

Tutoriels Direct3D 10 : Etat



par Msdn Romain Perruchon (relecteur) Cyril Doillon (relecteur)

Date de publication : 21/09/2008

Dernière mise à jour : 21/09/2008

Traduction d'un article original de la Library MSDN US

IV - Tutoriels sur DXUT.....	3
IV-14 - Tutoriel 14 : Gestion d'état.....	3
IV-14-a - Résumé.....	3
IV-14-b - Source.....	3
IV-14-c - Blend States.....	3
IV-14-d - Rasterizer States.....	8
IV-14-e - Etats Depth Stencil.....	11
IV-14-f - Rendu avec le Stencil Buffer.....	13

IV - Tutoriels sur DXUT

IV-14 - Tutoriel 14 : Gestion d'état



IV-14-a - Résumé

Ce tutoriel va explorer un aspect très important, mais souvent négligé, de la programmation sous Direct3D 10 : l'état. Bien qu'ils ne soient pas aussi glamours ni aussi flashy que les shaders, les changements d'état sont d'une importance indispensable en ce qui concerne la programmation graphique. En fait, le même shader peut avoir des résultats extrêmement différents basés uniquement sur l'état du device au moment du rendu.

Dans ce tutoriel, on explorera 3 principaux types d'objets état : **BlendStates**, **DepthStencilStates** et **RasterizerStates**. A la fin de ce tutoriel, vous devriez mieux comprendre la façon dont les états interagissent pour produire différentes représentations de la même scène.

IV-14-b - Source

```
(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial14
```

IV-14-c - Blend States

Avez-vous déjà entendu le terme Alpha Blending ? L'alpha blending implique de modifier la couleur d'un pixel tracé à une certaine position d'écran en utilisant la couleur du pixel qui existe déjà à cet emplacement. Les états Blend States (ID3D10BlendState quand on l'utilise directement à partir de l'interface de programmation et BlendState quand on l'utilise avec FX) permettent au développeur de spécifier cette interaction entre les anciens et les nouveaux pixels. Avec l'état Blend State réglé par défaut, les pixels écrasent simplement tout pixel déjà existant à n'importe quelle coordonnée d'écran pendant la rasterisation. Toute information sur le pixel précédemment tracé à cet emplacement est perdue.

Pour donner un exemple, imaginons que votre application dessine la vue d'une ville depuis l'intérieur d'un taxi. Vous avez dessiné des milliers de bâtiments, de trottoirs, de poteaux téléphoniques, de poubelles et autres objets qui font d'une ville ce qu'elle est. Vous avez même dessiné l'intérieur du taxi. Maintenant, dernière étape, vous voulez dessiner les vitres du taxi pour faire comme si vous regardiez effectivement à travers du verre.

Malheureusement, les pixels dessinés pendant la rasterisation du maillage de la vitre écrasent complètement les pixels déjà à ces emplacements sur l'écran. Cela inclut votre belle scène urbaine avec ses trottoirs, ses poteaux téléphone et tous les autres objets de votre cité. Ne serait-ce pas génial si vous pouviez inclure les informations qui étaient déjà à l'écran pendant le tracé des pare-brises du taxi ? Ne serait-ce pas génial aussi si nous pouvions le faire avec des changements minimaux de notre code actuel de pixel shader ?

C'est exactement ce que les états Blend States vous proposent. Pour démontrer cela, notre scène consiste en un modèle et un quadrilatère.

```
// Chargement du maillage
V_RETURN( g_Mesh.Create( pd3dDevice, L"Tiny\\tiny.x", (D3D10_INPUT_ELEMENT_DESC*)layout, 3 ) );
```

Notez que notre quadrilatère a un format de vertex différent et par conséquent qu'il a besoin d'une description de mise en page en entrée différente.

```
// Création d'un quad d'écran
const D3D10_INPUT_ELEMENT_DESC quadlayout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXCOORD0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 16, D3D10_INPUT_PER_VERTEX_DATA, 0 },
};

g_pTechniqueQuad[0]->GetPassByIndex( 0 )->GetDesc( &PassDesc );
V_RETURN( pd3dDevice->CreateInputLayout( quadlayout, 2, PassDesc.pIAInputSignature,
&g_pQuadLayout ) );

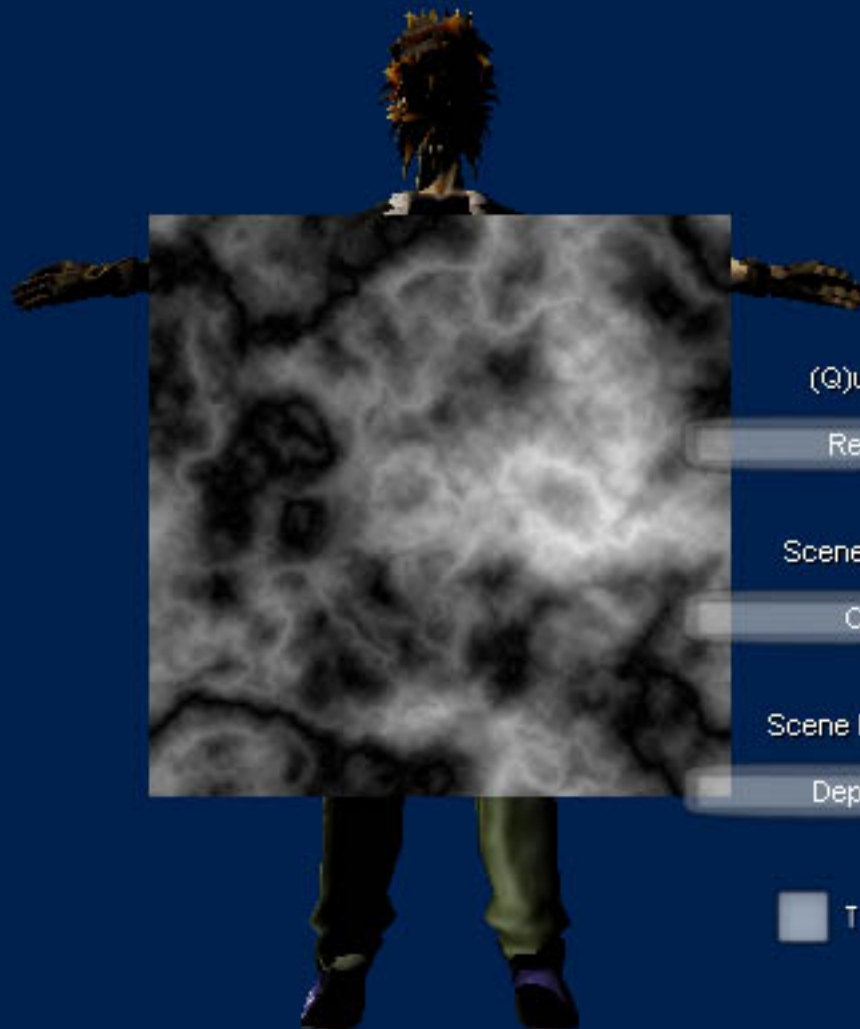
...

D3D10_SUBRESOURCE_DATA InitData;
InitData.pSysMem = svQuad;
InitData.SysMemPitch = 0;
InitData.SysMemSlicePitch = 0;
V_RETURN( pd3dDevice->CreateBuffer( &vbdesc, &InitData, &g_pScreenQuadVB ) );
```

Quand nous opérons le rendu de notre scène, nous voyons le modèle d'un personnage avec un quadrilatère dessiné au-dessus.

Tutorial14

10 Vsync on (640x480), RGB8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO



Toggle full screen

Toggle REF (F3)

Change device (F2)

(Q)uad Render Mode

RenderQuadSolid

Scene (R)asterizer Mode

CullOff/FillSolid

Scene Depth/(S)tencil Mode

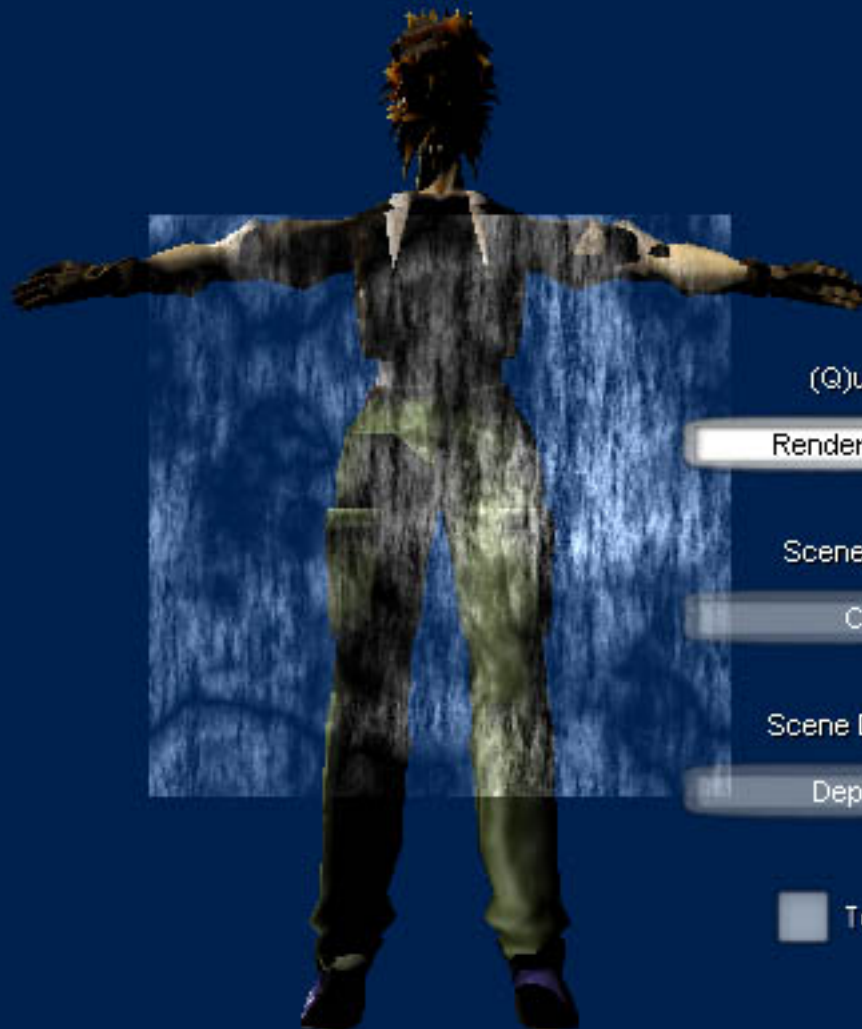
DepthOff/StencilOff

Toggle Spinning

Imaginez que ce personnage représente notre cité et que le quadrilatère est réellement un panneau de verre. Quand nous lançons l'application, la situation est exactement comme nous l'avons décrite plus haut : nous ne pouvons rien voir à travers le verre. Toutefois, en sélectionnant un mode de rendu de quad différent dans le menu déroulant, on peut soudainement voir à travers le quadrilatère. C'est comme si c'était devenu un carré de verre.

Tutorial14

10 Vsync on (640x480), RGB8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO



Toggle full screen

Toggle REF (F3)

Change device (F2)

(Q)uad Render Mode

RenderQuadSrcAlphaAdd

Scene (R)asterizer Mode

CullOff/FillSolid

Scene Depth/(S)tencil Mode

DepthOff/StencilOff

Toggle Spinning

En fait, tout ce que fait l'application quand nous choisissons un mode de rendu de quad différent, c'est changer la technique que nous utilisons dans le fichier FX pour le rendu du quadrilatère. Par exemple, **RenderQuadSolid** dans le menu déroulant se réfère à la technique suivante dans le fichier FX :

```
technique10 RenderQuadSolid
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, QuadVS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, QuadPS() ) );

        SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
    }
}
```

Les fonctions **SetVertexShader**, **SetGeometryShader** et **SetPixelShader** devraient toutes vous paraître familières à ce point. Sinon, révisez les didacticiels 10 à 12. La dernière ligne est ce sur quoi nous allons nous concentrer maintenant. Dans un fichier FX, c'est la façon dont on règle l'état Blend State. **NoBlending** se réfère à une structure d'état définie en haut du fichier FX. La structure désactive le mélange pour la première (0) cible de rendu.

```
BlendState NoBlending
{
    BlendEnable[0] = FALSE;
};
```


Comme nous avons dit plus haut, cet état Blend State ne fait rien d'intéressant. Parce que l'alpha blending est désactivé, les pixels du quad se contentent d'écraser les pixels existants placés à l'écran pendant le rendu du personnage. Toutefois, les choses deviennent plus intéressantes quand nous sélectionnons **RenderQuadSrcAlphaAdd** dans le menu défilant Quad Render Mode. Cela modifie le quadrilatère à rendre avec la technique **RenderQuadSrcAlphaAdd**.

```
technique10 RenderQuadSrcAlphaAdd
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, QuadVS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, QuadPS() ) );

        SetBlendState( SrcAlphaBlendingAdd, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
    }
}
```

Un examen attentif de cette technique dévoile deux différences entre elle et la technique **RenderQuadSolid**. La première est le nom. La seconde, d'une importance vitale, est que l'état *Blend State* est transmis à **SetBlendState**. Au lieu de transmettre au *blend state* **NoBlending**, nous transmettons le *blend state* **SrcAlphaBlendingAdd** qui est défini en haut du fichier FX.

```
BlendState SrcAlphaBlendingAdd
{
    BlendEnable[0] = TRUE;
    SrcBlend = SRC_ALPHA;
    DestBlend = ONE;
    BlendOp = ADD;
    SrcBlendAlpha = ZERO;
    DestBlendAlpha = ZERO;
    BlendOpAlpha = ADD;
    RenderTargetWriteMask[0] = 0x0F;
};
```

Ce Blend State est un peu plus complexe que le dernier. Le mélange est désactivé pour la première (0) cible de rendu. Pour une description approfondie de tous les paramètres du Blend State, reportez-vous à la documentation de  **D3D10_BLEND_DESC**. Voici un rapide aperçu de ce que cette fonction dit à Direct3D 10 de faire. Quand on opère le rendu du quadrilatère par cette technique, le pixel dont on va faire le rendu ne se contente pas de remplacer le pixel existant. Au lieu de cela, sa couleur est modifiée selon la formule suivante :

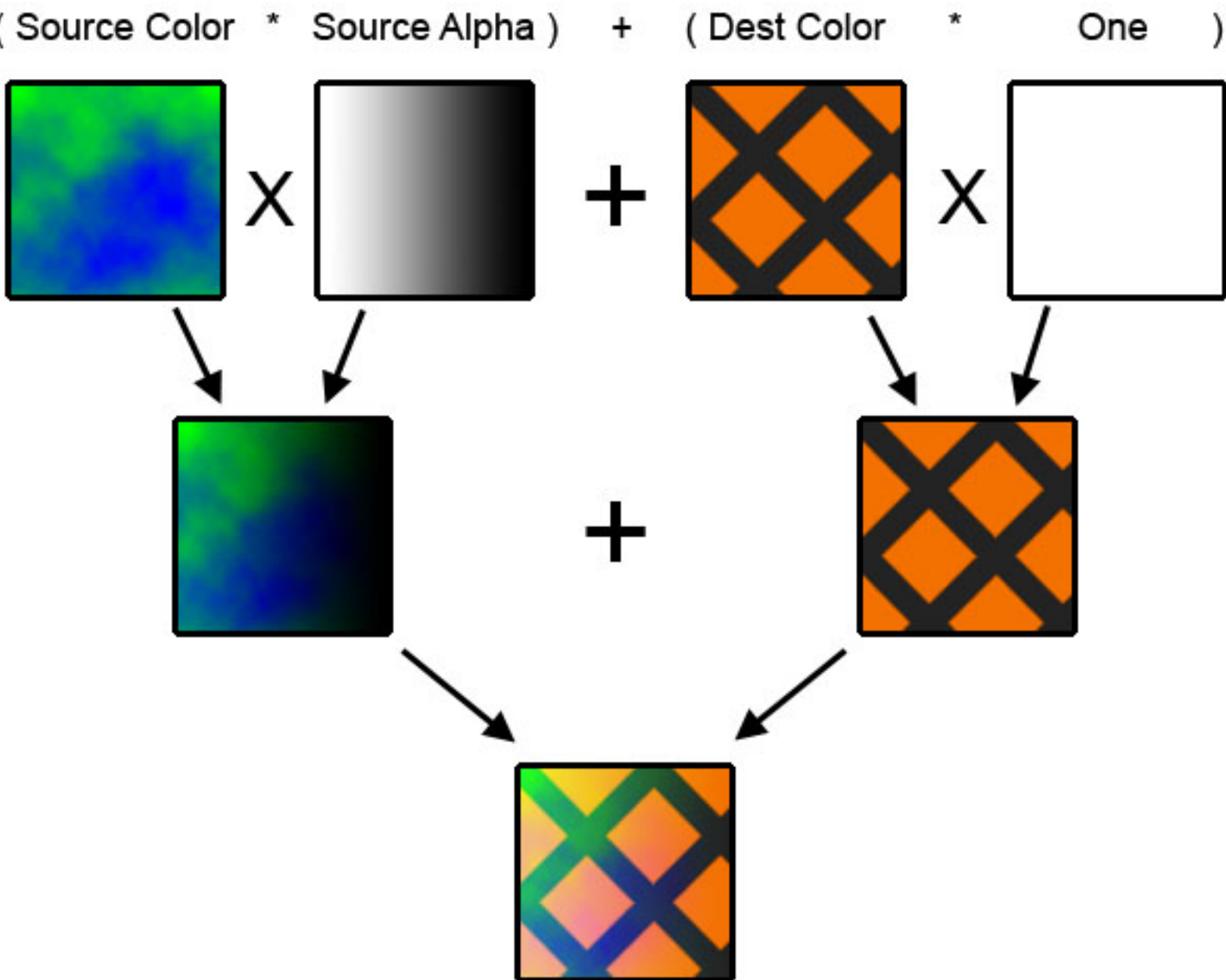
```
outputPixel = ( SourceColor*SourceBlendFactor ) BlendOp ( DestColor*DestinationBlendFactor )
```

SourceColor est un pixel rendu quand on trace le quad.
 DestColor est le pixel qui existe déjà dans le framebuffer (du dessin du personnage)
 BlendOp est le BlendOp de la structure de mélange SrcAlphaBlendingAdd
 SourceBlendFactor est SrcBlend dans la structure de mélange SrcAlphaBlendingAdd
 DestinationBlendFactor est DestBlend dans la structure de mélange SrcAlphaBlendingAdd

En utilisant la structure **SrcAlphaBlendingAdd** ci-dessus, nous pouvons traduire l'équation en la suivante :

```
OutputPixel = ( SourceColor.rgb * SourceColor.aaa ) ADD ( DestColor.rgb * (1,1,1) )
```

Graphiquement, cela peut se rendre par le diagramme suivant :



Par conséquent, la couleur produite dépend de la couleur qui était déjà dans la mémoire d'image au moment du tracé. Des sélections différentes dans le menu déroulant entraîneront différentes techniques de dessin du quadrilatère, et par conséquent différents *blend states*. Chaque état *blend state* change les variables de l'équation, aussi veillez à toutes les expérimenter. Voyez si vous pouvez prédire ce qui arrivera juste en regardant le *blend state*.

IV-14-d - Rasterizer States

Vous aurez peut-être noté que quelle que soit la façon dont l'état *blend state* a été choisi, la figure au milieu paraissait un peu pauvre. Certains polygones apparaissaient là où ils ne devraient pas être. Parfois, les jambes avaient l'air de tubes coupés en deux. Cela nous amène à notre objet état suivant : le *Rasterizer State*.

L'état *rasterizer state* (**ID3D10RasterizerState** quand on l'utilise directement à partir de l'interface de programmation et **RasterizerState** quand on l'utilise avec FX) contrôle comment les triangles est effectivement rendus à l'écran. Contrairement au chapitre précédent sur les *blend states*, nous n'allons pas contrôler les états rasterizer states via l'interface FX. Cela ne veut pas dire qu'on ne peut pas les utiliser via le système FX. En fait, il est juste aussi facile d'utiliser les états *rasterizer states* via FX qu'il est facile d'utiliser les *blend states* via FX. Toutefois, nous allons en profiter pour en apprendre davantage sur la gestion d'état non-FX grâce aux interfaces de programmation Direct3D 10.

Comme son nom l'indique, l'état *rasterizer state* contrôle comment les polygones sont rastérisés à l'écran. Quand le didacticiel démarre, les polygones ne sont pas triés : que les polygones pointent dans votre direction ou dans la direction opposée, ils sont tous dessinés de la même façon. C'est pourquoi certains polygones sombres "traversent" la figure. Pour modifier ce comportement, nous devons d'abord créer un *Rasterizer State* qui définira le comportement que nous voulons. Puisque nous faisons cela par l'interface de programmation et pas par l'interface FX, le processus est un peu différent des *blend states*. Le code qui suit provient de la fonction **LoadRasterizerStates** :

```

D3D10_FILL_MODE fill[MAX_RASTERIZER_MODES] =
{
    D3D10_FILL_SOLID,
    D3D10_FILL_SOLID,
    D3D10_FILL_SOLID,
    D3D10_FILL_WIREFRAME,
    D3D10_FILL_WIREFRAME,
    D3D10_FILL_WIREFRAME
};
D3D10_CULL_MODE cull[MAX_RASTERIZER_MODES] =
{
    D3D10_CULL_NONE,
    D3D10_CULL_FRONT,
    D3D10_CULL_BACK,
    D3D10_CULL_NONE,
    D3D10_CULL_FRONT,
    D3D10_CULL_BACK
};

for( UINT i=0; i<MAX_RASTERIZER_MODES; i++ )
{
    D3D10_RASTERIZER_DESC rasterizerState;
    rasterizerState.FillMode = fill[i];
    rasterizerState.CullMode = cull[i];
    rasterizerState.FrontCounterClockwise = true;
    rasterizerState.DepthBias = false;
    rasterizerState.DepthBiasClamp = 0;
    rasterizerState.SlopeScaledDepthBias = 0;
    rasterizerState.DepthClipEnable = true;
    rasterizerState.ScissorEnable = false;
    rasterizerState.MultisampleEnable = false;
    rasterizerState.AntialiasedLineEnable = false;
    pd3dDevice->CreateRasterizerState( &rasterizerState, &g_pRasterStates[i] );

    g_SampleUI.GetComboBox(IDC_SCENERASTERIZER_MODE)->AddItem( g_szRasterizerModes[i], (void*)
(UINT64)i );
}
    
```

Tout ce que ce code fait, c'est remplir une structure **D3D10_RASTERIZER_DESC** avec les informations nécessaires puis appeler **ID3D10Device::CreateRasterizerState** pour placer un pointeur sur l'objet **ID3D10RasterizerState**. La boucle crée un état pour chaque type d'état Rasterizer State que nous voulons montrer. Notez que le premier état qui utilise **D3D10_FILL_SOLID** comme mode de remplissage et **D3D10_CULL_NONE** comme mode d'élimination est ce qui donne un aspect bizarre à notre figure. Pour changer cela, nous pouvons choisir le deuxième état *rasterizer state* du menu déroulant *Scene Rasterizer Mode*.

Cet objet pourra ensuite être utilisé pour régler l'état avant le rendu. Voici comment l'état rasterizer state actuellement sélectionné est réglé dans **OnD3D10FrameRender** :

```

//
// Actualise le mode d'élimination (méthode non-FX)
    
```

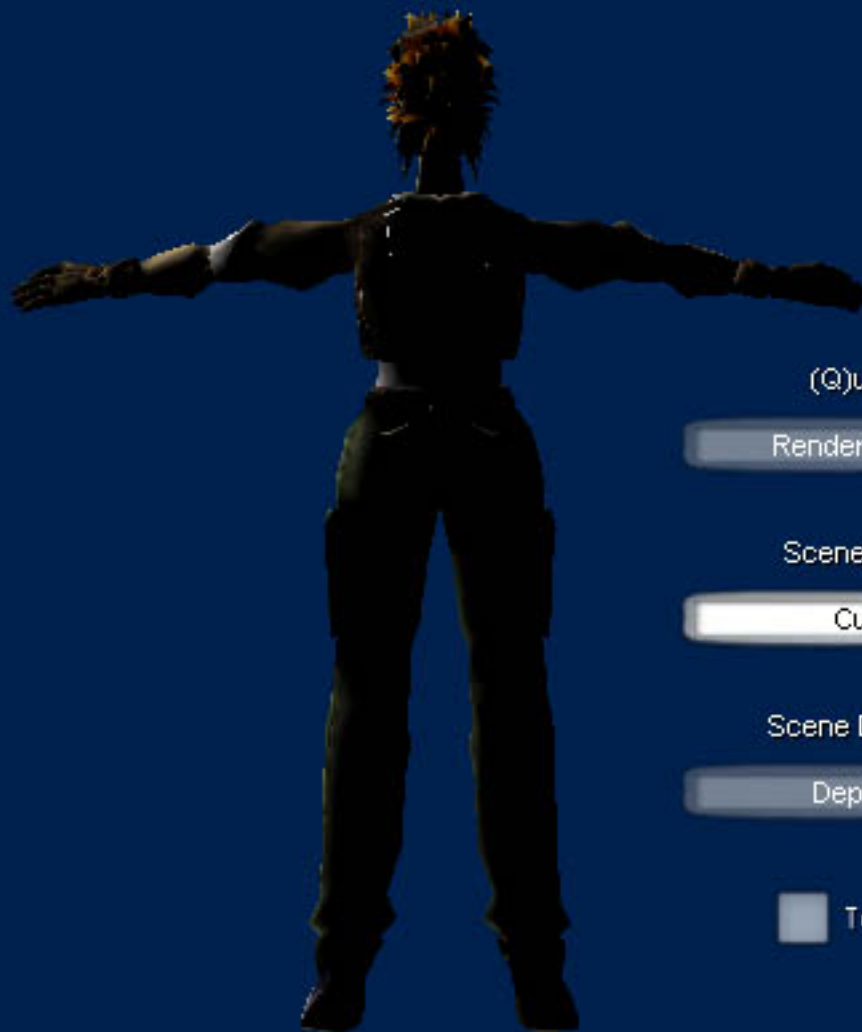
```
//
pd3dDevice->RSSetState(g_pRasterStates[ g_eSceneRasterizerMode ] );
```

Il faut noter que tout réglage d'état "persiste" à travers toutes les opérations de tracé suivantes jusqu'à ce qu'un état différent soit réglé ou que le device soit détruit. Cela signifie que lorsqu'on règle l'état *Rasterizer State* dans **OnD3D10FrameRender**, ce même *Rasterizer State* est appliqué à toutes les opérations de tracé qui viendront par la suite jusqu'à réglage d'un *Rasterizer State* différent.

Quand nous avons choisi la deuxième option du menu déroulant *Scene Rasterizer Mode*, notre figure s'est aggravée et le quadrilatère a tout simplement disparu.

Tutorial14

D3D10 Vsync on (640x480), RGB8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO



Toggle full screen

Toggle REF (F3)

Change device (F2)

(Q)uad Render Mode

RenderQuadSrcAlphaAdd

Scene (R)asterizer Mode

CullFront/FillSolid

Scene Depth/(S)tencil Mode

DepthOff/StencilOff

Toggle Spinning

C'est parce que nous avons choisi un mode de destruction qui a dit à Direct3D 10 de ne pas dessiner les triangles qui pointent vers nous. L'état *Rasterizer State* suivant (**D3D10_CULL_BACK** et **D3D10_FILL_SOLID**) devrait nous

donner les résultats que nous voulons (à l'exception de quelques triangles noirs égarés autour de la tête dont nous discuterons au chapitre suivant).

Tutorial14

D3D10 Vsync on (640x480), RGB8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO



Toggle full screen

Toggle REF (F3)

Change device (F2)

(Q)uad Render Mode

RenderQuadSrcAlphaAdd

Scene (R)asterizer Mode

CullBack/FillSolid

Scene Depth/(S)tencil Mode

DepthOff/StencilOff

Toggle Spinning

En plus des triangles à dessiner ou non, les rasterizer states peuvent contrôler de nombreux aspects du rendu. N'hésitez pas à expérimenter les différentes sélections de Scene Rasterizer Mode ou même à changer le code dans **LoadRasterizerStates** pour voir ce qui se passe.

IV-14-e - Etats Depth Stencil

Dans le dernier chapitre, régler le mode d'élimination sur **D3D10_CULL_BACK** et le mode de remplissage sur **D3D10_FILL_SOLID** a donné une figure correcte au milieu. Toutefois, il restait quelques polygones noirs autour du visage. C'est dû au fait que ces polygones restaient classés comme pointant vers l'avant bien qu'ils soient de l'autre

côté de la tête. C'est là que l'état *Depth Stencil State* entre en jeu. Cet objet état contrôle en fait deux différentes parties du pipeline Direct3D 10. La première est le **Depth Buffer**.

A la base, le **depth buffer** stocke des valeurs représentant la distance du pixel par rapport au plan de vue. Plus la valeur est grande, plus la distance est grande. Le **depth buffer** a la même résolution (hauteur et largeur) que le backbuffer. En soi, il ne semble pas très utile dans notre situation. Pourtant, l'état *depth stencil* nous permet de modifier le pixel actuel en fonction de la profondeur du pixel déjà situé à cet endroit dans le framebuffer.

Les états *Depth Stencil States* sont créés dans la fonction **LoadDepthStencilStates**. Cette fonction est similaire aux **LoadRasterizerStates**. Nous allons discuter de certains paramètres de la structure **D3D10_DEPTH_STENCIL_DESC**, le plus important d'entre eux étant **DepthEnable**. Il détermine si *Depth Test* fonctionne. Comme on peut le voir, quand l'application démarre, le *Depth Test* est désactivé. C'est pourquoi les mèches de cheveux dirigées vers l'avant à l'arrière de la tête apparaissent encore.

Régler l'état *Depth Stencil State* est également très similaire au réglage du *Rasterizer State* à une exception près : il y a un paramètre supplémentaire. Ce paramètre est le **StencilRef** qui sera expliqué au prochain chapitre.

```
//
// Actualise les états Depth Stencil States (méthode non-FX)
//
pd3dDevice->OMSetDepthStencilState(g_pDepthStencilStates[ g_eSceneDepthStencilMode ], 0);
```

Le deuxième état *Depth Stencil State* dans le menu déroulant *Scene Depth/Stencil Mode* active le *Depth Test*. Quand vous activez le *Depth Test*, il vous faut dire à Direct3D 10 quel type de test effectuer. L'équation se ramène à ceci :

```
DrawThePixel? = DepthOfCurrentPixel D3D10_COMPARISON CurrentDepthInDepthBuffer

D3D10_COMPARISON est la fonction de comparaison dans le paramètre DepthFunc de
D3D10_DEPTH_STENCIL_DESC
```

En substituant la valeur de **DepthFunc** du deuxième état *Depth Stencil State*, on obtient ceci pour l'équation :

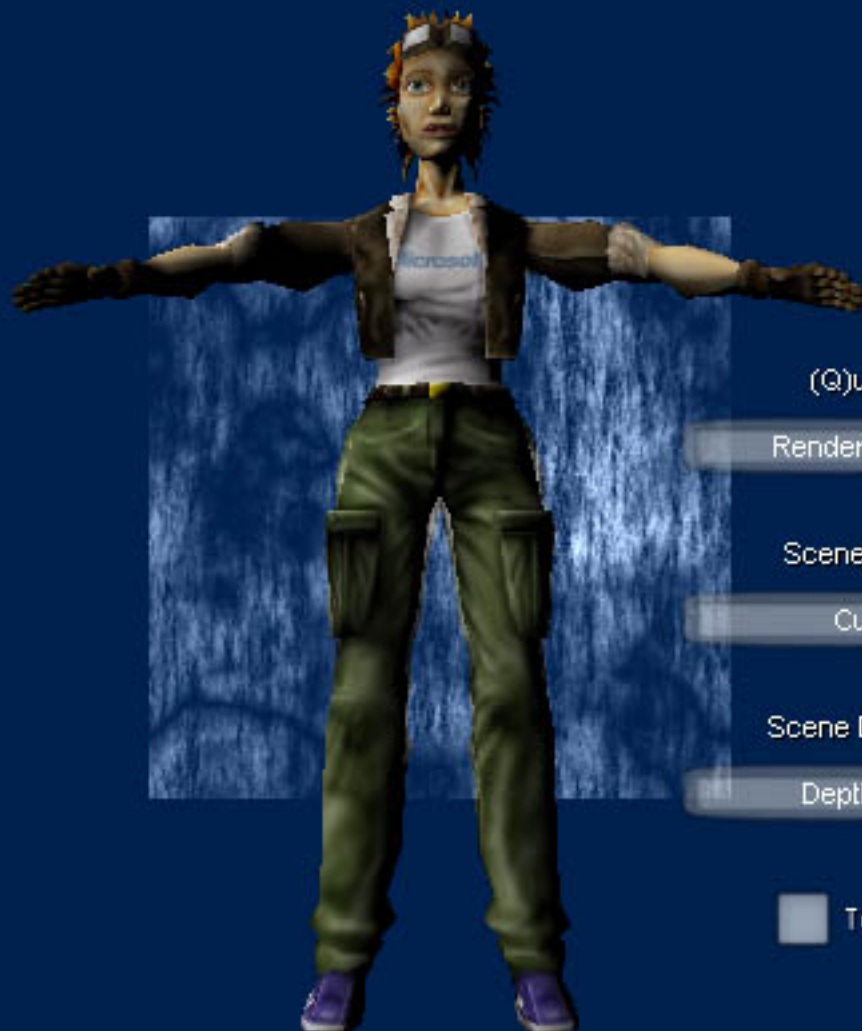
```
DrawThePixel? = DepthOfCurrentPixel D3D10_COMPARISON_LESS CurrentDepthInDepthBuffer
```

En français, cela veut dire "Tracer le pixel actuel seulement si sa profondeur est INFÉRIEURE à la profondeur déjà stockée à cet endroit". De plus, nous fixons le **DepthWriteMask** à **D3D10_DEPTH_WRITE_MASK_ALL** pour garantir que notre profondeur est entrée dans le *depth buffer* à cet endroit (si nous passons le test) afin que tout pixel ultérieur soit plus proche que notre valeur de profondeur à dessiner.

Cela entraîne que la figure est dessinée sans aucune marque noire. Les mèches de cheveux dirigées vers l'avant à l'arrière de la tête ne transparissent jamais. Soit elles sont écrasées par des triangles plus proches à l'avant du visage, soit elles ne sont jamais dessinées parce que les triangles proches ont une valeur de profondeur plus proche. Vous noterez aussi que cela révèle quelque chose à propos du quadrilatère : pendant tout ce temps, il a eu l'air d'être devant la figure. Maintenant, on s'aperçoit qu'en fait, il coupe la figure.

Tutorial14

10 Vsync on (640x480), RGB8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO



Toggle full screen

Toggle REF (F3)

Change device (F2)

(Q)uad Render Mode

RenderQuadSrcAlphaAdd

Scene (R)asterizer Mode

CullBack/FillSolid

Scene Depth/(S)tencil Mode

DepthLess/StencilOff

Toggle Spinning

Essayez de jouer avec les valeurs **DepthFunc** et voyez ce qui se passe.

IV-14-f - Rendu avec le Stencil Buffer

Les paramètres Stencil font partie de l'état *Depth Stencil State*, mais, en raison de leur complexité, ils méritent leur propre chapitre.

On peut se représenter le **Stencil Buffer** comme un tampon comme le **Depth Buffer**. Comme le **Depth Buffer**, il a les mêmes dimensions que la cible de rendu. Néanmoins, au lieu de stocker des valeurs de profondeur, le **Stencil Buffer** stocke des valeurs entières. Ces valeurs n'ont de sens que pour l'application. La partie *Stencil* de l'état *Depth Stencil State* détermine comment ces valeurs se retrouvent ici et ce qu'elles signifient pour l'application.

Comme le *Depth Test* du **Depth Buffer**, le **Stencil Buffer** a un *Stencil Test*. Ce *Stencil Test* peut comparer la valeur entrante du pixel à la valeur actuellement contenue dans le **Stencil Buffer**. Toutefois, il peut aussi augmenter cette comparaison avec des résultats issus du *Depth Test*. Passons en revue un des réglages de *Stencil Test* pour voir ce qu'il signifie. Les informations suivantes proviennent de la cinquième itération de la boucle dans la fonction **LoadDepthStencilStates**. Elles reflètent le cinquième état *Depth Stencil State* dans le menu déroulant *Scene Depth/Stencil Mode* :

```
D3D10_DEPTH_STENCIL_DESC dsDesc;
dsDesc.DepthEnable = TRUE;
dsDesc.DepthWriteMask = D3D10_DEPTH_WRITE_MASK_ALL;
dsDesc.DepthFunc = D3D10_COMPARISON_LESS;

dsDesc.StencilEnable = TRUE
dsDesc.StencilReadMask = 0xFF;
dsDesc.StencilWriteMask = 0xFF;

// Stencil operations if pixel is front-facing
dsDesc.FrontFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilDepthFailOp = D3D10_STENCIL_OP_INCR;
dsDesc.FrontFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilFunc = D3D10_COMPARISON_ALWAYS;

// Stencil operations if pixel is back-facing
dsDesc.BackFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilDepthFailOp = D3D10_STENCIL_OP_INC;
dsDesc.BackFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilFunc = D3D10_COMPARISON_ALWAYS;
```

Le précédent chapitre nous a enseigné que le *Depth Test* est conçu pour faire en sorte que tout pixel soit plus proche du plan que la précédente valeur de profondeur à cet endroit. Toutefois, maintenant, nous avons activé le *Stencil Test* en réglant **StencilEnable** sur *TRUE*. Nous avons également réglé **StencilReadMask** et **StencilWriteMask** à des valeurs qui garantissent que nous lisons et écrivons tous les bits du **stencil buffer**.

Maintenant, on arrive au cœur des réglages Stencil. L'opération stencil peut avoir différents effets selon que le triangle rasterisé pointe vers l'avant ou vers l'arrière (rappelez-vous que l'on peut choisir de ne dessiner que les polygones qui pointent vers l'avant ou vers l'arrière en utilisant le Rasterizer State). Par souci de simplicité, nous utiliserons les mêmes opérations aussi bien pour les polygones pointant vers l'avant et vers l'arrière. Aussi ne discutera-t-on que des opérations **FrontFace**. **FrontFace.StencilFailOp** est réglé sur **D3D10_STENCIL_OP_KEEP**. Cela signifie que si le *Stencil test* échoue, nous garderons la valeur actuelle dans le **stencil buffer**. **FrontFace.StencilDepthFailOp** est réglé sur **D3D10_STENCIL_OP_INC**. Cela signifie que si le *Depth test* échoue, nous incrémenterons la valeur dans le **stencil buffer** de 1. **FrontFace.StencilPassOp** est réglé sur **D3D10_STENCIL_OP_KEEP**. Cela signifie que si le test Stencil réussit, nous gardons la valeur actuelle. Finalement, **FrontFace.StencilFunc** représente la comparaison utilisée pour déterminer si nous réussissons le *Stencil Test* (pas le *Depth test*). Ce peut être n'importe quelle valeur de **D3D10_COMPARISON**. Par exemple, **D3D10_COMPARISON_LESS** ne réussirait le *Stencil test* que si la valeur actuelle de *Stencil* est inférieure à la valeur de *Stencil* attitrée dans le **Stencil Buffer**. Mais d'où vient cette valeur de test actuelle ? C'est le deuxième paramètre de la fonction **OMSetDepthStencilState** (le paramètre *StencilRef*).

```
//
// Actualise les états Depth Stencil States (méthode non-FX)
//
pd3dDevice->OMSetDepthStencilState(g_pDepthStencilStates[ g_eSceneDepthStencilMode ], 0);
```

Pour cet exemple, le paramètre **StencilRef** n'a pas d'importance dans la fonction de comparaison *Stencil*. Pourquoi ? Parce que nous avons réglé **StencilFunc** sur **D3D10_COMPARISON_ALWAYS**, ce qui signifie que nous réussissons TOUJOURS le test.

Vous aurez peut-être déjà déterminé en regardant cet état *Depth Stencil State* que tout pixel qui ECHOUE au *Depth Test* causera l'incrémentement de la valeur contenue dans le **Stencil Buffer** (pour cet emplacement dans le buffer). Si nous remplissons le stencil buffer de zéros (ce que nous faisons au début de **OnD3D10FrameRender**), alors toute valeur non nulle dans le **stencil buffer** est un emplacement où un pixel a échoué au *Depth test*.

Si nous pouvions effectivement visualiser ce **Stencil Buffer**, nous pourrions déterminer tous les emplacements dans la scène où nous avons essayé de dessiner quelque chose, mais où autre chose était plus proche. Par essence, nous pourrions dire où nous avons essayé de dessiner quelque chose derrière un objet qui était déjà là.

Malheureusement, il n'existe aucune fonction Direct3D 10 appelée **ShowMeTheStencilBuffer**. Heureusement, nous pouvons créer une technique FX qui fait la même chose. Pour montrer que l'état *Depth Stencil State* peut se régler tout aussi facilement par le système FX que par les interfaces de programmation Direct3D 10, nous montrerons comment utiliser un état *Depth Stencil State* sous FX pour visualiser le contenu du **stencil buffer**.

Après que la figure et le quadrilatère aient tous les deux été dessinés, nous rendons un autre quadrilatère qui couvre toute la fenêtre d'affichage. Parce que nous couvrons toute la fenêtre d'affichage, nous pouvons accéder à tous les pixels qui pourraient éventuellement être touchés par nos précédentes opérations de rendu. La technique utilisée pour rendre ce quadrilatère est **RenderWithStencil** dans le fichier FX.

```

technique10 RenderWithStencil
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, ScreenQuadVS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, QuadPS() ) );

        SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
        SetDepthStencilState( RenderWithStencilState, 0 );
    }
}
    
```

Notez qu'en plus de régler l'état *Blend State*, nous réglons aussi l'état *Depth Stencil State* sur **RenderWithStencilState**. Le deuxième paramètre règle la valeur **StencilRef** à 0. Jetons un œil sur **RenderWithStencilState** tel que défini en haut du fichier FX.

```

DepthStencilState RenderWithStencilState
{
    DepthEnable = false;
    DepthWriteMask = ZERO;
    DepthFunc = Less;

    // Réglage des états stencil
    StencilEnable = true;
    StencilReadMask = 0xFF;
    StencilWriteMask = 0x00;

    FrontFaceStencilFunc = Not_Equal;
    FrontFaceStencilPass = Keep;
    FrontFaceStencilFail = Zero;

    BackFaceStencilFunc = Not_Equal;
    BackFaceStencilPass = Keep;
    BackFaceStencilFail = Zero;
};
    
```

La première chose à noter est que c'est très similaire au **D3D10_DEPTH_STENCIL_DESC** que vous trouveriez dans la fonction **LoadDepthStencilStates** du fichier cpp. Les valeurs sont différentes, mais les membres sont les mêmes. D'abord, nous désactivons le test de profondeur. Ensuite, nous désactivons le *Stencil test*, MAIS nous réglons **StencilWriteMask** à 0. Nous ne voulons que lire la valeur *stencil*. Enfin, nous réglons **StencilFunc** sur *Not_Equal*, **StencilPass** sur *Keep* et **StencilFail** sur *Zero* à la fois pour les faces avant et arrière. Rappelez-vous que le *Stencil Test* teste la valeur entrante actuelle avec la valeur déjà stockée dans le **Stencil Buffer**. Dans ce cas, nous avons réglé la valeur entrante (**StencilRef**) à 0 quand nous avons appelé **SetDepthStencilState(RenderWithStencilState, 0)**. Par conséquent, nous pouvons traduire cet état pour dire que le test stencil réussit quand le **Stencil Buffer** n'est pas égal à 0 (**StencilRef**). En cas de réussite, nous obtenons de conserver le pixel entrant du *pixel shader* et de l'écrire dans le *frame buffer*. En cas d'échec, nous jetons le pixel entrant et il ne sera jamais dessiné. Donc, notre quadrilatère qui couvre tout l'écran ne sera dessiné que là où **Stencil Buffer** n'est pas égal à zéro.

Tutorial14

10 Vsync on (640x480), RGB8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO



Toggle full screen

Toggle REF (F3)

Change device (F2)

(Q)uad Render Mode

RenderQuadSrcAlphaAdd

Scene (R)asterizer Mode

CullBack/FillSolid

Scene Depth/(S)tencil Mode

DepthLess/StencilIncOnFail

Toggle Spinning

Choisir différents articles dans le menu déroulant *Scene Depth/Stencil Mode* change la façon dont **Depth Buffer** et **Stencil Buffer** se remplissent, et donc à quel point le quadrilatère couvrant l'écran tracé en dernier apparaît effectivement à l'écran. Certaines de ces combinaisons ne donneront aucun résultat. Certaines donneront des effets bizarres. Essayez de trouver lesquelles feront quoi avant de les appliquer.