

Tutoriels Direct3D 10 : Shaders



par Msdn Romain Perruchon (traducteur, relecteur) Cyril
Doillon (relecteur) Arnaud Feltz (traducteur, relecteur)

Date de publication : 12/05/2009

Dernière mise à jour : 12/05/2009

Traduction d'un article original de la Library MSDN US

III - Tutoriels sur DXUT.....	3
III-11 - Tutoriel 11 : Vertex shaders.....	3
III-11-a - Résumé.....	3
III-11-b - Source.....	3
III-11-c - Shader de sommet.....	3
III-11-d - Création de l'effet de vague.....	3
III-12 - Tutoriel 12 : Pixel shaders.....	4
III-12-a - Résumé.....	4
III-12-b - Source.....	4
III-12-c - Pixel shader.....	4
III-12-d - Environment Map.....	5
III-12-e - Configurer l'Environment Map.....	5
III-12-f - Implémenter le Lookup.....	5
III-13 - Tutoriel 13 : Geometry Shaders.....	6
III-13-a - Résumé.....	6
III-13-b - Source.....	6
III-13-c - Geometry Shader.....	7
III-13-c-1 - Formater un Geometry Shader.....	7
III-13-c-2 - Explosion du modèle.....	8
III-13-c-3 - Calculer la face normale.....	8

III - Tutoriels sur DXUT

III-11 - Tutoriel 11 : Vertex shaders



III-11-a - Résumé

Ce tutoriel traite des possibilités du vertex shader. Il est conçu pour montrer à l'utilisateur les capacités de manipulation des sommets. A la fin de ce tutoriel, nous aurons un effet de vague imposé au modèle du personnage du tutoriel précédent. Cet effet est créé entièrement par le GPU avec aucune interaction du CPU avec les données.

III-11-b - Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial11

III-11-c - Shader de sommet

Les tutoriaux précédents ont déjà montré l'utilisation du vertex shader, ce tutoriel va accentuer ce stage dans le pipeline. Les sommets contiennent différents types d'informations telles que les coordonnées, les normales, les coordonnées de texture, les matériaux, les couleurs, et même des données personnalisées. Par exemple, si vous regardez l'exemple ParticlesGS, vous verrez que chaque sommet passé dans le pipeline contiendra aussi une variable appelée *Timer*, laquelle peut faire varier l'apparence des particules sur le temps.

Le but du vertex shader est de décharger les calculs intensifs du CPU, vers le GPU. Ceci libère le CPU pour accomplir d'autres tâches dans l'application. Par exemple, les jeux déchargent parfois les traitements graphiques vers le GPU, pendant que l'intelligence artificielle et la physique sont effectuées par le CPU.

Ce tutoriel est conçu pour montrer les possibilités du vertex shader, plutôt que d'apprendre des techniques spécifiques. Il est encouragé d'essayer de nouveaux effets et de faire des essais en fonction des résultats. Pour les méthodes spécifiques pour accomplir les effets, se référer aux exemples inclus avec le SDK. Cependant, notez qu'ils sont bien plus complexes et appliquent des concepts bien plus avancés.

III-11-d - Création de l'effet de vague

L'effet de vague sur le modèle est créé en modulant la position de x de chaque point par une onde sinusoïdale. L'onde sinusoïdale est contrôlée par la position y, de ce fait créant une vague le long du modèle. Cependant, pour faire bouger la vague et donner l'apparence d'animation, l'onde sinusoïdale est également modulée par une variable de temps.

Finalement, il y a une variable pour contrôler le nombre de déplacement par le vertex shader, et ceci est la variable d'ondulation. Cette variable est passée dans le shader et contrôlée par un curseur graphique.

```
output.Pos.x += sin( output.Pos.y*0.1f + Time ) *Waviness;
```

Après que cette manipulation soit effectuée, le vertex shader devra toujours préparer les sommets pour affichage. Donc, il y a toujours la transformation de la matrice de monde, de la matrice de vue, et la matrice de projection. Cependant, on choisi de faire la transformation de monde avant l'effet de vague, ainsi il sera plus facile de déterminer les effets sur l'écran, car les axes x et y sont apparents.

```
output.Pos = mul( output.Pos, View );
output.Pos = mul( output.Pos, Projection );
```

III-12 - Tutoriel 12 : Pixel shaders



III-12-a - Résumé

Ce tutoriel se concentre sur le pixel shader et ses possibilités. Il y a plusieurs choses qui peuvent être accomplies avec le pixel shader, et quelques unes des fonctions les plus habituelles sont listées. Le pixel shader appliquera un *environment map* à l'objet.

À la fin de ce tutoriel, le modèle deviendra brillant, et il reflètera son environnement entourant.

III-12-b - Source

SDK root\Samples\C++\Direct3D10\Tutorials\Tutorial12

III-12-c - Pixel shader

Les pixel shaders sont utilisés pour manipuler la couleur finale d'un pixel avant qu'il n'atteigne l'*output merger*. Chaque pixel qui est affiché sera passé au travers de ce shader avant la sortie. Une fois que le pixel est passé au travers du pixel shader, les seules opérations qui peuvent être effectuées sont celles effectuées par l'*output merger*, telles que l'*alpha blending*, le test de profondeur, le *stencil testing*, et ainsi de suite.

Le pixel shader a évolué depuis le mapping de texture que l'on trouvait dans les hardwares précédents. Au lieu d'un simple texture de consultation, il est possible de calculer la couleur finale depuis des sources multiples, ainsi que de l'altérer selon les données du sommet. Pour des applications générales, cependant, un pixel shader effectue de multiples consultations sur différentes textures.


III-12-d - Environment Map

Sur des surfaces réfléchissantes, il n'est pas possible de juste faire une texture de consultation sur des coordonnées de texture fixes pour déterminer la couleur finale. Cela est dû au fait que quand l'objet ou le spectateur bouge, le reflet change. Donc, nous devons mettre à jour la réflexion chaque fois que quelque chose bouge.

Une méthode efficace pour faire croire à l'observateur qu'il y a une réflexion dynamique sur l'environnement est de générer une texture spéciale qui entoure l'objet. Cette texture s'appelle un *cube map*. Un *cube map* est effectivement de placer un objet au milieu d'un cube, avec chaque face du cube devenant une texture. Un *environment map* est un *cube map* qui a des textures qui correspondent à la vue de l'environnement sur cette face. Si l'environnement est statique, ces *environment maps* peuvent être pré-générés. Si l'environnement est dynamique, le *cube map* doit être mis à jour à la volée. Voir l'exemple CubeMapGS pour une illustration de comment effectuer ceci.

Depuis cet *environment map*, on peut calculer ce qu'une caméra verra comme reflet. D'abord on peut trouver la direction par laquelle la caméra voit l'objet, et de cela, refléter la normale de chaque pixel et effectuer une consultation basé sur ce vecteur reflété.

III-12-e - Configurer l'Environment Map

La configuration de l'*environment map* n'est pas le but de ce tutoriel. La procédure à suivre est très similaire à celle d'une map texture normale. Voir le  **Tutoriel 7**, pour une explication de comment initialiser correctement une *texture map* et ses vues ressources associées.

```
// Charge l'Environment Map
ID3D10Resource *pResource = NULL;
V_RETURN( DXUTFindDXSDKMediaFileCch( str, MAX_PATH,
L"Lobby\\LobbyCube.dds" ) );
V_RETURN( D3DX10CreateTextureFromFile( pd3dDevice, str, NULL, NULL,
&pResource ) );
if(pResource)
{
    g_pEnvMap = (ID3D10Texture2D*)pResource;
    pResource->Release();
    D3D10_TEXTURECUBE_DESC desc;
    g_pEnvMap->GetDesc( &desc );
    D3D10_SHADER_RESOURCE_VIEW_DESC SRVDesc;
    ZeroMemory( &SRVDesc, sizeof(SRVDesc) );
    SRVDesc.Format = desc.Format;
    SRVDesc.ViewDimension = D3D10_SRV_DIMENSION_TEXTURECUBE;
    SRVDesc.TextureCube.MipLevels = desc.MipLevels;
    SRVDesc.TextureCube.MostDetailedMip = 0;
    V_RETURN(pd3dDevice->CreateShaderResourceView( g_pEnvMap, &SRVDesc,
&g_pEnvMapSRV ));
}
// Défini l'Environment Map
g_pEnvMapVariable->SetResource( g_pEnvMapSRV );
```

III-12-f - Implémenter le Lookup

Nous allons parcourir l'algorithme simple décrit plus haut et effectuer un lookup propre d'un *environment map*. Les calculs sont effectués dans le vertex shader et ensuite interpolés vers le pixel shader pour le lookup. Il est préférable de le calculer dans le vertex shader et interpoler vers le shader de pixel car il y a moins de calculs requis.

Pour calculer le vecteur refléter pour le lookup, nous avons besoins de deux informations. D'abord, la normale du pixel en question, ensuite, la direction de l'oeil.

Car l'opération est effectuée dans l'espace vue (le view space), nous devons transformer la normale du pixel vers l'espace de vue correcte.

```
float3 viewNorm = mul( input.Norm, (float3x3)View );
```

Ensuite, nous devons trouver la direction de la caméra. Notez, cependant, que nous sommes déjà en view space, et donc, la direction de la caméra est celle-ci sur l'axe Z (0,0,-1.0), car nous regardons directement vers l'objet.

Maintenant que nous avons nos deux pièces d'informations, on peut calculer la réflexion du vecteur avec la commande **reflect**. La variable *ViewR* est utilisée pour stocker la réflexion résultante pour le traitement dans le pixel shader.

```
output.ViewR = reflect( viewNorm, float3(0,0,-1.0) );
```

Une fois que le vecteur correcte a été calculé dans le vertex shader, on peut traiter l'environnement map dans le pixel shader. Ceci est effectué en appelant une fonction toute faite pour échantillonner l'environnement map et retourner la valeur couleur depuis cette texture.

```
// Charge la texture environment map
float4 cReflect = g_txEnvMap.Sample( samLinearClamp, viewR );
```

Car tout est normalisé (et les coordonnées de textures ont une portée de 0 à 1), les coordonnées x et y correspondront directement aux coordonnées de textures requises pour le lookup.

En bonus, on inclut le code pour faire la texture lookup plate original, et on calcule cela en tant que terme diffus. Pour jouer avec le taux de réflexion, vous pouvez moduler **cReflect** par un facteur de mise à l'échelle par rapport à **cDiffuse** et expérimenter avec les résultats. Vous pouvez avoir un personnage qui reflète beaucoup ou un personnage quelque peu terne.

III-13 - Tutoriel 13 : Geometry Shaders



III-13-a - Résumé

Ce tutoriel va explorer une partie du pipeline graphique qui n'a pas encore été abordé dans les tutoriels précédent. Nous allons évoquer quelques fonctionnalités basique des geometry shader.

Le résultat de ce tutoriel sera que le modèle comportera une seconde couche extraite du modèle. Notons que le modèle original sera toujours préservé au centre.

III-13-b - Source

```
SDK root\Samples\C++\Direct3D10\Tutorials\Tutorial13
```

III-13-c - Geometry Shader

Le bénéfice des geometry shader (GS) est qu'ils permettent de manipuler les mailles sur la base par-primitive. Au lieu de lancer un calcul sur chaque sommet individuellement, il y a l'option pour exploiter sur une base par-primitive. C'est-à-dire, des sommets peuvent être passés dedans comme sommet simple, une ligne segment (deux sommets), ou comme triangle (trois sommets).


En permettant la manipulation à un niveau par-primitive, de nouvelles idées peuvent être approchées, et il y a plus d'accès aux données à tenir compte pour cela. Dans ce tutoriel, vous verrez que nous avons calculé la normale pour le visage. En sachant la position de chacun des trois sommets, nous pouvons trouver la normale du visage.

En plus de permettre l'accès à toutes les primitives, le geometry shader peut créer de nouvelles primitives à la volée. Précédemment dans Direct3D, le pipeline graphique pouvait seulement manipuler le contenu existant, et il pourrait amplifier ou desamplifier des données. Le geometry shader dans Direct3D 10 peut lire dedans une simple primitive (avec les primitives bord-adjacents facultatifs) et émettre zéro, une, ou de multiples primitives basés sur ça.

Il est possible d'émettre un type différent de géométrie que la source d'entrée. Par exemple, il est possible de lire différents sommets, et de générer des triangles multiples basées sur eux. Ceci permet un éventail de nouvelles idées pouvant être exécuté sur le pipeline graphique sans intervention du CPU.

Le geometry shader existe entre le vertex shader et le pixel shader dans le pipeline graphique. Puisque de nouvelles géométries peuvent être créés potentiellement par le geometry shader, nous devons nous assurer qu'ils sont également correctement transformés dans l'espace de projection avant que nous les passions dans le pixel shader. cela peut être fait par le vertex shader avant qu'il entre dans le geometry shader ou il peut être fait dans le geometry shader lui même.

En conclusion, le rendement du Geometry Shader peut être rerouté à un buffer. Vous pouvez lire dans un groupe de triangles, produire de nouvelles géométries, et les stocker dans un nouveau buffer. Cependant, le concept du flux de sortie est au delà de la portée des tutoriels, et il est mieux expliqué dans les exemples trouvés dans le SDK.

Plusieurs des échantillons trouvés dans le SDK Direct3D 10 illustrent les techniques spécifiques qui peuvent être réalisées avec le geometry shader. Si vous souhaitez un exemple simple pour commencer, vous pouvez essayer  **ParticulesGS**. ParticulesGS simule et affiche un système de particules dynamique (création, explosion et destruction de particules) entièrement sur le GPU.

III-13-c-1 - Formater un Geometry Shader

Contrairement au Vertex shader et au Pixel shader, le geometry shader ne produit pas nécessairement un nombre statique de sorties par entrée. En soi, le format pour déclarer le shader est différent des deux autres.

Le premier paramètre est *maxvertexcount*. Ce paramètre décrit le nombre maximum de sommets qui peuvent être sorti chaque fois que le shader est lancé. Ceci est suivi du nom du geometry shader, qui a été convenablement appelé GS.

Le nom de fonction est suivi des paramètres passés dans la fonction. Le premier contient le mot-clé *triangle*, qui spécifie que le Geometry Shader opérera sur des triangles en entrée. Le suivant est le format de sommet, et l'identifiant (avec le nombre signifiant la taille du tableau, 3 pour des triangles, 2 pour un segments). Le second est le format et flux de sortie. *TriangleStream* signifie que la sortie sera dans les triangles (bandes de triangle pour être exactes), puis le format est spécifié dans les chevrons. En conclusion, l'identifiant pour le flux est dénotée.

```
[maxvertexcount(12)]
void GS( triangle GSPS_INPUT input[3], inout TriangleStream<GSPS_INPUT> TriStream )
```

Si des sommets commencent à être émis à un *TriangleStream*, il supposera qu'ils tous sont liés ensemble comme une bande. Pour finir une bande, appelez **RestartStrip** dans le flux. Pour créer une liste de triangle, vous devez veiller que vous appelez bien **RestartStrip** après chaque triangle.

III-13-c-2 - Explosion du modèle

Dans ce tutoriel, nous couvrons les fonctions de base des geometry shader. Pour illustrer cela, nous allons créer un effet d'explosion sur notre modèle. Cet effet est créé par extrusion de chaque sommet dans la direction de la normale du triangle considéré.

Notez qu'un effet similaire a été implémenté dans des tutoriels précédent, par lequel nous expulsions chaque sommet par sa normale, commandé par le glisseur *puffiness*. Ce tutoriel démontre l'usage des information d'un triangle complet pour générer la face normale. la différence d'utiliser la face normale est que vous verrez des lacunes entre les triangles éclatés. Parce que les sommets entre différents triangles sont partagés, ils seront donc passés par deux fois dans le geometry shader. De plus, puisque chaque fois qu'une extrusion est effectuée dans la normale du triangle, par opposition au sommet, les deux sommets finaux peuvent finir vers le haut dans différentes positions.

III-13-c-3 - Calculer la face normale

Pour calculer la normale pour n'importe quel plan, nous avons besoin d'abord de deux vecteurs qui résident sur le plan. Puisque nous nous sommes donnés un triangle, nous pouvons soustraire deux sommets quelconques du triangle pour obtenir les vecteurs relatifs. Une fois que nous avons les vecteurs, nous prenons le produit en croix pour obtenir la normale. Nous devons également normaliser la normale, puisque nous reverrons l'échelle plus tard.

```
//
// Calculer la face normale
//
float3 faceEdgeA = input[1].Pos - input[0].Pos;
float3 faceEdgeB = input[2].Pos - input[0].Pos;
float3 faceNormal = normalize( cross(faceEdgeA, faceEdgeB) );
```

Une fois que nous avons la normale de la face, nous pouvons expulser chaque point du triangle dans cette direction. Pour faire ainsi, nous employons une boucle, qui fera trois tour et opérera sur chaque sommet. La position du sommet est expulsée par la normale, multipliée par un facteur. Puis, puisque le Vertex shader n'a pas transformé les sommets à l'espace approprié de projection, nous devons également faire cela dans le Geometry shader. En conclusion, une fois que nous empaquetons le reste des données, nous pouvons apposer ce nouveau sommet à notre *TriangleStream*.

```
for( int v=0; v<3; v++ )
{
    output.Pos = input[v].Pos + float4(faceNormal*Explode,0);
    output.Pos = mul( output.Pos, View );
    output.Pos = mul( output.Pos, Projection );

    output.Norm = input[v].Norm;

    output.Tex = input[v].Tex;

    TriStream.Append( output );
}
```

Une fois que les trois sommets ont été émis, nous pouvons couper la bande et recommencer. Dans ce tutoriel, nous voulons expulser chaque triangle séparément, ainsi nous finissons avec une liste de triangle.

```
TriStream.RestartStrip();
```

Ce nouveau flux de triangle est alors envoyé au Pixel shader, qui opérera sur ces données et les dessinera à la cible d'affichage.