Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZA MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Specialistica in Informatica

OGRE: analisi di un motore grafico 3D

Relatore:
Chiar.mo Prof.
Giulio Casciola

Presentata da:
Raffaele Bratta

Sessione III
Anno accademico 2010-2011

Indice

Elenco delle figure

Elenco dei sorgenti

Introduzione

Consideriamo le seguenti tre realtà: il Cineca, l'USMC[1] e Rocksteady Studios Ltd. La prima è un consorzio interuniversitario formato da più di cinquanta università italiane. Costituito nel 1969, a oggi, è il maggior centro di calcolo in Italia e uno dei più importanti a livello mondiale. La seconda è una delle forze armate d'élite dell'esercito statunitense. Creata nel 1775, ha partecipato a tutte le più importanti missioni a cui quest'ultimo ha preso parte. La terza e ultima è uno studio londinese che si occupa di sviluppo di videogame. Fondato nel 2004, ha pubblicato, nel 2009 e nel 2011, due videogiochi dedicati a Batman che si sono aggiudicati entrambi il titolo di GotY[2]. Si tratta quindi di tre realtà estremamente eterogenee e distanti, eppure, per quanto assurdo possa sembrare, esiste almeno un aspetto che le accomuna: tutte quante utilizzano un motore grafico 3D.

Ma che cos'è un motore grafico 3D? Non è facile darne una definizione precisa e infatti non ne esiste una universalmente riconosciuta, sia perché ci sono diverse tipologie di motori grafici, sia per il fatto che spesso sono integrati all'interno di middleware più grandi. Quest'ultimo, ad esempio, è il caso dei game engine che, anche se erroneamente accomunati ai motori grafici 3D, in realtà sono una loro estensione e specializzazione (vedi 1.1.5 a pagina 9). Una caratteristica comune a tutti i motori 3D è quella di essere librerie, ad alto livello, che mettono a disposizione un'API[3] per la creazione, gestione e visualizzazione di contenuti grafici. Possiamo quindi considerare questa, per quanto banale e generalista, come una possibile definizione di motore grafico.

La terminologia motore grafico trae invece origine dall'analogia col corrispettivo dispositivo meccanico. In una macchina, il motore è il componente che fornisce energia al sistema permettendogli di funzionare. Verosimilmente, in ambito software, il motore grafico è il componente che fornisce contenuti da visualizzare all'applicazione. Inoltre, sempre per analogia, così come un

[1] United States Marine Corps
[2] Game of the Year
[3] Application Programming Interface

motore meccanico, una volta avviato, è in grado di funzionare autonomamente, così anche un motore grafico, una volta inizializzato, non necessita di interventi esterni da parte dell'utente.

Il suffisso 3D sta invece a indicare che il motore è in grado di gestire grafica tridimensionale. Esistono infatti anche motori che non la supportano, dei quali però non parleremo in questa tesi. D'ora in avanti quindi, salvo dove diversamente specificato, anche se non utilizzeremo il suffisso 3D, esso sarà comunque sottinteso.

Ora che abbiamo un'idea, seppur vaga, di che cos'è un motore grafico, riprendiamo le tre realtà precedentemente introdotte e vediamo come queste lo utilizzano.

Il Cineca agisce da anello di congiunzione tra la realtà accademica e il settore dell'industria e della pubblica amministrazione. Gli ambiti in cui si trova a operare sono quindi molteplici: si va dal calcolo scientifico ad alte prestazioni fino allo sviluppo di applicazioni e servizi telematici. Tra le applicazioni sviluppate ce ne sono alcune di realtà virtuale il cui scopo è rendere possibile l'esplorazione di un territorio, più o meno vasto, ricostruito in 3D. Una delle più interessanti e innovative è Virtual Rome in cui è stata ricostruita la città di Roma durante l'età imperiale. Tramite questa applicazione, l'utente può esplorare liberamente la città, i sui dintorni ed entrare in alcuni edifici storici. Per lo sviluppo di Virtual Rome il Cineca ha utilizzato il motore grafico OpenSceneGraph. [5]

In ambito industriale, la simulazione da sempre ha costituito un ottimo strumento per la riduzione dei costi e per lo sviluppo di nuovi metodi e tecnologie. Nell'ultimo decennio, si è però affermata anche come un valido strumento di apprendimento, pensiamo ad esempio ai simulatori di volo utilizzati per il training dei piloti. Anche l'USMC ha deciso di adottare un simulatore per l'addestramento dei propri soldati. In collaborazione con Bohemia Interactive, ha quindi sviluppato Virtual Battlespace 2, un simulatore di guerra estremamente realistico capace di gestire contemporaneamente truppe di terra, veicoli terresti, mezzi navali, aerei ed elicotteri. Grazie a questo simulatore, l'USMC è riuscito a ridurre i costi delle esercitazioni e a studiare e sperimentare nuove tattiche militari. Virtual Battlespace 2 si basa sul motore grafico Real Virtuality 2, sviluppato anch'esso da Bohemia Interactive. [29]

Per quanto riguarda Rocksteady Studios Ltd., è superfluo descrivere come il motore grafico viene utilizzato all'interno di questa realtà. Ognuno di noi può facilmente immaginare il ruolo di primaria importanza che esso riveste all'interno di uno studio di sviluppo di videogame. Il motore grafico è infatti il componente software attorno a cui vengono costruite le funzionalità del videogioco e col quale tutte le altre librerie, necessarie allo sviluppo

del videogame stesso, devono interagire. Per la creazione dei suoi videogiochi Rocksteady Studios Ltd. ha utilizzato il motore grafico Unreal Engine (vedi 1.1.1 a pagina 5). [23]

Le tre realtà descritte rappresentano un valido esempio dei principali campi di applicazione dei motori grafici, ovvero la visualizzazione 3D, la simulazione e i videogame con quest'ultimo che, rispetto agli altri due, è di gran lunga il settore più importante. La nascita dei motori grafici deriva infatti da una specifica esigenza dell'industria dei videogiochi. Inoltre, sempre agli studi di sviluppo di videogame si devono la maggior parte delle innovazioni e della ricerca nel campo dei motori 3D.

Il motivo principale per cui si sceglie di utilizzare un motore grafico è dovuto alla complessità dell'ambiente da visualizzare. Nel caso del Cineca, ad esempio, la complessità deriva dalla dimensione del territorio mostrato dall'applicazione Virtual Rome. Già in epoca imperiale, la superficie della città di Roma si estendeva infatti per oltre 10 km^2. Nel caso dell'USMC la complessità è invece legata alla quantità di elementi visualizzati. In una simulazione di guerra, il numero delle entità presenti sul campo di battaglia, tra veicoli e fanteria, supera infatti abbondantemente le mille unità. Infine, nel caso di Rocksteady Studios Ltd. la complessità deriva dall'elevata qualità del rendering. Al giorno d'oggi, la resa grafica è infatti uno dei fattori determinanti affinché un videogame abbia successo. Un motore grafico, oltre a facilitare la gestione di tutte queste problematiche, mette a disposizione una serie di strumenti e di metodi che semplificano la creazione di applicazioni 3D. Ciò riduce quindi i tempi di sviluppo e permette ai programmatori di concentrarsi maggiormente nell'implementazione delle funzionalità dell'applicazione. Consideriamo, ad esempio, uno dei compiti più comuni nella programmazione grafica: il caricamento di un modello 3D. Utilizzando un motore grafico questa operazione può essere eseguita in poche righe di codice, molto spesso anche una sola. Utilizzando invece la libreria OpenGL, per quanto comunque potente e avanzata, il caricamento di un modello 3D non è supportato e l'implementazione di questa funzionalità è totalmente a carico dello sviluppatore.

In questa tesi descriveremo e analizzeremo il motore grafico OGRE, acronimo di Object-Oriented Graphics Rendering Engine. La scelta di analizzare proprio questo motore grafico è legata a diverse considerazioni. Innanzitutto, OGRE è rilasciato con licenza open source e quindi rende disponibile il suo codice sorgente. Questo è molto importante, in un contesto di studio e sperimentazione come quello universitario, perché permette di comprendere e analizzare anche il funzionamento interno del motore grafico. Inoltre, OGRE è un progetto maturo e stabile con una vasta comunità di sviluppatori e utilizzatori alle spalle. Esiste molta documentazione a riguardo, tra

wiki, libri e manuali, e un forum molto attivo per la richiesta di aiuto e consigli. A conferma, sia della bontà del progetto che delle ottime prestazioni del motore grafico, basta dire che OGRE è utilizzato anche da applicazioni commerciali, come videogame, editor 3D e simulatori. Infine, la caratteristica che contraddistingue OGRE da tutti gli altri motori grafici è il fatto di essere "solamente" un motore di rendering puro. Ciò significa che qualsiasi funzionalità non direttamente legata al rendering, come ad esempio la gestione degli input dell'utente, non è supportata da OGRE. Anche se questo può sembrare un difetto, in realtà ciò permetterà di concentrarci solamente sugli aspetti legati al rendering che, in un motore grafico, costituiscono la parte fondamentale.

Qualche riga fa abbiamo fatto un esempio riguardante OpenGL. In questa tesi daremo già per acquisita la conoscenza di tale libreria, o della sua analoga Direct3D. Il lettore di riferimento è infatti uno studente che abbia superato l'esame di Grafica del corso di laurea magistrale in informatica dell'università di Bologna. Tutti i concetti presentati durante tale corso verranno dati per scontati e non saranno oggetto di approfondimento all'interno di questa tesi. Inoltre, è richiesto che il lettore abbia anche una buona padronanza del linguaggio C++, sia perché è quello con cui è scritto OGRE, sia per il fatto che verrà usato per tutti gli esempi di questa tesi.

Nel primo capitolo (vedi a pagina 1) faremo una panoramica sul mondo dei motori grafici. Innanzitutto, ripercorreremo la loro storia dalla nascita fino ai giorni nostri. Durante questo viaggio, ci soffermeremo ad analizzare gli eventi più importanti mettendo in luce quali conseguenze hanno avuto sull'evoluzione dei motori 3D. Inoltre, vedremo qualche esempio di motore grafico del passato e di oggi e ne analizzeremo brevemente le peculiarità. Infine, descriveremo le caratteristiche principali di OGRE in modo da evidenziare le funzionalità base di un moderno motore grafico 3D.

Nel secondo capitolo (vedi a pagina 17) analizzeremo alcuni degli aspetti più importanti legati al funzionamento e all'utilizzo di OGRE. Questo è però un software molto vasto e complesso e alcune delle sue caratteristiche più avanzate, come ad esempio gli shader o le animazioni, richiedono una serie di prerequisiti e competenze che sono impossibili da fornire in questa tesi. Esistono infatti interi libri dedicati a questi argomenti e pertanto lasciamo al lettore, nel caso sia interessato, il compito di approfondirli. Nonostante ciò, il capitolo si pone come obiettivo quello di presentare una serie di nozioni che permettano comunque di creare le prime applicazioni basate su OGRE.

Nel terzo e ultimo capitolo (vedi a pagina 57) descriveremo l'utilizzo di OGRE all'interno di un'applicazione reale prendendo in esame il videogame Victory: the Age of Racing, sviluppato da Vae Victis Srl. Analizzeremo le cause che hanno portato quest'azienda a scegliere OGRE e vedremo come

questo è stato integrato con gli altri moduli e librerie usate dal videogame. Infine, mostreremo alcuni esempi di come Vae Victis ha esteso e personalizzato alcune funzionalità del motore grafico per soddisfare le proprie esigenze.

Capitolo 1

Analisi dei motori grafici

In questo capitolo faremo una panoramica sul mondo dei motori grafici 3D. Innanzitutto, descriveremo la loro storia partendo dagli anni ottanta fino ad arrivare ai giorni nostri (vedi 1.1). Nel fare questo, ci soffermeremo sugli eventi più importanti analizzando le conseguenze che questi hanno avuto sull'evoluzione dei motori grafici. A seguire, prendendo spunto da OGRE, ipotizzeremo invece di voler progettare un motore grafico 3D ex novo (vedi 1.2 a pagina 11). Ciò ci permetterà sia di mostrare le caratteristiche principali di OGRE, sia di mettere in luce gli aspetti fondamentali di un moderno motore grafico 3D.

1.1 Storia

La storia dei motori grafici inizia negli anni ottanta e si sviluppa contestualmente a quella dei videogiochi. Escludendo l'ultimo decennio, il termine motore 3D è sempre stato associato alla parola videogame i quali costituiscono il loro principale campo di applicazione. Proprio ai videogiochi si devono quindi la maggior parte delle innovazioni e delle tecniche introdotte, nel corso degli anni, all'interno dei motori grafici. Non deve dunque stupirci che i principali eventi riguardanti la storia dei motori grafici coincidano con altrettanti avvenimenti della storia dei videogame.

1.1.1 Timeline

Nella figura 1.1 nella pagina successiva possiamo vedere una timeline con i principali eventi della storia dei motori grafici.

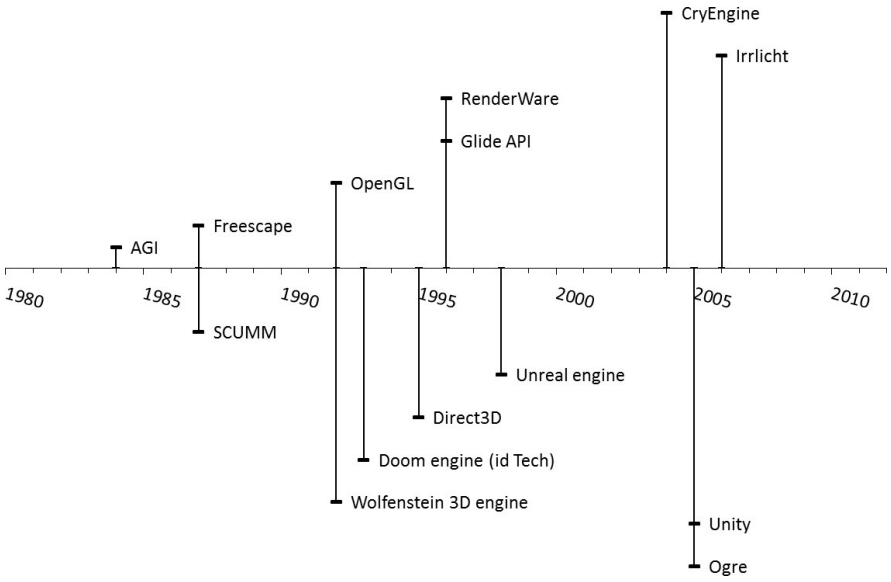

Figura 1.1: Timeline della storia motori grafici

1984

Nel 1983, IBM[1] commissiona a Sierra On-Line lo sviluppo di un videogame che metta in risalto le innovative caratteristiche tecniche del suo computer PCjr. Arthur Abraham, un dipendente di Sierra On-Line, sviluppa quindi un linguaggio di programmazione specializzato per lo scripting di avventure grafiche 2D e il suo interprete: l'AGI[2]. L'anno seguente, nel 1984, esce King's Quest: Quest for the Crown, il primo videogioco a sfruttare questo sistema. Il videogame avrà un successo commerciale enorme e ne saranno vendute oltre mezzo milione di copie. [3, 16]

1987

Aric Wilmunder e Ron Gilbert implementano per LucasArts, allora conosciuta come Lucasfilm Games, la prima versione dello SCUMM[3]. Questo sistema si pone a metà strada tra un motore grafico 2D, un game engine e un linguaggio di programmazione. Lo SCUMM mette infatti a disposizione degli sviluppatori una serie di tool utili alla produzione delle avventure grafiche, come ad esempio editor di dialoghi e strumenti musicali virtuali. È inoltre uno dei primi tentativi di successo, nel settore dei videogame, di creare un

[1] International Business Machines Corporation
[2] Adventure Game Interpreter
[3] Script Creation Utility for Maniac Mansion

middleware indipendente dalla piattaforma. Di conseguenza, le avventure grafiche che si avvalgono dello SCUMM riescono a funzionare su una grande varietà di sistemi: dall'Apple II a Microsoft Windows, passando per Atari ST e Commodore 64. [25]

Esce il videogame Driller basato sul motore Freescape. Quest'ultimo, sviluppato dallo studio britannico Incentive software, è considerato uno dei primi motori grafici 3D. Si tratta di un progetto tecnologicamente molto avanzato per quel periodo, tanto che l'azienda fatica a trovare personale in grado di lavorare sul motore. Nonostante la scarsa capacità di calcolo delle macchine dell'epoca, abbinata al fatto che il motore grafico utilizzava esclusivamente numeri discreti, esso consentiva comunque di creare una serie di primitive, come ad esempio triangoli, linee, cubi e piramidi. a ognuna di queste era possibile associare script da eseguire al verificarsi di certi eventi, dalla semplice visualizzazione della primitiva fino al rilevamento delle collisioni con altri oggetti. [9]

1992

id Software pubblica il videogioco Wolfenstein 3D che utilizza l'omonimo motore grafico sviluppato in larga parte da John Carmack. Le peculiarità di questo motore sono due: l'utilizzo per la prima volta delle texture e il rendering pseudo 3D che avviene tramite una tecnica di simil raytracing, chiamata raycasting. La finta terza dimensione comporta però alcune limitazioni, come ad esempio l'orientamento fisso della camera rispetto ad alcuni dei suoi assi di rotazione, quello di pitch e quello di roll, e l'obbligo di dover posizionare gli oggetti 3D tutti alla stessa altezza. [31]

Mark Segal e Kurt Akeley, due ingegneri della Silicon Graphics Inc., rilasciano la prima specifica di OpenGL. Tale documento descrive una serie di funzioni e il comportamento preciso che esse devono avere delegando la loro implementazione ai costruttori di chip grafici e/o a librerie di terze parti. L'obiettivo è quello di fornire, agli sviluppatori di applicazioni che fanno uso di grafica, un'API indipendente dalla piattaforma sottostante. [20]

1993

John Carmack e id Software continuano lo sviluppo del motore grafico di Wolfenstein 3D. Questa evoluzione prenderà inizialmente il nome di Doom engine, in onore del primo videogame in cui sarà impiegata, e in seguito di id Tech. Carmack e il suo team, oltre a rimuovere alcune limitazioni del motore

grafico, come il dover posizionare gli oggetti 3D tutti alla stessa altezza e la possibilità di utilizzare solamente un livello di illuminazione, aggiungono anche nuove feature. La più importante, sia a livello concettuale che di ottimizzazione del motore grafico, è sicuramente l'introduzione del BSP[4]. Tramite questa tecnica, è possibile suddividere l'ambiente visualizzato in aree in cui gli oggetti sono raggruppati gerarchicamente. Per la prima volta, viene quindi abbozzata l'idea di scena 3D, concetto alla base di tutti i moderni motori grafici. [7, 6]

1995

Nel 1992, Servan Keondjian fonda un'azienda, la RenderMorphics, che si occupa di sviluppare API di grafica 3D per applicazioni CAD[5] e medicali. Dopo qualche anno, nel febbraio del 1995, l'intera compagnia viene acquisita da Microsoft con l'intenzione di sfruttarne il know-how per creare, a sua volta, un'API di grafica per Windows 95, suo futuro sistema operativo. Quest'API prenderà il nome di Direct3D e diventerà in seguito parte integrante della libreria DirectX. [8]

1996

Nel 1994 tre ex dipendenti della Silicon Graphics Inc., Ross Smith, Gary Tarolli e Scott Sellers, avviano una compagnia, la 3dfx interactive, specializzata nella produzione di chip grafici. Due anni dopo, nel 1996, l'azienda rilascia il suo primo prodotto: la scheda video Voodoo. È un punto di svolta nel campo della computer grafica: la scheda è infatti una delle prime a essere dotata di una propria unità di calcolo, un antenato delle moderne GPU[6]. Le Voodoo e i chip grafici prodotti negli anni seguenti, che utilizzeranno la medesima tecnologia, saranno chiamate schede video accelerate. Questo appellativo è usato sia per differenziare le nuove schede da quelle della generazione precedente che per sottolineare il boost grafico che riusciranno a imprimere. La loro unità di calcolo è infatti specializzata nell'elaborazione di contenuti tridimensionali e quindi molto più veloce ed efficiente di un normale processore. Per permettere agli sviluppatori di utilizzare i nuovi chip grafici al massimo delle loro potenzialità 3fdx interactive mette a disposizione un'API chiamata Glide. Questa si differenzia dalle altre API sviluppate nello stesso periodo poiché non ha come obiettivo l'astrazione dall'hardware sottostante ma intende sfruttarlo appieno. Si può perciò considerare Glide

[4]Binary space partition
[5]Computer-Aided Design
[6]Graphics Processing Unit

come un sottoinsieme di OpenGL che mette però a disposizione solamente le funzioni che le schede video Voodoo sono in grado di eseguire. [2]

1997

Esce il primo di una lunga serie di videogame, tra i più famosi Burnout e Grand Theft Auto: San Andreas, basati sul motore RenderWare. Quella che una volta era una semplice API di grafica 3D si è dunque trasformata in uno dei più apprezzati game engine. Lo sviluppo di RenderWare inizia nel 1993 negli uffici di Criterion software Ltd., una sussidiaria inglese del Canon's European Research Lab. Inizialmente Renderware è utilizzato soprattutto per visualizzazioni 3D interattive ma, complice la comparsa delle schede video accelerate e delle prime console, l'azienda decide di trasformarlo prima in un motore grafico e successivamente in un middleware per la creazione di videogame. L'API proprietaria viene abbandonata in favore delle ormai consolidate OpenGL e Direct3D e vengono aggiunti moduli e strumenti per la gestione dell'intelligenza artificiale, della fisica, del sonoro, ecc. RenderWare è stato ampiamente impiegato, fino a diventare uno standard de facto, per lo sviluppo di videogiochi per la console PlayStation 2 grazie alle sue ottime prestazioni su quella piattaforma. [22]

1998

Epic games rilascia il videogame Unreal e concede in licenza l'utilizzo del motore grafico alla base del gioco. Oltre che per un utilizzo interno all'azienda, Unreal engine è il primo motore a essere progettato e realizzato fin da subito con l'idea di concederlo in licenza anche ad altri studi di sviluppo di videogame. Il motore, sviluppato in C++, è compatibile con una grande varietà di piattaforme, dai computer, tramite OpenGL e Direct3D, alle console. Inoltre, come tutti i game engine, mette a disposizione una serie di tool per lo sviluppo di videogame come ad esempio moduli per la gestione della fisica, delle comunicazioni via rete, ecc. La caratteristica che decreterà il successo dell'Unreal engine è pero il suo linguaggio di scripting ad alto livello, tanto potente quanto semplice da utilizzare. Molti videogame sono stati infatti realizzati utilizzando esclusivamente questo linguaggio. La grande portabilità del motore e la sua semplicità lo hanno reso ben presto il game engine più utilizzato nella produzione di videogame. [27]

2002

Nel 1999 tre fratelli turchi, Cevat, Avni e Faruk Yerli, creano a Coburg, in Germania, una piccola impresa chiamata Crytek. Inizialmente la neonata

azienda si occupa di sviluppare, per conto di NVIDIA, quelle che vengono chiamate tech demo. Si tratta di piccoli programmi il cui compito è pubblicizzare, nella maniera più spettacolare possibile, le caratteristiche tecniche delle schede video. In particolare la tech demo X-Isle, presentata all'ECTS[7] del 2000 e che poi evolverà nel videogame Far Cry, stupisce gli addetti ai lavori sia per l'eccellente qualità grafica che per una serie di effetti visivi particolarmente innovativi. L'esperienza maturata e le ottime critiche ricevute spingono i tre ragazzi a entrare nel mercato dei game engine. Nel 2002 viene infatti annunciato il CryEngine e due anni dopo, nel 2004, esce il videogame Far Cry, il primo a utilizzare questo nuovo motore. CryEngine riesce a conquistarsi un'importante fetta di mercato grazie alla sua elevata resa grafica. Il motore è infatti noto per essere uno dei principali innovatori nel campo degli effetti visivi. [13]

2005

La storia di OGRE inizia nel 1999, grazie a Steve Streeting, con la creazione di una piccola libreria il cui scopo era quello di facilitare e automatizzare la programmazione tramite Direct3D. Durante lo sviluppo il progetto evolve a un livello di astrazione tale da diventare indipendente dall'API sottostante. Questo convince Streeting ad aggiungere anche il supporto a OpenGL e a creare qualcosa di più ambizioso: un motore grafico. Nel 2000 OGRE viene registrato come progetto su SourceForge e nel 2005 viene rilasciata la versione 1.0. Fin da subito si crea attorno al progetto una nutrita comunità di utilizzatori e sviluppatori, sia per il fatto che OGRE viene pubblicato con licenza open source sia perché, nel panorama dei software liberi, non esistevano progetti simili. La principale caratteristica di OGRE è il fatto di essere solamente un puro motore di rendering: qualsiasi task non riguardante questo aspetto, come la gestione degli input dell'utente, il rilevamento delle collisioni, ecc., o non è implementato o è delegato a plugin/librerie esterne. [4]

Viene rilasciata la prima versione di Unity, un middleware per la produzione di videogame e animazioni 3D. La particolarità di questo motore è quella di supportare sistemi ignorati dai suoi competitor fino a quel momento come i browser, tramite un plugin dedicato, e i sistemi embedded, ovvero smartphone e tablet. Questa sua caratteristica lo rende quindi una scelta obbligata per molti sviluppatori di browser game e di applicazioni per telefoni cellulari che richiedano un certo livello di qualità nella resa grafica.

[7] European Computer Trade Show

In quegli anni infatti circa il 75% dei giochi con grafica tridimensionale delle piattaforme iPhone e Android sono basati su Unity. [26]

2006

Lo sviluppo del motore grafico Irrlicht Engine inizia nel 2003 da parte di Nikolaus Gebhardt, un giovane programmatore di videogame austriaco. Inizialmente il suo obiettivo era creare una libreria open source che permettesse lo sviluppo, in maniera semplice e veloce, di videogiochi 3D caratterizzati da un ridotto grado di complessità. Secondo Gebhardt infatti tutti i motori 3D e i game engine del periodo erano troppo complicati e difficili da utilizzare. La libreria si trasforma però ben presto in un vero e proprio motore grafico 3D. Nel 2006, grazie anche al lavoro della comunità che nel frattempo si è creata attorno al progetto, viene rilasciata la versione 1.0. L'obiettivo della semplicità rimane, tanto che alcune funzionalità ormai comuni ma di difficile utilizzo non sono supportate, e a esso si aggiunge quello della portabilità e della leggerezza. Irrlicht Engine è infatti noto per occupare poco in termini di risorse e per supportare una grande varietà di piattaforme, anche alcune molto vecchie e ormai obsolete. La natura open source del progetto ha fatto sì che la comunità rimediasse alle lacune del motore estendendolo con ulteriori funzionalità e con lo sviluppo plugin indipendenti. Di queste versioni estese, spesso chiamate IrrlichtNX, ne esistono diverse: da quella specializzata per lo sviluppo dei videogame a quella specializzata per lo sviluppo di simulazione fisiche, ecc. [14]

1.1.2 Anni ottanta: 2D e grafica dedicata

Gli anni ottanta vedono affermarsi il videogame come forma di intrattenimento di massa, sia per la comparsa dei primi personal computer sia per il fatto che le case costruttrici di hardware utilizzano spesso i videogiochi come veicolo promozionale per i loro prodotti. Quest'ultimo ad esempio è il caso dell'IBM PCjr (vedi 1.1.1 a pagina 2). Si crea quindi una nuova industria che, nel giro di pochi anni, vede nascere numerosi studi di sviluppo di videogame. Alcuni di questi, sia per velocizzare i tempi di rilascio di nuovi prodotti che per diminuire i costi di sviluppo, decidono di specializzarsi in determinate tipologie di giochi e creano quindi strumenti ad hoc per il loro sviluppo. Tra questi tool ci sono anche i primi prototipi dei motori grafici.

La scarsa capacità di calcolo dei computer dell'epoca non permette però grandi cose e infatti, salvo rare eccezioni (vedi 1.1.1 a pagina 3), la maggior parte dei motori fanno uso solamente di grafica bidimensionale. Inoltre, la vasta frammentazione hardware rende problematico ottimizzare i motori

grafici per più di una piattaforma. La conseguenza è che i motori non possono quindi essere concessi in licenza ad altri sviluppatori ma vengono usati solamente all'interno dell'azienda che li ha progettati. L'unica eccezione è quella dello SCUMM (vedi 1.1.1 a pagina 2) che sarà utilizzato anche dalla Humongous Entertainment.

1.1.3 Primi anni novanta: 2.5D e texture

John Carmack, programmatore e co-fondatore di id Software, è l'indiscusso protagonista di questi anni. La tecnica di rendering pseudo 3D, anche detta 2.5D, da lui inventata viene copiata da altri studi di sviluppo e diventa uno dei tratti caratteristici dei videogame di quel periodo. Sempre per merito di Carmack prendono forma idee e algoritmi che saranno alla base dei motori grafici odierni. Sfruttando la maggior capacità di calcolo dei computer dell'epoca, Carmack inserisce all'interno dei motori da lui sviluppati il supporto alle texture (vedi 1.1.1 a pagina 3) e abbozza uno dei concetti fondamentali dei motori grafici moderni: la scena 3D. Tramite l'utilizzo di un BSP (vedi 1.1.1 a pagina 3) nei motori di id Software è infatti possibile rappresentare in maniera gerarchica gli oggetti 3D visualizzati.

Sempre all'inizio degli anni novanta vengono fatti i primi tentativi per uniformare la produzione di contenuti grafici. Il soggetto più attivo è il consorzio ARB[8] che, con la pubblicazione della specifica OpenGL (vedi 1.1.1 a pagina 3), tenta di creare uno standard per la programmazione grafica 2D e 3D.

1.1.4 Metà anni novanta: la guerra delle API

In questi anni si assiste al declino di numerose piattaforme diventate ormai obsolete con la conseguente riduzione della frammentazione hardware. I vantaggi che ne derivano sono però vanificati dalla proliferazione delle API. Invece di cercare uno standard comune, molte aziende preferiscono creare e implementare una propria API di grafica 3D con la conseguenza che la frammentazione si sposta a livello software. Tutte le API sono accomunate dallo stesso obiettivo, far sì che gli sviluppatori non debbano interagire direttamente con l'hardware, ma si differenziano per il modo di raggiungerlo. Da una parte le API che fanno dell'astrazione dall'hardware sottostante il loro punto di forza, dall'altra quelle che mirano al suo pieno sfruttamento. La prime, di cui OpenGL (vedi 1.1.1 a pagina 3) e Direct3D (vedi 1.1.1 a pagina 4) sono i maggiori esponenti, si caratterizzano per una maggiore compatibilità

[8] Architecture Review Board

e per una minor efficienza mentre per le seconde, di cui Glide (vedi 1.1.1 a pagina 4) è il maggior esponente, vale il contrario.

Sempre in questo periodo iniziano a essere costruite le prime schede video accelerate, ovvero dotate di una propria unità di elaborazione. Inizialmente l'unico produttore in grado di utilizzare tale tecnologia è 3dfx interactive e la sua API Glide, più efficiente e meglio ottimizzata rispetto a OpenGL e Direct3D per quel tipo di scheda, diventa la più utilizzata sia nei motori grafici che dagli sviluppatori. Successivamente con l'ingresso nel mercato delle schede video di nuovi produttori, su tutti ATI Technologies Inc. e NVIDIA, si torna però a preferire la compatibilità rispetto all'efficienza e OpenGL e Direct3D diventano quindi le API più utilizzate. L'ascesa delle API più astratte coincide col declino di Glide e di 3dfx interactive che dichiarerà bancarotta nel 2002.

1.1.5 Fine anni novanta: dai motori grafici ai game engine

Grazie anche all'uscita delle console di sesta generazione, PlayStation 2, Xbox, Nintendo 64, ecc., si assiste in questi anni all'esplosione dell'industria dei videogiochi che, al pari di cinema e televisione, diventa uno dei settori più importanti dell'entertainment. L'aumentare degli studi di sviluppo di videogame fa sì che alcuni di loro come Criterion software Ltd. (vedi 1.1.1 a pagina 5), Epic Games (vedi 1.1.1 a pagina 5), id Software, ecc., forti della maggior esperienza accumulata nell'ambito della grafica 3D, inizino a concedere in licenza i motori grafici alla base dei loro giochi. In generale infatti i nuovi studi nati in questi anni non hanno né le competenze tecniche né le risorse necessarie, sia in termini di tempo che di denaro, per realizzare un proprio motore grafico e preferiscono quindi acquistarne uno già fatto. La richiesta di motori 3D è così grande che alcune aziende modificano il proprio modello di business passando dall'essere produttori di videogame all'essere sviluppatori di motori grafici.

In questo periodo lo sviluppo dei motori 3D procede per fasi. Se ne possono individuare tre ognuna consequenziale alla precedente. La prima è caratterizzata dal fatto che i motori grafici abbandonano le proprie API proprietarie, dove possibile, in favore delle ormai affermate OpenGL e Direct3D. L'utilizzo di questi due sottosistemi grafici permette ai motori 3D di essere subito compatibili con buona parte delle piattaforme del periodo. Di conseguenza gli sviluppatori possono concentrarsi maggiormente sull'innovare e ottimizzare i propri motori piuttosto che perdere tempo nel renderli compatibili con quel particolare tipo di hardware o di scheda video. La seconda fase

è caratterizzata invece dall'evoluzione della maggior parte dei motori grafici in game engine. La causa di questa trasformazione è il fatto che l'industria videoludica è la principale utilizzatrice dei motori 3D. Pertanto, per ottenere nuovi clienti e nel contempo distinguersi dalla concorrenza, gli sviluppatori di motori grafici aggiungono di volta in volta tool finalizzati alla produzione di videogame. Questi strumenti vanno da editor di eventi a linguaggi di scripting passando per moduli per la gestione della fisica, del networking, ecc. Non si parla quindi più di motori grafici ma di game engine dove questa terminologia sottolinea la loro nuova specializzazione. Inoltre, in questi middleware il rendering 3D non è più la componente principale ma spesso sono proprio le utility ausiliare a determinare la scelta di un game engine rispetto a un altro. Infine, la terza e ultima fase si caratterizza per il sensibile miglioramento della resa grafica dei motori 3D. Questo si rende possibile grazie alla comparsa degli shader, piccoli programmi personalizzabili che vengono eseguiti all'interno della GPU e che sono in grado di modificare la pipeline di rendering. Sfruttando questa tecnologia, i motori 3D possono infatti creare nuovi effetti grafici come HDRR[9], bump mapping, illuminazione dinamica, self shadowing, ecc. Uno dei game engine più innovativi nella creazione di effetti visivi è il CryEngine (vedi 1.1.1 a pagina 5).

1.1.6 Anni duemila: il ritorno dei motori grafici

In questi anni la grafica 3D real time viene utilizzata sempre più anche con altre finalità come la visualizzazione scientifica, la navigazione 3D interattiva, la simulazione industriale, ecc. Nel contempo si assiste alla crescita del fenomeno degli indie game, piccoli videogiochi sviluppati a basso costo da team di poche persone. Tutte queste tipologie di applicazioni sono accomunate dall'avere budget relativamente ridotti per il loro sviluppo. Le limitate risorse a disposizione non permettono infatti l'acquisto delle licenze dei game engine che hanno spesso un costo superiore alle centinaia di migliaia di dollari. Gli sviluppatori decidono quindi di creare proprie librerie grafiche o, preferibilmente, di affidarsi ai motori 3D open source che sono creati in quegli anni. Il rinnovato interesse nei confronti dei motori grafici permette a quelli già esistenti, come OGRE (vedi 1.1.1 a pagina 6) e OpenSceneGraph, di affermarsi sempre più e nello stesso pone le basi per creazione di nuovi progetti, come IrrlichtEngine (vedi 1.1.1 a pagina 7) e jMonkeyEngine.

[9]High dynamic range rendering

1.1.7 Tra presente e futuro: embedded e fotorealismo

Negli ultimi cinque anni si è assistito a un crescita costante delle piattaforme embedded, ovvero smartphone e tablet. L'industria videoludica si interessa fin da subito a questi nuovi dispositivi intuendone le potenzialità commerciali. Di conseguenza i produttori di motori 3D e game engine investono molte risorse nel cercare di supportare al meglio queste piattaforme. Anche se attualmente quasi tutti i motori grafici sono compatibili con i dispositivi embedded, Unity (vedi 1.1.1 a pagina 6) è stato il primo a supportarli. Il mercato delle piattaforme embedded rappresenterà infatti il campo su cui, nei prossimi anni, i motori grafici introdurranno le maggiori innovazioni e competeranno fra loro.

In un futuro più remoto invece la sfida che i motori grafici dovranno affrontare è la ricerca del fotorealismo, ovvero produrre rendering con una qualità tale da renderli sempre più indistinguibili da una fotografia. I primi passi in questa direzione sono già stati fatti ma sono ancora tentativi sporadici e lontani dall'obiettivo. Per esempio Team Bondi, nella realizzazione del suo ultimo videogame L.A. Noire, è riuscita a ottenere un'incredibile livello di dettaglio nella resa grafica di volti umani e capelli mentre Crytek, con l'ultima versione del CryEngine, si è distinta per il realismo con cui il motore riesce a rappresentare la vegetazione. Inoltre, Carmack ha dichiarato che la prossima release del motore id Tech, grazie alla maggior capacità di calcolo di computer e schede video, farà uso di algoritmi basati su raytracing per ottenere rendering sempre più realistici.

1.2 Troll engine

Per mettere in luce gli aspetti fondamentali di un motore grafico, ipotizzeremo di volerne implementare uno: il Troll engine. OGRE rappresenterà, per il motore da progettare, sia il modello di riferimento che il suo principale competitor. L'utilizzo di questo stratagemma ha un duplice vantaggio. Il primo è dato dal fatto che, essendo OGRE un generico motore grafico, potremo concentrarci solamente sugli aspetti legati al rendering. Il secondo vantaggio è invece che, descrivendo le feature e le funzionalità del Troll engine, descriveremo anche molte delle caratteristiche di OGRE. [10]

1.2.1 Struttura e implementazione

Analizziamo le caratteristiche del motore grafico dal punto di vista delle scelte progettuali. Queste sono dettate dagli obiettivi del Troll engine: elevate prestazioni, compatibilità multipiattaforma e semplicità d'uso.

C++

Il motore grafico sarà implementato utilizzando il linguaggio C++. Oltre a essere uno standard de facto nel campo delle applicazioni grafiche, questo linguaggio offre molti vantaggi, come ad esempio il supporto alla programmazione a oggetti, l'ottimizzazione e la velocità di esecuzione del codice prodotto, l'esistenza di numerose librerie esterne e la compatibilità con la maggior parte delle piattaforme a oggi esistenti.

API a oggetti

Troll engine sarà modellato e sviluppato utilizzando il paradigma di programmazione a oggetti. L'utilizzo di questa metodologia è una scelta obbligata per un progetto vasto e complesso come un motore grafico. I vantaggi legati all'utilizzo di questo paradigma, come ad esempio la modularità e il riutilizzo del codice, sono infatti tanto più evidenti quanto più il software da realizzare è di grandi dimensioni. Utilizzando la programmazione a oggetti, potremo inoltre avvalerci delle strategie definite dai design pattern e quindi produrre codice più semplice, ottimizzato e manutenibile. Infine, per l'utente è più semplice comprendere e utilizzare un'API a oggetti rispetto a una realizzata secondo il modello imperativo.

Compatibilità multipiattaforma

Dal punto di vista grafico, il motore sarà in grado di utilizzare indifferentemente sia l'API OpenGL che Direct3D. Così facendo, garantiremo quindi piena compatibilità sia con le piattaforme di derivazione Unix che con quelle sviluppate da Microsoft. Per i restanti moduli del Troll engine utilizzeremo invece solamente funzioni standard e/o librerie indipendenti dal sistema sottostante. Ad esempio, per la gestione della temporizzazione o del filesystem potremmo utilizzare la libreria Boost, compatibile con i sistemi Windows, Linux e Mac OS.

Multithread

Per sfruttare al meglio le moderne architetture a più processori, il motore grafico sarà in grado di funzionare in modalità multithread. Ad esempio, è possibile velocizzare il caricamento delle risorse grafiche, come ad esempio mesh e texture, parallelizzandolo tramite l'utilizzo di più thread. La modalità multithread, che potremmo implementare sempre tramite la libreria Boost, agirà in maniera autonoma e trasparente agli utenti del motore grafico.

1.2.2 Geometrie

Mesh

Troll engine sarà ovviamente in grado di gestire geometrie definite tramite mesh: sia che queste vengano caricate da file esterni, sia che vengano dichiarate, via codice, dall'utente. Per la memorizzazione delle mesh potremmo indifferentemente scegliere un formato proprietario, uno aperto o uno creato ad hoc. Qualunque esso sia è però importante che, oltre alla geometria, supporti anche informazioni aggiuntive come LOD[10], animazioni, ecc.

Curve e superfici

Le considerazioni fatte per le mesh sono altrettanto valide per curve e superfici come patch di Bezier, NURBS[11], ecc. In aggiunta il motore grafico sarà in grado di visualizzare in maniera parametrica sia le curve che le superfici e di discretizzarle in mesh.

1.2.3 Rendering

Materiali Troll engine metterà a disposizione dell'utente un semplice linguaggio di scripting per la definizione dei materiali, ovvero tutto quanto riguarda l'aspetto estetico di una geometria. Gli script potranno essere caricati da file esterni o definiti dall'utente via codice.

Shader Il motore grafico sarà in grado di utilizzare la versione più recente, attualmente la 3.0, di tutte e tre le tipologie di shader: vertex, fragment e geometry. Inoltre, Troll engine supporterà la definizione di questi programmi sia tramite i linguaggi GLSL[12] e HLSL[13] che tramite il metalinguaggio Cg[14].

Texture Il motore grafico supporterà diverse tipologie e formati di texture come immagini, cubemap, texture volumetriche, stream e buffer di memoria, ecc.

[10] Level of detail
[11] Non Uniform Rational B-Splines
[12] OpenGL Shading Language
[13] High Level Shading Language
[14] C for Graphics

1.2.4 Ambiente

Scene manager Troll engine utilizzerà un modulo chiamato scene manager per organizzare in maniera gerarchica le geometrie caricate. Inoltre, questo componente ottimizzerà di volta in volta il rendering delle entità presenti nella scena 3D a seconda che questa sia un un ambiente chiuso, uno spazio aperto, ecc.

Illuminazione e ombre Il motore grafico sarà in grado di gestire un numero virtualmente infinito di luci. Per il rendering delle ombre, a seconda della qualità visiva che l'utente vorrà ottenere, Troll engine potrà utilizzare diverse tecniche: da quelle modulative a quelle additive, da quelle basate su stencil buffer a quelle basate su texture, ecc.

Animazioni Troll engine sarà in grado di gestire sia le animazioni di tipo skeletal che quelle di tipo morph. Inoltre, il motore grafico metterà a disposizione dell'utente funzionalità per la gestione di interpolazioni e traiettorie.

1.2.5 Effetti speciali

Compositor Il motore grafico metterà a disposizione un modulo, chiamato compositor, il cui compito sarà quello di applicare effetti di post processing a ogni ciclo di rendering. L'utente potrà creare questi effetti tramite lo stesso linguaggio di scripting impiegato nella definizione dei materiali.

Particellari Troll engine sarà dotato di un componente per la creazione di effetti particellari come neve, fumo, polvere, ecc. La definizione di questi effetti avverrà in maniera analoga a quelli di post processing.

1.2.6 Automazione e ottimizzazioni

Culling, Z-Buffer e trasparenze Il motore grafico si occuperà autonomamente della gestione del culling degli oggetti presenti nella scena 3D. Tramite lo scene manager a ogni ciclo di rendering saranno identificati tutti gli oggetti non visibili, o perché nascosti da altri o perché non inquadrati, e verrà evitata la loro renderizzazione. Inoltre, Troll engine si occuperà anche di gestire in maniera automatica lo Z-Buffer degli oggetti trasparenti.

Geometrie statiche e caching delle risorse Per ottimizzare le prestazioni il motore grafico permetterà la definizione di geometrie statiche. Con

questo termine si identifica una collezione di geometrie che rimangono invarianti durante tutta l'esecuzione dell'applicazione. Sfruttando questa caratteristica, è possibile ottimizzare sensibilmente il loro processo di rendering. Troll engine si occuperà inoltre del caching delle risorse più utilizzate dall'applicazione evitando che queste debbano essere inutilmente ricaricate di volta in volta.

LOD Il motore grafico implementerà un sistema per gestire dinamicamente i dettagli di geometrie e materiali a seconda della loro distanza dal punto di osservazione. Questo sistema entrerà automaticamente in funzione per tutte le geometrie e i materiali a cui l'utente non avrà associato nessuna tecnica di LOD.

Capitolo 2

OGRE

In questo capitolo analizzeremo alcuni degli aspetti più importanti di OGRE. Innanzitutto, vedremo come installare il motore grafico e creeremo la prima semplice applicazione d'esempio (vedi 2.1). A seguire, descriveremo due dei concetti fondamentali di OGRE: la scena 3D (vedi 2.2 a pagina 26) e il suo manager (vedi 2.3 a pagina 31). Successivamente, vedremo come dare maggior profondità e realismo agli oggetti di scena (vedi 2.4 a pagina 34). Dopo di che, analizzeremo come OGRE gestisce il ciclo di rendering e come interagire con esso (vedi 2.5 a pagina 46). Per finire, vedremo un altro concetto molto importante del motore grafico: i materiali (vedi 2.6 a pagina 48). [15, 18, 17, 19]

2.1 Primi passi

Scaricare, installare e configurare OGRE è il primo passo nel processo di apprendimento del motore grafico stesso. Una volta portato a termine tutto ciò, arriva poi il momento in cui creare la prima applicazione vera e propria. Questo programma, che dovrà essere il più semplice possibile, si limiterà a visualizzare a video una geometria 3D.

2.1.1 Installazione

Per installare OGRE possiamo seguire due strade: o compilarne i sorgenti o scaricare un'SDK[1] che contiene le librerie già compilate. La prima opzione è utile quando dobbiamo ottimizzare il motore grafico per una certa architettura hardware o quando vogliamo personalizzare alcuni aspetti della libreria,

[1]Software development kit

come ad esempio abilitare o meno il supporto per i thread. La seconda modalità, quella consigliata, è invece più rapida e adatta alla maggior parte delle situazioni.

L'installazione è l'unica cosa di OGRE che differisce tra un sistema e l'altro. Queste differenze sono però minime e pertanto descriveremo il processo d'installazione dell'SDK di OGRE solamente per una piattaforma, in questo caso il sistema Microsoft Windows 7. L'installazione dell'SDK è un procedimento estremamente semplice:

1. Apriamo il browser alla pagina `http://www.ogre3d.org/download/sdk`.

2. Scarichiamo l'SDK più appropriata per il nostro ambiente di sviluppo, in questo caso quella per Visual C++ .Net 2008 (32-bit).

3. Apriamo il file appena scaricato e scegliamo il percorso in cui OGRE verrà installato.

4. Una volta che l'archivio avrà finito di estrarre i file, troveremo la directory `OgreSDK_vc9_v1-7-4` all'interno del percorso specificato precedentemente.

2.1.2 SDK

Soffermiamoci ad analizzare il contenuto di `OgreSDK_vc9_v1-7-4`, in particolare quello delle directory `lib` e `bin` che contengono, rispettivamente, le librerie statiche e dinamiche di OGRE. Al loro interno si trovano due ulteriori directory: `Debug` e `Release`. La prima contiene le librerie con i simboli di debug, meno ottimizzate, ma più utili durante la creazione e il debugging di un'applicazione. La seconda contiene invece le librerie da utilizzare una volta terminato lo sviluppo per ottenere, dal motore grafico, le massime prestazioni.

La figura 2.1 nella pagina successiva mostra il contenuto della directory `bin/Release` dell'SDK di OGRE appena installata. Al suo interno, possiamo vedere tre tipologie di file: eseguibili, DLL[2] e file di configurazione. Tralasciamo gli `.exe`, che non useremo in quanto servono solamente ad aggiornare mesh di precedenti versioni di OGRE, e concentriamoci sulle `.dll` e sui `.cfg`.

Tra le DLL la più importante è sicuramente `OgreMain.dll`. Questa è la libreria che contiene il codice di OGRE compilato e che viene caricata, a tempo d'esecuzione, dalle applicazioni che utilizzano il motore grafico. Le

[2]Dynamic-link library

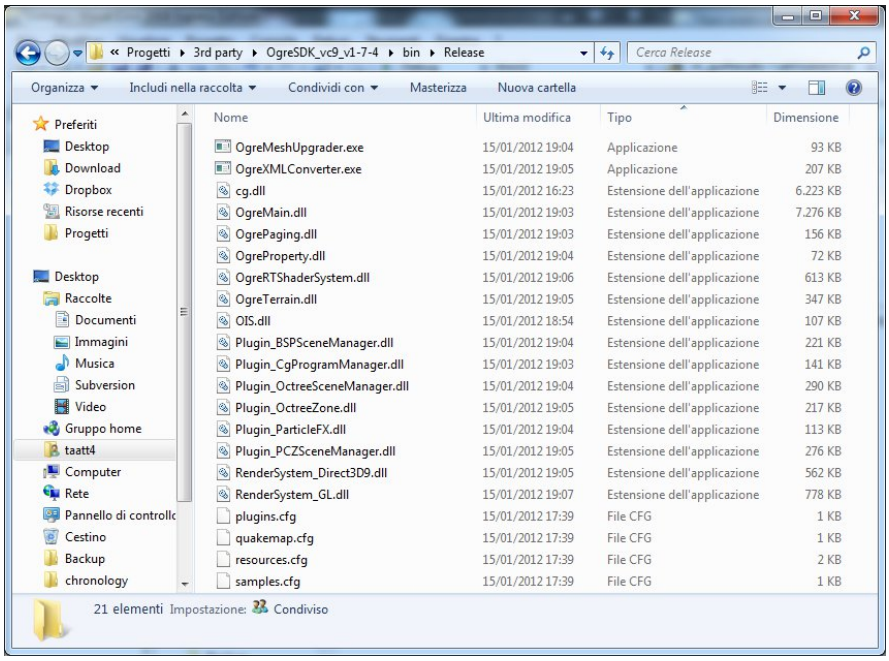

Figura 2.1: Il contenuto della directory `bin/Release` dell'SDK di OGRE

DLL il cui nome inizia con `Plugin_` sono invece librerie che aggiungono funzionalità extra a OGRE. Spesso sviluppati dalla comunità, di questi plugin ne esistono diverse tipologie, ma nell'SDK sono inclusi solamente quelli in genere più utilizzati. Infine, le DLL il cui nome inizia con `RenderSystem_` sono le librerie che OGRE utilizza per interfacciarsi col sottosistema grafico. Nell'SDK che abbiamo installato sono presenti sia il wrapper per OpenGL che quello per Direct3D9.

I `.cfg` sono file di configurazione di OGRE letti durante la sua inizializzazione. In `plugins.cfg` sono elencate tutte le librerie extra che il motore grafico dovrà caricare all'avvio. Il file è formato da una serie di righe del tipo `Plugin=RenderSystem_Direct3D9`. Questa, ad esempio, specifica a OGRE di caricare la libreria per interfacciarsi col sottosistema grafico Direct3D9. `resources.cfg` contiene invece la lista di tutte le risorse, come ad esempio mesh, texture e animazioni, che saranno caricate da OGRE durante la fase d'inizializzazione. Queste risorse posso essere caricate direttamente da una directory del filesystem oppure estratte da un file ZIP. Nel primo caso si aggiunge al file di configurazione un riga del tipo `FileSystem=../../media/thumbnails`, nel secondo una del tipo `Zip=../../media/packs/SdkTrays.zip`.

2.1.3 Hello world

Da tradizione, Hello world è il programma che stampa a video la stringa "hello, world". Per la sua semplicità è spesso usato per illustrare la sintassi base di un linguaggio di programmazione e, nel contempo, per verificare che quest'ultimo sia stato configurato correttamente.

A differenza dell'Hello world per un generico linguaggio di programmazione, quello di OGRE non stamperà nulla a video, ma si limiterà invece a inizializzare il motore grafico e ad avviare il ciclo di rendering. Questa applicazione ci permetterà di prendere confidenza con alcune classi di OGRE e di analizzare alcune caratteristiche del motore grafico.

Creazione e compilazione

Per lo sviluppo di questa applicazione, e di tutte le successive, utilizzeremo Microsoft Visual C++ 2008 Express Edition. Di conseguenza, alcuni degli step di seguito riportati saranno specifici per questo particolare ambiente di sviluppo. Facendo le opportune modifiche, nulla ci vieta però di utilizzare un altro IDE[3], come ad esempio Eclipse, o di non usarne nessuno.

1. Creiamo un nuovo progetto vuoto.
2. Aggiungiamo al progetto il file `main.cpp` contente il codice come quello del sorgente 2.1 a pagina 22.
3. Aggiungiamo ai percorsi di ricerca degli include le directory `OgreSDK_vc9_v1-7-4/include` e `OgreSDK_vc9_v1-7-4/boost_1_48`.
4. Aggiungiamo ai percorsi di ricerca delle librerie le directory `OgreSDK_vc9_v1-7-4/lib/debug` e `OgreSDK_vc9_v1-7-4/boost_1_48/lib`.
5. Aggiungiamo alle librerie da linkare `OgreMain_d.lib` e `OIS_d.lib`.
6. Compiliamo il progetto.
7. Impostiamo `OgreSDK_vc9_v1-7-4/bin/debug` come directory di lavoro.
8. Facciamo partire il programma e, se tutto è andato a buon fine, dovremmo vedere una finestra come quella nella figura 2.2 a pagina 22.
9. Clicchiamo su `OK` e partirà l'applicazione vera e propria.

[3]Integrated Development Environment

Al punto 3 abbiamo specificato al compilatore i percorsi in cui si trovano gli header di OGRE e di Boost. Quest'ultima è la libreria che il motore grafico utilizza per la gestione dei thread. Siccome nell'SDK questi sono abilitati di default, per far sì che l'applicazione compili correttamente, occorre specificare la directory in cui reperire gli header di Boost.

Al punto 4 abbiamo fatto la stessa cosa del punto 3, ma per le librerie. Le considerazioni fatte per gli header rimangono però valide anche in questo caso.

Al punto 5 abbiamo specificato al compilatore le librerie da linkare. `OgreMain_d.lib`, come possiamo facilmente immaginare, è la libreria principale di OGRE. `OIS_d.lib` è invece una libreria per la gestione degli input da mouse e tastiera. Quest'ultima non fa parte di OGRE, ma viene comunque installata con l'SDK. Il suffisso `_d`, utilizzato con analogo scopo anche nei file di configurazione, sta a indicare che si tratta di librerie di debug.

Infine, al punto 7 abbiamo impostato il percorso della directory di lavoro. All'interno di questa OGRE si aspetta di trovare sia le DLL che i file di configurazione. Notiamo che, così come al punto 5 abbiamo specificato al compilatore di linkare le librerie di debug, così anche per la directory di lavoro abbiamo specificato quella che contiene il medesimo tipo di DLL.

Finestra delle impostazioni di OGRE

La figura 2.2 nella pagina seguente mostra la finestra che appare prima che parta l'applicazione vera e propria. Tramite questa possiamo configurare sia impostazioni dell'applicazione stessa, come la risoluzione, che del motore grafico, come ad esempio l'utilizzo di singola o doppia precisione per la rappresentazione dei numeri a virgola mobile. Inoltre, sempre tramite questa finestra, è possibile configurare anche le impostazioni di alcuni dei plugin più complessi.

La finestra viene mostrata subito dopo una prima fase di inizializzazione del motore grafico in cui questo legge il file `plugins.cfg` e carica le DLL in esso specificate. Per verificare che sia effettivamente così, proviamo a cancellare la riga `Plugin=RenderSystem_Direct3D9_d` dal file `plugins_d.cfg` presente nella directory di lavoro del nostro progetto. Lanciamo di nuovo l'applicazione e noteremo che, dalle possibili scelte del menù `Rendering Subsystem`, è sparita la voce `Direct3D9 Rendering Subsystem`.

Se nel frattempo abbiamo anche cambiato qualche impostazione, noteremo che il valore scelto viene preservato tra un'esecuzione e l'altra dell'applicazione. Questo è dovuto al fatto che, cliccando su `OK`, le impostazioni vengono salvate in un file, chiamato `ogre.cfg`, all'interno della directory di lavoro specificata.

Figura 2.2: La finestra delle impostazioni di OGRE

Codice del programma

Il sorgente 2.1 contiene il codice della nostra applicazione. Il programma è strutturato in modo tale che possa essere utilizzato, come template, anche per gli esempi e le applicazioni successive.

```
1  #include <Ogre/ExampleApplication.h>
2
3
4  class PrimiPassi :
5    public ExampleApplication
6  {
7
8  public:
9
10   void createScene()
11   {
12   }
13 };
14
15
```

```
16 int main()
17 {
18   PrimiPassi primiPassi;
19   primiPassi.go();
20
21   return 0;
22 }
```

Sorgente 2.1: Hello world

Nel programma è presente una sola classe, `PrimiPassi`, che estende `ExampleApplication`. Quest'ultima fa parte dell'SDK di OGRE e ha lo scopo di favorire l'apprendimento del motore grafico. Essa si fa carico di alcuni dei compiti più complessi, come ad esempio l'inizializzazione di OGRE, e mette a disposizione un ambiente 3D precostituito. Inoltre, `ExampleApplication` si occupa anche della gestione degli input dell'utente. Questo è il motivo per cui, in fase di creazione del programma, abbiamo specificato al compilatore di fare il linking alla libreria `OIS_d.lib`.

L'utilizzo della classe `ExampleApplication` rende quindi la curva di apprendimento di OGRE molto più morbida e ci consente di introdurre un singolo concetto per volta. Nelle sezioni successive rimpiazzeremo, di volta in volta, sempre più funzionalità della classe `ExampleApplication`.

Nel metodo principale `main` abbiamo creato un'istanza della classe `PrimiPassi` e ne abbiamo invocato il metodo `go`. Quest'ultimo, ereditato dalla classe `ExampleApplication`, si occupa di inizializzare OGRE e di mostrare la finestra in cui configurare le impostazioni del motore grafico.

2.1.4 Caricamento di un modello 3D

Allo stato attuale, la nostra applicazione non fa altro che mostrare una finestra completamente nera. Estendiamola aggiungendo all'implementazione del metodo `createScene` il codice del sorgente 2.2.

```
11     Entity* entity = mSceneMgr->createEntity("sinbad",
           "Sinbad.mesh");
12
13     mSceneMgr->getRootSceneNode()->attachObject(entity);
```

Sorgente 2.2: Caricamento di un modello 3D

Ricompiliamo, lanciamo ancora una volta l'applicazione e, al centro della finestra, apparirà adesso una piccola sagoma verde. La classe `ExampleApplication` mette a disposizione, all'interno dell'ambiente 3D, una camera che possiamo liberamente spostare e ruotare tramite i tasti `WASD` e il

Figura 2.3: Il modello 3D di Sinbad

mouse. Avviciniamoci quindi alla figura verde e scopriremo che si tratta di un orco vestito da pirata, come quello nella figura 2.3. È Sinbad, la mascotte di OGRE.

Per caricare e visualizzare questo modello 3D ci è bastato aggiungere due semplici righe di codice. Da un lato ciò ci lascia intuire le potenzialità di OGRE, dall'altro ci fa capire come l'utilizzo di un motore grafico possa notevolmente semplificare la creazione di applicazioni 3D.

`createScene` è un metodo virtuale della classe `ExampleApplication` che viene richiamato durante la creazione della scena 3D. È quindi il posto ideale in cui aggiungere gli oggetti da visualizzare. Per fare questo, abbiamo creato un oggetto di tipo `Entity` tramite il metodo `createEntity`. Nella terminologia di OGRE, le entità rappresentano un qualsiasi oggetto renderizzabile. I parametri passati al metodo `createEntity` sono, rispettivamente, il nome assegnato all'entità e il modello 3D da caricare.

Nell'ultima riga abbiamo invece attaccato l'entità a un nodo della scena 3D. Senza entrare nei dettagli, per il momento ci basti sapere che, affinché OGRE effettui il rendering di un'entità, occorre che questa sia attaccata a un nodo della scena 3D (vedi 2.2.3 a pagina 28).

Manager

Notiamo che la creazione dell'entità non è avvenuta istanziando direttamente la classe `Entity`, bensì attraverso l'oggetto `mSceneMgr`. Questo è un membro della classe `ExampleApplication` di tipo `SceneManager`. Nello specifico, la classe `SceneManager` è quella responsabile di tutti gli aspetti legati alla gestione della scena 3D (vedi 2.3.1 a pagina 31). La creazione di tutti gli oggetti di scena, come ad esempio entità, camere e luci, avviene infatti tramite questa classe.

Tutte le classi il cui nome termina col suffisso `Manager` si basano sul design pattern Abstract Factory. Questa tecnica di programmazione è caratterizzata da due aspetti. Il primo è che centralizza la creazione di oggetti correlati fra loro, come ad esempio in OGRE dove tutti gli oggetti della scena 3D sono creati tramite la classe `SceneManager`. Il secondo, conseguenza diretta del primo, è invece che nasconde i dettagli dell'istanziazione di questi oggetti. Sfruttando quest'ultimo aspetto, OGRE riesce a implementare meccanismi di cache delle risorse molto avanzati. [12]

Il concetto di manager è molto utilizzato all'interno dell'API di OGRE. Per renderci conto di questo, basta osservare quante delle classi del motore grafico terminano col suffisso `Manager`.

Risorse

Nel nostro esempio abbiamo caricato il modello 3D `Sinbad.mesh`. È interessante chiederci dove OGRE va a recuperare questa risorsa. La risposta a questa domanda è nel file `resources_d.cfg`. All'interno di questo file, che si trova nella directory di lavoro specificata, esiste infatti una riga del tipo `Zip=../../media/packs/Sinbad.zip`. Estraendo l'archivio, vedremo che tra i file in esso contenuti ci sarà anche `Sinbad.mesh`.

Quando vogliamo caricare una mesh, o una qualsiasi altra risorsa, non è necessario preoccuparsi di dove essa sia collocata. È sufficiente infatti che questa si trovi all'interno di uno degli archivi o dei percorsi specificati nel file `resources.cfg` e provvederà in automatico OGRE a gestire il tutto. Per verificare che sia effettivamente così, proviamo a cancellare la riga `Zip=../../media/packs/Sinbad.zip` dal file `resources_d.cfg`. Lanciando nuovamente l'applicazione, essa ci restituirà un errore relativo al fatto che non riuscirà a trovare il file `Sinbad.mesh`.

2.2 Scena 3D e trasformazioni

All'interno di un motore grafico, col termine scena 3D si identifica l'insieme di tutti gli oggetti che contribuiscono al processo di rendering. La scena non è quindi costituita solamente da ciò che possiamo vedere, come ad esempio le mesh, ma anche da camere, luci e altro ancora. In OGRE, così come in tutti motori grafici, la scena 3D è strutturata e organizzata in modo da perseguire due scopi: ottimizzare, quanto più possibile, il processo di rendering e semplificare l'interazione con gli oggetti facenti parte della scena stessa. Estremizzando, possiamo dire che il ciclo principale di un'applicazione grafica è costituito da due fasi che si alternano continuamente: una prima in cui si configura la scena 3D e una seconda in cui viene effettuato il rendering di questa.

2.2.1 Nodi

Nella sezione precedente (vedi 2.1.4 a pagina 23) abbiamo creato un'applicazione che visualizzava il modello 3D di Sinbad. Ora, vedremo come ottenere lo stesso risultato, ma in maniera diversa. Sostituiamo all'implementazione del metodo `createScene` il codice del sorgente 2.3.

```
12    Entity* sinbadEntity = mSceneMgr->createEntity("Sinbad.mesh");
13
14    SceneNode* node = mSceneMgr->createSceneNode();
15    node->attachObject(sinbadEntity);
16
17    mSceneMgr->getRootSceneNode()->addChild(node);
```

Sorgente 2.3: Creazione di un nodo della scena 3D

Ricompiliamo, lanciamo nuovamente il programma e, come previsto, la mesh di Sinbad sarà ancora una volta al centro della finestra di fronte a noi.

Analogamente a quanto fatto nella sezione precedente, abbiamo creato dalla mesh di Sinbad l'entità corrispondente. Notiamo però che, questa volta, non abbiamo assegnato alcun nome a quest'oggetto. La specifica del nome di un'entità, così come per la maggior parte degli altri oggetti di OGRE, è infatti opzionale. In quest'ultima eventualità provvederà automaticamente il motore grafico a sceglierne uno al posto nostro. Se decidiamo invece di assegnare un nome a un oggetto, occorre che questo sia univoco all'interno della categoria di cui l'oggetto stesso fa parte. Ad esempio, non si possono avere né due entità né due luci con lo stesso nome. È invece possibile avere sia un'entità che una luce chiamate in modo uguale.

Nel programma creato nella sezione precedente avevamo invocato il metodo `attachObject` sull'oggetto restituito dal metodo `getRootSceneNode`. Questi due metodi servono, rispettivamente, ad attaccare un oggetto a un nodo, nel nostro caso un'entità, e a ottenere il nodo radice della scena 3D. In OGRE, un nodo rappresenta un punto di ancoraggio a cui è possibile attaccare vari oggetti, come ad esempio entità, luci, camere e animazioni. Tutti i nodi sono memorizzati in una struttura dati ad albero dove il nodo radice coincide con la sua origine. Possiamo pensare a questo nodo come a quello a cui è ancorata tutta la scena 3D.

Nella nostra applicazione, invece di utilizzare direttamente il nodo radice, abbiamo creato un nodo intermedio tramite il metodo `createChild`. Quest'ultimo restituisce un oggetto di tipo `SceneNode` che rappresenta un nodo della scena 3D. Infine, col metodo `addChild`, abbiamo aggiunto ai figli del nodo radice quello appena creato.

2.2.2 Sistemi di riferimento

Così come OpenGL, anche OGRE utilizza un sistema di coordinate destrorso. Ciò significa che, osservando lo schermo, l'asse x esce alla nostra destra, l'asse y esce verso l'alto e l'asse z esce, perpendicolarmente alla superficie del monitor, verso di noi.

Ogni nodo della scena 3D è caratterizzato da una posizione, un orientamento e una scala. La prima e la terza sono memorizzate in un vettore tridimensionale, mentre il secondo in un quaternione. Quest'ultimo è un concetto matematico che permette di rappresentare, in maniera compatta, qualsiasi rotazione attorno a un asse arbitrario. Per semplicità, possiamo però pensare che l'orientamento di un nodo sia definito tramite gli angoli di Eulero. OGRE utilizza le classi `Vector3` e `Quaternion` per la rappresentazione di vettori tridimensionali e quaternioni.

Quando si specifica la posizione e l'orientamento di un nodo occorre definire anche il sistema di riferimento. In OGRE, ne esistono tre tipologie: quello del mondo, quello locale e quello del padre. Il sistema di riferimento del mondo è univoco, fisso e immutabile. Si colloca idealmente al centro della scena e, in generale, coincide con la posizione e l'orientamento del nodo radice. Il sistema di riferimento locale è invece relativo e vincolato ai singoli nodi. Di conseguenza, le trasformazioni subite da un nodo si ripercuotono anche sul sistema di riferimento a esso associato. Inoltre, il sistema di riferimento locale di un nodo coincide col sistema di riferimento del padre dei nodi figli. Ad esempio, nella nostra applicazione il sistema di riferimento del padre di `node` è equivalente a quello locale del nodo radice.

2.2.3 Grafo della scena

Così come la maggior parte dei motori grafici, anche OGRE utilizza un grafo ad albero per memorizzare i nodi della scena 3D e tenere traccia dei loro legami di parentela. Il nodo radice rappresenta l'origine dell'albero, mentre gli oggetti di scena, attaccati ai vari nodi, rappresentano le foglie. Non esistono limiti né al numero di figli che un nodo può avere né alla profondità, all'interno dell'albero, a cui esso può essere collocato. Gli unici vincoli di questa struttura dati sono il fatto che un nodo può avere al massimo un solo padre e un solo oggetto di scena attaccato.

Oltre che per il processo di culling (vedi 2.3.3 a pagina 33), l'albero dei nodi viene utilizzato per sapere quali entità della scena 3D sono attive. Se esiste un cammino, diretto o meno, che permette di risalire da un nodo fino alla radice dell'albero, allora l'entità attaccata a quel nodo è considerata attiva. OGRE nel processo di rendering della scena 3D scarta tutte le entità inattive. Questo è il motivo per cui, nella nostra applicazione, abbiamo aggiunto tramite il metodo `addChild` il nodo `node` ai figli di quello radice.

La caratteristica principale del grafo della scena 3D è il fatto che qualsiasi trasformazione applicata a un nodo viene automaticamente propagata a tutti i suoi discendenti, diretti o meno. Questa funzionalità, apparentemente semplice e di poco conto, permette in realtà di strutturare in maniera intuitiva concetti molto complessi.

Consideriamo, ad esempio, il caso di un'automobile e del suo guidatore. Se vogliamo che le due entità si muovano assieme, basta impostare che il nodo del pilota sia figlio di quello della macchina. Così facendo, è come se avessimo creato un unico corpo rigido dove ogni spostamento e cambio di direzione della vettura viene trasmesso anche al pilota. Staccando i due nodi, le entità torneranno a essere di nuovo una indipendente dall'altra.

2.2.4 Trasformazioni

Per modificare la posizione, l'orientamento e la scala di un nodo esistono due tipologie di metodi: assoluti e relativi. Come si può facilmente intuire dai nomi, i primi non tengono conto dei valori correnti del nodo, mentre i secondi applicano un delta a questi ultimi. Inoltre, per i primi il sistema di riferimento rispetto al quale agirà la trasformazione non è modificabile, mentre per i secondi sì.

Assolute

Vediamo come utilizzare i metodi di trasformazione assoluti per modificare la posizione, l'orientamento e la scala di un nodo della scena 3D. Aggiungiamo il codice del sorgente 2.4 al metodo `createScene` della nostra applicazione.

```
19    mSceneMgr->setDisplaySceneNodes(true);
20
21    Entity* sinbadEntity2 = mSceneMgr->createEntity("Sinbad.mesh");
22
23    SceneNode* sinbadNode2 = node->createChildSceneNode();
24    sinbadNode2->setPosition(10, 0, 0);
25    sinbadNode2->setOrientation(Quaternion(Radian(Degree(-90)),
          Vector3::UNIT_Y));
26    sinbadNode2->setScale(1, 0.8, 1);
27    sinbadNode2->attachObject(sinbadEntity2);
```

Sorgente 2.4: Trasformazioni assolute di un nodo della scena 3D

Innanzitutto, abbiamo invocato il metodo `setDisplaySceneNodes` passandogli come parametro il valore `true`. Così facendo, OGRE visualizzerà il sistema di riferimento locale di ogni nodo della scena. Come possiamo vedere nella figura 2.4 nella pagina seguente, la freccia rossa corrisponde all'asse x, quella blu all'asse z e quella verde, che in questo caso è nascosta dalla mesh, all'asse y.

Dopo di che, abbiamo istanziato una nuova entità del modello 3D di Sinbad e l'abbiamo attaccata al nodo `sinbadNode2`. Questa volta, per creare il nodo abbiamo utilizzato il metodo `createChildSceneNode`. Questa funzione è molto comoda perché crea un nuovo nodo e contemporaneamente lo aggiunge ai figli del chiamante.

A seguire, tramite il metodo `setPosition`, abbiamo modificato la posizione di `sinbadNode2` spostandolo, rispetto al nostro punto di osservazione, a destra di 10 unità. Questa trasformazione agisce rispetto al sistema di riferimento del padre del nodo. Il riposizionamento di `sinbadNode2` avviene pertanto lungo l'asse x, ovvero la freccia rossa, del nodo più a sinistra rispetto al nostro punto di osservazione

La funzione `setOrientation` serve invece per modificare l'orientamento del nodo. A differenza del metodo precedente, questa trasformazione agisce rispetto al sistema di riferimento locale del nodo a cui è applicata. L'istanza della classe `Quaternion`, passata come parametro alla funzione, rappresenta un rotazione di $-90°$ attorno all'asse y. In questo modo, abbiamo fatto sì che la mesh attaccata a `sinbadNode2` sia rivolta verso quella attaccata a `node`.

Infine, col metodo `setScale`, abbiamo rimpicciolito del 20% il nodo `sinbadNode2` lungo la sua componente verticale. Così come la trasformazione

Figura 2.4: Il modello 3D di Sinbad spostato, ruotato e scalato tramite trasformazioni assolute

precedente, anche questa agisce rispetto al sistema di riferimento locale del nodo a cui è applicata. Naturalmente, modificare la scala di un nodo equivale a cambiare la scala della mesh a esso attaccata.

Nella figura 2.4 possiamo vedere il risultato delle trasformazioni applicate al nodo `sinbadNode2`. Il modello 3D più a sinistra è quello attaccato a `node`, ovvero quello originale di partenza, mentre quello più a destra è quello attaccato a `sinbadNode2`.

Relative

Vediamo come riportare il modello 3D di Sinbad al suo orientamento e alla sua dimensione originale tramite trasformazioni relative. Per farlo, aggiungiamo il codice del sorgente 2.5 all'implementazione del metodo `createScene`.

```
29      Entity* sinbadEntity3 = mSceneMgr->createEntity("Sinbad.mesh");
30
31      SceneNode* sinbadNode3 = sinbadNode2->createChildSceneNode();
32      sinbadNode3->translate(0, 0, -10, Node::TS_PARENT);
33      sinbadNode3->yaw(Radian(Degree(90)), Node::TS_LOCAL);
34      sinbadNode3->scale(1, 1.2, 1);
```

```
35      sinbadNode3->attachObject(sinbadEntity3);
```

Sorgente 2.5: Trasformazioni relative di un nodo

Analogamente a prima, abbiamo istanziato una nuova entità della mesh e l'abbiamo attaccata a un nodo figlio di `sinbadNode2`.

Ancora una volta, abbiamo spostato a destra di 10 unità, rispetto al nostro punto di osservazione, il modello di Sinbad appena creato. Per fare questo, abbiamo utilizzato il metodo `translate` i cui parametri sono, rispettivamente, il vettore di traslazione e il sistema di riferimento rispetto al quale la trasformazione agirà. In questo caso, abbiamo utilizzato il sistema di riferimento del padre e infatti, a causa del diverso orientamento di `sinbadNode2`, la traslazione avviene lungo la direzione negativa dell'asse z.

Dopo di che, tramite il metodo `yaw`, abbiamo applicato una rotazione di 90° attorno all'asse y del sistema di riferimento locale del nodo. Per la rotazione attorno agli assi x e z si usano invece gli analoghi metodi `pitch` e `roll`.

Per concludere, abbiamo riportato il modello di Sinbad alla sua dimensione originale. Notiamo che, per controbattere gli effetti della scalatura che era stata applicata al nodo padre `sinbadNode2`, abbiamo applicato un ingrandimento del 120%.

Nella figura 2.5 nella pagina successiva possiamo verificare che il modello 3D di Sinbad, quello più a destra, è tornato all'orientamento e alla dimensione originale.

2.3 Manager di scena

In un motore grafico, la gestione della scena 3D è uno degli aspetti più critici. Spesso, il fatto che un motore sia più performante, rispetto a un altro, è direttamente legato al maggior grado di ottimizzazione nella gestione della scena 3D. `SceneManager`, come si può intuire anche dal nome, è la classe di OGRE che si occupa di tutti questi aspetti. Oltre alla gestione del processo di creazione degli oggetti di scena (vedi 2.1.4 a pagina 25), questa classe di occupa anche di ottimizzare sia le trasformazioni applicate ai nodi (vedi 2.2.4 a pagina 28) che il processo di rendering.

2.3.1 SceneManager

La classe `SceneManager` è una delle più importanti di OGRE. Anche se finora l'abbiamo utilizzata solamente per creare nodi ed entità, le sue funzionalità vanno ben al di là di questo. Osservando la sua documentazione, possiamo

Figura 2.5: Il modello 3D di Sinbad riportato all'orientamento e alla dimensione originale tramite trasformazioni relative

infatti notare il grande numero di metodi che la classe mette a disposizione. Quelli che iniziano coi prefissi `create` e `destroy` servono, rispettivamente, a creare e distruggere oggetti di scena. I metodi il cui nome comincia per `set`, `get` e `has` servono invece a configurare e interrogare la scena 3D.

Uno dei compiti più importanti della classe `SceneManager`, anche se poco evidente all'utente, è quello di ottimizzare la gestione della scena 3D. Questo processo agisce sia sulla fase di rendering che su quella di trasformazione dei nodi. Ogni volta che viene applicata un trasformazione a un nodo questo viene marcato come mosso. Così facendo, al ciclo di rendering successivo, non è necessario ricalcolare posizione e orientamento di ogni nodo della scena. Inoltre, la classe `SceneManager` si occupa anche di fare il culling degli oggetti, evitando così inutili renderizzazioni.

L'implementazione predefinita della classe `SceneManager` si chiama `OctreeSceneManager`. Il suo nome deriva dal fatto che, per memorizzare il grafo ad albero della scena (vedi 2.2.3 a pagina 28), utilizza una struttura dati chiamata per l'appunto octree. Quest'implementazione non è l'unica disponibile in OGRE, ma è quella più utilizzata perché fornisce buone prestazioni nella maggior parte dei casi.

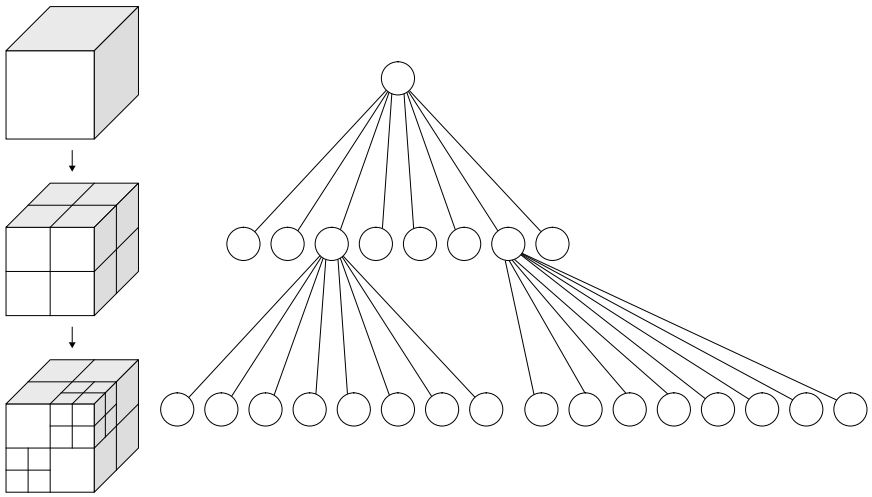

Figura 2.6: Rappresentazione grafica di un octree

2.3.2 Octree

Un octree, così come anche il nome suggerisce, è una struttura dati ad albero dove ogni nodo o non ha figli, nel caso sia una foglia, oppure ne ha otto. Questa caratteristica fa sì che l'octree si adatti molto bene a memorizzare gli oggetti di una scena 3D.

L'algoritmo per rappresentare una scena 3D con un octree è il seguente. Supponiamo di avere un cubo all'interno del quale sono racchiusi tutti gli oggetti della scena. Se dividiamo questo parallelepipedo a metà della sua larghezza, altezza e profondità, si formeranno otto nuovi cubi ognuno dei quali conterrà un ottavo della scena 3D. Possiamo pensare a questi come agli otto figli del parallelepipedo di partenza. Dei nuovi cubi creati, prendiamo solamente quelli che contengono almeno due oggetti della scena 3D e continuiamo applicando a loro il processo di suddivisione.

Dopo qualche iterazione, questo algoritmo si fermerà perché tutti i cubi creati conterranno uno o nessun oggetto della scena 3D. Questa caratteristica dell'octree renderà il processo di culling estremamente veloce.

Nella figura 2.6 possiamo vedere la rappresentazione grafica di un octree.

2.3.3 Culling

Il processo di culling serve a scartare tutte le geometrie che sono al di fuori dell'area inquadrata dalla camera. In questo modo, si ottimizza il processo di rendering della scena 3D, limitandolo solamente agli oggetti visibili.

Sfruttando le caratteristiche dell'octree e il modo in cui la scena 3D viene mappata al suo interno, OGRE utilizza un algoritmo di culling molto semplice e veloce. L'approccio utilizzato è simile a quello della ricerca negli alberi binari, con la differenza che, in questo caso, ogni nodo ha otto figli invece di due. Già dopo poche iterazioni, l'algoritmo riesce infatti a scartare buona parte della scena 3D non inquadrata dalla camera.

L'algoritmo di culling funziona in questo modo. Si considerano gli otto figli della radice dell'octree e si controlla se il cubo a essi associato è all'interno dell'area inquadrata dalla camera o meno. Si possono verificare tre casi: il cubo è completamente all'interno, completamente all'esterno o parzialmente incluso nell'area inquadrata. In entrambi i primi due casi l'algoritmo si ferma e non scende nei due sottoalberi associati. Questo perché, per come sono stati costruiti i cubi, anche i nodi figli ricadranno nella stessa casistica del padre. Se il cubo è completamente all'interno dell'area inquadrata dalla telecamera, allora tutti gli oggetti di scena in esso contenuti vengono renderizzati. In caso contrario gli oggetti vengono scartati. Infine, per i cubi che sono parzialmente dentro l'area inquadrata si ripete l'algoritmo per ognuno dei suoi figli.

2.4 Camere, luci e ombre

Così come un set cinematografico non è composto solamente dagli attori, così anche un scena 3D non è formata da sole entità. Altri due componenti fondamentali sono le luci e le camere. Le prime permettono di dare profondità e colore alla scena, le seconde definiscono invece l'aspetto con cui quest'ultima sarà visualizzata.

2.4.1 Creazione di un piano

Prima di aggiungere luci e altri elementi alla scena 3D, conviene creare un piano su cui questi oggetti possano proiettare la loro ombra. Inoltre, anche se la superficie non è rilevante ai fini dei calcoli di illuminazione, essa ci permetterà comunque di osservare meglio le varie tipologie di luci e come queste interagiscono con l'ambiente.

Per creare un piano riprendiamo il template usato per ogni applicazione sviluppata finora (vedi sorgente 2.1 a pagina 22), assicuriamoci che il metodo `createScene` sia vuoto e aggiungiamo, come sua implementazione, il codice del sorgente 2.6.

```
12      Plane plane(Vector3::UNIT_Y, -10.0);
13
```

```
14    MeshManager::getSingleton().createPlane("Plane.mesh",
          ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
          1000.0, 1000.0, 20, 20, true, 1, 2.0, 2.0, Vector3::UNIT_Z);
15
16    Entity* planeEntity = mSceneMgr->createEntity("plane",
          "Plane.mesh");
17    planeEntity->setMaterialName("Examples/BeachStones");
18
19    SceneNode* planeNode =
          mSceneMgr->getRootSceneNode()->createChildSceneNode();
20    planeNode->attachObject(planeEntity);
```

Sorgente 2.6: Creazione di un piano

Definizione di un piano

L'istanza della classe `Plane` contiene la definizione di un piano nello spazio 3D. Per definire un piano bastano due parametri: il vettore normale, ovvero la perpendicolare alla sua superficie, e lo scostamento lungo questo asse. Nel nostro caso abbiamo definito un piano parallelo al terreno, ma spostato verso il basso di 10 unità.

MeshManager

Utilizzando la classe `MeshManager`, abbiamo creato una mesh del piano appena definito. I primi tre parametri del metodo `createPlane` specificano rispettivamente il nome della mesh, il gruppo di risorse a cui questa appartiene e la definizione del piano utilizzata per la sua costruzione.

Il gruppo di risorse ha più o meno la stessa funzione dei namespace nel linguaggio C++. Da un lato permette di avere, in gruppi diversi, risorse con lo stesso nome, dall'altro permette di velocizzare il loro caricamento. Possiamo infatti specificare, in fase di istanziazione della risorsa, che questa venga ricercata solamente all'interno di un certo gruppo. Nel nostro esempio abbiamo associato la mesh del piano al gruppo di default `DEFAULT_RESOURCE_GROUP_NAME`.

I successivi quattro parametri definiscono la dimensione del piano, in questo caso un quadrato di 1000 unità, e da quanti segmenti deve essere formato ogni lato. OGRE costruisce i piani tramite una serie di triangoli affiancati l'uno all'altro. Se ad esempio per i parametri sei e sette avessimo scelto il valore 1, allora il nostro piano sarebbe stato formato solamente dai due triangoli che si creano suddividendolo lungo la diagonale.

Il parametro otto specifica al motore grafico di generare le normali di ogni triangolo che costituisce il piano. Questi vettori sono fondamentali affinché la luce sia riflessa in maniera corretta dalla superficie.

Infine, i restanti parametri definiscono alcune informazioni sulle coordinate texture del piano e il suo vettore di up.

La classe `MeshManager`, così come tutti gli altri manager di OGRE, utilizza il design pattern singleton. Questa tecnica di programmazione è utilizzata per due motivi. Il primo è che garantisce, in ogni momento del ciclo di vita dell'applicazione, che ci sia una e una sola istanza della classe. La seconda motivazione è invece che permette di accedere a questa istanza da qualsiasi punto del programma, evitando quindi di spargere nel codice inutili riferimenti all'oggetto. Questo design pattern ben si sposa con il concetto di manager, per cui sono importanti sia l'univocità che la facilità ad accedervi. In OGRE, il metodo statico `getSingleton` serve appunto per ottenere l'unica istanza di un manager. [12]

Creazione dell'entità

Come in tutte le applicazioni precedenti, abbiamo creato un'entità e, per fare in modo che questa sia renderizzata, l'abbiamo attaccata a un nodo della scena 3D.

L'entità creata corrisponde a un'istanza della mesh `Plane.mesh`. Questa non è però una delle risorse esterne caricate da OGRE, bensì la mesh del piano creata precedentemente tramite la classe `MeshManager`. Possiamo infatti notare che `Plane.mesh` corrisponde al valore del primo parametro del metodo `createPlane`. Quello che abbiamo fatto, tramite quest'ultima funzione, è quindi creare una sorta di risorsa virtuale definita solamente nella memoria dell'applicazione. OGRE non fa distinzioni tra modelli 3D caricati da file esterni e mesh definite via codice. Ciò permette quindi all'utente di interagire con entrambe in maniera trasparente e univoca.

Una novità di questo programma, rispetto a quelli sviluppati finora, è l'invocazione del metodo `setMaterialName`. Tramite questa funzione, abbiamo associato alla mesh del piano uno dei materiali (vedi 2.6 a pagina 48) predefiniti dell'SDK di OGRE, quello identificato dal nome `Examples/BeachStones`. Questo materiale applica una texture all'entità a cui viene associato e abbassa, quasi azzerandola, la quantità di luce ambientale che essa riflette.

2.4.2 Luci

Se provassimo a eseguire il programma adesso, non riusciremmo a vedere granché. La mesh del piano sarebbe molto scura e poco illuminata. Po-

tremmo pensare che ciò sia causato dall'assenza di luci nella scena, ma non è questo il vero motivo. Infatti, anche nelle precedenti applicazioni non avevamo aggiunto nessuna luce, eppure il modello di Sinbad era ben visibile. Questo perché, nella scena 3D, esiste una luce ambientale di fondo che illumina uniformemente ogni oggetto presente. Applicando però il materiale `Examples/BeachStones` al piano, abbiamo ridotto sensibilmente la sua riflettività a questo tipo di luce, rendendolo di fatto poco illuminato.

Per rimediare possiamo cambiare la sensibilità del materiale alla luce ambientale o, più semplicemente, aggiungere qualche fonte d'illuminazione alla nostra scena. Così come OpenGL, anche OGRE mette a disposizione tre tipologie di luci: point, spot e direzionali. Il motore grafico supporta un numero virtualmente infinito di luci nella scena, ma noi ci limiteremo ad aggiungerne, di volta in volta, solamente una di ogni tipo.

Point

Le luci point illuminano in ogni direzione tutto ciò che le circonda. Nel mondo reale, possiamo paragonare a questa fonte d'illuminazione il bulbo di una lampadina.

Per impostare un luce di questo tipo nella scena 3D, modifichiamo il metodo `createScene` aggiungendo il codice del sorgente 2.7.

```
22    Light* point = mSceneMgr->createLight("point");
23    point->setType(Light::LT_POINT);
24    point->setPosition(0.0, 20.0, 0.0);
25    point->setDiffuseColour(1.0, 1.0, 1.0);
```

Sorgente 2.7: Creazione di una luce point

L'oggetto `point`, istanza della classe `Light`, rappresenta una luce di OGRE. Così come per gli altri oggetti di scena, anche la luce non può essere istanziata direttamente, ma deve essere creata tramite lo scene manager.

Il metodo `setType` è autoesplicativo e serve a impostare il tipo di luce, in questo caso point.

Con il metodo `setPosition` si imposta invece la posizione della luce all'interno della scena 3D. Le coordinate sono relative al sistema di riferimento del mondo. A differenza di quanto avviene con le entità, perché una luce sia attiva non è necessario che questa sia attaccata a un nodo della scena. Questa incongruenza, mai risolta per motivi di retrocompatibilità, risale alle primissime versioni di OGRE in cui non erano stati ancora formalizzati i concetti di nodo e scena 3D (vedi 2.2 a pagina 26). Nulla ci vieta però di attaccare una luce a un nodo, ma per semplicità abbiamo preferito non farlo.

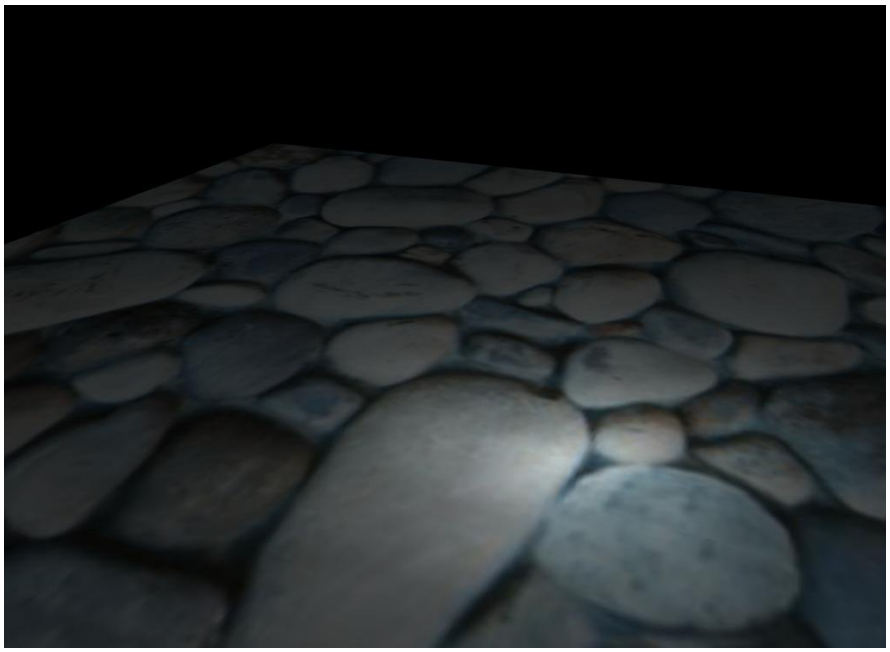

Figura 2.7: Una luce point

Infine, il metodo `setDiffuseColour` imposta il colore della componente diffusa della luce. I tre parametri rappresentano rispettivamente, in una scala da 0 a 1, la componente rossa, verde e blu della luce emessa. Nel nostro caso la fonte di illuminazione emetterà una semplice luce bianca. Analogamente, esiste anche un metodo, `setSpecularColour`, per impostare il colore della componente speculare della luce.

Nella figura 2.7 possiamo finalmente vedere gli effetti della luce appena creata sul nostro piano.

Spot

La sorgenti luminose spot sono invece luci orientabili che illuminano la scena solamente lungo una certa direzione. Queste infatti proiettano un cono di luce in cui l'intensità è tanto maggiore quanto più si è al centro di tale cono. Possiamo pensare a questa tipologia di luci come a quelle utilizzate ad esempio durante i concerti.

Commentiamo, all'interno del metodo `createScene`, il codice relativo alla luce point e aggiungiamo quello del sorgente 2.8 per impostare una luce di tipo spot.

```
27    Light* spot = mSceneMgr->createLight("spot");
28    spot->setType(Light::LT_SPOTLIGHT);
```

```
29      spot->setPosition(0.0, 100.0, 0.0);
30      spot->setDirection(0, -1.0, 0.0);
31      spot->setDiffuseColour(ColourValue(0.0, 1.0, 0.0));
32      spot->setSpotlightInnerAngle(Degree(5.0));
33      spot->setSpotlightOuterAngle(Degree(45.0));
```

Sorgente 2.8: Creazione di una luce spot

Tralasciamo i metodi utilizzati precedentemente, le cui descrizioni e funzionalità rimangono valide anche in questo caso, e concentriamoci su quelli specifici delle luci spot.

Il metodo `setDirection` imposta la direzione lungo la quale viene proiettato il cono di luce. Nel nostro caso la luce è orientata verso il basso lungo l'asse y.

Con i metodi `setSpotlightInnerAngle` e `setSpotlightOuterAngle` si impostano rispettivamente gli angoli di apertura del cono interno ed esterno della luce. Nel cono interno l'intensità della luce è massima, mentre in quello esterno decresce progressivamente mano a mano che ci si avvicina ai bordi. Gli oggetti che vengono a trovarsi al di fuori del cono esterno non ricevono nessun tipo di illuminazione.

Infine, notiamo che nel metodo `setDiffuseColour` invece di utilizzare tre parametri per definire il colore, come nel caso della luce point, abbiamo usato un unico oggetto di tipo `ColourValue`. In OGRE, le istanze di questa classe sono utilizzate per gestire e rappresentare i colori.

Compiliamo il programma, lanciamolo e dovremmo vedere un cerchio verde sul piano, risultato della proiezione del cono di luce. In realtà, vedremo invece qualcosa di più simile a un esagono, come quello nella parte sinistra della figura 2.8 nella pagina successiva. Ciò è causato dalla bassa risoluzione della mesh del piano. Nel metodo `createPlane` della classe `MeshManager` abbiamo infatti impostato che ogni lato del piano fosse suddiviso solamente in 20 segmenti. Se aumentiamo la risoluzione della mesh, portando questo valore a 200, ricompiliamo e lanciamo nuovamente l'applicazione, riusciremo effettivamente a vedere il cerchio di luce proiettato correttamente, come nella parte destra della figura 2.8 nella pagina seguente.

Direzionali

Entrambe le tipologie di luci presentate finora, point e spot, sono accomunate dal fatto che il fascio di raggi luminosi ha origine in un solo punto e si propaga, a raggiera, in più direzioni. Nelle luci direzionali esistono invece infiniti punti di origine e i raggi luminosi si propagano, paralleli fra loro, in un'unica direzione. L'effetto di una luce direzionale è quindi paragonabile al-

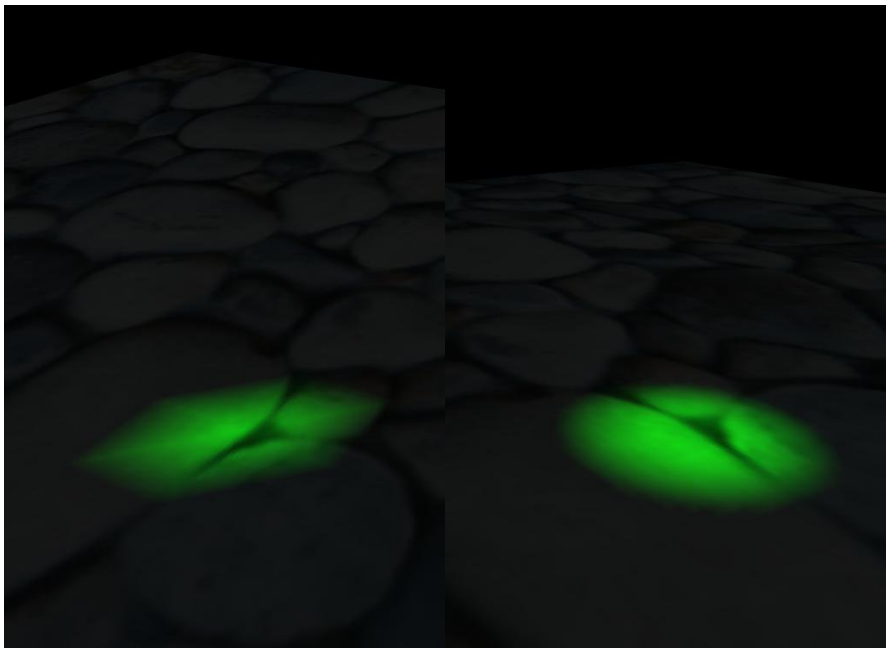

Figura 2.8: Una luce spot

l'illuminazione solare, in cui la dimensione della sorgente luminosa è talmente grande da considerarne i raggi paralleli.

Analogamente a quanto fatto precedentemente, prima di impostare una luce direzionale commentiamo il codice relativo alla luce spot. Aggiungiamo poi quello del sorgente 2.9.

```
35    Light* directional = mSceneMgr->createLight("directional");
36    directional->setType(Light::LT_DIRECTIONAL);
37    directional->setDirection(1.0, -1.0, 0.0);
38    directional->setDiffuseColour(ColourValue(1.0, 1.0, 1.0));
```

Sorgente 2.9: Creazione di una luce direzionale

Essendo infiniti i punti da cui hanno origine i raggi luminosi, nelle luci direzionali non è necessario definirne la posizione. Nel codice infatti abbiamo impostato solamente la direzione e il colore della luce.

Nella figura 2.9 a fronte possiamo osservare gli effetti della luce direzionale sul piano. Quest'ultimo appare molto più luminoso rispetto al caso in cui abbiamo utilizzato una luce point. Ciò è dovuto al fatto che l'intensità della luce direzionale non diminuisce all'aumentare della distanza dalla sorgente luminosa. Questo non è però vero per le luci point e spot dove oggetti più lontani sono meno illuminati. Possiamo renderci conto della cosa osservando i bordi del piano nella figura 2.7 a pagina 38.

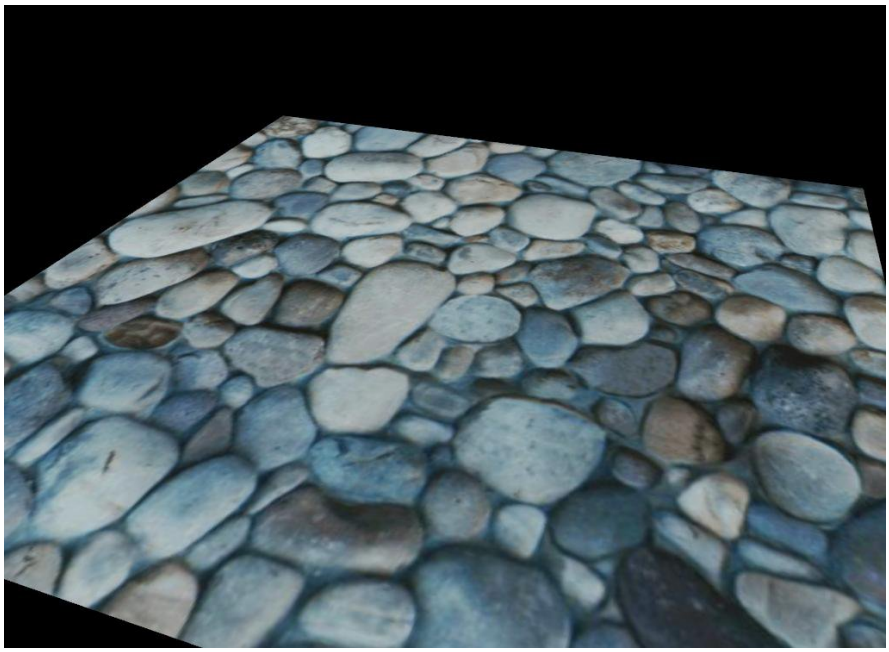

Figura 2.9: Una luce direzionale

Il fattore di attenuazione delle luci point e spot può essere regolato con il metodo `setAttenuation`.

2.4.3 Ombre

Ora che abbiamo visto come aggiungere luci alla scena 3D, è giunto il momento di attivare il rendering delle ombre. Prima di fare questo occorre però che venga caricato qualche oggetto che possa proiettare la sua ombra. Aggiungiamo quindi, al metodo `createScene`, il codice del sorgente 2.10.

```
40    Entity* sinbadEntity = mSceneMgr->createEntity("sinbad",
          "Sinbad.mesh");
41
42    SceneNode* sinbadNode =
          mSceneMgr->getRootSceneNode()->createChildSceneNode();
43    sinbadNode->translate(0.0, 136.0, 0.0);
44    sinbadNode->scale(30.0, 30.0, 30.0);
45    sinbadNode->attachObject(sinbadEntity);
46
47    mSceneMgr->setShadowTechnique(SHADOWTYPE_STENCIL_ADDITIVE);
```

Sorgente 2.10: Generazione delle ombre

Nella prima parte non c'è nulla di nuovo. Dopo aver istanziato la mesh di Sinbad, l'abbiamo riposizionata, scalata e infine attaccata a un nodo della scena.

Nell'ultima riga del nostro codice abbiamo invece attivato la generazione delle ombre. Il parametro del metodo `setShadowTechnique` identifica quale tecnica il motore grafico utilizzerà per il rendering delle ombre. OGRE ne mette a disposizione diverse, ognuna caratterizzata da un certo grado di accuratezza e costo computazionale. La tecnica utilizzata nel nostro esempio, basata sullo stencil buffer, è una delle più accurate, ma nel contempo anche una delle più costose. Per generare le ombre questa tecnica infatti esegue un ciclo di rendering in più per ogni luce presente nella scena. In generale, una resa accurata si accompagna sempre a un alto costo computazionale.

La gestione delle ombre è una delle sfide più difficili e complesse nel campo della grafica 3D real time. Esistono decine di libri sull'argomento e ogni anno vengono proposte nuove tecniche e soluzioni per ottenere ombre sempre più realistiche. Effetti avanzati come penombra e ombre dinamiche richiedono però un notevole lavoro artistico, di programmazione e di ottimizzazione. Spesso occorre gestire le ombre in maniera diversa a seconda del contesto e degli oggetti presenti nella scena 3D. Tutto ciò va ben al di là delle responsabilità di un motore grafico. OGRE infatti si limita a fornire tecniche più standard e meno realistiche, ma adatte alla maggior parte dei casi.

Nonostante questo, forse per la prima volta ci accorgiamo veramente delle potenzialità del motore grafico. Un compito difficile e complesso come il rendering delle ombre, che con OpenGL o Direct3D avrebbe richiesto centinaia di righe di codice, in OGRE può essere portato a termine con una semplice istruzione.

Nella figura 2.10 nella pagina successiva possiamo vedere il modello di Sinbad e l'ombra proiettata. Notiamo che un effetto non banale come quello del self shadowing, ovvero il fatto che la mesh proietti la sua ombra su sé stessa, viene gestito in maniera del tutto automatica da OGRE.

2.4.4 Camere e viewport

Fino a questo momento, abbiamo sempre utilizzato la camera messa a disposizione dalla classe `ExampleApplication`. Ora invece rimpiazzeremo quest'ultima utilizzando una camera creata da noi. Inoltre, creeremo anche una viewport da associare alla nuova camera. Per fare tutto ciò dobbiamo aggiungere due nuovi metodi alla nostra classe: `createCamera` e `createViewports`. Anche questi due, così come `createScene`, sono metodi virtuali della classe `ExampleApplication`.

Figura 2.10: Ombre

La viewport è la superficie, che in genere coincide con l'area della finestra dell'applicazione, sui cui viene mostrato il risultato del processo di rendering. La camera rappresenta invece il punto di vista dell'osservatore, all'interno della scena 3D, da cui viene effettuato il rendering. In OGRE, non ci sono limitazioni al numero di viewport e camere che è possibile creare e utilizzare. In ogni istante però sono attive solamente le camere che hanno una viewport associata.

Camera

Aggiungiamo quindi alla nostra classe il metodo `createCamera` come quello del sorgente 2.11

```
50  void createCamera()
51  {
52    mCamera = mSceneMgr->createCamera("camera");
53    mCamera->setPosition(0.0, 500.0, -500.0);
54    mCamera->lookAt(0.0, 0.0, 0.0);
55    mCamera->setFOVy(Degree(45));
56    mCamera->setNearClipDistance(5.0);
57    mCamera->setFarClipDistance(1000.0);
```

```
58  }
```

Sorgente 2.11: Creazione di una camera

Così come per luci ed entità, anche la creazione di una camera viene fatta attraverso lo scene manager. L'oggetto di tipo `Camera`, restituito dall'invocazione del metodo `createCamera` dell'`mSceneMgr`, è stato associato al membro `mCamera` della classe `ExampleApplication`. In questo modo riusciremo ancora a sfruttare alcune delle funzionalità messe a disposizione da quest'ultima classe, come ad esempio la possibilità di muovere liberamente la camera all'interno della scena 3D.

Il discorso fatto prima sull'interazione tra luci e nodi (vedi 2.4.2 a pagina 37), rimane valido anche per le camere. Anche queste infatti possono essere attaccate ai nodi, ma ciò, a differenza di quanto avviene ad esempio con le entità, non è necessario per il loro funzionamento. Nel nostro esempio abbiamo messo la camera in una posizione sopraelevata alle spalle della mesh di Sinbad.

Con il metodo `lookAt` abbiamo invece impostato il punto inquadrato dalla camera, in questo caso la base del modello 3D. Avremmo potuto utilizzare anche altri metodi per orientare la camera, come ad esempio `setDirection`, `setOrientation` o `rotate`, ma se non ci sono particolare esigenze, come in questo caso, l'utilizzo del metodo `lookAt` è più semplice e immediato.

Infine, con i metodi `setFOVy`, `setNearClipDistance` e `setFarClipDistance`, abbiamo configurato l'aspetto e le dimensioni della piramide di vista della camera. Il metodo `setFOVy` imposta l'angolo di apertura di quest'ultima, mentre i metodi `setNearClipDistance` e `setFarClipDistance` impostano la distanza dei piani di clip. Di tutti gli oggetti esterni alla piramide di vista o non compresi tra i due piani di clip viene fatto il culling, ovvero vengono scartati dalla fase di rendering.

Viewport

Completiamo il programma aggiungendo alla nostra classe anche il metodo `createViewports` come quello del sorgente 2.12

```
60  void createViewports()
61  {
62    Viewport* viewport = mWindow->addViewport(mCamera);
63    viewport->setBackgroundColour(ColourValue::White);
64
65    mCamera->setAspectRatio(Real(viewport->getActualWidth()) /
          Real(viewport->getActualHeight()));
```

Figura 2.11: Una camera e la sua viewport

```
66    }
```

Sorgente 2.12: Creazione di una viewport

La creazione di una viewport viene fatta tramite un riferimento alla classe `RenderWindow`. Nel nostro caso questo è rappresentato dal membro `mWindow` della classe `ExampleApplication`. Senza entrare nei dettagli, possiamo pensare a questo oggetto come alla finestra dell'applicazione. Il metodo `addViewport` esegue due compiti contemporaneamente. Da un lato aggiunge una viewport restituendone il riferimento come valore di ritorno. Dall'altro associa anche una camera alla viewport creata, nel nostro caso quella impostata precedentemente. La camera associata a una viewport può essere cambiata in qualsiasi momento tramite il metodo `setCamera`.

Con il metodo `setBackgroundColour` si imposta invece il colore di sfondo della viewport. Nella figura 2.11, oltre alla nuova posizione della camera, possiamo infatti notare che il colore attorno al piano è diventato bianco.

Infine, dopo aver creato una viewport, o comunque ogni volta vengono modificate le sue dimensioni, occorre impostare il rapporto d'aspetto della camera associata. Questo può essere fatto tramite il metodo `setAspectRatio`. Se ci si dimentica quest'ultimo passaggio, si rischia che il rendering della scena 3D risulti deformato.

2.5 Ciclo di rendering

Tutte le applicazioni che abbiamo sviluppato finora sono accomunate dal fatto di presentare una scena 3D statica. Questo perché il metodo `createScene` viene invocato solamente un volta, prima dell'inizio del ciclo di rendering. Utilizzando la classe `FrameListener` è possibile definire funzioni che vengono richiamate tra il rendering di un frame e l'altro. Nell'implementazione di queste funzioni è quindi possibile dare vita alla scena.

2.5.1 I listener

Le classi listener, assieme ai manager (vedi 2.1.4 a pagina 25), sono l'altro tratto caratteristico dell'API di OGRE. Dando un rapido sguardo alla documentazione del motore grafico, possiamo infatti vedere che dopo `Manager` il suffisso più utilizzato è `Listener`. Come anche il nome suggerisce, i listener sono oggetti che restano in ascolto del verificarsi di certi eventi, come ad esempio il caricamento di una risorsa, lo spostamento di un nodo o l'aggiornamento di una camera.

Tutte le classi listener si basano sul design pattern Observer. Questa tecnica di programmazione consiste in uno o più oggetti, gli osservatori, che vengono registrati presso l'oggetto da osservare. Quest'ultimo poi, al verificarsi di eventi o cambiamenti di stato, informerà di quanto accaduto tutti gli osservatori registrati. In OGRE, l'oggetto osservato è il motore grafico stesso, mentre gli osservatori sono le istanze delle varie classi listener. [12]

Il design pattern Observer, ampiamente utilizzato nello sviluppo di GUI[4], permette di creare complessi sistemi basati su eventi. Potremmo, ad esempio, sfruttare il listener dei nodi della scena 3D per creare un semplice sistema di rilevamento di collisioni.

2.5.2 FrameListener

La classe `FrameListener` è il listener che riceve le notifiche relative al processo di rendering della scena 3D. È di gran lunga il listener più utilizzato perché permette, all'applicazione, di modificare la scena 3D tra il rendering di un frame e l'altro.

Questa classe, così come tutti gli altri listener, è astratta e quindi non può essere né istanziata né utilizzata direttamente. Occorre invece creare un classe derivata tramite il meccanismo di ereditarietà. Nel sorgente 2.13 nella pagina successiva possiamo vedere l'interfaccia della classe `FrameListener` che è composta solamente da tre metodi virtuali.

[4] Graphical User Interface

```
1 virtual bool frameStarted(const FrameEvent& evt);
2 virtual bool frameRenderingQueued(const FrameEvent& evt);
3 virtual bool frameEnded(const FrameEvent& evt);
```

Sorgente 2.13: Interfaccia della classe FrameListener

Sfruttando il polimorfismo, OGRE invocherà poi i tre metodi della classe `FrameListener` per tutti i listener registrati. Il metodo `frameStarted` verrà invocato prima dell'inizio del processo di rendering. `frameRenderingQueued` sarà invece chiamato alla fine del processo di rendering, ma prima che avvenga lo scambio tra framebuffer e back buffer. Infine, un volta rimpiazzato il contenuto del framebuffer, verrà invocato `frameEnded`. Tutto questo si ripeterà, per tutta la durata dell'applicazione, ogni volta che sarà effettuato il rendering della scena 3D.

2.5.3 Animazione di una luce

Riprendiamo l'applicazione creata nella sezione precedente (vedi 2.4 a pagina 34) e vediamo come renderla un po' più viva. Quello che faremo sarà cambiare, durante il ciclo di rendering, l'orientamento della luce direzionale. Così facendo, sembrerà che questa stia ruotando attorno al modello 3D di Sinbad e vedremo l'ombra muoversi di conseguenza.

La prima cosa da fare è creare un listener estendendo la classe `FrameListener`. Aggiungiamo quindi, all'inizio del nostro programma, il codice del sorgente 2.5.3

```
4  class CicloRenderingFrameListener :
5    public FrameListener
6  {
7
8  public:
9
10   CicloRenderingFrameListener(SceneManager* sceneManager) :
11     mSceneManager(sceneManager),
12     mTime(0)
13   {
14   }
15
16   bool frameStarted(const FrameEvent& evt)
17   {
18     mTime += evt.timeSinceLastFrame;
19
20     Light* directional = mSceneManager->getLight("directional");
```

```
21      directional->setDirection(Math::Sin(mTime), -1.0,
            Math::Cos(mTime));
22
23      return true;
24    }
25
26  private:
27
28    SceneManager* mSceneManager;
29    Real mTime;
30  };
```

All'interno del costruttore di `CicloRenderingFrameListener` vengono inizializzati i membri della classe. `mSceneMgr`, come possiamo facilmente intuire, è un riferimento al manager della scena. `mTime` è invece un accumulatore in cui verrà salvato il tempo trascorso dall'inizio del ciclo di rendering.

La dichiarazione del metodo `frameStarted`, nella classe `CicloRenderingFrameListener`, serve per effettuare l'override dell'omonimo metodo della classe `FrameListener`. In questo modo, grazie al polimorfismo, quando OGRE invocherà il metodo `frameStarted` del listener verrà eseguito il codice della classe `CicloRenderingFrameListener`.

Il metodo `frameStarted` prende come parametro un oggetto di tipo `FrameEvent`. Si tratta di una semplice struttura formata solamente due campi: `timeSinceLastFrame` e `timeSinceLastEvent`. Il primo campo contiene il tempo trascorso dall'ultimo evento di rendering dello stesso tipo, in questo caso l'invocazione precedetente del metodo `frameStarted`. Il secondo contiene invece il tempo trascorso dall'ultimo evento di rendering qualsiasi esso sia, nel nostro caso l'ultima invocazione del metodo `frameEnded`.

Ad ogni invocazione del metodo `frameStarted`, il valore contenuto nel campo `timeSinceLastFrame` viene sommato a quello della variabile `mTime`. Quest'ultima viene poi passata, come parametro, alle funzioni `Sin` e `Cos` per cambiare la direzione della luce rispetto all'asse x e all'asse z. Utilizzando queste due funzioni matematiche, si otterrà un effetto simile a quello di una luce che ruota attorno al modello 3D di Sinbad.

Nella figura 2.12 a fronte possiamo vedere alcune immagini estratte da questa animazione.

2.6 Materiali

In OGRE, il concetto di materiale racchiude al suo interno tutti gli elementi legati alla visualizzazione di un oggetto. Con questo termine, non ci si riferi-

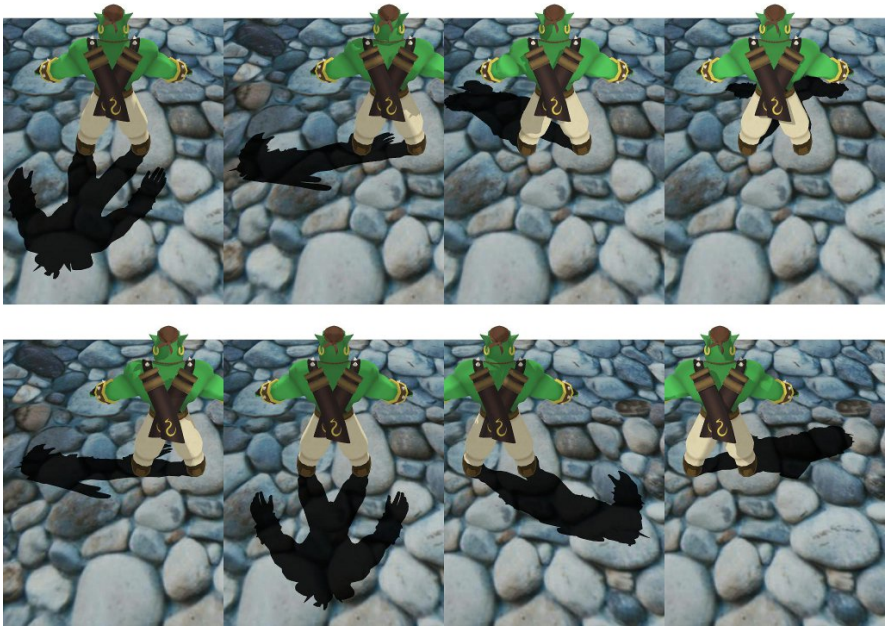

Figura 2.12: Animazione di una luce

sce solamente a impostazioni prettamente grafiche, come il colore dei vertici o la texture da utilizzare, ma anche ad aspetti più nascosti che però influenzano comunque il modo in cui verrà renderizzato l'oggetto, come ad esempio l'utilizzo o meno dello Z-buffer. In questo modo, da un unico punto è possibile configurare tutti i parametri che agiscono, direttamente o indirettamente, sull'aspetto finale di un oggetto.

2.6.1 Definizione tramite script

Un materiale può essere definito in due modi: via codice o tramite uno script. Quest'ultima modalità è quella predefinita per via della sua semplicità e praticità. La definizione dei materiali tramite script permette inoltre un maggior riutilizzo degli stessi.

Al pari delle altre risorse, anche gli script dei materiali vengono caricati automaticamente da OGRE durante la sua inizializzazione. Perché questo avvenga, occorre però che gli script abbiano estensione `.material` e siano collocati all'interno dei percorsi o degli archivi specificati nel file `resources.cfg` (vedi 2.1.4 a pagina 25).

All'interno di un singolo script possono essere definiti più materiali. La sintassi utilizzata è simile a quella del C++. Il codice del sorgente 2.14 nella pagina seguente mostra lo schema di definizione di un generico materiale.

```
material material_name
{
  technique
  {
    pass
    {
      attribute value
      attribute value

      texture_unit
      {
      }

      vertex_program_ref shader_name
      fragment_program_ref shader_name
    }
  }
}
```

Sorgente 2.14: Schema di definzione di un generico materiale

Ogni materiale è identificato da un nome, in questo caso la stringa `material_name`, che deve essere univoco tra tutti gli script definiti. Un materiale è composto da una o più sezioni `technique` che, a loro volta, contengono da una a sedici sezioni `pass`. All'interno dei vari `pass` è possibile definire attributi, shader e una o più sezioni `texture_unit`.

technique

Ogni sezione `technique` specifica una singola modalità di renderizzazione dell'oggetto a cui il materiale è associato. Anche se un materiale può essere composto da una sola `technique`, in genere se ne specificano almeno due. Da un lato ciò garantisce maggior compatibilità hardware, dall'altro permette di applicare la tecnica di LOD, oltre che alle geometrie, anche ai materiali.

Supponiamo, ad esempio, di aver definito un materiale, composto da una sola `technique`, che realizzi un effetto grafico estremamente avanzato utilizzando l'ultima specifica degli shader, la 3.0. Se la nostra scheda video supporta però gli shader solamente fino alla versione 2.0, allora la `technique` definita precedentemente verrà scartata e, di conseguenza, l'oggetto verrà renderizzato senza nessun materiale associato. Occorre quindi specificare una seconda `technique`, meno avanzata, che faccia da riserva alla prima nel caso questa non sia supportata dall'hardware del sistema.

Non ci sono limiti al numero di `technique` che possiamo definire in un materiale, ma ne verrà utilizzata sempre e soltanto una. Le `technique` vanno specificate nell'ordine di preferenza con cui vogliamo che vengano scelte. Di conseguenza, si definiscono prima gli effetti più complessi e avanzati e successivamente le `technique` di riserva.

pass

Ogni sezione `pass` corrisponde a un singolo processo di rendering dell'oggetto a cui il materiale è associato. I vari `pass` vengono eseguiti uno di seguito all'altro e, opportunamente combinati fra loro, permettono di ottenere effetti molto avanzati. Ovviamente, più `pass` vengono definiti più la `technique`, da un punto di vista computazionale, sarà costosa.

All'interno di un `pass`, oltre alle sottosezioni `texture_unit`, possiamo specificare una serie di attributi per configurare il processo di rendering. Alcuni dei più utilizzati sono:

`ambient` Imposta la riflettività del materiale alla luce ambientale.

`diffuse` Analogo all'attributo `ambient`, ma per la componente diffusa della luce.

`specular` Analogo all'attributo `ambient`, ma per la componente speculare della luce.

`emissing` Imposta la quantità di luce autonomamente emessa dal materiale.

`scene_blend` Nel caso di un materiale semitrasparente, imposta come questo verrà combinato col contenuto della scena sottostante.

`depth_write` Imposta se, durante il processo di rendering dell'oggetto a cui è associato il materiale, aggiornare i valori dello Z-buffer.

`depth_check` Specifica se effettuare il controllo di visibilità, dell'oggetto a cui è associato il materiale, nello Z-buffer.

`shading` Imposta il modello di shading utilizzato. È possibile scegliere tra flat, Gouraud e Phong.

`polygon_mode` Specifica il tipo di rasterizzazione dei poligoni dell'oggetto a cui il materiale è associato.

`vertex_program_ref` e `fragment_program_ref` servono, rispettivamente, per associare un vertex shader e un pixel shader al `pass` in cui vengono specificati. In questo caso, per non creare conflitti con gli shader, OGRE ignorerà però la maggior parte degli attributi dell'elenco precedente. `shader_name` è il nome con cui lo shader viene identificato all'interno delle risorse di OGRE.

texture_unit

Le sezioni `texture_unit` servono per definire le texture da utilizzare e alcuni loro parametri di configurazione. Questi, come nel caso dei `pass`, vengono specificati tramite una serie di attributi. Alcuni dei più utilizzati sono:

`texture` Il file immagine, che deve essere presente tra le risorse caricate da OGRE, da utilizzare come texture.

`colour_op` Imposta come la texture viene combinata col colore sottostante.

`tex_border_colour` Imposta il colore da assegnare al bordo della texture.

`tex_address_mode` Specifica come si comporta la texture nel caso le coordinate siano maggiori di 1.0.

`filtering` Imposta il filtro da applicare alla texture quando viene rimpicciolita o ingrandita.

Specificando più di una sezione `texture_unit` all'interno di un singolo `pass`, è possibile ottenere l'effetto chiamato multitexture, ovvero una serie di texture sovrapposte le une alle altre.

2.6.2 Applicazione di un materiale

In una delle sezione precedenti (vedi 2.4 a pagina 34), abbiamo già visto come applicare un materiale a un oggetto della scena 3D. Nello specifico, utilizzando la funzione `setMaterialName("Examples/BeachStones")`, avevamo applicato alla mesh di un piano il materiale chiamato come il parametro del metodo. Ora, definiremo tre nuovi materiali e, sfruttando la medesima applicazione, li applicheremo di volta in volta alla mesh del piano.

Per prima cosa, occorre creare un file `Materiali.material` all'interno del percorso `OgreSDK_vc9_v1-7-4/media/materials/scripts` della directory d'installazione dell'SDK di OGRE (vedi 2.1.1 a pagina 17. Siccome questo percorso corrisponde a uno di quelli specificati all'interno del file `resources.cfg`, tutti i materiali che andremo a definire nel file `Materiali.material` saranno automaticamente caricati da OGRE durante la sua fase d'inizializzazione.

Successivamente, basterà sostituire, di volta in volta, al parametero del metodo `setMaterialName` il nome del nuovo materiale, specificato all'interno del file `Materiali.material`, da applicare al piano.

Colore

Impostiamo il materiale definito nel sorgente 2.15 all'interno del file `Materiali.material`.

```
material ColoreRosso
{
  technique
  {
    pass
    {
      ambient 0.0 0.0 0.0
      diffuse 1.0 0.0 0.0
    }
  }
}
```

Sorgente 2.15: Materiale per colorare di rosso un oggetto

Questo materiale non fa altro che colorare di rosso l'oggetto a cui viene applicato. Ciò si ottiene impostando la riflettività del materiale alla luce diffusa con tale colore. I tre valori dopo l'attributo `diffuse` specificano infatti la componente rossa, verde e blu del colore che viene riflesso dal materiale a quel tipo di luce.

Nella figura 2.13 nella pagina seguente possiamo vedere il piano con il nuovo materiale applicato.

Texture

Ancora una volta, aggiungiamo il materiale definito nel sorgente 2.16 al file `Materiali.material`.

```
material TextureErba
{
  technique
  {
    pass
    {
      texture_unit
      {
        texture grass_1024.jpg
```

Figura 2.13: Il piano colorato di rosso

```
10          tex_address_mode clamp
11        }
12      }
13    }
14  }
```

Sorgente 2.16: Materiale per applicare una texture a un oggetto

Questo materiale applica invece una texture e specifica, nel caso le coordinate siano superiori a 1.0, di effettuare il clamp di quest'ultima.

Nella figura 2.14 a fronte possiamo vedere il piano con il nuovo materiale applicato. Notiamo che, siccome le coordinate del piano sono maggiori di 1.0, viene effettuato il clamp della texture.

Multitexture

Per l'ultima volta, aggiungiamo il materiale definito nel sorgente 2.17 al file `Materiali.material`.

```
1  material Multitexture
2  {
3    technique
4    {
5      pass
```

Figura 2.14: Una texture applicata al piano

```
6       {
7         texture_unit
8         {
9           texture terr_dirt-grass.jpg
10        }
11
12        texture_unit
13        {
14          texture ogrelogo.png
15        }
16      }
17    }
18  }
```

Sorgente 2.17: Materiale per ottenere un effetto multitexture

Questo materiale applica due texture all'oggetto a cui viene associato sovrapponendole l'una all'altra. Le texture vengono applicate nell'ordine con cui sono definite all'interno del `pass`. Di conseguenza, la prima specificata sarà quella che starà sotto a tutte le altre.

Nella figura 2.15 nella pagina seguente possiamo vedere il risultato ottenuto applicando il materiale `Multitexture` al piano. Come ci aspettavamo, la texture `ogrelogo.png` viene sovrapposta a `terr_dirt-grass.jpg`.

Figura 2.15: Il piano con due texture sovrapposte l'una all'altra

Capitolo 3

Victory: The Age of Racing

In questo capitolo vedremo come Vae Victis Srl, l'azienda in cui il sottoscritto lavora a partire dall'estate del 2008, ha utilizzato OGRE nello sviluppo del videogame Victory: the Age of Racing, che da questo punto in poi chiameremo semplicemente Victory. Innanzitutto, descriveremo brevemente la storia di Vae Victis e analizzeremo le cause che hanno portato l'azienda a scegliere OGRE per lo sviluppo del suo videogame (vedi 3.1). A seguire, descriveremo il gioco mostrando alcune delle sue caratteristiche principali e analizzeremo la piattaforma software su cui si basa (vedi 3.2 a pagina 60). Per concludere, mostreremo alcune parti del videogame, su cui il sottoscritto ha lavorato personalmente, e vedremo come OGRE viene utilizzato all'interno di queste (vedi 3.3 a pagina 62).

3.1 Vae Victis

Vae Victis Srl è una piccola azienda con sede a Diegaro, in provincia di Forlì-Cesena, che si occupa di sviluppo di videogame. Nasce ufficialmente alla fine del 2007 dalle ceneri di BlueLemon, ma la sua attività inizia in realtà ben prima. Sia i soci fondatori che il gruppo di lavoro di entrambe le aziende sono infatti gli stessi. [28]

3.1.1 Storia

La storia di BlueLemon comincia nel 1997 quando, approfittando del boom di internet in Italia, inizia a sviluppare siti web, applicazioni e a fornire servizi di grafica 3D. La parte multimediale prende ben presto il sopravvento su quella web, tanto che l'azienda inizia a produrre un cartone animato in 3D. Il trailer viene presentato, nel 2001, al festival dell'animazione di Berlino,

ma non ottiene il successo sperato. BlueLemon però non si scoraggia e, nello stesso anno, arriva subito l'occasione per rifarsi. L'azienda inizia lo sviluppo di Macio Net Race: un gioco online di corse motociclistiche. Questo progetto, su cui l'azienda continuerà a lavorare per altri tre anni, si aggiudica infatti importanti premi e riconoscimenti a livello nazionale. Le impressioni positive ricevute spingono i ragazzi di BlueLemon a proseguire nello sviluppo di applicazioni multimediali. All'inizio del 2005, l'azienda rilascia BlaBlaBar: una piattaforma online di chat 3D. Anche se molte delle sue caratteristiche sono diventate ormai comuni al giorno d'oggi, l'applicazione presentava idee e spunti molto innovativi per quel periodo. BlaBlaBar metteva a disposizione un ambiente virtuale 3D in cui era possibile creare e personalizzare il proprio avatar, stringere amicizie con altri utenti, creare contenuti da condividere con la comunità e ascoltare musica in streaming. Il salto di qualità avviene nel 2006 quando, forti dell'esperienza maturata in quasi dieci anni di lavoro, il team di BlueLemon inizia la progettazione di Victory, un vero e proprio videogame 3D. L'anno seguente, per rimarcare ancor di più il nuovo corso aziendale, tutte le risorse, le tecnologie e il know-how di BlueLemon vengono trasferiti all'interno di una nuova società: Vae Victis.

Dopo quasi due anni di sviluppo, nel 2008 il primo prototipo del gioco viene presentato alla più importante fiera, a livello europeo, dell'industria dei videogame: la Game Developer Conference di Parigi. Le ottime critiche ricevute convincono Vae Victis di aver intrapreso la strada giusta e infatti, a febbraio del 2009, viene sottoscritto un contratto di pubblicazione con K2 Network, uno tra i più importanti publisher americani. A metà gennaio del 2012, dopo oltre cinque anni di sviluppo, viene rilasciata al pubblico la prima versione di Victory.

3.1.2 Da OGRE a Virtools

Virtools è un middleware, sviluppato dall'azienda francese Dassault Systèmes, per la creazione di programmi 3D. Viene utilizzato prevalentemente nelle applicazioni di realtà virtuale e di visualizzazione interattiva. La sua filosofia, sebbene più generalista, ricorda molto da vicino quella di Unity (vedi 1.1.1 a pagina 6). Come quest'ultimo, anche Virtools mette a disposizione un unico ambiente integrato per lo sviluppo delle applicazioni. Quest'ambiente viene contemporaneamente utilizzato sia per creare la parte grafica che per gestire la logica del programma. Un'ulteriore caratteristica di Virtools, molto apprezzata, è la possibilità di programmare in maniera visuale. Ciò avviene manipolando graficamente degli elementi che rappresentano funzioni e moduli del programma. Collegando questi oggetti gli uni agli altri, si crea poi il flusso di esecuzione dell'applicazione. [1]

Figura 3.1: Uno screenshot di Victory

BlueLemon aveva utilizzato Virtools per la creazione del BlaBlaBar. L'azienda decide quindi, sfruttando l'esperienza acquisita con tale middleware, di impiegarlo anche nella creazione di Victory. Dopo circa un anno di lavoro, appare però chiara l'inadeguatezza di Virtools per la produzione del videogame. Le criticità più rilevanti si riscontrano nell'integrazione con libreria fisica e nelle comunicazioni via rete. Entrambi questi aspetti sono legati ad alcune delle caratteristiche di punta di Victory, ovvero un modello fisico estremamente accurato e complesso e una spiccata componente online del gioco, e non possono quindi essere trascurati. Ad esempio, la libreria fisica utilizzata da Victory, per funzionare correttamente, necessita che il suo ciclo di aggiornamento avvenga con una certa frequenza, almeno 300 volte al secondo, e che questa sia la più stabile possibile. In Virtools ciò non era possibile perché l'aumento della frequenza di aggiornamento andava a discapito della sua stabilità.

Constatata l'impossibilità di proseguire lo sviluppo del gioco utilizzando Virtools, Vae Victis decide quindi di rimpiazzare tale middleware con altre librerie. Per la parte di rendering 3D viene scelto il motore grafico OGRE. Le motivazioni alla base di questa scelta sono due: la licenza con cui viene reso disponibile il motore grafico e la sua flessibilità. Vae Victis è infatti una realtà molto piccola, formata da circa quindici dipendenti, e non ha a disposizione un budget tale da permettersi l'acquisto di un game engine (vedi 1.1.5 a pagina 9). L'azienda decide quindi di orientarsi su soluzioni più economiche, ma non per questo meno performanti. OGRE era infatti già stato utilizzato, da altre realtà, per progetti di una certa complessità e

di buon livello qualitativo. Inoltre, il fatto che OGRE sia scritto in C++ e gestisca solamente la parte di rendering 3D, permetterà a Vae Victis di utilizzarlo e integrarlo facilmente con le altre librerie del videogame.

3.2 Struttura del videogame

Victory è un progetto software vasto e complesso con funzionalità che vanno dalla simulazione fisica del modello di guida fino alla creazione di reti peer to peer per le partite online, passando per la gestione del comparto sonoro e dell'interfaccia utente. Il solo OGRE non è sufficiente a gestire tutti questi aspetti. Di conseguenza, per soddisfare ognuna di queste macro funzionalità viene utilizzata una libreria diversa che, unite e integrate fra loro, compongono un'unica piattaforma software, chiamata Numen.

3.2.1 Caratteristiche

Victory è un videogame di corse automobilistiche online. Scopo del gioco è gareggiare contro altri utenti cercando di ottenere quante più vittorie possibili. Alle fine di ogni gara, a seconda del piazzamento raggiunto, ogni utente riceve una serie di punti. Questi possono essere utilizzati per acquistare nuove automobili o per potenziare e personalizzare quelle che già si possiedono.

All'interno del videogame possiamo individuare due macro sezioni: una dove avvengono le gare vere e proprie, visibile nella figura 3.1 nella pagina precedente, e una, chiamata limbo, in cui si gestiscono tutti gli aspetti legati alla vettura. In quest'ultima sezione possiamo infatti creare nuove automobili, personalizzare il loro aspetto estetico e modificarne le caratteristiche tecniche. La sezione limbo, come possiamo anche vedere nella figura 3.2 nella pagina successiva, è facilmente riconoscibile per il fatto che mostra l'automobile poggiata al centro di un piano bianco.

Le due sezioni, oltre che per l'aspetto visivo, si differenziano l'una dall'altra anche per le caratteristiche e le necessità che hanno. Quella di gara, ad esempio, è sicuramente più interattiva, dinamica e frenetica. L'utente interagisce infatti con essa continuamente per modificare la direzione e la velocità della propria automobile. Inoltre, con la stessa frequenza, arrivano via rete analoghe informazioni riguardanti le vetture avversarie. Infine, rispetto al limbo, questa sezione necessita di alcuni sistemi aggiuntivi, come ad esempio il motore fisico o il generatore di particellari, per gestire il modello di guida delle automobili e alcuni effetti visivi come fumo e polvere.

Ognuna di queste due sezioni corrisponde ad una diversa scena 3D (vedi 2.2 a pagina 26), ma le loro differenze sono tali che vengono gestite però

Figura 3.2: Il limbo di Victory

da due diversi `SceneManager` (vedi 2.3 a pagina 31). Per la scena del limbo viene utilizzato il manager di scena standard, quello basato su Octree. Per la scena di gara, data la sua particolore conformazione, se ne utilizza invece uno specializzato nella gestione degli spazi aperti. [30]

3.2.2 Numen

Numen, la piattaforma software che è alla base di Victory, è composta da numerose librerie. Alcune delle più importanti sono:

JPEG[1] È la libreria che gestisce i file immagine dell'omonimo formato. Viene utilizzata sia per le texture che per il salvataggio di screenshot di gioco.

zlib È la libreria che gestisce gli archivi compressi contenenti le risorse del videogame.

Particle Universe È la libreria che gestisce gli effetti particellari utilizzati nel gioco, come ad esempio neve, fumo e polvere.

OIS[2] È la libreria che gestisce gli input dell'utente. È la stessa utilizzata dalla classe `ExampleApplication` di OGRE (vedi 2.1.3 a pagina 22).

blPhysics È la libreria che gestisce tutta la componente fisica del gioco. Modificata ad hoc per Victory, è stata utilizzata anche in altri videogame di corse automobilistiche e in simulatori semi-professionali di vetture di Formula 1.

Scaleform È la libreria che gestisce la GUI del videogame. Permette di creare e sviluppare l'interfaccia utente tramite il linguaggio Adobe Flash.

RakNet È la libreria che gestisce tutte le comunicazioni via rete durante le partite online.

SSL[3] È la libreria che gestisce la cifratura dei dati nelle comunicazioni remote.

fmod È la libreria che gestisce tutto il comparto sonoro del gioco.

Anche se non è presente nell'elenco, OGRE è la libreria principale e rappresenta il fulcro di tutta la piattaforma Numen. Oltre a occuparsi del rendering 3D, OGRE coordina, temporizza e gestisce le librerie della piattaforma. Tutte queste interagiscono infatti col motore grafico. Alcune, come ad esempio JPEG o Particle Universe, sono utilizzate direttamente da OGRE. Di altre, come ad esempio blPhysics o Scaleform, il motore grafico ne gestisce la temporizzazione e il ciclo di aggiornamento. Altre ancora, come ad esempio fmod o RakNet, si integrano invece col motore grafico per estenderne le funzionalità.

3.3 Utilizzo di OGRE

Per ovvi motivi di segretezza industriale, non è possibile riportare su questa tesi il codice sorgente del videogame. Di conseguenza, per vedere come OGRE è utilizzato all'interno di Victory e come è stato integrato con alcune delle librerie della piattaforma Numen, utilizzeremo esempi e pseudocodice. In questo modo non riusciremo a vedere i dettagli, che tra l'altro richiederebbero una conoscenza molto approfondita sia del motore grafico che di tutta la piattaforma Numen, ma potremo comunque capire il ruolo e l'importanza di OGRE all'interno del videogame.

3.3.1 Scaleform

La libreria Scaleform, diventata ormai uno standard de facto nello sviluppo di GUI per i videogame, si occupa di gestire tutta l'interfaccia utente di

Victory. Tramite questa libreria è possibile utilizzare il linguaggio Adobe Flash per lo sviluppo della GUI, sia per quanto riguarda la parte visiva che per quella logica. Sfruttando le caratteristiche del linguaggio, è possibile creare interfacce utente vettoriali e ricche di effetti visivi, come animazioni e transizioni.

Ogni schermata della GUI viene rasterizzata da Scaleform in una serie di primitive geometriche e di texture. Queste vengono poi passate al motore grafico che si occupa di effettuare il loro rendering. Il risultato di tutto questo processo viene poi visualizzato, sempre tramite OGRE, come overlay 2D al di sopra della scena.

Il motore grafico si fa carico anche di gestire il ciclo di aggiornamento della libreria Scaleform. Ciò è necessario affinché le animazioni e le transizioni della GUI siano temporizzate in maniera corretta. Per fare questo, viene invocato, all'interno del metodo `frameStarted` del `FrameListener` principale dell'applicazione (vedi 2.5.2 a pagina 46), il metodo `Advance` della libreria Scaleform passandogli come parametro il tempo trascorso dall'ultimo processo di rendering. La stessa cosa avviene, in maniera del tutto analoga, anche per la libreria fisica. [24]

3.3.2 Personalizzazione estetica dell'automobile

Una della caratteristiche più importanti di Victory è la personalizzazione delle automobili. Qualsiasi aspetto estetico di una vettura è infatti liberamente modificabile da parte dell'utente: dal colore al tipo di vernice, passando per adesivi e decalcomanie applicate. La figura 3.3 nella pagina seguente mostra un particolare di questo processo.

La personalizzazione estetica dell'automobile viene gestita modificando dinamicamente il materiale (vedi 2.6 a pagina 48) associato alla mesh della vettura. Ad esempio, nella figura 3.3 nella pagina seguente possiamo vedere che alla macchina è stata applicata una vernice opaca. Questo effetto si ottiene semplicemente modificando i valori di riflettività alla luce diffusa e speculare del materiale. Nel caso in oggetto, aumentando i primi e diminuendo i secondi.

Notiamo sotto al cursore del mouse due adesivi che si sovrappongono uno all'altro. Questo effetto, col quale si possono creare estetiche molto complesse, è realizzato tramite la tecnica di multitexture. Ogni adesivo corrisponde infatti a un texture e, di conseguenza, a una sezione `texture_unit` del materiale.

Figura 3.3: La personalizzazione estetica di un'automobile all'interno di Victory

3.3.3 Sistema sonoro

All'interno di un ambiente 3D, un sistema sonoro è composto da una serie di elementi che appartengono a due distinte categorie: gli emettitori e i punti d'ascolto. I primi, così come dice la parola, sono oggetti che agiscono da sorgenti sonore. Ogni suono emesso è caratterizzato da vari parametri, come ad esempio il volume, la durata, la direzione e l'intensità, che ne definiscono il comportamento e il modo di propagarsi nello spazio. Considerate tutte le sue caratteristiche, se un suono viene captato da un punto d'ascolto, allora viene riprodotto. Anche se molto semplificato, questo modello coincide con la gestione di fmod dei suoni in un ambiente 3D.

Così per come è stato descritto il sistema sonoro, viene molto intuitivo pensare agli emettitori e ai punti di ascolto come a oggetti della scena 3D di OGRE (vedi 2.2 a pagina 26). Nello specifico, un emettitore può essere rappresentato da un nodo in quanto caratterizzato sia da una posizione che da un orientamento. Quest'ultimo coincide infatti con la direzione del suono prodotto. Un punto d'ascolto, considerato il tipo di gioco che è Victory, può invece essere rappresentato da una camera (vedi 2.4.4 a pagina 43).

Nel sorgente 3.1 nella pagina successiva possiamo vedere un plausibile stralcio di codice che permette di integrare OGRE con fmod.

```
1  class SoundSceneNode :
2    public Ogre::SceneNode,
3    public FMOD::Emitter
4  {
5    ...
6  };
7
8  class SoundCamera :
9    public Ogre::Camera,
10   public FMOD::Listener
11 {
12   ...
13 };
```

Sorgente 3.1: Integrazione tra OGRE e il sistema sonoro fmod

`SoundSceneNode`, costruita derivando le classi `SceneNode` ed `Emitter`, rappresenta l'unione tra il concetto di nodo della scena 3D e quello di emettitore. In questo modo, possiamo utilizzare un unico oggetto per rappresentare entrambe le cose. Avremmo anche potuto modificare il metodo `attachObject` per far sì che fosse possibile attaccare un emettitore a un nodo della scena 3D. Il procedimento utilizzato per `SoundCamera` è analogo a quello della classe `SoundSceneNode`.

Nella figura 3.1 a pagina 59 possiamo vedere due automobili che si trovano davanti alla nostra. Supponiamo che le loro due mesh siano attaccate ad altrettanti nodi di tipo `SoundSceneNode`. Infine, supponiamo anche che la camera da cui stiamo osservando la scena sia di tipo `SoundCamera`. A questo punto, mano a mano che le vetture si avvicinano o allontano rispetto alla nostra posizione, anche la sorgente sonora si comporterà di conseguenza. In questo modo, il suono emesso da un'automobile, come ad esempio quello del motore, sarà automaticamente tanto più attenuato quanto questa sarà lontana dalla nostra posizione. [11]

3.3.4 Aggiornamento automobili remote

Come già ribadito, Victory è un gioco di corse automobilistiche online. Ciò significa che, durante una gara, ogni utente controlla dal proprio computer una singola automobile e che le informazioni relative a questa, come ad esempio posizione, direzione, accelerazione e velocità, vengono trasmesse, via rete, a tutti gli altri giocatori collegati.

Ogni automobile presente in gara corrisponde a un'entità di OGRE (vedi 2.1.4 a pagina 23) attaccata a un nodo della scena 3D. Siccome in Victory

le vetture non subiscono alcun danno in caso di incidente, la mesh dell'automobile rimarrà costante per tutta la durata della gara. L'unica cosa che cambierà sarà quindi la posizione e l'orientamento del nodo.

Durante una gara, a ogni step di aggiornamento del gioco, le operazioni eseguite possono essere riassunte in:

1. Si esegue un ciclo di aggiornamento della libreria fisica che aggiorna le informazioni relative alla propria vettura.

2. In base agli output prodotti al passo precedente, si aggiorna, di conseguenza, la posizione e l'orientamento del nodo a cui è attaccata l'automobile.

3. Il nodo viene serializzato e spedito, tramite la libreria RakNet, a tutti gli altri giocatori collegati.

Nel contempo arriveranno i nodi spediti dagli altri utenti con le informazioni aggiornate di posizione e orientamento. Questi andranno a rimpiazzare, nella scena 3D, i loro corrispettivi contenti i valori non aggiornati. [21]

Capitolo 4

Conclusioni

In questa tesi abbiamo introdotto alcuni degli aspetti principali del motore grafico OGRE. Con poche righe di codice abbiamo creato, di volta in volta, piccole applicazioni con un discreto livello di complessità. Pensiamo, ad esempio, al caricamento di una mesh o alla generazione delle ombre di una scena 3D. In OGRE, entrambe queste operazioni possono essere portate a termine con meno di dieci righe di codice. La stessa cosa non si può certo dire per OpenGL dove ognuno di questi compiti avrebbe invece richiesto qualche migliaio di righe di codice. Tutto questo ci fa capire due cose: da un lato la potenza di OGRE, dall'altro il suo maggior livello di astrazione rispetto a una libreria grafica a basso livello come OpenGL o Direct3D9.

A seconda di come ci si rapporta col motore grafico questi due aspetti possono costituire un vantaggio o uno svantaggio. Se ci si avvicina a OGRE con conoscenze pregresse sulla programmazione grafica, allora gli aspetti citati precedentemente costituiscono sicuramente un vantaggio. Questo perché OGRE aiuta e semplifica lo sviluppo di applicazioni 3D facendosi carico di numerosi compiti. Avvicinarsi invece a OGRE senza nessuna conoscenza di librerie a più basso livello, come ad esempio OpenGL o Direct3D9, può non essere la scelta più adatta. Questo proprio in funzione del maggior livello di astrazione del motore grafico che tende a nascondere all'utente alcuni degli aspetti sottostanti. Sia la struttura di OGRE che la sua documentazione danno infatti già per acquisti molti concetti sulla pipeline di rendering e possono quindi creare confusione in un utente alle prime armi. Di conseguenza, anche la comprensione di questa tesi è proporzionale alla conoscenza che il lettore ha di OpenGL e/o Direct3D9.

Riassumendo, librerie come OpenGL o Direct3D9 rappresentano tuttora il modo migliore con cui avvicinarsi al mondo della programmazione 3D. I motori grafici costituiscono invece uno step evolutivo delle due precedenti librerie. In particolare, OGRE si rivela essere uno strumento semplice e nel

contempo molto potente. A conferma di questo, il suo impiego anche nella applicazioni 3D per antonomasia: i videogame. Il fatto che OGRE si limiti però a gestire solamente gli aspetti legati al rendering 3D lo rende incompleto, ma, grazie alla sua flessibilità, facilmente integrabile con altre librerie.

Acronimi

USMC United States Marine Corps
Conosciuto in Italia come Corpo dei Marine, o più semplicemente Marine, è una delle forze armate degli Stati Uniti.

GotY Game of the Year
È un premio assegnato, da riviste e siti web specializzati, ai migliori videogame dell'anno.

API Application Programming Interface
Nel contesto di una libreria software, identifica l'insieme di metodi e funzionalità a disposizione dei programmatori.

IBM International Business Machines Corporation
Nota anche come Big Blue, è una delle maggiori aziende del settore informatico.

AGI Adventure Game Interpreter
È un game engine utilizzato da Sierra On-Line, durante buona parte degli anni ottanta, per creare ed eseguire le avventure grafiche 2D da lei prodotte.

SCUMM Script Creation Utility for Maniac Mansion
È un middleware per lo sviluppo di avventure grafiche 2D creato all'interno di LucasArts che lo ha utilizzato, fino al 1998, per la produzione di tutti i videogame di questa tipologia.

BSP Binary space partition
È un metodo che permette di suddividere, in maniera ricorsiva, uno spazio euclideo in insiemi convessi di iperpiani. Questa suddivisione si rappresenta poi tramite una struttura dati ad albero, chiamata BSP tree.

CAD Computer-Aided Design
Indica il settore dell'informatica volto all'utilizzo di tecnologie soft-

ware, in particolare della computer grafica, per supportare l'attività di progettazione di manufatti sia reali che virtuali.

GPU Graphics Processing Unit
È una tipologia di coprocessore specializzata nel rendering di immagini grafiche.

ECTS European Computer Trade Show
È una fiera, che si tiene ogni anno, in cui vengono presentati nuovi prodotti dell'industria elettronica, dei computer e dei videogame.

ARB Architecture Review Board
È stato il consorzio industriale che, dal 1992 al 2006, ha gestito la specifica OpenGL.

HDRR High dynamic range rendering
È una tecnica di rendering in cui si utilizza uno spazio di valori più ampio, rispetto a quanto fatto di solito, per i calcoli d'illuminazione.

LOD Level of detail
È una tecnica, utilizzata al fine ottimizzare il processo di rendering, con cui si diminuisce la complessita di oggetti 3D mano a mano che questi si allontanano dal punto di osservazione.

NURBS Non Uniform Rational B-Splines
È una classe di curve geometriche utilizzate in computer grafica per rappresentare curve e superfici.

GLSL OpenGL Shading Language
È un linguaggio di programmazione, ad alto livello, per la gestione delle unità shader di una GPU. È stato introdotto da ARB come estensione di OpenGL a partire dalla versione 2.0 della libreria.

HLSL High Level Shading Language
È un linguaggio, sviluppato da Microsoft, per la creazione di shader per Direct3D.

Cg C for Graphics
Definito e sviluppato da NVIDIA, è un metalinguaggio, con una sintassi simile a quella del C, che permette di creare shader compatibili sia con OpenGL che con Direct3D.

SDK Software development kit
È un insieme di strumenti per lo sviluppo di software.

DLL Dynamic-link library
Nell'ambiente Microsoft Windows, è una libreria software che viene caricata dinamicamente in fase di esecuzione del programma. Equivale ad una libreria condivisa dei sistemi Unix.

IDE Integrated Development Environment
È un software per semplicare ai programmatori lo sviluppo di applicazione. Solitamente è composto da un editor, da un compilatore e da un debugger.

GUI Graphical User Interface
È un tipo di interfaccia che consente all'utente di interagire con l'applicazione manipolando oggetti grafici.

JPEG Joint Photographic Experts Group
È uno standard internazionale di compressione dell'immagine digitale a tono continuo, sia a livelli di grigio che a colori.

OIS Object-Oriented Input System
È una libreria per la gestione di input da varie periferiche, come ad esempio tastiere, mouse e joystick.

SSL Secure Sockets Layer
È un protocollo crittografico che permette comunicazione sicura e integrità dei dati su reti internet.

Bibliografia

[1] *3DVIA Virtools - Dassault Systèmes*. 2012. URL: http://www.3ds.com/products/3dvia/3dvia-virtools/.

[2] *A Fallen Titans Final Glory: Part I*. 2012. URL: http://www.sudhian.com/content/?p=1249.

[3] *AGI Development Site - Intro*. 2012. URL: http://www.agidev.com/intro/.

[4] *Brief history of OGRE*. 2012. URL: http://www.ogre3d.org/tikiwiki/Brief+history+of+OGRE.

[5] Luigi Calori, Carlo Camporesi e Sofia Pescarin. *Virtual Rome: a FOSS approach to WEB3D*.

[6] *Doom Engine source code review*. 2012. URL: http://fabiensanglard.net/doomIphone/doomClassicRenderer.php.

[7] *Doom rendering engine - DoomWiki.org, the new home of the Doom Wiki - Doom, Heretic, Hexen, Strife, and more*. 2012. URL: http://doomwiki.org/wiki/Doom_rendering_engine.

[8] Michael Drummond. *Renegades of the Empire: How Three Software Warriors Started a Revolution Behind the Walls of Fortress Microsoft*. Three Rivers Press, 2000. ISBN: 0609807455.

[9] *Exploring the Freescape - Retro Feature at IGN*. 2012. URL: http://uk.retro.ign.com/articles/922/922505p1.html.

[10] *Features — OGRE - Open Source 3D Graphics Engine*. 2012. URL: http://www.ogre3d.org/about/features.

[11] *fmod - interactive audio middleware*. 2012. URL: http://www.fmod.org/.

[12] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[13] *Home — Crytek*. 2012. URL: http://crytek.com/.

[14] *Irrlicht Engine - A free open source 3D engine*. 2012. URL: http://irrlicht.sourceforge.net/.

[15] Felix Kerger. *OGRE 3D 1.7 Beginner's Guide*. Packt Publishing, 2010. ISBN: 1849512485.

[16] *Making adventure games with AGI - Feature - Adventure Classic Gaming - ACG - Adventure Games, Interactive Fiction Games - Reviews, Interviews, Features, Previews, Cheats, Galleries, Forums*. 2012. URL: http://www.adventureclassicgaming.com/index.php/site/features/143/.

[17] *OGRE: API Reference Start Page - OGRE Documentation*. 2012. URL: http://www.ogre3d.org/docs/api/html/index.html.

[18] *OGRE Manual v1.7 ('Cthugha'): OGRE Manual v1.7 ('Cthugha')*. 2012. URL: http://www.ogre3d.org/docs/manual/.

[19] *Ogre Wiki - Support and community documentation for Ogre3D*. 2012. URL: http://www.ogre3d.org/tikiwiki/.

[20] *OpenGL - The Industry Standard for High Performance Graphics*. 2012. URL: http://www.opengl.org/.

[21] *RakNet - Multiplayer game network engine*. 2012. URL: http://www.jenkinssoftware.com/.

[22] *RenderWare - Wikipedia, the free encyclopedia*. 2012. URL: http://en.wikipedia.org/wiki/RenderWare.

[23] *Rocksteady Studios*. 2012. URL: http://www.rocksteadyltd.com/.

[24] *Scaleform — Autodesk Gameware*. 2012. URL: http://gameware.autodesk.com/scaleform.

[25] *SCUMM History*. 2012. URL: http://members.fortunecity.com/harang/scumm.html.

[26] *UNITY: Game Development Tool*. 2012. URL: http://unity3d.com/.

[27] *Unreal Technology*. 2012. URL: http://www.unrealengine.com/.

[28] *Vae Victis / Independent Game Development Studio*. 2012. URL: http://vaevictis.it/.

[29] *VBS2 — Bohemia Interactive Simulations*. 2012. URL: http://armory.bisimulations.com/products/vbs2/overview.

[30] *Victory: The Age of Racing — Free to Play Online Racing Game*. 2012. URL: http://victorythegame.com/.

[31] *Wolfenstein 3D engine - Wikipedia, the free encyclopedia*. 2012. URL: http://en.wikipedia.org/wiki/Wolfenstein_3D_engine.