Shadow Mapping in D3D10

In questo corposo tutorial cercherò di spiegare, principalmente, il famoso algoritmo di Shadow mapping tuttora usato (con molte migliorie) nei giochi attuali. Implementeremo poi una variante importante (Variance Shadow Mapping) con una Point e Spot Light (di cui darò le necessarie spiegazioni nel corso del tutorial). Infine, vi spiegherò come funzionano gli shader a basso livello, ossia cosa c'è dietro le classi ID3D10Effect (eviteremo quindi D3DX fino all'osso).

Le richieste grafiche richieste sono poche ma devono essere ben consolidate, in particolare

1. Cube Map NON centrate nell'origine(dovete saperle fare bene, non in modo arronzato)
2. Geometry Shader
3. Render Targets (render to quad per fare postprocessing)
4. Texture Projection

Servirà pazienza a volontà per evitare di usare gli Effect.

Darò anche qualche accenno su come realizzare il Gaussian Blur necessario per completare lo shadow mapping, ma è comunque suggerita una lettura più approfondita (potete cominciare da qui http://www.gamedev.net/reference/articles/article2007.asp)

Shadow mapping o projective shadowing è un processo con il quale vengono calcolate e renderizzate le ombre nella computer grafica 3D. Il concetto è piuttosto vecchio e risale ad un articolo di Lance Williams del 1978 intitolato "Casting curved shadows on curved surfaces" (trad.: 'gettare ombre curve su superfici curve'). Da allora la tecnica viene utilizzata sia per i rendering pre-calcolati che per le scene realtime, soprattutto nei videogiochi. Lo shadow mapping viene utilizzato anche in produzioni cinematografiche, e fa parte dell'engine RenderMan della Pixar che lo ha utilizzato in film come Toy Story.

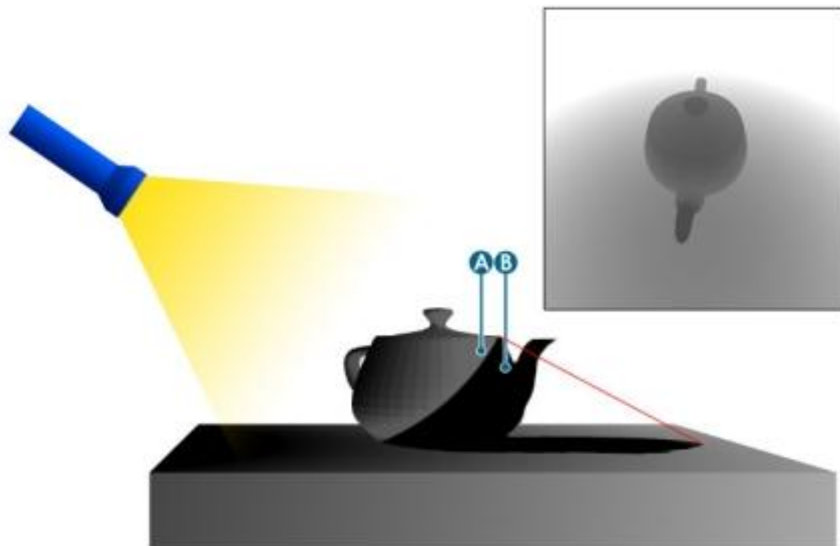
Le ombre sono create verificando se un frammento sia o meno visibile dalla sorgente di luce (preferibilmente una spot light); proiettando questa informazione nello spazio di vista della camera (utilizzando una apposita texture) e confrontando le "profondità" si può verificare se un punto sia o meno illuminato e colorarlo di conseguenza.

Qualunque oggetto visibile da una sorgente di luce appare illuminato. Qualunque cosa dietro tali oggetti sarà, necessariamente, in ombra. Questo è il principio di base dello shadow mapping. Si renderizza quindi la vista della luce disattivando tutte le computazioni non necessarie, visto che in questa fase siamo interessati unicamente al buffer di profondità (la componente z) e non ai colori.

Successivamente si renderizza la scena normalmente, confrontando la profondità di ogni punto con la proiezione della shadow map nello spazio di vista della camera: se un punto è "dietro" la shadow map, allora è in ombra.

La tecnica di shadow mapping è meno accurata dei volumi d'ombra o shadow volumes, ma risulta un'alternativa notevolmente più veloce e più semplice da implementare. Inoltre, non richiede buffer speciali (come lo stencil buffer), può essere modificata per simulare ombre morbide (con il percentage closer filtering) e non crea problemi complessi come il calcolo delle intersezioni dei volumi e della loro "terminazione". Tuttavia, la qualità delle

ombre create con lo shadow mapping è legata alla risoluzione della mappa di ombre stessa e risulta prona ad artefatti (soprattutto quando i frustum di luce e camera hanno stessa direzione ma verso opposto).



L'immagine qui sopra mostra in modo efficace il procedimento dell'algoritmo.
Supponiamo una sorgente di luce Spotlight che illumina la teiera in questione. Nel riquadro c'è il rendering della scena dalla posizione della luce, memorizzando solo le informazioni di profondità.
Per questo motivo, ogni pixel della parte invisibile della teiera avrà una profondità maggiore, che significa rendering in ombra.

Prima di implementare l'algoritmo nella pratica e spiegare i concetti che ci serviranno, apro qui una grossa parentesi su D3D10 e il sistema Shader ed Effect. Per i deboli di cuore saltate questo paragrafo.

D3D10 ha portato alla nascita dello Shader Model 4.0 che, oltre a novità dal punto di vista delle "features" ha introdotto i constant buffer, tanto utili quanto ostici.
Le classi ID3D10Effect mascherano questo in modo abbastanza buono, esponendo una interfaccia molto simile a quella usata in D3D9.
Il problema è che (come sottolineato al GDC piu' volte) semplici porting di shader da D3D9 a D3D10 porterà solo a cattive performances.
E' quindi necessario adattarsi quanto prima ai constant buffer.
Questi ultimi alla fine non sono altro che delle strutture (pari pari a quelle del C++) nei quali verranno inseriti i dati da usare negli shaders. La loro dichiarazione, quindi è scontata.

```
cbuffer cMultiFrame
{
	float4x4  WorldMatrix;
	float4    OtherValues;
	float4    Color;
};
```

L’utilizzo di questi valori negli shader è ancor piu’ trasparente, poiché verranno trattati come dati globali (quindi World Matrix potrà essere usato senza dover referenziare il suo constant buffer di appartenenza), quindi proprio come facevate in D3D9.

Premettendo che in D3D10 i valori negli shader si passano SOLO usando constant buffer, perché li hanno introdotti?
In linea di massima, per ridurre la banda di dati da inviare in upload alla GPU per ogni frame. Supponiamo di avere una immensa dichiarazione di dati (10 float4, 3 matrici, 10 boolean e un array di strutture) e di voler cambiare solo il valore booleano. Questo causa, per ragioni costruttive, il reinvio alla GPU di tutto il materiale (compresi i float, l’array di strutture e le matrici), sprecando un’assurdità di tempo per avere un cambio minimo. I constant buffer consentono invece di inviare alla GPU solo i dati strettamente necessari. Ad esempio, se abbiamo che ad ogni frame dobbiamo aggiornare solo le tre matrici, mentre il resto rimane invariato, è un’ottima idea racchiude il tutto in 2 constant buffer

```
cbuffer cMultiFrame
{
      row_major float4x4      WorldMatrix;
      row_major float4x4      ViewMatrix;
      row_major float4x4      ProjMatrix;
};

cbuffer cOther
{
      //qualsiasi altra cosa vi pare.
};
```

In questo modo, al nuovo Draw, l’upload alla GPU sarà limitato alle sole 3 matrici, mentre il resto, essendo invariato, rimane nella memoria della GPU.

I constant buffer vengono gestiti dal codice dell’applicazione attraverso un generico buffer.

```
struct
{
      D3DXMATRIX  WorldMatrix;
      D3DXMATRIX  ViewMatrix;
      D3DXMATRIX  ProjMatrix;
} cMultiFrame;

  D3D10_BUFFER_DESC cbDesc;
    cbDesc.ByteWidth = sizeof( cMultiFrame );
    cbDesc.Usage = D3D10_USAGE_DEFAULT;
    cbDesc.BindFlags = D3D10_BIND_CONSTANT_BUFFER;
    cbDesc.CPUAccessFlags = 0;
    cbDesc.MiscFlags = 0;
    hr = g_pd3dDevice->CreateBuffer( &cbDesc, &cMultiFrame, &g_pConstantBuffer10
);
    if( FAILED( hr ) )
       return hr;

//Cambio dei dati
D3DXMatrixIdentity(&cMultiFrame.WorldMatrix);
//Aggiorno il buffer
g_pd3dDevice->UpdateSubresource(&g_pConstantBuffer10,0,NULL,&cMultiFrame,0,0);
```

Ricordiamo alcune regole fondamentali:

- I constant buffer hanno il limite di 4096 float4 (se andate oltre ci sono i texture buffer).
- Non andare mai oltre i 5 cbuffer (diventano troppo difficili da gestire per la GPU)
- Il miglior modo di organizzare i constant buffer è per frequenza di aggiornamento (valori che cambiano piu' di una volta per frame, valori che cambiano una volta per frame…)
- cbuffer con un solo parametro sono da usare solo se effettivamente tale valore cambia in contesti così strani che è meglio che stia da solo
- I float2,3,4 mappateli con D3DXVECTOR2-3-4
- Le regole di aggiornamento dei buffer generici NON valgono per i cbuffers: anche se li aggiornate piu' di una volta a frame, NON usate D3D10_USAGE_DYNAMIC per fare un Map via CPU. Usate UpdateSubresource se avete tutto il cbuffer in un blocco (quasi sempre), Map se invece dovete "raccogliere" il buffer da varie aree di memoria sparse (se non avete capito meglio chiudere il tutorial).
- Per aumentare le performance (tramite il riuso della cache della GPU), mettete gli elementi in ordine di utilizzo nello shader

D: se l'unico modo di settare i dati è tramite i constant buffer, perché nei miei vecchi shader D3D10 non li creavo?
In realtà D3D10 racchiude tutti i dati che non sono in un constant buffer in un constant buffer chiamato $global.
D: Perché hai messo row_major alle matrici?
Il constant buffer è comunque una serie di bit inviati via GPU. E' quindi necessario specificare sempre in che modo vuoi che sia letta la matrice. Se si usa ID3D10Effect, D3DX farà questo per te,altrimenti dovrai specificarlo esplicitamente D3D10, per altro, legge di default le matrici al contrario (column_major). Perché? Leggere le matrici in column_major è piu' veloce perché permette di fare la moltiplicazione vettore-matrice come una serie di dot products.
D: Cosa significa row_major e column_major?

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Lettura elementi in row_major: 1 2 3 4 5 6
Lettura elementi in column_major: 1 4 2 5 3 6

Vediamo ora come funziona veramente uno shader.
Solitamente uno shader è formato da

1. Dichiarazioni cbuffer
2. Dichiarazioni texture
3. Dichiarazione Samplers
4. Rasterizer, Blend e DepthStencil state
5. Struct di input output
6. Vertex Shaders
7. Pixel Shaders
8. Geometry Shaders
9. Technique con Pass

Il VERO shader è formato soltanto dai punti 1-2-3-5-6-7. Gli altri punti sono delle scritture formali usate soltanto da ID3D10Effect per aiutarvi nella scrittura degli shader (che in realtà NON esistono!).

Quando parsate un file tramite un effect, questo

1. Apre il file e legge i dati
2. Trova tutti i depth, rasterizer e blend state, li parsa e li crea in C++ mantenedosi i puntatori
3. Allo stesso modo parsa e crea cbuffer e texture buffers.
4. Parsa tutti i pass costruendo una tabella di puntatori da usare nell'apply, secondo le vostre specifiche.

Quindi, creare gli states negli shader NON velocizza il codice, vengono sempre e comunque creati dalla CPU e settati al momento opportuno (potete verificare con Pix)

Una volta fatto tutto ciò manualmente (ci siete riusciti?), dovete compilare effettivamente gli shader. Per fare ciò c'è D3D10CompileShader che, dati i necessari parametri, darà un puntatore al byte code da usare nella funzione apposita(Create Pixel, Vertex o Geometry shader)

Piccola nota interessante: D3D10CompileShader utilizza il compilatore HLSL che viene fornito con Windows Vista. Quindi si, quello del lontano 2006 che, oltre ad avere vari problemi interni (io ne ho segnalato 1 alla Microsoft, in quanto il compilatore non riusciva a supportare 2 if di seguito), non supporta D3D10_SHADER_DEBUG e non è ottimizzato al massimo.
Per usare quello "attuale", dovrete usare D3DX10CompileShaderFromMemory, oppure compilare gli shader usando FXC e importarli poi manualmente.
Comunque, parlando con loro, non c'è intenzione di aggiornare il compilatore di Windows Vista.

Il device offre delle funzioni per inserire e ricevere dati dagli shader. Queste fondamentalmente sono

```
XSSetShader               -> Inserisce uno shader nel device
XSSetConstantBuffer       -> constant buffer
XSSetShaderResourceView   -> texture
XSSetSamplerState         -> sampler state
```

Dove X può essere V/S/G.
In effetti, se uno stesso constant buffer viene usato sia dal pixel che dal vertex shader, questo dovrà essere "bindato" sia al vertex e al pixel shader chiamando le relative funzioni.

Veniamo ora a una nota dolente, gli slot delle risorse.
Lo shader ha un numero di slot nei quali si può effettuare il bind di buffer, texture e samplers. La determinazione di questi ultimi è leggermente ostica, e può impedire di fare un adeguato debug.
Alla compilazione dello shader, HLSL determina il numero degli slot in modo cardinale in base all'utilizzo delle risorse. Per farla breve, con questo codice

```
cbuffer cMultiFrame
{
      row_major float4x4      WorldMatrix;
      row_major float4x4      ViewMatrix;
      row_major float4x4      ProjMatrix;
};
```

```
cbuffer cOther
{
      float4 other;
};

PS_INPUT vs_main(VS_INPUT in)
{
      PS_INPUT i;
      i.Pos = mul(in.Pos,WorldMatrix);

      return i + other;
}
```

Il constant buffer cMultiFrame avrà lo slot 1 (perché utilizzato per primo), mentre cOther si troverà nello slot 1.
Consideriamo il vertex shader di sopra accoppiato a questo Pixel Shader

```
float4 ps_main(PS_INPUT i)
{
 return othervalue + WorldMatrix._m11 * Text.Sample(SamplText,In.Tex);
}
```

La texture Text si troverà nello slot 1 degli shader resource. Il sampler nella stessa situazione, mentre come constant buffer, la situazione di prima sarà invertita, perché invertito è l'ordine di utilizzo.
L'ordine degli slot cambia inoltre se ci sono dei return incondizionati oppure se, valori presi precedentemente da texture non vengono mai usati.
Consideriamo questo pixel shader

```
PS_INPUT vs_main(VS_INPUT in)
{
      PS_INPUT i;
      float4 color = Text.Sample(SamplText,In.Tex);
      //altro codice con altri sample da diverse texture
      return i + code(x) + sampled;
}
```

Di primo avviso verrebbe da dire che lo slot 1 è dato dalla Texture Text. Invece non è così. Poiché questa texture è stata campionata, ma non compare in nessun calcolo utile, lo slot non viene mai usato e quindi tutto viene slittato di un posto. E' un GRANDISSIMO problema nel debug degli shader, quando si vogliono ritornare alcuni valori senza usarne altri. L'unico modo è usare in modo inutile quel valore.

```
min(color,float4(0,0,0,0));
```

Ritornerà 0, che possiamo aggiungere a quello che poi ci serve per davvero.

Passiamo ad un'altra nota ancor piu' pesante: il packing dei dati.
HLSL lavora a botte di float4 o meno. Inserisce quindi valori inutili per riempire i vuoti dei constant buffer. Gli array inoltre, vanno a 16 byte alla volta.

Dunque in primis, nel dichiarare le strutture da usare come constant buffer in C++, sarà bene attivare pragma pack 4 (e poi riportarlo a 8).
Vediamo alcuni esempi che chiariranno il problema

```
cbuffer cMultiFrame
```

```
{
	row_major float4x4	WorldMatrix;
	row_major float4x4	ViewMatrix;
	row_major float4x4	ProjMatrix;
};
```

Questo constant buffer è “perfetto”, poiché rispetta i multipli di 4 float alla volta.

```
cbuffer buf
{
	float2 data;
	float4 data2;
};
```

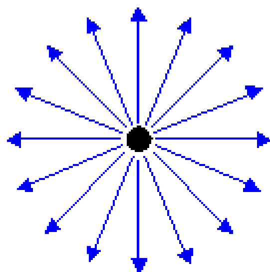
Questo constant buffer ha dei problemi: dopo i primi 2 float data, poiché HLSL effettua il packing 4 alla volta inserirà 2 float dummy per riempire il vettore da 4, e poi inserirà il prossimo float4-
In questo modo però, le dimensioni passano da 24 byte a 32, poiché effettivamente la nostra struttura sarà stata così trasformata.

```
cbuffer buf
{
	float4 data;
	float4 data2;
};
```

Il problema è evidente nel dover rifare una struttura uguale in C++ per settare i dati in modo corretto.
In questo caso la soluzione è semplice, basta invertire i 2 membri, ma ci saranno situazioni in cui sarà ardua tenere il buffer con offset puliti e corretti.
Il metodo che tutt’ora funziona è cercare di raggruppare i dati quanto più possibile in float4, lasciando all’ultimo i dati con offset sbagliati. Al limite inseriremo valori dummy anche nella struttura in C++ per eguagliare il buffer coerentemente.
Interessante anche la parola packoffset, che serve a decentrare i valori rispetto al loro usuale offset. Non ve ne spiegherò l’utilizzo perché è per scopi avanzati, potete leggere la documentazione per capirne qualcosa in piu’.

Finita questa grossa parentesi, prima di implementare l’algoritmo in modo definitivo, diamo uno sguardo veloce alla luce Spot e quella Point (la direzionale dovreste saperla già usare), perché sarà su queste che creeremo le shadow map.
Non mi dilungherò molto, sta a voi imparare a padroneggiarle.

La point light (luce puntiforme) è quella luce che irradia verso tutte le direzioni alla stessa intensità, con attenuazione che varia in ragione del quadrato della distanza. Una luce di questo tipo è la classica lampadina, o il lampione della strada.



Poiché la luce è proiettata in tutte le direzioni, la direzione effettiva di illuminazione verso un pixel va calcolata al momento, per poi usare le usuali equazioni di luce desiderate.
In codice shader, questo significa (usando il modello Lambertiano)

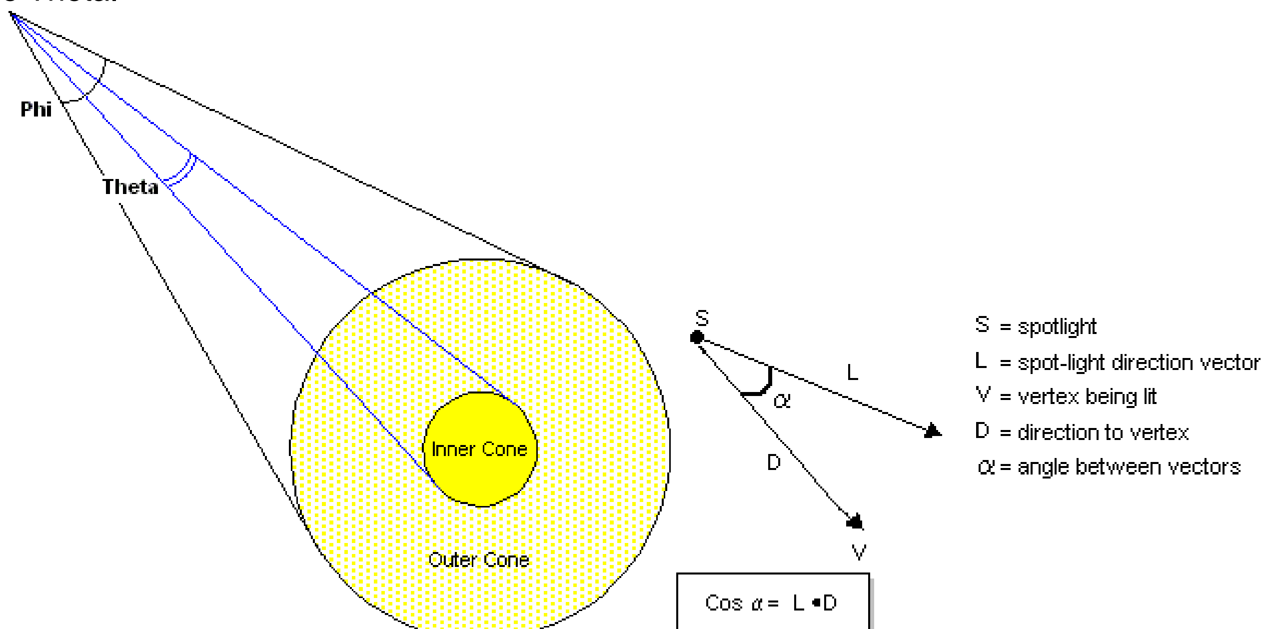
```
        float3 vLight = normalize(LightInfo.xyz-In.WPos);
        float NdotL = dot(In.Nor,vLight);
        float Att =
Attenuation(distance(LightInfo.xyz,In.WPos),0.003,0.003,0.003);
        float4 PointLight = (saturate(NdotL)) * Att;
```

L'attenuazione va calcolata semplicemente in questo modo standard

```
float Attenuation
    (
        float distance,
        float a,
        float b,
        float c
    )
{
    float Atten = 1.0f / ( a + b * distance + c * pow(distance,2) );
    return Atten;
}
```

C'è chi usa anche metodi propri, ma questo va bene per il 99% delle situazioni.

Mentre la Direction Light non ha il punto di partenza, e la Point Light non ha una direzione fissa, la Spot Light è un tipo di luce caratterizzata da Punto di inizio e direzione di illuminazione. E' tipicamente paragonabile ad una torcia o un faro. Quest'ultima è caratterizzata, oltre dalla sua attenuazione, da un falloff (che non spiego e non uso) e 2 angoli che determinano il cono di luce da creare. Questi vengono chiamati solitamente Phi e Theta.



Una spot light produce effettivamente (ho provato di persona, nemmeno io ci credevo all'inizio) un cono di luce più forte e uno più debole. Il primo avrà un certo angolo, il secondo un altro (che sceglieremo noi). Oltre un certo angolo non ci sarà piu' illuminazione da parte della spot light.
Dunque, per determinare se un pixel riceve luce da una spot light bisognerà calcolare l'angolo formato dalla direzione della spot light e dalla direzione Sorgente-Vertice. A seconda dell'angolo che ne verrà fuori (calcolabile tramite un dot product, previa normalizzazione dei vettori), ci comporteremo di conseguenza.
In codice, significa:

```
        float4 LightToVecDirection = normalize(LightInfo3 - In.WPos);
        float4 LightToSourceDirection = normalize(LightInfo3 - LightInfo2);
        float cosAlpha = dot(LightToVecDirection,LightToSourceDirection);

    float Atten = 0.0f;

    if( cosAlpha > Theta )
    {
        Atten = 1.0f;
    }
    else if( cosAlpha > Phi )
    {
        Atten = 1.0f;
    }
```

Non mi dilungherò oltre su quest'argomento, vi consiglio di fare prove su prove prima di continuare nella lettura-

Iniziamo ora a realizzare la nostra shadow map. La scena di riferimento (presa dal demo che è fornito col tutorial) sarà questa.

Che ho palesemente copiato da Riemers (i grafici scarseggiano).
Cominciamo a implementare la shadow map per una luce puntiforme.
Poiché questa, per definizione, proietta luce in tutte le direzioni, il suo angolo di illuminazione sarà 360 gradi. Per questo motivo saremo "costretti" ad usare una cubemap (6 texture) che ci permetterà di coprire tutte le possibili direzioni della luce. (Di fronte, dietro, a destra, a sinistra, in basso, in alto).
Cambieremo in primis la telecamera per posizionarla sul punto della luce, e anche la matrice di proiezione, per cambiare l'aspect ratio (invece di 4:3 sarà 1, poiché le texture, per poterle bindare ad uno shader, devono essere quadrate) e il FOV (la luce non ha lo stesso difetto dell'occhio, che ha un angolo limitato a 45 gradi).

Per avere la massima precisione, la texture che conterrà le informazioni di profondità verrà creata a 32 bit (potete fare i taccagni e usarne una da 16, ma poi farete i conti con gli artifacts).

Dalla texture create un render target (associato ad un inutile depth stencil view) e uno shader re source view per il bind.

```
      CD3D10_TEXTURE2D_DESC
desc(DXGI_FORMAT_R32_FLOAT,SMX,SMY,1,0,D3D10_BIND_SHADER_RESOURCE|D3D10_BIND_REN
DER_TARGET,D3D10_USAGE_DEFAULT,0,1,0,D3D10_RESOURCE_MISC_GENERATE_MIPS);
      ID3D10Texture2D *peppe;

      Device->CreateTexture2D(&desc,NULL,&peppe);
```

```
D3D10_RENDER_TARGET_VIEW_DESC f;
f.Format = desc.Format;
f.Texture2D.MipSlice = 0;
f.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2D;

Device->CreateRenderTargetView(peppe,&f,&SRT);

D3D10_SHADER_RESOURCE_VIEW_DESC d;
d.Format = desc.Format;
d.Texture2D.MipLevels = PowerOf(SMX);
d.Texture2D.MostDetailedMip = 0;
d.ViewDimension = D3D10_SRV_DIMENSION_TEXTURE2D;

Device->CreateShaderResourceView(peppe,&d,&SSV);

desc.Format = DXGI_FORMAT_D32_FLOAT;
desc.BindFlags = D3D10_BIND_DEPTH_STENCIL;
desc.MiscFlags = 0;
Device->CreateTexture2D(&desc,NULL,&peppe);

D3D10_DEPTH_STENCIL_VIEW_DESC ds;

ds.Format = DXGI_FORMAT_D32_FLOAT;
ds.Texture2D.MipSlice = 0;
ds.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2D;

Device->CreateDepthStencilView(peppe,&ds,&SDS);

peppe->Release();
```

SMX e SMY sono le dimensioni della shadow map (che dovranno coincidere). Attenzione ai MipLevels. Oltre a farli sempre generare perché salvano le performance, è bene generarne sempre il massimo numero. Poiché ogni livello ha dimensioni dimezzate rispetto al precedente, per sapere quanti mip levels generare basta trovare x tale che

```
2^x = Dimensione
```

La mia funzione PowerOf fa proprio questo: cerca in quale potenza di 2 si trova l'argomento.
Ovviamente, è sempre buona norma mantenere le dimensioni di una texture come potenza di 2.

D3D10 permette, comunque, di non usare un render target ma leggere direttamente il contenuto del depth stencil. Ciò è un pelo piu' difficile (la texture va creata come R32_TYPELESS per poter fare il bind in 2 diversi pipeline stages) ma può essere molto più veloce perché, se non c'è un render target attivato, le schede nVidia attivano lo SpeedZ (che è 2 volte piu' veloce). Anche ATI ha una cosa del genere, l' HyZ.

Ho preferito, comunque, non mostrare come farlo.

Le matrici saranno quindi fatte in questo modo:

```
    D3DXMatrixPerspectiveFovLH(&cNeverChange.LightProj,D3DXToRadian(90.0f),1.0
f,1.0f,200.0f);

    D3DXMatrixLookAtLH(&cNeverChange.view[0],&LPos,&(LPos+D3DXVECTOR3(1,0,0)),
&D3DXVECTOR3(0,1,0));
```

```
            D3DXMatrixLookAtLH(&cNeverChange.view[1],&LPos,&(LPos+D3DXVECTOR3(-
1,0,0)),&D3DXVECTOR3(0,1,0));

      D3DXMatrixLookAtLH(&cNeverChange.view[2],&LPos,&(LPos+D3DXVECTOR3(0,1,0)),
&D3DXVECTOR3(0,0,-1));

      D3DXMatrixLookAtLH(&cNeverChange.view[3],&LPos,&(LPos+D3DXVECTOR3(0,-
1,0)),&D3DXVECTOR3(0,0,1));

      D3DXMatrixLookAtLH(&cNeverChange.view[4],&LPos,&(LPos+D3DXVECTOR3(0,0,1)),
&D3DXVECTOR3(0,1,0));

      D3DXMatrixLookAtLH(&cNeverChange.view[5],&LPos,&(LPos+D3DXVECTOR3(0,0,-
1)),&D3DXVECTOR3(0,1,0));
```

Dove LPos è la posizione della luce puntiforme. Se non avete capito perché fare LPos + D3DXVECTOR3 meglio che andiate a vedere meglio il cubemapping.

Lo shader sarà abbastanza usuale. Il vertex shader ridotto all'osso, l'inutile geometry effettuerà il cubemapping.
Il pixel shader, conterrà il calcolo della Z

```
float ps_depth (VGS_OUTPUT In)      :     SV_TARGET
{
      return (distance(In.WPos,LightInfo) / FarClip) + 0.0001f;
}
```

Il FarClip in linea di massima settatelo come quello della matrice di proiezione. Serve a limitare i valori della Z tra 0 e 1 (cosicchè diventi proprio un vero depth buffer).

Sarebbe buona norma verificare subito con PIX il risultato ottenuto e vedere se le 6 texture sono riempite.

Per la spot light sarà tutto molto piu' semplice.
Con 1 sola texture e una matrice di proiezione uguale a una usuale. Solo il FOV, lo setterete con il massimo angolo della luce (Phi).

Nel pixel shader principale della scena, infine, dovrà essere effettuato un confronto tra i valori delle texture di profondità.

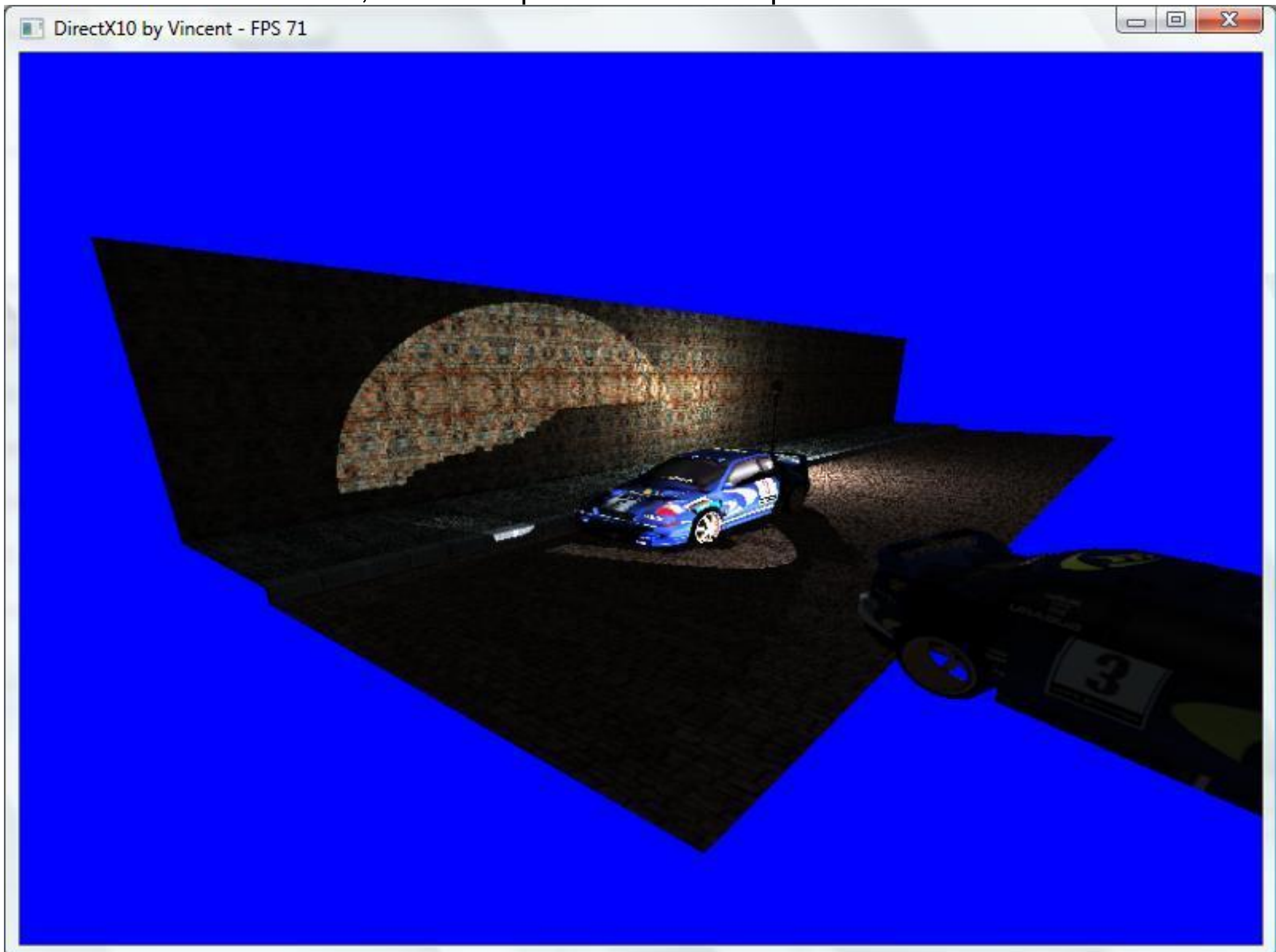
Per la spot light, proiettiamo le coordinate in D3DSpace e facciamo il sample del pixel, e lo confrontiamo con step

```
      float2 VTexC = float2(In.PSpot2D.x/In.PSpot2D.w * 0.5f + 0.5f,-
In.PSpot2D.y/In.PSpot2D.w * 0.5f + 0.5f);

      float2 SpotShadow = SpotText.Sample(SamplText,VTexC).xy;

int ShadowValue = (step((LightToVecDist/FarClip),SpotShadow.x) && Atten != 0);

//Cubemap sampling
            float CubeShadow =
Cubemap.Sample(SamplText,(LightToVec/LightToVecDist));
            int CubeValue =  step((LightToVecDist / FarClip),CubeShadow);
```

I 2 valori ricevuti (ShadowValue e CubeValue) andranno moltiplicati per i valori rispettivi della luce.

```
float4 Final = Ambient + (PointLight * CubeValue) + (Spotlight * ShadowValue)
```

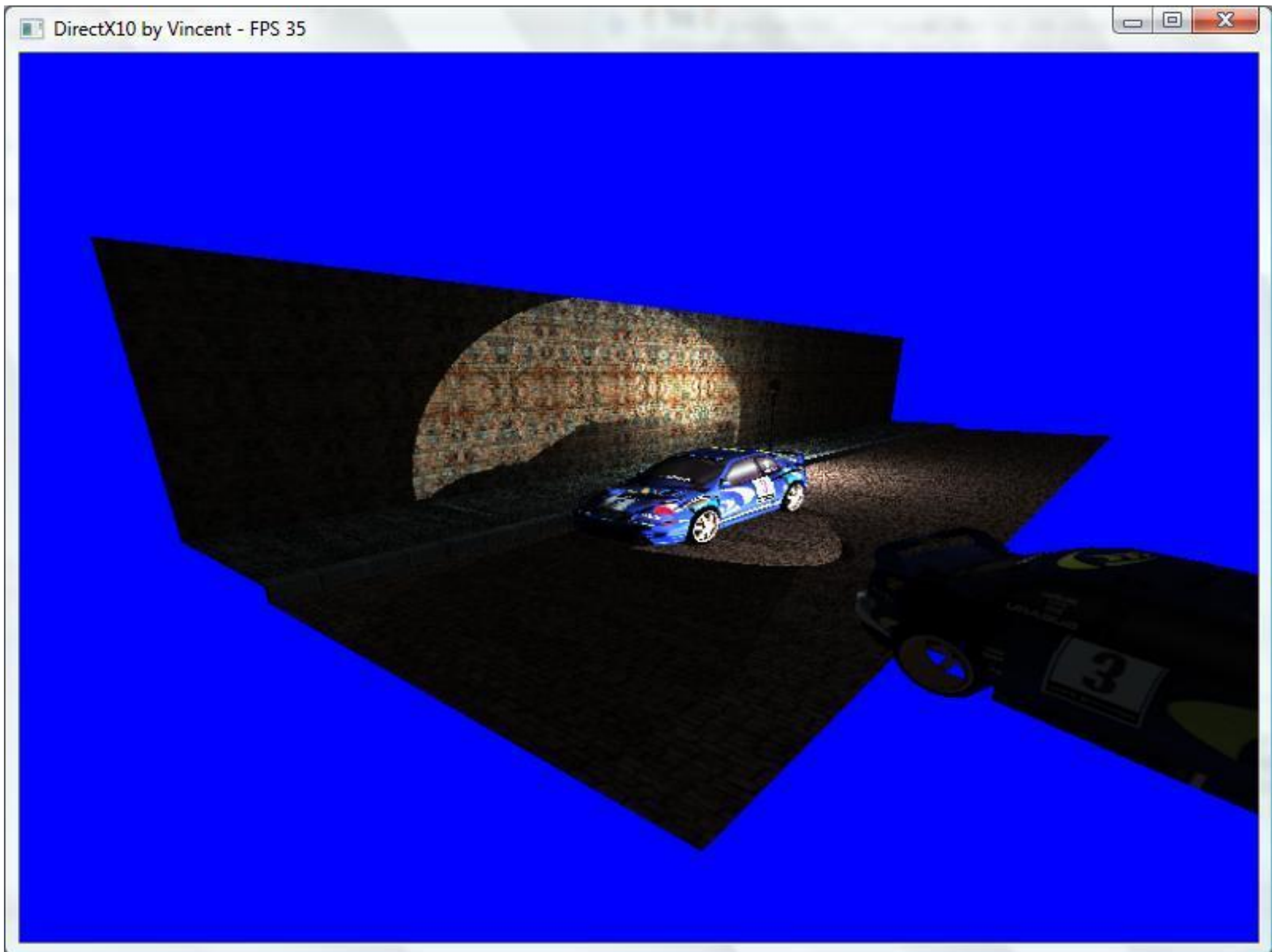
Alla fine moltiplichiamo la luce per il valore di ombra (si/no)
Con una texture 128x128, il risultato potrebbe essere questo.



Prima di dire che schifo continuate a leggere.

Potete notare l'ombra dell'auto proiettata sul muro dalla spot light (centrata nel mezzo della seconda auto) e della Pointlight (centrata sul lampione che proietta l'ombra dell'auto verso il pavimento).

Ok, l'aliasing si fa sentire.
La tecnica, effettivamente dipende molto dalle dimensioni della texture.
Notevoli miglioramenti si notano se viene aumentata a 1024x1024



Poiché questa è generata per ogni frame, potete notare l'abbassamento del frame rate (della metà)

Un buon metodo che da ottimi risultati con una texture così piccola è il Variance Shadow Mapping. Oltre a creare una penombra, l'aliasing viene quasi completamente a mancare. Fu "ideata" da Andrew Lauritzen (conosciuto come AndyTX in giro).

Nonostante abbia distribuito un buon paper sull'argomento (http://www.punkuser.net/vsm/vsm_paper.pdf), proverò a spiegarvi l'algoritmo in poche parole.

La tecnica (altro che semplice, come dicono quelli nVidia: "fast variance shadow mapping") è di difficile comprensione, non tanto per il codice, ma per i concetti statistici che ci stanno dietro. Spero che almeno sappiate cosa vuol dire funzione continua.

At first, la texture andrà cambiata predispondendola per 2 canali da 32 bit (R32G32). Nel primo canale manterremo, come al solito, la profondità. Nel secondo invece, metteremo la profondità al quadrato, variandola di un minimo per evitare artefatti che spiegherò dopo. Dunque avremo:

```
float2 ps_spotdepth (VPS_OUTPUT In) :     SV_TARGET
{
      float depth = (distance(In.WPos,LightInfo3) / FarClip) + 0.0001f;
      float dx = ddx(depth);
      float dy = ddy(depth);
      float depth_2 = depth * depth + 0.25 * (dx*dx+dy*dy);
```

```
    return float2(depth,depth_2);
}
```

In questo modo abbiamo dei dati che possono essere filtrati linearmente dall'hardware.

Consideriamo ora la profondità di un pixel come una funzione f(x). x è il pixel, il risultato è la profondità
Per le modifiche che abbiamo fatto sopra, disponiamo di f(x) e f(x^2).
Le due funzioni sono continue in tutto il dominio. Infatti, per ogni x,y compreso tra 0 e 1. f(x,y) ha sempre senso e restituirà un pixel.

Adesso, consideriamo il momento in statistica, definito come la media della *k*-esima potenza dei valori

$$\mu_k = \int_{-\infty}^{+\infty} x^k p_X(x) dx$$

Dove p(x) denota la funzione di densità.

Consideriamo quindi i momenti delle funzioni f(x) e f(x^2)

$$M_1 = E(x) = \int_{-\infty}^{\infty} xp(x)dx$$

$$M_2 = E\left(x^2\right) = \int_{-\infty}^{\infty} x^2 p(x)dx.$$

Essendo 2 momenti semplici, è possibile calcolare la Varianza (indice di dispersione) come M2 – (M1)^2, convenzionalmente indicato con sigma(quadro).

$$\mu = E(x) = M_1$$

$$\sigma^2 = E\left(x^2\right) - E(x)^2 = M_2 - M_1^2.$$

Grazie alla varianza, è possibile usare la diseguaglianza di **Čebyšëv,** che afferma:
"se la variabile casuale X ha valore atteso μ e la varianza σ^2 e λ è un reale positivo, allora la probabilità che X assuma un valore compreso tra μ-λσ e μ+λσ è maggiore di $1-1/\lambda^2$."

Grazie alla probabilità espressa dalla diseguaglianza, è possibile approssimare in modo molto vicino un pesante Percentage Closer Filtering su una superficie arbitraria.
Dunque alla fine, la percentuale di illuminazione è data dalla formula finale:

Variance / (Variance + d*d);

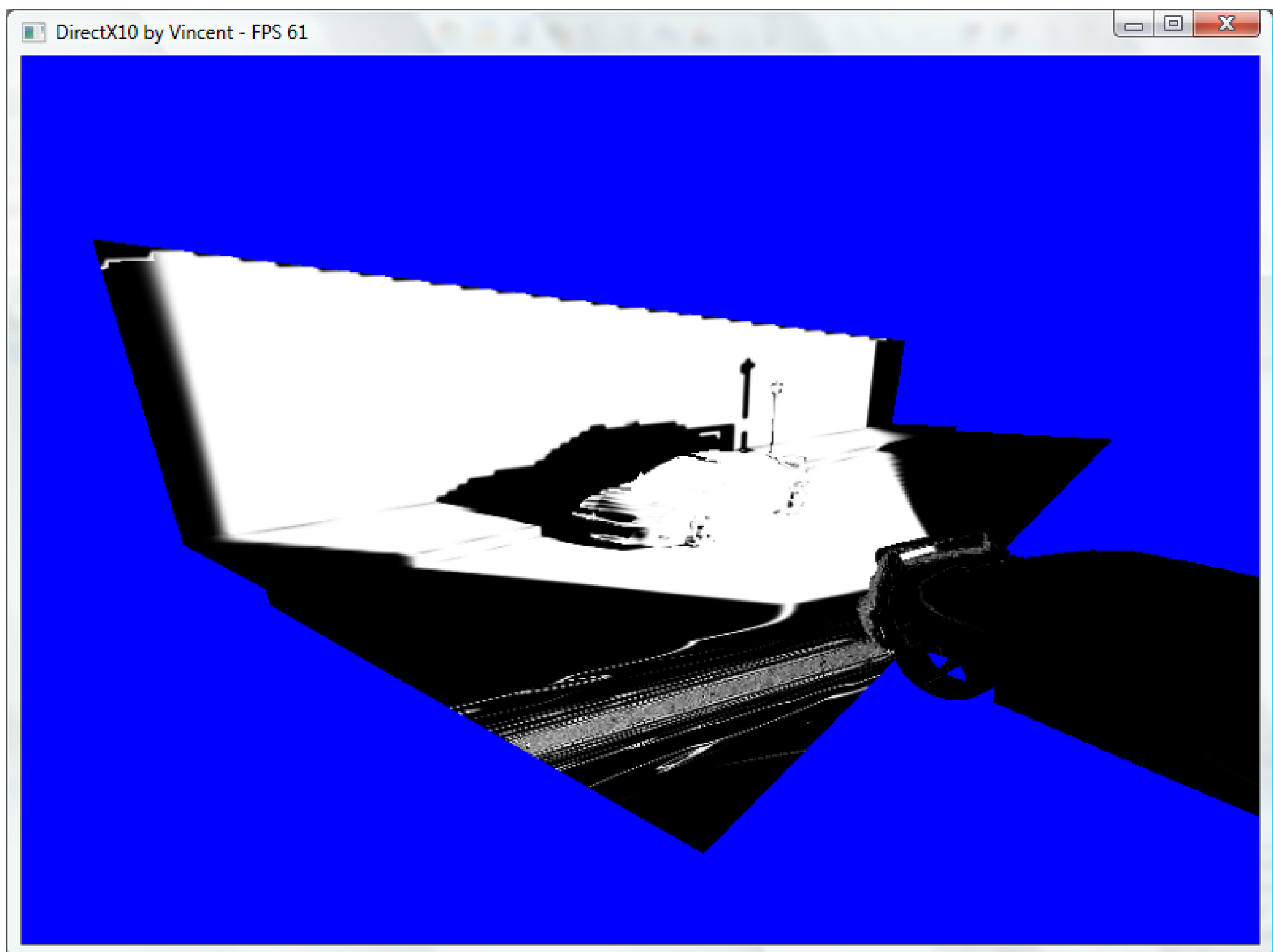
Se avete capito solo il procedimento matematico (e non la teoria), la stesura del codice è relativamente semplice.
Considerando SpotShadow un float2 con x = Depth e y = Depth^2,

```
float E_x2 = SpotShadow.y;
float Ex_2 = SpotShadow.x * SpotShadow.x;
float variance = E_x2 - Ex_2;
```

```
float mD = SpotShadow.x - (LightToVecDist/FarClip);
float mD_2 = mD * mD;
variance = max(variance,0.00001f);
float p = variance / (variance + mD_2);

lit = (smoothstep(0.3f,1.0f,p));
```

Lo smoohstep serve ad evitare il fenomeno di lightbleeding. Potete tranquillamente toglierlo.

E' sempre importante verificare man mano i risultati. Se tutto è stato fatto correttamente, ritornando dal pixel shader i semplici valori di lit, il risultato dovrebbe essere questo.



Per "importare" il risultato nella scena, basterà illuminare i valori di luce con Lit invece dei valori presi con step.

Si, c'è comunque aliasing. Il metodo, in effetti, funziona molto meglio se accompagnato con il gaussian blur, del quale darò una spiegazione veloce.

Il gaussian blur è il classico effetto sfumato che si vede anche nella realtà.
L'effetto (che alla fine è un postprocessing) prende il nome dalla caratteristica curva gaussiana a forma di campana (che useremo per calcolarci il valore del blur da effettuare).
In sintesi, l'effetto consiste nel campionare l'immagine più volte (oltre al pixel solito, prenderemo anche alcuni pixel circostanti) e, dato ad ognuno di essi un peso apposito

(calcolato tramite la curva di Gauss), ne verrà effettuata la media aritmetica (o meglio, i valori verranno normalizzati).

La funzione gaussiana ha un'equazione del tipo $G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$
Per nostra fortuna, questa può essere decomposta in una somma di 2 funzioni, permettendoci di divedere il blur in 2 pass (orizzontale e verticale)

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

Con Sigma parametro reale.
Perché la curva di Gauss?
Il caso vuole che anche questa è usata nella teoria delle probabilità.
Per saperne di più, cercate nei testi adeguati.

In primis, scriveremo una funzione che ci calcolerà i "pesi" dei pixel (rispetto a quello centrale) tramite la curva gaussiana.
In accordo alla funzione scritta sopra, avremo:

```
float GaussianWeight(float Sigma, float Displace)
{
	float value = (1/(sqrt(2*D3DX_PI) * Sigma)) * exp( -(Displace*Displace)/(2 * Sigma * Sigma));
	return value;
}
```

Questi valori saranno costanti ma andranno passati comunque allo shader, quindi sarebbe il caso di creare un constant buffer "fisso".
Ricordando che la funzione gaussiana è pari (ossia G(x) = G(-x)), basterà calcolare il valore della funzione solo per il displace positivo.

Parliamo ora di questo displace, ossia di quanto spostarci per poter prendere i pixel circostanti.

Dobbiamo spostarci, prima in orizzontale e poi in verticale, di alcuni pixel rispetto a quello centrale puntato dalle TexCoord correnti.
Dunque, calcolate le dimensioni della texture, avremo la dimensione del singolo texel effettuando semplicemente la divisione 1.0f/TextureSize

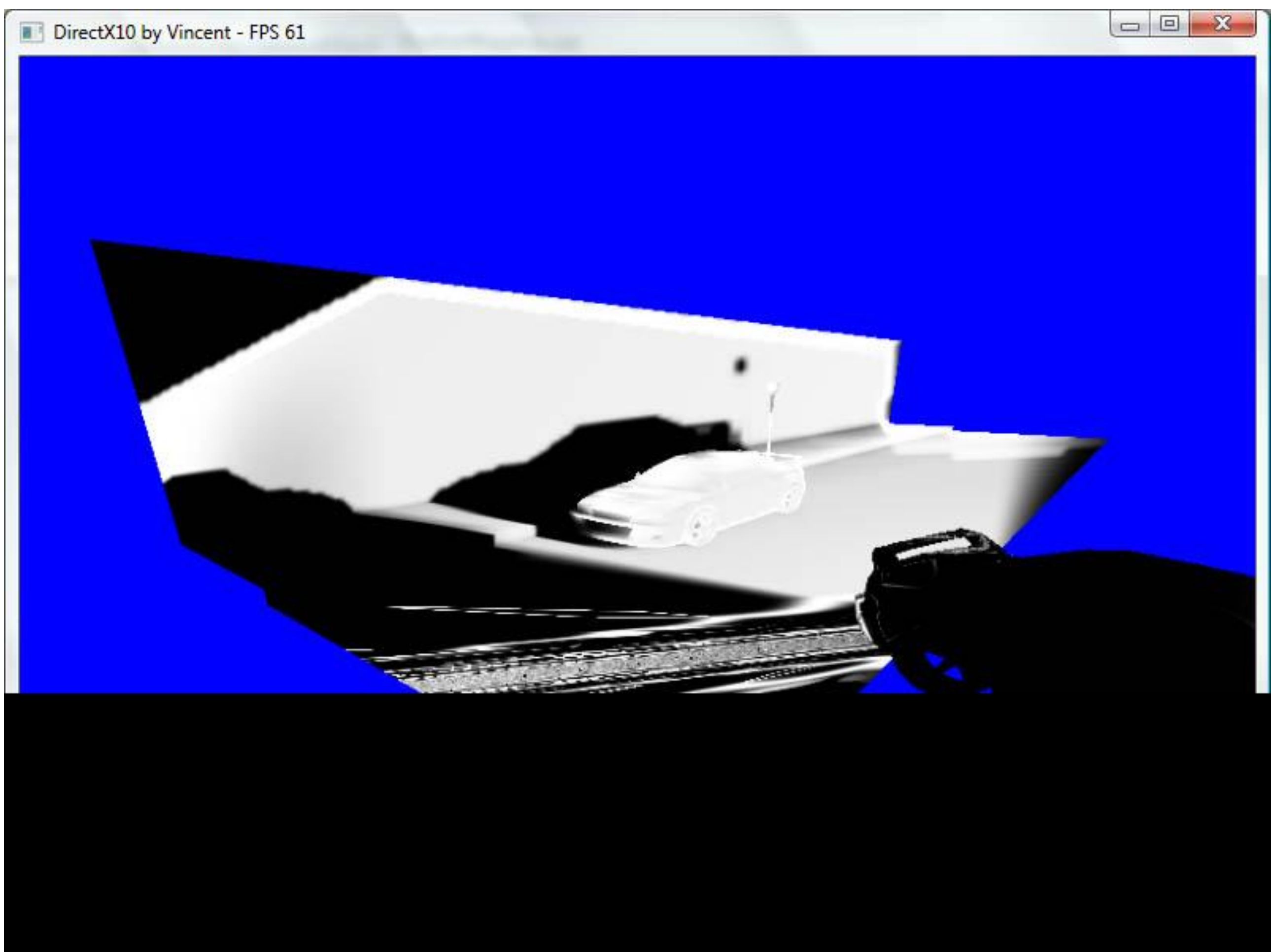
Essendo un effetto post processing, sarà necessario creare un render target adeguato e un quadrato (con vertex buffer) da disegnare.
Dopodichè, il pixel shader sarà

```
float4 Blur(VSB_OUTPUT Inp) : SV_TARGET
{
    float4 c = 0;

    float4 SampleOnZero = SpotText.Sample(SamplText,Inp.Tex);

    for (int i = 0; i < SAMPLE_COUNT / 4; i++)
    {
        int i2 = i * 2;
```

```
        c += SpotText.Sample( SamplText, Inp.Tex + SampleOffsets[i2].xy ) *
SampleWeights[i].x;
        c += SpotText.Sample( SamplText, Inp.Tex + SampleOffsets[i2].zw ) *
SampleWeights[i].y;
        c += SpotText.Sample( SamplText, Inp.Tex + SampleOffsets[i2 + 1].xy ) *
SampleWeights[i].z;
        c += SpotText.Sample( SamplText, Inp.Tex + SampleOffsets[i2 + 1].zw ) *
SampleWeights[i].w;
    }

    return c + (SampleOnZero*0.39);
}
```

Il codice è contorto a causa delle regole di packing degli shader che vi ho esposto sopra.
I sample offsets sono stati riempiti in C++.

Effettuando il blur, i risultati crescono notevolmente a livello qualitativo. (La terza immagine ha solo la luce un po' piu' potente per far notare meglio il miglioramento).



3

3

In linea prettamente teorica il Variance Shadow Mapping finisce qui. In pratica questo ottimo filtro può essere unito ad altri effetti d'ombra sulle mappe di profondità.
Famoso è il Parallel Split Variance Shadow Mapping:consiste nel dividere la scena in piu' shadow maps e fare il variance su quest'ultime, unendo i risultati. Permette di rende rizzare ad alta qualità anche ampie aree (per le viste da elicottero, per esempio).
Oppure la Summed Area Variance Shadow Map, di cui lascio a voi l'implementazione

L'esempio contiene l'implementazine della tecnica, nel quale è stato evitato completamente l'uso di D3DX per gli effetti e, considerando i problemi dovuti al packing dei constant buffer, il codice soffre dell'effetto Spaghetti Code. (Array di vertex shader, set imprecisi, constant buffer dichiarati senza un minimo di cognizione). Dunque non datelo per "perfetto".
Ho lasciato per voi la funzione SetDepthStencil2, che usa un depth buffer senza render target (ossia usa lo stesso depth buffer del rendering usuale).
Potete provare a usarla al posto della normale, e fare le vostre modifiche.

In particolare vi segnalo i comandi
F3 – Shadow Map normale 128x128
F5 – Lit del variance
F6 – Variance Shadow Map
F7 – Blurred Variance Shadow Mapping
F5 - Lit Blurred

L'effetto variance non è stato implementato sulla cubemap per far notare meglio le differenze.

Numerosi sono stati gli aiuti per questo lavoro. Le doverose citazioni (non in ordine di rilevanza) sono

Goz - (I calcoli in Vertex Shader sono diversi dagli stessi, se eseguiti in Pixel Shader)
Jarkkol - (Penso che dovresti normalizzare il risultato del blur. I coefficienti sono sbagliati…mi faresti vedere il codice?)
ReedBeta – (Hai un problema al terzo parametro)
Dr. Kappa – dalle infinite idee (Hai provato con questo? E quest'altro?)

Email e commenti sono sempre graditi.

Vincenzo Chianese

http://xvincentx.netsons.org/programBlog