Al giorno d'oggi le ombre, nelle simulazioni 3D real time e non, sono diventate componenti fondamentali: nel corso del tempo si sono succedute una serie di tecniche per realizzarle in modo piu' o meno realistico e, questa volta, proverò a spiegarvi lo shadow volume.

A differenza degli altri effetti, le ombre non sono oggetti renderizzati sullo schermo (anche se, nel caso dello shadow volume, esse posseggono un loro vertex buffer), ma sono semplicemente delle aree dello schermo piu' scure rispetto all'ambiente circostanze perché ricevono meno luce durante il calcolo dell'illuminazione. La parte del rendering delle ombre è trovare le zone nelle quali c'è effettivamente meno luce.

Lo shadow Volume è un'interessante tecnica per generare le ombre di un oggetto a partire semplicemente dall'oggetto stesso e dalle varie fonti di luce che generano appunto l'ombra. E', inoltre, un pratico esempio di utilizzo dello Stencil Buffer. E' stato utilizzato in giochi importanti come Doom 3 dell'Id Software, e consiste, in sintesi, nel marcare nello stencil buffer i pixel che sono in ombra, e poi disegnarli.

La tecnica in sé non è difficile, la sua implementazione può risultare un pò antipatica (almeno a me è risultata così per motivi che capirete sotto.)

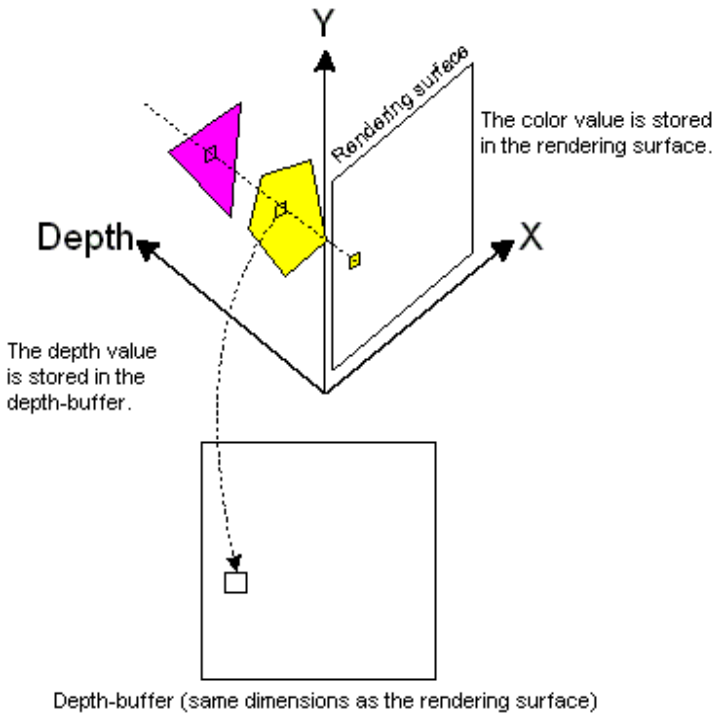
Attenzione però: il rendering di ombre è considerata comunque come tecnica avanzata, e richiede familiarità e conoscenze di computer graphics, anche se non troppo profonde (comunque in D3D10 la tecnica è molto più facile del normale). Quindi, se state leggendo questo tutorial, io riterrò implicite conoscenze dei fondamenti di matematica (sistemi di coordinate, trasformazioni, operazioni con vettori), che sappiate come funziona una luce e come trasformarla (potete leggere il mio precedente articolo), il caricamento e il draw di Mesh (di qualsiasi tipo, basta che abbia un Index Buffer e che sappiate generarne le adiacenze per il Geometry Shader) e l'uso di base del geometry shader (e potete leggere l'altro mio tutorial ancora, su questo).
Ovviamente SAPETE usare il C++.

Nonostante lo implementeremo in D3D10, l'algoritmo è completamente indipendente dall'API grafica usata. Per funzionare, questo richiede

Un DepthBuffer
Uno Stencil Buffer

Giusto per la cronaca, entrambi i buffer sono implementabili anche via software.

Darò una sommaria spiegazione di queste 2 superfici, tanto per chiarire in via definitiva a cosa servono.



Nell'immagine sopra proposta, la rendering surface è il back buffer. Supponiamo ci siano 2 geometrie da renderizzare. Possiamo facilmente notare che il pentagono giallo è in posizione meno profonda rispetto al triangolo viola. Ecco come il Depthbuffer lavora:
Renderizzo il triangolo viola; questo ha una profondità di x. Il DepthBuffer segna sulla sua superficie un riferimento al pixel del triangolo e segna come profondità x.
Renderizzo il pentagono, che ha profondità x/2. Il depth buffer incontra il nuovo pixel e lo confronta con il vecchio: se

triangolo.Profondità < Pentagono.Profondità, allora il pixel del pentagono passa dietro; altrimenti, quest'ultimo passa avanti.

E così è per tutti gli oggetti che vengono renederizzati su questa superficie. Il confronto può avvenire anche in maniera opposta, dipende dall'ordine di rendering che gli date.
I valori del DepthBuffer variano tra 0 e 1 (1 profondità massima, 0 profondità minima). Ad ogni rendering effettuiamo, come saprete, il Clear del Depth buffer, con un valore di riempimento che specifichiamo nella funzione. E' facile notare che, mettendo 1, come si fa di solito, gli oggetti verranno rende rizzati come di consueto; mettendo 0, niente verrà renderizzato (ogni confronto, infatti, fallirà, perché 1 è profondità massima). Mettendo un valore intermedio, verrà evitato il Draw dei pixel oltre una certa profondità. (Questa è una tecnica molto usata nei videogiochi.)

Il DepthBuffer può essere però programmato anche per svolgere azioni diverse dal normale. Possiamo, ad esempio, dirgli di far passare il pixel se la profondità è maggiore invece che minore, oppure far passare solo i pixel con uguale profondità rispetto alla precedente…interessanti usi che per ora non ci interessano, comunque il DepthBuffer può essere programmato dalla struttura D3D10_DEPTH_STENCIL_DESC.

Detto questo, passiamo brevemente allo Stencil.
Lo Stencil Buffer è una superficie analoga al DepthBuffer, ma è programmabile (può quindi svolgere determinati compiti. Ne vedremo un suo uso a tempo debito). Un'altra differenza, è che usa valori interi, e non float come il DepthBuffer

Veniamo adesso alla nostra ombra. Prendiamo una semplice geometria (una mesh, un triangolo, qualsiasi cosa, ma per semplicità vi consiglio di usare una sfera) e proviamo a pensare come generare un'ombra. Dopotutto, l'ombra di un oggetto non è l'altro che la proiezione dell'oggetto stesso a terra, su un muro, o qualsiasi superficie, rispetto alla direzione della luce.
Fate anche una prova pratica se volete: ponete un oggetto al sole, e guardate l'ombra che ne viene fuori. La posizione di quest'ultima dipende dalla posizione dell'oggetto, ma anche della direzione della luce. Sarete mai stati al mare vero? Ebbene, l'ombra degli ombrelloni cambia di ora in ora, proprio perché il sole varia la sua posizione nel tempo (cioè no, è la Terra che gira, si capisce, ma noi vediamo il contrario).

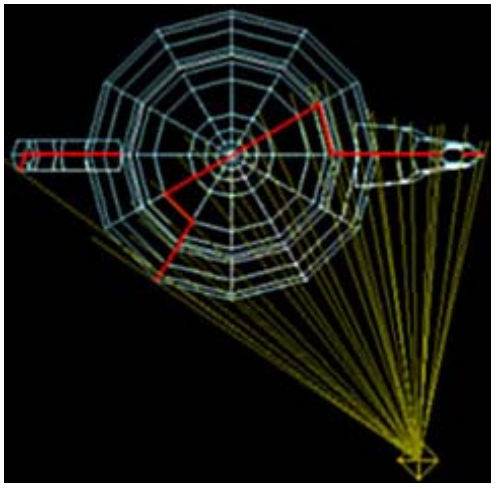
Poiché un'ombra non contiene, ovviamente, i particolari, ma soltanto la Silhouette (immagine di un oggetto che ne rende esclusivamente i contorni), possiamo costruire l'ombra a partire dai contorni della nostra geometria. Basterà dunque renderizzare questi ultimi lungo la direzione della luce. Attenzione però: un oggetto colpito dalla luce non genera sempre un'ombra completa, ma dipende dai contorni toccati dalla luce.

Ricordo che avremo un'ombra per ogni luce che colpisce l'oggetto, ma descriverò il processo, per semplicità, per una sola. Potrete poi calcolarne quante ne volete. (Piu' sotto ci sono alcune considerazioni su questo argomento)

Innanzi tutto, nella creazione del render target associato allo swap chain, dobbiamo specificare un formato che riservi alcuni bit per la superficie stencil, che useremo dopo. Quindi in DXGI_FORMAT, metteremo DXGI_FORMAT_D24_UNORM_S8_UINT, che, come dice la descrizione stessa
"A 32-bit z-buffer format that uses 24 bits for the depth channel and 8 bits for the stencil channel. "

Dobbiamo ora trovare il modo di trovare i contorni dell'oggetto rispetto alla luce che lo colpisce. Attenzione, **SOLO I CONTORNI**.
Ricordiamo che ogni geometria, anche se curva, è sempre fatta da triangoli, che è formato da lati. Ebbene, il contorno di un oggetto rispetto ad una luce è **l'ultimo lato colpito dalla luce prima che dietro di essi ci siano lati non colpiti.**

A prima vista il concetto non è chiaro sicuramente. Proviamo con un'immagine:

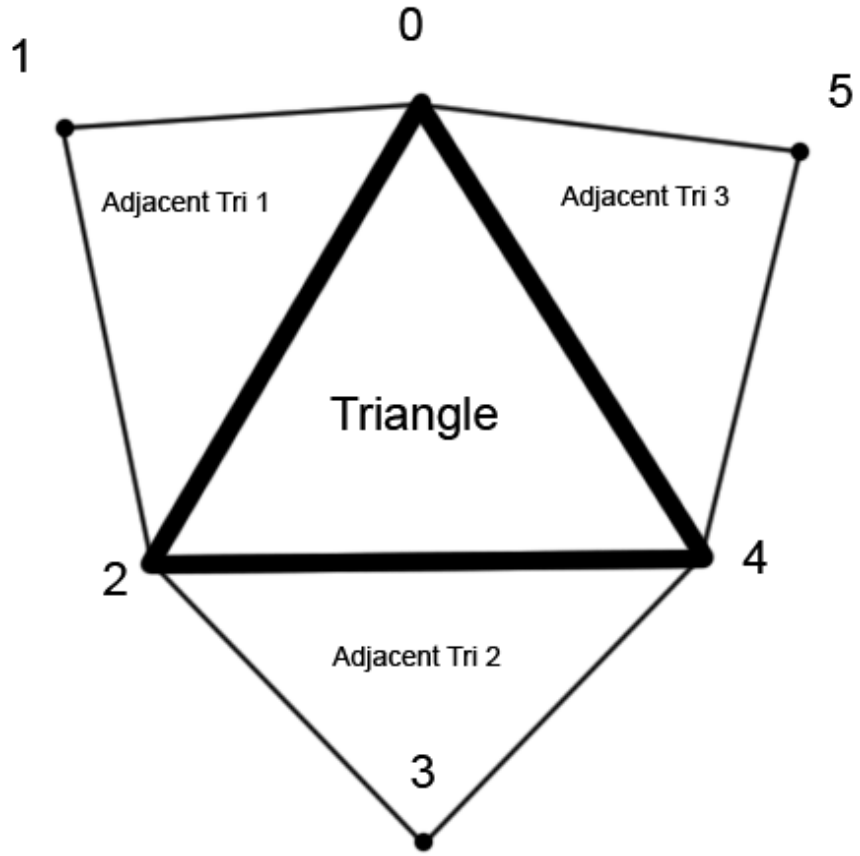


La linea spezzata (quindi formata da piu' segmenti) rossa è la silhouette. La caratteristica di quest'ultima è che ha un lato adiacente colpito dalla luce e un lato adiacente no. Dunque il contorno di un oggetto rispetto ad una luce, è,

volgarmente, quella geometria (quella serie di lati) nella quale i triangoli adiacenti ad essa verso una direzione, sono colpiti dalla luce, quelli nella direzione opposta no.

Qui sotto è la stessa cosa, rappresentato in 3D.

Saputa questa importante informazione, possiamo procedere. Per individuare i lati interessati, dobbiamo avere informazioni sui lati circostanti, o meglio, adiacenti, ad un dato lato, per effettuare i calcoli della luce.
In D3D9 tutto ciò veniva fatto via CPU, ora invece ci sono i geometry shaders, che per fortuna possono gestire intere primitive.
In particolare, può anche inviare informazioni proprio sulle adiacenze dei triangoli.



Ecco come il geometry shader invierà le informazioni necessarie: ci darà un array di 6 vertici, nei cui indici pari abbiamo il triangolo originale, in quelli dispari i lati delle adiacenze.

La generazione di un buffer di adiacenze non è una cosa proprio facilissima. Leggiamo questo estratto dalle FAQ di D3D10:

How does D3D10 know how to generate adjacency indices for my mesh? Or, why does D3D10 not render correctly when I specify that the geometry shader needs adjacency information.

> The adjacency information is not created by D3D10, but by the application. Adjacency indices are generated by the application and must contain six indices per primitive; of the six, the odd numbered indices are the edge adjacent vertices. ID3DX10Mesh::GenerateAdjacencyAndPointsReps can be used to generate this data.

Insomma, ci hanno detto che le adiacenze non vengono generate in automatico, ma dovete crearvele da soli. E non è mica una cosa facile. Io dopo molto lavoro sono riuscito a scrivere un algoritmo decente che faccia questo per me, ma ho trovato anche un'altra soluzione per voi.

La classe ID3DX10Mesh ha delle funzioni che fanno questo sporco compito. Dunque basterà creare una mesh in D3DX, inserire gli indici al suo interno, chiamare la funzione GenerateGSAdjacency(), riprendersi l'index buffer (che sarà raddoppiato in termini di dimensioni, perché per ogni triangolo ci sono altri 3 indici) e disegnare come prima.

Dunque:

1. Creare una nuova mesh con D3DX10CreateMesh.
2. Inseriteci gli indici
3. Chiamate GenerateAdjacencyAndPointReps() //Senza esagerare con il valore.
4. Chiamate GenerateGSAdjacency()
5. GetIndexBuffer
6. Prendetevi gli indici
7. Ricordatevi che gli indici sono raddoppiati in numero.

Rilasciate, ovviamente, tutte le risorse inutili.
Io vi consiglio comunque di sviluppare una vostra versione dell'algoritmo. Per provare poi se tutte le adiacenze sono state generate bene, potete fare un geometry shader che disegna soltanto i triangoli delle adiacenze (seguendo il disegno sopra): se le adiacenze sono corrette, queste ultime disegneranno, implicitamente, i triangoli originali della mesh.
Adesso che abbiamo anche le adiacenze, passiamo al Draw.
In primo luogo dobbiamo specificare come Primitive Topology un valore con _ADJ di suffisso. In questo modo D3D10 manderà al vertex shader, da un index buffer che contiene sia i vertici che i dati sulle adiacenze, soltanto i veri vertici (ciò è gli elementi pari dell'index buffer). Le adiacenze vere e proprie le prenderemo nel Geometry Shader.

Prima di generare l'ombra vera e propria, cerchiamo di renderci conto dei contorni che andremo a generare. E' una cosa in più che vi consiglio di fare, altrimenti passate direttamente alla generazione dello Shadow Volume qui sotto.

Prima di tutto renderizzate l'oggetto normalmente, con uno shader semplice. Avrete il vostro oggetto sullo schermo. Passiamo ora ai contorni.

Dovrete renderizzare di nuovo lo stesso oggetto, ma con un secondo Pass, che ora andremo a specificare

Il vertex shader resterà sempre lo stesso. L'unica cosa è che moltiplicherete la posizione soltanto per la WorldMatrix.
Il pixel shader sarà un semplice return (1,0,0,1), per colorare i contorni.

E' il geometry shader la roba tosta. Vediamo il codice con le dovute spiegazioni

```
float3 GetNormal( float3 A, float3 B, float3 C )
{
	float3 AB = B - A;
	float3 AC = C - A;
	return normalize( cross(AB,AC) );
}

 [maxvertexcount(6)]
void GSScene( triangleadj GSSceneIn input[6], inout LineStream<PSSceneIn> OutputStream )
{
	float3 TriNormal = GetNormal( input[0].Pos.xyz, input[2].Pos.xyz, input[4].Pos.xyz );



	if( dot( TriNormal, g_ViewSpaceLightDir ) > 0	{
		PSSceneIn output;

		for( uint i=0; i<6; i+=2 )
		{
			uint iNextTri = (i+2)%6;
```

```
            float3 AdjTriNormal = GetNormal( input[i].Pos.xyz, input[i+1].Pos.xyz, input[
iNextTri ].Pos.xyz );


            if ( dot( AdjTriNormal, g_ViewSpaceLightDir ) <= 0 )
                  {

                        output.Pos = mul( input[i].Pos, g_mProj );
                        output.Norm = input[i].Norm;
                        output.Tex = input[i].Tex;
                        OutputStream.Append( output );

                        output.Pos = mul( input[ iNextTri ].Pos, g_mProj );
                        output.Norm = input[ iNextTri ].Norm;
                        output.Tex = input[ iNextTri ].Tex;
                        OutputStream.Append( output );

                        OutputStream.RestartStrip();
                  }
            }
      }

}
```

Per prima cosa è cambiato l'input: triangleadj. Questo significa che il geometry shader preleverà dall'index buffer 6 indici per volta (e ci darà altrettanti vertici), in cui ci saranno le adiacenze negli indici dispari, il triangolo originale in quelli pari. Inoltre, stamo usano l'oggetto LineStream, per disegnare contorni, lati, composti da 2 vertici.

Per verificare se la luce tocca un vertice, basta fare il dot product tra la direzione della luce (normalizzata) e le normali del vertice.
Leggendo il mio precedente tutorial, avrete capito che il dot product restituisce l'angolo tra i 2 oggetti dati (se normalizzati) come operandi. Se l'angolo è > 0, l'oggetto è colpito dalla luce, altrimenti no.

Domanda lecita che può venire in mente: se io nella mia struttura che contiene i vertici ho già le normali, perché devo ricalcolarle con il cross product? Come posso calcolare la normale dai 3 dati che ho nei vertici?

In primo luogo, la normale alla superficie può essere ottenuto tramite questa pratica formula

```
float3 Normal = (Vertice1.Nor + Vertice2.Nor + Vertice3.Nor) / 3;
//oppure
float3 Normal = normalize((Vertice1.Nor + Vertice2.Nor + Vertice3.Nor));
```

Ossia la media aritmetica delle 3 normali, oppure la normalizzazione della loro somma (il secondo esempio è migliore)
Che differenza c'è tra questa normale e quella ottenuta tramite il cross delle posizioni? Perché si preferisce il primo metodo?

C'è, in effetti, una differenza tra I 2 metodi. Il primo calcola la normale alla faccia direttamente dalla geometria, ma questo potrebbe risultare scomodo, per esempio, nelle superfici curve.

Perché? Si sa ovviamente che anche la sfera piu' sferica in computer graphics è formata da triangoli e quindi questa non sarà mai "liscia" (una prova è l'utilizzo della luce lambertiana in vertex shader anziché nel pixel, o anche il semplice Goraud Shading). Questo perché le normali calcolate dalla faccia desiderata, sarà perpendicolare ovviamente alla faccia, ma non alla sfera ipotetica che vogliamo renderizzare. Per ottenere questo, sarebbe necessario (fisica) dividere la superficie della sfera in tanti Δs tale che ognuno di questi abbia la normale perpendicolare alla superficie. Ciò significa scomporre la sfera in talmente tanti triangoli che anche un'ottima scheda video morirebbe.

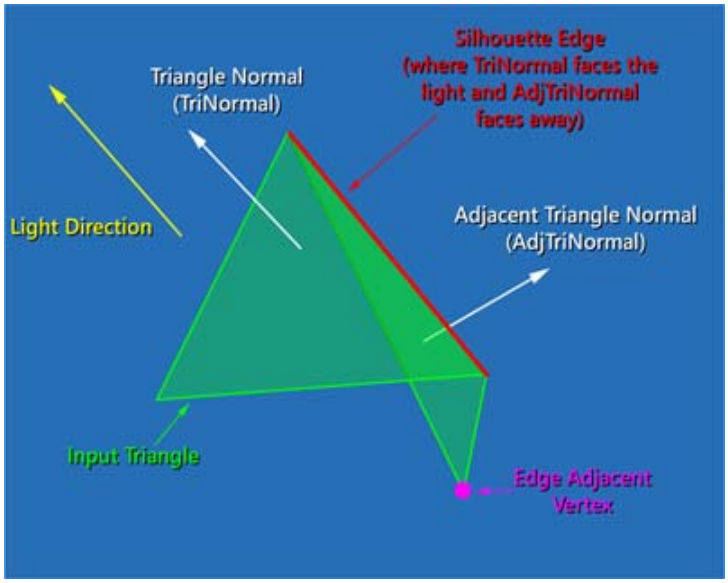
Dunque di sceglie di "modificare" un po' le normali che vengono scritte nella struttura dei vertici per assicurare una miglior resa degli effetti di luce. Tanta precisione in un calcolo così semplice!
Calcolando la normale dalle normali relative ai vertici ci assicuriamo che queste seguano la "curvatura" della superficie. In casi di precisioni, dunque, il secondo metodo è migliore.

Ma allora perché si usa ancora il primo metodo?
Semplicemente perché il calcolo delle normali dalle normali non sempre va a buon fine (anche se nella maggior parte dei casi riesce).
Essendo una semplice addizione, questa può crollare semplicemente nel caso in cui

```
normalize(vec3(-1,0,0) + vec3(1,0,0));
```

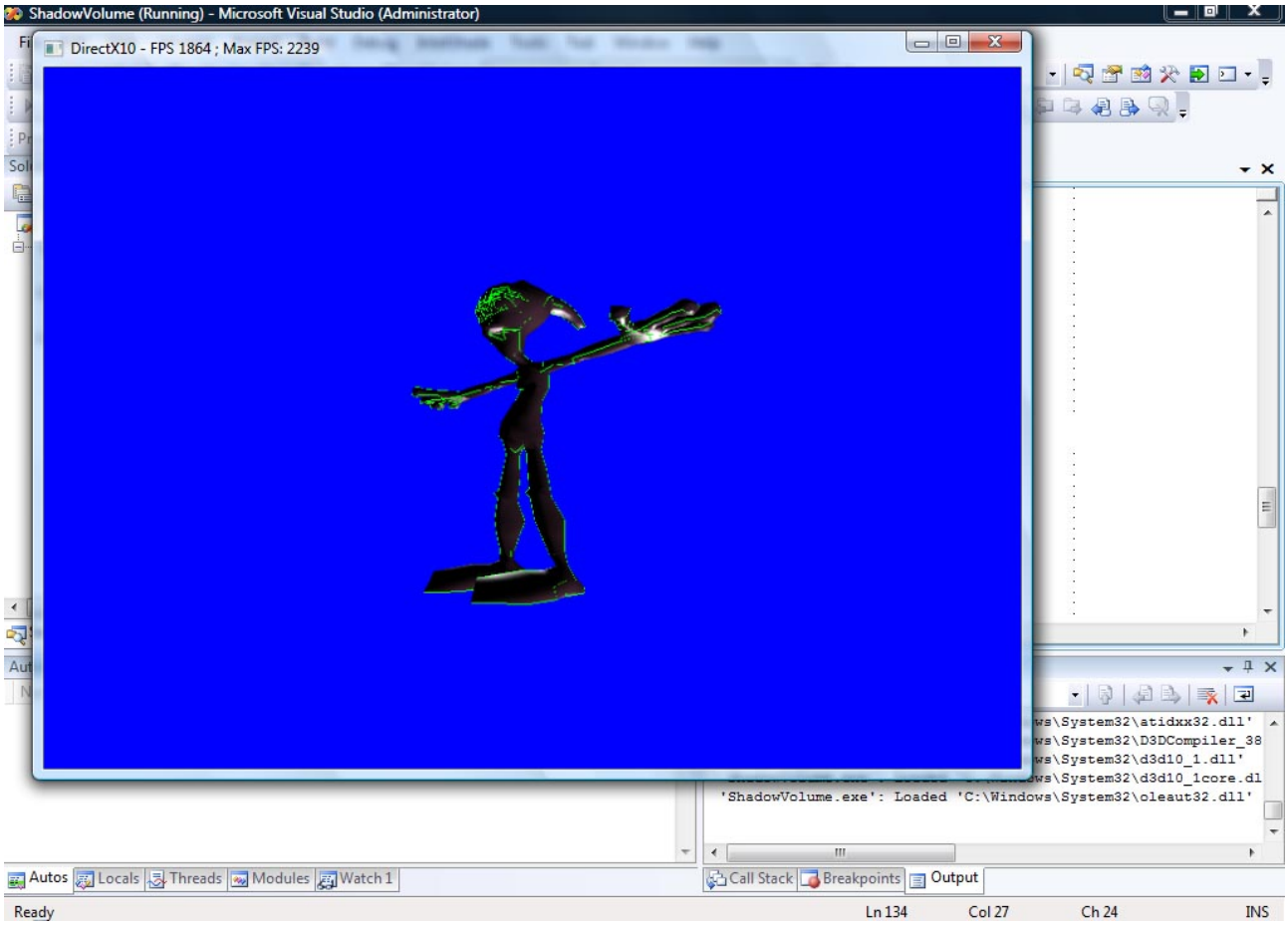
Comunque, un triangolo, per avere delle normali così, è talmente deformato che praticamente non ha ragione di esistere. E comunque, problemi di calcolo nelle normali possono accadere anche con il primo metodo. Scegliete quello a voi piu' congeniale, osservando le differenze nei risultati.
Ad ogni modo, ora andiamo avanti.

Se il triangolo interessato è effettivamente colpito dalla luce, dobbiamo verificare (come abbiamo visto sopra), che ci sia un triangolo adiacente NON colpito dalla luce. Se sono rispettate le 2 condizioni, disegniamo il lato interessato. Ovviamente è inutile controllare se sul lato opposto c'è la luce (bastano 2 su 3 condizioni) e risparmiamo in prestazioni.

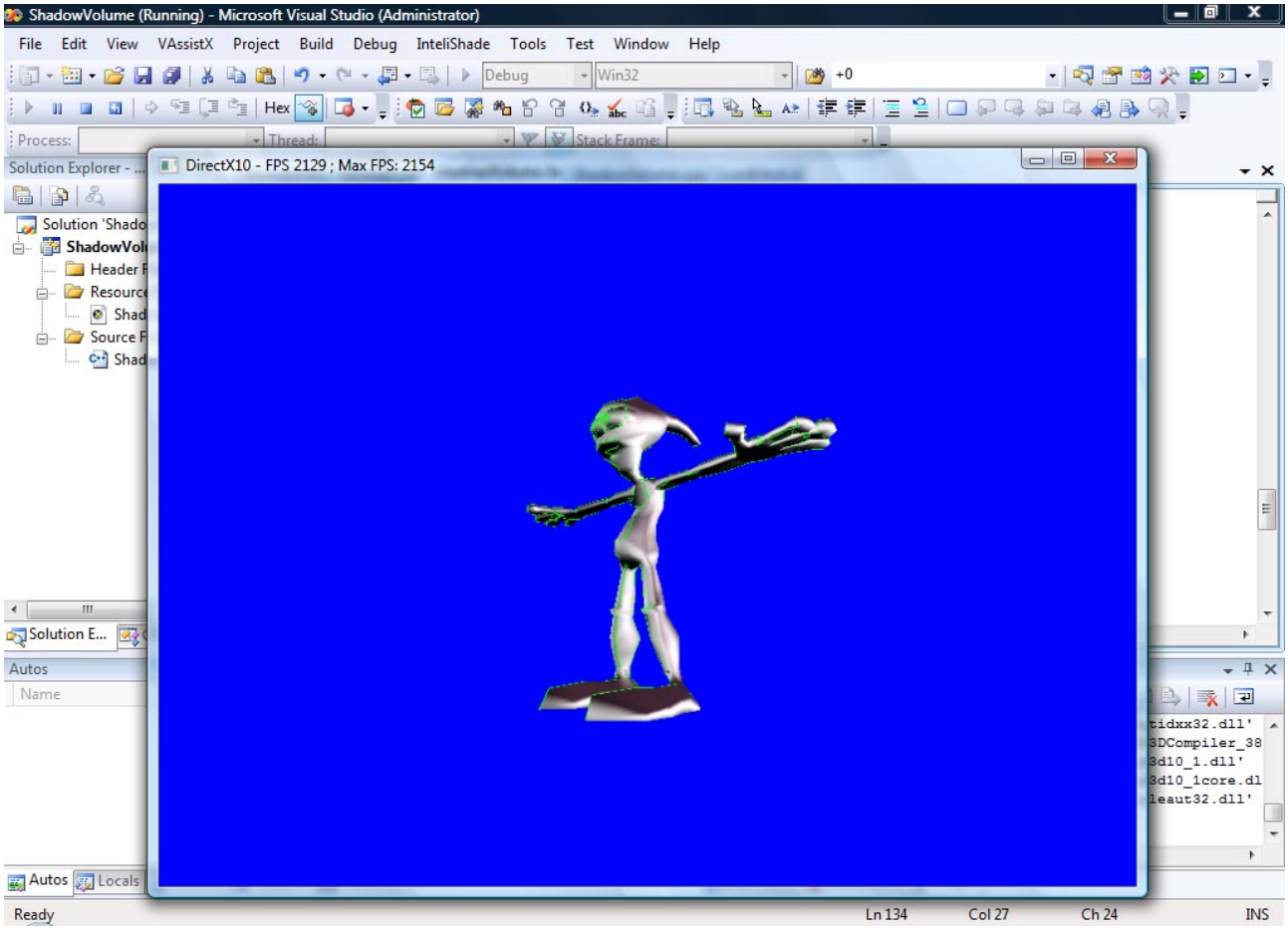


In rosso il lato che sarà disegnato.

Avviamo il nostro demo: se avremo fatto tutto bene, vedremo una scena come questa (io ho colorato i bordi di verde

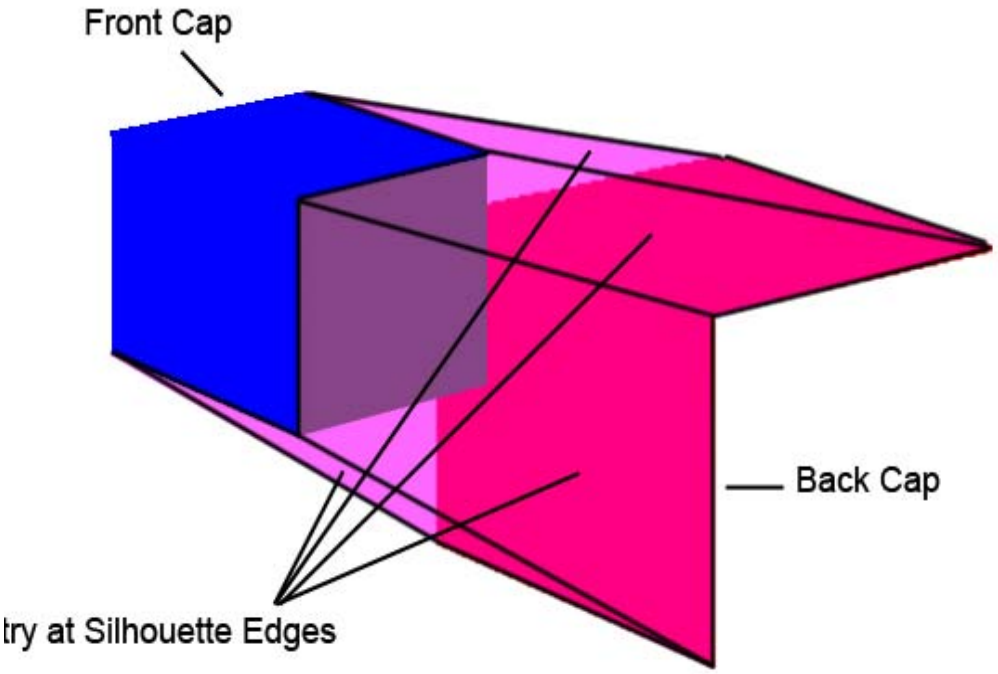


Questo è in situazione di minima luce.



Questa è la situazione al massimo di luce.
Se volete notare ancora meglio i bordi, potete evitare il rendering della geometria originale.

Passiamo ora al secondo step: generare il cono d'ombra.
Ora che abbiamo i bordi della geometria, dobbiamo proiettarla, sempre secondo la direzione della luce.



Guardate l'immagine: il cubo blu è l'oggetto originale, e proietteremo tutti quei bordi che rispettano la condizione dettata sopra.

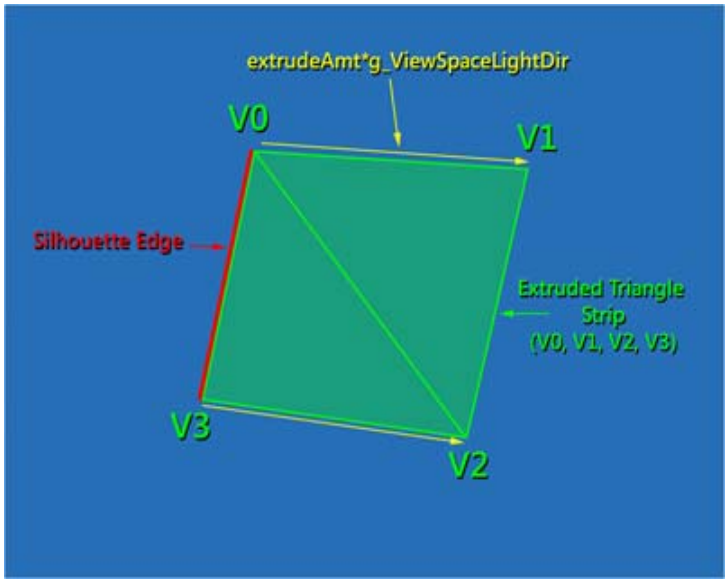
Per fare questo, creeremo, per ogni vertice che rispetta la solita condizione, un altro vertice con stesse coordinate, ma moltiplicato per la direzione della luce * valore scalare.

In questo modo i contorni si prolungheranno nel verso della luce.
Questo valore scalare, potete considerarlo come altezza dell'ombra: piu' sarà negativo, piu' l'ombra andrà lontano rispetto alla direzione della luce.
Modifichiamo dunque il Geometry Shader come segue

```
void gs_shadow(triangleadj VS_INPUT input[6],inout TriangleStream<PS_INPUT> Stream)
{

	float3 TriNormal = GetNormal( input[0].Pos.xyz, input[2].Pos.xyz, input[4].Pos.xyz );

	if( dot( TriNormal, normalize(LightDirection.xyz) ) > 0 )
	{
		PS_INPUT output = (PS_INPUT)0;

		for( uint i=0; i<6; i+=2 )
		{
			uint iNextTri = (i+2)%6;

			float3 AdjTriNormal = GetNormal( input[i].Pos.xyz, input[i+1].Pos.xyz, input[
iNextTri ].Pos.xyz );

			if ( dot( AdjTriNormal, normalize(LightDirection.xyz) ) <= 0 )
			{
				float4 v0 = input[i].Pos;
				output.Pos = mul(mul(v0,ViewMatrix),ProjMatrix);
				Stream.Append(output);

				float4 v1 = v0 + ExtrudeValue *
float4(normalize(LightDirection.xyz),0);
				output.Pos = mul(mul(v1,ViewMatrix),ProjMatrix);
```

```
                                Stream.Append(output);

                                float4 v2 = input[iNextTri].Pos;
                                output.Pos = mul(mul(v2,ViewMatrix),ProjMatrix);
                                Stream.Append(output);

                                float4 v3 = v2 + ExtrudeValue *
float4(normalize(LightDirection.xyz),0);
                                output.Pos = mul(mul(v3,ViewMatrix),ProjMatrix);
                                Stream.Append(output);

                                Stream.RestartStrip();
                        }
                }
        }

}
```

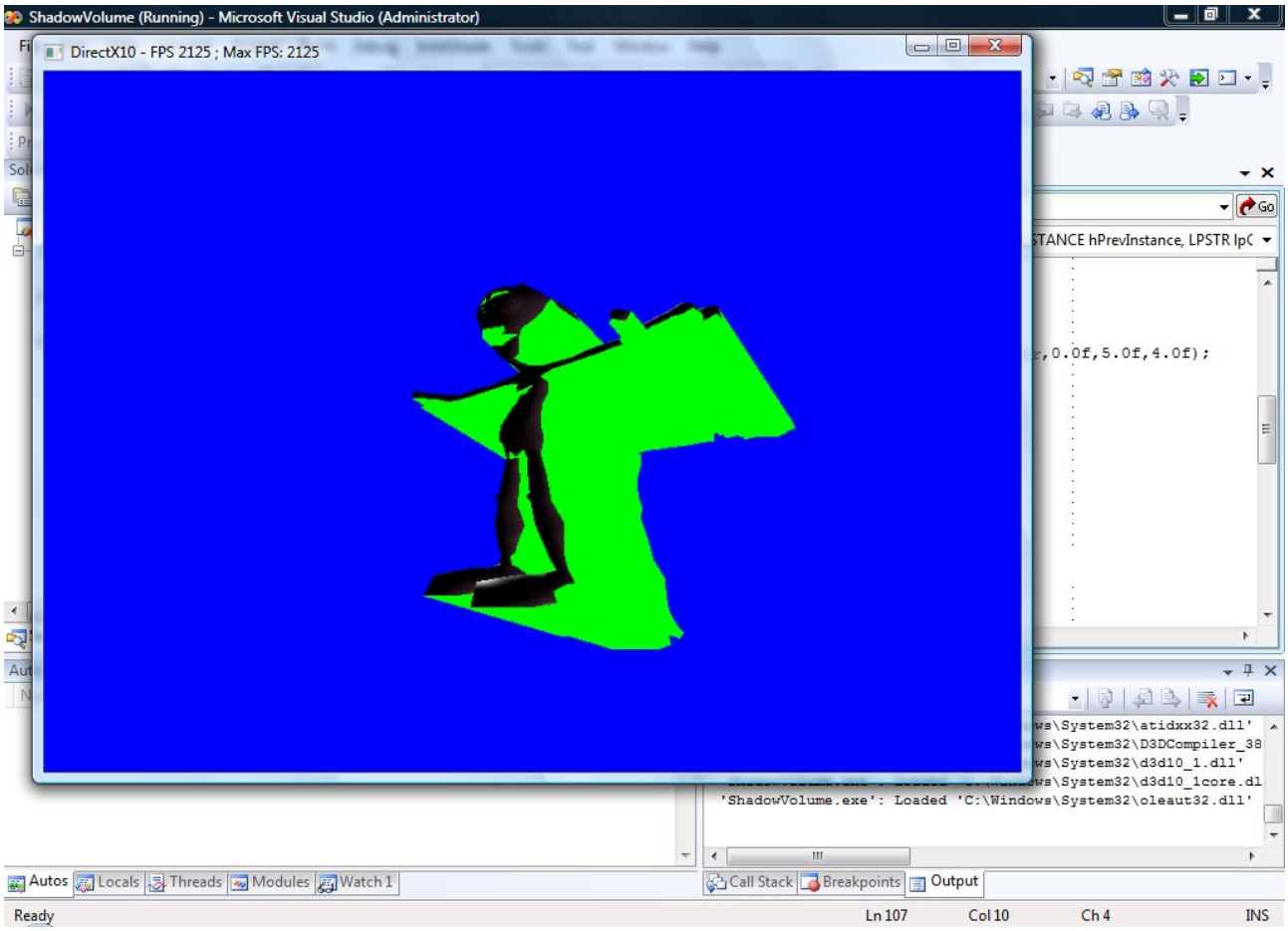
Notate in primo luogo che stiamo riutilizzando TriangleStream. Come potete vedere, dai vertici normali ne ricaviamo un altro "prolungato" lungo la direzione della luce, per un'"altezza" di ExtrudeValue (un float…-30 dovrebbe andar bene).



Il valore di estrusione indica quanto deve essere lungo il lato del poligono del volume dell'ombra. Oltre questo valore diciamo che l'ombra "non ci arriva", e esaurisce li il suo effetto.
Teoricamente questo dovrebbe essere pari a infinito ma con intensità che diminuisce con il quadrato della distanza. Comunque, assoggettate il valore ai vostri bisogni (se voglio che l'ombra vada sul muro, allungo il valore tale che questa arrivi, no?). Questa tecnica viene chiamata estrusione finita.
La magnitudine dell'estrusione, comunque, può essere anche infinita. La vedremo in seguito.

Come vedete, tramite questa tecnica, il lato V3-V0 è stato proiettato su V1-V2, creando un vero e proprio poligono, in effetti.

Il risultato di tutti questi poligoni (uno per lato), sarà il seguente.



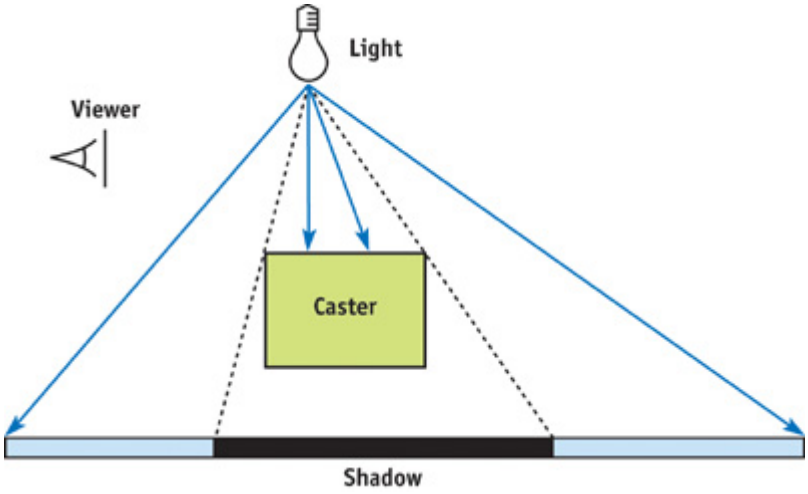
Tutto questo "volume" verde che si è creato, viene chiamato shadow volume, che non è altro che il nome della tecnica.

Una volta arrivati a questo punto, dobbiamo solo renderizzare un po' meglio il risultato dell'estrusione, dandogli le sembianze di un'ombra.
In primo luogo vi consiglio di creare un "ambiente", create una stanza texturizzata e piazzateci un oggetto al centro.
Per verificare la correttezza dell'ombra generata, vi consiglio di fare una stanza con delle scale.

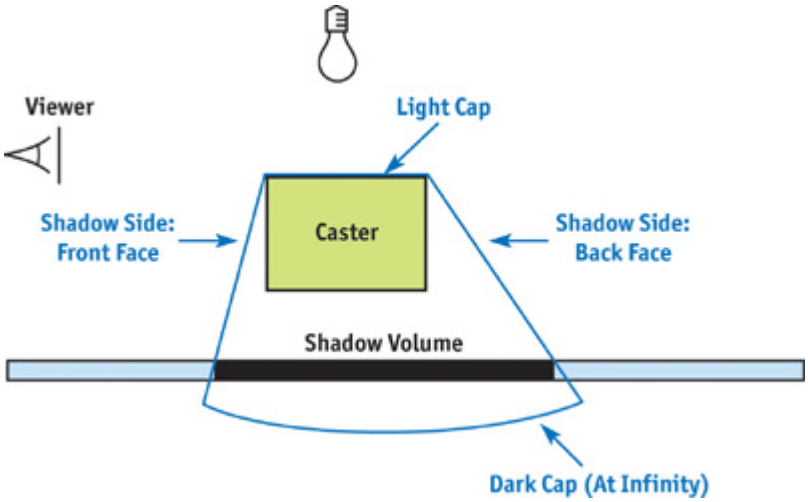
Prima cosa diamo una breve definizione di Culling.
Il culling è un valore che indica se il triangolo considerato di una geometria guarda o no verso la telecamera.
Solitamente, per aumentare le prestazioni delle applicazioni, si preferisce evitare il rendering di quelle primitive che non guardano verso la telecamera (tanto non possiamo vederli).
Come fa DirectX a capire se un poligono guarda o no verso la telecamera? In base al verso delle normali. E' importante che queste siano sempre calcolate come si deve, nei vostri software di modellazione 3D, per evitare inaspettate sorprese.

Per renderizzare la nostra ombra (soltanto l'ombra, non tutto lo shadow volume) dobbiamo trovare un modo per capire se il pixel interessato al rendering si trova effettivamente o no nella zona d'ombra, e segnarlo nello stencil buffer.

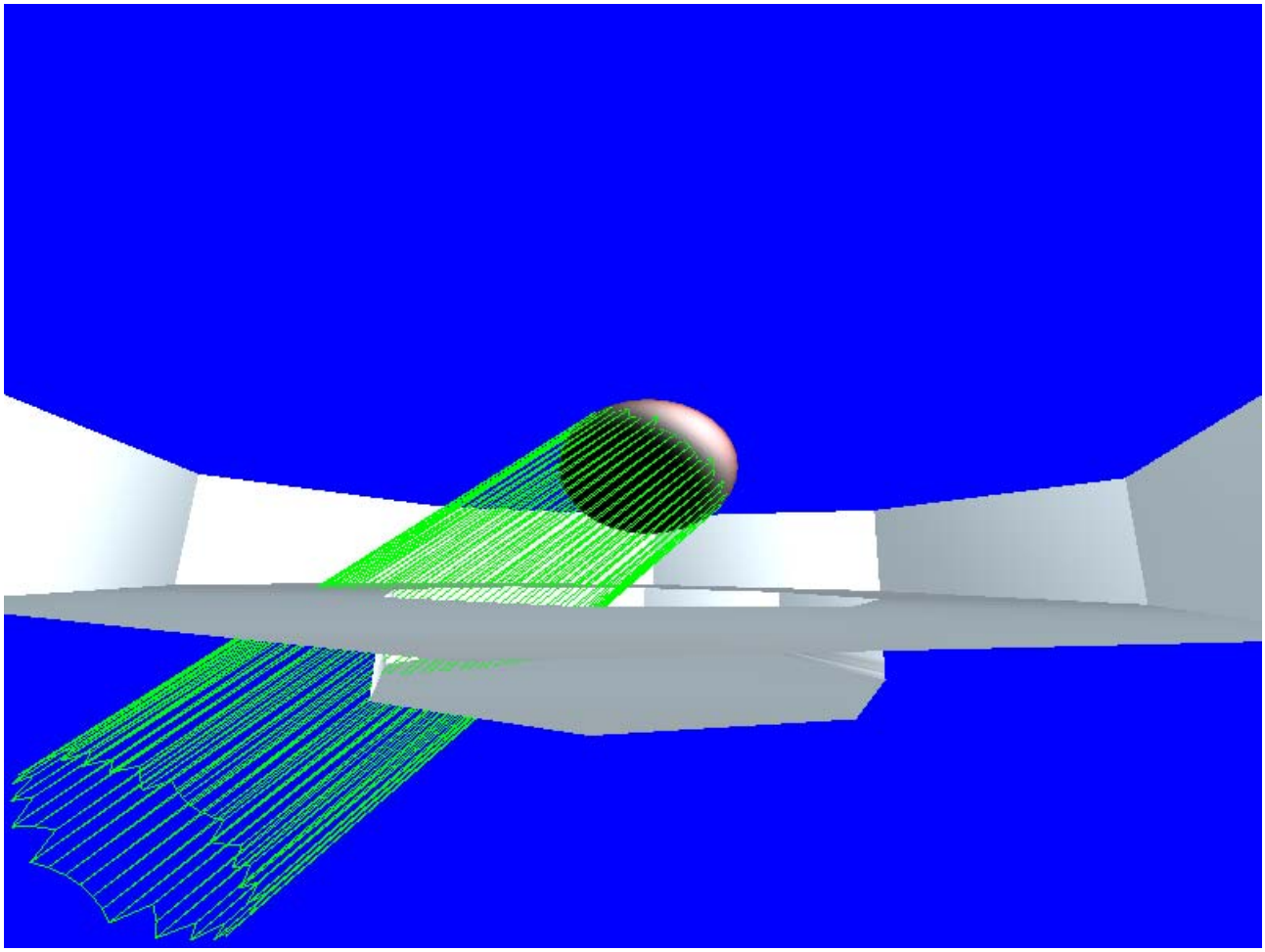
Prima di passare all'implementazione vera e propria, cerchiamo di capire i concetti matematici e logici che ci sono dietro.

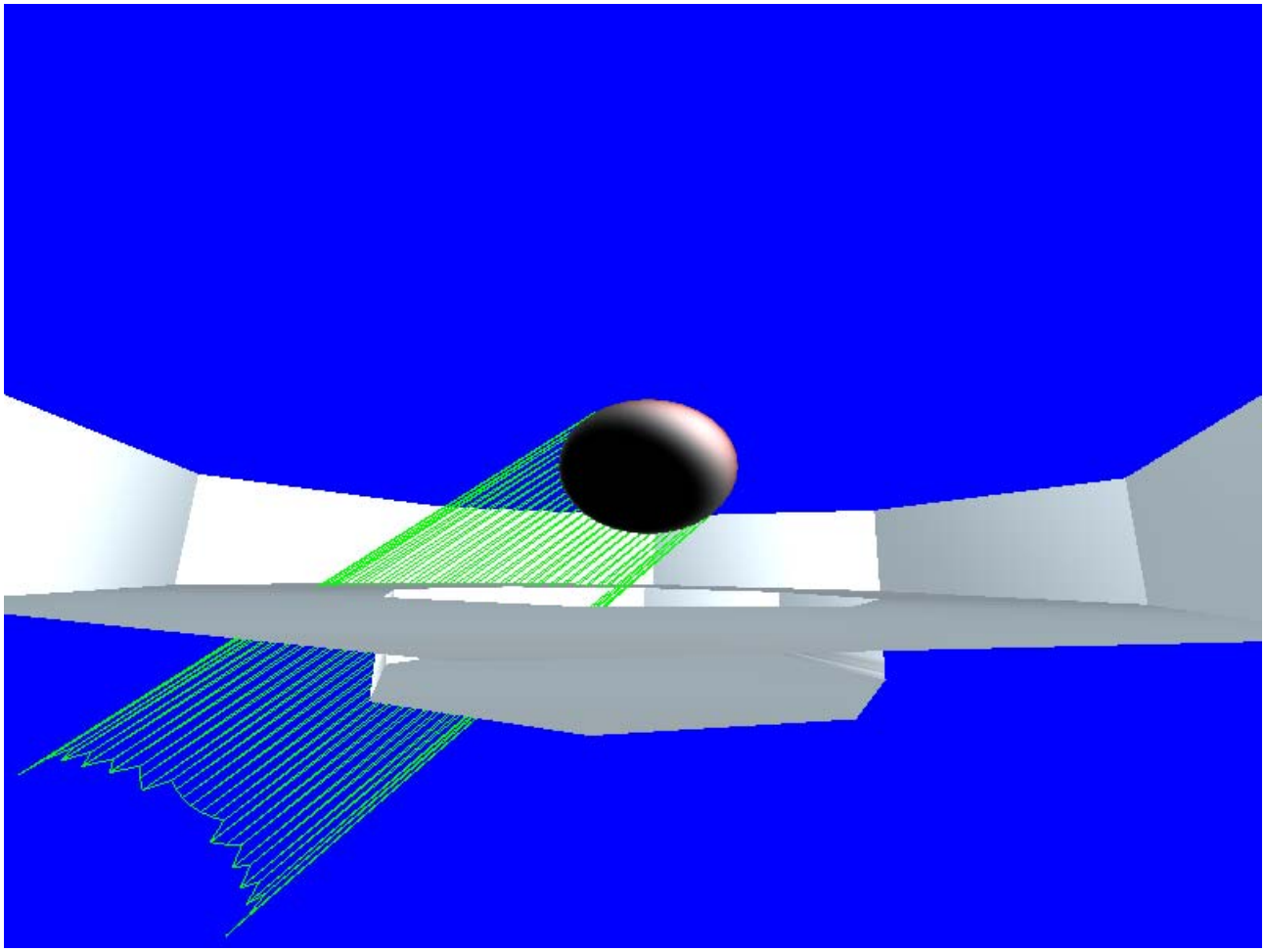


In questa semplice scena, abbiamo un osservatore con una singola sorgente di luce puntiforme che manda vari raggi di luce (ovviamente sono visualizzati solo alcuni: le point light vanno in tutte le direzioni): una scatola che è il caster dell'ombra.
Rivediamo ora la stessa immagine, ma con lo Shadow volume che abbiamo creato sopra.



Vediamo che lo Shadow Volume è formato da 4 parti essenziali: il LightCap, che è il punto nel quale inizia lo shadow volume (sono i poligoni in frontface rispetto alla luce), il dark cap,che non è altro che il light cap proiettato con l'estrusione (che può essere estruso anche all'infinito, come vedremo piu' tardi), oltre alla solita front e back face, che ci sono sempre in quanto lo shadow volume è comunque una geometria). L'ombra vera e propria è soltanto il pezzo colorato in nero.





Ecco una situazione pratica di ciò che abbiamo visto nell'immagine sopra: per mostrarvi meglio, ho rende rizzato prima solo in front e poi in back face: potete vedere il front e back cap, oltre che il light e dark cap (notate come il dark cap va oltre al muro su cui deve andare l'ombra).

Poiché tutto parte dallo shadow volume generato prima, questa tecnica funziona (con alcune eccezioni, di cui parleremo sotto) su qualsiasi tipo di geometria

Guardate l'immagine in back face: la zona sul muro, seppur immaginaria, è l'ombra del nostro oggetto in questione. Il rendering della superficie in front face, comprende invece tutto lo shadow volume. Sarà quindi sufficiente fare la differenza tra i 2 culling per ottenere l'ombra dell'oggetto richiesto!

C'è però un dubbio che potrebbe venirvi in testa: "Ma come può funzionare questa tecnica? L'intuizione è buona, ma front face e back face non sono sullo stesso piano, hanno valori di profondità differenti: in questo modo, la sottrazione tra i 2 culling porterebbe un'area a +1 e un'area a -1, avendo come risultato l'intero shadow volume come ombra (perché, come abbiamo detto sopra, i valori diversi da 0 li avremmo considerati "ombrati" "

Il dubbio è lecito, e in effetti il ragionamento fila, ma ecco ciò che vi è sfuggito (e anche a me, per molto tempo): lo ZBuffer NON percepisce la profondità! Lo ZBuffer è una semplice superficie sulla quale vengono scritti valori, e vengono quindi anche sovrascritti. Essendo una superficie paragonabile ad una texture (Anzi, è PROPRIO una texture), ogni punto sarà individuato soltanto da coordinate X e Y. Ecco come arriveranno sullo ZBuffer i poligoni in front e back face del nostro shadow volume:

Vedete come lo ZBuffer non tiene conto della profondità dei poligoni in front e back (che invece è ben evidente nelle 2 immagini sopra).
Di conseguenza, loro differenza (l'immagine è chiarissima) darà proprio l'ombra dell'oggetto in questione.

Dobbiamo solo renderizzare questa immagine pari pari nello stencil buffer: disegneremo la prima, e poi toglieremo la seconda, ottenendo la nostra ombra finale.

Come detto sopra,lo stencil buffer è una superficie programmabile, che scriverà valori interi a seconda di determinate condizioni che noi gli diremo.
Agiremo in questo modo: creeremo un contatore di intersezioni che, nel rendering dello shadow volume (non dell'ombra, ma del volume dell'ombra) incontra un poligono che guarda verso la telecamera, ma che supera il DepthTest, aumenteremo un contatore. Altrimenti, lo decrementeremo.
Tutti quei punti in cui il contatore non è stato decrementato, significa che sono punti che appartengono al volume d'ombra.

Nel rendering dello shadow volume aggiungeremo perciò, questo DepthStencilState

```
DepthStencilState StencilShadow
{
    DepthEnable = true;
    DepthWriteMask = ZERO;
    DepthFunc = LESS;

    StencilEnable = true;
    StencilReadMask = 0xff;
    StencilWriteMask = 0xff;

    FrontFaceStencilFunc = ALWAYS;
    FrontFaceStencilPass = INCR;
    FrontFaceStencilFail = Keep;

    BackFaceStencilFunc = ALWAYS;
    BackFaceStencilPass = DECR;
    BackFaceStencilFail = Keep;

};
```

Ricordiamoci di mettere su un rasterizerstate che disabiliti il CULL. Per effettuare il conteggio dei poligoni interessati, infatti, dobbiamo renderizzare tutti i triangoli e non evitare il rendering di quelli che non guardano la telecamera.
Il DepthTest è stato abilitato ma senza che possa scrivere (la profondità della scena è stata già calcolata in precedenza, con il semplice rendering. Utilizziamo il depth test soltanto per verificare la condizione descritta sopra).
Non diamo nessuna funzione particolare allo stencil, diciamo semplicemente di contare i poligoni in Back e Front Culling.
Ovviamente questa funzione, anche se con lo stencil programmato, produce uno shadow volume (come potete vedere nell'immagine sopra). Dovete quindi non renderizzarlo in qualche modo. Basta semplicemente azzerare la scrittura sul framebuffer, che si ottiene creando un blending e mettendo come maschera di scrittura del framebuffer un grandissimo 0x0. Per tornare alle condizioni normali, invece, 0x0f.

Vi consiglio di specificare **ESPLICITAMENTE** per ogni pass il proprio Rasterizer, DepthStencil e Blend States, perché questi, a quanto pare, vengono mantenuti di pass in pass. Ci ho perso la testa per questo problema.

Oramai è fatta: dobbiamo semplicemente creare un quadratone nero grande tutto lo schermo (senza matrici, semplicemente un quarato da -1 a 1) e renderizzarlo su schermo.

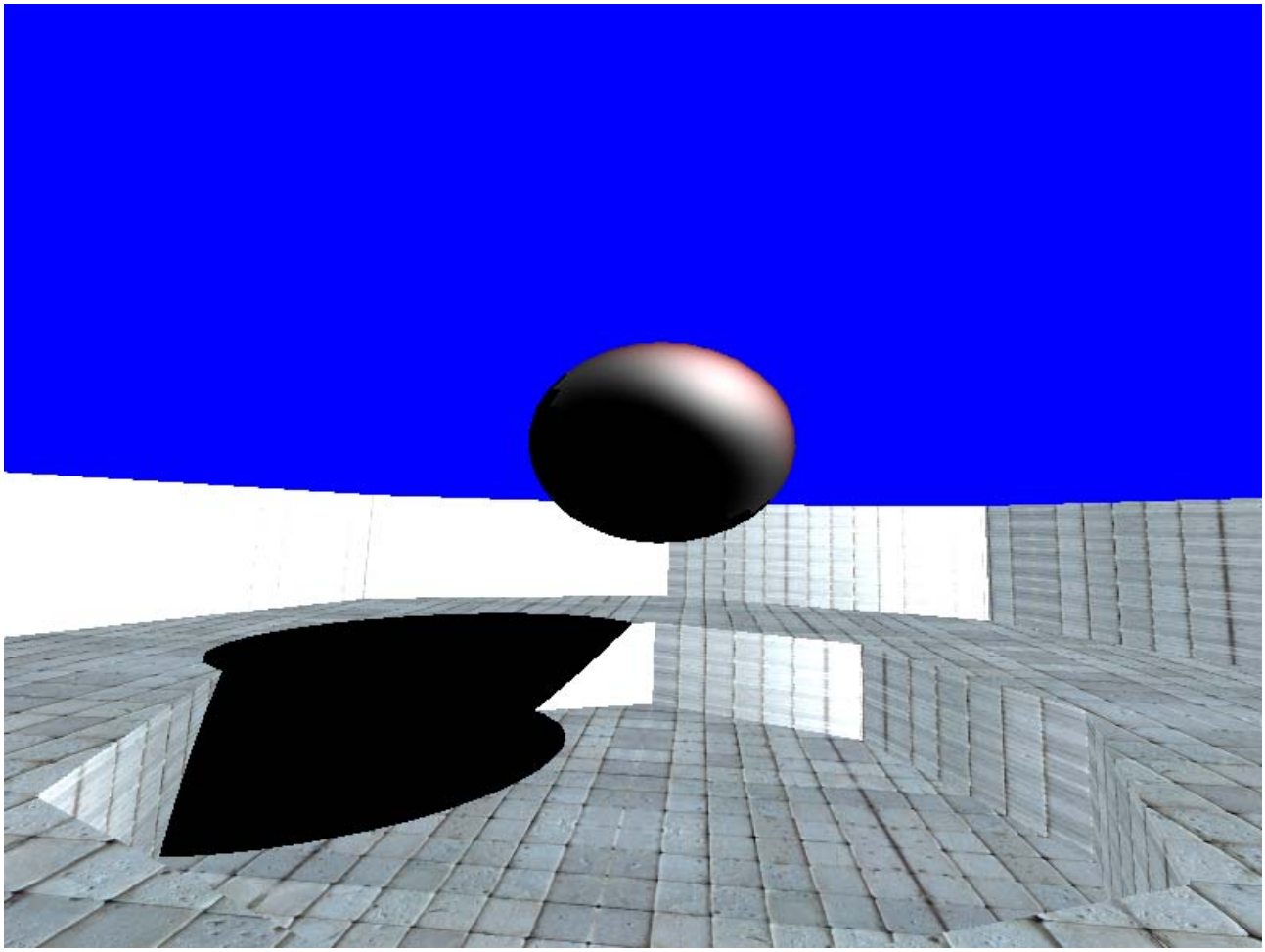
Nel rendering di quest'ultimo, però, utilizzeremo il valore dello stencil per mandare a video solo quei pixel che hanno come valore di stencil 1, in modo che sarà renderizzato solo l'ombra.

```
DepthStencilState StencilMask
{
    DepthEnable = false;
    DepthWriteMask = ZERO;
    DepthFunc = LESS;

    StencilEnable = true;
    StencilReadMask = 0xff;
    StencilWriteMask = 0xff;

    FrontFaceStencilFunc = LESS_EQUAL;
    FrontFaceStencilPass = Keep;
    FrontFaceStencilFail = Zero;

    BackFaceStencilFunc = EQUAL;
    BackFaceStencilPass = Keep;
    BackFaceStencilFail = Zero;

};
```

Ovviamente **NON** pulite mai lo stencil tra l'operazione di shadow volume e rendering del quadrato, altrimenti non funzionerà un bel niente.

Altra nota importante: nello Shader la funzione DepthStencilState prende, dopo appunto la struttura interessata, un valore int che indica il valore dello stencil con il quale effettuerà i vari confronti. Nello stencil shadow possiamo scriverci quello che ci pare, perché tanto la funzione di comparazione è ALWAYS quindi passerà sempre. Nel secondo è opportuno invece inserire il valore corretto.
Il valore con cui pulire lo stencil va ovviamente specificato nel clear.

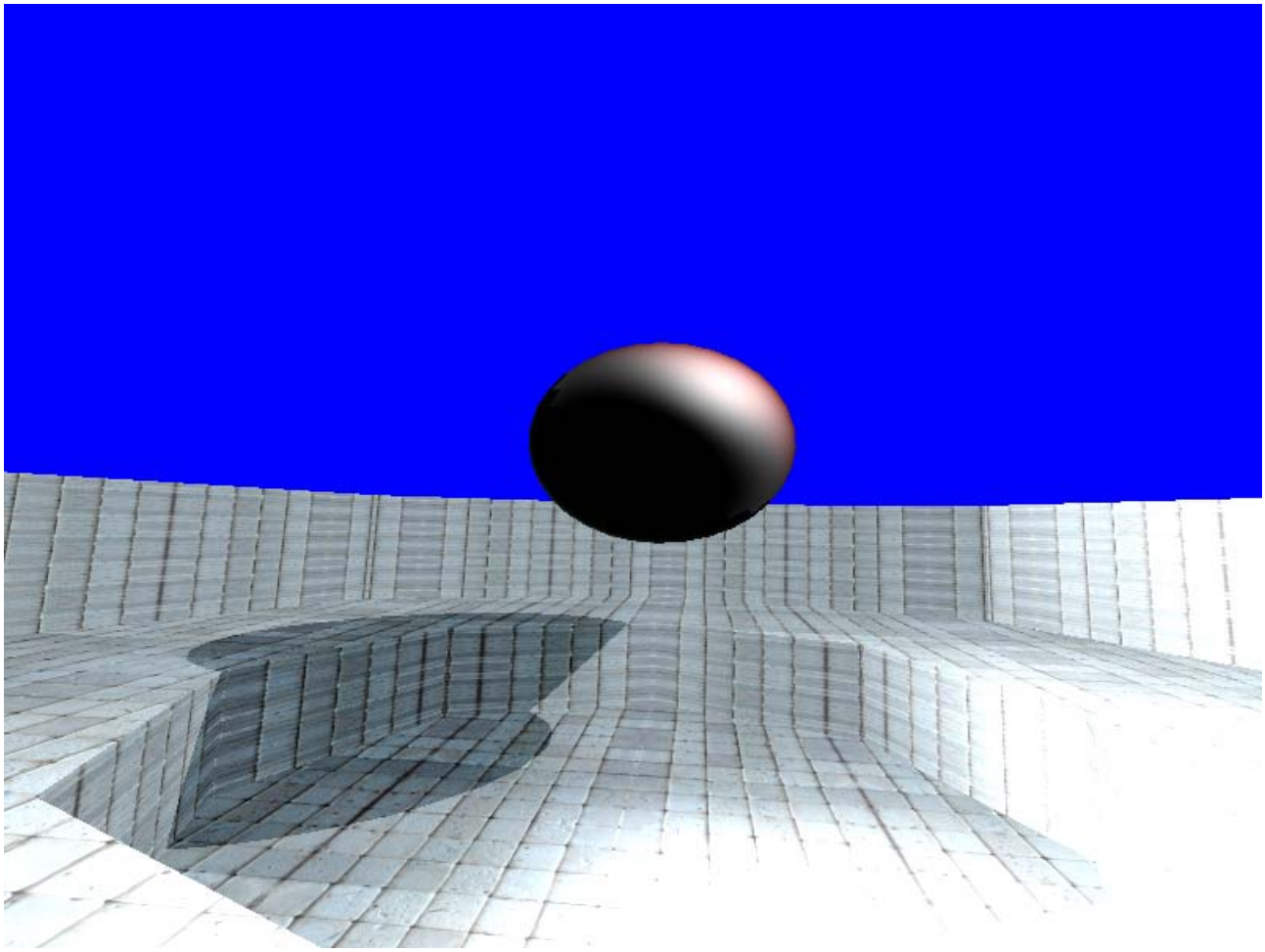
La prima volta la nostra ombra potrebbe presentarsi in questo modo



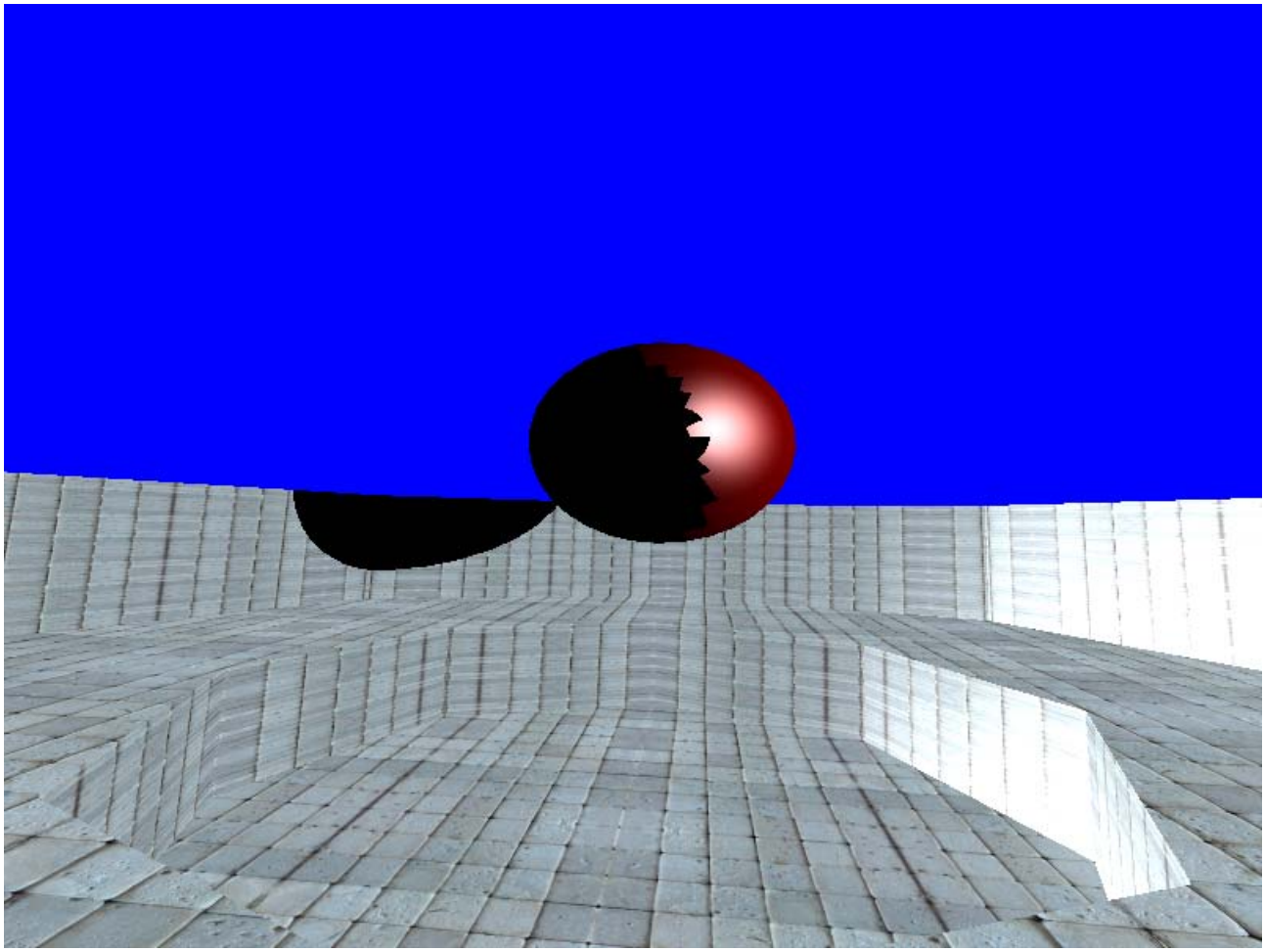
Come noterete sicuramente l'ombra è completamente nera e non lascia vedere niente sotto di essa. L'effetto è comunque realistico, ma se volete dare una leggera trasparenza, potete aggiungere un lieve effetto di blending.

```
BlendState AdditiveBlending
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = TRUE;
    SrcBlend = SRC_COLOR ;
    DestBlend = DEST_COLOR ;
    BlendOp = ADD;
    SrcBlendAlpha = ZERO;
    DestBlendAlpha = ZERO;
    BlendOpAlpha = ADD;
    RenderTargetWriteMask[0] = 0x0F;

};
```

Ed ecco il risultato finale



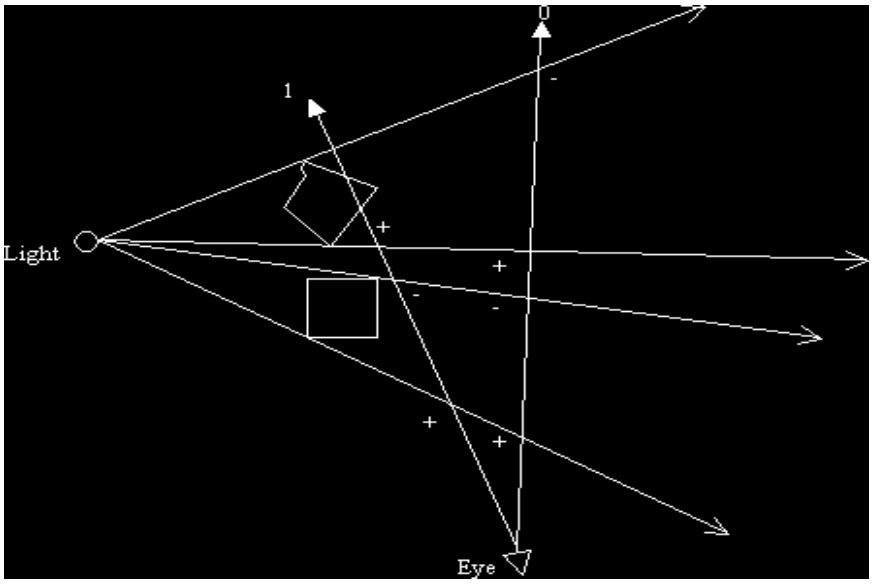
Guardate quest’altra immagine: notate i pixel sulla sfera.



Sembra che questo effetto sia dovuto al numero basso dei poligoni della sfera, determinando una pixellosità non desiderata.
Si può risolvere comunque, con un subdivision surface momentaneo (sempre se siete in grado di farlo). Anche nella documentazione di DirectX potete trovare un esempio di questo effetto "collaterale".
Il blending comunque, se usato nel modo giusto, riduce questo effetto di molto. (Questo va impostato nel pass del quadrato)

Se non riuscite a vedere l'ombra, provate ad aumentare il valore di extruding, magari non riesce a raggiungere la vostra superficie.

La tecnica funziona anche per piu' ombre, lo potete capire da qui.



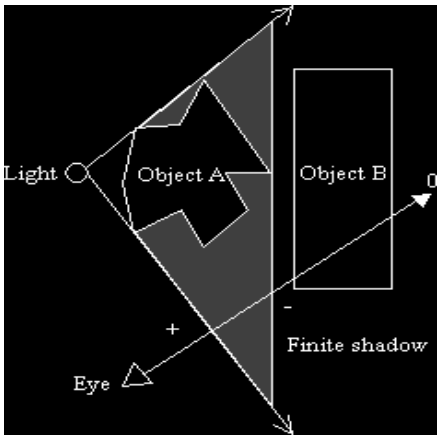
Tutti i valori tra + e – saranno ombrati.

C'è da fare però un altro appunto (anche se dovrebbe essere implicito).
Matematicamente, l'illuminazione in un punto è la somma di vari fattori. Avrete notato, nel mio precedente tutorial, nell'equazione della luce Phong, che la luce è la somma dei componenti ambient, emissive, diffuse e specular. Una scena con piu' luci ha un unico termine di ambient e emissive, ma per ogni luce c'è un termine diffuse e specular. Nel rendering senza ombre possiamo, con uno shader apposito, renderizzare tutte le luci in un unico pass.

Nel rendering con le ombre, invece, il contributo da una luce è zero in alcuni punti perché questi ultimi sono ovviamente ombrati. Per fare questo, il componente diffuse e specular deve essere calcolato, per ogni luce, in un pass separato, unendo poi tutti i risultati assieme all'emissive e ambient.

Da qui potete rendervi conto da soli, facendo prove con varie sorgenti di luce, che dopo un certo numero di luci il vostro software comincerà a rallentare inesorabilmente, segno che lo shadow volume, a quel punto, non è piu' indicato per rende rizzare le ombre della vostra scena.

Vediamo ora brevemente cos'è l'estrusione infinita.
Uno shadow volume infinito può essere utile quando la sorgente di luce è davvero troppo vicina ad un oggetto, come nella situazione descritta qui sotto



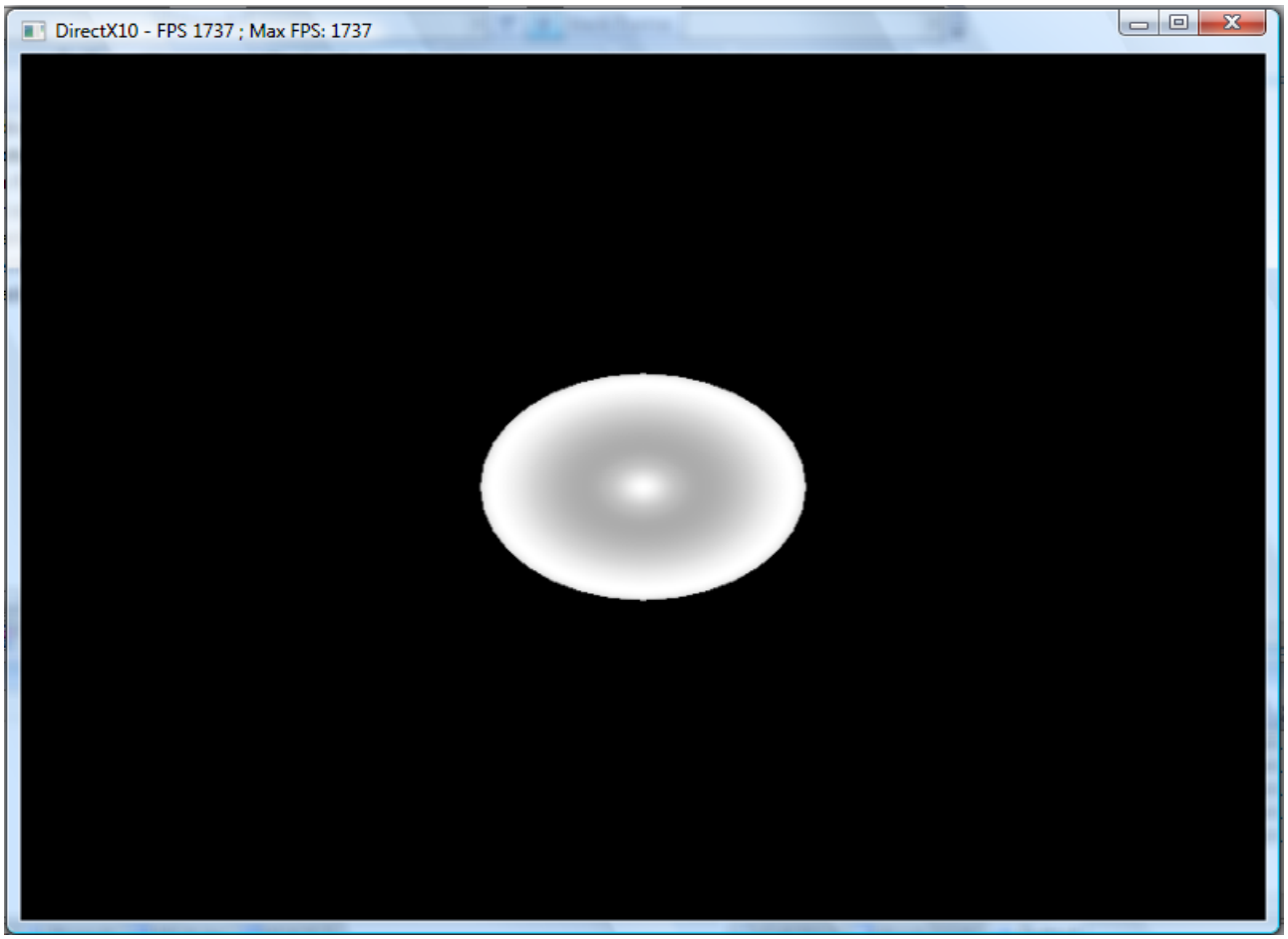
In questo scenario, l'eccessiva vicinanza tra luce e oggetto impedisce la creazione di un'ombra decente, ed è qui che è utile usare un valore infinito di estrusione. Inoltre risolve qualsiasi problema di distanza con un oggetto su cui deve andare a finire l'ombra (senza dover giocare continuamente con il valore di estrusione per far toccare l'ombra su sull'oggetto desiderato)

L'estrusione infinita è un argomento a parte, posso accennarvi che per fare un'estrusione infinita basta considerare i punti non come lati, ma come vettori. Ciò si ottiene sostituendo il valore w della coordinata da 1 a 0 nella creazione del buffer contenente i vertici, ma modificando opportunamente la matrice Projection per avere un far plane infinito, che si

ottiene calcolando il limite all'infinito del valore far.
Comunque, per le vostre prime applicazioni, non dovreste averne bisogno.

Depth Fail

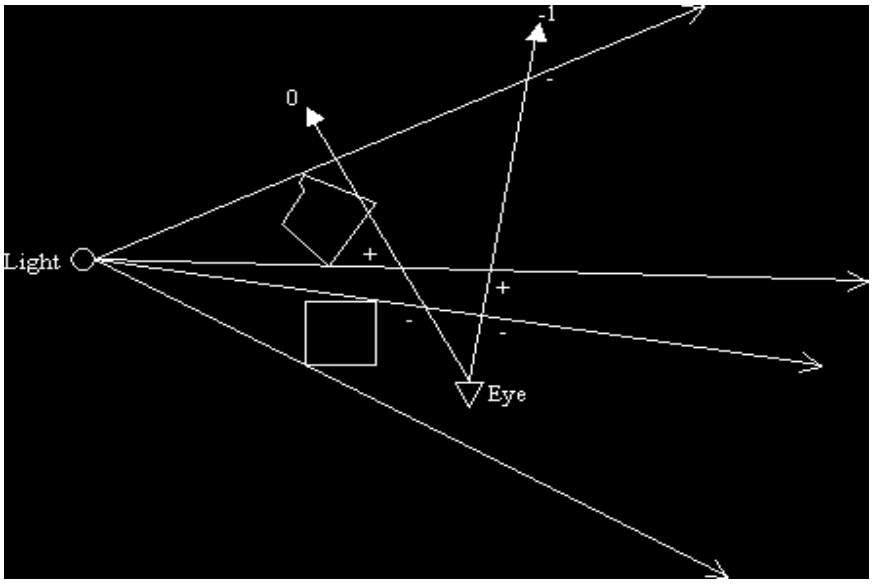
Guardate ora cosa succede però, se il cono d'ombra si trova nella telecamera



E' un effetto indesiderato. Questo accade perché il conteggio dei poligoni del culling in front face e back face sfasa (la telecamera, esseno all'interno del cono, non trova nessun poligono in back face) e quindi tutti gli oggetti in ombra diventano illuminati, e quelli illuminati diventano proprio neri.

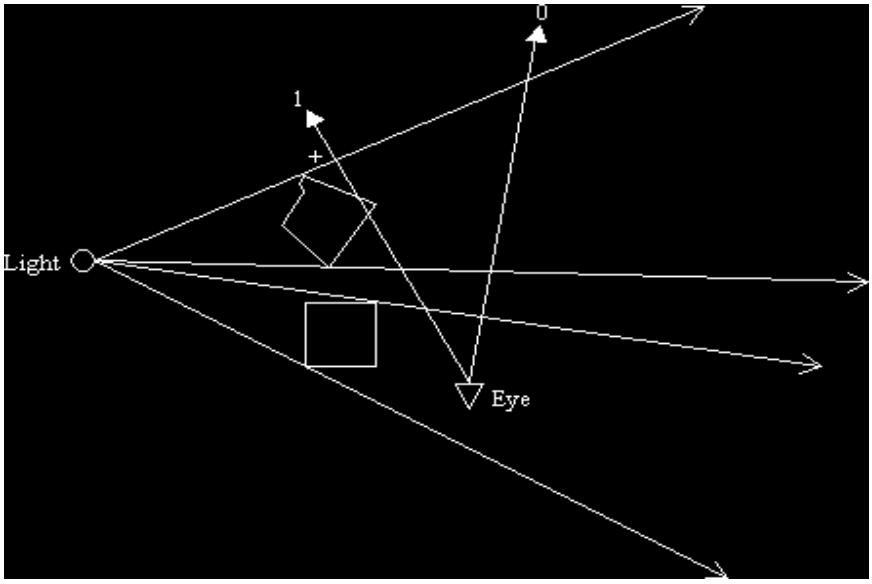
Il blending, sempre se usato in modo saggio riduce di molto anche questo effetto, ma non è comunque una soluzione valida.
Un problema di questo tipo comprometterebbe l'esperienza di gioco.

Capiamo dal punto di vista teorico



Sappiamo che la zona ombrata è quella tra + e -, quindi noteremo una totale inversione di zone luce ed ombra.

Fu così che vari cervelloni si craccarono la testa alla ricerca di un altro tipo di algoritmo di utilizzo dello stencil.
Il duro lavoro diede vita al depthfail, che ragiona in modo praticamente opposto al depthpass.



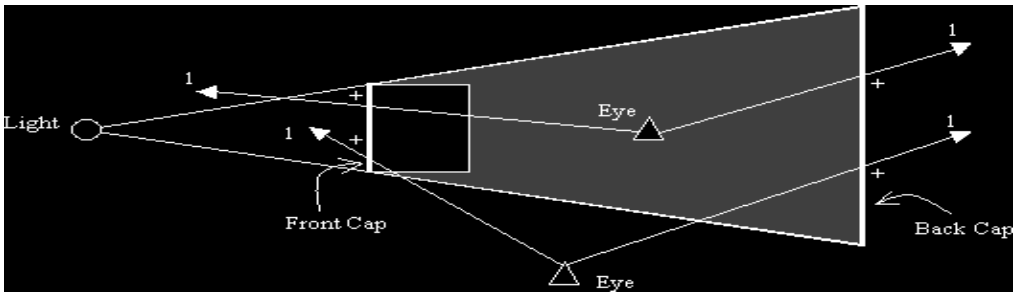
Semplicemente effettua le operazioni quando il depthtest fallisce. Cosi, dall'immagine sopra, il primo lato (a partire dalla telecamera) supera il test, il secondo lo fallisce, il terzo lo supera. Così, non c'è ombra tra il lato 2 e 3.

Se ci ragionate sopra, il DepthFail, in fondo, non fa altro che negare il contrario del DepthPass (pensateci bene). Dato che 2 negazioni affermano,il risultato alla fine è uguale,ma il DepthFail è immune, se lo shadow volume ha dei poligoni di capping (vedremo come costruirli a breve), al problema che invece affligge la tecnica precedente.

In linea di massima, il DepthFail funziona anche fuori dal cono d'ombra, ma comunque in alcuni casi non va bene; mettetevi in testa, insomma, che la tecnica perfetta, con lo shadow volume, non esiste, ma è possibile fare un mix delle 2 per creare ciò che nVidia chiama "Robust Shadow Volume Rendering"

C'è un altro punto da risolvere: il capping del poligoni, di cui abbiamo parlato anche prima (Light Cap e DarkCap nelle immagini delle prime pagine)

La tecnica del DeptFail si basa sul fallimento del DepthTest; è dunque necessario ricreare condizioni che nella semplice estrusione dei poligoni non sempre avviene. O meglio, se non ricreiamo delle front e back faces adeguate, il DethFail al 99% fallirà. Possiamo migliorare la situazione di molto effettuando il capping dei poligoni, ossia chiudere lo shadow volume.



I lati in grassetto bianco sono il nostro obiettivo.
Attezione, fare il capping dei poligoni è necessario soltanto se stiamo usando il depth fail. Aggiungiamo semplicemente alcuni poligoni che sono necessari per innescare il fail iniziale dell'ombra. Se state usando il depth pass, è un inutile spreco di risorse.

Per fare il capping dei poligoni, è sufficiente aggiungere, se il dot della luce è > 0, i vertici del triangolo originale, e la loro stessa versione estrusa e negativa. Quindi, il nostro geometry shader definitivo sarà questo

```
[maxvertexcount(24)]
void gs_shadow(triangleadj VS_INPUT input[6],inout TriangleStream<PS_INPUT> Stream)
{

	float3 TriNormal = GetNormal( input[0].Pos.xyz, input[2].Pos.xyz, input[4].Pos.xyz );

	if( dot( TriNormal, normalize(LightDirection.xyz) ) > 0 )
	{
		PS_INPUT output = (PS_INPUT)0;

		output.Pos = mul(mul(input[0].Pos,ViewMatrix),ProjMatrix);
		Stream.Append(output);
		output.Pos = mul(mul(input[2].Pos,ViewMatrix),ProjMatrix);
		Stream.Append(output);
		output.Pos = mul(mul(input[4].Pos,ViewMatrix),ProjMatrix);
		Stream.Append(output);

		Stream.RestartStrip();

		output.Pos = -mul(mul(input[4].Pos + ExtrudeValue *
float4(normalize(LightDirection.xyz),0),ViewMatrix),ProjMatrix);
		Stream.Append(output);
		output.Pos = -mul(mul(input[2].Pos + ExtrudeValue *
float4(normalize(LightDirection.xyz),0),ViewMatrix),ProjMatrix);
		Stream.Append(output);
		output.Pos = -mul(mul(input[0].Pos + ExtrudeValue *
float4(normalize(LightDirection.xyz),0),ViewMatrix),ProjMatrix);
		Stream.Append(output);

		Stream.RestartStrip();



		for( uint i=0; i<6; i+=2 )
		{
			uint iNextTri = (i+2)%6;

			float3 AdjTriNormal = GetNormal( input[i].Pos.xyz, input[i+1].Pos.xyz, input[
iNextTri ].Pos.xyz );

			if ( dot( AdjTriNormal, normalize(LightDirection.xyz) ) <= 0 )
			{
				float4 v0 = input[i].Pos;
				output.Pos = mul(mul(v0,ViewMatrix),ProjMatrix);
				Stream.Append(output);

				float4 v1 = v0 + ExtrudeValue *
float4(normalize(LightDirection.xyz),0);
				output.Pos = mul(mul(v1,ViewMatrix),ProjMatrix);
				Stream.Append(output);

				float4 v2 = input[iNextTri].Pos;
				output.Pos = mul(mul(v2,ViewMatrix),ProjMatrix);
				Stream.Append(output);
```

```
                                float4 v3 = v2 + ExtrudeValue *
float4(normalize(LightDirection.xyz),0);
                                output.Pos = mul(mul(v3,ViewMatrix),ProjMatrix);
                                Stream.Append(output);

                                Stream.RestartStrip();

                        }
                }
        }
}
```

A ciò seguirà un adeguato setting del solito DepthStencilState, per aumentare nel BackFaceStencilDepthFail, e diminuire nel FrontFaceStencilDepthFail.

Il capping dei poligoni comunque è una tecnica ultra superata. In D3D10 veramente è "vecchia".
Oggi si preferisce utilizzare il Clamp, supportato fin dalle vecchie nVidia 2X e finalmente sta diventando una funzionalità realmente standard nelle schede video.
Quest'ultimo viene settato nel RasterizerDesc: basta mettere a false il valore DepthClipEnable e attivare il rasterizer state: in questo modo potete tranquillamente evitare il capping dei poligoni. Nel rarissimo caso (non una scheda D3D10, comunque) l'ombra venisse mal renderizzata,potrebbe essere un problema di compatibilità con il depth clip enable.

Come verificare se "fisicamente" l'ombra è stata creata bene? (Fisicamente si intende se rispetta le leggi fisiche che l'accompagnano). Basta fare una piccola modifica: posizionate la telecamera nel punto in cui inizia la luce (se luce puntiforme) oppure in un punto che appartiene alla retta DirezioneLuce – Baricentro della Mesh.
Avviate il vostro software: se NON vedete nessun'ombra, allora la creazione è avvenuta correttamente. Fate ruotare prima la telecamera assieme alla luce e assicuratevi di non vedere mai l'ombra dell'oggetto. Dopo di che fate ruotare solo la telecamera: dovreste avere un effetto di aumento progressivo dell'ombra, fino ad avere un massimo e un minimo d'ombra. Se tutte queste condizioni avvengono, l'ombra è stata creata correttamente.

Ma qual è il merito di Carmack in questa vicenda?
La tecnica del Depth Fail fu scoperta da William Bilodeau e Michael Songy nell'ottobre del 1998 e la presentarono ad una conferenza della Creative Labs. Si fecero anche "proteggere" da eventuali copioni, presentando richiesta di certificazione per una tecnica di rendering delle ombre usando shadow volumes e stencil buffer.
Ad ogni modo, Carmack, ovviamente ignaro del lavoro dei due ragazzi di sopra, scoprì indipendentemente l'algoritmo nel 2000, durante lo sviluppo di Doom 3.
Proprio perché è stato Carmack a mostrarne le potenzialità nel suo gioco e soprattutto mostrò la tecnica al vasto pubblico, oggi viene chiamato in suo onore Carmack Reverse.
Probabilmente, senza Carmack non avremmo nemmeno mai saputo l'esistenza di questa tecnica.

Conclusa anche la parte storica, è d'obbligo fare qualche considerazione generale sull'uso di questa tecnica, perché, al di là delle difficoltà (che non sono poche), è tutt'altro che rose e fiori.

In primo luogo, DepthFail e DepthPass hanno i loro difetti, come comportarsi a riguardo? E' necessario creare una tavola di scelte secondo la quale usare l'una o l'altra tecnica (come ho detto sopra, nVidia definisce questo approccio "Robust Shadow Volume"); è da tener conto, comunque, che la generazione del volume richiede tempo (anche se il geometry shader è abbastanza rapido perché accolla il carico alla GPU), tempo che aumenta nel DepthFail per generare il capping (che in alcuni casi può essere anche molto oneroso).
La situazione è in parte risolvibile se le ombre non si muovono: se queste infatti sono statiche è possibile non solo precalcolarle una sola volta, ma renderizzarle anche tutte assieme (tanto il pixel shader che le colora è uguale per tutte)tramite lo stream output. In questo modo si possono avere significativi incrementi di velocità(30-50%).
Altrimenti è possibile generare lo shadow volume da una versione della geometria con meno poligoni, in modo da caricare meno il processore.
Insomma, cercate di ridurre al minimo i vertici, massimizzate l'uso dell'Index Buffer anche con valori di Epsilon esagerati. L'ombra è un fatto estetico, ma può essere anche non abbastanza preciso.

Altro problema che si pone, è se la zona è ricca di punti che fanno ombra. Prendete ad esempio una scena composta da una stanza piena di pilastri e con multiple sorgenti di luce. Per ogni pilastro sarà necessario generare svariati coni d'ombra, e ciò può distruggere la scheda video piu' potente (attualmente) in circolazione.

Inoltre, come avete sicuramente visto nelle immagini, l'ombra, anche se nelle scene si ferma nel muro, continua in realtà per tutto il valore dell'estrusione, e ciò può provocare notevoli problemi se è necessario continuare a disegnare l'ombra da dietro un muro.

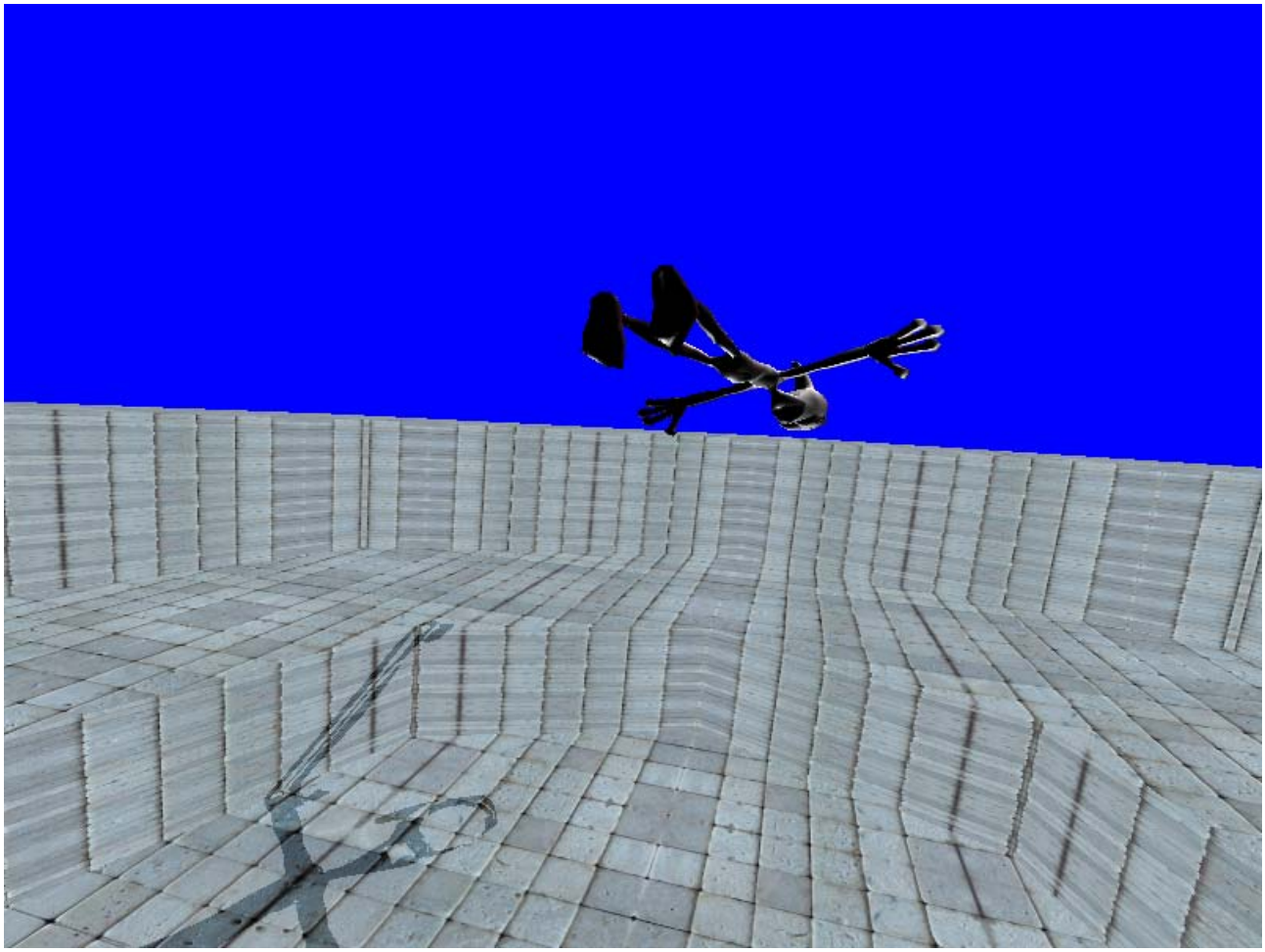
Unendo tutti questi piccoli consigli, lo shadow volume diventa una tecnica valida da usare in real time (ricordiamo l'esperienza di Doom3, in cui le ombre in stencil venivano fatte interamente da CPU uccidendo letteralmente il

processore). In particolare, un gioco in visuale top (come i giochi di calcio, che si vedono dall'alto) sono ottimi per lo shadow volume: il limitato numero di poligoni, il dover disegnare al massimo 22 ombre in un campo suggerisce l'utilizzo di questa tecnica.

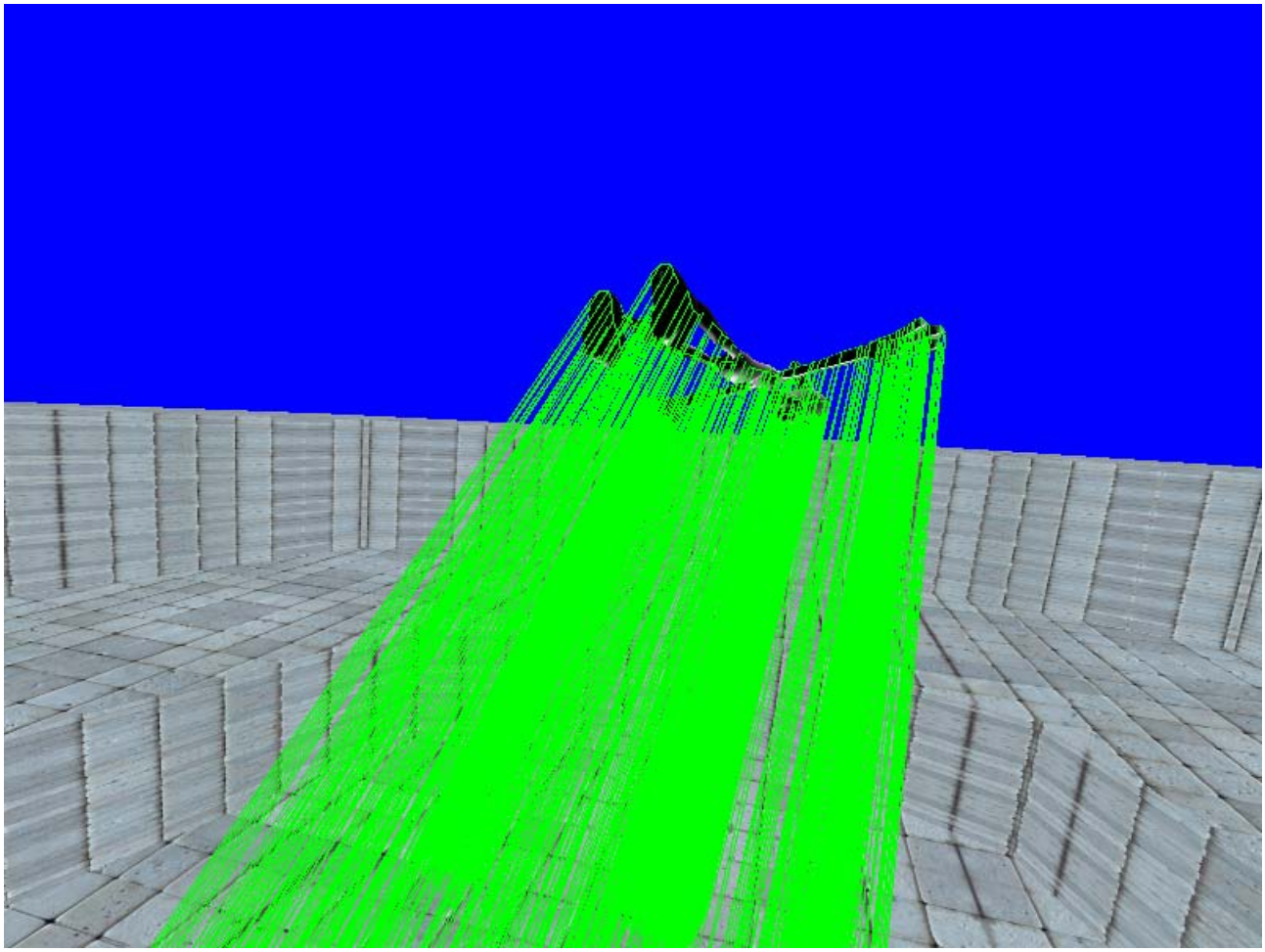
Un altro problema dovuto all'uso di questa tecnica si manifesta quando si renerizzano oggetti che hanno una sola faccia adiacente (e non 2, come vuole ogni triangolo). Questo effetto è chiamato Crack. Nel mondo reale, se guardi in un crack, vedi l'interno dell'oggetto. Ma, nel 3D Rendering, il risultato sarà una visione attravero l'oggetto e oltre di esso perché ovviamente ci sono varie faccende di culling. Ora, da certe angolazioni l'oggetto viene visto bene, da altre no. Nel peggiore dei casi, l'oggetto potrebbe risultare in ombra in questo strano modo.

Lasciando evidenti tracce bianche.

Io ho avuto lo stesso problema con l'alieno che ho usato negli esempi (perciò dopo ho usato la classica sfera), pensando di aver evidentemente sbagliato a calcolare i contorni. Invece erano fatti bene.
Con un po' di ottimizzazione (e l'aiuto di D3DX), sono riuscito ad ovviare, almeno in parte, a questo problema.


Il problema del crack: notate la faccia bucata dell’ombra

Lo shadow volume, però, è una tecnica che produce ombre per pixel molto accurate e precise, e funziona sia con le luci puntiformi, direzionali e spot. Comparato con gli altri algoritmi, lo shadow volume ha il vantaggio di funzionare nella maggior parte dei casi (anche i piu’ estremi: tutto in ombra e solo un piccolo pezzo in luce) mantenendo comunque un’alta qualità (qualità massima se si estrudono i vertici dalla geometria originale).


Esempio di shadow volume complesso

Ovviamente non tutte le scene vanno bene per essere rende rizzate con lo shadow volume: il crack, come avete visto, è uno dei problemi in cui si può incappare, ed inoltre, dato che l'ombra è creata a partire dalla geometria 3D iniziale, può essere non molto fedele nel caso di oggetti non perfettamente rappresentati dalle proprie mesh (per esempio i particle system, i billboard).

A ciò si aggiunge il fatto che la tecnica è stata ampiamente superata.
Insomma, fate vobis.

Spero vivamente di avervi aiutato nella comprensione del funzionamento dello shadow volume, ma se non avete capito qualcosa, potete inviare un commento o una mail

Vincenzo Chianese

http://xvincentx.netsons.org/programBlog