

UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

ANALISI E SVILUPPO DI SHADER GRAFICI
IN LINGUAGGIO GLSL

ANALYSIS AND DEVELOPMENT
OF GRAPHICS SHADERS USING GLSL

Relatore:
Ing. JACOPO ALEOTTI

Tesi di:
RICCARDO MONICA

22 settembre 2011

Ringrazio il mio relatore, il Dott. Ing. Jacopo Aleotti, non solo per il suo aiuto e la sua disponibilità, ma soprattutto perché con ogni probabilità è l'unica persona che mai leggerà questa tesi.

Indice

1	Introduzione	1
2	Il nuovo modello della GPU	4
2.1	La pipeline	4
2.2	Gli shader	7
2.3	Linguaggi di programmazione	10
2.4	Parallelismo	11
3	Implementazione in OpenGL	17
3.1	Versioni ed estensioni	17
3.2	Il concetto di program	19
3.3	Vertex shader	20
3.4	Fragment shader	22
3.5	Geometry shader	24
3.6	Il nuovo standard OpenGL	29
3.7	Tessellation control shader	31
3.8	Tessellation evaluation shader	35
4	Algoritmi grafici	40
4.1	Illuminazione	40
4.2	Calcolo delle normali	47
4.3	Bump mapping	50
4.4	Riflessione	53
4.5	Silhouette	56
4.6	Frattali	60
5	Conclusione	63

Capitolo 1

Introduzione

Il vantaggio dell'accelerazione grafica consiste nel trasferire alcuni dei compiti più ripetitivi e dispendiosi della visualizzazione dalla CPU a un hardware dedicato, la scheda video. Al giorno d'oggi, tutte le schede video per PC possono accettare in ingresso la descrizione di un oggetto tridimensionale e produrre autonomamente un'immagine (bidimensionale) sullo schermo. In più, è necessario che questa operazione sia eseguita più velocemente possibile, perché alcune applicazioni (per esempio i giochi per computer) richiedono che l'immagine sia modificata e rigenerata di continuo in tempo reale, decine di volte al secondo.

Siccome i passi da eseguire per trasformare un oggetto, proiettarlo su un piano e discretizzarlo in un numero finito di pixel non sono pochi, è ovvio che le schede video hanno bisogno di una elevata capacità di elaborazione. Ogni scheda video moderna comprende un proprio processore, la GPU, che si occupa dell'esecuzione di queste operazioni. Ormai, le GPU sono vicine alle CPU per complessità e numero complessivo di operazioni al secondo, nonché (purtroppo) per consumo e dissipazione di calore.

Tuttavia, per come sono nate e per le loro finalità, le GPU hanno un'architettura diversa da quella delle CPU. Hanno necessità di accedere continuamente alla memoria (per caricare texture, ad esempio), perciò devono disporre di un bus dati più ampio e sono comunque limitate dalla frequenza massima della RAM video. Inoltre, devono eseguire rapidamente calcoli vettoriali. In compenso, possono sfruttare un livello di parallelismo centinaia di volte superiore alle CPU, perché i dati sono

in genere scarsamente correlati tra loro.

Le GPU tradizionalmente eseguono il loro compito attraverso una sequenza fissa di operazioni, in gran parte rigidamente implementate nell'hardware del processore. Le operazioni costituiscono la "pipeline grafica", che trasforma le primitive di cui gli oggetti sono costituiti in colori dei pixel da inviare allo schermo. L'azione della pipeline può essere modificata con un certo numero di parametri, dal tipo di proiezione con cui l'oggetto è trasformato da 3D a 2D alle luci che lo illuminano, tuttavia la sequenza di operazioni deve essere sempre la stessa.

Questa è una grossa limitazione. Per creare nuovi effetti grafici sempre più realistici (o volutamente non realistici), la pipeline grafica deve essere modificata. Non è difficile capire, a questo punto, che invece di modificare l'hardware ogni volta è più conveniente implementare un processore di uso più generale che possa eseguire del codice caricato di volta in volta. Ciò significa renderla molto meno specifica, ma anche molto più adattabile. E si perde poco in efficienza: la GPU può essere più semplice e esegue solo gli effetti che servono.

In realtà, la possibilità di eseguire codice personalizzato sulle GPU ha cominciato ad essere disponibile come estensione su alcune schede video a partire dal 2002, ed è già diventata standard con le versioni di DirectX 9.0 (19 dicembre 2002) e di OpenGL 2.0 (7 settembre 2004). Piccoli frammenti di codice, detti "shader", possono essere caricati attraverso i driver video per sostituire ed ampliare parti della pipeline grafica. Negli anni successivi, gli standard sono stati modificati più volte e hanno reso programmabili parti sempre più ampie della pipeline. Parallelamente, sempre più funzioni sono state sottratte alla pipeline grafica originale, di conseguenza qualora fossero ancora necessarie devono essere reimplementate attraverso gli shader.

Tutto ciò ha portato a una piccola rivoluzione nel modo di sviluppare programmi di grafica. Lo sviluppo degli shader, cioè della parte del programma eseguita sulla GPU, è diventata la parte più importante e quella che decide gli effetti grafici che sono attivi. Il codice eseguito dalla CPU è solo quello necessario per caricare gli shader e inviare la geometria. Ciò ha anche l'effetto di liberare la CPU dalla gestione degli effetti grafici, perciò essa può occuparsi meglio della logica del programma e la simulazione fisica.

Lo scopo di questa tesi è appunto quello di analizzare questa innovazione radi-

cale, limitatamente allo standard OpenGL e al suo linguaggio di programmazione per shader, GLSL. Essa si compone fondamentalmente di tre capitoli. Il primo, intitolato “Il nuovo modello della GPU”, riporta le modifiche subite dal modello teorico della pipeline grafica e accenna anche ad alcuni dettagli dell’implementazione hardware, dove questa influisce sull’efficienza degli algoritmi. Il secondo, “Implementazione in OpenGL”, introduce le modifiche subite dal linguaggio OpenGL e gli shader in linguaggio GLSL. Il terzo, “Algoritmi grafici”, propone una serie di esempi di algoritmi grafici che, benché complessi, sono diventati particolarmente facili da implementare attraverso gli shader.

Capitolo 2

Il nuovo modello della GPU

2.1 La pipeline

La pipeline grafica può essere modellizzata come una sequenza di unità di elaborazione separate da buffer di elementi. Ciclicamente, ciascuna unità di elaborazione preleva uno o più elementi dal buffer precedente, esegue su di essi un (breve) programma, crea nuovi elementi (che possono essere di tipo diverso) e li consegna al buffer successivo. Questa operazione è ripetuta finché non ci sono più elementi nel buffer di origine.

Nella figura 2.1, è riportata una schematizzazione tradizionale della pipeline grafica. I riquadri rappresentano i buffer, con indicato il sistema di coordinate dei vertici tra parentesi. Le frecce che li collegano sono le fasi che elaborano e trasferiscono i dati, e le didascalie alla loro destra indicano le funzioni svolte in quella fase di elaborazione.

In sintesi, la descrizione delle primitive grafiche (solitamente, i triangoli che descrivono gli oggetti) proviene dal processore e si colloca nel primo buffer. Inizialmente le primitive sono descritte tramite le model coordinates nel sistema di riferimento dell'oggetto. Dopodiché esse attraversano le trasformazioni di modello e le trasformazioni di vista, che complessivamente le portano nelle coordinate riferite al punto di vista, le eye coordinates.

Qui il loro colore viene modificato a seconda dell'illuminazione. Passano poi attraverso le trasformazioni di proiezione, che applicano la prospettiva, il clipping

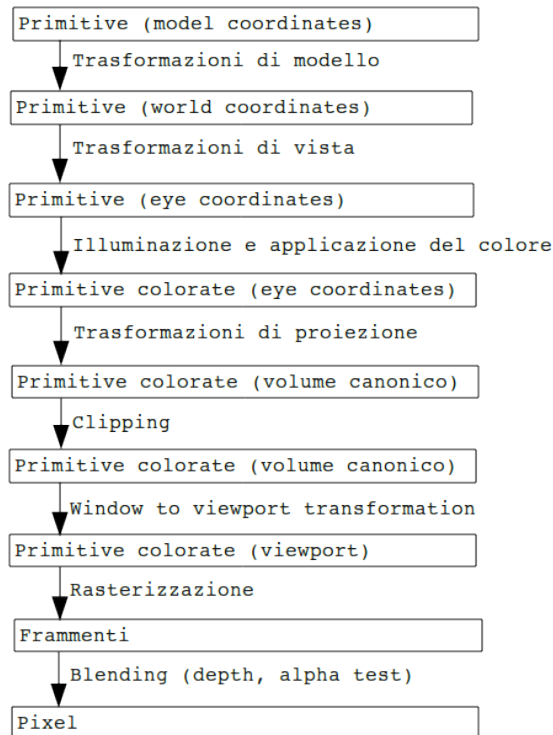


Figura 2.1: *La pipeline tradizionale.*

che rimuove le parti al di fuori del campo visivo. Le primitive acquisiscono quindi la corretta posizione sullo schermo (window to viewport) e sono discretizzate in frammenti (rasterizzazione), ciascuno dei quali corrisponde a una posizione di un pixel sullo schermo. Dei frammenti di varie primitive che possono essere disegnati su un pixel, viene selezionato quello che ha profondità (depth) minore, oppure viene calcolato un colore intermedio se ci sono effetti di trasparenza (alpha).

La figura 2.2 riporta una schematizzazione più specifica della pipeline grafica OpenGL com'era prima dell'introduzione degli shader programmabili. Per una migliore comprensione, sulla sinistra è stato aggiunto un riquadro che rappresenta sommariamente il contenuto del buffer a quel punto della pipeline.

Siccome questa pipeline rappresenta le funzionalità fisse delle GPU, modificabili solo attraverso alcuni parametri, viene detta "Fixed Function Pipeline" e sarà riferita con questo nome nel corso di tutta la trattazione seguente.

Ci sono alcune importanti differenze con lo schema precedente, tra cui:

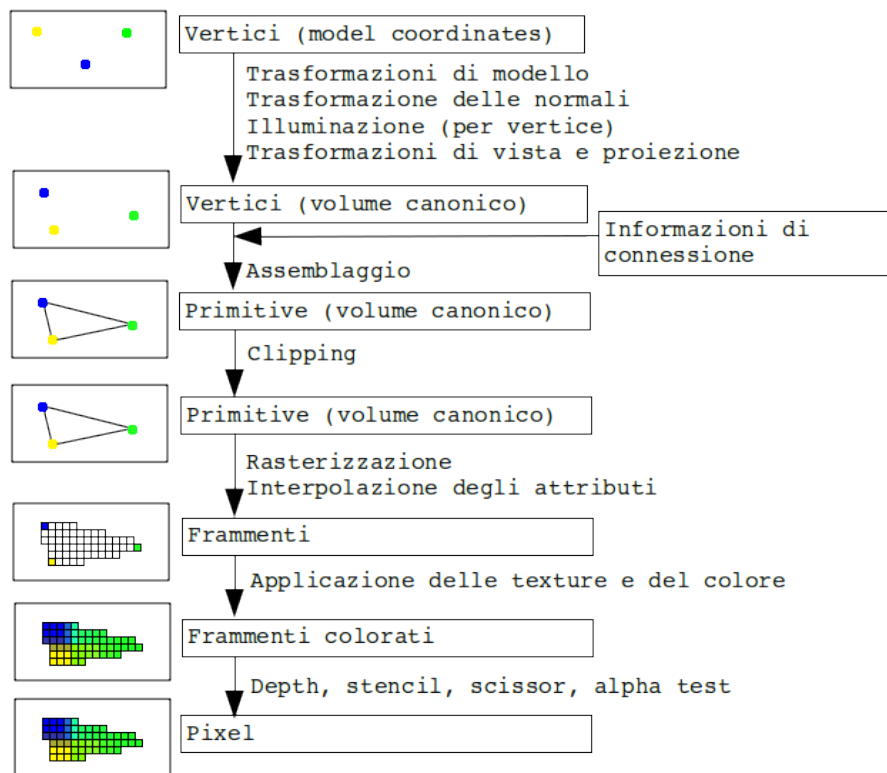


Figura 2.2: La Fixed Function Pipeline.

- Tutte le trasformazioni sui vertici sono state compresse in un unico passaggio, all'inizio della pipeline.
- Sono scomparse le trasformazioni di vista, che vengono assorbite dalle trasformazioni di modello o di proiezione.
- Le trasformazioni sui vertici sono eseguite senza conoscere le informazioni di connessione, cioè le primitive a cui quei vertici appartengono.
- Le informazioni di connessione sono recuperate ed aggiunte successivamente, nella fase di assemblaggio.
- Il colore dei vertici, le loro normali e l'illuminazione sono calcolati immediatamente all'inizio della pipeline.
- L'effettiva colorazione delle primitive avviene alla fine della pipeline, per interpolazione (lineare o prospettica) dei valori ai vertici.

2.2 Gli shader

Alcune delle unità di elaborazione della pipeline permettono di sostituire i programmi predefiniti (dal costruttore o dal driver) con programmi definiti dall'applicazione utente. Ciascuno di questi brevi programmi è chiamato Shader, e le unità che li eseguono sono dette Shader Processors. Uno shader è inizializzato ed eseguito una volta per ogni elemento (o gruppo di elementi) presente nel buffer di origine, e può in genere produrre un numero variabile (anche nessuno) di elementi in uscita. Il tipo di uno shader indica lo shader processor che dovrà eseguirlo, cioè il punto della pipeline in cui deve essere inserito per funzionare.

Il vantaggio principale dell'utilizzo degli shader programmabili, nella maggior parte delle applicazioni, è l'efficienza. Essi permettono di trasferire una certa quantità di elaborazione dalla CPU alla GPU. È un vantaggio notevole in termini di prestazioni, sia perché permette alla CPU di dedicarsi ad altri compiti (come la simulazione di complessi modelli fisici), sia perché la GPU è più ottimizzata per grandi quantità di calcolo parallelo. Tra l'altro, questo è il motivo per cui non esistono emulatori di shader processor che usano soltanto la CPU: sarebbero troppo lenti.

Come vantaggio secondario, capita spesso che con gli shader programmabili sia possibile implementare effetti che non sarebbe possibile ottenere altrimenti (non senza riscrivere l'intera pipeline grafica via software). Questo accade perché gli shader possono sfruttare informazioni parzialmente elaborate a cui normalmente non si ha accesso all'inizio della pipeline. Ad esempio, uno shader che agisce alla fine della pipeline è già a conoscenza dell'effettiva suddivisione in pixel che la primitiva avrà sullo schermo.

Nella figura 2.3, è mostrata l'architettura della pipeline che comprende i cinque tipi di shader definiti fino ad oggi, risalenti alla versione GLSL 4.0 del 10 marzo 2010. Queste specifiche sono cambiate e sono destinate a cambiare molto rapidamente, man mano che sempre più parti della pipeline diventano modificabili.

Per cominciare, si nota immediatamente che molte funzionalità della pipeline sono state inglobate all'interno degli shader e non sono più eseguite automaticamente. Tutte le trasformazioni sul singolo vertice sono ora inglobate all'interno del Vertex Shader. Allo stesso modo, tutte le funzioni che operano a livello del singolo

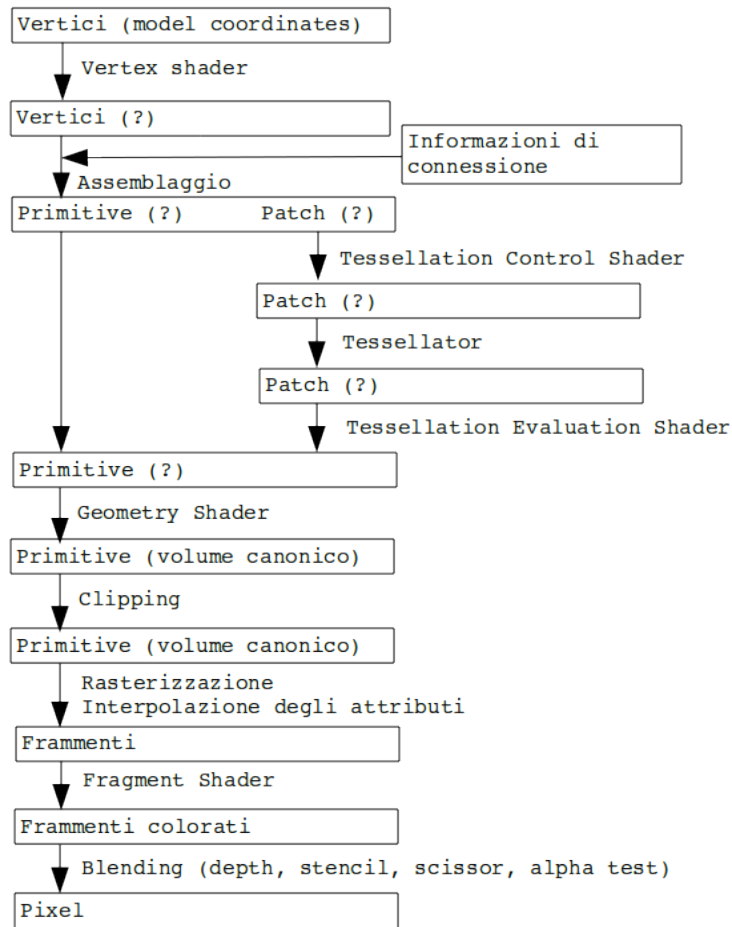


Figura 2.3: La pipeline e gli shader.

pixel per applicare il colore sono diventate parte del Fragment Shader.

Inoltre, è presente un percorso alternativo tra l'assemblaggio e il Geometry Shader, che viene percorso solo dalle primitive di tipo *patch*. Una *patch* è una primitiva che può avere un qualunque numero di vertici. Le *patch* passano attraverso il Tessellator, un procedimento che aumenta il numero di vertici della *patch* interpolando quelli esistenti. Può essere utile per migliorare dinamicamente la risoluzione della primitiva.

Si nota anche che sono comparsi dei punti di domanda dove prima erano indicate le coordinate di riferimento. Questo accade perché, siccome larga parte della pipeline è controllabile dal programmatore, è sufficiente che le coordinate siano passate in modo coerente da una fase a quella successiva e che il Geometry Shader

le produca nel volume canonico.

Segue una breve descrizione di ciascun tipo di shader. Queste tipologie saranno poi riprese nel capitolo 3 (Implementazione in OpenGL).

Vertex Shader Un vertex shader sostituisce le prime operazioni della pipeline grafica. Esso riceve in ingresso un singolo vertice, com'è definito dall'applicazione, e applica le trasformazioni di modello, di vista e di proiezione. Può inoltre modificare gli attributi, calcolare la normale e il colore.

Tessellation Control Shader (o Hull Shader) Un tessellation control shader si colloca dopo il vertex shader. Il tessellation control shader è istanziato una volta per ogni vertice della primitiva, ma al contrario del vertex shader può leggere tutti i vertici della patch durante la sua esecuzione, non solo i dati del vertice corrente. Lo scopo principale è quello di istruire il Tessellator, che ha il compito di suddividere ciascuna primitiva della patch in ulteriori primitive. Questo permette di aumentare dinamicamente la risoluzione della patch durante l'esecuzione.

Tessellation Evaluation Shader (o Domain Shader) Il Tessellation Evaluation Shader viene istanziato una volta per ogni vertice prodotto dal Tessellator e ha il compito di calcolare gli attributi di ciascuno dei vertici prodotti. Come il Tessellation Control Shader, ha accesso a tutti i vertici della primitiva.

Geometry Shader Un geometry shader si colloca subito dopo l'assemblaggio delle primitive. Esso riceve in ingresso una primitiva (definita come lista ordinata di vertici) e produce in uscita un qualunque numero di primitive.

Fragment Shader (o Pixel Shader) Un fragment shader è eseguito una volta per ogni frammento, cioè ciascuno dei pixel in cui una primitiva viene scomposta, prima del blending. Perciò un fragment shader è eseguito (a meno di ottimizzazioni) indipendentemente dal fatto che il suo frammento sia visualizzato oppure no (a causa del fallimento del depth test, per esempio).

2.3 Linguaggi di programmazione

Anche per gli shader, sono disponibili diversi linguaggi di programmazione, a diversi livelli di astrazione.

- Un linguaggio macchina;
- Un linguaggio assembly;
- Un linguaggio di alto livello.

Allo stato attuale, non esiste uno standard accettato da tutti i produttori per il linguaggio macchina.

In origine, quando è stato rilasciato OpenGL 2.0, era stato previsto che lo standard assembly sviluppato dall'Architecture Review Board (detto appunto assembly ARB) sarebbe diventato parte dello standard OpenGL. Così è stato, tuttavia esso supportava solamente i primi due tipi di shader (Vertex e Fragment) e non è più stato aggiornato nelle versioni di OpenGL successive. È diventato vecchio in breve tempo ed è raramente utilizzato, anche se le schede video NVIDIA supportano tutt'ora un suo discendente, l'assembly NVIDIA.

Per maggiore compatibilità, invece, sono stati sviluppati due linguaggi ad alto livello: GLSL (OpenGL Shading Language) da OpenGL, HLSL (High Level Shading Language) da Microsoft nello standard DirectX. Questi due linguaggi di programmazione sono accettati da pressoché tutti i driver per schede video recenti. Uno shader viene compilato dal driver e tradotto nel linguaggio macchina corretto per l'architettura della GPU. Perciò, gli shader devono essere compilati di volta in volta run-time sul computer dell'utente finale.

L'uso di GLSL o HLSL porta tutti i vantaggi classici della programmazione ad alto livello: codice più comprensibile, minor rischio di errore, minor tempo necessario per scrivere il codice, eccetera.

In più, l'uso di un linguaggio ad alto livello tende a produrre in media codice più ottimizzato. Infatti, anche se ci fosse uno standard condiviso, sarebbe impossibile sviluppare un programma assembly ottimizzato per tutte le architetture esistenti (e future). Invece, siccome il compilatore è all'interno del driver, esso conosce l'architettura della GPU e può eseguire le ottimizzazioni necessarie. Di solito, tutti i compilatori per linguaggi ad alto livello moderni contengono degli ottimizzatori molto

potenti, perché se il codice è più ottimizzato la scheda video sembra comunque più veloce all'utente finale.

Per superare la differenza tra GLSL e HLSL, è stato sviluppato da NVIDIA in collaborazione con Microsoft un linguaggio di livello ancora più alto, Cg (C for Graphics). Questo linguaggio può essere compilato sia in shader per DirectX che per OpenGL, o direttamente in linguaggio assembly per una scheda video.

2.4 Parallelismo

Come già accennato nell'introduzione, le GPU hanno un limite massimo in frequenza, dovuto alla necessità continua di accedere alla memoria. Tuttavia, i dati che devono essere elaborati sono per lo più indipendenti, perché ciascuno degli elementi da elaborare (un vertice, una primitiva, un pixel) può essere ottenuto correttamente anche senza avere informazioni sugli altri. Le interferenze avvengono solo alla fine della pipeline, in cui si deve trovare il frammento più vicino all'osservatore (depth test). In questa situazione, è vantaggioso aumentare le prestazioni tramite un elevato parallelismo delle operazioni.

Per ogni fase di elaborazione della pipeline, le GPU contengono molte (diverse centinaia) unità di elaborazione identiche, che hanno propria area di memoria temporanea e propri registri interni, ma che condividono l'unità di controllo. Queste unità sono dette (Shader) Processing Unit. A ciascuna Processing Unit è assegnato un diverso elemento presente nel buffer di origine e, quando un numero sufficiente di Processing Unit è pronta, tutti i dati sono elaborati contemporaneamente. Ovviamente, siccome l'unità di controllo è unica, unico è il programma (flusso di controllo) che le Processing Unit parallele possono eseguire.

Questa architettura è molto efficiente, perché l'esecuzione di un numero molto elevato di elaborazioni tutte uguali è molto comune nella pipeline grafica. Si consideri per esempio la colorazione dei singoli frammenti, che può richiedere molte operazioni simili per ogni singolo pixel della viewport (decine, centinaia di migliaia o anche milioni).

Un'architettura di questo tipo è un'implementazione del paradigma SIMD (Single Instruction Multiple Data). In particolare si parla della GPU come di uno stream

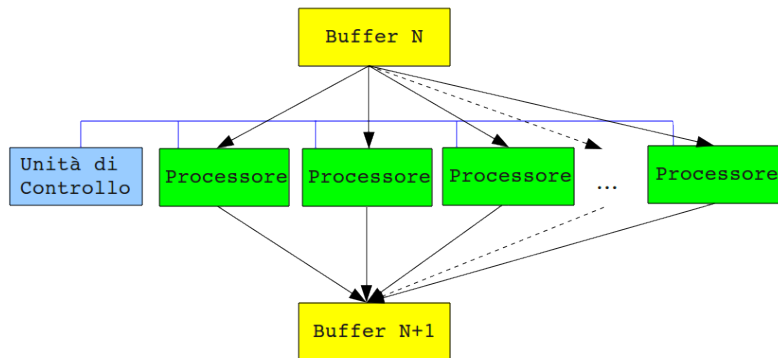


Figura 2.4: Le Shader Processing Unit condividono un'unica unità di controllo.

processor: dato uno stream di dati in ingresso (le primitive e le loro proprietà), essa applica la stessa operazione (sequenza di operazioni) a tutti i dati.

In teoria, lo standard OpenGL definisce (Shader) Processor diversi nella GPU per ogni fase della pipeline: Vertex Processor, Geometry Processor e così via. Tuttavia, dall'introduzione dello Shader Model versione 4.0 (Unified Shader Model), è stato imposto che tutti gli shader potessero eseguire le stesse identiche operazioni aritmetiche e operare sugli stessi tipi di dato. Così è diventato possibile usare lo stesso processore che di volta in volta si riconfigura come un diverso tipo di Shader Processor.

Nella pratica, ciascun modello di GPU ha la sua particolare architettura ed implementazione. Normalmente, sono presenti Shader Processor diversi, a volte ottimizzati per operazioni differenti, ma che possono eseguire qualsiasi tipo di shader e su cui i compiti della GPU sono allocati dinamicamente, a seconda di quali operazioni sono più richieste in quel momento.

Il parallelismo della GPU ha effetto sulla programmazione degli shader e porta alcuni svantaggi:

- Risulta impossibile comunicare dati tra shader allo stesso livello, perchè molto probabilmente sono eseguiti in parallelo.
- Non si può conoscere l'ordine in cui gli shader sono eseguiti, nè in fase di programmazione nè in fase di esecuzione. La GPU deve spesso cambiare l'ordine per ottimizzare e fare in modo che tutte le Processing Unit siano occupate.

- Le istruzioni di controllo di flusso (gli `if`, o i salti condizionati) sono il tipo di istruzione più problematico per l'architettura, perchè l'unità di controllo si ritroverebbe a produrre due sequenze diverse di segnali di controllo, destinate a diverse Shader Processing Unit. Siccome l'unità di controllo è unica, questo non è possibile.

I primi svantaggi non sono gravi e costituiscono una limitazione trascurabile per il programmatore degli shader. Ciononostante, di recente una funzione per ottenere una qualche sincronizzazione è stata aggiunta ai Tessellation Control Shader, perciò è possibile che la comunicazione tra shader sia aggiunta in futuro.

L'ultimo svantaggio invece costituisce un problema molto significativo. Sembrerebbe imporre che le istruzioni di controllo di flusso all'interno degli shader debbano essere rigorosamente vietate. Come esempio, sia dato lo pseudocodice C nel listato 2.1, dove `d1` e `d2` sono dati che provengono dall'esterno, diversi per ciascuna istanza dello shader.

```
float a = 1.0;
if (d1 == d2)
    a = f(); // f e' una generica funzione comunque complicata
```

Listato 2.1: Breve esempio di codice C.

In una ipotetica CPU, potrebbe essere compilato a livello assembly in qualcosa di simile a ciò che appare nel listato 2.2.

```
TEMPF a           ; dichiara la variabile temporanea float a
MOV a,1.0         ; la inizializza
CMP dl,d2        ; confronta
JNE @avanti      ; se non sono uguali, salta avanti (Jump if Not Equal)
    CALL f       ; altrimenti, esegue la funzione
    MOV a,reg1   ; e salva il risultato in a
@avanti:
```

Listato 2.2: Esempio di compilazione su una CPU.

Questo non può essere usato in una GPU, perché il Program Counter è unico e accessibile solo dall'unità di controllo, che decodifica e invia le stesse (identiche)

istruzioni a tutte le Processing Unit (listato 2.3). Invece, per le unità in cui $d1$ è diverso da $d2$, la chiamata a f non deve essere inviata e soprattutto il suo risultato deve essere ignorato e non salvato in a .

Processing Unit 1 ($d1 == d2$)	Processing Unit 2 ($d1 != d2$)
TEMPF a	TEMPF a
MOV a,1.0	MOV a,1.0
CMP d1,d2	CMP d1,d2
JNE @avanti ; non deve saltare	JNE @avanti ; deve saltare
CALL f	CALL f
MOV a,reg1	MOV a,reg1
@avanti:	@avanti:

Listato 2.3: L'unità di controllo può solo decidere una volta, per tutti, se eseguire il salto oppure no, ma la condizione dipende dai dati su ciascuna Processing Unit.

Per risolvere questo problema, e consentire l'uso di istruzioni di controllo, sono state previste alcune soluzioni. La più antica, implementata anche nell'assembly ARB, prevede l'uso di singole istruzioni condizionate.

Sono presenti singole istruzione macchina (che non alterano il flusso di controllo) che possono essere condizionate al risultato dell'istruzione precedente. Questa semplice operazione può essere eseguita anche senza intervento dell'unità di controllo, perchè si limita ad avere un effetto diverso a seconda di quel risultato. Il tempo di esecuzione e la sequenza di istruzioni successive non è alterata. Ad esempio, può essere presente una istruzione macchina che sposta il contenuto di un registro in un altro solo se l'uguaglianza valutata in precedenza è risultata vera (MOVE: MOVE if Equal). Se più operazioni devono essere eseguite sotto condizione, questa soluzione impone che siano eseguite comunque su variabili temporanee, poi sia valutata la condizione, e infine i risultati individuati siano salvati all'interno delle variabili di destinazione solo se la condizione finale è risultata vera.

Con questo metodo le istruzioni devono essere riordinate, ma sono le stesse per entrambe le processing unit e non ci sono problemi (listato 2.4).

Una seconda soluzione prevede che le Shader Processing Unit per cui la condizione è risultata falsa ignorano le istruzioni di controllo (sono mascherate). Al termine del blocco condizionato, l'unità di controllo invia un comando particolare (nell'esempio 2.5, UNMASK) e anche le Processing Unit in attesa riprendono a se-

Processing Unit 1 (d1 == d2)	Processing Unit 2 (d1 != d2)
TEMPF a	TEMPF a
MOV a,1.0	MOV a,1.0
TEMPF temp1	TEMPF temp1
CALL f	CALL f
MOV temp1,reg1	MOV temp1,reg1
CMP d1,d2	CMP d1,d2
MOVE a,temp1 ; questa esegue, sono uguali	MOVE a,temp1 ; questa non esegue, sono diversi

Listato 2.4: Con MOVE e istruzioni riordinate.

guire le istruzioni. Questa soluzione è più complessa da realizzare dal punto di vista dell'architettura della GPU, perché necessita la presenza di un sistema di mascheratura delle Shader Processing Unit. Però i consumi risultano ridotti, in quanto le unità in attesa consumano meno.

Processing Unit 1 (d1 == d2)	Processing Unit 2 (d1 != d2)
TEMPF a	TEMPF a
MOV a,1.0	MOV a,1.0
CMP d1,d2	CMP d1,d2
MASKNE ; MASK if Not Equal	MASKNE ; solo l'unita' 2 e' mascherata, qui
CALL f	<mascherata, ignora le istruzioni>
MOV a,reg1	<mascherata, ignora le istruzioni>
UNMASK ; gia' non mascherata	UNMASK ; mascheratura tolta, continua l'esecuzione

Listato 2.5: Soluzione con mascheratura.

Di solito, questo metodo introduce alcuni limiti per i condizionamenti nidificati (if dentro un altro if). Nell'esempio precedente, ad esempio, non esiste nessun modo per togliere la mascheratura solo a una parte delle unità mascherate e si dovrà ricorrere al riordinamento del codice o a qualche altro metodo se l'if interno non termina insieme all'if esterno.

Come si può osservare, le istruzioni di controllo non provocano nessun vantaggio in termini di velocità di esecuzione dello shader. A meno della possibilità piuttosto remota che tutte le (centinaia di) processing unit seguano lo stesso percorso nel codice condizionato, infatti, tutte le Processing Unit eseguono per il tempo massi-

mo possibile, cioè quello che servirebbe nel caso in cui tutto il codice condizionato debba essere eseguito.

```
float a;  
if (d1 == d2)  
    a = f();  
else  
    a = g();
```

Listato 2.6: *f e g sono funzioni eseguite in alternativa.*

Per esempio, dato il listato 2.6, in una CPU è eseguita o f o g , perciò il tempo medio di esecuzione di quel codice è circa la media (pesata) di quello di f e di g , ammesso che il tempo per valutare la condizione sia trascurabile. In una GPU, invece, il tempo di esecuzione è quasi sempre la somma dei tempi di esecuzione di f e g , indipendentemente da quella delle due che è stata selezionata.

In conclusione, è bene evitare il più possibile l'uso delle istruzioni di controllo di flusso all'interno degli shader, specialmente quelle nidificate.

Capitolo 3

Implementazione in OpenGL

3.1 Versioni ed estensioni

OpenGL non è una libreria grafica. Esso è uno standard, un'API libera che può (deve, allo stato attuale) essere implementata dai produttori di schede video per essere compatibili con le applicazioni che usano l'accelerazione 3D.

Però, è risultato presto evidente che la grande varietà di schede video in commercio implementa funzioni specifiche che sono molto diverse tra loro. Perciò, a fianco dell'API di base di OpenGL, che implementa le funzioni grafiche più comuni, sono state create le estensioni, che il produttore può decidere se implementare oppure no.

Un'estensione è identificata da una stringa senza spazi. Ad esempio, `GL_EXT_geometry_shader4` o `GL_ARB_geometry_shader4` è l'estensione che permette l'uso dei geometry shader. Quando un software ha bisogno di un'estensione che non è parte dello standard OpenGL, deve sempre verificare che sia implementata nella scheda video e nel suo driver.

Periodicamente, OpenGL seleziona alcune tra le estensioni più utili, le riorganizza e le implementa in una nuova versione, che diventa il nuovo standard obbligatorio per tutte le schede video. La retrocompatibilità non è sempre richiesta, perciò una scheda video che supporta una versione di OpenGL potrebbe non supportare parte delle versioni precedenti. Ciò significa che anche il supporto per una specifica versione di OpenGL deve essere sempre verificato dal software prima di essere

usato.

Normalmente, per OpenGL, la versione è indicata da due numeri, di cui il primo è il più significativo, separati da un punto: 1.5, 2.0, 3.2, eccetera.

La versione di GLSL è la versione del linguaggio di programmazione che il driver della scheda video è capace di compilare e di cui la scheda video supporta tutti i requisiti e le funzioni. Questa versione è indicata da tre numeri separati da due punti: 1.50.11, 3.30.6, e così via. Di tutte le versioni conosciute, il secondo numero è sempre divisibile per 10, e il terzo numero non sembra avere significato, perché esiste un'unico valore ammissibile data la combinazione dei due precedenti.

In origine, la numerazione di GLSL era indipendente da quella di OpenGL. La prima implementazione di GLSL in OpenGL 2.0 aveva infatti versione 1.10.59. Tuttavia, siccome gli shader ormai influiscono in pressochè tutte le parti della grafica, una versione di OpenGL corrisponde sempre a una di GLSL.

Il legame tra versione GLSL e OpenGL è diventato sempre più evidente. A partire dalla versione di OpenGL 3.3 (GLSL 3.30.6), il primo numero corrisponde, e il secondo numero differisce solo perché è moltiplicato per 10 nella versione di GLSL.

GLSL	OpenGL	Data rilascio	Novità importanti
1.10.59	2.0	7 settembre 2004	Vertex e Fragment Shader
1.20.8	2.1	2 luglio 2006	Geometry Shader come estensione
1.30.10	3.0	11 luglio 2008	
1.40.08	3.1	28 maggio 2009	
1.50.11	3.2	3 agosto 2009	Geometry Shader
3.30.6	3.3	11 marzo 2010	
4.00.9	4.0	11 marzo 2010	Tessellation Shaders
4.10.6	4.1	26 luglio 2010	
4.20.6	4.2	22 agosto 2010	(Versione corrente)

Tabella 3.1: *Corrispondenza tra la versione GLSL e quella OpenGL.*

Degli esempi riportati in questa tesi, gran parte sono stati scritti per OpenGL 2.1 o per OpenGL 4.0. La versione 2.1 è stata scelta perché può accedere ai primi tre tipi di shader (Vertex, Fragment e Geometry) in tutte le schede video moderne e retrocompatibili, e contemporaneamente lo standard non ha ancora subito quelle ra-

dicali modifiche a OpenGL che hanno reso l'API più difficile da capire. Si discuterà di queste ultime nella sezione 3.6: "Il nuovo standard OpenGL".

Le sezioni che seguono sono un'introduzione agli shader in OpenGL 2.1 e non sono intese come una guida alla sintassi GLSL, benché la conoscenza di quella sintassi sia richiesta per comprendere i numerosi frammenti di codice presentati.

3.2 Il concetto di program

Gli shader non possono essere caricati sul processore grafico direttamente, ma devono essere prima inseriti all'interno di strutture dette *program*. Un *program* è una raccolta di shader programmabili che hanno alcune caratteristiche che li rendono compatibili tra loro.

È possibile definire un elevato numero di *program*, a cui associare diversi shader. Ciascuno shader può essere associato a più *program*, ad alto livello, ma in realtà a basso livello di quello shader vengono eseguite più copie. Perciò risulta impossibile comunicare tra un programma e l'altro: tutte le variabili sono locali al *program* corrente.

Shader scritti in versioni di GLSL diverse possono essere collegati nello stesso *program*, se le versioni non sono troppo distanti tra loro. Ciascuna versione di GLSL specifica con quali altre versioni è compatibile.

In ogni istante può essere in uso della GPU un solo *program* e tutti gli shader che contiene sono attivi contemporaneamente. Mentre un *program* è attivo, tutte i vertici (primitive) inviati alla pipeline vengono processati da quel particolare *program*, a partire dall'istante in cui quel *program* è attivato. È possibile attivare un *program*, cambiare il *program* in uso o ripristinare la Fixed Function Pipeline in qualunque momento, eccetto durante la definizione della stessa primitiva (in OpenGL, tra i corrispondenti `glBegin` e `glEnd`).

Se all'interno di un *program* non è definito nessuno shader di un certo tipo, quella parte è sostituita con lo shader di default della Fixed Function Pipeline. Questo permette di non implementare le parti della pipeline che non si vogliono modificare. Inoltre, viene garantita anche una certa retrocompatibilità: *program* creati in epoche in cui una certa funzione della pipeline non poteva essere sostituito con uno shader

programmato continuano a funzionare esattamente allo stesso modo sulle schede video più recenti.

Tuttavia, quando uno shader di un certo tipo è definito, tutte le funzionalità che la Fixed Function Pipeline eseguiva a quello stadio sono disabilitate. Se servono ancora, sarà necessario reimplementarle manualmente all'interno dello shader.

Più shader dello stesso tipo possono essere assegnati allo stesso *program*. In questo caso, gli shader sono uniti (fase di linking) in modo simile ai file oggetto generati dai compilatori nei programmi per la CPU.

Questo significa che deve essere sempre definito uno shader principale, che viene invocato per primo. È questo shader che può chiamare le funzioni che sono definite negli altri shader, che sono trattati come semplici librerie. In GLSL, lo shader principale è sempre quello che definisce la funzione `void main()`.

3.3 Vertex shader

Un vertex shader è il primo shader all'inizio della pipeline. Esso viene istanziato una volta per ogni singolo vertice in ingresso e non ha alcuna informazione di connessione tra i vertici adiacenti. I vertex shader sono supportati fin da OpenGL 2.0, con l'introduzione del primo linguaggio ad alto livello, GLSL 1.10.

Un vertex shader sostituisce le seguenti funzioni della Fixed Function Pipeline di OpenGL:

- Trasformazioni di modello, di vista e di proiezione dei vertici;
- Illuminazione e applicazione del colore ai vertici;
- Trasformazione delle normali;
- Generazione delle coordinate delle texture.

Queste funzioni sono disabilitate da OpenGL appena si attiva un *program* che contiene almeno un vertex shader. Se sono ancora necessarie, devono essere reimplementate nel codice dello shader.

```
vec4 gl_Vertex;           // posizione del vertice (xyzw) da glVertex
vec3 gl_Normal;          // normale del vertice (xyz) da glNormal
vec4 gl_Color;           // colore del vertice (rgba) da glColor
vec4 gl_MultiTexCoord0; // coordinate delle texture (xyzw)
```

Listato 3.1: *Attributi del Vertex Shader accessibili in lettura*

```
vec4 gl_Position;        // nuova posizione del vertice (xyzw)
vec4 gl_FrontColor;      // colore del vertice per il
                        // davanti della primitiva (rgba)
vec4 gl_BackColor;       // colore del vertice per il
                        // retro della primitiva (rgba)
vec4 gl_TexCoord[];      // coordinate delle texture (xyzw)
```

Listato 3.2: *Attributi del Vertex Shader accessibili in scrittura*

In GLSL 1.20, un vertex shader ha accesso in sola lettura ad alcuni attributi dei vertici (listato 3.1) e accesso in scrittura ad altri attributi (listato 3.2). Quando il vertex shader termina, queste ultime diventano le nuove proprietà del vertice.

Inoltre, come tutti gli shader, un vertex shader ha accesso a molte delle variabili di stato di OpenGL, tra cui le matrici (listato 3.3).

```
mat4 gl_ProjectionMatrix; // matrice di proiezione
mat4 gl_ModelViewMatrix;  // matrice di modelview
mat4 gl_ModelViewProjectionMatrix; // prodotto già precalcolato di
                        // gl_ProjectionMatrix * gl_ModelViewMatrix
mat3 gl_NormalMatrix;     // matrice di trasformazione per le normali,
                        // pari alla parte in alto a sinistra dell'
                        // inversa della trasposta di
                        // gl_ModelViewMatrix
```

Listato 3.3: *Variabili di stato.*

Il listato 3.4 riporta un semplice esempio di Vertex Shader.

```
#version 120

void main()
{
    gl_FrontColor = gl_Color;
    gl_BackColor = gl_Color;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Listato 3.4: *Un semplice vertex shader, privo di illuminazione.*



Figura 3.1: *Immagine di un icosaedro ottenuta con solo il vertex shader di cui nel listato 3.4. Si nota che l'illuminazione non è presente, perché non è stata implementata nello shader.*

3.4 Fragment shader

Il fragment shader esegue le ultime operazioni sui frammenti prima che essi subiscano il blending e diventino pixel. Ciascuna istanza del fragment shader ha il compito di calcolare il colore di un singolo frammento ed eventualmente modificare la sua profondità.

Un fragment shader sostituisce le funzioni della Fixed Function Pipeline che agiscono sulla colorazione per singolo pixel, in particolare l'effetto nebbia e l'applicazione delle texture. Se queste sono ancora necessarie, dovranno essere implementate a mano.

I fragment shader sono stati introdotti, contemporaneamente ai vertex shader, a partire da OpenGL 2.0 (GLSL 1.10).

Il fragment shader ha accesso alle variabili globali indicate nel listato 3.5.

Di queste, le più importanti sono `gl_Color` e `gl_FragColor`. La prima contiene il colore originale del frammento calcolato per interpolazione del colore prodotto dal vertex shader in `gl_FrontColor` o `gl_BackColor` (o dalla Fixed Function Pipeline se non c'è vertex shader). Nella seconda deve essere scritto il colore finale del frammento, cioè quello che verrà visualizzato sullo schermo se il

```
vec4 gl_Color;  
vec4 gl_FragColor;  
float gl_FragDepth;  
vec4 gl_FragCoord;
```

Listato 3.5: *Input del fragment shader.*

depth test ha successo.

`gl_FragCoord` contiene le coordinate del pixel all'interno della viewport, nel formato `x,y,z,w`. Normalmente, l'origine è nell'angolo in basso a sinistra dello schermo. Le coordinate `x` e `y` variano da 0.0 al numero di pixel di larghezza e altezza della finestra. I pixel sono centrati a metà dell'unità: 0.5, 1.5, 2.5 eccetera. La coordinata `z` in questo caso indica la profondità del frammento.

Usata raramente, `gl_FragDepth` può essere scritta per impostare la profondità del frammento. Il valore iniziale di questa variabile è indefinito, non contiene il valore originale della profondità. Se `gl_FragDepth` non viene mai citato nel codice dello shader, la funzione della Fixed function Pipeline che calcola la profondità viene ripristinata ed è posto uguale a `gl_FragCoord.z / gl_FragCoord.w`.

Il fragment shader ha alcune limitazioni:

- Non sostituisce le funzioni di blending e non può accedere al frame buffer. Perciò, non ha nessuna possibilità di conoscere il colore attuale dei pixel sullo schermo o se il proprio frammento sarà veramente visualizzato oppure no.
- La posizione del frammento è già decisa sulla viewport e non può essere cambiata.

Il fragment shader è l'unico shader che può usare la parola chiave `discard`. Quando questa istruzione è eseguita, l'esecuzione dell'istanza dello shader si interrompe e il frammento non è mai visualizzato sullo schermo.

Il listato 3.6 riporta un semplice esempio di Fragment Shader.

```
#version 120
void main()
{
    gl_FragColor = gl_Color;
}
```

Listato 3.6: *Un semplice fragment shader, che reimplementa tutte le funzioni attive di default nella Fixed Function Pipeline a questo stadio.*

3.5 Geometry shader

Un'istanza di un geometry shader riceve in ingresso una primitiva e produce in uscita un qualunque numero di primitive, che possono essere di tipo diverso di quelle ricevute in ingresso. Tuttavia, un *program* che contiene geometry shader deve essere specializzato per ricevere in ingresso un unico tipo di primitiva e tutte le primitive prodotte devono essere dello stesso tipo. Inoltre, più istanze non possono mai collaborare a produrre la stessa primitiva.

L'installazione di un geometry shader non sostituisce nessuna parte della Fixed Function Pipeline.

Un geometry shader può svolgere tutte le funzioni di un vertex shader, più molte altre. Ciononostante, in genere il geometry shader è meno efficiente per quelle funzioni, e comunque è obbligatorio inserire almeno un vertex shader in ogni *program* che contiene un geometry shader.

È necessario specificare il tipo di primitive di input e di output di un geometry shader, quando esso viene creato. Nella versione 1.20 il tipo deve essere impostato appena prima del linking del *program* che contiene il geometry shader, tramite alcune chiamate a funzioni OpenGL.

Le primitive di output di un geometry shader possono essere soltanto di tre tipi:

- `GL_POINTS`: singolo punto,
- `GL_LINE_STRIP`: linea spezzata,
- `GL_TRIANGLE_STRIP`: triangoli ciascuno con un lato in comune al precedente.

Le possibili primitive di input sono le seguenti, invece:

- `GL_POINTS`: punto,
- `GL_LINES`: segmento,
- `GL_TRIANGLES`: triangolo,
- `GL_LINES_ADJACENCY`: segmento con informazioni di adiacenza,
- `GL_TRIANGLES_ADJACENCY`: triangolo con informazioni di adiacenza.

Per prima cosa, si nota che primitive diverse da punti, linee e triangoli (come quadrilateri e poligoni) non sono supportate dai geometry shader, e devono essere scomposte prima di poter essere inviate alla pipeline che usa geometry shader.

Sono comparsi invece altri due tipi di primitiva, `GL_LINES_ADJACENCY` e `GL_TRIANGLES_ADJACENCY`. Questi tipi di primitiva possono essere usati per definire primitive come quelli tradizionali (ad esempio possono diventare il parametro di `glBegin`), ma hanno significato solo se è in uso un geometry shader.

Questi tipi servono per estendere `GL_LINES` e `GL_TRIANGLES` ed includere informazioni sulle primitive adiacenti a quella corrente.

Con `GL_LINES_ADJACENCY`, un segmento viene definito ogni 4 vertici. Due vertici servono per il segmento e gli altri due per ciascuno dei due segmenti che hanno un vertice in comune con esso. L'ordine di inserimento dei vertici può essere quello in figura 3.2, anche se si vedrà poi che non è particolarmente rilevante.



Figura 3.2: Esempio di ordine di inserimento dei vertici di un segmento con informazioni di adiacenza. In rosso è segnato il segmento, mentre in verde i segmenti adiacenti.

Si nota che questo definisce un solo segmento, quello dal punto 2 al punto 3, e OpenGL non assume o verifica l'effettiva esistenza di segmenti adiacenti, né associa questo segmento agli altri. Il segmento successivo è trattato come una primitiva completamente separata da questa. Perciò, allo stesso modo, anche il segmento

adiacente deve essere definito con 4 vertici, di cui 3 sono in comune con questo (ad esempio, il 2, il 3, il 4 e il 5).

Con `GL_TRIANGLES_ADJACENCY`, ciascun triangolo è definito con 6 vertici. Tre sono i vertici del triangolo e tre sono i vertici che i triangoli adiacenti non hanno in comune con il triangolo corrente. Di nuovo, quello in figura 3.3 può essere l'ordine di inserimento dei vertici. Valgono considerazioni analoghe alle precedenti.

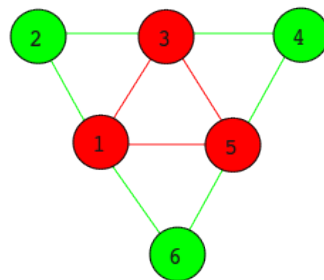


Figura 3.3: Esempio di ordine di inserimento dei vertici di un triangolo con informazioni di adiacenza. I vertici 1, 3 e 5 definiscono il triangolo, mentre i vertici 2, 4 e 6 i triangoli adiacenti.

Un geometry shader riceve in ingresso una primitiva i cui vertici sono già stati elaborati dal vertex shader. Esso riceve in ingresso un array di vertici, ciascuno con le proprietà esattamente come sono state prodotte dal vertex shader. Questo include tutte le proprietà predefinite (listato 3.7).

```
vec4 gl_FrontColorIn []; // da gl_FrontColor
vec4 gl_BackColorIn []; // da gl_BackColor
vec4 gl_PositionIn []; // da gl_Position
vec4 gl_TexCoordIn [][]; // da gl_TexCoord[]
```

Listato 3.7: Variabili di ingresso predefinite del Geometry Shader.

Il numero di elementi di questi array è il numero di vertici della primitiva (3, per un triangolo). Essi sono ordinati secondo la definizione originale dell'applicazione OpenGL. Si nota che la variabile `gl_TexCoord`, che era già un array, è diventata un array di array.

Inoltre, il geometry shader riceve tutte le variabili *varying* provenienti dal vertex shader. Queste variabili non possono più scavalcare il geometry shader per raggiungere il fragment shader.

Il geometry shader deve dichiarare le variabili *varying* in ingresso come variabili globali, aggiungendo la parola chiave `in` (listato 3.8).

```
varying in vec3 varyvar[3];
```

Listato 3.8: Esempio di dichiarazione di variabile *varying* in input a un Geometry Shader.

Tutte le variabili di ingresso sono considerate costanti all'interno del geometry shader.

Anche se ciascun vertex shader ha prodotto un solo valore per questa variabile, il geometry shader riceve un vettore di lunghezza pari al numero di vertici della primitiva di input. Il vettore è inizializzato, in ordine, con ciascuno dei valori della variabile prodotti dalle varie istanze dei vertex shader.

Risulta a questo punto evidente perché la sequenza con cui i vertici sono inviati alla pipeline, all'interno della stessa primitiva, non può essere definita a priori. Il geometry shader deve produrre i vertici nell'ordine corretto, ma può interpretare l'input in qualunque ordine, basta che ci sia la stessa convenzione di ordinamento tra programma OpenGL e geometry shader.

In ultima analisi, il tipo di primitiva di input definisce solo il numero di vertici che sono raggruppati insieme per essere elaborati dallo stesso geometry shader:

- `GL_POINTS`: un vertice,
- `GL_LINES`: due vertici,
- `GL_TRIANGLES`: tre vertici,
- `GL_LINES_ADJACENCY`: quattro vertici,
- `GL_TRIANGLES_ADJACENCY`: sei vertici.

Anche se il geometry shader dovrebbe produrre una primitiva, tale primitiva viene costruita un vertice alla volta. Sono predefinite le solite variabili globali per un singolo vertice (listato 3.9).

```
vec4 gl_FrontColor;  
vec4 gl_BackColor;  
vec4 gl_Position;  
vec4 gl_TexCoord[];
```

Listato 3.9: Variabili di uscita predefinite del Geometry Shader.

Il geometry shader può definire anche variabili *varying* personalizzate, ad uso del successivo fragment shader. Per distinguerle da quelle provenienti dal vertex shader, si aggiunge la parola chiave `out` (listato 3.10).

```
varying out vec3 varyvar[3];
```

Listato 3.10: Esempio di dichiarazione di variabile *varying* in output da un Geometry Shader.

Tutte le variabili di output possono essere lette e scritte ovunque nel codice del geometry shader. In qualunque momento può essere chiamata la funzione predefinita `EmitVertex`, che legge lo stato attuale di tutte le variabili di uscita e produce un vertice con quei valori. Dopodiché, le variabili possono subito essere modificate per l'emissione del vertice successivo.

Un'altra funzione, `EndPrimitive` conclude la primitiva corrente. Se lo shader sta producendo punti, questo non ha nessun effetto. Se sta producendo una linea spezzata, termina la linea corrente e ne comincia una nuova a partire dal prossimo vertice emesso. Se sta producendo una triangle strip, ne inizia un'altra.

Se `EndPrimitive` non viene chiamata al termine dello shader, la primitiva corrente viene conclusa automaticamente, se sono stati emessi abbastanza vertici, o scartata se non sono sufficienti.

Per fare in modo che certo valore scavalchi il geometry shader, si definisce una variabile di input dal vertex shader e poi si copia sempre il valore in una variabile

di output verso il fragment shader. Ovviamente, la variabile di uscita deve avere un nome diverso di quella di ingresso, perciò è necessaria una piccola modifica al vertex shader o al fragment shader.

```
#version 120
#extension GL_EXT_geometry_shader4: enable

void main()
{
    for (int i = 0; i < 3; i++)
    {
        gl_FrontColor = gl_FrontColorIn[i];
        gl_BackColor = gl_BackColorIn[i];
        gl_Position = gl_PositionIn[i];
        EmitVertex();
    }

    EndPrimitive();
}
```

Listato 3.11: *Il più semplice Geometry Shader, che si limita a copiare tutti i valori in ingresso in valori di uscita.*

3.6 Il nuovo standard OpenGL

A partire da OpenGL 3.0, la maggior parte delle funzionalità che per un qualche motivo sono state giudicate incompatibili o limitative per gli shader sono state gradualmente dichiarate deprecate o eliminate del tutto. Rimangono disponibili per quelle schede video che hanno deciso di mantenere la retrocompatibilità, che fortunatamente sono, per ora, la maggior parte.

La giustificazione di ciò riguarda la complessità crescente di dover mantenere sia gli shader che la vecchia Fixed Function Pipeline e le innumerevoli funzioni (più di 500) per modificare i suoi parametri. Questa complessità ricade quasi completamente sulle spalle dei produttori delle schede video e dei loro driver, che devono implementare quelle funzioni.

In particolare, l'architettura degli shader impone che il compilatore sia implementato all'interno del driver stesso, siccome ciascuna scheda video usa un diverso

linguaggio macchina proprietario e deve poter compilare e ottimizzare il codice GLSL in quel linguaggio.

Per evitare di appesantire l'API, e siccome attraverso gli shader tutte quelle funzionalità possono essere reimplementate comunque in modo molto più preciso ed adeguato alle esigenze, OpenGL ha scelto di eliminarne gran parte. In futuro, si ipotizza che la stessa Fixed Function Pipeline possa essere abolita e che l'uso degli shader divenga obbligatorio.

Purtroppo, ciò ha fatto sì che fosse molto più lungo e complicato partire da zero e avere un programma OpenGL funzionante, a causa del gran numero di funzionalità da reimplementare. Così sono nate un gran numero di librerie di alto livello, al di sopra di OpenGL, che tentano di proporre al programmatore un'API più amichevole.

Alla versione 4.0, un gran numero di funzionalità di OpenGL sono state abolite.

- L'illuminazione, che deve essere completamente reimplementata attraverso gli shader. Variabili come `gl_LightSource`, `gl_FrontMaterial`, `gl_BackMaterial` non sono più accessibili dall'interno degli shader.
- Le matrici di modelview (`gl_ModelViewMatrix`) e proiezione (`gl_Projection`), le altre matrici che sono calcolate da queste e, dal lato OpenGL, gli stack delle matrici (`glLoadMatrix`, `glMatrixMode`, `glPushMatrix` e `glPopMatrix`), nonché tutte le trasformazioni (`glTranslate`, `glRotate`, ecc). Se queste matrici sono ancora necessarie, devono essere reimplementate con opportune variabili uniform di tipo `mat4`. La reimplementazione della struttura con cui accedervi e la scelta della libreria matematica con cui calcolarle sono lasciate al programmatore, che in questo modo può scegliere quelle più utili alle sue esigenze. Un piccolo vantaggio di questo cambiamento è che non si è più legati alla struttura a 2 sole matrici (Projection e Modelview) ma se ne possono usare anche tre (Projection, View, Model) o qualsiasi numero, a seconda dell'algoritmo implementato negli shader.
- Gli attributi predefiniti dei vertici: `gl_Normal`, `gl_Vertex`, `gl_MultiTexCoord`, `gl_Color`. Questo porta all'abolizione delle funzioni OpenGL per impostarli: `glNormal`, `glVertex`, `glColor`,

`glTexCoord`. Tutti gli attributi, inclusa la posizione del vertice, devono essere forniti allo shader attraverso variabili attribute personalizzate.

Con l'abolizione di `glVertex`, diventa impossibile definire vertici con il metodo tradizionale. Perciò sono aboliti anche `glBegin` e `glEnd`.

Se non è più possibile definire singoli vertici, diventa obbligatorio usare le funzioni che si basano su array di vertici. Prima si deve allocare un gran numero di vertici in memoria e poi essi sono inviati alla scheda video tutti in blocco con la funzione `glDrawElements` o simili. Questo metodo è più efficiente per sfruttare il parallelismo degli shader e per visualizzare in un colpo solo una grande quantità di vertici, ma è certamente più lungo e complicato da implementare a mano.

Le display list sono state abolite. Per ottenere lo stesso aumento di efficienza, è possibile creare degli oggetti chiamati VBO (Vertex Buffer Object). Essi sono spazi di memoria che possono essere allocati direttamente sulla memoria video, in modo che non debbano essere trasferiti su di essa a ogni ciclo di rendering. Dopo averli configurati, il loro uso diventa equivalente a array di attributi allocati in RAM, eccetto che l'accesso da parte della CPU è più lento e deve passare attraverso OpenGL e il driver della scheda video.

Il nuovo standard ha portato anche numerose modifiche al linguaggio GLSL. La più notevole è l'introduzione delle direttive di tipo `layout`, che possono essere usate per configurare molte delle funzionalità di OpenGL legate agli shader dall'interno degli shader stessi, senza dover chiamare le funzioni C dell'API.

3.7 Tessellation control shader

Il tessellation control shader viene eseguito una volta per ogni vertice della patch in input, ma al contrario del vertex shader può accedere a tutte le informazioni della patch corrente. Ha lo scopo di istruire il Tessellator per aumentare il numero di vertici e di primitive della patch e dunque renderla più precisa.

Una patch è una particolare primitiva che viene definita tramite la costante `GL_PATCHES`, che è un tipo di primitiva utilizzabile analogamente a `GL_LINES` e `GL_TRIANGLES`. Le patch possono essere composte da un qualunque numero di vertici, definito dal programmatore.

È sempre necessario avere un Tessellation Control Shader attivo per poter usare il tipo di primitiva `GL_PATCHES`, e viceversa è obbligatorio usare `GL_PATCHES` se c'è un Tessellation Shader attivo. È inoltre obbligatoria la presenza di un Vertex Shader e di un Tessellation Evaluation Shader, ma non di un Geometry Shader.

La funzione principale di un Tessellation Control Shader è quella di impostare due variabili (listato 3.12). Esse devono essere impostate una volta da almeno una delle varie istanze, ma non è necessario che siano impostate da tutte, perché sono condivise da tutte le istanze dello shader, per ogni *patch*.

```
int gl_TessLevelInner[2];
int gl_TessLevelOuter[4];
```

Listato 3.12: *Le variabili che controllano il grado di tessellazione della patch.*

Queste due variabili hanno la funzione di configurare il Tessellator e hanno valore minimo 1 (figura 3.4). All'aumentare di `gl_TessLevelInner`, nuove primitive sono create all'interno della patch e connesse ai suoi vertici. Tuttavia, i lati della patch originali non vengono toccati.

All'aumentare di `gl_TessLevelOuter` i lati della patch originale sono suddivisi in segmenti e la primitiva originale è suddivisa a raggiera, con nuovi lati che vanno dal punto centrale ai nuovi vertici. `gl_TessLevelOuter` ha un numero di componenti pari al numero di lati (3 per i triangoli e 4 per i quadrati) del tipo di patch in ingresso. Se si assegnano valori diversi alle componenti del vettore, lati diversi saranno scomposti in un numero di segmenti diverso.

L'effetto migliore si ottiene per valori simili delle le due variabili.

Come nel caso del geometry shader, tutti gli attributi in uscita dal vertex shader sono raggruppati in vettori in input al Tessellation Control Shader. La parola chiave `varying` è stata abolita (listato 3.13).

Una variabile predefinita intera, `gl_InvocationID`, contiene l'id del vertice della patch per cui il tessellation control shader è stato allocato.

Il tessellation control shader richiede che, all'inizio, sia specificata una direttiva di tipo `layout`. La sintassi è quella mostrata nel listato 3.14.

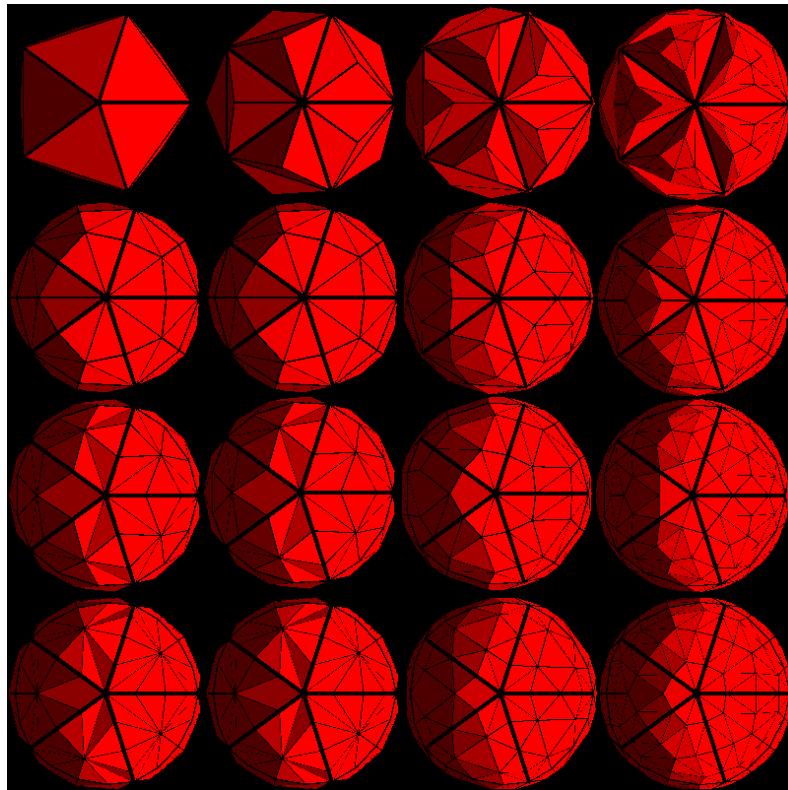


Figura 3.4: Tessellatura di un icosaedro. *TessLevelOuter* aumenta di 1 ad ogni riga. *TessLevelInner* aumenta di 1 a ogni colonna. Il tessellation evaluation shader normalizza la distanza dal centro, per questo la forma tende a una sfera. (shader originale proveniente da *The Little Grasshopper* [7])

La costante intera (in questo caso 3) visualizzata nel listato indica il numero di vertici che vengono prodotti dal tessellation control shader e inviati al Tessellator. Questo numero di solito è uguale al numero di vertici della patch in ingresso, ma se il tessellation control shader esegue già qualche interpolazione potrebbe non essere così.

Per ciascuna *patch*, il tessellation control shader è invocato quel numero di volte. Inoltre, tutte le variabili in uscita verso il tessellation evaluation shader devono essere array di quella lunghezza. Questo spiega perché nel listato 3.13 si sia lasciato vuoto lo spazio tra le parentesi quadre: la loro lunghezza è ottenuta automaticamente.

Infine si segnala la funzione predefinita `barrier`, mostrata nel listato 3.15, che permette di ottenere una rudimentale sincronizzazione tra le varie istanze dei

```
in vec3 invaryvar[];  
out vec3 outvaryvar[];
```

Listato 3.13: Esempio di variabili in ingresso e in uscita del tessellation control shader. La dimensione degli array è ricavata automaticamente dal compilatore e può essere omessa.

```
layout(vertices = 3) out;
```

Listato 3.14: Esempio di direttiva layout del tessellation control shader.

tessellation control shader di una stessa patch. Ciò è necessario perché esse possono scrivere tutte sugli stessi dati, cosa che può causare conflitti e situazioni di corsa.

```
void barrier();
```

Listato 3.15: Prototipo della funzione barrier.

La funzione `barrier` garantisce di essere eseguita contemporaneamente in tutte le istanze. Ciò significa che se per qualche motivo una istanza arriva ad eseguire la funzione prima delle altre, essa si deve fermare e attendere che tutte siano pronte. Se compare più volte nel codice, il suo effetto è ripetuto: prima sono sincronizzate le prime esecuzioni, poi le seconde, eccetera. Questa funzione può essere chiamata solo all'interno della funzione `main` e al di fuori di qualsiasi blocco di codice condizionato.

Nel listato 3.16 è riportato un esempio di Tessellation Control Shader. L'esempio è molto semplice, ma è stato necessario reimplementare un gran numero di variabili che, in versioni di OpenGL precedenti, erano predefinite.

```

#version 400

layout(vertices = 3) out;

in vec3 vPosition[];
in vec3 vNormal[];
in vec4 vColor[];
in float vmatshininess[];
in vec3 vmatspecular[];
out vec3 tcPosition[];
out vec3 tcNormal[];
out vec4 tcColor[];
out float tcmatshininess[];
out vec3 tcmatspecular[];
uniform float TessLevelInner;
uniform float TessLevelOuter;

#define ID gl_InvocationID

void main()
{
    tcPosition[ID] = vPosition[ID];
    tcNormal[ID] = vNormal[ID];
    tcColor[ID] = vColor[ID];
    tcmatshininess[ID] = vmatshininess[ID];
    tcmatspecular[ID] = vmatspecular[ID];
    if (ID == 0)
    {
        gl_TessLevelInner[0] = TessLevelInner;
        gl_TessLevelOuter[0] = TessLevelOuter;
        gl_TessLevelOuter[1] = TessLevelOuter;
        gl_TessLevelOuter[2] = TessLevelOuter;
    }
}

```

Listato 3.16: Esempio di tessellation control shader.

3.8 Tessellation evaluation shader

Il tessellation evaluation shader è eseguito una volta per ogni vertice prodotto dal Tessellator, nonché per i vertici che in origine facevano parte della patch. La presenza di un Tessellation Evaluation Shader rende obbligatorio l'uso del tipo di primitiva patch e la presenza di un tessellation control shader.

Ciascuna istanza può leggere tutti gli array che sono stati inviati dal tessellation control shader, esattamente come esso li ha prodotti, ma può produrre dati per un solo vertice.

L'utilità principale di questo shader è il calcolo degli attributi dei nuovi vertici creati dal Tessellator, in modo che possano proseguire la pipeline come quelli della

patch originale.

Anche il tessellation evaluation shader prevede una direttiva di tipo layout, del tipo indicato nel listato 3.17.

```
layout(triangles, equal_spacing, ccw) in;
```

Listato 3.17: Esempio di direttiva layout.

Questa direttiva indica come i gruppi di vertici emessi dal tessellation control shader debbano essere interpretati e quale tipo di primitive debba essere prodotto dal tessellation evaluation. Il primo parametro può essere:

- `triangles`: i vertici rappresentano (insiemi di) triangoli e devono rimanere triangoli,
- `quads`: i vertici rappresentano (insiemi di) quadrilateri e devono essere convertiti in triangoli,
- `isolines`: i vertici rappresentano (insiemi di) quadrilateri e devono essere convertiti in segmenti.

Il secondo parametro può essere:

- `equal_spacing`: i lati della patch originale sono suddivisi in segmenti di uguale lunghezza,
- `fractional_even_spacing`: i lati della patch originale sono suddivisi in un numero pari di segmenti di uguale lunghezza, più due più corti,
- `fractional_odd_spacing`: i lati della patch originale sono suddivisi in un numero dispari di segmenti di uguale lunghezza, più due più corti.

Il terzo parametro può essere:

- `cw`: i vertici delle primitive sono prodotti in senso orario,
- `ccw`: i vertici delle primitive sono prodotti in senso antiorario.

Oltre alle variabili personalizzate che provengono dal tessellation control shader, il tessellation evaluation shader non riceve né dati sulla patch né la nuova posizione dei vertici interpolati. Invece, ha accesso in lettura alla variabile `vec3 gl_TessCoord`. Questa variabile contiene i pesi per calcolare la posizione del vertice assegnato a quel tessellation evaluation shader come media delle posizioni degli altri vertici.

Nell'ipotesi in cui le primitive di input siano triangoli e che il tessellation control shader abbia passato avanti un array che contiene le posizioni dei vertici della patch (listato 3.18), la posizione può essere calcolata con il codice presentato nel listato 3.19.

```
out vec3 tcPosition[3];
```

Listato 3.18: Esempio di variabile personalizzata che il tessellation control shader può usare per passare avanti la posizione dei vertici della patch originale.

```
vec3 p0 = gl_TessCoord.x * tcPosition[0];  
vec3 p1 = gl_TessCoord.y * tcPosition[1];  
vec3 p2 = gl_TessCoord.z * tcPosition[2];  
vec3 position = p0 + p1 + p2;
```

Listato 3.19: Calcolo dell'effettiva posizione del vertice dato `gl_TessCoord`.

Nel listato 3.20 è riportato un esempio di tessellation evaluation shader, capace di applicare l'illuminazione a ciascun vertice. Il risultato può essere visto nelle figure 3.5 e 3.6.

```

#version 400
layout(triangles, equal_spacing, ccw) in;

in vec3 tcPosition[];
in vec3 tcNormal[];
in vec4 tcColor[];
in float tcmatshininess[];
in vec3 tcmatspecular[];
out vec4 tecolor;

uniform mat4 Projection;
uniform mat4 Modelview;
uniform mat3 NormalMatrix;
uniform vec4 LightPosition;
uniform vec3 LightDiffuse;
uniform vec3 LightAmbient;
uniform vec3 LightSpecular;

vec4 ApplyLight(in vec4 oColor, in vec3 relativepos, in vec3 normal,
in vec3 lightpos, in vec3 lightambient, in vec3 lightdiffuse,
in vec3 lightspecular, in float matshininess, in vec3 matspecular);

void main()
{
    // posizione del vertice
    vec3 p0 = gl_TessCoord.x * tcPosition[0];
    vec3 p1 = gl_TessCoord.y * tcPosition[1];
    vec3 p2 = gl_TessCoord.z * tcPosition[2];
    vec3 position = p0 + p1 + p2;

    // normale del vertice
    p0 = gl_TessCoord.x * tcNormal[0];
    p1 = gl_TessCoord.y * tcNormal[1];
    p2 = gl_TessCoord.z * tcNormal[2];
    vec3 normal = p0 + p1 + p2;
    normal = normalize(NormalMatrix * normal);

    // calcola questi valori come media dei valori in ingresso
    vec4 avgcolor = (tcColor[0] + tcColor[1] + tcColor[2]) / 3.0;
    float avgshini = (tcmatshininess[0] + tcmatshininess[1] +
    tcmatshininess[2]) / 3.0;
    vec3 avgspec = (tcmatspecular[0] + tcmatspecular[1] +
    tcmatspecular[2]) / 3.0;
    // illuminazione
    vec4 rp4 = Modelview * vec4(position, 1.0);
    vec3 relativepos = rp4.xyz / rp4.w;
    tecolor = ApplyLight(avgcolor, relativepos, normal, LightPosition.rgb /
    LightPosition.w, LightAmbient, LightDiffuse,
    LightSpecular, avgshini, avgspec);
    gl_Position = Projection * Modelview * vec4(position, 1.0);
}

```

Listato 3.20: Un tessellation evaluation shader.

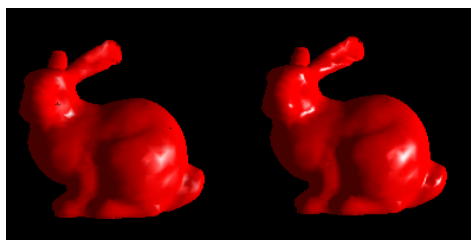


Figura 3.5: Lo Stanford Bunny non tessellato (a sinistra) e tessellato (a destra) con *TessLevelInner* e *TessLevelOuter* pari a 10 ciascuno. L'illuminazione è per vertice, perciò più vertici significa maggior precisione.

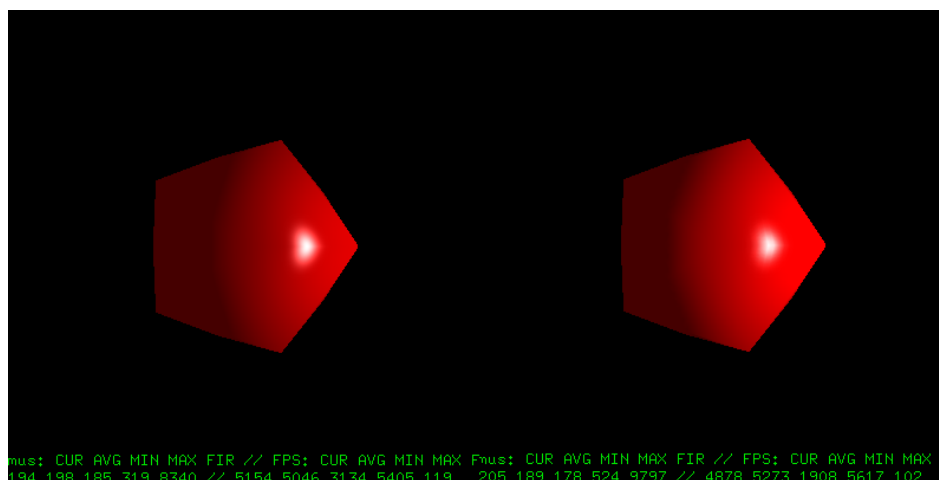


Figura 3.6: A sinistra, l'illuminazione per pixel di un icosaedro ottenuta tramite il Phong shading. A destra, lo stesso icosaedro illuminato per vertice (vedere sezione 4.1) e tessellato con valori di *TessLevelInner* e *TessLevelOuter* pari a 20 ciascuno. La suddivisione in primitive nel secondo caso non è visibile perché esse sono troppo piccole e le immagini appaiono molto simili. Purtroppo anche il tempo di visualizzazione medio è solo di pochi microsecondi favorevole all'uso dei tessellation shader. Non si può dire quale metodo sia il migliore.

Capitolo 4

Algoritmi grafici

4.1 Illuminazione

Purtroppo, la creazione di un vertex shader esclude immediatamente la funzione di illuminazione, che tra le altre cose permette di avere un'idea della profondità dell'oggetto. Perciò, è bene reimplementarla.

GLSL ha accesso a quasi tutte le variabili di illuminazione di OpenGL. Un vettore di strutture del tipo definito nel listato 4.1, una per ogni luce, è disponibile come variabile globale all'interno di ogni shader.

```
struct gl.LightSourceParameters
{
    vec4 position; // posizione della luce (xyzw)

    vec4 ambient; // componente ambientale della luce (rgba)
    vec4 diffuse; // componente diffusiva della luce (rgba)
    vec4 specular; // componente speculare della luce (rgba)
    vec4 halfVector;
    vec3 spotDirection;
    float spotExponent;
    float spotCutoff;
    float spotCosCutoff;
    float constantAttenuation;
    float linearAttenuation;
    float quadraticAttenuation;
}
```

Listato 4.1: Tipo della struttura che contiene le informazioni su una singola luce.

La costante globale `gl_MaxLights` contiene il numero massimo di luci che il driver (la scheda video) supporta, perciò questo è anche il numero di elementi del vettore (listato 4.2).

```
gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

Listato 4.2: *Definizione della variabile globale che contiene tutte le informazioni su tutte le luci.*

Allo stesso modo, il tipo definito nel listato 4.3 contiene le caratteristiche del materiale della primitiva corrente, ed è definito nelle variabili di cui al listato 4.4.

```
struct gl_MaterialParameters
{
    vec4 emission; // luce emessa dalla primitiva (rgba)
    vec4 ambient; // componente ambientale del materiale (rgba)
    vec4 diffuse; // componente diffusiva del materiale (rgba)
    vec4 specular; // componente speculare del materiale (rgba)
    float shininess; // tendenza del materiale a riflettere la luce speculare
}
```

Listato 4.3: *Tipo della struttura che contiene le informazioni su un singolo materiale.*

```
gl_MaterialParameters gl_FrontMaterial; // da glMaterialfv(GL_FRONT, ...)
gl_MaterialParameters gl_BackMaterial; // da glMaterialfv(GL_BACK, ...)
```

Listato 4.4: *Definizione delle variabili globali con l'informazione sul materiale, per le parti davanti (front) e dietro (back) della faccia.*

Uno shader che replica esattamente l'illuminazione di OpenGL sarebbe molto complesso, perché dovrebbe tenere conto della possibilità che la luce sia direzionale, puntiforme, spot light, abbia attenuazione eccetera.

Tuttavia, uno dei vantaggi degli shader è proprio la possibilità di implementare, della Fixed Function Pipeline, solo le funzionalità che servono.

A titolo di esempio, si suppone che si voglia applicare il modello di Phong a ciascun vertice (Gouraud shading). È leggermente diverso dal comportamento di

OpenGL di default, che usa il modello di Blinn-Phong, ma può bastare. C'è una singola luce attiva puntiforme omnidirezionale. Inoltre, il materiale non emette luce e riflette la luce ambientale e diffusiva secondo il suo colore.

```

vec4 ApplyLight(in vec4 oColor, in vec3 relativepos,
in vec3 normal,in int lightIndex)
{
vec3 intensity = vec3(0.0,0.0,0.0);

vec3 lightpos = gl_LightSource[lightIndex].position.xyz /
gl_LightSource[lightIndex].position.w;
vec3 lightrelative = normalize(lightpos - relativepos);
// incident light vector

/* ambient */
intensity += gl_LightSource[lightIndex].ambient.rgb *
gl_LightSource[lightIndex].ambient.a * oColor.rgb;

float dotp = dot(lightrelative,normal);
if (dotp > 0.0)
/* if face is not oriented towards light, skip specular and diffuse */
{
/* diffuse */
{
vec3 diffuse = gl_LightSource[lightIndex].diffuse.rgb *
dotp * gl_LightSource[lightIndex].diffuse.a * oColor.rgb;
intensity += diffuse;
}
/* specular */
{
vec3 reflected = reflect(lightrelative,normal);
vec3 viewr = normalize(relativepos);
float weight = pow(max(0.0,dot(reflected,viewr)),
gl_FrontMaterial.shininess/4.0);
// divide shininess by 4 to make it similar to Blinn-Phong model
intensity += gl_LightSource[lightIndex].specular.rgb *
max(weight,0.0) * gl_LightSource[lightIndex].specular.a *
gl_FrontMaterial.specular.rgb * gl_FrontMaterial.specular.a;
}
}
}

return vec4(clamp(intensity,0.0,1.0),oColor.a);
}

```

Listato 4.5: Funzione che applica il modello di Phong.

`oColor` è il colore originale del vertice, `relativepos` è la posizione del vertice in eye coordinates (dopo aver applicato la matrice di modelview), `normal` è la normale, anch'essa in eye coordinates. La funzione restituisce il nuovo colore del vertice, dopo aver applicato l'illuminazione.

Il parametro `lightIndex` è l'indice della luce che deve essere applicata. Infatti, si nota che nessuna delle strutture elencate in precedenza fornisce l'informazione sull'abilitazione o meno della luce (con `glEnable(GL_LIGHTn)`). Inoltre, in molte architetture i parametri di una luce non usata vanno per default quasi tutti a 1.0, non a 0.0, perciò risulta impossibile distinguere tra una luce disattivata e una bianca.

```
#version 120

vec4 ApplyLight(in vec4 oColor, in vec3 relativepos, in vec3 normal, in int ←
    lightIndex);

void main()
{
    vec3 normal = normalize(gl_NormalMatrix * gl_Normal);
    // la normale DEVE essere sempre normalizzata dopo
    // una moltiplicazione
    vec4 rp = gl_ModelViewMatrix * gl_Vertex;
    rp /= rp.w;
    gl_FrontColor = ApplyLight(gl_Color, rp.xyz, normal, 0);

    gl_BackColor = vec4(0.0, 0.0, 0.0, 1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Listato 4.6: *Vertex shader che usa la funzione `ApplyLight`.*

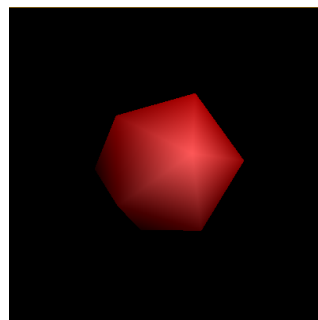


Figura 4.1: *Risultato dell'applicazione dello shader di cui al listato 4.6 a un icosaedro. L'illuminazione di OpenGL è stata ripristinata.*

In realtà, l'uso delle strutture sull'illuminazione è inefficiente e deprecato. Esse sono ridondanti: prevedono una dozzina di variabili diverse per ogni luce, di cui poche sono usate ogni volta. È consigliato (e versioni di GLSL superiori a 1.20 lo

impongono) che queste informazioni siano passate allo shader attraverso opportuni attributi e variabili uniform.

```

uniform vec3 lightlposition;
uniform vec3 lightldiffuse;
uniform vec3 lightlspecular;
uniform vec3 lightlambient;

attribute float matshininess;
attribute vec3 matspecular;

vec4 ApplyLight(in vec4 oColor, in vec3 relativepos, in vec3 normal)
{
    vec3 intensity = vec3(0.0,0.0,0.0);

    vec3 lightrelative = normalize(lightlposition - relativepos);
    // incident light vector

    /* ambient */
    intensity += lightlambient * oColor.rgb;

    float dotp = dot(lightrelative,normal);
    if (dotp > 0.0)
        /* if face is not oriented towards light, skip specular and diffuse */
        {
            /* diffuse */
            {
                vec3 diffuse = lightldiffuse * dotp * oColor.rgb;
                intensity += diffuse;
            }
            /* specular */
            {
                vec3 reflected = reflect(lightrelative,normal);
                vec3 viewr = normalize(relativepos);
                float weight = pow(max(0.0, dot(reflected,viewr)),matshininess);
                intensity += lightlspecular * max(weight,0.0) * matspecular;
            }
        }

    return vec4(clamp(intensity,0.0,1.0),oColor.a);
}

```

Listato 4.7: Nuova funzione *ApplyLight*, che non usa nessuna delle variabili di stato di OpenGL.

Tra le altre cose si è risparmiato un parametro per la funzione, nonché alcune moltiplicazioni per gli (inutili) alpha del colore della luce. Inoltre, la divisione per 4 della shininess non è più necessaria.

Se le variabili sono impostate correttamente dall'applicazione, il risultato dell'uso della versione nel listato 4.7 è identico a quello precedente (figura 4.1).

A questo punto, è possibile calcolare l'illuminazione per singolo pixel. Questa è una delle funzionalità che non possono essere implementate con la sola Fixed Function Pipeline, che invece si limita ad attribuire il colore ad ogni vertice e ad interpolarlo.

Il calcolo dell'illuminazione per pixel ha il vantaggio di una maggiore precisione. In più, alcuni effetti grafici (tra cui le texture) prevedono di calcolare il colore della primitiva all'interno del fragment shader, perciò alcune parti dell'illuminazione non possono che essere applicati successivamente. Lo svantaggio consiste ovviamente in un calo delle prestazioni, perché la GPU deve compiere i calcoli dell'illuminazione per singolo pixel e non per vertice.

Il Phong shading è un algoritmo che prevede il calcolo dell'illuminazione per singolo pixel. Deve essere calcolata la normale ad ogni vertice della primitiva, poi tale normale è interpolata su ogni frammento. Si applica poi il solito modello di Phong, con componente ambientale, diffusiva e speculare.

All'inizio, il vertex shader (listato 4.8) calcola la normale e la posizione (relativa al punto di vista) di ciascun vertice e le passa avanti come variabili *varying*. Inoltre, deve trasmettere al fragment shader anche i dati sul materiale, perché i fragment shader non possono accedere alle variabili *attribute*.

```
#version 120
attribute float matshininess;
attribute vec3 matspecular;
varying float vmatshininess;
varying vec3 vmatspecular;
varying vec3 veyeposition;
varying vec3 vnormal;

void main()
{
    vnormal = gl_NormalMatrix * gl_Normal;
    vec4 rp = gl_ModelViewMatrix * gl_Vertex;
    veyeposition = rp.xyz / rp.w;
    gl_FrontColor = gl_Color;
    gl_BackColor = vec4(0.0,0.0,0.0,1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vmatspecular = matspecular;
    vmatshininess = matshininess;
}
```

Listato 4.8: *Il vertex shader per il Phong shading.*

La funzione `ApplyLight` rimane uguale, eccetto che il suo prototipo deve avere due parametri in più, perché gli attributi `mat specular` e `mat shininess` non possono più essere letti come variabili globali. Ora, la funzione deve essere caricata come fragment shader, non come vertex shader.

Il fragment shader (listato 4.9) si limita ad applicare l'illuminazione per ogni pixel. Si noti che la normale deve essere normalizzata prima di essere utilizzata, perché a causa dell'interpolazione può non avere più modulo 1.

```
#version 120

varying float vmatshininess;
varying vec3 vmatspecular;

varying vec3 veyeposition;
varying vec3 vnormal;

vec4 ApplyLight(in vec4 oColor, in vec3 relativepos, in vec3 normal, in vec3 ←
    matspecular, in float matshininess);

void main()
{
    gl_FragColor = ApplyLight(gl_Color, veyeposition, normalize(vnormal),
        vmatspecular, vmatshininess);
}
```

Listato 4.9: *Il fragment shader per il Phong shading.*

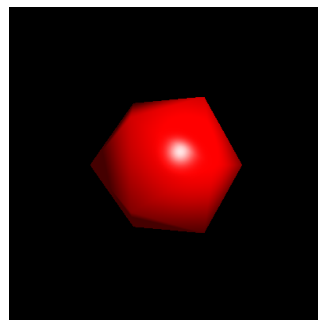


Figura 4.2: *Phong shading applicato all'icosaedro.*

4.2 Calcolo delle normali

È possibile usare un geometry shader per aumentare il numero di vertici allo stesso modo di un tessellation shader, perché può emettere qualunque numero di primitive data una singola primitiva in ingresso. Tuttavia, il geometry shader è inefficiente, perché deve emettere un vertice alla volta ed infrange il parallelismo della scheda video.

L'uso del geometry shader è consigliato più che altro per eseguire compiti che hanno la necessità di conoscere un'intera primitiva, non solo i suoi singoli vertici. Uno di questi compiti è il calcolo delle normali. La normale a un triangolo può essere ottenuta come il prodotto vettoriale tra due lati. Per cominciare, si definisce una funzione `ComputeNormal` che riceve in ingresso i tre vertici e restituisce la normale, tramite il prodotto scalare di due lati della primitiva (listato 4.10).

```
vec3 ComputeNormal(in vec3 v1, in vec3 v2, in vec3 v3)
{
    vec3 diff1 = v2 - v1; // primo lato
    vec3 diff2 = v2 - v3; // secondo lato

    vec3 crossproduct = cross(diff1, diff2);

    return normalize(crossproduct);
}
```

Listato 4.10: *La funzione `ComputeNormal`.*

Questo algoritmo è meno efficace di usare le normali calcolate per singolo vertice, se sono disponibili nella geometria originale, perché calcola la stessa normale per tutta la faccia (vedere il risultato alla fine della sezione, in figura 4.3). L'effetto finale è più simile al faceted shading, anche applicando l'illuminazione per pixel.

Il vertex shader non deve calcolare la normale, né passarla avanti con una variabile `varying` (listato 4.11). Il fragment shader è identico a quello visto nella sezione 4.1 (listato 4.9), con la differenza che per evitare duplicati è stato necessario rinominare tutte le variabili destinate al fragment shader.

Il geometry shader (listato 4.12) si limita a copiare verso il fragment shader tutte le variabili, eccetto `gnormal`, che invece deve calcolare. Il vertex shader fornisce

```

#version 120

attribute float matshininess;
attribute vec3 matspecular;
varying float vmatshininess;
varying vec3 vmatspecular;
varying vec3 veyeposition;

void main()
{
    vec4 rp = gl_ModelViewMatrix * gl_Vertex;
    veyeposition = rp.xyz / rp.w;

    gl_FrontColor = gl_Color;
    gl_BackColor = vec4(0.0,0.0,0.0,1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    vmatspecular = matspecular;
    vmatshininess = matshininess;
}

```

Listato 4.11: *Il vertex shader per il calcolo delle normali.*

già i vertici in eye coordinates (`veyeposition`), perciò non è necessaria nessuna moltiplicazione per `gl_NormalMatrix`.

```

#version 120
#extension GL_EXT_geometry_shader4: enable

varying out float gmatshininess;
varying out vec3 gmatspecular;
varying out vec3 geeyeposition;
varying out vec3 gnormal;

varying in float vmatshininess[3];
varying in vec3 vmatspecular[3];
varying in vec3 veyeposition[3];

vec3 ComputeNormal(in vec3 v1,in vec3 v2,in vec3 v3);

void main()
{
    gnormal = ComputeNormal(veyeposition[0],veyeposition[1],veyeposition[2]);

    for (int i = 0; i < 3; i++)
    {
        gl_FrontColor = gl_FrontColorIn[i];
        gl_BackColor = gl_BackColorIn[i];
        gl_Position = gl_PositionIn[i];

        gmatshininess = vmatshininess[i];
        gmatspecular = vmatspecular[i];
        geeyeposition = veyeposition[i];

        EmitVertex();
    }

    EndPrimitive();
}

```

Listato 4.12: *Il geometry shader per il calcolo delle normali.*

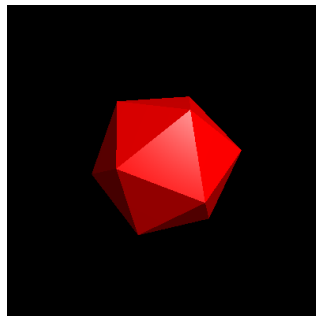


Figura 4.3: *Icosaedro visualizzato con le normali calcolate dal geometry shader.*

4.3 Bump mapping

Il bump mapping è una tecnica per simulare piccole imperfezioni su una superficie attraverso una piccola alterazione delle normali. Per ottenere questo effetto, di solito si applica una texture alla superficie e si usa il colore di quella texture come se fosse la normale all'interno del fragment shader. Tuttavia, se l'alterazione è casuale come in questo esempio, esiste un modo più semplice.

Non esiste una funzione `rand` all'interno degli shader. Anche se esistesse, non sarebbe utilizzabile in questa occasione, perché il suo valore cambierebbe ad ogni esecuzione dello shader e non sarebbe costante tra un ciclo di rendering e il successivo.

Invece, si immerge l'oggetto in una texture tridimensionale che si ripete in tutto lo spazio. In questo modo, si può usare la posizione (nelle coordinate di modello) del punto per indicizzare la superficie, senza bisogno di importare le coordinate aggiuntive della texture per ciascun vertice.

Oltre a quanto visto nella sezione 4.1, il vertex shader deve solo passare al fragment shader la posizione originale del punto, prima di essere trasformato dalla matrice di modelview e di proiezione (listato 4.13).

Per il fragment shader, invece, le modifiche sono leggermente più complesse. In questo esempio, si considera che la texture è inizializzata con valori casuali da 0.0 a 1.0 e che il programma la assegna alla variabile mostrata nel listato 4.14.

Si ottiene il punto in cui valutare la texture a partire dalla posizione originale proveniente dallo shader, opportunamente scalato. Modifiche a questo fattore di scala aumentano o diminuiscono le dimensioni delle imperfezioni sulla superficie, perché diminuiscono o aumentano le dimensioni dell'oggetto visualizzato rispetto alla texture.

Poi si ottiene la modifica alla normale sottraendo alla texture in quel punto 0.5, in modo che essa vari tra -0.5 e 0.5. Si moltiplica per un altro fattore di scala, che controlla la profondità delle imperfezioni.

La normale deve essere normalizzata due volte: una prima di applicare la modifica, perché è stata interpolata, e una dopo, per assicurarsi che abbia sempre modulo 1 dopo il bump mapping (listato 4.15).

```

#version 120

attribute float matshininess;
attribute vec3 matspecular;
varying float vmatshininess;
varying vec3 vmatspecular;
varying vec3 veyeposition;
varying vec3 vnormal;
varying vec3 oPosition;

void main()
{
    vnormal = gl_NormalMatrix * gl_Normal;
    vec4 rp = gl_ModelViewMatrix * gl_Vertex;
    veyeposition = rp.xyz / rp.w;
    oPosition = gl_Vertex.xyz / gl_Vertex.w;
    gl_FrontColor = gl_Color;
    gl_BackColor = vec4(0.0,0.0,0.0,1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vmatspecular = matspecular;
    vmatshininess = matshininess;
}

```

Listato 4.13: *Il vertex shader per il Bump Mapping.*

```

uniform sampler3D random3d;

```

Listato 4.14: *La texture casuale.*

Nel listato 4.16 è riportato il fragment shader finale, completo di illuminazione. Il risultato è visibile nella figura 4.4.

```
vec3 bump = (texture3D(random3d,oPosition * 0.1).xyz -
vec3(0.5,0.5,0.5)) * 0.1;
vec3 newnormal = normalize(normalize(vnormal) + bump);
```

Listato 4.15: *Il calcolo della nuova normale, dopo il bump mapping.*

```
#version 120

varying float vmatshininess;
varying vec3 vmatspecular;

varying vec3 veyeposition;
varying vec3 vnormal;

vec4 ApplyLight(in vec4 oColor, in vec3 relativepos, in vec3 normal,in vec3←
matspecular,in float matshininess);

uniform sampler3D random3d;

varying vec3 oPosition;

void main()
{
vec3 bump = (texture3D(random3d,oPosition * 0.1).xyz - vec3(0.5,0.5,0.5))←
* 0.2;
vec3 newnormal = normalize(normalize(vnormal) + bump);

gl.FragColor = ApplyLight(gl.Color,veyeposition,newnormal,vmatspecular,←
vmatshininess);
}
```

Listato 4.16: *Il fragment shader completo.*

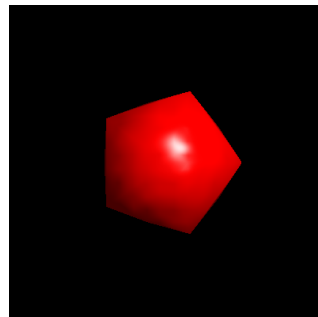


Figura 4.4: *Bump mapping applicato all'icosaedro.*

4.4 Riflessione

Esistono diversi modi di simulare la riflessione in grafica. Il metodo più grezzo è semplicemente la duplicazione dell'oggetto al di là della superficie riflettente. Questa soluzione può essere implementata, con qualche difficoltà, attraverso l'uso di un geometry shader che produca due primitive per ogni primitiva ricevuta, una nella posizione originale e una nella posizione in cui ci dovrebbe essere il riflesso. Si può fare, ma introduce alcuni problemi nel caso in cui la superficie riflettente non sia descritta da un'equazione analitica molto semplice, oltre a ridurre l'efficienza.

In questo esempio, invece, si usa un altro metodo, meno preciso. Per prima cosa, tutto l'ambiente in cui è immerso l'oggetto riflettente è proiettato su una cube map, di cui l'oggetto è al centro. La cube map viene fornita allo shader tramite la variabile uniform mostrata nel listato 4.17.

```
uniform samplerCube cubetex;
```

Listato 4.17: *La definizione di una variabile uniform per contenere una cube map.*

La proiezione può essere eseguita tramite OpenGL, ma non è oggetto di questa tesi, perciò sono state usate alcune texture già fatte. Tuttavia, è bene sapere che la proiezione introduce un'approssimazione, perché lo shader successivamente non può fare altro che considerare la cube map come se fosse a distanza infinita dal centro. Questo può causare problemi: oggetti vicini all'oggetto riflettente possono risultare riflessi nel posto sbagliato.

Il vertex shader (listato 4.18) deve passare al fragment shader la posizione (`relativepos`) del punto in eye coordinates, cioè l'opposto del vettore incidente che parte dal punto di vista e colpisce l'oggetto in quel vertice. Deve passare anche la normale al vertice. Entrambi questi valori vengono interpolati sulla superficie durante il passaggio.

Il fragment shader riceve la normale e la posizione in eye coordinates. A questo punto, riflette la posizione relativa rispetto alla normale e ottiene il raggio visivo riflesso, che intercetta la cube map e ottiene il colore per quel pixel. A seconda di

```
#version 120
varying vec3 normal;
varying vec3 relativepos;

void main()
{
    vec4 pos = gl_Vertex;

    normal = gl_NormalMatrix * gl_Normal;
    vec4 rp = gl_ModelViewMatrix * pos;
    relativepos = rp.xyz / rp.w;

    gl_Position = gl_ModelViewProjectionMatrix * pos;
}
```

Listato 4.18: *Il vertex shader per la riflessione.*

come la cube map è stata caricata, può essere necessario oppure no usare l'opposto del raggio riflesso.

Per accedere alla cube map si usa la funzione `textureCube`. Essa richiede come parametri la cube map e il vettore di tre componenti che contiene la direzione del raggio visivo con cui indicizzarla (listato 4.19).

```
#version 120

varying vec3 normal;
varying vec3 relativepos;

uniform samplerCube cubetex;

void main()
{
    vec3 reflray = reflect(relativepos,normal);
    vec4 texel = textureCube(cubetex,-reflray);

    gl_FragColor = texel;
}
```

Listato 4.19: *Il fragment shader per la riflessione.*

Nelle immagini che seguono (4.5 e 4.6) è stato aggiunto alla scena un cubo molto grande, su cui è stata applicata una texture tramite un'altro shader molto semplice che si limita ad accedere alla cube map usando la posizione corrente del pixel rispetto al centro della scena (listato 4.20).

```
#version 120
varying vec3 position;

uniform samplerCube environmentmap;

void main()
{
    vec4 texel = textureCube(environmentmap, position.xyz);
    gl_FragColor = texel;
}
```

Listato 4.20: *Il fragment shader per ottenere lo sfondo.*

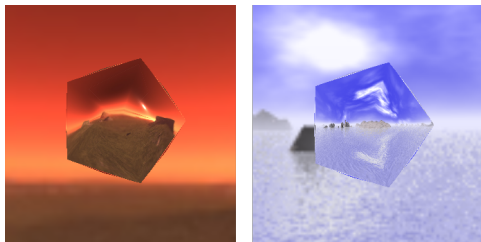


Figura 4.5: *Un icosaedro riflette due diversi paesaggi.*

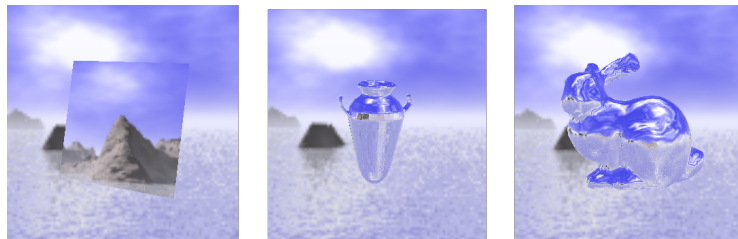


Figura 4.6: *Un piano, un'anfora e un coniglio riflettono lo stesso paesaggio.*

4.5 Silhouette

Un'applicazione interessante degli shader riguarda l'estrazione del contorno (silhouette) da una mesh di poligoni. Intuitivamente, il contorno è definito come l'insieme degli spigoli le cui facce adiacenti sono una rivolta verso il punto di vista e l'altra rivolta nel verso opposto.

Lavorando nel volume canonico, dopo le trasformazioni di vista e di proiezione, la regola si semplifica. Una faccia è rivolta lontano dall'osservatore se è rivolta nella direzione in cui la profondità aumenta, cioè la componente z della normale è positiva. Al contrario, se la componente z è negativa la faccia è rivolta verso l'osservatore. Ciò permette di affermare che se la normale di una faccia ha segno diverso da quella della faccia adiacente, lo spigolo tra le due è parte della silhouette.

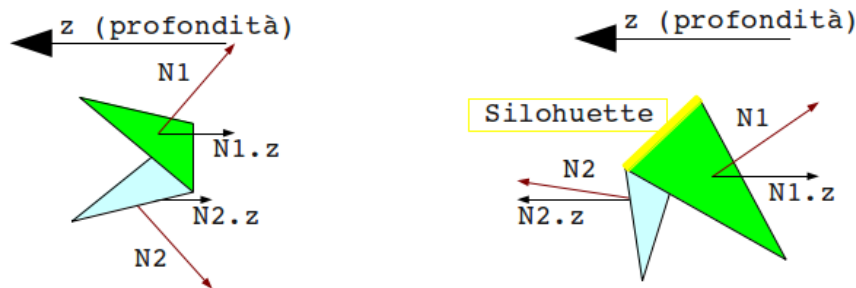


Figura 4.7: Lo spigolo tra le due facce a sinistra non è parte della silhouette, perché le proiezioni lungo z (in nero) delle normali (in rosso) hanno lo stesso segno. Al contrario, lo spigolo a destra è parte della silhouette.

Il vertex shader si limita a calcolare le coordinate nel volume canonico e a passare avanti il colore (listato 4.21).

Il fragment shader è ancora più semplice (listato 4.22), perché deve solo applicare il colore. Siccome si stanno disegnando delle linee, non c'è nessuna illuminazione.

L'elaborazione è eseguita tutta nel geometry shader (listato 4.23). Esso riceve in ingresso un triangolo con informazioni di adiacenza (`GL_TRIANGLES_ADJACENCY`) e produce in uscita i lati del triangolo (con `GL_LINE_STRIP`) che soddisfano la condizione di appartenere alla silhouette.

```
#version 120

varying vec3 position;

void main()
{
    vec4 newpos = gl_ModelViewProjectionMatrix * gl_Vertex ;
    position = newpos.xyz / newpos.w;

    gl_FrontColor = gl_Color;
    gl_BackColor = vec4(0.0,0.0,0.0,0.0);
}
```

Listato 4.21: *Vertex shader.*

```
#version 120

void main()
{
    gl_FragColor = gl_Color;
}
```

Listato 4.22: *Fragment shader.*

Oltre a quanto detto in precedenza, è stata applicata un'ottimizzazione. Per evitare che ciascuno spigolo della silhouette sia emesso due volte, uno per una faccia e uno per l'altra, il geometry shader si interrompe subito se la faccia non è orientata verso l'osservatore. Ciò ha effetto in alcuni casi anche se tutti gli spigoli della faccia non fanno parte della silhouette, ma questo non ha importanza, perché non sarebbe stata visualizzata comunque.

Per quanto detto nella sezione Parallelismo (2.4), l'ottimizzazione non ha l'effetto di ridurre il tempo di esecuzione del Geometry Shader, ma solo quello di evitare di produrre due segmenti per ogni spigolo della silhouette e appesantire la fase di rasterizzazione.

Il risultato è visibile nelle figure 4.8 e 4.9.

```

#version 120
#extension GL_EXT_geometry_shader4: enable

varying in vec3 position[6];

// FaceNormal.txt
vec3 ComputeNormal(in vec3 v1,in vec3 v2,in vec3 v3);

// emette il vertice che ha indice "index"
void Emit(in int index)
{
    gl_FrontColor = gl_FrontColorIn[index];
    gl_BackColor = gl_BackColorIn[index];
    gl_Position = vec4(position[index],1.0);
    EmitVertex();
}

void main()
{
    // calcolo della z della normale alla faccia
    float fz = ComputeNormal(position[2],position[0],position[4]).z;

    // ottimizzazione: non elaborare due volte lo stesso spigolo
    if (fz < 0.0)
        return;

    // calcolo della z delle normali alle facce adiacenti
    float z021 = ComputeNormal(position[0],position[2],position[1]).z;
    float z432 = ComputeNormal(position[4],position[3],position[2]).z;
    float z540 = ComputeNormal(position[5],position[4],position[0]).z;

    // sappiamo gia' che
    // la z della normale alla faccia corrente e' positiva
    // basta che quella adiacente sia negativa
    if (z021 < 0.0)
    {
        Emit(2);
        Emit(0);
        EndPrimitive();
    }

    if (z432 < 0.0)
    {
        Emit(2);
        Emit(4);
        EndPrimitive();
    }

    if (z540 < 0.0)
    {
        Emit(0);
        Emit(4);
        EndPrimitive();
    }
}

```

Listato 4.23: *Geometry shader.*

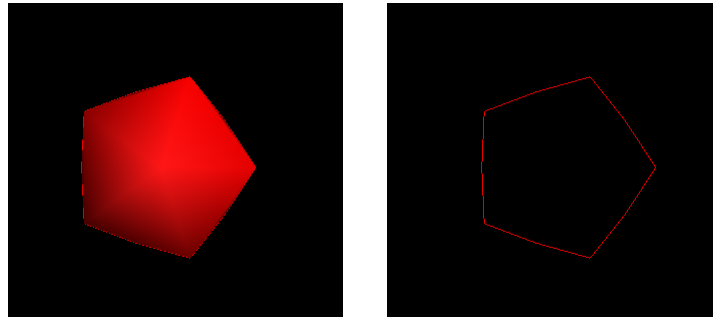


Figura 4.8: Silhouette di un icosaedro con (a sinistra) e senza (a destra) l'icosaedro disegnato all'interno.

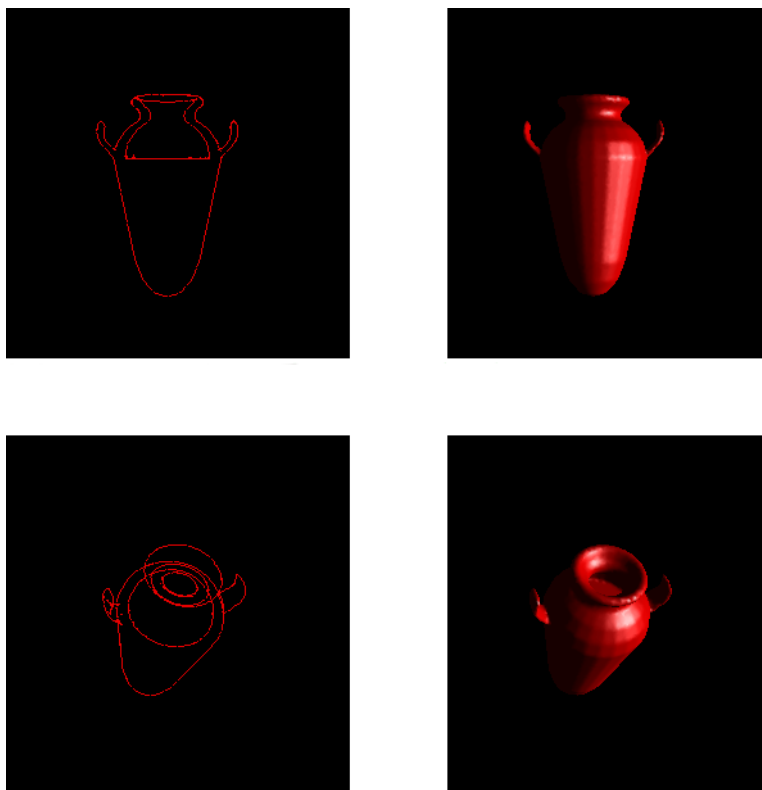


Figura 4.9: Un'anfora (a destra) e la sua silhouette (a sinistra). Si nota che chi ha fabbricato la mesh di poligoni dell'anfora ha troncato lo spazio interno in orizzontale per risparmiare.

4.6 Frattali

La visualizzazione di immagini che derivano da frattali può trarre grande vantaggio dagli shader. Queste immagini non possono essere precalcolate, perché possono essere assimilate a immagini a risoluzione infinita. D'altra parte, i pixel sono gli unici punti in cui importa veramente conoscere il valore dell'immagine, per cui ci si può limitare a calcolarla in questi punti.

Anche la CPU potrebbe eseguire l'algoritmo per ogni pixel, ma i Fragment Shader sono fatti apposta per processare ogni frammento di una primitiva, e possono anche sfruttare la capacità di parallelismo della GPU. Basta visualizzare una primitiva attraverso un particolare Fragment Shader che esegue l'algoritmo del frattale per ogni frammento e imposta il colore di conseguenza.

In questo esempio, è stata utilizzata la classica formula per l'insieme di Mandelbrot:

$$z_{n+1} = z_n^2 + c$$

I vari z_k e c sono numeri complessi. Per definizione, posto $z_0 = 0$, c appartiene all'insieme di Mandelbrot se z_n converge per n tendente all'infinito.

L'unico modo per sapere in un tempo finito se la successione converge oppure no è calcolarne un gran numero di valori e dire che è divergente se il modulo di uno di essi supera un valore molto grande rispetto ai dati iniziali. Questo algoritmo approssimato ha due svantaggi: è necessario decidere arbitrariamente il valore massimo, e soprattutto bisogna definire un numero massimo di iterazioni, altrimenti nei casi convergenti l'algoritmo proseguirebbe all'infinito.

Per ottenere l'insieme di Mandelbrot, si dovrebbe partire da $z_0 = 0$ e inserire le coordinate (di modello) del frammento in c , con la formula $c = x + iy$ (la z è costante). Dopodiché si assegnerebbe un colore diverso al pixel a seconda del numero di iterazioni necessarie per far superare alla successione il valore prestabilito.

In questo esempio, invece, si è tentato di ottenere qualcosa di più originale. Si assegnano le coordinate del frammento a z_0 e c invece è costante. Il colore è assegnato in funzione del contenuto di z quando il frattale diverge: i due colori sono mischiati insieme in funzione del valore di parte reale e immaginaria. Se la successione converge, invece, viene scelto un colore che è la media tra i due.

Il concetto matematico non è molto diverso, ma questo spiega perché compare quell'insolita forma spinosa invece di una classica immagine di un frattale famoso. Il listato 4.24 riporta il fragment shader utilizzato.

```

uniform vec2 initC;
uniform int numIter;
varying vec3 position;
uniform vec4 color1;
uniform vec4 color2;

vec2 Fractal(float x, float y)
{
    vec2 c = initC;
    vec2 z = vec2(x,y);

    for (int i = 0; i < numIter; i++)
    {
        vec2 sum = z + c;
        z = vec2(sum.x*sum.x - sum.y*sum.y, 2.0*sum.x*sum.y);

        if (abs(dot(z,z)) > 30.0)
            return normalize(z);
    }

    return vec2(0.5,0.5);
}

void main()
{
    vec2 res = Fractal(position.x,position.y);

    vec4 color = res.x * color1 + res.y * color2;
    gl_FragColor = clamp(color,0.0,1.0);
}

```

Listato 4.24: *Fragment shader per la visualizzazione di un frattale.*

Il vertex shader 4.25 è molto semplice e si limita a passare avanti la posizione originale di ciascun vertice, che sarà poi interpolata nel fragment shader:

Le immagini in figura 4.10 mostrano il risultato della visualizzazione di un quadrato con questi shader. La variabile `initC` è stata impostata a $0.285 + 0.013i$, mentre il numero di iterazioni `numIter` è 400. I colori 1 e 2 sono rispettivamente il verde (rgb: 0.0, 0.9, 0.0) e il blu (rgb: (0.0, 0.0, 0.9)).

Per finire, si nota che un'immagine di un frattale generato in questo modo si comporta esattamente come una texture (immagine a dimensione finita). La primitiva attraversa tutta la pipeline e può essere tralata, ruotata e illuminata. Può

```

varying vec3 position;

void main()
{
    vec4 newpos = gl_Vertex;
    position = newpos.xyz / newpos.w;
    gl_Position = gl_ModelViewProjectionMatrix * newpos;
}

```

Listato 4.25: *Vertex shader.*

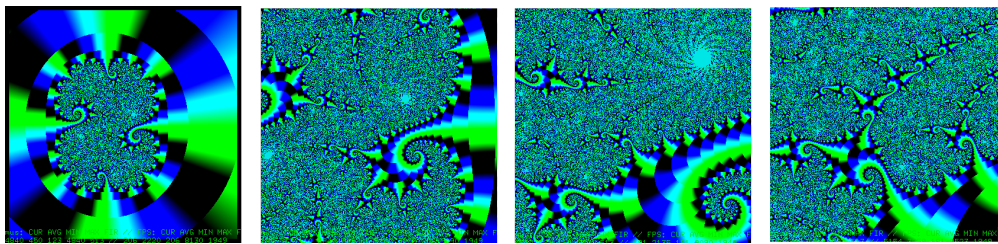


Figura 4.10: *Tra ciascuna immagine e quella alla sua sinistra c'è un fattore di ingrandimento di circa 2,5.*

comparire nella scena insieme ad altre primitive più tradizionali e i suoi frammenti subiscono il depth test come le altre (figura 4.11, a sinistra).

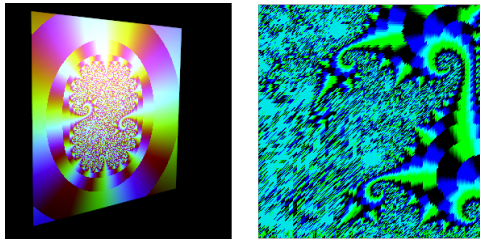


Figura 4.11: *A sinistra, la primitiva su cui è il frattale, ruotata ed illuminata. A destra, il frattale con ingrandimento pari a 13780 volte, che comincia ad essere eccessivo per la precisione delle variabili floating point utilizzate.*

Capitolo 5

Conclusione

Naturalmente, il discorso sugli shader è tutt'altro che concluso. Essi sono molto di più di un effetto grafico, un insieme di effetti grafici o un algoritmo. La programmazione degli shader consente l'uso di un nuovo linguaggio e di un intero processore, perciò è impossibile circoscrivere il numero di algoritmi e funzionalità che sono state sviluppate in più di otto anni e possono essere sviluppate in futuro.

Già in questa trattazione, per motivi di tempo e di spazio, è stato necessario ridurre il numero di esempi riportati, e di questi semplificare il codice. Del resto, ogni nuovo shader offriva spunti per modifiche e varianti, le quali a loro volta potevano generare nuove idee, sempre più complicate. Proseguendo per questa strada, il percorso non avrebbe mai avuto una fine.

E ci sarebbero state anche altre strade da prendere. La GPU sta cominciando a diventare un processore di uso generale, come la CPU. Già oggi alcuni programmi di uso comune possono avvalersi della sua capacità di elaborazione parallela per eseguire calcoli che hanno poco o nulla a che fare con la visualizzazione di grafica tridimensionale sullo schermo. Un esempio di questo sono i decoder di formati compressi per i filmati, che possono sfruttare l'accelerazione hardware della GPU.

Invece, si spera di aver mostrato almeno la *direzione* intrapresa dalla programmazione grafica. Non è più questione di impostare i parametri della Fixed Function Pipeline per ottenere l'effetto desiderato tra quelli disponibili. Piuttosto, si modifica la pipeline per ottenere quell'effetto in modo più efficiente, versatile e personalizzabile.

Vista la mole di lavoro necessaria per reimplementare da zero, ad esempio, l'illuminazione, questo approccio ha certamente i suoi svantaggi. Eppure, gli esempi riportati dimostrano che è stato possibile generare effetti altrimenti molto difficili e inefficienti da ottenere usando solo la programmazione tradizionale della CPU.

Bibliografia

- [1] Mike Bailey and Steve Cunningham, *Graphics Shaders: Theory and Practice*. AK Peters, 2009
- [2] *OpenGL 2.1 Reference Pages*
<http://www.opengl.org/sdk/docs/man/>
- [3] *OpenGL 4.2 Reference Pages*
<http://www.opengl.org/sdk/docs/man4/>
- [4] *OpenGL Shading Language (GLSL) Reference Pages*
<http://www.opengl.org/sdk/docs/manglsl/>
- [5] *GLSL Tutorial Lighthouse3d.com*
<http://www.lighthouse3d.com/tutorials/glsl-tutorial/>
- [6] *GLSL Core Tutorial Lighthouse3d.com*
<http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/>
- [7] *The Little Grasshopper*
<http://prideout.net/blog/>