

Guide de programmation Direct3D 10

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

Date de publication : 16/11/2008

Dernière mise à jour : 16/11/2008

Traduction d'un article original de la Library MSDN US.
Le guide de programmation contient des information sur comment utiliser les pipeline programmable Direct3D 10 afin de creer des graphismes 3D temps réel pour faire des jeux aussi bien que des application scientifique ou des applications bureautique.

| | |
|---|----|
| I - Caractéristiques de l'API (Direct3D 10)..... | 3 |
| II - Étapes de Pipeline (Direct3D 10)..... | 4 |
| III - Ressources (Direct3D 10)..... | 6 |
| IV - HLSL..... | 7 |
| V - Effets (Direct3D 10)..... | 8 |
| VI - Considérations sur le passage de Direct3D 9 à Direct3D 10 (Direct3D 10)..... | 9 |
| VI-1 - Aperçu des Principaux Changements Structurels dans Direct3D 10..... | 9 |
| VI-1-a - Elimination des Fonctions Fixes..... | 9 |
| VI-1-b - Validation de Temps de Création d'Application Objet..... | 10 |
| VI-2 - Abstractions / Séparation de Dispositifs..... | 10 |
| VI-2-a - Elimination directe de Dépendances Direct3D 9..... | 10 |
| VI-3 - Astuces pour résoudre rapidement les Problèmes de Construction d'Applications..... | 11 |
| VI-3-a - Correction de Types Direct3D 9..... | 11 |
| VI-3-b - Résolution de Problèmes de Liens..... | 11 |
| VI-3-c - Simulation de CAPs..... | 11 |
| VI-4 - Piloter l'interface API Direct3D 10..... | 11 |
| VI-4-a - Création de Ressource..... | 11 |
| VI-4-b - Vues..... | 12 |
| VI-4-c - Comparatif de l'Accès Statique et Dynamique aux Ressources..... | 12 |
| VI-4-d - Effets de Direct3D 10..... | 12 |
| VI-4-e - HLSL sans Effets..... | 12 |
| VI-4-f - Compilation de Shader..... | 12 |
| VI-4-g - Création de Ressources Shader..... | 13 |
| VI-4-h - Interface Shader Reflection Layer..... | 13 |
| VI-4-i - Présentation de l'Input Assembler - Vertex Shader / Liaison de Flux en Entrée..... | 13 |
| VI-4-j - Impact de l'Elimination des Codes de Shader morts..... | 13 |
| VI-4-k - Exemple de Structure en Entrée de Vertex Shader..... | 13 |
| VI-4-l - Création d'Objet Etat..... | 14 |
| VI-5 - Portage de Textures..... | 15 |
| VI-6 - Portage de Shaders..... | 17 |
| VI-6-a - Les Shaders Direct3D 10 sont autorisés sous HLSL uniquement..... | 17 |
| VI-6-b - Signatures et Liaison de Shaders..... | 17 |
| VI-6-c - Liens entre Shaders sous HLSL..... | 18 |
| VI-6-d - Tampons Constant..... | 18 |
| VI-7 - Différences supplémentaires de Direct3D 10 à surveiller..... | 18 |
| VI-7-a - Entiers en entrée..... | 18 |
| VI-7-b - Curseurs de souris..... | 18 |
| VI-7-c - Transposition de Texels en Pixels sous Direct3D 10..... | 18 |
| VI-7-d - Changements de Comportement dans le Comptage de Références..... | 19 |
| VI-7-e - Test de Niveau Coopératif..... | 19 |
| VII - Liaison de Bibliothèques Statiques et Dynamiques (Direct3D 10)..... | 20 |
| VII-1 - Fichiers DLL appropriés au Chargement des Bibliothèques..... | 20 |
| VII-2 - Redistribution des données binaires..... | 20 |
| VIII - Caractéristiques de Direct3D 10.1..... | 22 |
| VIII-1 - Accéder aux caractéristiques 10.1 sous Vista Gold et Vista Service Pack 1..... | 23 |
| VIII-2 - Accéder aux caractéristiques 10.1 sous Vista Service Pack 1 exclusivement..... | 23 |


I - Caractéristiques de l'API (Direct3D 10)

Le pipeline graphique Direct3D 10 représente un changement architectural fondamental, une reconstruction dès les fondations du hardware et du software pour alimenter la prochaine génération de jeux et d'applications 3D multimédia. Il utilise le modèle Windows Vista Display Driver Model (WDDM) qui permet des améliorations de performances et de comportement tels que la mémoire virtuelle GPU.

Les développeurs familiers de Direct3D 9 découvriront toute une série d'améliorations fonctionnelles et de performances sous Direct3D 10, y compris :

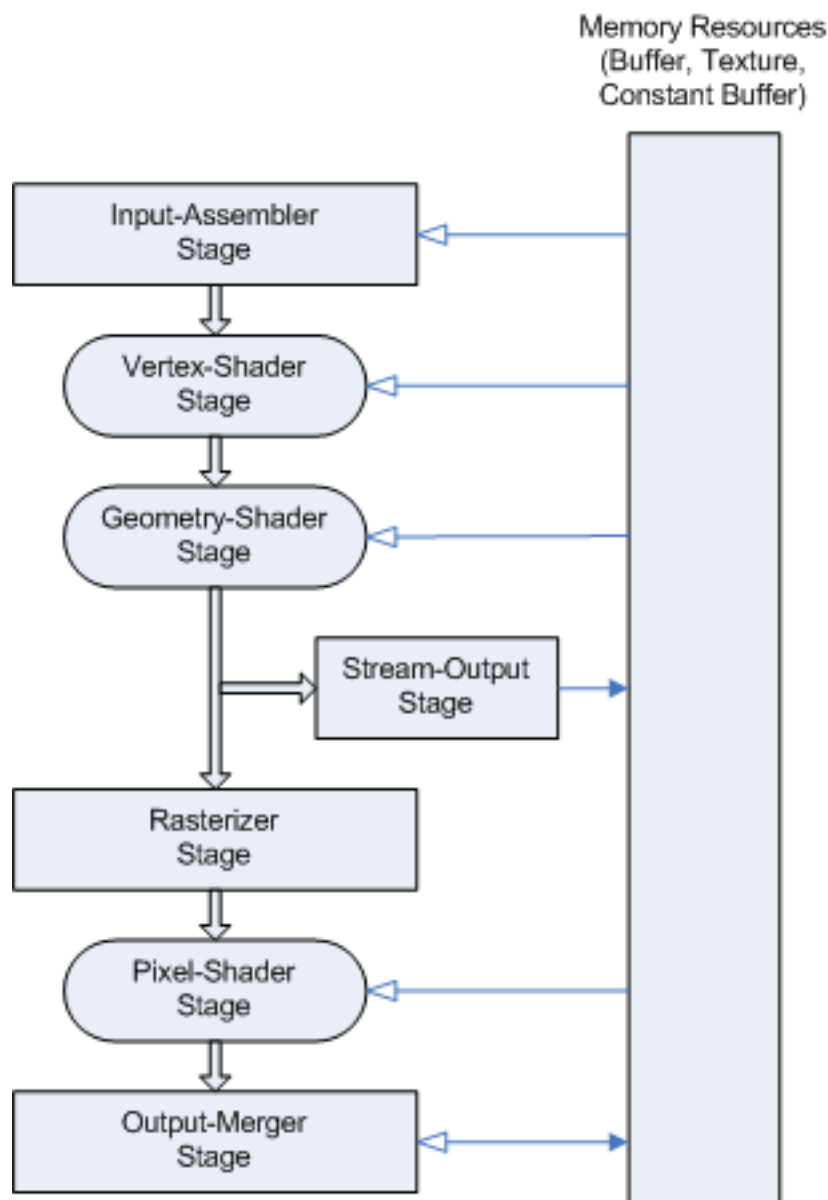
- La capacité de traiter des primitives entières (avec continuité), d'amplifier et de désamplifier les données dans la nouvelle étape  **geometry shader**.
- La capacité d'envoyer en mémoire les données des points générées par le pipeline en utilisant l'étape  **stream-output**.
- L'organisation de l'état du pipeline en 5  **objets états** immuables, permettant une configuration rapide du pipeline.
- L'organisation de shader constants en  **constant buffers**, ce qui minimise la déperdition de bande passante pour l'apport de données shader-constant.
- La capacité d'effectuer des échanges et des organisations "per-primitive material" en utilisant un geometry shader.
- De nouveaux  **types de ressource** (y compris des texture arrays qui peuvent être indexés à partir de shaders) et formats de ressource.
- Une généralisation accrue de l'accès aux ressource grâce à une  **vue**.
- Les capability bits (caps) des anciens systèmes hardware ont été éliminés en faveur d'un riche jeu de fonctionnalités garanties qui ciblent le hardware de classe Direct3D 10 (minimum).
-  **Layered Runtime** - Direct3D 10 API est conçu avec des couches, en partant de la fonctionnalité de base au cœur et des fonctionnalités optionnelles et d'assistance au développeur (debug, etc.) dans les couches externes.
- Intégration HLSL complète □ Tous les shaders Direct3D 10 sont écrits en HLSL et implémentés avec le  **common shader core**.
- Une augmentation du nombre de render targets, textures et samplers. Il n'y a pas de limite de longueur des shaders.
- Opérations integer et bitwise sur les shaders.
- Relecture d'une depth stencil surface ou d'une ressource multisamplée, une fois qu'elle n'est plus liée comme render target.
- Supporte l'alpha-to-coverage multisamplée.

Il existe des différences de comportement supplémentaires dont les développeurs sous Direct3D 9 devraient aussi être conscients (cf.  **Considérations sur le passage de Direct3D 9 à Direct3D 10**).



Voici une liste de caractéristiques de Direct3D 9 qui soit ne sont plus supportées, soit ont été révisées dans Direct3D 10 (cf.  **Caractéristiques supprimées**).






II - Étapes de Pipeline (Direct3D 10)

Le pipeline programmable Direct3D 10 est conçu pour générer des graphiques pour les applications de jeu en temps réel. Le diagramme conceptuel ci-dessous illustre le flux de données de l'input à l'output à travers chacun des étapes programmables.



Toutes les étapes peuvent se configurer avec l'interface de programmation applicative (API). Les étapes comportant des common shader cores (les blocs rectangulaires arrondis) peuvent se programmer en langage de programmation HLSL. Comme vous pouvez le voir, cela rend le pipeline extrêmement souple et adaptable. Le but de chaque étape est indiqué ci-dessous :

-  **Etape Input-Assembler** - L'étape input-assembler est responsable de l'alimentation en données (triangles, lignes et points) du pipeline.
-  **Etape Vertex-Shader** - L'étape vertex-shader traite les sommets des polygones, en effectuant typiquement des opérations telles que des transformations, des habillages et des éclairages. Un vertex shader prend toujours en entrée des points de données (vertex) du pipeline et produit en sortie des points de données (vertex).

-  **Etape Geometry-Shader** - L'étape geometry-shader traite des primitives entières. Son entrée est une primitive entière (c'est à dire trois sommets pour un triangle, deux pour une ligne, ou un seul point un point). De plus, chaque primitive peut aussi inclure les données de sommets pour toute primitive d'élément adjacent. Cela peut inclure au maximum trois sommets supplémentaires pour un triangle ou deux supplémentaires pour une ligne. Le Geometry Shader admet aussi les amplifications et désamplifications limitées de géométrie. Partant d'une primitive donnée en entrée, le Geometry Shader peut écarter la primitive ou en émettre une ou plusieurs nouvelles.
-  **Etape Stream-Output** - L'étape stream-output est conçue pour le flux de données primitives depuis le pipeline vers la mémoire sur le chemin vers le rasterizer. On peut extraire les données et/ou les passer dans le rasterizer. Les données extraites vers la mémoire peuvent être recirculées dans le pipeline comme données d'entrée ou relues depuis le CPU
-  **Etape Rasterizer** □ Le traceur par ligne (rasterizer) est responsable du découpage des primitives, de leur préparation pour le pixel shader et de la détermination de comment déclencher les pixel shaders.
-  **Etape Pixel-Shader** - L'étape pixel-shader reçoit les données interpolées pour une primitive et génère des données par pixel comme la couleur.
-  **Etape Output-Merger** - L'étape output-merger est responsable de la combinaison des différents types de données en sortie (valeurs de pixel shader, informations profondeur et stencil) avec le contenu des tampons de rendu cible et des tampons profondeur/stencil pour générer le résultat final du pipeline.

III - Ressources (Direct3D 10)

Une ressource est une zone mémoire à laquelle le pipeline Direct3D peut accéder. Afin que le pipeline accède efficacement à la mémoire, les données qui sont fournies au pipeline (comme une géométrie d'entrée, des ressources de shader, des textures, etc.) doivent être enregistrées dans une ressource. Il existe deux types de ressources dont dérivent toutes les ressources Direct3D : les tampons et les textures. Jusqu'à 128 ressources peuvent être actives à chaque étape du pipeline.







Typiquement, chaque application créera de nombreuses ressources. Des exemples de ressources incluent : vertex buffers, index buffer, constant buffer, textures et ressources shader. Il existe plusieurs options qui déterminent comment on peut utiliser les ressources. Vous pouvez créer des ressources fortement typées ou sans type ; vous pouvez contrôler si les ressources peuvent à la fois être lues et écrites ; vous pouvez rendre des ressources accessibles seulement au CPU, seulement au processeur graphique, ou aux deux. Naturellement, Il y aura un marchandage vitesse contre fonctionnalité - plus vous accordez de fonctionnalité à une ressource, moins vous pouvez attendre de performance.

Puisqu'une application utilise souvent de nombreuses textures, Direct3D introduit également le concept de tableau de texture pour simplifier la gestion des textures. Un tableau de texture contient une ou plusieurs textures (toutes de même type et de mêmes dimensions) qui peuvent être indexées depuis une application ou par des shaders. Les tableaux de texture vous permettent d'utiliser une unique interface avec de multiples index pour accéder à de nombreuses textures. Vous pouvez créer autant de tableaux de texture que vous en avez besoin pour gérer différents types de textures.

Une fois que vous avez créé les ressources que votre application utilisera, vous connectez ou liez chaque ressource aux étapes du pipeline qui les utilisera. Cela se fait en appelant une API de liaison, ce qui place un pointeur sur la ressource. Puisque plus d'une étape du pipeline peut avoir besoin d'accéder à la même ressource, Direct3D 10 introduit le concept de vue ressource (*resource view*). Un affichage identifie la portion d'une ressource à laquelle il est possible d'accéder. Vous pouvez créer m affichages d'une ressource et les lier à n étapes de pipeline, en supposant que vous suivez les règles de liaison pour les ressources partagées (sinon le runtime enverra des messages d'erreur pendant la compilation).

Une vue ressource fournit un modèle général d'accès à une ressource (textures, tampons, etc.). Puisque vous pouvez utiliser un affichage pour dire au runtime à quelles données accéder et comment y accéder, les vues ressource vous permettent de créer des ressources sans type. Autrement dit, vous pouvez créer des ressources pour une taille donnée au moment de la compilation, puis déclarer le type de données dans la ressource quand la ressource est liée au pipeline. Les affichages exposent de nombreuses capacités nouvelles pour utiliser les ressources, comme la possibilité de relire des surfaces profondeur/stencil dans le shader, ce qui génère des cubemaps dynamiques en une seule étape, et le rendu simultané de plusieurs coupes d'un volume.



Pour en savoir plus sur les types de ressources de base, les tableaux de texture, et comment créer et utiliser des ressources, voyez ces autres sujets :

-  **Types de Ressources**
-  **Choix d'une Ressource**
-  **Création de Ressources tampon**
-  **Création de Ressources texture**
-  **Copier et Accéder à des Données Ressources**
-  **Structure et Affichages de Mémoire**
-  **Compression de bloc**
-  **Table des Limites de Ressource**
-  **Système de coordonnées**
-  **Mapping de Formats anciens**

IV - HLSL

HLSL signifie High Level Shading Language, un langage de programmation pour DirectX. En utilisant HLSL, vous pouvez créer des shaders programmables en langages de type C pour le pipeline Direct3D.





HLSL a été créé à partir de DirectX 8 pour initialiser le pipeline 3D programmable. Sous DirectX 8, le pipeline était programmé avec une combinaison d'instructions d'assemblage, d'instructions HLSL et de déclarations de fonctions fixes. Avec l'introduction de l'API Direct3D 10, le pipeline est désormais 100% programmable de façon uniquement virtuelle, en n'utilisant que HLSL ; en fait, on n'utilise plus l'assemblage pour générer des codes shader avec Direct3D 10.

-  **Guide de programmation pour HLSL** □ Le guide de programmation contient des informations sur l'écriture des shaders, ainsi que sur la compilation, la création et la liaison de shaders au pipeline.
-  **Référence pour HLSL** □ La section de référence comprend un listing complet de la syntaxe du langage, ainsi que des fonctions intrinsèques qui sont incorporées dans le langage pour simplifier vos exigences de codage.

V - Effets (Direct3D 10)

Un effet DirectX est une collection d'état du pipeline, mis en place par des expressions écrites en HLSL, et une syntaxe spécifique au framework *effect*. Après avoir compilé un effet, utilisez les interfaces de programmation applicative (API) du framework *effect* pour obtenir le rendu. La fonctionnalité d'effet peut aller de quelque chose d'aussi simple qu'un vertex shader qui transforme la géométrie et un pixel shader qui produit une couleur homogène, jusqu'à une technique de rendu qui nécessite plusieurs passes, utilise toutes les étapes du pipeline graphique et manipule aussi bien un état de shader qu'un état de pipeline non associé aux shaders programmables.

La première étape est d'organiser l'état que vous voulez contrôler dans un effet. Cela inclut l'état de shader (capacité d'envoi en mémoire des données de points générées par le pipeline - vertices, geometry shaders et pixel shaders), la texture et l'état d'échantillonneur (*sampler*) utilisé par les shaders, et tout autre état de pipeline non-programmable. Vous pouvez créer un effet en mémoire sous la forme d'une chaîne texte, mais typiquement, la taille devient assez grande pour qu'il soit commode d'enregistrer l'état d'effet dans un fichier effet (un fichier texte dont le nom se finit par une extension *.fx*). Pour utiliser un effet, vous devez le compiler (pour vérifier la syntaxe HLSL ainsi que la syntaxe du framework *effect*), initialiser l'état de l'effet par des appels API, et modifier votre boucle de rendu pour appeler les API de rendu.

-  **Organiser un Etat dans un Effet**
-  **Interfaces Système Effet**
-  **Interfaces de Spécialisation**
-  **Rendu d'un Effet**

Un effet encapsule tout l'état de rendu nécessaire à un effet particulier dans une unique fonction de rendu appelée une technique. Une passe est un sous-ensemble d'une technique, qui contient l'état de rendu. Pour mettre en œuvre un effet de rendu en plusieurs passes, mettez en œuvre une ou plusieurs passes à l'intérieur d'une technique. Par exemple, disons que vous voulez rendre une géométrie avec un jeu de tampons profondeur/stencil, puis tracer quelques sprites au-dessus. Vous pourriez mettre en œuvre le rendu de géométrie dans la première passe et le tracé de sprite dans la seconde passe. Pour rendre l'effet, il vous suffit de rendre les deux passes dans votre boucle de rendu. Vous pouvez mettre en œuvre autant de techniques que vous voulez dans un effet. Bien sûr, plus grand est le nombre de techniques, plus longue est la durée de compilation de l'effet. Une façon d'exploiter cette fonctionnalité est de créer des effets avec des techniques conçues pour tourner sur différents hardwares. Cela permet à une application de réduire élégamment les performances basées sur les capacités détectées du hardware.

VI - Considérations sur le passage de Direct3D 9 à Direct3D 10 (Direct3D 10)

La page suivante fournit une description basique des différences clefs entre Direct3D 9 et Direct3D 10. Cette documentation est préliminaire et incomplète ; elle sert de prélude à un guide exhaustif de développement de portabilité et de plateforme croisée entre Direct3D 9 et Direct3D 10 qui sera fourni dans une future édition du kit de développement logiciel DirectX. Le plan ci-dessous donne un aperçu de départ pour aider les développeurs ayant l'expérience de Direct3D 9 à explorer et établir un rapport avec Direct3D 10.

VI-1 - Aperçu des Principaux Changements Structurels dans Direct3D 10

Le processus de rendu en utilisant l'appareil Direct3D 10 est structurellement similaire à celui de Direct3D 9.


- Créer une source de vertex stream
- Créer une présentation d'entrée en Direct3D 10 (créer une déclaration de vertex stream en Direct3D 9)
- Déclarer la topologie primitive
- Créer des textures
- Créer des objets états
- Créer des shaders
- Tracer

L'appel *Draw* (tracer) lie les opérations entre elles ; l'ordre des appels avec l'appel *Draw* est arbitraire. Les principales différences dans la conception de l'interface de l'API Direct3D 10 sont les suivantes :

- Elimination des Fonctions Fixes.
- Elimination des bits CAPS □ le jeu des caractéristiques de base de Direct3D 10 est garanti.
- Une gestion plus stricte de : l'accès aux ressources, l'état du dispositif, des shader constants, des liaisons de shaders (entrées et sorties vers les shaders) entre les étapes.
- Les changements du nom du point d'entrée de l'interface API reflète l'utilisation de la mémoire du processeur graphique virtuel (Map() au lieu de Lock()).
- On peut ajouter une couche debug au dispositif au moment de la création.
- La typologie primitive est désormais un état explicite (séparé de l'appel Draw).
- Les shader constants explicites sont désormais enregistrées dans des constant buffers.
- La création de shader se fait entièrement en HLSL. Le compilateur HLSL réside désormais dans le DLL primaire Direct3D 10.
- Nouvelle étape programmable - le geometry shader.
- Elimination de BeginScene()/EndScene().
- Les fonctionnalités 2D courantes, de focalisation et de gestion d'adaptation sont mises en œuvre dans un nouveau composant : DXGI

VI-1-a - Elimination des Fonctions Fixes

Il est parfois surprenant que même dans un dispositif Direct3D 9 qui exploite totalement le pipeline programmable, il subsiste un certain nombre de domaines qui dépendent du pipeline à fonctions fixes (FF). Les domaines les plus courants sont habituellement liés au rendu aligné sur l'espace écran pour l'Interface Utilisateur. C'est pour cette raison qu'il est probable que vous devrez construire un shader d'émulation FF ou un jeu de shaders qui vous fourniront les comportements de substitution nécessaires.

Cette documentation comprend un livre blanc contenant des sources de shader de substitution pour les comportements FF courants (cf.  **Echantillon d'Emulation de Fonctions Fixes**). Certains comportements de pixels par fonctions fixes comprenant un test alpha ont été transformés en shaders.

VI-1-b - Validation de Temps de Création d'Application Objet

Le pipeline Direct3D 10 a été reconçu intégralement en termes de hardware et de software avec pour intention primaire de réduire la perte de temps pour le CPU (au tracé). Pour réduire les coûts, à tous les types de données ont été assignés des objets respectifs avec des méthodes de création explicites fournies avec l'appareil lui-même. Cela permet la stricte validation des données au moment de la création d'objet plutôt qu'au moment de l'appel *Draw* comme c'est souvent le cas avec Direct3D 9.

VI-2 - Abstractions / Séparation de Dispositifs

Certaines applications, y compris des jeux, qui désirent supporter à la fois Direct3D 9 et Direct3D 10, ont besoin d'avoir des couches de rendu extraites du reste de la base de code. Il y a de nombreuses façons de le faire, mais la clef de tout cela, c'est la conception de la couche d'extraction pour les dispositifs Direct3D bas de gamme. Tous les systèmes doivent communiquer au hardware via la couche commune qui est conçue pour fournir au processeur graphique la gestion des ressources et des types de bas niveau.

VI-2-a - Elimination directe de Dépendances Direct3D 9

Pendant le portage de grandes bases de code précédemment testées, il est important de minimiser la quantité de changements dans le code pour les réduire à ce qui est absolument nécessaire à préserver les comportements précédemment testés dans le code. La meilleure pratique inclut la documentation claire de l'endroit où les articles changent, en utilisant des commentaires. Il est souvent utile d'avoir une norme de commentaire pour ce travail, ce qui permettra une navigation rapide dans la base de code.

Voici un exemple de listing de ligne unique standard / de commentaires de bloc de démarrage qui pourraient être utilisés pour ce travail :

```
// Direct3D 10 REMOVED
```

A utiliser là où des lignes / blocs de code ont été enlevés

```
// Direct3D 10 NEEDS UPDATE
```

Le commentaire *NEED UPDATE* suggère qu'il faudra avoir recours à un travail sur la nouvelle interface API lors de visites ultérieures du code pour la conversion du comportement. Un usage important de `assert(false)` devrait aussi être utilisé là où `Direct3D 10 NEEDS UPDATE` apparaît, pour vous garantir que vous ne faites pas tourner sans le savoir un code erroné

```
// Direct3D 10 CHANGED
```

Les zones où des changements majeurs sont survenus doivent être conservées pour s'y référer à l'avenir, mais mises en commentaire pour les extraire du code

```
// Direct3D 10 END
```

Qualificateur de fin de bloc de code

Avec plusieurs lignes de source, vous devriez aussi utiliser les commentaires de style C `/* */` mais y ajouter les commentaires de début et de fin qu'il faut de part et d'autre de ces zones.

VI-3 - Astuces pour résoudre rapidement les Problèmes de Construction d'Applications

VI-3-a - Correction de Types Direct3D 9

Il peut être utile d'insérer un fichier d'en-tête de haut niveau contenant des définitions / corrections pour les types de base de Direct3D 9 qui ne sont plus supportés par les en-têtes de Direct3D 10. Cela vous aidera à minimiser le nombre de changements dans le code et les interfaces là où il y a un jeu de correspondances directes d'un type Direct3D 9 au type Direct3D 10 nouvellement défini. Cette approche est également utile pour grouper les comportements de code dans un seul fichier source. Dans ce cas, c'est une bonne idée de définir des types indépendants de la version / à désignation générale qui décrivent des constructions courantes utilisées pour le rendu et couvrant à la fois les interfaces API de Direct3D 9 et Direct3D 10. Par exemple :

```
#if defined(D3D9)
typedef IDirect3DIndexBuffer9    IDirect3DIndexBuffer;
typedef IDirect3DVertexBuffer9  IDirect3DVertexBuffer;
#else //D3D10
typedef ID3D10Buffer            IDirect3DIndexBuffer;
typedef ID3D10Buffer            IDirect3DVertexBuffer
#endif
```

Parmi d'autres exemples spécifiques de Direct3D 10 :

```
typedef ID3D10TextureCube    IDirect3DCubeTexture;
typedef ID3D10Texture3D     IDirect3DVolumeTexture;
typedef D3D10_VIEWPORT      D3DVIEWPORT;
typedef ID3D10VertexShader  IDirect3DVertexShader;
typedef ID3D10PixelShader   IDirect3DPixelShader;
```

VI-3-b - Résolution de Problèmes de Liens

Il est conseillé de développer des applications Direct3D 10 et Windows Vista en utilisant la dernière version en date de Microsoft Visual Studio. Toutefois, il est possible de construire une application Windows Vista qui dépende de Direct3D 10 en utilisant la version antérieure 2003 de Visual Studio. Direct3D 10 est un composant de plateforme Windows Vista qui a des dépendances (comme avec le kit de développement logiciel plateforme Server 2003 SP1) sur la librairie suivante : *BufferOverflowU.lib* est nécessaire pour résoudre tout problème avec *buffer_security check linker*.

VI-3-c - Simulation de CAPs

Beaucoup d'applications contiennent des zones de code qui dépendent de données CAPS disponibles. Une solution de rechange pour cela consiste à corriger l'énumération et à forcer les CAPS à prendre des valeurs sensées. Prévoyez de revisiter par la suite les zones où il y a des dépendances envers des CAPS pour les éliminer complètement là où c'est possible.

VI-4 - Piloter l'interface API Direct3D 10

Ce chapitre se concentre sur les changements de comportement provoqués par l'interface API de Direct3D 10.

VI-4-a - Création de Ressource

L'interface API Direct3D 10 comporte des ressources conçues comme des types de tampons génériques qui ont des registres de liaison spécifiques de l'utilisation prévue. Cette conception a été choisie pour faciliter un accès quasi-universel aux ressources dans le pipeline pour des scénarii tels que le rendu avec un vertex buffer, suivi du tracé instantané des résultats sans interruption du CPU. L'exemple suivant montre l'allocation de vertex buffers et

d'un index buffer où vous pouvez voir que la description de ressource ne diffère que par les registres de liaison de ressource du processeur graphique.

L'interface API Direct3D 10 a fourni des méthodes d'aide à la texture pour créer explicitement des ressources de type de texture, mais comme vous pouvez l'imaginer, ce sont de véritables fonctions d'aide :

- *CreateTexture2D()*
- *CreateTextureCube()*
- *CreateTexture3D()*

Quand vous ciblez Direct3D 10, il est probable que vous voudrez allouer d'avantage d'objets pendant la durée de création de ressource que ce à quoi vous aviez l'habitude avec Direct3D 9. Cela deviendra particulièrement apparent avec la création de tampons de rendu cible et de Textures où vous aurez besoin de créer également un affichage pour accéder au tampon et pour placer la ressource sur le dispositif.

Tutoriel 1 : Les bases de Direct3D 10

VI-4-b - Vues

Une vue est une interface de type spécifique vers les données enregistrées dans un pixel buffer. Une ressource peut se voir allouer plusieurs vues en même temps, et cette caractéristique est mise en lumière dans l'exemple de Rendu à Simple Passe de Cubemap contenu dans ce kit de développement logiciel.


 **Une page du Guide du Programmeur sur l'Accès aux Ressources**

 **Exemple de CubeMap**

VI-4-c - Comparatif de l'Accès Statique et Dynamique aux Ressources

Pour obtenir la meilleure performance possible, les applications doivent répartir leur utilisation de données en termes de nature statique ou dynamique de ces données. Direct3D 10 a été conçu pour tirer avantage de cette approche, et en temps que tel, les règles d'accès aux ressources ont été rendues significativement plus strictes par rapport à Direct3D 9. Pour les ressources statiques, il vous faut idéalement peupler la ressource avec ses données pendant le temps de sa création. Si votre appareil a été conçu autour des points de conception Create, Lock, Fill, Unlock de Direct3D 9, vous pourriez repousser la population au-delà du moment de la création en utilisant une ressource d'activation et la méthode *UpdateSubResource* sur l'interface de ressource.

VI-4-d - Effets de Direct3D 10

L'utilisation du système des Effets de Direct3D 10 n'entre pas dans le cadre de cet article. Le système a été écrit pour tirer pleinement avantage des bénéfices architecturaux qu'apporte Direct3D 10. Reportez-vous au chapitre  **Effets (Direct3D 10)** pour plus de détails sur son utilisation.

VI-4-e - HLSL sans Effets


On peut piloter le pipeline de Direct3D 10 sans utiliser le système des Effets de Direct3D 10. Notez que dans ce cas, tous les constant buffers, shaders, échantillonneurs et liens de texture doivent être gérés par l'application elle-même. Reportez-vous au lien et aux chapitres suivants dans ce document pour plus de détails :

 **Exemple de Direct3D 10 sans Effet**


VI-4-f - Compilation de Shader

Le compilateur HLSL de Direct3D 10 apporte des améliorations à la définition du langage HLSL, et par conséquent il a la possibilité de fonctionner en 2 modes. Pour supporter totalement les fonctions intrinsèques et la sémantique

de style de Direct3D 9, la compilation doit être invoquée en utilisant le paramètre **COMPATIBILITY MODE** qui peut être spécifié sur une base par compilation.

On peut trouver les fonctions intrinsèques et sémantiques du langage HLSL spécifiques du modèle 4.0 de shader Direct3D 10 au lien référencé ci-dessous. Les changements les plus notables dans la syntaxe par rapport à HLSL sous Direct3D 9 sont dans le domaine de l'accès aux textures. La nouvelle syntaxe est la seule forme supportée par le compilateur hors du mode de compatibilité. Pour plus de détails, cf.  **HLSL**.

VI-4-g - Création de Ressources Shader

La création d'instances de shader compilées hors du système des Effets de Direct3D 10 se fait d'une façon très similaire à Direct3D 9. Toutefois, sous Direct3D 10, il est important de conserver la signature de Shader Input pour usage ultérieur. La signature est retournée par défaut comme élément du shader blob, mais on peut l'extraire pour réduire les exigences de mémoire si besoin est. Pour plus de détails, cf.  **Utiliser des Shaders sous Direct3D 10**.

VI-4-h - Interface Shader Reflection Layer

La shader reflection layer est l'interface par laquelle on peut obtenir des informations sur les exigences concernant les shaders. C'est particulièrement utile quand vous créez des liens d'Entrée sous Assembleur (voir plus bas) quand vous avez besoin de satisfaire les exigences en entrée des shaders pour garantir que vous fournissez la bonne structure en entrée pour le shader. Vous pouvez créer une instance de l'interface de la Reflection Layer en même temps que vous créez une instance d'un shader compilé.

VI-4-i - Présentation de l'Input Assembler - Vertex Shader / Liaison de Flux en Entrée

L'Input Assembler (IA) remplace la Déclaration de Vertex Stream de Direct3D 9 et la description de sa structure en est très similaire de forme. La principale différence qu'apporte l'IA est que l'objet de conception IA créé doit directement correspondre à un format spécifique de la signature d'entrée du shader. L'objet de correspondance créé pour lier le flux en entrée au shader peut s'utiliser à travers autant de shaders qu'on veut dès lors que la signature d'entrée du shader correspond à celle du shader utilisé pour créer la présentation d'entrée.

Pour piloter au mieux le pipeline avec des données statiques, vous devriez envisager les permutations de format du flux en entrée vers les possibles signatures d'entrée de shader, créer les instances d'objets de présentation IA aussi tôt que possible et les réutiliser là où c'est possible.

VI-4-j - Impact de l'Elimination des Codes de Shader morts

Le chapitre suivant détaille une différence significative entre Direct3D 9 et Direct3D 10 qui exigera probablement une manipulation délicate dans votre code machine. Les shaders qui contiennent des expressions conditionnelles voient souvent certains de leurs chemins de code être éliminés dans le cadre du processus de compilation. Sous Direct3D 9, deux type d'entrée entrée peuvent être supprimées (marquées comme supprimées) quand elles sont inutilisées : les signatures d'entrée (*signature inputs*, comme dans l'exemple ci-dessous) et les constantes d'entrée (*constant inputs*). Si la fin du buffer constant contient des entrées inutilisées, la taille déclarée dans le shader va refléter la taille du buffer contact sans les entrées inutilisées au final. Ce n'est plus le cas sous Direct3D 10, car l'intention est de permettre l'utilisation de la même Signature en Entrée parmi toutes les permutations possibles du code de shader sans invoquer un changement relativement coûteux d'input layout. Cela a un impact sur le moteur quand on manipule de grands shaders et que l'on crée des Input layout. Les éléments qui sont éliminés par optimisation de codes morts dans le compilateur doivent toujours être déclarés dans la Structure d'Entrée. L'exemple suivant montre cette situation :

VI-4-k - Exemple de Structure en Entrée de Vertex Shader

```
typedef struct
{
```

```
float4 pos: SV_Position;
float2 uv1 : Texcoord1;
float2 uv2 : Texcoord2; *
} VS_INPUT;
```

* L'élimination de code mort de Direct3D 9 éliminerait la déclaration dans le shader par suite de l'élimination de code mort conditionnel

```
float4x4 g_WorldViewProjMtx;
bool g_bLightMapped = false;

VS_INPUT main(VS_INPUT i)
{
    VS_INPUT o;
    o.pos = mul( i.pos, g_WorldViewProjMtx);
    o.uv1 = i.uv1;
    if ( g_bLightMap )
    {
        o.uv2 = i.uv2;
    }
}
```

Dans l'exemple ci-dessus, sous Direct3D 9, l'élément *uv2* serait éliminé en raison des optimisations de code mort dans le compilateur. Sous Direct3D 10, le code mort sera toujours éliminé, mais le Shader *input assembler layout* exige que la définition des données en entrée existent. Le Shader *Reflection Layer* fournit le moyen de traiter cette situation d'une manière générique, dans laquelle vous pouvez faire face aux exigences en entrée de Shader et garantir que vous fournirez une description complète du flux d'entrée au mapping de la signatures du shader.

Voici une fonction exemple pour detecter l'existence d'une sémantique nom/index dans une signature de fonction :

```
// Returns true if the SemanticName / SemanticIndex is used in the input signature.
// pReflector is a previously acquired shader reflection interface.
bool IsSignatureElementExpected(ID3D10ShaderReflection *pReflector, const LPCSTR SemanticName, UINT
    SemanticIndex)
{
    D3D10_SHADER_DESC shaderDesc;
    D3D10_SIGNATURE_PARAMETER_DESC paramDesc;

    Assert(pReflector);
    Assert(SemanticName);

    pReflector->GetDesc(&shaderDesc);

    for (UINT k=0; k<shaderDesc.InputParameters; k++)
    {
        pReflector->GetInputParameterDesc(k, &paramDesc);
        if (wcsncmp(SemanticName, paramDesc.SemanticName)==0 && paramDesc.SemanticIndex == SemanticIndex)
            return true;
    }

    return false;
}
```

VI-4-I - Création d'Objet Etat

Pendant le portage du code machine, il peut s'avérer utile d'utiliser au départ un jeu par défaut d'objets état et de désactiver tous les réglages d'état de rendu ou d'état de texture de Direct3D 9. Cela provoquera des artéfacts de rendu, mais c'est la façon la plus rapide de progresser. Vous pourrez par la suite construire un système de gestion des objets états qui pourra utiliser un index composé pour permettre un maximum de réutilisation du nombre d'objets états utilisés.

VI-5 - Portage de Textures

Le tableau suivant montre le jeu de correspondances des formats de texture de Direct3D 9 à Direct3D 10. Tout contenu sous un format non disponible sous DXGI devra être converti par des utilitaires.

| Format Direct3D 9 | Format Direct3D 10 |
|---------------------|---|
| D3DFMT_UNKNOWN | DXGI_FORMAT_UNKNOWN |
| D3DFMT_R8G8B8 | Non disponible |
| D3DFMT_A8R8G8B8 | Non disponible |
| D3DFMT_X8R8G8B8 | Non disponible |
| D3DFMT_R5G6B5 | Non disponible |
| D3DFMT_X1R5G5B5 | Non disponible |
| D3DFMT_A1R5G5B5 | Non disponible |
| D3DFMT_A4R4G4B4 | Non disponible |
| D3DFMT_R3G3B2 | Non disponible |
| D3DFMT_A8 | DXGI_FORMAT_A8_UNORM |
| D3DFMT_A8R3G3B2 | Non disponible |
| D3DFMT_X4R4G4B4 | Non disponible |
| D3DFMT_A2B10G10R10 | DXGI_FORMAT_R10G10B10A2 |
| D3DFMT_A8B8G8R8 | DXGI_FORMAT_R8G8B8A8_UNORM & DXGI_FORMAT_R8G8B8A8_UNORM_SRGB |
| D3DFMT_X8B8G8R8 | Non disponible |
| D3DFMT_G16R16 | DXGI_FORMAT_R16G16_UNORM |
| D3DFMT_A2R10G10B10 | Non disponible |
| D3DFMT_A16B16G16R16 | DXGI_FORMAT_R16G16B16A16_UNORM |
| D3DFMT_A8P8 | Non disponible |
| D3DFMT_P8 | Non disponible |
| D3DFMT_L8 | DXGI_FORMAT_R8_UNORM Note : Utilisez le .r swizzle dans le shader pour dupliquer le rouge dans les autres composants pour obtenir le comportement D3D9 |
| D3DFMT_A8L8 | Non disponible |
| D3DFMT_A4L4 | Non disponible |
| D3DFMT_V8U8 | DXGI_FORMAT_R8G8_SNORM |
| D3DFMT_L6V5U5 | Non disponible |
| D3DFMT_X8L8V8U8 | Non disponible |
| D3DFMT_Q8W8V8U8 | DXGI_FORMAT_R8G8B8A8_SNORM |
| D3DFMT_V16U16 | DXGI_FORMAT_R16G16_SNORM |
| D3DFMT_W11V11U10 | Non disponible |
| D3DFMT_A2W10V10U10 | Non disponible |
| D3DFMT_UYVY | Non disponible |
| D3DFMT_R8G8_B8G8 | DXGI_FORMAT_G8R8_G8B8_UNORM (dans DX9, les données étaient agrandies de 255.0f, mais cela peut se traiter en code shader). |
| D3DFMT_YUY2 | Non disponible |
| D3DFMT_G8R8_G8B8 | DXGI_FORMAT_R8G8_B8G8_UNORM (dans DX9, les données étaient agrandies de 255.0f, mais cela peut se traiter en code shader). |
| D3DFMT_DXT1 | DXGI_FORMAT_BC1_UNORM & DXGI_FORMAT_BC1_UNORM_SRGB |
| D3DFMT_DXT2 | DXGI_FORMAT_BC1_UNORM & DXGI_FORMAT_BC1_UNORM_SRGB Note |

| | |
|----------------------------------|--|
| | : DXT1 et DXT2 sont identiques de point de vue d'un hardware API... la seule différence est l' "alpha prémultiplié" qui peut être pisté par une application et ne nécessite pas un format séparé. |
| D3DFMT_DXT3 | DXGI_FORMAT_BC2_UNORM & DXGI_FORMAT_BC2_UNORM_SRGB |
| D3DFMT_DXT4 | DXGI_FORMAT_BC2_UNORM & DXGI_FORMAT_BC2_UNORM_SRGB Note : DXT3 et DXT4 sont identiques de point de vue d'un hardware API... la seule différence est l' "alpha prémultiplié" qui peut être pisté par une application et ne nécessite pas un format séparé. |
| D3DFMT_DXT5 | DXGI_FORMAT_BC3_UNORM & DXGI_FORMAT_BC3_UNORM_SRGB |
| D3DFMT_D16 & D3DFMT_D16_LOCKABLE | DXGI_FORMAT_D16_UNORM |
| D3DFMT_D32 | Non disponible |
| D3DFMT_D15S1 | Non disponible |
| D3DFMT_D24S8 | Non disponible |
| D3DFMT_D24X8 | Non disponible |
| D3DFMT_D24X4S4 | Non disponible |
| D3DFMT_D16 | DXGI_FORMAT_D16_UNORM |
| D3DFMT_D32F_LOCKABLE | DXGI_FORMAT_D32_FLOAT |
| D3DFMT_D24FS8 | Non disponible |
| D3DFMT_S1D15 | Non disponible |
| D3DFMT_S8D24 | DXGI_FORMAT_D24_UNORM_S8_UINT |
| D3DFMT_X8D24 | Non disponible |
| D3DFMT_X4S4D24 | Non disponible |
| D3DFMT_L16 | DXGI_FORMAT_R16_UNORM Note : Utilisez le .r swizzle dans le shader pour dupliquer le rouge dans les autres composants pour obtenir le comportement D3D9 |
| D3DFMT_INDEX16 | DXGI_FORMAT_R16_UINT |
| D3DFMT_INDEX32 | DXGI_FORMAT_R32_UINT |
| D3DFMT_Q16W16V16U16 | DXGI_FORMAT_R16G16B16A16_SNORM |
| D3DFMT_MULTI2_ARGB8 | Non disponible |
| D3DFMT_R16F | DXGI_FORMAT_R16_FLOAT |
| D3DFMT_G16R16F | DXGI_FORMAT_R16G16_FLOAT |
| D3DFMT_A16B16G16R16F | DXGI_FORMAT_R16G16B16A16_FLOAT |
| D3DFMT_R32F | DXGI_FORMAT_R32_FLOAT |
| D3DFMT_G32R32F | DXGI_FORMAT_R32G32_FLOAT |
| D3DFMT_A32B32G32R32F | DXGI_FORMAT_R32G32B32A32_FLOAT |
| D3DFMT_CxV8U8 | Non disponible |
| D3DDECLTYPE_FLOAT1 | DXGI_FORMAT_R32_FLOAT |
| D3DDECLTYPE_FLOAT2 | DXGI_FORMAT_R32G32_FLOAT |
| D3DDECLTYPE_FLOAT3 | DXGI_FORMAT_R32G32B32_FLOAT |
| D3DDECLTYPE_FLOAT4 | DXGI_FORMAT_R32G32B32A32_FLOAT |
| D3DDECLTYPED3DCOLOR | Non disponible |
| D3DDECLTYPE_UBYTE4 | DXGI_FORMAT_R8G8B8A8_UINT Note : Le shader récupère des valeurs UINT, mais s'il y a besoin de valeurs en virgule flottante de style Direct3D 9 (0.0f, 1.0f... 255.f), les |

| | |
|-----------------------|---|
| | valeurs UINT peuvent être simplement converties en float32 dans le shader. |
| D3DDECLTYPE_SHORT2 | DXGI_FORMAT_R16G16_SINT Note : Le shader récupère des valeurs SINT, mais s'il y a besoin de valeurs en virgule flottante de style Direct3D 9, les valeurs SINT peuvent être simplement converties en float32 dans le shader. |
| D3DDECLTYPE_SHORT4 | DXGI_FORMAT_R16G16B16A16_SINT Note : Le shader récupère des valeurs SINT, mais s'il y a besoin de valeurs en virgule flottante de style Direct3D 9, les valeurs SINT peuvent être simplement converties en float32 dans le shader. |
| D3DDECLTYPE_UBYTE4N | DXGI_FORMAT_R8G8B8A8_UNORM |
| D3DDECLTYPE_SHORT2N | DXGI_FORMAT_R16G16_SNORM |
| D3DDECLTYPE_SHORT4N | DXGI_FORMAT_R16G16B16A16_SNORM |
| D3DDECLTYPE_USHORT2N | DXGI_FORMAT_R16G16_UNORM |
| D3DDECLTYPE_USHORT4N | DXGI_FORMAT_R16G16B16A16_UNORM |
| D3DDECLTYPE_UDEC3 | Non disponible |
| D3DDECLTYPE_DEC3N | Non disponible |
| D3DDECLTYPE_FLOAT16_2 | DXGI_FORMAT_R16G16_FLOAT |
| D3DDECLTYPE_FLOAT16_4 | DXGI_FORMAT_R16G16B16A16_FLOAT |

Pour les formats non compressés, DXGI a limité ce qui est supporté pour les modèles de formats de pixels arbitraires ; tous les formats non compressés doivent être de type RGBA. Cela peut exiger le mixage des formats de pixels existants, ce que nous vous conseillons de calculer comme une passe pré-process hors ligne quand c'est possible.

VI-6 - Portage de Shaders

VI-6-a - Les Shaders Direct3D 10 sont autorisés sous HLSL uniquement

Direct3D 10 limite l'utilisation du langage assembleur au seul débogage. Par conséquent, tout shader assembleur écrit à la main sous Direct3D 9 devra être converti en HLSL.

VI-6-b - Signatures et Liaison de Shaders

Nous avons discuté plus haut dans ce document des exigences concernant la Liaison Input Assembly vers les signatures d'entrée du Vertex shader (voir plus haut). Notez que le module d'exécution de Direct3D 10 a également durci les exigences pour l'établissement de liens d'étape à étape entre Shaders. Ce changement affectera les sources de shaders là où le lien entre étapes n'aura pas été complètement décrit sous Direct3D 9. Par exemple :

```

VS_OUTPUT                                PS_INPUT
float4  pos : SV_POSITION;                float4 pos : SV_POSITION;
float4  uv1 : TEXCOORD1;                  float4 uv1 : TEXCOORD1;
float4x3 tangentSp : TEXCOORD2;           float4 tangent : TEXCOORD2; *
float4  Color : TEXCOORD6;                float4 color : TEXCOORD6;
    
```

* Lien VS - PS rompu - même si le pixel shader peut ne pas être intéressé par la matrice toute entière, le lien doit spécifier la virgule flottante complète float4x3.

Notez que la sémantique du lien entre les étapes doit correspondre exactement. Toutefois, les entrées des étapes cibles peuvent être un préfixe des valeurs en sortie. Dans l'exemple ci-dessus, le pixel shader aurait pu avoir position et *texcoord1* comme seules entrées, mais il n'aurait pas pu avoir position et *texcoord2* comme seules entrées, à cause des contraintes d'ordre.

VI-6-c - Liens entre Shaders sous HLSL

Les liens entre shaders peuvent se faire à n'importe lesquels de ces points dans le pipeline :


- d'Input Assembler à Vertex Shader
- de Vertex Shader à Pixel Shader
- de Vertex Shader à Geometry Shader
- de Vertex Shader à la sortie de flux
- de Geometry Shader à Pixel Shader
- de Geometry Shader à la sortie de flux

VI-6-d - Tampons Constant






Pour faciliter le portage de contenus depuis Direct3D 9, une approche initiale pour la gestion des constantes en dehors du système des Effets pourrait impliquer la création d'un tampon constant buffer unique contenant toutes les constantes nécessaires. Il est important pour les performances d'ordonner les constantes dans les tampons en fonction de la fréquence attendue de leur mise à jour. Cette organisation réduira la quantité de jeux de constantes redondantes au minimum.

VI-7 - Différences supplémentaires de Direct3D 10 à surveiller


VI-7-a - Entiers en entrée

Sous Direct3D 9, il n'y avait pas de véritable support hardware pour les types de données sous forme d'entiers. Toutefois, le hardware Direct3D 10 supporte les types entiers explicites. Si vous avez des données en virgule flottante dans votre vertex buffer, il vous faut une entrée en virgule flottante. Sinon, la représentation en configuration binaire de la valeur en virgule flottante sera de type entier. Un type entier n'est pas permis pour une entrée de pixel shader sauf si la valeur est marquée comme sans interpolation (cf.  **Modificateurs d'Interpolation**).

VI-7-b - Curseurs de souris


Sur les précédentes versions de Windows, les utilitaires standards de curseur de souris GDI ne fonctionnaient pas correctement sur certains appareils exclusivement plein-écran. Les API  **IDirect3DDevice9::SetCursorProperties**,  **IDirect3DDevice9::ShowCursor** et  **IDirect3DDevice9::SetCursorPosition** ont été ajoutées pour traiter ces cas. Puisque la version Windows Vista de GDI comprend entièrement les surfaces  **DXGI**, il n'y a plus besoin de cette API spécialisée pour curseur de souris, aussi n'y a-t-il pas d'équivalent Direct3D 10. Au lieu de cela, les applications Direct3D 10 doivent utiliser les  **utilitaires de curseur de souris GDI** pour les curseurs de souris.

VI-7-c - Transposition de Texels en Pixels sous Direct3D 10

Sous Direct3D 9, les centres des texels et les centres des pixels étaient à une demi-unité d'écart les uns des autres (cf.  **Transposer directement les Texels en Pixels (Direct3D 9)**). Sous Direct3D 10, les centres des texels sont déjà à des demi-unités, aussi n'est-il pas du tout besoin de décaler les coordonnées des vertices.

Le rendu des quads plein écran est plus direct avec Direct3D 10. Les quads plein écran doivent être définis en clip-space (-1,1) puis simplement être envoyé dans le vertex shader sans aucun changement. De cette façon, vous n'avez pas besoin de recharger votre vertex buffer chaque fois que la résolution d'écran change, et vous n'avez pas de travail supplémentaire à faire dans le pixel shader pour manipuler les coordonnées de texture.

VI-7-d - Changements de Comportement dans le Comptage de Références

Contrairement aux précédentes versions de Direct3D, les différentes fonctions Set ne feront pas référence aux applications objets. Cela signifie que l'application doit garantir qu'elle fera référence à l'objet aussi longtemps que cet objet désire être lié au pipeline. Quand le décompte de référence de l'objet tombe à zéro, l'objet est délié du pipeline en même temps qu'il est détruit. Ce style de référencement est également connu sous le nom de référence faible. Par conséquent, chaque emplacement de liaison sur l'application objet a une référence faible sur l'interface/objet. Sauf explicitement mentionné autrement, ce comportement doit être adopté pour toutes les méthodes de réglage. Toutes les fois où la destruction d'un objet a pour conséquence qu'un point de liaison est annulé, la couche Debug envoie un message d'alerte. Notez que les appels à des méthodes de type *Device Get* comme  **ID3D10Device::OMGetRenderTargets** augmentent le décompte de référence des objets retournés.

VI-7-e - Test de Niveau Coopératif

La fonctionnalité de l'API Direct3D 9  **IDirect3DDevice9::TestCooperativeLevel** est analogue au réglage de  **DXGI_PRESENT_TEST** quand on appelle  **IDXGISwapChain::Present**.



VII - Liaison de Librairies Statiques et Dynamiques (Direct3D 10)

Pour qu'une application fonctionne correctement, il faut que les fichiers DLL appropriés soient installés sur l'ordinateur hôte. Ces DLL peuvent être fournis soit par le système d'exploitation, soit par le package redistribuable de l'application.

VII-1 - Fichiers DLL appropriés au Chargement des Librairies




Les librairies incluses avec le kit de développement logiciel DirectX chargeront automatiquement les fichiers DLL adéquats à l'exécution. L'exception est à cette règle est *d3dx10.lib/d3dx10d.lib*, qui chargera le *d3dx10.dll* qui était embarqué avec cette version du kit de développement logiciel. Par exemple, si le kit de développement logiciel téléchargé inclut *d3dx10_33.dll* et *d3dx10_34.dll*, alors la librairie (*d3dx10.lib*) embarquée avec ce kit chargement *d3dx10_34.dll*. Si un autre kit est installé par la suite qui contient *d3dx10_35.lib*, le *d3dx10.lib* du kit précédent continuera encore à charger *d3dx10_34.dll*. La librairie *d3dx10.lib* du kit le plus récent chargera *d3dx10_35.dll*.

VII-2 - Redistribution des données binaires

Seul *d3dx10.dll* (et les versions suivantes du même fichier) peut être redistribué. Pour redistribuer ce fichier, vous devez utiliser la fonction  **DirectXSetup**. Pour tout détail sur l'utilisation de cette fonction et l'assemblage d'un package redistribuable, cf.  **Installer DirectX avec DirectXSetup**. Toutes les autres données binaires sont incluses dans Windows Vista. Les seules données binaires qui peuvent être redistribuées sont celles situées dans le répertoire suivant :

```
(SDK root)\Redist
```


Le tableau suivant décrit ce dont les développeurs de fichiers binaires doivent être conscients :

| Fichiers binaires Direct3D 10 | Description |
|-------------------------------|--|
| d3dx10.dll/d3dx10d.dll | Composants retail et debug D3DX10. Peuvent être redistribués et sont contenus dans le fichier CAB correspondant. |
| d3d10d.dll | Version debug de d3d10.dll. Inclus seulement comme élément du kit de développement logiciel DirectX et ne peut pas être redistribué. |
| d3d10ref.dll | Traceur de référence. Fournit l'implémentation logicielle du pipeline graphique. Inclus seulement comme élément du kit de développement logiciel DirectX et ne peut pas être redistribué. Le traceur de référence est destiné au débogage uniquement. Une liaison explicite n'est pas nécessaire ; tenter de créer un instrument de référence (cf.  D3D10CreateDevice) chargera ce dll s'il est présent. |
| d3d10sdklayers.dll | Une série d'utilitaires du kit de développement logiciel qui agissent comme une couche entre les appels de l'interface API et l'exécution, y compris la  couche debug et la  couche switch-to-reference . Une liaison explicite n'est pas nécessaire ; si un instrument est créé avec le registre de couche approprié, ce DLL est chargé automatiquement. Ce composant est |



| |
|---|
| conçu pour le développement et le débogage uniquement et ne peut pas être redistribué. |
|---|




VIII - Caractéristiques de Direct3D 10.1

Direct3D 10.1 étend le jeu de caractéristiques de Direct3D 10.0 avec les nouveautés suivantes :

- Modes de mélange Modes de mélange indépendants par cible de rendu utilisant la nouvelle interface d'état de mélange (cf.  **Interface ID3D10BlendState1**). Les opérations de mélange de sources duales sont limitées à rendre target slot 0 ; vous ne pouvez pas écrire vers d'autres sorties ou avoir des render targets liées à des slots autres que 0.
- Comportement en culling Les faces à surfaces nulle sont automatiquement soumises au culling ; cela n'affecte que le rendu de modèle fil de fer.
- Règles de virgule flottante Utilise les mêmes règles *IEEE-754* pour la virgule flottante SAUF QUE les opérations en virgule flottantes 32 bits ont été resserrées pour produire un résultat à 0,5 ULP du résultat infiniment précis. Cela s'applique à l'addition, la soustraction et la multiplication. (précision à 0,5 ULP pour la multiplication, 1,0 ULP pour sa réciproque).
- Formats - La précision du mélange *float16* a été portée à 0,5 ULP. Le mélange est également nécessaire pour les formats *UNORM16/SNORM16/SNORM8*.
- Anti-Aliasing multi-échantillon Le multisampling a été amélioré pour généraliser la transparence par couverture et pour rendre le travail du multisampling plus efficace avec un rendu multi-passe. Pour ce faire, toute la sémantique de multisampling est définie comme si le pixel shader fonctionnait toujours une fois par échantillon (fréquence par échantillon) et calculait une couleur distincte par échantillon. Si un pixel shader n'utilise aucun attribut par échantillon, il calculera la même valeur pour chaque échantillon couvert dans un pixel. Dans ce cas, cela équivaut au hardware exécutant le shader une fois par pixel (fréquence par pixel) et répliquant le résultat sur tous les échantillons couverts. Naturellement, faire tourner le shader à la fréquence par pixel produit toujours les mêmes résultats que faire tourner le même shader à la fréquence par échantillon quand les attributs sont échantillonnés à une fréquence par pixel. Les statistiques de pipeline PSInvocations s'incrémentent à la fréquence par échantillon sauf si le shader tourne à la fréquence par pixel.
- Bande passante d'Etage du Pipeline Augmente la quantité de données que l'on peut faire passer entre les shader stages :



| Ressource | Limites |
|---------------------------------------|-----------------------------|
| Registres entre Shader Stages | 32 (32 bits x 4 composants) |
| Registres d'entrée des Vertex Shaders | 32 |
| <i>Input Assembler input slots</i> | 32 |

- Règles de rasterisation - Les règles de rasterisation ont changé pour les lignes et les additions ; de nouvelles fonctionnalités ont été ajoutées :
 - *MultisampleEnable* n'affecte que la rasterisation de lignes (les points et les triangles ne sont pas affectés), et est utilisé pour choisir un algorithme de tracé de ligne. Cela signifie que certaines rasterisation multi-échantillons de Direct3D 10 ne sont plus supportées.
 - Nouvelle exécution de pixel shaders sous fréquence par échantillon avec rasterisation de primitives.
- Ressources - *CopyResource* est autorisé dans deux nouveaux scenarii :
 - Les surfaces *MSAA* colorées et profondeur/stencil peuvent désormais toutes les deux être utilisées avec *CopyResource* que ce soit comme source ou comme destination.
 - Conversion de Format pendant la copie entre certaines ressources typées préstructurées 32/64/128 bits et des représentations compressées de même durée d'impulsion binaire.
- Echantillonnage de texture Les instructions *sample_c* et *sample_c_lz* sont définies pour fonctionner à la fois avec *Texture2DArrays* et *TextureCubeArrays*. Utilisez l'élément de localisation (le composant alpha) pour spécifier un indice de matrice.
- Vues - *TextureCube* et le nouveau *TextureCubeArray* (cf.  **D3D10_TEXCUBE_ARRAY_SRV1**) ne sont pas de vraies ressources, mais sont des nouvelles vues sur une ressource *Texture2DArray*. Créez un affichage de ressource de la ressource *Texture2DArray* avec un nouveau registre d'utilisation (*D3D10_RESOURCE_MISC_TEXTURECUBE*), utilisez la nouvelle interface  **ID3D10ShaderResourceView1 Interface** pour lier un affichage de texture de cube au pipeline.

Les nouvelles caractéristiques nécessitent un type de dispositif 10.1 (cf.  **Interface ID3D10Device1**) que vous pouvez créer en appelant  **D3D10CreateDevice1**, ou vous pouvez créer le dispositif et lier une chaîne en même temps en appelant  **D3D10CreateDeviceAndSwapChain1**.

Sous Windows Vista Service Pack 1, les fichiers DLL Direct3D 10.0 et Direct3D 10.1 existent côte à côte sur le système. Pour accéder aux caractéristiques 10.1, vous avez deux possibilités :

VIII-1 - Accéder aux caractéristiques 10.1 sous Vista Gold et Vista Service Pack 1

Les développeurs qui désirent supporter Vista Gold ainsi que SP1 devront prendre en compte l'absence des nouvelles extensions 10.1 API sur Vista Gold. DXUT et D3DX10 fourniront tous les deux des fonctions pratiques pour créer le dispositif approprié, sur la base des fichiers DLL disponibles sur le système et sur la base du hardware disponible (10.0 ou 10.1). Le dispositif 10.1 dérive du dispositif 10.0 et peut être rétabli en utilisant *QueryInterface()*. Il est recommandé que chaque application conserve la trace du type de dispositif et maintienne un pointeur sur le dispositif 10.1 (si disponible) pour éviter de fréquents appels à *QueryInterface* quand les fonctionnalités de 10.1 sont désirées. De façon similaire, là où les affichages de ressources 10.1 et les objets états sont associés par une classe personnalisée de l'application, il est recommandé que l'application garde en mémoire si l'objet est de type 10.0 ou 10.1 pour éviter des appels redondants à *QueryInterface()*. D3DX10 inclut un jeu de fonctions utilitaires pour simplifier ce processus (cf.  **D3DX10CreateDevice** et  **D3DX10CreateDeviceAndSwapChain**).

VIII-2 - Accéder aux caractéristiques 10.1 sous Vista Service Pack 1 exclusivement

Certains développeurs peuvent vouloir choisir d'exiger Vista Service Pack 1, qui sera largement distribué aux utilisateurs finaux et qui inclut une série d'améliorations en dehors de Direct3D 10.1. Ces développeurs peuvent utiliser exclusivement les en-têtes et bibliothèques de Direct3D 10.1, créant une dépendance envers les fichiers DLL de Direct3D 10.1 qui supportent à la fois les hardware 10.0 et 10.1 (néanmoins, certains appels risquent d'échouer sur certains dispositifs 10.0 où la nouvelle fonctionnalité n'est pas supportée).

Quelques notes supplémentaires :

- Les API exposées dans le fichier *D3DX10.dll* acceptent à la fois les dispositifs 10.0 et 10.1 et profitent des fonctionnalités 10.1 quand elles sont disponibles.
- *D3D10SDKLayers.dll* accepte un dispositif 10.1 et peut produire le débogage approprié pour les caractéristiques 10.1.
- *D3D10Ref.dll* met en uvre les dispositifs logiciels 10.0 et 10.1.
- *D3DX10* et *FXC* supporte le modèle de shader mis à jour pour 10.1 avec les cibles suivantes : *vs_4_1*, *gs_4_1*, *ps_4_1* et *fx_4_1* qui peuvent être liées à un dispositif 10.1. Un dispositif 10.1 supporte les shaders modèle 4.0 et 4.1.
- Le framework effect Direct3D 10.0 supporte les dispositifs 10.0 et 10.1. Toutefois, toute technique qui inclut des shaders de modèle 4.1 ou les nouvelles caractéristiques 10.1 doit utiliser un dispositif 10.1.