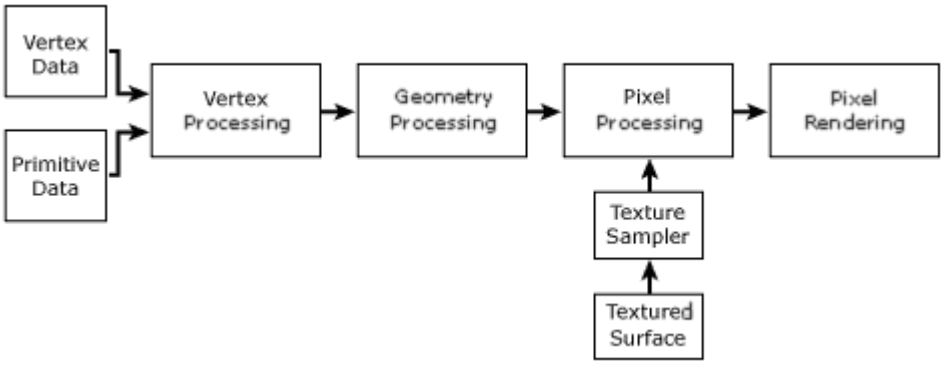
De Rerum Shader

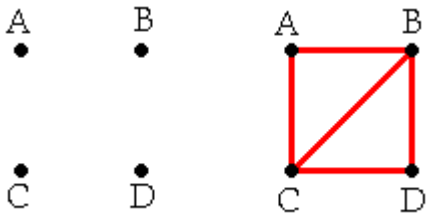
Roberto Nacchia
www.notjustcode.it

Questo vuole essere una sorta di articolo definitivo alla realizzazione degli shader. Sarà trattato in maniera generale in modo da adattarsi sia a DirectX9 che a DirectX10 e sarà diviso in più sezioni. Viene data per scontata una conoscenza sufficiente di DirectX e viene consigliata la lettura a chi sta iniziando a scrivere gli Shader e a chi vuole saperne di più sull'argomento.

Pipeline grafica



Questa che vedete è la pipeline grafica di DirectX9 che useremo per spiegare cosa succede nel momento in cui mandate in rendering un oggetto. Un oggetto 3D come sappiamo è composto da 2 dati principali: i vertici (Vertex Data) e gli indici (Primitive Data). I vertici sono i punti che compongono la figura e gli indici ne indicano l'ordine con cui vanno presi per formare i triangoli che rappresentano l'oggetto. Tali dati sono visibili nei 2 box di ingresso. I vertici sono strutture dati contenenti obbligatoriamente la posizione ed altri dati accessori (ad esempio normali per indicare la direzione in cui è rivolto il vertice, la coordinata texture, etc). Tali dati devono essere comunque considerati arbitrari in quando spesso si andranno ad utilizzare per scopi totalmente diversi (ad esempio usare una coordinata texture per indicare una direzione in cui muovere l'oggetto). Gli indici invece indicano solo l'ordine da utilizzare per prendere i vertici.



In questo esempio i vertici sono A, B, C e D. Immaginate ognuno come un insieme di byte. Gli indici per formare i 2 triangoli ABC e BCD saranno quindi 0, 1, 2, 1, 2, 3.
Negli indici si troveranno i valori 0,1,2,1,2,3 sotto forma di short (16bit fino a 65536 valori) o di int (32bit fino a 4 miliardi).
I vertici (VertexBuffer) e gli indici (IndexBuffer) sono i dati contenuti nel Vertex Data e nel Primitive Data. Quando utilizzate il rendering tramite DrawPrimitive li vedete perché sono contenuti negli oggetti VertexBuffer ed IndexBuffer. Quando utilizzate una mesh essi sono in realtà nascosti nella classe mesh (la mesh è semplicemente una classe che contiene VertexBuffer, IndexBuffer e metodi e proprietà per fare le operazioni di rendering al posto vostro).

Quello che accadrà sarà sempre un SetStream per inserire il VertexBuffer e un SetIndices per inserire gli indici. Oltre a questo nel device vengono di solito impostati il formato del vertice (per far conoscere a DirectX che tipo di dati ci sono nel VertexBuffer) ed una serie di dati arbitrari chiamati **costanti**.
Le costanti sono dati che la pipeline utilizza per processare i vertici ed i pixel. Un esempio sono le matrici World, View e Projection che vengono passate al device. Quando si utilizzano gli shader le costanti possono contenere qualsiasi cosa. Solitamente sono organizzati in registri da 4 float l'uno ma con l'avanzare delle versioni di Direct3D sono apparsi registri di altri tipi, come interi e booleani. Tra le costanti rientrano anche le texture applicate al modello.
Ora che tutti i buffer sono in memoria si può far partire il draw. DirectX fa questo:

1° Step

Ogni vertice viene elaborato dal vertex shader. Ognuno ha in input le stesse costanti di tutti gli altri vertici che vengono renderizzati con questa istruzione di draw. Le costanti comunque sono in sola lettura. Il Vertex Shader prenderà il vertice che ha posizione e dati così com'è stato caricato in memoria per poterlo trasformare nella posizione finale che avrà sullo schermo. Oltre alla posizione il Vertex Shader elabora anche altri dati che vogliamo utilizzare nelle fasi successive (esempio coordinate texture).
Il Vertex Shader restituisce un vertice modificato nei dati e nella struttura che viene inviato al Geometry processing.

2° Step

Terminata l'elaborazione di tutti i vertici il geometry processing crea i triangoli. Tale operazione è detta Rastering. I 3 vertici vengono caricati e si elaborano tutti i pixel contenuti nel triangolo. Ogni pixel è una struttura dati identica a quella restituita dal vertex shader ma con valori che sono la media pesata dei valori presenti nei 3 vertici in base alla distanza tra essi (al centro del triangolo il valore sarà la media esatta). Il geometry processing provvede anche ad eliminare i triangoli non validi (ad esempio fuori dallo schermo o che non supera uno stencil test) in modo che non vengano più elaborati dalle fasi successive. Al termine crea una serie di pixel, ognuno con una posizione sullo schermo ed un insieme di dati con se frutto del lavoro di interpolazione (il calcolo della media citata prima). I pixel vengono passati al pixel processing.

3° Step

Il pixel shader prende in ingresso ad uno ad uno la struttura dati di ogni pixel che viene rielaborato. Anche questo ha accesso alle costanti che sono condivise tra tutti i pixel. In ingresso si hanno anche le texture che possono essere lette come costanti e partecipare al calcolo. Il risultato finale sarà il colore di uscita espresso nei 4 canali Rosso, Verde, Blu ed Alpha (trasparenza).
Si passa quindi all'ultima fase, il fissaggio del colore. Se l'alphablending è attivato si legge il valore del pixel attualmente sullo schermo nel punto su cui stiamo per scrivere il nuovo pixel. Si fa quindi una operazione di confronto tra i due pixel (in base alle opzioni di blending) ed il risultato finale sarà il pixel che si troverà sullo schermo. A rientrare in gioco c'è anche l'alpha test che elimina i pixel in base al valore di alpha.

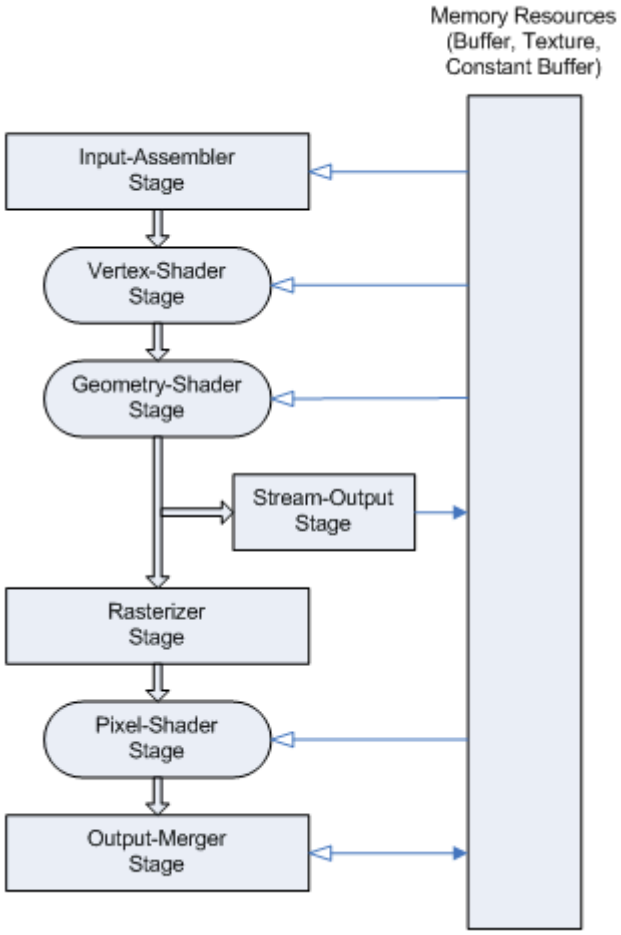
Il processo viene ripetuto per ogni vertice e quindi per ogni pixel dei triangoli generati dai vertici. Ogni vertice non vede gli altri ne può interagirci, stessa cosa per i pixel.
Questo significa che il Vertex ed il pixel shader devono essere regole uniche per tutti dato che l'unica differenza sarà il contenuto dei vertici che si scaricherà sul contenuto dei pixel generati.

Questo è stato fatto principalmente per parallelizzare il calcolo visto che le schede video elaborano centinaia di vertici e pixel contemporaneamente per ogni ciclo del processore grafico. Il rendering viene fatto su superfici che possono essere il backbuffer (quello che viene mandato a video) o su altre texture che verranno usate in altri cicli di rendering.

DirectX10

La struttura di DirectX10 è più o meno simile a quella di DirectX9. Le differenze (minime a livello logico ma grandi per funzionamento) sono le seguenti:

1. Le costanti sono organizzate in strutture di buffer e non in semplici registri. Questo permette di spostare interi settori di memoria o di utilizzare una texture come matrice di dati.
2. Il geometry processing include il Geometry Shader, una zona del processore programmabile che permette di gestire i triangoli che vengono creati dopo il Vertex Shader. Il risultato è che per ogni triangolo è possibile modificarne i vertici, eliminarli o creare nuovi triangoli. Il tutto nella scheda video con prestazioni elevate.
3. Il geometry processing è in grado di salvare ciò che è stato elaborato fino a quel punto in modo da usarlo in seguito e risparmiare calcoli nelle fasi successive (addirittura per far generare intere strutture geometriche alla scheda video invece di caricarle da file o farle generare dal codice dell'applicazione).



HLSL

La scheda video è di fatto un processore con banchi di memoria su di esso. Il processore (la GPU) è tuttavia ottimizzata per eseguire calcoli vettoriali in parallelo. Nella computer grafica 3D il tipo più comune è il vettore. La posizione è espressa come vettore XYZ, come anche le direzioni; anche i colori sono espressi come valori RGBA e non dimentichiamo che nella matematica 3D si usano le matrici per trasformare le posizioni che sono di fatto vettori di vettori.
Le GPU lavorano nativamente con i vettori e questo è già uno dei motivi per cui rispetto alle CPU riescono ad elaborare in modo così veloce la grafica.
Dalla versione DirectX8 le GPU hanno iniziato ad offrire uno spazio programmabile (prima era tutto fisso e solo i driver avevano il controllo della scheda).
Nascono così gli shader, codice che invece di essere elaborato dal processore è gestito dalla scheda video ma che a differenza di linguaggi come C o .Net sono compilati e caricati mentre il programma sta girando (quindi più simili a codice script).
Visto a basso livello si tratta di poter gestire le zone di memoria del processore, i cosiddetti registri. Attraverso il codice accediamo ai vari registri della GPU, eseguiamo delle operazioni e mandiamo i risultati in altri registri. La scheda video infatti fa questo: DirectX riempie i registri di input contenenti ad esempio i vertici, esegue il nostro codice e poi legge il contenuto dei registri di output che saranno quelli mandati a video. Ecco un esempio di codice assembler

```
vs.1.0
dcl_position v0
dcl_normal v3
dp4 r0.x, v0, c0
dp4 r0.y, v0, c1
dp4 r0.z, v0, c2
dp4 r0.w, v0, c3
dp3 r1.x, v3, c20
max r1, r1.x, c4.x
mul r0, r1.x, c21
add r0, r0, c22
min oD0, r0, c5.x
mov oPos, r0
```

Le prime 3 righe impostano il processore settando la versione shader da utilizzare (1.0) ed i registri (v0 conterrà la posizione mentre v3 la normale).
Le successive 10 righe sono calcoli in cui vengono eseguite operazioni; ad esempio dp3 r1,v3,c20 significa eseguire il prodotto fattoriale di vettori tra v3 (vettore contenente la normale) ed il registro costante c20 (un dato, per esempio una direzione della luce). Il risultato è inserito nel registro r0. Le ultime 2 righe mostrano che le operazioni scrivono su registri con prefisso ‘o’ (oD0 è il colore, oPos la posizione). Lettura input, esecuzione e scrittura, questo è un codice shader. I registri del processore sono di 4 tipi:

- Registri R, temporanei usati per i calcoli
- Costanti C, che gli passiamo da codice e contengono matrici, direzioni e valori arbitrari
- Input V, in cui la scheda inserisce gli input quali i dati in arrivo dal vertex data o dai vari shader
- Registri O, in cui noi scriveremo i valori da restituire alla scheda.

Fortunatamente dalla versione 9 di DirectX non dobbiamo più scrivere in assembler il codice degli shader, ma in un comodo linguaggio C-like, l’HLSL (High Level Shading Language).
Tenete però sempre presente il linguaggio assembler, perché bisognerebbe sempre ottimizzare in base a quello. La funzione Disassembler permette di ottenere il codice assembler compilato a partire dall’HLSL. Già osservare tra due versioni di shader quella che produce meno righe di codice

assembler è un primo modo di valutare la complessità dello stesso e capire quando si sta andando oltre i limiti della versione shader scelta.
Esistono infatti 4 versioni di shader, dalla 1 alla 4. A queste si aggiungono una serie di sotto versioni per il vertex e per il pixel. Queste cambiano per istruzioni utilizzabili e per lunghezza del codice Assembler.

Versione	Limiti Vertex Shader	Limiti Pixel Shader
1.1 1.2 1.3 1.4	128 istruzioni	8 istruzioni 12 istruzioni 14 istruzioni 28 istruzioni
2.0 2.0x	256 istruzioni	96 istruzioni > 96 istruzioni
3.0	> 512 istruzioni	> 512 istruzioni
4.0	> 65536	> 65536 istruzioni

Sarà quindi importante scrivere codice che generi meno istruzioni assembler possibili. Fortutanamente il compilatore applicherà numerose forme di ottimizzazione.

In questo articolo non rispiegherò la sintassi HLSL (ci sono già i tutorial sul sito), ma come imparare a ragionare con esso.

Nozioni Matematiche

Vettori
Un vettore è una tupla di valori. Ad esempio i valori 3,4,5 sono un vettore di numeri. I vettori utilizzati in DirectX sono in genere tuple da 2,3 o 4 valori di tipo float. Matematicamente i vettori rappresentano diversi tipi di informazioni. Un esempio è una coordinata sul piano cartesiano; il vettore V (3,4) rappresenta il punto di coordinate 3 sull'asse X e 4 sull'asse Y.
Un vettore è dotato di diversi tipi di operazioni. In HLSL è possibile eseguire le 4 operazioni tra vettori di pari dimensione. Somma, differenza, prodotto e divisione avvengono elemento per elemento (es

A (a,b) + B(c,d) = C(a+c,b+d)

Quando si applicano le operazioni tra un vettore ed uno scalare (un numero, paragonabile ad un vettore di dimensione unitaria) allora ogni valore del vettore viene ad esempio moltiplicato per lo scalare (es

A(a,b) * c = B (a*c,b*c)

Un vettore può rappresentare anche una direzione. Il vettore A(3,4) può indicare che bisogna spostarsi di 3 sull'asse X e 4 sull'Y. Lo spostamento è indicato dalla lunghezza del vettore calcolato con Pitagora in cui il vettore direzione rappresenta l'ipotenusa del triangolo rettangolo. Per A avremmo

$$L = \sqrt{3^2 + 4^2} = 5$$

Una direzione è calcolata come differenza tra 2 punti. Di solito però si preferisce la rappresentazione normalizzata. Un vettore V avente lunghezza L è normalizzato dividendo V per L
A (3,4)
L = 5
N = (3/5, 4/5).
Un vettore normalizzato ha lunghezza 1 ed è fondamentale utilizzare sempre vettori normalizzati per le direzioni nei calcoli usati negli shader.

Shader

Vertex Shader

Il vertex shader è il primo step con cui possiamo interagire. Il vertex shader è una funzione in codice C-like (ma senza puntatori) che prende in ingresso un vertice restituendo in uscita una nuova struttura contenente i dati che arriveranno al pixel shader.

```
struct VS_OUT
{
	float4 oPos:SV_POSITION;
	float3 oNormal:MYNORMAL;
};
Float4x4 transform;
Float4x4 world;
VS_OUT VS(float4 pos:POSITION,float3 normal:NORMAL){
	VS_OUT outP;
	outP.oPos= mul(pos,transform);
	outP.oNormal=mul(float4(normal,1),world);
	return outP;
}
```

Ecco un esempio di vertex shader. In input c'è un float4 che è la posizione ed un float3 che è la normale. Vengono moltiplicati per la matrice di trasformazione transform e quella del mondo world restituendo il risultato. L'output è a sua volta un vettore che deve avere per forza una posizione e poi dati arbitrari.
Un elemento è definito tramite tipo, nome e semantica.
Il tipo indica la dimensione dei dati (float4 significa vettore da 4 float), il nome è quello che vogliamo usare nel vertex shader, mentre la semantica è un nome usato come scambio tra gli shader.
L'output generato infatti viene passato al vertex shader che usando la stessa dimensione e semantica può riprendere i dati. La differenza tra DirectX10 e DirectX9 è essenzialmente per la semantica.
In DirectX9 infatti le semantiche erano fissate (POSITION, TEXCOORD0, COLOR0), mentre in DirectX10 le semantiche possono essere qualsiasi; esistono comunque delle semantiche obbligatorie riconoscibili tramite suffisso SV_. Un esempio è la posizione SV_POSITION che è un dato obbligatorio sia in input che in output per il vertex shader.

Il primo dato di input, obbligatorio, è la posizione.
Il modello 3D che viene creato dal grafico tramite editor come 3D Studio Max o Maya è un file dati che contiene le informazioni per la ricostruzione del modello nella posizione in cui è stato disegnato. Quindi se abbiamo il modello del cubo questo si troverà ad esempio centrato nell'origine.

Non è però questa la posizione che vogliamo per l'oggetto, quindi occorrerà trasformarlo in modo che si trovi nella posizione finale.
Questo processo si chiama in computer grafica "**trasformazione affine**".
La trasformazione avviene per mezzo di prodotti vettore per matrice. Spieghiamo perché.
Prendiamo un oggetto a 2 dimensioni (quindi coordinate XY). Banalmente per traslare un punto basta sommare ad XY due valori.
Es)
P1: (4,6) a destra di 3 ed in alto di 2 diventa
P2: = P1 + (3,2) = (4+3,6+2)= (7,8).

Se invece vogliamo ridurne le dimensioni delle coordinate di metà
P2= P1/2 = (4/2,6/2)=(2,3)

Due trasformazioni diverse fatte con metodi diversi. I matematici hanno quindi utilizzato la trasformazione matriciale ossia moltiplicando il punto impostato come vettore XYW per una matrice opportunamente preparata. La W è la coordinata omogenea di valore pari ad 1 utilizzata per poter ottenere le traslazioni come prodotto di un vettore per una matrice. Per questo se utilizza grafica 2D occorrono vettori di dimensioni pari a 3 e matrici di dimensione 3x3. In 3D si utilizzeranno vettori XYZW con matrici 4x4. I vettori utilizzati per le trasformazioni devono essere sempre omogeneizzati dividendo per W tutti i valori. Le matrici vengono create utilizzando formule fisse.

Ecco le due operazioni espresse in matrici

$$\begin{matrix} & & & & 1 & 0 & 0 & & & \\ 4 & 6 & 1 & * & 0 & 1 & 0 & = & 7 & 8 & 1 \\ & & & & 3 & 2 & 1 & & & \end{matrix}$$

Traslazione

$$\begin{matrix} & & & & 0.5 & 0 & 0 & & & \\ 4 & 6 & 1 & * & 0 & 0.5 & 0 & = & 2 & 3 & 1 \\ & & & & 0 & 0 & 1 & & & \end{matrix}$$

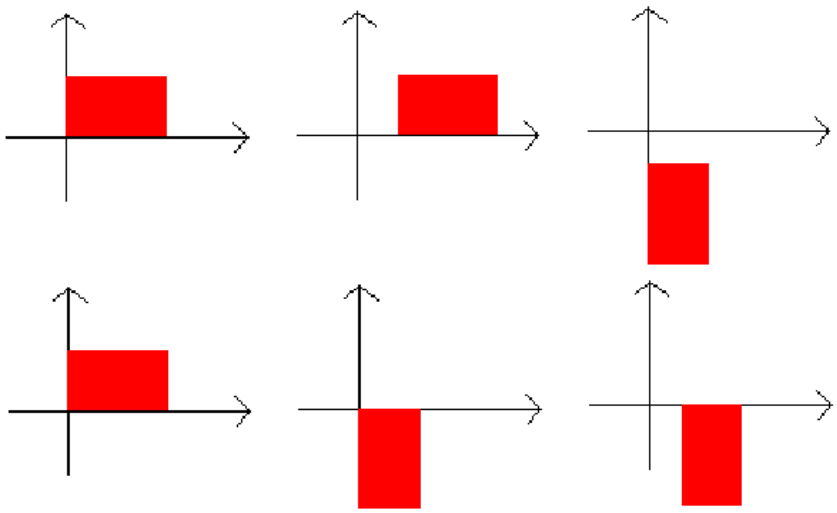
Scala

Le principali proprietà delle trasformazioni affini sono:

- che per trasformare un oggetto è sufficiente trasformare tutti i suoi vertici per una stessa matrice
- gli oggetti mantengono i parallelismi (se ruotate un cubo, rimane un cubo)
- le trasformazioni avvengono sempre rispetto al centro dello spazio cartesiano
- per applicare n trasformazioni in sequenza è sufficiente trasformare i vertici per il prodotto delle singole matrici.

Quest'ultima è una proprietà fondamentale. Se abbiamo un cubo che vogliamo prima ruotare poi e poi traslare creeremo 2 matrici, una rotazione ed una traslazione, le moltiplicheremo tra loro e poi le applicheremo. Ricordo che il prodotto tra matrici non è commutativo, quindi data R la matrice di rotazione e T la matrice di traslazione avremo che R * T != T * R.
Questo è logico anche geometricamente. Se le trasformazioni avvengono sempre rispetto al centro avremo 2 movimenti diversi. Nell'immagine sotto un esempio. I 3 piani cartesiani in alto

rappresentano il caso T * R, i 3 sotto il caso R * T. Il risultato è diverso perché nel primo caso allontanando l'oggetto dal centro tramite traslazione fa in modo che la rotazione avvenga lungo un raggio più ampio.



DirectX fornisce le funzioni per creare le matrici base (scala, rotazioni sugli assi, traslazioni). Ragionate sempre per somma di traslazioni, così riuscirete sempre ad ottenere i movimenti desiderati.

In HLSL la funzione da usare è mul.
Float4 Input
Float4 Output
Float4x4 Matrix

Output= mul (Input, Matrix)

Spazi
Uno dei concetti da capire bene quando si lavora con gli shader è quello di Spazio inteso come riferimento delle coordinate.
Un oggetto che viene importato in DirectX è definito come Object Space. Il centro dello spazio cartesiano coincide con le coordinate 0,0,0 dell'oggetto.
Se vogliamo che questo oggetto si trovi in una posizione XYZ allora dobbiamo moltiplicare i vertici per una matrice di traslazione. I vertici avranno quindi nuovi valori ed il centro 0,0,0 sarà quindi in una posizione diversa rispetto all'oggetto.
Si dice quindi che ora l'oggetto si trova in World Space.
L'importanza dei cambi di spazio si vede quando si iniziano ad utilizzare diversi vettori. Facciamo un esempio. La luminosità di un triangolo colpito dalla luce si calcola tramite il prodotto fattoriale tra la normale al triangolo (la direzione in cui guarda) e la direzione della luce.
L'oggetto inizialmente si trova in Object Space e quindi non nella posizione che dovrà occupare nella scena. La normale non è quindi orientata correttamente e se si calcola la luce con la direzione si avrà un risultato sbagliato. Trasformando oltre alla posizione anche la normale avremo la normale in world space e quindi corretta per effettuare il calcolo.

Non sarebbe però stato sbagliato fare il contrario, ossia trasformare la direzione della luce dal world space all'object space moltiplicandola per l'inversa della matrice. Computazionalmente è anche conveniente: la luce è una e quindi basta un solo prodotto per la matrice inversa mentre i vertici sono tanti.

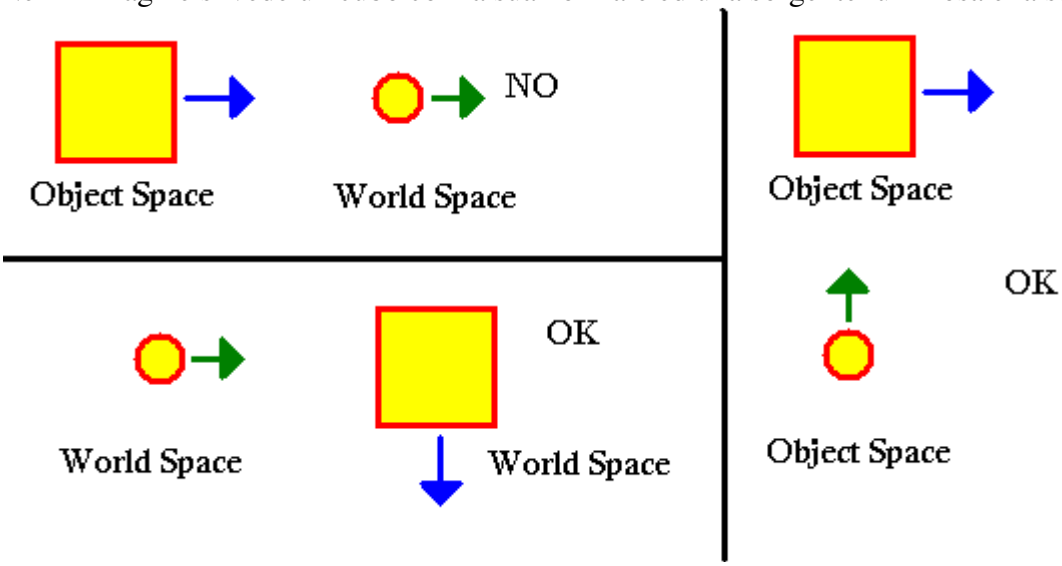
La regola quindi è che quando fate operazioni tra punti e direzioni assicuratevi che si trovino nello stesso spazio.

Gli spazi più usati in DirectX sono oltre all'object space ed il world space:

- View Space: è lo spazio nel quale gli oggetti sono orientati secondo la telecamera che si trova al centro dello schermo nel punto più esterno, virtualmente sul vetro del monitor
- Projection Space: tutte le coordinate sono riferite al tipo di proiezione. Tutto ciò che si trova tra le coordinate 1,1,1 e -1,-1,0 sono visibili sullo schermo. Tutto ciò che è fuori non compare.
- Texture Space: il piano del triangolo coincide con il piano dello schermo. Utilizzato per gli effetti di normal map. Si ottiene moltiplicando il vettore che ci interessa per la matrice che ha come righe i vettori Tangente, Binormale e Normale del vertice.

Facciamo un esempio:

Nell'immagine si vede un cubo con la sua normale ed una sorgente luminosa e la sua direzione.

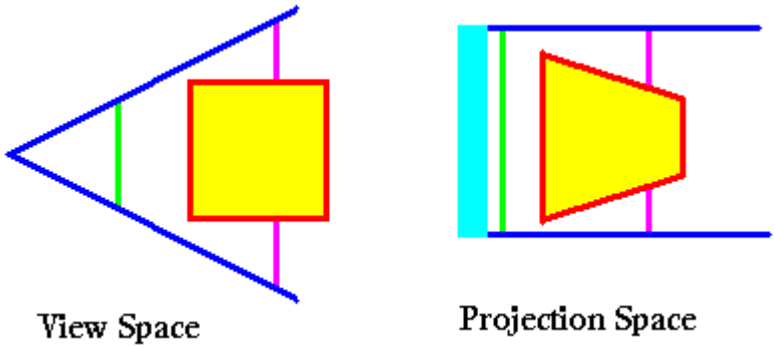


In alto a sinistra il cubo è in object space e la sua normale punta a destra (0°) mentre la sorgente si trova in world space e la sua direzione è anch'essa a destra (0°). Sapendo che l'intensità luminosa è l'angolo tra le direzioni della normale al lato e della direzione della luce abbiamo un angolo pari a 0°. Sotto invece spostiamo correttamente il cubo in world space, vediamo che l'angolo tra luce e normale è ora di 90°. Questo è corretto. Se però spostiamo la luce in object space (moltiplicando direzione e posizione per l'inversa della world matrice dell'oggetto) abbiamo la luce che punta in alto (+90°). La differenza sarà sempre di 90° e la posizione reciproca uguale. Il risultato sarà quindi il medesimo in entrambi i casi.

Camera e Proiezione

DirectX mostra ciò che esce dal vertex shader con posizione compresa tra -1,-1,0 e 1,1,1 (spazio omogeneo).

La matrice world sposta gli oggetti dalla loro posizione creata nell'editor alla posizione che vogliamo nella scena. Questo però non basterebbe in quanto solo gli oggetti compresi nell'intervallo verrebbero mostrati.
Vorremmo quindi inquadrare la scena rispetto ad un punto di vista, la telecamera. Le funzioni di DirectX creano matrici View che ruotano e traslano gli oggetti in modo che lo 0,0,0 risulti essere il punto in cui si trova la lente della telecamera. La scena però ha ancora dimensione naturale, è cambiato solo lo spazio. La view matrix permette solamente di impostare la posizione della telecamera, la direzione in cui guarda e l'orientamento lungo la direzione (variando questo potremmo ad esempio inquadrare la scena a testa in giù). Si utilizza quindi la projection matrix. Questa matrice non fa altro che portare i vertici che si trovano entro un certo angolo nella zona valida di DirectX. Ci sono 2 tipi di proiezione, quella prospettica e quella ortogonale. La prima permette di formare una piramide che parte dal punto 0,0,0 (la telecamera) fino al massimo che vogliamo vedere (il parametro far nella matrice). Su di essa tracciamo dei piani perpendicolari alla base (colori verde e viola). Applicando la proiezione vedremo che i piani vicino al vertice si allargheranno di più rispetto a quelli distanti. Ognuno di quei piani sarà adattato al quadrato unitario. Si creerà così l'effetto prospettico dove ciò che è vicino è più grande di ciò che è lontano.



Il lato azzurro rappresenta il monitor e vedete che il cubo (immaginatelo come stanza) ha il lato vicino alla camera più grande rispetto al lato distante.
Per la proiezione ortogonale invece semplicemente si prenderà un parallelogramma dalla scena e si ridurranno le coordinate allo spazio omogeneo.

Le tre matrici World, View e Projection devono essere moltiplicate per il vertice nello shader per avere l'oggetto correttamente sullo schermo. Per ottimizzare si usa la matrice di trasformazione che è prodotto delle 3.

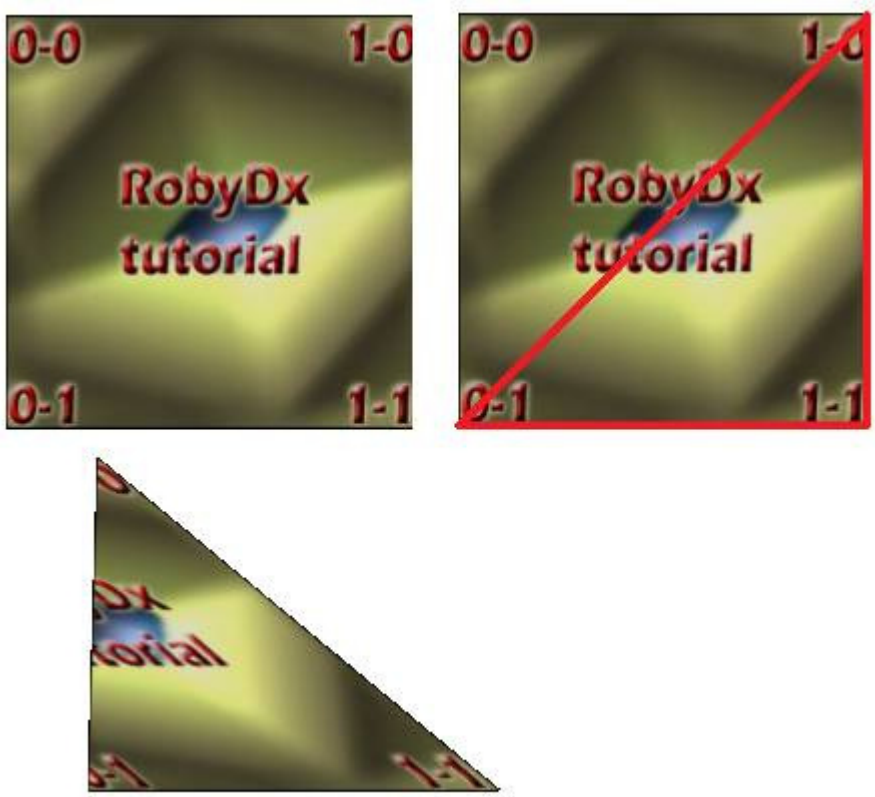
Direzioni e Luci

Esistono numerose formule per il calcolo della luce in computer grafica, la maggior parte di esse comunque usano come base la normale al triangolo (o al vertice) e la direzione della luce.
L'intensità è l'angolo tra i vettori normale e direzione della luce che in geometria si può calcolare come prodotto fattoriale.
Data la normal N (nx,ny,nz) e la direzione D (dx,dy,dz) il coseno dell'angolo è
Cos(angolo) = nx*dx + ny*dy + nz* dz. Questa operazione si chiama prodotto fattoriale (in inglese Dot Product).
Questo valore che varia tra -1 e +1 (ripassate trigonometria se non lo sapete) è anche pari all'intensità luminosa nella formula di illuminazione di Lambert.

Lambert = (N dot L) * coloreMateriale * coloreLuce + luceAmbientale

In HLSL per il prodotto fattoriale si usa l'istruzione dot. La direzione della normale è quella che arriva dalla mesh, quella della luce dipende dal tipo. Per la luce direzionale la direzione è data come parametro. Per quella puntiforme invece la direzione è la differenza normalizzata tra il vertice e la posizione della luce.
Tutte le formule di illuminazione utilizzano come base quella che è la formula di Lambert. Il prodotto fattoriale è utile comunque ogni qual volta serva calcolare l'angolo tra due vettori.

Coordinate Texture
Le coordinate texture indicano il punto della texture che verrà preso in quel vertice. Le coordinate sono espresse con valori da 0 ad 1 (una texture ha come coordinate 0,0 come angolo alto-sinistro e 1,1 come angolo basso-destro)



Nell'immagine si vede come un triangolo abbia i 3 vertici come
A=1,0
B= 0,1
C= 1,1.
Il triangolo avrà i 3 punti scelti nei vertici mentre l'interno conterrà l'area scelta. Le coordinate texture sono indipendenti dalla forma che il triangolo avrà. Immaginate sempre una texture come qualcosa di elastico: ne tagliate un triangolo e deformandolo lo applicate al triangolo. Questo rientra sempre nel discorso dell'interpolazione lineare. Le coordinate A,B e C sono quelle negli angoli: nei pixel centrali la coordinata sarà la media tra questi. Si vede che il punto centrale sul lato tra A e B ha coordinate 0.5-0.5, pari alla media tra A e B. Così come il punto centrale nel triangolo avrà coordinate 0.75, 0.75 pari alla media esatta tra i 3 punti.
Le coordinate texture vengono calcolate nel vertex shader (quasi sempre arrivano in input e si rimandano in output) ma spesso sono utilizzate per altri fini. In directX9 infatti si è soliti passare per mezzo delle coordinate texture i valori più disparati (colori, direzioni, valori generici). DirectX10 non esiste invece un vettore specifico per le coordinate texture in quanto ad ogni vettore si dà un nome che arriva al pixel shader.

Pixel Shader

Il pixel shader è il codice che viene eseguito dopo il rastering. Come detto più volte sopra, il pixel shader ha in ingresso la stessa struttura restituita dal vertex shader (esclusa la posizione del vertice). I valori però sono una media dei 3 valori corrispondenti dei vertici che includono il pixel.
Se ad esempio dal vertex shader abbiamo restituito la seguente struttura

```
Struct VS_OUT
{
        Float4 Position:SV_POSITION;
        Float3 Colors:MYCOLOR;
        Float Index:MYINDEX;
}
```

Se i risultati sono i seguenti

	Vertice A	Vertice C	Vertice C
SV_POSITION	0,0,0	100,50,40	200,100,40
MYCOLOR	90,0,0	0,120,0	0,0,180
MYINDEX	1	6	2

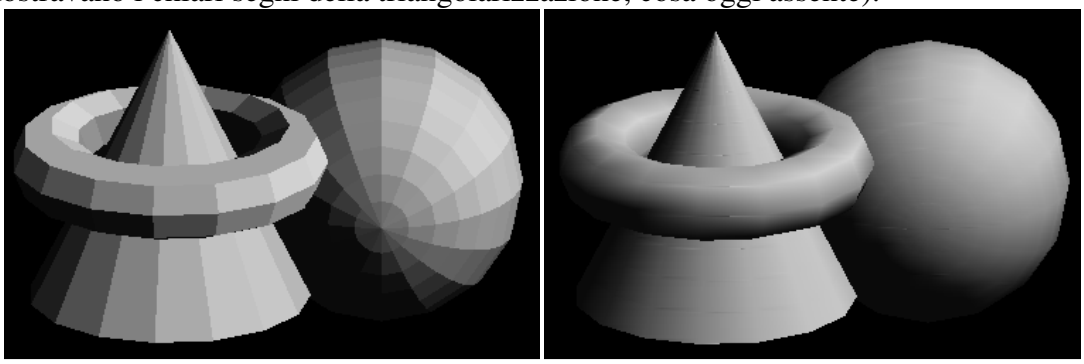
Il pixel shader centrale avrà i seguenti valori:
SV_POSITION= 100,50,30
MYCOLOR = 30,40,60
MYINDEX = 3

La prima cosa che si nota quindi è l'impossibilità di avere un valore esatto (salvo inserirlo uguale nei 3 vertici). I calcoli devono quindi essere fatti in linea generale con i valori che possono arrivare.
Il pixel shader ha in uscita solitamente un vettore corrispondente ai colori RGBA. Questo sarà il valore finale in uscita che, salvo utilizzo di alphablending, sarà il colore che vedrete a video.
Il pixel shader viene eseguito una volta per ogni pixel, un numero di solito nettamente superiore al vertex shader. Bisogna quindi cercare di non sovraccaricare il codice per non rallentare l'applicazione.
Rimane comunque il fatto che questo è lo shader più importante perché questo è il punto migliore per posizionare algoritmi di calcolo della luce e della texture.

Per Pixel vs Per Vertices
La formula di interpolazione lineare è la seguente:
$F= A * I + (1-I) * B$

Dove F è il valore finale, A e B i due valori da interpolare. Il valore I è l'indice di interpolazione tra 0 ed 1. Si vede che il valore finale F passa da essere A per I = 1 a B per I = 0 in modo continuo e lineare. Ciò che fa DirectX, nel passaggio di valori tra il vertex ed il pixel shader, è questo tipo di operazione, anche se applicata a 3 valori e corretta tramite l'uso di diversi filtri.
Il fenomeno trova una grande utilità in fase di ottimizzazione. Ci sono infatti operazioni lineari come ad esempio somma e moltiplicazione.
L'interpolazione della somma di due valori è uguale alla somma dei due valori interpolati. Quando questo si verifica è opportuno spostare i calcoli nel vertex shader e far arrivare i risultati al pixel.
Un cubo ha 8 vertici, se decidiamo che il colore di ogni lato sia la somma tra un valore inserito nel vertice e 2 costanti abbiamo 2 somme. Se fatte nel vertex shader saranno solo 16 operazioni, se invece le inseriamo nel pixel shader sono 2 operazioni per ogni pixel (se il cubo occupa tutto lo

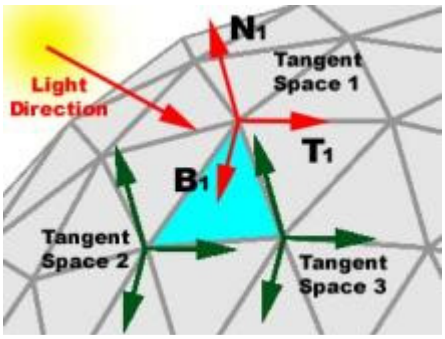
schermo siamo nell'ordine delle centinaia di migliaia). Uno spreco immenso senza avere nessuna differenza.
Altre operazioni però non sono lineari (è il caso del prodotto fattoriale, la base per le luci). Applicando ai vertici quindi avremo un risultato errato (se ricordate i vecchi giochi le superficie curve mostravano i chiari segni della triangolarizzazione, cosa oggi assente).



Se quindi le inserirete nel vertex shader avrete una netta perdita di qualità (ma che potreste preferire per un discorso di prestazioni).

Spazio Tangente

Lo spazio tangente è uno dei tanti possibili spazi su cu possono essere trasformati i vettori. Lo spazio tangente ruota i vettori rispetto alla posizione del triangolo che sarà il centro. Lo spazio tangente viene usato per le normal map. Nelle normal map infatti è il pixel shader a calcolare la luce ma lo fa utilizzando come normale il colore della normal map caricata. Il colore estratto dalla texture RGB viene convertito in XYZ semplicemente con
Float3 N= colore.rgb*2-1
La necessità è quella di ruotare la direzione della luce in modo che sia corretta rispetto alla texture che non è possibile far ruotare. La stessa normal map è identica indipendentemente da dove è posizionata nel modello e quindi la soluzione ottimale è quella di ruotare la direzione della luce o della telecamera in modo che abbia l'angolo corretto rispetto alla texture che è in posizione base.



In questa immagine è possibile vedere oltre al vettore normale (N) anche altri due vettori, quello tangente (T) che come dice la parola è tangente al triangolo e la binormale (B) anch'esso tangente ma che forma 90° con gli altri 2. I 3 vettori vengono posizionati a formare una matrice 3x3. Un vettore moltiplicato per questa matrice sarà il vettore in tangent space.

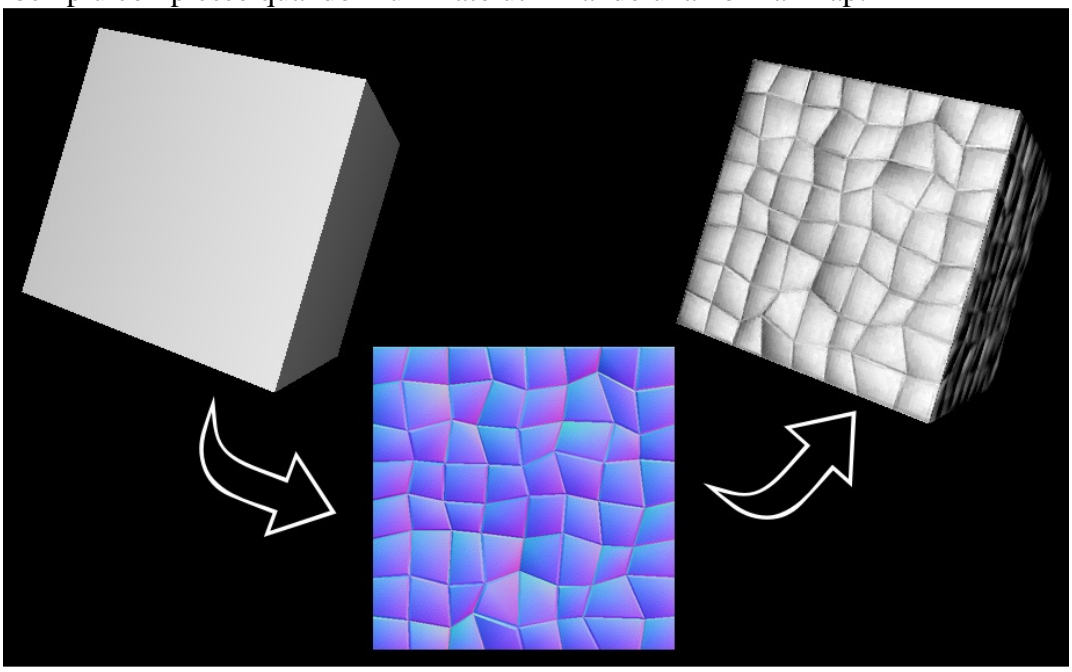
$$Lrel_x \quad Lrel_y \quad Lrel_z = L_x \quad L_y \quad L_z * \begin{matrix} Tx & Nx & Bx \\ Ty & Ny & By \\ Tz & Nz & Bz \end{matrix}$$

Per generare i due vettori tangente e binormale si usano diversi algoritmi. Il più comune è solitamente il seguente

```
Vector3D[] tan1 = new Vector3D[vertexCount * 2];
Vector3D[] tan2 = tan1 + vertexCount;

for (long a = 0; a < triangleCount; a++)
{
	long i1 = triangle.index[0];
	long i2 = triangle.index[1];
	long i3 = triangle.index[2];

	Point3D v1 = position[i1];
	Point3D v2 = position [i2];
	Point3D v3 = position [i3];

	Point2D w1 = texcoord[i1];
	Point2D w2 = texcoord[i2];
	Point2D w3 = texcoord[i3];

	float x1 = v2.x - v1.x;
	float x2 = v3.x - v1.x;
	float y1 = v2.y - v1.y;
	float y2 = v3.y - v1.y;
	float z1 = v2.z - v1.z;
	float z2 = v3.z - v1.z;

	float s1 = w2.x - w1.x;
	float s2 = w3.x - w1.x;
	float t1 = w2.y - w1.y;
	float t2 = w3.y - w1.y;

	float r = 1.0F / (s1 * t2 - s2 * t1);
	Vector3D sdir((t2 * x1 - t1 * x2) * r, (t2 * y1 - t1 * y2) * r,
		(t2 * z1 - t1 * z2) * r);
	Vector3D tdir((s1 * x2 - s2 * x1) * r, (s1 * y2 - s2 * y1) * r,
		(s1 * z2 - s2 * z1) * r);

	tan1[i1] += sdir;
	tan1[i2] += sdir;
	tan1[i3] += sdir;

	tan2[i1] += tdir;
	tan2[i2] += tdir;
	tan2[i3] += tdir;

	triangle++;
   }

   for (long a = 0; a < vertexCount; a++)
```

```
{
    Vector3D n = normal[a];
    Vector3D t = tan1[a];

    // Gram-Schmidt orthogonalize
    tangent[a] = (t - n * Dot(n, t)).Normalize();

    // Calculate handedness
    tangent[a].w = (Dot(Cross(n, t), tan2[a]) < 0.0F) ? -1.0F : 1.0F;
    binormal[a]=cross(n,tangent[a]);
}
```

In questa immagine si può vedere come un cubo (fatto di soli 12 triangoli) possa sembrare un poligono ben più complesso quando illuminato utilizzando una normal map.



Esistono vari algoritmi che migliorano ancora di più l'effetto, tra cui il parallax map ed il relief map.
Il primo sposta leggermente le coordinate texture a seconda della telecamera, in modo da dare l'effetto di parallasse (da cui il nome).
Il secondo ben più complesso, effettua un ciclo nella direzione della camera in si calcola in base alla normal map quale dovrebbe essere la vera profondità geometrica e creando dei veri e propri buchi ottici. Trovate esempi anche nella SDK di Microsoft.

Rastering

Le operazioni che il sistema compie quando i vertici vengono trasformati in triangolo sono numerose ed ampiamente personalizzabili. In DirectX10 queste opzioni vengono gestite tramite Shader. Questo non significa che siano codice gestito direttamente dalla scheda video, ma che semplicemente si possono bypassare un po' di overhead scrivendolo in HLSL dato che poi il sistema procederà ad impostare la scheda in modo più veloce.

In DirectX9 questo veniva impostato tramite il RenderState, sparito in DirectX10 e sostituito o dallo Shader o da opportune classi (esempio ID3D10RasterizerState).

- Culling

  Il culling è la prima forma di ottimizzazione che pratica DirectX. I triangoli sono entità bidimensionali immersi in spazi 3D. Questo significa che il loro spessore è 0.
  Un oggetto solido normalmente mostra di ogni triangolo soltanto un lato. Un cubo ad esempio verrà visto solamente dall'esterno, quindi i lati interni non si vedranno mai. DirectX utilizza come regola quella di non generare i triangoli che sono rivolti dalla parte opposta rispetto al loro verso. I vertici di un triangolo possono essere rivolti in senso orario o antiorario rispetto alla camera. Tramite l'opzione di culling si può decidere quale verso non rendere visibile (di default il verso è antiorario). Questa opzione è utile per poter decidere di vedere l'interno anziché l'esterno di un modello. L'opzione a None invece rende visibili entrambe le direzioni ma aumenta notevolmente il tempo necessario al rendering.

- FillMode

  Il fillMode indica il modo con cui il triangolo viene generato. Esistono 3 modalità: Solid, Wireframe e Point (quest'ultima non esiste in DirectX10).
  Solid è la modalità base in cui vengono generati tutti i punti interni al triangolo. Wireframe invece genera solo i lati di ogni triangolo creando una struttura a scheletro (wireframe significa fil di ferro). Point infine renderizza come punti solo i 3 vertici.

Testing

Il testing è l'ultima fase di ottimizzazione di DirectX. Il pixel ormai generato deve venire mandato a video. Tuttavia è possibile creare regole per il quale esso venga scartato. I test esistenti sono l'alpha test, il depth test e lo stencil test.

- Alpha Test

L'alpha test consiste nella possibilità di cancellare un pixel in base al suo valore di alpha. Impostando opportunamente il device è possibile ad esempio scartare i pixel il cui valore di alpha sia uguale, maggiore o minore di una certa soglia. Un esempio immediato è per la creazione di griglie. Passando una texture il cui valore alpha sia uguale a 0 dove ci sono i fori è possibile chiedere a DirectX di non stampare quei pixel. Questa operazione viene effettuata dopo il pixel shader, quindi il valore alpha và calcolato li.

- Depth Test

Questa è l'operazione di testing più importante e viene eseguita prima del Pixel Shader. Il depth test è alla base della gestione della profondità e si attiva utilizzando lo Z-Buffer. Ogni pixel mandato a video ha, oltre che un colore ed una posizione XY in pixel anche una profondità compresa tra 0 ed 1. Questo valore viene memorizzato sullo Z-Buffer. Quando un nuovo pixel viene mandato a video viene controllata la sua profondità con il valore già presente sullo Z-Buffer lasciato dai pixel precedenti. Se minore (quindi più vicino) il pixel viene disegnato ed il valore sostituito, altrimenti il pixel viene scartato perché sarebbe nascosto dai precedenti. Questa serie di operazioni può essere personalizzato (cambiando la regola ad esempio e mostrando ciò che è lontano, anziché ciò che è vicino). Queste operazioni sono alla base di metodi di ottimizzazione come lo Z-Only.

- Stencil Test

Lo stencil test utilizza una porzione dello Z-Buffer per effettuare un test simile ma più elaborato rispetto al depth test. Tramite lo stencil si può infatti non solo decidere quando un pixel viene scritto

o no, ma anche come viene scritto lo Z-Buffer (o meglio una sua porzione chiamata Stencil-Buffer). In questo modo è possibile utilizzare un poligono affinché valorizzi questa zona in modo da utilizzarla come maschera (ad esempio per tagliare la scena creando un buco al suo interno).

Blending

Il blending è una serie di operazione che DirectX effettua quando ha tracciato un pixel. Normalmente DirectX non è in grado di conoscere lo stato di alcun pixel sullo schermo durante l'elaborazione. L'unica eccezione avviene durante la fase di blending che è l'ultima ad essere eseguita. Dopo ogni esecuzione del pixel shader il nuovo pixel di colore sovrascrive quello attualmente sul backbuffer. Se si abilita l'alphablending (o semplicemente blending in DirectX10) il colore viene confrontato con quello attuale e si può utilizzare una operazione di fusione (blending significa mescolamento) tra il nuovo pixel ed il vecchio. La formula è questa:

Final Color = ObjectColor * SourceBlendFactor + PixelColor * DestinationBlendFactor

Dove ObjectColor è il colore dell'oggetto, PixelColor quello del backbuffer in quel punto (quindi ciò che è già stato renderizzato) mentre SourceBlendFactor e DestinationBlendFactor sono le impostazioni dell'alphablending.

Source e Destination Blend indicano il valore che viene usato come interpolazione. Prendiamo l'esempio in cui come SourceBlend si usi il valore alpha del colore del nuovo pixel, mentre per DestinationBlend si usa l'inverso (1 – alpha).
Il valore finale sarà

Final Color = ObjectColor * (alpha) + PixelColor * (1 – alpha)
Questa è la classica formula di interpolazione lineare, il colore varia tra i 2 estremi in base al valore di alpha. Questo è un esempio di impostazione per poter creare il vetro.

Geometry Shader

Il geometry shader è una novità esclusiva di DirectX10 ponendosi tra il vertex ed il pixel shader. Il limite degli shader è sempre quello che ogni elemento non può vedere ne l'input ne l'output degli altri elementi. Il geometry shader invece permette di lavorare con intere primitive.
Per primitive si può intendere sia un punto, una linea o un triangolo, ma anche strutture con le adiacenze, ossia una linea con le 2 linee ad essa collegata oppure il triangolo con i 3 triangoli adiacenti. In quest'ultimo caso avremo quindi ben 6 vertici contemporaneamente così come sono usciti dal vertex shader con cui poter applicare algoritmi molto più complessi. Un esempio è il calcolo delle normali che richiede l'avere l'intero triangolo per essere calcolate.
I vertici che arrivano al geometry sono quelli che sono stati restituiti dal vertex shader. Per questo dovrete comunque ragionare sul fatto che quasi sempre nel vertex shader i vertici vengono restituiti in projected space (dopo che sono state applicate le 3 matrici). Se quindi avete necessità di lavorare con le posizioni dovrete o lasciare invariate le posizioni (ed eseguire la trasformazione alla fine) oppure usare un dato aggiuntivo in cui mettere la posizione nello spazio che più vi è consono. Di solito sarà in world space. All'output stream passerete i vertici in uscita chiudendo lo strip tramite l'istruzione RestartStrip. Se ricordate le basi nel caso si utilizzino trianglelist ogni triangolo sarà

creato partendo dai 3 vertici indicati nell'indexbuffer mentre per il trianglestrip il secondo triangolo sarà formato dagli ultimi 2 vertici del primo ed il nuovo punto.
Una cosa da ricordarsi è anche il fatto che i vertici arrivano dal vertex shader esattamente com'erano (quindi non omogeneizzati, sarà necessario dividere la posizione per w).
La funzionalità comunque più importante è quella di poter aggiungere ed eliminare triangoli direttamente nello shader. Esistono moltissimi effetti che richiedono la creazione di strutture geometriche, ad esempio le ombre volumetriche o la creazione di un prato.

Lavorare in HLSL

Il codice HLSL è a tutti gli effetti un normale linguaggio di programmazione. Tuttavia presenta un numero di limitazioni e di aspetti nascosti che spiazzano chi comincia a lavorare in questo ambiente.
Capita spesso infatti di avere dal compilatore errori inattesi che bisogna imparare a capire. HLSL infatti genererà il numero minimo possibile di istruzioni ma il grosso del lavoro dipenderà da voi.

Ragionare in Shader

La cosa che occorre capire è che ogni vertice viene processato individualmente. Gli shader usano un algoritmo unico per ogni vertice e non possiamo pensare di gestire ogni vertice in modo unico ne tentare di capire cosa avviene negli altri vertici. Ci sono comunque dei sistemi per avere delle informazioni. La struttura del vertice può prevedere anche dei nostri dati. Possiamo inserire un intero nel vertice che ne indichi il numero o a che triangolo appartenga. Addirittura possiamo aggiungere alla posizione del vertice le posizioni dei vertici adiacenti (ogni vertice avrebbe altri 2 float3 per le posizioni degli altri vertici). Questo ovviamente è solo un trucco per colmare un limite proprio della tecnologia che deve processare più vertici possibili contemporaneamente senza potersi preoccupare di aspettare che alcuni vertici siano processati per dare spazio agli altri. DirectX10 fornisce comunque già valori utili. Questi non devono essere aggiunti alla mesh ma solo alla struttura di input e di output dello shader.

- SV_ClipDistance[n]
- SV_CullDistance[n]
- SV_Depth
- SV_InstanceID
- SV_IsFrontFace
- SV_Position
- SV_PrimitiveID
- SV_RenderTargetArrayIndex
- SV_Target[n]
- SV_VertexID
- SV_ViewportArrayIndex

Ad esempio SV_VertexID restituisce l'indice del vertice che stiamo utilizzando. Maggiori dettagli su ognuno di essi li trovate nell'help.

Blocchi Istruzione

In HLSL come in C sono possibili i classici blocchi istruzioni:
If e Swich per i blocchi condizionali, For e While per quelli ciclici. Non credo serva spiegare a cosa servano in un linguaggio di programmazione if, for, while e switch. Quello che bisogna capire è però cosa avviene negli shader quando li si usa.
Prendiamo ad esempio un blocco for.

Facciamo un esempio

```
Int risultato=0;
for(int i=0;i<10;i++)
{
    Risultato+= i;
}
```

Questo può essere un modo in cui il compilatore può lavorare

```
def c0, 0, 1, 0, 0
mov r0.z, c0.x
mov r0.w, c0.x
rep i0
  add r0.z, r0.z, r0.w
  add r0.w, r0.w, c0.y
endrep
```

Il senso delle operazioni che fa il codice è che utilizza r0.w per accumulare il valore da aggiungere (prima 0, poi 1, poi 2 e così via) e r0.z accumula il valore da restituire. Per ogni ciclo il programma compie 3 operazioni: 2 Add ed un rep che incrementa i0 finché i0 non scende a 0.
Totale quindi:
3 istruzioni x 10 cicli + 2 iniziali = 32 istruzioni

Un' alternativa però può essere questa

```
def c0, 0, 1, 0, 0
mov r0.z, c0.x
mov r0.w, c0.x
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
```

```
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
add r0.z, r0.z, r0.w
add r0.w, r0.w, c0.y
```

Sapendo che il ciclo si ripete 10 volte il codice può essere esplicitato eliminando il rep e ripetendo 10 volte il codice interno.
Totale:
22 istruzioni (per altro molto più elementari rispetto alla rep)

Ben 10 istruzioni complesse in meno.
Il compilatore quindi preferirà utilizzare questo sistema. In realtà il compilatore è ancora più intelligente ed il codice effettivo compilato sarà questo

```
def c0, 45, 0, 0, 0
mov oC0, c0.x
```

Una sola istruzione. Come potete vedere quindi il compilatore di DirectX è straordinariamente capace riuscendo a trovare quasi sempre strategie in grado di limitare l'errore umano.
Attenzione però. Se il valore 10 venisse sostituito da una variabile invece che da una costante si potrebbe tornare al primo risultato.
Quando si utilizzano i blocchi quindi bisogna cercare di limitare al massimo il codice al loro interno e nel caso di cicli limitare l'uso di iteratori di lunghezza variabile. Ad esempio nel caso abbiate realizzato uno shader che può gestire 4 luci potrebbe essere preferibile disattivare le luci settando i valori a 0 piuttosto che creare un ciclo for variabile da 0 a 3.
Queste ottimizzazioni sono limitate solo agli shader dal 3.0 in poi. Questo perché le versioni shader 1 e 2 non hanno la gestione dei blocchi di codice rendendoli sempre come espliciti e generando errore quando si cerca di creare un ciclo variabile. In DirectX10 è possibile definire il comportamento usando i prefissi unroll e loop per dire al compilatore cosa preferire.
La stessa cosa anche per le istruzioni condizionali if e switch.

Funzioni

Come in C anche in HLSL è possibile definire funzioni esterne alla funzione principale. Le funzioni possono restituire qualsiasi valore ed utilizzare parametri anche come riferimento o in output (in, out, inout). Anche in questo caso le funzioni vengono compilate sia come porzioni di codice separato che come codice inline.
Il primo caso (disponibile solo con gli shader 3.0 e superiori) crea linee di codice e funzioni di salto. Questo significa che il codice salterà tramite goto da un punto all'altro.
Nel secondo invece il codice verrà inserito direttamente nel punto in cui la funzione viene chiamata. Inserire il codice direttamente nel punto migliora notevolmente la velocità ma rende il codice più lungo nel caso la funzione sia chiamata in più punti. Anche in questo caso sarà il compilatore a valutare a meno di usare prefissi come inline.

Lookup table

Oltre a quello di realizzare i nostri algoritmi in modo ottimizzato ci sono molti trucchi per diminuire il numero di istruzioni.
L'obiettivo è sempre uno: diminuire le istruzioni assembler e preferire quelle più veloci. Se il primo è un concetto universale, il secondo è qualcosa che spesso dipende dall'hardware. Ci sono infatti schede che sono più veloci a leggere le texture, altri ad eseguire codice aritmetico.
DirectX10 ad esempio ha introdotto i TextureBuffer per contenere costanti che vanno lette in modo casuale.
Questo concetto rientra in quello più generico di lookup table map.
Una lookup table è una texture i cui valori corrispondono al risultato di un algoritmo. La maggior parte degli algoritmi di illuminazione dipendono spesso da pochi valori. Se pensiamo all'illuminazione abbiamo spesso che l'algoritmo dipende dalla normale e dalla direzione della luce. Ipotizziamo che un modello di illuminazione sia una formula di questo tipo

$$L = \sqrt{(N \cdot L) + 2}$$

Sono 3 istruzioni assembler (prodotto fattoriale, somma e radice). Tutto dipende però da 2 soli input, la direzione della normale e della luce. La luce è qualcosa di fissato per tutti i vertici, cambia solo la normale nell'esecuzione dello Shader. In questo caso è possibile creare una cubemap in cui ad ogni direzione corrisponde il valore dell'illuminazione. Leggendo questa texture avremo con una sola istruzione il valore di illuminazione. Per luci complesse il risparmio può essere veramente considerevole. Questo concetto si può applicare a moltissimi casi (esempio anche la normalizzazione di vettori che anche se vedete l'istruzione normalize in realtà si tratta sempre di 3 istruzioni assembler). La possibilità di utilizzare texture nel Vertex e nel Geometry Shader a partire dagli Shader 3.0 permette di estendere questo metodo di ottimizzazione anche al Vertex e al Geometry Shader.

Compilazione

Il compilatore di HLSL, vista la semplicità del linguaggio, è estremamente ottimizzato ed in grado di ridurre all'osso il numero delle istruzioni assembler necessarie per il codice che avete scritto.
Se disassemblate uno Shader vedrete che per ogni tipo di istruzione è in grado di trovare sempre la soluzione ideale per evitare assegnazioni inutili o istruzioni ridontanti. Ad esempio

```
Float3 D = normalize(input.normal) * L + 3; // 5 istruzioni
Float3 R = (normalize(input.normal) * V + 5; // 5 istruzioni
Float3 F = D + R; // 2 istruzioni
```

Il compilatore da solo è in grado di effettuare questi 2 passaggi

```
Float3 N = normalize(input.normal); // 3 istruzioni
Float3 F = N * L + 3 + N * V + 5; //  4 istruzioni
```

Ed infine

```
Float3 N = normalize(input.normal); //3 istruzioni
Float3 F = N * (L +  V) + 8; // 3 istruzioni
```

Il totale è passato da 12 a 6 istruzioni. Il compilatore è in grado di fare cose ben più avanzate di quelle descritte come ad esempio capire che un valore di un vettore può essere utilizzato più volte.

Questo vi permette di scrivere codice ordinato senza dover pensare troppo all'ottimizzazione. Ovviamente lui non può recuperare vostri errori logici.

Modelli di illuminazione

L'illuminazione è un fenomeno molto complesso. Ogni fotone proveniente da una sorgente luminosa attraversa l'aria e colpisce un oggetto che ne assorbe alcune frequenze. La luce quindi rimbalza e colpisce altri oggetti arrivando alla fine ai nostri occhi; questo per un numero infinito di fotoni. Un computer non può ovviamente simulare tale complessità e per questo si sono creati algoritmi che semplificano il fenomeno. Esistono decine di modelli, la maggior parte dei quali può essere usato insieme agli altri all'interno della scena e sullo stesso modello. La somma di tutte le illumin

$$\sum_{1}^{n} f(x)$$

Formula di Lambert

La formula di Lambert è la più semplice formula di illuminazione. Secondo Lambert la luce riflessa da un oggetto è il coseno dell'angolo tra la normale alla superficie e la direzione della luce. Il risultato varrà tra 1 a -1 (quindi possiamo ridurre l'interesse all'intervallo 0-1). Vediamo facilmente che il risultato sarà massimo nel caso della direzione della luce opposta alla normale (angolo uguale a 0° significa coseno uguale ad 1) e minimo per angoli di 90°.
Per calcolare l'angolo si utilizza, come già descritto, il prodotto fattoriale.
Il prodotto viene quindi moltiplicato per il colore della luce e del materiale colpito (che possono essere uniti sotto il nome di diffuse color).

$$L = \sum_{1}^{n} diffuse * (L \cdot N)$$

Formula di Phong

La formula di Phong espande quella di Lambert aggiungendo la luce speculare (quella che brilla sul metallo ad esempio).
La formula include l'utilizzo del vettore di riflessione. L'oggetto in pratica calcola come fosse uno specchio la provenienza della luce. In questo caso si utilizza la direzione della telecamera per calcolare il rimbalzo.

Il vettore di riflessione si calcola in questo modo

$$R = 2(L \cdot N)N - L$$

Mentre la formula è la seguente

$$L = \sum_{1}^{n} diffuse(L \cdot V) + specular(R \cdot V)^{Sharpness}$$

Come è possibile vedere il modello Phong aggiunge alla formula di Lambert la luce speculare in cui R è il vettore riflesso e V la direzione della camera.

Luce Direzionale, Puntiforme e Spot

Nelle formule precedentemente descritte è stata utilizzata una luce direzionale. Questa simula la luce proveniente da grandi distanze che non si attenua ne si disperde. Il Sole è una di queste fonti. Altre due forme di illuminazione è quella puntiforme e spot. La prima rappresenta una sorgente proveniente da un punto e che si espande in tutte le direzioni (esempio una lampadina). La direzione in ogni punto non è quindi quella della luce, ma quella che dal punto arriva alla sorgente. La formula di Lambert diventa così:

$$L = \sum_{1}^{n} diffuse * (norm(L - P) \cdot N)$$

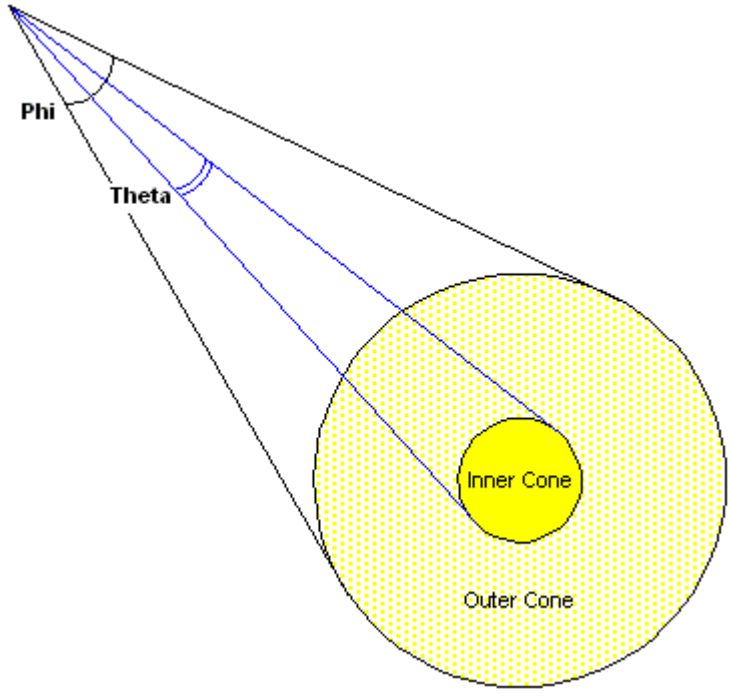
La direzione della luce è ora L (intesa stavolta come posizione della luce) e la posizione del punto, il tutto normalizzato. Uno dei fenomeni tipici delle luce puntiformi è l'attenuazione. Questo significa che con l'aumentare della distanza la luce si affievolisce. E' possibile calcolare l'attenuazione in molti modi; il più comune è moltiplicare per un valore compreso tra 0 ed 1 che varia in base alla distanza dalla sorgente. Ad esempio

$$atten = \frac{1}{ad^2 + bd + c}$$

La formula definitiva di Lambert diventa

$$L = \sum_{1}^{n} atten * diffuse * (norm(L - P) \cdot N)$$

Lo spot lightning rappresenta invece una sorgente direzionale che parte da un punto e produce un cono di luce.



In questo caso occorre sempre calcolare l'attenuazione del cono di luce in modo che sia massima nel cono interno e si affievolisca ai lati. Un esempio è quello usato come formula di illuminazione di base in DirectX fino alla versione 9.

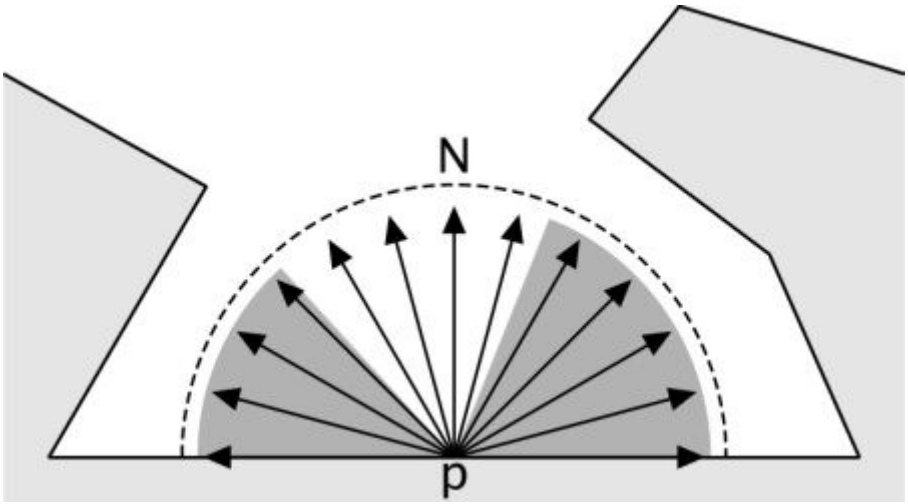
$$I = \frac{\cos alpha - \cos(phi/2)}{\cos(theta/2) - \cos(phi/2)}^{fallout}$$

Dove alpha è l'angolo tra la direzione della luce e la normale, phi e theta gli angoli in figura e fallout la variazione tra il cono interno e quello esterno.
Questo è solo un metodo visto che possono essere realizzati metodi più precisi o al contrario più semplici.

Occlusion Map

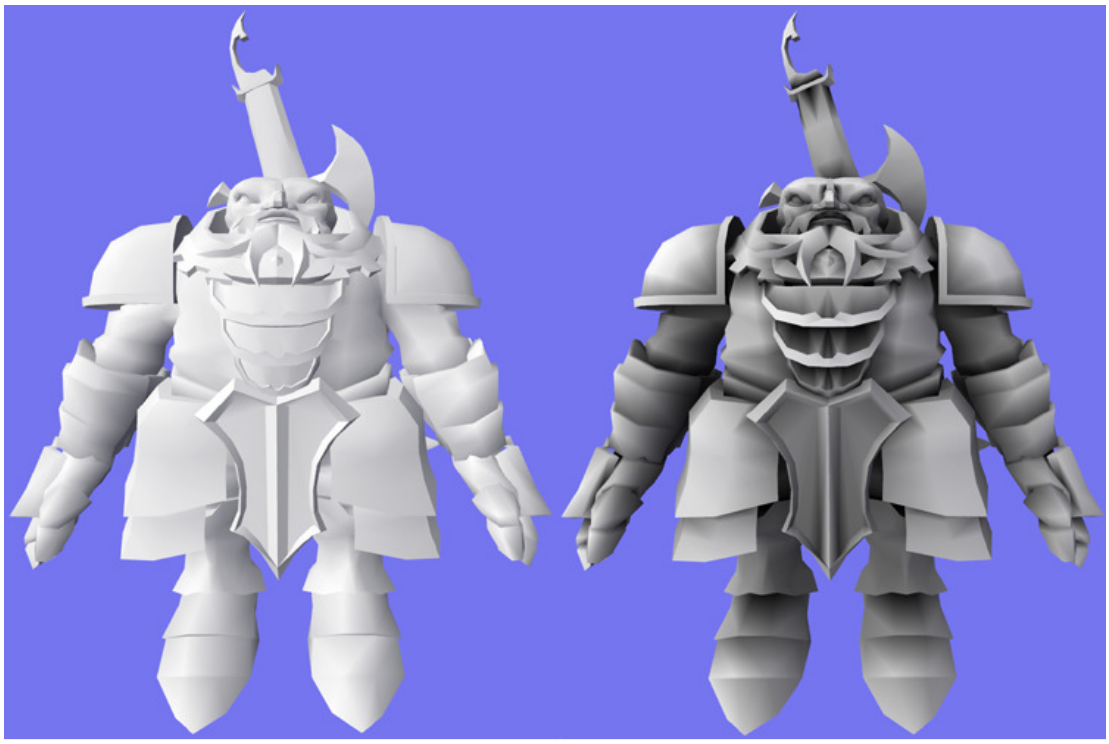
L'occlusion mapping è una tecnica che cerca di sostituire la reale gestione della luce, almeno per quanto riguarda le ombre. Attraverso l'uso di programmi come 3DS Max ad esempio è possibile generare texture che avvolte sul modello danno l'idea di ombreggiatura.

E' possibile scrivere in modo abbastanza semplice un algoritmo di generazione delle occlusion map. Basta creare un modello con una coordinata texture univoca (ogni punto di una texture corrispondente ad un unico punto sul modello) e per ogni punto calcolare i raggi che riescono ad uscire dal modello senza incontrare ostacoli.



E' sufficiente dare come valore 1 ai raggi uscenti (in cui quindi arriva la luce) e 0 a quelli che vengono bloccati. La media sarà quindi il valore di occlusione, ossia quanto viene ombreggiato quel punto. Il calcolo può essere lungo a seconda di quanto sia ottimizzato l'algoritmo o elevato il numero di raggi utilizzati.

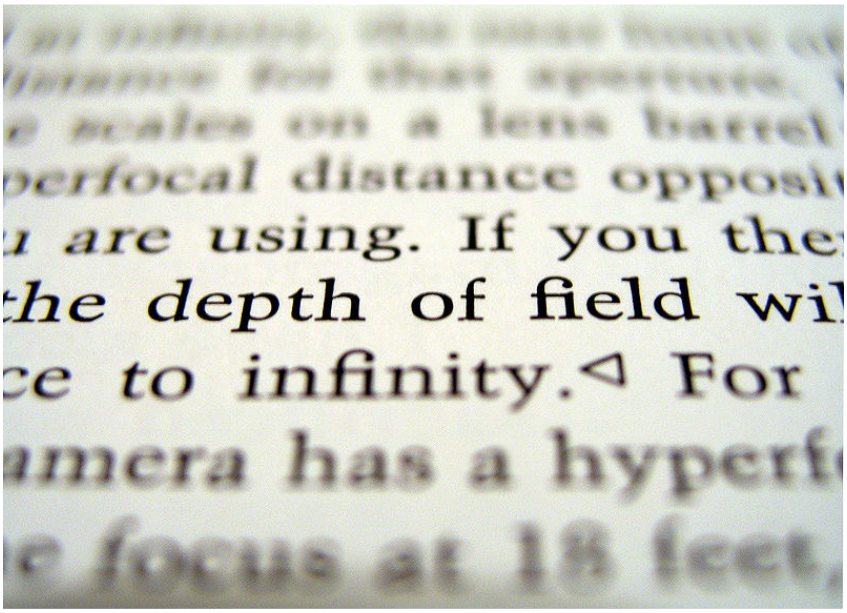
Utilizzando l'ambient occlusion insieme all'illuminazione tradizionale si può dare un forte effetto di realismo al modello.



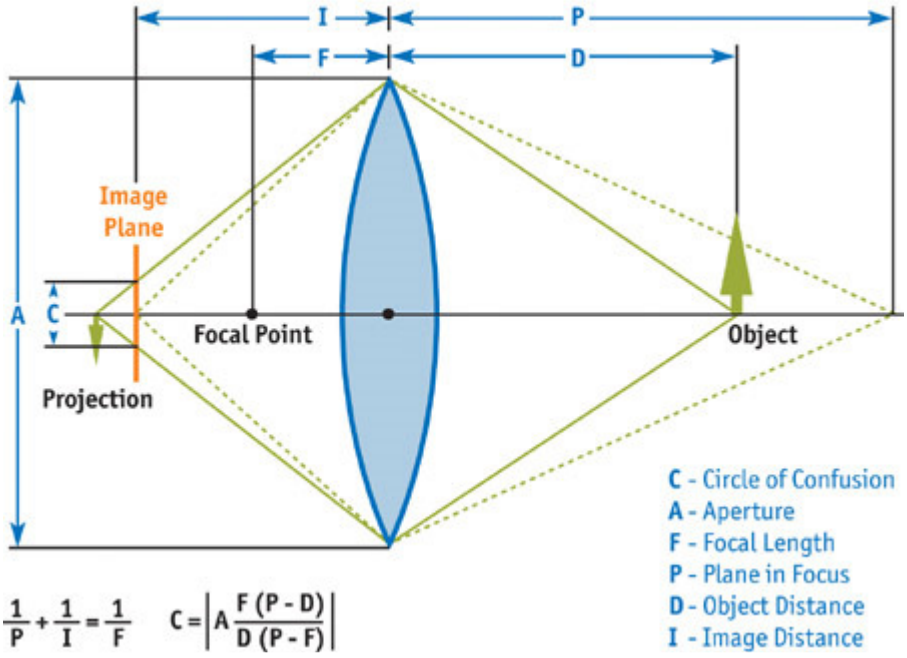
Render Target

Il processo di rendering avviene sempre su una superficie. Questa può essere il backbuffer, quello che vedete poi a video, oppure una superficie contenuta in una texture. In questo caso ciò che è stato renderizzato sarà utilizzabile nelle fasi successive. L'utilizzo più banale, utile che chiarire il funzionamento, è il rendering di un monitor che contiene un'altra scena.
Il primo rendering creerà la scena su una texture, il secondo userà questa texture sul modello del monitor. In realtà il render target trova il suo scopo negli effetti chiamati post processing. Come detto ogni vertice ed ogni pixel non hanno possibilità di vedersi mentre vengono processati. Al contrario una texture può essere letta più volte nel pixel shader in punti diversi. Se la scena si trova su una immagine è possibile prendere i colori da più punti e farci delle operazioni. Un esempio è il blur, la sfocatura, che si ottiene prendendo un' immagine e per ogni pixel calcolare la media con quelli attorno. Impossibile durante il rendering, ma salvando la scena in una texture ed applicandola ad un rettangolo che copre l'intero schermo allora diventa possibile.
L'unica cosa da tener conto è la dimensione. L'intero schermo è compreso tra le coordinate -1,1 e 1,-1. Le coordinate texture invece vanno da 0,0 ad 1,1. Per prendere esattamente un pixel dovrete quindi spostarvi di 1 diviso la dimensione della texture.

Filtri
Un filtro è una operazione che restituisce per ogni pixel una opportuna operazione tra pixel circostanti. Un classico filtro è il blur che consiste in una media tra il pixel analizzato e quelli circostanti. Solitamente si utilizza il concetto di circolo di confusione (circle of confusion), ossia una zona circolare attorno al pixel che peschi casualmente i valori attorno per farne una media. Il circolo può essere allargato a piacimento. Questo permette di creare vari effetti. Il più noto è il depth of field (profondità di campo o anche messa a fuoco). Se fate caso, guardando un oggetto vicino potrete notare che quelli distanti appaiono sfocati. L'occhio, come gli obiettivi fotografici, possono mettere a fuoco solo una porzione di spazio.



Esistono molte formule per la profondità di campo basate su formule matematiche di fisica ottica. La più semplice è questa.



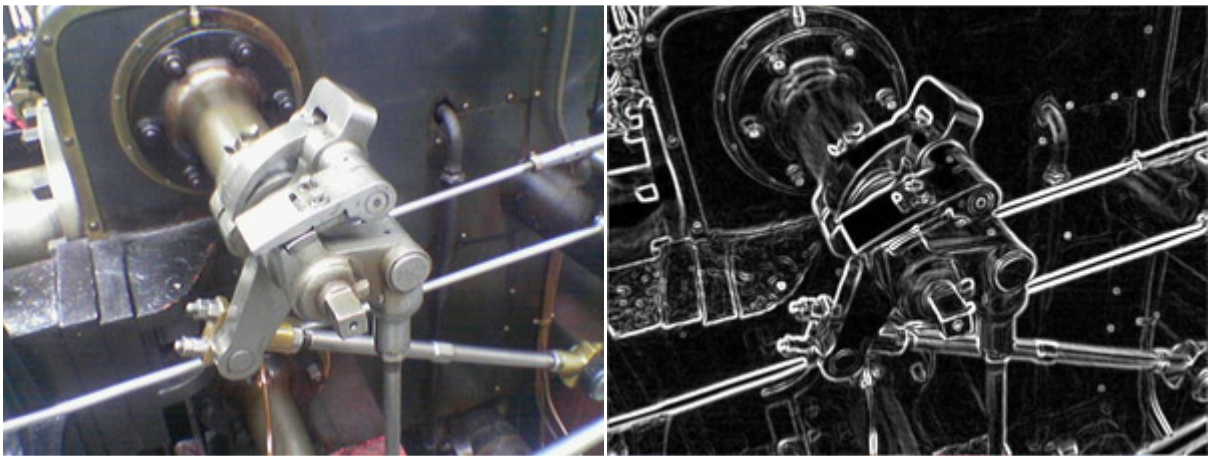
Il raggio del circolo di confusione diventa una funzione dell'apertura della lente e delle distanze tra i piani a fuoco e dell'oggetto. Questo valore sarà usato per calcolare di quanto allargare il circle of confusion ed ottenere l'effetto corretto.

Un altro esempio di filtro è il filtro sobel, un'operazione che permette di calcolare la differenza tra un pixel e quelli circostanti (gradiente). Ecco la formula

$$\mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

Semplificata si ottiene Gy e Gx come somma dei pixel attorno a quello corrente moltiplicati per il valore indicato nella matrice (ad esempio in Gy si prende il pixel in alto a sinistra e lo si moltiplica per +1).
Il valore di G finale è il gradiente. Banalmente si può rendere nero o bianco tutto ciò che supera una determinata soglia ottenendo il risultato qui sotto.



Questa tecnica viene spesso utilizzata nella tecnica del Toon Shading. Su un render target viene renderizzata la profondità della scena (due pixel a profondità molto diverse sicuramente apparterranno a contorni diversi), su un altro le normali (se c'è un cambio di direzione delle normali, il centro sarà uno spigolo). Calcolati i sobel sulle 2 e sommate si ottiene il contorno della figura

HDR
Il rendering HDR (High Dynamic Rendering) comprende tutte le tecniche per cui la luminosità degli oggetti supera i canonici 8bit per colore. Prendete una foto in cui c'è una forte sorgente luminosa, come il sole. Se con un software diminuite la luminosità vedrete tutto diventare più scuro, anche il sole. Questo non è reale. L'occhio umano infatti riesce a distinguere un contrasto di 1:1000000 anche se in realtà in un singolo istante riesce a contenere luminosità dell'ordine di 1:10000. L'occhio umano adatta l'occhio alla luce e per questo una forte luce improvvisa ci abbaglia. Proprio su questo giocano gli effetti basati sull'HDR. Entrando in una stanza buia le fonti luminose ci appaiono forti, al contrario se ci troviamo contro solo gli oggetti ci appaiono scuri.

(Far Cry della Crytek)

Le tecniche più famose basate sull'HDR sono il Bloom ed il Tone Mapping. La scena viene renderizzata su una texture con alta precisione di colore con valori di colore molto ampi. La scena viene quindi sfumata su una seconda texture prendendo solamente i valori di colore maggiori di una certa soglia. Il risultato viene unito alla texture originale e portata a video che perderà tutte le informazioni sui colori. Il risultato però sarà che gli oggetti illuminati appariranno molto più vivi perché circondati da un alone luminoso. La differenza tra il Bloom ed il Tone sta nel fatto che nel primo caso si ha una semplice copertura dell'oggetto da parte della luce che tende ad assorbirlo (utile per simulare un oggetto con una forte luce alle spalle) mentre il Tone esegue una calibrazione della luce (spesso dinamica in modo da simulare il fenomeno di adattamento dell'occhio alla luce). Esistono centinaia di formule che si differenziano per qualità e complessità delle stesse.

Con questo si conclude quest'articolo.
La tecnologia Shader è qualcosa che per sua natura non può comunque essere sintetizzata ma richiede anzi continui studi ed approfondimenti in quanto oggetto di continuo sviluppo da parte di programmatori e ricercatori universitari.
Spero tuttavia di aver dato un buon punto di partenza ed indicato la direzione per continuare.