Ringraziamenti

Si ringrazia la Fondazione Ultramundum, in particolare nelle persone dell'ing. Fulvio Dominici Carnino e dell'ing. Lorenzo Bidone, per l'indispensabile supporto che ci hanno fornito nella realizzazione di questo lavoro e per la disponibilità e la pazienza dimostrate.

Indice

Obiettivi

Questo lavoro ha lo scopo di presentare lo studio e la realizzazione di tecniche per il calcolo di ombre dinamiche in tempo reale in un'applicazione grafica tridimensionale. Tale progetto è stato realizzato utilizzando la tecnologia degli *Shader*, programmi che si basano su propri linguaggi di programmazione e che sono eseguiti direttamente dai microprocessori a bordo delle schede video.

Questo lavoro è stato realizzato presso la Fondazione *Ultramundum*, con lo scopo di migliorare la resa visiva dell'ambiente 3D che sta alla base degli applicativi che questa utilizza. La storia della Fondazione e le attività che questa porta avanti sono brevemente illustrate nella prima parte di questo documento.

Nella stestura di questo lavoro abbiamo considerato che il lettore fosse in possesso delle nozioni di base dell'informatica e della grafica 3D. Di seguito elenchiamo la suddivisione degli obiettivi del progetto in quattro categorie distinte.

Passaggio dal paradigma con pipeline di rendering fissa ad una gestione tramite shader in un'applicazione complessa

Il primo obiettivo della nostra tesi è eseguire l'adattamento di un sistema complesso (UltraPort, il sistema operativo su cui si fonda la tecnologia UltraPeg) che si poggiava su di un motore grafico con pipeline di rendering fissa (Wild Magic 3D Engine versione 3) ad uno basato invece sugli shader (Wild Magic 3D Engine versione 4). Quest'opera è necessaria ad Ultramundum per rendere migliore il suo prodotto (gli shader offrono possibilità di creare effetti molto più numerosi e qualitativamente migliori rispetto alla pipeline di rendering fissa), mentre dal canto nostro è fondamentale per poter implementare le ombre degli oggetti e gli effetti climatici.

Ricerca nel campo degli shader

Un'importante parte della nostra tesi riguarda la ricerca nel campo degli shader, un settore della grafica 3D in rapida evoluzione la cui documentazione liberamente accessibile è attualmente molto frammentata.

L'obiettivo che ci siamo prefissi è cercare di fare un po' di chiarezza e radunare la documentazione disponibile, in modo da avere un punto di partenza per effettuare la

conversione al paradigma a shader. Inoltre sarebbe opportuno studiarne le ancora numerose problematiche di applicazione e condivisione, cercando di individuare quali elementi possono essere considerati standard nel processo di creazione degli shader e quali no.

Questa analisi potrebbe inoltre essere utile ad altre persone che in futuro affrontino le problematiche dell'approccio a shader.

Ricerca di algoritmi per il calcolo di ombre ed effetti climatici

Il passaggio all'approccio a shader è una parte fondamentale per poter integrare nel modello luminoso di Ultraport nuove tecniche di illuminazione ed effetti climatici. Un'altra parte della nostra tesi è quindi improntata a ricercare, approfondire e, se possibile, migliorare gli algoritmi che eseguono l'ombreggiatura di una scena, cercando di scegliere il più adatto per il nostro contesto di utilizzo. Potrebbe essere necessario progettarne di appositi usando come punto di partenza quelli già esistenti.

Creazione di shader per illuminazione ed effetti climatici

Una volta individuato uno o più algoritmi per produrre le ombre, sarà necessario scrivere gli shader ed il codice necessario per l'integrazione di questi nel motore grafico Wild Magic. Lo scopo è la creazione di shader che migliorino il modello luminoso esistente tramite l'introduzione di ombreggiatura per oggetti statici e dinamici in modo da renderlo più realistico possibile. Si vorrebbero inoltre creare shader che effettuino effetti climatici quali nebbia e nuvole in una scena. L'introduzione di questi effetti migliorerebbe sensibilmente il realismo e la rappresentazione grafica dei modelli tridimensionali, in particolare delle città, creati con la tecnologia UltraPeg.

Capitolo 1

La Fondazione Ultramundum

La Fondazione Ultramundum è nata il primo gennaio 2001, nella forma di comitato, dopo cinque anni di sviluppo della Tecnologia UltraPeg, brevettata in Europa e negli Stati Uniti dall'attuale presidente Fulvio Dominici Carnino.

La Fondazione è un'organizzazione no-profit, nata per la realizzazione del progetto Ultramundum, dettagliato nel libro omonimo. Alla base di Ultramundum si trova la tecnologia UltraPeg, di cui parleremo in seguito [cfr. sezione 1.2]. Dopo due anni di ricerca fondi, la Fondazione si è data personalità giuridica nel 2003. Da allora ha realizzato e ha allo studio importanti progetti con enti pubblici come il Comune di Torino, la Provincia di Torino, la Regione Piemonte, la Regione Valle d'Aosta, il Ministero dell'Istruzione e della Ricerca, il Politecnico di Torino ed altri.

La struttura della Fondazione si basa su di un nucleo di esperti di alto livello, con esperienza pluridecennale nel settore, che sviluppa il cuore della tecnologia UltraPeg con la collaborazione di neolaureati, laureandi e semplici collaboratori molto motivati. La scelta del modello *no-profit* è motivata dalla volontà di sviluppare una tecnologia che si basi sulla diffusione degli strumenti per la realizzazione di una sorta di "scatola di costruzioni virtuale". I "mattoncini" presenti in questa scatola, sviluppabili liberamente da chiunque, possono essere usati in vari contesti per la facile e veloce realizzazione di contenuti tridimensionali interattivi. Se la realizzazione e la diffusione di un tale modello di sviluppo fossero stati affidati ad una struttura di tipo aziendale, molti dei collaboratori avrebbero potuto pensare che la loro opera sarebbe stata utilizzata da terzi per fini di lucro. Un ente come una Fondazione, invece, garantisce che non vi sia distribuzione di utili. Come già dimostrato in altri casi in passato, un modello di sviluppo di questo tipo è l'unico percorribile nel mondo di Internet per la diffusione di un nuovo standard tecnologico. Inoltre ha partecipato alle più importanti conferenze del settore di informatica grafica e realtà virtuale, come *Virtuality*, organizzata dal Virtual Reality e MultiMedia Park, i congressi di *MIMOS* (Movimento Italiano Modellazione e Simulazione), *Imagina* a Montecarlo, *SMAU* a Milano e altri. La Fondazione è stata invitata a presentare le proprie tecnologie ed i propri progetti, acquisendo notorietà a livello internazionale.

La Fondazione ha sede a Grugliasco all'interno del Parco Culturale delle Serre, in cui il Comune di Grugliasco ha messo a disposizione nuovi appositi spazi. Per maggiori

Figura 1.1. Alludrum, il logo di Ultramundum

informazioni, rimandiamo a [1].

1.1 Obiettivi della Fondazione

La Fondazione Ultramundum nasce per sviluppare e diffondere una nuova tecnologia software, denominata *UltraPeg*, che consente, tra le altre cose, di realizzare un nuovo tipo di televisione, chiamata *Ultravisione*. La principale differenza con la televisione che tutti conosciamo risiede nel fatto che i contenuti ultravisivi sono tridimensionali, mentre quelli della televisione classica sono bidimensionali. Un'ulteriore novità risiede nel mezzo trasmissivo: utilizzare la rete Internet invece delle onde radio.

La Fondazione ha come obiettivo la diffusione di questa nuova tecnologia e l'agevolazione di tutti coloro, persone, enti o aziende, intendano produrre contenuti ultravisivi. Chiunque potrà usufruire dell'Ultravisione tramite Internet: l'idea di base è simile a quella della televisione classica, cioè realizzare vari canali numerati tematici. Ogni canale potrà veicolare film, documentari, ricostruzioni architettoniche e archeologiche, videogiochi, didattica o altro. La maggior parte del materiale, inoltre, è distribuito in open source, mantenendo tuttavia, ove necessario, strettamente rispettati tutti i diritti di proprietà intellettuale. Il canale principale sarà, nelle intenzioni della Fondazione, 4DGEA, con scopi prettamente scientifici, che realizzerà in tre dimensioni il mondo fisico nel quale viviamo, a partire dalla galassia fino al dettaglio più ricercato. L'idea è di affiancare a 4DGEA, un altro canale, 4DGEA+, che integra 4DGEA rendendolo più ricco e spettacolare, a scapito della precisione e dell attendibilità scientifica di ciò che vi è rappresentato. Tutto ciò che non ha riscontri scientifici nella realtà sarà creato proceduralmente, rendendo l'ambiente il più verosimile possibile. Ad esempio, sarà l'opportunità di fare una passeggiata virtuale su Marte, ottenendo una rappresentazione tridimensionale molto affascinante, seppure poco scientifica.

Molti progetti sono già stati portati avanti con il sostegno di svariati enti pubblici, a livello comunale, provinciale, regionale e ministeriale. Queste realizzazioni consentono di mettere alla prova la tecnologia e costituiscono una importante base di esempi e spunti da cui partire per proprie realizzazioni.

Tra gli obiettivi futuri della Fondazione ci sono la crescita e la diffusione ulteriore della tecnologia Ultrapeg, coinvolgere enti e aziende nel progetto, completare il modello 3D

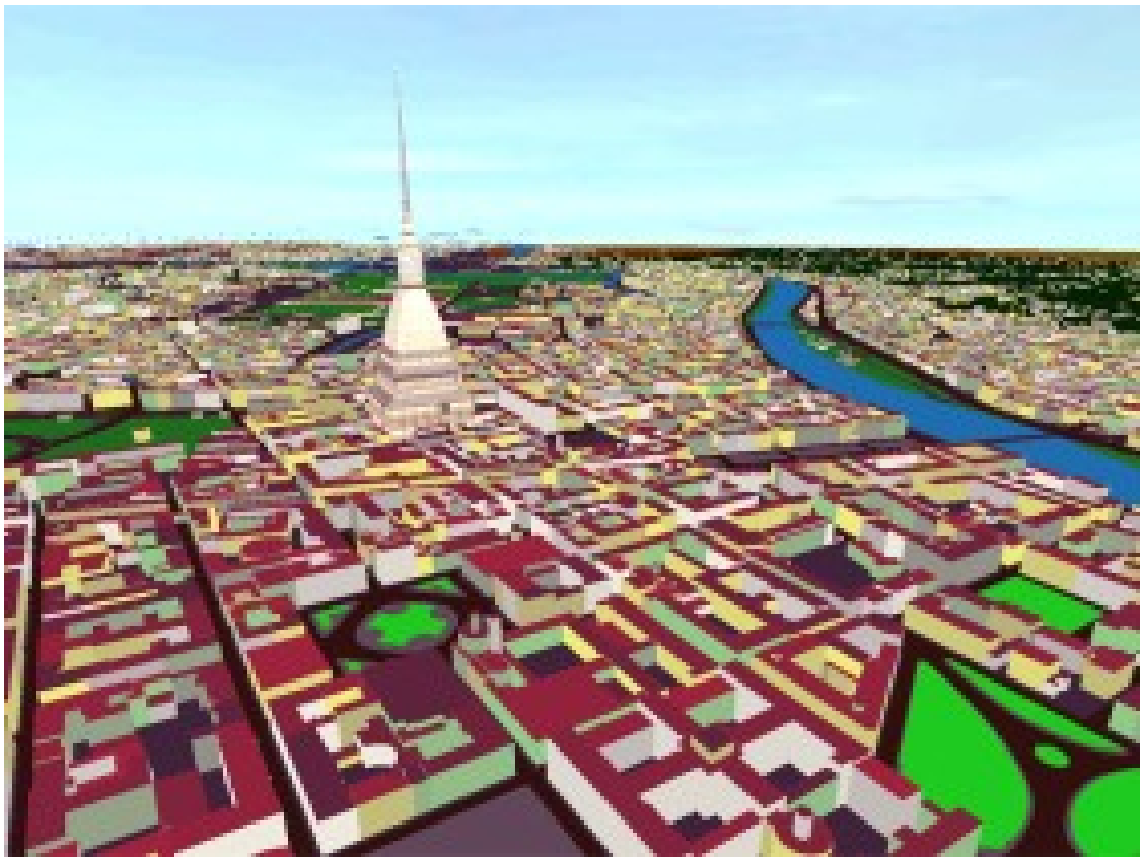


Figura 1.2. Veduta di Torino in 3D

dell'Italia e dell'Europa e delle più importanti città d'arte del mondo, entrare nel mercato dei videogiochi, sviluppo di sistemi immersivi per la realtà virtuale che consentano la fruizione in modo estremamente realistico del materiale sviluppato, con la realizzazione di nuovi servizi e nuovi media.

1.2 Ultrapeg

Ultrapeg è una tecnica di rappresentazione e memorizzazione dei dati di ambienti tridimensionali. L'utilizzo di Ultrapeg si rende necessario quando si desidera realizzare e rendere disponibili tramite Internet, applicazioni multimediali nei quali l'osservatore possa essere immerso.

Esempi dei campi di applicazione di Ultrapeg sono i videogiochi, i documentari interattivi tridimensionali, i film e telefilm di sintesi, la televisione tridimensionale interattiva (ultravisione) e in generale tutti quei casi nei quali si desideri porre l'utente all'interno di un ambiente tridimensionale liberamente esplorabile.

Figura 1.3. Scorcio di Torino in 3D

Tra le novità e vantaggi che presenta Ultrapeg c'è la possibilità di scaricare da Internet e visualizzare in pochi secondi qualsiasi ambiente digitale (poi liberamente esplorabile), di creare ambienti tridimensionali esplorabili liberamente nello spazio e nel tempo, di ascoltare dei contenuti testuali direttamente da una voce narrante nella propria lingua, di riutilizzare automaticamente ed in maniera "intelligente" interi ambienti (o porzioni di essi) all'interno di nuove produzioni, di migliorare in modo automatico ogni ambiente nel tempo grazie allo sviluppo tecnologico (senza richiedere alcun lavoro aggiuntivo all'autore), di creare canali digitali interattivi gratuiti o a pagamento grazie al supporto al business, di creare di titoli anche da parte di piccoli gruppi di lavoro con poche risorse economiche. Inoltre ogni prodotto è automaticamente multiutente, potendo collegare migliaia di persone contemporaneamente.

A livello di approccio, la sostanziale differenza rispetto alle tecniche normalmente usate per la realizzazione di ambienti tridimensionali interattivi risiede nella memorizzazione dei dati come concetti, invece che come collezioni di informazioni digitali. Fino ad oggi infatti la realizzazione di ambienti tridimensionali ha richiesto la fase di modellazione, nella quale artisti esperti di computer grafica letteralmente "scolpivano" ogni oggetto che si rendeva necessario, tecnica assimilabile a lavorare un blocco di creta grezza e trarne i modelli di

volta in volta necessari.

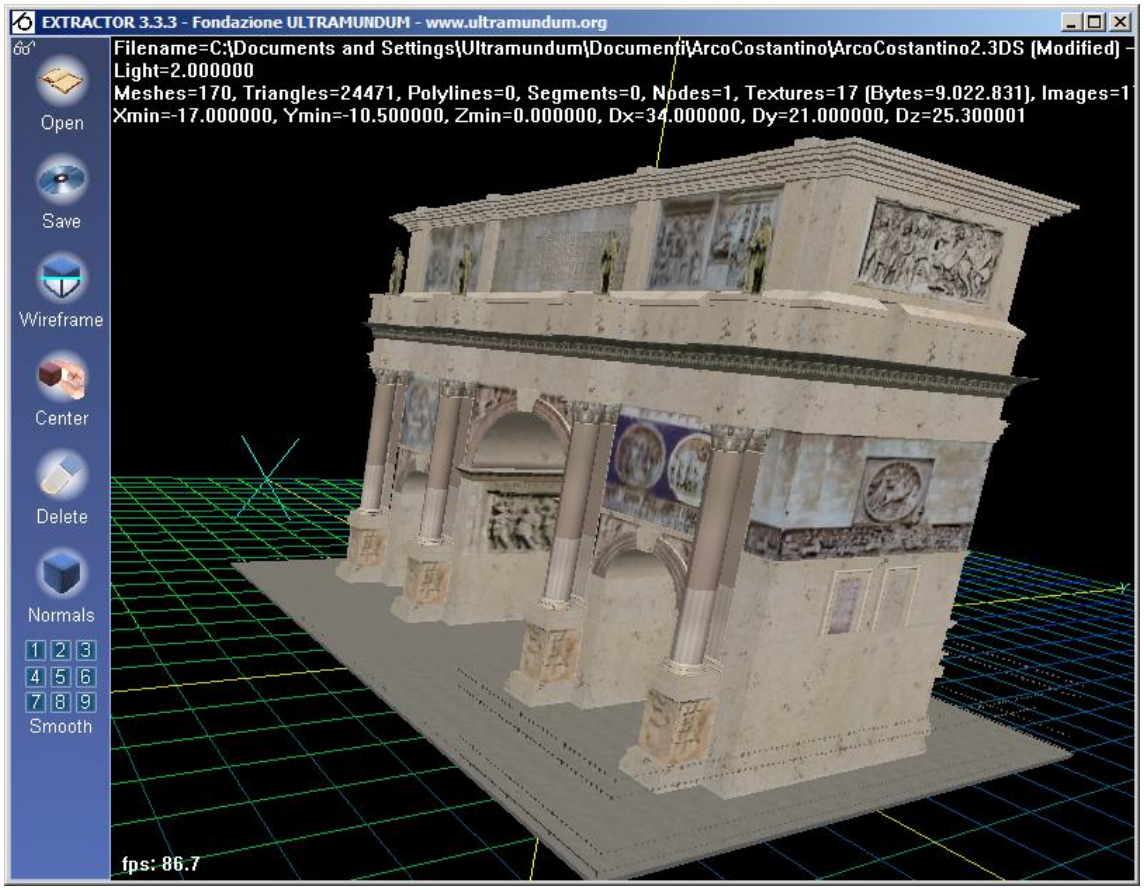


Figura 1.4. Modellazione dell'arco di Costantino per la città di Roma

Ultrapeg utilizza invece un approccio completamente diverso. L'ambiente virtuale viene visto come una gigantesca scatola di costruzioni che offre una vasta gamma di pezzi diversi utili alla realizzazione della scena. In tal modo si evita all'utente di modellare o creare ogni volta un oggetto dell'ambiente, ma gli si offre la possibilità di sceglierlo fra i numerosi già esistenti ed eventualmente di modificarlo facilmente secondo le sue esigenze, generandolo impostando parametri diversi. Ogni elemento eventualmente non presente dovrà essere sì realizzato, ma sarà poi disponibile nella raccolta di base per chiunque altro ne necessiti in futuro; Ultrapeg si basa infatti su di una filosofia collaborativa.

In tal modo si memorizza (e poi trasmette su Internet) solo l'elenco dei numeri di serie degli elementi della scena, non tutti i loro dati, facendo sì che la dimensione dei dati di un ambiente tridimensionale si riduca enormemente e rendendone possibile la trasmissione in tempo reale su Internet. La fase di sviluppo di nuovi contenuti è estremamente semplice: basta scegliere dalla raccolta di oggetti i singoli elementi che interessano e disporli sulla

scena. Si crea così un elenco che a tutti gli effetti è un ambiente tridimensionale completo, scaricabile da Internet da chiunque lo desideri.

Tra i mattoncini di base per costruire una scena non ci sono solo i modelli tridimensionali, ma anche molti altri elementi necessari per la produzione di un applicazione multimediale come musiche di sottofondo, animazioni, descrizioni testuali degli elementi e così via. Tutti i testi che descrivono parti degli ambienti sono memorizzati in uno speciale formato che può essere tradotto automaticamente in ogni lingua del mondo per poi essere letto, sempre automaticamente, nel momento in cui l'utente lo desidera. I mattoncini elementari messi a disposizione da Ultrapeg possono essere singoli elementi o aggregati molto complessi. Si possono prendere e usare interi edifici (come la Mole Antonelliana o Palazzo Madama), loro parti (come porticati, arcate, portoni) o singoli elementi (come fregi, capitelli, semplici mattoni).

Un'altra caratteristica importante di ogni elemento di Ultrapeg è l'"intelligenza", ovvero la capacità di adattarsi all'utilizzo che se ne vuol fare. Non essendo una collezione fissa di dati, bensì un programma vero e proprio, un porticato può, ad esempio, allungarsi o ridursi in modo da adattarsi al particolare uso che se ne vuol fare.

Inoltre un ambiente, essendo una serie di riferimenti agli elementi della raccolta di base, può migliorare nel tempo in modo automatico. Se un autore, infatti, decide di usare la Mole Antonelliana in un certo punto, memorizza il numero di serie dell'elemento e non i suoi dati effettivi. Se in un secondo momento il mattoncino della Mole viene migliorato, automaticamente si migliorano anche tutti i canali che ad esso fanno riferimento, senza che gli autori debbano fare nulla.

Grazie a queste ed altre caratteristiche, Ultrapeg permetterà la creazione della vera televisione tridimensionale (Ultravisione), nella quale gli utenti potranno seguire programmi simili a quelli attuali, ma nei quali sarà possibile "entrare" nei contenuti [cfr. sezione 1.3].

L'Ultravisione permetterà inoltre il video-on-demand, cioè la possibilità di accedere ad un qualsiasi contenuto in qualunque momento, senza attendere che venga trasmesso. Ogni prodotto, infatti, sarà scaricabile e visualizzabile a richiesta, proprio come le pagine del World Wide Web.

1.3 Ultravisione

L'Ultravisione è un servizio basato su tecnologia software brevettata, ma di libero utilizzo. E' possibile utilizzare questa nuova tecnologia per attivare un proprio canale ultravisivo sui server della Fondazione. Rispetto alla televisione classica, la grande novità è costituita dalla possibilità per l'utente di esplorare liberamente gli ambienti visualizzati. Ad esempio, se vediamo un documentario in Ultravisione di una città, viene proposta all'utente una visita seguendo un percorso classico. Tuttavia, è consentito in ogni istante avere il controllo della telecamera in modo da svoltare in una strada diversa da quella del percorso originale, oppure chiedere informazioni su luoghi, edifici e monumenti con risposta vocale generata al volo, oppure ancora interagire con gli oggetti.

Figura 1.5. Piazza San Carlo in 3D

Un filmato ultravisivo può essere quindi visto in maniera passiva, ma ogni scena può essere esplorata da prospettive personali, diverse da quelle del regista. La storia può essere interrotta per esplorare gli ambienti in luoghi anche molto lontani dall'azione.

Un videogioco ultravisivo invece ha il vantaggio di non avere necessità di download o di supporti hardware come cd-rom. I contenuti di tali applicazioni conservano le peculiarità dell'Ultravisione: gli spazi sono di estensione illimitata e ogni elemento ha un livello di dettaglio praticamente infinito.

Lo sviluppo dei contenuti ultravisivi si basa sul concetto di utilizzo di una serie di mattoncini "intelligenti" in grado di creare, come in una sorta di gioco di costruzioni, qualsiasi ambiente tridimensionale in modo semplice e veloce. Questo approccio ha il grande vantaggio di ridurre i tempi di produzione e i costi di qualsiasi realizzazione.

1.4 Progetti

La Fondazione ha realizzato numerose applicazioni e alcune altre sono in corso di sviluppo, alcune commissionate da enti pubblici o privati, alcune per migliorare il prodotto già esistente. Tra i progetti portati a compimento ci sono il plastico interattivo digitale di

piazza San Carlo a Torino, della città di Torino e comuni limitrofi, dell'aspetto futuro del parco Zappata a Torino, dell'intera provincia di Torino, della città di Aosta ricostruita nel primo secolo d.C. e della città di Roma.

Figura 1.6. Aosta antica in 3D

Tra i progetti in corso, ci sono la realizzazione della Luna e di Marte in 3D a partire dai dati Nasa, il miglioramento dei dettagli del modello di Torino, il modello dell'Italia in 3D, la ricostruzione dell'intero pianeta in quattro dimensioni, ossia dando all'utente la possibilità di viaggiare nel tempo, il più lungo e ambizioso progetto. Inoltre è in corso la realizzazione del primo film tridimensionale in open source al mondo, un cartone animato per bambini e non.

Figura 1.7. Veduta della basilica di San Pietro a Roma

Capitolo 2

Il motore grafico alla base di Ultraport

2.1 La necessità di un motore grafico

Per supportare la tecnologia UltraPeg più a basso livello, era necessario ricorrere all'utilizzo di un motore grafico del genere di quelli usati solitamente per i videogiochi e le applicazioni interattive. Le numerose funzioni che svolge UltraPort richiedevano infatti che questo sistema operativo si basasse su un motore grafico dalle buone prestazioni.

La scrittura di un motore grafico è un'operazione molto complessa, che richiede un impiego ingente di uomini e risorse. Con i mezzi ancora ridotti della Fondazione, non sarebbe stato possibile dedicare tempo ed energie alla creazione di un proprio motore grafico su cui basare UltraPort; si è perciò deciso di procedere con la ricerca di un motore grafico già esistente che fosse il più possibile compatibile con le esigenze di Ultramundum.

Numerose case sviluppatrici di software e videogiochi hanno prodotto motori grafici in grado di effettuare operazioni molto complicate con ottime prestazioni; tuttavia pochissime di queste hanno scelto di divulgare i codici sorgenti del loro prodotto, rendendone impraticabile l'uso, oppure sono disposte a cederne i diritti di utilizzo a cifre molto alte. La scelta di Ultramundum è quindi caduta sul motore *Wild Magic 3D Engine*, il prodotto che meglio univa le caratteristiche richieste dalla Fondazione.

2.2 Il motore grafico Wild Magic

Wild Magic è un motore grafico sviluppato da Geometric Tools, compagnia specializzata nello sviluppo di software per la computer grafica, l'analisi di immagini e metodi numerici; è stato ideato in particolare dal suo presidente David H. Eberly, ex professore di matematica in Texas e fondatore dell'azienda.

La caratteristica principale del motore è di essere *open source*, ovvero ne viene fornito gratuitamente il codice sorgente[1] ed è possibile per l'utente modificarlo adattandolo alle

[1] scaricabile da chiunque al sito www.geometrictools.com

proprie esigenze, anche se per utilizzarlo per scopi commerciali bisogna acquistare una ulteriore licenza.

Inoltre questo motore grafico è stato scelto da Ultramundum per la portabilità e la buona documentazione fornita in dote al momento dell'acquisto. Wild Magic è infatti pensato per poter funzionare su macchine dotate di sistema operativo Windows, Linux o Mac ed è in grado di supportare le *API* (Application Programming Interface) grafiche *OpenGL* (open source) e *DirectX* (specifiche di Microsoft per Windows). Visto che anche la tecnologia di Ultramundum non è pensata per nessuna piattaforma in particolare, la portabilità è una caratteristica fondamentale del motore grafico da scegliere, oltre alla relativa facilità d'uso, l'economicità (esistono motori grafici da milioni di dollari...) e la buona libertà di utilizzo.

Di contro, va considarato il fatto che Wild Magic è pensato più per scopi didattici che applicativi, quindi non sempre le soluzioni adottate sono efficienti al massimo. Wild Magic si è evoluto nel corso degli anni, aggiornandosi costantemente e sono state rilasciate diverse nuove versioni. Al momento del nostro arrivo in Fondazione per lo svolgimento della tesi, Ultramundum utilizzava la versione 3.11 di Wild Magic, basata sulla pipeline di rendering fissa, mentre Geometric Tools aveva già rilasciato la versione 4 (in particolare la 4.4), basata sull'utilizzo degli shader. Per realizzare gli obiettivi della nostra tesi, era necessario l'uso della versione del motore basata su shader, versione che durante i mesi di lavoro ha raggiunto l'evoluzione 4.6, anche grazie ad alcune migliorie suggerite da noi a GeometricTools.

E' stato quindi necessario effettuare tutta l'opera di conversione da una versione del motore all'altra dell'interfaccia tra Ultraport e WildMagic visto che gli shader erano assolutamente necessari per supportare gli effetti che volevamo integrare in UltraPort.

2.3 La struttura di Wild Magic

Wild Magic è essenzialmente basato su sei librerie:

- Foundation: contiene i file più a basso livello che gestiscono le API, le funzioni matematiche, le approssimazioni e le figure geometriche elementari;
- Graphics: sono implementati gli effetti delle scene, gli shaders e i cicli di rendering;
- Renderers: contiene le chiamate alle API vere e proprie. E'divisa a seconda che si usi DirectX, OpenGL o che si elabori tutto sulla CPU via software;
- Imagics: riguarda le immagini, i filtri e le operazioni binarie;
- Physics: per le intersezioni e i sistemi particellari;
- Applications: è anch'essa divisa per DirectX, OpenGL e software generico. Contiene le chiamate alle API per gestire le finestre e le applicazioni in genere.

Affinchè fossero supportate alcune funzioni, ad esempio modalità di applicazioni per le texture o applicazioni multithread, è stato necessario apportare alcune modifiche al codice

sorgente del motore che quindi differisce in parte da quello originale. Si è cercato tuttavia di ridurre al minimo questi interventi per non stravolgere l'architettura originale. La maggior parte delle modifiche apportate riguarda la libreria Graphics e quella che gestisce i cicli di rendering. I cambiamenti effettuati saranno illustrati nel dettaglio in seguito [cfr. sezione 2.6].

E' possibile compilare le librerie del motore in due modalità, Debug e Release, in base all'uso che se ne vuol fare. Inoltre è possibile compilare il motore come DLL e in modalità Memory.

Wild Magic contiene numerose applicazioni d'esempio grazie alle quali abbiamo preso confidenza col funzionamento del motore, non sempre chiarissimo a causa dell'approccio gerarchico e della struttura ad oggetti molto elaborata.

2.4 La scena in Wild Magic

La scena in Wild Magic viene salvata all'interno di un grafo. Scorrendo questo grafo, è possibile conoscere non solo gli oggetti che fanno parte del mondo rappresentato, ma anche le loro caratteristiche fisiche e grafiche. Questa gestione della scena consente di renderizzare solo gli oggetti visibili o parzialmente visibili all'osservatore, cosa che aumenta in modo drastico le prestazioni.

Questa struttura è molto flessibile, e consente a chi crea la scena di organizzare gli oggetti secondo i criteri che più preferisce. Solitamente per velocizzare la fase di resa grafica è bene seguire due principi di coerenza: la coerenza spaziale, organizzando la gerarchia in modo da mantenere nodi spazialmente vicini nella stessa porzione di grafo, e la coerenza dello stato di rendering, ossia raggruppare gli oggetti secondo la modalità con cui verranno rappresentati sullo schermo (con luce/senza luce, con texture/senza texture, in wireframe, eccetera...).

La principale funzione del grafo della scena è preservare le relazioni spaziali tra gli oggetti. E' pratica molto diffusa nella modellazione 3D di combinare parti di scena, create in maniera indipendente, utilizzando sistemi di riferimento relativi per collocare gli oggetti gli uni rispetto agli altri. Grazie a questa struttura dati è possibile mantenere questa coerenza spaziale.

Le classi che supportano questa organizzazione sono:*Spatial*, *Node* e *Geometry*. Gli ultimi due oggetti sono derivati dal primo, che rappresenta l'astrazione di un elemento del grafo, e presenta soltanto informazioni relative al sistema di riferimento relativo ed allo stato di Rendering. Il *Node* offre in aggiunta la possibilità di attaccare e staccare una serie di oggetti *Spatial* o derivati. La *Geometry* è invece la classe più complessa e mantiene tutte le informazioni necessarie per renderizzare quell'oggetto.

2.5 Il ciclo di rendering in Wild Magic

Oltre alle componenti geometriche che caratterizzano gli oggetti, come normali e posizione dei vertici, per effettuare il rendering c'è bisogno soprattutto di definire in che modo questi

vertici saranno resi a video; più banalmente, che colore dovranno avere e come questo colore deve essere generato. Le strutture che definiscono queste proprietà sono dette *effetti*.

Gli effetti racchiudono essenzialmente dei riferimenti agli shader che consentono di visualizzare in una certa maniera l'oggetto, insieme con i parametri necessari in ingresso a questi programmi (come texture, matrici o luci). Il motore Wild Magic nativamente fornisce un numero cospicuo di questi effetti: quelli che emulano la pipeline fissa sono inseriti direttamente nel motore e consentono di applicare le texture, i colori per vertice o le luci; nei moltissimi esempi allegati, invece, ci sono alcune versioni dimostrative di effetti speciali, come ad esempio l'applicazione di iridescenza agli oggetti, rugosità(*bump mapping*) o riflessioni attraverso mappe ambientali.

Gli effetti sono integrati perfettamente con la struttura del grafo della scena. Un effetto, infatti, può essere attaccato direttamente ad una geometria (in questo caso di definisce locale) oppure si può associare ad un nodo: così facendo questo effetto sarà applicato a tutta la porzione di grafo sottostante (effetto definito globale).

Questa struttura, così flessibile e potente, richiede di contro un complesso sistema che ne mantiene la coerenza e l'ottimizzazione. Attraverso l'oggetto *Culler*, ad esempio, Wild Magic è in grado di scandire una porzione della scena e memorizzare in un opportuno vettore, chiamato *VisibleSet*, i riferimenti ai soli oggetti che rientrano nel campo visivo (*frustum*) di una camera: questo consente di scartare dal ciclo di rendering le geometrie che sicuramente non visibili dall'osservatore, con un grande vantaggio in termini di prestazioni.

Per concludere, la sequenza di azioni che portano dalla creazione di un grafo alla resa effettiva di una scena sono:

1. **Creazione degli oggetti** In questa fase si dispongono spazialmente le geometrie, e le si organizza in un grafo; si decide come ogni oggetto deve essere renderizzato associandogli una serie di effetti;

2. **Aggiorna lo stato geometrico** si mantiene la coerenza spaziale degli oggetti propagando i cambiamenti dei sistemi di riferimento e le dimensioni delle *Bounding Box*, cioè le figure geometriche che circondano gli oggetti per stimarne le dimensioni, a seguito di cambiamenti nella topologia del grafo;

3. **Aggiorna lo stato di rendering** si propagano i cambiamenti di stato nel rendering, come l'aggiunta di effetti o materiali globali, impostazione di trasparenze o altro;

4. **Culling** si determina l'insieme di oggetti potenzialmente visibili dalla camera corrente e li si memorizza in forma lineare (passa da grafo a vettore);

5. **Disegno** si applicano tutti gli effetti agli oggetti potenzialmente visibili disegnando di fatto la scena sullo schermo attraverso le primitive proprie dell'API per cui è stato compilato il motore.

2.6 Modifiche apportate a Wild Magic

2.6.1 Modifiche Ultramundum

Alcune modifiche al motore sono state introdotte da Ultramundum per meglio adattarlo all'interfacciamento con Ultraport. Ad esempio, è stato sostituito il *memory manager* del motore, ovvero il gestore delle allocazioni di memoria, che risultava inadeguato, con uno creato appositamente, utilizzato anche in Ultraport, in modo da velocizzare e ottimizzare le prestazioni, in particolar modo quelle dell'interfaccia, e poter usufruire dei dati delle statistiche prodotte.

Inoltre è stato necessario introdurre un nuovo tipo di rendering, denominato *RenderingAlpha*, per visualizzare le scene contenenti anche oggetti semitrasparenti. Si sono infatti verificati in passato problemi con questo tipo di scene legati all'ordine che gli oggetti avevano sullo z-buffer. Infatti, quando viene renderizzato un oggetto semitrasparente, gli oggetti retrostanti ad esso devono necessariamente essere già stati rappresentati in maniera definitiva, altrimenti non vengono visualizzati. In questo tipo di rendering viene semplicemente verificato se un oggetto possiede la componente *alpha* che ne specifica la trasparenza. In caso affermativo, esso viene salvato in un vettore, altrimenti si procede con la normale renderizzazione. Al termine della rappresentazione di tutti gli elementi opachi, gli oggetti contenuti nel vettore saranno ordinati dal più lontano al più vicino e quindi renderizzati uno ad uno.

Queste modifiche approntate da Ultramundum sulla vecchia versione del motore sono state quindi riportate anche sulla versione 4.6, quella a shader usata da noi.

2.6.2 Modifiche Rendering-to-texture

Nell'esecuzione del rendering di una scena su di una texture, nella versione OpenGL del renderer, ci siamo accorti che il motore non agiva in maniera corretta. Per implementare questa funzionalità, vengono usati dei *Frame Buffer Object*(FBO), che rappresentano la modalità più veloce per effettuare *Rendering-to-texture*, operazione molto lenta in proporzione alle altre di un normale ciclo di rendering. Abbiamo verificato che quando si creano i buffer FBO, bisogna istanziare un depth buffer per ognuno di essi, altrimenti si verificano degli artefatti che derivano dalla mancanza dello z-test. Nel motore, i depth buffer vengono dichiarati, ma non vengono mai istanziati ed utilizzati.

Abbiamo quindi introdotto una modifica alla classe *FrameBuffer* del motore che sopperisce a questa mancanza consentendo al Rendering-To-Texture di funzionare nella maniera corretta. Nella versione modificata, i depth buffer vengono istanziati, associati al corrispettivo FBO, utilizzati ed infine distrutti.

2.6.3 Modifiche Shader

Per rendere praticabili le nostre idee e per fornire ad Ultramundum un'applicazione che fosse più completa possibile e mantenesse le stesse funzioni della versione precedente del motore, è stato necessario integrare gli shader forniti nativamente in Wild Magic con degli

altri scritti da noi. In questo modo il motore grafico è ora in grado di replicare il funzionamento della vecchia versione del motore per quanto riguarda Ultraport, di supporartare nuovi effetti e fornire molte più possibilità di crearne.

Un primo aspetto da migliorare ha riguardato le modalità di applicazione delle texture all'interno dell'effetto multitexture. Questa funzione era peraltro ben implementata nella vecchia versione del motore, quella a pipeline di rendering fissa. La prima modifica necessaria ha riguardato l'introduzione di un pixel shader che implementasse la modalità Decalcomania (*Decal* per il motore), ovvero di dare la possibilità di applicare una texture sopra un'altra, esattamente come si attacca una decalcomania ad un oggetto nel mondo reale. Per utilizzare questa funzione è necessario che la texture da applicare abbia un canale *alpha* che ne specifichi la trasparenza. Lo shader riceve in input le coordinate UV per le due diverse texture e le elabora secondo questa equazione:

ColoreFinale = (1 - ColoreDecal.a) * ColoreBase + ColoreDecal.a * ColoreDecal;

Dove *ColoreDecal* è il vettore con le componenti RGBA della texture da applicare come decalcomania, *ColoreBase* il vettore con gli RGBA della texture base e *ColoreFinale* il colore del pixel in output. Come si evince dalla suddetta equazione il canale alpha della texture decalcomania modula i contributi delle due texture al colore finale.

In Wild Magic vengono forniti gli shader per l'effetto multitexture (che associa ad una oggetto più texture contemporaneamente), ma è possibile applicarli a due texture solamente. Sono quindi stati scritti gli shader che rendono possibile l'utilizzo del multitexture per un numero di texture compreso fra tre e otto (massimo limite di texture supportate dagli shader). Per quanto riguarda i vertex shader, essi si limitano a passare in output al pixel shader le coordinate UV delle texture da combinare. I pixel shader campionano le texture in input secondo le corrispettive coordinate UV ricavando un colore per ciascuna di esse. I colori ottenuti in quest'operazione vengono modulati, ossia moltiplicati fra loro, ottenendo così il colore finale del *texel*, il pixel di una texture.

Si è inoltre reso necessario, per effettuare una conversione completa da Wild Magic 3 a Wild Magic 4 per Ultraport, aggiungere una funzionalità che la vecchia versione aveva e che sulla nuova non è più stata implementata. Si tratta delle modalità di applicazione di una texture ad un oggetto sia esso illuminato da una luce o tramite colorazione per vertice, texturizzato o no. Nell'effettuare questa modifica, abbiamo fatto in modo che il ciclo di rendering effettuasse una passata in meno, ottenendo un notevole guadagno in fatto di prestazioni. Il motore infatti, in presenza di un effetto luce o di colorazione per vertice e di un effetto texture o multitexture, eseguiva uno shader per ogni effetto, con conseguente perdita di tempo nell'abilitazione di vertexbuffer e passaggio di parametri. Abbiamo quindi scritto degli appositi shader per l'effetto luce e per l'effetto colorazione per vertice abbinati all'effetto texture o multitexture e modificato il motore in modo che li supportasse [cfr. paragrafo 2.6.4]. Questi nuovi shader sono contraddistinti dal suffisso *_T*. Rispetto a quelli di base da cui sono ricavati, i nuovi vertex shader passano in output ai pixel shader una (o più nel caso del multitexture) coordinata UV, mentre i nuovi pixel shader sono in grado di specificare la modalità di applicazione della texture in input all'oggetto. Esistono quattro possibilità, oltre a quella predefinita dal motore:

- *Modulate*: le componenti RGB o RGBA della luce vengono moltiplicate con quelle della texture. La texture diventa così trasparente e ricopre l'oggetto illuminato.

  Agisce secondo la seguente equazione:

  ColoreFinale = ColoreTexture * ColoreIll;

  dove *ColoreTexture* è il vettore contenente le componenti RGBA del colore ottenuto campionando la texture secondo la sua coordinata UV (entrambi questi parametri sono forniti in input al pixel shader), *ColoreIll* è il vettore, passato dal vertex shader, contenente le componenti RGBA dell'oggetto illuminato o colorato per vertice e *ColoreFinale* è il colore che assumerà il texel al termine dell'operazione;

- *Hard Add*: le componenti di colore di ogni pixel vengono ottenute sommando quelle della luce con quelle della texture. La semplice somma implica che se una o più componenti RGB saturano, ossia superino 1, esse vengano poste a 1.

  Agisce secondo la seguente equazione:

  ColoreFinale = saturate(ColoreTexture + ColoreIll);

- *Soft Add*: le componenti di colore di ogni pixel vengono ottenute sommando quelle della luce con una porzione di quelle della texture. Il risultato è molto simile a quello della tecnica Hard Add, ma il colore è meno brillante e "violento", perchè le due immagini si fondono senza saturare.
  Agisce secondo la seguente equazione:

  ColoreFinale = (1.0f - ColoreIll) * ColoreTexture + ColoreIll;

- *Decal*: la componente alpha della texture modula le componenti RGB della luce. Tanto più è elevato il valore di alpha (ossia tanto più la texture è opaca), tanto più il colore di quel pixel prevale su quello della luce. Per motivi di comodità, spesso si assegnano i valori 0 e 1 (ovvero il minimo e il massimo) alla componente alpha di ciascun pixel della texture in modo da ritagliare la decalcomania che va applicata all'oggetto illuminato.
  Agisce secondo la seguente equazione vista in precedenza all'interno di questo paragrafo.

- *Replace*: è quella predefinita dal motore. Qualunque colore assegnato al pixel dal pixel shader, viene rimpiazzato con quello dell'immagine texture. Un oggetto viene quindi renderizzato solamente con l'effetto texture, senza considerare l'effetto luce o colorazione per vertice, perciò questa modalità non crea solitamente elementi realistici.

  Agisce secondo la seguente equazione:

  ColoreFinale = ColoreTexture;

Introducendo queste modalità, è possibile rendere le scene molto più realistiche. Per vedere esempi della loro applicazione, rimandiamo al capitolo su Ultraport [cfr. paragrafo 4.3.5].

Inoltre, tramite la creazione di opportuni vertex e pixel shader, contraddistinti dal suffisso *_MT*, abbiamo fatto in modo che sia possibile, in questo contesto di illuminazione, far interagire due o più texture fra di loro [cfr. listato A.1]. Nel caso di due texture sono previste sedici modalità d'applicazione, le quattro della luce combinate con le quattro del multitexture, mentre per tre o più texture è possibile applicare solo la Modulate.

Nel caso ad un oggetto siano assegnate due texture (ovvero abbia un effetto multitexture), il programmatore è ora in grado di specificare come esse debbano combinarsi con l'illuminazione della scena (o la colorazione per vertice) e fra loro. Ad esempio, è possibile far sì che su una texture agisca la luce (Modulate) e sul risultato prodotto applicare una texture come una decalcomania (Decal). Da notare che in questo caso il motore agisce in cascata: inizialmente combina la prima texture con l'effetto luce o colorazione per vertice, al risultato viene applicata la seconda texture e così via.

Per quanto riguarda un numero maggiore di texture, è stata fatta l'assunzione che dalla terza in poi le texture possono essere combinate con le prime due solo in modalità Modulate. Ciò è dovuto al fatto che la modulazione è la tecnica più usata in questi casi. Inoltre non avrebbe avuto molto senso creare uno shader apposito per ognuna delle altre combinazioni, ottenendo un grande numero di shader che sarebbero stati utilizzati molto poco.

2.6.4 Modifiche Rendering

Per rimediare ad alcune lacune di Wild Magic su aspetti del rendering considerati importanti, abbiamo dovuto operare alcune modifiche a classi, per fare in modo di supportare determinate funzioni, e al ciclo di rendering, per renderlo più veloce ed efficiente e soprattutto per fare in modo che supportasse i nuovi shader che abbiamo scritto.

Innanzitutto si sono modificate le procedure che creano i nomi degli shader per l'applicazione delle texture nei file *LightingEffect.cpp* e *MultitextureEffect.cpp* della libreria Graphics. Abbiamo fatto in modo che venissero replicati i nomi degli shader originali con l'aggiunta dei suffissi *_T* per le texture singole e *_MT* per le multitexture. Inoltre sono state create le funzioni necessarie al loro caricamento run-time, anche queste replicate dalle originali degli effetti luce e multitexture.

All'interno della funzione *Draw* del file *Rendering.cpp* della libreria Graphics, abbiamo inserito un controllo sugli effetti: nel caso sia applicato all'oggetto un effetto luce, si cerca se fra gli altri effetti dell'oggetto, già noti in quanto associati ad esso in una fase precedente, sia presente un effetto texture o un effetto multitexture. In caso affermativo, viene chiamata un'opportuna funzione da noi creata (*ApplyEffect_T* per la texture, *ApplyEffect_MT* per il multitexture) al posto di quella normalmente evocata dal motore per caricare gli shader di un generico effetto (*ApplyEffect*). Questo stratagemma consente non solo di utilizzare gli shader di cui abbiamo parlato nel paragrafo precedente [cfr. paragrafo 2.6.3], ma anche di effettuare una sola passata di rendering al posto delle due solitamente necessarie quando un elemento della scena ha attaccati due effetti. Ciò è possibile perchè, quando vengono chiamate la ApplyEffect_T o la ApplyEffect_MT, tramite una semplice

implementazione software, si fa in modo che l'effetto texture o multitexture non venga attaccato all'oggetto, bensì venga ignorato.

All'interno della funzione ApplyEffect_T, a meno che non ci si trovi nella modalità di applicazione della texture Replace (in tal caso viene eseguita normalmente l'ApplyEffect in quanto l'effetto texture sovrascrive l'effetto luce), l'immagine texture contenuta nell'effetto texture viene copiata all'interno dell'effetto luce. In questo modo quando si andrà a renderizzare l'elemento della scena, l'effetto luce conterrà l'immagine e l'oggetto apparirà texturizzato e illuminato. In questa fase viene inoltre effettuato il caricamento degli shader: il vertex shader è lo stesso impostato in fase di configurazione dell'illuminazione e dipende dal tipo di luce (ambientale, direzionale, puntiforme o spot), mentre il pixel shader è selezionato in base alla modalità di applicazione (Modulate, Hard Add, Soft Add o Decal) è associata all'immagine texture.

Per quanto riguarda la procedura ApplyEffect_MT, il funzionamento è a grandi linee quello della ApplyEffect_T. Le uniche differenze riguardano vertex e pixel shader che in questo caso, oltre che dalle modalità di applicazione delle texture, dipendono anche dal numero di texture associate all'effetto. Per due texture, il pixel shader viene nuovamente selezionato a seconda della modalità di applicazione è associata all'immagine texture, mentre per tre o più viene impostato automaticamente quello che esegue la modulazione del colore fra le texture e con la luce.

Anche in presenza di un effetto di colorazione per vertice (solitamente alternativo a quello di illuminazione) e di un effetto texture o multitexture il motore effettuava due passate di rendering. Si è provveduto affinchè la passata diventasse una sola, stavolta intervenendo in fase di associazione degli effetti all'oggetto e non in fase di disegno. All'interno della funzione *AttachEffect* del file *Spatial.cpp* della libreria Graphics, che ha il compito di associare un effetto ad un oggetto, andiamo ora a verificare se ci troviamo in presenza di un effetto texture. Se questa ipotesi è verificata, scorriamo la lista degli effetti già attaccati cercando se vi sia un effetto di colorazione per vertice, tenendo presente che questo tipo di effetto viene sempre associato dal motore all'oggetto da disegnare prima di quello texture. Nel caso in cui ciò avvenga, l'effetto texture o multitexture non sarà attaccato all'effetto e verrà chiamata una funzione denominata *AddTexture*, creata appositamente da noi per gestire questa situazione. Questa procedura abbina una o più immagini all'effetto di colorazione per vertice, aggiorna il numero di texture associate a questo effetto (è stata creata anche in questo caso un'apposita funzione) e gli assegna gli shader necessari in base alla modalità di applicazione. In questo modo, in fase di rendering, quando si effettuerà la colorazione per vertice, si andranno anche ad applicare una o più texture. Nel caso si stia trattando invece un effetto multitexture, viene chiamata la funzione *AddMTexture*. Essa si comporta in modo del tutto identico alla AddTexture appena vista.

Wild Magic dà l'opportunità di scegliere tra la colorazione per vertice utilizzando tre parametri di colore (RGB) o quattro (RGBA). Le modifiche sopra citate sono implementate analogamente per entrambe queste possibilità.

2.6.5 Modifiche colorazione per vertice

Questa modifica nasce dalla necessità di inibire l'effetto di illuminazione in presenza di un effetto di colorazione per vertice. Infatti abbiamo verificato che, in caso la scena abbia entrambi gli effetti attaccati, la luce modulava il colore assegnato ad un vertice; il comportamento desiderato dev'essere invece che il vertice presenti il colore impostato dal programmatore senza alterazioni.

Per ottenere ciò si è reso necessario inserire un controllo che, in caso di presenza di un effetto di colorazione per vertice, stacca l'effetto luce eventualmente presente ed evita che se ne attacchino di nuovi. Questo controllo è stato inserito nella funzione di aggiornamento dello stato di una geometria (*Update state*) che viene chiamata ogni qualvolta si deve propagare un effetto attaccato ad un nodo a monte sulle geometrie sottostanti.

2.6.6 Modifiche Multithread

Il motore grafico Wild Magic nella versione nativa non è rientrante, ovvero non supporta il *multithreading*, l'accesso simultaneo da parte di più thread al codice del motore. Durante l'esecuzione di alcune applicazioni in Ultraport infatti, abbiamo riscontrato più di un errore dovuto a questa mancanza. Due o più thread d'esecuzione di Ultraport accedevano alla stessa variabile contemporaneamente.

In genere per risolvere questo tipo di problemi si ricorre a variabili semaforiche o all'implementazione di una regione critica; tuttavia queste tecniche non sono state implementate in Wild Magic, probabilmente perchè ha scopi più didattici che prettamente applicativi.

Per rimediare allora a questa lacuna, la variabile statica è stata trasformata in un vettore di quattro elementi. In questo modo, tramite l'utilizzo di un contatore, si replica il contenuto della variabile per supportare fino a quattro accessi contemporanei. Il contatore viene incrementato ad ogni lettura e riazzerato quando si raggiungono quattro accessi.

Capitolo 3

Shader Program

Gli *shader* sono essenzialmente degli insiemi di istruzioni, quindi dei programmi, utilizzati nella grafica 3D per specificare l'aspetto finale di un oggetto. Ad esempio, possono determinare di quale materiale è costituito tale oggetto e riprodurne le caratteristiche visive e il comportamento fisico, ma anche simulare operazioni ed eventi più complessi quali diffusione e rifrazione della luce, collisioni o riprodurre comportamenti fluidodinamici nella scena.

Caratteristica fondamentale di questi programmi che ne ha favorito il successo è la riutilizzabilità. Infatti, una volta scritto uno shader che simuli ad esempio le caratteristiche del marmo o del legno, è possibile assegnarlo a differenti e numerosi oggetti nella scena o utilizzarlo nuovamente in scene successive, senza più doverlo modificare, con un notevole risparmio in fatto di tempistica e risorse.

Per maggiori informazioni, rimandiamo a [4].

3.1 Storia

Gli shader sono nati come alternativa alla cosiddetta *Pipeline di rendering fissa* (fixed function pipeline - FFP). Essi hanno il compito di sostituire alcuni stadi della pipeline con un modello specifico per ombreggiare gli oggetti della scena. Ciò può significare assegnare ad una figura geometrica un semplice colore solido oppure una complicata combinazione di effetti che crea l'aspetto desiderato.

La principale novità introdotta dagli shader rispetto alla FFP riguarda il modo di processare e inviare dati riguardanti geometria e texture alla scheda video. In passato le schede video disponevano solo di pochi algoritmi fissi per compiere le operazioni di passaggio dei dati. La FFP consentiva di scegliere una di queste funzioni e di impostare alcuni stati della scheda video; quest'approccio era tuttavia molto limitante: non era infatti possibile pensare e creare un effetto elaborato per un gioco o un video con una sola funzione, bensì era necessario assemblarlo in diverse parti.

Per entrare più nel dettaglio, la FFP (o Transform & Lighting (T&L) pipeline), poteva essere controllata solo impostando stati di renderer, matrici e parametri di luci e materiali, quindi era scarsamente ottimizzabile.

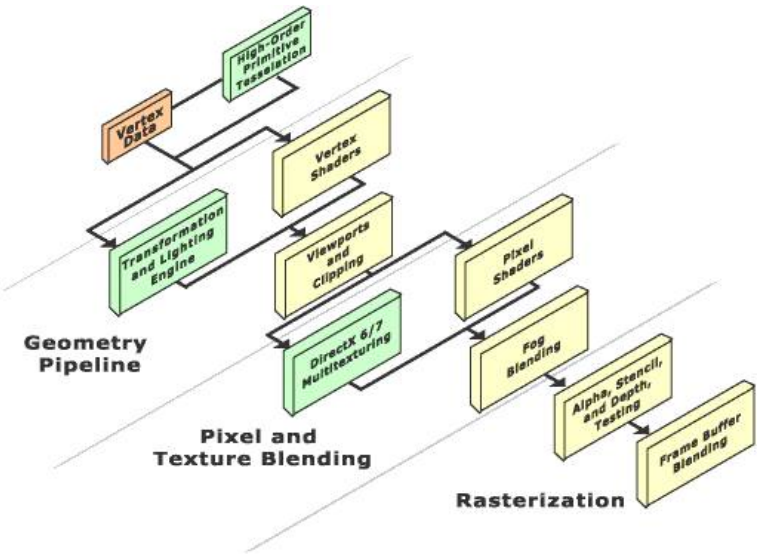


Figura 3.1. La pipeline grafica

Gli shader nascono quindi per fornire al programmatore maggiore libertà e consentono una veduta più ad ampio raggio sul progetto, oltre ad un notevole incremento della potenza e rapidità di calcolo. Su queste basi viene creato (anche se implementato solo via software) il primo shader, nel 2001, dalla Pixar's RenderMan technology che produce cartoni animati in 3D di successo quali *Toy Story* e *A Bug's Life*.

Non vengono più impostati dei parametri per controllare una pipeline, bensì viene scritto un programma che viene eseguito direttamente sull'hardware della scheda video. L'hardware di una scheda video che utilizza la FFP invece non supporta gran parte dei calcoli necessari per i vertici, ma li delega alla *CPU* (Central Processing Unit) e al motore grafico, causando a volte anche ridondanza di operazioni.

3.2 Tipi di shader

Gli shader si dividono essenzialmente in due categorie, a seconda della parte di pipeline grafica che intendono emulare:

- *Vertex shader* : emulano le funzioni di trasformazione e proiezione della luce della pipeline grafica. Gestiscono quindi i vertici dell'oggetto e ne possono alterare la posizione, i colori, le coordinate texture, eccetera...
- *Pixel shader* (o *fragment shader*) : emulano la parte pipeline che somma i colori e li applica all'oggetto (texture mapping della pipeline grafica). Vanno ad elaboare i singoli pixel per applicare effetti come ad esempio bump mapping o nebbia.

Per le nuove API (Application Programming Interface) *DirectX10* per Windows Vista è stato creato un terzo tipo di shader, i *geometry shader*, che si collocano a metà fra i vertex e i pixel shader. Vengono usati per combinare una serie di vertici per formare un oggetto più complesso al quale applicare effetti di pixel shading. Essi prendono, come ingresso, i vertici che sono stati trasformati dal vertex shader e per ogni vertice generarano fino a 1024 vertici in uscita con una tecnica nota come amplificazione dei dati. Il geometry shader è anche in grado di rimuovere dati dalla GPU emettendo in uscita un numero di vertici inferiore a quello di ingresso.
Nel nostro progetto tuttavia non sono stati considerati in quanto non supportati dal motore grafico usato.

3.3 Vertex shader

I vertex shader consentono di modificare gli attributi dei vertici della geometria della scena. Sono infatti i vertex shader ad elaborare secondo precise formule matematiche le matrici, i vertici e tutte le altre impostazioni puramente geometriche per generare gli oggetti della scena. Solo dopo la loro esecuzione vengono applicati tutti gli altri effetti.

Fra gli attributi dei vertici, i più importanti sono le coordinate dei vertici stessi, i parametri di illuminazione e le coordinate UV per le texture. Si può dunque modificare la geometria di partenza in modo massiccio. Il calcolo di questi attributi non viene più diviso fra CPU e motore grafico, ma viene eseguito dalla *GPU* (Graphics Processing Unit) direttamente sulla scheda video. Questa scelta consente una grande potenza di calcolo perchè sfrutta la possibilità di eseguire in parallelelo più vertex e pixel shader nelle numerose pipeline presenti nella GPU, ottimizzando così il tempo di elaborazione dei dati.

In realtà, i vertex shader non modificano il tipo di dati ma, semplicemente, ne cambiano i valori; in questo modo, un vertice appare con una texture o un colore diverso, oppure con una posizione diversa.

I vertex shader consentono di eseguire le seguenti operazioni:

- Costruzione di geometrie procedurali (simulazione di vestiti, bolle di sapone, eccetera...);
- Generazione di texture;
- Advanced Keyframe Interpolation (per modelli facciali complessi);
- Rendering di sistemi di particelle;
- Modifiche real-time della percezione visiva (effetto delle lenti, effetto sott'acqua, eccetera...);
- Modelli avanzati d'illuminazione;
- Primi passi del Displacement Mapping;

Questi sono solo esempi delle potenzialità dei vertex shader, ci sono numerose altre possibilità di utilizzo grazie alla flessibilità che questi programmi danno.

3.3.1 Processamento del vertice

Gli oggetti in una scena tridimensionale sono formati da poligoni, aventi un certo numero di vertici; questi ultimi vengono sottoposti ad una serie di trasformazioni che portano al disegno della scena stessa, chiamato in inglese rendering. A questa serie di trasformazioni si dà il nome di *pipeline*, per l'analogia con una catena di montaggio.

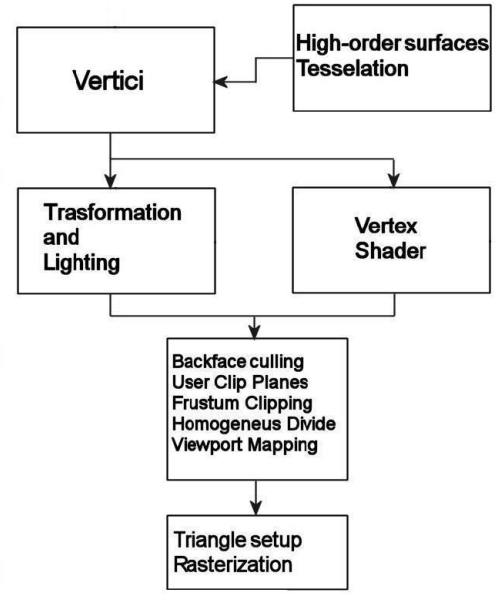


Figura 3.2. Pipeline per i vertici

Nella figura 3.2 è rappresentata la pipeline per il processamento dei vertici di una scena: essa viene alimentata da un flusso (detto anche *stream*) di vertici, che possono essere forniti direttamente dall'applicazione oppure provenire dallo stadio di *Tessellation*. Per spiegare il compito di quest'ultimo, è necessario introdurre le *high-order surfaces*, ovvero delle superfici geometriche create a partire da una famiglia di curve parametriche come Bezier, B-Spline e Catmull-Rom, attraverso le quali è possibile rappresentare superfici curve in maniera molto realistica, senza la spigolosità di quelle approssimate da triangoli. Esse sono l'ideale per rappresentare parti anatomiche nei personaggi animati. Lo stadio di Tesselation riceve in input i parametri per la creazione di queste superfici, ad esempio i punti di controllo delle curve di Bezier, e fornisce in output uno stream di vertici, in cui

è stata decomposta la superficie di partenza. Questa operazione consente di aumentare il livello di realismo della scena.

Uscito dallo stadio di Tesselation, il flusso di vertici ha due alternative a seconda del modello scelto: pipeline di rendering fissa o shadel model. Nel primo caso i vertici saranno processati nel Transformation and Lighting (T&L), nel secondo dalla Vertex Shader Unit. Il compito di quest'ultima è quello di eseguire, per ogni vertice in input, un programma, chiamato vertex shader, cui sono delegate le trasformazioni geometriche e l'illuminazione. Di questo particolare stadio parleremo più avanti [cfr paragrafo 3.3.2].

Il *Transformation and Lighting (T&L)* ha il compito di applicare ai vertici le trasformazioni geometriche e calcolare l'intensità luminosa di ciascuno di essi. Le trasfomazioni geometriche riguardano la definizioni delle matrici World, View e Projection. Per poter disegnare l'intera scena, è necessario posizionare gli oggetti nel mondo: ciò si realizza moltiplicando tutti i vertici per una matrice di rototraslazione; in questo modo si passa dal sistema di riferimento dell'oggetto a quello dell'intero mondo: questa trasformazione viene chiamata *world transformation*. Dopo aver posizionato gli oggetti, bisogna ora aggiustare la posizione della telecamera con la quale viene inquadrata la scena: è necessario traslare e ruotare gli oggetti in modo che la telecamera sia posta nell'origine del sistema di riferimento. Questa operazione viene svolta dalla matrice View, o di vista, e viene chiamata ovviamente *view transformation*. Siccome la superficie del monitor è bidimensionale, è necessaria un'ulteriore trasformazione per passare dallo spazio a tre dimensioni a quello a due: la *projection transformation*, che effettua la proiezione prospettica di ciascun vertice.

Questo stadio calcola anche l'intensità luminosa di ciascun vertice in base a un modello di illuminazione, cioè in maniera fissa; il programmatore può impostare una serie di parametri di esso quali la posizione e il colore delle luci, i materiali degli oggetti, l'attenuazione della luce e altri.

Dopo essere stati trasformati e illuminati, i vertici vengono sottoposti al *Backface Culling*, operazione che rimuove le facce che presentano la normale rivolta in direzione opposta all'osservatore.

A seguire vengono eliminati tutti i poligoni che non sono compresi nella parte di spazio individuata dalla normale agli *User Clip Planes*, che sono molto utili nel portal rendering e possono essere al massimo sei.

Successivamente viene effettuato il *Frustum Clipping*, utilizzando il *viewing frustum*, che è un tronco di piramide con la telecamera nel suo vertice; il suo volume rappresenta la porzione di spazio visibile attraverso di essa. L'operazione di clipping può essere descritta così: se un poligono giace completamente fuori dal volume del frustum, viene scartato; se giace completamente al suo interno allora esso passa allo stadio successivo; se invece interseca il frustum, viene tagliato (da qui deriva il termine clipping) e solo la parte in comune con il frustum passa allo stadio successivo.

Questo è rappresentato dall'*Homogeneus Divide*, in cui le coordinate x,y,z dei vertici vengono divise per quella omogenea w, per ottenere le coordinate normalizzate: x,y sono comprese tra (-1,1) mentre la z è compresa in (0,1). Infine i vertici vengono sottoposti al *Viewport Mapping*, che serve per mappare le coordinate normalizzate in quelle dello schermo in base alla risoluzione video di quest'ultimo.

A questo punto i vertici vengono trasformati in pixel dallo stadio di triangle setup, che funge da "ponte" e che sarà descritto insieme ai successivi passi quando parleremo dei pixel shader [cfr. paragrafo 3.4.1].

3.3.2 Architettura della Vertex Shaders Unit

In questo paragrafo ci occupiamo di analizzare più dettagliatamente lo stadio Vertex Shader Unit della pipeline grafica. Esso riceve in input un vertice, applica ad esso le operazioni descritte nel vertex program e lo ritorna in output.

Tutti i dati presenti in un vertex shader sono rappresentati sui 128 bit di un quad-floats, ossia 4 variabili da 32 bit ciascuna (dimensione di un float), denominate rispettivamente x, y, z, w.

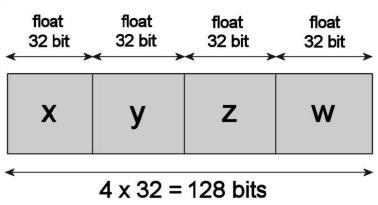


Figura 3.3. Registri di un vertex shader

A livello hardware, un vertex shader funziona come un processore di tipo SIMD che esegue un'istruzione per volta applicandola su dati multipli e opera su un insieme di variabili di 32-bit (fino a quattro per istruzione). Questa scelta è dovuta al fatto che la maggior parte dei calcoli per trasformazioni e illuminazioni sono eseguiti mediante matrici 4x4 o quaternioni.

L'architettura di questa unità è rappresentata in figura 3.4: essa riceve i dati relativi vertice (ad esempio: posizione, normale, colore, coordinate texture) dai registri di input. Inoltre può avere in input anche delle costanti (matrici, posizione della luce, ...) fornite dagli appositi registri. Esegue quindi le istruzioni contenute nel vertex program, avvalendosi di registri temporanei a lettura/scrittura per memorizzare risultati intermedi ed infine memorizza i risultati delle trasformazioni nei registri di output, che conterranno il vertice trasformato ed illuminato.

Esiste anche un ulteriore registro intero, denominato address register a0, che serve per un indirizzamento relativo delle costanti. L'architettura appena descritta è relativa alla versione 1.1 dei vertex shader, la quale si è successivamente evoluta nella versione 2.0. Quest'ultima differisce dalla precedente non solo per il maggior numero di registri costanti (256 contro 96), ma anche per l'introduzione di sedici nuovi registri costanti di tipo intero ed altrettanti di tipo booleano. Inoltre è stato aggiunto un registro intero che funge da contatore nei loop.

Esiste anche la versione 3.0, che introduce altre due tipologie di registri (predicate e sampler) e rende possibile specificare la semantica dei registri di output.

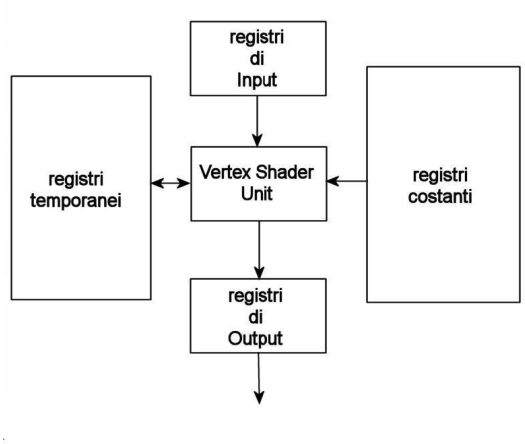


Figura 3.4. Architettura di un vertex shader

La versione 4.0 è invece relativa a Windows Vista e presenta un notevole incremento del numero di registri e degli slot per le istruzioni.
Da notare che, in assenza di supporto hardware, è possibile usare l'emulazione software della VS Unit, che tuttavia usufruisce di prestazioni di calcolo inferiori, visto che sarà la CPU stessa ad eseguire il vertex program ed i risultati dovranno poi attraversare il bus per giungere alla pixel unit.

La sintassi delle istruzioni per la Vertex Shaders Unit ricalca quella di un tipico assembler per CPU RISC, ovvero:

NomeOp dest, [-]s0 [, [-]s1 [, [-]s2]] ; commento

dove *NomeOp* è il nome dell'istruzione, *dest* è l'operando destinazione, mentre *s0*,*s1*,*s2* sono quelli sorgenti; le parentesi quadre indicano che l'operando è opzionale, come pure lo è il segno meno davanti ad uno di essi.

Oltre alle istruzioni normali, ci sono le macro-ops, che combinano quelle normali per fornire funzionalità di alto livello come moltiplicazione di un vettore per una matrice, prodotto vettoriale e altre. Ciascuna istruzione può occupare uno o più slots: ad esempio le istruzioni normali ne occupano uno mentre le macro-ops più di uno; ci sono anche delle istruzioni da zero slots: si tratta di quelle per impostare le costanti.

Per la specifica 1.1 un programma può occupare al massimo 128 slots, mentre nella 2.0 questo limite è stato portato a 256 mentre nella 3.0 si può superare 512.

Nel vertex shader versione 1.1, a livello assembler, non sono permessi cicli, salti ed espressioni condizionali; ciò permette un'esecuzione lineare del programma, un'istruzione dopo l'altra, con guadagno in semplicità, chiarezza ma impatta sulla lunghezza di codice. Già la versione 2.0 tuttavia presenta alcune novità che smentiscono in parte questa filosofia, come l'introduzione della chiamata a subroutine, dell'istruzione condizionale *if*, dei cicli, del prodotto vettoriale, della normalizzazione, dell'interpolazione lineare e della

funzione potenza. Infatti ad alto livello, ad esempio nel linguaggio di programmazione per shader chiamato *C for Graphics* (o più brevemente *Cg*, di cui parleremo diffusamente in seguito), le espressioni condizionali sono comunque permesse. Esse sono tradotte come due rami diversi dello stesso programma, ma vengono eseguiti entrambi. In seguito, in base a quale delle condizioni risulti verificata, vengono restituiti i dati contenuti nei registri "corretti", mentre gli altri vengono cancellati. Abbiamo verificato che quest'approccio non risulta per niente efficiente, in quanto vi è un notevole spreco di tempo e potenza di calcolo. Per ottimizzare le prestazioni delle nostre applicazioni abbiamo, in taluni casi, modificato dei vertex shader ottenendo due versioni diverse degli stessi, caricando ora uno ora l'altro in base alle esigenze del programma. In tal modo si ha effettivamente un maggior numero di shader, ma si guadagna sicuramente molto in efficienza e velocità di calcolo, doti indispensabili in un'applicazione real-time.

Riportiamo di seguito una tabella riassuntiva per i diversi modelli di vertex shader [cfr. figura 3.5].

	VS 1.1	VS 2.0	VS 2.0a	VS 3.0	VS 4.0
Numero di slot di istruzioni	128	256	256	≥ 512	4096
Predizione istruzioni	No	No	Sì	Sì	Sì
Registri temporanei	12	12	13	32	4096
Registri per le costanti	96	≥ 256	≥ 256	≥ 256	16x4096
Operatori di bitwise	No	No	No	No	Sì

Figura 3.5. Tabella riassuntiva per i modelli di vertex shader

3.3.3 Scrittura di un vertex shader

Questi programmi possono ricevere in input due tipi di dati relativi alla scena: i dati costanti e i dati variabili in base al vertice processato.

Il primo tipo di dati, caratterizzato a livello di programmazione Cg dalla specifica *uniform* anteposta alla dichiarazione, riguarda solitamente matrici di trasformazione, vettori, dati per interpolazioni di vario tipo, posizione della luce; più in generale, tutti i parametri propri dell'effetto da realizzare che abbiano la caratteristica di essere costanti per tutti i vertici. A livello di esecuzione infatti, la scheda video esegue in parallelo più vertex shader, processando i vertici in modo indipendente; tuttavia i parametri uniform non variano. I registri in cui sono contenute queste costanti sono caricati dalla GPU prima che il vertex shader riceva e processi i parametri definiti dal programmatore; ovviamente questi registri non possono essere scritti o modificati dal vertex shader.

Il secondo tipo di dati è trasferito alla GPU tramite un buffer, chiamato *Vertex Buffer*, che contiene le informazioni sui singoli vertici della scena. Queste possono essere: la posizione dei vertici, le normali,i colori, le coordinate texture associate ad ogni singolo vertice. Il tipo di informazione specificata si esprime in Cg tramite il *semantic*, un'etichetta che accompagna la dichiarazione del dato. Esempi di semantic sono *POSITION*, *COLOR*, *NORMAL* che identificano dati relativi a posizione, colore e normali. Ogni vertex shader

deve obbligatoriamente avere in input dei dati di tipo POSITION, altrimenti non sarebbe possibile identificare il vertice per cui eseguire i calcoli specificati nel programma.

Di seguito è riporato un esempio semplice di vertex shader, scritto in linguaggio Cg:

```
void v_ProjectedDepth
(
    in float4        kModelPosition : POSITION,
    out float4       kClipPosition : POSITION,
    out float        fDepth : TEXCOORD0,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(kModelPosition,WVPMatrix);

    // Save the normalized distance from the light source.
    fDepth = kClipPosition.z/kClipPosition.w;
}
```

Nella dichiarazione del programma deve anche essere specificato se i dati sono in input (*in* anteposto alla dichiarazione) o in output (*out*). Solitamente in output vengono forniti gli stessi vertici passati in input, ma modificati, ovvero trasformati dalle tre dimensioni alle due del *clip space* dello schermo tramite l'uso di una matrice omogenea e di una matrice di proiezione. Inoltre possono essere forniti in output dati utili al pixel shader che variano di applicazione in applicazione (non sempre sono necessari). Per contenere questi dati, ci sono a disposizione fino a tredici registri (identificati da una *o* all'inizio del nome) accessibili solo in scrittura dal vertex shader.

A seguire riportiamo invece la traduzione in assembler effettutata dal compilatore Cg del vertex shader appena considerato:

```
!!ARBvp1.0
# cgc version 1.5.0014, build date Sep 18 2006 20:36:59
# command line args: -profile arbvp1
# source file: ProjectedShadow.cg
#vendor NVIDIA Corporation
#version 1.5.0.14
#profile arbvp1
#program v_ProjectedDepth
#semantic v_ProjectedDepth.WVPMatrix
#var float3 kModelPosition : $vin.POSITION : POSITION : 0 : 1
#var float4 kClipPosition : $vout.POSITION : HPOS : 1 : 1
#var float fDepth : $vout.TEXCOORD0 : TEX0 : 2 : 1
#var float4x4 WVPMatrix :  : c[1], 4 : 3 : 1
PARAM c[5] = { program.local[0..4] };
TEMP R0;
TEMP R1;
MUL R0, vertex.position.y, c[2];
MAD R0, vertex.position.x, c[1], R0;
MAD R0, vertex.position.z, c[3], R0;
```

```
ADD R0, R0, c[4];
RCP R1.x, R0.w;
MUL result.texcoord[0].x, R0.z, R1;
MOV result.position, R0;
END
# 7 instructions, 2 R-regs
```

La prima parte di codice, quella le cui righe sono iniziate dal carattere # specifica attributi quali data di creazione, versione del Cg compiler e altri. Inoltre vengono definite le variabili che il programma andrà ad utilizzare divise fra quelle costanti (identificate dalla parola semantic) e quelle numeriche che vanno inizializzate. La seconda parte del programma è costituita da istruzioni assembler vere e proprie, solitamente il compilatore fa in modo che esse siano le più semplici possibile in modo da guadagnare in fatto di velocità ed efficienza.

3.4 Pixel shader

Il pixel shader è un programma per l'applicazione di effetti grafici a modelli poligonali che viene eseguito direttamente sul microprocessore della scheda video, la GPU. Si basa sull'elaborazione in tempo reale dei singoli pixel. Le attuali schede video più evolute hanno dei pixel shader in grado di elaborare circa 3,3 miliardi di pixel al secondo e di applicare per ogni fotogramma, considerando una frequenza di cento immagini al secondo, fino a sedici effetti grafici. Numerosi limiti pratici, a cominciare dalla velocità della memoria video, impediscono di raggiungere questo risultato.

Il pixel shader, o fragment shader secondo la nomenclatura OpenGL, consente quindi di applicare effetti come il bump mapping, ombre, esplosioni, rifrazioni, effetti focali e distorsioni permettendo una migliore simulazione degli effetti di illuminazione e un aspetto più realistico di superfici dalle proprietà ottiche particolari.

Hanno il compito di personalizzare il processo di elaborazione finale di ogni singolo pixel in base al colore di base, alle texture e ai vertici elaborati dal vertex shader da cui dipendono. I vertex shader, infatti, restituiscono semplicemente la scena completa con tutti i vertici in posizione finale sullo schermo, ognuno con un colore e la posizione delle texture, senza specificare come questi attributi vengano utilizzati su ogni singolo pixel, cosa di cui si occupano i pixel shader.

Insieme ai vertex shader, formano un tandem davvero efficace per creare effetti grafici ad alta qualità, impossibili da ottenere con la pipeline fissa.

I pixel shader consentono di eseguire, fra le altre, queste operazioni:

- Per-pixel lighting;
- True phong shading;
- Illuminazione anisotropica;
- Effetti volumetrici;

- Bump mapping avanzato;
- Texture procedurali e perturbazione di texture;

3.4.1 Processamento del pixel

Dopo che un vertice è stato trasformato, illuminato, sottoposto a clipping e mappato nella viewport [cfr. paragrafo 3.3.1], arriva nello stadio di *Triangle Setup/Rasterization*, che fa da "ponte" tra il mondo dei vertici e quello dei pixel.

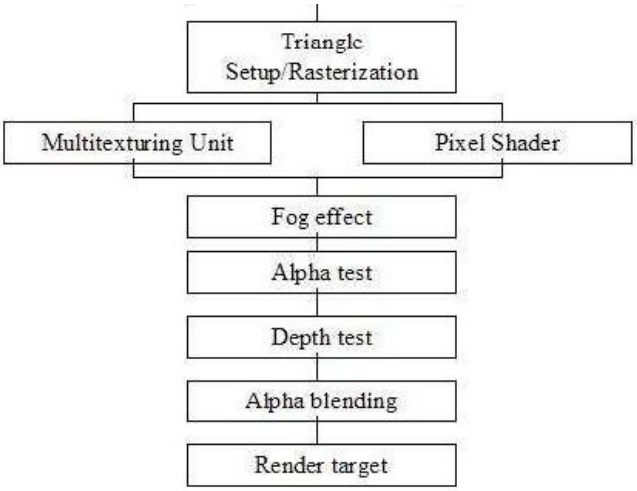


Figura 3.6. Pipeline per un pixel

La fase di *Triangle Setup* calcola i parametri per la rasterizzazione di un triangolo: esso infatti viene scomposto [cfr. figura 3.7] in linee orizzontali, chiamate *scanline*; in questa fase, per ogni scanline, si calcolano le coordinate del primo e dell'ultimo pixel da disegnare.

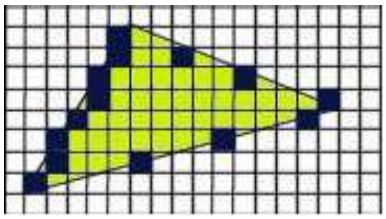


Figura 3.7. Triangolo scomposto in scanline

Successivamente la fase di *Rasterization* riempie ogni scanline con i pixel; il colore, la coordinata z e quelle texture di ciascun pixel vengono calcolate per interpolazione lineare di quelle dei vertici.

Ai pixel così calcolati possono essere aggiunte le texture ed il colore speculare mediante la *Multitexturing Unit* oppure un Pixel Shader. Di quest'ultimo parleremo diffusamente fra poco [cfr. paragrafo 3.4.2], mentre per quanto riguarda la Multitexturing Unit, semplificando, diciamo che consente l'applicazione di un massimo di otto texture ad un pixel, specificando le operazioni consentite fra di esse.

Dopo questa fase, il pixel arriva nello stadio successivo: il *Fog*. Questo passaggio serve ad aggiungere un effetto nebbia alla scena: per ciascun pixel viene calcolato un fattore fog che viene combinato con il colore precedentemente calcolato; il fattore fog dipende dalla distanza del pixel dal'osservatore, ovvero dalla sua coordinata z.

Successivamente il pixel viene sottoposto all'*Alpha Test*: se il suo valore alpha, ossia quello che definisce la trasparenza, è minore di una soglia impostabile dal programmatore, il pixel viene scartato; in questo modo si possono applicare texture con bordi trasparenti; questa tecnica è usata ad esempio per rappresentare gli alberi.

Il passo successivo è noto come *Depth Test*, che confronta la coordinata z del pixel con quella presente, alle stesse coordinate x ed y, nello *Z-buffer*; quest'ultimo ha le stesse dimensioni della finestra video e memorizza la coordinata z di ciascun pixel disegnato. Supponendo che l'asse delle z sia entrante nello schermo, se la coordinata z del pixel è maggiore di quella memorizzata nel buffer, allora significa che esso si trova dietro al pixel già disegnato e quindi sarà scartato in quanto non visibile.

Il successivo stadio è quello di *Alpha Blending*, in cui i dati relativi al colore del pixel vengono fusi con quelli dei pixel già presenti nel Render Target. L'equazione usata per la "fusione"(in inglese *blending*), è la seguente:

$$\begin{aligned} \text{ColoreFinale} = \text{ColorePixelSorgente} * \text{SourceBlendFactor} + \\ + \text{ColorePixelDestinazione} * \text{DestBlendFactor} \end{aligned}$$

Dove *ColorePixelSorgente* è il colore del pixel corrente nella pipeline, mentre *ColorePixelDestinazione* è quello del pixel giò presente nel Render Target; *Source* e *DestBlendFactor* sono dei numeri reali, compresi tra 0 e 1, che indicano l'ammontare della fusione.

Infine il pixel arriva nel *RenderTarget*, lo stadio che specifica dove sarà renderizzato l'oggetto. Normalmente questo sarà il backbuffer della scheda video, ma si potrebbe reindirizzare su una texture; questa tecnica è usata, ad esempio, per rappresentare i videowall nei giochi di guida o il riflesso dell'ambiente circostante su di un oggetto, tecnica chiamata *environment cube mapping*.

Come ultimo passaggio, il contenuto del backbuffer viene copiato nel framebuffer, in sincronia con il pennello elettronico, per visualizzare la scena, in genere sul monitor.

3.4.2 Architettura della Pixel Shaders Unit

Dopo aver visto la collocazione della Pixel Shaders Unit nella pipeline grafica, ne descriviamo l'architettura. Le caratteristiche delle Pixel Shader Unit sono in continua evoluzione; di seguito riportiamo brevemente come si sono evoluti in termini di prestazioni e potenzialità.

Le versioni 1.1-1.3 sono troppo limitate e non le prendiamo in esame.
La versione 1.4 è la prima degna di nota, il set di istruzioni adotta un approccio di tipo RISC, con poche istruzioni semplici e versatili; inoltre il numero massimo di istruzioni è fissato a 28.

Questa unità è costituita dalla ALU e dai registri (cfr. figura 3.8). La *ALU* (Arithmetic Logic Unit - Unità aritmetica e logica) esegue le istruzioni dello shader, utilizzando i registri per l'input/output e per memorizzare i risultati temporanei.

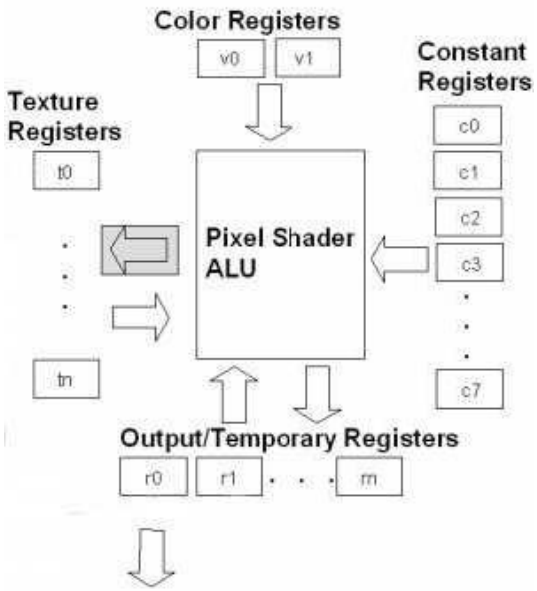


Figura 3.8. Architettura di un pixel shader

Il formato dei registri è un vettore a quattro componenti fixed point, chiamati *r*, *g*, *b*, *a* che rappresentano le componenti rossa, verde, blu ed alpha di un colore.

Ci sono quattro tipi di registri: colore, costanti, temporanei e texture. I registri colore v0,v1 contengono il colore del vertice, calcolato precedentemente dal vertex shader o dalla pipeline fissa, nell'intervallo [0.0 - 1.0] per ciascuna componente. Sono registri da cui si può soltanto leggere, dato che rappresentano un input; solitamente v0 contiene il colore diffuso, mentre v1 quello speculare. I registri costanti sono otto, etichettati c0-c7; ciascuno può contenere un vettore di quattro elementi fixed point, nel range [-1,+1]. Dal punto di vista del pixel shader, essi sono in sola lettura, all'applicazione invece è consentito leggerli e scriverli.

I sei registri texture t0-t5 sono in sola lettura e contengono le coordinate texture.
I sei registri temporanei r0-r5 sono utilizzati per memorizzare risultati intermedi, quindi è possibile un accesso sia in lettura che in scrittura. Il registro r0 viene utilizzato come output del pixel shader: in esso sarà memorizzato il colore definitivo del pixel.

In seguito è stata introdotta la versione 2.0, che porta il numero massimo di istruzioni a 96, ed aggiunge nuove istruzioni e registri. Rispetto all'architettura della pixel shader unit 1.4, la versione 2.0 conserva l'impostazione di base della versione precedente, introducendo alcune novità.

Il miglioramento più importante consiste nel formato di tutti i registri, che adesso sono composti da quattro componenti floating point da 32 bit, mentre prima erano in notazione fixed point. Ciò consente una maggiore precisione nei calcoli aritmetici, che si tramuta in una maggiore qualità visiva. Inoltre i registri costanti sono diventati 32, denominati sempre c0-c31, mentre nella versione precedente erano solo otto. I registri temporanei sono passati da sei a dodici.

Sono stati introdotti sedici nuovi registri denominati sampler s0-s15 che rappresentano le unità di texture sampling; in queste unità, come abbiamo visto, si ricava il texel corrispondente alle coordinate texture selezionate. Sono stati introdotti cinque nuovi registri di output: oC0-oC3 per il colore e oDepth per la coordinata z da memorizzare nello z buffer.

Un'importante differenza rispetto alla precedente versione è la mancanza dei modificatori _bias, _bx2, _x2 per i registri sorgenti.
Una notevole miglioria è rappresentata dal numero massimo di istruzioni utilizzabili in un pixel shader: si è passati da 12 texture e 16 aritmetiche della precedente versione a ben 32 texture e 64 aritmetiche della nuova. Inoltre non c'è più bisogno di effetturare due passate di rendering quando il numero di istruzioni supera le sei di tipo texture e le otto aritmetiche.

Con l'introduzione della versione 2.0, ci sono anche nuove istruzioni aritmetiche: *abs* per calcolare il valore assoluto; *crs* per il prodotto vettoriale; *exp* e *log* rispettivamente per la funzione esponenziale ed il logaritmo; *frc* ritorna la parte frazionale delle componenti; le istruzioni *m4x4*, *m4x3*, *m3x4*, *m3x3*, *m3x2* calcolano i prodotti tra un vettore ed una matrice; *nrm* normalizza un vettore; *pow* calcola una potenza; *rcp* calcola il reciproco; *rsq* calcola la radice quadrata del reciproco; infine *sincos* calcola seno e coseno.

La versione 3.0 specifica solo il numero minimo di istruzioni che l'hardware deve supportare: 512; inoltre aggiunge le istruzioni per il controllo del flusso del programma.
Per Windows Vista è stata introdotta la versione 4.0 della pixel shader unit con numero di istruzioni illimitate. Il modello di shader introdotto in Vista utilizza una stessa gamma di istruzioni per gestire tutte e tre le tipologie di shader (vertex shader, geometry shader e pixel shader) con notevoli vantaggi in termini di semplicità e flessibilità.

La tabella sottostante riassume le più importanti caratteristiche di ciascun modello di pixel shader [cfr. figura 3.9].

3.4.3 Scrittura di un pixel shader

Un semplice esempio di pixel shader, scritto in linguaggio Cg, è riporato di seguito:

```
void p_ProjectedDepth
(
    in float   fDepth : TEXCOORD0,
    out float4 kPixelColor : COLOR)
```

	PS 1.4	PS 2.0	PS 2.0a	PS 3.0	PS 4.0
Limite di istruzioni texture	12	32	Illimitate	Illimitate	Illimitate
Slot di istruzioni	12 + 16	32 + 64	512	≥ 512	≥ 65536
Istruzioni eseguite	12 + 16	32 + 64	512	65536	Illimitate
Registri temporanei	6	12	22	32	4096
Registri costanti	8	32	32	224	16x4096
Swizzling arbitrario	No	No	Yes	Yes	Yes
Registro contatore dei cicli	No	No	No	Yes	Yes
Operatori di bitwise	No	No	No	No	Yes

Figura 3.9. Tabella riassuntiva per i modelli di pixel shader

```
{
    kPixelColor.rgb = fDepth;
    kPixelColor.a = 1.0f;
}
```

Per i pixel shader l'unico parametro obbligatorio in uscita è il colore del pixel che si sta processando. Questo valore sarà il colore finale del pixel sul monitor. In input invece è necessario avere le coordinate del vertice elaborate dal vertex shader associato.

Come i vertex shader, anche i pixel shader ammettono due tipi di dati, quelli costanti per tutti i pixel e quelli che identificano ognuno di essi, come come posizione, coordinate texture, normali e altri.

La corrispondente traduzione in assembler del pixel shader in esame, effettuata con il Cg compiler, è qui riportata:

```
!!ARBfp1.0
# cgc version 1.4.0000, build date Jun  9 2005 12:09:02
# command line args: -profile arbfp1
# source file: ProjectedShadow.cg
#vendor NVIDIA Corporation
#version 1.0.02
#profile arbfp1
#program p_ProjectedDepth
#var float fDepth : $vin.TEXCOORD0 : TEX0 : 0 : 1
#var float4 kPixelColor : $vout.COLOR : COL : 1 : 1
#const c[0] = 1
PARAM c[1] = { { 1 } };
MOV result.color.xyz, fragment.texcoord[0].x;
MOV result.color.w, c[0].x;
END
# 2 instructions, 0 R-regs
```

Si noti come gran parte del testo riguarda attributi del programma e definizioni di variabili, mentre le istruzioni vere e proprie sono soltanto due, peraltro molto semplici.

3.5 Programmazione degli shader

Gli shader si possono sviluppare direttamente codice assembler per la scheda video; per semplificare il lavoro dei programmatori, tuttavia, sono stati inventati dei linguaggi a più alto livello, simili al C, e degli opportuni compilatori si preoccupano di generare l'assembler per noi.

Questi linguaggi stati sviluppati sia da case produttrici di schede video, NVDIA e ATI, sia da software house come Microsoft e OpenGL.

Attualmente i linguaggi disponibili sono:

- *HLSL* (High-Level Shading Language): creato da Microsoft per DirectX, è il più usato;
- *GLSL* (OpenGL Shading Language): creato da OpenGL;
- *CG* (C for Graphics): creato da NVDIA, è pressochè identico all'HLSL;
- *ASHLI* (Advanced Shading Language Interface): creato da Ati, scarsamente usato;

L'evoluzione degli shader ha portato alla creazione del formato *fx* (*CGfx* per il Cg) che ha la caratteristica di racchiudere in un unico file il vertex e pixel shader che descrivono un effetto e di definire in testa le strutture dati comuni ad essi.

Questo tipo di effetti è concepito per essere interpretato e compilato da apposite librerie fornite dalle API grafiche; queste potranno essere usate sia dai motori grafici che dagli ambienti di modellazione. In questo modo chi genera le scene avrà la stessa resa visiva di chi le utilizzerà all'interno di un motore grafico. Questi formati saranno supportati nella versione 5 del motore Wild Magic.

Per il nostro progetto abbiamo scelto di utilizzare Cg come linguaggio di scrittura degli shader. Esso era l'unico che desse garanzia di funzionamento sul motore grafico su cui avremmo lavorato, Wild Magic 3D Engine 4.6 [cfr. capitolo 2], scelto da Ultramundum per i rendering 3D. In linea teorica il motore dovrebbe supportare anche Shader scritti in HLSL, ma dopo alcune prove non riuscite e dopo aver analizzato i problemi di portabilità (si vuole infatti creare un'applicazione in grado di girare su più piattaforme e DirectX è compatibile solo con Windows), abbiamo optato per utilizzare Cg. E' possibile inoltre compilare shader in Cg in base a diversi profili a seconda dell'API che si sta utilizzando.

Wild Magic delega la compilazione degli shader all'apposito compilatore Cg, ma ne esegue lui stesso il caricamento a run-time. Il motore, tramite un opportuno parser contenuto in una delle sue classi, analizza il codice assembler dello shader stabilendo quali variabili saranno necessarie in input e di che tipo esse siano. Sulla base del nome di queste variabili, esse vengono classificate in tre tipologie: *RenderConstant*, *NumericalConstant* e *UserConstant*. Le prime due identificano parametri geometrici, di illuminazione e di materiale usati frequentemente negli effetti del motore, mentre l'ultima viene usata per i dati di ingresso definiti dal programmatore. Vengono quindi riempite le strutture dati apposite atte ad immagazzinare questi parametri in modo da renderli disponibili in fase di

caricamento. Quando sarà necessario applicare lo shader, questi dati saranno trasferiti alla scheda video, rendendo possibile l'esecuzione del programma. Questo approccio presenta tuttavia lo svantaggio che il programmatore difficilmente si accorge di eventuali errori fino a che l'applicazione non viene lanciata; inoltre è praticamente impossibile il debug degli shader, cosa che causa notevole perdita di tempo nella ricerca degli errori, come abbiamo purtroppo verificato più volte a nostre spese.

Inoltre abbiamo verificato la portabilità del linguaggio Cg su schede di marca ATI: il funzionamento è corretto in linea di massima, anche se si perde un po' in fatto di qualità.

A livello di sintassi, il Cg è molto simile al linguaggio C. Presenta tuttavia alcune modifiche atte a rendere più veloce il processamento dei dati, soprattutto nei tipi di variabili e funzioni riguardanti i vettori. Ad esempio supporta lo *swizzle* per i vettori, ossia la possibilità di identificare il primo parametro del vettore con la dicitura *vettore.x* senza utilizzare un indice apposito, nonché di agire con una stessa istruzione su più componenti del vettore (es. vettore.xyz = 3 assegna il valore 3 sia ad x che a y che a z).

3.6 Problematiche del Cg

Grazie al continuo svilippo della NVIDIA, nel tempo ci sono state varie evoluzioni dei compilatori Cg (chiamati *cgc*). Al momento dell'inzio delle nostre prove di scrittura degli shader, la più recente era la versione 1.5Beta del febbraio 2007. Tutti i primi shader da noi realizzati sono stati compilati con questa versione che inizialmente sembrava funzionare in maniera corretta. Tuttavia, al momento di realizzare gli shader per disegnare in una sola passata di rendering un oggetto su cui agivano contemporaneamente un effetto luce e un effetto texture [cfr. sezione 2.6.4], abbiamo riscontrato un grave baco che presenta questa versione del compilatore. Alcuni vertici della scena in esame apparivano bianchi a dispetto della colorazione loro assegnata; inoltre i numero di vertici bianchi aumentava o diminuiva ruotando la scena in funzione della luce. Dopo numerosi tentativi di isolare il problema per capire a cosa fosse dovuto (sul sito della NVIDIA non è emerso nulla e anche le ricerche in Internet sono state infruttuose), siamo riusciti a capire che l'origine dell'errore era da ricercarsi nella funzione matematica *pow*, che esegue l'elevamento a potenza di un numero, utilizzata all'interno dello shader per calcolare la componente speculare del materiale dell'oggetto che si sta disegnando. In particolare i calcolo specifico riguardava la definizione del parametro di *Shineness* del materiale, ovvero la *macchia* (spesso assimilabile ad un puntino) creata dalla luce sull'oggetto riflettente [cfr. figura 3.10]. Questo parametro si usa per definire quanto è luccicante un oggetto.

Non potendone cambiare l'implementazione, l'unica via che si è prospettata era emularne il funzionamento tramite l'utilizzo di una funzione lineare che andasse ad approssimarne la curva tramite la definizione di alcuni parametri. Dopo aver pensato a questa funzione, abbiamo cambiato conseguentemente il motore in maniera abbastanza pesante aggiungendo alcuni parametri nelle strutture dati interne, scrivendo nuove funzioni e soprattutto creando nuovi shader per il supporto delle modifiche eseguite. Queste modifiche hanno prodotto un discreto risultato di emulazione della funzione pow, apprezzabile anche a livello grafico con differenze pressoché impercettibili con l'originale.

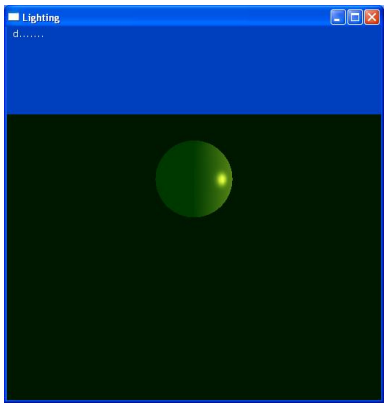


Figura 3.10. Un oggetto su cui è possibile apprezzare l'effetto del parametro *Shininess*

Lo stesso problema si è presentato per il parametro *Exponent* delle luci spot, anch'esso soggetto ad un elevamento a potenza. Anche in questo caso si è intervenuti sulle classi del motore, in particolare su quelle riguardanti la definizione della luce, impostando nuovi parametri per il calcolo dell'esponente, riscrivendo gli shader in maniera adeguata e raggiungendo un buon risultato finale.

Tuttavia, esattamente il giorno dopo la fine di queste modifiche, è stato notificato sul sito dell'autore del motore grafico questo baco del compilatore cgc 1.5Beta e suggerita come soluzione l'utilizzo del compilatore precedente, cgc versione 1.4. Dopo aver verificato che in effetti con il vecchio compilatore la funzione pow funzionava a dovere, abbiamo deciso di ripristinare tutte le vecchie impostazioni del motore e di compilare tutti gli shader passati e futuri del nostro progetto con la versione 1.4 di cgc, più affidabile della nuova. Rileviamo quindi come una versione più vecchia del compilatore funzionasse in maniera corretta, mentre quella più recente creasse problemi, cosa che sinceramente non ci saremmo mai immaginati.

Gli shader creati apposta per supportare queste modifiche sono comunque risultati utili come punto di partenza per gli altri interventi che abbiamo apportato al motore per gli effetti texture. In pratica li abbiamo riconvertiti al nuovo uso che avrebbero avuto [cfr. sezione 2.6.4]. In tal modo questo lavoro non è risultato del tutto inutile, anche se ci ha sottratto tempo prezioso.

A parte questo grosso problema che abbiamo per fortuna risolto, non abbiamo riscontrato altri grandi ostacoli a lavorare con il linguaggio Cg.

3.7 Problematiche degli shader

Un grosso limite che abbiamo rilevato nel lavoro con gli shader è la mancata segnalazione di errori qualora essi accadano. Se ad esempio viene effettuata una divisione per zero o un registro va in overflow all'interno del codice assembler, il risultato prodotto a video sarà il colore bianco. Esso può essere assunto solo da uno o più vertici della geometria se fallisce

il vertex shader (cfr. figura 3.11), mentre in caso di fallimento del pixel shader sarà tutta la figura ad apparire bianca.

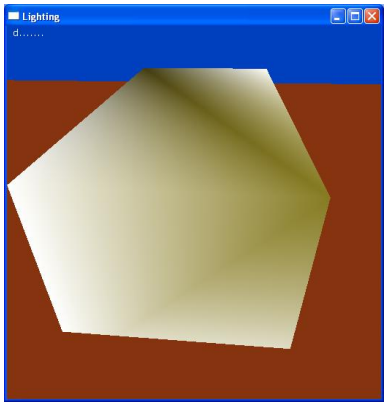
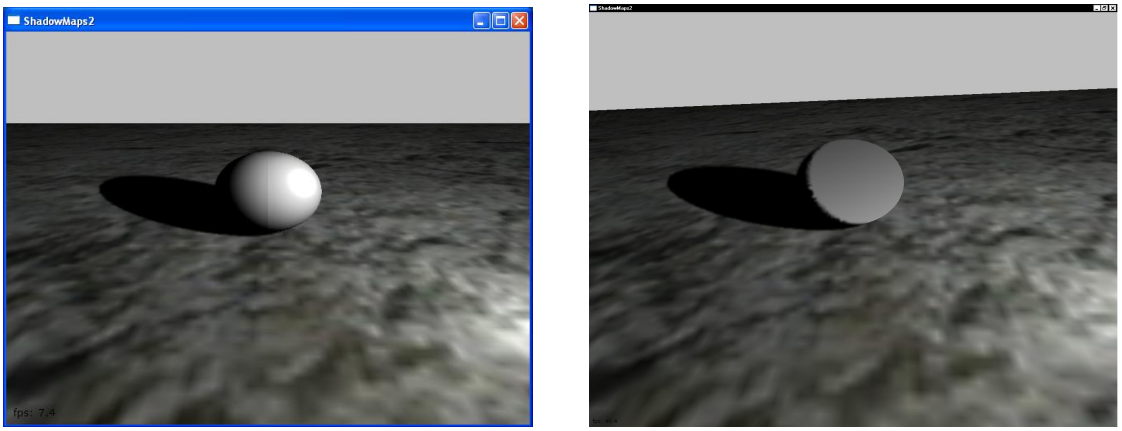


Figura 3.11. L'errore generato a video dagli shader compilati con cgc 1.5Beta

Un altro limite degli shader è costituito dal modo di valutare le espessioni condizionali. Come abbiamo già avuto modo di dire, a livello assembler i due o più rami di un'espressione *if-else* o di uno *switch*, vengono comunque eseguiti. Quest'approccio fa sì che venga sottratta potenza di calcolo in quanto si hanno alcuni registri occupati inutilmente; inoltre aumentano le possibilità che si verifichi un errore. Se ad esempio in uno dei due rami si verifica una divisione per zero o un overflow, come abbiamo già avuto modo di vedere, la figura apparirà bianca a video. Ciò accade nel caso l'espressione condizionale non risulti vera per il ramo di codice che ha generato l'errore.



(a) Scheda NVIDIA (b) Scheda ATI

Figura 3.12. Il differente comportamento delle ombre su schede NVIDIA e ATI

Inoltre ci sono diversi problemi di compatibilità tra schede grafiche di marca NVIDIA e schede grafiche di marca ATI. Durante il nostro lavoro, è capitato diverse volte che effetti o intere applicazioni che davano un buon risultato su di una scheda, ne fornissero un altro di scarsa qualità sull'altra. Ciò è dovuto alla diversa implementazione che hanno alcuni operatori come la divisione e alla diversa precisione nei registri floating point fornite per queste due schede.

La figura [cfr. figura 3.12] mostra la stessa applicazione che disegna le ombre, eseguita su una macchina dotata di scheda NVIDIA e una con scheda ATI. Si può apprezzare che su una scheda NVIDIA venga un'ombra molto più sfumata e l'effetto della luce speculare sia più evidente. D'altro canto abbiamo notato che in genere su schede ATI si ottengono prestazioni velocistiche migliori, a parità di caratteristiche hardware.

Capitolo 4

Ultraport

Ultraport è il primo sistema operativo per il 3D ed è basato interamente sulla tecnologia UltraPeg [cfr. sezione 1.2]. Per chiarire lo stretto rapporto che c'è tra UltraPort e UltraPeg, si potrebbe dire che se UltraPort fosse un *dvd*, UltraPeg sarebbe il laser che ne consente la lettura.

In questo capitolo viene analizzato UltraPort, presentando le sue strutture dati, il modo per importare in esso modelli 3D e per programmarlo.

4.1 Strutture dati

Nei prossimi paragrafi verranno illustrate le entità proprie di UltraPort, ovvero gli elementi su cui si basa che ne consentono il funzionamento. Come da abitudine di Ultramundum, i nomi di questi elementi sono in latino per sottolineare che questa tecnologia proviene dal Mediterraneo e dall'Italia, non dagli Stati Uniti o altre nazioni.

4.1.1 Le Tabulae

I dati in UltraPort comprendono una molteplicità di categorie, per ognuna delle quali sono necessarie specifiche tipologie. Tutti i dati dei mondi di Ultramundum sono contenuti in file chiamati *Tabulae*, di cui struttura ed evoluzione sono estremamente complesse.

La *Tabula* è l'elemento base di tutta la tecnologia UltraPeg. Deve essere vista come "un mattoncino" di un gioco delle costruzioni. All'interno di una tabula si può memorizzare un elemento tridimensionale (una porta, una colonna, un'automobile), un suono, un'animazione o ogni altro tipo di dati. In effetti, la tabula può essere vista come un record di un database binario (il *Tabularium*) presente sull'Ultraserver, che può essere scaricata da UltraPort dal database centrale di Ultramundum (il *Primum Tabularium*). Dopo lo scaricamento, ogni tabula viene memorizzata in un database sul disco del computer (il *Tabularium locale*) per usi futuri.

Una tabula è univocamente indicata dal proprio numero di serie, la cui dimensione in byte può crescere nel tempo onde accomodare un numero sempre crescente di Tabulae.

Ogni tabula può contenere tre porzioni: Interfaccia, Procedurale e Dati. L'Interfaccia contiene i dati necessari a definire come la Tabula si relaziona con il mondo esterno.

La porzione Procedurale, che può anche non essere presente per Tabulae di soli dati, contiene tutto il codice, scritto in linguaggio chiamato *Carmina C*, molto simile al C di base, che consente alla tabula di eseguire materialmente le proprie operazioni.

La parte Dati contiene tutte le informazioni che sono necessarie alla Tabula. Un'altra caratteristica fondamentale delle tabulae è la gerarchia: è consentito collegare una tabula con altre contenenti elementi "inferiori". In tal modo, una tabula che definisce un certo tipo di automobile può referenziare una tabula per le ruote. La stessa tabula viene referenziata da tutte le altre che contengono ruote al loro interno [cfr. figura 4.1].

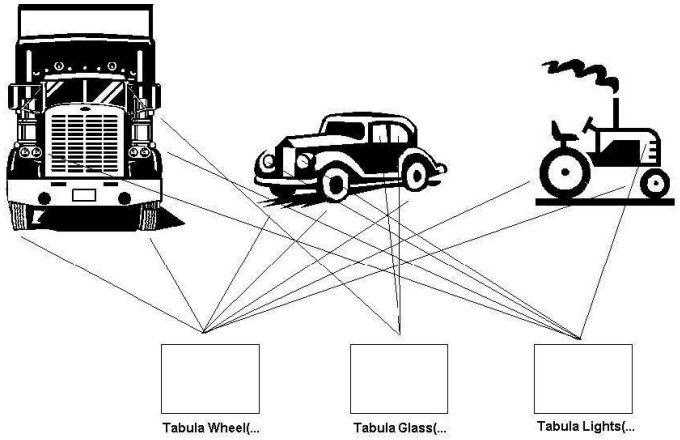


Figura 4.1. Viene illustrata la possibilità di riutilizzo delle tabulae all'interno di altre che descrivono oggetti più complessi. In questo semplice esempio la tabula *ruota* (wheel) viene richiamata nella tabula *camion*, in quella *auto* e in quella *trattore*

Le tabulae sono astrazioni concettuali e inglobano archetipi di intere categorie di entità, potendo esprimere ogni singolo elemento specifico della categoria di cui si occupano mediante l'opportuna impostazione dei parametri passati all'Interfaccia.

Ad esempio, la tabula Ferrari 550 consente di creare tutte le possibili copie di questa specifica automobile indicando all'interfaccia parametri come Colore, Tipo di cerchioni, Stato di usura e così via. Grande attenzione deve essere posta nella stesura delle tabulae in modo da catturare al più alto livello possibile di astrazione la categoria che si va a definire.

4.1.2 Le Cellulae

Una *Cellula* può essere definita come una porzione di spazio-tempo che contiene un'entità. In particolare ogni cellula rappresenta:

- un sistema di riferimento quadri-dimensionale, nel quale le informazioni che lo definiscono sono:

  - un centro (3 coordinate float);
  - un raggio (utile a mantenere l'insieme di cellule figlie) (1 float);
  - un orientamento completo a 3 assi (3 angoli float);
  - coordinate in metri;

- un modo per mantenere le equazioni del moto quadri-dimensionale (es. terra-sole) delle cellulae figlie;
- un servizio attraverso cui poter ottenere informazioni su: quali periodi in cui si differenzia una certa cellula (es. cellula-terra =¿ ...terziario, quaternario....), in quali luoghi;
- un involucro per tutte le tabulae che contiene lo stato della/e tabulae in essa contenute;

In particolare, all'interno delle cellulae è presente un processo di calcolo della posizione (in metri) dell'*ultranauta*, ovvero sia l'utente che utilizza Ultraport per l'esplorazione di mondi virtuali, espresso in coordinate della cellula root. Infatti, grazie al fatto di aver memorizzato la posizione dell'utente in formato concettuale, viene percorso il path di cellulae sino ad arrivare all'utente. Dopodichè, sapendo la posizione dell'utente rispetto alla cellula root, si procede con la generazione vera e propria.

4.1.3 Architettura di Ultraport

UltraPort identifica la parte *Client* del sistema complessivo client-server che consente ad un ultranauta l'esplorazione in ultravisione dei canali di Ultramundum.

In tale accezione, UltraPort può essere costituito, nel caso minimo, da un singolo software che viene eseguito su di un computer (con sistema operativo Windows, Linux, Mac OSX, eccetera...) o da un opportuno decoder connesso ad un televisore (Xbox, PlayStation, eccetera...). In questo caso si parla di UltraPort monolitico.

Le varie funzionalità necessarie ad UltraPort sono suddivise in sottosistemi.
Loschema dell'architettura di Ultraport è visibile in figura [cfr. figura 4.2].

4.1.4 Nucleus

L'insieme delle tabulae in essere su di uno specifico UltraPort prende il nome di *Nucleus*. Nel Nucleus avviene la vera e propria elaborazione dell'evoluzione del mondo digitale in cui è immerso l'ultranauta.

Si compone essenzialmente di una struttura dati programmabile rappresentata dalle tabulae, di una tabula *Avatar* rappresentante l'ultranauta nel mondo digitale, di tabulae tunnel per l'ingresso/uscita dati da/per il Nucleus e dalla *Psaltria*, una *virtual machine* utile a gestire/interpretare in modo completo le tabulae (compresa l'esecuzione della parte programmabile delle stesse).

Le tabulae debbono comunicare intensamente tra di loro onde consentire l'evoluzione voluta del mondo digitale. Le comunicazioni tra tabulae e tra esse e gli elementi del Nucleus

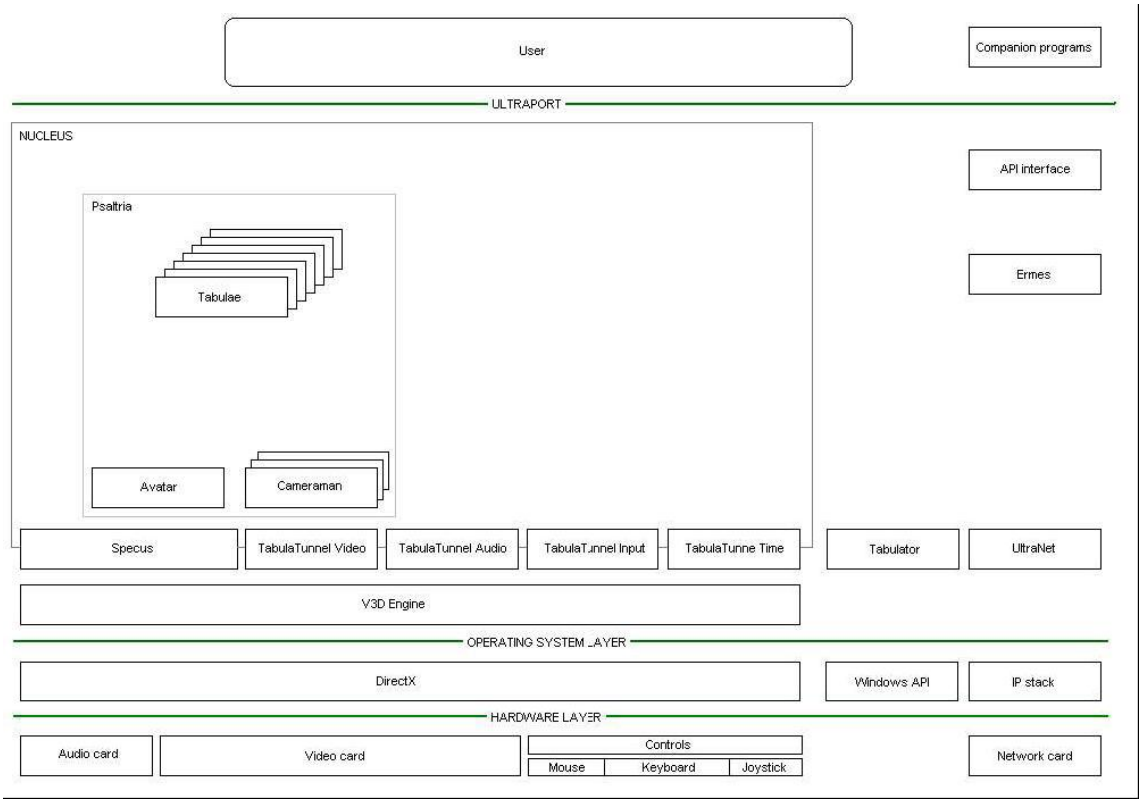


Figura 4.2. Schema dell'architettura di Ultraport

sono governate dal sottosistema *Ermes*, consentendo al Nucleus di essere distribuito su più computer. Fino al momento in cui non avviene uno scambio di messaggi, il Nucleus è in quiete e le tabulae non sono in esecuzione. I sottosistemi di output sensoriale continuano a rigenerare l'immagine del mondo per l'utente (nel caso siano ottimizzati, solo quando vi siano variazioni), ma nulla avviene a video.

Qualunque movimento dell'utente, o anche il semplice passaggio di un intervallo di tempo, produce la generazione di messaggi che attivano le operazioni del Nucleus per far evolvere il mondo digitale in modo corretto.

I comportamenti delle tabulae possono essere definiti in termini di reazioni a messaggi (meccanismo *Event-Action*). Ogni tabula non è attiva finché non riceve un messaggio da un'altra tabula o dal Nucleus, dopodichè va in esecuzione per elaborare una risposta allo stesso. Questo può portare alla generazione di altri messaggi e così via.

4.1.5 Specus

Lo *Specus*, nome derivato dal latino (significa *caverna*) a citazione di Platone, è un modulo istanziato su tutti i computer che necessitano di un qualsiasi riferimento allo *SceneGraph*, il grafo dei nodi che devono essere renderizzati.

Ogni elemento della scena è rappresentato da tre nodi. Il primo descrive la *mesh*, ossia

una struttura dati che contiene i dettagli della mesh stessa (come il numero dei vertici, il tipo dei vertici...). A questo nodo, ne vengono attaccati altri due: il primo è il nodo che rappresenta il materiale relativo alla mesh, il secondo definisce le primitive utilizzate, ossia i triangoli veri e propri, che saranno poi renderizzati nella scena. La costruzione dei nodi dello SceneGraph rispetta questa sequenza di passi. Il grafo dei nodi dello Specus ha origine nel nodo radice e si sviluppa attaccando i nodi successivi. Inoltre i nodi possono anche essere tolti dal grafo e distrutti per ottimizzare le risorse della macchina nella descrizione dell'ambiente.

Lo Specus ha un'interfaccia pubblica, accessibile mediante messaggi di Ermes, e un'interfaccia diretta, accessibile mediante chiamata ai suoi metodi. La scrittura e modifica dello SceneGraph può avvenire sia tramite chiamata diretta ai metodi che tramite messaggi.

Ecco i principali messaggi che vengono inviati allo Specus per la gestione dei nodi:

- UM_EMESS_TYPE_CREATESGNODE: crea un nodo;
- UM_EMESS_TYPE_SETSGNODE: imposta la descrizione di un nodo creato;
- UM_EMESS_TYPE_INITIALIZESGNODE: inizializza un nodo precedentemente creato e generare così tutte le variabili interne a partire dalla descrizione impostata;
- UM_EMESS_TYPE_ATTACHSGNODETOSGNODE: "attacca" il nodo creato ad un altro esistente per creare la struttura ad albero;

Altre funzioni inseriscono e cancellano nodi, liberando così lo SceneGraph.

Per il nostro progetto, lo Specus è stato modificato in maniera abbastanza consistente a causa del fatto che esso funge anche da interfaccia col motore grafico Wild Magic [cfr. capitolo 2]. Era quindi necessario fare in modo che lo Specus si basasse sulla versione del motore da noi usata e modificata per la gesione delle ombre, ovvero la 4.6.

Sostanzialmente il lavoro svolto è stato di aggiornamento. Ciò ha riguardato in particolare alcune funzioni che nel passaggio dalla vecchia versione 3.11 alla nuova 4.6 hanno modificato la loro implementazione o non esistevano, come ad esempio le modalità di applicazione delle texture di cui si parla nella sezione sulle modifiche al motore grafico [cfr. sezione 2.6]. Di riflesso, questi cambiamenti hanno richiesto l'aggiornamento di altre procedure di Specus e di UltraPort in genere che si basano sul motore, come ad esempio il rendering off-screen, il *picking* degli oggetti o il rendering di figure semitrasparenti. Inoltre è stato necessario in taluni casi, per supportare funzioni di Wild Magic che in precedenza non esistevano, l'inserimento di nuove strutture dati o di nuovi flags.

Nel compiere questo lavoro, si è svolta anche un'opera di riorganizzazione e ordinamento di alcune parti del motore. Ad esempio è stato uniformato lo spazio dei nomi secondo le specifiche Ultramundum (non tutte le parti di Specus e di UltraPort erano conformi ad esse) e le librerie di Wild Magic necessarie al funzionamento di Ultraport sono state racchiuse in un unico header file.

4.1.6 Sequenza di rendering

Il Nucleus è istanziato in una singola classe, fisicamente presente su un solo computer. La sequenza di rendering avviene sulla base di una cadenza temporale definente il target di qualità che si vuole ottenere. Ad esempio, cinquanta fotogrammi al secondo è una buona qualità di rendering. Una base tempi per il rendering è contenuta all'interno di ogni Specus ed avvia in parallelo due processi: la creazione del nuovo stato e il rendering dello stato precedente. Se uno dei due processi non è ancora terminato nell'istante di avvio del nuovo periodo, lo Specus conteggia il tempo necessario al termine.

Indicatori specifici per i due processi sono inviati al Nucleus che può così porre in essere strategie di ottimizzazione sulla base di quanto tempo è stato risparmiato o di quanto ne è stato necessario in più rispetto alla cadenza temporale dell'obiettivo di qualità.

La gestione della base tempi è demandata al motore grafico, che gestisce direttamente le sequenze di transformazione e rendering in multithreading. Tali sequenze sono temporizzate in modo da lasciare tempo di CPU per le funzioni del sistema operativo e degli altri moduli di UltraPort. Modifiche a tali temporizzazioni sono demandate al Nucleus, che riceve statistiche aggiornate in real-time dal motore.

Ad ogni intervallo di tempo il motore grafico invia un messaggio interno allo Specus che lo invia con un messaggio Ermes locale all'avatar dell'utente. L'avatar calcola la propria nuova posizione e frame d'animazione, generando messaggi di Ermes per l'interfaccia di manipolazione dello SceneGraph dello Specus.

4.1.7 Generazione iniziale di una scena

All'apertura di un canale si rende necessario generare la prima volta la scena, per la quale si utilizza una tecnica a doppia passata di rendering.

Nella prima passata vengono svolte le seguenti azioni:

1. Il Nucleus richiama la prima cellula madre, con in mano la posizione dell'utente;

2. La cellula madre si attiva per stabilire quale fra le cellulae figlie andrà istanziata. Questo processo viene fatto in base a diversi parametri, tra cui dimensione, distanza dall'utente e coefficiente d'importanza;

3. Per ogni cellula figlia effettivamente istanziata si calcola un vettore di un numero variabile di elementi, rappresentanti i costi relativi alle risorse ritenute fondamentali per l'elaborazione, ad esempio numero di triangoli, dimensione texture, numero di istruzioni di linguaggio carmina eseguite, numero di canali audio, eccetera...);

4. Tale vettore deve essere retropropagato dalle cellulae figlie alle cellulae madri, in modo da garantire che ogni cellula madre possieda la somma dei costi indicati nei vettori delle cellulae figlie;

5. Si continua il processo ricorsivamente;

6. Alla fine ogni cellula madre saprà per ogni Ccllula figlia effettivamente istanziata, di quali risorse ha bisogno per generarsi rispetto alla posizione ed alla distanza dall'ultranauta;

7. Ad ogni tabula viene passata la distanza e un coefficiente di importanza definito dall'autore;

Nella seconda passata:

1. Il Nucleus, disponendo di un vettore di risorse massime spendibili, proveniente da un processo preliminare di analisi della configurazione hw/sw del computer utilizzato, procede con l'avvio di un processo ricorsivo di allocazione e consumo delle risorse usate, partendo dalla cellula root;

2. Ciascuna cellula procederà ricorsivamente alla distribuzione delle risorse ancora spendibili alle cellulae figlie istanziate, sino a raggiugere tutte le foglie della scena corrente;

3. Ciascuna cellula salverà dentro di sè le risorse che gli sono arrivate e che tocca distribuire alle cellulae figlie. Tale informazione sarà utilizzata nella fase di real-time;

4. Ciascuna cellula si genererà in base alle risorse che gli sono state effettivamente assegnate;

4.2 Esportare ed importare modelli in UltraPort

Il sistema UltraPort è concepito per generare le scene in maniera procedurale, a partire da file testuali detti *Tabulae*. A volte però è necessario importare modelli di edifici generati con pacchetti di modellazione come *3D Studio Max* o *AutoCad*; in questo modo si possono integrare opere realizzate in altri formati e convertirle in *Tabulae*, oppure inserirle direttamente nella scena. A questo scopo Ultramundum ha sviluppato un applicativo chiamato *Extractor*, che è in grado di importare modelli dai più comuni formati, come *VRML*, *3DS* e *LightWave*. Questo programma consente di caricare una scena da questi file e genererare un grafo in Wild Magic. A questa funzionalità di base si aggiungono altri strumenti di manipolazione della scena per eliminare eventuali imperfezioni, come ad esempio il ricalcolo delle normali, o la modifica delle impostazioni riguardanti la trasparenza.

L'output generato da *Extractor* è un file detto *Legacy Scene Graph*(LSG), ossia un grafo della scena ereditato da altri formati; questo file LSG deriva dalla serializzazione di una scena Wild Magic, ed il motore stesso ne supporta nativamente il caricamento.

Questo applicativo è stato realizzato utilizzando le librerie del motore grafico Wild Magic in versione 3, cioè la versione con la pipeline di rendering fissa. Naturalmente anche i file LSG generati sono compatibili soltanto con quella versione del motore.

Per utilizzare i modelli con il motore basato su shader è stato quindi necessario riprogettare il formato LSG, rendendolo compatibile con tutte le versioni di Wild Magic.

4.2.1 Struttura del formato

Esportare scene dai formati del pacchetti di modellazione non è semplice. Anche se spesso sono forniti i pacchetti di sviluppo (SDK), l'esportazione delle scene risulta spesso macchinosa e con molti errori. Per questo motivo abbiamo deciso di mantenere il tool *Extractor* basato sul motore grafico a pipeline fissa; invece di riscrivere tutte le classi che esportano dai formati dei modellatori alla scena di Wild Magic, abbiamo concepito un nuovo formato che esportasse una scena a partire dall'interpretazione di *Extractor* dei modelli. Grazie a questo approccio abbiamo potuto limitare le informazioni esportate alle sole realmente utilizzate in UltraPort: vertici, normali, colori, materiali e texture.

Onde limitare l'occupazione in memoria dei file abbiamo deciso di codificare le informazioni direttamente in binario, e per semplificare il trasporto dei dati abbiamo incluso in questo file anche le texture.

La struttura concettuale di un file LSG ricalca abbastanza fedelmente quella di un grafo della scena ed ogni informazione rilevante è stata codificata secondo particolari enumerazioni. Riportiamo, come esempio, il formato in cui è codificato un nodo [fig. 4.3].

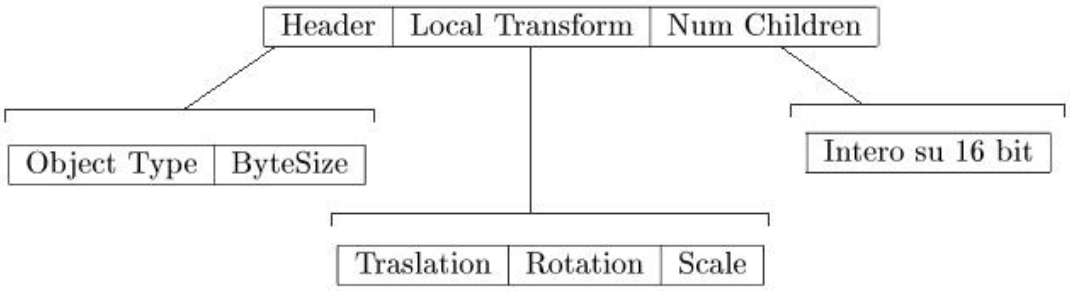


Figura 4.3. La struttura di un nodo del formato LSG.

Abbiamo già detto che le texture presenti nella scena vengono allegate al file durante la serializzazione. Quando si generano scene complesse, per ottimizzare le prestazioni e favorire l'esecuzione in real-time, si cerca di ottimizzare il più possibile l'uso delle texture, limitandone il numero e cercando di riutilizzarle il più possibile. Questo in termini di implementazione nel motore si traduce con la condivisione dei puntatori ad oggetti *Texture*, o *Image*.

Nella serializzazione questo approccio è stato mantenuto; abbiamo infatti realizzato un catalogo di immagini e texture evitando di memorizzare più volte le immagini duplicate.

4.2.2 Realizzazione

Le classi che abbiamo realizzato per supportare l'esportazione e l'importazione dal formato LSG cercano di essere il più generali ed estendibili possibile, sfruttando al massimo il

polimorfismo e l'ereditarietà.

La definizione delle strutture dati è contenuta nella classe *UM_LSG*: qui definiamo i tipi e tutte le enumerazioni necessarie; qui si definisco anche i metodi fondamentali che tutti gli esportatori di questo formato devono implementare: il caricamento da file o da un puntatore in memoria, ed il salvataggio su file.

Per l'accesso in input e output abbiamo definito una classe apposita, chiamata *UM_LSG_IO*: è una classe che utilizza metodi statici e nasconde alle classi i dettagli di accesso e controllo sulle zone di memoria, siano esse disco o memoria di sistema.

Da *UM_LSG* abbiamo poi derivato una classe che definisce l'esportazione ed importazione da e verso un generico motore Wild Magic: questo è utile perchè il motore grafico, pur evolvendosi, mantiene sempre una certa struttura concettuale (ad esempio il grafo della scena); con questa classe, chiamata *UM_LSG_WM*, raggruppiamo gli elementi comuni a tutte le versioni del motore.

Infine, da quest'ultima classe abbiamo derivato due classi: *UM_LSG_WM3* e *UM_LSG_WM4*. E' qui che si specificano i dettagli propri di ogni versione riguardo la creazione della scena o la collocazione delle informazioni di interesse. Con questo approccio risulta più semplice adattare questo insieme di classi alle successive versioni di Wild Magic, o addirittura ad un altro motore grafico.

4.3 Programmazione

Nei prossimi paragrafi verranno illustrate le specifiche del modo di programmare proprio della Fondazione Ultramundum per UltraPort. Si è infatti creato uno standard di programmazione che non dipendesse dal sistema operativo, dalle API o dal motore grafico, in modo che UltraPort fosse portabile al massimo ed eseguibile su ogni computer.

4.3.1 Convenzione sui nomi

Al fine di uniformare la stesura del codice da parte di più programmatori, è ovviamente necessario stabilire in UltraPort alcune semplici convenzioni sulla denominazione dei vari elementi base costituenti un programma; nella fattispecie, per la programmazione ad oggetti, classi, strutture, variabili, costanti e funzioni. La maggior parte delle convenzioni si basa essenzialmente sull'uso di un particolare prefisso per i nomi delle suddette entità. Ad esempio la lettera *m* indica che l'elemento è membro della classe che stiamo scrivendo, *p* identifica un puntatore, *b* un dato booleano e così via. Nei prossimi paragrafi si farà ulteriore chiarezza su questo concetto. Inoltre è buona regola cominciare con la lettera maiuscola ciascun nome o parte di esso.

Di seguito vengono forniti alcuni semplici esempi di quanto appena indicato:
m_dwResolutionX, m_dwResolutionY, g_bIsInitialized, pNode, eccetera.

4.3.2 Nomi di variabili membro e globali

Parallelamente ai prefissi usati per indicare l'area di validità di un particolare dato, è necessario utilizzarne uno per far riconoscere immediatamente il tipo di una qualsiasi

variabile o di una costante.

Tipo	Prefisso	Tipo Ridefinito
void	v	UM_void
bool	b	UM_bool
char	s8	UM_s8
unsigned char	u8	UM_u8
int	n	UM_int
unsigned int	un	UM_uint
float	f32	UM_f32
double	f64	UM_f64

Figura 4.4. Principali tipi di dati di UltraPort con il rispettivo prefisso

Nella tabella riportata [cfr. figura 4.4] sono inseriti i principali tipi di dati con il rispettivo prefisso e il nome del tipo ridefinito da utilizzarsi nei sorgenti Ultramundum in modo da evitare l'uso di nomi propri del linguaggio C che possono non avere implementazioni identiche sulle varie piattaforme.

4.3.3 Ulteriori prefissi di base

Parallelamente ai prefissi usati per indicare il tipo di una qualsiasi variabile o di una costante, occorre utilizzarne altri da anteporre ad ogni altra entità che si possa incontrare nei programmi.

Nella tabella riportata [cfr. figura 4.5] sono inseriti i principali tipi di entità con il rispettivo prefisso.

Tipo entità	Prefisso	Esempio
Nomi classi	C	Cstato
Nomi strutture	S	SWindData
Membro di una classe	m_	M_nElemento
Nomi di funzioni membro di una classe	UM_	UM_Salta(3)
Variabile globale	g_	g_nValore
Variabile statica	s_	s_nXCoord
Puntatore	p	ps32Counter
Puntatore a funzione	pfn	PftCalcola()
Puntatore FILE	pf	FILE* pfSource
Stringa Unicode con terminatore /o	sz	szNome

Figura 4.5. Principali tipi di entità di UltraPort con rispettivo prefisso

4.3.4 Il codice Carmina C

Il codice *Carmina C* è un linguaggio di alto livello, simile all'ANSI C 1989, nel quale vengono scritte le porzioni procedurali delle tabulae di UltraPort per essere poi compilate in Carmina.

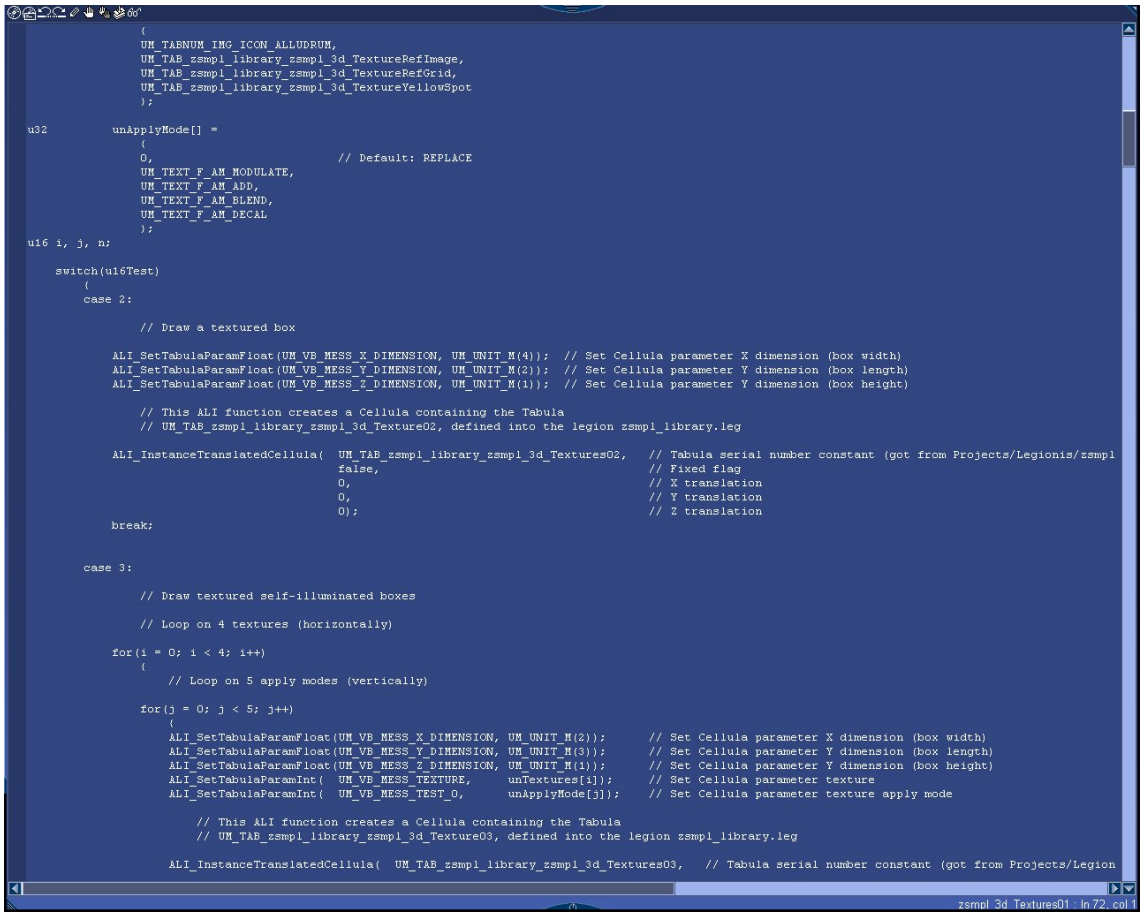


Figura 4.6. Un esempio di applicazione di UltraPort scritta in codice Carmina C

Qualsiasi programma Carmina C, in qualunque versione, è costituito da una serie di sezioni che possono essere presenti in tutto o in parte. Le sezioni sono le stesse di un normale programma C, ma con alcune varianti:

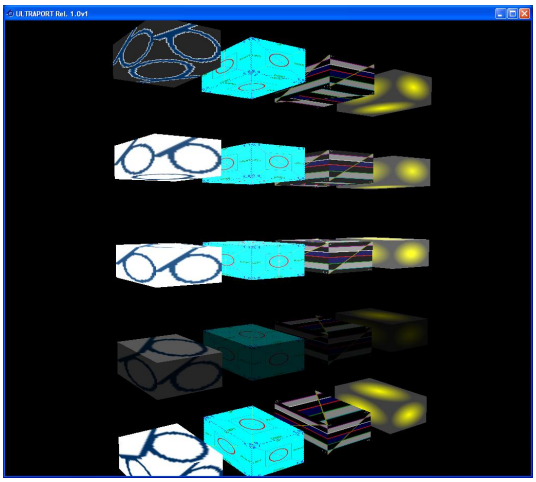
1. **Commenti iniziali**, dove l'autore illustra lo scopo della tabula;
2. **Blocco di #include**, dove sono inclusi file esterni necessari alla compilazione;
3. **Blocco di #define**, dove sono definite le costanti della tabula;
4. **Funzione iniziale**, detta costruttore della tabula (_new_), che contiene il codice eseguito quando si avvia la tabula;
5. **Funzioni ulteriori**, che contengono il codice di operazioni specifiche;

Ulteriori informazioni su questo linguaggio e su Ultraport in genere, nonchè alcuni esercizi per prendere confidenza con la tecnologia UltraPeg, sono disponibili in Internet sul sito della Fondazione[1].

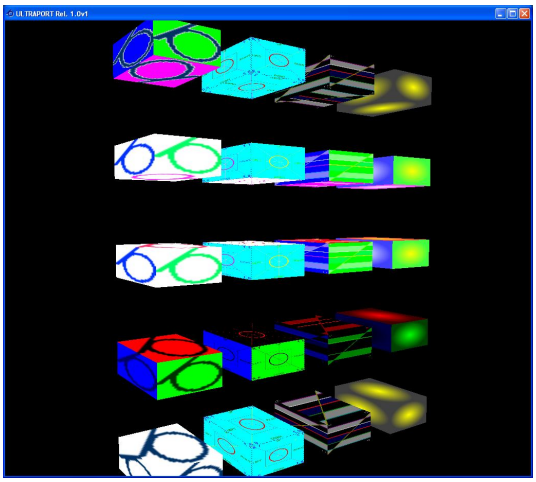
4.3.5 Applicazioni create con codice Carmina C

Per verificare in UltraPort l'effettivo funzionamento di tutti i cambiamenti apportati al motore [cfr. sezione 2.6], in particolare quelli relativi alle texture e al rendering, abbiamo fatto ricorso alla creazione di alcuni semplici esempi di codice Carmina C che creassero scene e situazioni che vanno ad interessare le suddette modifiche. E' stato perciò creato un apposito *Spatium*, ossia uno spazio di lavoro che contiene progetti di tabule, in cui abbiamo scritto alcuni file sorgenti che danno vita a diverse tabule. In queste tabule mostriamo esempi di applicazione delle diverse possibilità di utilizzo delle texture in seguito al nostro lavoro.

Abbiamo creato una tabula di base che richiama altre tabule, ad esempio quella che crea un parallelepipedo. In essa, cambiando semplicemente un parametro, siamo in grado di eseguire tutti i diversi esempi creati. Essi sono molto simili come funzionamento: viene creata una serie di parallelepipedi a cui sono assegnate differenti texture con tutte le diverse modalità di applicazione.



(a) Luce direzionale che illumina oggetti texturizzati



(b) Colorazione per vertice applicata su oggetti texturizzati

Figura 4.7.

In figura 4.7, ad esempio, sono illustrati due degli esempi sopracitati. Innanzitutto abbiamo individuato quattro texture con caratteristiche diverse: una con un canale alpha, una con i riferimenti allo spazio e quattro coordinate UV, una con una griglia di colori e

[1] consultare il sito http://www.ultramundum.it/italia/downupload/manuals/Enchiridion/index.htm

una con una "macchia" di colore chiara sfumata. Per ciascuna di esse è stata creata una colonna formata da cinque parallelepipedi. Ogni riga mostra una differente modalità di applicazione della texture; partendo dal basso si hanno Replace (che rappresenta anche le texture di base sopracitate), Modulate, Hard Add, Soft Add e Decal [cfr. paragrafi 2.6.3]. L'immagine di sinistra concerne una scena illuminata da una luce direzionale, quella di destra raffigura delle scatole cui è stata applicata la colorazione per vertice e una texture.

Sulla falsariga degli esempi appena illustati, ne sono stati creati diversi altri:

- Applicazione di una singola texture;
- Applicazione di una singola texture su un solido illuminato;
- Applicazione di più texture su un solido illuminato;
- Applicazione di una singola texture su un solido auto-illuminato (colorato per vertice);
- Applicazione di più texture su un solido auto-illuminato;
- Applicazione di texture proiettate;
- Applicazione di texture proiettate su solidi;
- Applicazione di texture semitrasparenti [cfr. figura 4.8];
- Applicazione di una singola texture su solidi semitrasparenti;
- Applicazione di più texture su solidi semitrasparenti;
- Applicazione di una singola texture semitrasparente su solidi illuminati;
- Applicazione di più texture semitrasparenti su solidi illuminati;
- Applicazione di una singola texture su solidi illuminati formati da materiale semitrasparente;
- Applicazione di più texture su solidi illuminati formati da materiale semitrasparente (che riportiamo a titolo di ulteriore esempio in Figura 4.9);

Per gli esempi relativi al multitexturing, viene introdotta un'ulteriore serie di parallelepipedi che specifica i vari modi di combinare fra loro le texture.

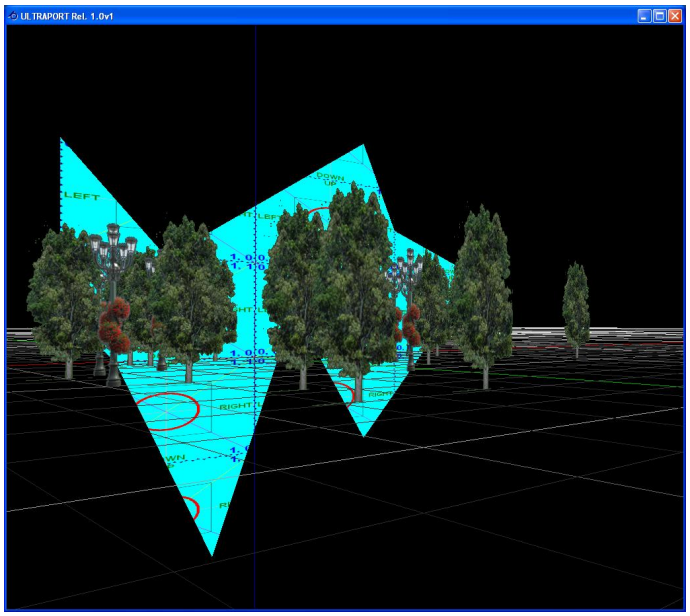


Figura 4.8. Esempio di texture semitrasparenti: lo sfondo retrostante alberi e lampioni rimane ugualmente visibile

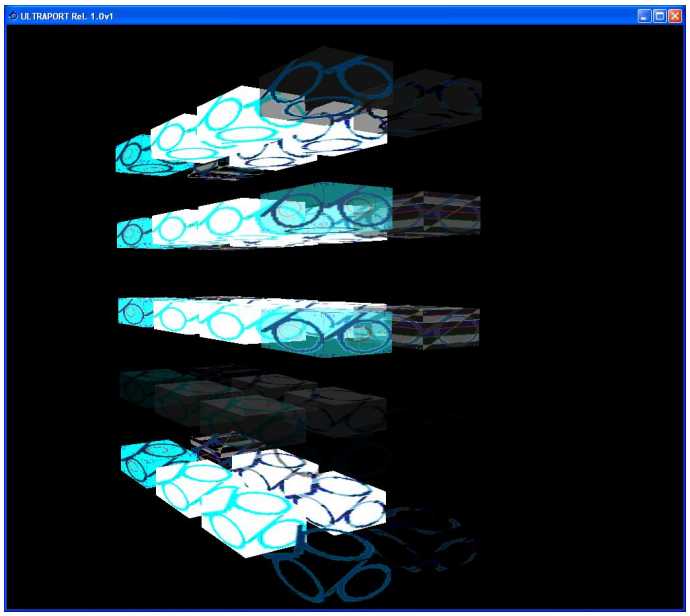


Figura 4.9. Applicazione di più texture su solidi illuminati formati da materiale semitrasparente

Capitolo 5

L'illuminazione ad armoniche sferiche

L'illuminazione ad armoniche sferiche, in inglese *spherical harmonics lighting* o più brevemente *SH Lighting*, è una tecnica che consente di simulare, in tempo reale, l'ombreggiatura di una sorgente luminosa di tipo direzionale, a bassa frequenza, posta a distanza infinita, su un insieme di oggetti statici. Fu introdotta nel 2002 in un articolo del Siggraph da Sloan, Kautz e Snyder come tecnica di illuminazione dei modelli ultra realistica; vedremo che ciò non è poi così vero, ma alcune peculiarità di questo algoritmo lo rendono particolarmente adatto per i nostri scopi.

La sua attuazione prevede due fasi: una precomputazione ed un calcolo in fase di generazione dell'immagine. La precomputazione consente di immagazzinare in forma compatta le caratteristiche luminose dei vertici della scena, grazie alla rappresentazione in armoniche sferiche. In fase di resa dell'immagine è sufficiente un semplice prodotto scalare tra i dati del vertice e la sorgente luminosa per ottenere il fattore di ombreggiatura. La rappresentazione compatta dei dati, insieme con la rapidità nel calcolo del fattore di ombreggiatura consentono di utilizzare potenzialmente un numero di sorgenti luminose molto elevato. Grazie al precalcolo abbiamo inoltre la possibilità di muovere a piacere la sorgente luminosa ricalcolando l'ombreggiatura della scena ad un costo computazionale molto basso. Questo rende l'illuminazione ad armoniche sferiche molto adatta ad un'applicazione che opera in tempo reale e che presenta scene all'aperto; vedremo però che per utilizzare questa tecnica all'interno di Wild Magic ed in particolare di UltraPort è stato necessario apportare alcune modifiche.

5.1 Descrizione del problema e funzionamento dell'algoritmo

Lo *SH Lighting* è davvero efficace se il contesto di utilizzo riguarda la resa di ambienti esterni con oggetti statici e luce solare; il sole si può infatti considerare una sorgente direzionale, a bassa frequenza e posta a distanza infinita. Per questo motivo lo consideriamo

un algoritmo adatto ad UltraPort ed ai suoi scenari di utilizzo, le città virtuali. Il principale problema è che UltraPort è un sistema dinamico in cui non è possibile effettuare alcuna operazione prima che la scena sia renderizzata, dato che le geometrie vengono generate in modo procedurale senza la possibilità di prevedere la distribuzione spaziale degli oggetti. E' quindi necessario effettuare il precalcolo parallelamente alla resa della scena.

Ciò che finora abbiamo definito precomputazione è di fatto una approssimazione dell'equazione della luce realizzata tramite una integrazione numerica; in altre parole, quello che si fa è un *raytracing* della scena immagazzinando i dati sotto forma di armoniche sferiche. La matematica che sta alla base di questo algoritmo è abbastanza complessa, per cui abbiamo deciso di concentrarci più sugli aspetti concettuali e rimandiamo ai riferimenti bibliografici per i dettagli e le formule.

5.1.1 La luce come emisfera

Per descrivere le proprietà luminose di una superficie in termini prettamente fisici, molto spesso si utilizzano funzioni chiamate BRDF (*Bidirectional reflectance Distribution Functions*). Questo tipo di rappresentazione consente di calcolare l'intensità luminosa uscente dalla superficie a qualsiasi angolazione ma è costosa in termini computazionali in quanto il suo calcolo prevede un integrale su una emisfera. Nella maggior parte delle tecniche di illuminazione in tempo reale, si usa un approccio più *fenomenologico* e meno fisico; si cerca quindi di produrre gli effetti desiderati senza rispettare i vincoli fisici imposti dalle equazioni.

L'idea dell'emifera di luce è solo una base concettuale, non è possibile calcolare tale integrale in tempo reale, con le tecnologie attuali. Con questo approccio si velocizzano i calcoli a scapito del realismo, perché non si riescono a simulare le luci ad area e le riflessioni di oggetti su oggetti. Consideriamo come esempio la figura 5.1. Per elaborare in tempo reale una scena del genere, di solito le quattro sorgenti luminose sarebbero considerate puntiformi. Questa approssimazione preserva le caratteristiche più essenziali della luce, ma ne trascura molte altre. L'idea è trovare una via di mezzo tra il raytracing e il modello a luce puntiforme. Grazie a SH possiamo avere una rappresentazione delle informazioni molto compatta in termini di occupazione di memoria e molto facile da manipolare. La compressione comporta ovviamente una perdita di dati, in particolare vengono trascurate le componenti della luce ad alta frequenza; questa perdita non è così grave quando si vuole simulare una scena esterna illuminata da luce solare.

5.1.2 La teoria delle armoniche sferiche

In questa sezione forniremo alcune nozioni per comprendere la matematica che sta alla base delle SH.

In matematica, le armoniche sferiche sono un insieme ortogonale di soluzioni dell'equazione di Laplace in coordinate sferiche. In altre parole, sono una base ortonormale per ricostruire qualunque funzione. Come la trasformata di Fourier opera sulla circonferenza unitaria, così le SH operano sulla sfera unitaria. Il termine "base" indica che questo tipo

Figura 5.1. Una scena illuminata da quattro luci ad area

di funzioni possono essere scalate e combinate per approssimare ogni altra funzione matematica; più basi si usano, meglio si approssima la funzione. I fattori usati per ricombinare le basi sono noti come Coefficienti.

Nella loro forma più generale le SH sono definite usando numeri complessi ma nel nostro contesto, in cui vogliamo approssimare funzioni reali, si utilizzala forma reale:

$$y_l^m(\theta,\phi) = \begin{cases} \sqrt{2}\Re(Y_l^m), m > 0 \\ \sqrt{2}\Im(Y_l^m), m < 0 \\ Y_l^0, m = 0 \end{cases}$$

Tra i termini di questa equazione sono presenti i fattoriali doppi, definiti come $x!! = x(x-2)(x-4)(x-6)...$ ed i Polinomi di Legendre associati, che sono particolari funzioni caratterizzate dai parametri l e m (in generale numeri complessi) definiti rispettivamente grado e ordine della funzione di Legendre associata.

$$\tilde{f}(s) = \sum_{i=0}^{n^2} c_i y_i(s) \; dove \; i = l(l+1) + m$$

Figura 5.2. Una funzione $f(s)$ approssimata usando armoniche sferiche di ordine n.

Per ulteriori dettagli sui Polinomi di Legendre, rimandiamo a [5].Da una definizione più compatta della formula deduciamo che l'approssimazione di una funzione in SH di ordine n richiede dunque n^2 coefficienti, il cui valore è dato dal risultato di un integrale.Questo viene ottenuto tramite una stima utilizzando l'integrazione nota come MonteCarlo[1]; questa tecnica si basta sulla probabilità e consente di calcolare i coefficienti tramite una sommatoria di prodotti. Per fare questo è però necessario effettuare un campionamento opportuno della emisfera attorno al punto suddividendo l'emisfera in una griglia regolare e poi si prendono campioni casuali all'interno delle griglie. Questa tecnica è nota come *Jittering* ed è usata, ad esempio, in alcune tecniche di *anti-aliasing* nel campo dell'elaborazione dei segnali. La sua peculiarità è che in campionando in questo modo si diminuisce la varianza complessiva del campionamento: si dimostra che la somma delle varianze per ogni cella non sarà mai più alta della varianza di campioni casuali presi su tutta l'area, spesso è anzi molto più bassa.

Riassumendo, tramite le SH e l'integrazione MonteCarlo possiamo approssimare l'equazione della luce; attraverso dei campioni dell'emisfera di luce e l'integrazione numerica otteniamo i coefficienti.

La rotazione dei coefficienti Una proprietà molto importante della proiezione di funzioni in coefficienti ad armoniche sferiche è che questa diviene invariante alla rotazione. In altri termini, ruotare una funzione e proiettarla in uno spazio SH è equivalente a proiettare e poi ruotare. Grazie a questa proprietà possiamo ruotare la sorgente di luce intorno agli oggetti senza dover proiettare la sorgente ruotata. Ci basteranno i coefficienti di una luce con un qualche orientamento predefinito e applicare una matrice di rotazione per avere rappresentare la luce in una nuova posizione.

5.1.3 Le armoniche sferiche nel calcolo dell'illuminazione

Nell'informatica grafica una singola equazione può descrivere la completa distribuzione della luce in una scena: l'equazione di rendering. Questa formula sta alla base di qualunque algoritmo di illuminazione globale che sia basato sul comportamento fisico della luce. La sua formulazione più comune è la seguente:

$$L_o(x,\overrightarrow{w}) = L_e(x,\overrightarrow{w}) + \int_s f_r(x,x' \rightarrow x,\overrightarrow{w}) L_i(x' \rightarrow x) V(x,x') G(x,x') dA'$$

Questa equazione dice che la radianza in uscita da un punto x su una superficie nella direzione ω è la somma della radianza emessa e della radianza riflessa da questo punto in

[1]Per ulteriori dettagli, Cfr. [6].

questa direzione. La radianza riflessa può essere espressa come l'integrale sull'emisfera delle direzioni che colpiscono il punto x del prodotto della BRDF e della radiazione incidente[2]. Come si può notare, l'intensità di luce che emessa da un punto può essere espressa come un integrale. L'idea alla base dell'illuminazione a SH è calcolare questo integrale in fase di precomputazione e proiettare le diverse funzioni sulle basi.

In altre parole, il preprocessamento può essere suddiviso in due fasi:

- proiettare la sorgente luminosa sulle basi SH. La luce si può rappresentare sia numericamente, ad esempio usando come sorgente un campione HDR (High Dynamic-Range light probe) oppure simbolicamente tramite coefficienti;
- per ogni vertice della scena proiettare sulle basi SH il prodotto della BRDF, della funzione di visibilità e del termine geometrico. Possiamo definire ciò come una funzione di trasferimento che, ponendo all'ingresso la sorgente luminosa, fornisce in uscita il termine di ombreggiatura del vertice.

Alcune considerazioni sulla scena da rappresentare consentono di semplificare ulteriormente i calcoli. In particolare se consideriamo gli oggetti come diffusori Lambertiani, che cioè riflettono la luce in egual misura in tutte le direzioni, riduciamo la BRDF ad una costante; gli oggetti emissivi non sono tenuti in considerazione anche la radiance emessa si può trascurare. Il termine geometrico è costituito dal coseno tra la direzione della luce e la normale nel vertice. La parte più impegnativa dal punto di vista computazionale rimane il calcolo del termine di visibilità, poichè necessita di fatto di un raytracing della scena. La versione più rudimentale dell'algoritmo non ne tiene in considerazione, ma come possiamo vedere dal confronto in figura 5.3 la resa è decisamente più realistica se si tiene conto dell'occlusione degli oggetti.

5.1.4 Illuminazione diffusa e interriflessioni

Uno dei pregi dell'illuminazione ad armoniche sferiche è la flessibilità. E' possibile infatti approssimare l'equazione della luce a livelli di precisione sempre più raffinati. Di seguito mostreremo come va modificato l'algoritmo per tener conto delle riflessioni reciproche di oggetti su oggetti. In questa maniera non si tiene conto soltanto della luce irradiata dalla sorgente luminosa, ma si aggiunge il contributo della luce che ha rimbalzato intorno alla scena prima di raggiungere il vertice corrente. Aumentando il numero dei rimbalzi si va sempre più verso un calcolo simile alla radiosity. I passi dell'algoritmo si modificano nel seguente modo:

- nel primo passo si calcolano i cofficienti di luce diffusa;
- per tutti i vertici, si fanno partire i raggi finchè non si incontra il primo ostacolo;
- se si trova un'intersezione bisogna memorizzare i coefficienti dei vertici del triangolo colpito;

[2]Per approfondimenti Cf. [10]

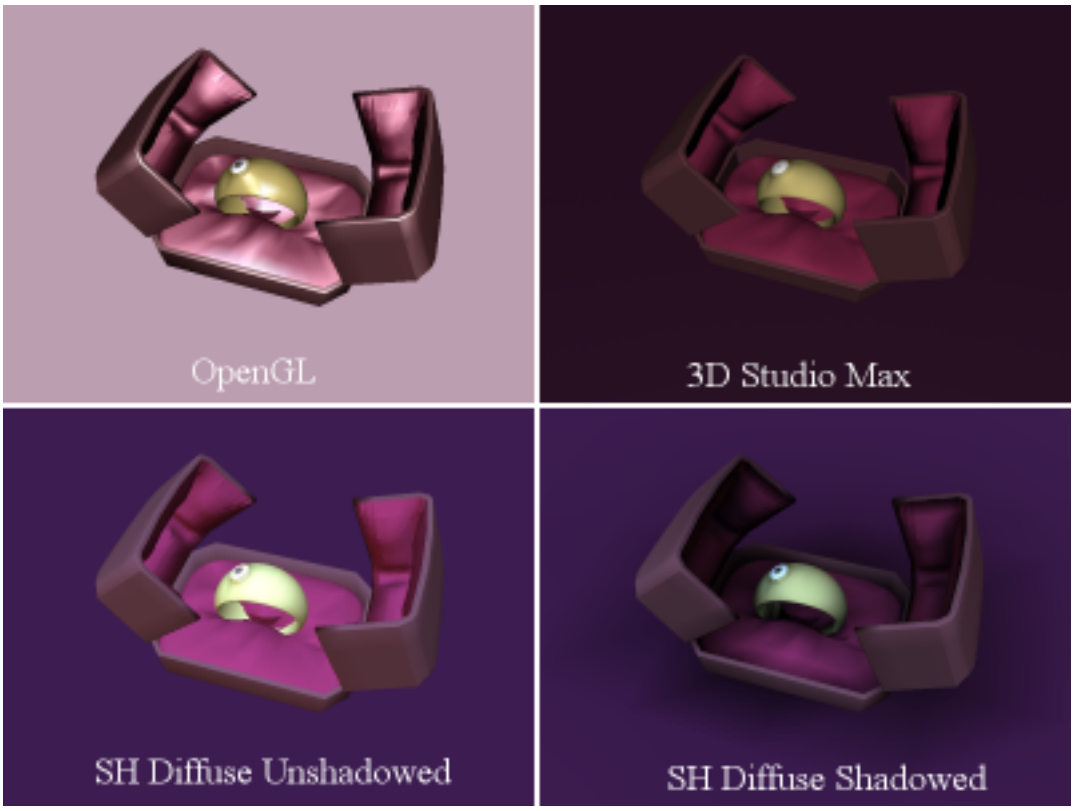


Figura 5.3. Confronto tra le diverse tecniche di illuminazione. Notare in particolare la differenza tra SH Unshadowed ed SH Shadowed: se il calcolo tiene conto delle occlusioni il realismo è decisamente maggiore.

- utilizzando il baricentro del triangolo, si combinano i coefficienti dei vertici per ottenere una nuova serie di coefficienti. Questa rappresenta il contributo di luce fornito da quel triangolo al vertice in esame;
- questo contributo viene ulteriormente moltiplicato per il prodotto scalare tra la luce e la normale del vertice in esame;
- i coefficienti così calcolati vengono poi scalati di altri fattori relativi al numero dei campioni ed allo stimatore MonteCarlo;
- questa procedura si può ripetere un numero di volte a piacere, sempre partendo dai nuovi coefficienti che si ottengono al rimbalzo precedente.

Al termine dei calcoli di interriflessioni otterremo i coefficienti di ogni vertice, che sono dati dalla somma della componente diretta più le componenti dovute a tutti i rimbalzi. Tutto questo comporta naturalmente calcoli più complessi il cui effetto è veramente apprezzabile soltanto in scene in cui le geometrie si prestano a questo tipo di simulazione; in particolare è necessario che le geometrie presentino un numero molto elevato di vertici per descrivere

le superfici. In generale il calcolo che tiene conto solamente dell'occlusione tra le geometrie si può considerare un ottimo compromesso tra realismo e complessità di calcolo.

5.2 Realizzazione

In questo capitolo descriveremo una possibile realizzazione dell'algoritmo di illuminazione ad armoniche sferiche che tiene conto dell'occlusione reciproca delle geometrie. Buona parte delle scelte implementative sono state dettate dalla struttura interna del motore grafico Wild Magic e dalle necessità della piattaforma UltraPort. In termini pratici, l'obiettivo di questa implementazione è stato l'integrazione del calcolo dell'ombreggiatura all'interno del motore grafico, evitando di snaturare la sua struttura interna e realizzando una implementazione che sfruttasse gli shader. La scelta di effettuare la procedura di precalcolo interamente in software ha consentito di mantenere il codice compatibile con tutte le piattaforme supportate da Wild Magic, al prezzo di una lentezza maggiore nell'elaborazione. I vincoli che imposti dalla piattaforma UltraPort, inoltre, imponevano che l'insieme di strutture dati utilizzate per l'imbreggiatura fosse il più possibile traspartente al sistema sovrastante ed ovviamente l'impossibilità di effettuare precomputazioni.

Ciò che abbiamo realizzato è un sistema di facile integrazione sia in Wild Magic che in UltraPort, che sfrutta al massimo le strutture dati del motore grafico. Nello sviluppo dell'algoritmo ci siamo fermati ad un basso ordine di armoniche sferiche, in particolare al quarto grado, e non abbiamo considerato le interriflessioni; ci limitiamo soltanto alle occlusioni di oggetti su oggetti. Questo è un buon compromesso tra resa visiva e tempi di elaborazione.

Per quanto riguarda le prestazioni, ciò che abbiamo fatto è individuare il collo di bottiglia dell'algoritmo e proporre una soluzione per velocizzare il raytracing, attraverso una semplice struttura di indicizzazione spaziale della scena. Quanto segue è una descrizione delle classi che abbiamo realizzato.

5.2.1 Le strutture dati

Gli elementi fondamentali che costituiscono il motore Wild Magic sono: il grafo della scena e gli effetti. Per questo motivo, dovendo scegliere delle strutture dati per memorizzare le informazioni sullo stato dell'algoritmo e sui coefficienti generati abbiamo scelto di estendere queste classi. Abbiamo dunque creato un derivato del nodo che chiameremo *Spherical Harmonic Lighter* ed un derivato dell'effetto, lo *Spherical Harmonic Effect*.

Il *Lighter* costituisce la radice della porzione del grafo della scena che si desidera ombreggiare con SH. Questo semplifica la creazione di una scena ombreggiata con SH: basterà creare un grafo normalmente e poi attaccarlo sotto al nostro nodo. In questa classe sono contenute tutti i parametri e le informazioni necessari per realizzare l'algoritmo.

I parametri essenziali sono: il numero di campioni utilizzato nell'integrazione numerica e l'ordine dello sviluppo in armoniche sferiche. Il primo dato determina quanto fitto sarà il campionamento dell'emisfera di luce: più campioni si considerano, più precisa sarà l'approssimazione dell'integrale, più realistica sarà l'ombreggiatura, maggiore il tempo di calcolo.

Il secondo determina la quantità di coefficienti che ricaveremo dal calcolo; come abbiamo già detto è il quadrato dell'ordine cui arrestiamo lo sviluppo in serie.

Tra le informazioni contenute in questa classe le più importanti sono: le luci ed i campioni. Naturalmente per rappresentare questi dati abbiamo creato delle classi apposite, chiamate *Sample* e *Spherical Harmonic Light* in cui memorizzare i coefficienti calcolati. Ogni *Sample* contiene un vettore direzione e le coordinate polari θ *e* ϕ della direzione, oltre che i coefficienti di tale raggio proiettati sulle basi SH. La luce invece mantiene un vettore di coefficienti base, che rappresenta una luce a coordinate polari (θ *e* ϕ) pari a zero e uno identico con la luce ruotata alle coordinate polari desiderate. Quest'ultimo è usato nel calcolo dell'ombreggiatura, mentre il primo serve in caso di spostamento della sorgente luminosa: in tal caso si parte dal primo vettore e con opportune moltiplicazioni matriciali si ottiene un nuovo vettore ruotato. In altre parole, si cambia la direzione della luce.

Gli oggetti ed i parametri descritti finora hanno validità generale per tutta la scena da illuminare; ciò che resta da illustrare è la struttura dati utilizzata per immagazzinare i coefficienti per ogni vertice. Tenendo sempre a mente la necessità di realizzare un sistema dinamico, in cui si può decidere di passare da uno stato di illuminazione ad armoniche sferiche ad uno tradizionale in qualunque momento, abbiamo scelto di associare i coefficienti ad ogni geometria tramite un effetto. Questo effetto ha associato lo shader che combina il calcolo dell'ombreggiatura con la colorazione sottostante dell'oggetto. In questo modo ogni geometria in maniera assolutamente indipendente dal resto della scena è in grado di calcolarsi la sua ombreggiatura; questa possibilità risulterà utile nel caso di un'implementazione del precalcolo in forma distribuita in parallelo su più processi.

5.2.2 Il funzionamento

Attraverso la descrizione dei metodi e l'ordine in cui questi sono chiamati sarà forse più chiaro il funzionamento dell'algoritmo nella sua realizzazione pratica. Per utilizzare queste classi all'interno di Wild Magic è sufficiente creare un grafo della scena ed attaccarlo come figlio di un'istanza di un nodo *Lighter*; fatto questo, basterà aggiungere almeno una luce ed effettuare una chiamata al metodo *SHShade*. In questi tre passi si nascondono tutti i complessi calcoli necessari a ricavare i coefficienti di luce e geometrie.

La prima cosa da fare è generare il vettore dei campioni secondo la tecnica del *Jittering*, ossia dividere uniformemente lo spazio da campionare (nel nostro caso un quadrato) e poi prendere dei valori casuali all'interno delle suddivisioni. Ogni valore casuale è poi convertito in coordinate polari per ottenere una direzione; questa è proiettata sulla base SH da cui ricaviamo i coefficienti.

Grazie ai campioni appena calcolati possiamo ricavare i coefficienti delle luci, anche se non ancora ruotati nelle direzioni desiderate.

Il passo successivo è ruotare le luci, attraverso moltiplicazioni matriciali. Questi calcoli possono essere ottimizzati sfruttando i set di istruzioni specifici forniti dalle diverse architetture, ma ciò limiterebbe la portabilità del codice.

A questi passi preparatori segue il nucleo della fase di precalcolo: la generazione dei coefficienti di ogni vertice. Per fare questo si effettua una visita in profondità del grafo

della scena interessato al calcolo, chiamando per ogni geometria trovata il metodo *GenerateDirectCoeff*. Per ogni vertice della mesh si effettua una scansione dei campioni: se il prodotto scalare tra la normale in quel vertice e il raggio-campione è maggiore di zero, e se non ci sono ostacoli tra il vertice e la luce, allora si sommerà il contributo di quel raggio ai coefficienti del vertice. Questo contributo è moltiplicato per il prodotto scalare calcolato in precedenza, per dare maggior peso ad alcuni raggi rispetto che ad altri.

Ora tutto è pronto per calcolare il fattore di ombreggiatura. Abbiamo deciso di calcolare questa componente in software, e inserire questo dato nel canale dei colori. Questa scelta deriva dalla assunzione che un oggetto illuminato non dovrebbe mai presentare una colorazione per vertice, ma soltanto materiali e/o textures, e dunque il canale dei colori nel vertex buffer è sempre libero. L'alternativa sarebbe stata passare i coefficienti di ogni vertice come parametri variabili allo shader e le luci come parametri costanti; dovendo però associare ogni parametro ad una caratterizzazione semantica[3]. avremmo occupato quattro canali di coordinate texture solo per passare i sedici coefficienti, appesantendo l'implementazione senza trarne un effettivo vantaggio, in quanto il valore di ombreggiatura non cambia finché non si sposta la sorgente luminosa. Calcolato il prodotto scalare, l'ultimo passo è analizzare gli effetti attaccati alla geometria e sostituirli con la rispettiva versione che tiene conto della componente SH. In questo modo si nasconde totalmente al motore grafico la modifica e la scena viene renderizzata come al solito; saranno gli shader a modulare (con una banale moltiplicazione nel nostro caso) l'ombreggiatura per vertice con la texture o il materiale proprio della geometria in esame.

5.2.3 Il ruolo degli shader

La scelta di inserire nel canale dei colori il parametro di ombreggiatura calcolato in software, semplifica moltissimo il codice degli shader per questa applicazione. In questa implementazione, che ha il fine di illustrare l'algoritmo, abbiamo scritto uno shader che combina lo *SH Lighting* con la texture. Volendo utilizzare questo calcolo in un contesto più ampio, data la semplicità dello shader, sarebbe una buona soluzione integrare il calcolo all'interno di altri effetti, in modo da limitare le passate di rendering.

Il vertex shader ha banalmente il ruolo di *passthrough* per le coordinate texture e per il colore; ciò significa che non modifica i dati in input, ma li trasferisce inalterati al pixel shader, cui ovviamente arriveranno interpolati sul triangolo interessato.

Il pixel shader riceve in ingresso il fattore di ombreggiatura e la coordinata texture. Dopo aver campionato l'immagine combina tramite una moltiplicazione il colore ottenuto e il fattore di ombreggiatura.

5.2.4 Ottimizzazioni

Una delle critiche che si possono portare a questo algoritmo, è la lentezza della fase di precalcolo. Oltre al fatto che il numero di conti per ogni vertice varia con il quadrato del numero dei campioni, che sono necessari prodotti scalari e matriciali per le rotazioni della luce e che tutti i calcoli sono da farsi in virgola fissa data la precisione necessaria, il collo

[3] Cfr.3.3.3

di bottiglia dell'algoritmo è un altro. Come accennato in precedenza tutti i calcoli sopra citati si possono potenzialmente velocizzare sfruttando i set di istruzioni particolari dei processori, come ad esempio l'*MMX*, o il *SIMD*[4].

Ciò che di fatto rallenta pesantemente i calcoli è ciò che nel paragrafo precedente abbiamo liquidato con la frase "se non ci sono ostacoli tra il vertice e la luce". In un primo momento abbiamo utilizzato il sistema di intersezione tra un raggio e una geometria compreso nel motore Wild Magic noto come *Picker*; avendo fini prettamente didattici, questa implementazione non è per nulla ottimizzata, ed è risultata il collo di bottiglia dell'intero algoritmo. Il principale difetto di questo sistema è che effettua un attraversamento in profondità della scena, triangolo per triangolo, senza avere alcuna informazione spaziale a limitare il campo di ricerca.

Per velocizzare il raytracing abbiamo realizzato un sistema di indicizzazione dello spazio che consentisse una ricerca più rapida delle intersezioni tra raggi e geometrie; tutto questo in cambio di una maggiore occupazione di memoria del programma. In letteratura sono presenti moltissimi algoritmi e continuamente se ne propongono di nuovi. Non essendo questo il tema della nostra tesi, abbiamo realizzato il più semplice degli algoritmi: l'*Octree*[5]. Questo algoritmo è relativamente semplice da implementare e non richiede un eccessiva occupazione in memoria. L'idea che sta alla base di questa tecnica è di suddividere il volume occupato dalla scena in esame in un cubo allineato con gli assi. Questo cubo è poi suddiviso a metà lungo i tre assi, formato otto sotto-cubi. Ognuno di questi è ricorsivamente suddiviso fino a contenere un elemento base della scena: un triangolo. Con questa suddivisione, più fitta nelle parti di scena dove ci sono più geometrie, abbiamo la possibilità di escludere molto più rapidamente le geometrie che sicuramente non si trovano sulla traiettoria del nostro raggio. Per combinare velocità e precisione abbiamo scelto di realizzare la struttura utilizzando la matematica in virgola fissa a 64 bit, decidendo a priori il massimo livello di profondità in cui suddividere l'octree. Questa scelta è stata quasi obbligata, poiché le dimensioni dei triangoli non sono note a priori, e possono variare di molto da geometria a geometria. Con questo sistema ogni triangolo verrà ad occupare potenzialmente più di un sotto-cubo.

Non ci dilunghiamo in ulteriori dettagli su questo algoritmo, in quanto è stato utilizzato solamente per mostrare la necessità di un qualche sistema di indicizzazione dello spazio per poter effettuare il raytracing in tempi ragionevoli, con la consapevolezza che l'algoritmo dell'octree, se pur semplice, non è certamente la soluzione migliore.

5.3 Conclusioni

L'illuminazione ad armoniche sferiche è un algoritmo dalle basi matematiche abbastanza complesse, ma che consente di ottenere dei buoni livelli di realismo in alcuni casi paragonabili alla *Global Illumination*, senza appesantire l'esecuzione del rendering ad ogni frame.

[4]Quasi tutti i processori moderni offrono istruzioni particolari ottimizzate per eseguire calcoli frequenti nella computer grafica, come ad esempio moltiplicazioni tra matrici, o di vettori con matrici.

[5]Per la realizzazione ci siamo ispirati al template liberamente scaricabile all'indirizzo: http://www.hxa7241.org/articles/content/octree-general-cpp_hxa7241_2005.html

I suoi limiti più evidenti sono la risoluzione al vertice e la necessità del precalcolo. Questi difetti ne impediscono al momento l'utilizzo in UltraPort, per due motivi: primo, le mesh delle città virtuali di UltraPort sono generalmente ottimizzate per il rendering in tempo reale, e sono costituite da un numero molto basso di vertici, soprattutto i piani delle strade su cui si proietta il maggior numero di ombre; secondo, la fase di precalcolo deve essere eseguita parallelamente al rendering.

Nonostante ciò la semplicità del calcolo di ombreggiatura, la possibilità di simulare la presenza di molteplici sorgenti luminose e non ultimo il realismo con cui si calcola l'illuminazione secondo noi giustificano lo sforzo di ottimizzare l'implementazione al fine di integrarlo nel sistema UltraPort.

A questo proposito, le soluzioni che proponiamo vanno da un lato nella direzione di velocizzare il raytracing, con tecniche di indicizzazione più efficienti e più mirate a questo particolare contesto di utilizzo, come ad esempio il *Bounding Interval Hierarchy*[7], e dall'altro nel rendere il calcolo dei coefficienti parallelo al ciclo di rendering. L'idea di base è quella di procedere per raffinamenti successivi: in una prima iterazione si calcolano i coefficienti dei vertici tenendo conto solamente di alcuni campioni, quelli che hanno un'inclinazione simile a quella della luce. Abbiamo realizzato alcuni esperimenti in merito che hanno dimostrato come il contributo di questi pochi campioni sia sufficiente per generare delle ombreggiature, anche se imprecise. Questo consentirebbe agli utenti di avere immediatamente una scena illuminata con SH ma con ombre appena accennate; in seguito il processo che è eseguito in parallelo completerà il calcolo dei coefficienti, e le ombre andranno via via perfezionandosi durante l'esecuzione del programma.

Capitolo 6

Calcolo di ombre dinamiche

In questo capitolo descriveremo come è possibile realizzare il calcolo di ombre dinamiche in tempo reale; in particolare descriveremo le tecniche da utilizzare nel caso si voglia utilizzare il paradigma a shader. L'algoritmo di illuminazione ad armoniche sferiche, descritto nel capitolo 5, effettua un calcolo basato sulla equazione fisica della luce, ma si può applicare soltanto a geometrie statiche, perchè la lentezza del calcolo dei coefficienti ne impedisce l'aggiornamento ad ogni fotogramma. Per realizzare le ombre di una scena qualsiasi, in cui potenzialmente possono muoversi luce e geometrie, è necessario abbandonare l'ambizione di attenersi al mondo fisico. Ciò che si cerca di fare è simulare un effetto di ombreggiatura che risulti abbastanza credibile alla vista e che rispecchi i rapporti geometrici tra gli oggetti. Inserire le ombre in una scena, infatti, non porta solo un abbellimento estetico, favorisce anche una maggiore percezione della profondità. Questo calcolo deve essere

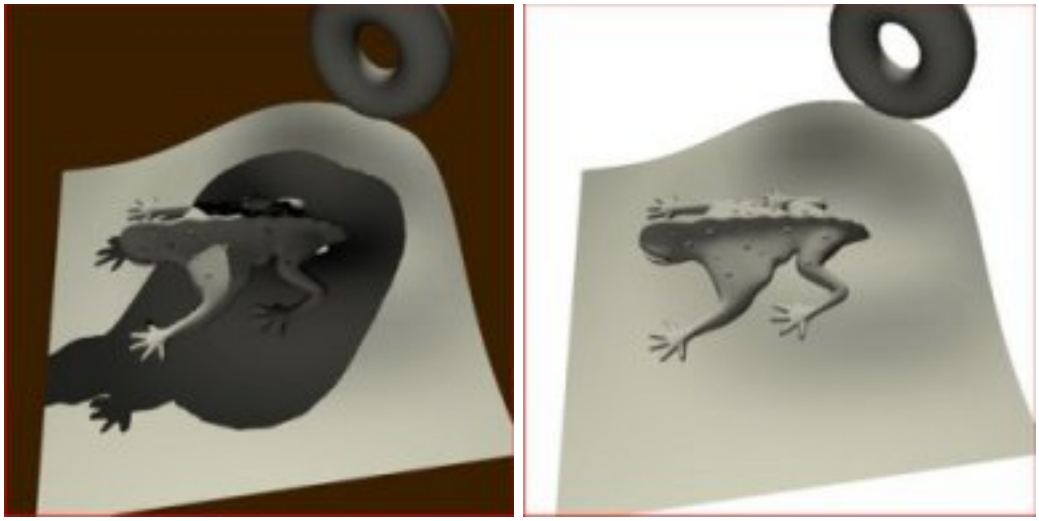


Figura 6.1. La presenza di ombre, oltre a conferire realismo ad una scena, aiuta a comprendere meglio le relazioni spaziali tra gli oggetti.

ripetuto per ogni fotogramma, perciò non deve essere troppo oneroso per non pregiudicare l'esecuzione in tempo reale. Naturalmente per ottenere effetti più realistici occorre una maggiore elaborazione; per questo motivo bisogna decidere il giusto bilanciamento tra prestazioni e grafica, tenendo conto della disponibilità di potenza di calcolo di cui disporrà l'utilizzatore medio cui il nostro prodotto è rivolto. Calcolare le ombre sfruttando la GPU sicuramente incrementa le prestazioni, ma comporta anche numerose insidie. Il calcolo di ombre dinamiche può ritenersi un ottimo banco di prova per le potenzialità degli shader, in particolare del linguaggio *CG* da noi utilizzato; i programmi che devono essere eseguiti sulle GPU hanno messo in luce alcune sostanziali differenze tra le schede video, che spesso producono risultati molto diversi a parità di shader.

Ciò che presentiamo è una analisi dettagliata dell'algoritmo che abbiamo ritenuto più adatto alla nostra piattaforma, lo *Shadow Mapping*, descrivendone la sua forma base e la nostra implementazione della stessa. Compreso il funzionamento di base sarà più semplice comprendere i difetti dell'algoritmo e le soluzioni che proponiamo.

6.1 Effettuare il render su una texture

Il concetto di *off-screen rendering* è fondamentale per comprendere e realizzare le ombre dinamiche. Senza il supporto per questo tipo di operazione, sarebbe stato impossibile implementare l'algoritmo in Wild Magic. Memorizzare il risultato di un ciclo di rendering su di un'immagine per poi riutilizzarlo è una tecnica usata in molti algoritmi per ottenere effetti speciali nell'informatica grafica; per questo motivo le schede video supportano nativamente questa funzionalità. Naturalmente abbiamo accesso alla memoria video ed alla GPU soltanto tramite le cosiddette *API Grafiche*, in particolare le librerie *OpenGL* e *DirectX*. Queste realizzano il render-to-texture in maniera differente, ma il motore Wild Magic, attraverso le sue strutture dati e le sue astrazioni, ci consente di non preoccuparci di questi dettagli. Le strutture dati necessarie per renderizzare su una texture sono: un oggetto *FrameBuffer* e un oggetto *Texture*, opportunamente marcato come *offscreen*.

Il *Frame Buffer* di una scheda video è quella porzione di memoria destinata a contenere l'immagine che generalmente viene mostrata sullo schermo. Contiene essenzialmente il colore di ciascun punto e la sua dimensione varia a seconda della risoluzione dello schermo (quanti pixel in larghezza e in altezza) e alla profondità del colore, cioè quanti bit si utilizzano per memorizzare quel dato. Al *Frame Buffer* si affiancano altre aree di memoria di supporto, tra cui lo *z-Buffer*, che risulta indispensabile per realizzare le mappe di profondità: è lo strumento che ci serve per stabilire se un oggetto è davanti o dietro all'altro rispetto alla posizione del punto di osservazione.

6.2 Shadow Mapping e Shadow Volumes

Esistono in letteratura due tipologie di algoritmi per realizzare le ombre dinamiche: lo *Shadow Mapping*, che analizziamo nel dettaglio e che riteniamo più adatto alle nostre applicazioni, e lo *Shadow Volumes*, una tecnica di tipo geometrico.

La tecnica degli *Shadow Volumes* richiede l'estrusione di una geometria nella direzione della luce, per generare un volume infinito. Dopo di che, attraverso il ray casting, le porzioni ombreggiate della scena possono essere determinate (di solito usando lo stencil buffer). Questa tecnica ha una risoluzione al pixel e non soffre di nessun problema di aliasing, ma come ogni tecnica presenta alcuni difetti. I due principali problemi sono: la complessità varia pesantemente a seconda della geometria della scena, inoltre è un algoritmo che intrinsecamente richiede numerosi calcoli. Ciò vincola l'applicazione ad un ristretto numero di poligoni per mantere l'esecuzione in tempo reale.

Per questi motivi abbiamo deciso di utilizzare lo *Shadow Mapping*, un algoritmo più semplice, ma che ha ugualmente risoluzione al pixel e richiede molto meno carico elaborativo. Esistono tuttavia diverse elaborazioni ed evoluzioni di quest'ultimo algoritmo. Molti programmatori hanno usato Shadow Map come base per le loro applicazioni, ma lo hanno adattatato alle loro esigenze, cercando di migliorarne rapidità di calcolo e qualità visiva delle immagini prodotte. Per il nostro progetto sono stati analizzati diversi algortmi di questo tipo, sia per prendere confidenza con il calcolo delle ombre, sia per verificare come altre persone abbiano risolto alcuni dei problemi ci si ponevano di fronte. Riportiamo di seguito i più significativi:

- *Reflective Shadow Maps*: con questo algoritmo si ottiene un'ottima resa visiva perchè tutte le superfici della scena sono riflettrici di luce diffusa; tuttavia richiede grande potenza di calcolo alle schede video anche per geometrie non troppo complesse [12];
- *Variance Shadow Maps*: questo algortimo consente di supportare diversi tipi di shadow maps di differente qualità, ottenendo una resa visiva ottima; tuttavia è di difficile compresione e richiede calcoli onerosi [1];
- *Fast Correct Soft Shadows for NVIDIA GPUs*: questo algoritmo possiede rapidità di calcolo e relativa facilità di implementazione; inoltre la resa visiva non è disprezzabile. Tuttavia è implementato in larga parte per la pipeline di rendering fissa ed è specifico per le schede video NVIDIA [2];
- *Light Space Perspective Shadow Maps*: algoritmo valido di cui si parlerà in seguito [cfr. paragrafo 6.5.1];

Per i motivi sopra riporati, dopo rapide fasi di test, questi algoritmi sono stati scartati. Abbiamo invece usato come punto di partenza per il nostro progetto l'algoritmo open-source *Soft-Edged Shadows*, di cui si parlerà diffusamente più avanti [cfr. sezione 6.6].

6.3 Descrizione dell'algoritmo Shadow Map

Il concetto di mappe di ombreggiatura o Shadow Maps fu introdotto nel 1978 da Lance Williams ed è un concetto di fatto molto semplice. Una mappa di ombreggiatura è una qualche rappresentazione di una scena che sia in grado di dirci se, dal punto di vista di

[1] cfr. il sito web http://www.punkuser.net/vsm/
[2] cfr. il sito web http://dee.cz/fcss/

una sorgente luminosa, un oggetto è illuminato o no. Questo si traduce nel catturare una vista della scena dalla sorgente luminosa guardando nella direzione della luce. Tutti gli oggetti visibili risulteranno illuminati, mentre quelli non visibili saranno in ombra.

Questo si traduce in un render della scena dal punto di vista della luce, memorizzando la profondità dei pixel illuminati (per questo motivo le *Shadow Maps* sono anche chiamate mappe di profondità); questo render viene effettuato su di una texture, ed utilizzato nella resa finale della scena. In questa fase si confronta la distanza dalla luce di ogni pixel con quella presente nella mappa di profondità. Visto che, grazie allo Z-Buffer, nella mappa sono memorizzate le distanze dei pixel più vicini alla luce; se la distanza del pixel da renderizzare risulta maggiore di quella nella *Shadow Map* tale punto è da considerarsi in ombra.

Da questa semplice descrizione emerge che per calcolare le ombre in tempo reale è necessario, per ogni fotogramma, renderizzare la scena due volte: la prima dal punto di vista della luce, e la seconda, come di consueto, dal punto di vista dell'osservatore. Il primo render non ha come destinazione lo schermo, ma una porzione di memoria video differente, che ha la caratteristica di poter essere usata come texture. Questa pratica prende il nome di *off-screen rendering*, poichè l'immagine della scena non viene mostrata all'utente, ma è utilizzata per altri scopi.

6.4 Realizzazione dello Shadow Map semplice

La versione base dell'algoritmo è abbastanza semplice da implementare in Wild Magic, sia come strutture dati che come shader. E' stato realizzato un effetto nuovo, chiamato *ShadowMapEffect*, che ha la peculiarità di avere influenza globale. Un effetto globale si distingue da quelli normali perchè si attacca ad un nodo, ma viene applicato a tutte le geometrie presenti al di sotto di questo. Gli effetti globali hanno il vantaggi di poter ridefinire il metodo di disegno; possono quindi alterare, soltanto per le geometrie sotto la loro influenza, il ciclo naturale di rendering. Ciò risulta ideale per realizzare le ombre dinamiche perché da la possibilità di eseguire due volte il ciclo di rendering.

6.4.1 Strutture dati

Vediamo ora con maggiore dettaglio lo *ShadowMapEffect*. Tra i dati che questa classe mantiene, il più importante è il riferimento al *FrameBuffer* (FBO) su cui eseguire la *Shadow Map*. Questa classe è uno degli elementi chiave di Wild Magic, poichè rappresenta l'astrazione del *Frame Buffer* della scheda video. Le API grafiche danno la possibilità di istanziare degli FBO aggiuntivi rispetto a quello predefinito usato per lo schermo. Associando a qusti FBO una texture, la scheda video reindirizzerà il suo output direttamente sull'immagine, che può essere poi riutilizzata in fasi successive.

Un altro elemento chiave di questo effetto è la telecamera associata alla luce. Per effettuare una ripresa della scena è necessaria una camera, e la taratura dei suoi parametri (frustum e frame[3]) diventa fondamentale per ottenere delle ombre di qualità.

[3]Per i dettagli del modello di camera Cf. http://en.wikipedia.org/wiki/Viewing_frustum

Infine, si utilizzano due effetti, uno per il primo passo di rendering ed uno per il secondo, oltre alla texture su cui memorizzare la mappa di profondità. Riportiamo per maggiore chiarezza lo scheletro della classe *ShadowMapEffect*:

```
class ShadowMapEffect : public Effect
{
public:
    ShadowMapEffect (Camera* pkProjector,
                         const std::string& rkProjectionImage,
                         int iDepthWidth,
                         int iDepthHeight,
                         float fDepthBias);

    virtual ~ShadowMapEffect ();

    virtual void Draw (Renderer* pkRenderer, Spatial* pkGlobalObject,
        int iMin, int iMax, VisibleObject* akVisible);

    void SetDepthBias (float fDepthBias);
    float GetDepthBias () const;

protected:
    ShadowMapEffect ();

    CameraPtr m_spkProjector;           // Shadow Caster camera
    ShaderEffectPtr m_spkDepthEffect;   // Depth Map generator effect
    FrameBuffer* m_pkDepthBuffer;       // Frame Buffer Object
    ImagePtr m_spkDepthImage;           // Image for offscreen rendering
    ShaderEffectPtr m_spkShadowEffect;  // Final rendering effect
    Texture* m_pkDepthTexture;  // For testing (copy depth to system memory).

    // The depth bias is stored at index 0.  The other array values are
    // unused.
    float m_afDepthBias[4];
};
```

Questa classe ha una struttura molto semplice, ma il suo utilizzo è di contro un po' macchinoso. Il puntatore alla camera chiamato *m_spkProjector* , ad esempio, rappresenta la camera per la proiezione dal punto di vista della luce. Nella gestione a shader questo oggetto si traduce in una matrice omogenea che viene calcolata implicitamente dal motore e poi passata agli shader.

Questa matrice è il risultato di un prodotto di tre matrici di trasformazione note come *World*, *View* e *Projection*. Le ultime due dipendono dalla configurazione del *Frame* e del *Frustum*, e sono quindi configurabili comodamente chiamando dei metodi della classe Camera. La matrice *World*, invece, dipende dalla posizione e dall'orientamento della camera rispetto al sistema di riferimento globale.

Per impostare la posizione della camera è necessario attaccarla ad un nodo particolare, il *CameraNode*, e poi applicare la traslazione al nodo stesso con la sequenza di istruzioni:

```
m_spkFlashlight = WM4_NEW Node;
m_spkFlashlight->Local.SetTranslate(Vector3f(0.0f,0.0f,4.0f));
m_spkScene->AttachChild(m_spkFlashlight);
Camera* pkProjector = WM4_NEW Camera;
pkProjector->SetFrustum(120.0f,1.0f,1.0f,5.0f);
m_spkFlashlight->AttachChild(WM4_NEW CameraNode(pkProjector));
```

Questo ha lo svantaggio di disaccoppiare l'oggetto che identifica la luce dalla camera necessaria per proiettare le ombre; per completare l'inizializzazione è necessario infatti attaccare anche un oggetto *Light* alla scena. Questo può generare situazioni in cui la sorgente di luce si trova a determinate coordinate, mentre le ombre fanno riferimento ad una posizione differente.

Degna di nota è la presenza del puntatore ad una texture, il membro *m_pkDepthTexture*; questo riferimento non è strettamente necessario, in quanto le texture utilizzate come destinazione del render prima, e come mappa di profondità poi, sono tutte associazioni ad aree di memoria opportunamente riservate sulla scheda video. La chiave per avere prestazioni real-time è infatti quella di tenere più dati possibili stabilmente sulla scheda video, evitando l'oneroso passaggio di dati sul bus. Questo puntatore è utilizzato in fase di debug per poter scaricare dalla scheda video il risultato intermedio del primo passo di rendering. Tramite le istruzioni:

```
m_pkDepthBuffer->CopyToTexture(true);
Image* pkImage = m_pkDepthBuffer->GetTarget()->GetImage();
pkImage->Save("shadowmap.wmif");
```

è possibile salvare l'immagine in un formato interno di Wild Magic, il *wmif*. Per poter visualizzare il contenuto dell'immagine sono necessari i programmi di conversione e di visualizzazione forniti con il motore.

Per concludere, questa struttura dati è internamente semplice ed aderente all'algoritmo base, ma render difficoltosa al programmatore la sua configurazione.

6.4.2 Funzionamento

L'inizializzazione dei parametri è demandata totalmente al programma chiamante: in questo caso è stata privilegiata l'integrazione nel motore a scapito della facilità di utilizzo. Per configurare correttamente questa classe, infatti, bisogna impostare manualmente non solo i parametri della luce, ma anche la dimensione del frustum della telecamera, nonché l'entità del parametro di *bias* per la fase di confronto. Ciò che manca del tutto è un sistema dedicato che consenta di stabilire quali oggetti sono nel campo visivo della luce (un *Culler*).

Una volta inizializzata la scena, lo *Shadow Mapping* è assolutamente automatico. Durante il ciclo di rendering tutti gli oggetti che stanno al di sotto del nodo cui è attaccato l'effetto di ombreggiatura non sono elaborati nella maniera tradizionale; la loro resa è infatti demandata alla funzione *Draw* del nostro effetto. Questa, per ogni geometria, applica in una prima fase soltanto l'effetto che crea la mappa di profondità, e successivamente effettua il rendering tradizionale.

Per chiarezza riportiamo lo pseudocodice di questo metodo:

```
Se il Frame Buffer non esiste ancora
{
        Crea ed inizializza il Frame Buffer della shadow map(...);
}

Imposta la camera della luce come corrente, ed abilita il Frame Buffer.

Per ogni oggetto visibile
        attacca l'effetto Mappa Di Profondità
        disegna applicando solo questo effetto
        stacca l'effetto

Reimposta la camera dell'osservatore come corrente
Disabilita il Frame Buffer

Imposta la camera della luce come parametro dello shader, con il comando:
pkRenderer->SetProjector(m_spkProjector);

Per ogni oggetto visibile
        Attacca l'effetto che disegna le ombre
        Disegna applicando tutti gli effetti propri di quell'oggetto
        Stacca l'effetto che disegna le ombre

Rimuovi la camera della luce dai parametri:
pkRenderer->SetProjector(0);
```

Possiamo considerare questa implementazione la base di partenza, utile per comprendere il meccanismo, ma ancora troppo grossolana.

6.4.3 Descrizione degli shader

Prima di perfezionare la descrizione delle ombre dinamiche, è il caso di soffermarsi sugli shader necessari per ottenere le ombre dinamiche semplici, dette anche "dure" per il passaggio netto tra zone in luce e zone in ombra. Il codice degli shader è abbastanza auto-esplicativo[4]. Il vertex-shader per la mappa di profondità riceve in ingresso soltanto la posizione del vertice; grazie alla matrice di proiezione della luce(la *World View Projection*) trasforma quelle coordinate nel *Light space* le restituisce in uscita dopo una opportuna divisione per la coordinata omogenea w. In questo modo sulla shadow map compariranno soltanto valori di profondità normalizzati tra zero e uno. Il pixel shader riceve in ingresso la profondità del pixel, derivata dall'interpolazione sul triangolo, e restituisce in uscita quello stesso valore.

Leggermente più complessa è la coppia di shader per la fase di confronto. Nel vertex si trasformano le coordinate tridimensionali del vertice con due matrici diverse, producendo due output. Il primo è il vertice nel *Clip space* dell'osservatore, il secondo è la coordinata in cui si deve campionare la mappa di profondità per avere il valore corrispondente a

[4]Per i listati si veda l'Appendice B

quel vertice. La matrice utilizzata per questo calcolo è sempre una WVP *World View Projection*, cui è stata pre-moltiplicata una matrice di scalamento per adattare il risultato alle coordinate texture.

Nel pixel shader avviene il confronto. Si confronta ogni coordinata in ingresso con il campione corrispondente a quelle coordinate nella mappa di profondità: se risulta in luce, lo shader ritorna in output il colore proprio del pixel, altrimenti ritorna nero. Degno di nota è il termine *depth bias*, un termine che serve per amplificare la differenza di profondità tra i pixel; questa componente è necessaria perchè il confronto tra numeri in virgola mobile, quando i valori sono prossimi allo zero, produce delle fluttuazioni che generano sgradevoli imperfezioni sui bordi delle ombre. Questo fenomeno è noto come *Z-fighting*, e una delle soluzioni a questo problema è amplificare la differenza tra i valori; l'inconveniente di questa tecnica è che il valore del bias deve essere impostato empiricamente e non c'è una formula esatta che consente di stabilirlo. Molto delicata è anche la scelta della matrice di proiezione

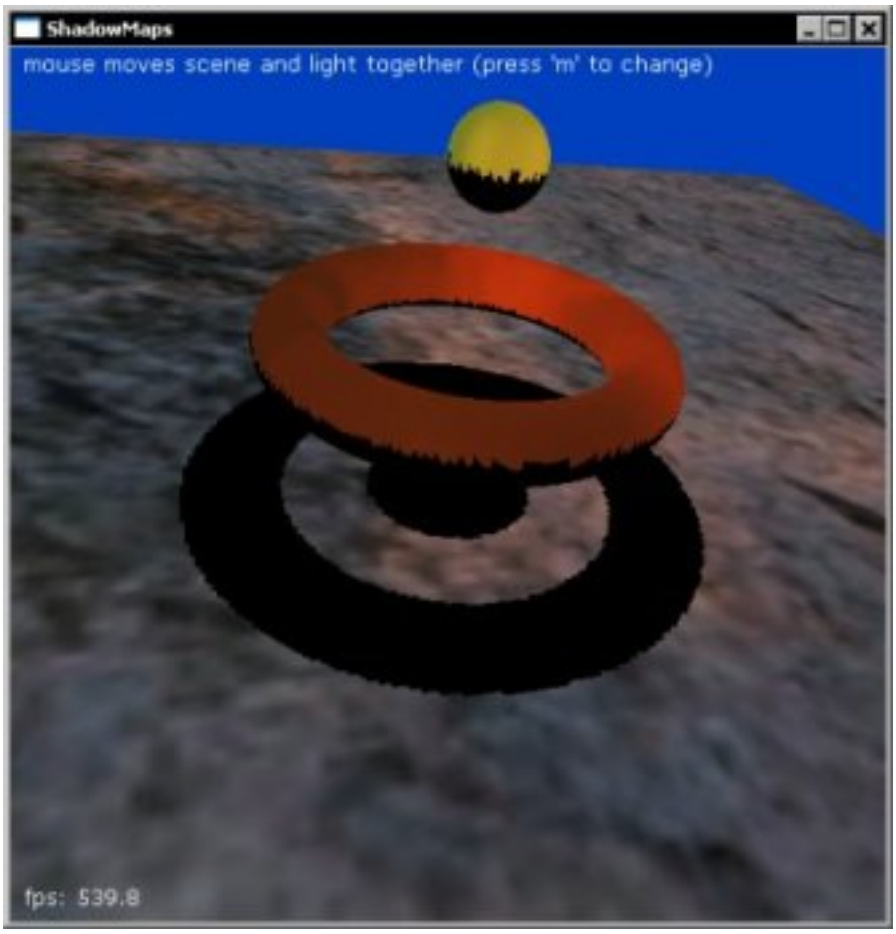


Figura 6.2. Una impostazione errata del bias può generare imperfezioni sui bordi tra luce e ombra. Notare in particolare gli effetti dello Z-Fighting sul toro e sulla sfera.

della luce, perchè non basta che punti nella direzione giusta, il frustum deve anche essere

abbastanza grande da contenere la scena che vogliamo illuminare senza essere troppo grande e sprecare spazio nella mappa di ombreggiatura. Questa matrice è in generale prospettica, ma nel caso di luce posta all'infinito può ridursi ad una matrice ortografica; fortunatamente il calcolo di questi parametri può essere fatto automaticamente a partire dalle informazioni geometriche della scena (in genere del suo Bounding Volume) e dalla direzione della luce.

6.4.4 Conclusione

L'ombreggiatura tramite mappe di profondità è un'idea semplice ed efficace per realizzare ombre che rispecchiano le relazioni geometrice tra gli oggetti. L'impostazione minimale descritta nei paragrafi precedenti produce un risultato gradevole visivamente solo in contesti molto limitati. Nella sezione successiva illustremo quali sono le modifiche necessarie per realizzare le ombre dinamiche che simulino in maniera credibile la luce solare, producendo ombre dall'aspetto più realistico.

6.5 I limiti dello Shadow Map semplice

In questa sezione illustreremo quali sono i principali limiti dello Shadow Map semplice, proponendo alcune soluzioni tratte da articoli pubblicati recentemente. Il nostro obiettivo rimane quello di realizzare ombre dinamiche prodotte da una sorgente luminosa posta a distanza infinita, come il sole; questo requisito porta l'algoritmo a funzionare in condizioni limite, mettendo in evidenza i suoi difetti. Le soluzioni abbiamo realizzato cercano di mettere in pratica alcuni degli articoli pubblicati recentemente, al fine di combinare basso impatto sul *framerate* e buona resa visiva.

L'hardware delle schede video influisce sensibilmente sulla realizzazione di questo algoritmo tramite shader, in particolare nella fase di confronto tra le profondità dei pixel. La precisione limitata dei registri in virgola mobile e gli errori di arrotondamento nei registri a virgola fissa hanno un impatto rilevante sull'operato del *depth buffer*, al punto tale da meritare l'appellativo in letteratura di *Z-fighting*. Lo *shadow mapping* però presenta altri due grossi difetti, il cui superamento è tuttora oggetto di studi e pubblicazioni. Questi problemi sono noti in letteratura come *aliasing prospettico* e *acne*.

6.5.1 L'aliasing prospettico

La cattura di una vista della scena dal punto di vista della luce comporta inevitabilmente un campionamento; come avviene in ogni rappresentazione di una scena 3D, questo genera aliasing. I fattori che determinano l'aliasing sono due: la dimensione della texture su cui memorizziamo la mappa di profondità e la configurazione della camera con cui effettuiamo la ripresa.

Consideriamo come esempio di aver scelto una texture per l'*offscreen rendering* di dimensioni 512x512 pixel: questo significa che tutta la scena ripresa dal punto di vista della luce verrà memorizzata in uno spazio di quelle dimensioni. Consideriamo ancora che la finestra in cui vogliamo mostrare la scena ombreggiata abbia risoluzione 1024x768. In

questo caso, anche con la migliore delle impostazioni della camera, dovremo applicare una mappa di profondità ad una scena di dimensioni quasi doppie, con conseguente necessità di interpolare dei dati e dunque aliasing.

Una soluzione banale a questo problema è utilizzare texture più grandi per contenere la mappa di profondità. Questa soluzione è praticabile, ma non risolve tutti i problemi. C'è anche da dire che se su questa texture è necessario compiere ulteriori elaborazioni, non è consigliabile utilizzarne una troppo estesa onde evitare un eccessivo carico computazionale a livello di pixel shader.

Una volta impostate le dimensioni della *Shadow Map* ad una grandezza ragionevole (solitamente il doppio della finestra video se questa è piccola, altrimenti non superare 2048x2048 pixel), ciò che resta da fare per ridurre l'aliasing è sfruttare al meglio questa mappa di profondità. Bisogna inoltre tener presente che nel nostro caso stiamo cercando di simulare la luce solare, per cui nella mappa di ombreggiatura finirà tutta la scena, indipendentemente dalla posizione dell'osservatore; questo si traduce, in fase di resa finale, nell'avere la stessa risoluzione di ombre per gli oggetti presenti nel campo visivo dell'osservatore, indipendentemente dalla loro distanza.

Le soluzioni all'aliasing presenti in letteratura si possono categorizzare in base allo spazio geometrico in cui operano. Nella catena di trasformazioni geometriche che portano un punto nel mondo tridimensionale a proiettarsi sulla finestra del nostro schermo ci sono due fasi in cui si può intervenire: lo spazio dell'immagine e lo spazio prospettico.

Soluzioni nello spazio dell'immagine Operare nello spazio dell'immagine significa calcolare la mappa di profondità in maniera tradizionale, ed attuare modifiche direttamente sull'immagine subito prima di mostrarla a schermo. In termini pratici questo si traduce in veri e propri filtraggi i cui algoritmi derivano dal campo dell'*Image Processing*. Queste elaborazioni consentono di rendere i bordi delle ombre più sfumati e in alcuni casi attenuano gli effetti di *Z-fighting*. Una delle tecniche più usate è nota come PCF (*Percentage Closer Filtering*), presentata da Reeves e altri nel 1987 [11]. Come tutte le tecniche di elaborazione di immagini, agisce pixel per pixel, combinando opportunamente i contributi di un determinato in torno del punto; questo intorno, in genere quadrato, è detto Kernel.

E' anche vero che la mappa di profondità non è una texture come le altre, quindi applicare un filtraggio classico porterebbe risultati inconsistenti in fase di confronto. Quello che si fa è confrontare la profondità del pixel corrente con quelle del kernel; si stabilisce per quanti pixel risulterebbe in luce e quanti in ombra e si ricava la percentuale di ombreggiatura. Con questo principio si ottengono anche effetti sofisticati, come per esempio ombre più marcate nei punti di contatto tra gli oggetti e più soffuse alle altre estremità[5]. Per ridurre l'aliasing usando il PCF o altri filtraggi come quello gaussiano, spesso sono necessari kernel molto ampi; questo accade se la luce è molto lontana o l'inquadratura dedica poco spazio a zone che invece ne occupano molto nel campo visivo dell'osservatore. Usare kernel ampi comporta uno sgranamento eccessivo del bordo dell'ombra, senza contare che su alcune schede video può non essere implementabile.

[5]Cf Percentage Closer Soft Shadows,
http://developer.download.nvidia.com/shaderlibrary/docs/shadow_PCSS.pdf

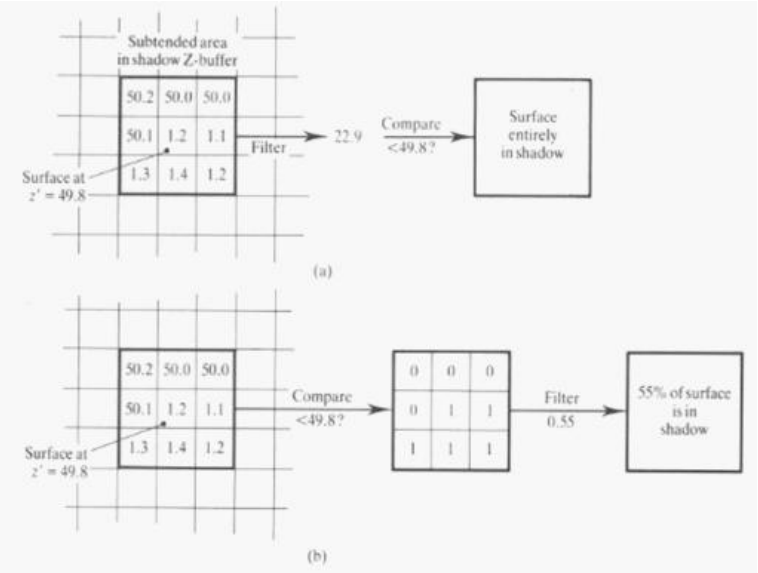


Figura 6.3. a. Filtro con media aritmetica sulle profondità e poi test b. PCF: Prima test tra tutte le profondità e poi media aritmetica dei valori binari

Soluzioni nello spazio prospettico Modificare lo spazio prospettico consente di sfruttare la mappa di profondità al meglio. L'idea è effettuare il calcolo della shadow map ed il successivo confronto di profondità in uno spazio geometrico diverso da quello del frustum della luce. Portando le coordinate della scena in uno spazio che si definisce *post-prospettico* siamo in grado di applicare una distorsione tale da ingrandire gli oggetti più vicini all'osservatore rispettando le relazioni di profondità ma eliminando di fatto l'aliasing.

Questi algoritmi, che prendono il nome di *Perspective Shadow Maps* e *Light Space Perspective Shadow Maps*[6] hanno come difetto principale che rendono il calcolo delle ombre dipendente dalla posizione dell'osservatore e dalla configurazione della sua camera. Ciò comporta il ricalcolo delle matrici necessarie ad ogni fotogramma, ma è un prezzo decisamente basso da pagare confrontato con i vantaggi che porta, tanto più che essendo dipendenti dal campo visivo dell'osservatore, si limita sensibilmente il numero di oggetti da renderizzare nella shadow map.

6.5.2 L'acne

"Acne superficiale" è il termine che in informatica grafica indica gli artefatti di falsa auto-ombreggiatura su superfici che dovrebbero essere illuminate, ed esiste indipendentemente dalla risoluzione o dalla precisione del campionamento. La soluzione che presentiamo prende il nome di *Gradient Shadow Mapping*[7], ed è in grado di ridurre l'acne introducendo alcuni ulteriori parametri che modificano la funzione di confronto tra le profondità. Come si può vedere nella figura 6.4 circa il 50% della scena è ricoperto da false ombre disposte

[6] Cf. [12]
[7] Cf. [12] p. 289

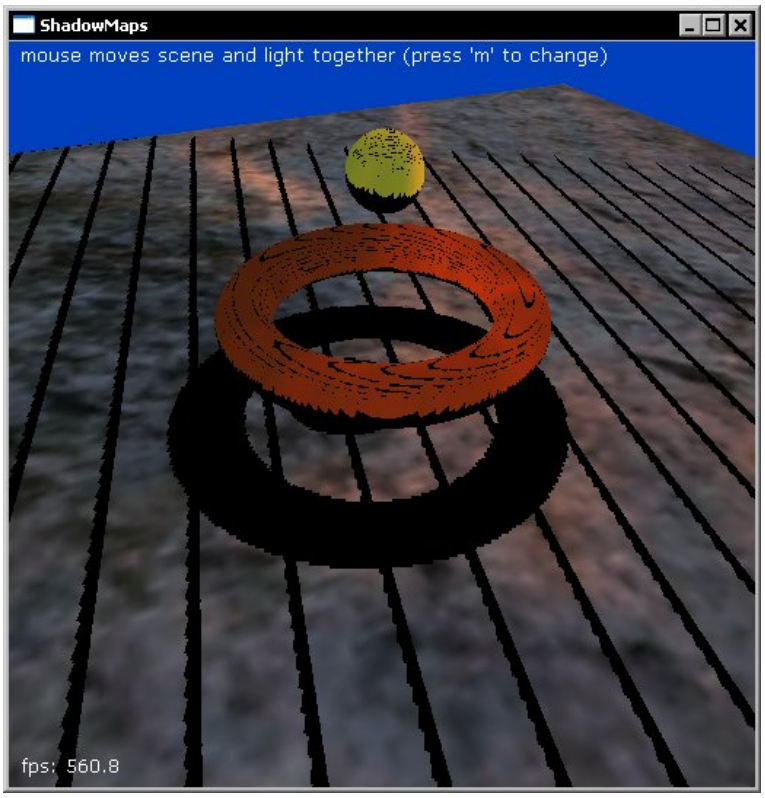


Figura 6.4. Nell'implementazione base delle shadow map si verifica il fenomeno dell'acne.

in bande o talvolta in triangoli. Aumentando la risoluzione della mappa gli artefatti rimangono, sono solo più piccoli e più fitti. La ragione di questo artefatto è che la mappa di profondità contiene una rappresentazione campionata delle superfici più vicine alla sorgente luminosa, e a causa del campionamento è a gradini. La superficie continua è poi confrontata con quella discretizzata. Sui confini tra luce ed ombra, se il pixel in questione è esattamente al centro del campione, la sua profondità sarà identica a quella della shadow map(assumendo precisione infinita). In tutti gli altri casi, c'è il 50% di possibilità che un punto casuale sulla faccia originale si ritrovi dietro alla superficie campionata, dando origine ad un'ombra falsa. Vale la pena ricordare che ciò non è dovuto alla precisione limitata dei calcoli, è un limite intrinseco nel sistema.

La prima soluzione a questo problema, introdotta già dall'autore dell'algoritmo nel 1978 e citata già in precedenza, è l'introduzione di una costante, il *depth-bias*, che amplifica la differenza da confrontare. Abbiamo già detto che questa è una soluzione puramente euristica, e che il valore giusto da impostare dipende dalla scena in questione, e spesso si impostano valori conservativi a scapito di alcuni inevitabili artefatti. Nonostante questo, è tuttora la soluzione più usata per ridurre l'acne.

Dato che questo fenomeno è strettamente collegato all'angolo di incidenza della luce sulla superficie, il *bias* costante non è sufficiente quando si ha a che fare con curvature

elevate e variazioni brusche delle normali. E' necessario introdurre un altro parametro, che tenga conto di questo fattore. Questo coefficiente è proporzionale alla tangente dell'angolo di incidenza della luce, e si usa direttamente per moltiplicare il bias.

Per ricavare questo dato è sufficiente, prima di fare il confronto tra le profondità, calcolare la differenza massima tra i campioni vicini; l'esperienza consiglia anche di porre un limite di saturazione, poichè il gradiente potrebbe variare troppo e portare risultati inconsistenti.

Per completare l'algoritmo si modifica la funzione di confronto, passando da quella a gradino (determinata dalla soglia costante) ad una funzione a rampa, la cui pendenza è un ulteriore parametro di taratura. Dai nostri esperimenti risulta che questo algoritmo elimina l'acne in un buon numero di casistiche; l'inconveniente maggiore è ancora una volta la euristica con cui bisogna scegliere i parametri: non è stato ancora messo a punto un sistema automatico di stima di questi parametri basandosi sulla geometria della scena.

6.6 Realizzazione dello Shadow Map avanzato

Con il nome di *Shadow Map avanzato* intendiamo indicare la versione dell'algoritmo cui abbiamo aggiunto le migliorie atte a limitarne i principali difetti. Partendo dalla versione semplice, chiamata *Soft-Edged Shadows*, abbiamo cercato di implementare diversi algoritmi[8], fino ad ottenere un insieme di classi e strutture dati che costituissero uno scheletro per un sistema di ombreggiatura di qualità. Questo ci consente da un lato di offrire sin da ora il calcolo di ombre dinamiche, e dall'altro è aperto ad ulteriori miglioramenti.

6.6.1 Il funzionamento

L'implementazione che proponiamo realizza ombre dai confini sfumati, che tenta di correggere l'aliasing operando nello spazio dell'immagine con un filtraggio gaussiano, e che rimuove l'acne grazie all'algoritmo del *Gradient Shadow Mapping*[12].

Per ottenere questo risultato sono necessari due passi di rendering in più rispetto all'algoritmo base. Vediamoli nel dettaglio:

Mappa di profondità: In questa fase si genera la mappa di profondità esattamente come nella versione base; l'unica differenza sta nel supporto utilizzato per memorizzare i dati: in questo caso infatti usiamo texture in virgola mobile su 32-bit, perchè ci serve un ventaglio maggiore di dati affinché abbiano effetto le elaborazioni successive. Nella versione base erano sufficienti interi su 8-bit.

Maschera di ombreggiatura: Si effettua un render della scena dal punto di vista dell'osservatore, effettuando il test di profondità. E' in questo stadio che si applicano le misure contro l'acne. Il risultato è una texture grande come la finestra che mostreremo a video in cui per ogni pixel è presente il termine di visibilità: uno se totalmente in luce, e poi valori intermedi fino a zero per le parti completamente in ombra.

[8] Per le elaborazioni nello spazio dell'immagine ci siamo ispirati all'articolo [16]

Filtraggio gaussiano: Si applica un filtraggio gaussiano[9] alla maschera di ombreggiatura per smussare ogni imperfezione sui bordi delle ombre.

Resa finale: Nel renderizzare la scena finale si calcola il colore del pixel come se l'ombra non ci fosse, poi si combina il risultato con il rispettivo pixel della maschera di ombreggiatura.

6.6.2 Le strutture dati

Per integrare il calcolo di ombre dinamiche nel motore Wild Magic abbiamo derivato due classi: una dal tipo nodo e una dal tipo effetto. La nuova classe nodo, chiamata *Shadower*, rappresenta la radice della scena che verrà ombreggiata con le Shadow Maps e mantiene le informazioni di stato e gli strumenti necessari per realizzare l'algoritmo. Il nuovo effetto, chiamato *LightShadowEffect* è attaccato a tutte le geometrie da ombreggiare e si occupa di combinare l'illuminazione tradizionale con la maschera di ombreggiatura.

Il nodo Shadower La classe nodo contiene i dati che hanno validità per tutta la scena; per semplificarne la comprensione riporteremo lo scheletro della classe più volte, raggruppando i campi per modalità di utilizzo. I membri necessari per gestire la mappa di profondità sono riportati in appendice [B.2].

Come si può notare, i membri fondamentali sono gli stessi dello *ShadowMapEffect* base. La differenza maggiore sta nel fatto che abbiamo deciso di calcolare la matrice di proiezione della luce direttamente nella classe, senza demandare questo compito al motore grafico. In questo modo abbiamo la possibilità di impostare i valori della camera a nostro piacimento, e soprattutto possiamo impostare la posizione della luce direttamente in coordinate mondo. Il calcolo della matrice all'interno della classe nodo ne facilita anche la manipolazione in caso si decidesse di implementare algoritmi per la riduzione dell'aliasing nello spazio prospettico. Abbiamo inoltre inserito il *Culler*, necessario per limitare il numero di oggetti da renderizzare ai soli davvero presenti nel campo visivo della luce. Questo aggiunge ulteriore flessibilità al sistema, consentendo, ad esempio, di attaccare al di sotto del nodo *Shadower* anche geometrie che si trovano fuori dal frustum della luce, senza correre il rischio di applicare shader su mesh che verranno poi scartate dalla scheda video.

Vediamo ora quali sono gli oggetti necessari per realizzare la seconda fase, in cui si realizza una maschera di ombreggiatura grande quanto la finestra che apparirà sullo schermo [B.2].

I membri destinati a questo scopo sono ovviamente dello stesso tipo dello shadow map, dovendo effettuare anche qui una passata di *Render-to-texture*. Le variabili in più saranno passati allo shader per ridurre l'acne [6.5.2]; questi dati sono impostati in maniera puramente euristica ed hanno la seguente funzione:

- DepthBias è il termine costante che amplifica la differenza di profondità (ordine di grandezza: 0 - 0.3);

[9] Cf. http://en.wikipedia.org/wiki/Gaussian_filter

- GradientClamp è la soglia massima cui limitiamo il termine dipendente dal gradiente di profondità (ordine di grandezza: 0 - 0.2);
- GradientScaleBias è un coefficiente che determina il peso del gradiente nel calcolo della soglia dinamica (ordine di grandezza: 0 - 2);
- FuzzyWidth determina la pendenza della funzione di sogliatura; nell'algoritmo del *Gradient Shadow Mapping*, infatti, è prevista la sostituzione del confronto a soglia costante (funzione a gradino) con una rampa (ordine di grandezza: 0 - 2).

La successiva fase di filtraggio(Blur) richiede due passate: l'elaborazione dei pixel basata tenendo conto dei contributi dei pixel vicini in senso orizzontale e successivamente verticale. Per limitare il numero di offscreen texture, abbiamo scelto di utilizzare come destinazione del Blur orizzontale un'apposita area di memoria, mentre per il successivo Blur verticale riutilizziamo la texture della fase di Unlit. I campi necessari per questa fase sono riportati in appendice [B.2].

Anche in questo caso abbiamo i campi necessari al *render-to-texture*, il *Frame Buffer* l'immagine di destinazione e la texture di verifica. A questi si aggiungono una camera ed un poligono. Il poligono è un rettangolo cui verrà applicata l'immagine da filtrare, attraverso i *BlurEffect*; la camera sarà impostata in modalità ortografica e inquadrerà solamente il rettangolo. Questo passo di render, infatti, non appesantisce l'elaborazione dal punto di vista dei vertici da processare, in quanto di fatto si disegnano soltanto i quattro vertici del rettangolo; il vero carico sta nel processamento dei pixel, in cui lo shader, come vedremo più avanti, effettuerà una somma di prodotti tra numerosi campioni della maschera di ombreggiatura. I campionamenti in aree di memoria sono tra le operazioni più costose nell'economia generale degli shader, e applicare lo stesso calcolo per tutti i pixel influisce considerevolmente sulle prestazioni. Per questo motivo si consiglia di limitare l'estensione della maschera di ombreggiatura ad una dimensione di 1024x768 o 1280x1024 pixel.

I metodi principali Tra i molti metodi di questa classe, ci soffermeremo su due categorie: i metodi della classe generica *Node* che sono stati ridefiniti, ed i metodi statici per la lista dinamica. I metodi che sono stati ridefiniti sono:

```
int AttachChild(Wm4::Spatial *pkChild);
int DetachChild(Wm4::Spatial *pkChild);
void UpdateRS (std::vector<GlobalState*>* akGStack = 0, std::vector<Light*>* pkLStack = 0);
```

Grazie al polimorfismo siamo riusciti a rendere il funzionamento di questo nodo molto semplice, nascondendo all'interno di metodi che sono chiamati abitualmente su tutti i nodi le nuove necessità di questo algoritmo. I metodi AttachChild e DetachChild, oltre alla loro funzione base di gestione del grafo della scena, richiamano l'aggiornamento dello stato interno del nodo. Nel caso della AttachChild si attacca ai nuovi figli l'effetto *LightShadowMap*, nella Detach questo viene rimosso. In entrambi i metodi si ricalcola l'insieme di oggetti visibili dal punto di vista della luce.

Il metodo *UpdateRS*, che può essere chiamato in qualsiasi condizione della scena (se lo *Shadower* non è ancora stato inizializzato si comporta come un nodo tradizionale): questo

metodo consente di propagare ricorsivamente un cambiamento di stato a tutto l'albero che sta al di sotto di un nodo. Ciò che abbiamo aggiunto è un insieme di istruzioni che attaccano alla geometria un effetto *LightShadow* e che ne modificano i parametri all'occorrenza.

Per identificare un *LightShadowEffect* confrontiamo i puntatori agli oggetti *Light* e *Texture* dell'effetto con gli analoghi oggetti del nodo; per questo motivo nel nodo dobbiamo conservare due copie di questi dati, una vecchia ed una nuova, in modo da propagare correttamente la variazione di questi dati. Lo pseudocodice dell'aggiornamento è il seguente:

```
// Metodo ricorsivo che valuta il puntatore pkMesh:
Se pkMesh è di tipo Mesh
pkLS = Prendi il LightShadowEffect di pkMesh;
spTexEff = Prendi il texture effect di pkMesh;
se ne ha uno
{
memorizza le impostazioni di trasparenza ed il puntatore alla texture;
stacca spTexEff da pkMesh;
}
else
{
utilizza una texture predefinita bianca
}
se pkMesh non ha un LightShadowEffect
{
creane uno ed attaccalo a pkMesh
}
//Configura il LightShadowEffect con i parametri aggiornati
pkLS->UpdateMembers(m_spLight, m_spkUnlitTarget, spApply, pkTexture);
```

Questa classe definisce infine un'interfaccia statica, che ha la seguente struttura:

```
class UM_Shadower: public Node
{
public:
        //      Linked list members and methods
        static UM_Shadower* sm_pCHeadList;
        UM_Shadower*    m_pCNextItem;
        static void InsertIntoList(UM_Shadower *pC);
        static void RemoveFromList(UM_Shadower *pC);
        static int      ApplyAll(Renderer *pkRenderer, Camera *pkCamera);
        static void OnResize(Renderer *pkRenderer, int iScreenWidth, int iScreenHeight);
};
```

L'obiettivo è avere una *linked list* di nodi di tipo *Shadower* e un insieme di metodi statici che operano su tutta la lista. In questo modo l'applicazione che utilizzerà questi nodi non dovrà tener traccia della loro quantità e collocazione quando dovrà modificarne lo stato. Un esempio è il metodo *ApplyAll*, che andrà inserito nel ciclo di rendering tradizionale dell'applicazione e si occuperà di generare la maschera di ombreggiatura attraverso i tre passi di rendering [6.6.1].

Il LightShadowEffect Nella versione tradizionale del motore Wild Magic, per renderizzare una scena in cui gli oggetti presentano un materiale ed una texture e sono illuminati da una qualche sorgente luminosa, sono necessari due shader (e quindi due effetti). Il *LightShadowEffect* si sostituisce al *TextureEffect* ed al *LightingEffect*, applicando anche le ombre dinamiche.

Questa versione è concepita per illustrare il comportamento dell'algoritmo, e non tiene conto di effetti speciali quali multitexture o altro: gestisce soltanto il caso più tipico, quello di un oggetto cui è associato un materiale ed una texture; gli altri effetti richiederebbero l'implementazione di shader appositi. I membri del *LightShadowEffect* ricalcano banalmente i parametri in ingresso agli shader che utilizzano: la luce, la maschera di ombreggiatura e la texture di colore dell'oggetto.

Per realizzare questo effetto siamo partiti dalla struttura del *LightingEffect*, la classe di Wild Magic che si occupa dell'illuminazione. Anche il nostro effetto, tramite l'*overload* dei metodi che modificano lo stato del render, può impostare la luce corrente nel *Renderer*, che potrà così passarne i parametri allo shader in maniera implicita. Questo effetto è sottoposto a potenziali modifiche nell'arco della sua esistenza, poichè quando si sposta la luce è necessario che la sua variabile membro sia aggiornata, e quando si ridimensiona la finestra e necessario ridimensionare la texture contenente la maschera di ombreggiatura. Da qui la necessità di avere i metodi *Getter, Setter e UpdateMembers*.

6.6.3 Utilizzo delle classi

In questa sezione descriveremo come utilizzare le classi *Shadower* e *LightShadowEffect* in una applicazione esterna.

La prima modifica che va fatta ad una applicazione generica per potervi applicare le ombre dinamiche è l'inserimento del metodo statico *ApplyAll* nel ciclo di rendering, subito prima delle chiamate tradizionali di disegno.

Questo metodo scorre tutta la lista di nodi *Shadower* presenti nella scena e per ognuno richiama il metodo *ApplyShadows*: è il metodo chiave dell'algoritmo, per cui ne riportiamo lo pseudocodice:

```
// Depth Map
Attacca l'effetto ShadowMap a tutti gli oggetti
Abilita il Frame Buffer associato
Applica solo questo effetto a tutte le geometrie visibili dalla camera della luce
Disabilita il Frame Buffer associato
Stacca l'effetto ShadowMap da tutti gli oggetti

//Unlit
Attacca l'effetto Unlit a tutti gli oggetti
Abilita il Frame Buffer associato
Applica solo questo effetto a tutte le geometrie visibili dalla camera della luce
Disabilita il Frame Buffer associato
Stacca l'effetto Unlit

// Disegna nello spazio dell'immagine
Imposta la camera ortografica
//Applica il blur orizzontale
Abilita il Frame Buffer del Blur
Applica l'effetto al solo rettangolo
```

```
Disabilita il Frame Buffer del Blur
Stacca l'effetto dal poligono

Idem per il Blur verticale, ma abilitando il Frame Buffer dell'unlit

Ripristina la camera dell'osservatore
```

La seconda modifica deve essere fatta in fase di creazione della scena. La porzione di grafo che si desidera ombreggiare deve essere attaccata ad una istanza della classe *Shadower*, la quale dovrà essere attaccata al resto del grafo della scena.

Dopo di che si devono chiamare le funzioni di inizializzazione dei Frame Buffer e della luce (*InitFBO* e *InitLight*): con la prima chiamata la classe istanzia gli oggetti necessari e inizializza le *Offscreen texture* alle dimensioni desiderate; con la seconda si imposta la direzione della luce ed il sistema, automaticamente, configura la camera in modo da inquadrare l'intera scena. In fase di configurazione il programmatore imposterà solamente la direzione della luce, sarà il sistema a collocare la camera ad una determinata distanza dalla scena, configurando il frustum per inquadrarla tutta.

Quello che si fa è posizionare la camera molto lontano dal centro della scena, per simulare la luce solare, e si imposta il frustum con i piani *near* e *far* a contatto con la *bounding sphere* della scena. Gli altri parametri sono calcolati con banali formule trigonometriche.

I nostri esperimenti hanno messo in evidenza qualche piccolo difetto nel calcolo del *Bounding Volume* della scena da parte del motore Wild Magic: spesso capita che, utilizzando la sfera per identificare i volumi occupari, le dimensioni della bounding sphere risultino leggermente superiori alle dimensioni effettive della scena. Il nostro algoritmo tara la camera in modo da inquadrare tutta la scena, e questo piccolo difetto fa sì che parte dello spazio della shadowmap vada sprecando inquadrando zone dove non c'è nulla, aumentando l'aliasing prospettico. Grazie al *Culler* ed alla camera il sistema è in grado di stabilire quali oggetti vanno parte dell'insieme visibile e che saranno quindi renderizzati. A questo punto è sufficiente chiamare il metodo *UpdateRS*, che sarebbe stato chiamato comunque in qualsiasi creazione di scena.

Questo completa tutte le inizializzazioni necessarie per generare le ombre. Nel ciclo di rendering tradizionale, per le geometrie ombreggiate sarà applicato il *LightShadowEffect*; questo effetto combinerà una texture, un materiale e la mappa di ombreggiatura nella resa finale.

6.6.4 Descrizione degli shader

Gli shader per realizzare questo algoritmo sono quattro, uno per ogni passo di rendering. Il primo è quello che genera la mappa di profondità, ed è identico a quello della versione base. Il vertex-shader riceve in ingresso le coordinate tridimensionali dei vertici e restituisce la distanza di ognuno di essi dalla luce. Il pixel shader non fa altro che restituire in uscita il valore che riceve in ingresso. Con questa fase trasferiamo su una texture floating point i valori di distanza dalla luce di tutti i pixel che risultano illuminati; ciò avviene perchè lo Z-test scarta tutti i pixel che risultano coperti, e lascia passare solo i più vicini all'osservatore, in questo caso la luce.

Unlit La seconda coppia di shader, chiamata *Unlit*, è più complessa. Il vertex shader riceve in ingresso i seguenti parametri costanti:

- la matrice di proiezione dell'osservatore (WVPMatrix);
- la matrice di roto-traslazione dalle coordinate locali della geometria alle coordinate del mondo (WMatrix);
- la matrice di proiezione della luce (LightVPMatrix);
- la matrice per calcolare le coordinate texture della maschera di ombreggiatura(LightVPBSMatrix).

Per il listato completo rimandiamo all'appendice B.2. Le matrici relative alla luce sono calcolate dalla classe *Shadower* nel metodo *CalcLightMatrices* della classe *Shadower*: come già detto, entrambe derivano dal prodotto di una matrice *View* con una matrice *Projection*. La prima descrive l'orientamento della camera nello spazio mentre la seconda specifica le dimensioni del frustum. La matrice LightVPBS deriva dalla prima, LightVP, cui viene premoltiplicata una matrice costante, necessaria per mappare le coordinate sulla texture correttamente, che ha i seguenti valori:

$$\mathcal{B}ias = \begin{bmatrix} 0.5 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.5 & 0.5 & 0.0 & 1.0 \end{bmatrix}$$

Grazie a queste matrici il vertex shader produce due risultati: il primo è la distanza di quel vertice dalla luce; il secondo è la coordinata texture dove andare a prendere il termine di confronto nella mappa di profondità.

Il pixel shader realizza l'algoritmo del *Gradient Shadow Mapping* [6.5.2]. Grazie ai parametri passati in ingresso calcola il termine di visibilità di ogni pixel; questo termine è una evoluzione dell'informazione binaria derivante dal confronto tra le profondità: invece di restituire in luce/in ombra si restituisce una valore intermedio. Le istruzioni chiave sono quelle per il calcolo del gradiente: grazie alla funzione *tex2D* del CG si campiona la mappa di profondità nei pixel sopra, sotto, a destra e a sinistra. Si calcola poi la differenza massima tra questi quattro valori ed il campione centrale. Per il gradiente si tiene il minimo tra questo valore ed una soglia cautelativa impostata dall'esterno. Per il listato dello shader rimandiamo all'appendice B.2.

Blur L'applicazione di questo shader è un po' elaborata. Per applicare il filtraggio gaussiano sull'immagine, infatti, occorre creare una mesh a forma di rettangolo cui applicare la maschera di ombreggiatura come texture. Poi bisogna creare una opportuna telecamera ortografica che riprenda soltanto quel rettangolo. Solo così nel ciclo di render si può elaborare la nostra texture. Gli shader del filtraggio sono molto semplici, calcolano una semplice somma di prodotti tra campioni della maschera e un vettore di pesi opportunamente precalcolato dal nodo *Shadower*. Questo passo di rendering è tra i più onerosi

dal punto di vista computazionale dell'intero algoritmo; potenzialmente potrebbe essere rimpiazzato dal filtraggio PCF [6.5.1] con un kernel abbastanza ampio, ma i nostri test hanno dimostrato che l'implementazione di tale filtraggio presenta problemi di portabilità, almeno per quanto riguarda lo *Shader Model 2.0*.

Light Shadow Effect Gli shader di questo effetto sono costruiti a partire dallo shader della luce direzionale di Wild Magic, con l'aggiunta della componente di ombreggiatura. Riutilizzando quel codice abbiamo evitato di scrivere le istruzioni necessarie a passare i numerosi parametri di cui questo shader ha bisogno: le componenti della luce e le componenti del materiale dell'oggetto sono passate in maniera implicita da Wild Magic.

Il vertex shader calcola dunque l'illuminazione per vertice secondo il classico modello di Phong combinando le componenti della luce ed il materiale; al termine dei calcoli restituisce il colore risultante, oltre che le coordinate texture e le coordinate nello spazio dell'immagine per campionare la maschera di ombreggiatura: questa è l'unica aggiunta rispetto allo shader vertex shader della luce direzionale.

L'istruzione per calcolare queste coordinate è una semplice traslazione e scalamento della posizione del vertice trasformata nel *Clip Space*:

```
// Transform the position from model space to clip space.
kClipPosition = mul(float4(kModelPosition,1.0f),WVPMatrix);
// Compute the screen-space texture coordinates.
kScreenTCoord.x = 0.5f*(kClipPosition.x + kClipPosition.w);
kScreenTCoord.y = 0.5f*(kClipPosition.y + kClipPosition.w);
kScreenTCoord.z = kClipPosition.w;
kScreenTCoord.w = kClipPosition.w;
```

Il pixel shader riceve in ingresso, variabili per ogni pixel, le coordinate con cui campionare la texture dell'oggetto e la maschera di ombreggiatura. Come parametri costanti riceve la componente ambientale della luce e le informazioni sulla modalità di campionamento (i *Sampler*). Il colore finale è il risultato della moltiplicazione tra: il colore derivante dal calcolo della luce; il colore della texture; il coefficiente di ombreggiatura campionato nella maschera, incrementato di un offset pari alla componente ambientale della luce: in questa maniera si cerca di simulare il fatto che anche nelle zone in ombra, come nel mondo reale, c'è sempre un certo chiarore derivante dalle riflessioni ambientali.

6.7 Considerazioni finali sulle ombre dinamiche

La nostra implementazione di ombre dinamiche in tempo reale cerca di ridurre gli artefatti principali causati dalle shadow map tradizionali: l'acne e l'aliasing prospettico. Per fare questo abbiamo combinato due algoritmi: l'applicazione della sogliatura dinamica per evitare l'acne, il *Gradient Shadow Mapping* [12], e l'elaborazione nello spazio dell'immagine per ottenere bordi delle ombre più sfumati, il *Soft-Edged Shadows* [16].

Avendo come obiettivo la portabilità del codice, non abbiamo potuto realizzare anche il filtraggio PCF, perchè la nostra implementazione dello stesso funzionava soltanto su schede del produttore NVIDIA.

Questa realizzazione dimostra che si possono ottenere ombre dinamiche in tempo reale sfruttando gli shader cg nello *Shader Model 2.0* senza penalizzare troppo le prestazioni. Per migliorare ulteriormente la resa visiva delle ombre e ridurre maggiormente l'aliasing, una strada da percorrere potrebbe essere l'implementazione dell'algoritmo noto come *Light Space Perspective Shadow Maps*[12].

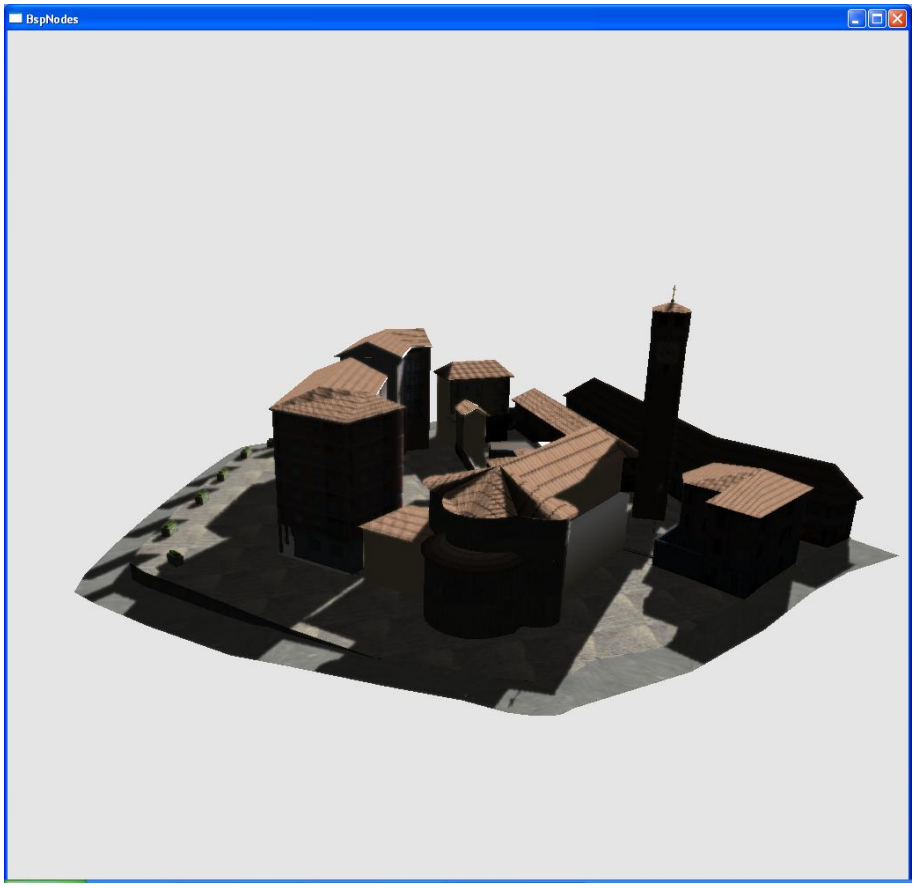


Figura 6.5. Una scena complessa illuminata con le ombre dinamiche in Wild Magic

Conclusioni e sviluppi futuri

Questa esperienza è stata molto gratificante dal punto di vista formativo, perchè ci ha consentito di conoscere in maniera approfondita la grafica tridimensionale, in particolare una tecnologia molto utilizzata ed in continua evoluzione: gli shader.

La documentazione in questo campo è ancora molto frammentata e piuttosto scarna; prendere confidenza con questa tecnologia è risultato più difficile del previsto. Presa coscienza di questa situazione abbiamo operato la scelta di concentrarci maggiormente sulla fase di ricerca.

L'idea iniziale infatti era realizzare effetti grafici, in particolare climatici, che migliorassero la resa visiva dell'ambiente UltraPort aumentandone il realismo. L'utilizzo degli shader nel nostro contesto, a causa dei requisiti tecnologici imposti da UltraPort e Wild Magic (come portabilità, assenza di precalcoli e velocità di esecuzione) è stata più complicata del previsto. Queste problematiche ci hanno consentito di acquisire una considerevole esperienza nel campo della programmazione ad oggetti e dell'informatica grafica in generale.

Il lavoro che abbiamo portato avanti fornisce gli strumenti per migliorare UltraPort aggiungendo effetti di illuminazione e di simulazione del clima; in particolare si potrebbe pensare a sviluppare un'applicazione che sfrutti queste potenzialità per visualizzare in 3D le previsioni meteo di tutti i modelli di città che già fanno parte del 4DGEA, il canale scientifico che riproduce il mondo reale sfruttando la tecnologia UltraPeg.

Gli algoritmi che abbiamo presentato e tutti gli effetti che si possono ottenere grazie alla tecnologia degli shader, possono trovare applicazione anche al di fuori del contesto per cui sono stati pensati e realizzati. Riteniamo di aver realizzato un sistema di calcolo di ombre dinamiche in tempo reale accessibile a tutti e piuttosto valido, in quanto non richiede eccessiva potenza di calcolo né hardware troppo sofisticato, pur avendo una buona resa visiva.

Appendice A

Listati

A.1 Pixel Shader _MT

Questi pixel shader fanno in modo che il renderer della scena effettui una sola passata in presenza di un effetto luce e di un effetto multitexture.

Per due textures sono disponibili sedici shader che combinano le quattro possibilità (Modulate, Hard Add, Soft Add, Decal) di unire l'effetto multitexture all'effetto luce con le quattro possibilità (sempre Modulate, Hard Add, Soft Add, Decal) di combinare fra loro le due texture. Esiste una quinta alternativa per entrambi i casi, il pixel shader di default del motore Wild Magic, che funziona in modalità Replace, ovvero non modifica la multitexture secondo l'influenza della luce, ma viene implementata a livello di motore grafico.

Per un numero di texture compreso fra tre e otto (numero massimo di texture supportate), le texture sono messe in modalità Modulate fra loro e fra loro stesse e la luce per motivi di semplicità ed efficienza.

Per motivi di spazio, oltre a quelli che combinano la luce con due texture, viene riportato solo lo shader che agisce su tre texture. Gli altri differiscono da quest'ultimo solo nel numero maggiore di coordinate texture.

```
// Due texture in modalità MODULATE  Luce e multitexture in modalità MODULATE
//-----------------------------------------------------------------------------
void p_T0s1d0T1s2d0_MT_MOD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);
```

```
    kPixelColor = kColor0*kColor1;
    kPixelColor *= kInPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità HARD ADD - Luce e multitexture in modalità MODULATE
//-----------------------------------------------------------------------------
void p_T0s1d0T1s1d1_MT_MOD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and add the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = saturate(kColor0 + kColor1);
    kPixelColor *= kInPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità SOFT ADD - Luce e multitexture in modalità MODULATE
//-----------------------------------------------------------------------------
void p_T0s1d0T1s3d1_MT_MOD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1.0f - kColor0)*kColor1 + kColor0;
    kPixelColor *= kInPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità DECAL - Luce e multitexture in modalità MODULATE
//-----------------------------------------------------------------------------
void p_T0s1d0T1s11d10_MT_MOD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
```

```
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1 - kColor1.a) * kColor0 + kColor1.a * kColor1;
    kPixelColor *= kInPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità MODULATE - Luce e multitexture in modalità HARD ADD
//-----------------------------------------------------------------------------
void p_TOs1d0T1s2d0_MT_HAD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = kColor0*kColor1;
    kPixelColor = saturate(kPixelColor + kInPixelColor);
}
//-----------------------------------------------------------------------------

// Due texture in modalità HARD ADD - Luce e multitexture in modalità HARD ADD
//-----------------------------------------------------------------------------
void p_TOs1d0T1s1d1_MT_HAD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and add the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = saturate(kColor0 + kColor1);
    kPixelColor = saturate(kPixelColor + kInPixelColor);
}
//-----------------------------------------------------------------------------

// Due texture in modalità SOFT ADD - Luce e multitexture in modalità HARD ADD
//-----------------------------------------------------------------------------
void p_TOs1d0T1s3d1_MT_HAD
```

```
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1.0f - kColor0)*kColor1 + kColor0;
    kPixelColor = saturate(kPixelColor + kInPixelColor);
}
//-----------------------------------------------------------------------------

// Due texture in modalità DECAL - Luce e multitexture in modalità HARD ADD
//-----------------------------------------------------------------------------
void p_T0s1d0T1s11d10_MT_HAD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1 - kColor1.a) * kColor0 + kColor1.a * kColor1;
    kPixelColor = saturate(kPixelColor + kInPixelColor);
}
//-----------------------------------------------------------------------------

// Due texture in modalità MODULATE - Luce e multitexture in modalità SOFT ADD
//-----------------------------------------------------------------------------
void p_T0s1d0T1s2d0_MT_SAD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = kColor0*kColor1;
    kPixelColor = (1.0f - kInPixelColor)*kPixelColor + kInPixelColor;
```

```
}
//-----------------------------------------------------------------------------

// Due texture in modalità HARD ADD - Luce e multitexture in modalità SOFT ADD
//-----------------------------------------------------------------------------
void p_T0s1d0T1s1d1_MT_SAD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and add the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = saturate(kColor0 + kColor1);
    kPixelColor = (1.0f - kInPixelColor)*kPixelColor + kInPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità SOFT ADD - Luce e multitexture in modalità SOFT ADD
//-----------------------------------------------------------------------------
void p_T0s1d0T1s3d1_MT_SAD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1.0f - kColor0)*kColor1 + kColor0;
    kPixelColor = (1.0f - kInPixelColor)*kPixelColor + kInPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità DECAL - Luce e multitexture in modalità SOFT ADD
//-----------------------------------------------------------------------------
void p_T0s1d0T1s11d10_MT_SAD
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
```

```
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1 - kColor1.a) * kColor0 + kColor1.a * kColor1;
    kPixelColor = (1.0f - kInPixelColor)*kPixelColor + kInPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità MODULATE - Luce e multitexture in modalità DECAL
//-----------------------------------------------------------------------------
void p_T0s1d0T1s2d0_MT_DEC
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = kColor0*kColor1;
    kPixelColor = (1 - kPixelColor.a) * kInPixelColor + kPixelColor.a * kPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità HARD ADD - Luce e multitexture in modalità DECAL
//-----------------------------------------------------------------------------
void p_T0s1d0T1s1d1_MT_DEC
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and add the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = saturate(kColor0 + kColor1);
    kPixelColor = (1 - kPixelColor.a) * kInPixelColor + kPixelColor.a * kPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità SOFT ADD - Luce e multitexture in modalità DECAL
//-----------------------------------------------------------------------------
void p_T0s1d0T1s3d1_MT_DEC
(
    in float4         kInPixelColor : COLOR,
```

```
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1.0f - kColor0)*kColor1 + kColor0;
    kPixelColor = (1 - kPixelColor.a) * kInPixelColor + kPixelColor.a * kPixelColor;
}
//-----------------------------------------------------------------------------

// Due texture in modalità DECAL - Luce e multitexture in modalità DECAL
//-----------------------------------------------------------------------------
void p_T0s1d0T1s11d10_MT_DEC
(
    in float4         kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);

    kPixelColor = (1 - kColor1.a) * kColor0 + kColor1.a * kColor1;
    kPixelColor = (1 - kPixelColor.a) * kInPixelColor + kPixelColor.a * kPixelColor;
}
//-----------------------------------------------------------------------------

// Pixel shader per tre texture
//-----------------------------------------------------------------------------
void p_T0s1d0T1s2d0T2s2d0_MT
(
    in float4       kInPixelColor : COLOR,
    in float2         kTCoord0 : TEXCOORD0,
    in float2         kTCoord1 : TEXCOORD1,
    in float2         kTCoord2 : TEXCOORD2,
    out float4        kPixelColor : COLOR,
    uniform sampler2D Sampler0,
    uniform sampler2D Sampler1,
    uniform sampler2D Sampler2)
{
    // Sample the texture images and multiply the results.
    float4 kColor0 = tex2D(Sampler0,kTCoord0);
    float4 kColor1 = tex2D(Sampler1,kTCoord1);
    float4 kColor2 = tex2D(Sampler2,kTCoord2);
    kPixelColor = kColor0*kColor1*kColor2*kInPixelColor;
}
```

Appendice B

Le ombre dinamiche

In questa sezione alleghiamo gli shader ed alcune porzioni di codice sorgente relative alla nostra implementazione delle ombre dinamiche.

B.1 Shadow Map semplici

```
//-----------------------------------------------------------------------------
void v_ProjectedDepth
(
    in float3        kModelPosition : POSITION,
    out float4       kClipPosition : POSITION,
    out float        fDepth : TEXCOORD0,
    uniform float4x4 WVPMatrix)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(float4(kModelPosition,1.0f),WVPMatrix);

    // Save the normalized distance from the light source.
    fDepth = kClipPosition.z/kClipPosition.w;
}
//-----------------------------------------------------------------------------
void p_ProjectedDepth
(
    in float   fDepth : TEXCOORD0,
    out float4 kPixelColor : COLOR)
{
    kPixelColor.rgb = fDepth;
    kPixelColor.a = 1.0f;
}
//-----------------------------------------------------------------------------
void v_ShadowMap
(
    in float3        kModelPosition : POSITION,
    out float4       kClipPosition : POSITION,
    out float4       kProjectedTCoord : TEXCOORD0,
    uniform float4x4 WVPMatrix,
    uniform float4x4 ProjectorMatrix)
```

```
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(float4(kModelPosition,1.0f),WVPMatrix);

    // Compute the projected texture coordinates.
    kProjectedTCoord = mul(float4(kModelPosition,1.0f),ProjectorMatrix);
}
//-----------------------------------------------------------------------------
void p_ShadowMap
(
    in float4         kProjectedTCoord : TEXCOORD0,
    out float4        kPixelColor : COLOR,
    uniform float     DepthBias,
    uniform sampler2D ColorSampler,
    uniform sampler2D DepthSampler)
{
    float4 kColor = tex2Dproj(ColorSampler,kProjectedTCoord);
    float4 kDepth = tex2Dproj(DepthSampler,kProjectedTCoord);

    float fPointDepth = kProjectedTCoord.z/kProjectedTCoord.w;
    float fBiasedDiff = fPointDepth - kDepth.r - DepthBias;
    if (fBiasedDiff >= 0.0f)
    {
        kPixelColor.rgb = 0.0f;
        kPixelColor.a = kColor.a;
    }
    else
    {
        kPixelColor = kColor;
    }
}
//-----------------------------------------------------------------------------
```

B.2 Shadow Map avanzate

Di seguito riportiamo la classe nodo *Shadower* che abbiamo realizzato; per maggiore chiarezza la sua interfaccia è scomposta in più parti, ognuna relativa ad un singolo passo di rendering.

Mappa di profondità:

```
class UM_Shadower: public Node
{
public:
        UM_Shadower(void);
        ~UM_Shadower(void);

        //Setup and update methods

        int InitFBO(Renderer *pkRenderer, int iScreenWidth, int iScreenHeight);
        void InitLight(Vector3f &rkDir,
        ColorRGB &rkAmbient, ColorRGB &rkDiffuse, ColorRGB &rkSpecular,
        float fIntensity);
        void OnChangeLight(Vector3f &rkDir,
```

```
        ColorRGB &rkAmbient, ColorRGB &rkDiffuse, ColorRGB &rkSpecular,
        float fIntensity);
        void OnChangeViewPort(Renderer *pkRenderer,
        int iScreenWidth, int iScreenHeight);
        float* GetIntensity();
        void SetAmbient(ColorRGB &rkCol);
        void SetDiffuse(ColorRGB &rkCol);
        void SetSpecular(ColorRGB &rkCol);
        //Draw steps method
        int ApplyShadows(Renderer *pkRenderer, Camera *pkCamera);

protected:
        // Members that describe the state of the node
        CameraPtr       m_spLightCamera;
        LightPtr        m_spLight;
        Culler          m_kLightCuller;

        // shader constants
        Matrix4f m_kLightVPMatrix;
        Matrix4f m_kLightVPBSMatrix;

        int m_iShadowSize;              //Dimension of shadowmap ( in pixel )
        float m_fTexelSize;             //1/shadowsize

        // shadow objects
        ImagePtr m_spkShadowImage;
        TexturePtr m_spkShadowTarget;
        FrameBuffer* m_pkShadowFBO;
        ShaderEffectPtr m_spkShadowEffect;
};
```

Maschera di ombreggiatura:

```
class UM_Shadower: public Node
{
public:
        UM_Shadower(void);
        ~UM_Shadower(void);
        /* Acne reduction coeffs */
        float m_fDepthBias;
        float m_fGradientClamp;
        float m_fShadowGradientScaleBias;
        float m_fShadowFuzzyWidth;

protected:
        // unlit objects
        ImagePtr        m_spkUnlitImage;
        TexturePtr      m_spkUnlitTexture;
        FrameBuffer*    m_pkUnlitFBO;
        ShaderEffectPtr m_spkUnlitEffect;
};
```

Filtraggio gaussiano:

```
class UM_Shadower: public Node
{
protected:
        /* Blur weights */
        enum { NUM_WEIGHTS = 15 };
        Vector3f m_akHWeights[NUM_WEIGHTS];
        Vector3f m_akVWeights[NUM_WEIGHTS];
        Vector2f m_akHOffsets[NUM_WEIGHTS];
        Vector2f m_akVOffsets[NUM_WEIGHTS];
        // Gaussian blur objects
        CameraPtr m_spkScreenCamera;
        TriMeshPtr m_spkScreenPolygon;
        ImagePtr m_spkHBlurImage;
        TexturePtr m_spkHBlurTarget;
        FrameBuffer* m_pkHBlurFBO;
        ShaderEffectPtr m_spkHBlurEffect;
        ShaderEffectPtr m_spkVBlurEffect;

        /* Gaussian filtering */
        Vector3f GetGaussianDistribution( float x, float y, float rho );
        void GetGaussianOffsets( Vector2f vViewportTexelSize );
        /* Creates the orthographic camera and the screen-sized polygon for the blur step */
        void CreateBlurObjects ();
};
```

Riportiamo ora gli shader necessari per realizzare l'algoritmo:

```
//-----------------------------------------------------------------------------
void v_Unlit
(
    in float3 kModelPosition : POSITION,
    out float4 kClipPosition : POSITION,
    out float4 kProjTCoord : TEXCOORD0,
    out float fDepth : TEXCOORD1,
    uniform float4x4 WVPMatrix,
    uniform float4x4 WMatrix,
    uniform float4x4 LightVPMatrix,
    uniform float4x4 LightVPBSMatrix
)
{
    // Transform the position from model space to clip space.
    float4 kHModelPosition = float4(kModelPosition,1.0f);
    kClipPosition = mul(kHModelPosition,WVPMatrix);

    // Transform the position from model space to light space.
    float4 kWorldPosition = mul(kHModelPosition,WMatrix);
```

```
    float4 kLSPosition = mul(kWorldPosition,LightVPMatrix);

    // Compute the projected texture coordinates.
    kProjTCoord = mul(kWorldPosition,LightVPBSMatrix);

    // Output the distance from the light source.
    fDepth = kLSPosition.z;
}
//-----------------------------------------------------------------------------
//GRADIENT SHADOW MAPPING implementation SHADER x4 p. 293
void p_Unlit
(
    in float4 kProjTCoord : TEXCOORD0,
    in float fDepth : TEXCOORD1,
    out float4 kPixelColor : COLOR,
    uniform float DepthBias,
    uniform float TexelSize,  // 1.0f/textureWidth (square texture)
    uniform float GradientClamp,
    uniform float ShadowGradientScaleBias,
    uniform float ShadowFuzzyWidth,
    uniform sampler2D ShadowSampler
)
{
    // Generate the texture coordinates for the specified depth-map size.
        float4 akTexCoords[9];
        akTexCoords[0] = kProjTCoord;

        //Profondità del pixel, dal punto di vista della luce,
        //cui viene sottratto il Bias.
        float fDepthBiased = fDepth - DepthBias;
        // Pixel offset per la valutazione del gradiente nell'intorno del pixel
        float pixeloffset = TexelSize / 2;

        // Fattore di visibilità tra [0 1]. E' il valore che ritorniamo
        float visibility = 0.0f;
        // Differenza tra la profondità del pixel
        // e la profondità di quel punto in SM, opportunamente scalato.
        float delta_z;
        // E' la differenza massima tra i valori di profondità
        // dei campioni intorno al pixel, in verticale ed orizzontale.
        float gradient;
        float2 differences;     // Variabile "di appoggio" per il calcolo di gradient
        float4 depths;  // Contiene i 4 campioni di profondità intorno al pixel.
        float2 shadowP; // Le coordinate xy del pixel sulla shadow map
        float centerdepth;      // La profondità del pixel in quel punto

        // Valuta il fattore di visibilità del singolo pixel
        // Trova le coordinate del pixel sulla shadow map
        shadowP = akTexCoords[cnt].xy / akTexCoords[cnt].w;
        // Valuta la profondità
        centerdepth = tex2D( ShadowSampler, shadowP ).r;
        //Valuta la shadow map nell'intorno del punto,
        //  ( sopra sotto destra e sinistra )
        depths = float4(
```

```
                tex2D( ShadowSampler, shadowP + float2( -pixeloffset, 0 ) ).x,
                tex2D( ShadowSampler, shadowP + float2( +pixeloffset, 0 ) ).x,
                tex2D( ShadowSampler, shadowP + float2( 0, -pixeloffset ) ).x,
                tex2D( ShadowSampler, shadowP + float2( 0, +pixeloffset ) ).x );
        differences = abs( depths.yw - depths.xz );
        //Calcola il gradiente: il Clamp serve solo per limitare l'aumento
        // del gradiente, bisogna tararlo in modo che non sogli troppo.
        gradient = min( GradientClamp, max( differences.x, differences.y ) );
        delta_z = centerdepth + gradient * ShadowGradientScaleBias - fDepthBiased;
        // Funzione di visibilità: calcola quanto è in luce il pixel.
        visibility += saturate( 1 + delta_z / ( gradient * ShadowFuzzyWidth ) );

        kPixelColor.rgb = visibility;
        kPixelColor.a = 1.0f;
}
//-----------------------------------------------------------------------------
void v_Blur
(
    in float3 kModelPosition : POSITION,
    in float2 kModelTCoord : TEXCOORD0,
    out float4 kClipPosition : POSITION,
    out float2 kTCoord : TEXCOORD0,
    uniform float4x4 WVPMatrix
)
{
    // Transform the position from model space to clip space.
    kClipPosition = mul(float4(kModelPosition,1.0f),WVPMatrix);

    // Pass through the texture coordinates.
    kTCoord = kModelTCoord;
}
//-----------------------------------------------------------------------------
void p_HorizontalBlur
(
    in float2 kTCoord : TEXCOORD0,
    out float4 kPixelColor : COLOR,
    uniform float3 Weights[15],
    uniform float2 Offsets[15],
    uniform sampler2D BaseSampler
)
{
    kPixelColor = 0.0f;
    for (int i = 0; i < 15; i++)
    {
        kPixelColor.rgb +=
            Weights[i]*tex2D(BaseSampler, kTCoord + Offsets[i]).rgb;
    }
    kPixelColor.a = 1.0f;
}
```

Bibliografia

[1] Fondazione Ultramundum
url:http://www.ultramundum.org

[2] David H. Eberly, *3D Game Engine Design*, Morgan Kaufmann Publishers, Second Edition.

[3] David H. Eberly, *3D Game Engine Architecture*, Morgan Kaufmann Publishers.

[4] Teoria su Vertex e Pixel shader url: http://www.csc.calpoly.edu/ zwood/ teaching/csc471/finalproj24/jsoldo/
url: http://www.toymaker.info/Games/html/shaders.html#in
url: http://www.gamedev.net/columns/hardcore/dxshader1/
url: http://it.wikipedia.org/wiki/Shader
url: http://vincedx.altervista.org/Lessons.php?id=89&from=0
url: http://sisinflab.poliba.it/coppi/graphics/shaders4.pdf

[5] Associated Legendre Functions, Wikipedia,
url: *http://en.wikipedia.org/wiki/Associated_Legendre_functions*

[6] Siggraph 2003, Course 44 *Monte Carlo Ray Tracing* disponibile all' url: *http://www.cs.rutgers.edu/˜decarlo/readings/mcrt-sg03c.pdf*

[7] Carsten Wächter e Alexander Keller, *Instant Ray Tracing: The Bounding Interval Hierarchy* disponibile all' url: *http://graphics.uni-ulm.de/BIH.pdf*

[8] Robin Green, *Spherical Harmonic Lighting: The Gritty Details*
url: http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf

[9] Emmanuel Viale e Kelly Dempski, *Advanced Lighting and Materials with Shaders*, Wordware Publishing, Inc.

[10] Dutré, Bekaert, Bala, *Advanced Global Illumination*, A.K. Peters, Ltd. 2001

[11] Reeves, W. T., Salesin, D. H., and Cook, R. L. 1987. *Rendering antialiased shadows with depth maps. In Computer Graphics (Proceedings of SIGGRAPH 87)*, vol. 21.

[12] AA. VV. *Shader X4 Advanced Rendering Tecniques*

[13] AA. VV. *Shader X5 Advanced Rendering Tecniques*

[14] Stamminger and Drettakis, 2002 *Perspective Shadow Maps* http://www-sop.inria.fr/reves/publications/data/2002/SD02/?LANG=gb

[15] Michael Wimmer, Daniel Scherzer, Werner Purgathofer, 2005 *Light Space Perspective Shadow Maps*

[16] Anirudh.S Shastry, *Soft Edged shadows*,
url: http://www.gamedev.net/reference/articles/article2193.asp