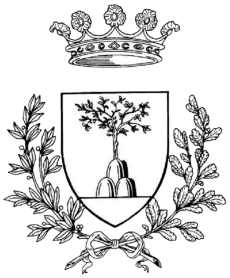
UNIVERSITÀ DEGLI STUDI DI FERRARA



Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Relazione progetto di tirocinio per la Laurea Triennale in Informatica

Tecniche di visualizzazione avanzata in tempo reale per lo sviluppo di applicazioni nell'ambito dei Beni Culturali

Candidato:

Michele Pio Mischitelli

Proponente:

Dott. Paolo Cignoni

Tutore Accademico:

Dott. Stefano Marchetti

Tutore Aziendale:

Ing. Massimiliano Corsini

Anno Accademico 2006/07

Alla mia famiglia.

Indice

Capitolo 1

Introduzione

La presente relazione documenta il lavoro svolto durante il progetto di tirocinio per il conseguimento della laurea triennale.

Il tirocinio si è svolto presso il *Visual Computing Lab*, un laboratorio di ricerca dell'ISTI, l'*Istituto di scienza e tecnologie dell'informazione "A. Faedo"*, appartenente al CNR di Pisa. Lo sviluppo del progetto e la stesura della presente relazione si sono protratti dal 15 Ottobre al 21 Dicembre 2007.

1.1 Il progetto

Il progetto si compone di tre parti:

- Realizzazione di un plug-in per il software opensource *MeshLab* che consenta di calcolare l'Ambient Occlusion di una qualsiasi figura poligonale caricata.

- Studio sugli shaders, piccoli programmi eseguiti in hardware dalla GPU, con successiva realizzazione di effetti real-time per la simulazione fotorealistica di marmo e granito.

- Implementazione delle *Translucent Shadow Maps* (TSM), una tecnica realizzata tramite shader, per la simulazione di materiali translucidi in tempo reale.

Questi tre progetti, solo apparentemente scollegati, concorrono in maniera diversa al raggiungimento di uno scopo comune, che è poi l'obiettivo di questo lavoro di tirocinio: l'ottenimento di un maggiore realismo visivo nella visualizzazione interattiva di *mesh*[1].

Si pensi ad esempio ad una statua realizzata in marmo; se si decidesse di digitalizzarla tramite particolari scanner 3D, si otterrebbe un modello ben dettagliato nella sua geometria, ma completamente privo di qualsiasi dettaglio relativo al materiale con cui è stato realizzato dallo scultore. Volendo visualizzare con un certo grado di realismo questo modello su un'applicazione grafica, il passo immediatamente successivo può essere quello di applicarvi sopra un immagine che simuli il marmo e le sue venature. A questo punto il modello presenta una buona resa visiva del materiale ma manca di profondità in quanto, seppur illuminato da una sorgente luminosa, manca dell'attenuazione della luce stessa a causa della geometria dell'oggetto. Per tale motivo, si considera l'illuminazione del modello tridimensionale da più punti di vista, memorizzando l'effetto delle ombre risultanti ed applicandole successivamente in modo statico. Grazie alla combinazione di queste due tecniche si ha un modello che comunica una sensazione di profondità e che visivamente riesce a dare anche l'impressione di essere fatto di marmo. Tuttavia, per raggiungere un grado di fotorealismo elevato manca ancora qualcosa: se non fosse per l'immagine del marmo applicatavi sopra, sarebbe difficile intuire che la statua è fatta proprio di marmo. Questo perché il marmo è un materiale caratterizzato dal fatto che parte della luce che lo colpisce vi penetra all'interno, esegue dei rimbalzi a causa delle venature ed esce da un'altra direzione.

[1] Mesh e modello tridimensionale sono da intendersi come sinonimi.

Il fenomeno prende il nome di *Sub-Surface Scattering* ed è comune a tutti i materiali translucidi, come ad esempio la pelle, alcuni tipi di pietre, il latte, *et cetera*. Pertanto il terzo passo per ottenere un'elevata resa fotorealistica della statua di partenza, consiste nell'implementazione di una tecnica che simuli in tempo reale la translucentezza del materiale.

Realizzare una scena che presenti i dettagli grafici specificati poco sopra, su di un Personal Computer corrente, di fascia medio-alta (CPU dual core, 2GB di Ram), richiederebbe dai 10 ai 15 minuti circa di elaborazione con algoritmi detti di *raytracing*. Grazie al raytracing è possibile simulare con l'accuratezza di complessi modelli matematici l'ottica di un intera scena, producendo il più elevato livello qualitativo possibile.

Tuttavia, a seguito degli avanzamenti nel campo delle schede grafiche, la sfida si è spostata dalla ricerca di una migliore resa visiva alla possibilità di effettuare un numero sufficientemente elevato di elaborazioni, tale per cui sia possibile riprodurre a schermo una sequenza animata in modo fluido. Per raggiungere tale traguardo è necessario effettuare 30 o più elaborazioni nell'arco di un solo secondo; per rendere meglio l'idea, se con una tecnica raytracing tradizionale si è parlato di circa 13 minuti per calcolare l'intera scena, oggigiorno si punta ad effettuare il tutto in *meno di 30 millisecondi*.

Nonostante le GPU siano diventate particolarmente efficienti e prestanti nel campo dell'elaborazione floating point, è in ogni caso impensabile poter effettuare le stesse identiche operazioni del raytracing in soli 30 millisecondi. Bisogna, quindi, introdurre approssimazioni e semplificazioni che rendano l'intero processo calcolabile in tempo reale.

Grazie a tali accorgimenti è stato possibile realizzare i tre sotto-progetti esaminati nella presente relazione: non solo la qualità visiva risultante è particolarmente elevata, ma anche la velocità con cui tali effetti vengono calcolati permette di ottenere la visualizzazione interattiva di modelli caratterizzati da materiali complessi, come appunto nell'esempio di cui sopra, il marmo.

Riassumendo le tecniche utilizzate nell'esempio precedente e collegandole agli algoritmi real-time implementati durante il tirocinio, è possibile schematizzare quanto segue:

- L'attenuazione dell'illuminazione, a causa della geometria dell'oggetto, viene calcolata con una tecnica chiamata *Ambient Occlusion*.
- Si genera l'immagine del marmo in maniera totalmente dinamica sfruttando il *Perlin Noise* all'interno di uno shader.
- La simulazione del Sub-Surface Scattering si ottiene con l'implementazione tramite shaders di una moderna tecnica denominata *Translucent Shadow Map* TSM.

La prima parte è stata sviluppata in C++ utilizzando diverse tecnologie di supporto, quali ad esempio la libreria *VCGlib*, creata dal Visual Computing Lab stesso, la *OpenGL Extension Wrangler Library* e la versione opensource del framework *QT*® di *Trolltech*®.

VCGlib è una libreria templetizzata C++ portabile e opensource per la manipolazione, l'elaborazione ed il disegno in OpenGL di complessi simpliciali.

La OpenGL Extension Wrangler Library (*GLEW*) permette di gestire in modo semplice tutte le estensioni sviluppate dagli IHV (*Indipendent Hardware Vendors*) che ampliano e completano le funzionalità dell'OpenGL standard.

QT® è un framework di sviluppo C++ che, attraverso la *QT class library*, offre funzionalità per il disegno di componenti grafici, manipolazione di stringhe, I/O, parsing XML, e molto altro. Nella realizzazione della prima parte del tirocinio tale framework ha rivestito un ruolo chiave, permettendo di realizzare un plug-in in maniera estremamente semplice, eliminando l'overhead che normalmente si ha in questi casi.

Il plug-in, facendo parte di un progetto multipiattaforma, è stato realizzato tenendo conto di varie possibili combinazioni hardware/software; come analizzato in seguito, riuscire a realizzare una compatibilità così ampia richiede diverse configurazioni hardware su cui testare il codice. D'altro canto, proprio affrontando queste problematiche, si è in grado di apprendere una gran quantità di informazioni e, conseguentemente, di esperienza che portano ad una crescita culturale, seppur settoriale, del programmatore stesso.

Per quanto riguarda la seconda e la terza parte, si è ricorsi al linguaggio GLSL che consente di scrivere codice simil-C eseguito completamente in hardware dalle schede grafiche che lo supportano. Il vantaggio rispetto all'OpenGL standard è l'estrema libertà con cui è possibile manipolare i singoli vertici e pixel di una scena tridimensionale: ciò si concretizza riprogrammando in modo custom la pipeline standard di OpenGL con uno o più *shader*, ossia piccoli programmi scritti appunto in GLSL, che specificano a basso livello *come* trattare i dati di una scena durante la visualizzazione.

Per la scrittura di codice GLSL è possibile sfruttare un qualsiasi editor testuale: del resto, come per ogni linguaggio di programmazione, si sente la necessità di un ambiente di sviluppo che assista il programmatore nella scrittura, ad esempio segnalando la sintassi con colori appropriati, suggerendo il prototipo di una certa funzione, facendo debug, *et cetera*. Per tale progetto si è deciso di utilizzare l'ambiente *RenderMonkey*, sviluppato dalla canadese ATI (ora di proprietà di AMD) in collaborazione con 3Dlabs.

Il lavoro è stato svolto seguendo le direttive del Proponente e del Tutore Accademico, che, pur seguendo i progressi con costante attenzione e rispondendo tempestivamente ad ogni dubbio, hanno lasciato ampi margini di decisione personale, sia a livello di design che implementativo.

1.2 Struttura della relazione

Nel prossimo capitolo saranno esposte dettagliatamente le tecnologie utilizzate: il C++, OpenGL, XML, la libreria VCG, il framework QT, il GLSL, l'ambiente di sviluppo per shaders RenderMonkey, ed infine qualche breve accenno alle GPU.

Nel terzo capitolo si descriverà nel dettaglio il plug-in per il calcolo dell'Ambient Occlusion realizzato, le varie tecniche implementate e le possibilità offerte dal plug-in stesso.

Il quarto capitolo è dedicato ad una panoramica generale sugli shaders, unitamente alla descrizione dell'algoritmo procedurale con cui sono stati realizzati pattern per la resa visiva di materiali simili al marmo ed al granito.

L'elaborazione della translucentezza mediante la tecnica delle Translucent Shadow Maps verrà analizzata invece nel quinto capitolo.

Infine, l'ultimo capitolo offrirà un riepilogo degli aspetti principali del progetto, analizzando in particolare il risultato dell'unione dei tre sotto progetti.

Capitolo 2

Tecnologie utilizzate

In questo capitolo vengono analizzate le tecnologie utilizzate nello sviluppo del progetto.

La maggior parte di tali tecnologie sono state già state utilizzate per un piccolo progetto relativo al corso di *Tecniche Avanzate per la Grafica*, tenuto nell'anno accademico 2006/07 dal dott. Massimiliano Corsini presso l'Università di Ferrara. L'unica eccezione è data dalle librerie QT di Trolltech, mai usate prima del tirocinio.

2.1 C++

Il *C++* è un linguaggio di programmazione orientato agli oggetti, con tipizzazione statica. Ideato da Bjarne Stroustrup nel 1983, esso è un'evoluzione del linguaggio *C*.

Rispetto al C, il C++ offre funzionalità quali le classi, le funzioni virtuali, l'overloading degli operatori, l'ereditarietà multipla, i template e la gestione delle eccezioni.

La grande ricchezza semantica del C++ lo rende un linguaggio estre-

mamente espressivo e potente. Purtroppo questo linguaggio richiede molto tempo per venire appreso e padroneggiato completamente.

Il C++ ha una libreria standard. Di particolare importanza in essa è la *Standard Template Library* o *STL*, la parte che utilizza i template per implementare contenitori generici, come vettori, code, array associativi e altre funzionalità.

Per ulteriori informazioni su questo linguaggio è possibile consultare [1, 2].

2.2 XML

L'*eXtensible Markup Language* come il nome stesso dice, è un linguaggio markup generale. Esso viene definito estensibile, perché permette agli utenti di definire i propri tags. Lo scopo primario dell'XML è facilitare la condivisione dei dati attraverso sistemi diversi, in particolar modo attraverso internet. L'XML è un sottoinsieme semplificato di *SGML*, lo *Standard Generalized Markup Language*, ed è progettato per essere relativamente leggibile dall'uomo.

Aggiungendo vincoli semantici, linguaggi applicativi possono essere implementati in XML; esempi ne sono i linguaggi *XHTML*, *RSS*, *MathML*, *GraphML*, *Scalable Vector Graphics (SVG)*, *MusicXML* e migliaia di altri.

XML è uno standard libero ed aperto, ed è una raccomandazione del *W3C*, il *World Wide Web Consortium*.

Per ulteriori approfondimenti su questa tecnologia si veda [3].

2.3 Qt

Qt è un framework multipiattaforma per lo sviluppo di programm, principalmente dotati di interfaccia utente grafica, sempre più usato in ambito aziendale. Esempi di sistemi largamente diffusi che usano Qt sono: *KDE*,

Opera, *Google Earth* e *Skype*. Qt usa una fase di metacompilazione in cui alcuni comandi aggiuntivi al C++ standard sono preprocessati e trasformati in codice C++ puro. Questo permette al framework di realizzare operazioni molto flessibili e potenti dal punto di vista del programmatore. Le librerie di cui Qt dispone non si limitano alla gestione dell'interfaccia utente, contengono anche classi per la gestione di stringhe, I/O, multi-threading, connessioni di rete, containers generici, SQL, e molto altro; contengono inoltre un parser XML, seppur privo di validatore.

La documentazione è disponibile in [4].

2.4 VCGlib

VCGlib è una libreria templetizzata C++ portabile ed opensource per la manipolazione, l'elaborazione ed il disegno in OpenGL di complessi simpliciali, in particolare *mesh triangolari*[1]. Questa libreria è la base di tutti gli strumenti software sviluppati nel Visual Computing Lab. Essa è composta da più di cinquantamila righe di codice ed include molti algoritmi per l'elaborazione di mesh, ad esempio: navigazione efficiente nei punti, negli spigoli e nelle facce, gestione automatica della memoria occupata, pulizia, rimozione di rumore di campionamento[2], approssimazione di superfici irregolari e molto altro. Il progetto sfrutta pesantemente le risorse offerte da questa libreria.

Per ulteriori informazioni si rimanda a [5].

[1] Le mesh triangolari sono complessi simpliciali di ordine 2.

[2] Di solito le mesh vengono acquisite utilizzando dispositivi chiamati *3D scanners*, che introducono errori di campionatura.

2.5 GPU

La *Graphics Processing Unit* è un componente hardware dedicato che si trova negli attuali elaboratori sotto forma di una scheda di espansione dedicata o di un chip integrato sulla scheda madre. Le moderne GPU sono particolarmente adatte a compiere elaborazioni legate alla pipeline di rendering e, grazie alla loro natura altamente parallela, sono estremamente efficienti nell'esecuzione di una certa categoria di algoritmi come ad esempio l'elaborazione delle immagini.

Al di là dell'utilizzo nell'ambito dello sviluppo di videogiochi, le GPU stanno ricoprendo un ruolo sempre maggiore nell'ambito del calcolo scientifico, in cui la loro capacità di processare efficacemente una gran quantità (o *stream*) di dati *floating point*, le pone su un'altro livello prestazionale rispetto anche ai più potenti processori attualmente in commercio.

Da poco, infatti, i due maggiori produttori di GPU, *AMD* e *nVidia*, vista la crescente domanda da parte delle compagnie di ricerca, hanno addirittura creato linee di prodotti hardware/software interamente dedicati a questo tipo particolare di applicazioni. In questo caso si parla di soluzioni GPGPU, ossia General-Purpose GPU.

Nonostante la loro specializzazione a svolgere determinate operazioni, le GPU sino a poco fa erano estremamente limitate per una serie di motivi, fra cui l'impossibilità di eseguire correttamente operazioni sugli interi (fondamentali per l'implementazione di un algoritmo *RNG - Random Number Generator*) e la mancanza di una pipeline interna che garantisse operazioni di filtraggio e fusione (blending) tra valori a virgola mobile a singola precisione (*Floating Point a 32bit*). Con le ultime generazioni di GPU (*AMD R600* e *nVidia G80*) gran parte di questi problemi sono stati risolti anche se la mancanza di una pipeline a doppia precisione si continua a far sentire in particolari ambiti: tale limite verrà rimosso probabilmente con le future

versioni delle architetture grafiche.

Per ulteriori informazioni si rimanda a [6].

2.6 OpenGL

OpenGL è un'interfaccia software verso il sistema grafico hardware, sviluppata da Silicon Graphics Inc nel 1992. Tale libreria nasce dalla necessità di visualizzare immagini ad alta qualità che rappresentassero oggetti e ambienti realistici a partire da informazioni di tipo geometrico e ottico.

Questa interfaccia è composta da circa 150 comandi distinti il cui scopo è quello di specificare oggetti geometrici in due o tre dimensioni, definendone le coordinate spaziali e specificandone gli attributi quali le proprietà ottiche del materiale, creare una gerarchia di dipendenze tra di essi e impostare il punto di vista. Queste operazioni sono utilizzate per sviluppare applicazioni interattive che fanno uso di grafica tridimensionale.

OpenGL è concepita come un'interfaccia indipendente dall'hardware. Per raggiungere questa indipendenza, non vengono incluse alcune operazioni come quelle per l'utilizzo di interfacce grafiche a finestra, che sono specifiche del Sistema Operativo, o per il controllo dei dispositivi di input.

In maniera analoga, OpenGL non fornisce comandi ad alto livello per descrivere modelli di oggetti tridimensionali. I comandi forniti permettono di specificare oggetti dall'aspetto relativamente complesso tramite l'uso di un ristretto insieme di primitive geometriche quali punti, linee e poligoni.

Il modo in cui esse vengono visualizzate si basa sull'esistenza di un *framebuffer*, un'area di memoria nella quale è possibile memorizzare l'immagine prodotta. Il suo utilizzo si limita soltanto alla lettura e scrittura di questa porzione di memoria, non esiste alcun'altra periferica hardware con la quale si può interagire.

La seguente lista descrive sommariamente le operazioni grafiche principali che OpenGL compie per disegnare un'immagine sullo schermo:

- Costruzione di forme a partire da primitive geometriche, creando una descrizione matematica degli oggetti[3].
- Disposizione degli oggetti in uno spazio tridimensionale e selezione del punto di vista.
- Calcolo del colore di tutti gli oggetti. Il colore può essere esplicitamente assegnato dall'applicazione, determinato da condizioni di illuminazione specificate, ottenuto applicando una *texture*[4] sugli oggetti, oppure con una qualsiasi combinazione di queste tre azioni.
- Conversione della descrizione matematica degli oggetti e delle informazioni sul colore ad esse associate in pixel sullo schermo. Questo processo è chiamato *rasterizzazione*.

Nel corso di questi stadi, OpenGL può compiere altre operazioni, come l'eliminazione di parti di oggetti che sono nascoste da altri oggetti. La maggior parte delle implementazioni OpenGL seguono lo stesso ordine di operazioni, una serie di stadi di elaborazione chiamata *rendering pipeline*.

Tramite le trasformazioni è possibile traslare, ruotare e applicare operazioni di scalatura ad ogni oggetto tridimensionale. La chiave nell'utilizzo delle trasformazioni risiede non nel trasformare l'oggetto stesso ma nell'agire sul sistema di riferimento in cui questo è stato inizialmente specificato. In OpenGL e, più in generale, in un qualsiasi sistema grafico, esistono diversi tipi di trasformazioni che vengono applicate dal momento della specifica dei vertici alla visualizzazione su schermo:

[3]OpenGL considera punti, linee, poligoni, immagini e bitmap come primitive.

[4]Una texture, o tessitura, è un'immagine bidimensionale applicata alla superficie di un oggetto tridimensionale.

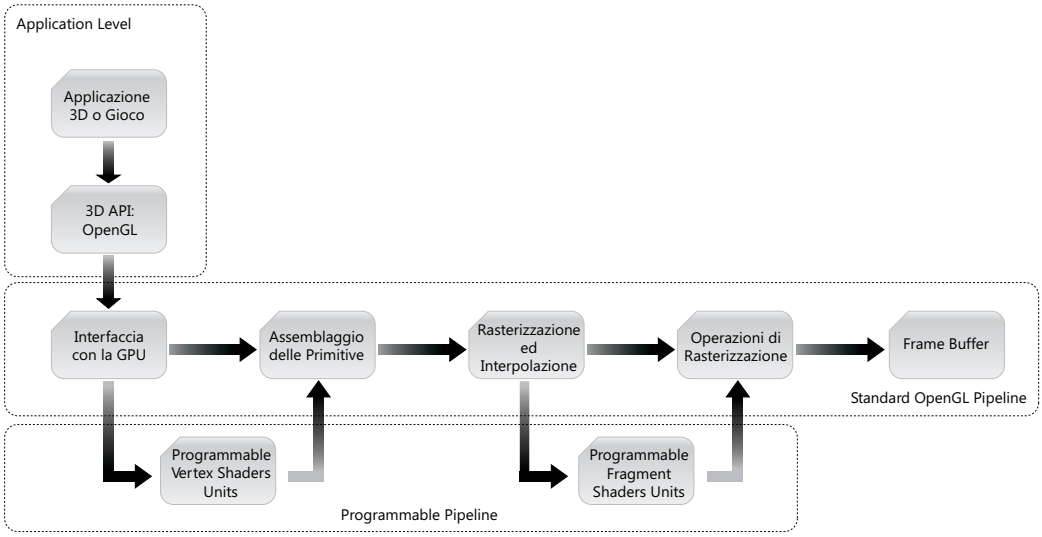


Figura 2.1: *Una rappresentazione schematica della pipeline fissa OpenGL*

- **Viewing**: specifica della posizione e orientamento dell'osservatore.
- **World**: posizionamento dell'oggetto nello spazio.
- **Projection**: definizione del volume di visuale.
- **Viewport**: trasformazione finale per l'output su schermo.

Nel corso degli anni, diverse nuove caratteristiche sono state aggiunte nell'OpenGL nelle versioni 1.1, 1.2, 1.3, 1.4 e 1.5 ma l'architettura base della pipeline è rimasta sostanzialmente invariata. Si usa il termine di *pipeline fissa* o *pipeline standard* perché ogni implementazione delle OpenGL è costretta ad avere la stessa funzionalità e produrre gli stessi risultati, in maniera consistente con le specifiche OpenGL, per una determinata serie di input. In questo caso sia le operazioni che l'ordine con cui vengono eseguite sono definite a priori dalle specifiche.

Il trend attuale, comunque, è quello di fornire la possibilitá di rimpiazzare la pipeline fissa con parti programmabili tramite semplici programmi (detti

Shader) che permettono di manipolare i singoli vertici ed i frammenti di una scena OpenGL. Sui vertici, in genere, si applicano tramite Shader trasformazioni e calcoli utili all'illuminazione. I frammenti sono delle strutture create dal rasterizzatore per contenere informazioni sui singoli pixel della scena: in un unico frammento ci sono tutte le informazioni di colore e di profonditá necessarie ad aggiornare una singola locazione nel frame buffer. Per tale motivo, le operazioni effettuate tramite Shader sui singoli frammenti hanno un campo di applicazione così vasto che risulta impossibile immaginare dei limiti entro cui definire tali operazioni.

Per ulteriori informazioni su OpenGL si rimanda a [7, 8].

2.7 GLSL

L'*OpenGL Shading Language* (GLSL) costituisce un linguaggio di programmazione di alto livello con cui scrivere *Shader* che verranno eseguiti dalla *Graphics Processing Unit* al posto della pipeline fissa che caratterizza l'OpenGL standard. Con il termine OpenGL Shading Language si distingue un particolare linguaggio, che una volta compilato e mandato in esecuzione, viene processato interamente dalla GPU invece che dalla CPU. Dal momento che esistono in OpenGL due tipi fondamentali di unità programmabili, ci sono a loro volta due diverse tipologie di Shader: i *Vertex Shader* ed i *Fragment Shader*.

L'*OpenGL Shading Language* nasce con l'intento di semplificare lo sviluppo di shaders. Prima del GLSL, infatti, gli Shaders venivano scritti direttamente in codice Assembly. Successivamente, l'*OpenGL Architecture Review Board* decise di creare un linguaggio ad alto livello, facile da programmare e che permettesse ai programmatori di effetti grafici di poter sfruttare efficacemente l'hardware: nacque così il GLSL, chiamato anche GLslang. Le caratteristiche principali di questo linguaggio sono:

- Il GLSL è un linguaggio procedurale di alto livello.
- Stessa sintassi e semantica, con alcune piccole eccezioni, per i vertex e fragment shader.
- Basato sulla sintassi del C/C++.
- Supporta in maniera nativa operazioni sui vettori e sulle matrici, in quanto spesso utilizzate nelle operazioni grafiche.
- Più rigoroso del C per quanto concerne i tipi di dati.
- Invece di usare meccanismi di Read/Write, gli input e gli output sono gestiti da qualificatori di tipo.
- Non ci sono particolari restrizioni sulla lunghezza degli Shader.

Per ulteriori informazioni si rimanda a [9].

2.8 RenderMonkey

RenderMonkey è un ambiente di sviluppo realizzato da *AMD* in collaborazione con *3Dlabs* che consente di sviluppare rapidamente Shader sia in linguaggio GLSL che in *HLSL* (utilizzato in DirectX). RenderMonkey è stato creato per venire incontro alle esigenze dei programmatori fornendo strumenti per la creazione di Shader in un ambiente di sviluppo flessibile e potente, ma al tempo stesso facile da utilizzare ed intuitivo. Con l'introduzione di linguaggi di programmazione di alto livello, come ad esempio *DirectX*, *OpenGL* ed *OpenGL ES*, la complessità degli Shader in tempo reale è cresciuta in maniera esponenziale. L'ambiente di sviluppo RenderMonkey non solo permette di prototipizzare e sviluppare facilmente Shader, ma fornisce inoltre un meccanismo integrato per la gestione di tutte le risorse ad essi associate quali modelli 3D, immagini e parametri.

RenderMonkey permette vari livelli di gestione degli Shader, schematizzati come segue.

2.8.1 Sviluppo di Shader

Editor di Shader

Per ogni linguaggio supportato da RenderMonkey, è presente un particolare editor con rispettiva evidenziazione della sintassi. Ciascun editor fornisce al programmatore una integrazione degli errori di compilazione, associando ciascun errore ad una linea ben precisa del codice sorgente.

Modifica dei parametri di uno Shader

Tutti i parametri associati ad uno Shader possono essere modificati tramite una interfaccia utente intuitiva. I colori, ad esempio, possono essere modificati scegliendoli da una palette a forma di ruota; valori scalari e vettoriali possono essere modificati tramite l'ausilio di slider; le texture e gli oggetti che compongono la scena e le relative texture possono essere osservate tramite un'anteprima e molto altro.

2.8.2 Debug degli Shader

Anteprima interattiva con segnalazione di errori

La finestra in cui viene renderizzata la scena risponde immediatamente a qualsiasi cambio di parametri durante il rendering degli effetti. Ciò significa che modificando un qualsiasi parametro, istantaneamente il programmatore avrà un riscontro visuale della modifica, rendendo semplice la messa a punto degli effetti. Nella finestra di rendering, inoltre, ogni errore relativo a risorse mancanti (texture, variabili, et cetera) viene prontamente segnalato tramite

scritte in primo piano e rendendo la geometria della scena tutta di colore bianco.

Debug visivo di ciò che l'algoritmo sta effettivamente facendo

Fare il debug degli Shader è un processo molto complesso in quanto non ci sono riscontri numerici ma soltanto visuali. Nel caso di effetti complessi che richiedono più passate di rendering per essere completati, RenderMonkey consente di reindirizzare l'output di ciascuna passata ad una porzione della finestra di rendering: così facendo si è in grado di vedere in ogni istante i singoli output e quindi capire dove è situato l'errore.

2.8.3 Gestione delle risorse XML

RenderMonkey salva tutte le impostazioni e i dati necessari relativi allo Shader sviluppato in un file di lavoro *XML*. Una volta caricato tale file in RenderMonkey, sul lato sinistro dell'interfaccia compare una schematizzazione ad albero di ciascun nodo del file XML, ognuno identificante di una specifica risorsa. Data la natura portabile e chiara del formato XML, è possibile integrare questi file di lavoro in altri programmi, come ad esempio *MeshLab*, rendendo immediato il porting da questo ambiente di sviluppo alle applicazioni grafiche. In altre parole, rendendo quindi gli Shader prototipati immediatamente utilizzabili dalle applicazioni.

Per ulteriori informazioni si rimanda a [10].

Capitolo 3

Ambient Occlusion

L'Ambient Occlusion è una tecnica utilizzata in Computer Grafica che aiuta ad aumentare il realismo dei modelli di illuminazione locale, tenendo conto dell'attenuazione della luce dovuta alla visibilità che una determinata porzione di superficie possiede. A differenza dei modelli di illuminazione locale come il Phong, l'Ambient Occlusion è un modello globale: questo significa che l'illuminazione di ogni punto di una superficie è in funzione della geometria dell'intera scena. Ad ogni modo, esso è solamente un approssimazione di vero e proprio modello di illuminazione globale. L'apparenza morbida dovuta all'applicazione esclusiva dell'Ambient Occlusion è similare a quanto osservabile su di un oggetto in una giornata nuvolosa.

L'Ambient Occlusion è spesso sinonimo di *accessibility shading*, una tecnica che, sfruttando l'accessibilità di un particolare punto rispetto all'ambiente circostante, determina l'apparenza della superficie in funzione di vari elementi (quali sporco, luce, fenomeni atmosferici, *et cetera*). Questa tecnica è divenuta uno degli effetti cinematografici più sfruttati in quanto è particolarmente semplice ma nel contempo efficace. Nell'industria, ci si riferisce spesso all'Ambient Occlusion con il termine di *sky light*.

Il modello descritto dall'Ambient Occlusion possiede l'interessante pro-

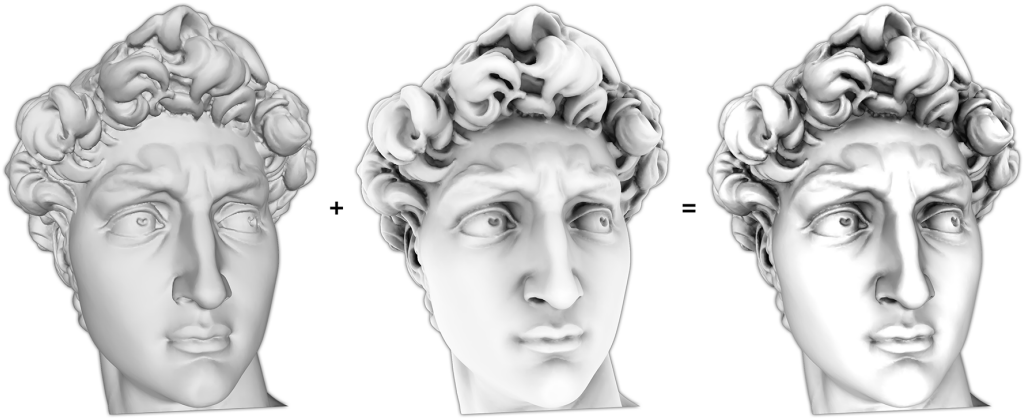


Figura 3.1: *Da sinistra: modello di illuminazione locale; Ambient Occlusion; locale con Ambient Occlusion*

prietà di restituire una migliorata percezione di profondità dell'oggetto visualizzato. Da uno studio pubblicato nel 2000[11] basato su esperimenti di percezione, si evince come il discernimento della profondità di un oggetto illuminato da una sky light diffusa ed uniforme sia superiore a quello fornito da un modello di illuminazione diretto (Fig. 3.1).

3.1 Cenni sulle problematiche di calcolo

Per simulare il contributo ambientale su di un modello tridimensionale bisognerebbe tenere conto della visibilità dell'illuminazione in ogni punto della superficie, dato che l'oggetto è completamente immerso nell'ambiente e riceve illuminazione da ogni direzione. Com'è facilmente intuibile, un algoritmo che esegua tale tipo di simulazione risulterebbe particolarmente costoso, computazionalmente parlando.

Normalmente, la tecnica più utilizzata per il calcolo dell'Ambient Occlusion consiste nel tracciare diversi raggi in ogni direzione da ogni punto della

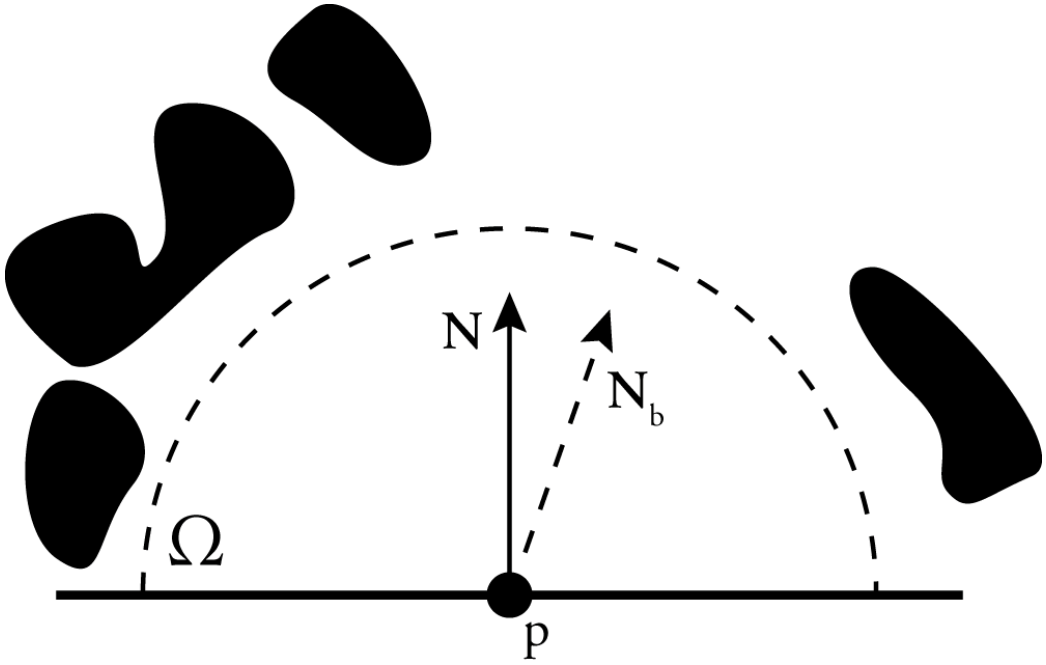


Figura 3.2: *Illustrazione dell'integrazione usata per determinare l'Ambient Occlusion*

superficie (Fig. 3.2). Ciascun raggio che raggiunge lo sfondo della scena aumenta la luminosità specifica del punto da cui esso è partito; al contrario, ogni raggio che collide con un qualsiasi altro punto dell'oggetto non fornisce alcun aumento di illuminazione. Come risultato di tale procedimento, i punti circondati da molta geometria saranno rappresentati da un colore scuro, mentre i punti che hanno il proprio emisfero libero da qualsiasi ostruzione risulteranno più illuminati.

Integrare un numero elevato di raggi per ciascun punto della superficie permette, sicuramente, di ottenere un risultato qualitativamente elevato seppur a costo di altrettanto elevati tempi di elaborazione. Per tale motivo, questa tecnica viene usata principalmente in software per il *Raytracing offline*, quali ad esempio *Cinema4D* e *MaYa*.

Scopo della presente relazione è, invece, l'elaborazione dell'Ambient Oc-

clusion in *real-time*, il che porta ad escludere questa tipologia di elaborazione.

Individuare una tecnica adatta allo scopo ci porta ad effettuare, sostanzialmente, una divisione dei possibili algoritmi in due principali categorie:

- Algoritmi Statici
- Algoritmi Dinamici

Ciascuna delle due famiglie presenta pregi e difetti: in generale si può comunque dire che gli algoritmi statici godono di maggiore precisione ma richiedono tempi di elaborazione relativamente lunghi, che ne impediscono attualmente il calcolo ad ogni frame. D'altro canto gli algoritmi dinamici sfruttano l'elevata potenza elaborativa delle GPU accoppiata ad approssimazioni che, a scapito di una minore precisione, riducono nel contempo il carico di lavoro. Grazie a tali accorgimenti si in grado di aggiornare le informazioni sull'occlusion ad ogni frame, rendendo possibile l'interazione dinamica con la geometria mantenendo la coerenza con il contributo ambientale.

L'algoritmo ivi discusso appartiene alla categoria statica, a cui è stato aggiunta la possibilità di sfruttare la GPU in modo da ridurre (in alcuni casi fino ad un ordine di grandezza) il tempo necessario per l'elaborazione.

La tecnica mira a calcolare il contributo ambientale sui singoli vertici della mesh ed scomponibile in tre principali fasi:

- Si osservano i vertici da V punti di vista disposti uniformemente su una sfera immaginaria che li contiene.
- Per ciascuna vista si effettua una *Occlusion Query* per determinare l'insieme dei vertici nascosti.
- Accumulando ciascuna Occlusion Query si ottiene una *Occlusion Map* della mesh. La mappa così generata dovrà essere rapportata al numero totale di viste, ottenendo una normalizzazione della stessa tra zero ed

uno. Negando tale valore, i vertici maggiormente esposti riceveranno un valore prossimo ad uno che identifica l'intera quantita di luce, mentre valori prossimi allo zero non ne ricevono in quanto completamente occlusi.

Dalle caratteristiche elencate si evince come sia richiesto un numero sufficientemente elevato di punti di vista per ottenere un buon risultato (attualmente viene impostato come valore di default 256 viste), ma grazie all'hardware moderno tutto il processo risulta sicuramente più veloce del classico metodo raytraced. Tuttavia, altri fattori, come analizzato in seguito, concorrono ad un buon risultato finale.

Si descrivono di seguito le tre fasi dell'algoritmo.

3.2 Prima Fase - Generazione delle viste

Come anticipato in precedenza, per osservare la mesh ci avvaliamo di una sfera immaginaria di punti disposti in modo tale che ciascun vertice sia contenuto al suo interno. Caricata la mesh da file, si genera il cosiddetto *Bounding Box*, un cuboide che contiene l'oggetto. In simulazioni dinamiche, i bounding box sono preferibili ad altri tipi di forme (come ad esempio una sfera o un cilindro) quando si effettuano di test di intersezione non particolarmente accurati o quando c'è semplicemente la necessità di calcolare il punto più lontano dal centro della mesh.

La VCGlib mette a disposizione un metodo per ottenere le due coordinate che identificano rispettivamente l'angolo maggiore e minore del bounding box. Identificando con b_{max} il massimo tra i valori assoluti così ottenuti, è possibile utilizzare la distanza che separa il centro della mesh con b_{max} per calcolare il raggio r di una sfera immaginaria su cui verranno posizionati i k punti di vista, garantendo inoltre che l'intera mesh sia contenuta in essa.

Una volta individuata la sfera bisogna distribuire uniformemente le viste su di essa. Una corretta disposizione è fondamentale per ottenere una buona qualità finale della scena. Si potrebbe pensare di utilizzare le coordinate polari per disporre le sorgenti luminose in punti equidistanti lungo la latidutine e la longitudine, ma ai poli della sfera si tenderebbe ad accumulare troppe viste rispetto alla zona equatoriale. Per questo motivo si è ricorsi alla VCGlib in quanto implementa già un metodo che permette di generare k coordinate disposte uniformemente su di una sfera di raggio unitario. Moltiplicando ciascuna coordinata così ottenuta per il raggio r della sfera calcolato in precedenza, con l'aggiunta di una piccola quantità per evitare che la camera possa *toccare* in alcuni punti la mesh, si ottengono le viste da cui, in seguito, verranno effettuare le diverse inquadrature.

Posizionando la camera su ciascun punto v_i, si procede al rendering nel cosiddetto *Depth Buffer*, o *Z-Buffer*.

3.2.1 Rendering nello Z-Buffer

In computer grafica, lo Z-Buffering identifica la gestione della profondità di un immagine tridimensionale, generalmente fatta in hardware. Quando un oggetto viene renderizzato dalla scheda grafica, la profondità di ciascun pixel (coordinata z) viene salvata in questo particolare buffer (Fig. 3.3). Il depth buffer viene solitamente rappresentato da un array bidimensionale (x–y) con ciascun elemento corrispondente ad un pixel dello schermo. Se un altro oggetto deve essere renderizzato nello stesso punto, la scheda grafica effettuerà un confronto tra la posizione sull'asse z di ciascun oggetto e sceglierà quello più vicino all'osservatore. La coordinata z relativa a tale oggetto verrà salvata nel depth buffer, sovrascrivendo il vecchio valore (nel caso di una sostituzione). Al termine dell'algoritmo, nel buffer sarà presente una mappa dei vertici più vicini alla camera: tale mappa prende il nome di *Depth Map*.

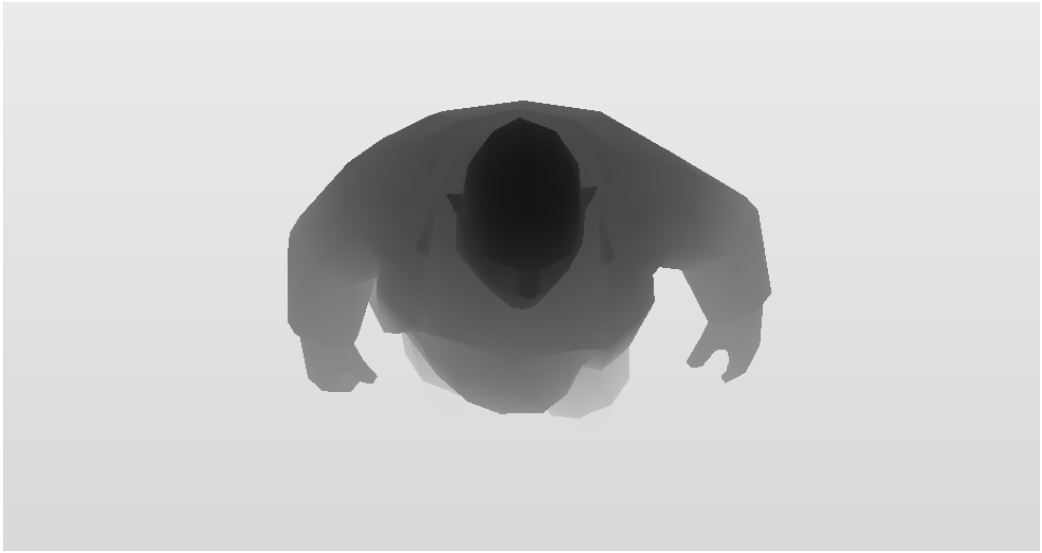


Figura 3.3: *Come appare una Depth Map se visualizzata su schermo*

La *granularità* dello Z-Buffer è determinante ai fini di una buona qualità finale della scena: usando soltanto 16-bit per tale buffer si rischia di avere artefatti visivi (fenomeno che prende il nome di *Z-Fighting*) quando due oggetti sono molto vicini l'uno con l'altro. Aumentando la precisione del buffer a 24-bit o 32-bit si risolve quasi completamente il problema, anche se ulteriori accorgimenti sono necessari per eliminarlo del tutto.

Per il calcolo dell'Ambient Occlusion si è scelto di usare una precisione di 24-bit, questo perché non tutte le schede video supportano correttamente i 32-bit per lo Z-Buffer. Tuttavia, come accennato poco sopra, bisogna utilizzare altri accorgimenti per prevenire artefatti che influenzerebbero la qualità finale. Questo problema sarà affrontato successivamente.

3.3 Seconda Fase - Occlusion Query

Creata la Depth Map, si procede con le Occlusion Query. Per Occlusion Query si intende quel particolare meccanismo in base a cui un applicazione riesce a ottenere il numero di pixel (o meglio, frammenti) disegnati da una primitiva od un gruppo di primitive.

Lo scopo principale di tale algoritmo è quello di determinare la visibilità di un oggetto. Normalmente un applicazione renderizza dapprima tutti gli oggetti che compongono la scena; successivamente si effettua un confronto sul bounding box di ciascun oggetto rispetto a quello degli altri. In questo modo si in grado di determinare la profondità e quindi, automaticamente, la visibilità (gli oggetti più vicini hanno la coordinata z più piccola). Lo stesso può essere fatto per ciascun frammento della scena: in OpenGL per frammento si intende un pixel potenziale, che, per via dell'algoritmo dello Z-Buffer, non detto venga renderizzato.

Per tale motivo, nella seconda fase dell'Ambient Occlusion, si testa la visibilità di ciascun vertice rispetto alla vista v_i. In precedenza è stato effettuato il rendering nello Z-Buffer: ciò comporta che ogni singolo vertice è stato proiettato prima in *Light Space* e quindi scritto sulla Depth Map. In OpenGL ogni oggetto, prima di essere visualizzato, necessita di essere proiettato in uno spazio comune a tutti gli altri. A tale risultato si arriva moltiplicando ogni vertice dell'oggetto per una matrice, chiamata *Model-View*: il modello si dirà quindi *proiettato in View Space* proprio perché ciascun vertice ha subito una trasformazione spaziale che lo pone in un sistema di coordinate differente da quello di origine. Nel caso specifico in cui la posizione della luce di una scena coincida con la posizione della camera, l'oggetto proiettato si dirà essere in *Light Space*. Tale convenzione risulta comoda quando si lavora con differenti sistemi di riferimento, riuscendo subito ad individuare il sistema in cui si sta lavorando.

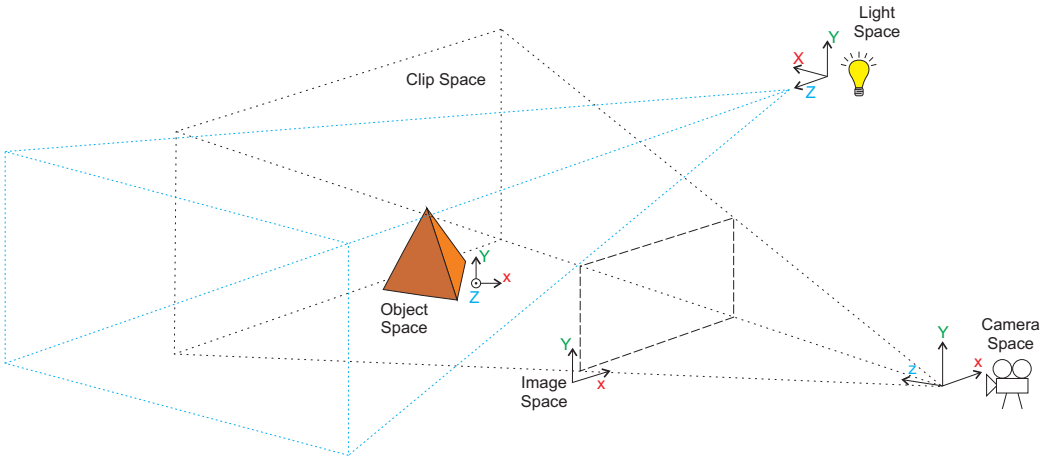


Figura 3.4: *Schematizzazione grafica del camera (o view) space e del light space*

Le trasformazioni Model-View vengono eseguite al volo, prima di visualizzare la scena: ciò comporta che la struttura dati presente in memoria non viene in alcun modo modificata. Quando si esegue una Occlusion Query, la lettura delle coordinate di un qualsiasi vertice ritorna, quindi, la posizione di quel particolare vertice rispetto all'oggetto di cui esso fa parte. Dato che lo scopo dell'algoritmo è quello di verificare l'insieme dei vertici visibili, risulta necessario proiettare ciascun vertice in Light Space, ponendolo automaticamente nello stesso spazio della Depth Map.

A questo punto testare la visibilità di ciascun vertice risulta immediata: è sufficiente utilizzare le coordinate (x_p, y_p) proiettate in Light Space per eseguire un *Texture Lookup*[1] ed ottenere la coordinata z_d presente nella Depth Map. Dato che in quest'ultima sono memorizzate unicamente le informazioni di visibilità relative ai vertici più vicini alla camera, basta effettuare un semplice confronto per determinarne la visibilità:

$$z_p < z_d \tag{3.1}$$

[1]Per Texture Lookup ci si riferisce alla lettura di uno specifico elemento (chiamato *Texel*) da una texture

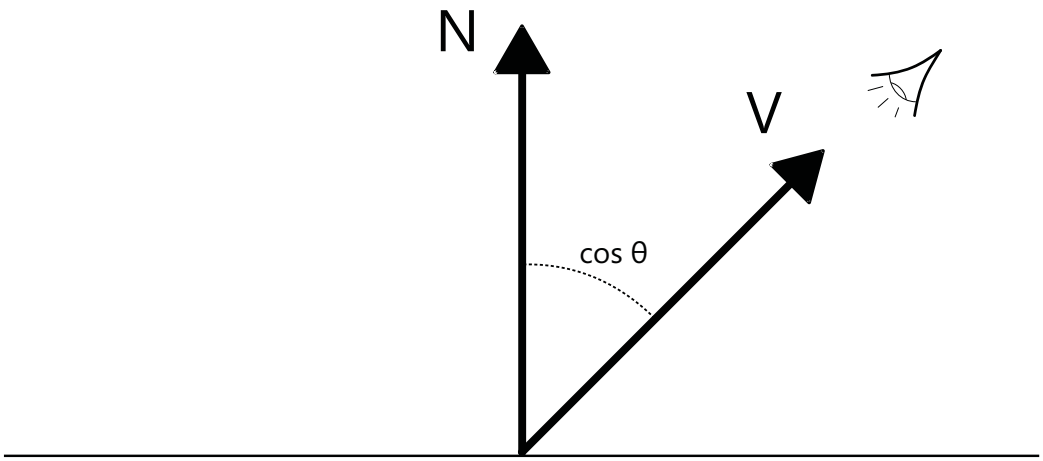


Figura 3.5: *Prodotto scalare tra la normale N e la vista V*

Se la (3.1) è vera, il vertice proiettato risulterà occluso.

3.3.1 Ridurre gli artefatti nelle Occlusion Query

Come anticipato durante la generazione delle viste, bisogna tenere conto di alcuni problemi che possono nascere utilizzando le Depth Map senza particolari accorgimenti. Nel caso specifico affrontato durante le Occlusion Query, effettuare dei Texture Lookup lungo i bordi della mesh ed in corrispondenza di rientranze o sporgenze della superficie può portare ad artefatti sotto forma di striature.

Ciò avviene perché in corrispondenza di tali zone sono presenti più vertici con coordinate (x–y) simili ma con z differente. Effettuare un Texture Lookup in tali punti, a causa di problemi di precisione numerica, spesso porta ad assegnare la stessa profondità a vertici che in realtà sono diversi.

Al fine di ridurre il problema è necessario tenere conto dell'inclinazione del vertice rispetto al punto di vista: questo si ottiene eseguendo un prodotto scalare tra il vettore di vista incidente la superficie e la normale in quel particolare punto (Fig. 3.5). La normale rappresenta un vettore unitario

perpendicolare alla superficie su cui è poggiato, mentre il vettore di vista altri non è che la coordinata normalizzata del punto di vista v_i. Nella geometria Euclidea, il prodotto scalare, la lunghezza e gli angoli sono tutti strettamente legati tra di loro: ad esempio, per un vettore a, eseguire il prodotto scalare $a \cdot a$ equivale al quadrato della sua lunghezza, e, più genericamente, se b è un'altro vettore:

$$a \cdot b = \mid a \mid\mid b \mid \cos(\theta)$$

Dato che il coseno di 90^o è zero, il prodotto scalare di due vettori perpendicolari è sempre zero. Inoltre, se a e b sono vettori unitari, il prodotto scalare ritorna semplicemente il coseno dell'angolo tra di loro.

Per queste proprietà matematiche, effettuare il prodotto scalare tra la normale ed il vettore di vista è utile per pesare il contributo di occlusione: questo concetto è stato descritto nella legge del coseno di Lambert, che afferma che l'intesità luminosa è direttamente proporzionale all'angolo θ tra la direzione di vista e la normale.

3.4 Terza Fase - Utilizzare la Occlusion Map

Analizzate in dettaglio le problematiche di generazione delle viste e delle interrogazioni sulla visibilità dei vertici, resta da affrontare ancora *come* sfruttare tali dati in modo da ottenere l'Ambient Occlusion del modello.

3.4.1 Accumulazione delle Occlusion Query

Se eseguendo una Occlusion Query relativa alla vista v_i si ottiene la Occlusion Map di tale vista, per ottenere la Occlusion Map di tutto il modello è necessario eseguire le Occlusion Query da tutti i punti $\{v_0, v_1, \ldots, v_n\}$ generati durante la prima fase dell'algoritmo. A tale scopo è sufficiente disporre

di una struttura temporanea (texture, array, *et cetera*) su cui *accumulare* i risultati di ogni singola query.

Per accumulazione si intende una delle più semplici espressioni utilizzate nella programmazione, ovvero:

$$x = x + k$$

dove x indica l'elemento su cui si sta accumulando e k la quantità da accumulare.

Nell'algoritmo presentato, l'accumulazione viene eseguita sommando di volta in volta il risultato delle Occlusion Query: in questo modo, una volta utilizzate tutte le viste da cui osservare il modello, si avrà a disposizione la Occlusion Map di tutti i vertici della mesh.

3.4.2 Conversione della Occlusion Map

La mappa così realizzata risulta ancora inutilizzabile, in quanto c'è necessità di tradurre le informazioni salvate su di essa in valori utilizzabili da una qualsiasi applicazione. Genericamente, un applicazione che presenti la possibilità di calcolare l'Ambient Occlusion, una volta chiamato il metodo deputato all'elaborazione, si aspetta come risultato una serie di valori, ciascuno rappresentante la visibilità del corrispettivo vertice: come già detto, quelli maggiormente esposti riceveranno un valore prossimo ad uno che identifica la massima visibilità da parte dell'ambiente circostante, mentre valori prossimi allo zero indicano vertici completamente occlusi. Dal momento che l'Ambient Occlusion è utilizzabile per diversi scopi, sarebbe conveniente salvare in una struttura la Occlusion Map, per potervi accedere in maniera diretta ogni qual volta se ne abbia la necessità.

In MeshLab, una volta caricato un modello tridimensionale, si assegna a ciascun vertice una serie di *campi* in cui inserire informazioni che ne descrivono diverse caratteristiche: queste possono essere di tipo geometrico, come

ad esempio la posizione del vertice e la sua normale, oppure di tipo visivo come il colore, la *qualità* ed altri parametri configurabili dinamicamente in fase di caricamento della mesh stessa. La Occlusion Map viene memorizzata nel parametro *qualità* in quanto genericamente non utilizzato.

Al fine di ottenere un output visivo, si è resa necessaria la conversione della mappa in modo che ciascun valore rappresenti un colore: in MeshLab, ciascun vertice possiede un campo in cui è possibile specificare il colore tramite un valore ad 8-bit per canale, fino ad un massimo di quattro canali (rosso, verde, blu e trasparenza). La mappatura da Occlusion Map a Color Map viene realizzata in una serie di passi:

- Si cerca il massimo ed il minimo valore all'interno dell'Occlusion Map.
- Successivamente si calcola un valore di scala S tale per cui il massimo dell'Occlusion Map corrisponda a $2^8 - 1$ secondo la formula:

  $$S = \frac{2^8 - 1}{\max - \min}$$

  Dove max e min identificano rispettivamente il massimo ed il minimo valore della Occlusion Map.
- Si scrive in ciascuno dei tre canali rosso, verde e blu il valore convertito con la seguente formula:

  $$C_{rgb} = (\mathrm{OM}_i - \min) * S$$

  Dove OM indica l'insieme dei valori della Occlusion Map.

È interessante notare come ciascun canale, facente parte dello stesso vertice, possiede identico valore: per tale motivo l'output visivo sarà in scala di grigi. A questo punto è possibile salvare la mesh con il contributo ambientale memorizzato in essa. Di seguito è possibile osservare una comparativa dello stesso modello tridimensionale, renderizzato in MeshLab, con un modello di

illuminazione locale (Fig.3.6) e successivamente con l'aggiunta del contributo ambientale calcolato dal plug-in descritto (Fig.3.7).

3.5 Ottimizzazioni implementate

Quanto descritto in precedenza rappresenta la sequenza di operazioni eseguite prevalentemente sulla CPU, se si esclude la generazione della Depth Map. Questo metodo di calcolo dell'Ambient Occlusion non presenta limiti di sorta e consente di pre-calcolare, su un PC consumer di media potenza, in circa un minuto il contributo ambientale di un modello composto da diverse centinaia di migliaia di poligoni. L'operazione di pre-calcolo si effettua *una tantum*, per poi venire applicata staticamente sulla mesh durante il rendering interattivo. L'impatto prestazionale dovuto al rendering con Ambient Occlusion è praticamente nullo.

Volendo, pertanto, migliorare le performance dell'algoritmo ivi descritto bisogna cercare di ridurre la durata dell'operazione di pre-calcolo. Attualmente il costo computazionale dell'algoritmo è pari a $O(2n \cdot k)$, dove n rappresenta il numero di vertici e k il numero delle viste. Si è scritto $2n$ in quanto per ogni k-esima vista sono necessarie due passate di rendering dei vertici: la prima serve per ottenere il Depth Buffer, la seconda per effettuare le Occlusion Query.

Tuttavia è bene notare come l'utilizzo del Depth Buffer sia già di per sé un ottimizzazione: esso permette infatti di ridurre considerevolmente i tempi di calcolo dell'Occlusion Query calcolata in modo standard (ossia lanciando k raggi da ogni vertice e testando l'intersezione con la geometria circostante).

Dal momento che la Depth Map viene generata dalla scheda grafica e che quest'ultima dispone di hardware specializzato per l'esecuzione di tale operazione, è ragionevole ritenere che il tempo impiegato dalla GPU per renderizzare su schermo una qualsiasi geometria, non può essere paragonato

Figura 3.6: *Testa del David illuminata con un modello di illuminazione locale (in questo caso, Phong)*

Figura 3.7: *Testa del David illuminata tramite il modello di Phong in combinazione con l'Ambient Occlusion*

al tempo impiegato dalla CPU per scorrere un array dimensionato in base al numero dei vertici della geometria precedentemente considerata.

Una volta generata la Depth Map, infatti, bisogna in qualche modo leggerla dalla scheda grafica per poter effettuare i confronti tramite CPU. L'OpenGL mette a disposizione una primitiva, la `glReadPixels()` che consente di leggere il contenuto del *framebuffer*[2]; una volta impostato l'OpenGL in modo che tale framebuffer contenga esclusivamente le informazioni di profondità della scena, effettuando una lettura su di esso si è in grado di ottenere la Depth Map. Leggere da framebuffer ha ovviamente un costo, che d'ora in poi quantificheremo con μ.

Si può quindi riassumere il costo della generazione di una Depth Map con $O(\varepsilon + \mu)$. Computazionalmente parlando, μ risulta un operazione particolarmente costosa, che riduce in parte il vantaggio prestazionale inizialmente immaginato: bisognerebbe quindi cercare di ridurre al minimo le letture da framebuffer. Ciò è reso possibile da due tecnologie forniteci dall'OpenGL che verranno analizzate nella prossima sezione: i *Framebuffer Objects* e gli *Shaders*.

Dato che non tutte le schede grafiche attualmente disponibili forniscono le tecnologie necessarie per calcolare l'Ambient Occlusion utilizzando interamente la GPU (*modalità hardware*), si è scelto di non sostituire completamente l'algoritmo per il calcolo tramite processore (detto anche *in modalità software*). Nell'applicativo finale l'utente ha quindi la possibilità all'utente di selezionare o meno l'accelerazione hardware tramite un interfaccia grafica del plug-in stesso.

[2]Il framebuffer è la porzione di memoria video in cui viene disegnata la scena OpenGL al termine delle operazioni di rasterizzazione prima che essa venga spedita allo schermo. Per maggiori informazioni consultare [12].

3.5.1 Utilizzo della GPU per accelerare i calcoli

Come anticipato poco sopra, il problema che attualmente presenta l'algoritmo è quello di dover leggere, per ogni vista, il contenuto del framebuffer della scheda in modo da poter effettuare i confronti necessari per stabilire la visibilità dei vertici. Il problema non si porrebbe se fosse possibile spostare tutta la fase delle Occlusion Query direttamente sulla scheda grafica. Tale possibilità avrebbe un importante implicazione: fino a pochi anni fa, le schede grafiche avevano l'unico compito di velocizzare il rendering della scena, possibilmente effettuando anche qualche operazione di filtraggio in modo da rendere gli oggetti più definiti, e poco altro. La possibilità, invece, di eseguire una serie di operazioni definite dal programmatore in maniera arbitraria, rende di fatto la GPU *programmabile*.

La programmabilità di una GPU, in ambiente OpenGL, è stata resa disponibile con la versione 2.0 delle librerie grafiche. Essa definisce una serie di primitive, unitamente ad un supporto hardware adeguato ed un linguaggio con cui programmare tale hardware. Dal momento che la GPU resta pur sempre un chip specializzato, per sfruttare adeguatamente le risorse hardware a disposizione è importante separare le operazioni da eseguire su processori *General Purpose* da quelle maggiormente adatte ad essere trattate sotto forma di *stream* e quindi adatte ad essere eseguite su GPU[3].

La programmazione di una GPU richiede la realizzazione di due programmi relativi alle due fasi principali: nel **vertex shader** si manipolano i vertici che costituiscono la geometria sorgente, mentre nel **fragment shader** vengono eseguite manipolazioni sui singoli frammenti della scena. Al termine di queste due fasi, i frammenti, seppur manipolati, verranno scritti sempre

[3]Questo è in parte non corretto, in quanto ultimamente stanno sorgendo diversi tool di sviluppo che mirano ad unificare la programmazione di GPU e comuni processori identificando ciascuno di essi come una generica risorsa per l'elaborazione. Per maggiori informazioni si veda [6]

in corrispondenza della loro posizione elaborata dal vertex shader. I vertici ed i frammenti vengono elaborati in base a degli algoritmi specificati dal programmatore, avvalendosi di eventuali ulteriori input: ad uno shader è possibile fornire come parametri degli scalari, vettori o texture che saranno disponibili durante tutta l'esecuzione dello stesso.

La programmazione di uno shader avviene mediante un linguaggio denominato OpenGL Shading Language (*GLSL*) che, una volta compilato, viene tradotto in una serie di istruzioni di basso livello, in tutto simile all'Assembly delle architetture *x86*. Lo shader così strutturato viene caricato in una porzione di memoria della scheda video, insieme a tutti i parametri specificati in precedenza e resta in attesa di essere eseguito.

Se durante l'esecuzione di codice OpenGL si attiva uno shader, parte della pipeline di rendering viene sostituita da quest'ultimo. Il programma di shading si prenderà dunque carico di ogni aspetto del rendering, dalle trasformazioni dei vertici alla colorazione dei frammenti.

Nel caso dell'Ambient Occlusion, la programmabilità dell'hardware rende possibile demandare alla scheda grafica tutto il calcolo delle Occlusion Query, insieme dell'accumulazione dei risultati. Questo è un grande vantaggio, che si traduce in tempi di esecuzione drasticamente ridotti come riassunto alla fine del capitolo in una tabella.

Tuttavia, anche nella versione hardware c'è il problema di leggere lo Z-Buffer. Mentre nella versione software non si poteva fare altro che leggere tutti i pixel, uno ad uno e salvarli in una struttura temporanea, nella versione hardware si hanno a disposizione varie strade per evitare tutto il processo di *readback*.
Dato che il problema principale sta nel leggere dei dati, bisognerebbe trovare una struttura dati che sia scrivibile e leggibile direttamente dalla scheda video; altro requisito fondamentale è che tale struttura risieda completamente nella memoria video, evitando di dover ricorrere al lento bus di comunicazione

che collega la GPU con la Ram di sistema. La soluzione a tale problema è utilizzare una *texture*.

Una texture altro non è che una matrice bidimensionale di valori numerici, chiamati *texel*, ciascuno dei quali identifica un colore. Nelle texture i colori vengono solitamente rappresentati nello spazio RGB, ossia i colori sono rappresentati tramite le rispettive componenti di rosso, verde e blu. In particolare, le texture possono essere a tre o quattro canali per colore. In genere il quarto canale identifica la trasparenza, mentre i primi tre le componenti di colore. Le texture a tre canali vengono semplicemente chiamate *Textures RGB* (o BGR, in base all'ordine con cui vengono interpretati i canali), mentre quelle a quattro si dicono *Textures RGBA* (o BGRA). Le schede video sono particolarmente adatte a trattare dati sotto forma di textures, in quanto dispongono di circuiteria delegata esclusivamente a questo scopo.
Per ottimizzare l'accesso alle texture si ricorre al *Render To Texture*. In questa particolare modalità le operazioni di rendering non vengono effettuate direttamente nel framebuffer ma reindirizzate in una texture.

In questo modo si evita la fase di copia da framebuffer a texture, fornendo prestazioni decisamente superiori, oltre ad avere un risparmio di memoria video in quanto il risultato di un operazione di rendering sarà presente in memoria soltanto una volta.

Il render to texture è uno dei pilastri su cui si basano molte delle moderne tecniche real-time. La quantità di effetti visivi che è possibile realizzare ricorrendovi è vastissima; di seguito sono riportati solo alcune delle più importanti applicazioni del render to texture:

- Generazione dinamica di textures procedurali
- Realizzazione di superfici riflettenti
- Tecniche che richiedono più passate come ad esempio l'anti-aliasing, la sfocatura di movimento, la messa a fuoco (o profondità di campo).

- Effetti di elaborazione di immagini (come ad esempio la sfocatura).
- Utilizzo della scheda grafica come unità di elaborazione generica, fornendo un meccanismo per il ritorno in input di dati già elaborati.

Fino a poco tempo fa, l'unica strada adottabile per effettuare un rendering su di una texture consisteva nell'utilizzo di un *Pixel-Buffer* (o P-Buffer). Esso rappresentava a tutti gli effetti una finestra di sistema *invisibile* in cui venivano effettuate le operazioni di rendering. Tuttavia, il P-Buffer aveva una serie di problemi di efficenza che lo rendevano più lento, oltre ad essere particolarmente oneroso da gestire a livello di codice. L'assegnazione di una texture come render target, infine, era poco flessibile in quanto si basava su implementazioni specifiche del sistema su cui esso veniva eseguito.

Successivamente questa tecnica venne sostituita dai *Framebuffer Objects*, analoghi nel funzionamento ai render target usati in DirectX[4].

3.5.2 I Framebuffer Objects

I Framebuffer Objects (FBO) costituiscono un estensione dell'OpenGL standard per effettuare il rendering in un area diversa (off-screen) rispetto al tipico framebuffer utilizzato per visualizzare le immagini su schermo; una delle aree sulle quali è possibile renderizzare con gli FBO può essere ad esempio una texture. Catturare le immagini che normalmente verrebbero visualizzate su schermo permette di implementare vari tipi di filtraggio delle immagini, oltre ad effetti di *post processing*[5]. Gli FBO sono analoghi ai render target presenti in DirectX. In OpenGL, i framebuffer objects sono largamente utilizzati per via della loro elevata efficenza e facilità di utilizzo; essi hanno di

[4]Le DirectX sono le librerie grafiche sviluppate da Microsoft, in tutto analoghe all'OpenGL, ma non cross-platform

[5]Il termine post processing viene utilizzato spesso nell'industria cinematografica ed indica il processo con cui si modifica l'apparenza di una serie di immagini in movimento

fatto rimpiazzato altri metodi per il rendering off-screen, come ad esempio i p-buffer.

Tipicamente, parlando di framebuffer, in Opengl ci si riferisce al *color buffer*, ovvero quel particolare buffer in cui si memorizzano i colori da inviare allo schermo. Come affrontato in precedenza, però, ci sono altri buffer di cui la scheda grafica fa uso come ad esempio lo z-buffer. In generale, quindi, con framebuffer ci si riferisce ad un *insieme* di buffer. Normalmente i buffer su cui è consentito il rendering sono definiti dal sistema; con i framebuffer objects, invece, è possibile definire ulteriori buffer, chiamati *Framebuffer Attachable Images*.

Una Framebuffer Attachable Image può essere costituito da un buffer su cui effettuare il rendering off-screen (chiamato *Renderbuffer*) o da una texture (Fig.3.8).

Una particolarità dei Framebuffer Objects è che tutte le Framebuffer Attachable Images ad esso associate possono essere utilizzati esclusivamente in lettura o in scrittura. Bisogna pertanto usare più FBO per eseguire più operazioni di lettura/scrittura. Dal momento che l'Ambient Occlusion necessita, in una prima fase, di leggere da una texture usata come render target e successivamente scrivere sul framebuffer il risultato di ciascuna Occlusion Query, si è reso necessario utilizzare due Framebuffer Objects. Fortunatamente il costo di un operazione di cambio FBO è estremamente contenuto, pertanto le prestazioni non risentono di questo aggràvio.

Nel primo FBO verrà quindi associata un unica texture in cui verrà scritta la depth map, mentre nel secondo saranno presenti n texture in cui scrivere i risultati delle occlusion queries. Di seguito verrà affrontato il motivo che spinge ad avere n texture per memorizzare il risultato delle queries.

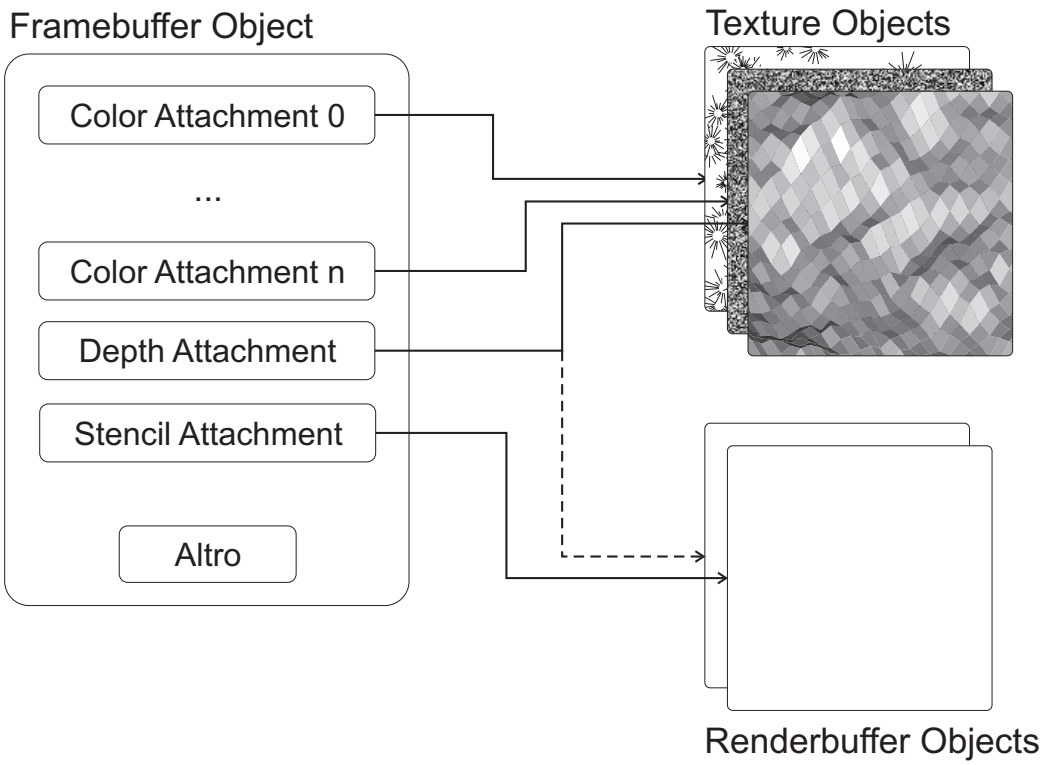


Figura 3.8: *Una rappresentazione schematica del concetto di Framebuffer Object*

3.5.3 Input/Output con la GPU

Parlando di Graphics Processing Unit, spesso si ricorre al concetto di *stream*. In Inglese uno stream è sostanzialmente un *fiume*, un *torrente* di informazioni che necessitano di essere processate. In generale, processare uno stream di dati risulta costoso per una CPU classica. Questo perché bisogna essere in grado di elaborare ogni singola informazione in maniera estremamente efficiente e rapida per poter essere in grado di rallentare lo stream il meno possibile. Le GPU, d'altro canto, utilizzano un approccio differente: esse possiedono un architettura estremamente parallela e specializzata, con diverse *pipeline* di rendering, ciascuna di esse costituita a sua volta da più unità di elaborazione elementari, conosciute anche con il nome di *Arithmetic Logic Units* o ALU. Un architettura così realizzata risulta particolarmente efficiente per l'elaborazione di grossi stream di dati, riuscendo a produrre un numero elevato di dati in output per ogni ciclo di clock.
Per fornire in input ad una scheda grafica uno stream di dati ci si avvale principalmente di textures. Queste vengono utilizzate anche per fornire dati aggiuntivi necessari oltre all'input principale durante l'esecuzione delle operazioni.

L'esecuzione di comandi, in una GPU, avviene sostanzialmente in due fasi distinte: nei vertex shaders si manipolano i vertici che costituiscono la geometria sorgente, mentre nei fragment shaders vengono eseguite manipolazioni sui singoli frammenti della scena. Al termine di queste due fasi, i frammenti, seppur manipolati, verranno scritti in corrispondenza della loro posizione sullo schermo elaborata dal vertex shader. Per tale motivo, elaborare nel vertex shader direttamente la geometria dell'oggetto, costringe i frammenti ad essere scritti in corrispondenza dei vertici: in un secondo momento sarebbe quindi impossibile leggere il valore associato da ciascuna occlusion query ad ogni vertice.

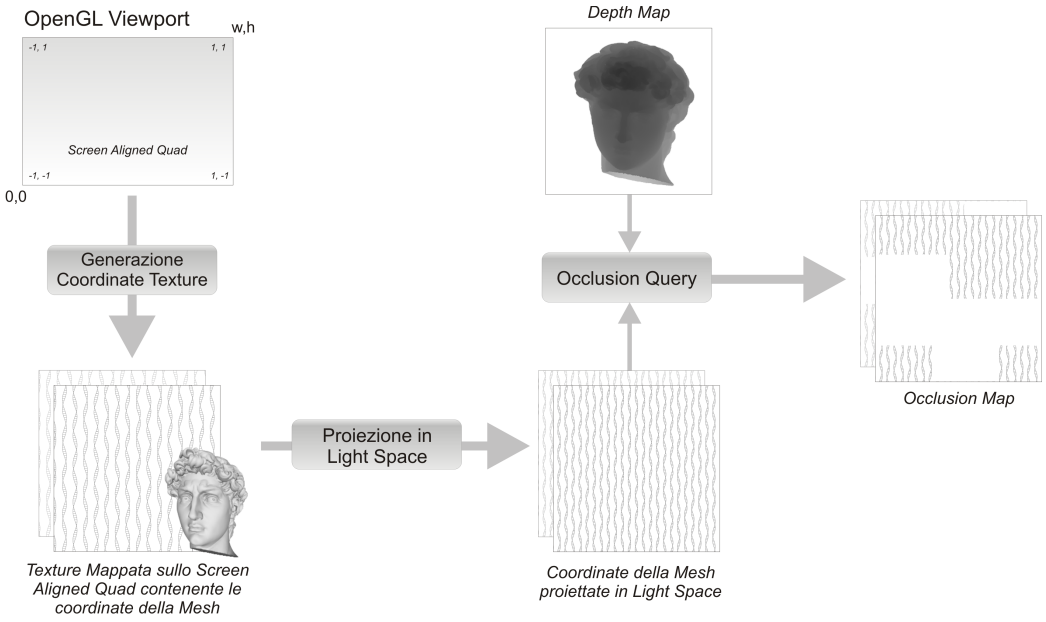


Figura 3.9: *Schema dell'esecuzione hardware dell'Ambient Occlusion usando gli Screen Aligned Quads*

Proprio per permetterne la lettura in un secondo momento, bisogna fare in modo che le coordinate di ciascun vertice vengano mappate su di un piano piuttosto che su una geometria tridimensionale. A tal fine si utilizza come geometria sorgente un quadrilatero che ricopra interamente lo schermo (*screen aligned quad*), fornendo allo shader le coordinate e le normali del modello originario tramite textures (Fig.3.9). Nelle textures, le singole componenti di rosso, verde e blu vengono memorizzate secondo lo schema RGBRGBRG.... Analogamente, volendo memorizzare le coordinate (x, y, z) dei vertici in una texture, esse verranno assegnate secondo lo schema XYZXYZXY....

Durante al fase di generazione delle coordinate texture, il vertex shader processa la geometria dello screen aligned quad: le coordinate di ciascun frammento saranno quindi generate mediante interpolazione, in maniera totalmente automatica.

Avendo a disposizione le coordinate del modello di cui si vuole calcolare l'Ambient Occlusion sotto forma di texel, per essere in grado di leggerle bisogna fare in modo che la texture sia applicata allo screen aligned quad ricoprendolo completamente (*mapping 1:1*): in questo modo, le coordinate generate dal vertex shader potranno essere utilizzate per accedere ad ogni singolo elemento contenente le coordinate dei vertici della mesh originaria.

A questo punto, avendo a disposizione sia le coordinate dei vertici che la direzione delle normali di ciascuno di essi, è possibile eseguire un Occlusion Query, proprio come visto nella versione software. L'output, tuttavia, sarà generato sulla base dello screen aligned quad: pertanto un'altra texture sarà necessaria per memorizzare il risultato dell'accumulazione delle Occlusion Query.

La dimensione di tali textures definisce anche il massimo numero di vertici che è possibile passare allo shader per eseguire i calcoli. Usando una texture particolarmente grande, è possibile risolvere in parte il problema. Attualmente la dimensione massima della texture è fissata a 2048×2048, consentendo di calcolare l'Ambient Occlusion in hardware per modelli composti da oltre *4 milioni* di vertici. Dal momento che passare ad uno shader texture così grandi risulta gravoso, è possibile in parte risolvere il problema con texture *tridimensionali*.

Una texture tridimensionale è analoga alla controparte bidimensionale; l'unica differenza sta nel fatto che nella versione 3D è possibile indirizzare n texture bidimensionali differenti, ognuna posta più in fondo della precedente lungo una immaginaria coordinata z. All'atto pratico le texture tridimensionali sono più simili ad un array di texture 2D, in cui l'indice che permette di scegliere una di esse è dato dalla z.

L'utilizzo di texture tridimensionali impone anche la possibilità di scrivere il risultato di ciascuna Occlusion Query su di un *layer* o *pagina* differente,

in modo coerente alle letture della z-esima texture. Purtroppo non si può usare una texture 3D per l'output di un fragment shader anche se esiste un meccanismo simile: i *Multiple Render Targets* (MRT).

Da qualche anno a questa parte, le schede video sono in grado di reindirizzare l'output del fragment shader su più buffer, semplicemente specificando tramite un indice di tipo intero su quale di questi buffer bisogna scrivere. Una limitazione dei MRT è che non se ne possono avere a disposizione un numero arbitrario: le GPU nVidia G80 ed AMD R600/RV670 mettono a disposizione fino a 8 MRT, mentre altre più datate supportano soltanto 4 target. Ciò non costituisce una forte limitazione in quanto anche nel caso di 4 MRT, si è in grado di processare in hardware una geometria pari a ben *16 milioni* di vertici.

Non tutte le schede grafiche supportano i MRT, per tale motivo l'algoritmo deve essere abbastanza elastico da gestire le tre casistiche: nessuno, quattro o otto MRT. In generale, comunque, se l'hardware supporta i MRT è sempre bene sfruttarli tutti, potendo quindi usare texture di dimensioni inferiori.

Infatti, l'algoritmo attualmente implementato prevede il dimensionamento automatico delle texture usate per l'I/O di dati sia in base al numero di vertici (è inutile usare una texture da 2048 per una geometria di 2000 vertici..) che in base al numero di MRT supportato.

Tanto per fare un esempio, caricando un modello composto da 1700 vertici, l'algoritmo così realizzato avrà bisogno solamente di:

$$\sqrt{\frac{1700 \text{ vertici}}{4 \text{ MRT}}} \simeq 21$$

una texture 32x32 (32 è la potenza di due più vicina a 21[6]) nel caso di 4 MRT, che si abbassa a 16x16 potendo usufruire di 8 MRT. L'accorgimento

[6]Le texture, per essere gestite in modo efficiente dall'hardware grafico, devono avere dimensioni pari ad una potenza di due

così realizzato permette incrementi di prestazioni proporzionali all'aumentare dei vertici.

3.5.4 Occlusion Queries in hardware

L'esecuzione di Occlusion Query in hardware si realizza tramite shaders. Uno shader è sostanzialmente una porzione di codice di alto livello che, una volta compilato, viene tradotto in codice Assembly e memorizzato su di una porzione di memoria video; successivamente, quando il programma lo richiede, esso viene eseguito sostituendo la pipeline standard OpenGL.

In questo caso specifico, si chiede all'hardware di eseguire dapprima una proiezione in light-space dei vertici e successivamente di effettuare un confronto tra la coordinata z_p con la corrispettiva letta dalla Depth Map. La proiezione in light space è necessaria per avere coerenza tra le coordinate dei vertici con le coordinate lette dalla Depth Map: a questo scopo è necessario conoscere in che modo OpenGL trasforma i vertici quando vengono proiettati. Tutte le trasformazioni, in OpenGL, avvengono moltiplicando una matrice per i vertici della scena: tale matrice viene generata in automatico dall'OpenGL una volta impostata la camera da cui si intende osservare la scena e prende il nome di matrice *Model-View*; tuttavia, esiste un'altra matrice, ugualmente importante, anch'essa generata in automatico che però identifica il tipo di proiezione (prospettica, ortogonale, *et cetera*): essa prende il nome di matrice *Projection*.

Al fine di ottenere una corretta trasposizione dei vertici, è necessario ottenere entrambe le matrici e moltiplicarle tra di loro, per poi passare il risultato allo shader che eseguirà le Occlusion Query. In OpenGL ottenere queste matrici è possibile ed è sufficiente invocare il metodo `glGetIntegerv()` passando come parametro `GL_MODELVIEW_MATRIX` per ottenere la matrice Model-View

ed in maniera analoga, specificando `GL_PROJECTION_MATRIX` per ottenere la matrice Projection.

Una volta ottenuto il prodotto di tali matrici, è possibile disegnare nel secondo FBO uno screen aligned quad ed attivare lo shader, passando come parametri:

- I vertici e le normali della geometria sotto forma di texture RGBA tridimensionali a 32bit per canale.
- La depth map relativa alla vista v_i.
- La direzione della vista v_i, necessaria per calcolare il $\cos(\theta)$.
- Il prodotto delle matrici Model-View e Projection.

L'algoritmo all'interno dello shader usato per testare la visibilità di ciascun vertice risulta in tutto analogo a quello visto in precedenza: le uniche differenze sono nel metodo con cui si ottengono i vertici e le normali del modello. Man mano che il vertex shader avanza nell'esecuzione, produce le coordinate da utilizzare per leggere dalle texture. Effettuando un texture lookup su tali coordinate è possibile ottenere i vertici e le normali cercate. A questo punto l'esecuzione dello shader effettua le stesse operazioni viste nella versione software: i vertici vengono prima proiettati in light space, successivamente si utilizzano le coordinate (x_p, y_p) per leggere dalla depth map il valore z_d, quindi si effettua un confronto tra quest'ultimo e z_p per testare la visibilità del vertice; nel caso in cui il vertice risulti occluso, andrà scritto nel frammento il prodotto scalare tra la normale (letta in precedenza) ed il vettore identificante la direzione di vista (normalizzata in modo da avere lunghezza unitaria).

La gestione della profondità delle texture 3D in ingresso viene gestita *manualmente* (incrementando esplicitamente una variabile) nel caso in cui

la scheda video non supporti correttamente la presenza di cicli nello shader; per le schede che invece li supportano, si è fatto ricorso al loro utilizzo per avere un codice più compatto.

Per completezza, si riporta di seguito il codice in linguaggio GLSL dello shader (con supporto ai cicli) che effettua le Occlusion Query:

```
vec4 project(vec4 coords)
{
   coords = mvprMatrix * coords; // clip space [-1 .. 1]
   return vec4(coords.xyz * 0.5 + 0.5, coords.w);
}

vec4 occlusionQuery(float zLevel, float zLevelMax)
{
      vec4 R = vec4(0.0, 0.0, 0.0, 1.0);

      float zCoord = zLevel/zLevelMax;
      zCoord += 1.0/(zLevelMax*2.0);

      vec3 vCoord3D = vec3(gl_FragCoord.xy/viewpSize, zCoord);

      vec4 N = texture3D(nTexture, vCoord3D);
      vec4 P = project(texture3D(vTexture, vCoord3D));

      //scaling based on viewport size
      P *= (viewpSize/texSize);

      if ( shadow2DProj(dTexture, P).r > 0.5 )
         R.r = max(0.0, dot(N.xyz, viewDirection));
```

```
      return R;
}

void main(void)
{
   for (float z3D=0.0; z3D<numTexPages; z3D+=1.0)
      gl_FragData[int(z3D)] = occlusionQuery(z3D, numTexPages);
}
```

Tutta la fase di accumulazione è stata demandata all'OpenGL grazie al metodo `glBlendFunc()` che definisce una funzione di *hardware blending*. Attivando il blending, si fa in modo che l'output corrente non sovrascriva il precedente, ma si unisca ad esso. La modalità con cui i due buffer si *fondono* è definita da una equazione:

$$C_{\text{out}}^{k} = \min(2^{b-1}, (C_{\text{out}}^{k-1} * \alpha + C_{\text{in}}^{k} * \beta))$$

dove α e β sono due parametri specificati nella `glBlendFunc()` e b il numero di bit memorizzabili in ciascun canale. Volendo, quindi, eseguire una semplice somma è sufficiente specificare $\alpha = 1$ e $\beta = 1$.

Accelerazione della geometria

Si è notata, comunque, una cattiva scalarità dell'algoritmo, spesso più veloce della controparte software soltanto di un 20%. Dopo attente analisi, si è giunti alla conclusione che la velocità di esecuzione dell'algoritmo era fortemente penalizzata dal costo computazionale che ridisegnare la geometria ad ogni vista comportava. La libreria OpenGL, a riguardo, fornisce sostanzialmente due alternative per velocizzare le operazioni di rendering geometrico: le *display lists* ed i *vertex buffer objects*.

Le display lists (*DL*) erano ampiamente utilizzate fino a due–tre anni fa, quando le GPU non erano ancora abbastanza evolute da permettere altre alternative. L'uso di DL, tuttavia, era estremamente vantaggioso, permettendo di velocizzare in maniera tangibile tutte le operazioni di rendering geometrico. Le DL si utilizzavano, sostanzialmente, associando una generica serie di operazioni ad un *handle* di riferimento. Tutte le operazioni di una DL venivano, quindi, memorizzate in memoria video ed ogni qual volta nel codice OpenGL si chiedeva di eseguire le operazioni identificate dall'handle precedentemente creato, essere si trovavano già in memoria, pronte per essere eseguite dall'hardware grafico.

Purtroppo, le DL presentavano problemi nella gestione di geometrie particolarmente complesse, fornendo risultati impredicìbili. Dal momento che in MeshLab si lavora spesso con mesh particolarmente complesse, costituite da centinaia di migliaia (a volte addirittura milioni) di vertici, l'utilizzo delle DL risulta obbligatoriamente precluso.

Per risolvere le limitazioni delle Display Lists, vennero introdotti i *Vertex Buffer Objets* (VBO). Con i vertex buffer, sostanzialmente, si chiede alla GPU di mantenere in memoria una copia delle geometria che in seguito si pensi verrà utilizzata spesso. Tramite questo semplice meccanismo non si avrà necessità di ricaricare le informazioni da visualizzare ad ogni frame, aumentando le prestazioni.

Dal momento che possono nascere problemi con l'utilizzo dei VBO principalmente a causa dell'implementazione software del produttore, si è deciso di lasciare all'utente la possibilità di utilizzarli o meno.

È bene notare, comunque, che si è data la possibilità di abilitare i VBO indipendentemente dall'utilizzo della GPU per eseguire le Occlusion Query. Questo perché, pur mantenendo un esecuzione completamente software dell'algoritmo, è possibile velocizzare le operazioni correntemente già delegate dalla scheda grafica.

3.5.5 Problematiche derivanti dall'uso della GPU

Purtroppo, l'utilizzo delle schede grafiche per l'accelerazione degli algoritmi presenta delle problematiche peculiari. In *primis*, non sapendo a priori su quale hardware l'algoritmo verrà eseguito, bisogna cercare di immaginare tutte le casistiche ed implementare un *path* di esecuzione per ognuna di esse, eventualmente prevedendone uno di *fall-back* per i casi particolari. In *secundi* a causa di alcune limitazioni che hanno afflitto, fino a poco tempo fa, le GPU rendendole inadatte a determinate operazioni.

Attualmente l'algoritmo presenta una limitazione nel calcolo accelerato in hardware. Come accennato in precedenza, le texture utilizzate per passare i vertici e le normali sono in formato RGBA a 32-bit per canale (per comodità ci si riferità a questo tipo di texture con `RGBA_32F`): una particolarità di questo tipo di texture consiste nell'avere i texel in formato *floating point* a differenza delle normali texture che sono di tipo intero.

Quasi la totalità dell'hardware oggi a disposizione supporta questo formato di texture, ma una percentuale altrettanto elevata di GPU manca di una caratteristica che impedisce all'algoritmo, per com'è strutturato, di funzionare in hardware.

Durante l'esecuzione in modalità accelerata, al termine di ogni Occlusion Query il risultato viene scritto su di una texture `RGBA_32F` collegata ad un framebuffer object. L'accumulazione dei singoli valori, come si è detto, è stata demandata all'OpenGL tramite una funzione di blending: il problema che affligge l'attuale implementazione sta proprio nel fatto che, tranne nelle più recenti schede grafiche, non è possibile effettuare l'hardware blending su valori floating point a 32-bit, ma solo a 16-bit.

Un problema che potrebbe sembrare banale, ma che in realtà non lo è. Per consentire una corretta gestione di numeri floating point a 16-bit (detti a *mezza precisione*) bisogna fare in modo che tutti i valori circolanti all'in-

terno dell'algoritmo abbiano identica precisione. Eseguire, infatti, operazioni su numeri con diversa precisione risulta in errori di calcolo dovuti principalmente alla incapacità di C++ di interpretare correttamente la mantissa e l'esponente dei numeri a mezza precisione.

3.6 Conclusioni e possibili evoluzioni future

Una delle prime migliorie implementabili può essere sicuramente l'implementazione di un path specifico per le schede video che mancano del supporto al floating point 32-bit blending, ampliando il parco macchine in grado di eseguire l'algoritmo con accelerazione hardware.

Un'altra possibile miglioria, che però cambierebbe completamente la struttura stessa dell'algoritmo, potrebbe prevedere la rielaborazione del contributo ambientale ad ogni frame e non soltanto all'inizio. Così facendo si potrebbe modificare interattivamente la geometria, mantenendo l'Ambient Occlusion coerente con essa. L'algoritmo attuale, ad ogni modo, non si presta bene per essere eseguito ad ogni frame, se non scendendo a compromessi sulla qualità: per questo motivo bisognerebbe implementare un'altra tecnica, pensata proprio per garantire un elevato frame rate della scena interattiva.

Tuttavia, per la natura statica dei modelli importati in MeshLab, non c'è necessità di implementare un simile algoritmo: è preferibile, in questo caso, attendere una frazione di secondo in più pur di avere un risultato qualitativamente superiore.

3.6.1 Analisi delle prestazioni

Concludendo quest'interessante capitolo sul calcolo dell'Ambient Occlusion, si riporta una tabella contenente i dati relativi alle prestazioni dell'algoritmo. Il sistema su cui è stato testato è costituito da un processore AMD Athlon64

X2 2,5 GHz, 2 GByte DDR Ram, scheda video nVidia GeForce 8800 GTS 320 MB e sistema operativo Windows Vista Business.

I modelli utilizzati per questa breve analisi sono la testa del David di Michelangelo, gentilmente fornita dal *Visual Computing Lab* stesso, e da un dragone in stile orientale. I parametri impostati nel plug-in prevedono 256 viste e una risoluzione di 1024 × 1024 per la generazione della Depth Map[7]. Di seguito i risultati espressi in secondi (tra parentesi sono riportati il numero di vertici per ciascun modello) :

	David (125.275)		Dragon (437.645)	
	Senza VBO	Con VBO	Senza VBO	Con VBO
Senza GPU	44,4	16,8	133,7	45,8
Con GPU	25,9	3,3	100,8	8,3

[7]La risoluzione della Depth Map non influisce in alcun modo sulla risoluzione delle texture usate per l'I/O

Capitolo 4

Generazione Procedurale di Materiali

Essendo OpenGL una piattaforma aperta e orientata all'evoluzione, molte delle esigenze più comuni vengono facilmente coperte dal modello statico descritto nei precedenti capitoli. Esiste però una serie di casistiche che non possono essere gestite in questo modo: quando è richiesta una maggiore flessibilità si fa ricorso agli shaders dell'OpenGL.

L'*OpenGL Shading Language* permette alle applicazioni di definire il comportamento dei singoli stadi della pipeline OpenGL tramite un linguaggio di alto livello specificamente indirizzato a questo tipo di necessità.

La libertà fornita dalla programmabilità della pipeline non solo permette l'implementazione di un numero elevatissimo di effetti grafici, ma ne aumenta ulteriormente le possibilità ad ogni nuova generazione di GPU messa in commercio. Segue un breve elenco che riporta una parte di ciò che possibile realizzare con gli shaders:

- Realizzazione realistica di materiali, quali metalli, pietre, legno, plastiche e così via.

- Simulazione di effetti ambientali come fuoco, nebbia, ghiaccio, pioggia.
- Effetti di luce e ombre realistici.
- Simulazione della translucentezza dei materiali, come latte, marmo e pelle.
- Non solo fotorealismo: è possibile anche realizzare effetti stile cartoon e disegno a mano libera.
- Elaborazione di immagini: implementazione di algoritmi di filtraggio, conversione colori, e così via.

Alcune di queste operazioni sarebbero implementabili anche con la classica pipeline statica OpenGL, ma risulterebbero limitate in qualche modo o sarebbero talmente lente da risultare inutili. Grazie agli shaders, invece, è possibile eseguire tutti gli effetti riportati poco sopra sfruttando l'hardware specializzato delle schede grafiche, ottenendo anche un notevole guadagno in termini prestazionali.
Nel corso del tirocinio presso il VC Group del CNR di Pisa, sono stati realizzati due shader per la simulazione di superfici quali marmo e granito, da integrare con la tecnica dell'Ambient Occlusion descritta nel capitolo precedente e con le Translucent Shadow Maps affrontate nel prossimo, al fine di raggiungere un elevato fotorealismo nella simulazione di tali materiali.

4.1 OpenGL Shading Language

Non appena venne introdotto il concetto di shader all'interno dell'OpenGL, la loro programmazione avveniva tramite linguaggi a basso livello, simili in molti aspetti all'Assembly delle architetture x86. Il problema principale di questo approccio, come naturalmente intuibile, risiedeva nella natura stessa

del linguaggio di programmazione: l'Assembly è, per definizione, specifico per ogni singolo produttore di hardware, rendendo particolarmente difficile e lungo la creazione di codice compatibile con un numero elevato di configurazioni differenti.

L'idea, quindi, era quella di creare un linguaggio di programmazione ad alto livello, che si slegasse dalle implementazioni specifiche dei singoli produttori di hardware, ma nello stesso momento che fosse ugualmente flessibile e sufficientemente funzionale da poter divenire uno standard duraturo nel tempo.

La soluzione giunse nel Giugno del 2003, quando l'*OpenGL Architecture Review Board (ARB)* fornì un implementazione, tramite estensioni dell'OpenGL standard, del cosiddetto *OpenGL Shading Language*, successivamente divenendo parte integrante dello standard ed abbandonando la propria natura di estensione.

Struttura del Linguaggio GLSL

Come accennato precedentemente, il GLSL si basa sulla sintassi del linguaggio di programmazione ANSI C, infatti, ad una prima osservazione, i programmi scritti in questi due linguaggi tendono ad essere molto simili. Ciò è stato intenzionale da parte dell'ARB, proprio per fare in modo che il linguaggio fosse di immediato utilizzo, dato che la quasi totalità dei programmatori di grafica interattiva utilizzano C o C++. Il punto di inizio, da cui parte l'esecuzione del programma stesso, si ha in corrispondenza di un `void main()` ed il corpo del programma è limitato da parentesi graffe. Le strutture condizionali `if-then-else`, i cicli tramite `for`, la dichiarazione delle variabili, *et cetera* sono molto simili al C.
Dal momento che l'OpenGL Shading Language mira, tramite una sintassi conosciuta, ad essere un linguaggio semplice ma nel contempo potente e

flessibile, funzioni matematiche utili a codificare algoritmi di natura grafica sono presenti nel linguaggio base: funzioni trigonometriche, moltiplicazioni tra vettori e matrici, prodotti scalari e vettoriali, *et cetera*.

Per quanto riguarda i tipi di dato, nel mondo della grafica tridimensionale spesso si fa riferimento a vettori di tre o quattro elementi, siano essi booleani, interi o floating point. A tale scopo sono stati inseriti diversi tipi di dati specifici per vettori e le matrici. Per dati di tipo floating point, tali vettori sono identificati da `vec2` (due float), `vec3` (tre float) e `vec4` (quattro float); anteponendo una "i" o una "b" si specificano invece vettori rispettivamente di tipo intero e booleano.

Per quanto riguarda la gestione delle matrici si hanno a disposizione i tipi `mat2`, `mat3` e `mat4`. Esistono poi tipi di dati speciali detti *campionatori* che consento di effettuare l'operazione di lettura da textures (`sampler2D` per texture bidimensionali, `sampler3D` per quelle tridimensionali). Ciascun tipo di variabile, se dichiarato al di fuori di un blocco, può presentare un qualificatore di tipo, che ne specifica la semantica di gestione (ad esempio se rimane costante lungo l'invio di una primitiva geometrica, se deve essere interpolato dal rasterizzatore e così via).
Per ulteriori approfondimenti sul linguaggio GLSL si rimanda a [13].

4.2 Texture Procedurali

Un applicazione pratica degli Shader è data dalla possibilità di simulare superfici particolarmente complesse, composte da venature irregolari, quali ad esempio il marmo. Un approccio tradizionale a questo problema sarebbe stato quello di scattare una fotografia ad una lastra composta di marmo, magari illuminata da una sorgente diffusa in modo da evitare fenomeni speculari, ottenendo così una texture da applicare sul modello e rendere l'idea del marmo. Questa tecnica risulta efficace a patto che la foto scattata sia

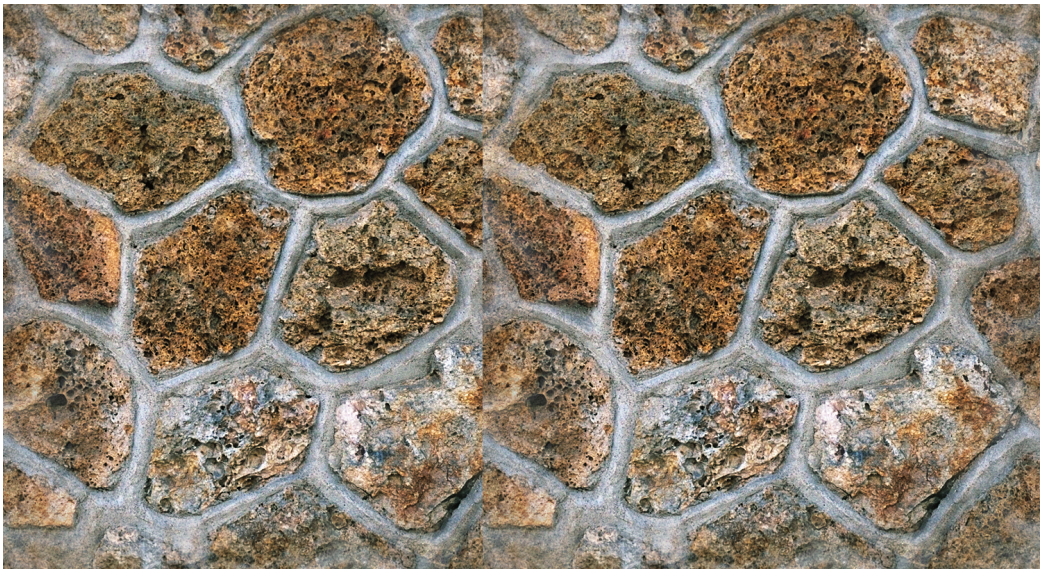


Figura 4.1: *Artefatti visivi causati da una texture non studiata per il tiling*

di elevata qualità: nel caso di specularità nell'immagine sorgente, o di foto a bassa risoluzione, l'efficacia della tecnica viene enormemente ridotta. Tuttavia, essa presenta un limite particolarmente importante: pur tenendo in considerazione la scala del modello-texture, nel caso generale si rende necessario affiancare la stessa immagine più volte per coprire l'intera superficie del modello. Questo prende il nome Inglese di *tiling* e, con texture non opportunamente realizzate, presenta vistosi artefatti proprio in corrispondenza dei bordi in cui le due copie della stessa immagine si uniscono (Fig.4.1) : nel caso del marmo, tali artefatti si manifestano sotto forma di discontinuità delle venature.

Per cercare di risolvere il fenomeno degli artefatti nel tiling e più in generale, per sintetizzare immagini rappresentanti fenomeni fisici altrimenti difficilmente simulabili, si ricorre alle *texture procedurali* (Fig.4.2).

Con il termine *texture procedurale* ci si riferisce alla possibilità di creare matematicamente delle superfici realistiche (come le pietre, il legno, la carta

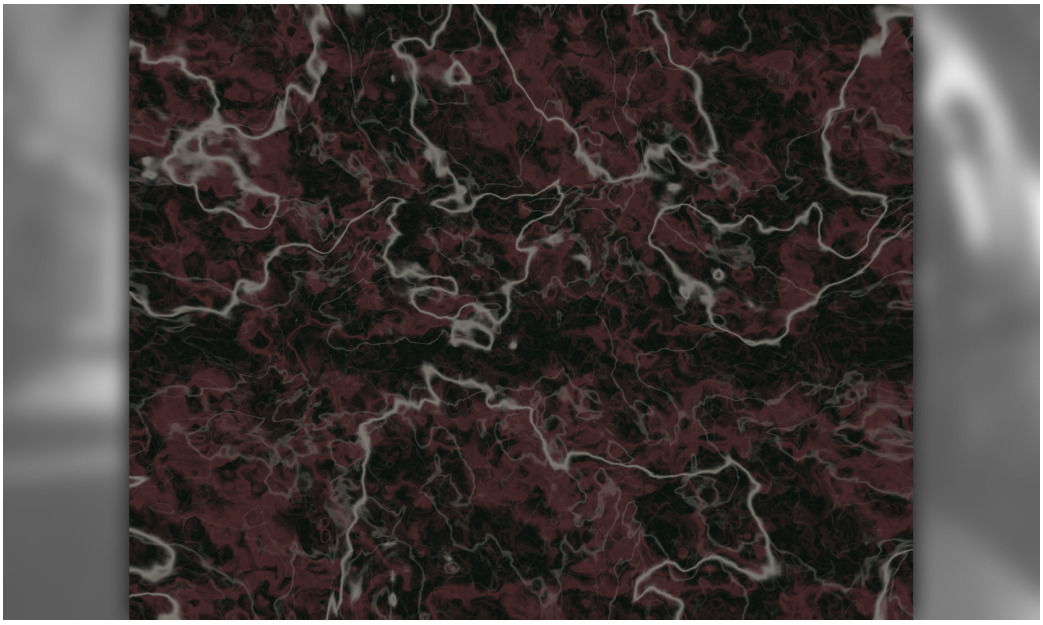


Figura 4.2: *Esempio di texture procedurale (marmo rosso)*

da parati, *et cetera*) evitando la simulazione fisica dei fenomeni che portano alla tipica apparenza che le contraddistingue.

Uno dei vantaggi dell'approccio procedurale sta nella flessibilità di rappresentare un fenomeno con un qualsiasi livello di accuratezza; si può andare, quindi, da approcci pesantemente indirizzati alla simulazione fisica del fenomeno ad altri basati esclusivamente sull'effetto artistico finale.

Per quanto detto sopra, la generazione procedurale si rivela un potente paradigma per la sintesi di immagini. Con questo approccio, più che specificare dettagliatamente e memorizzare una gran quantità di dati complessi al fine di riprodurre una determinata superficie, si cerca di descrivere l'apparenza mediante una funzione od un algoritmo (sinonimo di procedura, da cui il nome). In questo modo non c'è necessità alcuna di memorizzare particolari informazioni, risparmiando spazio e delegando all'elaboratore tutta la parte di computazione dell'algoritmo.

Descrivendo l'apparenza di una superficie tramite un algoritmo, si ottiene automaticamente anche un altro importante vantaggio: si immagini di avere un oggetto fatto di legno; nel caso in cui tale oggetto venga tagliato in due, la parte interna mostrerà l'evoluzione interna delle venature presenti sulla superficie. Volendo trasporre le informazioni di volume del materiale su di un oggetto digitalizzato, bisogna ricorrere necessariamente a texture tridimensionali, molto costose in termini di spazio occupato e parallelamente, particolarmente difficili da realizzare. Grazie alle texture procedurali, invece, il problema è automaticamente risolto: le coordinate di ogni punto della superficie di una qualsiasi mesh tridimensionale possono costituire l'input di una funzione o di una procedura che simuli un determinato materiale. Immaginando di dividere in due un modello digitale, la parte interna viene automaticamente *texturizzata* coerentemente con la posizione dei singoli vertici: quanto mostrato su schermo sarà, infatti, il risultato della funzione calcolato nei punti che compongono la parte interna della mesh.

4.3 Simulazione del Marmo

Utilizzando gli shader ed il concetto di texture procedurale, è possibile realizzare in tempo reale una simulazione visivamente realistica di superfici con tratti pseudo-casuali, come ad esempio il marmo e le sue tipiche venature. Al fine di creare venature pseudo-casuali si è ricorsi al *Perlin Noise* per generare una serie di pattern casuali, ripetibili nel tempo.

4.3.1 Perlin Noise

Osservando il mondo circostante, ci si rende conto di come molte seguono il comportamento dei frattali. Ci sono, infatti, vari livelli di dettaglio che descrivono ad esempio una montagna: osservandone la superficie da molto

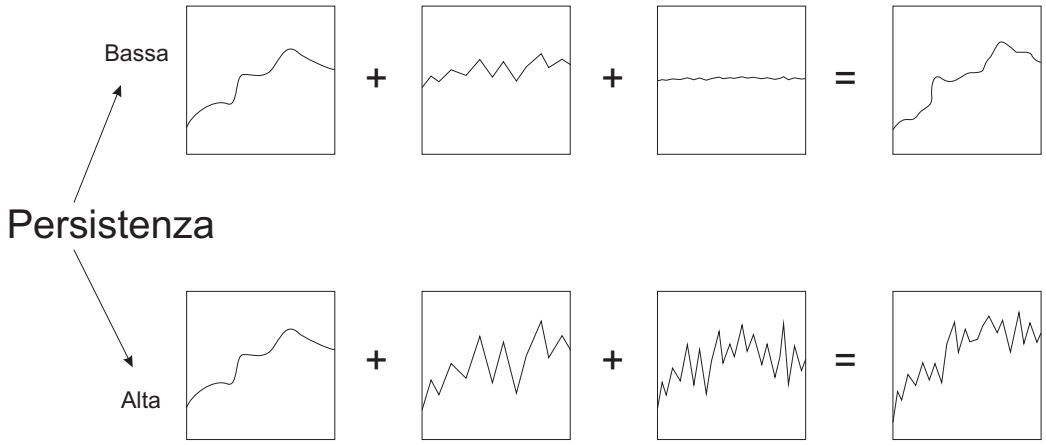


Figura 4.3: *Mantenendo costante la frequenza, si varia l'ampiezza in modo da poter controllare la ricaduta dei vari livelli di rumore sulla funzione finale*

lontano si notano frastagliature lungo i contorni; avvicinandosi ad essa, si possono notare altri dettagli via via più fini: macigni, rocce, pietre . . . e così via. Questo è solamente un esempio, in quanto questo insieme di variazioni su larga e piccola scala lo si può osservare in moltissimi fenomeni naturali.

Il *Perlin Noise*, che prende il nome dal suo inventore Ken Perlin, mira a ricreare questi fenomeni sommando tra loro varianti di una stessa funzione basata sulla generazione di numeri casuali, ogni volta con scala diversa.

Il Perlin Noise è quindi un esempio di texture procedurale, attualmente ampiamente utilizzato dai programmatori e dagli artisti per aumentare il realismo nella generazione di effetti. La funzione procedurale descrive un apparenza pseudo-casuale, anche se ciascun dettaglio della trama così generata presenta la stessa dimensione di tutti gli altri dettagli. Per questa importante proprietà, il Perlin Noise presenta un elevato grado di controllabilità.

I parametri sui quali è possibile intervenire per modificare la trama del noise sono tre, la *frequenza*, la *persistenza* e le *ottave*: la prima indica la rapidità delle variazioni che intercorrono tra valori di noise adiacenti. Con il termine persistenza, invece, si indica quanto un ulteriore somma di noise

influenzerà il risultato finale: questo si traduce in una riduzione dell'ampiezza che i valori del noise assumeranno alla i-esima iterazione (Fig.4.3). La relazione che lega frequenza ed ampiezza è data da:

$$\text{frequenza} = 2^i \tag{4.1}$$

$$\text{ampiezza} = \text{persistenza}^i \tag{4.2}$$

Le ottave, infine, rappresentano il numero di iterazioni che l'algoritmo eseguirà, sommando di volta in volta rumore a frequenze ed ampiezze differenti.

In uno shader, il Perlin Noise lavora in funzione delle coordinate dei vertici di un modello ed usa un interpolazione tra i valori calcolati per produrre dettagli a livello di frammento. Quindi, sostanzialmente, per ottenere il Perlin Noise servono due elementi: una funzione che generi numeri casuali (chiamata in gergo *RNG*) ed un'altra che esegua un interpolazione.

4.3.2 Il problema dei RNG in Hardware

Dal momento che l'obiettivo di questo capitolo è la realizzazione tramite shaders di texture procedurali, c'è la necessità di eseguire tutte le operazioni in hardware: questo vale sia per la generazione di numeri casuali, che per l'interpolazione. Purtroppo, sino a poco tempo fa, le GPU non erano in grado di eseguire operazioni su interi in maniera sicura in quanto, mancando di unità aritmetiche per la gestione degli interi, delegavano le operazioni sulle unità floating point: tuttavia, eseguendo calcoli semplici che mantengano il risultato finale entro un certo limite (normalmente 65.536) si riusciva ad essere abbastanza certi del risultato.

Ma dal momento che implementare un algoritmo *RNG* (Random Number Generator) prevede l'utilizzo di operatori bit a bit e di aritmetica intera ai limiti dell'intervallo rappresentabile[1], avrebbe significato un comportamento

[1]Normalmente un intero contiene 2^{32} valori.

non prevedibile della funzione, diverso da caso a caso. Un buon generatore di numeri casuali, infatti, assicura di riprodurre la stessa sequenza pseudo-casuale ogni qual volta si fornisce in ingresso lo stesso seme[2] (*seed*).

Data la limitazione imposta dall'hardware grafico, si è scelto di precalcolare una serie di numeri casuali e successivamente, passarla allo shader tramite una texture: ciascun texel[3] contiene uno dei numeri generati.

Tradizionalmente, realizzando in C/C++ un programma *ad hoc* per visualizzare il risultato dello shader, si sarebbe potuto dapprima generare la texture di numeri casuali e successivamente, passarla allo shader come visto in occasione dell'Ambient Occlusion. Dal momento che per questo sotto-progetto si è deciso di utilizzare *RenderMonkey*, ciò non è richiesto: fra le texture contenute nel software, è presente una adatta allo scopo descritto poco sopra.

4.3.3 L'algoritmo

Descritte le problematiche che rendono utile implementare una texture procedurale per la simulazione del marmo, ed affrontati i concetti che sono alla base del Perlin Noise, si passa alla descrizione dell'algoritmo implementato. Prendendo come spunto diverse fonti[14] che trattano tecniche procedurali per riprodurre fedelmente il marmo, si evince che l'algoritmo base prevede di utilizzare il Perlin Noise come fattore di spostamento (*shifting*) di una funzione coseno:

$$\text{texture} = \cos(x + \text{perlin}(x, y, z))$$

[2]Una sequenza di numeri casuali viene generata a partire da un parametro (detto appunto *seme*). Questo consente alla sequenza generata di essere ripetibile

[3]Texel sta per **TEX**ture **EL**ement ed identifica, appunto, un elemento della texture.

In realtà, una funzione così definita rende ben poco l'idea di una superficie marmorea in quanto mancante delle classiche venature chiare e sottili, alternate a zone più scure ed ampie.
Partendo da queste basi, comunque, si è pensato di elaborare contemporaneamente due funzioni identiche ma con parametri differenti per avere maggiore libertà e poter separare le venature chiare da quelle scure (Fig.**??**). Al pari dei metodi suggeriti, è stata applicata l'idea di utilizzare la funzione coseno per dare una ulteriore sensazione di irregolarità e di alternanza nelle venature. Dopodichè grazie a delle funzioni di turbolenza, si è riusciti a manipolare i valori casuali letti dalla texture di supporto, in modo da ottenere venature sottili o spesse a seconda del risultato desiderato. Infine si diminuisce l'ampiezza di ciascuna ottava, incrementando nel contempo la frequenza in maniera differente a seconda del tipo di venature.
Si riporta di seguito il codice della funzione Perlin Noise realizzata:

```
for (i = 0.0;  i < 4.0;  i++)
{
   turb1 = 1.0 - abs(cos(snoise(P1)).x);
   turb2 = 1.0 - abs(cos(snoise(P2)).x);

   turb1 = pow( smoothstep(0.85, 1.0, turb1), 2.0) / ampl;
   turb2 = pow( smoothstep(0.80, 1.0, turb2), 64.0) / ampl;

   t1 += (1.0-t1) * turb1;
   t2 += (1.0-t2) * mix(turb1, turb2, 0.95);

   ampl*=2.3;

   P1 = P1 * 2.3 + 0.50;
```

```
    P2 = P2 * 1.7 + 0.25;
}
```

Nel codice sono presenti riferimenti a funzioni tipiche del GLSL che ne impediscono la comprensione a tutti coloro non abbiano mai studiato tale linguaggio. Per tale motivo si fornisce una descrizione delle funzioni utilizzate:

- `abs`: La funzione ritorna semplicemente il modulo del parametro fornito in ingresso.
- `smoothstep`: Accetta tre parametri, `edge1`, `edge2`, `x`. Ritorna zero se $x \leq$ edge1, uno se $x \geq$ edge2 ed esegue una interpolazione di Hermite nel caso in cui edge1 $\leq \ x \ \leq$ edge2.
- `mix`: Esegue un interpolazione lineare tra i primi due parametri in ingresso utilizzando come α il terzo parametro.

Eseguito l'algoritmo di cui sopra, si provvede ad un interpolazione lineare tra i colori più scuri e quelli più chiari: nel caso delle venature scure, si utilizza un interpolazione tra due colorazioni, opportunamente scelte, utilizzando $\alpha = t_1$. Per definire le venature più chiare e sottili, si interpola linearmente il valore precedentemente calcolato con una colorazione tendente al bianco, utilizzando $\alpha = t_2$.

La scelta dei colori per la realizzazione dello shader nelle sue varianti rossa, verde e nera è stata fatta basandosi su fotografie di marmi reali.

4.3.4 Miglioramenti

Al fine di rendere la simulazione del marmo ancora più realistica, si è deciso di implementare, oltre a quanto già visto fino ad ora, due tecniche semplici ma particolarmente efficaci: le riflessioni dell'ambiente circostante ed il modello di illuminazione locale proposto da G. J. Ward.

La Riflessione

La riflessione è un operazione particolarmente semplice ma di sicuro effetto se applicata su superfici lucide, quali il marmo appunto. E' sufficiente effettuare un lookup su di una texture che rappresenti lo sfondo della scena secondo la direzione del vettore $\vec{R}$: tale vettore rappresenta la direzione della riflessione del vettore di vista $\vec{V}$ rispetto alla normale $\vec{N}$ della superficie. Per calcolarlo è sufficiente richiamare la funzione `reflect()` già definita in GLSL o applicare la seguente formula:

$$\vec{R} = 2\vec{N}(\vec{N} \cdot \vec{L}) - \vec{L}$$

Dove $\vec{L}$ indica la direzione della luce.

Modello di illuminazione di Ward

Il modello di illuminazione locale implementato venne proposto nel 1992 da Gregory J. Ward[15] e si presenta come un modello basato sulla fisica dei materiali per simularne l'apparenza su schermo. Modelli empirici come il Phong, ad esempio, prendono in considerazione semplicemente un elenco di parametri su cui impostare l'apparenza di un determinato oggetto in un determinato contesto: cambiare la scena richiede un cambio di questi parametri. I modelli basati sulla fisica, invece, oltre ad essere più accurati in quanto basati su equazioni che descrivono le proprietà di un materiale, presentano generalmente una serie di parametri che permettono di modificare caratteristiche fisiche del materiale stesso.

Il modello di Ward si caratterizza proprio per l'accuratezza con cui è in grado di simulare superfici *isotropiche* ed *anisotropiche*[4], oltre che per la semplicità matematica che lo rende particolarmente adatto ad un implementazione in hardware.

[4]In fisica, l'isotropia è la proprietà di indipendenza dalla direzione, da parte di una grandezza definita nello spazio. Il suo contrario è l'anisotropia.

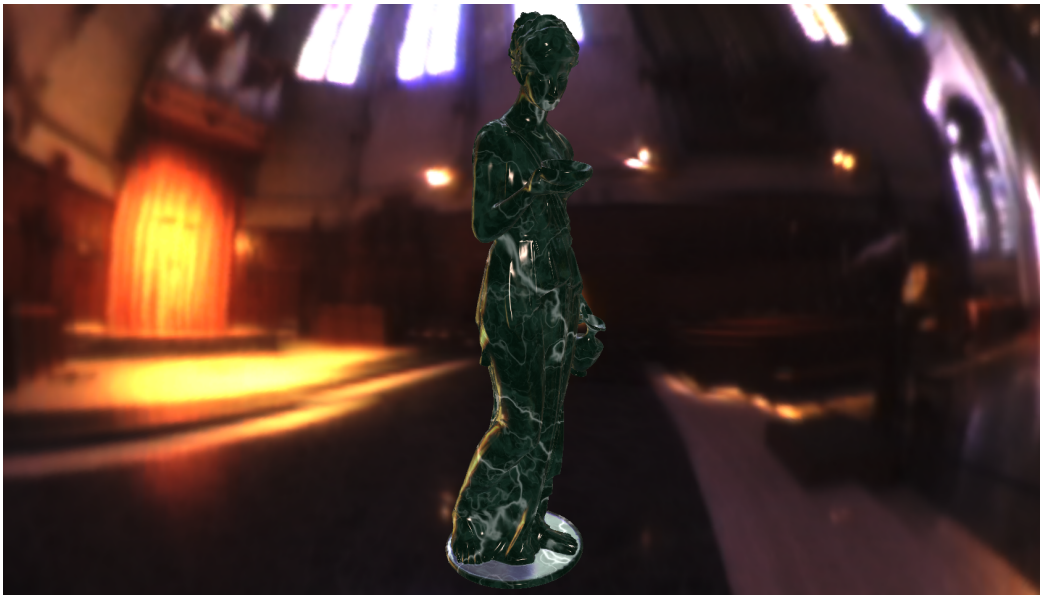


Figura 4.4: *Risultato finale dello shader "Marmo" con l'aggiunta di riflessioni e illuminazione locale di Ward*

Nel caso dello shader sul marmo si è scelti di implementare la versione isotropica, in quanto il marmo può essere collocato in questa categoria di materiali. Essa è descritta dalla formula:

$$I = \frac{1}{4\pi\alpha^2} \frac{\exp[-\tan^2 \delta/\alpha^2]}{\sqrt{(\vec{N} \cdot \vec{L})(\vec{N} \cdot \vec{V})}} \tag{4.3}$$

Dove I è l'illuminazione risultante, δ è l'angolo tra il vettore $\vec{N}$ ed il vettore $\vec{H}$[5], mentre α identifica la rugosità media della superficie.

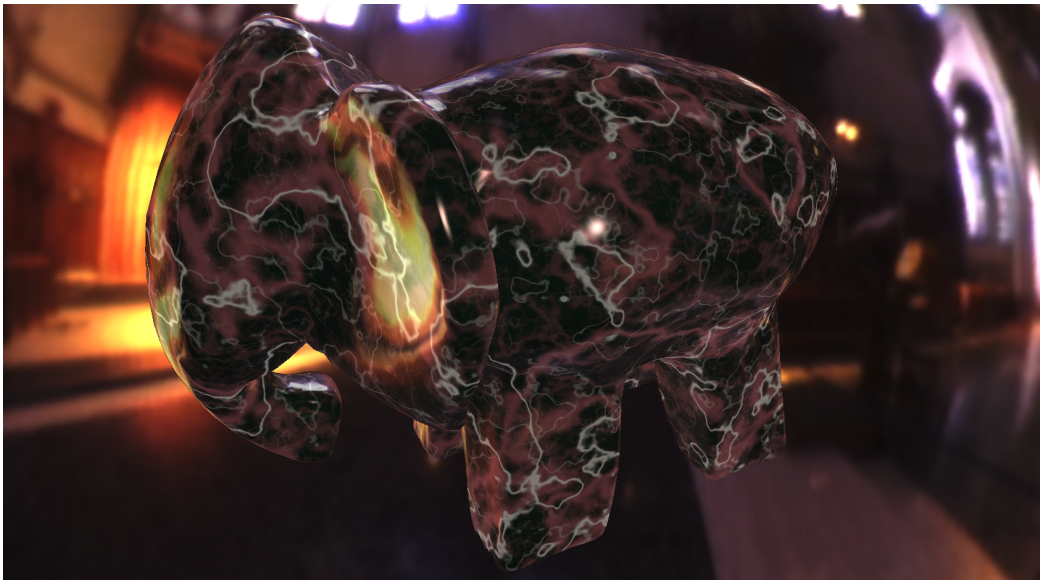


Figura 4.5: *Risultato finale dello shader "Marmo" nella variante rossa*

4.3.5 Risultato Finale

La texture procedurale così realizzata, le riflessioni ambientali ed il modello di illuminazione di Ward concorrono al raggiungimento del risultato mostrato nelle Fig.4.4 e 4.5.

Lo shader riproduce fedelmente l'apparenza tipica del marmo, mostrando le sottili venature chiare poste al di sopra della nube di colori che ne identificano la colorazione predominante.

Nonostante l'elevato livello qualitativo raggiunto, la visualizzazione viene eseguita ad un alto numero di fotogrammi per secondo, garantendo quindi una riproduzione fluida anche su schede grafiche non professionali.

[5]Il vettore $\vec{H}$ rappresenta l'*halfway vector* ossia il vettore posto a metà fra i vettori di vista e luce – È un approssimazione accurata che evita di dover ricalcolare il vettore di riflessione $\vec{R}$.

4.4 Simulazione del Granito

Descritti in dettaglio i passi necessari per realizzare uno shader per il marmo, crearne uno per la riproduzione del granito richiede soltanto di apportare alcuni cambiamenti alla funzione di Perlin Noise ed al tipo di colorazione, mantenendo però tutta la parte relativa alle riflessioni e l'illuminazione con Ward.

Data la natura altamente irregolare e puntiforme del granito, si è scelto di implementare una versione classica del Perlin Noise, con ampiezza e frequenza definite come nel paragrafo 4.3.1. Di seguito è riportato il codice GLSL relativo all'implementazione di tale metodo:

```
float granite3D(vec3 vpos)
{
   float freq=0.0, ampl=0.0,
         turb=0.0,  div=0.0;

   //apply a geometry scale factor
   vpos *= geom_scale;

   for(int i = 0; i < octaves; i++)
   {
      ampl = pow(persistence, float(i));
      freq = pow(2.0, float(i));

      turb += snoise(vpos*freq) * ampl;
      div += ampl;
   }

   //apply contrast + 15%
```

```
    turb = (turb / div) - 0.5;
    turb = (turb * 1.15) + 0.5;

    //increase the white level
    turb += 0.6;

    return clamp(turb, 0.0, 1.0);
}
```

Le coordinate che descrivono il rumore, prima di venire utilizzate per l'elaborazione delle ottave, sono opportunamente scalate per un valore definito esternamente allo shader, in modo da fornire più controllo sull'output. Tale fattore di scala deve essere opportunamente settato sulla base della dimensione della mesh.

Come giustificato durante l'introduzione all'utilizzo di texture procedurali, l'artista è libero di utilizzare valori empirici o fisicamente corretti al fine di ottenere il miglior risultato visivo, o di simulazione fisica, possibile.

Nel caso del granito, si è utilizzato un approccio misto: osservando un pezzo di granito reale ci si rende conto che presenta un elevata frequenza nei dettagli ma scarsamente dettagliata a medi livelli di zoom. Per questo motivo è corretto utilizzare un'alta persistenza, accompagnata da poche ottave. Come osservato, al fine di ottenere un apparenza realistica, si è scelto di usare 3 ottave con una persistenza pari a 0,85.

Per incrementare ulteriormente la qualità finale della visualizzazione, si è ricorsi anche a manipolazioni basate esclusivamente sull'osservazione empirica del risultato finale, aumentando il contrasto del 15% ed il livello medio dei colori di un fattore 0.6.

Il risultato di questo shader, come visibile in Fig.4.6, è particolarmente realistico, sicuramente vicino all'aspetto vero del granito.

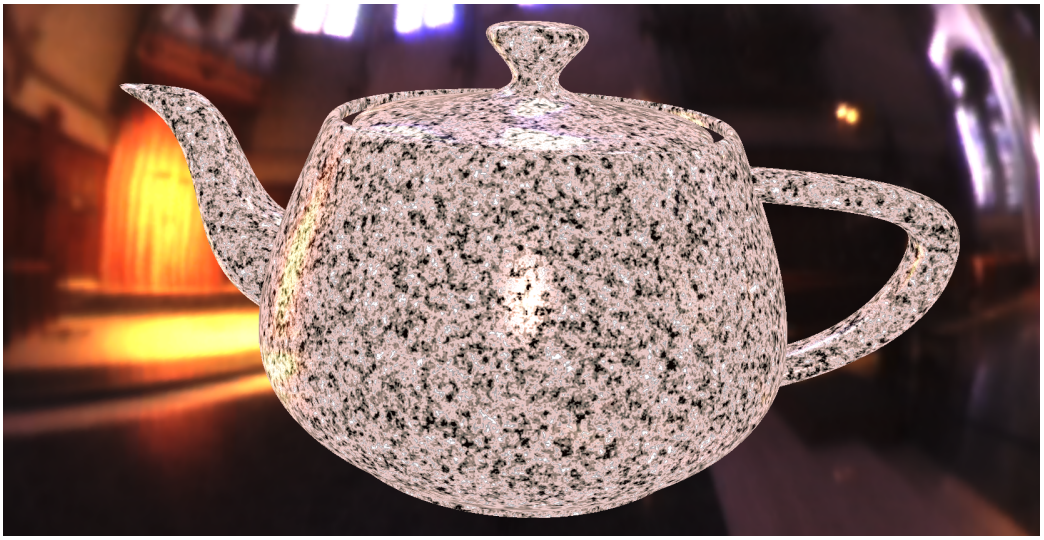


Figura 4.6: *Risultato finale dello shader "Granito" con riflessi ed illuminazione di Ward*

4.5 Conclusioni

In questo capitolo sono state descritte le problematiche di *tiling* relative al texturing, la risoluzione di queste ultime mediante approcci procedurali che consentono di definire l'apparenza di un materiale tramite un algoritmo ed un approccio di generazione procedurale di texture basato sul Perlin Noise.

Durante la realizzazione degli shader analizzati con queste tecniche, ci si è scontrati con il problema della generazione dei numeri casuali in hardware, risolto con un programma di supporto scritto per l'occorrenza. Si è tuttavia ricorso, per superare le limitazioni dell'hardware ed aumentare le prestazioni a leggere i valori di noise da una apposita texture: infatti un'operazione di lookup è sicuramente meno costosa che eseguire una serie di operazioni per la generazione di numeri casuali.

Questo avviene spesso in Computer Grafica; vista l'impossibilità di ese-

guire in tempo reale algoritmi di rendering complessi, come il *raytracing*[6], si ricorre ad approssimazioni e semplificazioni del problema in modo da renderlo più semplice.

La lettura da una texture dello stesso risultato che sarebbe possibile calcolare in tempo reale non porta a differenze qualitative nel risultato, in altri casi è necessario fare assunzioni a priori, in modo da semplificare il problema che altrimenti non sarebbe scomponibile ulteriormente.

È questo il caso dei modelli di illuminazione fin'ora affrontati: il modello di Ward, seppur più complesso ed accurato del classico Phong, risulta pur sempre un approssimazione della *Rendering Equation* che descrive la radianza in ciascun punto della scena. Questi ed altri temi saranno affrontati nel corso del prossimo capitolo, dedicato al *Sub Surface Scattering* applicato mediante Translucent Shadow Maps.

[6]Il raytracing è la tecnica di rendering off-line che viene utilizzata da programmi quali Cinema4D, MaYa, 3D Studio.

Capitolo 5

Translucent Shadow Maps

Molto spesso, l'obiettivo ultimo della Computer Grafica è il raggiungimento di un livello qualitativo che rasenti il più possibile il fotorealismo. Nel caso di applicazioni che non richiedano la visualizzazione interattiva di modelli tridimensionali, è possibile utilizzare tecniche che simulino con precisione matematico-fisica gli eventi del mondo circostante: ci si riferisce a questa casistica con il nome di *offline rendering*. Alcuni esempi possono essere ad esempio il *raytracing*, in cui si modella la luce come raggi che avanzano all'interno della scena effettuando rimbalzi, rifrazioni, subendo attenuazioni *et cetera*. Un altro modo di pensare alla luce e le sue interazioni con la scena viene fornito dalla *Photon Optics*, in cui si applicano i princìpi della meccanica quantistica al fine di descrivere i fenomeni luminosi.

Nel caso di applicazioni in tempo reale tali modelli risultano improponibili, in quanto troppo complessi da calcolare. Partendo da questa constatazione, molti matematici e fisici hanno cercato, nel corso della storia, di elaborare formulazioni più semplici e quindi meno costose, computazionalmente parlando. La strada che si segue in questi casi è quella dell'approssimazione: spesso infatti si assumono determinate ipotesi a priori, in modo da semplificare il problema che altrimenti non sarebbe scomponibile ulteriormente.

I modelli di illuminazione fin'ora affrontati permettevano, ciascuno a diversi livelli di accuratezza, di approssimare in tempo reale solo una parte della *Rendering Equation* [16]. L'equazione di rendering a cui ci si riferisce descrive in modo completo il trasporto dell'energia luminosa ed è data da:

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \Omega)$$

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + \int_\Omega f_r(x, \vec{\omega}_i, \vec{\omega}_r) L_i(x, \vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n}) \mathrm{d}\vec{\omega}_i \qquad (5.1)$$

Mentre il modello di Ward descritto nel precedente capitolo tratta esclusivamente la riflessione della luce (modello riflessivo), il modello di Phong utilizzato dall'OpenGL standard combina linearmente una componente ambientale, una diffusiva ed una speculare. La componente ambientale rappresenta la luce che si assume essere uniformemente incidente sulla mesh da parte dell'ambiente e che viene riflessa in tutte le direzioni in maniera uniforme; la componente diffusiva rappresenta la luce dispersa in ogni direzione; la componente speculare, infine, modella la luce concentrata attorno alla direzione riflessa.

Entrambi i modelli, comunque, risultano inadatti quando applicati a materiali translucidi, come ad esempio la cera o la pelle. In questi materiali, infatti, la luce non viene riflessa completamente ma in parte vi penetra all'interno, esegue dei rimbalzi e quindi esce da una direzione diversa. Questo fenomeno prende il nome di *Subsurface Scattering*.

5.1 Approssimare la Rendering Equation con la BSSRDF

Quando la luce incontra un ostacolo sul suo cammino, può subire una riflessione o essere assorbita. Gli ostacoli possono essere superfici di varia natura

composte da materiali con differenti proprietà fisiche, oppure fluidi in cui le particelle sono molto distanti tra loro, come ad esempio la nebbia o il fuoco. Dal momento che i calcoli necessari per generare l'illuminazione nei fluidi sono computazionalmente molto costose, risulta poco efficiente utilizzare un unico modello volumetrico per tutte le superfici translucide, questo anche se il subsurface scattering è un fenomeno volumetrico. A tal fine si utilizza il modello denominato *Bidirectional Surface Scattering Reflectance Distribution Function* (BSSRDF) che fu proposto da Henrik W. Jensen[17] ed è basato fortemente sulla teoria del trasporto dell'energia luminosa:

$$S(x_i, \vec{\omega}_i, x_o, \vec{\omega}_o) = \frac{\mathrm{d}L_r(x_i, \vec{\omega}_i)}{\mathrm{d}\Phi_i(x_o, \vec{\omega}_o)} \tag{5.2}$$

in cui x_i rappresenta il punto di entrata della luce e $\vec{\omega}_i$ la direzione con cui entra; analogamente, x_o e $\vec{\omega}_o$ identificano il punto di uscita della luce e la sua direzione di uscita.
Una semplificazione del problema della dispersione della luce la si ottiene assumendo la superficie come puramente Lambertiana: il modello Lambertiano, che prende il nome dal suo ideatore Johann Heinrich Lambert, prevede che materiali molto opachi abbiano una superficie composta a livello microscopico da tante piccole sfaccettature. I raggi provenienti da una sorgente luminosa, quindi, riflettendosi su tali micro-facce si disperdono uniformemente in tutte le direzioni, mantenendo però inalterata la radianza[1] in ciascuna di esse.

Molte ricerche sono state focalizzate sullo sviluppo di modelli per la più generica *Bidirectional Reflectance Distribution Function* (BRDF). Il modello BRDF descrive la superficie di un oggetto in funzione della luce riflessa a partire da una sorgente luminosa incidente. Ci sono diversi tipi di BRDF, alcuni semplici come nel caso del modello Lambertiano che produce una BRDF co-

[1]La radianza esprime il flusso totale di energia radiante, per unità di area per angolo solido, nell'unità di tempo.

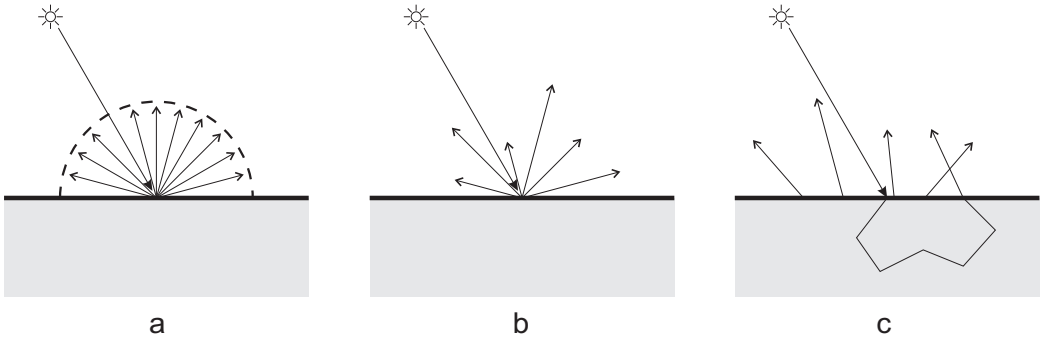


Figura 5.1: *Dispersione della luce in un modello Lambertiano (a), BRDF(b) e BSSRDF(c)*

stante, altri più complessi come visto nel modello di Ward. Assumere, però, che la luce riflessa provenga esclusivamente dal punto in cui essa incide sulla superficie funziona bene per molti materiali, ma non per quelli translucidi.

Quando avviene il fenomeno della dispersione, tutto il processo di riflessione della luce avviene al di sotto della superficie esterna di un dato materiale. Per questo motivo tutti i contributi luminosi, in particolare le ombre, vengono sfuocati, ammorbiditi e nel caso alcune parti dell'oggetto siano poco spesse, divengono translucide. Le funzioni di distribuzione bidirezionale della riflessione (BRDF) non possono riprodurre tali effetti: per questo motivo si ricorre alla BSSRDF (5.2), che descrive esattamente questo tipo di comportamento. E' interessante sottolineare come la BRDF sia un caso particolare ($x_i = x_o$) della BSSRDF.

Nonostante la BSSRDF sia già di per sé un'approssimazione della Rendering Equation (5.1), risulta comunque troppo costosa per essere calcolata in tempo reale da parte di una scheda grafica. A tal fine si è scelto di implementare una tecnica, proposta nel 2003 da C. Dachsbacher e M. Stamminger, che prende il nome di *Translucent Shadow Maps* (TSM).

5.2 La tecnica delle Translucent Shadow Maps

Le *shadow maps*[18] sono un metodo ampiamente utilizzato per aggiungere ombre a scene tridimensionali calcolate in tempo reale, utilizzando semplici textures bidimensionali. Ogni texel[2] di detta shadow map contiene le informazioni di profondità spaziale relative a ciascun vertice rispetto alla posizione della luce. Quest'informazione, come descritto in occasione dell'Ambient Occlusion, descrive la visibilità di un punto: se esso è presente nella shadow map vuol dire che è visibile e quindi riceve luce, altrimenti esso è occluso, quindi in ombra. Il procedimento per determinare la visibilità di un punto arbitrario è riconducibile ad un semplice texture lookup, caratterizzato da complessità computazionale $O(1)$ costante. Il metodo delle shadow maps non viene utilizzato solo nella grafica in tempo reale, ma anche in rendering raytraced da programmi come RenderMan o Cinema4D vista l'efficienza e generalità di cui tale tecnica gode [19].

Nel caso delle Translucent Shadow Maps, le informazioni associate ad una shadow map vengono estese, permettendo tramite un semplice filtro di implementare il fenomeno del subsurface scattering. Un texel della Translucent Shadow Map non memorizza solamente la profondità di un dato pixel su schermo, ma anche l'irradianza[3] dal punto di vista della luce e le normali della superficie calcolate per pixel. L'integrale proposto da Jensen (5.2) può quindi essere calcolato sotto forma di un filtro, dove il peso di ciascun valore è in relazione alle normali ed allo spessore della mesh in ciascun punto. Sfruttando questi accorgimenti, si è in grado di calcolare in tempo reale il subsurface scattering, anche se la tecnica limita all'utilizzo esclusivo di sorgenti luminose parallele.

Dal momento che l'obiettivo ultimo di questo sotto-progetto è la simula-

[2]Texel sta per **TEX**ture **EL**ement ed identifica, appunto, un elemento della texture.

[3]L'irradianza rappresenta l'integrale della radianza incidente lungo tutte le direzioni.

zione della translucentezza del materiale, seguendo le considerazioni di Lensch et al.[20], è lecito considerare esclusivamente il contributo fornito dalla dispersione multipla: la dispersione multipla considera più punti da cui la luce penetra il materiale, di fatto trascurando la direzione da cui essa proviene ed esce, semplificando ulteriormente il problema. Oltre alla dispersione multipla, infatti, esiste il fenomeno della dispersione singola in cui la luce, attraversando un materiale od un fluido, viene dispersa al più una volta. Lo scattering singolo è osservabile ad esempio nella nebbia (le luci appaiono circondate da un alone). Altri materiali, come appunto il marmo, la pelle, il latte, presentano quasi esclusivamente fenomeni di dispersione multipla, subendo quasi per nulla la dispersione singola.

In questa tipologia di materiali, il subsurface scattering può essere suddiviso in tre fasi principali:

- La luce incidente la superficie si disperde nel materiale secondo la legge di Fresnel (5.4).
- Una volta penetrata nel materiale, la luce si diffonde in esso, effettuando dei rimbalzi approssimati dalla funzione di dispersione diffusa della riflessione (5.3).
- Infine, la luce esce da un punto, nuovamente pesata secondo la legge di Fresnel.

Prima fase – Calcolo della Dispersione

Come detto precedentemente, la luce incidente la superficie in un generico punto x_{in} si disperde nel materiale secondo la legge di Fresnel. Ciascun impulso irradiante $I(\omega_{\text{in}})$ proveniente da una luce parallela è descritto semplicemente da:

$$E(x_{\text{in}}) = F_t(\eta, \omega_{\text{in}})||N(x_{\text{in}}) \cdot \omega_{\text{in}}||I(\omega_{\text{in}}) \tag{5.3}$$

in cui η rappresenta la densità ottica del materiale. Il termine F_t è dato da:

$$F_t(\eta, \omega_{\text{in}}) = (2 - N \cdot V)^5 + \mu(1 - N \cdot V)^5 \tag{5.4}$$

che risulta una buona approssimazione della legge di Fresnel, come descritto da C. Schlick [21].

Seconda fase – Diffusione della luce nel materiale

Una volta che la luce è penetra nel materiale, eseguirà dei rimbalzi, diffondendosi in esso. Il processo di diffusione è particolarmente complesso da calcolare, per questo motivo si approssima con la funzione di dispersione diffusa della riflessione $R_d(x_{\text{in}}, x_{\text{out}})$ con x_{in}, $x_{\text{out}} \in S$. R_d descrive il trasporto dell'energia di una sorgente luminosa incidente in x_{in} ed uscente da x_{out}. Questa funzione in quattro dimensioni non solo dipende da $\|x_{\text{out}} - x_{\text{in}}\|$, ma anche dall'angolo fra $x_{\text{out}} - x_{\text{in}}$ e dalla normale in x_{in}:

$$B(x_{\text{out}}) = \int_S E(x_{\text{in}}) R_d(x_{\text{in}}, x_{\text{out}}) \mathrm{d}x_{\text{in}} \tag{5.5}$$

Nella tecnica TSM si assume che il percorso tra x_{in} ed x_{out} sia completamente all'interno dell'oggetto, come avviene nella maggior parte dei casi. Per via di questo assunto, possono presentarsi problemi nel caso di oggetti concavi, anche se in pratica questo tipo di errori risulta spesso visivamente trascurabile. Una schematizzazione grafica è mostrata in Fig.5.2.

Terza Fase – Applicazione dello scattering

La luce, una volta dispersa all'interno del materiale, esce da un generico punto x_{out} pesata nuovamente per il coefficiente F_t secondo la formula:

$$L(x_{\text{out}}, \omega_{\text{out}}) = \frac{1}{\pi} F_t(\eta, \omega_{\text{in}}) B(x_{\text{out}}) \tag{5.6}$$

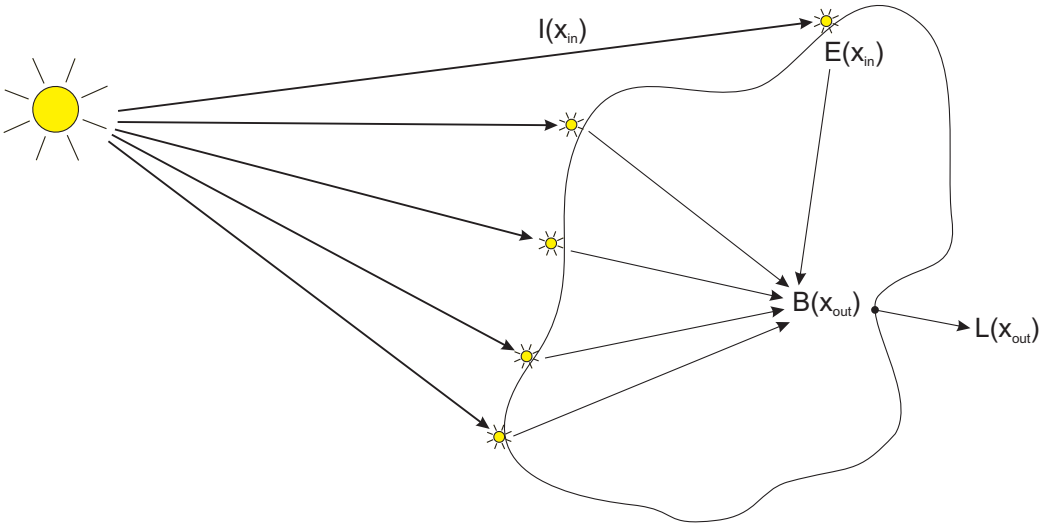


Figura 5.2: *Calcolo della translucentezza mediante integrazione*

5.2.1 Dettagli Implementativi

Dal momento che la tecnica delle Translucent Shadow Map estende il concetto di shadow map, oltre a memorizzare quest'ultima su di una texture, bisogna fare altrettanto con l'irradianza descritta dalla 5.3 e con le normali per-pixel: queste tre componenti saranno ovviamente calcolate dal punto di vista della luce, ovvero in Light Space.

Calcolare la depth map risulta banale in quanto, una volta effettuate le proiezioni in light space nel vertex shader, è sufficiente leggere il valore della coordinata z e scriverlo sull'output del fragment shader. In RenderMonkey, fare in modo che un determinato output venga scritto su di una texture piuttosto che nel framebuffer risulta un operazione immediata. In generale, RenderMonkey consente di comporre l'effetto finale eseguendo più *passate* (cicli) di rendering: in ciascuna di esse è possibile definire uno shader diverso, oltre che poter lavorare su geometrie differenti. Per fare in modo, quindi, che l'output di una determinata passata venga utilizzata da un'altra, si ricorre al

render to texture: per utilizzare tale feature, è sufficiente associare un *render target* di tipo *texture renderizzabile*[4]. In questo modo, la passata relativa al calcolo della depth map scriverà l'output non sul framebuffer, ma su di una texture.

Analogamente, creare una seconda texture su cui memorizzare le normali (normalizzate) per-pixel risulta immediato: una volta impostata come render target una nuova texture, è sufficiente proiettare le normali in light space per poi scriverle come risultato di tale passata. La proiezione in light space si riduce a moltiplicare ciascun vertice per una matrice di rototraslazione in modo da porre il punto di vista in corrispondenza della luce. In questo caso RenderMonkey risulta abbastanza limitante: maggiori informazioni su questo problema saranno fornite nella prossima sezione.

Infine, bisogna calcolare la mappa d'irradianza secondo la (5.3). Come per il calcolo della depth map e delle normali, nel vertex shader si effettuano le proiezioni in light space dei vertici e delle normali. Nel fragment shader, invece, si calcola dapprima il termine F_t secondo quanto riportato nella 5.4, quindi si moltiplica per il coseno dell'angolo tra la normale e la direzione della luce: in questo modo, si attenua l'intensità dell'irradianza in corrispondenza dei bordi della mesh. In ultimo, quanto calcolato sopra viene ulteriormente moltiplicato per l'intensità della sorgente luminosa, fornendo la possibilità di regolare in un secondo momento l'intensità del subsurface scattering.

Le tre textures così generate rappresentano la Translucent Shadow Map che si voleva calcolare. Grazie ad essa è possibile valutare quindi la risposta locale e globale per ogni punto della superficie, indicato con x_{out}, presi dalla visuale della camera. Le coordinate di ciascun punto vengono quindi proiettate in light space, producendo di conseguenza le coordinate $(u, v)^T$ ne-

[4]Una texture renderizzabile in RenderMonkey è analoga ad una qualsiasi texture: l'unica differenza è data dalla possibilità di essere utilizzata come destinazione per le operazioni di rendering

cessarie per leggere dalla Translucent Shadow Map e la distanza d dal punto di luce.

La risposta locale è determinata da due filtri, selezionabili richiamando due funzioni differenti nello shader, uno a 9 punti e un'altro a 21 punti in cui i texel in prossimità di x_{out} vengono usati come input. Dal momento che per ciascun punto bisogna ricalcolare la 5.5, alcune schede video potrebbero non essere in grado di determinare l'output del filtro a 21 punti in maniera fluida: proprio per questo motivo si è scelto di implementare un'altro filtro, più leggero ma caratterizzato comunque da una buona qualità. La radianza incidente sui texel vicini ad x_{in} viene considerata solamente se la loro profondità risiede in prossimità di d. Se, infatti, il valore memorizzato nella depth map è decisamente inferiore di d, lo spessore del materiale è tale per cui il contributo dovuto allo scattering è considerarsi nullo.

Per la risposta globale, bisogna calcolare la luce proveniente dall'esmisfero (rivolto verso l'interno) di x_{out}. Il trasporto dell'energia luminosa che avviene internamente al materiale da x_{in} ad x_{out} è calcolato secondo la formula in Fig.5.3. Come evidenziato, essa dipende dal vettore $R = x_{\text{in}} - x_{\text{out}}$ e dalla normale $N = (n_x, n_y, n_z)^T$ del punto x_{in}. Grazie alla potenza dell'hardware odierno, è possibile eseguire tutte queste operazioni in tempo reale, senza necessità alcuna di precomputare la Translucent Shadow Map. Si provvede quindi, per ciascun punto x_{out}, a sommare tra loro tutti i texel che contribuiscono alla generazione dello scattering, ognuno pesato per un fattore costante, precalcolato in base alla dimensione del filtro ed in nessun modo legato alla differenza di profondità. In questo modo, sia il filtro a 9 che a 21 punti produrranno la stessa intensità di dispersione, con le uniche differenze dovute alla velocità di esecuzione e qualità visiva.

Infine, applicare quanto calcolato risulta di immediata implementazione in quanto è sufficiente eseguire le operazioni indicate in 5.6, pesando nuovamente il contributo dello scattering per il termine F_t.

coefficiente di dispersione ridotto	$\sigma_s^{'}$
coefficiente di assorbimento	σ_a
indice di rifrazione relativo	η
fattore di riflessione di Fresnel	$F_t(\eta, \omega_{\text{in}})$
posizione di entrata ed uscita della luce	$x_{\text{in}}, x_{\text{out}}$
direzione di entrata ed uscita della luce	$\omega_{\text{in}}, \omega_{\text{out}}$
normale della superficie in x_{in}	N_{in}

$$
\begin{aligned}
R_d(x_{\text{in}}, x_{\text{out}}) &= \frac{\alpha^{'}}{4\pi}\left[z_r(\sigma_{tr}d_r+1)\frac{e^{-\sigma_{tr}d_r}}{\sigma_t^{'}d_r^3} + z_v(\sigma_{tr}d_v+1)\frac{e^{-\sigma_{tr}d_v}}{\sigma_t^{'}d_v^3}\right] \\
z_r &= 1/\sigma_t^{'} \quad z_v = z_r + 4AD \\
d_r &= \|x_r - x_{\text{out}}\|, \text{con } x_r = x_{\text{in}} - z_r \cdot N_{\text{in}} \\
d_v &= \|x_v - x_{\text{out}}\|, \text{con } x_v = x_{\text{in}} + z_v \cdot N_{\text{in}} \\
A &= \frac{1+F_{dr}}{1-F_{dr}} \quad D = \frac{1}{3\sigma_t^{'}} \\
F_{dr} &= -\frac{1.440}{\eta^2} + \frac{0.710}{\eta} + 0.668 + 0.0636\eta \\
\sigma_{tr} &= \sqrt{3\sigma_a\sigma_t^{'}} \quad \sigma_t^{'} = \sigma_a + \sigma_s^{'} \quad \alpha^{'} = \frac{\sigma_s^{'}}{\sigma_t^{'}}
\end{aligned}
$$

Figura 5.3: *Quantità ed equazioni che descrivono il termine diffusivo della BSSRDF*

5.2.2 Codice GLSL

Di seguito è riportato il codice GLSL relativo all'implementazione della 5.3. Il metodo così definito viene richiamato più volte, in modo da filtrare la translucent shadow map ed ottenere lo scattering della luce.

```
vec4 multipleScattering (vec4 Xin, vec4 Xout, float lvl)
{
   vec4 finalColor = vec4(0.0);
```

```
/***********
 * irradiance, depth and normals must take
 * into account Xin.xy shifting!
 ***********/
vec4 irradIN = texture2D(Irradiance, Xin.xy,lvl);
vec4 depthIN = texture2D(DepthBuff, Xin.xy,lvl);
vec4 sNormIN = texture2D(SNormals, Xin.xy,lvl);

/***********
 * sigma_a & s both define the light
 * frequencies absorbed by the material
 ***********/
vec4 sigma_a = lightFreqAbsorbed * tsm_freqAbsorption;
vec4 sigma_s = lightFreqAbsorbed * (1.5-tsm_freqAbsorption);

vec4 extinctionC = (sigma_a + sigma_s);
vec4 reduced_albedo = sigma_s / extinctionC;
vec4 effective_extinctionC = sqrt(3.0 * sigma_a * extinctionC);
vec4 D = 1.0/(3.0*extinctionC);

float fresnel_diff = -(1.440/(refr_index*refr_index));
fresnel_diff += (0.710/refr_index);
fresnel_diff += 0.668+(0.0636*refr_index);

float A = (1.0+fresnel_diff)/(1.0-fresnel_diff);

vec4 zr = 1.0/extinctionC;
vec4 zv = zr + 4.0*A*D;
```

```
    vec4 xr = Xin - zr * sNormIN;
    vec4 xv = Xin + zv * sNormIN;

    float dr = length(xr - Xout);
    float dv = length(xv - Xout);

    vec4 f1 = reduced_albedo/(4.0*3.1415296);

    vec4 f2 = zr *( effective_extinctionC * dr + 1.0);
    vec4 f3 = exp (-effective_extinctionC * dr);
    f3 /= (extinctionC * pow(dr,3.0));

    vec4 f4 = zv *( effective_extinctionC * dv + 1.0);
    vec4 f5 = exp (-effective_extinctionC * dv);
    f5 /= (extinctionC * pow(dv,3.0));

    finalColor = f1 * ( f2 * f3 + f4 * f5);

    return irradIN*finalColor;
}
```

5.3 Limitazioni di RenderMonkey

L'ambiente di sviluppo dedicato agli shader di cui si è fatto uso durante il tirocinio è, come già detto, RenderMonkey. Il motivo che ha spinto questa scelta è stato dettato principalmente da due fattori:

- Comodità: in RenderMonkey è estremamente semplice effettuare un render to texture o lavorare con più passate di rendering contempora-

neamente, senza considerare che non bisogna preoccuparsi di scrivere il codice relativo al caricamento di mesh tridimensionali. Ci si può quindi concentrare solo sullo sviluppo degli shaders, tralasciando il resto.

- Assenza di alternative valide: seppur molto comodo, RenderMonkey presenta alcune limitazioni. Purtroppo, non ci sono alternative valide, se non ricorrendo ad ambienti decisamente più complessi e potenti, come ad esempio l'*nVidia FX Composer 2*[22].

Una delle limitazioni presenti in RenderMonkey, che ha influenzato lo sviluppo della tecnica ivi riportata, è stata l'impossibilità di ottenere la matrice Model-View dal punto di vista della luce, necessaria per effettuare le proiezioni in light space. Diverse strade sono state intraprese al fine di risolvere il problema: dapprima si utilizzava una texture 2x2 su cui si scriveva in ciascun texel una riga della matrice Model-View. Questo metodo, tuttavia, presentava un problema non risolvibile senza avere forti errori di approssimazione sui dati scritti: la posizione della camera, infatti, viene scritta sulla quarta colonna della Model-View in maniera non normalizzata, con valori che possono essere anche molto elevati. Pertanto non si è in grado di eseguire il mapping tra 0 ed 1 necessario per convertire la matrice in colori memorizzabili su texture: valori superiori od inferiori verrebbero, pertanto, semplicemente limitati in quell'intervallo.

Si potrebbe normalizzare prima di eseguire la conversione, ma si è scelto di risolvere il problema diversamente: è stata creata una variabile di tipo matrice 4x4 in cui è stato scritto manualmente il valore di ciascun elemento della matrice Model-View dal punto di vista della luce. Purtroppo anche in questo caso non si è riusciti a risolvere completamente il problema della precisione numerica, pur essendo molto vicini ai valori reali che tale matrice dovrebbe avere. A causa di questo problema di precisione, in alcuni particolari casi si possono notare degli artefatti sulla translucent shadow map,

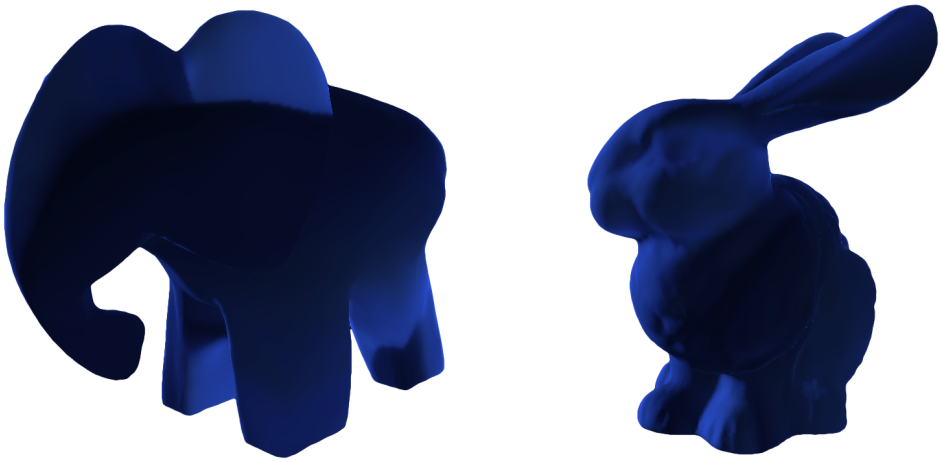


Figura 5.4: *Due modelli illuminati posteriormente che presentano il subsurface scattering*

risolvibili ugualmente modificando leggermente i parametri dello shader o spostando leggermente la posizione della luce.

5.4 Risultati

Applicando la translucent shadow calcolata durante la prima fase dell'algoritmo (5.2) ed applicandola secondo gli algoritmi descritti nelle precedenti sezioni (5.2, 5.2), permette di ottenere quanto mostrato in Fig.5.4.

Analizzando le prestazioni che è possibile ottenere con l'algoritmo descritto, si nota come sia sufficientemente veloce da calcolare, permettendo l'esecuzione in tempo reale su schede di fascia media e medio-alta. Su di un computer con processore AMD Athlon64 X2 2,5 GHz, 2 GByte DDR Ram, scheda video nVidia GeForce 8800 GTS 320 MB e sistema operativo Windows Vista Business le prestazioni ottenute alla risoluzione di 1680x1050 sono riportate nella seguente tabella:

Modello	FPS	n^o Vertici	Filtro (pt)
Teapot	59.0†	39800	9
Teapot	31.0	39800	21
Hebe	58.9†	63932	9
Hebe	29.2	63932	21
Bunny	58.9†	69451	9
Bunny	28.5	69451	21
Elephant	59.9†	13714	9
Elephant	29.9	13714	21

I risultati[5] dimostrano un andamento interessante: dal momento che tutte le operazioni vengono eseguite nel fragment shader, l'algoritmo evidenzia una complessità che varia in maniera direttamente proporzionale con la grandezza dello schermo piuttosto che in base al numero di vertici. Questo è vero nel caso si abbia a disposizione una scheda video particolarmente performante, che non risulta limitata dal numero di vertici su schermo; su GPU meno potenti, il numero di vertici può divenire un fattore determinante. Realizzando un software in C++ sarebbe possibile sfruttare la tecnica dei Vertex Buffer Object, vista in occasione dell'Ambient Occlusion (3.5.4), per aggirare il problema.

[5]I valori contraddistinti dal simbolo † non sono veritieri in quanto limitati dal VSync che in RenderMonkey non è disattivabile in alcun modo.

Capitolo 6

Conclusioni

Durante il capitolo introduttivo era stato proposto, come obiettivo ultimo della relazione, l'unione dei tre sotto progetti per formare un unica pipeline di operazioni che, presa in input una geometria qualsiasi, permettesse si simularne in tempo reale l'apparenza visiva nel caso in cui fosse fatta di un materiale che presenta caratteristiche complesse da un punto di vista visivo, quale il marmo. Mentre la prima fase del tirocinio è stata caratterizzata dalla realizzazione di un plug-in per MeshLab, le ultime due fasi del progetto hanno riguardato l'implementazione in RenderMonkey di shaders, rispettivamente per la simulazione del pattern caratteristico del marmo (e del granito) e della translucentezza che tale materiale presenta. Il punto di unione dei tre sotto progetti sarebbe stato, quindi, MeshLab: da alcuni mesi il laboratorio Visual Computing del CNR di Pisa è al lavoro su di un plug-in che permetta di importare file di progetto realizzati in RenderMonkey, consentendone l'integrazione con tutti gli strumenti disponibili in MeshLab. Sfortunatamente, al termine del periodo di tirocinio, detto plug-in non è stato ancora ultimato: pertanto, l'unica possibilità di unione dei sotto progetti resta all'interno di RenderMonkey.

Vista la natura indipendente degli shaders di marmo e translucentezza, la

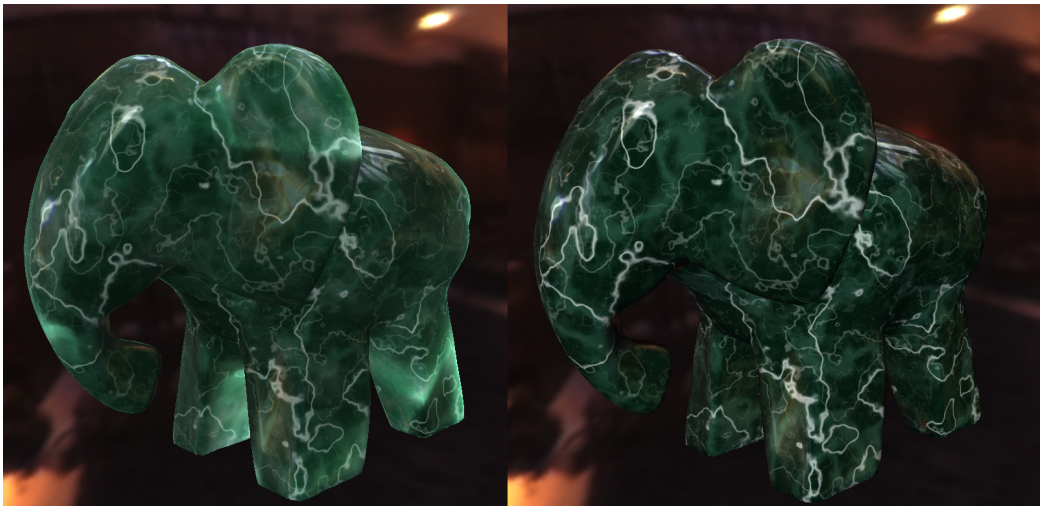


Figura 6.1: *Confronto: un modello con applicatovi lo shader del marmo (destra) e marmo con Translucent Shadow Map (sinistra)*

loro unione è risultata diretta. È stato sufficiente utilizzare il codice per la generazione del marmo all'interno dello shader delle Translucent Shadow Maps, utilizzandone l'output come componente diffusiva. Mediante combinazione lineare sono stati uniti i contributi del marmo e della translucentezza, oltre che dalle riflessioni. Seppur mancante il contributo dell'Ambient Occlusion, il risultato finale risulta particolarmente realistico (Fig.6.2), trasmettendo l'idea di una superficie marmorea "piena", in cui la luce passandovi attraverso aggiunge spessore e morbidezza al modello tridimensionale.

Le prestazioni risultanti dalla combinazione di tali shaders ne permettono l'esecuzione interattiva ad alte risoluzioni, su macchine di fascia alta. Tuttavia, come visto in occasione delle Translucent Shadow Maps, l'algoritmo scala in maniera direttamente proporzionale con la risoluzione dello schermo: riducendola, infatti, è possibile ottenere notevoli incrementi di prestazione che ne consentono l'esecuzione anche su macchine di fascia medio-alta.

Concludendo, nella presente relazione sono state affrontare diverse tecniche avanzate di visualizzazione, ciascuna pensabile come ad un lavoro a

Figura 6.2: *Il risultato finale dello shader del marmo unito alla translucentezza calcolata mediante TSM*

sé stante, ma che in realtà concorrono insieme al raggiungimento di una realistica simulazione visiva per materiali molto utilizzati nell'ambito della Computer Grafica applicata ai Beni Culturali. Gli obiettivi relativi ai sotto-progetti prefissati all'inizio del tirocinio sono stati ampiamente raggiunti ed hanno permesso al candidato di maturare esperienze, conoscenze e competenze avanzate nell'ambito della Computer Grafica in tempo reale. L'esperienza

forse più preziosa è stata tuttavia maturata dal confronto con una realtà lavorativa di alto livello, in ambito di ricerca.

Elenco delle figure

Bibliografia

[1] Bjarne Stroustrup. *C++ - Linguaggio, libreria standard, principi di programmazione*. Addison-Wesley Longman Italia Editoriale s.r.l., Milano, Italia, third edition, 2000.

[2] The C++ Resources Network. cplusplus.com - The C++ Resources Network.
`http://www.cplusplus.com/`.

[3] W3 Consortium. XML.
`http://www.w3.org/XML/`.

[4] Trolltech®. Documentazione QT®.
`http://doc.trolltech.com/`.

[5] Visual Computing Group. VCGlib.
`http://vcg.sourceforge.net/`.

[6] General Purpose GPU. GPGPU.org.
`http://www.gpgpu.org/`.

[7] Edward Angel. *Interactive Computer Graphics, A Top-Down Approach Using OpenGL*. Pearson, Addison-Wesley, Boston, Massachusetts, third edition, 2003.

[8] OpenGL. OpenGL.org.
`http://www.opengl.org/`.

[9] OpenGL Shading Language. OpenGL.org.
`http://www.opengl.org/documentation/glsl/`.

[10] RenderMonkey. AMD in collab. con 3Dlabs.
`http://ati.amd.com/developer/rendermonkey`.

[11] H. H. Buelthoff M.S. Langer. Depth discrimination from shading under diffuse lighting. *Perception*, 29(6):649–660, 2000.

[12] Wikipedia. Framebuffer.
`http://en.wikipedia.org/wiki/Framebuffer`.

[13] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley, first edition, 2004.

[14] K. Musgrave D. Ebert. *Texturing and Modeling: A Procedural Approach*. AP Professional, first edition, September, 1994.

[15] G. J. Ward. *Measuring and modeling anisotropic reflection*, volume 26(2). 1992.

[16] James T. et al. Kajiya. The rendering equation. In *In Computer Graphics' Proceedings of ACM SIGGRAPH 1986*, volume 20(4), pages 143–150, 1986.

[17] Henrik Wann Jensen et al. A practical model for sub-surface light transport. In *In Computer Graphics' Proceedings of ACM SIGGRAPH 2001*, pages 551–518, 2001.

[18] Lance Williams. Casting curved shadows on curved surfaces. In *In Computer Graphics' Proceedings of ACM SIGGRAPH 1978*, volume 12, pages 270–274, 1978.

[19] W. Reeves et al. Rendering anti-aliased shadows with depth maps. In *In Computer Graphics' Proceedings of ACM SIGGRAPH 87*, page 283291, 1987.

[20] Hendrik P. A. Lensch et al. Interactive rendering of translucent objects. In *In Computer Graphics' Proceedings of ACM SIGGRAPH 2002*, pages 214–224, 2002.

[21] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *In Computer Graphics Forum's Proceedings of Eurographics 1994*, volume 13(3), pages 149–162, Oslo, Norway, 194.

[22] FX Composer 2. nVidia. `http://developer.nvidia.com/object/fx_composer_home.html`.

[23] Eric Lengyel. *Mathemathics for 3D Game Programming & Computer Graphics*. Charles River Media, Hingham, Massachusetts, first edition, 2002.