UNIVERSITÀ DI PISA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

STUDIO E REALIZZAZIONE DI UNA LIBRERIA SOFTWARE PER LA VISUALIZZAZIONE INTERATTIVA DI AMBIENTI VIRTUALI COMPLESSI

Tesi di Daniele Giannetti

RELATORI

Prof. Domenici Andrea
Dott. Ing. Tecchia Franco
Prof. Frisoli Antonio

SESSIONE DI LAUREA DEL 17 DICEMBRE 2010

ANNO ACCADEMICO 2009-2010

Dedicato a tutti coloro che mi hanno sostenuto.

Ringraziamenti

Al termine di questi anni passati studiando all'Università di Pisa, sono decisamente soddisfatto del percorso di studi che ho scelto e dovuto affrontare. Non si può dire che sia stato facile arrivare fino a questo punto, tuttavia gli insegnamenti ricevuti mi hanno formato e il bagaglio culturale fornitomi durante il percorso risulterà determinate per il resto della mia vita. Un primo ringraziamento va allora a tutti i docenti e ricercatori che si sono sforzati di rendere i propri insegnamenti veramente efficaci.

Vorrei poi ringraziare tutti i membri del laboratorio PERCRO della Scuola Superiore S. Anna, a partire dall'ing. Franco Tecchia che mi ha dato la possibilità di svolgere questo lavoro di tesi. Inoltre ringrazio Marcello Carrozzino, Sandro Bacinelli e Chiara Evangelista per la gentilezza e la disponibilità con cui mi hanno seguito durante il lavoro, e il prof. Antonio Frisoli per avermi fatto scoprire il laboratorio ed avermi presentato ai membri del gruppo Computer Graphics and Virtual Environments.

Ringrazio i miei genitori che mi hanno saputo sopportare anche quando lo stress mi rendeva intrattabile, mio fratello che è sempre riuscito ad appoggiarmi anche nei momenti più difficili, ed infine tutti gli amici che mi sono stati vicini.

Indice

1 Introduzione **1**

1.1 La Realtà Virtuale 1

1.2 Il Sistema XVR e la VRLib 3

1.3 Scopo della Tesi 5

1.4 Fondamenti di Grafica Tridimensionale 6

1.4.1 Geometria degli Oggetti Virtuali e Mesh 6

1.4.2 Trasformazioni nel Processo di Rendering 10

1.5 Il Sistema Grafico OpenGL 16

1.5.1 OpenGL e Direct3D 17

1.5.2 Contesto ed Estensioni OpenGL 18

1.5.3 Evoluzione del Sistema Grafico OpenGL 19

1.5.4 OpenGL e VRLib 26

1.6 Gli Shader 27

1.7 Fisica in Tempo Reale 32

1.8 Organizzazione del Testo 34

2 Rendering di Oggetti Virtuali **36**

2.1 Illuminazione di Oggetti Virtuali 36

2.1.1 Materiali e Modello di Riflessione di Phong 36

2.1.2 Flat e Smooth Shading 40

2.1.3 Diffuse Texture Mapping 43

2.1.4 Light Mapping 45

2.1.5 Environment Mapping e Image-Based Lighting 47

2.2 Complessità Geometrica e Texture 56

2.2.1 Normal Mapping 56

2.2.2 Displacement Mapping 59
2.3 Trasparenze 62

3 Proiezione di Ombre 66
3.1 Tecniche di Base 67
3.1.1 Shadow Mapping 67
3.1.2 Shadow Volumes 70
3.1.3 Ombre e VR3Lib 73
3.2 Contributi alla Tecnica di Shadow Mapping 74
3.2.1 Shadow Biasing 74
3.2.2 Perspective Shadow Maps (PSM) 76
3.2.3 Cascaded Shadow Maps (CSM) 78
3.2.4 Omnidirectional Shadow Maps 80
3.3 Soft Shadow Mapping 82
3.3.1 Percentage-Closer Filtering (PCF) 83
3.3.2 Percentage-Closer Soft Shadows (PCSS) 85
3.3.3 Variance Shadow Maps (VSM) 87
3.3.4 Exponential Shadow Maps (ESM) 93
3.3.5 Exponential Variance Shadow Maps (EVSM) 97

4 Realizzazione delle Tecniche Presentate 104
4.1 Shading di Base 105
4.2 Diffuse Texture Mapping 109
4.3 Light Mapping 111
4.4 Normal Mapping 113
4.5 Environment Mapping 115
4.6 Displacement Mapping 118
4.7 Shadow Mapping con Tecnica EVSM 125
4.7.1 Costruzione della Shadow Map 126
4.7.2 Filtraggio della Shadow Map 129
4.7.3 Rendering della Scena 133
4.8 Altri Shader 136

5 Struttura e Funzionamento della VR3Lib 138
5.1 Librerie Utilizzate dalla VR3Lib 139
5.2 La Scena e gli Oggetti Virtuali 141
5.3 Gestori e Replicazione di Dati 149

5.4 Gestione degli Shader 152
5.5 Proiezione di Ombre: la Classe VR3ShadowController 155
5.6 Simulazione Fisica e Interazione con PhysX 161
5.7 Rendering di Testo 168
5.8 Un Frame con la Libreria VR3Lib 172
5.9 Funzionalità Rimosse 176
5.10 Il Formato AAM 177
5.10.1 Potenziamento della Specifica AAM 178
5.10.2 L'Esportatore per 3ds Max 180

6 Testing e Analisi delle Prestazioni 183
6.1 Complessità Geometrica 185
6.2 Tecniche basate su Texture 195
6.3 Proiezione di Ombre 203
6.4 Simulazione Fisica 206

7 Conclusioni e Futuri Sviluppi 209
7.1 Conclusioni 209
7.2 Futuri Sviluppi 211

Elenco delle figure

1.1 Alcuni sistemi sviluppati nell'attività del laboratorio PERCRO 3
1.2 La libreria VRLib nell'ambito del framework XVR 4
1.3 Una semplice mesh di triangoli 7
1.4 Poligoni front-facing e back-facing 9
1.5 Trasformazione di proiezione e NDC 13
1.6 Trasformazioni nel processo di rendering 14
1.7 Analogia con una fotocamera 15
1.8 Pipeline OpenGL 1.5 20
1.9 Pipeline OpenGL 3.1 24
1.10 Pipeline OpenGL 3.3 e shader 30

2.1 Proprietà dei materiali e delle sorgenti luminose 40
2.2 Flat shading 41
2.3 Smoothing groups 43
2.4 Texture coordinates 44
2.5 Diffuse texture mapping 45
2.6 Esempio di scena renderizzata con una tecnica di ray tracing 46
2.7 Light mapping 47
2.8 Esempio di sphere map 49
2.9 Esempi di cube environment map 50
2.10 Kernel di un filtro di convoluzione gaussiano 52
2.11 Specular e diffuse cube environment maps 52
2.12 Environment mapping e skybox 53
2.13 Due diversi tipi di normal map 58
2.14 Normal mapping 58

2.15 Esempio di height map e normal map corrispondente 61
2.16 Displacement mapping 62

3.1 Shadow mapping 69
3.2 Shadow volumes 72
3.3 Peter Panning 75
3.4 Perspective shadow mapping 77
3.5 Cascaded shadow mapping 80
3.6 Omnidirectional shadow mapping 81
3.7 Percentage-closer filtering 84
3.8 Percentage-closer soft shadows 86
3.9 Variance shadow mapping 90
3.10 Light bleeding 91
3.11 Light bleeding nella tecnica VSM 92
3.12 Artefatti di bordo con tecnica VSM 93
3.13 Exponential shadow mapping 96
3.14 Light bleeding nella tecnica ESM 97
3.15 Light bleeding nella tecnica EVSM 99
3.16 Artefatti di bordo con tecnica EVSM 102
3.17 Exponential variance shadow mapping 103

4.1 Tassellamento a suddivisione lineare con $TF = 3$ 119

5.1 Classi `VR3Scene`, `VR3Light` e `VR3Camera` 143
5.2 Classi `VR3Obj`, `VR3Mesh`, `VR3Material` e `VR3Texture` 147
5.3 Classe `VR3Scene` (ulteriori relazioni) 148
5.4 Gestori nella VR3Lib 151
5.5 Classi `VR3ShaderProgram`, `VR3ShaderManager` e `VR3FBO` 154
5.6 Classe `VR3ShadowController` 160
5.7 Classi `VR3PhysicsSimulator` e `VR3PhyMesh` 166
5.8 Classe `VR3Text` 170
5.9 GUI del plugin 3ds Max modificato 182

6.1 Mesh utilizzate per i test sulla complessità geometrica (1) 186
6.2 Mesh utilizzate per i test sulla complessità geometrica (2) 187
6.3 Percorso della telecamera 188
6.4 Prestazioni al variare della complessità geometrica - 1024×768 (1) . 190
6.5 Prestazioni al variare della complessità geometrica - 1024×768 (2) . 191

6.6 Prestazioni al variare della complessità geometrica - 1920×1080 (1) . 193
6.7 Prestazioni al variare della complessità geometrica - 1920×1080 (2) . 194
6.8 Confronto di prestazioni al variare della complessità geometrica . . . 195
6.9 Oggetti utilizzati nei test sulle tecniche basate su texture 196
6.10 Prestazioni delle tecniche basate su texture - 1024×768 198
6.11 Prestazioni delle tecniche basate su texture - 1920×1080 199
6.12 Prestazioni della tecnica di light mapping 200
6.13 Oggetto utilizzato nei test sulla tecnica di displacement mapping . . . 201
6.14 Prestazioni della tecnica di displacement mapping al variare di TF . . 202
6.15 Scena utilizzata nei test sulla tecnica di soft shadow mapping 203
6.16 Prestazioni della tecnica di soft shadow mapping 205
6.17 Prestazioni della tecnica di shadow mapping al variare del filter size . 206
6.18 Prestazioni del modulo di simulazione fisica 207

7.1 Un'applicazione costruita con la VR3Lib 210

Elenco degli algoritmi

3.1 Shadow mapping 68
3.2 Shadow volumes 71
3.3 Percentage-closer soft shadow mapping (fragment processing) 85
3.4 Variance shadow mapping 89
3.5 Exponential shadow mapping 95
3.6 Exponential variance shadow mapping 101

Capitolo 1

Introduzione

1.1 La Realtà Virtuale

Quando si parla di *realtà virtuale* (o in modo equivalente di *realtà artificiale*) ci si riferisce ad ambienti che non esistono realmente, costruiti mediante complessi sistemi computerizzati, capaci di fornire degli stimoli agli utenti e di evolvere secondo l'interazione con gli stessi in modo tale da far loro credere di essere parte integrante del mondo virtuale. Gli utenti di un sistema di realtà virtuale vengono proiettati all'interno di ambienti virtuali con cui dovrebbero essere in grado di interagire e da cui dovrebbero ricevere stimoli sensoriali per avere l'impressione di essere effettivamente inseriti nel mondo virtuale; si parla infatti di *esperienza immersiva multisensoriale*. L'utente inserito nell'ambiente virtuale percepisce e interagisce con gli *oggetti virtuali* che lo compongono; un oggetto virtuale corrisponde dunque alla rappresentazione nell'ambiente virtuale di un oggetto che esiste o potrebbe esistere nel mondo reale.

L'ambiente virtuale in cui l'utente di un sistema di realtà virtuale è inserito può essere una riproduzione di un particolare ambiente reale (che esiste nel mondo reale), oppure può essere un ambiente volutamente diverso dalla realtà ma comunque abbastanza convincente da dare all'utente l'impressione di essere parte di esso.

Non sono ancora disponibili sistemi capaci di fornire stimoli sensoriali per tutti e 5 i sensi umani e capaci di generare ambienti virtuali indistinguibili da ambienti reali. Nella pratica, molti sistemi di realtà virtuale non forniscono infatti una risposta sensoriale su tutti i sensi: i videogiochi ad esempio (che in alcuni casi possono essere intesi come sistemi di realtà virtuale) normalmente generano solamente stimoli visivi e uditivi verso l'utente e dunque non sono in grado di fornire un'esperienza immersiva

completa.

I sistemi ad oggi più avanzati sfruttano diverse tecnologie per ottenere esperienze immersive molto più convincenti:

- rappresentazione grafica tridimensionale in tempo reale[1] ad elevato livello di realismo,
- visualizzazione stereoscopica dell'ambiente virtuale (per dare la sensazione di profondità delle immagini),
- rilevamento della posizione e della traiettoria di varie parti del corpo dell'utilizzatore,
- suoni olofonici,
- sensazioni tattili tramite ritorno di forza (force feedback),
- sintesi ed analisi vocale,
- ecc...

Ci si basa allora soprattutto sui sensi di vista e udito, ma con particolari dispositivi (che prendono il nome di *interfacce aptiche*[2]) è possibile anche ricevere sensazioni tattili in risposta alle azioni svolte nell'ambiente virtuale.

La qualità di un sistema di realtà virtuale deriva da diversi fattori:

1. *Quantità di informazione sensoriale generata verso l'utente del sistema*
   Coinvolgendo più sensi nella simulazione e stimolandoli in modo più realistico si ottiene un'esperienza immersiva più coinvincente.

2. *Libertà di controllo sull'ambiente virtuale*
   Quanto più l'utente si sente libero di interagire con l'ambiente virtuale (spostare oggetti, rompere oggetti, ecc...), tanto più immersiva risulterà l'esperienza dello stesso.

3. *Libertà di movimento durante l'utilizzo del sistema*
   Se l'utente non è in grado di ruotare la testa o muovere le braccia durante l'utilizzo del sistema di realtà virtuale, la sua esperienza multisensoriale risulterà

[1] Notare che in questo ambito l'espressione 'tempo reale' non si riferisce alla presenza di vincoli real-time hard o soft gestiti con algoritmi specifici, suggerisce invece semplicemente che l'applicazione deve essere usata in modo interattivo e dunque un importante obiettivo sono le prestazioni.

[2] La parola 'aptico' deriva dal greco 'apto', che significa 'tocco'.

in generale meno convincente rispetto a quella ottenibile con un sistema dove l'utente ha piena libertà di movimento.

Il laboratorio PERCRO[3] della Scuola Superiore S. Anna di Pisa (presso cui è stato svolto il presente lavoro di tesi) opera nell'ambito della realtà virtuale per la costruzione di sistemi immersivi di simulazione di vario genere (alcune fotografie sono riportate in fig. 1.1).

Figura 1.1: Alcuni sistemi sviluppati nell'attività del laboratorio PERCRO

1.2 Il Sistema XVR e la VRLib

XVR (*eXtreme Virtual Reality*) [1] è un framework per lo sviluppo di applicazioni di realtà virtuale; è stato sviluppato presso il laboratorio PERCRO a partire dal 2001 in collaborazione con VRMedia Srl, un'impresa spin-off della Scuola Superiore S. Anna che attualmente gestisce il sistema.

Il sistema XVR è disponibile gratuitamente online insieme ad una serie di tool addizionali che possono far comodo nelle sviluppo di applicazioni tramite questo fra-

[3] PERCRO sta per 'PERCeptual RObotics laboratory'; la missione di questo centro di ricerca è lo sviluppo di nuovi sistemi e tecnologie nell'ambito della realtà virtuale e tele-robotica.

mework. Tale sistema è utilizzato nella realizzazione della maggior parte dei progetti sviluppati presso il laboratorio PERCRO (come quelli in fig. 1.1). Una presentazione della struttura del framework XVR è fuori dallo scopo di questo testo, mettiamo in evidenza solo alcuni punti fondamentali per la comprensione delle tematiche che seguono.

Lo sviluppo di applicazioni di realtà virtuale tramite il sistema XVR passa attraverso la scrittura di script in un linguaggio orientato alla realtà virtuale che è stato concepito appositamente per questo framework, il linguaggio S3D. Tramite questo linguaggio di scripting è possibile costruire in modo semplice anche complesse applicazioni interattive di realtà virtuale. Gli script in linguaggio S3D vengono compilati in un opportuno *bytecode* (come accade in Java) in modo tale da essere facilmente interpretati ed eseguiti su diverse piattaforme.

Il motore che si occupa di interpretare gli script compilati in bytecode ed eseguire le istruzioni che contengono è la *XVR-VM* (*XVR Virtual Machine*). La macchina virtuale contiene il vero nucleo della tecnologia XVR ed ha una struttura modulare capace di caricare moduli esterni con funzionalità aggiuntive. Una delle componenti fondamentali della macchina virtuale XVR è il motore grafico, che viene utilizzato per la visualizzazione in tempo reale di ambienti virtuali tridimensionali. In particolare il motore grafico di XVR si basa sul sistema OpenGL (vedi sez. 1.5) per la visualizzazione di scene tridimensionali. L'accesso ad OpenGL avviene attraverso una potente libreria interna a XVR, che fornisce un'interfaccia molto semplice e in gran parte direttamente corrispondente alle funzioni del linguaggio S3D, la libreria *VRLib* (*Virtual Reality Library*), che rappresenta quindi il componente fondamentale del motore grafico XVR (vedi fig. 1.2).

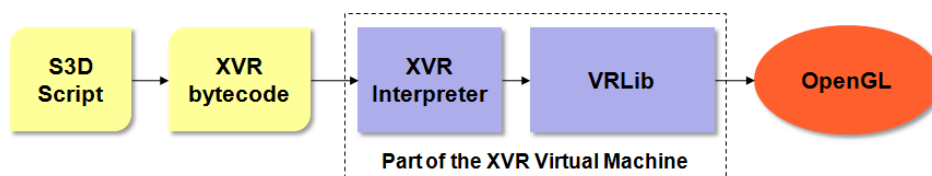


Figura 1.2: La libreria VRLib nell'ambito del framework XVR

Il sistema XVR è stato pensato anche per la realizzazione di applicazioni di realtà virtuale in ambito web: applicazioni realizzate con questo framework (come semplici giochi tridimensionali) possono essere eseguite all'interno di un browser che si occupa di dialogare con la macchina virtuale XVR (scaricata dinamicamente dal client) tramite un'opportuna interfaccia software. Visto che il client si deve

quindi procurare dinamicamente la macchina virtuale con cui eseguire le applicazioni XVR scaricate, è necessario mantenere le dimensioni di questo componente molto contenute; la libreria VRLib (che è uno dei componenti fondamentali della XVR-VM) è stata infatti pensata per avere dimensioni molto ridotte una volta compilata all'interno del motore XVR.

La VRLib inizialmente era pensata come libreria stand-alone per lo sviluppo di applicazioni di grafica tridimensionale in tempo reale; la sua prima versione risale infatti al lontano 1998. Solo successivamente la libreria venne adattata per essere utilizzata nell'ambito del progetto XVR. Nel corso degli anni, fino ad oggi, numerosi cambiamenti sono stati apportati alla VRLib con lo scopo di estenderne le funzionalità mano a mano che il sistema grafico OpenGL e le capacità degli adattatori grafici andavano evolvendosi. Con le ultime versioni di OpenGL, è stato introdotto un cambiamento radicale nel paradigma di programmazione per la grafica tridimensionale (come vedremo in sez. 1.5), e questo significa che per continuare lo sviluppo della libreria VRLib è diventata necessaria una vera e propria riscrittura.

1.3 Scopo della Tesi

Vista la necessità di innovare in modo profondo la libreria VRLib su cui il sistema XVR si basa per visualizzare ambienti virtuali complessi, risulta conveniente progettare la nuova versione integrando anche nuove e diverse funzionalità innovative che precedentemente non erano supportate dalla VRLib in modo nativo.

La tesi ha quindi come scopo fondamentale la costruzione di una nuova libreria per la visualizzazione di ambienti virtuali complessi, comprendente alcune caratteristiche che la differenziano dalla precedente versione:

- Supporto nativo a svariate tecniche avanzate di rendering nate negli ultimi anni, con lo scopo di produrre scene virtuali estremamente realistiche in modo molto semplice.

- Supporto nativo alla simulazione fisica (meccanica) degli oggetti presenti nella scena, in modo tale da integrare all'interno di un singolo modulo software (la nuova libreria) sia gli aspetti di simulazione fisica che quelli prettamente di visualizzazione e grafica tridimensionale.

Mentre allora la VRLib aveva come scopo principale una elevata flessibilità, la nuova libreria (che è stata chiamata *VR3Lib*[4]) ha come scopo fondamentale la *visualizzazione in tempo reale di ambienti ad elevato livello di realismo che evolvono secondo le basilari leggi fisiche in base all'interazione con l'utente.*

Questa nuova libreria nasce come modulo stand-alone per lo sviluppo di applicazioni di grafica tridimensionale e simulazione fisica ad elevato livello di realismo, tuttavia la sua interfaccia (ereditata ed estesa dalla VRLib) ne consentirà in futuro una semplice integrazione all'interno del framework XVR. La libreria VR3Lib è stata sviluppata interamente nel linguaggio di programmazione orientato agli oggetti C++, come anche la precedente VRLib.

1.4 Fondamenti di Grafica Tridimensionale

In questa sezione si introducono alcuni concetti basilari di grafica tridimensionale che verranno richiamati in seguito quando si presenteranno i vari algoritmi utilizzati per ottenere una visualizzazione realistica di ambienti virtuali complessi.

Introduciamo immediatamente il concetto di *artefatto*: nell'ambito della grafica tridimensionale, un artefatto è un effetto indesiderato visibile nella scena virtuale visualizzata che rende evidente il fatto che l'immagine è sintetica. Le tecniche per la visualizzazione di ambienti virtuali ad elevato livello di realismo talvolta introducono artefatti più o meno visibili, e in questo testo ne vedremo alcuni esempi.

1.4.1 Geometria degli Oggetti Virtuali e Mesh

Abbiamo introdotto gli oggetti virtuali come entità che vengono visualizzate e con cui è eventualmente possibile interagire nel contesto della realtà virtuale. Dal punto di vista puramente geometrico, un oggetto virtuale dovrebbe avere una rappresentazione tale da essere quantomeno simile all'oggetto reale (o ideale) che rappresenta all'interno dell'ambiente virtuale. La complessità geometrica degli oggetti reali è tale da proibirne una rappresentazione fedele all'interno della scena virtuale; normalmente si opera infatti fornendo una rappresentazione geometrica approssimativa dell'oggetto reale.

Vi sono diversi approcci per rappresentare oggetti tridimensionali su un calcolatore, ma il metodo dominante (e l'unico a cui facciamo riferimento) prevede di

[4] Il numero 3 si riferisce al numero della versione del sistema grafico OpenGL che ha creato l'esigenza di una profonda rivisitazione della struttura della VRLib: la versione 3.0.

associare ad ogni oggetto virtuale un insieme di poligoni interconnessi che ne approssimano la superficie. Un gruppo di poligoni interconnessi per rappresentare la geometria di un oggetto virtuale prende il nome di *mesh*. I poligoni utilizzati possono essere di vario tipo; nel nostro caso ci limitiamo a considerare mesh di triangoli come quella in fig. 1.3.

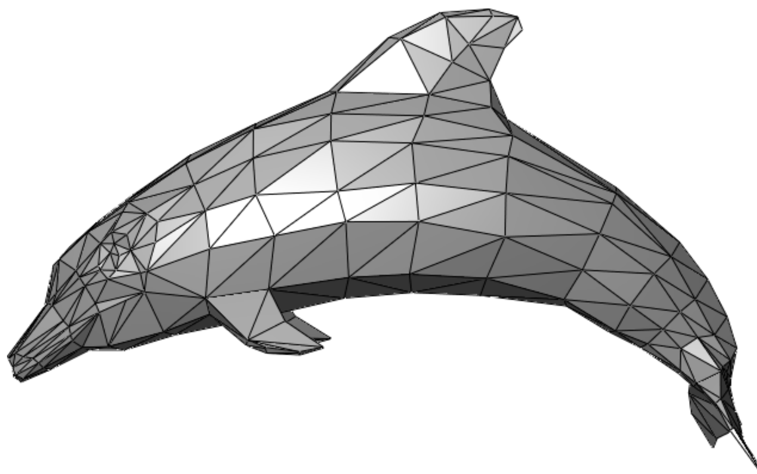


Figura 1.3: Una semplice mesh di triangoli

Naturalmente, rappresentando un oggetto tramite un insieme di triangoli, la geometria risultante sarà solamente un'approssimazione di quella effettiva. Tale approssimazione può risultare più o meno buona in dipendenza dalla forma dell'oggetto reale che stiamo virtualizzando, ma è comunque evidente che aumentando il numero di triangoli utilizzati possiamo ottenere un'approssimazione sempre più fedele all'originale.

Ad una mesh di triangoli deve venir associata una rappresentazione nella memoria del calcolatore, e anche in questo caso si può ricorrere a diverse soluzioni. Per quanto ci riguarda, non è importante il modo in cui una mesh di triangoli viene effettivamente organizzata in memoria e possiamo supporre semplicemente una rappresentazione *face-vertex* (dove ad ogni triangolo nella mesh sono associati 3 riferimenti ai vertici di cui è composto). Nel seguito una mesh verrà intesa semplicemente come una collezione di triangoli che definiscono la superficie dell'oggetto virtuale.

Una mesh occupa quindi in memoria uno spazio che dipende dal numero di poligoni di cui è composta, l'effettiva relazione è funzione del particolare modo in cui la mesh viene organizzata ed esula dagli scopi di questo testo. È evidente dunque che aumentando la fedeltà alla geometria dell'oggetto reale (tramite l'utilizzo di più

poligoni) avremo anche un aumento nello spazio occupato in memoria e di conseguenza un incremento del tempo necessario a elaborare i dati geometrici (operazione che risulterà obbligatoria durante la visualizzazione dell'oggetto reale). Questo è un problema fondamentale nella grafica tridimensionale ed esistono anche tecniche che mirano ad incrementare il realismo di un oggetto virtuale senza aumentare il numero di poligoni di cui è composto (vedi sez. 2.2).

Introduciamo adesso la terminologia che verrà utilizzata nel seguito con riferimento alle mesh:

- *Vertice (vertex)*
  Una posizione all'interno dell'ambiente virtuale; ogni poligono della mesh ha un certo numero di vertici che ne definiscono la forma (3 vertici nel caso di mesh di triangoli).

- *Spigolo (edge)*
  Una connessione tra due vertici che definisce il lato di uno o più poligoni che formano la mesh.

- *Poligono (polygon)*
  Un insieme chiuso di spigoli; costituisce una porzione della superficie della mesh e nel nostro caso si tratta sempre di un triangolo.

È utile stabilire una convenzione che ci permetta di determinare da che parte è orientato un poligono. Questo è conveniente per almeno due motivi:

- senza avere una convenzione che ci suggerisca come è orientato il poligono non è possibile associare allo stesso un vettore normale perchè non sapremmo in che verso proiettarlo (dei due possibili);

- nel caso di mesh chiusa[5] sarebbe utile discriminare i poligoni che non guardano la telecamera, per evitare di elaborarli ulteriormente una volta che sono stati identificati come nascosti.

Il modo classico per decidere tra i due possibili versi si basa sull'ordine dei vertici nella specifica del poligono. Nel nostro caso, supponendo di prendere un semplice triangolo posizionato in uno spazio tridimensionale, possono verificarsi due situazioni (vedi fig. 1.4):

[5] Definiamo chiusa una mesh dove ogni spigolo fa parte di esattamente due triangoli; queste mesh definiscono un volume isolato (a differenza delle mesh aperte).

1. i vertici del triangolo sono ordinati in modo tale che l'ordine di scorrimento dal nostro punto di vista è antiorario; in tal caso il verso della normale è uscente dal poligono (secondo la direzione ortogonale al poligono stesso) e si dice che il poligono è *front-facing;*

2. i vertici del triangolo sono ordinati in modo tale che l'ordine di scorrimento dal nostro punto di vista è orario; in tal caso il verso della normale è entrante nel poligono (secondo la direzione ortogonale al poligono stesso) e si dice che il poligono è *back-facing.*

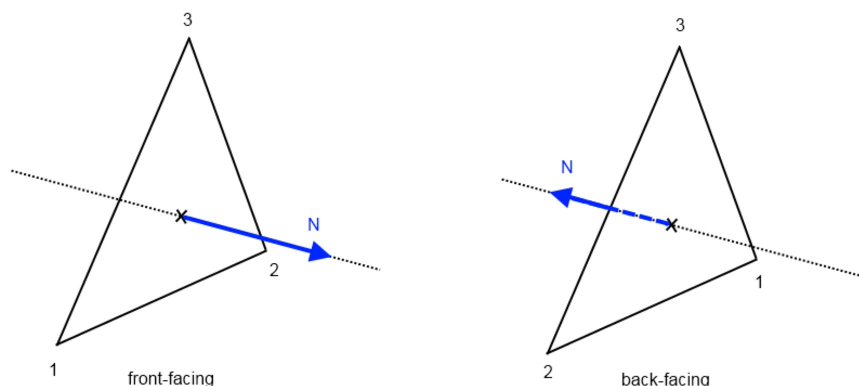


Figura 1.4: Poligoni front-facing e back-facing

Se v_1, v_2 e v_3 sono rispettivamente il primo, il secondo e il terzo vertice di un triangolo, la normale N a questo poligono può essere calcolata come segue:

$$\begin{array}{l} v' = v_2 - v_1 \\ v'' = v_3 - v_1 \end{array} \rightarrow n = v' \times v'' \rightarrow N = \frac{n}{\|n\|}$$

Dove '$\times$' identifica il prodotto vettoriale e $\|n\|$ è la norma euclidea del vettore n.

Nel caso di una mesh chiusa come quella in fig. 1.3, tutti i triangoli dovranno essere tali da avere la loro normale orientata verso l'esterno della mesh; questo garantisce che la normale nei singoli poligoni approssima quella che sarebbe la normale alla superficie dell'oggetto reale. Inoltre tutti i triangoli che risultano back-facing (che non guardano verso l'osservatore) durante il disegno della mesh non saranno visibili perchè coperti da triangoli front-facing più vicini alla telecamera (si deve infatti definire un volume chiuso) e potremo evitare di disegnarli (*back-face culling*).

1.4.2 Trasformazioni nel Processo di Rendering

Col termine '*rendering*' si intende quel complesso processo che, a partire dalla descrizione di un ambiente virtuale presente in qualche forma nella memoria del calcolatore, genera un'immagine della scena virtuale 'fotografata' da un certo punto di vista. Il processo di rendering viene eseguito normalmente da una catena di elaborazione (pipeline) composta da diversi stadi che operano uno alla volta per produrre il risultato finale. Uno stadio può essere visto come un modulo software che svolge una certa funzione su dei dati in ingresso, tuttavia con i moderni adattatori grafici spesso succede che molte delle funzionalità sono realizzate direttamente in hardware sulla scheda video.

Dato che nel nostro caso stiamo parlando di un'applicazione grafica interattiva, il processo di rendering dovrà avvenire in modo continuo. Ogni volta che un frame (una singola immagine, un fotogramma) viene prodotto, comincia il lavoro di rendering per il frame successivo, in un processo continuo dipendente dall'interazione con l'utente. Il *frame rate* è il numero di frame che si riescono a produrre in un secondo (misurato in fps); tale quantità deve essere mantenuta sopra una certa soglia minima per consentire un utilizzo agevole dell'applicazione e per dare l'impressione all'utente che l'immagine sullo schermo evolva in modo continuo. Il valore della soglia dipende dal modo in cui bisogna utilizzare l'applicazione e dalla velocità di evoluzione attesa per la scena virtuale visualizzata. In generale quando si scende sotto circa 30 fps l'utilizzo interattivo comincia a diventare scomodo e l'evoluzione discreta della scena dinamica è facilmente percepibile.

Descrivere in modo completo il processo di rendering esula dagli scopi di questo testo; inoltre tale processo dipende anche dal sistema grafico utilizzato per interagire con l'adattatore grafico. In sez. 1.5 e 1.6 vedremo qualche dettaglio sulla pipeline di rendering utilizzata dal sistema grafico OpenGL, ma non sarà comunque un'analisi completa.

In questo paragrafo vogliamo invece fornire al lettore un'idea precisa su come sia possibile ottenere, a partire dalla rappresentazione della scena all'interno del calcolatore, un'immagine della stessa da un certo punto di vista. In particolare ci concentriamo sulle trasformazioni geometriche che un oggetto tridimensionale deve subire prima che sia possibile disegnarlo a schermo alterando un certo numero di pixel.

La procedura che presentiamo non è l'unica possibile, ma è quella che viene usata all'interno della VRLib e della nuova libreria e dunque è l'unica di nostro interesse.

Abbiamo detto che la geometria di un oggetto virtuale viene rappresentata tramite una mesh, e i vertici della mesh definiscono le posizioni dei poligoni di cui è composta. Ogni vertice è identificato da una posizione in uno spazio tridimensionale, un vettore $v_{obj} = (x_{obj}, y_{obj}, z_{obj})$. Lo spazio tridimensionale all'interno del quale sono normalmente definite le posizioni dei vertici prende il nome di *object space* e le coordinate in questo spazio si dicono *object coordinates*. Si tratta tipicamente di un sistema di riferimento destrorso.

Un oggetto virtuale viene posizionato e orientato all'interno dell'ambiente virtuale tramite una trasformazione detta trasformazione di *modeling*, grazie alla quale possiamo ad esempio spostare una mesh cubica in una data posizione all'interno di un nuovo sistema di coordinate. La trasformazione di modeling viene applicata moltiplicando una matrice di trasformazione per le posizioni di tutti i vertici in object space, trasferendo così i vertici in un nuovo sistema di coordinate destrorso che prende il nome di *world space* (e le coordinate si dicono *world coordinates*).

Normalmente le operazioni vengono svolte lavorando con *coordinate omogenee*: la posizione di un vertice in uno spazio tridimensionale viene rappresentata con un vettore di 4 componenti (x, y, z, w) dove w rappresenta un fattore di scala tale per cui la posizione effettiva nello spazio euclideo è data da $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$. Le matrici usate per le trasformazioni sono quindi matrici quadrate di dimensione 4 e ciò ci consente di rappresentare anche trasformazioni che altrimenti non si potrebbero esprimere con matrici di dimensione 3 (come la traslazione).

Trasformando i vertici in object space tramite la matrice di modeling si ottengono i vertici in world space come segue:

$$\begin{pmatrix} x_{wrld} \\ y_{wrld} \\ z_{wrld} \\ w_{wrld} \end{pmatrix} = M_{modeling} \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ 1 \end{pmatrix}$$

Per poter catturare un'immagine dalla scena virtuale, bisogna poi posizionare la fotocamera virtuale per scattare la foto. La posizione e l'orientamento della fotocamera definiscono una nuova trasformazione che si dice trasformazione di *viewing*. Anche tale trasformazione viene eseguita moltiplicando i vertici per una matrice e si applica subito dopo la modeling transformation. Questa trasformazione porta i vertici in un nuovo sistema di riferimento destrorso che prende il nome di *eye space* (e le coordinate si dicono *eye coordinates*). Solitamente (e così accade nelle VRLib),

in questo nuovo sistema di riferimento la fotocamera è posizionata sempre nel centro e guarda verso l'asse negativo delle z; tutti gli oggetti nella scena saranno stati spostati e ruotati di conseguenza da questa trasformazione.

Trasformando i vertici in world space tramite la matrice di viewing si ottengono i vertici in eye space come segue:

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{viewing} \begin{pmatrix} x_{wrld} \\ y_{wrld} \\ z_{wrld} \\ w_{wrld} \end{pmatrix}$$

Le operazioni applicate fino ad ora sui vertici saranno tipicamente combinazioni di rotazioni, traslazioni e cambiamenti di scala. Spesso accade che le trasformazioni di modeling e viewing vengano applicate tramite un singolo prodotto matriciale sfruttando una matrice di trasformazione combinata che prende il nome di matrice di *modelview*, come segue:

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelview} \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ 1 \end{pmatrix} = (M_{viewing} \cdot M_{modeling}) \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ 1 \end{pmatrix}$$

Posizionati gli oggetti e la fotocamera nella scena virtuale, il prossimo passo è quello di proiettare gli oggetti della scena che risultano di nostro interesse per prepararli al disegno sullo schermo. Vi sono due tipi fondamentali di proiezione: *ortografica* e *prospettica*. Dato che l'apparato visivo umano funziona dandoci una visione prospettica dell'ambiente circostante, ci concentriamo solo sulla seconda tipologia di proiezione. Si definisce allora una nuova matrice di trasformazione che rappresenta la *projection* transformation. Applicando questa trasformazione alle coordinate in eye space si ottengono i vertici in un nuovo sistema di coordinate detto *clip space* (e dunque si parla di *clip coordinates*). Queste nuove coordinate definiscono un sistema di riferimento sinistrorso (a differenza dei precedenti).

La matrice di proiezione prospettica definisce un volume nella scena virtuale che identifica la porzione di spazio che risulterà visibile dopo aver effettuato la trasformazione prospettica. Questo volume prende il nome di *volume di vista* (*view frustum*) e nel caso di proiezione prospettica ha la forma di un tronco di piramide a base rettangolare. Tutto ciò che sta all'esterno del volume di vista non risulterà

visibile nell'immagine disegnata sullo schermo.

Trasformando i vertici in eye space tramite la matrice di proiezione si ottengono i vertici in clip space come segue:

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

Le clip coordinates sono ancora coordinate omogenee. Si opera a questo punto una divisione (chiamata *perspective division*) per trasformare le coordinate in forma cartesiana, producendo le *normalized device coordinates (NDC)*. Il volume di vista è stato dunque trasformato con successo in un cubo centrato nell'origine, allineato con gli assi cartesiani e tale che tutte le coordinate dei suoi vertici hanno modulo unitario (vedi fig. 1.5). Tutto ciò che stava fuori dal volume di vista viene mappato all'esterno del cubo e scartato dal sistema grafico utilizzato. Il sistema di riferimento resta sinistrorso.

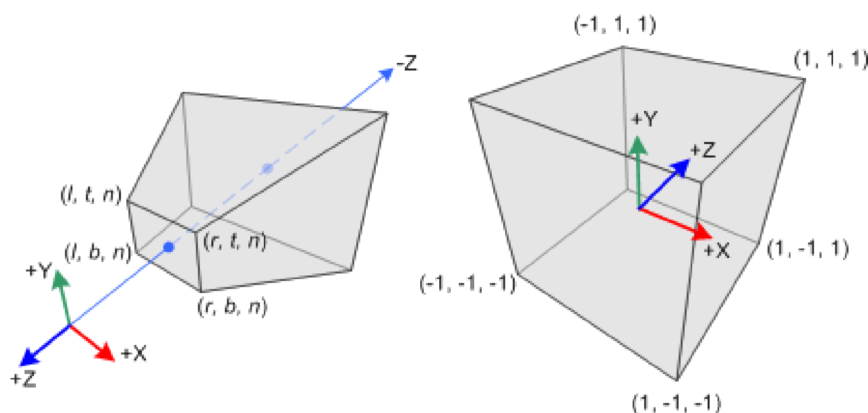


Figura 1.5: Trasformazione di proiezione e NDC

$$\begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

L'ultima trasformazione rimanente prima di poter passare alla generazione effettiva dei pixel da alterare sullo schermo è la trasformazione di *viewport*. Le coordinate dei vertici in NDC vengono tradotte in *window coordinates*, che definiscono finalmente la posizione del vertice all'interno della finestra dell'applicazione (x_{win}, y_{win})

e una profondità z_{win} (normalmente compresa tra 0 ed 1) che può venire utilizzata per evitare che oggetti lontani dalla fotocamera virtuale vengano disegnati su pixels associati ad oggetti più vicini.

La trasformazione di viewport dipende da quello che è il viewport corrente (l'area della finestra dove si sta disegnando), che viene specificato mediante 4 valori interi (x, y, w, h) che rappresentano rispettivamente le coordinate nella finestra dell'angolo in basso a sinistra del viewport, la sua larghezza e altezza (il tutto specificato in pixel).

$$\begin{pmatrix} x_{win} \\ y_{win} \\ z_{win} \end{pmatrix} = \begin{pmatrix} \frac{w}{2} \cdot x_{NDC} + \left(x + \frac{w}{2}\right) \\ \frac{h}{2} \cdot y_{NDC} + \left(y + \frac{h}{2}\right) \\ \frac{1}{2} \cdot z_{NDC} + \frac{1}{2} \end{pmatrix}$$

Le trasformazioni viste possono essere riassunte come in fig. 1.6.

Figura 1.6: Trasformazioni nel processo di rendering

In fig. 1.7 si riassumono nuovamente le trasformazioni necessarie durante il processo di rendering, fornendo anche un'analogia con la situazione reale di voler fotografare un oggetto.

La trasformazione di viewport in una situazione reale potrebbe essere intesa come regolazione delle dimensioni finali della fotografia e posizionamento della foto sul foglio di carta fotografica dove verrà stampata. In alternativa (come in fig. 1.7) potremmo pensarla come posizionamento della foto all'interno dell'area che un proiettore è in grado di visualizzare su un telo bianco.

È importante notare che fino ad ora abbiamo parlato solamente di trasformazioni subite dai vertici, e non abbiamo detto nulla su come è possibile alterare i pixel nella finestra corrispondenti ad intere superfici: queste considerazioni spettano al sistema grafico utilizzato (vedi sez. 1.5) e non ne parliamo in questo paragrafo.

La catena di trasformazioni è stata presentata ad un livello di dettaglio piuttosto spinto ed è valida in generale. La procedura potrebbe cambiare lievemente se si cambia sistema grafico, ma risulterà comunque molto simile. Quanto appena visto è esatto nel caso di OpenGL [2, 3, 5]. Alcune delle trasformazioni viste sopra sono

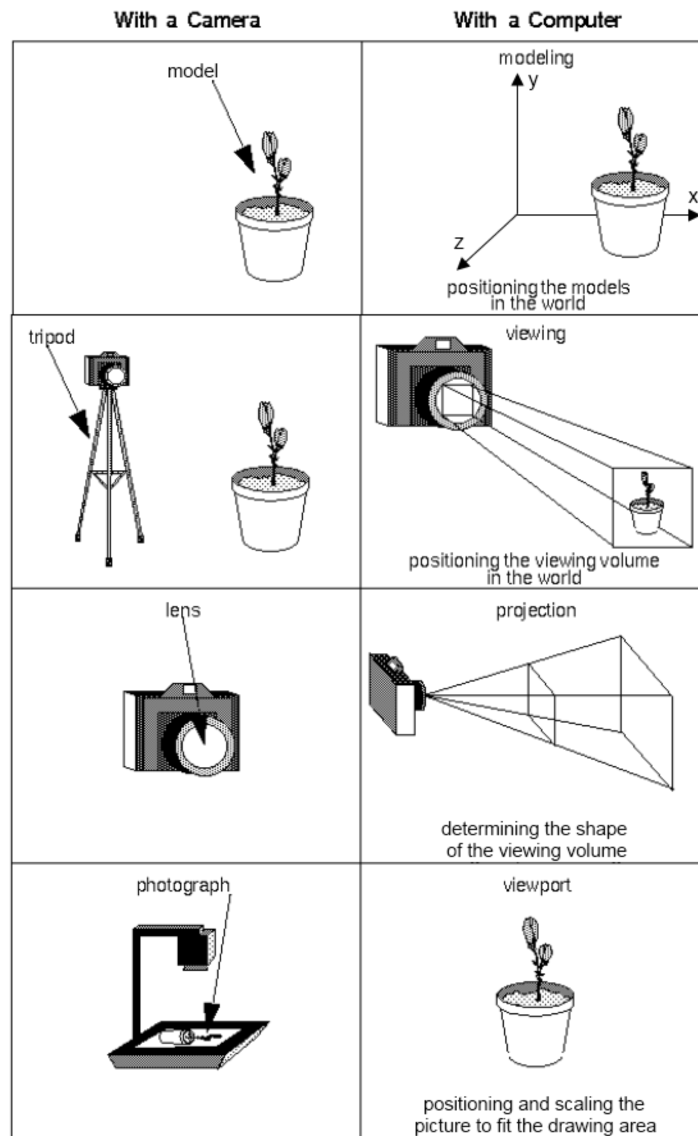


Figura 1.7: Analogia con una fotocamera

a carico dell'utente, altre invece vengono tipicamente eseguite dal sistema grafico in modo automatico (dopo un'opportuna configurazione).

1.5 Il Sistema Grafico OpenGL

OpenGL è il sistema grafico che è stato utilizzato per implementare la versione precedente della VRLib, ed è sempre utilizzando OpenGL che è stata costruita la nuova versione della libreria.

OpenGL va visto come un'interfaccia software verso l'hardware grafico gestito dal sistema operativo, un'insieme di funzioni (e dunque una libreria) che consentono la produzione di immagini a colori di oggetti tridimensionali di elevata qualità. Questa interfaccia software viene tipicamente implementata dai produttori di adattatori grafici perché le operazioni di OpenGL devono operare sull'hardware grafico: questo implica che il supporto alle operazioni utilizzate deve essere implementato dal driver dell'adattatore grafico utilizzato.

Non si hanno particolari requisiti sull'hardware gestito (sostanzialmente l'unico vero requisito è la presenza di un *framebuffer*[6]), si definisce soltanto l'insieme di funzioni che devono venir supportate. Il modo in cui le varie operazioni OpenGL vengono implementate dipende dal particolare adattatore grafico in uso. Se l'hardware disponibile consiste solamente in un framebuffer, praticamente tutte le operazioni del sistema grafico andranno implementate via software ed eseguite sulla CPU. Tipicamente oggi avremo a che fare con hardware grafico dotato di svariati processori in virgola mobile capaci di effettuare trasformazioni su dati geometrici; in tal caso colui che implementa le operazioni OpenGL (che scrive il driver per l'adattatore grafico) dovrà dividere il lavoro tra la CPU e l'hardware grafico disponibile al fine di ottenere le migliori prestazioni possibili durante l'esecuzione.

OpenGL sta per 'Open Graphics Library'. Il sistema si definisce 'open' perché rientra nella categoria degli standard aperti: lo standard OpenGL è disponibile per tutti e può essere utilizzato gratuitamente nello sviluppo di applicazioni grafiche. Le implementazioni del sistema OpenGL sono disponibili gratuitamente e alcune volte sono addirittura open source.

[6] Un framebuffer è una zona di memoria, normalmente gestita direttamente dall'adattatore grafico, che contiene informazioni per ogni singolo pixel visualizzato a schermo. I dati presenti nel framebuffer comprendono i colori che vengono disegnati dal dispositivo di visualizzazione, e solo alterando il contenuto del framebuffer è possibile cambiare l'immagine mostrata.

Le chiamate OpenGL possono venire interpretate con un modello *client-server*: un programma che usa il sistema grafico (il client) lancia dei comandi che vengono interpretati ed eseguiti dall'implementazione OpenGL (il server o server GL).

1.5.1 OpenGL e Direct3D

OpenGL non è l'unico sistema disponibile per lo sviluppo di applicazioni di grafica tridimensionale in tempo reale, ma il famosissimo sistema Microsoft Direct3D è la sola alternativa che viene usata su larga scala proprio come OpenGL. In questo paragrafo discutiamo le differenze tra i due sistemi mettendo in evidenza i motivi per cui la scelta per implementare la nuova libreria ricade su OpenGL.

Direct3D è uno standard proprietario Microsoft sviluppato appositamente per il sistema operativo Microsoft Windows, mentre OpenGL nasce come standard aperto e sue implementazioni sono disponibili nei driver degli adattatori grafici sotto una grande varietà di sistemi operativi (Windows, sistemi Unix-like, ecc...). Direct3D è molto diffuso nel campo dei videogiochi per PC su piattaforma Windows e viene impiegato su larga scala anche nello sviluppo di videogiochi per le console Microsoft Xbox e Xbox 360; OpenGL viene invece tipicamente utilizzato per applicazioni di grafica professionale. L'efficienza relativa dei due sistemi dipende fondamentalmente dall'implementazione, e talvolta i produttori di schede grafiche si sforzano di massimizzare le prestazioni di Direct3D penalizzando lievemente OpenGL in modo da vendere i propri prodotti più facilmente alla preponderante clientela interessata esclusivamente alle performance dei videogiochi su piattaforma Windows.

Nel nostro caso bisogna fare alcune considerazioni:

- anche se la nuova libreria (come la precedente) assume di lavorare in ambiente Windows, future estensioni potranno allargare il supporto ad altri sistemi operativi (a tal proposito la nuova libreria è stata progettata in modo tale da usare il meno possibile funzionalità dipendenti dal sistema operativo);

- la precedente versione della libreria è stata costruita sfruttando OpenGL e dunque cambiare sistema grafico è impensabile in quanto complicherebbe notevolmente l'estensione della nuova VR3Lib con software già pronto per la vecchia versione.

Ne deriva che il sistema grafico scelto per realizzare la nuova libreria è OpenGL.

1.5.2 Contesto ed Estensioni OpenGL

Un'applicazione che sfrutta il sistema grafico OpenGL tipicamente comincia la propria esecuzione costruendo una finestra per la visualizzazione delle immagini all'utente. La creazione della finestra corrisponde anche all'ottenimento di un framebuffer fornito dal gestore delle finestre (dipendente dal sistema operativo). Questo si configura come il buffer dove le chiamate OpenGL andranno a scrivere dei valori per i singoli pixel al fine di modificare quello che è visualizzato a schermo nella finestra dell'applicazione.

La gestione del framebuffer è a carico del gestore delle finestre proprio del sistema operativo. Tipicamente nelle applicazioni OpenGL si sfruttano librerie ulteriori per il dialogo semplificato col gestore delle finestre (e magari per svincolarsi dalla dipendenza dal sistema operativo). OpenGL ha accesso diretto al framebuffer, ma non si cura di come questo servizio venga fornito; potrebbe darsi ad esempio che tale buffer sia gestito tramite una coppia di buffer distinti che vengono usati alternativamente (*double buffering*): mentre su un primo buffer si scrivono delle informazioni grazie alle chiamate OpenGL (*back buffer*), il secondo viene visualizzato a schermo (*front buffer*), e quando si termina la scrittura si esegue uno scambio dei buffer (che potrebbe risultare in una copia del contenuto in determinati casi).

Dopo la creazione della finestra, normalmente si costruisce il *contesto OpenGL*, che corrisponde ad una struttura dati contenente una serie di informazioni sullo stato OpenGL necessarie per l'esecuzione delle chiamate sul server. Il contesto viene normalmente associato con la finestra OpenGL appena creata. Un singolo server GL può gestire diversi contesti, ma un client si dovrà connettere ad un singolo contesto per l'esecuzione dei propri comandi sul server. La creazione di un contesto è anch'essa dipendente dal sistema operativo. Una qualunque chiamata OpenGL descritta nelle specifiche può essere eseguita correttamente solamente quando il contesto è già stato allocato; tipicamente quindi un'applicazione OpenGL portabile sfrutterà i servizi di una libreria esterna (che possa funzionare sotto una molteplicità di sistemi operativi) per la creazione del contesto.

Fin dalle prime versioni di OpenGL, l'interfaccia verso l'hardware grafico è stata continuamente potenziata tramite l'utilizzo di *estensioni*. Le estensioni includono caratteristiche che non sono incluse direttamente nella specifica di base OpenGL e possono venire proposte da diverse entità:

- *Produttori di hardware grafico e altre aziende*

  I produttori possono proporre estensioni OpenGL allo scopo di consentire ac-

cesso tramite interfaccia OpenGL ad alcune funzionalità proprie delle loro schede video. Anche altre aziende possono proporre estensioni, ad esempio per sfruttare funzionalità proprie del sistema operativo.

- *OpenGL Architecture Review Board (ARB)*
  Questo era un consorzio indipendente che governava le specifiche OpenGL e decideva sul rilascio di nuove versioni. Nel 2006 il controllo delle specifiche venne trasferito al gruppo Khronos e si formò all'interno dello stesso un nuovo gruppo ARB (OpenGL ARB Working Group) per continuare lo sviluppo del sistema grafico. L'ARB decide anche quali estensioni possono essere aggiunte alle specifiche di base OpenGL (quando il supporto a tali meccanismi diventa sufficientemente diffuso).

L'utilizzo di un'estensione può comportare l'introduzione di nuove chiamate OpenGL: in tal caso bisogna ottenere gli entry point delle funzioni necessarie dal driver grafico prima di effettuare le chiamate. Per ottenere i puntatori a funzione necessari si utilizzano nuovamente delle chiamate dipendenti dal sistema operativo e dunque conviene normalmente affidarsi ad una libreria capace di funzionare sotto diversi sistemi operativi per la gestione delle estensioni OpenGL: in questo modo avremo a disposizione tutte le funzionalità supportate dalla piattaforma corrente semplicemente inizializzando la libreria.

Bisogna inoltre notare che l'API[7] per accedere alle funzionalità OpenGL (chiamata anche OpenGL ABI, Application Binary Interface) non viene aggiornata da svariati anni sia sotto Windows che sotto altri sistemi (in pratica fornisce direttamente solo le funzionalità delle prime versioni, 1.1 o 1.2 in dipendenza dal sistema operativo). Questo significa che anche per accedere alle funzionalita OpenGL di base per le versioni più recenti bisogna comunque ottenere dei puntatori a funzione dal display driver. Una libreria per la gestione delle estensioni normalmente si occupa anche di ottenere tutti gli entry point per le funzionalità OpenGL di base (in accordo con la versione utilizzata).

1.5.3 Evoluzione del Sistema Grafico OpenGL

La prima versione di OpenGL (versione 1.0) venne rilasciata nel 1992 e le funzionalità all'epoca erano molto limitate. Una trattazione delle prime versioni di OpenGL esula

[7] API significa *Application Programming Interface*, si tratta dell'interfaccia esportata da un componente software per l'utilizzo da parte di altri programmi.

dagli scopi di questa sezione in quanto quello che si desidera mettere in evidenza è soprattutto il modo in cui le nuove versioni di OpenGL hanno reso obsoleta la struttura della vecchia libreria grafica di XVR.

Nel presentare l'evoluzione di OpenGL partiamo quindi dalla versione 1.5 (rilasciata nel 2003).

OpenGL 1.5

Nelle revisioni precedenti alla 1.5 tutte le innovazioni introdotte possono essere considerate ad oggi delle features di base da usare nel rendering. Con la versione 1.5 sono state apportate diverse modifiche alla pipeline OpenGL, ma la cosa più interessante è che con questa revisione sono state introdotte una serie di estensioni per consentire all'utente il caricamento e l'utilizzo di *shader* (programmi scritti dall'utente che si sostituiscono ad alcune funzioni della pipeline OpenGL di base).

Prima della versione 1.5 OpenGL metteva a disposizione dell'utente la sola pipeline 'fissa', in cui le funzioni di ogni stadio sono preimpostate e l'utente può alterarne il funzionamento solamente tramite opportuni parametri di configurazione.

Nonostante questa novità degli shader, la pipeline OpenGL di default rimane quella fissa, che si può schematizzare in modo semplice come in fig. 1.8.

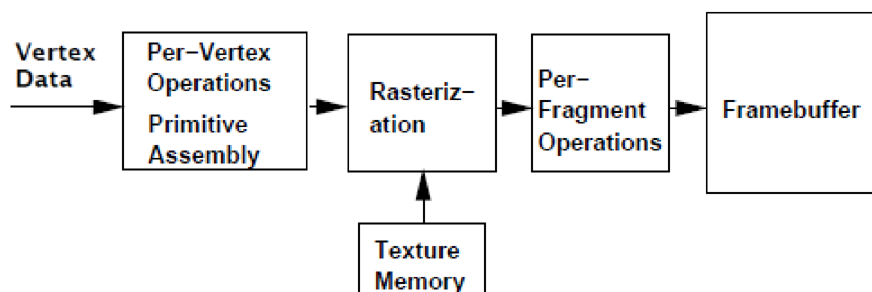


Figura 1.8: Pipeline OpenGL 1.5

La figura rappresenta una versione semplificata della pipeline OpenGL 1.5: in realtà è anche possibile prelevare dei pixel dal framebuffer una volta che sono stati scritti, e utilizzarli per generare dei nuovi dati di ingresso per la pipeline (per cui nella figura manca un ciclo). Trascuriamo inoltre alcuni stadi le cui funzionalità non sono di interesse in questa sezione.

Diamo adesso una breve descrizione dei vari stadi della pipeline fissa:

- *Per-Vertex Operations & Primitive Assembly*
  In questo stadio si ricevono i dati geometrici dei singoli vertici e si opera

trasformando quei dati sulla base delle trasformazioni di modeling, viewing, projection e viewport; si calcolano i parametri di illuminazione dei singoli vertici e infine si assemblano delle vere e proprie primitive a partire dai singoli vertici (poligoni, linee, segmenti, ecc...). Dopo aver applicato la trasformazione di projection, i poligoni risultanti vengono troncati in funzione del volume di vista corrente dell'utente (che dipende dalle trasformazioni stesse). Primitive situate fuori dal volume di vista vengono scartate, mentre primitive che sono in parte interne e in parte esterne vengono modificate producendo nuovi vertici (scartando la parte che giace all'esterno del volume di vista); questa operazione prende il nome di *primitive clipping*.

- *Rasterization*
  Le coordinate dei vertici trasformate come sopra e le descrizioni delle primitive risultanti consentono allo stadio di rasterizzazione di produrre una serie di indirizzi all'interno del framebuffer (spazio di memoria che viene visualizzato a schermo) che corrispondono alle posizioni nel frame buffer delle primitive che si stanno disegnando.

- *Per-Fragment Operations*
  Lo stadio di rasterizzazione produce quindi una serie di frammenti che si portano dietro dei valori con cui alterare il contenuto del frame buffer (per esempio, informazioni di colore). Tuttavia, prima che queste informazioni vengano effettivamente scritte, possono venire eseguite alcune operazioni sui singoli frammenti prodotti causandone l'eliminazione, alterando in diverso modo i valori presenti nel frame buffer, ecc...

- *Framebuffer*
  Questa è la zona di memoria che viene letta per visualizzazione a schermo la scena e contiene i dati per i singoli pixel (almeno quelli di colore). Potrebbe gestire anche altre informazioni oltre a quelle di colore, utili durante le operazioni di OpenGL ma non durante il vero e proprio rendering a schermo (per esempio informazioni di profondità spaziale del frammento). Questo framebuffer viene fornito dal sistema di gestione della finestra su cui OpenGL sta operando, che fondamentalmente dipende dal sistema operativo.

Una trattazione completa delle operazioni svolte da OpenGL esula dagli scopi di questo testo; si rimanda alla bibliografia per tutti i dettagli che in questa sede verranno omessi.

OpenGL 2.0

La versione 2.0 (rilasciata nel 2004) segna l'integrazione del supporto agli shader nelle funzionalità di base di OpenGL. Nonostante ciò, le vecchie funzionalità vengono comunque mantenute e i programmi che usano le versioni precedenti di OpenGL continuano a funzionare senza modifiche. Notare che la pipeline statica OpenGL non subisce modifiche e il funzionamento di default resta quello valido per la versione 1.5.

OpenGL 2.1

In questa revisione del 2006 [2] abbiamo una serie di piccole modifiche e aggiunte che incrementano la potenza del sistema grafico, ma discuterne in questa sede esula dallo scopo del documento. Il funzionamento di base resta identico alla versione 2.0 e la pipeline statica di default di OpenGL 1.5 rimane valida.

OpenGL 3.0

Con la versione 3.0 viene introdotto un grande cambiamento. Nelle versioni precedenti abbiamo assistito alla tendenza crescente di lasciare all'utente la gestione delle funzioni della pipeline GL tramite l'utilizzo di programmi appositi (gli shader). Nella revisione 3.0 del 2008 questa tendenza sfocia in quello che è conosciuto come *Deprecation Model*.

Si cerca di abbandonare del tutto la pipeline fissa in favore di un modello completamente programmabile, dove il programmatore deve sempre ricorrere all'utilizzo degli shader anche per renderizzare la più semplice delle scene.

La pipeline fissa pone dei limiti severi a quello che può fare l'utente in quanto le funzionalità sono modificabili solo nella misura in cui l'API lo permette. Mano a mano che le applicazioni grafiche diventano sempre più complesse, si richiede una maggiore libertà di utilizzo da parte dell'utente. Ad esempio, una delle funzionalità introdotte nelle precedenti versioni di OpenGL è quella di renderizzare delle scene dove si ha della 'nebbia': questa è una caratteristica specifica e indubbiamente interessante, ma avendo a disposizione una pipeline completamente programmabile abbiamo il controllo assoluto sul modo in cui ogni vertice e ogni pixel viene disegnato, e dunque qualsiasi effetto risulta possibile.

Il principale concorrente di OpenGL nel mercato delle applicazioni grafiche è il sistema microsoft Direct3D, che già dal 2007 ha abbandonato la pipeline fissa adottando il cosiddetto *Shader Model 4.0*.

I sistemi grafici come OpenGL e Direct3D definiscono degli standard di interfaccia a cui i produttori di adattatori grafici come Nvidia e ATI devono adattarsi fornendo dei driver capaci di supportare le diverse funzionalità in modo tale che le schede video lavorino correttamente ed efficientemente con un'ampia gamma di applicazioni grafiche. Un cambiamento così radicale nelle API grafiche si riflette anche sul futuro delle schede video, che tende sempre di più verso la realizzazione di processori paralleli altamente programmabili.

Con il deprecation model, in OpenGL 3.0 tutte le funzionalità che risultavano legate alla pipeline fissa sono state marcate come deprecate, evidenziando che nelle successive versioni sarebbero state eliminate. Sono state aggiunte altre caratteristiche interessanti in OpenGL 3.0, al fine di potenziare l'interfaccia verso la pipeline programmabile, ma non ne discutiamo in questa sede.

Tutte le funzionalità OpenGL marcate come deprecate (ma supportate) vengono realizzate internamente tramite l'utilizzo di shader e dunque non si richiede più alcun supporto specifico da parte del driver dell'adattatore grafico.

Oltre ad incrementare la libertà di gestione della pipeline, questa spinta all'utilizzo di shader scritti dal programmatore avvicina lo stesso al modello di esecuzione proprio dell'adattatore grafico e dunque lo forza a programmare in modo più efficiente. Questo naturalmente si riflette in un incremento di complessità nella programmazione OpenGL, che ha reso l'apprendimento dell'API assai più arduo. Tutti i metodi di disegno 'diretto' (dove si specificano uno ad uno i vertici e gli altri dati necessari), che consentivano inizialmente un facile apprendimento dell'API, sono stati deprecati in quanto altamente inefficienti visto l'enorme numero di chiamate OpenGL necessarie per renderizzare una scena complessa. L'unico metodo di disegno che viene mantenuto è quello 'efficiente' che prevede di immagazzinare i dati relativi ai vertici in buffer nella memoria gestita dal server GL (*Vertex Buffer Objects, VBO*), in modo tale che possano venire utilizzati direttamente per il rendering senza doverli trasferire ad ogni singolo frame. Anche il comune utilizzo delle *Display List* (che prevede di compilare un insieme di comandi di disegno diretto per l'esecuzione efficiente ad ogni frame) è stato deprecato, infatti teoricamente le possibilità di ottimizzazione da parte dei driver quando si usano i VBO sono superiori e ci si aspettano migliori performance.

Le caratteristiche deprecate risultano comunque presenti ed utilizzabili nella versione 3.0, tuttavia risulta possibile la creazione di contesti GL *forward compatible* che impediscono l'utilizzo delle funzionalità deprecate.

La struttura della pipeline di alto livello rimane la stessa di OpenGL 1.5, dove tut-

tavia si scoraggia l'utilizzo delle funzionalità fisse in favore di una programmazione esplicita dei seguenti stadi:

- *Per-Vertex Operations*
  Programmato tramite l'utilizzo di opportuni shader detti *vertex shader.*

- *Per-Fragment Operations*
  Programmato tramite l'utilizzo di opportuni shader detti *fragment shader.*

Discuteremo in modo approfondito degli shader in sez. 1.6.

OpenGL 3.1

Come predetto nelle specifiche OpenGL 3.0, con l'arrivo della versione 3.1 (24 Marzo 2009) tutte le caratteristiche che erano state marcate come deprecate sono state rimosse. Questo significa che non si ha compatibilità con applicazioni scritte utilizzando le versioni precedenti dell'API.

Tutte le caratteristiche rimosse sono state spostate in una nuova estensione (`GL_ARB_compatibility`) in modo tale da mantenere il funzionamento di vecchie applicazioni con modifiche minime.

Gli altri cambiamenti che si osservano nella versione 3.1 non sono significativi dal nostro punto di vista.

La pipeline GL assume la forma in fig. 1.9, trascurando comunque alcuni stadi che non sono interessanti in questa trattazione.

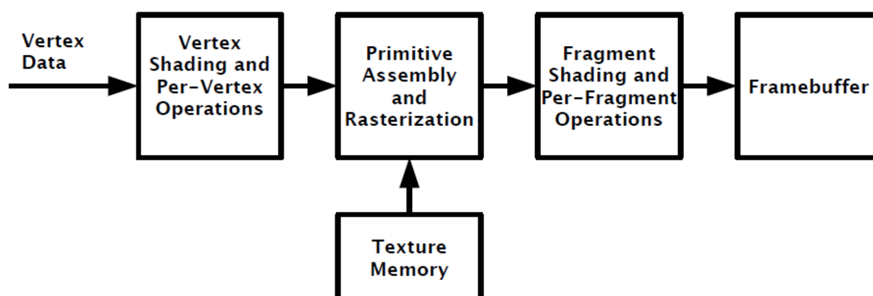


Figura 1.9: Pipeline OpenGL 3.1

OpenGL 3.2

La versione 3.2 del sistema grafico OpenGL venne rilasciata il 3 agosto 2009. In questa versione si rende ancora più evidente la separazione tra il vecchio modello di

programmazione grafica e la nuova pipeline programmabile con l'introduzione dei *profili*. I profili sono sottoinsiemi di funzionalità OpenGL mirate a specifici domini applicativi. In particolare vengono definiti due diversi profili:

- *Core profile*
  Supporta tutte le funzionalità OpenGL non deprecate tramite il deprecation model e tutte le funzionalità che verranno aggiunte nelle successive revisioni. È consigliabile l'utilizzo di questo profilo quando si realizzano nuove applicazioni (in modo tale da non utilizzare funzionalità deprecate).

- *Compatibility profile*
  Supporta tutte le funzionalità OpenGL di tutte le versioni e dunque è il profilo ideale quando si vuole far funzionare vecchie applicazioni con contesti GL più recenti. In particolare, il supporto a questo tipo di profilo è opzionale nelle implementazioni di OpenGL 3.2.

Il tipo di profilo utilizzato viene scelto al momento della creazione del contesto OpenGL.

Un altro aspetto interessante di questa revisione (tra le varie modifiche apportate) è l'integrazione nella specifica di base delle funzionalità relative ai geometry shader. I geometry shader sono un nuovo tipo di shader che modifica le primitive assemblate subito dopo lo stadio di primitive assembly. Consentono la creazione di nuovi vertici, il posizionamento degli stessi, la trasformazione di un tipo di primitiva in un'altra primitiva, ecc... Questi oggetti erano già disponibili sotto forma di estensioni, ma solo con la versione 3.2 sono stati integrati nella specifica di base.

OpenGL 3.3

La versione 3.3 è stata rilasciata nel Marzo 2010. Viene mantenuta la separazione tra i due profili compatibility e core, e vengono aggiunte una serie di features che in questa sede non risultano particolarmente interessanti.

OpenGL 4.0

Sempre nel 2010, contestualmente alla versione 3.3, vennero rilasciate anche le specifiche per la dodicesima revisione (OpenGL 4.0). Si mantiene ancora la separazione tra i due profili compatibility e core, ma si ha un potenziamento generale della API

sulla base delle nuove tecnologie disponibili nelle schede video. Tra le varie innovazioni che si trovano in questa revisione, possiamo evidenziarne due particolarmente significative:

1. Gestione di numeri reali in virgola mobile a precisione doppia negli shader; funzionalità che consente di realizzare algoritmi di rendering sempre più avanzati.
2. Aggiunta di due nuovi stadi programmabili nella pipeline OpenGL, che operano in sequenza immediatamente dopo il vertex shading. Si tratta di due stadi programmabili dove vengono eseguiti programmi che prendono il nome rispettivamente di *tessellation control shader* e *tessellation evaluation shader*. Questi due nuovi stadi hanno come scopo esplicito quello di generare nuove primitive a partire dai dati ricevuti dallo stadio di processing dei vertici, primitive che verranno utilizzate nei successivi stadi della pipeline. Un lavoro di tassellamento può anche venire eseguito lavorando con i geometry shader, ma l'applicazione risulterà in tal caso molto meno efficiente.

OpenGL 4.1

Rilasciate il 25 Luglio 2010 [5], questa è la versione di OpenGL più recente al momento della stesura di questo documento. Non ci interessano tutte le differenze con la versione precedente del sistema grafico; l'unico aspetto che vogliamo sottolineare è che con la versione 4.1 sono state aggiunte alla specifica di base le funzionalità per il salvataggio e successivo caricamento di shader program precompilati e precollegati, pronti per il rendering.

1.5.4 OpenGL e VRLib

Scopo di questo lavoro di tesi è quello di produrre, partendo dalle vecchie librerie grafiche XVR (VRLib) una nuova versione il cui obiettivo sia quello di renderizzare scene interattive realistiche sfruttando le ultime versioni di OpenGL e implementando tecniche di rendering avanzate.

La precedente versione della libreria è stata scritta sfruttando il vecchio paradigma di programmazione OpenGL che prevedeva la pipeline fissa. Le funzionalità della stessa vengono largamente utilizzate all'interno del codice della VRLib, col risultato che gran parte delle caratteristiche utilizzate risultano rimosse considerando le ultime versioni di OpenGL (profilo core).

Questo non implica assolutamente che un motore realizzato sfruttando le vecchie VRLib smetterà di funzionare in futuro: è infatti ragionevole che il supporto alle vecchie versioni di OpenGL verrà mantenuto nei driver rilasciati dai produttori di adattatori grafici (anche per le nuove schede).

Il motivo che ha spinto ad innovare le VRLib è quindi soprattutto quello di adottare il nuovo paradigma di programmazione basato su shader con lo scopo di aumentare l'efficienza di rendering e trarre vantaggio dall'architettura degli adattatori grafici più recenti, introducendo nella libreria tecniche capaci di renderizzare scene ad elevato livello di realismo. Basando inoltre la nuova libreria sulle versioni più recenti del sistema grafico, sarà semplice utilizzare tutte le funzionalità che verranno introdotte in future revisioni di OpenGL.

Inizialmente la VRLib non forniva alcun supporto agli shader e si affidava solamente alle funzionalità della pipeline fissa per il rendering; successive estensioni della libreria hanno poi aggiunto un sistema di gestione degli shader che consente l'utilizzo di tecniche di rendering arbitrarie tramite shader scritti dall'utente della VRLib.

Questa possibilità di estendere le funzionalità della VRLib è stata conservata anche nella nuova versione, implementando un'interfaccia molto simile alla precedente.

Per quanto riguarda le funzionalità OpenGL utilizzate nel nuovo motore grafico, il contesto su cui lavora la libreria deve supportare tutte le chiamate OpenGL 3.3 nel profilo core. Non si utilizza nessuna funzionalità esterna al profilo core e si ha completa compatibilità con le successive versioni di OpenGL.

Il motivo per cui non si richiede un contesto OpenGL di versione 4.0 o 4.1 è che queste ultime due versioni del sistema grafico non sono ancora largamente supportate dai driver degli adattatori grafici, specialmente da quelli con prestazioni medio-basse.

1.6 Gli Shader

Utilizzando il nuovo paradigma di programmazione OpenGL, è evidente l'importanza che assumono gli shader nella realizzazione della nuova VR3Lib. In questa sezione presentiamo dunque alcuni dettagli sull'utilizzo degli shader nella versione 3.3 di OpenGL.

Uno shader è un programma scritto in un apposito linguaggio di programmazione che viene compilato in un eseguibile pensato per l'esecuzione su uno degli stadi della

pipeline grafica usata per il rendering. Le istruzioni di cui è composto uno shader sono tutte pensate per il rendering e normalmente uno shader fa del processing al solo scopo di ottenere determinati effetti sulla scena visualizzata.

Nonostante non sia sempre vero, attualmente l'oggetto prodotto dalla compilazione di uno shader viene normalmente eseguito su dei processori propri della GPU (*Graphics Processing Unit*, parte dell'adattatore grafico) e non all'interno della CPU. Le GPU sono infatti ottimizzate per le operazioni che si trovano tipicamente negli shader e per i calcoli in virgola mobile che sono assai comuni durante il rendering. Una GPU è normalmente in grado di eseguire le operazioni specificate da uno shader su diversi core in parallelo, per processare diversi vertici o pixel allo stesso tempo. Solo raramente uno shader può venire eseguito come un programma ordinario sul processore centrale (CPU), qualora ad esempio delle istruzioni utilizzate non siano supportate dal particolare adattatore grafico: questa situazione si chiama *software fallback* e normalmente risulta in bassi frame rate che impediscono la realizzazione di tecniche di rendering complesse per mantenere la responsiveness dell'applicazione ai comandi dell'utente, dunque è normalmente da evitare.

I linguaggi che possono venire usati per la scrittura di shader sono molteplici, in particolare si dividono in due categorie:

- Linguaggi utilizzati nel *rendering offline*
  Equipaggiati con funzionalità specifiche per ottenere la migliore qualità possibile senza mirare ad elevate prestazioni (utilizzati ad esempio nel tipo di rendering con cui si ottengono i film in computer graphics).

- Linguaggi utilizzati nel *rendering in tempo reale*
  Utilizzati in applicazioni grafiche interattive che devono mantenere un certo frame rate per ottenere risultati soddisfacenti, mirano ad ottimizzare le performance e sono progettati per venire eseguiti direttamente sulla GPU.

Utilizzando OpenGL per realizzare un motore grafico capace di visualizzare ambienti virtuali complessi in modo interattivo, dobbiamo ricorrere alla seconda categoria di linguaggi di shading. Notare che anche in questo caso 'tempo reale' si riferisce semplicemente al fatto che come target principale abbiamo le prestazioni.

Alcuni dei linguaggi più conosciuti in questo ambito sono:

- *OpenGL Shading Language (GLSL)*
  Questo specifico linguaggio è pensato per venire utilizzato con OpenGL e alcune chiamate dell'API sono appunto previste per gestire shader scritti in GLSL.

- *High Level Shader Language (HLSL)*
  Linguaggio proprietario Microsoft pensato per venire utilizzato insieme al sistema grafico Microsoft Direct3D, analogo a GLSL nei confronti di OpenGL.

- *C for Graphics (Cg)*
  Sviluppato da Nvidia in collaborazione con Microsoft, può venire utilizzato sia nel sistema grafico OpenGL che in Direct3D usando librerie apposite distribuite insieme alle specifiche del linguaggio.

Lavorando con OpenGL, e dato che tutti gli shader necessari nella nuova libreria grafica dovranno essere scritti da zero, la scelta ricade su GLSL (più facile da gestire con l'API OpenGL rispetto a Cg).

Le specifiche GLSL, inizialmente approvate dall'OpenGL Architectural Review Board (ARB) come estensione nella versione 1.5 dell'API, sono state inserite nel core OpenGL dalla versione 2.0. Il linguaggio GLSL viene normalmente potenziato e modificato con le nuove revisioni di OpenGL, portando a nuove versioni della specifica GLSL.

Lavorando con la versione 3.3 dell'API OpenGL, gli shader scritti per il funzionamento della VR3Lib useranno la versione 3.30 di GLSL (corrispondente a OpenGL 3.3). Nella versione 3.30, GLSL può venire usato per programmare 3 diversi tipi di processori nella pipeline OpenGL:

1. *Vertex Processor*
   Questo stadio programmabile lavora sui singoli vertici manipolando tutti i dati ad essi associati (ad esempio la posizione). I programmi che possono venire compilati per lavorare su questi processori sono detti vertex shader e quando uno o più vertex shader vengono compilati e collegati insieme si ottiene un *vertex shader executable*, un oggetto che viene eseguito su un vertex processor.

2. *Geometry Processor*
   Questo stadio programmabile lavora su insiemi di vertici assemblati in una primitiva dopo il processing da parte dei vertex shader. I programmi che possono venire compilati per lavorare su questi processori sono detti geometry shader e quando uno o più geometry shader vengono compilati e collegati insieme si ottiene un *geometry shader executable*, un oggetto che viene eseguito su un geometry processor.

3. *Fragment Processor*
   Questo stadio programmabile lavora sui singoli pixel dopo lo stadio di raste-

rizzazione. I programmi che possono venire compilati per lavorare su questi processori sono detti fragment shader e quando uno o più fragment shader vengono compilati e collegati insieme si ottiene un *fragment shader executable*, un oggetto che viene eseguito su un fragment processor. La parola 'fragment' ha in quest'ottica lo stesso significato di 'pixel', anche se bisogna ricordare che non è detto che i frammenti processati da un fragment processor arrivino nel framebuffer (diventando effettivamente pixel sullo schermo).

I 3 diversi tipi di eseguibili vengono normalmente assemblati in un unico oggetto che si dice *shader program*. Mentre la presenza di un vertex shader executable e di un fragment shader executable è obbligatoria per qualsiasi rendering, non è obbligatorio prevedere un geometry shader executable (dunque si può fare a meno dei geometry shader qualora le primitive assemblate non debbano essere modificate).

Per chiarire il funzionamento, la pipeline OpenGL 3.3 può venire rappresentata come in fig. 1.10, dove si ha un livello di dettaglio superiore rispetto alle precedenti illustrazioni, ma abbiamo al solito introdotto alcune semplificazioni.

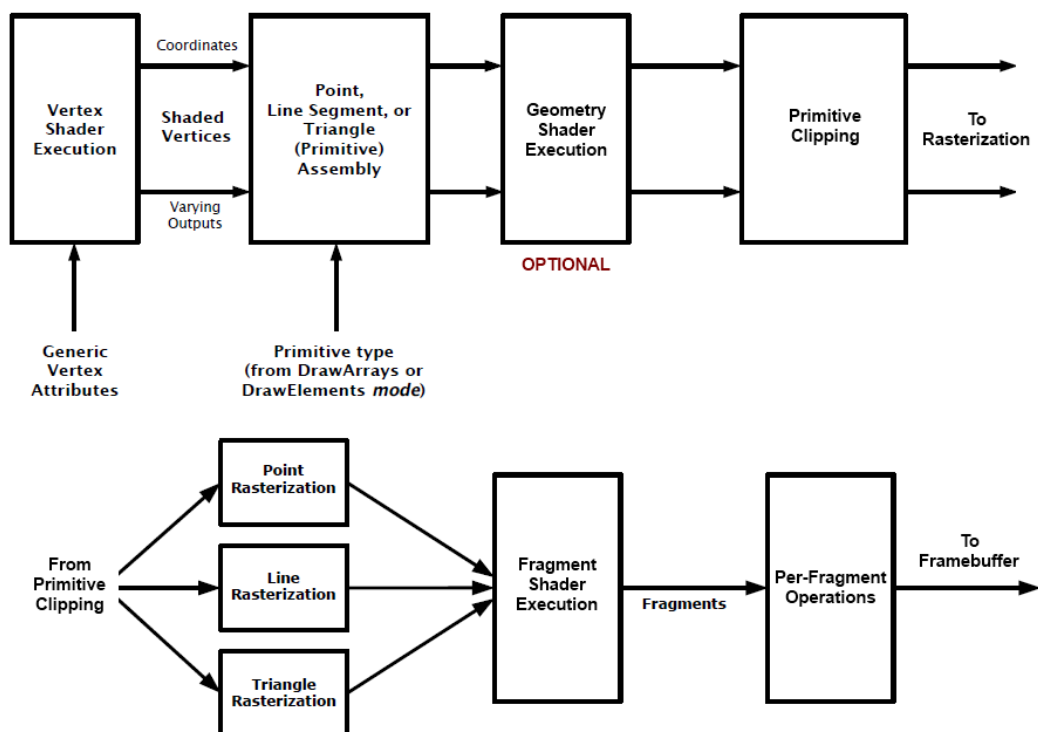


Figura 1.10: Pipeline OpenGL 3.3 e shader

Notare che non è facile determinare a priori il numero di esecuzioni di ogni tipo di shader executable pur sapendo cosa si vuole disegnare.

Supponendo infatti di voler disegnare un triangolo come una singola primitiva, sfruttando anche un geometry shader che sposta i 3 vertici dello stesso, abbiamo che:

- il vertex shader executable dovrà venire eseguito 3 volte (una volta per ogni vertice);
- il geometry shader executable dovrà venire eseguito 1 singola volta (una volta per ogni primitiva);
- il fragment shader executable dovrà venire eseguito un numero di volte pari al numero di pixel nel framebuffer che risultano interessati dalla primitiva in esame dopo lo stadio di rasterizzazione (potenzialmente migliaia).

Ogni tipo di eseguibile opera su ingressi diversi e inoltra allo stadio successivo il risultato del processing:

- Il vertex shader executable opera sui dati dei vertici in ingresso provenienti da buffer predisposti dall'utente; in generale si utilizzeranno dei VBO per immagazzinare i dati che verranno processati dal vertex processor. L'eseguibile inoltre produce in uscita dei dati relativi ai singoli vertici che vengono consumati dal successivo stadio della pipeline.
- Il geometry shader executable opera su dati relativi a intere primitive provenienti dallo stadio di assemblaggio delle primitive. In uscita abbiamo invece nuovi dati relativi alle primitive trasformate dal geometry shader.
- Il fragment shader executable opera su dati provenienti dallo stadio di rasterizzazione sui singoli frammenti. In uscita abbiamo dati relativi ai frammenti processati (informazioni di colore, profondità e altro) che subiranno altre operazioni prima di finire eventualmente nel framebuffer.

L'unico eseguibile su cui l'utente ha controllo diretto per quanto riguarda i dati di ingresso è allora il vertex shader executable.

Ogni tipo di shader può passare agli shader che lavorano negli stadi successivi delle variabili che prendono il nome di *varying variables*, queste possono venire trasformate in vario modo passando da uno shader ad un altro. Ad esempio, un vertex shader può definire una variabile varying di uscita che contiene per ogni vertice processato la somma delle 3 componenti spaziali della posizione del vertice; supponendo che lo stadio di geometry shading venga bypassato perché non è previsto

un geometry shader executable, il fragment shader riceverà dei valori su una variabile di ingresso varying corrispondente a quella in uscita dal vertex shader. I valori per i singoli frammenti verranno prodotti a partire da quelli dei vertici nello stadio di rasterizzazione e in un modo dipendente da come l'utente ha scritto gli shader. Le specifiche GLSL definiscono anche un set di variabili varying predefinite per ogni tipo di shader, che possono venire lette o modificate durante le operazioni (influenzando magari il comportamento degli stadi successivi nella pipeline).

Un altro tipo di variabili che possono essere accedute in sola lettura durante l'esecuzione dello shader program sono le *uniform variables*. Queste variabili vengono impostate dall'utente prima di lanciare in esecuzione il programma e mantengono il valore fino a che lo shader program non subisce un nuovo collegamento (linking).

Per quanto riguarda le trasformazioni effettuate automaticamente dal sistema grafico OpenGL, le trasformazioni di modeling, viewing e projection sono a carico dell'utente e vanno eseguite all'interno del vertex shader. Le altre trasformazioni (perspective division e viewport transformation) vengono invece effettuate automaticamente dal sistema grafico prima di arrivare allo stadio di fragment processing.

1.7 Fisica in Tempo Reale

La nuova libreria costruita durante il lavoro di tesi deve integrare (oltre agli aspetti di visualizzazione in tempo reale di ambienti virtuali ad elevato livello di realismo) anche le funzionalità di simulazione fisica degli oggetti presenti nell'ambiente virtuale. Quello che si vuole ottenere è un'interazione fisicamente realistica tra gli oggetti presenti nella scena virtuale. I risultati devono essere convincenti ma non è richiesta una simulazione particolarmente accurata dell'evoluzione meccanica degli oggetti.

Naturalmente quando si parla di simulazione fisica in applicazioni di realtà virtuale si intendono interazioni prettamente meccaniche tra gli oggetti presenti nell'ambiente virtuale. Nella realtà le interazioni meccaniche tra diversi oggetti sono spesso non banali e, nonostante in molti casi ci si possa limitare a considerare la fisica dei corpi rigidi, talvolta bisogna ricorrere a concetti avanzati di fisica meccanica (ad esempio per spiegare il comportamento dei fluidi).

Volendo integrare nella VR3Lib anche un modulo per la simulazione fisica degli oggetti nell'ambiente virtuale, dobbiamo compiere una scelta fondamentale tra due alternative:

1. Realizzare il modulo di simulazione partendo da concetti di fisica meccanica e implementando i vari aspetti direttamente nella nuova libreria.

2. Sfruttare un sistema di simulazione già disponibile e implementare il modulo di simulazione nella VR3Lib come una semplice interfaccia verso il sistema precostituito.

Naturalmente la prima alternativa richiede uno sforzo molto superiore, dovendo affrontare tematiche avanzate di fisica meccanica e aspetti di simulazione discreta attraverso un calcolatore. Inoltre bisogna evidenziare che realizzare un motore di simulazione fisica capace di ottenere un comportamento realistico senza instabilità nella simulazione è tutt'altro che banale e normalmente si richiede comunque la regolazione di alcuni parametri da parte dell'utente finale.

Per ottenere una simulazione fisica flessibile e affidabile, la soluzione migliore nel caso della VR3Lib è quella di sfruttare un motore precostituito piuttosto che ricorrere ad una soluzione fatta in casa. Affrontando il problema di scegliere il motore da utilizzare, sono disponibili molte alternative:

- *PhysX*
  Motore fisico gestito dalla Nvidia, di provata affidabilità, efficienza e stabilità.

- *Havok*
  Motore fisico molto popolare la cui compagnia produttrice (Havok) è stata acquisita dalla Intel nel 2007, ancora oggi è tra i più utilizzati.

- *Bullet*
  Motore fisico indipendente ed open source, meno utilizzato rispetto ai precedenti ma comunque piuttosto comune.

- ...

I sistemi sopra sono i più utilizzati e sono tutte soluzioni gratuite, ma esistono anche altre alternative che non trattiamo in questo testo. In passato è stato costruito un modulo XVR per l'accesso alle funzionalità fornite dal motore fisico Nvidia PhysX per la simulazione di corpi rigidi (*PhysXVR*). Visto che PhysX è stato sfruttato in passato all'interno del progetto XVR e considerato che si tratta di una piattaforma molto comune e di provata affidabilità e stabilità, si è deciso di utilizzarlo per la simulazione fisica degli oggetti virtuali all'interno della VR3Lib.

La nuova libreria integrerà pertanto un modulo di simulazione fisica che si occupa di dialogare con PhysX e che dunque opera semplicemente come un'interfaccia verso i servizi offerti dal motore Nvidia.

PhysX fornisce svariate funzionalità per la simulazione di oggetti virtuali, può simulare corpi rigidi, corpi molli, fluidi e tessuti, inoltre può sfruttare alcune GPU per spostare porzioni di simulazione sull'architettura ad elevato parallelismo dell'adattatore grafico. Questa tendenza a sfruttare l'architettura parallela della GPU per la simulazione fisica oltre che per il rendering di ambienti virtuali è comune a tutti i motori visti sopra. Nella attuale versione della nuova libreria è stato integrato solamente il supporto alla simulazione di corpi rigidi tramite PhysX, in futuro si prevede di estendere la struttura per la simulazione di oggetti più complessi.

1.8 Organizzazione del Testo

I capitoli rimanenti di questo testo sono organizzati come segue:

Capitolo 2 *Rendering di Oggetti Virtuali*
Si presentano le tecniche di base e alcune tecniche avanzate per la visualizzazione e l'illuminazione realistica di oggetti virtuali.

Capitolo 3 *Proiezione di Ombre*
Partendo da una presentazione delle tecniche di base per la proiezione di ombre di oggetti virtuali su altri oggetti virtuali in applicazioni di rendering in tempo reale, vediamo alcuni passi significativi nell'evoluzione di queste tecniche fino ai metodi più innovativi attualmente disponibili.

Capitolo 4 *Realizzazione delle Tecniche Presentate*
Si fornisce una visione semplificata di come le tecniche viste nei capitoli 2 e 3 possano venire implementate mediante la costruzione di shader in linguaggio GLSL.

Capitolo 5 *Struttura e Funzionamento della VR3Lib*
Si illustrano alcuni dettagli sulla struttura della nuova libreria in termini di moduli e classi; si forniscono inoltre alcuni chiarimenti sul funzionamento di base del ciclo di rendering in un'applicazione che sfrutta la VR3Lib.

Capitolo 6 *Testing e Analisi delle Prestazioni*
Si misurano le prestazioni della libreria ottenuta prendendo a riferimento alcu-

ne situazioni notevoli e mettendo in evidenza l'impatto delle varie funzionalità supportate sul tempo necessario a tracciare un singolo fotogramma.

Capitolo 7 *Conclusioni e Futuri Sviluppi*
Si riassumono i risultati raggiunti e si propongono alcune direzioni per futuri lavori di estensione della libreria.

Capitolo 2

Rendering di Oggetti Virtuali

Nel capitolo 1, abbiamo introdotto il sistema grafico OpenGL e alcune tematiche di grafica tridimensionale. In questo capitolo vediamo inizialmente alcune tecniche di base per la visualizzazione di oggetti virtuali, per poi introdurre tecniche avanzate che hanno lo scopo di rendere la scena più 'convincente'.

2.1 Illuminazione di Oggetti Virtuali

In questa sezione trattiamo un aspetto fondamentale nella visualizzazione realistica di scene virtuali: l'illuminazione di oggetti virtuali. La presentazione comincia con algoritmi di base molto datati ma che comunque sono ancora i modelli utilizzati in combinazione con le tecniche più recenti.

Abbiamo già visto nel capitolo 1 che da un punto di vista geometrico un oggetto virtuale può essere descritto tramite una mesh. Ricordiamo che una mesh viene intesa come una collezione di triangoli che definiscono la struttura di un oggetto virtuale. Le tecniche presentate in questa sezione sono valide indipendentemente da come vengono rappresentate le mesh associate agli oggetti virtuali nella memoria del calcolatore.

2.1.1 Materiali e Modello di Riflessione di Phong

Quando bisogna visualizzare un oggetto virtuale, alla mesh viene normalmente associato un materiale. Mentre la mesh descrive le proprietà geometriche di un oggetto, il materiale definisce come l'oggetto reagirà se illuminato da una certa sorgente luminosa.

Il caso più semplice si ha quando un oggetto non viene illuminato da alcuna sorgente luminosa (e non emette luce propria): in tal caso l'oggetto risulterà completamente nero. Supponendo ad esempio di stare in una camera completamente buia, ogni oggetto apparirà nero indipendentemente da quello che noi consideriamo essere il suo colore.

Quando però nella realtà gli oggetti vengono illuminati da alcune sorgenti luminose, l'occhio umano percepisce delle informazioni aggiuntive: riusciamo ad associare ad ogni oggetto un colore, siamo in grado di dire se un oggetto riflette o meno della luce e in che misura, e possiamo addirittura riconoscere oggetti che risultano in qualche modo trasparenti perché il nostro occhio riceve raggi luminosi riflessi da oggetti situati dietro l'oggetto trasparente.

In computer graphics tutte queste proprietà di risposta di un oggetto alla luce sono racchiuse nel concetto di materiale associato ad una mesh.

I modelli che sono stati proposti per rappresentare le proprietà di un materiale e utilizzarle per calcolare il colore dei punti di oggetti virtuali sono diversi [9, 10]. Nel nostro caso utilizziamo un modello molto comune presente anche nella vecchia VRLib: il *modello di riflessione di Phong* [9].

Un materiale è definito dal seguente insieme di proprietà:

1. E_M - *material emission color*
   Vettore di 3 componenti positive che indica la quantità di luce emessa per ogni componente di colore (R, G, B) in ogni direzione. Per oggetti che non emettono luce, tutte le componenti di questo termine saranno nulle. Il colore emissivo non è presente nella versione di base del modello di Phong, ma risulta comunque utile nella pratica.

2. A_M - *material ambient color*
   Vettore di 3 componenti positive che indica la quantità di luce riflessa per ogni componente di colore (R, G, B) per quanto riguarda la luce ambientale (che arriva sull'oggetto da ogni direzione e viene riflessa in ogni direzione).

3. D_M - *material diffuse color*
   Vettore di 3 componenti positive che indica la quantità di luce riflessa per ogni componente di colore (R, G, B) per quanto riguarda la luce che arriva da una sorgente. La luce in questo caso viene riflessa in ogni direzione (*riflessione Lambertiana*).

4. S_M - *material specular color*
   Vettore di 3 componenti positive che indica la quantità di luce riflessa per ogni componente di colore (R, G, B) per quanto riguarda la luce che arriva da una sorgente. La luce in questo caso viene riflessa in una singola direzione (come la riflessione di un singolo raggio su uno specchio, secondo la popolare *legge di riflessione*).

5. n - *material shininess*
   Coefficiente reale positivo che assume valori tanto superiori quanto più la superficie è liscia, maggiore è il suo valore e minori saranno le dimensioni della sorgente luminosa riflessa sull'oggetto.

Esamineremo più avanti il problema di materiali con un certo grado di trasparenza.

Il modello prescrive anche che ad ogni sorgente luminosa vengano associate alcune proprietà, in particolare abbiamo:

1. A_L - *light ambient color*
   Vettore di 3 componenti positive che indica la quantità di luce ambiente emessa dalla sorgente luminosa per ogni componente di colore (R, G, B). Questa luce colpisce gli oggetti nella scena da ogni direzione.

2. D_L - *light diffuse color*
   Vettore di 3 componenti positive che indica la quantità di luce a riflessione diffusa emessa dalla sorgente luminosa per ogni componente di colore (R, G, B). Questa luce si riflette in tutte le direzioni secondo il modello Lambertiano.

3. S_L - *light specular color*
   Vettore di 3 componenti positive che indica la quantità di luce a riflessione speculare emessa dalla sorgente luminosa per ogni componente di colore (R, G, B). Questa luce si riflette in una singola direzione secondo la legge di riflessione.

Supponendo quindi di avere un oggetto composto dal materiale M e illuminato dalla sorgente di luce L caratterizzati dai parametri sopra e di prendere in esame un singolo punto P sulla superficie dell'oggetto osservato da una posizione O, e definendo:

- N - il vettore ortogonale alla superficie nel punto P normalizzato (la normale nel punto P);
- L - il vettore che va dal punto P alla sorgente luminosa L (normalizzato);

- V - il vettore che va dal punto P alla posizione dell'osservatore O (normalizzato);
- R - il vettore che indica la direzione e il verso di un raggio incidente nel punto P riflesso secondo la legge di riflessione (normalizzato);

introducendo inoltre la notazione per i vettori: '$\cdot$' per il prodotto scalare e '$\otimes$' per il prodotto componente per componente, abbiamo che secondo il modello di illuminazione di Phong il colore del punto P risulterà:

$$C_P = E_M + A_M \otimes A_L + (D_M \otimes D_L)(L \cdot N) + (S_M \otimes S_L)(V \cdot R)^n \tag{2.1}$$

dove però bisogna scartare ogni termine che nella somma risulti negativo (a causa di un prodotto scalare che da come risultato un valore negativo).

Nel caso di molteplici sorgenti luminose, definendo:

- L_i - il vettore che va dal punto P alla sorgente luminosa i-esima normalizzato;
- R_i - il vettore direzione di un raggio proveniente dalla sorgente L_i, incidente nel punto P, riflesso secondo la legge di riflessione e normalizzato;
- A_{Li} - il vettore per il colore ambient dell'i-esima sorgente luminosa;
- D_{Li} - il vettore per il colore diffuse dell'i-esima sorgente luminosa;
- S_{Li} - il vettore per il colore specular dell'i-esima sorgente luminosa;
- K - il numero di diverse sorgenti luminose;

la formula per l'illuminazione del punto P secondo il modello di Phong può essere facilmente estesa come segue:

$$C_P = E_M + \sum_{i=1}^{K} \left[A_M \otimes A_{Li} + (D_M \otimes D_{Li})(L_i \cdot N) + (S_M \otimes S_{Li})(V \cdot R_i)^n\right] \tag{2.2}$$

Per capire l'effetto che le diverse proprietà del materiale e delle sorgenti luminose possono avere sul rendering, consideriamo una semplice mesh che rappresenta una teiera illuminata da una singola sorgente luminosa e vediamo cosa succede aggiungendo e rimuovendo i singoli termini (fig. 2.1). Supponiamo che E_M sia sempre nullo in ogni sua componente (ossia la teiera non emette luce), tenendo presente che comunque si tratta di un semplice termine additivo. Nella figura abbiamo, in

ordine: solo la componente ambientale; solo la componente diffuse; solo la componente specular; componente ambientale e diffuse; componente ambientale, diffuse e specular.

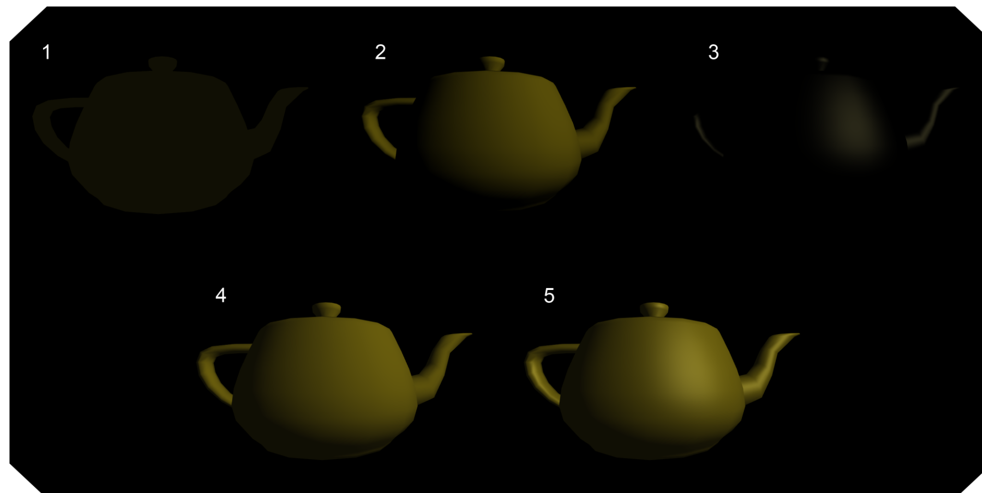


Figura 2.1: Proprietà dei materiali e delle sorgenti luminose

Come si vede, con la combinazione di tutti i termini si riesce ad ottenere una percezione piuttosto convincente dell'oggetto tridimensionale (immagine 5).

2.1.2 Flat e Smooth Shading

Nelle formule per l'illuminazione secondo Phong è richiesta la normale nel punto in cui si vuole calcolare il colore della mesh a cui è applicato il materiale. Tale normale deve essere disponibile nel momento in cui si va a calcolare il colore di un punto sull'oggetto virtuale. Supponiamo per adesso di aver bisogno di tale informazione nel momento in cui si calcola il colore di un frammento (nel fragment shader). Gli unici dati che abbiamo a disposizione per quanto riguarda la geometria di una mesh sono le posizioni dei vertici di ogni singolo triangolo. Potremmo dunque pensare di usare come normale in un frammento la normale del poligono di cui fa parte. Il risultato per la teiera numero 5 in figura 2.1 è riportato in fig. 2.2.

Figura 2.2: Flat shading

Questo tipo di *shading* (inteso come modulazione del colore di un materiale applicato ad una mesh sulla base dei parametri di illuminazione) prende il nome di *flat shading*[8]. Sebbene questo effetto possa essere voluto in determinate situazioni, spesso quello che vorremmo ottenere è uno shading più 'smooth' (omogeneo): pur sapendo che la geometria è composta da superfici piane regolari, l'oggetto virtuale dovrebbe apparire liscio (la sua superficie non dovrebbe contenere spigoli evidenti, come nella precedente figura 2.1).

In passato sono state pensate diverse tecniche per ottenere smooth shading di superfici curve (rappresentate geometricamente da insiemi di triangoli), e le principali sono riportate in [8, 9].

- *Gouraud shading* [8]
  Per ogni vertice della superficie che deve subire smooth shading si calcola una singola normale ottenuta mediando le normali dei poligoni che si incontrano nel vertice (eventualmente con una media pesata). Queste normali vengono poi usate per ottenere il colore dei vertici della superficie secondo un qualche modello di illuminazione (come quello di Phong). Il colore dei frammenti di ogni singolo poligono viene quindi ottenuto interpolando linearmente i valori ottenuti per i 3 vertici (interpolazione bilineare).

[8] In realtà il flat shading classico non prevede l'utilizzo del Phong reflection model, ma di un modello semplificato che considera solamente l'angolo con cui arrivano i raggi luminosi e la normale al poligono. Il calcolo dell'illuminazione non dovrebbe avvenire su ogni frammento ma una singola volta per ogni poligono.

- *Phong interpolation*[9]
  Questo metodo è superiore al precedente e rappresenta una migliore approssimazione di quello che si otterrebbe facendo shading secondo il modello di riflessione di Phong di una superficie effettivamente curva. In questo caso si calcolano le normali nei vertici come sopra con un operatore di media, tuttavia non si calcola direttamente il colore del vertice. Il colore dei frammenti di ogni poligono viene ottenuto interpolando linearmente i vettori normali calcolati per i 3 vertici e normalizzando il risultato, usando il vettore ottenuto (normale nel frammento) per il calcolo del colore secondo il Phong reflection model.

Nonostante il secondo metodo risulti computazionalmente più costoso (perché richiede l'applicazione del reflection model per ogni frammento e non per ogni vertice), i risultati sono decisamente superiori soprattutto nel caso in cui le mesh utilizzate abbiano un basso numero di poligoni. Con i moderni adattatori grafici le differenze di performance tra i due metodi non sono particolarmente significative, per cui normalmente risulta conveniente applicare il metodo di Phong. Le immagini in figura 2.1 sono state ottenute usando la Phong interpolation.

Notare inoltre che l'idea di eseguire il calcolo dell'illuminazione per ogni pixel (*per-pixel lighting*) è fondamentale allo scopo di migliorare i risultati anche nel caso di superfici spigolose: anche se non cambia la normale nel frammento, la posizione dell'osservatore nei confronti del frammento si modifica ogni volta. Il Phong reflection model infatti si fonda sull'ipotesi che l'illuminazione venga calcolata per ogni singolo pixel.

Quando si ha a che fare con oggetti più complessi, che hanno sia superfici curve che superfici spigolose, bisogna avere a disposizione informazioni aggiuntive sulla geometria della mesh. Nella pratica in questi casi spesso si ricorre alla soluzione degli *smoothing groups*: insieme ai dati geometrici che caratterizzano una mesh vengono anche definiti dei raggruppamenti all'interno dell'insieme di triangoli, che prendono il nome di smoothing groups. Gli spigoli dei triangoli appartenenti allo stesso gruppo vengono 'nascosti' quando si arriva al rendering (con una delle tecniche sopra), come se la superficie fosse effettivamente curva. In pratica allora ad ogni vertice verrà assegnata una normale diversa per ogni smoothing group a cui appartiene (creando spigoli visibili tra smoothing groups). In fig. 2.3 è riportato un esempio di rendering di una sedia con l'utilizzo degli smoothing groups: come si vede, i braccioli sembrano avere una superficie curva (i poligoni che li compongono formano uno smoothing group) mentre ai lati degli stessi abbiamo uno spigolo netto (triangoli non facenti parte dello stesso gruppo).

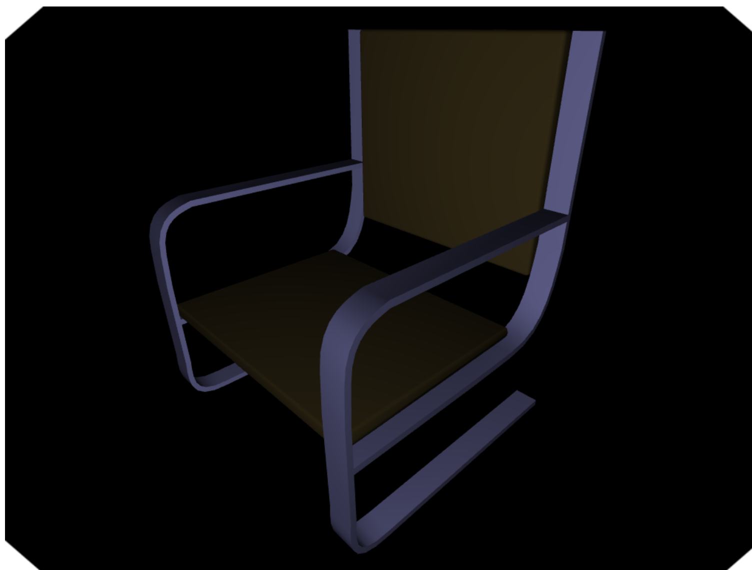


Figura 2.3: Smoothing groups

2.1.3 Diffuse Texture Mapping

Una texture è fondamentalmente una raccolta di una o più immagini che possono venire usate per aggiungere dettagli alla mesh renderizzata con un materiale come descritto sopra. In questa sezione ci limitiamo a considerare l'utilizzo più semplice e comune delle texture: modulare il colore del materiale applicato ad una determinata mesh sulla base di un'immagine (talvolta catturata direttamente da oggetti reali); tecnica che prende il nome di diffuse texture mapping.

Un'immagine va considerata come una matrice di celle ad ognuna delle quali sono associate alcune componenti di colore (per esempio R, G e B). In questo paragrafo ci limitiamo ad esaminare texture bidimensionali (composte da una singola immagine). Nel contesto delle texture bidimensionali, una cella nell'immagine che la compone si dice *texel* (texture element) o *texture pixel*. Esistono altri tipi di texture oltre a quelle bidimensionali: texture tridimensionali e cube map ne sono un esempio.

Data un'immagine che viene utilizzata come diffuse texture (per modulare il colore), bisogna dunque capire come avviene l'applicazione della stessa alla mesh renderizzata. Insieme ai dati geometrici che descrivono una mesh normalmente vengono fornite anche delle coordinate di texture per ogni vertice, che determinano a quale punto dell'immagine corrisponde il vertice considerato. Le coordinate dei vertici nella texture (che potremmo chiamare *texture vertices*) sono date da una coppia di valori (u, v) compresi tra 0 e 1, che indicano la distanza del vertice dall'angolo in basso a sinistra dell'immagine (in OpenGL), vedi fig. 2.4.

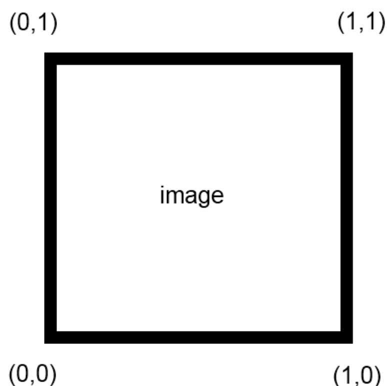


Figura 2.4: Texture coordinates

È ovvio che, quando una mesh viene disegnata, andranno colorati sulla base del contenuto della texture di diffuse tutti i pixel nel framebuffer che risultano influenzati rasterizzando i vari poligoni, non solamente i vertici. Per ogni frammento che si arriva a disegnare, le corrispondenti coordinate si ottengono interpolando quelle dei vertici del poligono di cui fa parte; questa operazione viene fatta automaticamente dall'hardware grafico (o via software dal sistema OpenGL se l'adattatore non ne è capace).

Supponiamo adesso di avere una diffuse texture composta da un'immagine quadrata di 256×256 pixel, e di volerla applicare ad una mesh composta da due triangoli che formano un piano rettangolare che copre l'intero campo visivo. Se questo piano va disegnato in una finestra di 1024×768 pixel allora ovviamente avremo che ad ogni texel di texture corrisponderanno diversi pixel dello schermo. Una situazione diversa, ma con problemi simili, è quella in cui ad un solo pixel dello schermo corrispondono diversi texel. Si parla nel primo caso di *texture magnification* (ingrandimento) e nel secondo di *texture minification* (riduzione).

In queste situazioni si possono scegliere diversi approcci; ad esempio, nel caso di magnification:

- usare il colore del texel su tutti i pixel interessati;
- combinare i colori di alcuni texel nell'immagine sulla base dell'esatta posizione del pixel all'interno della stessa;
- ecc...

Questi metodi vengono chiamati metodi di *filtraggio* delle texture. Nel caso di riduzione, una tecnica particolarmente interessante è quella del *mipmapping*, che

consiste fondamentalmente nel generare delle copie dell'immagine a minor livello di dettaglio (con meno pixel) prima di arrivare al rendering, sfruttando algoritmi di vario tipo per evitare artefatti evidenti (ad esempio basandosi sulla teoria dei segnali). Durante il rendering si userà una particolare mipmap (o anche più di una) per ottenere il colore associato ad un dato frammento in modo efficiente. Anche la gestione del mipmapping è sostanzialmente a carico dell'hardware nei moderni processori grafici e in ogni caso il programmatore OpenGL non deve preoccuparsene.

Un esempio di mesh a cui è applicata una texture di diffuse è quello in fig. 2.5.

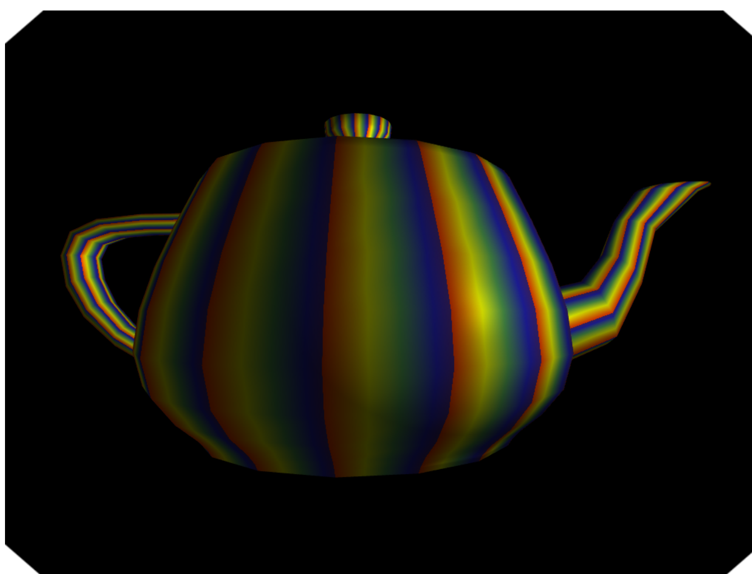


Figura 2.5: Diffuse texture mapping

2.1.4 Light Mapping

Vi sono molti algoritmi per il calcolo dell'illuminazione che forniscono risultati visivamente molto soddisfacenti ma sono troppo costosi per essere utilizzati on-line (durante il rendering in un'applicazione interattiva). Si parla di algoritmi di *global illumination* (in contrasto con i metodi di *direct illumination* visti fino ad ora), tecniche che considerano non solo la luce proveniente da sorgenti esplicite, ma anche quella riflessa da altri oggetti nella scena. Alcuni di questi algoritmi sono in grado di ottenere immagini estremamente realistiche ma purtroppo il loro costo

computazionale[9] ne ha proibito (almeno fino ad ora) l'utilizzo in applicazioni real-time interattive, esempi sono: *ray tracing* [11], *radiosity* [12], *photon mapping* [13], *ambient occlusion* [14], *algoritmi basati su antiradiance* [15], ecc...

È bene evidenziare che ad oggi la spinta verso l'applicazione di queste tecniche ad elevato realismo in tempo reale per applicazioni interattive è forte e la ricerca si sta spostando molto in questa direzione. Alcuni algoritmi vengono già applicati anche in prodotti interattivi commerciali, ma purtroppo sono gestibili solo sulle migliori schede video.

Una scena renderizzata con una tecnica di ray tracing è quella in fig. 2.6.

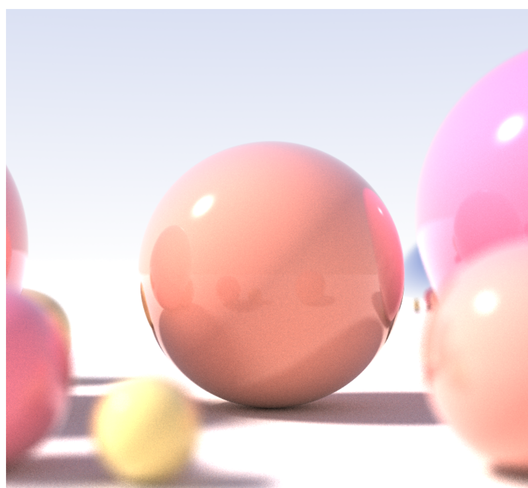


Figura 2.6: Esempio di scena renderizzata con una tecnica di ray tracing

Queste tecniche vengono comunemente applicate per il rendering offline tipico degli ambienti CAD per ingegneri e architetti e per i film in computer grafica, ma possono anche essere utilizzate per precalcolare dei parametri di illuminazione da utilizzare poi durante il rendering della scena. Spesso usando algoritmi così elaborati si riescono ad ottenere anche altre informazioni oltre allo shading del singolo oggetto, come la presenza di ombre proiettate da alcuni oggetti che impediscono a parte della luce di raggiungere altri oggetti nella scena.

L'utilizzo più semplice per queste informazioni di illuminazione è l'inserimento delle stesse in una texture (*light map*) che venga poi usata al posto della tipica il-

[9] Il costo di esecuzione di un algoritmo viene detto anche *complessità* ed è valutabile sotto diversi aspetti (tempo di esecuzione, quantità di memoria utilizzata, ecc...). Nella nostra trattazione con 'costo computazionale' o 'complessità' ci riferiamo sempre alla complessità temporale di un algoritmo (ci interessa sostanzialmente il tempo di esecuzione dello stesso), se non specificato altrimenti.

luminazione diretta durante il rendering. Supponiamo ad esempio di avere per la scena in figura 2.6 la light map calcolata renderizzando su una texture la luminosità risultante in ogni punto per il piano su cui poggiano le sfere. Questa informazione verrà applicata insieme al materiale e alla texture di diffuse del piano (se presente), arrivando effettivamente ad un approccio di multitexturing allo scopo di modulare il colore risultante dal materiale e dalla diffuse texture sulla base di dati di illuminazione precalcolati.

Un esempio di scena renderizzata sfruttando delle light maps è quello in fig. 2.7.

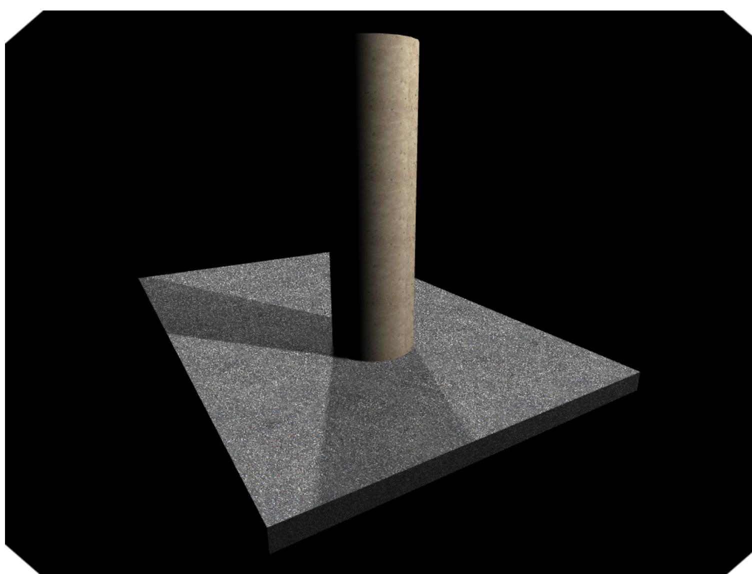


Figura 2.7: Light mapping

Ovviamente non è sempre possibile utilizzare il light mapping per risolvere i problemi di illuminazione ottenendo risultati realistici: questa tecnica si basa infatti su informazioni precalcolate e dunque non è adatta al caso di illuminazione che cambia nel tempo o oggetti mobili in ambienti dove l'illuminazione è statica. La tecnica di light mapping non è inoltre sensibile al punto di vista e dunque non è possibile sfruttarla per illuminare oggetti riflettenti (se questi vengono esaminati da diversi punti di vista).

2.1.5 Environment Mapping e Image-Based Lighting

Un'altra tecnica che può essere inserita nell'ambito dei metodi di global illumination è quello che si chiama *Image-Based Lighting (IBL)* [17]. Nel contesto dell'IBL rientano diversi approcci e diversi algoritmi, ma l'idea fondamentale è quella di

utilizzare immagini catturate in ambienti reali per illuminare oggetti virtuali della scena renderizzata.

Scopo di tale tecnica è quello di produrre per la scena virtuale un'illuminazione più realistica di quella che si riesce ad ottenere utilizzando solo sorgenti luminose puntiformi. Nonostante in letteratura la terminologia sia abbastanza confusa, noi consideriamo la dicitura *'environment mapping'* del tutto equivalente a *'image-based lighting'*. In questa sede discuteremo sostanzialmente di un particolare algoritmo di IBL adatto ad applicazioni in tempo reale, che si configura anche come quello più comune e diffuso. Altri algoritmi che rientrano nell'ambito dell'IBL sono stati sviluppati col tempo, alcuni dei quali producono risultati migliori della versione che vedremo adesso, ma non sono adatti ad applicazioni interattive.

Alcuni testi identificano la tecnica che descriviamo in questo paragrafo come *environment mapping*, distinguendola però dall'approccio IBL a cui invece si richiede l'utilizzo di metodi complessi di illuminazione globale.

Consideriamo un ambiente reale, come una stanza illuminata dalla luce solare che arriva da una finestra. La tecnica IBL consiste nel catturare una o più immagini da tale ambiente reale per poi illuminare oggetti virtuali in modo realistico. Le immagini catturate a questo scopo vengono spesso chiamate *light probe images*, e possono assumere diverse forme a seconda di come viene catturata la scena. Alcuni metodi sono:

1. Fotografare da distanza elevata una sfera perfettamente riflettente posta al centro della scena che si vuole catturare. Contrariamente a quanto si possa pensare, una sfera fotografata in questo modo contiene informazioni su tutta la scena attorno ad essa e non solo sulla metà che si trova dal lato della fotocamera. L'unico punto effettivamente non disponibile è quello direttamente dietro la sfera.

2. Catturare un insieme di immagini usando una fotocamera situata al centro della scena e facendola guardare in una diversa direzione ad ogni scatto.

3. Utilizzare delle lenti particolari da applicare alla fotocamera dette fisheye lens che consentono in due scatti di coprire l'intera scena circostante.

4. Utilizzare fotocamere speciali a scansione panoramica.

La caratteristica fondamentale che deve avere una light probe image è quella di essere omnidirezionale, ossia deve contenere informazioni riguardanti tutte le direzioni dal

centro della scena. Sulla base di come l'immagine viene catturata, abbiamo una diversa forma della light probe image e un diverso modo di applicare l'illuminazione descritta dall'immagine agli oggetti nella scena.

La prima tecnica utilizzata per fare environment mapping prende il nome di *sphere environment mapping*, in questo caso si sfrutta una light probe image ottenuta col primo metodo descritto sopra o ottenuta sintetizzando il risultato di un altro metodo nella forma che risulterebbe dall'applicazione della tecnica 1. L'immagine risultante in questo caso si dice *sphere environment map* e questa può essere utilizzata come texture durante l'illuminazione di mesh nella scena virtuale (trasformando opportunamente le coordinate per il lookup sulla texture). In fig. 2.8 è riportato un esempio di sphere environment map.

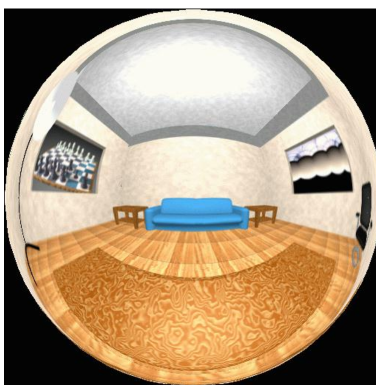


Figura 2.8: Esempio di sphere map

Non vediamo in dettaglio il funzionamento dell'algoritmo di sphere environment mapping perchè tale tecnica è stata superata dalla più recente tecnica di *cube environment mapping* [18]. In questo caso la light probe image è composta da 6 diverse immagini corrispondenti alle facce di un cubo e viene detta *cube environment map*. Le 6 diverse immagini possono venire catturate col metodo 2 descritto sopra, oppure sintetizzate a partire dal risultato ottenuto usando uno degli altri metodi. Il modo in cui le 6 diverse immagini sono organizzate nella cube map può variare a seconda dei casi, ma comunemente vengono posizionate a formare una croce (verticale o orizzontale) come in fig. 2.9.

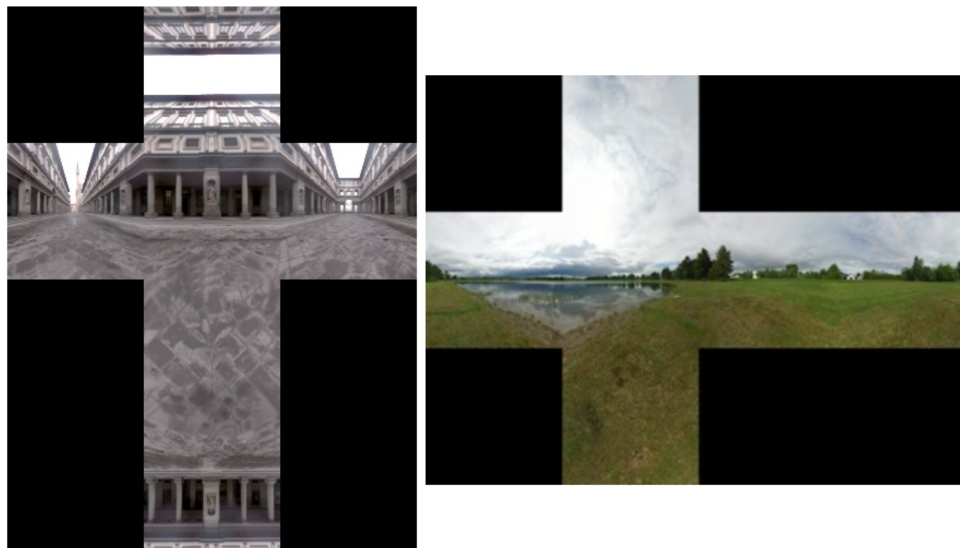


Figura 2.9: Esempi di cube environment map

Una cube environment map può venire caricata in OpenGL come un tipo particolare di texture (`GL_TEXTURE_CUBE_MAP`) e va considerata in tal caso come un cubo le cui facce rappresentano le 6 diverse immagini specificate. Quando si lavora con le cube map in OpenGL, la selezione di un texel avviene con un vettore di 3 componenti (s, r, t) che viene trattato come un vettore direzione applicato nel centro del cubo, il texel restituito sarà in questo caso quello indicato dal vettore (s, r, t), opportunamente filtrato eventualmente considerando le facce adiacenti.

Il cube environment mapping è superiore allo sphere environment mapping per diversi motivi:

- Lo sphere environment mapping introduce una forte distorsione nell'immagine catturata dovuta al fatto che l'intera scena reale è applicata ad una sfera, e questo può portare a degli artefatti durante il rendering;

- Lo sphere environment mapping funziona in modo soddisfacente solamente quando si renderizza esattamente dallo stesso punto di vista della fotocamera che ha catturato la sphere map, guardando nella stessa direzione.

Bisogna dunque capire come utilizzare una cube map per illuminare gli oggetti virtuali presenti nella scena. Gli approcci sono diversi anche in questo caso, quello più diffuso comporta l'utilizzo contemporaneo di due diverse cube map durante il rendering per il calcolo dell'illuminazione [19, 20, 21] a cui potremmo dare i nomi di *specular cube environment map* e *diffuse cube environment map*. L'immagine

catturata dalla scena reale può essere sostanzialmente utilizzata direttamente come specular cube environment map, anche se spesso un prefiltraggio con un effetto di blurring[10] può migliorare l'impatto visivo. La diffuse cube environment map da usare assieme alla prima si ottiene eseguendo una convoluzione della cube map catturata tale da perdere qualsiasi tipo di dettaglio presente nell'immagine.

Eseguire la convoluzione di un'immagine significa eseguire un filtraggio della stessa scegliendo una funzione bidimensionale da utilizzare per la convoluzione. Dato un tipo di filtro di convoluzione (ad esempio un filtro gaussiano) si costruisce una matrice bidimensionale di un certo numero di pixel contenente i fattori moltiplicativi campionati dalla funzione scelta; questa matrice si dice *kernel* del filtro. Il colore di ogni pixel nell'immagine è rimpiazzato con la somma dei prodotti tra i colori dei pixel vicini e i rispettivi membri del kernel del filtro, come nella formula seguente:

$$O(x,y) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left[I\left(x - \frac{m-1}{2} + i, y - \frac{n-1}{2} + j\right) \cdot K\left(i,j\right)\right]$$

Dove:

- $I(x, y)$ è il colore del pixel dell'immagine originale nella posizione (x, y),
- $O(x, y)$ è il colore del pixel dell'immagine risultante nella posizione (x, y),
- m è la larghezza in pixel del kernel del filtro (supponiamo dispari),
- n è l'altezza in pixel del kernel del filtro (supponiamo dispari),
- $K(i, j)$ è il fattore moltiplicativo nel kernel del filtro in posizione (i, j).

Ovviamente ogni tipo di filtraggio dovrà avvenire considerando che l'immagine è in realtà una cube map (ossia considerando anche le facce adiacenti) per evitare la creazione di cuciture[11] visibili (*seams*) nell'illuminazione degli oggetti virtuali.

In fig. 2.10 è riportato come esempio il kernel di un filtro gaussiano con diametro di 5 pixel.

---

[10] Il filtro di blur è un particolare filtro di convoluzione che ha l'effetto di rendere un'immagine sfocata e poco chiara mescolando i colori in un intorno di ogni pixel per produrre un singolo pixel filtrato.

[11] Le cuciture sono artefatti che nascono in corrispondenza degli spigoli delle facce del cubo invalidando l'illuminazione risultante.

$\frac{1}{273}$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Figura 2.10: Kernel di un filtro di convoluzione gaussiano

Come si vede, sommando tutti i fattori moltiplicativi nel kernel si arriva all'unità.

In fig. 2.11 abbiamo invece una coppia di cube map: la cube map sulla sinistra è sostanzialmente quella catturata dall'ambiente e viene usata inalterata come specular cube environment map, quella sulla destra è invece una versione filtrata della prima con l'utilizzo di un filtro di convoluzione che può venire usata come diffuse cube environment map.

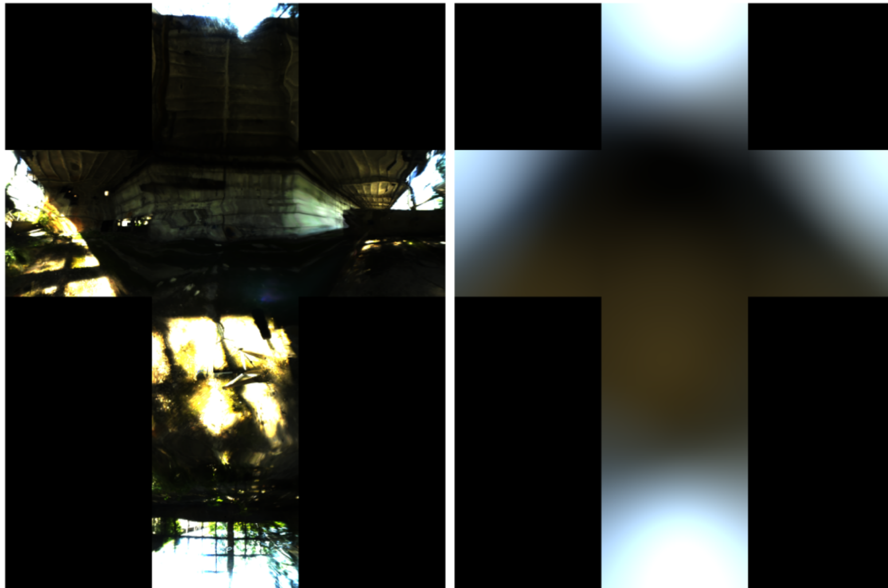


Figura 2.11: Specular e diffuse cube environment maps

Per ottenere infine lo shading desiderato, illuminando gli oggetti nella scena sulla base delle cube environment map ottenute dal mondo reale, si lavora nel seguente modo (estendendo semplicemente il modello di riflessione di Phong):

- Il colore di illuminazione a riflessione diffusa (D_L) viene ottenuto campionando la diffuse cube environment map nella direzione dettata dalla normale nel

punto considerato (nel frammento quando si opera con la Phong interpolation). Indichiamo con $DiffMap\,(N)$ questo termine campionato.

- Il colore di illuminazione a riflessione speculare (S_L) viene ottenuto campionando la specular cube environment map nella direzione dettata dalla riflessione del vettore che va dalla posizione dell'osservatore al punto considerato (tramite la legge di riflessione e normalizzando il risultato). Indichiamo con R questo vettore riflesso e con $SpecMap\,(R)$ il termine campionato secondo il vettore R.
- Il termine per la componente ambientale ($A_M \times A_L$) viene abbandonato in quanto l'illuminazione ambientale è superflua quando abbiamo a disposizione la diffuse cube environment map che ci suggerisce la luce a riflessione diffusa che arriva da ogni direzione dell'ambiente reale.

La formula per l'illuminazione del punto P secondo questo modello di environment mapping è allora:

$$C_P = E_M + (D_M \otimes DiffMap\,(N)) + (S_M \otimes SpecMap\,(R)) \qquad (2.3)$$

Un esempio di rendering con IBL secondo la formula sopra è riportato in figura 2.12.

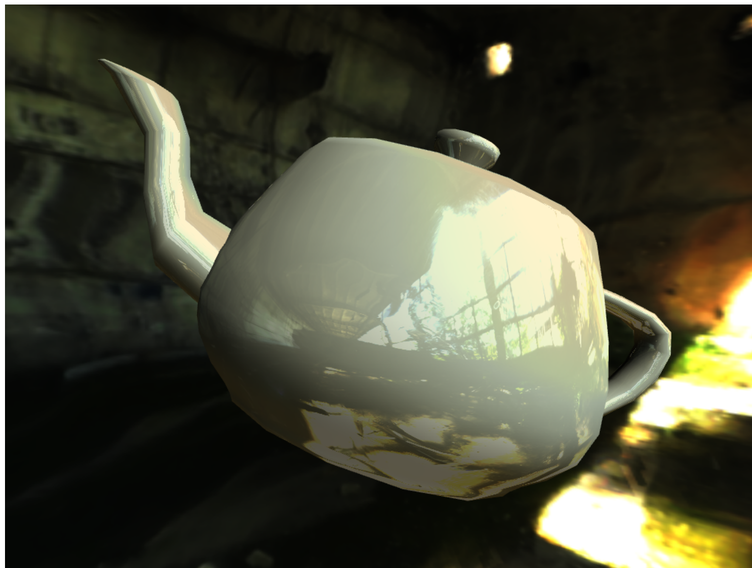


Figura 2.12: Environment mapping e skybox

In figura va notato che la teiera sembra effettivamente inserita nell'ambiente che la sta illuminando. La tecnica di environment mapping si occupa solamente di illu-

minare gli oggetti, ma in figura si usa anche una tecnica diversa che prende il nome di *skybox* per creare uno sfondo che circondi completamente la scena tramite l'utilizzo di una cube map renderizzata su un cubo a distanza teoricamente infinita. Nella pratica il cubo viene renderizzato ad una certa distanza dalla telecamera direttamente attorno ad essa, ma senza registrarne la profondità, in modo tale che qualsiasi oggetto disegnato dopo la skybox sembri interno ad essa. Evitare di registrare la profondità dei frammenti della skybox è fondamentale per evitare che il depth test impedisca a frammenti di altri oggetti esterni al cubo di raggiungere il framebuffer (vedi sez. 2.3). A differenza di uno sfondo statico, una skybox ruota a seconda dei movimenti dell'osservatore come se la scena raffigurata sulla stessa fosse a distanza infinita. L'utilizzo di una skybox normalmente accompagna la tecnica di environment mapping perchè inserisce effettivamente l'oggetto all'interno dell'ambiente che lo sta illuminando.

Un'aspetto che fino ad ora è stato trascurato in questa discussione sull'IBL sono le difficoltà di rappresentazione dei dati di illuminazione prelevati dalla realtà all'interno delle light probe images. Nonostante sia molto comune utilizzare i formati classici di rappresentazione di immagini digitali anche per le light probe images (png, jpeg, bmp, ecc...) spesso accade che con questi formati[17]:

- i livelli di luminosità sono codificati in modo non lineare per garantire un rendering più piacevole dell'immagine su dispositivi di visualizzazione non lineari, per cui i livelli di luminosità registrati spesso non sono proporzionali a quelli della scena reale;
- dato la limitata escursione di valori possibili per ogni colore, si rappresenta normalmente solo un range limitato di valori di luminosità, troncando al valore massimo o minimo tutti i casi che escono dal range.

Per questo, quando si vuole ottenere un rendering più convincente con tecniche di IBL, si ricorre a immagini acquisite con metodi di fotografia ad elevato range dinamico[12]. Normalmente immagini ad elevato range dinamico (*high dynamic range, HDR*) vengono costruite scattando una serie di fotografie alla stessa scena con periodi variabili di esposizione, che poi vengono processate in un'unica immagine digitale che viene salvata in uno dei formati HDR ad oggi disponibili. Talvolta i formati HDR utilizzano un numero in virgola mobile a precisione singola per ciascuno dei canali

[12] Il range dinamico di un'immagine viene di solito inteso come il rapporto di luminosità che si ha tra il pixel più brillante e quello più scuro che è possibile rappresentare in modo accurato.

di colore, ma ad oggi il formato più comune di immagine digitale HDR è il formato *RGBE* introdotto in origine da Greg Ward nel sistema di rendering Radiance. Nel formato RGBE ogni canale di colore (R,G,B) è immagazzinato in un singolo byte, tuttavia un ulteriore byte viene previsto per ogni pixel come esponente a comune tra i canali di colore (E): questo espande notevolmente il range di possibili valori rappresentabili consentendo sia pixel molto brillanti che pixel molto scuri. I file contenenti immagini nel formato RGBE vengono normalmente indicati con estensione '.hdr'.

Usando immagini HDR come environment map, durante il rendering siamo sicuri che non si presenteranno artefatti dovuti alla non linearità della codifica ed inoltre siamo in grado di ricavare valori molto precisi di illuminazione dato l'elevato range dinamico. Questo produce ottimi risultati come quello in fig. 2.12.

Bisogna tuttavia notare che i display tradizionali non sono in grado di visualizzare un range di colori vasto come quello rappresentato in immagini HDR. Dunque, nonostante il risultato sia indubbiamente corretto e preciso grazie all'utilizzo di immagini HDR, quando i colori verranno scritti nel framebuffer della finestra OpenGL al termine dell'esecuzione del fragment shader questi verranno troncati e l'effetto non risulterà molto diverso da quello che si sarebbe ottenuto usando immagini non HDR (a meno di eventuali non linearità nell'illuminazione).

La procedura per mappare un elevato range dinamico come quello che si ottiene durante il rendering con immagini HDR su un range più ridotto come quello che i moderni display hanno a disposizione è quello che si dice *tone mapping* [20]. Esistono vari operatori di tone mapping e in particolare anche il semplice troncamento eseguito in automatico da OpenGL può essere visto come uno di questi.

Per applicare complessi operatori di tone mapping, spesso si procede renderizzando prima la scena off-screen, in un buffer non visualizzato ad elevato range dinamico (ad esempio in virgola mobile a precisione singola). Dopo di che si esegue l'operatore di tone mapping che può produrre immagini diverse sulla base di alcuni parametri (come il livello di esposizione desiderato). Questa procedura ci permette anche di applicare diversi effetti sul buffer prima del tone mapping, sfruttando l'elevato range dinamico dei valori al suo interno; un esempio comune è l'effetto di *bloom* che espande le aree a forte illuminazione oltre i loro bordi effettivi come accade nella realtà grazie a fenomeni ottici.

2.2 Complessità Geometrica e Texture

Consideriamo una situazione in cui si voglia renderizzare un oggetto con un livello di dettaglio molto elevato. Normalmente in questi casi il numero di triangoli da disegnare cresce a dismisura. Se ad esempio vogliamo disegnare un pavimento nel modo più realistico possibile, è ovvio che le scanalature tra le piastrelle (o sulle piastrelle stesse) possono introdurre un numero non indifferente di triangoli aggiuntivi (a seconda di quanto precisa deve essere l'approssimazione). Nella pratica, modelli complessi possono essere composti da svariate centinaia di migliaia di triangoli.

Quando alla pipeline OpenGL viene chiesto di renderizzare un numero molto elevato di triangoli, inevitabilmente le prestazioni calano perché sale il carico di lavoro richiesto ad ogni stadio. In particolare un numero maggiore di vertici comporta più lavoro sullo stadio di vertex processing e su quello di geometry processing (composti normalmente da svariati processori), ed inoltre spesso provoca anche un incremento del lavoro sullo stadio di fragment processing. Un problema che si pone è allora quello di renderizzare delle mesh ad elevato livello di dettaglio senza però dover ricorrere ad un numero enorme di poligoni.

In questa sezione vediamo alcune tecniche nate con questo obiettivo che si basano su texture .

2.2.1 Normal Mapping

Nel paragrafo 2.1.2 abbiamo visto come superfici composte da un certo numero di poligoni possano essere rese 'smooth' agli occhi di un utente utilizzando il metodo di interpolazione di Phong, ossia lavorando semplicemente sulle normali che vengono approssimate ed utilizzate per l'illuminazione di ogni frammento.

In pratica quello che si sta facendo in questo contesto è mascherare l'effettiva geometria della mesh alterando il modo in cui questa viene percepita dall'utente sulla base delle direzioni delle normali.

Potremmo allora pensare che perturbando le direzioni delle normali in altro modo si possano ottenere anche diversi effetti, come ad esempio dare l'impressione che ci siano delle protuberanze o concavità sulla superficie. Questo in effetti è possibile facendo in modo che l'andamento delle normali perturbate segua quello della superficie ad elevato livello di dettaglio che vogliamo ottenere.

La tecnica del normal mapping funziona leggendo la direzione delle normali perturbate da un'immagine usata come texture bidimensionale durante il rendering. Il normal mapping è assai comune e apprezzato nei moderni software di rendering

interattivo (come ad esempio i videogames) in quanto consente di raggiungere un elevato grado di realismo senza aumentare il numero di poligoni.

Il normal mapping è sostanzialmente una tecnica di *bump mapping*, intendendo con questa categoria tutte le tecniche capaci di simulare complessità geometrica senza aumentare l'effettivo numero di poligoni. La prima tecnica di bump mapping venne proposta da Blinn nel 1978 [22], ma purtroppo risultava inapplicabile in real-time con gli adattatori grafici disponibili all'epoca e divennero quindi popolari diverse tecniche semplificate ma efficienti (come l'*emboss bump mapping* [23]), con risultati non molto convincenti. L'idea di trasferire i dettagli tipici di oggetti ad elevata complessità geometrica all'interno di immagini dette normal map nacque solamente nel 1998 [24], ma occorsero alcuni anni prima che il normal mapping divenisse una necessità per tutte le applicazioni ad elevato livello di realismo. Oggi sono disponibili anche altre tecniche per fare bump mapping (come ad esempio il *parallax mapping* [25]).

Limitandoci per adesso all'analisi delle normal map, illustriamone brevemente il funzionamento. A tutti gli effetti una normal map è un'ulteriore texture 2D da applicare su oggetti virtuali; in questa immagine si codificano però adesso le direzioni dei vettori normali. Le normali calcolate sulla base della geometria come nel paragrafo 2.1.2 diventano quindi in questo contesto del tutto inutili in quanto basta avere a disposizione delle coordinate con cui accedere alla normal map per scoprire il valore della normale in ogni singolo punto di un oggetto virtuale.

Una normal map riporta in ogni componente di colore una componente del vettore normale modificato; in particolare potremmo assumere che il rosso corrisponda alla componente x della normale, il verde alla componente y e il blu alla componente z. Se ad esempio una normal map con 256 possibili livelli per ogni singolo colore ha un pixel di colore dato da (255, 127, 127), quel pixel rappresenta sostanzialmente un vettore diretto verso l'asse positivo delle x, mentre (0, 127, 127) rappresenta un vettore diretto verso l'asse negativo delle x.

Una normal map può contenere le normali come viste in spazio oggetto (*object-space normal map*) oppure può contenere le direzioni normali nel cosiddetto *tangent-space* (*tangent-space normal map*); in quest'ultimo caso le tre componenti sono relative alla superficie della faccia su cui stiamo lavorando, con il blu che normalmente rappresenta la direzione uscente dalla faccia. Un confronto tra i due diversi tipi di normal map è riportato in fig. 2.13: a sinistra abbiamo una object-space normal map e a destra una tangent-space normal map.

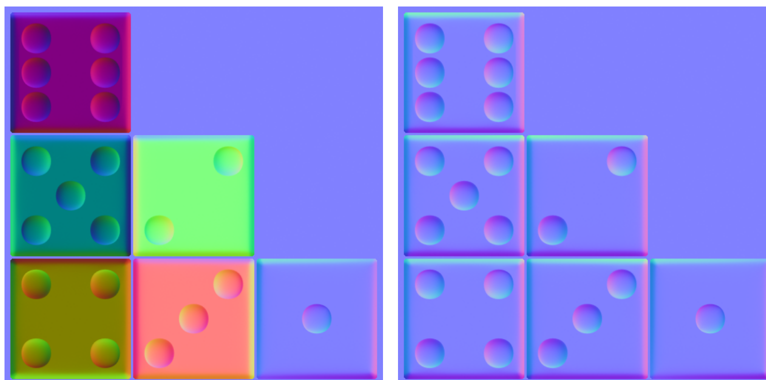


Figura 2.13: Due diversi tipi di normal map

Le normali ricavate dall'immagine sulla base delle coordinate di lookup per la texture possono poi essere usate per qualsiasi tipo di shading (Phong shading[13], environment mapping, ecc...).

In fig. 2.14 è riportato un esempio di oggetto a cui è applicata una normal map: la scena sfrutta IBL e una skybox; a sinistra abbiamo l'oggetto renderizzato con una diffuse texture, a destra invece alla texture di diffuse viene aggiunta anche una normal map. È evidente che l'oggetto sulla destra risulta molto più convincente nonostante la geometria delle due mesh sia identica (due triangoli per ogni faccia del dado).

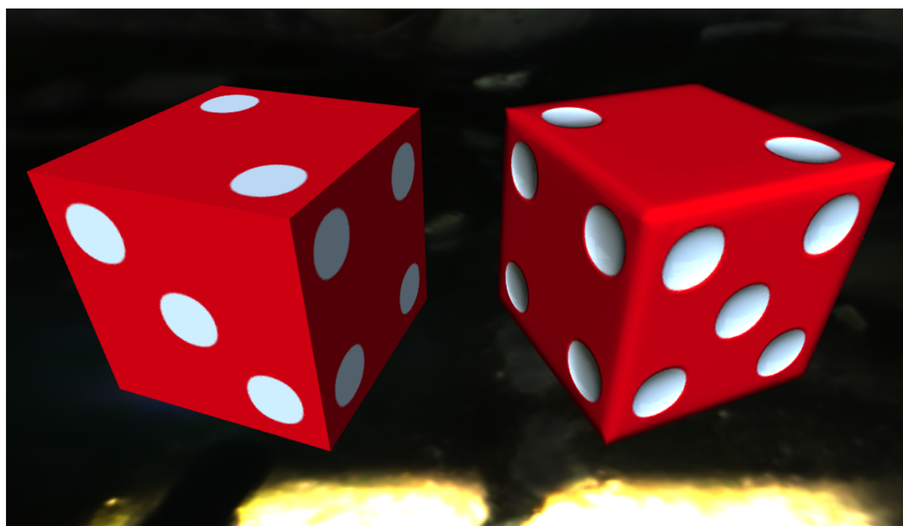


Figura 2.14: Normal mapping

[13] Con Phong shading si intende l'applicazione combinata della Phong interpolation e del Phong reflection model per l'illuminazione diretta degli oggetti virtuali.

2.2.2 Displacement Mapping

Nonostante i risultati che si ottengono utilizzando la tecnica del normal mapping siano molto soddisfacenti, bisogna capire che il metodo ha comunque dei limiti dovuti al fatto che si basa su una semplice alterazione delle normali durante il calcolo dell'illuminazione per ogni frammento. È infatti possibile accorgersi di questi limiti guardando attentamente la fig. 2.14: nonostante il dado sulla destra abbia teoricamente degli angoli smussati (come si capisce osservando quello che connette le 3 facce visibili), gli angoli esterni che collegano le facce visibili a quelle invisibili sembrano erroneamente acuminati. Per quanto infatti si possa alterare la direzione della normale in ogni singolo punto, bisogna capire che la geometria resta quella originale e i frammenti dove si va ad applicare la normale alterata saranno quelli generati rasterizzando i poligoni originali (per esempio, protuberanze nella geometria simulata tramite normal map non saranno mai visibili se si guardano i poligoni di profilo).

Il displacement mapping è un metodo alternativo alle classiche tecniche di bump mapping descritte sopra, in cui l'effettiva geometria della mesh viene alterata sulla base di un'immagine usata come texture bidimensionale che prende il nome di *height map* o *displacement map*. Nonostante siano stati proposti dei metodi alternativi [26], l'approccio originale al displacement mapping prevede il tassellamento dinamico delle superfici renderizzate seguito da una alterazione della posizione dei vertici generati sulla base della height map (spostandoli lungo la normale locale ai singoli vertici).

Inizialmente il displacement mapping venne evitato nei sistemi di rendering in tempo reale basati su OpenGL e Direct3D perché mancava del tutto la possibilità di eseguire un tassellamento dinamico dei poligoni a tempo di rendering. Col tempo vennero sviluppati diversi approcci:

- Per-vertex displacement mapping
  Data una struttura con un elevato numero di triangoli, è possibile modificarne la geometria nello stadio di vertex processing sulla base di una height map. Tuttavia con questo approccio perdiamo il vantaggio di usare tecniche come il displacement mapping perchè il carico sulla pipeline quando si renderizzano mesh complesse è comparabile a quello che si avrebbe se la geometria fosse direttamente quella finale.

- Per-pixel displacement mapping (bump mapping)
  Queste tecniche lavorano all'interno dei fragment shader; a questo punto è in effetti troppo tardi per alterare la geometria dell'oggetto e la rasterizzazione

ha già avuto luogo. In questo caso non si può parlare propriamente di displacement mapping ma si cade nell'ambito del bump mapping. Si opera alterando le coordinate di accesso alle texture sulla base di considerazioni svolte sulla height map, per poi accedere alle stesse con le nuove coordinate ottenendo informazioni di colore e normali nel punto considerato. Essendo tecniche nella famiglia del bump mapping, hanno limiti simili a quelli delle normal map e talvolta possono avere un costo troppo elevato per essere eseguite in applicazioni interattive.

La dicitura 'displacement mapping' viene spesso usata impropriamente in letteratura: in questa sede intendiamo con displacement mapping una tecnica che effettivamente altera la geometria degli oggetti virtuali come nel significato originale del termine (distinguendola in questo modo dalle più semplici tecniche di bump mapping).

Dato lo scarso vantaggio fornitoci dai metodi di vertex displacement mapping, l'applicazione di tecniche di displacement mapping ad applicazioni di rendering in tempo reale è divenuta conveniente solo recentemente, con l'introduzione dello stadio di geometry processing nelle pipeline OpenGL e Direct3D. I geometry shader ricevono in ingresso primitive e possono generare nuovi vertici, sono allora perfetti per essere utilizzati come stadio di tassellamento e displacement mapping col modello che segue [26, 27]:

1. I vertex processor lavorano sulla *macrostruttura* composta da pochi poligoni;
2. I geometry processor tassellano la macrostruttura e la modulano secondo la height map, ottenendo quella che normalmente viene definita *mesostruttura*;
3. I fragment processor eseguono il calcolo dell'illuminazione per i frammenti risultanti dalla rasterizzazione della mesostruttura.

I vantaggi di questo approccio rispetto all'utilizzo diretto nella pipeline di una geometria ad alta complessità sono evidenti:

- Il numero di vertici che devono venir trasferiti col modello della mesh da caricare è molto inferiore;
- Il carico di lavoro sullo stadio di vertex processing è minimo;
- Poche informazioni (pochi vertici) devono essere trasferite all'adattatore grafico, per cui il consumo di banda sul bus dati è minimo.

Modificata la geometria nello stadio di geometry processing, le normali devono essere alterate in accordo alle modifiche. A tale scopo si può fare in modo che il calcolo delle nuove normali avvenga direttamente nel geometry shader, oppure si può utilizzare semplicemente una normal map contenente le normali alterate (ed eventuali perturbazioni dovute alla *microstruttura* dell'oggetto, non riportate nella height map).

In fig. 2.15 abbiamo un esempio di height map di un oggetto cubico con una protuberanza su ogni faccia, accompagnata dalla object-space normal map utilizzata da cui si prelevano le normali alterate dopo lo stadio di geometry processing. Le height map hanno normalmente una sola componente cromatica e un valore elevato (colore chiaro) corrisponde ad una protuberanza sulla superficie, mentre un colore scuro corrisponde ad una concavità. L'entità della deformazione basata sul colore letto dalla height map dipende da alcuni parametri decisi dall'utente.

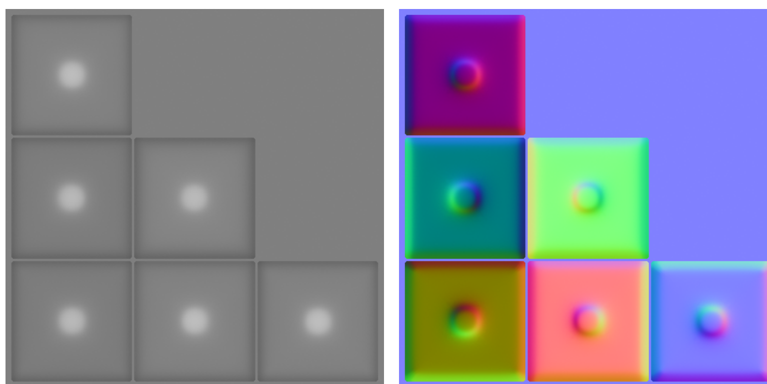


Figura 2.15: Esempio di height map e normal map corrispondente

In fig. 2.16 abbiamo l'oggetto di cui sopra renderizzato con la tecnica di displacement mapping (applicando anche una diffuse texture e illuminando la scena tramite environment mapping).

Figura 2.16: Displacement mapping

Come si nota, le protuberanze dell'oggetto sono visibili anche se i relativi poligoni sono visti di profilo e gli spigoli dell'oggetto sono effettivamente smussati come dettato dalla height map. Tutta la complessità geometrica derivante dalle protuberanze e dagli angoli smussati viene costruita a tempo di rendering nello stadio di geometry processing, mentre la mesh originale contiene un numero molto ridotto di poligoni.

Nonostante l'utilizzo dello stadio di geometry processing sulla GPU, in dipendenza dall'entità del tassellamento l'overhead introdotto può essere troppo elevato per ottenere prestazioni accettabili in applicazioni interattive e dunque un tassellamento troppo fitto non è attuabile all'interno dei geometry shader. Proprio per questo motivo le ultime versioni di OpenGL e Direct3D hanno introdotto degli stadi dedicati al tassellamento nella pipeline di rendering (come visto nel paragrafo 1.5.3).

Come ottimizzazione, spesso il tassellamento della geometria a tempo di rendering è un tassellamento *adattivo*, che cioè opera in modo diverso in dipendenza dal punto di vista dell'osservatore (è inutile eseguire un tassellamento particolarmente costoso se l'utente non può comunque apprezzarne i risultati).

2.3 Trasparenze

Un aspetto che fino ad ora abbiamo trascurato è la gestione di oggetti che risultano (almeno parzialmente) trasparenti, ossia tali che la luce è in parte capace di passargli attraverso.

Quando si ricorre ad algoritmi complessi di illuminazione globale come il ray tracing, anche la gestione delle trasparenze viene supportata in modo nativo e non

bisogna ricorrere a tecniche ulteriori. Purtroppo questi algoritmi, come già detto, non sono ancora adatti ad essere utilizzati in applicazioni interattive di grafica tridimensionale perché risultano troppo time-consuming e penalizzano fortemente il frame rate.

Il metodo classico per gestire le trasparenze in modo efficiente si basa sul concetto di *alpha blending*. Tipicamente nel frame buffer utilizzato per il rendering è presente un buffer di colori visualizzati a schermo contenente 4 canali: i canali R, G e B sono quelli utilizzati effettivamente per visualizzare le informazioni, tuttavia è normalmente presente anche un canale A (alpha) che viene usato durante le operazioni di rendering.

L'alpha channel viene di solito sfruttato per mescolare il colore di un frammento che sta venendo scritto sul framebuffer col colore precedentemente disponibile sullo stesso per il pixel interessato. L'effetto di trasparenza può allora essere ottenuto evitando di sovrascrivere col colore del nuovo frammento quello presente in precedenza nel framebuffer, ma mescolando i due valori sulla base di un coefficiente di trasparenza per l'oggetto.

I materiali, coi parametri introdotti nel paragrafo 2.1.1, vengono allora potenziati aggiungendo un ulteriore parametro reale t compreso tra 0 ed 1. Questo parametro suggerisce il modo in cui i colori dell'oggetto andranno mescolati con quelli presenti in precedenza nel frame buffer. Un valore di trasparenza pari a 0 significa che l'oggetto risulterà opaco (i colori precedentemente nel framebuffer verrano semplicemente sovrascritti), se invece la trasparenza è pari a 1 significa che l'oggetto è completamente trasparente e risulterà pertanto invisibile (i colori precedentemente nel framebuffer non verranno alterati).

Notare che, quando si fa diffuse texture mapping, l'immagine usata come texture può anche avere il canale alpha. In questo modo (modulando il materiale dell'oggetto, eventualmente parzialmente trasparente) possiamo ottenere oggetti trasparenti solo in alcune zone, in base ai valori nell'alpha channel della texture di diffuse.

Di solito, quando si esegue alpha blending, il colore in arrivo per un dato frammento viene combinato con quello disponibile nel framebuffer come segue (sulla base del valore α nell'alpha channel del nuovo frammento):

$$C' = C_P \cdot \alpha + C \cdot (1 - \alpha)$$

Dove C_P è il colore RGB per il nuovo frammento e C è il valore presente nel framebuffer prima dell'arrivo del nuovo pixel. Viene prodotto il colore C' che va a

sostituirsi a C all'interno del framebuffer.

Ne deriva dunque che un materiale completamente opaco avrà $\alpha = 1$ e per un oggetto completamente trasparente avremo $\alpha = 0$.

Con questa espressione in pratica si sta ignorando il valore dell'alpha channel precedentemente memorizzato nel framebuffer: l'unica cosa che conta è il valore di α per il nuovo frammento e il nuovo valore di alpha scritto nel framebuffer non è dunque di nostro interesse. Naturalmente questo non è l'unico modo di procedere e vi sono anche molti altri metodi per mescolare i colori al fine di ottenere un effetto di trasparenza.

Sembra quindi che la gestione delle trasparenze sia particolarmente semplice. In realtà però vi sono dei problemi di difficile risoluzione che si basano sulla seguente osservazione: un frammento non arriva mai al framebuffer quando fallisce il depth test effettuato nello stadio OpenGL 'per-fragment operations' (vedi fig. 1.10).

Normalmente nei framebuffer utilizzati per il rendering è sempre disponibile un *depth buffer* che viene utilizzato per evitare di disegnare oggetti che sono nascosti da altri oggetti grazie ad un test sulla profondità (*depth test*) eseguito dal sistema grafico (OpenGL nel nostro caso). Il depth test è necessario per evitare che un oggetto A nascosto da un oggetto B compaia davanti allo stesso perchè i frammenti di A vengono disegnati dopo quelli di B durante un frame. Questa funzionalità è fondamentale per visualizzare scene realistiche e dunque bisogna mantenere il depth test attivo anche quando si lavora con materiali trasparenti.

Questo porta ad alcuni problemi. Se infatti un oggetto in parte trasparente viene disegnato prima degli oggetti dietro di esso, questi non compariranno nella scena perché i loro frammenti non riusciranno a superare il depth test dato che un'altro oggetto più vicino al punto di vista dell'utente è già stato disegnato di fronte ad essi (l'oggetto trasparente). Il problema è ancora più grave in quanto lo stesso ragionamento si applica ai poligoni di un singolo oggetto: a differenza di quello che succede nella realtà, i poligoni sul lato back-facing di un solido non saranno visibili per oggetti trasparenti se non vengono disegnati prima di quelli front-facing.

Il modo classico di affrontare il problema è quello di disegnare prima tutti i poligoni opachi di tutti gli oggetti presenti nella scena, dopo di che si procede disegnando i poligoni in qualche modo trasparenti cominciando da quelli più lontani dalla telecamera e arrivando a quelli più vicini (*depth sorting*). Anche questa soluzione tuttavia può non risultare perfetta (nel caso in cui i poligoni trasparenti siano organizzati in modo tale che non esiste un ordine di disegno che evita ogni problema), ma non esaminiamo in dettaglio questi problemi.

Nella VR3Lib il disegno dei poligoni di oggetti trasparenti avviene (per quanto riguarda l'ordinamento) come nel caso di oggetti opachi e non vi è dunque un supporto completo agli oggetti trasparenti. Molte situazioni possono essere gestite semplicemente cambiando l'ordine di disegno degli oggetti nella scena, tuttavia queste considerazioni vengono lasciate all'utente della libreria.

Capitolo 3

Proiezione di Ombre

In questo capitolo esaminiamo un altro problema fondamentale che si incontra quando si cerca di visualizzare scene virtuali ad elevato livello di realismo: il disegno di ombre proiettate da oggetti virtuali su porzioni di loro stessi o su altri oggetti virtuali.

Parliamo di ombre dinamiche, che evolvono in base al movimento relativo delle sorgenti luminose e degli oggetti nella scena. Nel caso di sorgenti e oggetti statici (o privi di movimento relativo) conviene sempre ricorrere alla tecnica del light mapping con ombre precalcolate ottenendo un'elevata qualità ad un costo minimo.

Esiste una enorme varietà di algoritmi pensati per la proiezione di ombre in scene virtuali e negli ultimi anni vi è stato un fortissimo sviluppo di queste tecniche, che assumono sempre più importanza nelle applicazioni di grafica tridimensionale ad elevato realismo. La quantità di diverse tecniche è tale da consentire solamente una breve trattazione in questo capitolo, che non mira a dare un'idea completa sul panorama di metodi disponibili ma piuttosto a fornire alcuni dettagli sul funzionamento delle tecniche ad oggi più diffuse.

Al solito, nell'esaminare gli algoritmi per la proiezione di ombre dobbiamo ricordare che stiamo cercando tecniche utilizzabili in applicazioni interattive di visualizzazione. Non parleremo infatti di tutti i metodi che forniscono ottimi risultati ma sono utilizzabili solo offline (a causa del loro costo computazionale).

Prima di cominciare è bene evidenziare che alcune tecniche di illuminazione globale (vedi par. 2.1.4) applicate sostanzialmente offline, come il ray tracing e l'ambient occlusion, sono in grado di generare ombre in modo implicito: nonostante lo scopo di queste tecniche sia quello di illuminare oggetti in modo realistico, come effetto collaterale della loro esecuzione spesso abbiamo la generazione di ombre

proiettate. Vari sono stati gli sforzi per migliorare e rendere più efficienti questi algoritmi ad elevato livello di realismo anche per quanto riguarda la generazione di ombre. Non approfondiamo ulteriormente queste tecniche perché purtroppo sono ancora adatte solamente ad applicazioni di rendering non in tempo reale, però evidenziamo che in futuro è molto probabile che verranno applicate su vasta scala anche in applicazioni interattive (grazie anche al continuo potenziamento degli adattatori grafici).

3.1 Tecniche di Base

Per introdurre le tecniche moderne di proiezione di ombre, bisogna partire dai metodi classici proposti per la prima volta più di 30 anni fa; ovviamente questi algoritmi fornivano risultati nettamente inferiori alle tecniche disponibili oggi, però i metodi moderni si sono sviluppati sostanzialmente estendendo queste antiche tecniche e dunque bisogna capirne bene il funzionamento prima di procedere.

Gli algoritmi classici per la proiezione di ombre in tempo reale sono sostanzialmente due (come descritto nelle pubblicazioni [28, 29]):

- Shadow mapping
- Shadow volumes

Vediamo queste due tecniche una per volta in questa sezione; ognuna delle due ha avuto nel tempo svariate modifiche e potenziamenti, producendo diversi algoritmi che ad oggi possono venir utilizzati per generare ombre realistiche.

3.1.1 Shadow Mapping

L'algoritmo di shadow mapping venne inizialmente proposto da L. Williams nel 1978 [30]. Oggi sono disponibili in rete diverse implementazioni di tale algoritmo, tra le quali anche delle versioni basate su shader.

L'algoritmo di shadow mapping è un algoritmo di *multi-pass rendering*: il rendering di una scena dove si vogliano generare ombre mediante l'algoritmo di shadow mapping avviene in diversi passi. Nel seguito descriviamo gli algoritmi per la proiezione di ombre supponendo di avere a che fare con una scena in cui una singola sorgente luminosa causa la proiezione di ombre da parte degli oggetti nella scena. Se diverse sorgenti sono presenti nell'ambiente virtuale, bisognerà estendere le procedure sotto descritte per gestire questa molteplicità.

Va sottolineato che che il concetto di 'sorgente luminosa' in questo contesto viene inteso come entità che causa la proiezione di ombre da parte degli oggetti nella scena; l'illuminazione degli oggetti può avvenire indipendentemente da queste entità sfruttando qualunque tecnica descritta in sez. 2.1.

L'algoritmo procede come segue (in una implementazione basata su shader):

Algoritmo 3.1 Shadow mapping

---

1. PRIMO PASSO DI RENDERING
   Si renderizza off-screen la scena dal punto di vista della sorgente luminosa, salvando i valori di profondità D_s in un buffer che prende il nome di *shadow map*.

2. SECONDO PASSO DI RENDERING
   Si renderizza la scena dal punto di vista dell'osservatore, confrontando per ogni frammento renderizzato il valore di profondità del frammento dal punto di vista della luce D_f con il valore memorizzato nella shadow map.

   $$\begin{cases} D_f > D_s & \Rightarrow \quad fragment\ is\ shadowed \\ D_f \leq D_s & \Rightarrow \quad fragment\ is\ lit \end{cases}$$

   Se il valore di D_f supera quello memorizzato nella shadow map significa semplicemente che vi è un oggetto più vicino alla luce del frammento che sta attualmente venendo disegnato e che il frammento è investito dall'ombra proiettata da questo oggetto più vicino.

In questo contesto con *profondità* si intende tipicamente il valore di z_{win} ottenuto dopo la trasformazione di proiezione, la divisione prospettica e la trasformazione di viewport (vedi par. 1.4.2). Questo però non è affatto obbligatorio e si possono utilizzare anche metriche diverse all'interno della shadow map. Con la tecnica di shadow mapping, le informazioni di profondità vengono quindi non solo usate come al solito per prevenire il disegno di oggetti nascosti (tramite il depth test), ma anche memorizzate nella shadow map per venire usate in un successivo passo di rendering.

Implementazioni moderne della tecnica di shadow mapping prevedono di memorizzare i valori di profondità in una shadow map da usare poi durante l'effettivo passo di rendering come texture bidimensionale. Quando si lavora in questo modo bisogna prestare attenzione al fatto che le comuni tecniche di filtraggio delle texture che mescolano i colori di diversi texel non possono venire applicate alla shadow map, perché si produrrebbero dei valori di profondità intermedi che invaliderebbero le ombre ottenute durante la fase di rendering.

In fig. 3.1 è riportata una scena disegnata sfruttando la tecnica di shadow mapping per la proiezione delle ombre.

Figura 3.1: Shadow mapping

In figura sono ben visibili alcuni dei difetti di questa tecnica di base; la seguente è una lista completa dei problemi che si possono incontrare:

- Artefatti di *aliasing*
  Quando un singolo texel di shadow map copre diversi pixel dello schermo, dato che la mappa non viene filtrata prima di essere applicata alla scena, le ombre appaiono frastagliate sulla base dei texel di shadow map usati per il confronto.

- *Self-shadowing*
  Quando un singolo texel di shadow map copre un vasto range di profondità nella scena (come ad esempio il caso di un piano quasi parallelo alla direzione della luce) il test $D_f \leq D_s$ può fallire anche se in realtà non c'è nessun oggetto a bloccare la luce, generando delle zone d'ombra dove non dovrebbero essere. Inoltre, il problema può verificarsi anche in casi ottimali grazie alla quantizzazione dei valori di profondità all'interno della shadow map. L'artefatto a cui questo problema da origine viene tipicamente chiamato *shadow acne*.

- Proiezione *non omnidirezionale*
  Quando si posiziona nella scena una sorgente luminsa, ci si aspetta che le ombre vengano proiettate da tutti gli oggetti attorno ad essa nel modo corretto. Sfruttando la tecnica dello shadow mapping di fatto si renderizza la scena come

vista dalla luce con una matrice di proiezione, che definisce un ben preciso volume di vista all'interno del quale le ombre vengono proiettate correttamente. Le ombre degli oggetti esterni a tale volume di vista non risulteranno visibili.

- Le ombre sono *'hard'*
  Un dato frammento è completamente in ombra oppure completamente illuminato, non abbiamo alcuna zona di penombra con la tecnica classica di shadow mapping. Ombre dove compaiono anche zone di penombra vengono comunemente definite *soft shadows*, mentre quelle che stiamo esaminando ora sono decisamente *hard shadows*.

- *Shadow Flickering*
  Nel caso di oggetti mobili nella shadow map, visto che la minima quantità di informazione che possiamo alterare è un texel della mappa, avremo che l'ombra evolve in modo discreto (risulta 'tremolante'). Questo artefatto è evidente quando la risoluzione della shadow map è bassa rispetto al punto di vista dell'osservatore, ma tende a farsi meno pronunciato quando la risoluzione viene aumentata e si combatte anche introducendo delle regioni di penombra (è meno visibile quando i bordi dell'ombra non sono netti).

Per quanto riguarda la complessità computazionale della tecnica, si ha una dipendenza dalle dimensioni della texture scelta per immagazzinare i valori di profondità (shadow map). Normalmente gli shader costruiti per ottenere la shadow map da utilizzare durante la vera fase di rendering saranno molto più semplici di quelli usati durante l'aggiornamento del framebuffer della finestra OpenGL per il disegno della scena. Lo shadow mapping è il metodo meno costoso che vediamo per disegnare ombre.

3.1.2 Shadow Volumes

Proposta inizialmente da F. C. Crow nel 1977 [31], questa tecnica opera in modo completamente diverso dalla precedente e si basa sulla creazione di poligoni che delimitano le zone d'ombra.

Prima di vedere l'algoritmo, bisogna introdurre il concetto di *silhouette edge*. Un silhouette edge è uno spigolo di una mesh che giace tra un poligono front-facing e un poligono back-facing (con la terminologia introdotta nel par. 1.4.1). L'insieme di questi spigoli forma quella che si dice la silhouette della mesh (rispetto ad un determinato punto di vista).

Il modo classico di implementare questo algoritmo sfrutta lo *stencil buffer*, un buffer che può essere affiancato al depth buffer e ai buffer di colore all'interno del framebuffer. Lo stencil buffer contiene un valore per ogni pixel dello schermo e tipicamente è possibile scegliere il numero di bit da utilizzare in ognuna di queste celle; il valore memorizzato viene normalmente interpretato come un intero senza segno. Proprio come per il depth buffer, allo stencil buffer è associato un test che viene eseguito automaticamente (quando abilitato) dopo il fragment shading, nello stadio 'per-fragment operations' (vedi fig. 1.10). A differenza del depth test, è possibile configurare una serie di parametri di funzionamento di questo test e aggiornamento dello stencil buffer. Nella pratica lo stencil buffer può venire usato per costruire una maschera da usare durante il rendering, che impedisce il disegno di alcuni pixel.

L'algoritmo di shadow volumes può allora essere descritto come segue:

Algoritmo 3.2 Shadow volumes

---

1. Creazione dei poligoni del volume d'ombra
   Si individuano tutti gli spigoli che formano la silhouette dal punto di vista della sorgente luminosa e si estendono questi spigoli allontanadosi dalla luce per formare una serie di poligoni che delimitano l'ombra proiettata.

2. Rendering della scena in ombra
   Si disegna la scena come se fosse completamente in ombra; nel caso di illuminazione diretta questo significa ad esempio considerare solo l'illuminazione ambientale. Come effetto collaterale si ha l'aggiornamento del depth buffer con tutti i valori di profondità degli oggetti visibili.

3. Costruzione della maschera nello stencil buffer
   Sfruttando le informazioni presenti nel depth buffer, si costruisce nello stencil buffer una maschera da usare per il passo successivo senza alterare il contenuto del depth buffer o dei color buffers associati al framebuffer.

4. Rendering della scena illuminata sulla base della maschera
   Si disegna nuovamente la scena, ma stavolta come se fosse completamente illuminata, ripulendo prima il depth buffer. I frammenti che non superano lo stencil test non verranno disegnati e risulteranno delle zone in ombra (il colore rimane quello del passo 1).

Notare che il passo 1 deve essere fatto ad ogni frame perchè stiamo assumendo che vi sia movimento relativo continuo tra la sorgente luminosa e gli oggetti nella scena. Potremmo tuttavia pensare di ottimizzare l'algoritmo ricalcolando i poligo-

ni del volume d'ombra di ogni oggetto solamente quando si ha effettivamente del movimento.

Sono stati proposti diversi approcci per la costruzione della maschera nello stencil buffer, e alcuni di essi possono fallire in casi particolari ottenendo un risultato erroneo durante il rendering. Realizzare una versione robusta dell'algoritmo di shadow volumes (che possa funzionare in tutti i casi) non è affatto semplice [32]. Alcuni approcci diffusi sono i seguenti (non ne descriviamo il funzionamento):

- *depth pass* (Heidmann, 1991);
- *depth fail* o *Carmack's reverse* (Bilodeau, Songy, Dietrich e Carmack, 1999-2000);
- *exclusive-or.*

In dipendenza dall'approccio utilizzato è anche possibile che si debbano aggiungere dei poligoni ai volumi d'ombra generati (front-cap e/o back-cap) per costruire eventualmente una forma chiusa.

Una scena renderizzata con un algoritmo di shadow volumes è riportata in fig. 3.2.

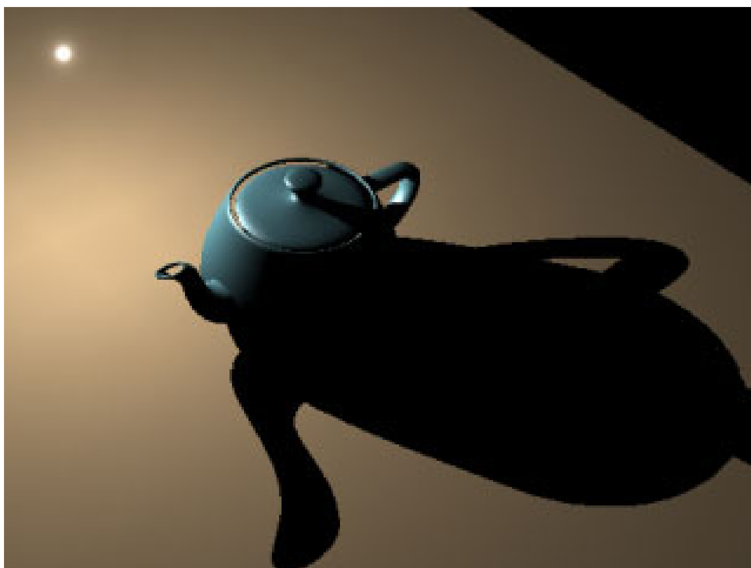


Figura 3.2: Shadow volumes

Di tutti i difetti evidenziati nella tecnica di shadow mapping, in questo caso l'unico che permane è che le ombre generate sono intrinsecamente *hard shadows*, non abbiamo alcun problema di aliasing, di self shadowing o di altro tipo.

Vi sono però dei problemi aggiuntivi:

- data la natura geometrica della procedura, la complessità della scena ha un'influenza determinante sulle prestazioni dell'algoritmo e spesso il lavoro di creazione dei poligoni d'ombra può comportare una riduzione significativa del frame rate;
- tipicamente l'algoritmo di shadow mapping è più veloce, anche per il fatto che i poligoni d'ombra che definiscono gli shadow volumes spesso coprono porzioni molto grandi dello schermo e il loro disegno per la costruzione della maschera nello stencil buffer può richiedere un tempo non indifferente;
- vista la dipendenza dalla geometria degli oggetti nella scena, ci si aspettano malfunzionamenti qualora gli oggetti nella scena non siano tutti composti da mesh *chiuse* (dove ogni spigolo fa parte di esattamente due triangoli) e risulta difficile ottenere un funzionamento corretto in tutte le situazioni.

Questa naturalmente è la versione di base dell'algoritmo di shadow volumes; nel corso degli anni sono state proposte diverse estensioni (come anche per lo shadow mapping) per ottenere ombre sempre più convincenti.

3.1.3 Ombre e VR3Lib

Una tematica molto interessante in entrambe le tecniche viste sopra è la questione delle *soft shadows*. Ombre convincenti, da poter utilizzare in un motore di rendering in tempo reale ad elevato livello di realismo, possono essere ottenute solamente quando si riesce a generare della penombra nella scena. Sono state proposte diverse estensioni all'algoritmo di shadow mapping (vedi sez. 3.3) per ottenere soft shadows, questa tecnica è infatti maggiormente flessibile e GPU-friendly rispetto agli shadow volumes.

Per quanto riguarda la generazione di soft shadows mediante estensioni alla tecnica di shadow volumes, solo recentemente sono state proposte alcune soluzioni soddisfacenti [33, 34, 35], ma con questi approcci si complica notevolmente la gestione della procedura e la complessità geometrica della scena ha un'impatto ancora superiore sulle prestazioni dell'algoritmo.

Visto che l'algoritmo implementato nella nuova versione della VRLib è un'innovativa e recente estensione della tecnica di shadow mapping, nel seguito ci concentriamo esclusivamente su tematiche che riguardano la tecnica proposta da Williams. Una soluzione basata su shadow mapping è preferibile per i nostri scopi perché vogliamo ottenere prestazioni che consentano l'utilizzo dell'applicazione in modo interattivo,

riuscendo comunque a visualizzare soft shadows convincenti anche quando la complessità geometrica della scena è significativa e siamo in presenza di diverse sorgenti luminose.

Prima di concludere questa sezione, accenniamo inoltre al fatto che sono stati proposti anche *approcci ibridi*, che utilizzano concetti propri dello shadow mapping e dello shadow volumes allo stesso tempo. Questi approcci, per quanto interessanti, non risultano vantaggiosi per i nostri scopi rispetto alle moderne tecniche di shadow mapping e dunque non ne discutiamo ulteriormente. Un esempio di tecnica ibrida è quella proposta da McCool [36].

3.2 Contributi alla Tecnica di Shadow Mapping

In sez. 3.1 abbiamo accennato al fatto che l'algoritmo di shadow mapping è stato progressivamente migliorato ed esteso per ottenere ombre sempre più realistiche. In questa sezione vediamo alcune di queste estensioni che risolvono molti dei problemi che si incontrano utilizzando le shadow map.

3.2.1 Shadow Biasing

Abbiamo discusso del problema di self-shadowing evidenziando come sia dovuto ad una duplice azione:

- imprecisioni all'interno del depth buffer possono causare la visualizzazione di ombre erronee;
- texel di shadow map che coprono una vasta profondità geometrica (come nel caso di un piano quasi parallelo alla direzione della luce) creano ombre dove non dovrebbero comparire.

Per gestire il problema, nella pratica si può ricorrere a diverse tecniche.

Una prima soluzione può essere quella di disegnare, durante il rendering della shadow map, solamente i poligoni back-facing. Supponendo allora di avere una sfera, nella shadow map i valori di profondità saranno quelli corrispondenti alla parte non visibile della sfera, guardando dalla luce. Durante il rendering non avremo il problema del self-shadowing per frammenti della sfera che vengono illuminati direttamente dalla sorgente luminosa. Con questo metodo, gli artefatti di shadow acne possono verificarsi solamente sulla parte non illuminata direttamente dalla luce (che comunque dovrebbe risultare completamente in ombra). Purtroppo, usando

questa tecnica bisogna limitare la geometria degli oggetti (che deve risultare chiusa in modo tale da avere sempre tutte le facce back-facing necessarie).

Una tecnica molto comune applicabile in generale invece prende il nome di *shadow biasing* e prevede di aggiungere ai valori di profondità salvati nella shadow map un termine additivo detto *bias* che sia tale da evitare i problemi di self shadowing e shadow acne. Teoricamente il bias applicato dovrebbe essere proporzionale al massimo range di profondità coperto da un singolo texel della shadow map, ma spesso accade che tale valore viene impostato con un fine-tuning sulla particolare scena da parte dell'utente.

La quantità di bias necessario per evitare qualsiasi artefatto di self-shadowing in una particolare scena può essere arbitrariamente elevata; l'applicazione di un certo bias ai valori di profondità nella shadow map causa però come effetto collaterale l'allontanamento delle ombre renderizzate dai loro oggetti proiettanti (*casters*). Si genera dunque un artefatto che da l'impressione che gli oggetti stiano fluttuando sopra la superfice che riceve l'ombra (su cui sono appoggiati) e prende il nome di *Peter Panning* (dal celebre personaggio di favole per bambini). L'artefatto è evidenziato in fig. 3.3.

Figura 3.3: Peter Panning

Un forte miglioramento si può ottenere fissando un bias per ogni texel di shadow map dipendente dall'inclinazione (rispetto alla direzione della luce) del poligono che funge da caster [37]. Tuttavia anche con questa tecnica non si risolve del tutto il problema.

Bisogna comunque notare che spesso una texture standard ad 8 bit per pixel (a singola componente) non risulta sufficiente per immagazzinare i valori di profondità per problemi di quantizzazione e precisione. Normalmente si ricorre infatti a texture con 16 o 24 bit per pixel o addirittura a texture con un numero in virgola mobile a precisione singola per ogni texel. Con questo accorgimento si riescono normalmente ad evitare i problemi correlati alla quantizzazione dei valori all'interno della shadow map.

3.2.2 Perspective Shadow Maps (PSM)

Il problema forse più grave che si incontra lavorando con le shadow map è quello dell'aliasing. Supponiamo adesso di non avere interesse a realizzare ombre realistiche che includano anche zone di penombra, ma di voler solamente ottenere delle ombre hard che risultino il più precise possibile.

Il problema dell'aliasing emerge fondamentalmente quando la risoluzione usata per il rendering della shadow map è troppo bassa con riferimento alla porzione di scena visualizzata nella finestra OpenGL. Un metodo semplice per nascondere il problema è quello di aumentare la risoluzione della shadow map, questo però incrementa di conseguenza il costo computazionale della tecnica e talvolta si possono anche raggiungere i limiti dell'implementazione del sistema grafico senza riuscire ad ottenere risultati soddisfacenti.

L'artefatto di aliasing (evidente in fig. 3.1) è causato da molti fattori. Uno dei principali contributi è chiamato *perspective aliasing* ed è dovuto fondamentalmente alla posizione della sorgente luminosa e dell'osservatore nei confronti dell'oggetto osservato su cui viene proiettata un'ombra. Sono state proposte alcune tecniche molto interessanti per ridurre il contributo di questo tipo di aliasing, in particolare si parla di *perspective shadow map* [38, 39].

Il principio di base su cui si fonda la tecnica di perspective shadow mapping è quello di disegnare i valori di profondità della scena nella shadow map solo dopo che gli oggetti hanno subito le ordinarie trasformazioni che li portano nello spazio post-proiezione dal punto di vista dell'osservatore (rappresentabili tramite una singola matrice in coordinate omogenee); naturalmente, anche le sorgenti luminose andranno trasformate utilizzando la stessa matrice prima di graficare la shadow map come visto in precedenza.

Come risultato, quello che si ottiene è una shadow map in cui gli oggetti vicini alla telecamera occupano uno spazio maggiore di quelli lontani. Di fatto in questo modo

si riduce il problema dell'aliasing perché gli oggetti vicini all'osservatore avranno a disposizione un numero maggiore di texel di shadow map durante il rendering.

In fig. 3.4 abbiamo un confronto tra lo *uniform shadow mapping* (quello ordinario, in alto) e la tecnica di perspective shadow mapping (in basso). Le immagini sulla sinistra sono una rappresentazione monocromatica delle shadow map generate (dove un colore più chiaro significa un oggetto più vicino alla sorgente luminosa). Notare come la shadow map generata con questa nuova tecnica fornisca una risoluzione maggiore per gli oggetti vicini al punto di vista dell'utente, nascondendo in parte il problema dell'aliasing.

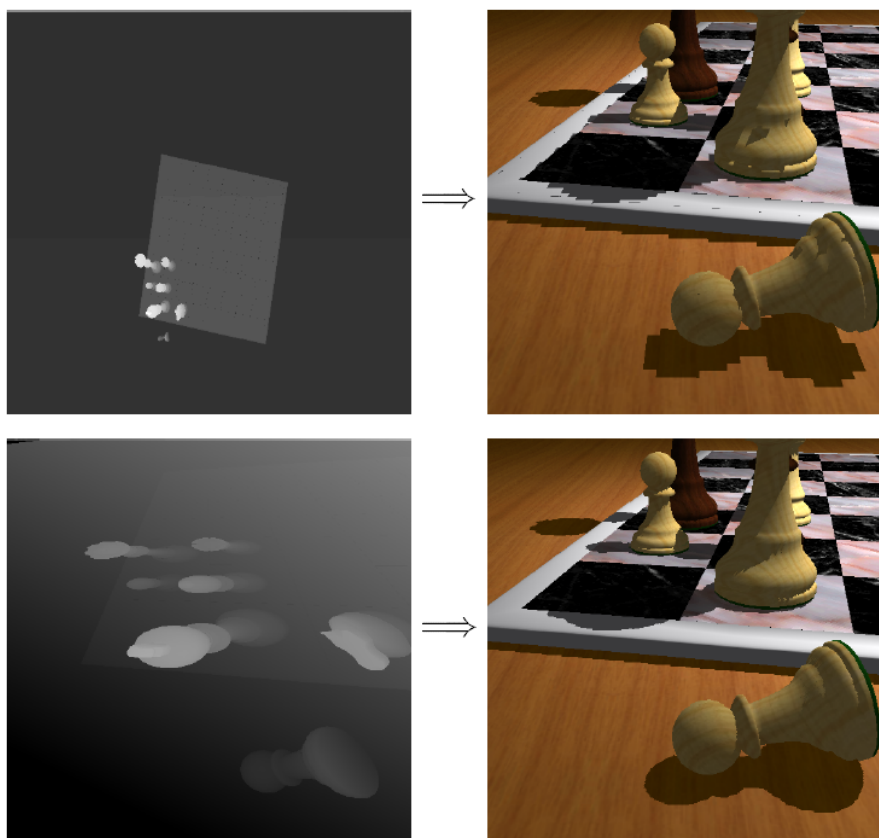


Figura 3.4: Perspective shadow mapping

Questo metodo, per quanto potente, ha una serie di casi particolari che richiedono una gestione opportuna. Non sempre è possibile ottenere un miglioramento così evidente come quello in figura 3.4 e talvolta si arriva allo stesso risultato che si avrebbe con la tecnica di base. Il problema del biasing viene aggravato dalle perspective shadow map perché l'area occupata da un texel in world space non è

uniforme, ma comunque rimane gestibile sfruttando un bias non costante calcolato opportunamente per ogni texel.

Naturalmente, quello delle perspective shadow map non è l'unico lavoro che è stato svolto per risolvere il problema dell'aliasing nelle shadow map. Esempi sono:

- *Adaptive shadow maps (ASM)* [40]
  Si costruisce una struttura gerarchica al posto della tradizionale shadow map intesa come immagine, purtroppo il costo computazionale di questa tecnica ne impedisce l'utilizzo in applicazioni di rendering in tempo reale quando non si ha a disposizione un adattatore grafico altamente performante.

- *Trapezoidal shadow maps (TSM)* [41]
  Si approssima il volume di vista dell'osservatore, come visto dalla sorgente luminosa, con un trapezoide che viene usato poi per calcolare la shadow map nel modo opportuno. Questa operazione risulta in una risoluzione migliorata sia per oggetti vicini all'osservatore che per oggetti lontani all'interno della shadow map; inoltre non si peggiora il problema del biasing (come nel caso del perspective shadow mapping).

- *Light-space perspective shadow maps (LiSPSM)* [42]
  Il metodo è simile a quello delle PSM, ma si opera con una trasformazione prospettica diversa rispetto a quella dell'osservatore ottenendo una shadow map diversa che può venire usata in modo simile al caso delle PSM, evitando però diversi problemi che si hanno col metodo di base appena presentato. Purtroppo, la texture di shadow map in questo caso viene sfruttata in misura inferiore.

Il motivo per cui non entriamo maggiormente nel dettaglio delle tecniche di PSM, TSM e LiSPSM è che per i nostri scopi tutti i metodi che si basano su trasformazioni dipendenti dal punto di vista dell'utente sono inapplicabili. Vedremo infatti in sez. 3.3 che le tecniche moderne di shadow mapping generano soft shadows basandosi sulla dimensione dei texel della shadow map e se tale dimensione viene alterata muovendo la telecamera si ottengono regioni di penombra più o meno estese a seconda della posizione e dell'orientamento del volume di vista.

3.2.3 Cascaded Shadow Maps (CSM)

Quando la tecnica dello shadow mapping viene utilizzata per proiettare ombre in ambienti virtuali molto vasti, diventa complesso regolarne i parametri al fine di otte-

nere ombre soddisfacenti, dove non siano visibili artefatti evidenti. Spesso in questi casi occorre utilizzare delle shadow map con una risoluzione molto elevata al fine di ottenere un risultato accettabile, ma così facendo si aumenta significativamente il costo computazionale dell'algoritmo in quanto il disegno della shadow map risulta più time-consuming.

La tecnica delle *cascaded shadow map* [43, 44] opera dividendo il volume di vista dell'osservatore in diverse 'fette' (slices) sulla base della distanza dal punto di vista e generando una shadow map distinta per ognuna di esse. Si fa in modo di generare una shadow map apposita per gli oggetti nelle immediate vicinanze producendo delle ombre dettagliate. Allontanadosi dal punto di vista, si inseriscono all'interno delle shadow map regioni sempre più vaste e si producono quindi ombre più imprecise per gli oggetti lontani (che non dovrebbero venire apprezzate dalla posizione dell'osservatore).

Notare che in pratica quello che si sta facendo è sostituire l'esigenza di generare una singola shadow map avente una risoluzione molto elevata con un approccio basato su diverse shadow map a bassa risoluzione. A tempo di rendering si andrà a selezionare la shadow map associata alla giusta slice del volume di vista sulla base della posizione del frammento.

Nonostante l'approccio CSM sia utile per combattere diversi tipi di artefatti negli algoritmi di shadow mapping (anche del tipo che vedremo più avanti parlando di soft shadows), si ha anche una riduzione dell'effetto di aliasing visto in precedenza. Le tecniche come PSM, TSM e LiSPSM possono venire applicati insieme alle CSM ad ogni singola slice, ma il miglioramento della qualità visiva non è normalmente significativo mentre abbiamo una decisa complicazione dell'algoritmo. Tipicamente le CSM non vengono dunque combinate con approcci di *warping*[14] della shadow map come quelli sopra.

In fig. 3.5 è illustrato il funzionamento di base dell'algoritmo CSM con una luce direzionale come quella del sole.

[14] Distorsione, deformazione.

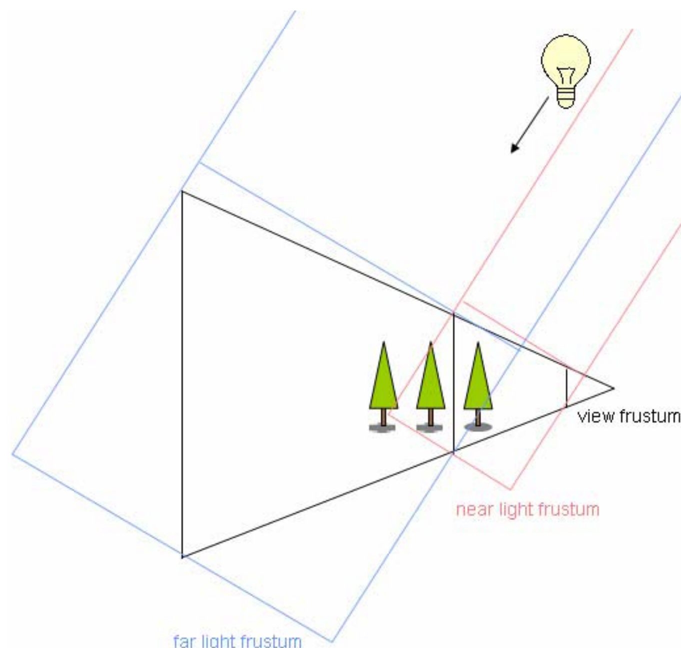


Figura 3.5: Cascaded shadow mapping

Questa tecnica, a differenza dei metodi di warping visti in precedenza, viene spesso applicata anche insieme ai metodi che vedremo successivamente per generare soft shadows (che generano penombra sulla base dei texel nella shadow map). Nonostante la dimensione del texel non sia uniforme su tutta la scena virtuale, possiamo trattare in modo diverso le shadow map generate per ogni slice e ottenere la giusta quantità di penombra in ogni caso. Bisogna però considerare che con la tecnica di CSM bisogna affrontare alcuni problemi ulteriori (come i punti di contatto tra diverse slices), e inoltre i vantaggi di questo metodo sono apprezzabili solamente quando si renderizzano paesaggi molto vasti che richiedono ombre completamente dinamiche (situazione non molto comune).

3.2.4 Omnidirectional Shadow Maps

Il problema di realizzare shadow map omnidirezionali deriva dall'esigenza di proiettare ombre generate da una sorgente di luce in tutte le direzioni. Fino ad ora abbiamo visto l'algoritmo di shadow mapping affermando che la shadow map viene calcolata sulla base della posizione della luce con certi parametri di proiezione (che definiscono di fatto un volume all'interno del quale le ombre degli oggetti vengono proiettate).

Quando abbiamo una sorgente di luce al centro di una scena virtuale e circondata da oggetti, vorremmo ottenere una tecnica capace di proiettare le ombre di tutti gli oggetti attorno alla sorgente.

Con quanto visto fino ad ora, ciò non è possibile perché la shadow map consiste in un'immagine catturata dal punto di vista della luce con una certa matrice di proiezione. Si potrebbe pensare che aumentando il campo di vista definito dalla matrice di proiezione si possa arrivare ad una shadow map omnidirezionale. Purtroppo però il valore limite del campo di vista (per come sono costruite le matrici di proiezione) è 180 gradi. Inoltre, avvicinandosi al valore limite di 180 gradi (che non può essere utilizzato) si verifica una deformazione progressiva del risultato, che può causare artefatti indesiderati quando si va ad utilizzare la shadow map.

Per realizzare shadow mapping omnidirezionale, la tecnica più comune è quella di ricorrere ad una *cube shadow map* [45, 46]. Si renderizza una shadow map classica per ogni faccia del cubo utilizzando un angolo di vista di 90 gradi e si costruisce una cube map che contiene tutte le shadow map ottenute; durante il rendering si accede a questa cube map per ricavare i valori di profondità necessari per il confronto.

Un'esempio di applicazione di questa tecnica è riportato in fig. 3.6.

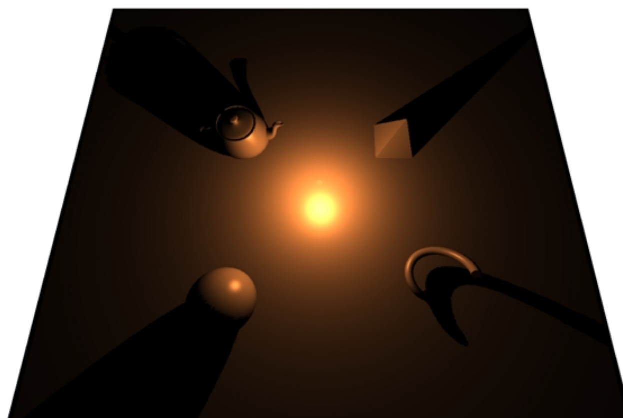


Figura 3.6: Omnidirectional shadow mapping

Questa tecnica è applicabile anche in combinazione con i metodi che vediamo in sez. 3.3 per ottenere soft shadows.

Sfruttare una cube shadow map non è equivalente in generale ad utilizzare 6 diverse shadow map ordinarie. Le due procedure possono fornire lo stesso risultato nel caso di shadow mapping standard (par. 3.1.1, trascurando problemi di efficienza), ma il risultato è diverso quando si sfruttano algoritmi complessi che si basano sul filtraggio della shadow map come quelli di cui parliamo nella prossima sezione (in tal

caso il filtraggio deve avvenire sulla cube shadow map per gestire opportunamente le interazioni tra le varie facce).

Bisogna anche aggiungere che spesso la gestione di sorgenti omnidirezionali può essere emulata associando ad ogni sorgente più di una shadow map calcolata nel modo consueto e orientando i volumi di proiezione delle ombre in modo opportuno (per contenere gli oggetti di interesse). Questo ci consente di risparmiare sul rendering di alcune facce della cube shadow map che potrebbero risultare inutili.

3.3 Soft Shadow Mapping

Un algoritmo per la proiezione di ombre che possa dirsi capace di ottenere ombre convincenti deve essere in grado di generare soft shadows. Come già detto, questo equivale a generare delle regioni di penombra all'interno della scena, in corrispondenza dei bordi delle ombre hard che si sono ottenute fino ad ora.

A questo scopo sono stati proposti diversi potenziamenti dell'algoritmo di shadow mapping: una buona raccolta si trova in [47]. Per quanto ci riguarda, introduciamo prima una tecnica che da ottimi risultati ma ad un costo piuttosto elevato e vediamo una sua estensione per generare penombra di ampiezza variabile; a seguire presentiamo alcune tecniche di diverso tipo che recentemente sono diventate molto popolari; infine accenniamo alla tecnica che è stata implementata nella nuova VR3Lib (molto innovativa, ancora non opportunamente documentata in letteratura): l'algoritmo di *Exponential Variance Shadow Mapping (EVSM)*.

Naturalmente, i metodi presentati in questo paragrafo sono solamente quelli più comuni e che risultano migliori in termini di qualità ed efficienza. In letteratura sono disponibili anche altre tecniche, ma noi ce ne disinteressiamo.

Mettiamo in evidenza che un algoritmo che genera soft shadows contribuisce in modo sostanziale al livello di realismo di una scena, ma non è assolutamente necessario che i risultati siano fisicamente accurati. Il nostro obiettivo è quello di ottenere risultati convincenti, anche se non esatti da un punto di vista ottico.

Generando zone di penombra il problema dell'aliasing diventa molto meno evidente anche quando la risoluzione della shadow map è bassa, per cui la costruzione di regioni di penombra realistiche assume maggiore importanza rispetto alla minimizzazione degli artefatti di aliasing sulla base del punto di vista dell'osservatore (lo stesso vale per lo shadow flickering).

Talvolta in letteratura quando si parla di soft shadows ci si riferisce esclusivamente a tecniche capaci di generare aree di penombra ad ampiezza variabile, in cui

le ombre diventano tanto più nette quanto più il caster e l'oggetto ricevitore sono vicini tra loro. Noi invece con l'espressione 'soft shadows' ci riferiamo a tutte le tecniche capaci di generare penombra. I metodi che comportano una variazione della dimensione delle regioni di penombra sulla base della distanza tra caster e receiver si dicono invece metodi con penombra ad ampiezza variabile.

3.3.1 Percentage-Closer Filtering (PCF)

Nel paragrafo 3.1.1 abbiamo messo in evidenza come molte tecniche di filtraggio classiche (come il filtraggio bilineare) non siano applicabili alle shadow map: operando un prefiltraggio sul depth buffer memorizzato nella shadow map si introducono degli artefatti e non si riesce comunque a generare delle aree di penombra.

Quello che si può pensare di fare è filtrare non i valori di profondità salvati nella shadow map, ma il risultato binario di vari confronti in un intorno del texel che stiamo controllando. Durante il rendering, quando si accede ad una shadow map per scoprire se un frammento va considerato in ombra o meno, possiamo effettuare il controllo $D_f \leq D_s$ non solo una volta, ma su un insieme di texel di shadow map nell'intorno di quello corrispondente al frammento in esame.

Durante il confronto nell'intorno del texel examinato, si costruisce una struttura che contiene i vari risultati raccolti fino a quel momento, come segue (i-esimo texel nell'intorno del frammento):

$$\begin{cases} D_f > D_s^i + bias & \Rightarrow & result = 0 \\ D_f \leq D_s^i + bias & \Rightarrow & result = 1 \end{cases}$$

Dopo di che si filtra questa struttura sfruttando un certo filtro di convoluzione (che nel caso più semplice opera calcolando il valore medio dei risultati di tutti i confronti). Il risultato sarà un valore compreso tra 0 ed 1 che indica la percentuale di texel nella regione esaminata che sono più lontani dalla sorgente luminosa rispetto al frammento considerato, questa è la procedura di *pencertage closer filtering* [48, 49]. Il valore ottenuto può essere direttamente usato come fattore moltiplicativo per calcolare 'quanto' il frammento è illuminato.

A tutti gli effetti questa tecnica si configura come un filtraggio della shadow map durante lo shading dei frammenti; va notato che al solito è impossibile effettuare qualsiasi tipo di prefiltraggio sulla mappa perché il suo contenuto rimane lo stesso (valori di profondità).

Aumentando le dimensioni dell'intorno esaminato, è inoltre possibile rendere l'ombra sempre più 'soft' aumentando lo spazio occupato dalle regioni di penombra. Quando si usa un'ampiezza del filtro piuttosto elevata possono comparire degli artefatti simili a bande nelle zone di penombra. Per eliminare questa imprecisione normalmente si opera secondo il *metodo Monte Carlo* scegliendo i campioni da filtrare in modo casuale (di fatto in questo modo si sostituiscono le bande con un rumore ad elevata frequenza spaziale e variabile nel tempo che agli esseri umani risulta difficile percepire).

Purtroppo la tecnica PCF ha dei difetti piuttosto evidenti:

- il problema del self-shadowing viene ereditato dalla tecnica originale di shadow mapping, però in questo caso il bias (anche se fisso) dovrebbe essere proporzionale alla profondità coperta dall'intera regione di filtraggio nella shadow map, per cui superiore rispetto al caso precedente;
- quando si usa una regione di filtraggio significativa, il processo rischia di essere troppo lento perchè l'operazione va ripetuta per ogni frammento che viene disegnato a schermo.

Nonostante la semplicità della tecnica, quando si vogliono ottenere zone di penombra piuttosto estese si osservano problemi di efficienza, e il frame rate cala velocemente sotto i minimi livelli accettabili. Soprattutto per le scarse prestazioni di questo algoritmo, nella nuova libreria si usa una tecnica diversa e più raffinata che non esegue il filtraggio a tempo di fragment processing (vedi oltre). In fig. 3.7 è riportato un esempio di scena renderizzata sfruttando la tecnica PCF.

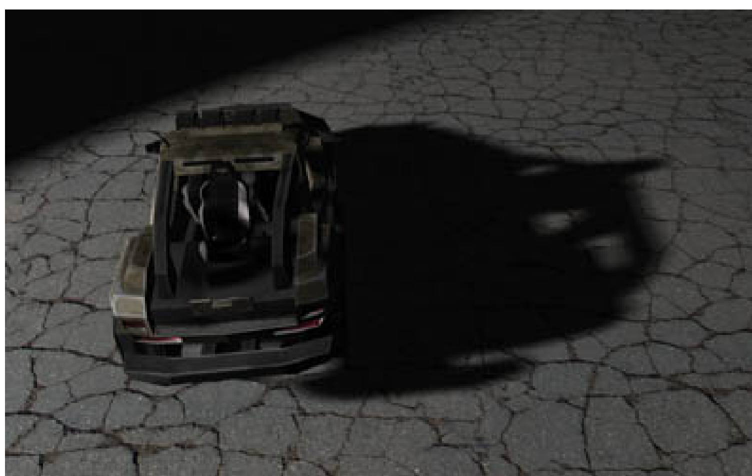


Figura 3.7: Percentage-closer filtering

Quella in figura è a tutti gli effetti una soft shadow e le aree di penombra sono ben visibili. È inoltre importante notare come, sfruttando la tecnica PCF, l'ampiezza della penombra dipenda solamente dall'ampiezza (fissa) della regione filtrata della shadow map.

3.3.2 Percentage-Closer Soft Shadows (PCSS)

Visto che il filtraggio di tipo PCF avviene all'interno del fragment shader, non ci sono delle condizioni che ci impediscano di variare dinamicamente l'ampiezza dell'area di shadow map esaminata per ogni frammento. L'area del filtro PCF influenza l'ampiezza della penombra generata, e una sua variazione dinamica porta a costruire ombre con penombra di ampiezza variabile.

Il metodo *percentage-closer soft shadows* [50] sfrutta questo principio estendendo l'approccio PCF di filtraggio ad ampiezza fissa. Si introduce il nuovo parametro W_L corrispondente all'ampiezza della sorgente luminosa (più la sorgente è ampia e maggiore sarà la penombra generata) e si opera secondo i seguenti passi durante il rendering del singolo frammento (ignorando i problemi di self-shadowing):

Algoritmo 3.3 Percentage-closer soft shadow mapping (fragment processing)

---

1. Blocker search
   Esaminando la shadow map nei dintorni del texel corrispondente al frammento, si mediano i valori di profondità D_s che soddisfano la relazione $D_s < D_f$ (più vicini del frammento alla sorgente luminosa), individuando infine il valore D_b corrispondente alla profondità media dell'oggetto 'blocker' che blocca parte della luce diretta verso il frammento. L'ampiezza dell'area esplorata dipende dal parametro W_L e dalla distanza del frammento dalla sorgente luminosa.

2. Stima dell'ampiezza della penombra
   Si calcola la seguente grandezza:

   $$W_P = \frac{(D_f - D_b)}{D_b} \cdot W_L$$

   Che rappresenta l'ampiezza della penombra sull'oggetto ricevitore.

3. Filtraggio PCF
   Si esegue un filtraggio ordinario secondo la tecnica PCF sfruttando un'ampiezza del filtro proporzionale al valore di W_P.

Quando si lavora con l'algoritmo PCSS, si assume di lavorare con una metrica lineare per D_f, D_s e D_b, come un valore proporzionale alla distanza dalla sorgente

luminosa. La profondità che si usa nel caso tipico è invece quella post-proiezione, scalata e traslata sull'intervallo $[0, 1]$, la quale è non lineare.

Naturalmente, questa tecnica ha un costo tipicamente superiore al PCF ordinario. Inoltre, visto che la regione di filtraggio PCF viene decisa dinamicamente, è impossibile configurare un bias da applicare al texel che vada bene per ogni ampiezza del filtro PCF e il problema del self-shadowing si aggrava ulteriormente.

In fig. 3.8 è riportato un esempio di scena renderizzata con tecnica PCSS.

Figura 3.8: Percentage-closer soft shadows

Come si vede, l'ampiezza della penombra generata cresce mano a mano che la distanza tra blocker e receiver sale. Questo effetto è presente anche nella realtà fisica e in effetti contribuisce a rendere le soft shadows generate ancora più convincenti.

Sfortunatamente, l'algoritmo PCSS si basa su una procedura costosa che richiede la lettura di diversi campioni dalla shadow map all'interno dello stadio di fragment processing: questa procedura si porta dietro i problemi di efficienza del filtraggio PCF ordinario.

Le tecniche che vedremo di seguito si basano su un filtraggio che non avviene a tempo di fragment shading: si utilizza una particolare struttura della shadow map che consente il prefiltraggio della stessa.

Prefiltrando la shadow map non è possibile ottenere penombra ad ampiezza variabile come in fig. 3.8 (perché la larghezza di filtraggio dovrebbe dipendere dal risultato della ricerca del blocker, che a sua volta dipende dalla posizione dei singoli frammenti), tuttavia bisogna considerare che non è strettamente necessario ottenere questo effetto per disegnare soft shadows convincenti (vedi fig. 3.7).

Vi sono stati anche degli sforzi per stimare la profondità media del blocker basandosi solamente su prefiltraggio (come in [52]), ma si richiede una complicazione sostanziale dell'algoritmo (che impatta sul frame rate) ed è necessario gestire una serie di problemi che si possono verificare. Queste tecniche non sono state applicate nella VR3Lib in quanto la loro estensione agli ultimi algoritmi disponibili basati su prefiltraggio non è ancora stata proposta. L'algoritmo implementato nella VR3Lib non genera penombre di ampiezza variabile.

3.3.3 Variance Shadow Maps (VSM)

La tecnica di *variance shadow mapping* [53, 51] disegna la scena dal punto di vista della sorgente luminosa come nel caso dello shadow mapping ordinario, ma si salvano due diversi valori all'interno dell'immagine (che deve dunque avere a disposizione due canali predisposti per lo scopo): la profondità D_s e la profondità elevata al quadrato D_s^2.

Supponiamo adesso di filtrare la shadow map ottenuta prima di renderizzare la scena: il filtraggio comporta una combinazione dei valori nella shadow map in una certa area definita dal kernel del filtro secondo opportuni coefficienti moltiplicativi. I valori che si trovano nella shadow map filtrata per ogni texel saranno calcolati come segue (e prendono il nome di *momenti*):

$$M_1 = \sum \left[D_s \cdot p\left(D_s\right) \right] \qquad M_2 = \sum \left[D_s^2 \cdot p\left(D_s\right) \right]$$

Dove $p\left(D_s\right)$ rappresenta i valori della funzione usata per la convoluzione (i coefficienti moltiplicativi). Queste espressioni ricordano molto da vicino quelle note per il calcolo del valor medio μ e della varianza σ^2 di una variabile aleatoria X con una certa distribuzione. I momenti sono infatti approssimazioni delle seguenti quantità (imponendo $D_s = X$, variabile aleatoria):

$$M_1 \approx E\left(X\right) = \int_{-\infty}^{\infty} x \cdot p(x)\, dx \qquad M_2 \approx E\left(X^2\right) = \int_{-\infty}^{\infty} x^2 \cdot p(x)\, dx$$

da cui:

$$\mu = E\left(X\right) \approx M_1 \qquad \sigma^2 = E\left(X^2\right) - E\left(X\right)^2 \approx M_2 - M_1^2$$

Quindi, dopo aver filtrato la shadow map (ottenendo M_1 e M_2), è possibile calcolare una stima del valor medio e della varianza di una distribuzione probabilistica di valori di D_s per ogni texel nella shadow map. La forma della densità di probabilità

dipende dal modo in cui si è fatto il filtraggio, ma in ogni caso abbiamo che una certa quantità di valori di profondità nell'intorno di ogni texel della shadow map originale viene adesso rappresentata da una distribuzione probabilistica di cui abbiamo valor medio e varianza per ogni texel.

Dato un valore di profondità per il frammento D_f noto a tempo di rendering, possiamo stimare la quantità di luce che arriva sul frammento calcolando la $P(D_f \leq D_s)$, dove D_s è una variabile aleatoria di cui abbiamo media e varianza (note prelevando i valori di un singolo texel dalla shadow map). Questo equivale ad effettuare un filtraggio PCF sulla shadow map visto che la quantità ricavata rappresenta la frazione di texel nell'area esaminata che risultano più lontani dalla luce del frammento a profondità D_f (questo è evidente almeno nel caso di PCF e prefiltraggio della mappa dato dalla media dei valori nel kernel).

Il metodo delle VSM prevede a questo punto di usare la nota *disuguaglianza di Chebyshev* per dare un'approssimazione (in realtà un limite superiore) del valore di $P(D_f \leq D_s)$ come segue:

$$P(D_f \leq D_s) \leq \frac{\sigma^2}{\sigma^2 + (D_f - \mu)^2} = P_{max}(D_f) \tag{3.1}$$

Il valore $P_{max}(D_f)$ così calcolato verrà utilizzato direttamente per il rendering come fattore moltiplicativo (compreso tra 0 e 1) e rappresenta la quantità di luce ricevuta dal frammento. Notare che la disuguaglianza di Chebyshev è valida solamente se abbiamo che $D_f > \mu \approx M_1$, in tutti gli altri casi si considera il frammento completamente illuminato.

Riassumendo, la procedura è la seguente:

Algoritmo 3.4 Variance shadow mapping

1. PRIMO PASSO DI RENDERING
   Si renderizza off-screen la scena dal punto di vista della sorgente luminosa, salvando i valori di profondità D_s e D_s^2 in una variance shadow map.

2. FILTRAGGIO OFFLINE
   Prima di disegnare la scena dal punto di vista della telecamera, si esegue un filtraggio della shadow map con un filtro di blur o simili. L'ampiezza del kernel utilizzato è direttamente proporzionale alla quantità di penombra che si vuole ottenere (e influenza le prestazioni dell'algoritmo).

3. SECONDO PASSO DI RENDERING
   Si disegna la scena dal punto di vista dell'osservatore, calcolando per ogni frammento renderizzato un fattore moltiplicativo P che indica la quantità di luce ricevuta dal frammento. Si opera fatto prelevando una singola volta i valori M_1 e M_2 dalla shadow map e calcolando le seguenti approssimazioni.

   $$\mu \approx M_1 \qquad \sigma^2 \approx M_2 - M_1^2$$

   $$\begin{cases} D_f \leq M_1 & \Rightarrow \qquad P = 1 \\ D_f > M_1 & \Rightarrow \quad P = P_{max}(D_f) = \dfrac{\sigma^2}{\sigma^2 + (D_f - \mu)^2} \end{cases}$$

   Basterà infine moltiplicare il fattore P per il colore del frammento, considerato completamente illuminato (eventualmente modulando solamente le componenti non ambientali).

In questo caso, dato che la mappa deve subire un filtraggio, possiamo anche usare le funzionalità di filtraggio fornite (ad oggi) direttamente dall'hardware: si parla di filtraggio bilineare, anisotropico e mipmapping con interpolazione lineare (filtraggio trilineare). Tutti questi metodi di filtraggio delle texture dovrebbero venire utilizzati se supportati dall'hardware, in tal caso infatti si aumenta la qualità della penombra sfruttando delle funzionalità altamente efficienti.

Grazie al fatto che la disuguaglianza di Chebyshev fornisce una sovrastima della probabilità cercata, con le VSM si risolve facilmente ogni problema di self-shadowing nelle regioni che dovrebbero essere completamente illuminate. Indipendentemente dall'ampiezza del kernel per il filtraggio eseguito, per eliminare l'artefatto di shadow acne è sufficiente imporre un valore minimo molto piccolo alla varianza σ^2 calcolata come sopra (operazione comunque necessaria per problemi di stabilità numerica).

Per ottenere un'implementazione robusta dell'algoritmo VSM, bisogna utilizzare due canali per i due momenti contenenti numeri reali a precisione singola. Inoltre

è fortemente consigliato l'utilizzo di una metrica lineare per la profondità (come un valore proporzionale alla distanza dalla sorgente luminosa). In questo modo si evitano problemi di instabilità numerica.

Una scena renderizzata sfruttando la tecnica VSM è quella in fig. 3.9.

Figura 3.9: Variance shadow mapping

I risultati sono eccellenti, tuttavia le variance shadow map hanno alcuni problemi di cui è necessario discutere.

Light bleeding

Quando un caster proietta la propria ombra su un ricevitore, tale ombra risulterà avere spigoli 'soft', contenenti delle zone di penombra. Supponiamo però che dietro il ricevitore vi sia un'altro ricevitore: questo dovrebbe ricevere solo l'ombra proiettata dal primo ricevitore; eppure in determinate circostanze abbiamo che lo spigolo soft dell'ombra proiettata dal caster risulterà visibile anche sul secondo ricevitore (vedi fig. 3.10).

Questo effetto è dovuto al fatto che la disuguaglianza di Chebyshev rappresenta solo un limite superiore alla probabilità cercata. Proviamo infatti ad applicare la disuguaglianza nel punto indicato in fig. 3.10 per il secondo ricevitore (oggetto C). Supponendo che i tre oggetti A, B e C siano a distanza rispettivamente a, b e c dalla sorgente luminosa, e che i momenti nel punto in esame siano i seguenti dopo il filtraggio:

$$M_1 = \frac{a+b}{2} \qquad M_2 = \frac{a^2+b^2}{2}$$

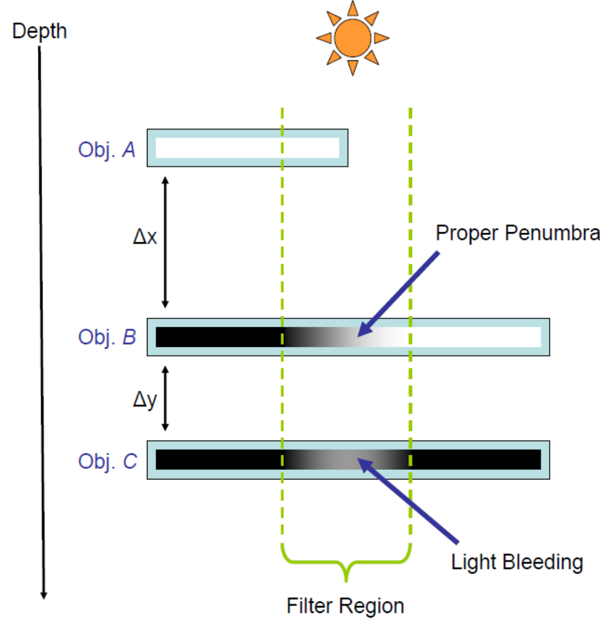


Figura 3.10: Light bleeding

otteniamo:

$$\mu \approx \frac{a+b}{2} \qquad \sigma^2 \approx \frac{a^2+b^2}{2} - \frac{a^2+b^2+2ab}{4} = \frac{(b-a)^2}{4}$$

per cui:

$$P_{max} = \frac{\sigma^2}{\sigma^2 + (c-\mu)^2} \approx \frac{\frac{(b-a)^2}{4}}{\frac{(b-a)^2}{4} + \left(c - \frac{a+b}{2}\right)^2} = \frac{\frac{\triangle x^2}{4}}{\frac{\triangle x^2}{4} + \left(\triangle y + \frac{\triangle x}{2}\right)^2}$$

Considerando allora un particolare $\triangle x$, il valore di P_{max} (che dovrebbe essere sempre 0 per l'oggetto C nel caso ideale) decresce come $1/\triangle y^2$. Dunque si potranno notare degli spigoli luminosi in zone che dovrebbero essere completamente in ombra, e il fenomeno sarà tanto più grave quanto più il rapporto $\triangle x/\triangle y$ è elevato.

Varie sono state le proposte per ridurre questo fastidioso fenomeno, dalla semplice rimozione delle penombre che risultano luminose sotto una certa intensità [51] (infatti il light bleeding è sempre tale che $P_{max} < 1$) all'utilizzo di approcci stratificati che generano diverse shadow map corrispondenti ad una suddivisione della scena in diversi livelli di profondità [54]. Una soluzione completa ed efficiente al problema non è ancora nota, tuttavia si ha un forte miglioramento sfruttando la

tecnica di EVSM, che vediamo più avanti.

L'artefatto di light bleeding è messo in evidenza in fig. 3.11.

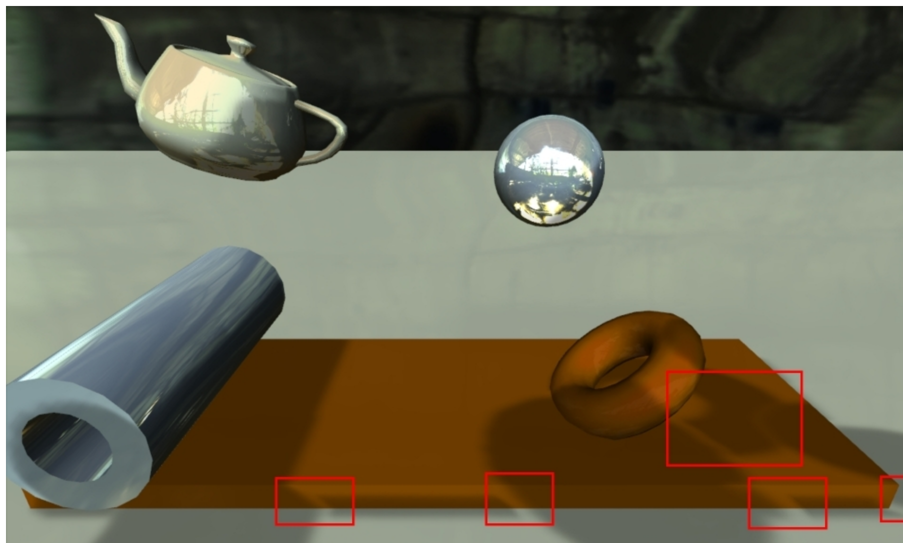


Figura 3.11: Light bleeding nella tecnica VSM

Artefatti per ricevitori non planari

La disuguaglianza di Chebyshev ci fornisce il valore esatto per $P(D_f \leq D_s)$ quando abbiamo a che fare con ricevitori planari; l'approssimazione è normalmente buona anche nel caso di ricevitori non planari ed utilizzabile direttamente. Tuttavia, quando la geometria che riceve l'ombra presenta delle discontinuità piuttosto pronunciate, possono verificarsi artefatti in prossimità dei bordi degli oggetti ricevitori dovuti nuovamente al modo in cui si calcola il fattore moltiplicativo. Filtrando in prossimità del bordo, quando abbiamo un forte dislivello nella shadow map, si può spostare il valor medio della profondità D_s ad una distanza superiore a quella del bordo e dunque questo può causare una completa illuminazione dei frammenti vicino alla discontinuità. L'artefatto è tanto peggiore quanto più è esteso il kernel del filtro utilizzato.

L'effetto appena descritto è messo in evidenza in fig. 3.12.

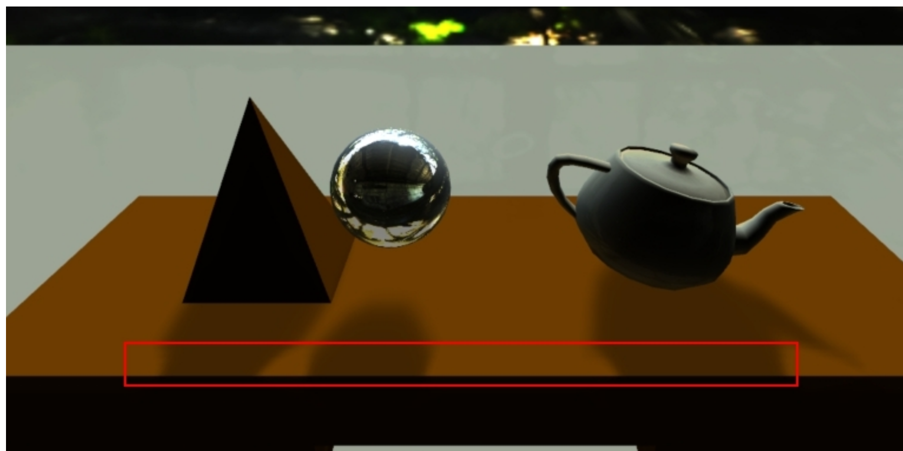


Figura 3.12: Artefatti di bordo con tecnica VSM

Questo problema può venire affrontato in diversi modi. La tecnica EVSM riduce fortemente anche questo artefatto, come vedremo più avanti.

3.3.4 Exponential Shadow Maps (ESM)

Questo algoritmo ha un approccio simile a quello delle VSM ed è stato introdotto da due diversi lavori svolti in contemporanea e sostanzialmente indipendenti [55, 56].

Nel caso delle ESM partiamo supponendo di inserire all'interno della shadow map solamente i valori di profondità D_s come nel caso standard e di filtrare la mappa combinandone le profondità in una certa area in dipendenza del filtro utilizzato. I valori che si trovano nella shadow map filtrata per ogni texel saranno calcolati come segue (momenti del primo ordine):

$$M = \sum \left[D_s \cdot p\left(D_s\right) \right]$$

dove $p\left(D_s\right)$ rappresenta i valori della funzione usata per la convoluzione (i coefficienti moltiplicativi). Come già osservato, questa espressione è simile a quella utilizzata per il calcolo del valor medio nella teoria della probabilità; infatti abbiamo che (imponendo $D_s = X$, variabile aleatoria):

$$\mu = E\left(X\right) = \int_{-\infty}^{\infty} x \cdot p(x)\, dx \approx M$$

Considerando allora i valori di D_s salvati inizialmente nella shadow map come i valori assunti da una variabile aleatoria, quello che si trova all'interno della shadow map una volta effettuato il filtraggio sono i valori M che approssimano il valor medio

di una distribuzione probabilistica di D_s per ogni texel. Per ogni texel abbiamo dunque una distribuzione probabilistica di valori determinata dal tipo di filtraggio effettuato, e il momento che si trova nella shadow map è una approssimazione del valor medio di tale distribuzione.

Dato un valore di profondità per il frammento D_f noto a tempo di rendering, possiamo stimare la quantità di luce che arriva sul frammento calcolando la $P(D_f \leq D_s)$, dove D_s è una variabile aleatoria di cui abbiamo stimato il valor medio (noto prelevando un singolo elemento dalla shadow map).

Per calcolare tale probabilità possiamo utilizzare la *disuguaglianza di Markov* (nota in teoria della probabilità) che sostanzialmente nel nostro caso si può scrivere semplicemente come segue (visto che i valori per D_s e D_f sono sempre positivi):

$$P(D_f \leq D_s) \leq \frac{E(D_s)}{D_f} \tag{3.2}$$

Questo ci da un limite superiore alla probabilità cercata. Purtroppo normalmente tale valore è piuttosto lontano dalla quantità desiderata e le ombre risultano troppo luminose. Infatti, anche quando abbiamo che $D_f > E(D_s)$, la discesa verso zero è troppo lenta.

Per migliorare il risultato possiamo applicare una funzione non lineare, monotona crescente e positiva $f(x)$ alla variabile aleatoria D_s, ottenendo una nuova variabile aleatoria $Y = f(D_s)$ sempre positiva. La disuguaglianza di Markov si applica naturalmente anche alla variabile aleatoria Y ed è infatti lecito scrivere:

$$P(f(D_f) \leq Y) = P(f(D_f) \leq f(D_s)) \leq \frac{E(f(D_s))}{f(D_f)}$$

ma siccome la funzione $f(x)$ è monotona crescente, vale anche che:

$$f(x) \leq f(y) \Longleftrightarrow x \leq y$$

e dunque:

$$P(D_f \leq D_s) = P(f(D_f) \leq f(D_s)) \leq \frac{E(f(D_s))}{f(D_f)}$$

La funzione $f(x)$ di warping (distorsione) deve essere tale da accelerare la discesa verso 0 quando $D_f > E(D_s)$, diverse funzioni soddisfano questo requisito.

Bisogna inoltre notare che, se si calcola l'intensità luminosa con la disuguaglianza di Markov come nella formula sopra, le ombre risulteranno più o meno intense a

seconda della profondità del frammento D_f e della profondità media memorizzata nel texel $E(D_s)$. Questo significa che l'intensità dell'ombra dipende anche dalla posizione della luce nella scena. Per rimuovere questa dipendenza dalla posizione della luce, bisogna che il risultato sia identico quando a D_s e D_f si aggiunge o sottrae uno stesso valore t. Questo porta ovviamente alla soluzione:

$$f(x) = Ce^{kx}$$

Riassumendo, l'algoritmo di ESM risulta allora il seguente:

Algoritmo 3.5 Exponential shadow mapping

1. PRIMO PASSO DI RENDERING
   Si renderizza off-screen la scena dal punto di vista della sorgente luminosa, salvando i valori di profondità trasformati e^{kD_s} nella shadow map.

2. FILTRAGGIO OFFLINE
   Prima di disegnare la scena dal punto di vista della telecamera, si esegue un filtraggio della shadow map con un filtro di blur o simili. L'ampiezza del kernel utilizzato è direttamente proporzionale alla quantità di penombra che si vuole ottenere (e influenza le prestazioni dell'algoritmo).

3. SECONDO PASSO DI RENDERING
   Si disegna la scena dal punto di vista dell'osservatore, calcolando per ogni frammento renderizzato un fattore moltiplicativo P che indica la quantità di luce ricevuta dal frammento. Si opera prelevando una singola volta il valore filtrato dalla shadow map $M = \sum \left[e^{kD_s} \cdot p(D_s)\right] \approx E\left(e^{kD_s}\right)$.

   $P = \min\left(1, \frac{M}{e^{kD_f}}\right)$

   Basterà infine moltiplicare il fattore P per il colore del frammento, considerato completamente illuminato (eventualmente modulando solamente le componenti non ambientali).

Naturalmente, anche nel caso delle ESM è possibile sfruttare tutti i metodi di filtraggio hardware validi per le VSM. Inoltre le ESM hanno il vantaggio di sfruttare un singolo termine da calcolare e memorizzare nella shadow map e risultano dunque leggermente più efficienti delle VSM (tuttavia il costo computazionale di queste tecniche è dovuto principalmente al filtraggio offline).

Un esempio di scena renderizzata con la tecnica di exponential shadow mapping è riportato in fig. 3.13.

Figura 3.13: Exponential shadow mapping

Bisogna comunque evidenziare che questo metodo ha alcuni difetti in parte simili a quelli visti per il variance shadow mapping, che descriviamo di seguito.

LIGHT BLEEDING

Il tipo di bleeding esaminato nel caso di VSM è del tutto sparito, tuttavia abbiamo introdotto un nuovo tipo di light bleeding che dipende dal rapporto dettato dalla disuguaglianza di Markov. Usando la funzione di warp esponenziale il light bleeding viene a dipendere solamente dalla distanza tra caster e ricevitore.

Questo effetto dipende dal fatto che per quanto la disuguaglianza possa tendere velocemente verso lo zero, non arriverà mai a zero e il colore risultante dipenderà dal valore relativo di $E\left(e^{kD_s}\right)$ e e^{kD_f}. In particolare, questo tipo di light bleeding scompare per k tendente all'infinito e dunque è conveniente impostare questo parametro con un valore molto elevato.

A questo proposito, ricordiamo che i valori di e^{kD_s} dovranno venire memorizzati in una texture e dunque si raccomanda di utilizzare un canale con numeri reali in virgola mobile a precisione singola (almeno) per ottenere il più elevato range possibile (e sfruttare di conseguenza valori di k molto elevati, intorno a 80 se il D_s massimo è 1).

In fig. 3.14 è messo in evidenza il tipo di light bleeding che si genera usando la tecnica ESM.

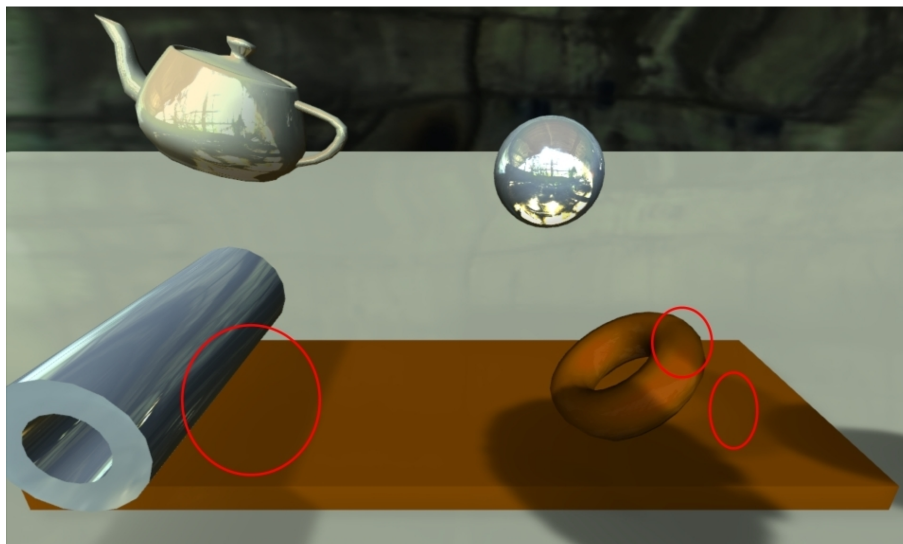


Figura 3.14: Light bleeding nella tecnica ESM

Artefatti per ricevitori non planari

Il problema che abbiamo esaminato nelle VSM per ricevitori non planari si presenta nuovamente nel caso delle ESM: la causa è identica e l'artefatto che si genera ha la stessa forma. Vedere il paragrafo 3.3.3 per maggiori informazioni.

Come nel caso delle VSM, l'effetto è tanto peggiore quanto più è esteso il kernel del filtro utilizzato. Inoltre, il problema si aggrava utilizzando le ESM perché grazie al warp esponenziale i dislivelli presenti nella shadow map standard vengono esasperati per ridurre il light bleeding (aumentando il parametro k) e il filtraggio può portare ad artefatti molto evidenti.

3.3.5 Exponential Variance Shadow Maps (EVSM)

Questa è l'ultima tecnica che presentiamo ed è quella implementata nella libreria costruita col presente lavoro di tesi. Non è ancora stata prodotta molta documentazione al riguardo e la maggior parte delle informazioni sono reperibili solamente in forum online specializzati come Beyond3D e GameDev.net.

La tecnica è stata proposta inizialmente da Lauritzen e McCool nell'articolo che presenta le *layered variance shadow maps* [54] (una tecnica piuttosto costosa per combattere il light bleeding in ambito VSM). In tale documento il metodo era soltanto accennato, tuttavia nel forum Beyond3D vi sono alcune discussioni riguardanti le EVSM ed è anche disponibile un esempio di implementazione proposto da Lauritzen stesso.

Il grande vantaggio di questo algoritmo è che ad un costo solo lievemente superiore è possibile combattere simultaneamente entrambi i difetti della tecnica di VSM. Si combinano le tecniche di VSM e ESM per produrre soft shadows ancora più robuste mitigando in modo sostanziale i difetti dei due metodi basati su prefiltraggio visti fino ad ora.

Riduzione del light bleeding

Osservando la figura 3.10, ricordiamo che il fenomeno del light bleeding è tanto più grave quanto più il rapporto $\triangle x/\triangle y$ è elevato. Con la tecnica di EVSM, l'exponential warp proprio del metodo ESM viene riutilizzato allo scopo di ridurre questo rapporto. Se consideriamo gli oggetti A, B e C in figura a distanza rispettivamente a, b e c dalla sorgente luminosa, vale che:

$$a < b < c$$

e il rapporto di interesse è dato da:

$$(b - a) \,/\, (c - b) = \triangle x/\triangle y$$

Assumendo di lavorare con distanze che sono state trasformate col warp esponenziale, il rapporto di interesse diventa:

$$\left(e^{kb} - e^{ka}\right) / \left(e^{kc} - e^{kb}\right)$$

e sostituendo $b = a + \triangle x$ e $c = a + \triangle x + \triangle y$ questo rapporto si può scrivere come:

$$\frac{\left(e^{k\triangle x} - 1\right)}{\left(e^{k\triangle y} - 1\right) \cdot e^{k\triangle x}}$$

Si osserva dunque che (come volevasi dimostrare):

$$\frac{\left(e^{k\triangle x} - 1\right)}{\left(e^{k\triangle y} - 1\right) \cdot e^{k\triangle x}} < \frac{\triangle x}{\triangle y} \quad \forall \triangle x, \triangle y \geq 0 \, e \, k > 0$$

Fare il warp esponenziale corrisponde a trasformare la nostra variabile aleatoria D_s in una nuova variabile aleatoria data da e^{kD_s}. Visto che la funzione esponenziale è monotona crescente, continua a valere la disuguaglianza di Chebyshev introdotta

con le VSM nel modo seguente:

$$P\left(D_f \leq D_s\right) = P\left(e^{kD_f} \leq e^{kD_s}\right) \leq \frac{\sigma^2}{\sigma^2 + \left(e^{kD_f} - \mu\right)^2} = P_{max}\left(D_f\right)$$

Dove μ e σ^2 sono approssimazioni che si ricavano grazie al filtraggio, come segue:

$$\mu = E\left(x\right) \approx M_1 = \sum \left[e^{kD_s} \cdot p\left(D_s\right)\right]$$
$$\sigma^2 = E\left(x^2\right) - E\left(x\right)^2 \approx M_2 - M_1^2 = \sum \left[e^{2kD_s} \cdot p\left(D_s\right)\right] - \left(\sum \left[e^{kD_s} \cdot p\left(D_s\right)\right]\right)^2$$

Dunque è sufficiente memorizzare nella shadow map i due valori e^{kD_s} e e^{2kD_s} ed eseguire il calcolo di $P_{max}\left(D_f\right)$ come nelle VSM ma sfruttando il warp esponenziale per ottenere una riduzione sostanziale del light bleeding (nella forma vista con le VSM). Inoltre aumentando k si ottiene una riduzione ancora più forte degli artefatti di light bleeding, come per le ESM dunque è necessario impostare per k un valore il più elevato possibile (intorno a 40 assumendo un massimo D_s pari ad 1: stavolta infatti bisogna memorizzare anche e^{2kD_s} all'interno della texture).

Il light bleeding non viene eliminato, ma solo fortemente ridotto. Gli effetti che si possono osservare sono quelli illustrati in fig. 3.15, corrispondente alla stessa situazione vista in fig. 3.11 (notare il miglioramento rispetto alle VSM ordinarie: il light bleeding è appena visibile).

Figura 3.15: Light bleeding nella tecnica EVSM

Riduzione degli artefatti per ricevitori non planari

Come già osservato in precedenza, entrambe le tecniche di VSM e ESM sono affette

da artefatti in prossimità di discontinuità geometriche dei ricevitori. L'effetto è tanto più visibile quanto più elevato è il dislivello nella continuità e il raggio del filtro di convoluzione applicato, peggiora inoltre al salire di k nelle ESM.

Nel caso delle EVSM è possibile affrontare il problema aggiungendo due ulteriori canali nella shadow map (da 2 canali floating point a precisione singola si passa a 4) che vengono utilizzati per il warp esponenziale negativo.

Si usano allora due diversi warp esponenziali della profondità D_s in modo simultaneo. Il warp positivo in due dei quattro canali (e^{kD_s}, e^{2kD_s}) riduce il light bleeding avvicinando di fatto l'oggetto B in fig. 3.10 all'oggetto A (relativamente). Il warp negativo nei restanti canali ($-e^{-kD_s}$, e^{-2kD_s}) ha invece l'effetto opposto: avvicina l'oggetto B all'oggetto C (sempre in modo relativo), combattendo dunque gli effetti per ricevitori non planari avvicinando tra loro le geometrie degli stessi, infatti:

$$\frac{\left(e^{-k\triangle x}-1\right)}{\left(e^{-k\triangle y}-1\right)\cdot e^{-k\triangle x}} > \frac{\triangle x}{\triangle y} \quad \forall \triangle x, \triangle y \geq 0 \, e \, k > 0$$

Notare che anche la funzione $-e^{-kD_s}$ è una funzione monotona crescente e dunque la disuguaglianza di Chebyshev applicata a questo secondo warp continua ad essere valida per calcolare $P\left(D_f \leq D_s\right)$.

I due warp vengono utilizzati congiuntamente: si calcola il valore di due diverse stime per $P\left(D_f \leq D_s\right)$ date dai due diversi limiti superiori di Chebyshev e si sceglie il minimo tra i due in ogni frammento. Ognuna delle due stime infatti fallisce in casi particolari dando una pessima approssimazione per eccesso del valore di $P\left(D_f \leq D_s\right)$ cercato; scegliendo il minimo si ottiene sempre il migliore tra i due valori (nascondendo alcune delle situazioni in cui una delle due stime fallirebbe, se presa singolarmente). Si potrebbe pensare di usare solamente il warp negativo, ma in questo modo il light bleeding risulterebbe peggiorato visto che si sta di fatto avvicinando l'oggetto B all'oggetto C in fig. 3.10 incrementando il valore del rapporto che determina la gravità dell'artefatto.

Anche in questo caso, gli artefatti dovuti alle discontinuità geometriche non vengono eliminati, ma solo fortemente ridotti. La situazione riportata in fig. 3.16 corrisponde a quella osservata in fig. 3.12 nel caso delle VSM. In questa scena l'artefatto è ancora presente, ma non lo si riesce più a percepire con facilità.

Riassumendo, l'algoritmo EVSM opera come segue (usando al solito canali reali in virgola mobile a precisione singola e una metrica lineare per D_s):

Algoritmo 3.6 Exponential variance shadow mapping

1. PRIMO PASSO DI RENDERING
   Si renderizza off-screen la scena dal punto di vista della sorgente luminosa, salvando i valori di profondità trasformati $\begin{pmatrix} e^{kD_s} & e^{2kD_s} & -e^{-kD_s} & e^{-2kD_s} \end{pmatrix} = \begin{pmatrix} M_1 & M_2 & M_3 & M_4 \end{pmatrix}$ in una exponential variance shadow map a 4 canali.

2. FILTRAGGIO OFFLINE
   Prima di disegnare la scena dal punto di vista della telecamera, si esegue un filtraggio della shadow map con un filtro di blur o simili. L'ampiezza del kernel utilizzato è direttamente proporzionale alla quantità di penombra che si vuole ottenere (e influenza le prestazioni dell'algoritmo). I valori prodotti saranno i veri e propri momenti.

3. SECONDO PASSO DI RENDERING
   Si disegna la scena dal punto di vista dell'osservatore, calcolando per ogni frammento renderizzato un fattore moltiplicativo P che indica la quantità di luce ricevuta dal frammento. Si opera prelevando una singola volta i valori memorizzati nella shadow map filtrata e calcolando le seguenti approssimazioni:

   $$\mu_1 \approx M_1 \qquad \sigma_1^2 \approx M_2 - M_1^2$$
   $$\mu_2 \approx M_3 \qquad \sigma_2^2 \approx M_4 - M_3^2$$
   $$P_1 = \begin{cases} 1 & if \quad e^{kD_s} \leq M_1 \\ \dfrac{\sigma_1^2}{\sigma_1^2 + \left(e^{kD_s} - \mu_1\right)^2} & if \quad e^{kD_s} > M_1 \end{cases}$$
   $$P_2 = \begin{cases} 1 & if \quad -e^{-kD_s} \leq M_3 \\ \dfrac{\sigma_2^2}{\sigma_2^2 + \left(-e^{-kD_s} - \mu_2\right)^2} & if \quad -e^{-kD_s} > M_3 \end{cases}$$
   $$P = \min\left(P_1, P_2\right)$$

   Basterà infine moltiplicare il fattore P per il colore del frammento, considerato completamente illuminato (eventualmente modulando solamente le componenti non ambientali).

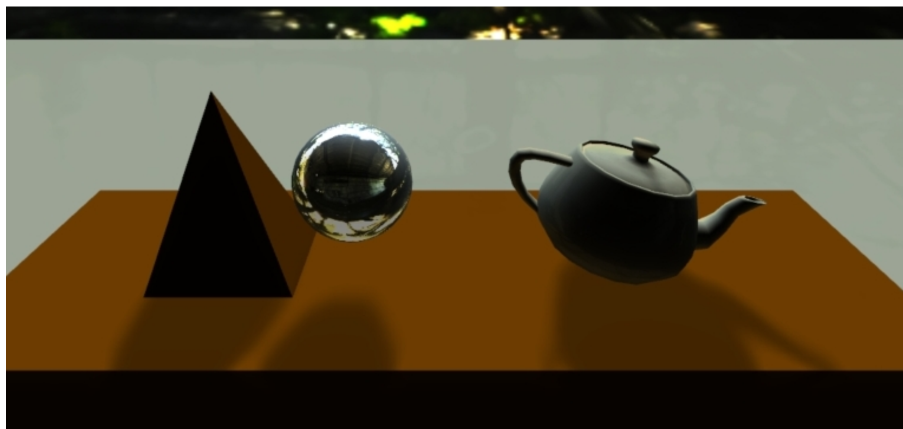


Figura 3.16: Artefatti di bordo con tecnica EVSM

In fig. 3.17 sono riportate delle scene renderizzate con tecnica EVSM: si nota che le soft shadows ottenute sono piuttosto convincenti e non vi sono artefatti visibili.

In termini di prestazioni (tempo di esecuzione), il warp esponenziale non risulta particolarmente costoso. Il numero di canali utilizzati all'interno della shadow map invece ha un certo peso sulle prestazioni dell'algoritmo e pertanto la tecnica ESM risulta la più efficiente tra quelle viste basate su prefiltraggio; il metodo EVSM è invece il più costoso. Il contributo maggiore però in termini di costo computazionale è dovuto al tempo necessario al filtraggio della shadow map prima di renderizzare la scena come vista dall'osservatore; infatti si richiedono svariati accessi alla texture da filtrare. Le differenze di prestazioni tra i 3 algoritmi di ESM, VSM ed EVSM sono quindi dovute alla differenza nel numero di canali, ma non sono comunque particolarmente marcate. L'efficienza di questi algoritmi dipende infatti principalmente da come viene effettuato il filtraggio prima del rendering finale ad ogni frame. Ne deriva che la scelta più logica è utilizzare il metodo EVSM per la nuova VR3Lib, visto che è meno affetto da artefatti e solo lievemente meno efficiente.

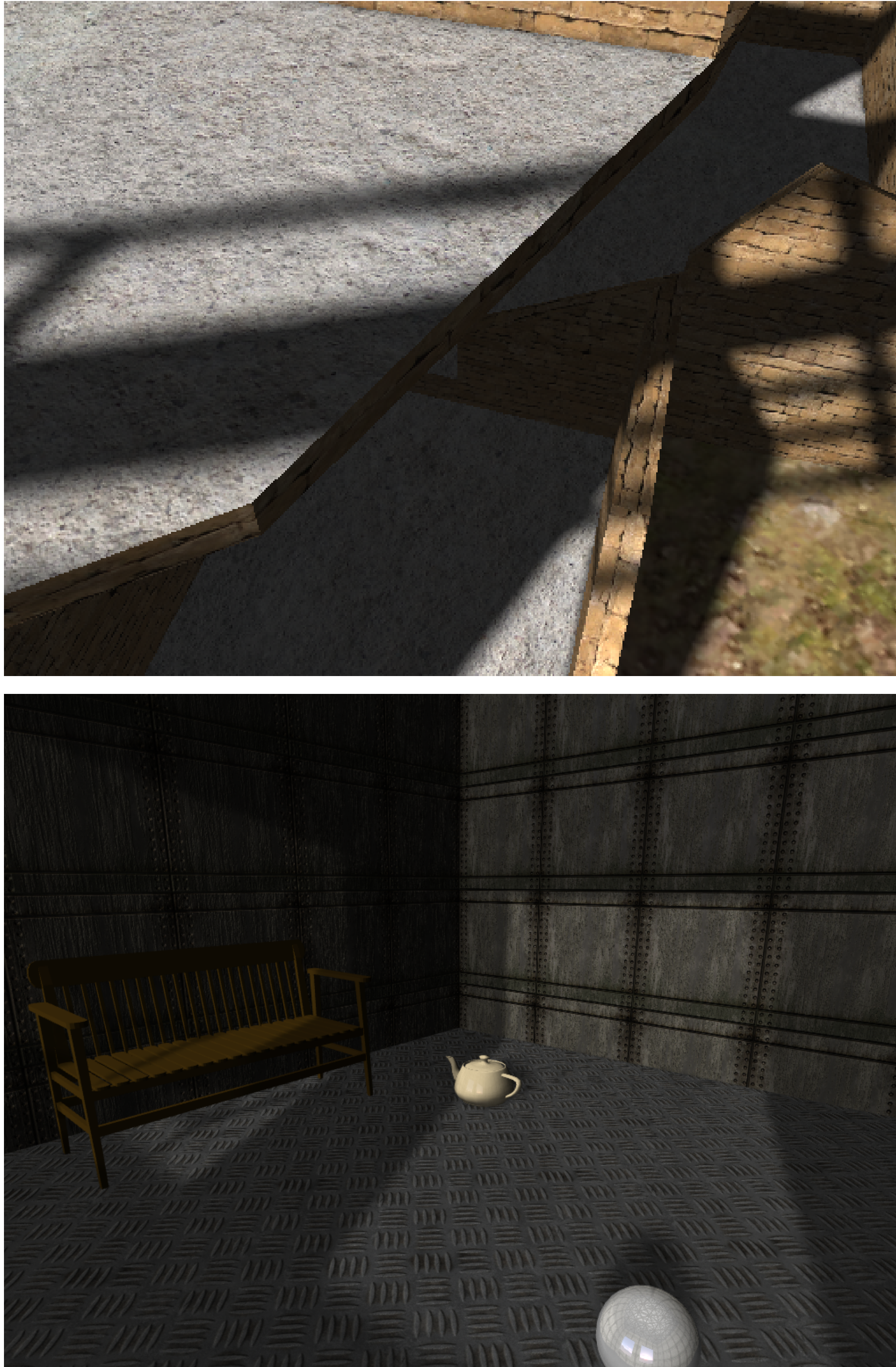


Figura 3.17: Exponential variance shadow mapping

Capitolo 4

Realizzazione delle Tecniche Presentate

In questo capitolo vediamo come sono state realizzate le tecniche di rendering presentate nei capitoli 2 e 3 all'interno della nuova VR3Lib. In particolare faremo riferimento a porzioni di codice GLSL e introdurremo alcuni concetti di OpenGL necessari per capire il funzionamento. Visto che con le ultime versioni del sistema grafico qualsiasi operazione di rendering viene effettuata usando shader scritti esplicitamente dal programmatore, gli algoritmi visti in precedenza vengono sostanzialmente implementati sotto forma di shader.

Dato che durante la visualizzazione di ambienti virtuali si possono usare una molteplicità di combinazioni delle tecniche viste in precedenza, all'interno della VR3Lib abbiamo una varietà di shader pronti all'uso, e spesso succede che il codice sorgente dello shader program utilizzato viene costruito a runtime sulla base delle features richieste dall'utente per evitare un'inutile replicazione di codice GLSL. Per facilitare la lettura, trascuriamo questo aspetto implementativo della libreria nel presente capitolo fornendo direttamente i frammenti di codice finali usati a tempo di rendering (che potrebbero venir costruiti dinamicamente).

Oltre a questo insieme di shader disponibili all'interno della VR3Lib e invisibili all'utente (built-in) è anche possibile effettuare il caricamento di shader forniti dall'utente della VR3Lib allo scopo di ottenere funzionalità ed effetti non supportati in modo nativo dalla libreria; vedremo meglio questo aspetto nel prossimo capitolo.

In tutte le sezioni che seguono facciamo riferimento come al solito alla versione 3.3 dell'API OpenGL e dunque lavoriamo con la versione 3.30 del linguaggio GLSL. I dati relativi ai vertici ricevuti dai vertex shader sono di diverso tipo, fondamentalmente possiamo considerare:

- Posizione in object space dei vertici,

- Normali ai vertici in object space,
- Coordinate di accesso alle varie texture per ogni vertice.

In dipendenza dal tipo di shading eseguito per un particolare oggetto, non sarà necessario fornire tutti i dati di cui sopra in ingresso al vertex shader e potremo usarne solo un sottoinsieme.

Ovviamente non è richiesto che il lettore abbia dimestichezza col linguaggio GLSL (una comprensione di base delle operazioni svolte è intuitiva e gli aspetti importanti verranno messi in evidenza nel testo), tuttavia alcuni semplici concetti di questo linguaggio che possono facilitare la lettura degli shader sono i seguenti:

- l'entry point per ogni tipo di shader executable è costituito dalla funzione `main()`, in analogia col linguaggio C;
- gli statement hanno la stessa struttura che hanno nel linguaggio C e i commenti si inseriscono allo stesso modo;
- i tipi `float`, `vec2`, `vec3`, `vec4` e `mat4` identificano rispettivamente un numero reale in virgola mobile a precisione singola, un vettore di 2 `float`, un vettore di 3 `float`, un vettore di 4 `float`, una matrice quadrata di `float` con 4 righe e 4 colonne;
- i qualificatori `in`, `out` e `uniform` identificano rispettivamente variabili di input, output e variabili uniform (accessibili in sola lettura).

4.1 Shading di Base

In sez. 2.1 abbiamo esaminato il problema dello shading di oggetti virtuali dotati solamente di un certo materiale. Abbiamo considerato il flat shading, alcuni metodi per ottenere smooth shading, e introdotto il concetto di smoothing groups. In questa sezione vediamo come realizzare queste tecniche al livello di GLSL.

Nella nuova libreria si utilizza l'interpolazione secondo Phong per ottenere uno shading smooth di triangoli appartenenti allo stesso smoothing group secondo il modello di riflessione di Phong. Non viene invece fornito uno shader program apposito per il flat shading (inteso come utilizzo della normale al poligono in ogni frammento in combinazione col Phong reflection model).

Al fine di incrementare le performance nel caso si richiedesse questo tipo di shading sarebbe conveniente fornire uno shader program apposito nel caso di flat

shading (infatti si potrebbe evitare di interpolare le normali passando al fragment shader). Bisogna tuttavia considerare che in ogni caso l'illuminazione viene calcolata per-pixel (all'interno del fragment shader) allo scopo di ottenere un risultato il più realistico possibile anche quando non vi sono superfici curve 'smooth', e dunque il vantaggio effettivo di prevedere uno shader program apposito per il flat shading è del tutto trascurabile.

Sia che si operi facendo flat shading con illuminazine per-pixel, sia che si operi tramite la Phong interpolation con eventualmente smoothing groups multipli, lo shader program resta quindi identico in quanto l'unica informazione a cambiare sono le normali per ogni vertice di ogni triangolo (che vengono precalcolate in object space e immagazzinate in un VBO come dati di ingresso al vertex shader). Nel caso di flat shading avremo che le normali ai vertici di un triangolo saranno tutte identiche e date dalla normale al triangolo; invece se abbiamo smooth shading con un solo smoothing group le normali ai vertici saranno tutte calcolate mediando le normali di diversi triangoli).

Nel nostro caso la media tra le normali dei triangoli che si toccano in un vertice (nel caso di smooth shading) viene calcolata come media classica (non pesata). Vi sono ovviamente altri approcci possibili.

Il vertex shader utilizzato risulta il seguente:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform mat4 transformMatrix; // proj*view*model matrix

in vec3 vertex; // vertex position in obj coordinates
in vec3 normal; // normal in obj coordinates
in vec2 diffCoord; // diffuse texture coordinates
in vec2 lightCoord; // light map coordinates
in vec2 normalCoord; // normal map coordinates

smooth out vec3 fragPositionObj;
smooth out vec3 fragNormalObj;
smooth out vec2 diffTexCoord;
smooth out vec2 lightNormalMapCoord;

void main(void) {
    gl_Position = transformMatrix*vec4(vertex,1.0); // clip coordinates
    fragPositionObj = vertex;
    ...
}
```

dove notiamo che:

- La matrice utilizzata per fissare la posizione dei vertici post-proiezione (`gl_Position`) è la matrice `transformMatrix` fornita come variabile uniform (vedi sez. 1.6); questa è data dal prodotto delle matrici di proiezione, viewing e modeling (in accordo a quanto descritto in sez. 1.4.2). Le altre trasformazioni (perspective division e viewport transformation) vengono eseguite in automatico a posteriori dal sistema grafico OpenGL.
- La parte omessa della funzione `main()` si occupa sostanzialmente di scrivere nelle variabili varying di uscita `fragNormalObj`, `diffTexCoord` e `lightNormalMapCoord` i valori opportuni, quando richiesti dal particolare tipo di shading che bisogna eseguire. Questa porzione di codice cambia a seconda del tipo di shading voluto per l'oggetto che sta venendo renderizzato. Nel caso di shading basato solo su materiali serviranno le sole normali e dunque bisognerà aggiungere:

```
...
void main(void) {
    ...
    fragNormalObj = normal;
}
```

È da notare il fatto che, tramite le variabili varying di uscita, alcune informazioni in object space vengono passate agli stadi programmabili successivi della pipeline (fragment shader ed eventualmente geometry shader). Lavorare in spazio oggetto conviene in alcune situazioni ma è sconveniente in altri casi: è molto positivo nel caso di normal mapping quando si prelevano dalla normal map le normali in spazio oggetto, ma quando servono le normali in world space (come vedremo nel seguito) queste andranno calcolate nel fragment shader. Comunque bisogna notare che una moltiplicazione matrice-vettore ha un costo molto basso anche nello stadio di fragment shading nelle moderne schede video e non si notano perdite significative di prestazioni. Naturalmente, lavorando in spazio oggetto, tutti i dati disponibili in world space nella applicazione andranno tradotti in object space prima di venire trasmessi agli shader come variabili uniform (un esempio può essere la posizione delle sorgenti luminose).

Lo stadio di geometry shading viene utilizzato solamente quando l'utente usufruisce del servizio di displacement mapping della libreria. Per lo shading di base che stiamo esaminando in questa sezione tale stadio verrà quindi bypassato.

Si opera invece tramite il seguente fragment shader:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform vec3 matEmission; // material emission color
uniform vec3 matAmbient; // material ambient color
uniform vec3 matDiffuse; // material diffuse color
uniform vec3 matSpecular; // material specular color
uniform float matOpacity; // material opacity
uniform float matShininess; // material shininess
uniform vec3 lightAmbient[MAX_NUM_LIGHTS]; // lights ambient colors
uniform vec3 lightDiffuse[MAX_NUM_LIGHTS]; // lights diffuse colors
uniform vec3 lightSpecular[MAX_NUM_LIGHTS]; // lights specular colors
uniform vec3 lightPosObj[MAX_NUM_LIGHTS]; // lights positions (obj coord)
uniform int numActiveLights; // number of active lights
uniform vec3 eyePosObj; // eye position in object coordinates

smooth in vec3 fragPositionObj; // fragment position (obj coord)
smooth in vec3 fragNormalObj; // fragment normal (obj coord)

out vec4 fragColor; // resulting fragment color (with alpha)

void main(void) {
    fragColor = vec4(1.0,1.0,1.0, clamp(matOpacity,0.0,1.0));
    vec3 fragColorRGB = matEmission;
    vec3 N = normalize(fragNormalObj);
    vec3 eyeObj = normalize(eyePosObj - fragPositionObj);
    vec3 lightObj;
    for(int i = 0; i < numActiveLights; i++) {
        lightObj = normalize(lightPosObj[i] - fragPositionObj);
        fragColorRGB += matAmbient*lightAmbient[i];
        fragColorRGB +=
            matDiffuse*lightDiffuse[i]*clamp(dot(N,lightObj),0.0,1.0);
        if((matSpecular != vec3(0.0,0.0,0.0)) && (matShininess != 0.0))
            fragColorRGB += matSpecular*lightSpecular[i]*
                pow(clamp(dot(eyeObj,reflect(-lightObj,N)),0.0,1.0),
                matShininess);
    }
    fragColor *= vec4(fragColorRGB,1.0);
}
```

dove notiamo che:

- La costante `MAX_NUM_LIGHTS` identifica il numero massimo di sorgenti luminose ad illuminazione diretta che la libreria è in grado di gestire.
- La variabile di uscita `fragColor` rappresenta il colore risultante per il frammento che eventualmente potrebbe andare a finire nel framebuffer dopo il controllo sulla profondità e l'alpha blending.

- Tutte le operazioni avvengono in spazio oggetto e le informazioni per le varie sorgenti luminose sono racchiuse in alcuni variabili uniform di tipo array. Per ciascuna sorgente luminosa è disponibile la posizione in object space che viene utilizzata insieme agli altri parametri per applicare il Phong reflection model sul frammento per ognuna delle sorgenti (i vari contributi vengono cumulati).

- La quantità `matOpacity` definisce la trasparenza del materiale ed è data da $1 - t$, dove il parametro t per il materiale è stato introdotto in sez. 2.3. Come già detto, i materiali trasparenti vengono gestiti (per quanto riguarda l'ordine di disegno dei poligoni) come i materiali opachi. La risoluzione di tutti i problemi che possono nascere in presenza di oggetti trasparenti viene lasciata all'utente.

- Il vettore normale ricevuto (`fragNormalObj`) viene normalizzato prima di venire utilizzato, questa operazione è necessaria perché l'interpolazione (secondo il metodo di Phong) che viene eseguita automaticamente dalla pipeline OpenGL passando dal vertex shader al fragment shader (grazie al qualificatore `smooth`) non produce necessariamente un vettore con norma unitaria.

4.2 Diffuse Texture Mapping

Abbiamo esaminato questa tecnica nel par. 2.1.3. Sostanzialmente si tratta di utilizzare un'immagine come texture durante il rendering allo scopo di modulare il colore proprio del materiale della mesh. L'implementazione di questa tecnica basandosi sugli shader illustrati in sez. 4.1 è banale in quanto è sufficiente estendere il vertex shader per propagare agli stadi successivi della pipeline le coordinate di accesso alla texture per ogni vertice; queste coordinate verranno interpolate prima di arrivare allo stadio di fragment shading.

Le coordinate ricevute dal fragment shader verranno poi utilizzate per accedere alla texture ed impostare in questo modo il colore iniziale del frammento che poi viene modulato tramite il materiale e i parametri di illuminazione come visto per lo shading di base. In GLSL l'accesso da shader alle texture avviene per mezzo di oggetti detti *sampler*, i quali vengono impostati prima di lanciare il rendering e consentono, date le coordinate di accesso alla texture, l'estrazione del valore nella locazione desiderata (opportunamente filtrato).

Pertanto nel vertex shader dovremo aggiungere:

```
...
void main(void) {
    ...
    fragNormalObj = normal;
    diffTexCoord = diffCoord;
}
```

Nel fragment shader invece sarà sufficiente aggiungere alcune linee per le nuove variabili varying e uniform in ingresso e per eseguire la semplice operazione di modulazione del colore del materiale, come segue:

```
...
uniform sampler2D diffSampler; // diffuse texture sampler
uniform mat4 diffMatrix; // diffuse texture coordinates transformation
...
smooth in vec2 diffTexCoord; // diffuse texture coordinates

out vec4 fragColor; // resulting fragment color (with alpha)

void main(void) {
    fragColor = vec4(1.0,1.0,1.0, clamp(matOpacity,0.0,1.0));
    vec2 diffCoord = (diffMatrix*vec4(diffTexCoord,0.0,1.0)).st;
    fragColor *= texture(diffSampler, diffCoord);
    vec3 fragColorRGB = matEmission;
    ...
    fragColor *= vec4(fragColorRGB,1.0);
}
```

dove notiamo che:

- La diffuse texture viene applicata insieme al materiale e non al posto di esso: la texture modula i colori risultanti e dunque l'effetto finale dipende ancora dalle proprietà del materiale applicato all'oggetto virtuale e dall'illuminazione presente nella scena (che determinano il contenuto di `fragColorRGB`).
- Prima di campionare la texture sfruttando l'oggetto sampler `diffSampler`, si trasformano le coordinate ricevute tramite l'utilizzo di una matrice di trasformazione `diffMatrix`. Questo ci consente di ottenere particolari effetti sulla texture, come ad esempio ruotarla sulla mesh oppure usarla come un elemento di base (*tile*) da replicare sulla superficie dell'oggetto stesso. Una simile trasformazione di coordinate viene prevista per tutte le texture utilizzate nella

VR3Lib e consente (visto che viene effettuata a tempo di rendering) anche la realizzazione di texture dinamiche.

- Come accennato in sez. 2.3, anche nelle texture può essere presente un canale alpha. Il valore iniziale di `fragColor` viene infatti modulato in tutte e 4 le sue componenti dal contenuto estratto dalla diffuse texture (e dunque anche l'alpha value usato per il blending può venire alterato).

4.3 Light Mapping

La tecnica del light mapping è particolarmente conveniente quando abbiamo a disposizione informazioni di illuminazione precalcolate e memorizzate in una texture accessibile a tempo di rendering (come visto nel par. 2.1.4). Quando si utilizza questa tecnica si assume che le informazioni di illuminazione siano tutte e sole quelle che abbiamo a disposizione grazie alla texture (light map) fornitaci. In questa situazione non si utilizza il Phong reflection model o metodi più sofisticati di illuminazione, ma si modula semplicemente il colore risultante dal materiale (componente diffuse) e dalla eventuale diffuse texture sulla base delle informazioni precalcolate di illuminazione.

L'illuminazione così ottenuta può essere arbitrariamente realistica sulla base della tecnica utilizzata per produrre la light map, ma risulta valida solamente fintanto che non si ha movimento relativo tra gli oggetti e le sorgenti luminose. Inoltre in questo caso l'illuminazione è indipendente dal punto di vista e dunque non saremo in grado di visualizzare in modo realistico oggetti riflettenti con la tecnica di light mapping (o almeno non ne saremo in grado se questi oggetti vengono visualizzati da molteplici angolazioni).

Sostanzialmente quindi per implementare la tecnica del light mapping possiamo usare lo stesso modello di vertex shader visto in sez. 4.1, in cui però non è più necessario propagare il valore delle normali ma invece bisogna fornire semplicemente le coordinate per l'accesso alla light map:

```
...
void main(void) {
    ...
    lightNormalMapCoord = lightCoord;
}
```

Si noti che la tecnica del light mapping si può combinare con il diffuse texture mapping (e in tal caso è necessario propagare anche le coordinate di accesso alla texture di diffuse oltre lo stadio di vertex shading). Questo principio è valido anche per molte altre combinazioni di tecniche, ma in questa trattazione ci limitiamo a mostrare gli shader nelle loro versioni di base. Naturalmente la nuova VR3Lib supporta tutte le combinazioni valide.

Dalla linea che abbiamo aggiunto al vertex shader iniziale si evince anche che la tecnica di light mapping non è compatibile con quella di normal mapping (infatti, per questioni di efficienza, si usa la stessa variabile varying di uscita per le due tipologie di informazioni). Se si opera secondo la tecnica di light mapping non è infatti necessario fornire le normali in quanto i parametri di illuminazione vengono letti direttamente dalla light map e le normali risulterebbero inutili.

Il fragment shader questa volta è notevolmente diverso in quanto non è necessario effettuare il calcolo per il Phong reflection model e le sorgenti luminose presenti nella scena vengono sostanzialmente ignorate. La forma è la seguente:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform vec3 matEmission; // material emission color
uniform vec3 matDiffuse; // material diffuse color
uniform float matOpacity; // material opacity
uniform sampler2D lightSampler; // light map sampler
uniform mat4 lightMatrix; // light map coordinates transformation

smooth in vec2 lightNormalMapCoord; // light map coordinates

out vec4 fragColor; // resulting fragment color (with alpha)

void main(void) {
    fragColor = vec4(1.0,1.0,1.0, clamp(matOpacity,0.0,1.0));
    vec3 fragColorRGB = matEmission;
    vec2 lightCoord = (lightMatrix*vec4(lightNormalMapCoord,0.0,1.0)).st;
    fragColorRGB += matDiffuse*texture(lightSampler, lightCoord).rgb;
    fragColor *= vec4(fragColorRGB,1.0);
}
```

dove notiamo che:

- Solamente la componente diffuse dei parametri di colore del materiale viene modulata con la light map per produrre il risultato finale di illuminazione. La componente ambient viene ignorata in quanto le informazioni di illuminazione ambientale (se necessarie) sono presenti nella light map. La componente

specular viene ignorata perchè l'illuminazione che si ottiene con la tecnica del light mapping non può dipendere dal punto di vista.

- Visto che si utilizzano tre diversi canali della light map (`r`, `g` e `b`), è possibile introdurre nella stessa informazioni di colore oltre che informazioni di luminosità (questo può essere utile in alcuni casi dove si vogliono realizzare particolari effetti di illuminazione).

4.4 Normal Mapping

Nel par. 2.2.1 abbiamo discusso la tecnica del normal mapping, avente lo scopo di simulare la complessità geometrica degli oggetti tramite una perturbazione della normale alla superficie guidata dal contenuto di una texture (la normal map). Questa tecnica aumenta notevolmente il livello di realismo senza incrementare il numero di poligoni ed è infatti ormai comune in applicazioni ad elevato livello di realismo.

Nella nuova VR3Lib è stata implementata una versione del normal mapping che si aspetta di avere a che fare con una object-space normal map (per cui i vettori estratti dalla mappa sono direttamente le normali in spazio oggetto). Normalmente i programmi per la creazione di modelli tridimensionali permettono di generare questo tipo di normal map senza problemi.

Nel nostro caso quindi la realizzazione di uno shader program capace di effettuare normal mapping è molto semplice in quanto è sufficiente partire nuovamente dagli shader presentati in sez. 4.1 ed applicare solamente alcune modifiche.

Per quanto riguarda il vertex shader, quello visto in sez. 4.1 continua ad essere valido. In questo caso tuttavia non sarà necessario fornire agli stadi successivi della pipeline le normali calcolate ai vertici: basterà infatti propagare le coordinate di accesso alla normal map (supponendo di non combinare questa tecnica con altri approcci come il diffuse texture mapping). Bisogna quindi aggiungere al modello di vertex shader in sez. 4.1 quanto segue:

```
...
void main(void) {
    ...
    lightNormalMapCoord = normalCoord;
}
```

Il fragment shader invece si modifica leggermente in quanto le normali non vengono ricevute dal vertex shader e non si sta infatti utilizzando la Phong interpolation, invece bisogna accedere alla normal map e prelevare la normale necessaria usando le coordinate ricevute dal vertex shader (dopo l'usuale interpolazione). La struttura del fragment shader viene perciò alterata come segue:

```
...
uniform sampler2D normSampler; // normal map sampler
uniform mat4 normMatrix; // normal map coordinates transformation

smooth in vec3 fragPositionObj; // fragment position (obj coord)
smooth in vec2 lightNormalMapCoord; // normal map coordinates

out vec4 fragColor; // resulting fragment color (with alpha)

void main(void) {
    fragColor = vec4(1.0,1.0,1.0, clamp(matOpacity,0.0,1.0));
    vec2 normCoord = (normMatrix*vec4(lightNormalMapCoord,0.0,1.0)).st;
    vec3 N = normalize(texture(normSampler, normCoord).xyz * 2.0 - 1.0);
    vec3 fragColorRGB = matEmission;
    ...
}
```

dove notiamo che:

- Nella parte mancante al termine della funzione `main()` si calcola l'illuminazione come visto nel fragment shader in sez. 4.1, sfruttando però la normale in spazio oggetto appena ottenuta dalla normal map.
- L'accesso alla normal map ci fornisce in generale un vettore di 4 numeri reali tra 0 ed 1; l'alpha channel in questo caso (anche se presente) verrà ignorato. Per ottenere la normale effettiva si procede mappando ognuna delle componenti del vettore ottenuto dall'intervallo $[0, 1]$ all'intervallo $[-1, 1]$ (in accordo con quanto visto nel paragrafo 2.2.1). Si effettua anche una normalizzazione allo scopo di evitare problemi dovuti alla quantizzazione dei valori memorizzati nell'immagine usata come normal map (il vettore recuperato potrebbe non avere norma unitaria a causa di questa quantizzazione).
- L'approccio di lavorare in object coordinates è di grande aiuto visto che nella mappa sono memorizzate le normali in spazio oggetto: è infatti possibile sfruttare direttamente il vettore recuperato senza ulteriori trasformazioni.

4.5 Environment Mapping

Nel par. 2.1.5 abbiamo discusso in maniera approfondita delle tecniche di image-based lighting (IBL) arrivando anche a dare una formula per l'illuminazione di un punto P sulla base di una coppia di immagini usate per l'illuminazione della scena (diffuse cube environment map e specular cube environment map).

L'applicazione dell'illuminazione basata su immagini non dipende dal tipo particolare di oggetto che si sta disegnando: mentre l'utilizzo dello shader program per il normal mapping (ad esempio) è legato al particolare oggetto virtuale (che deve possedere i parametri necessari), l'environment mapping è una modalità alternativa all'illuminazione diretta tradizionale e dipende dalla configurazione della scena (può anche venire disattivato e riattivato più volte durante l'esecuzione).

Quando si ha a che fare con oggetti light mapped, tutte le informazioni di illuminazione sono statiche e ricavabili dalla light map, ma in tutti gli altri casi dobbiamo fare in modo che gli shader program utilizzati per il rendering funzionino anche nel caso di environment mapping (oltre che nel caso di illuminazione diretta).

L'environment mapping consiste semplicemente in un diverso calcolo da compiere nello stadio di fragment processing sulla base dei valori ricavati dalle cube map. I vertex shader presentati nelle precedenti sezioni di questo capitolo rimangono pertanto validi, mentre i fragment shader visti in precedenza (escludendo il caso di light mapping) sono incompleti in quanto è necessario aggiungere una nuova modalità di funzionamento in ognuno di essi che esegua i calcoli di illuminazione in accordo alla tecnica di environment mapping (eq. 2.3).

Quello che si fa è allora introdurre una variabile uniform per discriminare la modalità di illuminazione diretta dall'IBL, e sulla base del suo valore si calcola il vettore `fragColorRGB` in modo opportuno (Phong reflection model o equazione 2.3). Nel caso in cui venga richiesto environment mapping, si invoca una funzione per il calcolo del vettore `fragColorRGB`.

La funzione per il calcolo dell'illuminazione secondo l'algoritmo di environment mapping è stata scritta in un fragment shader separato ed ha la seguente forma:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform mat4 normalMatrix; // matrix for normals (obj -> wrld space)
uniform mat4 modelMatrix; // modeling matrix (obj -> wrld space)
uniform vec3 matDiffuse; // material diffuse color
uniform vec3 matSpecular; // material specular color
```

```
uniform vec3 eyePosObj; // eye position in world coordinates
uniform int envMode; // environment mapping mode
uniform samplerCube envDiffSampler; // diffuse environment map sampler
uniform samplerCube envSpecSampler; // specular environment map sampler

smooth in vec3 fragPositionObj;

vec3 ComputeIBL(vec3 normalObj) {
    vec3 retVal = vec3(0.0, 0.0 ,0.0);
    vec3 normalWorld = (normalMatrix*vec4(normalObj,0.0)).xyz;
    if( (envMode & 0x0001) != 0) { // diffuse environment map active
        retVal += matDiffuse*texture(envDiffSampler, normalWorld).rgb;
    }
    if( (envMode & 0x0002) != 0) { // specular environment map active
        vec3 eye = normalize(eyePosObj -
            (modelMatrix*vec4(fragPositionObj,1.0)).xyz);
        vec3 cubeCoord = reflect(-eye, normalWorld);
        retVal += matSpecular*texture(envSpecSampler, cubeCoord).rgb;
    }
    return retVal;
}
```

dove notiamo che:

- La variabile uniform intera `envMode` determina il tipo di illuminazione con cui si ha a che fare: 0 significa che non si sta facendo environment mapping, se la diffuse cube environment map è attiva avremo il bit meno significativo ad 1, e se la specular cube environment map è disponibile avremo il secondo bit meno significativo ad 1. Quando una delle due mappe manca, il relativo termine nella eq. 2.3 viene considerato nullo.
- Per accedere alle cube map si usano (come accennato nel par. 2.1.5) vettori a tre componenti (s, r, t) che vanno pensati applicati nel centro del cubo; non si usano più le coordinate ordinarie (u, v) come nel caso di accesso a texture bidimensionali.
- Manca il termine relativo al colore emissivo del materiale che si nota in eq. 2.3: questo viene inserito prima di chiamare la `ComputeIBL()` ed è sempre un semplice parametro additivo.
- L'accesso alla diffuse cube environment map avviene (come stabilito dall'equazione 2.3) tramite la normale nel frammento. Questa normale dipende dal tipo di shading utilizzato (normal mapping o Phong interpolation di base) e viene

ricevuta dalla funzione come parametro (in object coordinates). La normale deve essere trasformata in world coordinates prima di venire utilizzata per accedere alla diffuse environment map e questa operazione avviene per mezzo della matrice `normalMatrix`.

- L'accesso alla specular cube environment map avviene (come stabilito dall'equazione 2.3) in una direzione dettata dalla riflessione del vettore che va dalla posizione dell'osservatore al frammento considerato (ovviamente considerato in world coordinates). La variabile uniform `eyePosObj` contiene infatti in questo fragment shader la posizione dell'osservatore in world coordinates, si usa il nome `eyePosObj` per riutilizzare la variabile uniform che nel caso di illuminazione diretta contiene la posizione della telecamera in object space.
- La trasformazione in world space della posizione del vertice e della normale aggiunge un costo a questa procedura derivante dal fatto che tutte le altre operazioni avvengono in object space. Questo è lo svantaggio che si affronta lavorando in object coordinates, ma ricordiamo che lavorando in world space otterremmo altri svantaggi in altre situazioni di shading (ad esempio in caso di normal mapping).

Notare che per trasformare le normali in world space non abbiamo usato la matrice di modeling `modelMatrix`, ma ci siamo invece affidati alla matrice (potenzialmente diversa) `normalMatrix`. Questo deriva dal fatto che se la modeling matrix contiene dei cambiamenti di scala il vettore che ricaviamo trasformando la normale mediante la `modelMatrix` è diverso dal vettore normale in world space: potremmo ottenere un vettore alterato con una direzione diversa da quella originale (se prima ad esempio la normale era ortogonale alla superficie, dopo la moltiplicazione per la matrice di modeling potrebbe non esserlo più). Per trasformare i vettori normali in world space è in realtà necessario utilizzare una matrice diversa:

$$M_{normal} = \left(M_{modeling}^{-1}\right)^T$$

I fragment shader visti in precedenza devono essere modificati per sfruttare i servizi offerti dalla funzione `ComputeIBL()` nel caso in cui si stia facendo environment mapping. La modifica introduce semplicemente un salto condizionale sulla base della nuova variabile uniform, come segue:

```
...
uniform int envMode; // environment mapping mode
...
out vec4 fragColor; // resulting fragment color (with alpha)

vec3 ComputeIBL(vec3 normalObj); // function declaration
void main(void) {
    ...
    vec3 fragColorRGB = matEmission;
    ...
    if(envMode == 0) { // no environment mapping
        [ USUAL SHADING ]
    } else { // environment mapping
        fragColorRGB += ComputeIBL(N);
    }
    fragColor *= vec4(fragColorRGB,1.0);
}
```

Dunque se si utilizza illuminazione diretta (`envMode` = 0) si entra nel primo ramo del costrutto `if`, che contiene le istruzioni per lo shading come descritto nelle precedenti sezioni di questo capitolo.

Il fragment shader contenente la funzione `ComputeIBL()` e quello classico modificato come sopra verranno compilati e collegati insieme in un unico fragment shader executable da eseguire nello stadio di fragment processing (come se si trattasse di un singolo fragment shader).

Il modo in cui opera la funzione `ComputeIBL()` risulta valido anche nel caso di environment map ad elevato range dinamico (HDR). L'utilizzo di immagini HDR garantisce che non si presenteranno artefatti dovuti alla non linearità della codifica e che avremo un calcolo molto preciso dei valori di illuminazione (par. 2.1.5). È importante notare però che non si applicano particolari operatori di tone mapping e non si sfruttano effetti realizzati in un nuovo passo di rendering. Ci si affida invece alle operazioni di base OpenGL per effettuare il mapping sul ridotto range dinamico del display (per cui l'immagine non risulterà molto diversa da quella ottenuta con illuminazione non HDR).

4.6 Displacement Mapping

La tecnica del displacement mapping è stata esaminata nel par. 2.2.2 e corrisponde ad una alterazione della geometria dell'oggetto a tempo di rendering, sulla base di un'immagine monocromatica detta height map (o displacement map). Nella nuova

VR3Lib tale tecnica è supportata nativamente e la geometria viene modificata nello stadio di geometry processing.

Affidarsi ad un geometry shader è la scelta più ovvia usando la versione 3.3 del sistema grafico OpenGL. Si opera dunque effettuando un tassellamento di ogni triangolo all'interno dello stadio di geometry processing. Il tassellamento può essere di vario tipo, nel nostro caso la procedura è molto semplice e prende il nome di *tassellamento a suddivisione lineare*: si utilizza un parametro intero TF (*Tessellation Factor*) che indica in quante parti vanno divisi i tre spigoli di ogni triangolo e si uniscono i punti generati con linee parallele agli spigoli del triangolo (generando dunque nuovi triangoli). Da un punto di vista pratico da ogni triangolo si costruiscono TF diverse *triangle strip*[15] (fasce di triangoli) parallele ad uno degli spigoli. A partire da un singolo triangolo si producono TF^2 triangoli nello stadio di geometry processing; vedi fig. 4.1 (valida per $TF = 3$, 9 triangoli risultanti dal tassellamento).

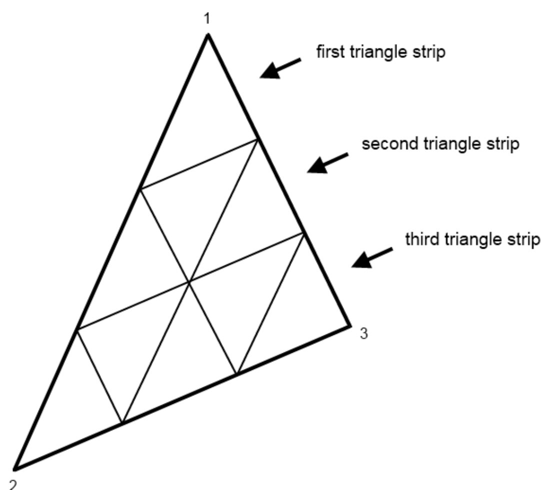


Figura 4.1: Tassellamento a suddivisione lineare con $TF = 3$

I vertici risultanti saranno i 3 vertici del triangolo iniziale più diversi vertici generati grazie all'algoritmo di tassellamento, questi vengono spostati lungo una normale calcolata localmente di una quantità data da:

$$BI + SC \cdot height$$

[15] Una triangle strip è un tipo di primitiva OpenGL che specifica un certo numero di triangoli organizzati in una fascia fornendo i primi tre vertici (per il primo triangolo) e poi un singolo vertice addizionale per ogni triangolo aggiunto alla fascia.

dove BI ($BIas$) e SC ($SCale$) sono ulteriori parametri che dipendono dall'oggetto e indicano come è stata calcolata la height map (e dunque come alterare la posizione dei vertici in accordo alla stessa), *height* invece è il valore letto dalla height map (un numero reale compreso tra 0 ed 1 che indica l'intensità dell'immagine in un certo punto).

Ogni striscia viene generata separatamente e dunque alcuni vertici emessi (appartenenti a diverse triangle strips) coincideranno. Il numero di vertici emessi in totale dal geometry shader (compresi quelli coincidenti) è dato da:

$$TF \cdot (TF + 2)$$

Quest'ultima informazione è importante perchè lo stadio di geometry processing ha un limite massimo al numero totale di componenti (variabili varying) emesse per tutta la geometria generata. Sapendo allora quante componenti vengono emesse per ogni vertice possiamo calcolare il massimo valore per TF. Quando si utilizza un TF superiore al valore massimo per un particolare oggetto non si verifica nessun errore, ma la geometria emessa è incompleta e l'oggetto presenterà dei poligoni mancanti una volta visualizzato a schermo. Il numero massimo di componenti totali di cui stiamo parlando dipende dall'implementazione di OpenGL: possiamo ricavarne il valore N_{max} richiedendolo al server GL tramite l'identificatore `GL_MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS`. Esistono anche altri vincoli da rispettare nel geometry shading, ma questo risulta quello più stretto per quanto riguarda la geometria emessa.

Il numero di componenti emesse per vertice cambia a seconda del tipo di oggetto. Chiamando questo valore n, la seguente relazione risulta valida (per come sono stati scritti gli shader e realizzate le altre tecniche presentate):

$$n = \begin{cases} 9 \quad if \quad not\ diffuse\ texture\ mapped \\ 11 \quad if \quad diffuse\ texture\ mapped \end{cases}$$

per cui il massimo valore di TF per un particolare oggetto sarà dato da:

$$TF_{max} = \max TF \, : \, n \cdot TF \cdot (TF + 2) \leq N_{max}$$

Quando si fa displacement mapping i vertici vengono spostati lungo le normali calcolate a partire dalla geometria. Questo significa che se l'oggetto è solido e vi sono diversi smoothing groups l'effetto ottenuto potrebbe non essere quello desiderato in quanto i poligoni generati potrebbero interesecarsi dopo l'applicazione della height

map. In generale quando si fa displacement mapping conviene inserire tutti i triangoli in un unico smoothing group allo scopo di evitare questi problemi. Le normali di cui sopra, utilizzate per effettuare il displacement mapping, non vengono poi più considerate nello shading: quando si fa displacement mapping è infatti obbligatorio fornire anche una normal map contenente le normali finali da usare per i calcoli di illuminazione oppure una light map che specifichi direttamente come l'oggetto con la geometria finale deve essere illuminato.

Il geometry shader dovrà quindi sempre ricevere le normali (anche nel caso di light mapping), e dovrà inoltre ricevere le coordinate per l'accesso alla height map. Quando si fa displacement mapping il vertex shader di base assume quindi la seguente forma:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

in vec3 vertex; // vertex position in obj coordinates
in vec3 normal; // normal in obj coordinates
in vec2 diffCoord; // diffuse texture coordinates
in vec2 lightCoord; // light map coordinates
in vec2 normalCoord; // normal map coordinates
in vec2 dispCoord; // displacement map coordinates

smooth out vec3 verPositionObj;
smooth out vec3 verNormalObj;
smooth out vec2 verDiffTexCoord;
smooth out vec2 verLightNormalMapCoord;
smooth out vec2 verDispMapCoord;

void main(void) {
    verPositionObj = vertex;
    verNormalObj = normal;
    verDispMapCoord = dispCoord;
    ...
}
```

dove notiamo che:

- La posizione post-proiezione dei vertici (`gl_Position`) non viene impostata in quanto questa operazione avviene all'interno del geometry shader (il dato `gl_Position` serve fondamentalmente prima di arrivare alla rasterizzazione dei poligoni e dunque non è necessario fissarne il valore prima di arrivare allo stadio di geometry processing). Dato che non si deve scrivere su `gl_Position`, non è più necessario ricevere la matrice di trasformazione `transformMatrix` come variabile uniform.

- La porzione mancante della funzione `main()` viene utilizzata in modo analogo a quanto descritto nelle sezioni precedenti in dipendenza dal particolare tipo di shading che si vuole effettuare per l'oggetto.

Adesso esaminiamo la struttura del geometry shader che si occupa di effettuare il tassellamento e di alterare la geometria. Assumiamo di non avere una diffuse texture e di calcolare i parametri di illuminazione nello stadio di fragment shading sulla base delle normali memorizzate in una normal map; in tal caso al vertex shader di cui sopra bisognerà aggiungere:

```
...
void main(void) {
    ...
    verLightNormalMapCoord = normalCoord;
}
```

Il geometry shader avrà invece la seguente struttura:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

// triangles are received
layout(triangles) in;
// triangle strips are produced
layout(triangle_strip, max_vertices = MAX_OUTPUT_VERTICES) out;

uniform mat4 transformMatrix; // proj*view*model matrix
uniform int dispTF; // tessellation factor
uniform float dispBI; // bias
uniform float dispSC; // scale
uniform sampler2D dispSampler; // height map sampler
uniform mat4 dispMatrix; // displacement map coordinates transformation

smooth in vec3 verPositionObj[3]; // vertex positions
smooth in vec3 verNormalObj[3]; // vertex normals (geometry-based)
smooth in vec2 verLightNormalMapCoord[3]; // normal map coordinates
smooth in vec2 verDispMapCoord[3]; // displacement map coordinates

smooth out vec2 lightNormalMapCoord; // normal map coordinates
smooth out vec3 fragPositionObj; // resulting vertex positions

void main(void) {
    int i, j, k;
    float disp;
    vec2 dispCoord;
```

```
vec3 norm;
vec3 v01 = (verPositionObj[1] - verPositionObj[0]) / dispTF;
vec3 v12 = (verPositionObj[2] - verPositionObj[1]) / dispTF;
vec2 delta_uv01_map =
    (verLightNormalMapCoord[1]-verLightNormalMapCoord[0]) / dispTF;
vec2 delta_uv12_map =
    (verLightNormalMapCoord[2]-verLightNormalMapCoord[1]) / dispTF;
vec2 delta_uv01_disp = (verDispMapCoord[1]-verDispMapCoord[0]) / dispTF;
vec2 delta_uv12_disp = (verDispMapCoord[2]-verDispMapCoord[1]) / dispTF;
vec3 delta_norm01 = (verNormalObj[1]-verNormalObj[0]) / dispTF;
vec3 delta_norm12 = (verNormalObj[2]-verNormalObj[1]) / dispTF;
vec3 bv0;
vec3 bv1;
for(i = 0; i <= dispTF-1; i++) { // for each strip
    bv0 = verPositionObj[0] + v01*i;
    bv1 = verPositionObj[0] + v01*(i+1);
    // emit the first vertex of the first triangle
    lightNormalMapCoord = verLightNormalMapCoord[0] +
        delta_uv01_map*(i+1) + delta_uv12_map*(i+1);
    dispCoord = verDispMapCoord[0] +
        delta_uv01_disp*(i+1) + delta_uv12_disp*(i+1);
    dispCoord = (dispMatrix*vec4(dispCoord,0.0,1.0)).st;
    norm = normalize(verNormalObj[0] +
        delta_norm01*(i+1) + delta_norm12*(i+1));
    disp = texture(dispSampler, dispCoord).r;
    fragPositionObj = bv1+v12*(i+1)+norm*(dispBI+disp*dispSC);
    gl_Position = transformMatrix*vec4(fragPositionObj,1.0);
    EmitVertex();
    // emit the second vertex of the first triangle
    lightNormalMapCoord = verLightNormalMapCoord[0] +
        delta_uv01_map*i + delta_uv12_map*i;
    dispCoord = verDispMapCoord[0] +
        delta_uv01_disp*i + delta_uv12_disp*i;
    dispCoord = (dispMatrix*vec4(dispCoord,0.0,1.0)).st;
    norm = normalize(verNormalObj[0] +
        delta_norm01*i + delta_norm12*i);
    disp = texture(dispSampler, dispCoord).r;
    fragPositionObj = bv0+v12*i+norm*(dispBI+disp*dispSC);
    gl_Position = transformMatrix*vec4(fragPositionObj,1.0);
    EmitVertex();
    // emit the third vertex of the first triangle
    lightNormalMapCoord = verLightNormalMapCoord[0] +
        delta_uv01_map*(i+1) + delta_uv12_map*i;
    dispCoord = verDispMapCoord[0] +
        delta_uv01_disp*(i+1) + delta_uv12_disp*i;
    dispCoord = (dispMatrix*vec4(dispCoord,0.0,1.0)).st;
    norm = normalize(verNormalObj[0] +
        delta_norm01*(i+1) + delta_norm12*i);
    disp = texture(dispSampler, dispCoord).r;
    fragPositionObj = bv1+v12*i+norm*(dispBI+disp*dispSC);
    gl_Position = transformMatrix*vec4(fragPositionObj,1.0);
    EmitVertex();
```

```
        // emit more vertices building the strip
        for(j = 2*i; j >= 1 ; j--) {
            if(j%2 != 0) {
                k = (j-1)/2;
                lightNormalMapCoord = verLightNormalMapCoord[0] +
                    delta_uv01_map*(i+1) + delta_uv12_map*k;
                dispCoord = verDispMapCoord[0] +
                    delta_uv01_disp*(i+1) + delta_uv12_disp*k;
                dispCoord = (dispMatrix*vec4(dispCoord,0.0,1.0)).st;
                norm = normalize(verNormalObj[0] +
                    delta_norm01*(i+1) + delta_norm12*k);
                disp = texture(dispSampler, dispCoord).r;
                fragPositionObj = bv1+v12*k+norm*(dispBI+disp*dispSC);
                gl_Position = transformMatrix*vec4(fragPositionObj,1.0);
                EmitVertex();
            } else {
                k = j/2-1;
                lightNormalMapCoord = verLightNormalMapCoord[0] +
                    delta_uv01_map*i + delta_uv12_map*k;
                dispCoord = verDispMapCoord[0] +
                    delta_uv01_disp*i + delta_uv12_disp*k;
                dispCoord = (dispMatrix*vec4(dispCoord,0.0,1.0)).st;
                norm = normalize(verNormalObj[0] +
                    delta_norm01*i + delta_norm12*k);
                disp = texture(dispSampler, dispCoord).r;
                fragPositionObj = bv0+v12*k+norm*(dispBI+disp*dispSC);
                gl_Position = transformMatrix*vec4(fragPositionObj,1.0);
                EmitVertex();
            }
        }
        EndPrimitive(); // finalize the triangle strip
    }
}
```

dove notiamo che:

- Le funzioni built-in nel linguaggio GLSL `EmitVertex()` e `EndPrimitive()` hanno lo scopo, rispettivamente, di emettere un vertice per la primitiva corrente e di finalizzare una primitiva (per iniziare ad emetterne una nuova). Nel nostro caso le primitive emesse sono triangle strips.
- La costante `MAX_OUTPUT_VERTICES` identifica il numero massimo di vertici che possono essere emessi dal geometry shader (dipende dall'implementazione di OpenGL e dal particolare tipo di shading che stiamo facendo). Nel nostro caso non abbiamo diffuse texture mapping e dunque questa costante avrà il valore:

$$\texttt{MAX_OUTPUT_VERTICES} = \left\lfloor \frac{N_{max}}{n} \right\rfloor = \left\lfloor \frac{N_{max}}{9} \right\rfloor$$

in accordo con quanto detto sopra.

Nel par. 2.2.2 abbiamo anche discusso di un'ottimizzazione all'algoritmo che prevede di effettuare il tassellamento in modo adattivo sulla base del punto di vista dell'osservatore. Purtroppo, lavorando col tassellamento a suddivisione lineare visto sopra, un'operazione del genere comporta che l'utente è spesso in grado di percepire le modifiche alla geometria sulla base del suo punto di vista. Per realizzare in modo efficace l'ottimizzazione descritta nel par. 2.2.2 bisogna usare algoritmi di tassellamento più complessi, ma non entriamo nei dettagli accontentandoci di questa soluzione indipendente dal punto di vista.

Visualizzando oggetti con la tecnica del displacement mapping realizzata come sopra può accadere che siano visibili delle cuciture sui punti di contatto tra i triangoli della mesh originale. Questo deriva dal fatto che i valori prelevati dalla height map per un dato vertice della mesh originale possono essere molteplici e in diverse posizioni della stessa, inoltre si esegue un filtraggio lineare sulla texture. Se i valori prelevati non sono sempre esattamente gli stessi per un dato vertice, avremo un'alterazione diversa delle posizione di questo vertice per i diversi poligoni di cui fa parte e di conseguenza la creazione di cuciture. Naturalmente questo fenomeno può essere evitato con una particolare costruzione della height map e delle coordinate per l'accesso ad essa (ad opera del programma usato per modellare l'oggetto), ma non entriamo nei dettagli.

4.7 Shadow Mapping con Tecnica EVSM

Nel capitolo 3 abbiamo esaminato svariate tecniche per proiettare ombre di oggetti virtuali su porzioni di loro stessi e altri oggetti virtuali, abbiamo visto come realizzare ombre convincenti tramite tecniche di soft shadow mapping e presentato l'innovativo algoritmo di exponential variance shadow mapping (EVSM) nel par. 3.3.5.

Nella nuova libreria si è inserita un'implementazione dell'algoritmo EVSM allo scopo di fornire all'utente un modo per ottenere scene con ombre proiettate ad elevato livello di realismo senza dover scrivere gli shader appositi. Ricordiamo che la procedura da seguire per disegnare la scena completa di ombre proiettate è quella descritta dall'algoritmo 3.6. Si pongono quindi alcuni problemi:

- Per quanto riguarda la creazione della shadow map, bisogna effettuare un primo passo di rendering off-screen che produca una texture a 4 canali in virgola mobile a precisione singola con i dati di interesse. Questo primo rendering deve fare affidamento su shader appositi e deve avvenire non nel framebuffer relativo alla finestra dell'applicazione OpenGL (*default framebuffer*) ma in un framebuffer generato appositamente dalla libreria per ogni sorgente luminosa (quello che in OpenGL viene chiamato *framebuffer object*, *FBO*). Le funzioni disponibili nell'interfaccia OpenGL ci consentono di effettuare direttamente rendering all'interno di una texture da usare successivamente.

- Bisogna trovare un modo per filtrare velocemente le shadow map generate prima di renderizzare la scena. Il tempo di filtraggio si va ad aggiungere al periodo di frame delle nostre applicazioni e dunque deve essere il più piccolo possibile.

- I fragment shader visti fino ad ora vanno modificati per modulare il colore ottenuto sulla base della quantità di luce ricevuta dal frammento (stimata come quantità P nell'algoritmo 3.6).

Esaminiamo le varie questioni separatamente. Il supporto alle ombre con tecnica EVSM integrato nella VR3Lib è in grado di gestire diverse sorgenti luminose. In questo contesto le sorgenti vengono intese come entità che causano la proiezione di ombre da parte degli oggetti nella scena e nella nuova libreria vengono distinte dalle luci di scena (usate nell'illuminazione diretta), il termine più appropriato in questo caso è *sorgente d'ombra* (*shadow source*). Nel seguito della sezione consideriamo la situazione semplificativa in cui abbiamo una singola sorgente (e dunque è necessario costruire e filtrare una singola shadow map); gli shader che mostreremo sono quelli che si utilizzano in tale caso (e quindi sono una semplificazione di quelli effettivamente presenti nella VR3Lib in quanto la struttura dipende in realtà dal numero di sorgenti d'ombra che si vogliono supportare).

4.7.1 Costruzione della Shadow Map

Il primo passo nell'algoritmo EVSM è quello di costruire la shadow map all'interno di una texture a 4 canali in virgola mobile a precisione singola. Questo avviene per mezzo di un FBO cui è agganciata una texture che viene usata come buffer di colore (dove verranno salvati i valori della profondità trasformata) e un depth buffer per abilitare il depth test durante il rendering della shadow map. I valori salvati non

possono ancora essere chiamati momenti in quanto il termine è appropriato solo una volta che è avvenuto il filtraggio.

Trascuriamo gli aspetti di utilizzo dell'interfaccia OpenGL per preparare un FBO per il rendering come descritto sopra, concentriamoci invece sugli shader che vengono utilizzati per ottenere la shadow map dal punto di vista della sorgente d'ombra.

Il vertex shader utilizzato risulta il seguente:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform mat4 modelViewMatrix; // view(from_source)*model matrix
uniform mat4 projMatrix; // shadow source proj matrix

in vec3 vertex; // vertex position in object coordinates

smooth out vec3 fragPosition; // vertex position in eye space

void main(void) {
    vec4 eyepos = modelViewMatrix*vec4(vertex,1.0);
    fragPosition = eyepos.xyz;
    gl_Position = projMatrix*eyepos; // clip coordinates
}
```

dove notiamo che:

- Riceviamo in ingresso solamente le posizioni dei vertici; tutti gli altri dati disponibili nei vertex buffer objects vengono ignorati.
- Passiamo agli stadi successivi della pipeline la posizione dei vertici in eye space, dove ricordiamo che la scena viene fotografata dal punto di vista della sorgente d'ombra.
- Anche se la mesh corrente è dotata di una displacement map, la geometria considerata durante il tracciamento della shadow map sarà quella originale. La tecnica di displacement mapping ha infatti come obiettivo fondamentale quello di aumentare la complessità geometrica senza caricare eccessivamente la pipeline grafica, aumentando il livello di dettaglio degli oggetti virtuali. Siccome si tratta appunto di dettagli, non si ottiene un vantaggio significativo alterando la geometria durante il disegno della shadow map (ma si rischia invece di caricare troppo la pipeline inserendo anche in questo primo passo di rendering un geometry shader come quello in sez. 4.6).

Il fragment shader assume la seguente forma:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform float depthScale; // depth scale EVSM parameter
uniform float zFar; // far clipping plane distance
uniform float zNear; // near clipping plane distance

smooth in vec3 fragPosition; // fragment position in eye space

out vec4 moments; // output vector for this shadow map texel

void main(void) {
    float d = length(fragPosition);
    d = (d-zNear)/(zFar-zNear);
    d = min(d,1.0);
    moments[0] = -exp( -depthScale * d );
    moments[1] = moments[0] * moments[0];
    moments[2] = exp( depthScale * d );
    moments[3] = moments[2] * moments[2];
}
```

dove notiamo che la metrica di profondità utilizzata è una metrica lineare (cresce linearmente all'aumentare della distanza dalla sorgente), e viene calcolata come segue:

$$D_s = \frac{d - zNear}{zFar - zNear}$$

dove d è la distanza dalla sorgente d'ombra, e $zFar$ e $zNear$ sono dei parametri che dipendono dal volume di vista usato per la proiezione delle ombre. Il valore risultante per D_s va da 0 ad 1 per frammenti che giacciono direttamente di fronte alla shadow source. Un D_s calcolato in questo modo può superare 1 per frammenti che stanno in fondo al volume di vista ma spostati verso gli angoli della base del tronco di piramide, questa situazione è potenzialmente pericolosa perchè il valore massimo di D_s definisce anche il valore utilizzato per `depthScale` (che abbiamo chiamato k nell'algoritmo 3.6 e deve essere il più alto possibile). Si evita allora che D_s superi 1 con un semplice troncamento (questo non crea problemi perché frammenti oltre una distanza $zFar$ dalla shadow source non comporteranno mai un accesso alla shadow map durante il rendering della scena anche se interni al view frustum della relativa shadow source, vedi par. 4.7.3).

4.7.2 Filtraggio della Shadow Map

Il secondo problema che bisogna affrontare è la realizzazione di un filtraggio efficiente da poter effettuare ad ogni fotogramma sulla shadow map. Il filtro utilizzato deve essere un filtro di convoluzione che combina i valori presenti nella mappa producendo dei momenti utilizzati poi per la stima della quantità di luce ricevuta. La funzione utilizzata per la convoluzione può essere di vario tipo (nel caso più semplice la media dei valori in un certo intorno). Nel nostro caso usiamo una funzione gaussiana per cercare di dare un peso ragionevole ad ognuno dei texel nell'intorno esaminato. Il filtro che ci si propone di realizzare è pertanto un filtro di *blur gaussiano* (*gaussian blur*).

Il filtraggio di una texture è un'operazione semplice ma normalmente onerosa: per produrre un singolo texel della mappa filtrata è necessario elaborare diversi elementi della texture originale, combinandoli tra loro con operazioni in virgola mobile. Data la natura dei calcoli da eseguire, risulta conveniente effettuare il filtraggio sfruttando i core della GPU in parallelo (questo vale per tutte le moderne schede video). Ne deriva che per ottenere una procedura efficiente si opera predisponendo un nuovo FBO con una nuova texture e renderizzando la texture filtrata direttamente all'interno della nuova texture, combinando i texel dell'immagine direttamente nello stadio di fragment shading (sfruttando dunque la pipeline OpenGL).

Nella pratica questo è fattibile disegnando un piano che copre l'intero viewport (corrispondente alle dimensioni della texture) e calcolando il colore di ogni frammento combinando diversi campioni dalla shadow map originale (vista come texture). L'accesso alla shadow map non filtrata avviene direttamente mediante le coordinate del frammento nel viewport.

Supponiamo adesso che ci venga dato il kernel quadrato del filtro da utilizzare (di dimensione m). Se l'immagine da filtrare è quadrata di dimensione n (in pixel) avremo una complessità computazionale complessiva per filtrare gli n^2 pixel di $O(n^2m^2)$ in termini di operazioni elementari: per ognuno degli n^2 pixel bisogna effettuare m^2 moltiplicazioni per i coefficienti del kernel e $m^2 - 1$ somme.

Questa complessità è considerevole e proibisce di espandere il kernel del filtro (aumentare m) oltre un certo livello (nonostante il filtraggio non venga effettuato durante il rendering della scena, a differenza del caso PCF presentato nel paragrafo 3.3.1).

Per migliorare le prestazioni si ricorre ad un *filtro separabile*. Un filtro si dice separabile quando è possibile effettuare il filtraggio in due passate ognuna delle quali

opera in una singola direzione (con un kernel monodimensionale). Il filtro gaussiano è separabile nelle due direzioni e dunque per ottenere un approccio più efficiente possiamo scomporre il filtraggio in due diversi filtraggi con kernel monodimensionali (filtraggio orizzontale e a seguire filtraggio verticale).

Per ottenere lo stesso effetto del filtro bidimensionale con kernel quadrato di dimensione m si deve ricorrere ad un doppio filtraggio con un kernel monodimensionale di dimensione m, prima orizzontale e poi verticale. La complessità di uno dei due passi di filtraggio diventa $O(n^2m)$ perché per ognuno degli n^2 pixel si effettuano solamente m moltiplicazioni per i coefficienti e $m-1$ somme. La complessità del filtraggio in due passate risulta pertanto $O(n^2m)$.

Ne deriva che useremo un diverso shader program per ognuna delle due passate. La shadow map filtrata dopo il primo passo viene costruita in una nuova texture di un nuovo FBO; questa viene poi fornita al secondo passo di filtraggio come immagine in ingresso.

Il vertex shader è identico nelle due passate e si occupa semplicemente di fissare la posizione dei vertici del piano, la sua struttura è la seguente:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

in vec2 vertex; // plane vertex position (clip space, (x,y) only)

void main(void) {
    gl_Position = vec4(vertex,0.0,1.0); // clip coordinates
}
```

dove notiamo che le coordinate (x, y) per i vertici del piano vengono già fornite in clip space in modo tale da riempire tutta l'area del viewport indipendentemente dalle sue dimensioni.

Il fragment shader utilizzato invece dipende dal particolare filtraggio eseguito (orizzontale o verticale), la forma è simile nei due casi e come esempio diamo solo la struttura dello shader scritto per eseguire il filtraggio orizzontale della shadow map:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform sampler2D texSampler; // sampler for the image to filter
uniform vec2 texSize; // image size (in pixels)
uniform int filterSize; // gaussian filter size (0 to 9)
```

```
out vec4 fragColor; // filtered value for this fragment

const float coefficients0[10]  = float[10]( 0.9734, 0.5984, 0.3990,
    0.2994, 0.2396, 0.1996, 0.1712, 0.1498, 0.1332, 0.1198 );
const float coefficients1[10]  = float[10]( 0.0133, 0.1942, 0.2421,
    0.2260, 0.2001, 0.1762, 0.1562, 0.1396, 0.1260, 0.1146 );
const float coefficients2[10]  = float[10]( 0.0000, 0.0066, 0.0540,
    0.0972, 0.1166, 0.1212, 0.1186, 0.1131, 0.1067, 0.1002 );
const float coefficients3[10]  = float[10]( 0.0000, 0.0000, 0.0044,
    0.0238, 0.0474, 0.0648, 0.0749, 0.0796, 0.0808, 0.0799 );
const float coefficients4[10]  = float[10]( 0.0000, 0.0000, 0.0000,
    0.0033, 0.0134, 0.0270, 0.0393, 0.0486, 0.0547, 0.0584 );
const float coefficients5[10]  = float[10]( 0.0000, 0.0000, 0.0000,
    0.0000, 0.0027, 0.0088, 0.0172, 0.0258, 0.0332, 0.0390 );
const float coefficients6[10]  = float[10]( 0.0000, 0.0000, 0.0000,
    0.0000, 0.0000, 0.0022, 0.0063, 0.0119, 0.0180, 0.0237 );
const float coefficients7[10]  = float[10]( 0.0000, 0.0000, 0.0000,
    0.0000, 0.0000, 0.0000, 0.0019, 0.0048, 0.0087, 0.0132 );
const float coefficients8[10]  = float[10]( 0.0000, 0.0000, 0.0000,
    0.0000, 0.0000, 0.0000, 0.0000, 0.0017, 0.0038, 0.0067 );
const float coefficients9[10]  = float[10]( 0.0000, 0.0000, 0.0000,
    0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0015, 0.0031 );
const float coefficients10[10] = float[10]( 0.0000, 0.0000, 0.0000,
    0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0013 );

void main(void) {
    float blurSize = 1.0/texSize.x;
    vec2 fragTexCoord = gl_FragCoord.xy/texSize;
    vec4 sum = vec4(0.0);
    switch(filterSize) {
    case 9:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            10.0*blurSize, fragTexCoord.y)) * coefficients10[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            10.0*blurSize, fragTexCoord.y)) * coefficients10[filterSize];
    case 8:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            9.0*blurSize, fragTexCoord.y)) * coefficients9[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            9.0*blurSize, fragTexCoord.y)) * coefficients9[filterSize];
    case 7:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            8.0*blurSize, fragTexCoord.y)) * coefficients8[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            8.0*blurSize, fragTexCoord.y)) * coefficients8[filterSize];
    case 6:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            7.0*blurSize, fragTexCoord.y)) * coefficients7[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            7.0*blurSize, fragTexCoord.y)) * coefficients7[filterSize];
    case 5:
        sum += texture(texSampler, vec2(fragTexCoord.x -
```

```
            6.0*blurSize, fragTexCoord.y)) * coefficients6[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            6.0*blurSize, fragTexCoord.y)) * coefficients6[filterSize];
    case 4:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            5.0*blurSize, fragTexCoord.y)) * coefficients5[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            5.0*blurSize, fragTexCoord.y)) * coefficients5[filterSize];
    case 3:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            4.0*blurSize, fragTexCoord.y)) * coefficients4[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            4.0*blurSize, fragTexCoord.y)) * coefficients4[filterSize];
    case 2:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            3.0*blurSize, fragTexCoord.y)) * coefficients3[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            3.0*blurSize, fragTexCoord.y)) * coefficients3[filterSize];
    case 1:
        sum += texture(texSampler, vec2(fragTexCoord.x -
            2.0*blurSize, fragTexCoord.y)) * coefficients2[filterSize];
        sum += texture(texSampler, vec2(fragTexCoord.x +
            2.0*blurSize, fragTexCoord.y)) * coefficients2[filterSize];
    }
    sum += texture(texSampler, vec2(fragTexCoord.x -
        blurSize, fragTexCoord.y)) * coefficients1[filterSize];
    sum += texture(texSampler, vec2(fragTexCoord.x +
        blurSize, fragTexCoord.y)) * coefficients1[filterSize];
    sum += texture(texSampler,
        vec2(fragTexCoord.x, fragTexCoord.y)) * coefficients0[filterSize];
    fragColor = sum;
}
```

dove notiamo che:

- Nella variabile di uscita `fragColor` si scrivono i valori filtrati per il texel (frammento) corrente sui 4 canali di colore (R, G, B e A).
- È possibile specificare un raggio per il filtro gaussiano tramite la variabile uniform `filterSize` (che va da 0 a 9): dato il valore di questa variabile, il kernel bidimensionale equivalente al filtraggio in due passate applicato avrà ampiezza (in pixel) pari a $2 \cdot \texttt{filterSize} + 3$.
- I coefficienti nel kernel monodimensionale da applicare in questa passata sono memorizzati in una serie di vettori costanti di numeri reali. Ogni vettore contiene tutti i coefficienti ad una certa distanza dal centro del filtro (un coefficiente per ogni `filterSize` legale).

- Il costrutto `switch` si comporta in modo analogo all'omonimo costrutto in C.
- Si usa la variabile `gl_FragCoord` built-in nel linguaggio GLSL per calcolare le coordinate con cui accedere alla texture. Possiamo considerare il contenuto di questa variabile come le window coordinates del frammento (anche se è una semplificazione).
- Le dimensioni dell'immagine da filtrare vengono passate esplicitamente come variabile uniform (`texSize`). Si potrebbe pensare in alternativa di estrarre le dimensioni con delle funzioni apposite di GLSL che interrogano il sistema OpenGL a partire dal sampler, tuttavia la soluzione risulterebbe meno efficiente.

Naturalmente il filtraggio verticale avviene con un fragment shader molto simile a quello sopra, in cui cambiano solamente dei dettagli di accesso alla texture (shadow map) da filtrare.

4.7.3 Rendering della Scena

Per completare la realizzazione della tecnica EVSM (nel caso semplificato di singola sorgente d'ombra), resta da vedere come modificare i fragment shader visti sopra per sfruttare la shadow map generata e filtrata.

Come spiegato nell'algoritmo 3.6, si procede calcolando un fattore P compreso tra 0 e 1 che stima la quantità di luce effettivamente ricevuta. Il fattore P viene usato come semplice fattore moltiplicativo per modulare il colore calcolato come se il frammento fosse completamente illuminato. Il fattore moltiplicativo viene applicato ai soli termini diffusivi e speculari dell'illuminazione del frammento (non avrebbe senso modulare il termine ambientale o quello emissivo perché questi non dipendono dall'occlusione della luce ricevuta per la presenza di altri oggetti).

Il fattore P si calcola chiamando una funzione apposita scritta in un diverso fragment shader (che viene compilato insieme a quelli visti nelle precedenti sezioni per formare un singolo fragment shader executable). I fragment shader visti precedentemente andranno modificati per chiamare questa funzione; per dare un esempio possiamo mostrare le modifiche da apportare al fragment shader per illuminazione diretta basata sul solo materiale:

```
...
float ComputeShadows(); // function declaration
...

void main(void) {
    float prob = ComputeShadows();
    ...
    vec3 fragColorRGB = matEmission;
    ...
    for(int i = 0; i < numActiveLights; i++) {
        ...
        fragColorRGB += matAmbient*lightAmbient[i];
        fragColorRGB +=
            matDiffuse*lightDiffuse[i]*clamp(dot(N,lightObj),0.0,1.0)*prob;
        if((matSpecular != vec3(0.0,0.0,0.0)) && (matShininess != 0.0))
            fragColorRGB += matSpecular*lightSpecular[i]*
                pow(clamp(dot(eyeObj,reflect(-lightObj,N)),0.0,1.0),
                matShininess)*prob;
    }
    fragColor *= vec4(fragColorRGB,1.0);
}
```

La variabile `prob` è quindi il fattore indicato con P nell'algoritmo 3.6. Per completare la realizzazione dell'algoritmo EVSM bisogna semplicemente scrivere questa funzione per stimare la quantità di luce ricevuta.

Nel caso semplificato di una singola sorgente d'ombra, possiamo scrivere il nuovo fragment shader contenente la funzione `ComputeShadows()` come segue:

```
#version 330 core // GLSL version 3.30, core OpenGL profile

uniform sampler2D shaSampler; // shadow map sampler
uniform mat4 shaModelView; // view(from_source)*model matrix
uniform mat4 shaProj; // shadow source proj matrix (modified for sampling)
uniform float shaMinVariance; // minimum variance (algorithm parameter)
uniform float shaIntensity; // shadow intensity (algorithm parameter)
uniform float shaBleedRedFactor; // bleeding reduction (algorithm parameter)
uniform float shaDepthScale; // depth scale (algorithm parameter)
uniform float shazFar; // shadow source far clipping plane distance
uniform float shazNear; // shadow source near clipping plane distance

smooth in vec3 fragPositionObj; // fragment position in obj space

// computes Chebyshev's inequality
float ChebyshevUpperBound(vec2 moments, float t, float minVariance) {
    float p;
    if(t<=moments.x)
        p = 1.0;
```

```
    else p = 0.0;
    float variance = moments.y - moments.x*moments.x;
    variance = max(variance, minVariance);
    float d = t-moments.x;
    float p_max = variance/(variance+d*d);
    return max(p,p_max);
}

// estimates the quantity of received light
float ComputeShadows() {
    float shadowContrib;
    vec3 sourceSpacePos =
        (shaModelView*vec4(fragPositionObj,1.0)).xyz;
    float d = length(sourceSpacePos);
    vec4 shaMapCoord = (shaProj*vec4(sourceSpacePos,1.0));
    shaMapCoord = shaMapCoord/shaMapCoord.w;
    if( shaMapCoord.z < 0.0 || d > shazFar ||
        shaMapCoord.x < 0.0 || shaMapCoord.x > 1.0 ||
        shaMapCoord.y < 0.0 || shaMapCoord.y > 1.0 )
        shadowContrib = 1.0;
    else {
        d = (d-shazNear)/(shazFar-shazNear);
        float negDepth = -exp( -shaDepthScale * d );
        float posDepth = exp( shaDepthScale * d );
        vec4 moments =
            texture(shaSampler, vec2(shaMapCoord.x, shaMapCoord.y));
        float negDepthScale = shaDepthScale * negDepth;
        float negMinVariance =
            (shaMinVariance * negDepthScale) * negDepthScale;
        float negShadowContrib = ChebyshevUpperBound(
            vec2(moments[0], moments[1]),
            negDepth,
            negMinVariance );
        float posDepthScale = shaDepthScale[vr3lib_i] * posDepth;
        float posMinVariance =
            (shaMinVariance * posDepthScale) * posDepthScale;
        float posShadowContrib = ChebyshevUpperBound(
            vec2(moments[2],moments[3]),
            posDepth,
            posMinVariance );
        shadowContrib = min(posShadowContrib, negShadowContrib);
        shadowContrib = clamp((shadowContrib-shaBleedRedFactor[vr3lib_i])/
            (1.0-shaBleedRedFactor),0.0,1.0);
        shadowContrib = shadowContrib*(shaIntensity[vr3lib_i])+
            (1.0-shaIntensity);
    }
    return shadowContrib;
}
```

dove notiamo che:

- Viene effettuato un controllo per verificare se il frammento in esame è interno

al volume di vista della shadow source e se la sua distanza dalla stessa è minore o uguale a `shazFar`; in caso contrario il frammento risulterà completamente illuminato perché si impone `shadowContrib` $= 1$.

- La matrice di proiezione utilizzata (`shaProj`) è una versione modificata della matrice di proiezione usata durante il rendering della shadow map: nel nostro caso le coordinate trasformate (dopo aver subito la perspective division) devono venire utilizzate per accedere alla shadow map e dunque almeno le componenti (x, y) devono venir rimappate dall'intervallo $[-1, 1]$ all'intevallo $[0, 1]$ (nel nostro caso anche la componente z viene alterata).

- L'algoritmo produce come migliore approssimazione della quantità di luce ricevuta il minimo delle stime ottenute col warp positivo e negativo, come previsto dall'algoritmo 3.6 (EVSM).

- Vi sono alcuni parametri di cui non abbiamo ancora parlato nell'algoritmo implementato da questo fragment shader, in particolare abbiamo:

  - `shaBleedRedFactor`
    Viene utilizzato prima di restituire il fattore moltiplicativo calcolato al fine di ridurre il light bleeding che potrebbe essere troppo evidente anche con la tecnica EVSM. La riduzione dell'artefatto per mezzo di questo parametro peggiora la qualità della penombra e dunque il suo utilizzo (se possibile) andrebbe evitato lasciandone a 0 il valore.

  - `shaIntensity`
    Determina quanto intense sono le ombre generate, ossia quanto un frammento viene inscurito se completamente in ombra.

  - `shaMinVariance`
    Minimo valore ammissibile per la varianza; è necessario impostare un valore molto piccolo per questo parametro per evitare problemi di shadow acne (come visto nel par. 3.3.3). Il parametro viene trattato in modo diverso per il warp positivo e negativo, ma non entriamo nei dettagli.

4.8 Altri Shader

Gli shader visti per la realizzazione delle tecniche presentate nei precedenti capitoli (che sono una semplificazione di quelli effettivamente integrati nella VR3Lib) sono

solo un sottoinsieme di tutti gli shader che sono stati scritti e inseriti nella nuova libreria per implementare funzionalità built-in.

Non esaminiamo la struttura di tutti gli shader che sono stati scritti per la realizzazione di ulteriori funzionalità, tuttavia mettiamo in evidenza che nella VR3Lib sono stati integrati anche degli shader per:

- disegnare uno sfondo caricando un'immagine specificata dall'utente (sfondo statico);
- disegnare la skybox (tecnica vista nel par. 2.1.5), intesa come un tipo particolare di sfondo;
- disegnare del testo specificato dall'utente, sia come entità nello spazio tridimensionale che come semplice messaggio di testo in una data posizione della finestra OpenGL.

Tutti questi shader aggiuntivi, insieme a quelli visti precedentemente, sono inseriti nella nuova libreria come stringhe costanti contenenti frammenti di codice GLSL. Il corpo completo degli shader viene costruito a tempo di esecuzione sulla base delle operazioni richieste dall'utente. Gli shader completi di tutti il codice sorgente vengono poi compilati e collegati per formare degli shader program da usare nella pipeline OpenGL. Questa compilazione a tempo di esecuzione può avere un certo peso sul tempo di startup dell'applicazione. Un approccio alternativo sarebbe quello di mantenere una cache di shader precompilati da utilizzare ad ogni esecuzione (che viene riempita quando si compilano nuovi shader), vedi capitolo 7.

Capitolo 5

Struttura e Funzionamento della VR3Lib

Nel capitolo 4 abbiamo visto come realizzare le tecniche di rendering esaminate in precedenza tramite l'utilizzo di shader scritti in linguaggio GLSL; la struttura vista per questi shader è simile a quella presente effettivamente all'interno della nuova libreria.

In questo capitolo esaminiamo come la VR3Lib è stata organizzata in moduli e classi. Faremo occasionali riferimenti alla struttura della precedente libreria per mettere in evidenza le differenze con la nuova versione. È necessario tenere presente che l'obiettivo principale della libreria realizzata durante il lavoro di tesi è quello di riuscire a renderizzare scene ad elevato livello di realismo dove gli oggetti si comportino in modo fisicamente plausibile. Non ci si propone dunque di reimplementare tutte le funzionalità proprie della vecchia VRLib (che aveva come obiettivo principale un'elevata flessibilità).

La libreria risultante fornisce all'utente un'interfaccia molto semplice simile a quella della VRLib, e anche le funzionalità aggiunte che non erano previste nella vecchia libreria sono state inserite mantenendo il paradigma di utilizzo della precedente versione.

In analogia con la precedente VRLib, la nuova libreria assume che ogni utilizzo da parte dell'utente avvenga solamente dopo che la finestra dell'applicazione è stata creata e il contesto OpenGL allocato. Sotto queste condizioni, tipicamente si procede inizializzando la libreria ed eseguendo alcune operazioni di startup (come caricare le mesh per gli oggetti virtuali), per poi iniziare il ciclo principale dell'applicazione che esegue un rendering continuo e si occupa di gestire l'input da parte dell'utente. La gestione del flusso di esecuzione dell'applicazione non è a carico della libreria VR3Lib. Si possono sfruttare delle librerie per impostare questo tipo di esecuzione

tramite una serie di callback, per avere una gestione appropriata dei comandi dell'utente e per inizializzare la finestra e il contesto OpenGL: un esempio è la libreria open source e cross-platform *freeglut* (progettata per sistemi Windows e Unix-like).

Un'applicazione minimale che sfrutta la libreria VR3Lib potrebbe avere la seguente forma:

```
VR3Scene* scene; // reference to a scene
VR3Obj* obj; // reference to an object

// Executed at startup
OnInit() {
    VR3Init(); // initialize VR3Lib

    scene = new VR3Scene(WIDTH,HEIGHT,60.0f); // create a scene
    obj = new VR3Obj("obj.AAM"); // create an object
    obj->SetPosition(0.0f, 0.0f, -2.0f); // set the object position
}

// Executed once per frame
OnFrame() {
    scene->Begin(); // begin scene rendering
    obj->Draw(); // draw the object
    scene->End(); // end scene rendering
}

// Executed on exit
OnExit() {
    delete obj; // destroy the object
    delete scene; // destroy the scene

    VR3Deinit(); // deinitialize VR3Lib
}
```

dove si nota la semplicità dell'interfaccia esportata verso l'utente. L'applicazione minimale di cui sopra non fa altro che creare un oggetto virtuale `obj` caricando delle informazioni da un file 'obj.AAM' e disegnarlo ogni frame davanti alla telecamera a distanza di due unità (di default infatti la telecamera è nell'origine e guarda verso l'asse negativo delle z).

5.1 Librerie Utilizzate dalla VR3Lib

I puntatori a funzione necessari per effettuare con successo le chiamate OpenGL vengono gestiti nella VR3Lib tramite l'utilizzo della libreria open source e cross-platform *GLEW* (*OpenGL Extension Wrangler Library*), questa si occupa di fornire

alla VR3Lib tutti gli entry point per le chiamate OpenGL interrogando opportunamente il display driver e gestisce anche le chiamate relative alle estensioni disponibili sulla particolare macchina. Nella libreria VRLib invece non si usava alcun modulo esterno per la gestione delle chiamate OpenGL e delle estensioni (l'interrogazione del display driver avveniva direttamente ad opera della VRLib).

Una libreria da sfruttare per il rendering di scene virtuali deve essere in grado di caricare correttamente immagini memorizzate in file esterni con una certa varietà di formati (ad esempio per la costruzione di texture). La VRLib sfrutta per le sue operazioni sulle immagini la libreria cross-platform *PaintLib* (contenente solamente le funzionalità essenziali) la quale ha il vantaggio di essere estremamente semplice e lightweight[16], con la conseguenza che sfruttando tale libreria siamo in grado di mantenere le dimensioni della VRLib entro limiti accettabili. Le dimensioni sono un parametro fondamentale perché si ricorda che le VRLib sono il componente principale del motore XVR, e, siccome questo può venire scaricato dinamicamente da un server per eseguire applicazioni tridimensionali interattive online, è necessario mantenere delle dimensioni contenute. Nonostante la VR3Lib sia stata scritta come libreria generica per applicazioni di realtà virtuale ad elevato livello di realismo, in futuro si prevede di integrare la nuova libreria all'interno del sistema XVR: dunque il parametro delle dimensioni è importante anche per la nuova libreria.

Nella VR3Lib si è deciso di non sfruttare più i servizi offerti dalla PaintLib (la cui manutenzione è cessata nel 2005), ma di affidarsi invece alla libreria cross-platform *DevIL* (*Developer's Image Library*) che risulta più potente, supporta una grande varietà di formati e viene ancora mantenuta. La libreria DevIL può venire configurata a tempo di compilazione per integrare solamente un sottoinsieme delle funzionalità e possiamo restringere a piacimento i formati dei file da supportare. Grazie a questa personalizzazione delle funzioni esportate dalla libreria, si riescono ad ottenere delle dimensioni molto ridotte e come vantaggio ulteriore è possibile semplicemente ricompilare la libreria qualora si vogliano aggiungere delle funzionalità precedentemente non supportate. Sfruttando DevIL è inoltre possibile caricare immagini ad elevato range dinamico memorizzate nel formato Radiance RGBE (par. 2.1.5), ma queste non sono supportate da PaintLib.

Ricordando che la nuova libreria integra anche un supporto alla simulazione fisica in tempo reale di corpi rigidi, si sfruttano i servizi del motore fisico Nvidia PhysX che si occupa di gestire tutti gli aspetti della simulazione (compresa la generazione di

[16] Un software si dice 'lightweight' quando le risorse di calcolo richieste per la sua esecuzione sono molto contenute.

flussi di esecuzione appositi). Le precedenti librerie sono state scelte cross-platform per facilitare l'estensione della VR3Lib a sistemi Unix-like, ma purtroppo il PhysX SDK[17] (che contiene le librerie dinamiche che costituiscono il motore fisico) è disponibile nelle ultime versioni soltanto per sistemi Windows (e noi utilizziamo la versione più recente al momento della stesura, la 2.8.4).

Sia GLEW che DevIL sono state compilate generando librerie statiche che sono state integrate nella VR3Lib a tempo di compilazione (in un unico file). Le librerie PhysX purtroppo sono disponibili gratuitamente solamente nella versione precompilata (un insieme di file .dll che implementano le funzionalità del motore). Queste librerie dinamiche dovranno venire distribuite insieme al motore XVR per supportare la simulazione fisica tramite PhysX (in particolare si tratta di 3 file .dll per un totale di circa 4 MB); naturalmente i file saranno necessari solamente quando le funzionalità di simulazione fisica vengono effettivamente utilizzate.

Visto che PhysX non è disponibile su sistemi non Windows, opportuni flag di compilazione sono stati previsti per disabilitare completamente la porzione della VR3Lib relativa alla simulazione fisica, in modo tale da facilitare l'estensione della libreria a sistemi Unix-like.

5.2 La Scena e gli Oggetti Virtuali

In questa sezione vediamo come viene rappresentata la scena all'interno della VR3Lib e come vengono organizzati gli oggetti virtuali all'interno di essa. Per facilitare la comprensione della struttura, forniremo dei diagrammi che illustrano le classi in notazione UML (Unified Modeling Language) sia in questa che nelle successive sezioni[18]. In tali diagrammi nasconderemo tutte le funzioni membro e i membri dati privati allo scopo di renderne più agevole la lettura; inoltre le definizioni di attributi e operazioni verranno semplificate senza esplicitare il tipo dei membri dati, il tipo dei valori di ritorno e i parametri delle funzioni. Alcuni membri dati privati vengono mostrati in questi diagrammi perché definiscono delle associazioni a navigabilità unidirezionale tra classi che vogliamo mettere in evidenza, oppure perché risultano in altro modo rilevanti. Quando la molteplicità viene omessa agli estremi dei segmenti che identificano le relazioni, si intende che essa ha valore unitario.

[17] SDK sta per 'Software Development Kit'; in generale identifica un insieme di strumenti da utilizzare per realizzare un'applicazione che sfrutta determinati software esterni (come PhysX) o che lavora su particolari piattaforme.

[18] I diagrammi sono stati realizzati con il tool gratuito BOUML, reperibile online.

La classe `VR3Scene` corrisponde alla vecchia classe `VRScene` della VRLib, essa rappresenta una scena virtuale (all'interno del quale vengono posizionati oggetti virtuali) e gestisce una serie di parametri di rendering dell'ambiente virtuale:

- le sorgenti luminose presenti nella scena (istanze della classe `VR3Light`) usate nel caso di illuminazione diretta,
- le telecamere presenti nella scena (istanze della classe `VR3Camera`),
- l'eventuale presenza di uno sfondo statico o di una skybox nella scena,
- le cube environment map utilizzate nel caso di illuminazione basata su immagini,
- le opzioni di rendering come il backface culling e il *view frustum culling*[19].

Quindi una scena definisce un set di caratteristiche di base dell'ambiente virtuale. Un oggetto può venire disegnato nel corso di un frame solo una volta che si è deciso in quale scena si sta renderizzando. Per cominciare il rendering di una scena si utilizza una funzione come la `VR3Scene::Begin()` e per terminarlo si utilizza la `VR3Scene::End()`: una coppia di chiamate come quelle sopra saranno eseguite ad ogni frame (blocco `Begin()...End()`).

È importante notare che ad una scena non sono associati gli oggetti che vengono disegnati all'interno di essa. In effetti la scena non rappresenta un ambiente virtuale completo, ma solamente alcune caratteristiche dello stesso (illuminazione, punto di vista, sfondo, ecc...) indipendenti dagli oggetti che poi verranno effettivamente disegnati.

Un'istanza della classe `VR3Scene` rappresenta allora una scena col significato definito sopra. È possibile gestire simultaneamente diverse scene qualora sia necessaria una rapida commutazione tra l'una e l'altra durante il rendering, e in particolare gli stessi oggetti potrebbero venir disegnati all'interno delle diverse scene.

In fig. 5.1 viene mostrata la struttura della classe `VR3Scene` e delle classi `VR3Light` e `VR3Camera` che rappresentano luci e telecamere che la classe `VR3Scene` gestisce.

---

[19] Tecnica che prevede un controllo aggiuntivo sulla visibilità degli oggetti virtuali per evitare qualsiasi chiamata di disegno OpenGL qualora questi risultassero esterni al volume di vista; questi oggetti risulteranno infatti comunque non visibili, ma in assenza di view frustum culling verranno scartati automaticamente solo in fase di primitive clipping (vedi fig. 1.10).

Figura 5.1: Classi `VR3Scene`, `VR3Light` e `VR3Camera`

Va notata la presenza di attributi puntatori privati che connettono un singolo oggetto `VR3Scene` a diverse istanze delle classi `VR3Light` e `VR3Camera`. L'attributo privato `ms_activescene` è statico e identifica la scena correntemente attiva (che sta venendo disegnata perchè siamo dentro un blocco `Begin()...End()` di tale oggetto `VR3Scene`).

Un oggetto virtuale che può venire disegnato in una determinata scena è un'istanza della classe `VR3Obj`. Ad ogni oggetto della classe `VR3Obj` è associata una posizione nel world space, una matrice di rotazione e un vettore per il cambiamento di scala su ognuno dei 3 assi. Un oggetto di questo tipo può allora essere spostato, ruotato e scalato tramite opportune funzioni. Di per sé un oggetto `VR3Obj` rappresenta dunque soltanto un insieme di trasformazioni.

Ad un'istanza della classe `VR3Obj` può venire associata una mesh (oggetto appartenente alla classe `VR3Mesh`). La mesh rappresenta la geometria dell'oggetto virtuale e comprende anche tutte le informazioni sui materiali e le coordinate per applicare le

eventuali texture. Quando un oggetto viene disegnato (tramite la `VR3Obj::Draw()`) in realtà quello che si renderizza è la mesh associata all'oggetto, dove le trasformazioni subite dai vertici sono guidate dai parametri nell'oggetto `VR3Obj`. Ne deriva che a diversi `VR3Obj` può essere associata la stessa mesh (stessi dati geometrici e materiali, ma diverse trasformazioni). Per essere più precisi, ad un singolo oggetto è anche possibile associare diverse mesh: quando l'oggetto viene disegnato si deciderà quale mesh renderizzare sulla base della distanza dalla telecamera. La possibilità di specificare mesh multiple è prevista per associare ad ogni oggetto diversi livelli di dettaglio (*level of detail* o *LOD*, mesh che rappresentano lo stesso oggetto reale ma con diversa complessità geometrica) e sfruttare approssimazioni più grossolane mano a mano che la distanza dal punto di vista aumenta, diminuendo il carico sulla pipeline grafica rimuovendo i dettagli che l'osservatore non può comunque apprezzare sulla base della distanza dall'oggetto. Un oggetto viene tipicamente disegnato tramite il metodo `VR3Obj::Draw()`, il quale provvederà a chiamare il metodo `VR3Mesh::Draw()` della mesh col giusto livello di dettaglio (se almeno una mesh è disponibile).

Gli oggetti della classe `VR3Obj` possono essere combinati in relazioni di parentela dove ad un singolo padre vengono associati 1 o più figli. Quando si disegna un oggetto virtuale la chiamata viene propagata a tutti gli oggetti figli in maniera ricorsiva fino ad arrivare alle foglie (oggetti che non hanno figli). Le operazioni geometriche da applicare ai vertici si cumulano discendendo la gerarchia in modo tale che le trasformazioni specificate per gli oggetti figli siano sempre relative al sistema di riferimento trasformato del padre. Questo meccanismo consente di eseguire complesse trasformazioni grazie ad una catena di oggetti in relazione gerarchica, ed è inoltre possibile prevedere oggetti privi di mesh all'interno della catena di parentela al solo scopo di rappresentare una determinata trasformazione. A tutti gli effetti, gli alberi di istanze della classe `VR3Obj` che si vengono a creare sono delle porzioni dello *scene graph*[20] per la `VR3Lib`. È fondamentale evitare di creare dei cicli all'interno di questo grafo di relazioni gerarchiche, perché in caso di ciclo si continuerebbe a ricorrere durante il disegno arrivando infine alla terminazione erronea dell'applicazione.

Una mesh è un'istanza della classe `VR3Mesh`; questa classe contiene informazioni sulla geometria e sui materiali associati ad un oggetto virtuale, ed è organizzata in un insieme di *subset* (istanze del tipo struttura `VR3Mesh::Subset`) che rappresentano

[20] In generale lo scene graph è la struttura dati che rappresenta l'organizzazione dell'ambiente virtuale da un punto di vista logico e spaziale; tipicamente si tratta di un grafo dove i nodi rappresentano gli oggetti nella scena.

gruppi di triangoli che condividono lo stesso materiale (includendo nel materiale anche le varie texture). Una mesh può quindi avere porzioni di diverso materiale nella VR3Lib. Il vero disegno degli oggetti e le chiamate OpenGL per effettuare il rendering vengono eseguite all'interno della funzione `VR3Mesh::Draw()` che viene normalmente chiamata da un'oggetto `VR3Obj` che sta venendo disegnato quando viene scelta la particolare mesh da renderizzare.

Ad ogni subset all'interno della mesh viene associato un materiale, ossia un'istanza della classe `VR3Material`. Ogni materiale contiene le informazioni per consentire l'applicazione del Phong reflection model e della tecnica di environment mapping (vedi sez. 2.1). Inoltre ad ogni materiale possono venire associate diverse texture sulla base della particolare combinazione di tecniche di rendering che si vuole utilizzare per disegnare i poligoni cui il materiale è applicato:

- diffuse texture (`VR3Material::m_ptex`),
- light map (`VR3Material::m_stex`),
- normal map (`VR3Material::m_ntex`),
- displacement map (`VR3Material::m_dtex`).

La combinazione di tecniche scelte influenza lo shader program utilizzato per renderizzare poligoni a cui è applicato il particolare materiale (come visto nel capitolo 4). Notiamo però che lo shader program dipende anche dal particolare subset oltre che dal materiale ad esso associato perché se il subset non ha i dati necessari per l'applicazione di una certa texture alla mesh lo shader program usato non dovrà sfruttare la texture (anche se l'immagine è fisicamente disponibile nel materiale associato al subset). L'identificatore OpenGL dello shader program utilizzato è memorizzato all'interno del particolare subset, nel membro dati `VR3Mesh::Subset::shader_program`.

Ad ogni particolare tipo di texture associata al materiale (tra i 4 sopra) vengono anche assegnati una serie di parametri per l'applicazione della texture ai poligoni (come le trasformazioni da applicare alle texture coordinates). Tutti questi parametri insieme alla texture (immagine) vera e propria vengono immagazzinati in un oggetto del tipo struttura `VR3Material::MatTexture` all'interno del quale si ha un riferimento all'effettiva immagine da utilizzare (rappresentata da un oggetto di tipo `VR3Texture`).

Un oggetto di tipo `VR3Texture` rappresenta sostanzialmente un'immagine caricata e disponibile al server GL per essere utilizzata come texture. Il caricamento di queste immagini da file esterni avviene mediante la libreria DevIL.

La struttura delle classi sopra descritte è illustrata in fig. 5.2, dove il simbolo '$\oplus$' viene utilizzato in alcune relazioni per indicare che la classe (struttura C++) dal lato opposto al simbolo '$\oplus$' è definita all'interno della classe dal lato del simbolo '$\oplus$'. Va notato che un oggetto `VR3Obj` immagazzina i riferimenti ai suoi figli in un contenitore di tipo `std::set` fornito dalla *STL*[21]; il contenitore scelto deve consentire accesso sequenziale durante il disegno, ma non è richiesta una particolare efficienza nelle operazioni di aggiunta e rimozione di oggetti figli (per cui il contenitore `std::set` risulta idoneo).

Una texture (oggetto di tipo `VR3Texture`) può essere una texture bidimensionale oppure una cube map (e nel caso delle texture associate ai materiali si tratta sempre di texture bidimensionali). Nel diagramma in fig. 5.1 abbiamo trascurato il fatto che la scena gestisce anche lo sfondo statico o la skybox se viene configurata per farlo, ed inoltre amministra le cube environment map necessarie per fare image-based lighting quando richiesto.

Per ottenere un diagramma completo bisogna quindi aggiungere alla fig. 5.1 le relazioni illustrate in fig. 5.3, dove notiamo che la gestione dello sfondo per la scena corrente viene affidato ad una istanza della classe `VR3Background`, facente parte della scena (notare la relazione di composizione). Ad una scena possono allora venire associate 2 ulteriori texture usate per l'illuminazione basata su immagini con l'algoritmo di environment mapping realizzato in sez. 4.5.

La struttura secondo cui sono organizzate le classi sopra presentate è molto simile a quella della VRLib. Come differenze significative possiamo evidenziare che nella precedente versione della libreria non esisteva una classe per rappresentare una sorgente luminosa (in quanto queste venivano gestite direttamente dal server GL), non vi era una classe per la gestione dello sfondo di scena, ed inoltre l'organizzazione delle texture all'interno dei materiali era diversa in quanto non vi era necessità di prevedere 4 diversi strati di texture con significati specifici. La vecchia VRLib non forniva un supporto integrato completo alle tecniche di normal mapping e displacement mapping. Le singole funzioni membro nelle varie classi della VR3Lib sono molto diverse dalla vecchia libreria in alcuni casi perché le operazioni necessarie

[21] STL sta per 'Standard Template Library', una libreria standard contenente alcune funzioni e classi di base utili in una grande varietà di applicazioni (come contenitori di vario tipo).

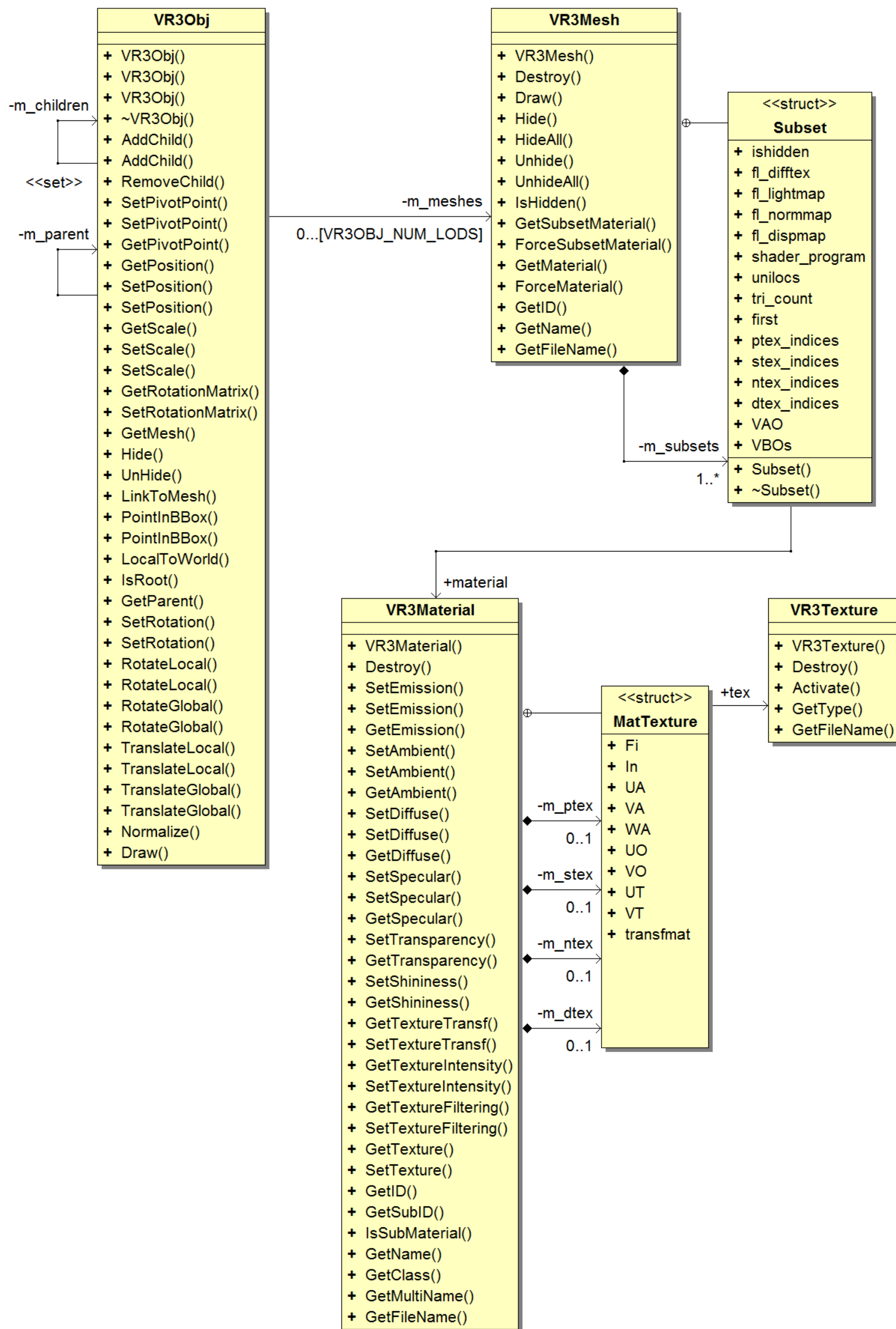


Figura 5.2: Classi `VR3Obj`, `VR3Mesh`, `VR3Material` e `VR3Texture`

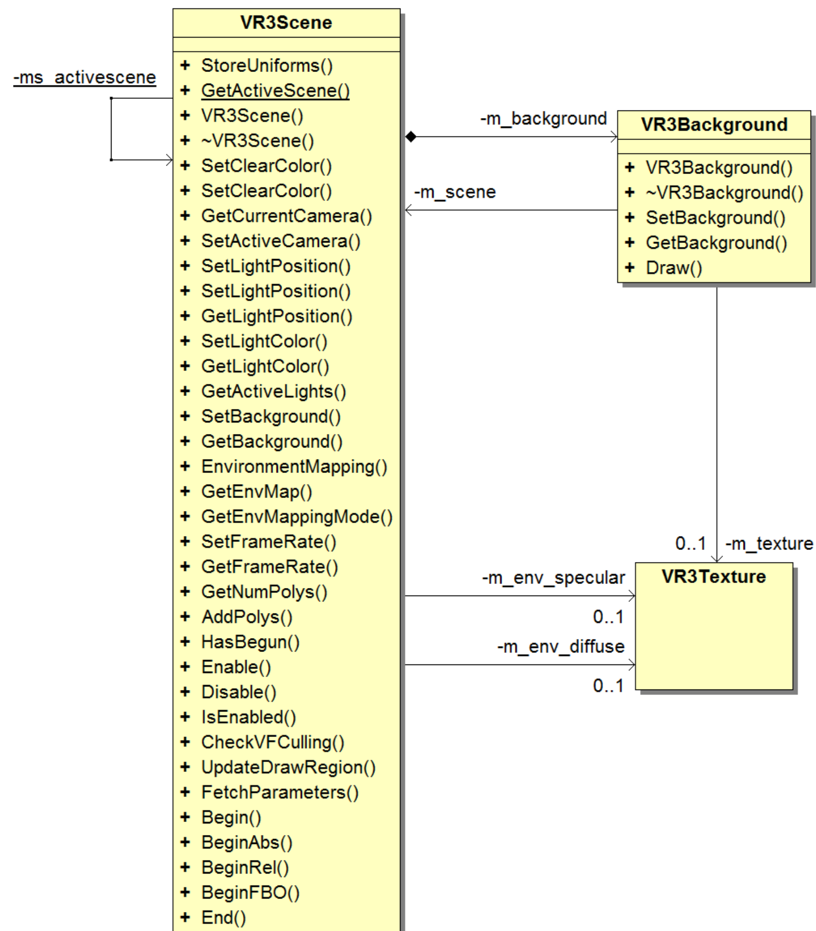


Figura 5.3: Classe `VR3Scene` (ulteriori relazioni)

possono cambiare significativamente.

5.3 Gestori e Replicazione di Dati

Nel presentare l'organizzazione degli oggetti virtuali e della scena abbiamo trascurato un aspetto fondamentale: la gestione delle istanze delle classi. Per capire quale può essere il problema supponiamo di creare due oggetti virtuali (istanze di `VR3Obj`) specificando un file da cui prelevare i dati per l'unica mesh negli oggetti. Se per i due oggetti virtuali viene specificato lo stesso file e la stessa mesh al suo interno, risulta conveniente associare ai due la stessa istanza di `VR3Mesh` per evitare una inutile replicazione di dati (multiple istanze di `VR3Mesh` identiche tra loro).

Questo principio si applica a diverse classi:

- `VR3Mesh`
  È inutile creare diverse mesh se alcuni oggetti virtuali fanno riferimento allo stesso file e alla stessa mesh al suo interno.

- `VR3Material`
  È inutile creare molteplici istanze della classe `VR3Material` quando diverse mesh fanno riferimento allo stesso materiale (i cui dati vengono estratti da un file esterno).

- `VR3Texture`
  È inutile caricare e costruire sul server GL due diverse texture che si riferiscono alla stessa immagine (stesso file): diversi materiali che sfruttano la stessa immagine come texture possono fare riferimento allo stesso oggetto di tipo `VR3Texture`.

Inoltre si applica anche alla classe `VR3PhyMesh` di cui non abbiamo ancora parlato e che verrà introdotta in sez. 5.6; fondamentalmente questa classe incapsula la geometria e il materiale da utilizzare nella simulazione fisica di un oggetto virtuale.

Ad ognuna delle classi sopra è stato quindi assegnato un *gestore*: una classe non istanziabile che fornisce una serie di funzioni membro statiche. Un gestore è semanticamente simile al concetto di namespace, ma vi sono anche dei membri dati statici accessibili alle funzioni membro statiche. Ogni gestore mantiene un insieme di riferimenti a istanze della classe gestita ed è in grado di recuperare un qualunque riferimento sfruttando dei parametri che identificano la particolare istanza (una *chiave*, come il path assoluto del file caricato per le istanze di `VR3Texture`).

Quando si deve ottenere un riferimento ad un oggetto gestito (come una mesh durante la costruzione dell'oggetto virtuale), la prima operazione svolta è l'interrogazione del gestore al fine di scoprire se tale oggetto è già stato caricato (in tal caso si recupera un riferimento e in questo modo si evita una inutile replicazione di dati). Se l'oggetto non è ancora stato creato si costruisce in memoria dinamica e si incarica il relativo gestore di caricarne il contenuto. Il gestore quindi registra il nuovo oggetto nell'insieme di istanze gestite e carica il contenuto sulla base del tipo di oggetto (ad esempio caricherà l'immagine come texture nel caso di `VR3Texture`). I vari gestori sono allora anche dei *caricatori* per istanze della classe gestita.

Visto che un oggetto gestito (ad esempio una istanza di `VR3Mesh`) può essere riferito da molteplici entità (diverse istanze di `VR3Obj`), quando si distrugge un'entità che lo riferisce non è sempre lecito distruggere anche l'oggetto gestito (questo potrebbe ancora servire ad altre entità). Si è allora previsto all'interno di ogni gestore un meccanismo di reference counting che consente di deallocare l'oggetto gestito solamente quando il numero di entità che lo riferiscono scende a 0. Anche l'utente può possedere un riferimento esplicito ad un oggetto gestito (ad esempio un riferimento ad una istanza di `VR3Mesh`) e anche questo tipo di riferimento conta al fine di capire se l'oggetto può venire o meno deallocato. Per facilitare le operazioni con gli oggetti gestiti, viene proibita la chiamata al distruttore da parte delle entità che riferiscono l'oggetto gestito: si fornisce invece una funzione `Destroy()` che sfrutta i servizi del gestore per modificare il contatore di riferimenti ed eventualmente deallocare l'oggetto gestito (se il contatore scende a 0).

La lista dei gestori e le relative classi gestite è riportata di seguito:

- `VR3MeshManager`, gestore di oggetti di tipo `VR3Mesh`;
- `VR3MaterialManager`, gestore di oggetti di tipo `VR3Material`;
- `VR3TextureManager`, gestore di oggetti di tipo `VR3Texture`;
- `VR3PhyMeshManager`, gestore di oggetti di tipo `VR3PhyMesh`.

La chiave utilizzata per identificare una particolare istanza cambia di gestore in gestore; per esempio, nel caso di oggetti di tipo `VR3Texture` avremo come chiave il path assoluto del file contenente l'immagine caricata, mentre per identificare un'istanza di `VR3Mesh` useremo il path assoluto del file contenente la mesh in combinazione con un identificatore della mesh all'interno del file stesso.

La struttura dei gestori è riportata in fig. 5.4.

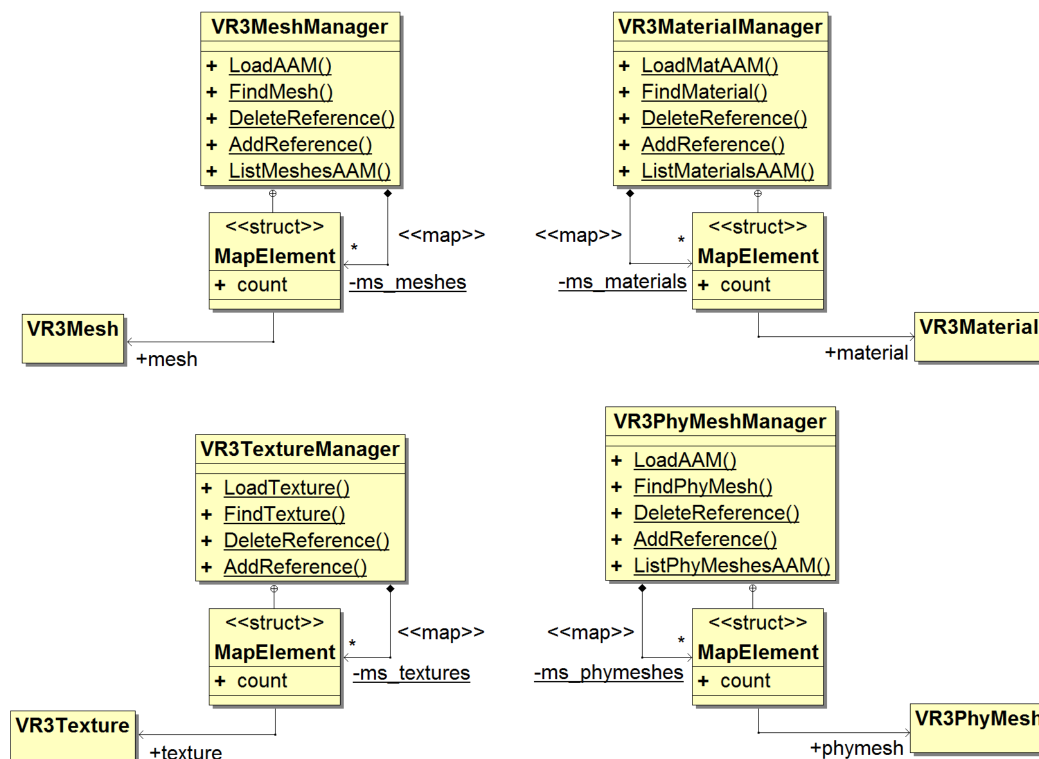


Figura 5.4: Gestori nella VR3Lib

Notiamo che si utilizzano dei contenitori associativi di tipo `std::map` (forniti dalla STL) per associare ad ogni chiave un particolare riferimento all'oggetto gestito. Non è richiesta una particolare efficienza nell'accesso a queste strutture associative perchè gli accessi avverranno tipicamente in fase di inizializzazione e in fase di terminazione dell'applicazione, ma non durante il ciclo di rendering principale (quindi contenitori di tipo `std::map` risultano idonei).

Alcune funzioni vengono fornite dai gestori (nella forma `List*()`) per consentire all'utente di recuperare una lista di oggetti che si possono caricare da un file specificato. Questo è utile quando un singolo file può contenere diversi oggetti che possono essere caricati singolarmente tramite il gestore (non si applica dunque al caso delle texture dove si carica un intero file che rappresenta un'immagine). Queste funzioni di esplorazione sono attualmente disponibili solamente nel caso di file AAM (vedi sec. 5.10).

Nella precedente versione della VRLib non erano disponibili gestori come quelli presentati in questa sezione: un'approccio simile a quello visto veniva applicato nel caso delle texture, ma non si evitava automaticamente la replicazione dei dati relativi a mesh e materiali.

5.4 Gestione degli Shader

Nel capitolo 4 abbiamo esaminato come realizzare mediante shader GLSL le tecniche presentate nei capitoli 2 e 3. Il codice per gli shader visti è una semplificazione di quello che è stato effettivamente integrato nella VR3Lib; abbiamo inoltre detto che le stringhe di codice GLSL vengono modificate a run time durante la costruzione degli shader program in base alla combinazione di tecniche che bisogna utilizzare per i vari oggetti virtuali.

Non è ancora chiaro il momento in cui questi shader program vengano costruiti; bisogna inoltre ricordare che la VR3Lib è anche in grado di gestire shader esterni scritti dall'utente, ma fino ad ora non si è discusso di come tale funzionalità sia stata realizzata.

La classe `VR3ShaderManager` è una classe non istanziabile che fornisce una serie di funzioni membro statiche e possiede un insieme di membri dati statici. Questa si occupa di gestire tutti gli shader integrati all'interno della VR3Lib che sono stati scritti in una forma simile a quella vista nel capitolo 4: questi frammenti di codice GLSL vengono memorizzati in stringhe costanti e prelevati dal `VR3ShaderManager` quando è necessario costruire un nuovo shader program.

Uno shader program viene costruito dal codice GLSL integrato nella VR3Lib solamente quando effettivamente i suoi servigi vengono richiesti. Supponiamo di cominciare la fase di inzializzazione caricando una mesh contenente un unico subset avente un materiale che contiene come unica texture una normal map. Quando l'oggetto di tipo `VR3Mesh::Subset` viene costruito bisognerà impostare il valore di `VR3Mesh::Subset::shader_program`: solo a quel punto si sfrutteranno le funzionalità del `VR3ShaderManager` causando la costruzione dello shader program per lo shading con normal mapping, senza diffuse texture e senza displacement mapping. Questo tipo di approccio alla creazione degli oggetti solo quando effettivamente necessari si dice *lazy instantiation* ed è una tecnica molto comune per limitare il consumo di risorse di processing.

La classe `VR3ShaderManager` mantiene un insieme di identificatori OpenGL contenenti i nomi sul server GL degli shader program già compilati e collegati. In questo modo se uno stesso shader program viene nuovamente richiesto si evita una seconda costruzione dello stesso (a partire dalle stringhe GLSL) risparmiando tempo e memoria lato server GL.

Naturalmente lo shader manager gestisce non solo gli shader program relativi al rendering degli oggetti virtuali nella scena, ma anche tutti gli altri shader integrati

nella VR3Lib: per il disegno di testo nella scena, per il rendering dello sfondo e per tracciare e filtrare shadow maps con tecnica EVSM. Ovviamente una funzionalità del genere non era prevista nella precedente VRLib in quanto non vi erano shader built-in da gestire in modo ottimale. Nella VR3Lib la classe `VR3ShaderManager` fornisce anche tutti i metodi per la compilazione e il collegamento di shader generici (impiegati naturalmente nella costruzione degli shader program built-in).

Come nella vecchia VRLib, anche nella VR3Lib è stata inserita una classe che rappresenta uno shader program definito dall'utente (la classe `VR3ShaderProgram`). Questo tipo di shader program viene costruito sulla base di codice GLSL fornito dall'utente come stringa o prelevato da un file esterno; la versione di GLSL utilizzata deve essere la 3.30 (infatti la definizione del preprocessore '`#version`' viene aggiunta dalla VR3Lib e dunque l'utente deve necessariamente adattarsi a questa versione). Uno shader program costruito da codice fornito dall'utente viene incapsulato in un oggetto di tipo `VR3ShaderProgram`, e l'utente può attivare per lo shading questo shader program chiamando la `VR3ShaderProgram::Start()` e disattivarlo chiamando la `VR3ShaderProgram::Stop()`. Se uno shader program esterno è attivo durante il disegno di una mesh, tale programma verrà utilizzato al posto di quelli di default per i vari subset (che vengono costruiti sulla base degli shader built-in). Il riferimento allo shader program esterno correntemente attivo viene mantenuto in un membro dati statico della classe `VR3ShaderProgram` (con la stessa struttura con cui un membro statico della classe `VR3Scene` mantiene un riferimento alla scena correntemente attiva).

Tramite la classe `VR3ShaderProgram` è possibile anche impostare il valore delle varie variabili uniform che l'utente può aver previsto negli shader forniti (particolari variabili uniform predefinite nella VR3Lib verranno impostate con valori forniti in automatico dalla libreria, ad esempio i sampler per accedere alle texture associate al materiale). Gli ingressi al vertex shader definito dall'utente dovranno avere una forma nota simile a quella degli shader built-in (in questo modo riceveranno i dati relativi ai vertici come accade per gli shader visti nel capitolo 4). Le uscite del fragment shader dovranno avere una particolare forma per andare a scrivere il risultato nei giusti buffer di colore (vedere la documentazione per maggiori dettagli). Nella vecchia VRLib è presente la classe `VRShaderProgram` con funzionalità molto simili, ma la classe `VR3ShaderProgram` estende il supporto anche ai geometry shader scritti dall'utente.

Per facilitare operazioni di rendering complesse (eseguite magari tramite shader definiti dall'utente), nelle VRLib e nelle VR3Lib è prevista una classe `VR3FBO` che

incapsula la gestione di un FBO (framebuffer object) per il rendering off-screen. Il concetto di FBO è stato introdotto in sez. 4.7. Rispetto alla classe `VRFbo` presente nella vecchia VRLib, la nuova classe riesce a gestire rendering in diversi color buffer contemporaneamente e ognuno di questi potrà venire utilizzato come texture in un successivo passo di rendering.

La struttura delle classi di cui abbiamo discusso in questa sezione è illustrata in fig. 5.5.

Figura 5.5: Classi `VR3ShaderProgram`, `VR3ShaderManager` e `VR3FBO`

In figura vengono mostrati i membri dati privati e statici della classe `VR3ShaderManager` che contengono gli identificatori di tutti i programmi caricati. Ognuno di questi membri dati ha una particolare struttura idonea ad immagazzinare i nomi lato server GL di un certo tipo di shader program built-in nella nuova VR3Lib:

- `ms_std_programs`
  Immagazzina gli identificatori degli shader program costruiti per fare rendering degli oggetti virtuali in dipendenza dal tipo di materiale ad essi applicato: in

questo membro dati si può inserire uno shader program per ogni combinazione legale di texture applicate al materiale.

- `ms_bg_programs`
  Immagazzina gli identificatori degli shader program costruiti per disegnare lo sfondo della scena: in questo membro dati si può inserire uno shader program per ogni tipo di sfondo supportato (attualmente i tipi legali sono 2: immagine bidimensionale e skybox).

- `ms_shad_program`
  Memorizza l'identificatore dello shader program utilizzato per disegnare la versione non filtrata dello shadow map all'interno di un FBO previsto per lo scopo (vedi par. 4.7.1).

- `ms_blur_programs`
  Memorizza la coppia di identificatori lato server GL degli shader program necessari per fare il filtraggio offline della shadow map prima di arrivare al rendering effettivo della scena (vedi par. 4.7.2).

- `ms_text_program`
  Memorizza l'identificatore dello shader program utilizzato per disegnare del testo specificato dall'utente all'interno della scena.

In ogni caso tutti questi shader program vengono costruiti solamente quando il loro utilizzo diventa necessario (come detto sopra).

5.5 Proiezione di Ombre: la Classe `VR3ShadowController`

Una nuova funzionalità inserita nella VR3Lib che non era prevista nella vecchia VRLib è il supporto integrato alla generazione di ombre proiettate da oggetti virtuali. Abbiamo discusso di come ottenere questo effetto da un punto di vista teorico nel capitolo 3 e abbiamo visto una possibile realizzazione dell'algoritmo EVSM basata su shader GLSL in sez. 4.7. Gli shader visti nel capitolo 4 sono una semplificazione di quelli effettivamente costruiti per la proiezione di ombre nella VR3Lib, ma la struttura è simile e in ogni caso gli shader program che ne derivano vengono gestiti dal `VR3ShaderManager`.

Per attivare e configurare il servizio di shadow mapping, l'utente della VR3Lib deve creare un'istanza della classe `VR3ShadowController` e renderla attiva. Fintanto che nessun oggetto di tipo `VR3ShadowController` risulta attivo, non si ha la proiezione di ombre nella scena e il rendering avviene in un singolo passo guidato dalle chiamate di disegno svolte dall'utente all'interno di un blocco `Begin()...End()` per la particolare scena. La creazione di uno shadow controller corrisponde anche alla sua attivazione (si rende tale controllore quello attivo). Un'applicazione può gestire diversi controllori durante il rendering, ma solamente uno di essi sarà attivo ad ogni istante.

Lo shadow controller attivo determina i parametri di scena che riguardano la proiezione delle ombre, in particolare:

- Definisce l'insieme degli oggetti (istanze di `VR3Obj`) che proiettano ombre (*casters*);
- Definisce l'insieme degli oggetti (istanze di `VR3Obj`) che ricevono ombre proiettate (*receivers*);
- Definisce un insieme di sorgenti d'ombra, e per ognuna di esse un insieme di parametri per la generazione delle ombre proiettate (per ogni sorgente d'ombra attiva verrà generata una shadow map da utilizzare durante il rendering della scena).

Visto come realizzare l'algoritmo EVSM in sez. 4.7, si pone subito il problema di dover effettuare due passi di rendering per tutti gli oggetti che devono proiettare ombre: prima di tutto bisogna disegnare tali oggetti all'interno delle shadow map (una per ogni sorgente d'ombra), dopo di che bisogna effettivamente renderizzarli nella finestra durante il passo di rendering della scena. Per facilitare le operazioni dell'utente, vorremmo evitare una chiamata esplicita alla `VR3Obj::Draw()` per ogni oggetto che deve essere disegnato all'interno di una shadow map. La soluzione adottata è quindi quella di costruire un oggetto shadow controller che si occupi di effettuare automaticamente i passi di rendering addizionali all'inizio del rendering della scena (alla chiamata di una funzione `Begin()` tra quelle a disposizione della scena). Questo procedimento garantisce trasparenza in fase di rendering, ma è necessario sapere quali oggetti vanno disegnati all'interno delle shadow map prima della eventuale invocazione della funzione `VR3Obj::Draw()` da parte dell'utente nel blocco `Begin()...End()`; per questo motivo ogni shadow controller mantiene l'insieme degli oggetti caster.

Volendo utilizzare tutti gli oggetti della scena sia come caster che come receiver, la situazione tipica sarà quella in cui ogni oggetto definito fa parte sia dell'insieme dei caster che di quello dei receiver. In effetti quando si crea un oggetto `VR3ShadowController` è possibile configurarlo come '`automatic`' e in tal caso la gestione dei due insiemi di cui sopra avviene automaticamente disegnando nella shadow map tutti gli oggetti come verrebbero renderizzati nella scena chiamando la `Draw()` su tutti gli oggetti senza padre (dal punto di vista della sorgente d'ombra). In modalità automatica dunque gli insiemi di caster e receiver verranno automaticamente riempiti con tutti gli oggetti non figli di altri oggetti definiti dopo la creazione dello shadow controller. I figli vengono intesi come estensioni dell'oggetto padre quando disegnati come conseguenza del disegno del padre: quando un oggetto è un caster o un receiver, anche i suoi figli si comporteranno come tali durante il suo disegno.

Un oggetto `VR3ShadowController` può anche venir configurato in modalità '`manual`' e in tal caso l'utente dovrà specificare singolarmente quali oggetti aggiungere all'insieme dei caster e a quello dei receiver. Questo complica l'interfaccia dal punto di vista dell'utente, ma consente una certa flessibilità nell'utilizzo del servizio di shadow mapping che può anche portare ad un miglioramento nelle prestazioni (evitando ad esempio di aggiungere all'insieme dei ricevitori oggetti che sicuramente non verranno investiti da ombre proiettate per come sono posizionati i vari elementi della scena).

Gli insiemi di oggetti caster e receiver vengono quindi utilizzati come segue:

- Insieme di oggetti caster (`VR3ShadowController::m_casters`)
  La funzione membro `Draw()` di questi oggetti verrà invocata una volta per ogni shadow source attiva quando si disegnano le shadow map, la chiamata si propaga agli oggetti figli indipendentemente dal fatto che questi siano o meno nell'insieme caster e quindi anche gli oggetti figli proietteranno un'ombra. In modalità automatica non vi saranno oggetti figli nell'insieme dei caster (normalmente infatti non si desidera disegnare direttamente un oggetto figlio).

- Insieme di oggetti receiver (`VR3ShadowController::m_receivers`)
  Durante il rendering della scena finale, dopo aver prodotto le shadow map filtrata, quando un oggetto viene disegnato, si controlla se fa parte dell'insieme dei ricevitori. In caso affermativo si attiva un flag che viene propagato ai figli causando anche per essi la ricezione di ombre proiettate. In modalità automatica non vi saranno oggetti figli nell'insieme dei receiver: l'aggiunta di un oggetto figlio a questo insieme è infatti utile solamente nel caso in cui

vogliamo disegnare direttamente nella scena un oggetto figlio bypassando il legame di parentela (e normalmente non è così).

Ogni shadow controller gestisce un insieme di sorgenti d'ombra. Ognuna di queste sorgenti può venire attivata e disattivata singolarmente e causa (se attiva all'inizio del rendering) la costruzione e il filtraggio di una EVSM da usare durante il disegno effettivo della scena. All'interno di tutte le shadow map generate in questo modo vi saranno gli oggetti visibili facenti parte dell'insieme dei caster (e relativi figli). Ogni sorgente d'ombra che si vuole utilizzare deve venire inizializzata e attivata prima di produrre effettivamente l'effetto sperato sugli oggetti virtuali. In fase di inizializzazione di una sorgente d'ombra bisogna specificare un certo numero di parametri che possono essere raggruppati come segue:

- *Parametri relativi al volume di vista della sorgente d'ombra*
  Influenzano la regione di spazio entro la quale sarà possibile osservare la proiezione di ombre da parte degli oggetti virtuali correttamente registrati come caster e la ricezione delle ombre da parte dei receiver.

- *Parametri relativi alla texture (shadow map) generata*
  La dimensione della mappa generata in termini di risoluzione è fondamentale per regolare le prestazioni dell'algoritmo: con una mappa troppo piccola si ottengono buone prestazioni ma le ombre possono sembrare troppo spigolose o tremolanti (flickering), mentre con una mappa troppo grande il tempo necessario per il filtraggio può risultare inaccettabile.

- *Parametri relativi all'algoritmo EVSM*
  Con questo insieme di parametri si configura l'algoritmo per ottenere ombre più o meno realistiche, e si può avere una notevole influenza sulle prestazioni. Questi parametri sono stati presentati in sez. 4.7 e si possono riassumere nella lista che segue:

  - Filter size (dimensione del filtro gaussiano utilizzato).
  - Intensity (intensità dell'ombra generata)
  - Minimum variance (varianza minima della tecnica EVSM)
  - Bleeding reduction factor (fattore di riduzione del light bleeding)
  - Depth scale (costante k applicata all'esponente, vedi algoritmo 3.6)

Naturalmente è possibile alterare i parametri di una sorgente d'ombra anche dopo l'inizializzazione (ad esempio se si vuole cambiare il raggio di filtraggio del filtro di blur gaussiano) utilizzando opportune funzioni dell'oggetto `VR3ShadowController`.

La classe `VR3ShadowController` gestisce anche una serie di opzioni per il rendering delle shadow map:

- *backface culling* (per poligoni disegnati nella shadow map),
- *shadow source view frustum culling* (per evitare il disegno nella shadow map di oggetti completamente esterni al volume di vista della shadow source, che verrebbero comunque scartati dalla pipeline OpenGL nello stadio di primitive clipping),
- *shadow source culling* (per evitare il disegno di una shadow map e il filtraggio della stessa nel caso in cui gli effetti della sorgente d'ombra corrispondente non fossero visibili in base alla posizione e orientamento del volume di vista dell'osservatore).

La creazione di diversi shadow controller può essere desiderabile qualora l'illuminazione (e di conseguenza le ombre) cambi istantaneamente ad un certo punto durante l'esecuzione dell'applicazione. In tal caso si possono modificare tutti i parametri di proiezione delle ombre in modo molto rapido attivando un diverso shadow controller configurato in precedenza. È importante notare inoltre che alcune sorgenti d'ombra possono essere inizializzate ma mantenute disattive fino a che non sono effettivamente necessarie (in questo modo ci si risparmia il costo di inizializzazione durante il ciclo di rendering, che è piuttosto elevato perché si devono creare i FBO e le texture necessarie per la shadow map e il filtraggio della stessa).

Nel caso in cui ogni frame si componga di diversi blocchi `Begin()...End()` è possibile calcolare le shadow map solamente per il primo blocco e riutilizzarle per il blocchi successivi tramite il flag `VR3BEGFL_KEEPSHADOWMAPS` da fornire alla funzione utilizzata per entrare nel blocco (una delle funzioni `Begin()` della classe `VR3Scene`).

In fig. 5.6 è riportato un diagramma UML che illustra la struttura della classe `VR3ShadowController`.

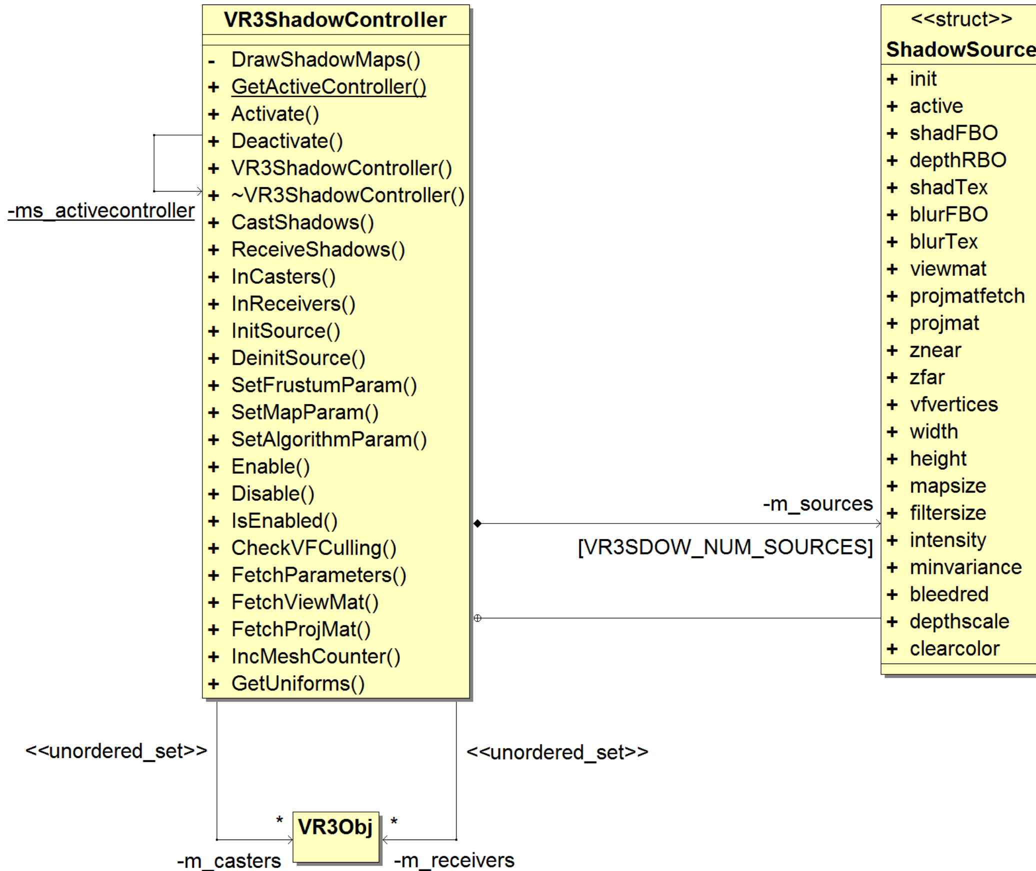


Figura 5.6: Classe `VR3ShadowController`

In figura notiamo la funzione membro privata `DrawShadowMaps()` della classe `VR3ShadowController`. Questa funzione viene chiamata durante l'ingresso in un blocco `Begin()...End()` nel caso in cui vi sia uno shadow controller attivo e causa il disegno e il filtraggio di tutte le shadow map necessarie (una per ogni sorgente d'ombra attiva). Gli insiemi degli oggetti caster e receiver sono membri dati di tipo `std::unordered_set`: questo tipo identifica un contenitore disordinato ad accesso rapido (basato su funzione hash). La funzione hash nel nostro caso opera su dei puntatori; non entriamo nel dettaglio delle sue operazioni, ma è sufficiente sapere che la funzione prevista di default è idonea nel caso di puntatori. Il contenitore `unordered_set` è incluso nel nuovo standard del linguaggio di programmazione C++ '`C++0x`' ed è già disponibile su una grande varietà di sistemi operativi e compilatori. Si utilizza un contenitore ad accesso rapido per l'insieme dei ricevitori perché il controllo di appartenenza di un oggetto a questo insieme avviene durante il rendering. Per l'insieme dei caster invece non sarebbe necessario un contenitore di questo tipo, ma si è scelto di utilizzare comunque un approccio con hashing per consentire

ricerche rapide da parte dell'utente della libreria anche durante il rendering.

5.6 Simulazione Fisica e Interazione con PhysX

Un nuovo servizio che non era presente nella vecchia VRLib è il servizio di simulazione fisica di corpi rigidi integrato all'interno della VR3Lib.

Abbiamo detto che il supporto alla fisica in tempo reale viene realizzato nella nuova libreria appoggiandosi al motore fisico Nvidia PhysX, disponibile nelle ultime versioni solamente sotto Windows. Le informazioni che vengono fornite in questa sezione si applicano solamente quando la VR3Lib viene compilata sotto Windows: se si tratta di un altro sistema operativo le classi che presentiamo adesso vengono estromesse dalla libreria finale e dunque non sono utilizzabili dall'utente della VR3Lib. È possibile escludere dalla compilazione queste classi anche sotto il sistema operativo Windows definendo la macro `VR3CONF_NOPHYSICS`.

La simulazione di corpi rigidi in PhysX avviene sulla base di una geometria definita da una mesh che in generale può essere diversa dalla mesh utilizzata per il disegno degli oggetti virtuali. I motivi per cui è utile differenziare la mesh per la simulazione fisica da quella utilizzata per il disegno sono diversi:

- Visto che la mesh fisica non è visibile, possiamo pensare di semplificarne la struttura rispetto alla mesh per il disegno dell'oggetto se (come spesso accade nella pratica) non ci interessa una simulazione fisicamente accurata. Questo rende più rapida la simulazione consentendoci di gestire scene con interazioni fisiche molto complesse ottenendo comunque risultati plausibili.

- In PhysX non è possibile simulare corpi rigidi rappresentati da mesh di triangoli non convesse se questi devono muoversi. Ne consegue che per simulare oggetti mobili non convessi normalmente si opera costruendo diverse mesh convesse composte tra loro che nell'insieme approssimano la forma del corpo rigido concavo. Si costruisce un composto PhysX associando ad un singolo attore (un oggetto simulato) diverse mesh (in questo caso convesse).

La geometria definita per la simulazione fisica può quindi essere composta anche da diverse mesh convesse, e tutti questi dati vengono immagazzinati in un oggetto di tipo `VR3PhyMesh`. Per quanto riguarda i parametri del materiale fisico utilizzato (coefficiente di attrito statico, coefficiente di attrito dinamico e coefficiente di restitu-

zione[22]), anche questi vengono memorizzati direttamente nell'oggetto `VR3PhyMesh`. Ad ogni mesh fisica è associato un unico materiale fisico (a differenza delle mesh grafiche che nelle VR3Lib possono essere multimateriale) e, visto che i parametri sono fondamentalmente 3 valori reali, non c'è ragione di evitare la replicazione dei dati tramite un gestore di materiali fisici come quello visto per la classe `VR3Material`. È quindi accettabile salvare i parametri del materiale fisico direttamente nell'oggetto `VR3PhyMesh`. Anche la massa dell'oggetto verrà salvata all'interno dell'istanza della classe `VR3PhyMesh`.

I parametri dei materiali vengono definiti per ogni singolo oggetto, ma quando si ha collisione tra diversi oggetti (o strisciamento di un oggetto su un altro) la fisica ci insegna che servirebbe un singolo coefficiente di attrito statico, uno di attrito dinamico e un coefficiente di restituzione dipendenti dai due materiali coinvolti. In PhysX quando due oggetti entrano in contatto si calcolano dei coefficienti da usare come dettato dalle formule fisiche combinando i coefficienti dei materiali assegnati ai due oggetti. Vi sono diversi modi di combinare i coefficienti e nella VR3Lib si fa riferimento alla modalità di default: si mediano i singoli valori per i due materiali in gioco.

All'interno della mesh fisica è anche indicato il tipo di simulazione da effettuare per l'oggetto virtuale, che può essere uno dei seguenti:

- Oggetto *dinamico* (`VR3PHYMESH_DYNAMIC`)
  La simulazione è completa: l'oggetto è influenzato da tutte le forze applicate su di esso e collide con ogni altro oggetto nella scena. In questo caso si usano composti di mesh convesse per approssimare la geometria (arbitraria) dell'oggetto da un punto di vista fisico.

- Oggetto *statico* (`VR3PHYMESH_STATIC`)
  L'oggetto non si muove nella scena e non è influenzato da forze di alcun tipo, si ha collision detection (rilevamento delle collisioni) solamente se l'altro oggetto interessato è dinamico. La geometria utilizzata per la simulazione fisica viene in questo caso rappresentata con una singola mesh di triangoli arbitraria. Nel caso di oggetti statici, PhysX sfrutta algoritmi ottimizzati ed efficienti fondati sul fatto che l'oggetto non si muove, e si può quindi utilizzare direttamente la mesh grafica anche per la simulazione fisica.

---

[22] Il coefficiente di restituzione rappresenta il rapporto tra la velocità ottenuta dopo una collisione e la velocità che si aveva prima della stessa.

- Oggetto *cinematico* (`VR3PHYMESH_KINEMATIC`)
  In questo caso l'oggetto può essere visto come un oggetto dinamico avente una massa infinita, non influenzato dalla gravità o da altre forze che agiscono su di esso e non influenzato dalle collisioni con altri oggetti. L'oggetto può venire spostato dall'utente con l'utilizzo delle funzioni fornite dal simulatore in un blocco `ObjBeginMove()...ObjEndMove()`. Gli oggetti cinematici respingeranno gli oggetti dinamici incontrati, ma non avremo collisioni con altri oggetti cinematici o con oggetti statici. L'utilizzo principale di questo tipo di oggetti consiste nella creazione di corpi inarrestabili che seguono un determinato percorso (con una simulazione all'interno di PhysX molto più semplice rispetto al caso di oggetti dinamici).

Alla classe `VR3PhyMesh` viene associato un gestore `VR3PhyMeshManager` simile a quello previsto per la classe `VR3Mesh` (come visto in sez. 5.3), che si occupa di evitare la creazione di nuove istanze della classe `VR3PhyMesh` se diversi oggetti virtuali fanno riferimento alla stessa mesh fisica. All'interno delle VR3Lib si assume che vi sia una corrispondenza univoca tra la mesh utilizzata da un oggetto per il disegno e la mesh fisica da caricare per simulare l'oggetto all'interno di PhysX; in particolare, si assume che il caricamento della mesh fisica debba avvenire dallo stesso file da cui è stata caricata la mesh grafica. Questo vincolo deriva dal fatto che la corrispondenza esiste sempre nel caso di file AAM (vedi sez. 5.10), ma potrà essere rilassato in future estensioni della libreria per disaccoppiare completamente i due tipi di mesh.

In analogia con la gestione delle ombre, per sfruttare il servizio di simulazione di corpi rigidi tramite PhysX l'utente deve creare un istanza della classe `VR3PhysicsSimulator`. Questo oggetto si occupa di gestire tutti gli aspetti della simulazione fisica e in particolare mantiene i seguenti insiemi di oggetti `VR3Obj`:

- `VR3PhysicsSimulator::m_objects`
  Insieme di oggetti registrati per la simulazione fisica.

- `VR3PhysicsSimulator::m_simobjects`
  Insieme di oggetti che effettivamente stanno venendo simulati nel motore PhysX. Quando si termina il riempimento dell'insieme `m_objects` e si prepara la simulazione con la `VR3PhysicsSimulator::PrepareSimulation()` si ha la copia nell'insieme `m_simobjects` dei riferimenti a tutti gli oggetti registrati per la simulazione. Questo insieme non può venire alterato direttamente dall'utente perché l'accesso diretto è consentito solamente a `m_objects`. L'in-

sieme `m_simobjects` è implementato con un contenitore ad accesso rapido perché dovrà venire esaminato a tempo di rendering.

È possibile creare molteplici simulatori (istanze di `VR3PhysicsSimulator`) con lo scopo di simulare diversi insiemi di oggetti (o diverse scene) in tempi diversi con una veloce commutazione tra gli stessi. Ad ogni istante vi sarà però un unico simulatore attivo che è quello utilizzato per recuperare i dati sugli oggetti simulati (come la posizione e l'orientamento) durante il rendering.

L'utente non deve costruire manualmente degli oggetti `VR3PhyMesh` da passare al simulatore fisico: costui si occuperà di dialogare con il gestore di questi oggetti e costruirà automaticamente tutte le istanze di `VR3PhyMesh` necessarie per preparare la simulazione fisica (nella `PrepareSimulation()`). Ovviamente il simulatore fisico si occuperà anche di distruggere i riferimenti a tutti questi oggetti una volta che avrà terminato le proprie operazioni (con la funzione `VR3PhyMesh::Destroy()`).

Un `VR3PhysicsSimulator` può essere configurato in due modalità:

- `automatic`
  Gli oggetti creati vengono automaticamente registrati per la simulazione col simulatore attivo (se questo è configurato come automatico).

- `manual`
  L'utente dovrà registrare manualmente tutti gli oggetti con cui vuole sfruttare il servizio di simulazione di corpi rigidi.

Quando si hanno relazioni di parentela tra gli oggetti della scena, queste verranno considerate solamente per determinare la forma, la posizione e l'orientamento iniziale degli oggetti. Una volta che la simulazione viene preparata e successivamente avviata (con la `VR3PhysicsSimulator::StartSimulation()`) gli oggetti evolveranno in modo tra loro indipendente anche se in relazione di parentela.

È possibile imporre dei vincoli sull'evoluzione fisica degli oggetti virtuali; questi vincoli possono limitare il movimento relativo di oggetti simulati rimuovendo alcuni gradi di libertà. I vincoli vengono detti *giunti* e possono essere creati tra due oggetti virtuali imparentati o meno oppure tra un oggetto virtuale e il mondo (inteso come oggetto statico implicito). In ogni caso la creazione di giunti è consentita solamente su oggetti dinamici o cinematici. L'esempio più semplice di giunto è quello che rimuove tutti i possibili gradi di libertà tra due oggetti, di fatto fissando la posizione relativa dei due: in questo caso i due oggetti evolveranno come un singolo corpo rigido.

Una volta che si è preparata la simulazione, tutti i parametri che definiscono la posizione e l'orientamento degli oggetti usati durante il rendering della scena vengono letti direttamente dal simulatore fisico che a sua volta li preleva da PhysX. I dati di posizione, orientamento e cambiamento di scala presenti all'interno delle particolari istanze `VR3Obj` (utilizzati per costruire la scena iniziale dal punto di vista fisico nella `PrepareSimulation()`) vengono ignorati durante la simulazione degli oggetti e la modeling matrix viene estratta solamente sulla base dei risultati prodotti da PhysX. Se l'utente vuole operare su oggetti virtuali mentre questi vengono simulati (applicando una forza o spostando brutalmente l'oggetto) dovrà allora sfruttare delle funzioni apposite messe a disposizione dall'oggetto `VR3PhysicsSimulator` che interagiscono direttamente con PhysX, e i cambiamenti non verranno conservati una volta che la simulazione è terminata.

Durante il disegno di gerarchie di parentela, la chiamata alla `Draw()` si propaga come al solito di padre in figlio. Ad ogni chiamata la modeling matrix viene passata di padre in figlio come al solito (per comporre le trasformazioni). La matrice ricevuta dal padre viene ignorata se l'oggetto sta venendo simulato (in tal caso si estrae la modeling matrix direttamente da PhysX), ma verrà invece composta con i parametri nell'oggetto VR3Obj per calcolare la modeling matrix finale se l'oggetto non fa parte del set di oggetti simulati `m_simobjects`.

I calcoli svolti da PhysX per simulare l'evoluzione dei corpi rigidi all'interno della scena vengono svolti in un thread separato rispetto al flusso di esecuzione principale dell'applicazione. Questo ci consente di sfruttare anche processori multi-core in modo molto semplice cercando di parallelizzare le operazioni di rendering grafico e i calcoli fisici per il prossimo frame. La gestione del thread per la simulazione fisica è affidata al motore PhysX. Quello che si fa lato VR3Lib per massimizzare le prestazioni è cominciare il tracciamento di un frame recuperando i risultati della simulazione lanciata all'inizio del frame precedente (se disponibili) e avviarne una nuova (che può dunque procedere in parallelo alle operazioni di rendering sulla pipeline OpenGL se molteplici core sono disponibili). Ogni simulazione effettuata fa procedere l'evoluzione dinamica dei corpi rigidi di un certo tempo che viene calcolato ad ogni frame. I risultati della simulazione fisica per il simulatore attivo vengono recuperati all'inizio di un blocco `Begin()...End()` (se la simulazione era stata effettivamente preparata), in particolare all'interno di una delle funzioni `Begin()` della classe `VR3Scene` tramite la funzione `VR3PhysicsSimulator::CollectResults()`.

Naturalmente un oggetto `VR3PhysicsSimulator` gestisce anche una serie di parametri per il fine tuning della simulazione che possono venire utilizzati per aumentare

la stabilità della stessa o renderla più rapida nel caso di scene complesse. Non esaminiamo in dettaglio tutti questi parametri (riferirsi alla documentazione per maggiori informazioni).

Nel caso in cui ogni frame si componga di diversi blocchi `Begin()...End()`, i risultati della simulazione dovrebbero venire recuperati solo all'inizio del primo blocco e riutilizzati in tutti i blocchi successivi: questo si può fare tramite il flag `VR3BEGFL_KEEPPHYSICS` da fornire alla funzione utilizzata per entrare nel blocco (una delle funzioni `Begin()` della classe `VR3Scene`).

Le classi presentate in questa sezione sono illustrate in fig. 5.7.

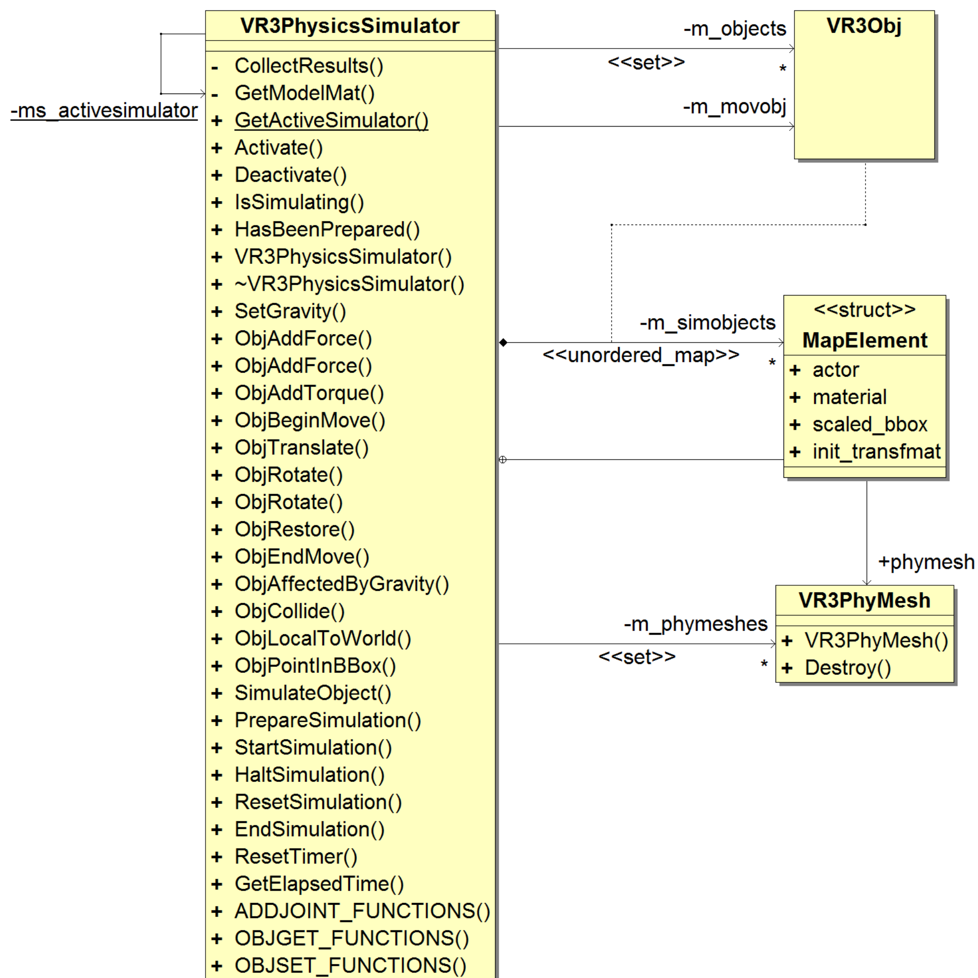


Figura 5.7: Classi `VR3PhysicsSimulator` e `VR3PhyMesh`

Notiamo che:

- Per facilitare la lettura, svariate funzioni membro pubbliche della classe `VR3PhysicsSimulator` sono state nascoste in alcuni gruppi di funzioni

riportati di seguito:

  - `ADDJOINT_FUNCTIONS()`
    Funzioni per aggiungere un giunto tra oggetti simulati o tra un oggetto simulato e il mondo.
  - `OBJGET_FUNCTIONS()`
    Funzioni per recuperare informazioni sulla simulazione fisica relative ad un particolare oggetto simulato.
  - `OBJSET_FUNCTIONS()`
    Funzioni per impostare alcuni parametri di simulazione per lo specifico oggetto simulato.

- Il movimento di oggetti cinematici e dinamici durante la simulazione avviene per mezzo di un blocchi `ObjBeginMove()...ObjEndMove()` all'interno del quale si operano le trasformazioni. Il riferimento all'oggetto che sta venendo spostato viene mantenuto nel membro dati privato `m_movobj` (visibile come associazione in figura).

- Il contenitore `m_phymeshes` viene utilizzato per immagazzinare tutte le mesh fisiche caricate durante le operazioni del simulatore. Questi riferimenti verranno cancellati (tramite la `VR3PhyMesh::Destroy()`) al termine della vita del simulatore.

- Il contenitore `m_simobjects` è un contenitore hash associativo. La chiave utilizzata è il puntatore all'oggetto `VR3Obj` i cui dati si vogliono accedere (questo si è indicato in figura con la linea tratteggiata).

- La classe `VR3PhyMesh` non ha alcuna funzione per leggere o scrivere dei parametri per la simulazione fisica (come la massa dell'oggetto). Questi parametri vengono infatti modificati durante la simulazione operando direttamente con l'oggetto `VR3PhysicsSimulator` attivo (usando le apposite funzioni), che dialoga con PhysX; non si alterano i valori memorizzati negli oggetti di tipo `VR3PhyMesh`. Questo significa che, al termine della simulazione, una nuova simulazione riparte con i valori originali letti durante la costruzione delle istanze di `VR3PhyMesh`. Questo modo di operare consente di diversificare i parametri in diversi simulatori fisici anche se gli oggetti `VR3PhyMesh` riferiti sono gli stessi, ed inoltre è coerente col fatto che ogni volta che una simulazione viene

conclusa tramite la `EndSimulation()` l'insieme `m_simobjects` viene svuotato per essere poi ricostruito alla preparazione di una nuova simulazione (tramite la `PrepareSimulation()`).

- La funzione membro privata `CollectResults()` della classe `VR3PhysicsSimulator` è la funzione che viene utilizzata entrando in un blocco `Begin()...End()` per recuperare i risultati della simulazione prodotti da PhysX. Questa funzione (se la simulazione si è conclusa) aggiorna tutte le informazioni gestite da PhysX come la posizione e l'orientamento dei vari attori nella scena (che identificano gli oggetti simulati). La funzione membro privata `GetModelMat()` viene invece utilizzata durante il rendering dei vari oggetti virtuali e consente loro di recuperare le informazioni di posizione e orientamento se stanno venendo simulati nel motore PhysX.

5.7 Rendering di Testo

Nella VRLib la gestione delle stringhe di testo visualizzate all'interno della scena tridimensionale avviene grazie alla classe `VRText`. L'implementazione di questa classe è molto semplice nella vecchia libreria, perché sfrutta i servizi messi a disposizione dall'interfaccia *WGL* (o *Wiggle*) che rappresenta l'API verso le funzioni necessarie per l'utilizzo di OpenGL su una piattaforma Windows.

L'interfaccia WGL è fortemente legata al sistema operativo Windows. L'utilizzo della stessa è obbligatorio almeno per la creazione del contesto OpenGL sotto Windows (anche utilizzando una libreria cross-platform per questa operazione, le chiamate WGL verranno incapsulate nel codice della libreria se si lavora sotto Windows). Una API simile a WGL esiste per piattaforme che lavorano con il sistema *X11* (o semplicemente *X*) per la gestione delle finestre (come Unix): si tratta dell'interfaccia *GLX* (*OpenGL Extension to the X Window System*). Come anche la precedente libreria VRLib, la VR3Lib è attualmente disponibile solamente per sistemi Windows; si è cercato però di rendere agevole una sua futura estensione a piattaforme diverse (come Linux) sfruttando dei flag di compilazione condizionale basati anche sul sistema operativo e lavorando solamente con librerie disponibili anche per sistemi Unix-like (eccetto PhysX).

Per renderizzare stringhe di testo nel modo più flessibile possibile, purtroppo, è necessario ricorrere a funzioni esterne all'API OpenGL come quelle fornite da WGL nel caso di sistemi Windows. Queste consentono di disegnare stringhe scegliendo

tra svariati font e aggiungendo anche gli effetti tipici degli avanzati editor di testo. I servizi offerti da WGL si appoggiano sull'interfaccia *GDI* (*Graphics Device Interface*), un componente fondamentale di Windows che si occupa degli aspetti di interazione con l'adattatore grafico. GDI viene utilizzato sotto Windows in tutte le applicazioni che devono disegnare qualcosa a schermo, ma fornisce anche un insieme di funzionalità aggiuntive per svariate situazioni.

Quando si deve fare rendering di caratteri tramite una libreria grafica come OpenGL, possiamo identificare diversi tipi di font che si possono ottenere:

- *Bitmap fonts*
  Ogni carattere viene rappresentato mediante un'immagine che viene applicata come texture ad un piano renderizzato per produrre il carattere nell'ambiente virtuale. Questi font sono molto efficienti, ma visto che si usano delle immagini possiamo ottenere dei caratteri visibilmente 'frastagliati' quando si usa una risoluzione troppo bassa (dunque per aumentare le dimensioni della stringa senza rendere l'artefatto troppo evidente è necessario utilizzare una nuova immagine a maggiore risoluzione).

- *Outline fonts*
  Si usano formule matematiche per descrivere la forma di ogni carattere. Per renderizzare in OpenGL un carattere di questo tipo possiamo associargli una mesh che approssima la forma esatta ottenuta mediante le formule matematiche (con un certo numero di poligoni). Questi font possono essere facilmente modellati (ingranditi o ristretti) senza perdere qualità.

- *Stroke fonts*
  Si usano una serie di vertici per definire i singoli tratti usati per tracciare i caratteri. Manipolare questi font è molto più semplice che nel caso degli outline fonts e si ha comunque libertà di scalare e trasformare i caratteri senza perdere qualità.

Sfruttando i servizi offerti da WGL è possibile costruire delle display list per il disegno di stringhe sfruttando bitmap fonts o outline fonts. Queste display list possono dunque essere usate direttamente per ottenere il risultato desiderato, data la stringa da renderizzare.

Un problema che quindi si nota è che con le ultime versioni di OpenGL (a partire dalla 3.1) non è più possibile utilizzare le display list in quanto questi oggetti

sono stati rimossi dalla specifica delle funzioni incluse nel core OpenGL[23]. Per la realizzazione del servizio di disegno di testo all'interno della scena nella VR3Lib si deve ricorrere perciò ad un nuovo approccio.

Si potrebbe pensare di abbandonare completamente l'interfaccia propria del sistema operativo Windows che si basa su GDI e WGL cercando una soluzione completamente cross-platform. A tal proposito esistono svariate librerie che consentono il disegno di testo su diverse piattaforme differenziando le procedure eseguite nei vari casi, ma queste si basano ancora sul concetto di display list che è stato abbandonato da OpenGL nel profilo core. Implementare all'interno della VR3Lib una soluzione cross-platform senza utilizzare funzioni dipendenti dal sistema operativo è possibile (ad esempio sfruttando solo bitmap fonts e distribuendo con la libreria alcune immagini da cui recuperare le texture necessarie per il disegno delle stringhe); tuttavia si otterrebbe poca flessibilità nei caratteri usati per il testo e bisognerebbe occuparci della distribuzione delle immagini aggiuntive. La soluzione migliore risulta quindi quella di sfruttare i servizi che ci vengono messi a disposizione dal sistema operativo, diversificando la soluzione per le diverse piattaforme. Nella versione attuale della nuova libreria il rendering di testo è supportato solamente sotto Windows tramite i servizi offerti dall'interfaccia GDI. Un'estensione a sistemi basati su X11 potrà essere aggiunta in futuro.

VR3Text

+ VR3Text()
+ ~VR3Text()
+ LoadText()
+ UseFont()
+ GetColor()
+ SetColor()
+ SetColor()
+ GetPivotPoint()
+ SetPivotPoint()
+ SetPivotPoint()
+ GetPosition()
+ SetPosition()
+ SetPosition()
+ GetScale()
+ SetScale()
+ SetScale()
+ GetRotationMatrix()
+ SetRotationMatrix()
+ Hide()
+ UnHide()
+ SetRotation()
+ SetRotation()
+ RotateLocal()
+ RotateLocal()
+ RotateGlobal()
+ RotateGlobal()
+ TranslateLocal()
+ TranslateLocal()
+ TranslateGlobal()
+ TranslateGlobal()
+ Draw3D()
+ Draw2D()

Figura 5.8: `VR3Text`

Visto che i servizi per il rendering di caratteri offerti da WGL non possono più essere utilizzati perché basati su display list, nella nuova classe `VR3Text` si sfruttano direttamente le funzioni dell'interfaccia GDI di Windows. Tramite queste funzioni è possibile generare delle immagini per i singoli caratteri utilizzati e poi renderizzarle nella scena applicandole come texture su piani creati appositamente lavorando con semplici shader (ottenendo dunque font di tipo bitmap). Le immagini ottenute da GDI possono contenere caratteri di diversi font con diversi effetti applicati ad essi; inoltre è possibile ottenere immagini di

[23] Naturalmente questo non vale quando si usa OpenGL con un profilo compatibility (il cui supporto da parte delle implementazioni ricordiamo essere opzionale).

diversa risoluzione sulla base della precisione richiesta. Nella classe `VR3Text` sostanzialmente si ha quindi un dialogo con l'interfaccia GDI, che ci fornisce le texture da utilizzare per il rendering.

Nella vecchia VRLib una stringa di testo (oggetto `VRText`) poteva essere disegnata come un oggetto virtuale qualsiasi, e veniva infatti posizionata a piacere nella scena. Nella nuova libreria si è estesa l'interfaccia consentendo trasformazioni complesse simili a quelle possibili per gli oggetti virtuali, e si è inoltre inserita una nuova modalità di disegno del testo aggiuntiva a quella tradizionale:

- *Disegno tradizionale* (`VR3Text::Draw3D()`)
  Il testo viene disegnato come un oggetto qualsiasi. Tutte le trasformazioni specificate vengono applicate in modo simile a quanto avviene nel disegno di un oggetto `VR3Obj` (senza vincoli di parentela). Questa modalità serve quando si vuole disegnare il testo all'interno della scena come se fosse un oggetto virtuale.

- *Disegno bidimensionale* (`VR3Text::Draw2D()`)
  Si considerano solamente le trasformazioni di traslazione e cambiamento di scala per disegnare il testo in un punto preciso della finestra specificato in numero di pixel dall'angolo in basso a sinistra. Questa modalità facilita l'utilizzo della classe `VR3Text` per scrivere messaggi di notifica all'interno della finestra OpenGL.

La struttura della classe `VR3Text` è illustrata nel diagramma UML in fig 5.8.

Volendo estendere il servizio di rendering di stringhe di testo anche a sistemi basati su X11, possiamo diversificare le operazioni di `VR3Text` per sfruttare funzioni proprie di tali sistemi simili a quelle che ci mette a disposizione l'interfaccia GDI di Windows. Notiamo inoltre che con le nuove versioni di OpenGL e l'abbandono delle display list è comunque probabile che le note librerie cross-platform per il rendering di testo verranno presto rinnovate utilizzando i vertex buffer objects durante il disegno[24]; in futuro si potrebbe pertanto pensare di delegare l'intera gestione del disegno di stringhe di testo ad una di queste librerie.

[24] Esempi di queste librerie sono *OGLFT* (OpenGL-FreeType) e *FTGL*, attualmente basate su display list.

5.8 Un Frame con la Libreria VR3Lib

Le classi presentate nelle precedenti sezioni sono sufficienti a comprendere il funzionamento di base della nuova libreria; in questa sezione vediamo come procede il rendering di un frame esaminando le operazioni eseguite all'interno di ogni chiamata principale e osservando come l'unico flusso di esecuzione passa da un oggetto all'altro.

Visto che questa sezione ha lo scopo di chiarire il funzionamento di base della libreria, ci poniamo nel caso più semplice possibile: si vuole disegnare un singolo oggetto `obj` nella scena `scene` e nel corso del frame abbiamo un singolo blocco `Begin()...End()`. Diversi blocchi `Begin()...End()` possono servire quando si vogliono disegnare due scene diverse (o due viste diverse della medesima scena) in porzioni distinte della finestra OpenGL, oppure quando il rendering deve avvenire in diversi passi sfruttando oggetti di tipo `VR3FBO`. Supponiamo di non essere in nessuno dei due casi.

All'interno del ciclo principale dell'applicazione (che si occupa di tracciare un frame ad ogni iterazione) avremo una cosa del tipo:

```
// Executed once per frame
OnFrame() {
    scene->Begin(); // begin scene rendering
    obj->Draw(); // draw the object
    scene->End(); // end scene rendering
}
```

Trascuriamo tutti gli aspetti di inizializzazione della scena e dell'oggetto virtuale `obj`, come di qualsiasi altro componente della libreria che potrebbe essere stato attivato. Le chiamate esaminate potrebbero sfruttare altre funzioni membro della stessa classe, di altre classi o di particolari namespace interni alla VR3Lib per svolgere alcune delle loro operazioni: in questo caso il passaggio non sempre viene messo in evidenza (perché può risultare di scarso interesse in questa trattazione).

L'esecuzione procede come segue:

1. `scene->Begin()`

    (a) Si imposta la trasformazione di viewport eseguita dalla pipeline OpenGL per ricoprire con la nuova immagine generata l'intera finestra dell'applicazione.

(b) Si imposta la scena corrente `scene` come scena attiva (sostituendo il valore in `VR3Scene::ms_activescene`).

(c) Se viene specificata una telecamera diversa da quella corrente, si attiva la nuova telecamera come telecamera corrente (nel nostro caso non viene esplicitamente specificata la telecamera da utilizzare e dunque si lavora con quella corrente).

(d) Se non viene specificato altrimenti, si calcolano le matrici di projection e viewing sulla base della telecamera corrente e delle dimensioni del viewport, e i risultati vengono salvati in un'opportuna struttura da consultare in seguito. Questo passaggio può essere evitato usando un particolare flag da fornire alla funzione `Begin()`.

(e) Se non viene specificato altrimenti, si salvano i parametri delle sorgenti luminose in una apposita struttura dati, nella forma in cui verranno eventualmente passati agli shader. Questo passaggio può essere evitato usando un particolare flag da fornire alla funzione `Begin()`. L'operazione avviene anche quando si ha illuminazione tramite environment mapping perché questi dati potrebbero servire nel caso in cui si utilizzi uno shader esterno specificato dall'utente.

(f) Se sono attive delle environment map (diffuse o specular), queste vengono registrate come cube map attive sulle giuste texture unit OpenGL (per futuri accessi da parte degli shader).

(g) Se non viene specificato altrimenti, si ripuliscono il color buffer e il depth buffer del framebuffer associato alla finestra OpenGL. È possibile evitare la ripulitura del color buffer usando un particolare flag da fornire alla funzione `Begin()`.

(h) Se vi è un simulatore fisico (oggetto `VR3PhysicsSimulator`) attivo, questo è già stato preparato per la simulazione e non viene specificato altrimenti, si provvede a recuperare i risultati della simulazione PhysX (se disponibili). Se l'operazione ha successo, si avvia una nuova simulazione nel motore PhysX (i dati da essa prodotti verranno utilizzati nel prossimo frame, se disponibili). Questa operazione avviene mediante la funzione `CollectResults()` del simulatore fisico e può venire evitata usando un particolare flag da fornire alla funzione `Begin()` (in tal caso si utilizzeranno i risultati precedentemente ottenuti).

(i) Se vi è un controllore di ombre (oggetto `VR3ShadowController`) attivo e non viene specificato altrimenti, si disegnano e filtrano tutte le shadow map per le varie sorgenti d'ombra attive nel controllore (trascurando eventuali ottimizzazioni come i vari tipi di culling visti in sez. 5.5). Queste vengono inoltre attivate come texture bidimensionali sulle giuste texture unit OpenGL per futuri accessi da parte degli shader. Alcuni parametri delle sorgenti d'ombra attive vengono salvati in un'apposita struttura dati nella forma in cui verranno eventualmente passati agli shader per la modulazione del colore sulla base delle ombre. Questa operazione avviene mediante la funzione `DrawShadowMaps()` del controllore delle ombre e può venire evitata usando un particolare flag da fornire alla funzione `Begin()` (in tal caso si utilizzeranno le mappe prodotte dall'ultima chiamata di `DrawShadowMaps()`). Naturalmente, durante il disegno delle shadow map bisogna disegnare i vari oggetti registrati come caster all'interno del controllore (ma non entriamo nei dettagli).

(j) Si disegna lo sfondo statico o la skybox, se presente, tramite la funzione `Draw()` del membro privato `m_background` della scena (di tipo `VR3Background`). Non esaminiamo in dettaglio questa funzione di disegno.

2. `obj->Draw()`

(a) Nel caso in cui vi sia uno shadow controller attivo, si verifica se l'oggetto `obj` appartiene all'insieme dei ricevitori. In caso affermativo si attiva un flag ad indicare che l'oggetto dovrà subire una modulazione di colore dipendente dalle shadow map generate al passo 1i. Tale flag viene propagato nelle chiamate successive di funzioni `Draw()` su eventuali oggetti figli e sulla eventuale mesh selezionata per il disegno.

(b) Si calcola la matrice di modeling per l'oggetto `obj` sulla base della matrice ricevuta dal padre (in questo caso la matrice identità) e delle trasformazioni specificate per il particolare oggetto. Nel caso in cui sia attivo un simulatore fisico e l'oggetto risulti tra quelli simulati, si provvede a recuperare la matrice di modeling direttamente da PhysX tramite la funzione `GetModelMat()` del simulatore fisico.

(c) Si invoca la `Draw()` su tutti gli oggetti di tipo VR3Obj che risultano figli dell'oggetto `obj`. Nel nostro caso abbiamo supposto che `obj` fosse l'unico

oggetto nella scena e dunque non avrà oggetti figli. Ai figli verrebbe passata la matrice di modeling calcolata al passo precedente in modo tale da rendere le trasformazioni dei figli relative al sistema di riferimento del padre (nel caso in cui non siano simulati).

(d) Si seleziona un livello di dettaglio tra quelli disponibili e si disegna la mesh corrispondente (se ne è stata trovata una) tramite la sua funzione di disegno (`m_meshes[LOD]->Draw()`). Il disegno di una mesh procede secondo i seguenti passi quando si utilizzano gli shader predefiniti nella libreria e non si sta disegnando l'oggetto all'interno di una shadow map (trascurando eventuali ottimizzazioni come il view frustum culling).

i. Si calcolano i valori di alcune variabili uniform dipendenti dalla mesh complessiva, ad esempio la matrice di trasformazione totale (modeling, viewing e projection) da applicare ai vertici in spazio oggetto all'interno degli shader. Tali valori vengono salvati in una struttura dati opportuna da consultare in seguito. I valori calcolati vengono prodotti anche sulla base dei dati memorizzati nelle strutture dati introdotte nei passi 1d e 1e.

ii. Per ognuno dei subset presenti nella mesh:

- Si verifica se lo shader program correntemente attivo (usato per il precedente subset) deve essere usato anche per il subset corrente, in caso negativo si attiva il nuovo shader program e si caricano i valori delle variabili uniform dipendenti dalla mesh. Se lo shader program non deve essere modificato si evita invece il caricamento delle variabili uniform dipendenti dalla mesh (che avranno già il giusto valore dal subset precedente). Nel passaggio di questi valori si sfruttano le informazioni memorizzate nella struttura dati di cui al passo 2(d)i, ma anche i dati nelle strutture introdotte nei passi 1d, 1e, 1i.
- Si carica il valore delle variabili uniform dipendenti dal particolare subset (come quelle relative al materiale) attivando le eventuali texture sulle giuste texture unit OpenGL per renderle accessibili da shader.
- Si disegnano tutti i poligoni del subset con lo shader program attualmente attivo.

3. `scene->End()`
   In questa funzione si ha soltanto il ripristino dello stato che si aveva prima della chiamata alla funzione `Begin()`, ed inoltre si aggiornano le statistiche sulle prestazioni per la scena corrente che vengono mantenute dall'oggetto `scene` stesso. Qualsiasi operazione di disegno tentata fuori da un blocco `Begin()...End()` verrà rifiutata dalla VR3Lib e l'applicazione terminerà con errore.

Nonostante alcune operazioni minori siano state trascurate in questo semplice esempio, lo schema sopra dovrebbe chiarire il modello d'esecuzione adottato dalla VR3Lib per il rendering di ogni frame.

5.9 Funzionalità Rimosse

Nella costruzione della nuova libreria secondo la struttura presentata sopra, alcune caratteristiche precedentemente supportate dalla VRLib sono state omesse perché ritenute superflue visto lo scopo della tesi. Di seguito riportiamo una lista delle principali funzionalità rimosse:

- Nella vecchia VRLib, vi erano varie opzioni da attivare durante il rendering per ottenere effetti particolari come il motion blur[25]. Questi effetti erano stati realizzati basandosi su funzionalità presenti nelle vecchie versioni del sistema grafico OpenGL che sono state rimosse con le ultime revisioni, per cui non sono più supportati direttamente dalla libreria VR3Lib. Ovviamente, con l'utilizzo di shader esterni, l'utente sarà comunque in grado di ottenere tutti gli effetti desiderati.

- Nonostante la VRLib non abbia un supporto completo ad oggetti trasparenti (simile a quanto succede nella VR3Lib), esiste una routine automatica per l'ordinamento dei subset all'interno di una mesh per cercare di disegnare i vari subset in modo tale da evitare problemi durante l'operazione di alpha blending per materiali trasparenti (vedi sez. 2.3). Un servizio di questo tipo non è stato incluso nella nuova libreria perché non costituisce una soluzione completa al problema: per ottenere un risultato soddisfacente bisogna ordinare i singoli poligoni e non interi subset.

[25] Perdita di nitidezza dell'immagine al movimento degli oggetti osservati o della telecamera.

- La VRLib supporta anche animazioni e character[26] (che possono venire caricati sfruttando l'opportuno formato AAM presente nella vecchia specifica, vedi sez. 5.10). Queste funzionalità sono state abbandonate nella VR3Lib perché la nuova libreria ha come scopo fondamentale la visualizzazione di scene realistiche che evolvono in base all'interazione con l'utente e tramite simulazione fisica (ottenuta mediante PhysX). Il supporto ad animazioni predeterminate potrebbe comunque essere comodo nella costruzione di varie applicazioni, per cui future estensioni della VR3Lib potranno reinserire queste funzionalità mancanti. I character rappresentano semplicemente particolari gerarchie di oggetti: il motivo per cui non sono supportati nella VR3Lib deriva dal fatto che il loro utilizzo è conveniente soprattutto quando vengono animati.

Notare che quella sopra non è una lista completa delle vecchie funzionalità abbandonate nella nuova VR3Lib: contiene solamente le principali caratteristiche rimosse.

5.10 Il Formato AAM

Fino ad ora esaminando il funzionamento delle varie componenti della VR3Lib abbiamo trascurato alcune questioni fondamentali:

- Come si ottengono i dati geometrici sulla base dei quali si costruiscono le mesh per gli oggetti virtuali?
- Come si ottengono i parametri dei vari materiali e come si fa a sapere quali immagini utilizzare come texture?
- Come si ottengono i dati geometrici e i parametri da utilizzare per la simulazione fisica degli oggetti virtuali?

Tutte queste informazioni vengono salvate in file esterni che la VR3Lib si incarica di caricare eseguendo il parsing degli stessi. Le informazioni devono quindi essere organizzate in un particolare formato che la libreria sia in grado di comprendere.

Nella vecchia VRLib i formati supportati sono diversi, tra questi il formato AAM[27] è quello utilizzato più di frequente. Il formato AAM è stato concepito

[26] I character sono particolari gerarchie di oggetti con eventualmente un'animazione associata.

[27] AAM sta per 'Ascii Animated Mesh', a sottilineare che si tratta di un formato testuale capace di rappresentare anche mesh animate.

appositamente per il motore XVR e per questo consiste in una specifica dei dati nella forma con cui questi vengono elaborati nella VRLib. Un file AAM è un file di testo contente una serie di tag organizzati in alcuni blocchi. Tutti i tag utilizzati sono case-sensitive; inoltre l'ordinamento è fissato dalla specifica e dunque bisogna fornire i parametri nella giusta sequenza.

Una descrizione completa della specifica AAM è fuori dagli scopi di questa sezione. Per adesso è sufficiente osservare che nella versione originale del formato AAM è possibile inserire informazioni di carattere geometrico (necessarie a costruire oggetti di tipo `VR3Mesh`) e informazioni sui vari materiali applicati alle mesh (per la costruzione di oggetti di tipo `VR3Material`).

Rispetto alla vecchia VRLib, nella nuova VR3Lib abbiamo a che fare con materiali più complessi in grado di gestire 4 strati di texture con significato predeterminato. Inoltre notiamo che nel formato AAM originale non sono previste informazioni per simulare gli oggetti da un punto di vista fisico.

Volendo continuare ad utilizzare il formato AAM anche nella VR3Lib, bisogna estendere la specifica per introdurre tutte le informazioni mancanti di cui la nuova libreria potrebbe aver bisogno. Inoltre è necessario trovare un modo comodo per generare un file in formato AAM a partire da un modello costruito con un qualche programma di modellazione.

Un file AAM può contenere informazioni relative a tutti i materiali e le mesh presenti nell'ambiente virtuale da visualizzare, ma è anche possibile caricare mesh e materiali da file separati e usarli all'interno di una stessa scena grazie al supporto fornito dalla VR3Lib.

5.10.1 Potenziamento della Specifica AAM

La specifica del formato AAM è stata estesa in modo tale da fornire ulteriori informazioni che possono venire utilizzate dalla nuova libreria. File AAM nel vecchio formato vengono comunque supportati anche se ovviamente non saranno in grado di sfruttare le nuove caratteristiche inserite nella VR3Lib.

La specifica AAM originale prevede un formato particolare del file per la specifica di animazioni e character. Siccome questi oggetti non sono supportati nella VR3Lib, la specifica estesa è solamente quella relativa a scene senza character o animazioni. Un file AAM privo di animazioni e character è organizzato (secondo le specifiche originali) in due sezioni: `MATERIALS` e `GEOMETRY`.

Un primo potenziamento della specifica prevede di aggiungere ai materiali due nuovi livelli di texture (addizionali ai due già disponibili nella specifica originale). I 4 livelli risultanti vengono allora utilizzati per identificare, rispettivamente: diffuse texture, light map, normal map e displacement map. I due livelli aggiuntivi sono comunque opzionali in modo tale da avere compatibilità con file nel vecchio formato (che si adattano pertanto alle nuove specifiche). Questa modifica si applica nella sezione `MATERIALS`.

Per quanto riguarda la sezione `GEOMETRY`, le modifiche sono banali: visto che si aggiungono fino a due livelli di texture bisognerà eventualmente aggiungere tutte le coordinate per l'accesso alle due nuove texture (ossia bisognerà aggiungere nuovi texture vertices, e questo avviene in modo molto semplice visto che la precedente specifica AAM è stata concepita per essere facilmente estendibile. Inoltre nella sezione `GEOMETRY` sono stati aggiunti alcuni parametri opzionali da applicare nel caso di displacement mapping (tessellation factor, bias e scale, vedi sez. **??**).

Infine, al nuovo formato AAM è stata aggiunta una sezione opzionale `PHYSICS` per fornire tutti i dati geometrici con cui costruire le istanze di `VR3PhyMesh`, contenente anche tutti gli altri parametri necessari per la simulazione fisica di un oggetto (ossia le informazioni sul materiale fisico, la massa e il tipo di simulazione da effettuare).

Le informazioni geometriche per la mesh fisica sono diverse in dipendenza dal tipo di simulazione:

- Oggetti *statici*
  I dati geometrici per la simulazione fisica sono gli stessi usati per il disegno: la geometria della mesh fisica (`VR3PhyMesh`) corrisponde in modo esatto alla geometria della mesh usata per il rendering (`VR3Mesh`). Le informazioni necessarie verranno allora lette dalla sezione `GEOMETRY`.

- Oggetti *dinamici* o *cinematici*
  I dati geometrici vengono forniti con una nuova serie di mesh convesse che andranno inserite in un composto PhysX per simulare oggetti dinamici o cinematici di forma arbitraria. Il numero e la complessità di queste mesh convesse dipende dall'accuratezza richiesta nella simulazione. Nonostante questo approccio sia stato guidato dai limiti propri di PhysX, è valido anche in altri motori di simulazione fisica come *Havok* e *Bullet* (infatti deriva da problemi di efficienza computazionale).

Non tutti gli oggetti all'interno del file AAM devono necessariamente comparire nella sezione `PHYSICS` quando questa è presente. Se ad un oggetto sono però associate

delle informazioni per la simulazione fisica, è necessario fornire anche i parametri del materiale fisico da applicare all'oggetto:

- coefficiente di *attrito statico*,
- coefficiente di *attrito dinamico*,
- coefficiente di *restituzione* (o *bounciness*).

Inoltre, per oggetti dinamici, è necessario fornire la massa dell'oggetto. La massa è intesa uniformemente distribuita nel volume dell'oggetto, come se il materiale fisico di cui è composto lo riempisse interamente; dunque se l'oggetto subisce un cambiamento di scala (prima di cominciare la simulazione) la massa verrà adattata di conseguenza.

Notare che tutte le quantità specificate all'interno del file AAM sono prive di unità di misura. Sulla base delle impostazioni di default all'interno di un simulatore fisico (oggetto `VR3PhysicsSimulator`), come ad esempio l'accelerazione di gravità di default, i valori letti dal file AAM e le quantità fornite a posteriori durante la simulazione vengono interpretate in accordo alle seguenti unità di misura per le grandezze fondamentali:

- lunghezze in metri (m),
- masse in chilogrammi (kg),
- intervalli di tempo in secondi (s).

Tuttavia l'utente è libero di interpretare in modo diverso i valori letti dal file AAM modificando le impostazioni di default per il simulatore fisico e fornendo delle quantità per applicare forze e modificare velocità che siano in accordo con le unità di misura scelte (il tempo verrà comunque considerato sempre in secondi e la simulazione procederà con la cadenza temporale scandita dallo scorrere del tempo nella realtà).

5.10.2 L'Esportatore per 3ds Max

Viste le modifiche che sono state apportate alla specifica del formato AAM in accordo con le capacità della nuova libreria grafica, si pone adesso il problema di costruire dei file nel nuovo formato. La creazione manuale di file AAM è impensabile in quanto normalmente la quantità di informazioni da scrivere in questi file è enorme

anche per scene relativamente semplici. In realtà l'unico modo agevole per costruire un file AAM è modellare gli oggetti di interesse con l'ausilio di un programma di modellazione e poi esportare i dati sotto forma di file AAM.

Esistono vari programmi di modellazione a cui possiamo fare riferimento:

- Autodesk *3ds Max*,
- Autodesk *Maya*,
- *Blender*,
- ...

In passato è stato realizzato (sempre nell'ambito del progetto XVR) un plugin per 3ds Max capace di esportare i dati della scena modellata direttamente in formato AAM. 3ds Max risulta ad oggi una delle scelte più comuni per costruire modelli per applicazioni di rendering in tempo reale e dunque risulta adatto anche nel caso della nuova VR3Lib.

Durante il lavoro di tesi, il plugin sviluppato secondo le vecchie specifiche del formato AAM è stato quindi modificato per esportare file nel nuovo formato. 3ds Max ci consente di sfruttare in modo semplice tutte le caratteristiche della nuova VR3Lib e di creare anche le varie texture da utilizzare durante il rendering.

Per evitare di presentare una serie di questioni poco rilevanti, non entriamo nei dettagli della struttura del plugin. Accenniamo soltanto a come sia stata realizzata l'esportazione della geometria per la simulazione fisica di oggetti non statici a partire dalla scena modellata.

I dati geometrici a disposizione del plugin, forniti da 3ds Max, corrispondono a tutte le mesh che rappresentano gli oggetti nella scena. Queste mesh potranno essere in generale delle mesh concave: in tal caso abbiamo detto che nel formato AAM ci si aspetta di ricevere una serie di mesh convesse da comporre per la simulazione dinamica o cinematica dell'oggetto. All'interno del plugin è stato perciò inserito un modulo per la *decomposizione convessa* di mesh arbitrarie, in grado di produrre un insieme di mesh convesse che approssimano una mesh arbitraria in base ad alcuni parametri di decomposizione.

Il modulo software aggiunto si basa sul codice scritto originariamente da John W. Ratcliff, attualmente senior software engineer presso Nvidia. Con il plugin modificato, l'utente 3ds Max è in grado di specificare, per ogni oggetto nella scena, quanto accurata deve essere l'approssimazione ottenuta mediante la decomposizione

convessa, e diversi parametri aggiuntivi per regolare la decomposizione (ricordando comunque che tale processo si applica soltanto nel caso di oggetti non statici).

In fig. 5.9 è riportato uno screenshot della GUI (*Graphical User Interface*) del plugin modificato, lavorando con 3ds Max 2010.

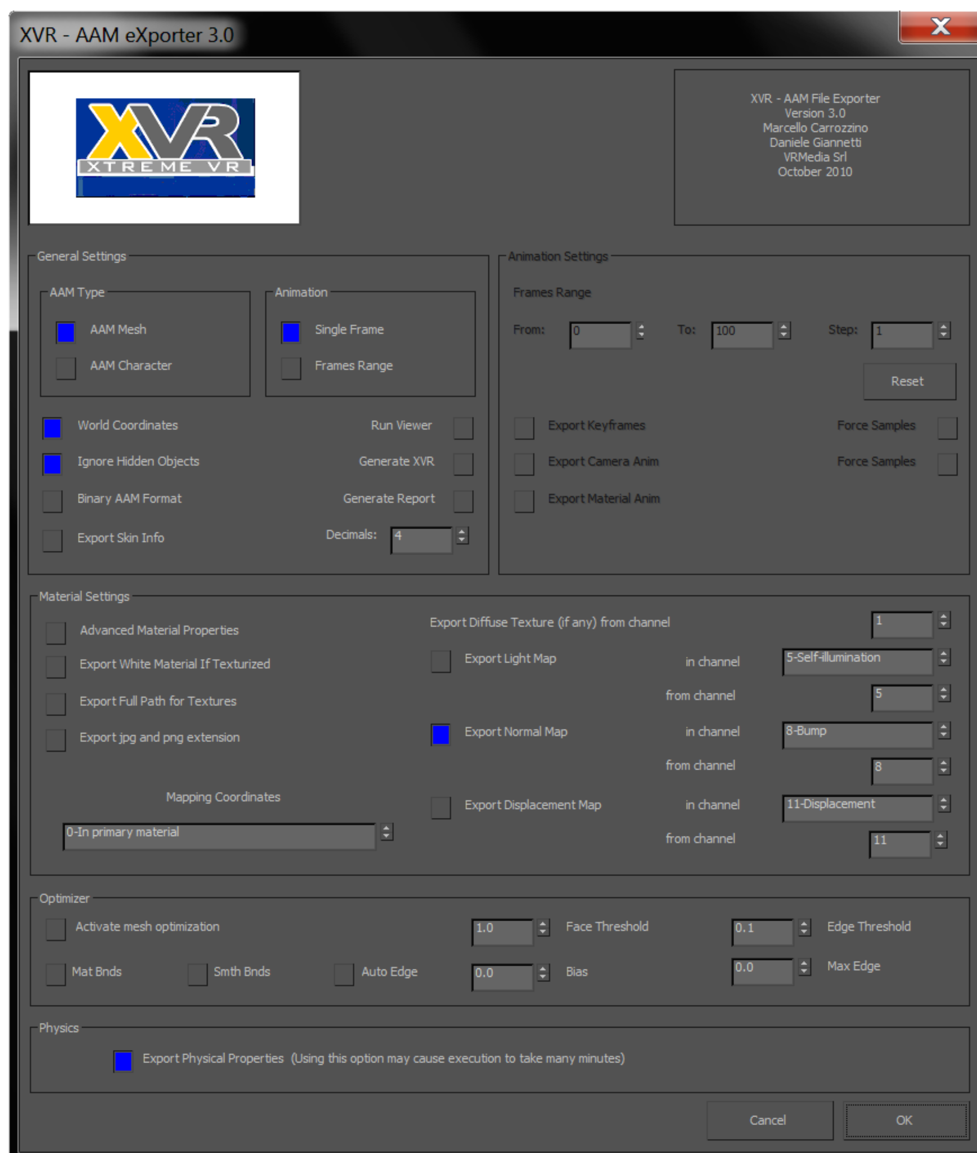


Figura 5.9: GUI del plugin 3ds Max modificato

Capitolo 6

Testing e Analisi delle Prestazioni

Nei precedenti capitoli abbiamo visto le moderne tecniche integrate nella nuova libreria e come implementarle con riferimento alla struttura degli shader; inoltre sono stati forniti alcuni dettagli sulla struttura della VR3Lib e sul suo funzionamento di base. In questo capitolo cerchiamo di misurare le prestazioni della libreria che è stata costruita.

Il parametro principale di interesse quando si valutano le prestazioni di un motore di rendering e simulazione come la VR3Lib è il tempo necessario per completare il disegno di un frame, che ha un'influenza diretta sul frame rate dell'applicazione. In questo capitolo pertanto ignoriamo qualsiasi aspetto di efficienza durante l'inizializzazione dell'applicazione e ci concentriamo solamente sulle prestazioni nel ciclo di rendering.

La VR3Lib è una libreria da utilizzare puramente per il rendering e la simulazione fisica: non si occupa in alcun modo di gestire la finestra o il framebuffer della stessa, ma semplicemente esegue il disegno dell'ambiente virtuale all'interno di un framebuffer fornito al server OpenGL. Particolari politiche di gestione del framebuffer (come il double buffering, par. 1.5.2) vengono realizzate indipendentemente dalla libreria. Il frame rate non dipenderà allora solamente dalle operazioni della VR3Lib ma anche da tutte le altre attività che vengono effettuate all'esterno della stessa ad ogni frame (come appunto lo scambio dei buffer). L'inverso del tempo necessario a tracciare il frame con la VR3Lib rappresenta quindi il limite teorico al frame rate; quanto effettivamente si riesca ad avvicinarsi al valore teorico dipende dalle operazioni non relative alle attività della VR3Lib effettuate nel corso di un frame. Noi ci limiteremo a misurare i tempi direttamente influenzati dalle attività della VR3Lib e dunque trascureremo le operazioni eseguite all'esterno della libre-

ria, anche se facenti parte del ciclo di rendering (tali operazioni dipendono anche dall'approccio utilizzato per gestire il contesto e la finestra OpenGL).

Tutti i test che seguono sono stati effettuati sulla medesima macchina, un laptop Sager NP9280 con le seguenti specifiche:

- nasato su chipset Intel X58,
- processore Intel Core i7 920,
- 6 GB di RAM DDR3 a 1066 MHz,
- scheda video Nvidia GeForce GTX 280M.

Questo capitolo è stato organizzato sulla base degli aspetti del funzionamento della VR3Lib che si sono testati, ed ogni sezione illustra il comportamento utilizzando alcune delle funzionalità disponibili.

Per misurare le prestazioni della libreria a tempo di rendering, si misura il tempo che intercorre tra la chiamata ad una funzione `Begin()` della scena e la fine della funzione `End()` corrispondente (nei test che vediamo si ha sempre un unico blocco `Begin()...End()` nel tracciamento di un frame). La struttura è pertanto la seguente:

```
VR3Scene* scene; // reference to a scene

...

// Executed once per frame
OnFrame() {
    glFinish();
    time1 = GetTime(); // get begin time

    scene->Begin(); // begin scene rendering
    ...
    scene->End(); // end scene rendering

    glFinish();
    time2 = GetTime(); // get end time

    frametime = time2 - time1;
}

...
```

Notiamo che la chiamata a `glFinish()` è indispensabile per far terminare tutti i comandi inviati al server GL prima di procedere a determinare il tempo trascorso (si vuole infatti misurare il tempo di esecuzione dei comandi lanciati all'interno del blocco `Begin()...End()`). Il tempo calcolato come sopra ad ogni frame si dice *tempo di tracciamento del frame* (e nel seguito verrà chiamato semplicemente *tempo di frame* o *frame time*); ricordiamo che l'inverso di questo tempo è un limite teorico al frame rate, ma quest'ultimo risulterà in genere più basso (talvolta in modo deciso) per le operazioni indipendenti dalla VR3Lib svolte all'esterno del blocco `Begin()...End()`.

Tutti i test che seguono sono stati effettuati disabilitando le ottimizzazioni disponibili all'interno della libreria, si fa pertanto a meno di:

- back-face culling (anche nel rendering delle shadow map),
- view frustum culling (anche nel rendering delle shadow map),
- shadow source culling.

6.1 Complessità Geometrica

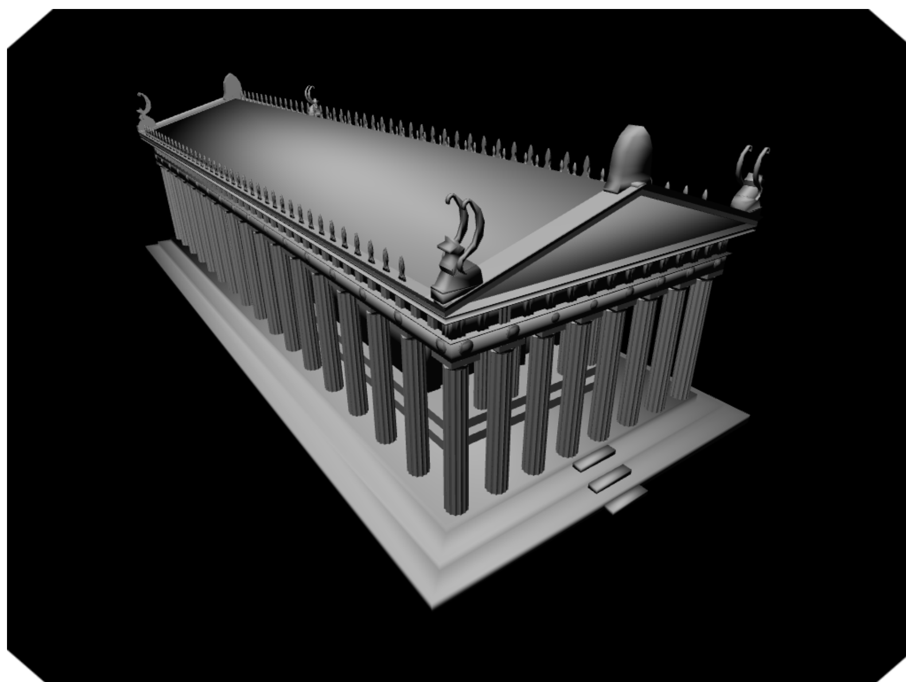
Esaminiamo adesso le funzionalità di base della VR3Lib considerando le prestazioni al variare della complessità geometrica delle scene renderizzate (ossia del numero di poligoni disegnati ad ogni frame). Per eseguire i test sono state scelte 4 diverse mesh con crescente complessità geometrica, illustrate nelle fig. 6.1 e 6.2.

Nome	**Numero di triangoli**
Statua A	9518
Partenone	107191
Automobile	654083
Statua B	2936704

Tabella 6.1: Mesh utilizzate per i test sulla complessità geometrica

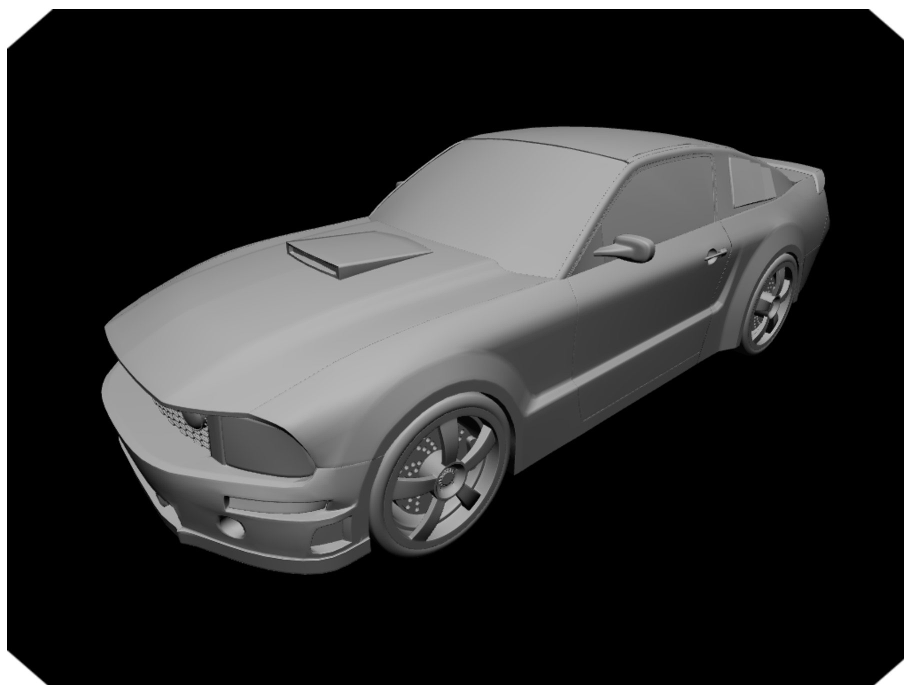
Come si vede, la mesh più complessa contiene quasi 3 milioni di poligoni, ma non appare di qualità superiore alle precedenti (la sua struttura sfaccettata suggerisce infatti una bassa qualità). In generale, non sempre aumentando il numero di poligoni si ottengono mesh più convincenti (quella in fig. 6.2b è stata infatti ricavata tassellando una mesh molto più semplice).

(a) Statua A - 9518 triangoli



(b) Partenone - 107191 triangoli

Figura 6.1: Mesh utilizzate per i test sulla complessità geometrica (1)



(a) Automobile - 654083 triangoli

(b) Statua B - 2936704 triangoli

Figura 6.2: Mesh utilizzate per i test sulla complessità geometrica (2)

Naturalmente, un maggior numero di triangoli comporta un carico superiore su molti stadi della pipeline di rendering OpenGL. Un altro parametro importante però è la porzione della finestra OpenGL occupata dalla mesh: anche nel caso di pochi triangoli se la mesh occupa un'ampia area dello schermo abbiamo una forte attività nello stadio di fragment processing perchè molti pixel possono venire alterati rasterizzando un singolo poligono. Nei test sulla complessità geometrica è stata utilizzata una telecamera mobile secondo un percorso predeterminato e si è fatto in modo che i 4 oggetti di cui sopra avessero approssimativamente le stesse dimensioni nella scena virtuale: ci si aspetta una variazione del frame time sulla base del percorso della telecamera (visto che il numero di pixel interessati dalla rasterizzazione dei vari poligoni nella mesh cambia nel tempo).

Nella scena è presente un unico oggetto posto nell'origine degli assi a cui viene associata una mesh tra quelle nelle fig. 6.1 e 6.2. Il percorso secondo cui si muove la telecamera è illustrato in fig. 6.3.

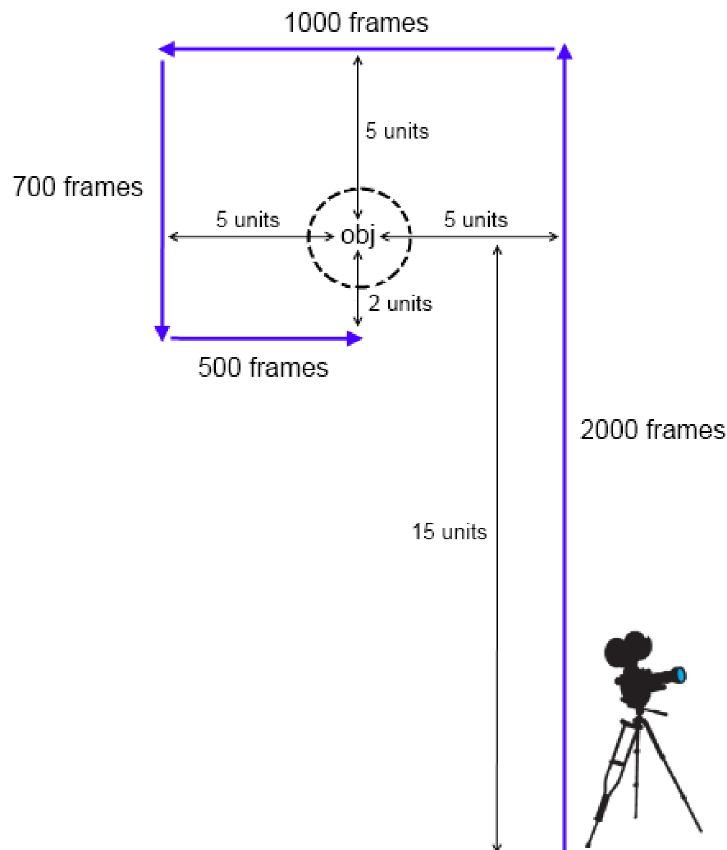


Figura 6.3: Percorso della telecamera

Il tracciato ha una durata predeterminata in termini di numero di fotogrammi: si percorrono in totale 42 unità di distanza e si catturano 4200 fotogrammi (ad ogni frame ci si muove quindi di 0.01 unità). Durante il movimento la telecamera resta sempre puntata verso l'oggetto (obj in figura).

Per ognuna delle 4 mesh di cui sopra abbiamo eseguito 4 test lavorando con due possibili dimensioni della finestra e del viewport OpenGL (1024×768 e 1920×1080[28]) ed inoltre ripetendo il test nel caso di:

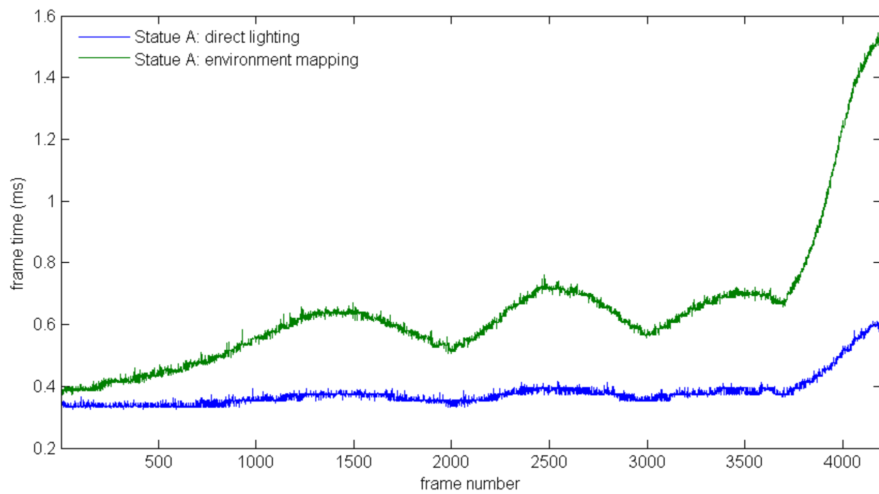
- illuminazione diretta con una singola sorgente luminosa,
- environment mapping con immagini HDR.

Nelle figure 6.4 e 6.5 sono riportati i risultati per viewport e finestra OpenGL 1024×768, dove in ogni grafico si possono confrontare gli andamenti nel caso di illuminazione diretta e image-based lighting.

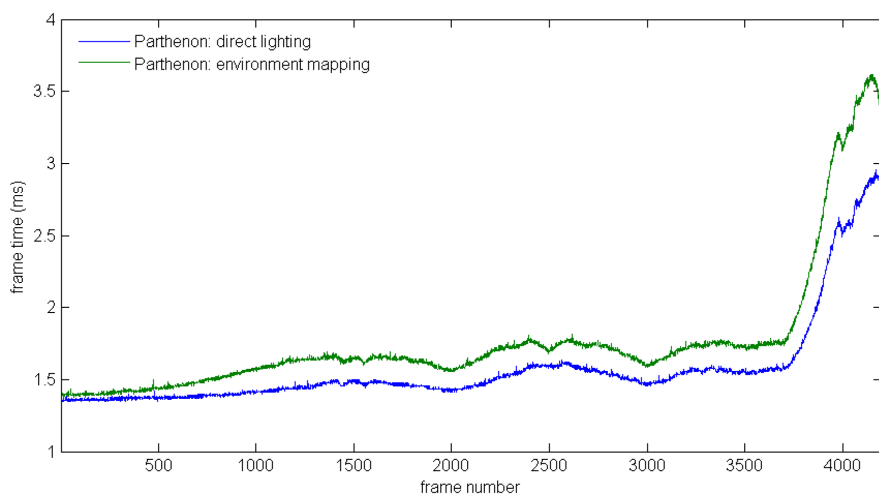
In questi grafici possiamo notare che:

- Riportando il valore calcolato per ogni singolo frame, l'andamento non è molto regolare e i grafici risultano 'rumorosi'. Non ci si può aspettare di ottenere esattamente lo stesso risultato in due diversi fotogrammi (anche se la scena risulta identica): i moderni calcolatori general purpose come la macchina utilizzata hanno infatti un'esecuzione in qualche modo impredicibile grazie alle note politiche di caching, alle varie attività che vengono portate avanti in concorrenza e ad altri meccanismi di basso livello che non esaminiamo in questo testo.
- Il tracciamento del frame richiede un tempo (in media) maggiore quando si utilizza la tecnica di environment mapping: questo deriva dal fatto che bisogna accedere alle cube environment map nel fragment shader e si complicano allora le operazioni nello stadio di fragment processing.
- L'andamento oscillatorio che si nota facilmente in fig. 6.4a è presente in tutti i casi, sia quando si utilizza illuminazione diretta che nel caso di environment mapping. Questo deriva dal modo in cui si muove la telecamera, che provoca una variazione nel tempo del numero di pixel interessati dalla rasterizzazione dei poligoni dell'oggetto e dunque un carico variabile sullo stadio di fragment

[28] La risoluzione 1920×1080 è tipica delle modalità video cosiddette 'Full HD' con aspect ratio di 16:9.

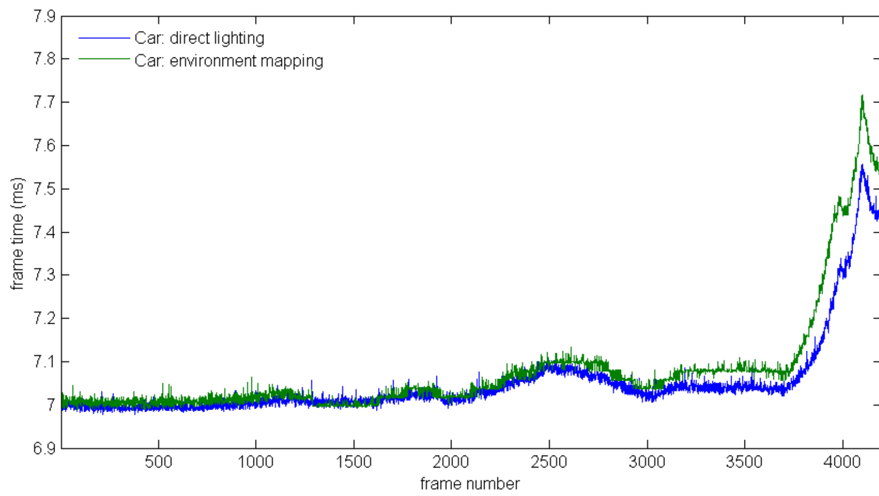


(a) Statua A

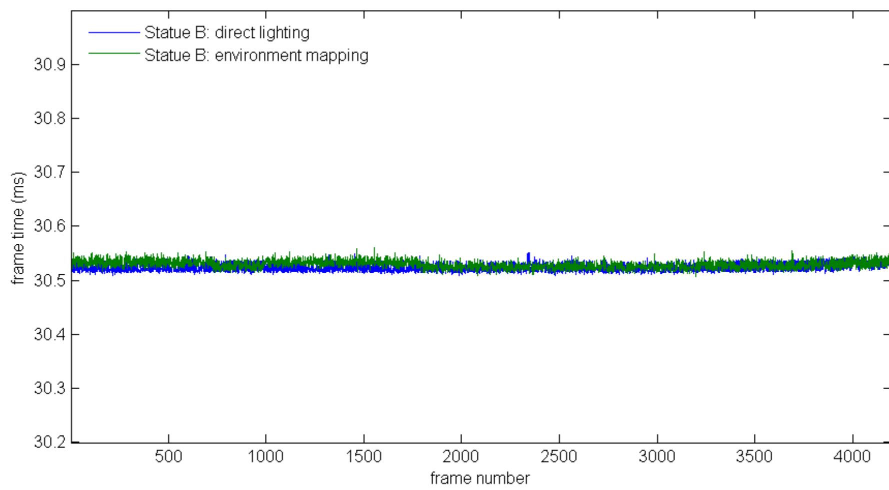


(b) Partenone

Figura 6.4: Prestazioni al variare della complessità geometrica - 1024×768 (1)



(a) Automobile



(b) Statua B

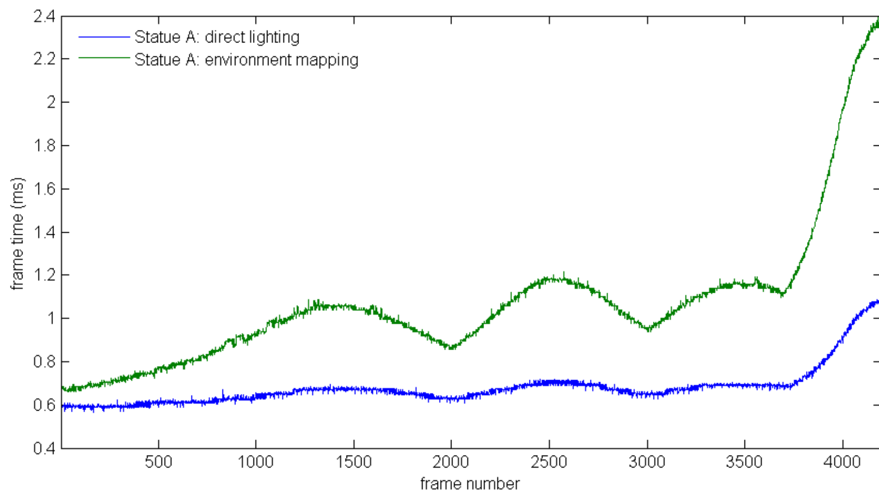
Figura 6.5: Prestazioni al variare della complessità geometrica - 1024×768 (2)

processing. L'andamento oscillatorio è sempre meno visibile all'aumentare della complessità geometrica perché fondamentalmente non si incrementa il carico sullo stadio di fragment processing in modo significativo, mentre il lavoro negli altri stadi della pipeline aumenta fino a rendere trascurabili le differenze nel carico sullo stadio di fragment processing (e questo spiega anche perché all'aumentare della complessità geometrica le differenze tra illuminazione diretta e environment mapping siano sempre meno marcate).

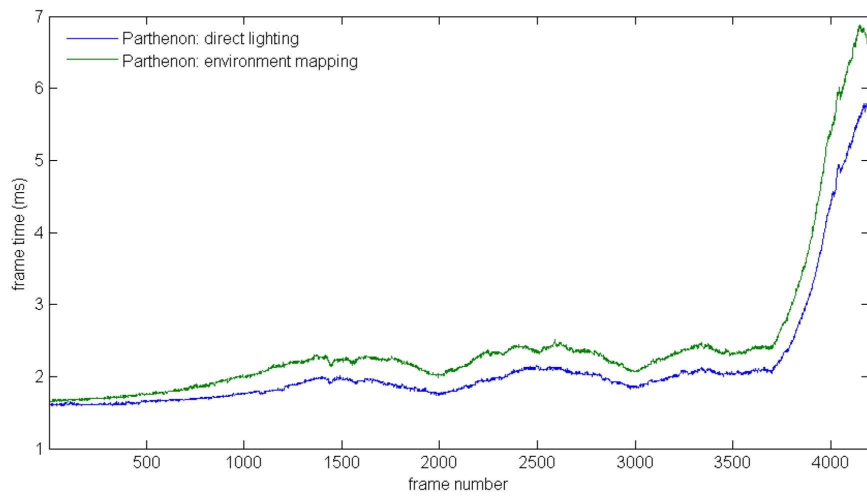
- Nel caso della Statua B abbiamo che i due andamenti non sono distinguibili e in pratica si percepisce soltanto una oscillazione attorno ad un valor medio di circa 30.53 ms. Il carico sul resto della pipeline grafica in questo caso supera di gran lunga quello sullo stadio di fragment processing e dunque non sono visibili variazioni sulla base della posizione della telecamera e neanche differenze significative in base al tipo di illuminazione utilizzata. Un frame time di 30.53 ms implica un frame rate teorico massimo di circa 32.75 fps; questo è il limite nel caso di applicazioni interattive (valori più bassi rendono l'utilizzo dell'applicazione poco agevole).

I test sono stati ripetuti nel caso di risoluzione 1920×1080 e i risultati sono riportati nelle fig. 6.6 e 6.7; si nota un peggioramento generale delle prestazioni rispetto al caso 1024×768 dovuto al maggior carico sullo stadio di fragment processing (più pixel nella finestra OpenGL). Questo peggioramento si fa meno marcato all'aumentare della complessità geometrica (infatti il lavoro nello stadio di fragment processing ha un'influenza sempre più ridotta sul frame time all'aumentare del numero di poligoni, visto che gli oggetti hanno tutti approssimativamente le stesse dimensioni).

Per mettere in evidenza le differenze di prestazioni al variare della complessità geometrica, in fig. 6.8 è riportato l'andamento dell'istante in cui si termina il tracciamento di un frame in funzione del numero del frame per ognuna delle 4 mesh sopra. In questo test abbiamo utilizzato solamente illuminazione diretta e risoluzione 1024×768. I tempi vengono calcolati a partire dalla fine della fase di inizializzazione dell'applicazione. L'inclinazione delle curve rappresenta l'inverso del frame rate sperimentato dall'utente.

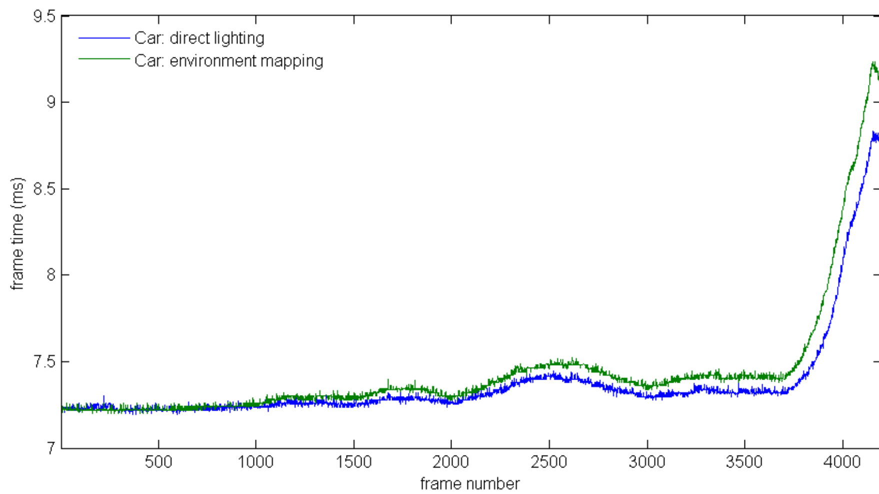


(a) Statua A

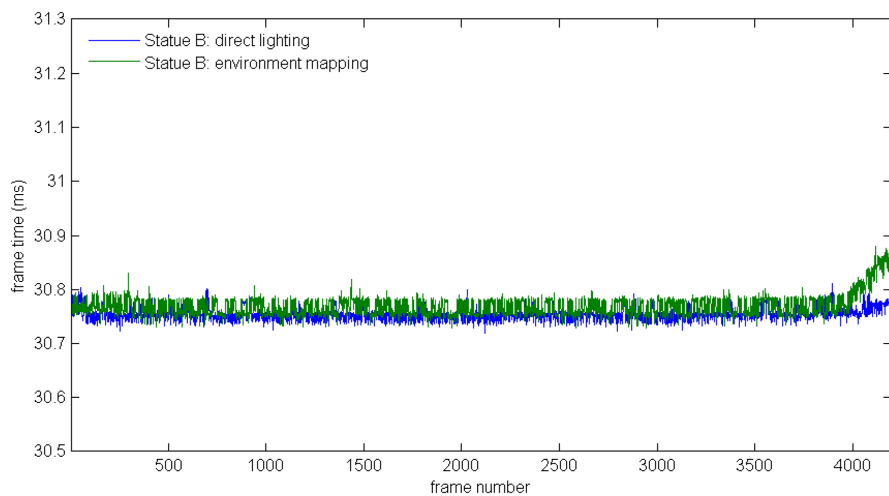


(b) Partenone

Figura 6.6: Prestazioni al variare della complessità geometrica - 1920×1080 (1)



(a) Automobile



(b) Statua B

Figura 6.7: Prestazioni al variare della complessità geometrica - 1920 $\times$ 1080 (2)

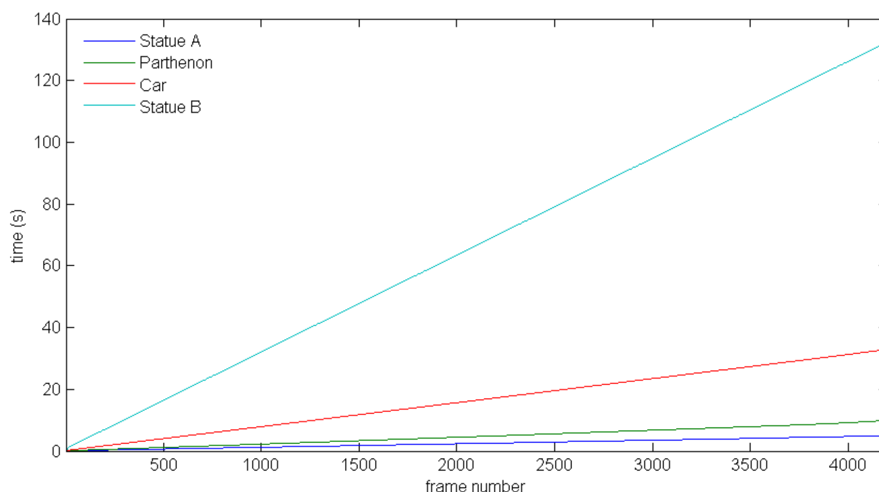


Figura 6.8: Confronto di prestazioni al variare della complessità geometrica

Notare allora come le operazioni eseguite all'esterno del blocco `Begin()...End()` allontanino il frame rate dal valore teorico dato dall'inverso del tempo di tracciamento del frame all'interno delle VR3Lib. Nel caso della statua A otteniamo dal grafico sopra un frame rate medio di circa:

$$\frac{4200}{5} = 840\,\text{fps}$$

mentre per quanto visto in fig. 6.4a il valore teorico massimo è compreso tra 1500 e 3000 fps. Questa differenza diventa meno significativa al crescere della complessità della scena: il valore teorico e quello effettivo sono infatti molto vicini nel caso della statua B (tra 32 e 33 fps) perché il tempo necessario alla VR3Lib per completare le sue operazioni nel blocco `Begin()...End()` è molto superiore al tempo impiegato all'esterno del blocco per tutte le altre attività. Per gestire la finestra, il framebuffer e il contesto OpenGL in questo test abbiamo utilizzato la libreria open source *freeglut* (lavorando con la tecnica del double buffering).

6.2 Tecniche basate su Texture

Visto il modo in cui la complessità geometrica degli oggetti visualizzati influenza il tempo di tracciamento di un frame, vediamo adesso l'effetto che hanno alcune delle tecniche presentate nel capitolo 2, in particolare:

- diffuse texture mapping,

- normal mapping,
- displacement mapping,
- light mapping.

Continueremo anche in questo capitolo a confrontare i risultati che si ottengono nel caso di illuminazione diretta (con una singola sorgente luminosa) e di image-based lighting.

È stata scelta una mesh molto semplice per la quale fossero disponibili versioni che sfruttano ognuna delle combinazioni delle tecniche sopra. La mesh selezionata è quella riportata in fig. 2.16. Questa mesh ha forma cubica e si compone di 108 triangoli, 18 per ogni faccia del cubo. Sono disponibili diffuse texture, normal map e displacement map per la mesh, che producono l'effetto in fig. 2.16. Tutte le texture (o mappe) utilizzate nelle simulazioni di questa sezione hanno una dimensione di 512×512 texel: naturalmente le dimensioni delle texture sono un altro parametro che occorre tenere presente quando si vogliono regolare le prestazioni dell'applicazione.

Dato che 108 triangoli sono pochi per fare una prova significativa (non avremo mai scene così semplici nella pratica), si è deciso di costruire la scena per il test replicando la mesh di cui sopra in una matrice tridimensionale di $4 \times 4 \times 4$ oggetti, per un totale di 6912 poligoni nella scena. Gli oggetti visualizzati sono riportati in fig. 6.9 (nella versione dove si applica solo diffuse texture mapping). Notare che 6912 poligoni sono ancora pochi nella pratica, ma sono idonei a mettere in evidenza le differenze tra le tecniche presentate.

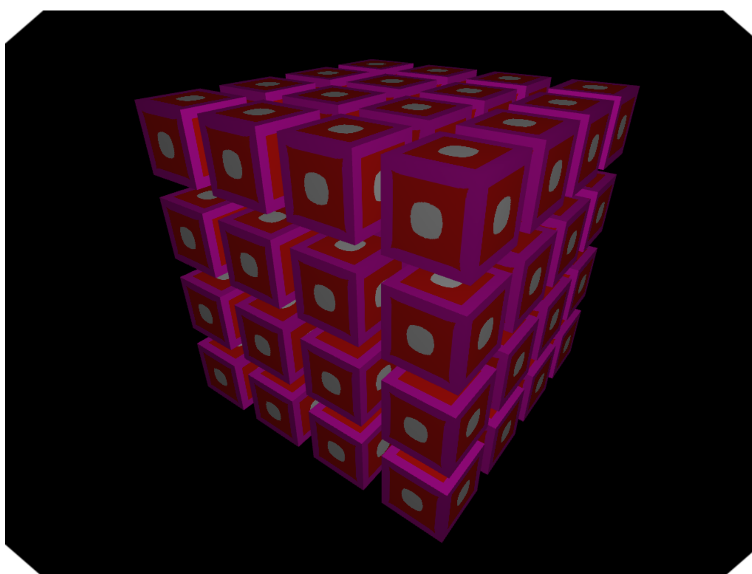


Figura 6.9: Oggetti utilizzati nei test sulle tecniche basate su texture

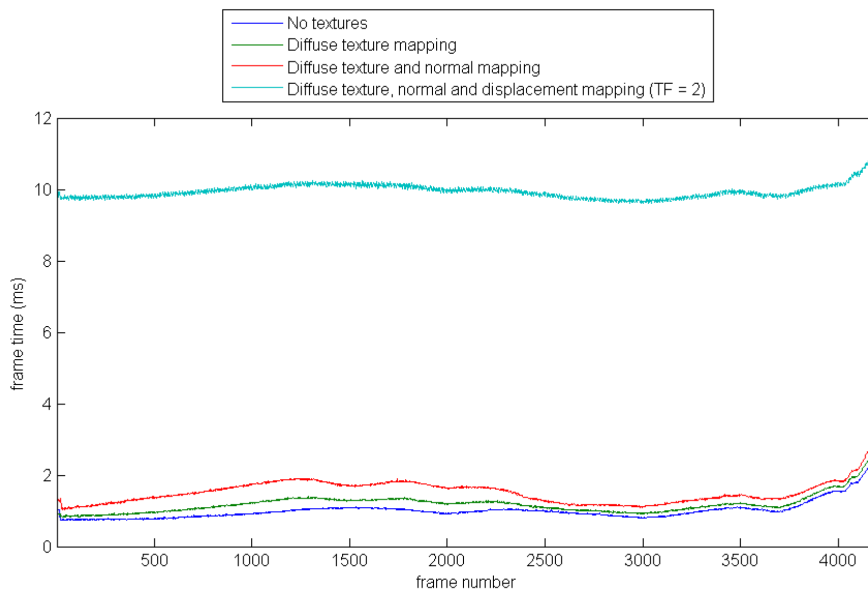
Il gruppo di 64 oggetti viene posizionato nella stessa maniera degli oggetti usati nei test effettuati in sez. 6.1 e si fa in modo che la telecamera evolva sempre seguendo il percorso in fig. 6.3. I test sono stati effettuati con le seguenti combinazioni di tecniche basate su texture:

- nessuna tecnica utilizzata;
- diffuse texture mapping;
- diffuse texture mapping e normal mapping;
- diffuse texture mapping, normal mapping e displacement mapping.

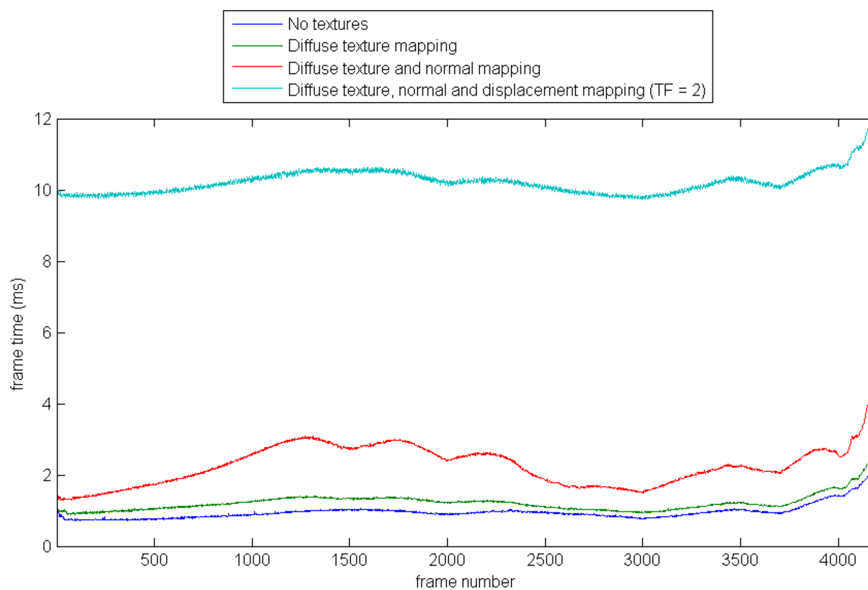
Per ognuno dei casi sopra sono stati eseguiti i test per due diverse dimensioni della finestra e del viewport OpenGL e nei due casi di illuminazione diretta e image-based lighting. In fig. 6.10 vengono mostrati due grafici, ognuno dei quali riporta gli andamenti per i 4 casi di cui sopra: nel primo grafico abbiamo illuminazione diretta, nel secondo sfruttiamo la nota tecnica di environment mapping con immagini HDR, in entrambi i grafici la risoluzione utilizzata per il viewport è 1024×768.

Notiamo che è nuovamente presente un'oscillazione nell'andamento, dovuta al movimento della telecamera e alla particolare forma degli oggetti che causa una variazione nel tempo del numero di frammenti interessati dalla rasterizzazione dei poligoni (e dunque del carico sullo stadio di fragment processing). Interessante è inoltre notare che passando da nessun livello di texture (linea blu) a diffuse texture e normal mapping (linea rossa) si nota un aumento moderato del frame time; quando invece si introduce la tecnica di displacement mapping (azzurro) si assiste ad un incremento considerevole.

Per i test di cui sopra abbiamo usato displacement mapping con tessellation factor pari a 2 (il minimo possibile), per cui da ogni triangolo si generano solamente 4 diversi triangoli nello stadio di geometry processing. L'algoritmo di tassellamento utilizzato all'interno del geometry shader impone quindi un carico piuttosto elevato visto che (come si evince anche dai risultati in sez. 6.1) sarebbe conveniente utilizzare una mesh contenente fin da subito tutti i poligoni che nel nostro caso vengono generati dinamicamente dal displacement mapping. Le prestazioni che si ottengono dalla tecnica di displacement mapping non sono allora completamente soddisfacenti; questo deriva dal fatto che realizzare il tassellamento all'interno del geometry shader non porta a risultati eccellenti, e anche per questo motivo nelle ultime versioni di OpenGL e Direct3D sono stati introdotti due nuovi stadi programmabili appositamente per questo scopo. In ogni caso la funzionalità di displacement mapping

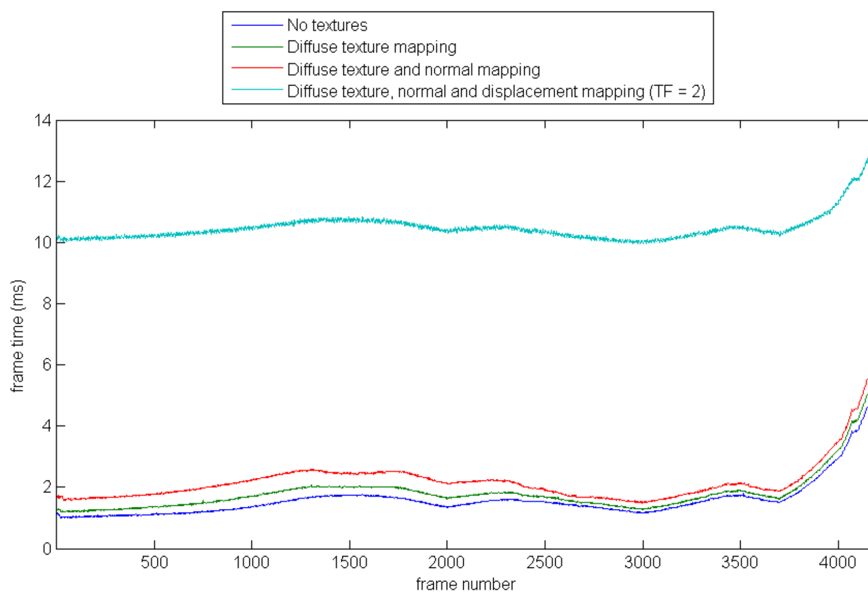


(a) Illuminazione diretta (unica sorgente)

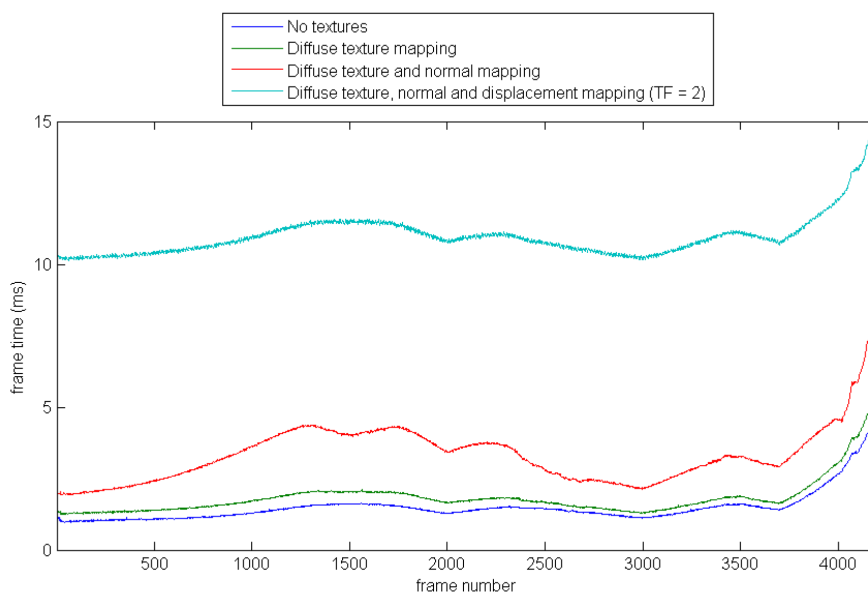


(b) Environment mapping

Figura 6.10: Prestazioni delle tecniche basate su texture - 1024×768



(a) Illuminazione diretta (unica sorgente)



(b) Environment mapping

Figura 6.11: Prestazioni delle tecniche basate su texture - 1920 × 1080

può comunque risultare interessante in quanto comporta un risparmio di memoria in termini di informazioni immagazzinate per una singola mesh e può venire usata per alterare dinamicamente la geometria modificando a run-time la texture usata come displacement map.

In fig. 6.11 vengono riportati i risultati degli stessi test di figura 6.10, utilizzando però una finestra e un viewport OpenGL con risoluzione 1920×1080. Quello che si nota è nuovamente un incremento globale del frame time dovuto al maggior lavoro necessario all'interno dello stadio di fragment processing.

Con riferimento alla tecnica di light mapping (da utilizzare quando è possibile determinare preventivamente l'illuminazione di un oggetto e immagazzinarla in una texture), in fig. 6.12 viene mostrato l'andamento per le due risoluzioni confrontate fino ad ora qualora agli oggetti nella scena sopra venisse applicata solamente una light map (senza altre tecniche basate su texture).

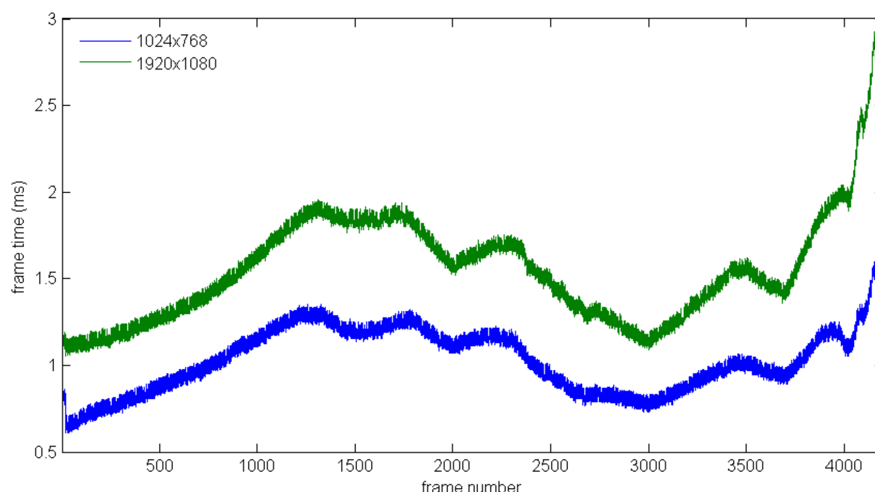


Figura 6.12: Prestazioni della tecnica di light mapping

Come si nota, le prestazioni sono confrontabili col caso di oggetti renderizzati senza applicare alcuna texture. In effetti nel caso di light mapping (come visto nel par. 2.1.4) il calcolo dell'illuminazione per un pixel si riduce ad un accesso alla light map, per cui è sempre conveniente usare questa tecnica quando possibile.

Tra le tecniche viste sopra, quella di displacement mapping è l'unica che non si basa su fragment shading: la sua realizzazione con la versione 3.3 del sistema grafico OpenGL (come visto in sez. 4.6) riguarda infatti sostanzialmente lo stadio di geometry processing. Visto che si devono costruire svariati poligoni a partire da ogni triangolo dell'oggetto interessato, le prestazioni di questa tecnica dipendono dalla

complessità geometrica dell'oggetto cui viene applicata (che normalmente dovrebbe essere bassa quando si applica questa tecnica). Ricordiamo che il numero di poligoni generati a partire da un singolo triangolo dipende dal parametro TF (tessellation factor) che ha un limite superiore dipendente dall'implementazione utilizzata del sistema grafico OpenGL.

Per verificare le prestazioni della tecnica di displacement mapping al variare del tessellation factor, è stata utilizzata una telecamera immobile che guarda l'oggetto a cui viene applicata la tecnica, e si è calcolato il frame time medio su 1000 frame per i vari valori possibili di TF. Tutti i test sono stati effettuati con una risoluzione di 1024×768. Nell'implementazione OpenGL utilizzata abbiamo che il valore massimo di TF per oggetti privi di diffuse texture (come quello utilizzato) risulta 9. Il calcolo del frame time medio per ogni valore di TF è stato effettuato considerando due diversi casi con diversa complessità geometrica:

1. Oggetto singolo di 108 triangoli, come quello in fig. 2.16 ma privo di diffuse texture.

2. Matrice $2 \times 2 \times 2$ di oggetti identici a quello di cui al punto 1.

L'oggetto utilizzato nel caso 1 e replicato in una matrice $2 \times 2 \times 2$ nel caso 2 è quello illustrato in fig. 6.13 (mostrato con TF pari a 7).

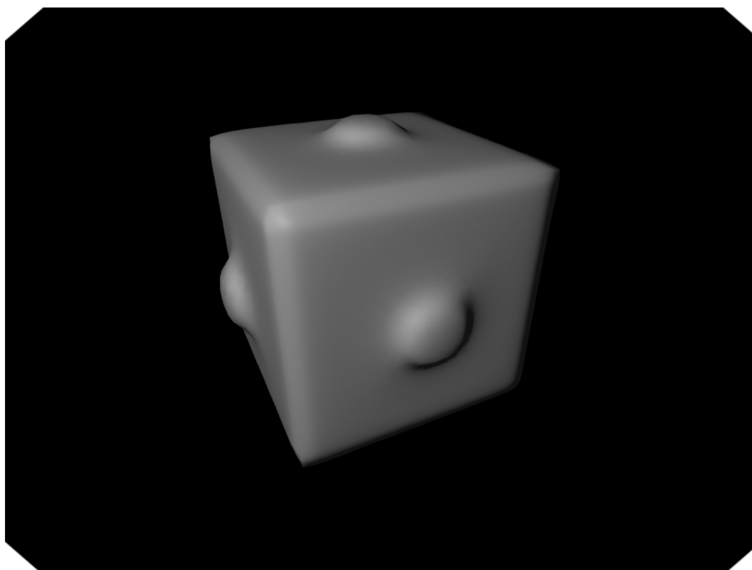
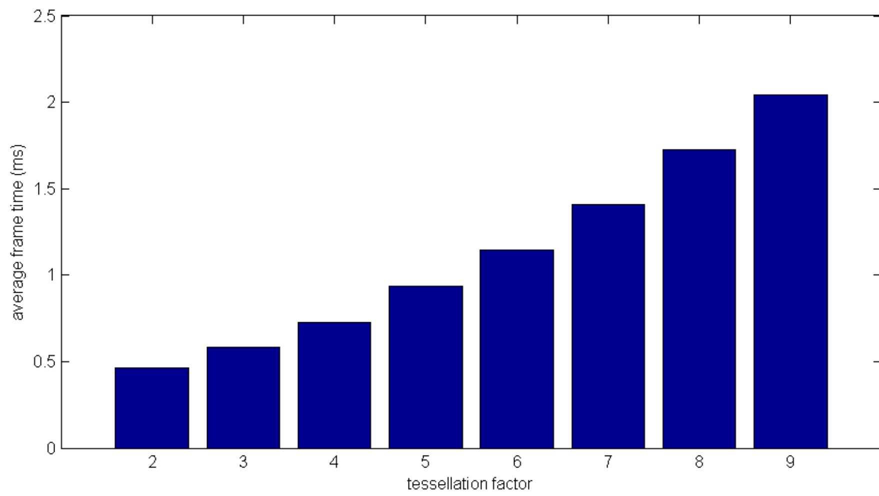
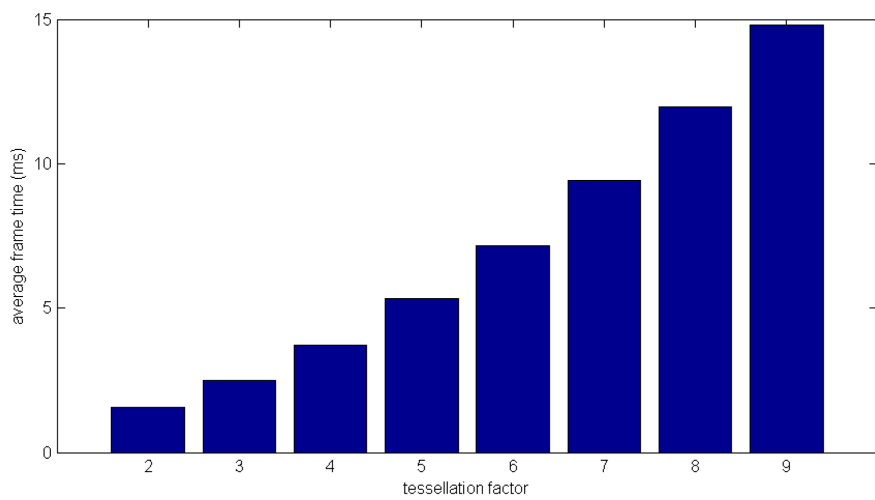


Figura 6.13: Oggetto utilizzato nei test sulla tecnica di displacement mapping

I risultati che si ottengono per il frame time medio sono riportati in fig. 6.14. Come si nota, abbiamo una crescita quadratica del tempo medio di tracciamento di un



(a) Singolo oggetto, 108 poligoni



(b) Matrice di 8 oggetti ($2 \times 2 \times 2$), 864 poligoni

Figura 6.14: Prestazioni della tecnica di displacement mapping al variare di TF

fotogramma in funzione di TF (fenomeno consistente con la struttura del geometry shader vista in sez. 4.6 e col fatto che il numero di poligoni emessi per ogni triangolo in ingresso è pari a TF^2). Notiamo inoltre che le prestazioni degradano velocemente al crescere della complessità geometrica degli oggetti a cui viene applicata questa tecnica.

6.3 Proiezione di Ombre

In questa sezione esaminiamo le prestazioni dell'algoritmo di soft shadow mapping con tecnica EVSM implementato all'interno della VR3Lib. Per misurare l'incremento del frame time dovuto all'utilizzo della funzionalità di shadow mapping, si lavora nuovamente con una telecamera mobile e si confrotano i risultati nel caso di shadow mapping abilitato e disabilitato. I test sono stati effettuati con la scena rappresentata in fig. 3.17 in basso: una stanza dove la sorgente d'ombra è posizionata all'esterno e la luce entra da due aperture rettangolari presenti nella parete. Una diversa immagine della stessa scena è fornita in fig. 6.15, questa scena è composta da 10572 poligoni.

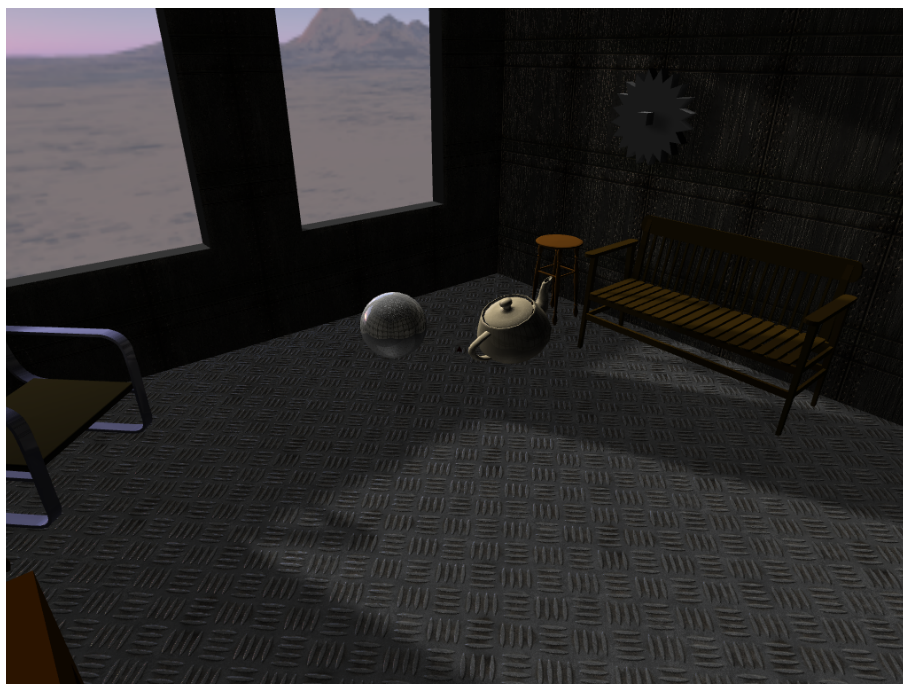


Figura 6.15: Scena utilizzata nei test sulla tecnica di soft shadow mapping

Alcuni oggetti visualizzati nella scena utilizzata subiscono uno shading basato su diffuse texture e normal mapping. Quello che a noi interessa però è semplice-

mente la differenza di prestazioni tra quando si utilizza la funzionalità di shadow mapping (come nell'immagine sopra) e quando invece se ne fa a meno. Tutti i test in questa sezione sono stati effettuati lavorando con una singola sorgente d'ombra e producendo una shadow map di 512×512 pixel.

La telecamera è stata collocata al centro della stanza e viene fatta ruotare su se stessa attorno all'asse verticale di una quantità pari a 0.1 gradi ad ogni frame: una rotazione completa viene ultimata in 3600 frame e dunque quello che viene misurato è l'andamento del frame time in questi 3600 cicli. Per questo test si utilizza l'algoritmo di EVSM secondo quanto visto in sez. 4.7, con filtraggio gaussiano offline avente kernel di dimensione 3×3 (il minimo possibile). Ricordiamo che il raggio del filtro gaussiano utilizzato viene configurato per ogni singola sorgente d'ombra nel controllore delle ombre e il parametro impostato corrisponde a:

$$filter\,size = \frac{kernel\,size - 1}{2}$$

E visto che il kernel è almeno di dimensione 3, abbiamo che il minimo valore del filter size è 1. La variabile uniform `filterSize` vista nel par. 4.7.2 viene allora impostata dalla VR3Lib col valore:

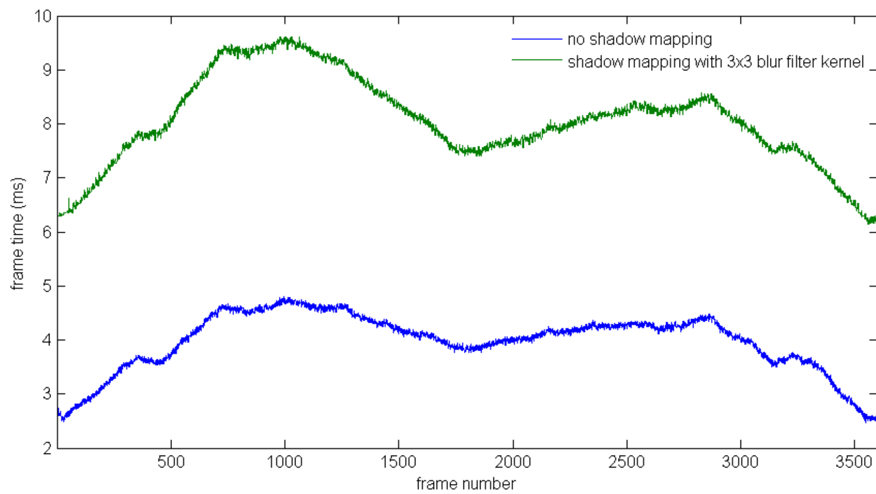
$$\texttt{filterSize} = filter\,size - 1$$

Gli andamenti ottenuti per le due risoluzioni 1024×768 e 1920×1080 sono riportati in fig. 6.16.

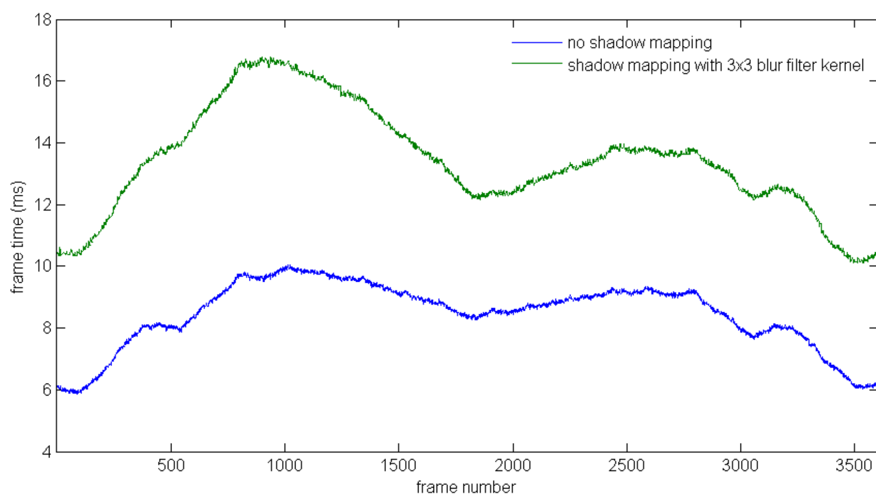
Nonostante questi test siano stati effettuati con il minimo filter size configurabile, l'overhead derivante dalla proiezione di ombre risulta comunque considerevole. L'utilizzo di tecniche efficienti per la proiezione di ombre realistiche è quindi fondamentale (e l'algoritmo di EVSM è una delle tecniche più efficienti attualmente disponibili). L'overhead introdotto è comunque gestibile nella maggior parte di casi se si utilizza un filter size non troppo elevato (ottenendo comunque regioni di penombra molto soddisfacenti).

Le oscillazioni che si osservano nelle curve in fig. 6.16 derivano dal fatto che ad ogni istante la telecamera ha il campo visivo rivolto verso una diversa porzione della stanza, e dunque il frame time verrà influenzato in modo significativo anche dalla porzione di scena (dal sottoinsieme di oggetti) visibile dalla telecamera per quel fotogramma.

Il parametro fondamentale per le prestazioni dell'algoritmo di shadow mapping è la dimensione del filtro di blur gaussiano utilizzato. Per esaminare le differenze di performance al variare di questo parametro, la telecamera è stata posizionata



(a) 1024×768



(b) 1920×1080

Figura 6.16: Prestazioni della tecnica di soft shadow mapping

fissa all'interno della stanza, orientata come in fig. 3.17. Si misura a questo punto il tempo di tracciamento medio al variare delle dimensioni del filtro, considerando 1000 fotogrammi. Questo test è stato effettuato con risoluzione del viewport OpenGL e dimensione della finestra di 1024×768, e i risultati sono riportati in fig. 6.17.

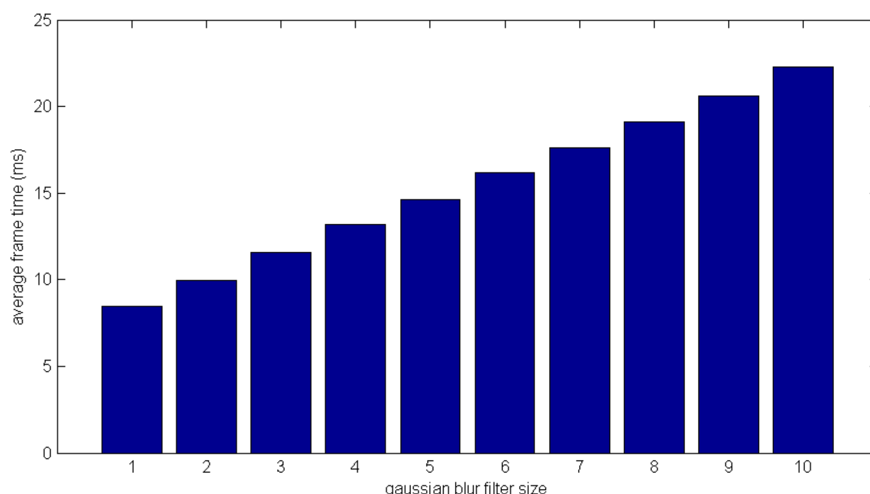


Figura 6.17: Prestazioni della tecnica di shadow mapping al variare del filter size

Come si nota, l'andamento è lineare in funzione delle dimensioni del kernel del filtro, in accordo con quanto detto nel par. 4.7.2 a proposito della complessità computazionale di filtraggio in due passate con l'utilizzo di un filtro separabile. Notare che il massimo valore per la dimensione del filtro è 10, per come sono stati realizzati gli shader che si occupano di effettuare il filtraggio. Ricordando che la tecnica EVSM è una delle più efficienti attualmente disponibili, e osservando i livelli raggiunti dal frame time per alti valori delle dimensioni del filtro, è ovvio che una tecnica di filtraggio online come quella presentata nel par. 3.3.1 (che risulta inferiore all'algoritmo EVSM in termini di prestazioni) permetterebbe di sfruttare solamente kernel di dimensioni molto contenute (e dunque otterremmo penombre di qualità inferiore).

6.4 Simulazione Fisica

Come ultimo aspetto per la valutazione delle prestazioni della nuova libreria, esaminiamo in che modo l'utilizzo delle funzionalità di sumulazione fisica influenza il tempo di tracciamento di un frame all'interno del ciclo di rendering dell'applicazione.

Si ricorda che il motore Nvidia PhysX ha una struttura multithreaded e tutti i calcoli per la simulazione dei corpi rigidi nella scena avvengono in flusso di esecuzione separato da quello principale dell'applicazione. Visto che sulla macchina utilizzata per i test è presente un processore con diversi core, ci si aspetta che la simulazione fisica possa procedere in parallelo al rendering del frame (in due thread separati). Questo significa sostanzialmente che non dovrebbero esserci incrementi significativi nel tempo di tracciamento di un fotogramma (trascurando la vera e propria simulazione a carico di un thread generato da PhysX, le operazioni svolte dalla libreria sono infatti molto simili al caso in cui si fa a meno della simulazione fisica).

Per verificare quanto sopra, si è posizionata la telecamera al centro della stanza in fig. 6.15, facendola ruotare come descritto in sez. 6.3. Si confronta quindi l'andamento del frame time nel caso di scena simulata con quello ottenuto nella scena non simulata. È importante notare che quando si testano le prestazioni in caso di simulazione fisica è necessario fare in modo che gli oggetti continuino a muoversi per tutta la durata della simulazione: motori avanzati come PhysX provvedono infatti a rendere la simulazione il più efficiente possibile mandando gli oggetti dinamici in uno stato di *sleep* quando si fermano per un certo tempo (e questa operazione evita di simulare ulteriormente gli oggetti dormienti fino a che una nuova azione non viene applicata su di essi, come una forza esterna). Mantenere gli oggetti dinamici in continuo movimento forza il motore fisico a continuare la simulazione per tutti gli oggetti.

I risultati ottenuti sono illustrati in fig. 6.18.

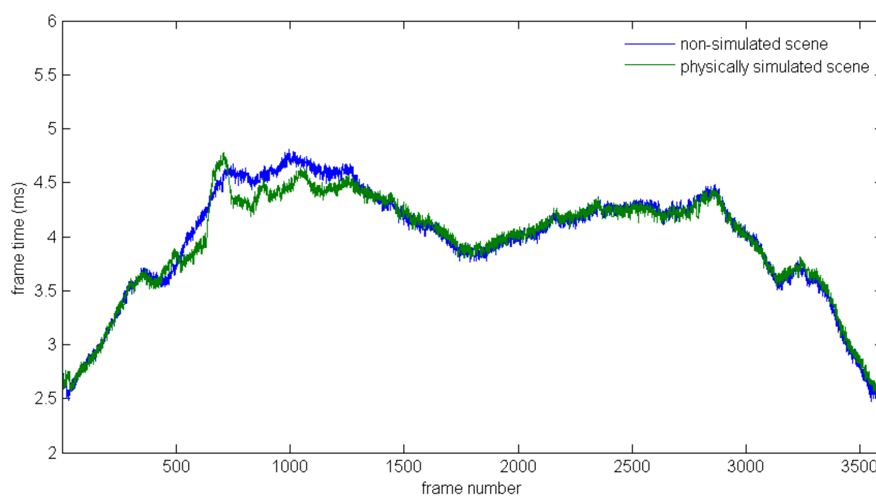


Figura 6.18: Prestazioni del modulo di simulazione fisica

Come si nota, gli andamenti sono molto simili. Questo deriva dal fatto che la vera simulazione fisica avviene in un thread separato da quello di rendering, che procede in parallelo al ciclo principale dell'applicazione. Le differenze visibili nella prima parte della simulazione sono dovute al movimento degli oggetti nella scena: visto che si osservano oggetti in movimento, questi escono ed entrano nel volume di vista in base alla loro evoluzione fisica e dunque comportano un carico diverso (nel tempo) sullo stadio di fragment processing rispetto al caso non simulato (in cui gli oggetti sono immobili).

Capitolo 7

Conclusioni e Futuri Sviluppi

7.1 Conclusioni

In questo testo è stato affrontato il problema di realizzare una libreria per la visualizzazione interattiva di ambienti virtuali complessi, capace di animare gli oggetti come se fossero corpi rigidi, simulando le interazioni fisiche tra di essi. Nella nuova libreria sono state inserite svariate tecniche di rendering di ultima generazione che possono essere utilizzate congiuntamente per ottenere un risultato molto realistico.

Il risultato del lavoro svolto è un modulo software disponibile sotto forma di libreria (la VR3Lib) che offre all'utente un'interfaccia molto semplice ereditata ed estesa dalla vecchia VRLib. Questo oggetto incapsula complessi shader utilizzati durante il rendering, interagisce con alcune librerie di secondo livello (GLEW per la gestione delle estensioni e DevIL per il caricamento delle immagini), e sfrutta i servizi offerti dal motore di simulazione fisica PhysX della Nvidia per fare evolvere la scena in modo fisicamente realistico.

La nuova libreria è idonea ad essere utilizzata per un'ampia gamma di applicazioni interattive di realtà virtuale:

- sistemi immersivi di realtà virtuale, come quelli su cui si lavora nel laboratorio PERCRO,
- simulatori per l'attività di macchinari automatizzati,
- complesse interfacce utente tridimensionali,
- videogiochi,
- ecc...

Figura 7.1: Un'applicazione costruita con la VR3Lib

Inoltre, dato che l'interfaccia alle funzionalità della VR3Lib è molto simile a quella della precedente VRLib, ci si aspetta una semplice integrazione della nuova libreria all'interno del framework XVR. XVR è il sistema di sviluppo di applicazioni di realtà virtuale utilizzato nel laboratorio PERCRO della Scuola Superiore S. Anna, di cui abbiamo discusso in sez. 1.2. Sarà tuttavia necessario estendere le potenzialità dell'ambiente per sfruttare tutte funzionalità messe a disposizione dalla nuova libreria.

Tutta la libreria è stata realizzata tenendo presente l'efficienza delle operazioni come uno dei requisiti fondamentali, e i risultati dei test risultano infatti soddisfacenti (considerando anche la macchina utilizzata per gli stessi). Nei capitoli 2 e 3 sono state esaminate svariate tecniche di rendering, mirando ad ottenere algoritmi efficienti capaci di fornire risultati soddisfacenti in tempo reale (per un utilizzo interattivo dell'applicazione).

Molte delle immagini di scene virtuali presenti in questo testo sono state ottenute sfruttando la nuova libreria: il suo utilizzo per mostrare i vari effetti disponibili nativamente è infatti molto semplice e rapido. Inoltre la VR3Lib risulta flessibile in quanto è utilizzabile per realizzare qualsiasi effetto di rendering, fornendo gli shader necessari.

In fig. 7.1 si riportano due ulteriori immagini di una scena virtuale visualizzata e simulata tramite la nuova libreria. Si tratta come al solito di rendering in tempo reale, e l'applicazione che ne risulta è in grado di interagire con l'utente mentre la scena evolve in modo fisicamente plausibile. Nella costruzione dell'ambiente virtuale riportato in fig. 7.1 sono state utilizzate molte delle tecniche illustrate in questo testo, sfruttando solamente la semplicissima interfaccia esportata dalla VR3Lib.

7.2 Futuri Sviluppi

La libreria ottenuta può essere estesa in molti modi. Alcune alternative per futuri sviluppi sono state accennate durante la trattazione dei precedenti capitoli, e in questa sezione si cercano di riassumere le varie direzioni percorribili.

- *Integrazione della libreria nel sistema XVR*
  La VR3Lib è attualmente un modulo software indipendente per lo sviluppo di applicazioni interattive di realtà virtuale. La nuova libreria è stata tuttavia costruita mantenendo un'interfaccia simile alla precedente libreria VRLib utilizzata nell'ambito del sistema XVR per lo sviluppo di applicazioni di realtà

virtuale. In futuro si potrebbe quindi pensare di effettuare la vera e propria integrazione della VR3Lib nel sistema XVR, sostituendola o affiancandola alla precedente libreria.

- *Supporto ad animazioni e character*
  Abbiamo già discusso del fatto che l'attuale struttura della VR3Lib non supporta nativamente le animazioni predeterminate e il concetto di character: in futuro potrebbe essere interessante reintrodurre queste funzionalità (presenti nella precedente libreria). Le animazioni sarebbero quindi un nuovo modo per far evolvere gli oggetti, diverso rispetto alla simulazione fisica degli stessi.

- *Potenziamento delle tecniche di rendering degli oggetti virtuali*
  Alle tecniche di rendering degli oggetti virtuali integrate nella nuova libreria (presentate nel capitolo 2) possono venire affiancati altri metodi moderni utilizzabili per ottenere risultati realistici. Ad esempio potremmo introdurre la tecnica di *parallax mapping* (con lo scopo di aumentare il realismo degli oggetti virtuali senza incrementare il numero di poligoni, introdotta in [25]) di cui ad oggi sono disponibili numerose varianti. Potrebbe inoltre essere interessante potenziare le tecniche attualmente disponibili nella VR3Lib, ad esempio estendendo la gestione del normal mapping per consentirne il funzionamento anche in caso di tangent-space normal map. Con le più recenti versioni dei sistemi grafici OpenGL e Direct3D, vengono inoltre messi a disposizione del programmatore due nuovi stadi programmabili predisposti per il tassellamento della geometria, che potrebbero venire utilizzati per implementare la tecnica di displacement mapping in modo più efficiente, cercando di ottenere prestazioni migliori rispetto a quelle risultanti dall'utilizzo dei soli geometry shader. La tecnica di displacement mapping si potrebbe inoltre ottimizzare sfruttando un algoritmo di tassellamento adattivo (dipendente dal punto di vista dell'osservatore). Potremmo addirittura sfruttare lo shadow mapping per ottenere self-shadowing sulla base della displacement map (qualora le prestazioni ottenute ottimizzando il displacement mapping lo permettessero).

- *Potenziamento della tecnica EVSM per la proiezione di ombre*
  La tecnica attualmente utilizzata per la proiezione di ombre è una delle più avanzate disponibili in letteratura. Tuttavia bisogna notare che la versione implementata può venire potenziata sfruttando congiuntamente altri metodi che ne permettono una applicazione più generale. Ad esempio, potremmo

sfruttare la tecnica CSM (cascaded shadow maps) in combinazione con l'attuale algoritmo di EVSM per ottenere una tecnica CEVSM capace di generare ombre realistiche anche in vasti paesaggi dove si richiedono ombre dinamiche (in tale situazione con la tecnica attuale può non essere possibile configurare i parametri in modo soddisfacente, volendo ottenere buone prestazioni). Un'altra possibile direzione per futuri sviluppi dell'algoritmo di shadow mapping è quella di gestire nativamente le sorgenti d'ombra omnidirezionali (come visto nel par. 3.2.4).

- *Estensione della VR3Lib a diversi sistemi operativi*
  La VR3Lib è stata scritta sfruttando per quanto possibile funzionalità standard e librerie cross-platform, in modo tale da essere facilmente estendibile a diversi sistemi operativi. In futuro si prevede che estensioni della VR3Lib ad altri sistemi operativi si dirigano verso Linux e altri sistemi Unix-like. Nonostante il motore fisico Nvidia PhysX non sia disponibile su tali piattaforme nelle ultime versioni, si potrebbe pensare di usare un'altra soluzione per la simulazione fisica. Per quanto riguarda il rendering di testo, le funzionalità attualmente implementate nella libreria sfruttano servizi disponibili solo su piattaforme Windows, ma l'estensione a sistemi basati su X11 dovrebbe essere molto semplice.

- *Potenziamento del modulo di simulazione fisica*
  Attualmente, tutti i corpi simulati con l'ausilio del motore PhysX vengono gestiti come corpi rigidi. Tuttavia il motore Nvidia ci consente di simulare anche oggetti più complessi come corpi molli (*soft bodies*), tessuti (*clothes*) e fluidi (*fluids*). La simulazione di questi oggetti complessi può anche essere condotta sfruttando le capacità di calcolo della GPU su alcuni adattatori grafici (ad esempio quelli aventi architettura CUDA[29]).

- *Gestione di oggetti virtuali trasparenti*
  Una tematica interessante che è rimasta aperta dai precedenti capitoli è la gestione di oggetti che presentano trasparenze. Visualizzare oggetti trasparenti con la tecnica di alpha blending vista in sez. 2.3 è complesso; il problema non viene affrontato direttamente con la VR3Lib attuale, ma in molte situazioni l'utente potrà risolvere la questione disegnando gli oggetti nel giusto ordine. Si

[29] CUDA sta per 'Compute Unified Device Architecture' e si tratta di un'architettura per il calcolo parallelo sviluppata da Nvidia per le proprie GPU.

potrebbe pensare di inserire un modulo apposito nella libreria per la gestione degli oggetti trasparenti, che potrebbe occuparsi di disegnare i singoli poligoni nell'ordine corretto (limitando il più possibile gli artefatti residui), oppure sfruttare un altro approccio completamente diverso per il rendering di oggetti trasparenti.

- *Utilizzo di shader precompilati*
  È interessante notare che gli shader presenti nella libreria sono disponibili alla stessa come stringhe di codice sorgente GLSL. Ad ogni avvio di un'applicazione che sfrutta la VR3Lib è perciò necessario costruire il codice completo per gli shader utilizzati, compilarli e collegarli per ottenere gli eseguibili da sfruttare durante il rendering. Questo comporta una certa perdita di tempo allo startup dell'applicazione che potrebbe essere evitata sfruttando shader precompilati e collegati che vengano direttamente caricati dalla libreria all'avvio dell'applicazione. Nonostante questa possibilità non venga contemplata nelle specifiche delle funzionalità OpenGL 3.3 di base, esiste un'estensione (`ARB_get_program_binary`, approvata dall'ARB nel Giugno 2010) che consente il salvataggio e successivo recupero di rappresentazioni binarie di programmi di shading che possono venire usate direttamente per il rendering. Le funzionalità dell'estensione sono state aggiunte alle specifiche di base OpenGL con la versione 4.1 del sistema grafico. L'applicazione potrebbe allora compilare e collegare tutti gli shader al primo avvio o durante l'installazione e successivamente utilizzare direttamente le versioni precompilate; questo risulterebbe in un notevole risparmio di tempo allo startup. Visto che comunque il tempo di avvio non è uno dei parametri fondamentali nelle applicazioni interattive di realtà virtuale, questa ottimizzazione è di importanza secondaria.

In un settore in rapida evoluzione come quello della realtà virtuale e grafica tridimensionale, ci si aspetta che in pochi anni molte delle tecniche implementate nella VR3Lib vengano superate o migliorate. Per questo motivo la libreria costruita dovrà subire una continua manutenzione al fine di risultare competitiva anche in futuro.

Bibliografia

[1] M. Carrozzino, F. Tecchia, S. Bacinelli, C. Cappelletti, M. Bergamasco. *Lowering the development time of multimodal interactive applications: the real-life experience of the XVR project*. Proceedings of ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, 2005.

[2] M. Segal, K. Akeley. *The OpenGL Graphics System: A Specification*. Version 2.1, 2006.

[3] M. Segal, K. Akeley. *The OpenGL Graphics System: A Specification*. Version 3.3 (Core Profile), 2010.

[4] J. Kessenich, D. Baldwin, R. Rost. *The OpenGL Shading Language*. Version 3.30, 2010.

[5] M. Segal, K. Akeley. *The OpenGL Graphics System: A Specification*. Version 4.1 (Core Profile), 2010.

[6] J. Kessenich, D. Baldwin, R. Rost. *The OpenGL Shading Language*. Version 4.10, 2010.

[7] D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Also known as *The Redbook*. Addison-Wesley Professional, 2009.

[8] H. Gouraud. *Continuous Shading of Curved Surfaces*. IEEE Transactions on Computers, vol. C-20, n. 6, p. 623-629, 1971.

[9] B. T. Phong. *Illumination for Computer Generated Pictures*. Communications of the ACM, vol. 18, n. 6, p. 311-317, 1975.

[10] J. F. Blinn. *Models of light reflection for computer synthesized pictures.* Proceedings of the 4th annual conference on Computer Graphics and Interactive Techniques, p. 192-198, 1977.

[11] J. Arvo. *Backward Ray Tracing.* Developments in Ray Tracing, SIGGRAPH 1986 Course Notes, vol. 12, p. 259-263, 1986.

[12] C. M. Goral, K. E. Torrance, D. P. Greenberg, B. Battaile. *Modeling the Interaction of Light Between Diffuse Surfaces.* ACM SIGGRAPH Computer Graphics, vol. 18, issue 3, p. 213-222, 1984.

[13] H. W. Jensen. *Global Illumination using Photon Maps.* Rendering Techniques 1996, p. 21-30, 1996.

[14] J. Hoberock, Y. Jia. *High-Quality Ambient Occlusion.* GPU Gems 3, p. 257-285, 2007.

[15] C. Dachsbacher, M. Stamminger, G. Drettakis, F. Durand. *Implicit Visiblity and Antiradiance for Interactive Global Illumination.* ACM Transactions on Graphics (SIGGRAPH Conference Proceedings), vol. 26, n. 3, art. n. 61, 2007.

[16] N. Green. *Environment Mapping and Other Applications of World Projections.* IEEE Computer Graphics and Applications, vol. 6, issue 11, p. 21-29, 1986.

[17] P. Debevec. *Image-Based Lighting.* IEEE Computer Graphics and Applications, vol. 22, issue 2, p. 26-34, 2002.

[18] Nvidia Corporation. *Technical Brief: Perfect Reflections and Specular Lighting Effects With Cube Environment Mapping.* Online document, 2000.

[19] C. Bloom. *Cube Map Lighting.* Online document, 2004.

[20] F. Houlmann, S. Metz. *High Dynamic Range Rendering in OpenGL.* UTBM University of Technology, online document, 2006.

[21] E. Johansson-Evegård. *Image-Based Lighting and Tone Mapping.* Online document, 2009.

[22] J. F. Blinn. *Simulation of Wrinkled Surfaces.* ACM SIGGRAPH Computer Graphics, vol. 12, issue 3, p. 286-292, 1978.

[23] M. I. Gold, Nvidia Corporation. *Emboss Bump Mapping.* Game Developers Conference, 1999.

[24] J. Cohen, M. Olano, D. Manocha. *Appearance-Preserving Simplification.* International Conference on Computer Graphics and Interactive Techniques, p. 115-122 , 1998.

[25] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, S. Tachi. *Detailed Shape Representation with Parallax Mapping.* Proceedings of the ICAT, 2001.

[26] L. Szirmay-Kalos, T. Umenhoffer. *Displacement Mapping on the GPU - State of the Art.* Computer Graphics Forum, vol. 27, n. 6, p. 1567-1592, 2008.

[27] N. Suni. *Dynamic Patch Tessellation.* D3DBook, online document, 2008.

[28] A. Woo, P. Poulin, A. Fournier. *A Survey of Shadow Algorithms.* IEEE Computer Graphics and Applications, vol. 10, issue 6, p.13-32, 1990.

[29] A. V. Nealen. *Shadow Mapping and Shadow Volumes: Recent Developments in Real-Time Shadow Rendering.* University of British Columbia, project report, 2002.

[30] L. Williams. *Casting Curved Shadows on Curved Surfaces.* ACM SIGGRAPH Computer Graphics, vol. 12 , issue 3, p. 270-274, 1978.

[31] F. C. Crow. *Shadow algorithms for computer graphics.* ACM SIGGRAPH Computer Graphics, vol. 11 , issue 2, p. 242-248, 1977.

[32] C. Everit, M. J. Kilgard. *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering.* Nvidia white paper, 2002.

[33] T. Akenine-Möller, U. Assarsson. *Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges.* ACM International Conference Proceeding Series, vol. 28, p. 297-306, 2002.

[34] U. Assarsson, T. Akenine-Möller. *A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware.* ACM Transactions on Graphics (TOG), vol. 22, issue 3, p. 511-520, 2003.

[35] V. Forest, L. Barthe, M. Paulin. *Realistic Soft Shadows by Penumbra-Wedges Blending*. SIGGRAPH / EUROGRAPHICS Conference On Graphics Hardware, p. 39-46, 2006.

[36] M. D. McCool. *Shadow volume reconstruction from depth maps*. ACM Transactions on Graphics (TOG), vol. 19, issue 1, p. 1-26, 2000.

[37] M. Kilgard. *Shadow Mapping with Today's OpenGL Hardware*. Presentation at CEDEC, 2001.

[38] M. Stamminger, G. Drettakis. *Perspective Shadow Maps*. International Conference on Computer Graphics and Interactive Techniques, p. 557-562. 2002.

[39] S. Kozlov. *Perspective Shadow Maps: Care and Feeding*. GPU Gems, p. 217-244, 2004.

[40] R. Fernando, S. Fernandez, K. Bala, D. P. Greenberg. *Adaptive Shadow Maps*. International Conference on Computer Graphics and Interactive Techniques, p. 387-390, 2001.

[41] T. Martin, T.-S. Tan. *Anti-aliasing and Continuity with Trapezoidal Shadow Maps*. Eurographics Symposium on Rendering, p. 153-160, 2004.

[42] M. Wimmer, D. Scherzer, W. Purgathofer. *Light Space Perspective Shadow Maps*. Eurographics Symposium on Rendering, 2004

[43] W. Engel. *Cascaded Shadow Maps*. ShaderX^5- Advanced Rendering Techniques, p. 197-206, 2006.

[44] R. Dimitrov, Nvidia Corporation. *Cascaded Shadow Maps*. Online document, 2007.

[45] P. S. Gerasimov. *Omnidirectional Shadow Mapping*. GPU Gems, p. 193-203, 2004.

[46] E. Miandji. *Cubic Shadow Mapping in Direct3D*. Online document, 2008.

[47] L. Bavoil, Nvidia corporation. *Advanced Soft Shadow Mapping Techniques*. Presentation at GDC, 2008.

[48] W. T. Reeves, D. H. Salesin, R. L. Cook. *Rendering Antialiased Shadows with Depth Maps*. International Conference on Computer Graphics and Interactive Techniques, p. 283-297, 1987.

[49] Y. Uralsky, Nvidia Corporation. *Efficient Soft-Edged Shadows Using Pixel Shader Branching*. GPU Gems 2, p. 269-282, 2005.

[50] R. Fernando, Nvidia Corporation. *Percentage-Closer Soft Shadows*. International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH Sketches, 2005.

[51] A. Lauritzen. *Summed-Area Variance Shadow Maps*. GPU Gems 3, p. 157-182, 2007.

[52] B. Yang, Z. Dong, J. Feng, H.-P. Seidel, J. Kautz. *Variance Soft Shadow Mapping*. Pacific Graphics, vol. 29, n. 7, 2010.

[53] W. Donnelly, A. Lauritzen. *Variance Shadow Maps*. Symposium on Interactive 3D Graphics and Games, p. 161-165. 2006.

[54] A. Lauritzen, M. McCool. *Layered Variance Shadow Maps*. Proceedings of graphics interface, p. 139-146, 2008.

[55] M. Salvi. *Rendering Filtered Shadows with Exponential Shadow Maps*. ShaderX^6- Advanced Rendering Techniques, p. 257-274, 2008.

[56] T. Annen, T. Mertens, H.-P. Seidel, E. Flerackers, J. Kautz. *Exponential Shadow Maps*. Proceedings of graphics interface, p. 155-161, 2008.