

Infographie avec DirectX 11

Théorie et ateliers

Automne 2022

François Jean

Pierre-Marc Bérubé

Mykel Leclerc Brisson



TABLE DES MATIÈRES

TABLE DES MATIÈRES	1
Introduction	9
Pourquoi DirectX ?.....	10
Pourquoi des graphiques en 3D ?.....	10
Qu'est-ce que l'interactivité ?.....	11
Le monde réel et le monde virtuel	11
Les éléments du monde 3D	13
L'éclairage.....	13
La mise en place et le découpage	14
Le rendu	14
L'évolution des composantes.....	15
Le matériel	15
DirectX.....	15
DirectX Graphics.....	16
1. Le pipeline de rendu.....	19
1.1 Le pipeline	20
1.2 Les données originales des objets 3D	23
1.3 Le traitement des sommets	27
1.4 Le traitement de la géométrie.....	32
2. Un modèle de base d'animation 3D.....	34
2.1 Le design d'un modèle d'animation.....	35
Organisation du programme	35
La performance.....	35
PetitMoteur3D.....	35
2.2 Application de base	36
2.3 Configuration de l'environnement.....	37
2.4 Notre application.....	37
Ce que l'on a	37
Détails du modèle de départ.....	37
Ce que l'on veut par la suite	41
2.5 La fonction InitAnimation.....	43
Notre horloge	45
2.6 La fonction Animation.....	47
L'animation optimisée	48
Le modèle Préparer-Afficher	48
Le modèle Afficher-Préparer	49
2.10 Règles de base sur la performance des animations en 3D.....	50

3. Premiers pas avec Direct3D	51
Initialisation de l'environnement 3D.....	52
Initialisation de la scène	53
Animation de la scène.....	53
Rendu de la scène	53
Le « nettoyage »	54
3.1 Préparation de l'application	55
Les fichiers stdafx.cpp et stdafx.h	56
3.2 Gestion des erreurs DirectX	57
Erreurs et débogage de l'application	57
Erreurs d'environnement et erreurs d'installation	57
3.3 Fonctions de gestion des erreurs et des pointeurs COM.....	58
3.4 Classes de dispositifs	60
Modification de CMoteur::Initialisations.....	62
3.5 Les objets Direct3D	65
La chaîne d'échange (<i>swap chain</i>)	65
Le dispositif (<i>device</i>)	66
Le contexte (<i>device context</i>)	66
Vue de surface de rendu (<i>render target view</i>).....	66
3.6 Création des objets Direct3D	68
3.7 Création des objets Direct3D (suite).....	73
3.8 La fonction d'animation.....	75
3.9 Rendu de la scène	77
3.10 Faire le ménage	79
4. Première scène	81
4.1 Initialisation de la scène.....	82
La fonction InitScene	82
4.2 Le bloc est mis en scène.....	83
4.3 Le bloc est défini.....	85
4.4 Copier le bloc en ressource.....	92
4.5 Les <i>shaders</i>	95
4.6 Nos premiers shaders	97
Le petit vertex shader – premiers éléments de HLSL	97
Le petit pixel shader.....	99
4.7 Compilation et chargement des shaders.....	100
4.8 Mise en scène – les transformations	105
Les matrices de transformation.....	105
La matrice de transformation dans le monde	105
La matrice de vision.....	105
La matrice de projection	105
Les matrices en résumé	106

4.9	Initialisation des matrices	107
	Matrice de transformation dans le monde	107
	Matrice de vision et matrice de projection	107
4.10	L'animation de la scène	109
4.11	Le rendu de la scène	111
4.12	Faire le ménage	115
5.	Contrôle de l'affichage.....	117
5.1	L'adaptateur graphique	118
5.2	Obtenir l'information sur un adaptateur	119
	Types de mémoire vidéo	122
5.3	Déterminer les dimensions d'affichage	124
5.4	Obtenir l'affichage le plus intéressant.....	126
	Nouveau constructeur pour CInfoDispositif.....	126
	Nouveaux paramètres pour les objets du dispositif	128
5.5	Affichage « plein écran »/« fenêtré » sur demande.....	130
	Nouvelle version du constructeur de CDispositifD3D11	131
5.6	Corriger la projection	134
5.7	Les tris de profondeur	136
	Optimisation des Z-Buffers.....	138
5.8	Implanter un tampon de profondeur.....	139
5.9	Implanter le « backface-culling »	142
6.	Éclairage	145
6.1	Les modèles d'illumination et les types de sources de lumière	147
	L'illumination « émise ».....	148
	L'illumination ambiante et la source ambiante	148
	L'illumination diffuse.....	148
	L'illumination spéculaire (ou réflexion spéculaire)	149
	La couleur de la lumière	150
	Les sources de lumières de type point	151
	Les sources de lumière divergentes ou « spot ».....	151
	Les sources de lumière directionnelles	152
	Sources lumineuses multiples.....	152
	Les matériaux.....	153
6.2	Éclairage simplifié avec spécularité.....	154
	Une petite version de nuancement Phong (<i>Phong Shading</i>)	154
	Retour sur la théorie	154
6.3	Écriture des shaders.....	157
	Le vertex shader.....	159
	Le pixel shader	160
6.4	Nos shaders dans le programme	162
6.5	Les effets	166
	Effects 11	166

Notre effet MiniPhong.....	168
Une classe de bloc avec effet.....	169
Initialisation de l'effet	170
Rendu avec effet.....	173
6.6 Le compilateur d'effets et de shaders.....	175
7. Les textures.....	177
7.1 Les textures	178
Format des textures.....	178
7.2 Coordonnées d'application des textures	179
7.3 Établir les coordonnées d'application de la texture.....	182
7.4 Le chargement et la gestion des textures	185
Nouvelles classes nécessaires pour la gestion des textures	185
7.5 Les textures dans nos shaders	190
7.6 Associer une texture à un objet	193
Implanter le chargement de la texture (des textures...)	193
L'état d'échantillonnage	194
Activation de la texture lors du rendu.....	195
7.7 L'échantillonnage.....	196
Problème #1 – La magnification	196
Problème #2 – La minification	197
Solutions	198
7.8 L'échantillonnage dans notre application.....	203
8. Objets 3D – Les maillages.....	205
8.1 Les maillages	206
Formats	206
8.2 Construire notre format d'objet.....	207
8.3 Une classe pour nos objets <i>mesh</i>	209
La classe	209
Le constructeur	209
La fonction TransfertObjet	212
La fonction InitEffet.....	216
La fonction Draw	218
La fonction Anime	221
Faire le ménage	222
8.4 Modifications à notre effet	223
8.5 Des classes pour le chargement de maillages	224
La classe-interface IChargeur	225
8.6 Travailler avec un objet 3D.....	227
8.7 Solution à la lecture très lente	229
9. Affichage 2D.....	230

9.1 Les sprites	232
9.2 Sprites et textures	232
Méthode 1- Principe de base	233
Méthode 2 - L'afficheur de sprites.....	234
9.3 Méthode 1- Implantation	235
La classe CSpriteTemp.....	235
9.5 Méthode 1- L'effet.....	243
9.5 Méthode 1- L'affichage du sprite	244
9.6 Méthode 1- La transparence (mélange alpha)	248
9.7 Méthode 1 - Position et dimensions d'affichage	252
Dimensions réelles de la texture	252
Position et dimensions d'affichage.....	253
9.8 Méthode 2 - L'afficheur de sprites	255
La classe CAfficheurSprite	255
L'utilisation de l'afficheur.....	259
Pourrait-on faire mieux ?.....	260
9.9 Désactiver/Activer le Z-buffer	261
9.10 Affichage sur un panneau (<i>billboarding</i>).....	262
Panneau orientable linéaire-Z.....	262
Est-ce complet ?	266
Pourrait-on faire mieux ?.....	266
9.12 Orientation linéaire	267
Linéaire-Z.....	267
Linéaire-XZ et Lineaire-YZ	267
Linéaire-XYZ.....	268
9.13 Le texte	269
DirectX et le texte	269
Solution temporaire	270
9.14 Principe de travail avec GDI+	271
9.15 La classe CAfficheurTexte.....	272
Fonctions statiques.....	273
Le constructeur	274
Le destructeur.....	275
La fonction Ecrire	276
9.16 Utilisation de CAfficheurTexte	277
Ajout de la fonction CAfficheurSprite ::AjouterSpriteTexte	277
Exemple d'utilisation (dans CMoteur).....	278
9.17 L'instanciation (intro).....	279
Instancier	279
10. Introduction aux dispositifs d'entrée.....	281
10.1 DirectInput.....	282
10.2 Les objets DirectInput.....	283

L'objet DirectInput.....	283
L'objet DirectInputDevice – le dispositif DirectInput.....	283
10.3 Une classe pour la gestion de DirectInput.....	284
10.4 Acquisition du clavier.....	289
10.5 Travailler avec le clavier	291
10.6 Travailler avec la souris	293
11. Les ombres.....	297
11.1 Différentes façons de faire de l'ombre.....	299
11.2 Le <i>shadow mapping</i>	305
11.3 Programmation du <i>shadow map</i>	307
1. La création de la texture.....	307
2. Autres préparatifs	310
3. Rendu de la scène – Préparation et rendu du <i>shadow map</i>	313
4. Les shaders pour la création du <i>shadow map</i>	315
5. Rendu de la scène – Préparation et rendu avec MiniPhong.....	317
6. Les shaders MiniPhong avec <i>shadow mapping</i>	320
12. Les « post-effects ».....	324
12.1 Le principe de base	325
12.2 La classe CPanneauPE.....	326
Déclarer et créer le panneau	330
12.3 La texture pour l'affichage de la scène	331
12.4 L'affichage de la scène sur la texture.....	334
Fonction pour activer l'affichage sur la texture	334
Fonction pour désactiver l'affichage sur la texture	335
Activer l'affichage sur la texture	336
12.5 Utiliser le panneau de post-effect.....	337
Initialisation des effets.....	337
La fonction Draw	337
12.6 L'effet « Nul »	339
12.7 L'effet « RadialBlur ».....	341
Ajustements pour plusieurs effets (techniques).....	342
La fonction Draw	344
13. Les « geometry shaders »	345
13.1 La programmation des geometry shaders.....	347
13.2 Une petite tessellation.....	349
Pour débuter	349
À faire	350
Le geometry shader	351
Petite modification au vertex shader	352
La technique	352

13.3 En liste de triangle	353
Exercice	353
14. Les shaders de tessellation.....	354
14.1 Les étapes de tessellation.....	355
Qu'est-ce qu'une <i>patch</i> ?	356
L'étape Hull Shader (HS).....	356
L'étape Tessellation	357
L'étape Domain Shader	358
14.2 Un exemple.....	359
Pour débuter	359
Modifications au panneau	359
Modification à la fonction Draw	360
Modifications au vertex shader.....	360
Construction du hull shader.....	361
Construction du domain shader	364
ANNEXE 1. Bases mathématiques	366
A.1 Les transformations sur le plan	367
Les vecteurs, les points et les sommets.....	367
Transformations	367
Translation	368
Rotation.....	368
Homothétie.....	369
Coordonnées homogènes	370
A.2 Opérations sur les matrices	372
Addition de matrices	372
Multiplication de matrices.....	372
A.3 Opérations en 2D	374
A.4 Les transformations en 3D	375
Translation	376
Rotation.....	377
Rotations avec les angles d'Euler	378
Homothétie.....	379
Opérations sur les matrices	379
Les quaternions	380
Références	382

Introduction

Presque tous les ordinateurs disponibles aujourd'hui disposent de graphiques avancés tant au niveau du traitement de l'image que du 2D et du 3D. Beaucoup de logiciels utilisent le 3D (par exemple, les jeux qui ne sont pas en 3D sont aujourd'hui l'exception) tant pour les logiciels d'application que certains logiciels de design. Par contre, la programmation d'applications 3D est encore l'apanage de professionnels utilisant des bibliothèques spécialisées (OpenGL ou DirectX).

Dans ce manuel, nous étudierons l'écriture d'applications informatiques interactives utilisant le 3D ainsi que d'autres aspects multimédias tels le son et les interactions avec l'utilisateur. Nous écrirons nos applications avec DirectX, une bibliothèque spécialisée dans le rendu d'image 3D.

Objectifs du chapitre

- ✓ Présenter les choix d'outils qui seront utilisés dans ce volume;
- ✓ Se familiariser avec DirectX et ses composants;
- ✓ Faire un tour d'horizon des différents éléments utilisés en infographie 3D ainsi que de la terminologie qui l'accompagne;
- ✓ Voir l'impact des systèmes graphiques modernes sur la programmation en infographie;
- ✓ Se familiariser brièvement avec l'évolution de DirectX.

Introduction

Pourquoi DirectX ?

En réalité, les choix sont assez limités quand on parle de développement d'applications multimédias interactives.

Pour le 3D, le grand classique est **OpenGL** de Silicon Graphics (maintenant OpenGL est géré par Khronos Group). OpenGL existe depuis longtemps, il est enseigné dans les universités depuis un certain temps et il est aussi disponible sous plate-forme PC. Malheureusement, c'est un produit qui ne vise que le 3D donc on doit utiliser d'autres bibliothèques pour l'interactivité et le multimédia. De plus, c'est un produit qui a été élaboré au début des années 1990 et que Silicon Graphics avait beaucoup négligé. (Note : même si des versions plus récentes d'OpenGL sont apparues depuis 2001 soient 1.3, 1.4, 1.5, 2.0, 2.1 et même les spécifications de la 3.0, la plupart des « experts » font encore référence à la version 1.2 de 1998). Une tentative pour rendre OpenGL « objet » a été effectuée au début de 1995 avec les extensions **OpenInventor** mais celles-ci ont connu peu de succès et n'ont pas été transposées sur PC par Silicon Graphics. De plus, la librairie OpenGL a été récemment déclarée obsolète sur les appareils Apple et remplacée par la librairie **Metal**. Il est important de noter que l'« open » d'OpenGL n'a rien à voir avec le mouvement « open source » (même si certaines versions *open source* existent maintenant). Sa portabilité « relative » sur Windows et sur Linux en fait toutefois un choix intéressant pour les gens qui développent sur ces deux plateformes.

D'autres bibliothèques 3D et multimédias existent, mais depuis quelques années, la bibliothèque la plus utilisée sur PC-Windows est DirectX de Microsoft qui contient tous les aspects du multimédia et implante une certaine approche orientée objet parfaitement compatible avec des langages comme C++, C#, Java ou même Visual Basic. La version « *managed* » de la bibliothèque est entièrement orientée objet. DirectX est d'ailleurs la base multimédia du X-Box 360 (un produit Microsoft) et de XNA.

Pourquoi des graphiques en 3D ?

Les graphiques en 3D d'aujourd'hui sont utilisés pour recréer des environnements réalistes sur un écran d'ordinateur. Ils permettent à l'utilisateur une certaine forme d'immersion dans l'environnement de travail (ou de jeu). On pense aux environnements industriels, aux laboratoires, aux ateliers de design, aux simulations et évidemment aux jeux vidéos qui sont généralement une certaine forme de simulation impliquant beaucoup d'interactivité.

Qu'est-ce que l'interactivité ?

Les premiers jeux par ordinateur n'utilisaient pas une forme d'interactivité très évoluée : la situation était décrite par un texte et l'utilisateur pouvait au moyen de certaines « actions » : se déplacer, observer, ou saisir des objets, puis le jeu réaffichait le nouveau statut du jeu. On ne pouvait pas tout faire, mais seulement des actions prédéterminées. Puis les jeux ont intégré des éléments graphiques, parfois 3D, pour en arriver à maintenant où l'environnement graphique 3D offert par les jeux a encore ses limites, mais offre une latitude très intéressante d'actions « réelles ».

Par opposition, un film au cinéma n'est pas interactif, les graphiques 3D ont été grandement utilisés dans les effets spéciaux depuis quelques années, mais l'auditeur ne peut pas manipuler ou interagir avec les images qui lui sont présentées (du moins pas encore...).

La création d'un monde 3D visible sur un écran d'ordinateur et interagir avec l'utilisateur est notre objectif. C'est un travail difficile puisque les difficultés sont partout : la vitesse des ordinateurs, la qualité des graphiques, des sons, les délais de réponse des périphériques. Tous ces détails se donnent la main pour ralentir notre application.

Le monde réel et le monde virtuel

Le monde réel est en 3D (et même plus...). On ne parle pas ici de 3D en tant que modèle mathématique, mais de 3D selon les lois de la physique. Bien sûr, il est possible d'expliquer certains phénomènes au moyen des mathématiques, mais ces explications ne couvrent qu'une partie de la réalité. De plus, la plupart des lois de la physique sont **automatiques**, c.-à-d. qu'on n'a pas à les connaître pour qu'elles s'appliquent, par exemple :

- essayez de traverser un mur, vous en serez incapable ;
- sautez, vous retomberez ;
- regardez, les objets éloignés vous semblent plus petits ;

Mais lorsque vient le temps de recréer virtuellement ces éléments au moyen de l'ordinateur, ce n'est pas toujours simple. Une loi de la physique doit être appliquée correctement pour produire son effet. Par exemple, si on ne dispose d'aucun moyen pour vérifier les propriétés solides des objets, il sera alors possible de traverser ceux-ci.

De plus, un de nos principaux problèmes est que l'écran de l'ordinateur est en 2D, il faut donc effectuer une projection de notre monde 3D sur cet écran. Et, pire encore, nous devons conserver des informations sur tous les objets que nous désirons afficher : leur modèle mathématique, leur position, leur couleur, leur matériel, leur vitesse de déplacement, etc. Ces détails deviennent rapidement très lourds à gérer.

Et pour compléter le tout, nous devons implanter notre monde image par image, comme pour un dessin animé, mais contrairement au dessin animé nous devrons :

- réagir aux actions de l'utilisateur
- vérifier ces actions selon le « scénario » (les règles du jeu, l'intelligence artificielle, les actions des autres utilisateurs, etc.)
- modifier la position de certains objets selon ces actions ou le « scénario »
- Construire chaque image en temps réel au moment de la visualisation.

Et le tout à une vitesse de plus de 60 images par seconde.

Les éléments du monde 3D

La composition d'images 3D par ordinateur existe depuis plusieurs années. En fait, les informaticiens et les mathématiciens y travaillaient avant même que des dispositifs comme les cartes graphiques ne voient le jour. Comme beaucoup de problèmes, celui-ci a été divisé en plusieurs éléments qui généralement se suivront dans le processus de création, un peu comme une chaîne de montage. Chaque étape du processus aura sa spécialité.

Une étape importante vise la logique de l'animation, cette étape dépend évidemment du type d'application et des interactions avec l'utilisateur. Nous y planterons habituellement les règles relatives aux mouvements, les lois de la physique et quelques trucs bien à nous pour accélérer l'affichage.

Une autre étape est la partie plus classique de l'affichage 3D, c.-à-d. la transformation d'éléments mathématiques en une image à l'écran. Il s'agit de la gestion de ce que l'utilisateur verra à l'écran :

- Quels sont les objets dans son champ de vision ?
- Quelle est la couleur des objets ?
- Quelle sorte d'éclairage est présent ?
- Quel niveau de détails désirons-nous observer ?
- Et plus...

Cette dernière étape est aujourd'hui principalement exécutée sur la carte graphique. On l'appellera souvent **pipeline de rendu** (chapitre 1).

L'éclairage

La création d'une illusion de luminosité dans notre scène est une autre étape importante de notre pipeline. Il est bien sûr possible de donner des couleurs fixes aux objets (comme nous le ferons au chapitre 3), mais les objets sont alors ternes et on en distingue mal les formes, nous ajouterons donc de l'éclairage à nos scènes, que ce soit pour simuler l'éclairage du soleil, des néons, des spots ou d'une bougie. Sans éclairage, nos objets seront « plats » puisque l'éclairage ajoute énormément à l'illusion de perspective.

L'éclairage dans une scène 3D demandera beaucoup de temps de calcul par le CPU et/ou par la carte graphique. Parce que la lumière devra être appliquée à chaque « morceau » de chaque objet, il en résulte des millions de calculs. Et nous devons parfois combiner plusieurs sources lumineuses.

Certains programmes trichent avec la lumière (et c'est souvent une très bonne idée...) en la préparant à l'avance.

La mise en place et le découpage

Cette étape permet de déterminer quels sont les objets à afficher à l'écran et quels sont ceux qui ne seront pas affichés. Il sera en effet inutile de calculer des effets spéciaux coûteux en temps de *CPU* ou de *GPU* sur des objets ne figurant pas dans l'image courante. N'oubliez pas que le 3D est comme le cinéma, un comédien qui n'apparaît pas dans une scène n'a pas à porter son costume (ni même à être là...). C'est à ce moment que nos objets sont représentés par des polygones (presque toujours des triangles) pour pouvoir les traiter plus facilement lors du rendu.

Le découpage permet de retirer partiellement ou totalement des polygones partiellement ou totalement cachés par d'autres objets.

Le rendu

Le rendu est le processus permettant à nos polygones mathématiques d'être combinés au champ de vision, à leurs propriétés physiques et graphiques et à l'éclairage pour obtenir l'image finale. Ce n'est pas très clair, donc je m'explique.

Les écrans d'ordinateur sont des surfaces planes en deux dimensions. On parle de 2D même s'il s'agit en réalité d'un très petit sous-ensemble du 2D. L'image finale sera une collection de pixels disposés correctement. Et notre programme devra créer cette image.

Prenons une partie toute simple d'un de nos objets, un triangle :

- Selon la position du triangle par rapport à la caméra, celui-ci sera plus grand, plus petit, plus mince, plus large.
- La couleur originale du triangle et/ou sa texture seront combinées aux différentes sources d'éclairage pour obtenir une série de pixels de couleurs
- Ces pixels seront projetés à l'écran selon la position déterminée au premier point.

Cela semble simple, mais il s'agit en réalité de l'étape la plus complexe de notre pipeline. Heureusement, cette étape est gérée presque totalement par Direct3D, et ce, automatiquement. Mais lorsque nous utiliserons des *shaders*, ce sera une autre histoire...

L'évolution des composantes

Le matériel

Malgré la puissance des ordinateurs d'aujourd'hui, les possibilités des processeurs centraux (**CPU**) sont limitées. En réalité, il faudrait dire que plus les ordinateurs sont capables de rendre du 3D de qualité, plus les demandes des utilisateurs et de l'industrie augmentent. C'est dans l'ordre des choses.

En plus des processeurs centraux, on parle aujourd'hui de **GPU** (*Graphic Processing Unit*), des microprocesseurs spécialisés pour l'affichage graphique et le 3D que l'on retrouve sur les cartes graphiques modernes. On retrouve donc sur les GPU récents la prise en charge d'une grande partie des étapes de transformations et d'éclairage. Des fonctions 3D de plus en plus sophistiquées se retrouvent sur ces cartes et on y retrouve ainsi des fonctions qui étaient très coûteuses en temps de processeur, ce qui rend aujourd'hui possibles des affichages inimaginables il y a deux ou trois ans.

Les systèmes multi-processeurs changent évidemment ce rapport, mais il y a aussi des multi-GPU !

DirectX

Microsoft® DirectX® est un ensemble de technologies créées par Microsoft pour améliorer l'aspect multimédia de Windows et tirer profit des nouvelles composantes multimédias. Le but principal est de faciliter le développement et l'implantation d'application utilisant le 3D en temps réel, la vidéo, la musique interactive, et les effets sonores. Construit à partir d'un petit noyau de développement appelé le « **Game SDK** » vers 1993 puis implanté comme un ajout à Windows 95, DirectX fait maintenant partie de l'architecture des plates-formes Windows, d'abord comme composantes annexes pour Windows 98 et Windows 2000 puis comme élément du système d'exploitation pour Windows XP et finalement comme source graphique principale avec Windows Vista (et bien sûr dans Windows 7 et Windows 8). Les mises à jour standardisées permettent à des applications plus récentes de s'installer en mettant à jour DirectX.

L'intérêt principal de DirectX pour les développeurs est d'avoir une **interface de programmation (API)** standardisée permettant l'accès aux composantes de haute performance comme les cartes 3D et les cartes de son. Cette interface contrôle toute une série de fonctions de « bas niveau » incluant la gestion de la mémoire vidéo et du rendu ; le support pour des périphériques tels *joysticks*, claviers, et souris ; et le contrôle du mixage des sons et de la musique. Ces fonctions se retrouvent dans des sous-ensembles de DirectX qui sont : **DirectX Graphics**, **DirectX Input** (XInput et Direct Input), **DirectX Audio**

Avant DirectX, les développeurs d'applications multimédias devaient souvent créer leurs propres pilotes pour avoir accès au maximum de périphériques. DirectX fournit un étage appelé « *hardware abstraction layer* » (HAL) qui implémente les fonctions de bas-niveau que l'on retrouve sur les cartes modernes au moyen d'une interface standardisée. Le résultat est que le développeur peut écrire une application qui fonctionnera sur une multitude de composantes.

DirectX Graphics

Malgré une tentative dans le SDK de renommer les aspects graphiques Graphics, le nom Direct3D (la partie 3D de DirectX Graphics) est demeuré le plus utilisé. Il permet aux applications d'accéder presque directement aux plus récentes technologies de cartes 3 D. Il permet aux développeurs d'atteindre rapidement la qualité graphique nécessaire pour les jeux et applications d'aujourd'hui. Aujourd'hui, DirectX Graphics regroupe Direct3D, Direct2D, DirectWrite, DXGI et quelques autres aspects graphiques.

Certains ajouts ont été effectués depuis les premières versions de Direct3D, ce sont :

DirectX 6.0 :

- Formats plus flexibles de sommets pour la définition de la géométrie
- Tampons de sommets pour l'organisation de la géométrie
- Rendu à plusieurs textures
- Gestion automatique des textures
- Tri de profondeur interchangeable (*z-buffers* ou *w-buffers*).
- Mapping d'environnement sur une base « par pixel » pour effectuer entre autres des effets d'eau

DirectX 7.0 :

- Support pour les transformations matérielles
- Support pour l'éclairage matériel
- Mapping d'environnement au moyen de cubes d'environnement
- Mélanges de géométrie
- Gestion automatique des textures améliorée
- Génération automatique des coordonnées de textures, transformations de textures, projection de textures, et plan de *clipping* arbitraires.
- La bibliothèque utilitaire D3DX

DirectX Graphics (DirectX 8.x et DirectX 9.x)

En plus de maintenir une compatibilité presque parfaite avec les versions précédentes, plusieurs nouvelles caractéristiques ont été ajoutées à la version 8.0. En fait Direct3D et DirectDraw ont été intégrés pour donner **DirectX Graphics**, une toute nouvelle API qui offre une certaine ressemblance avec les versions précédentes, mais est en réalité une réécriture complète.

- Langage programmable de traitement des sommets

Permet au développeur d'écrire ses propres fonctions (shaders) pour le morphing et l'animation des sommets, et aussi pour l'animation des palettes. Il permet aussi des modèles d'éclairage, des opérations géométriques, et du *mapping* définis par le développeur.

- Langage programmable de traitement des pixels

Permet au développeur d'écrire ses propres fonctions (shaders) d'accès au matériel pour le traitement des textures et des mélanges de couleurs, l'éclairage par pixel, ou tout autre algorithme défini par le développeur.

- Support pour le rendu multi-échantillons

Permet l'anti-aliasing (anticrénelage) au niveau de la scène ainsi que des effets multi-échantillons tels les mélanges de mouvement (motion blur) et les mélanges de profondeur de champ (depth-of-field blur).

- Support pour les « *sprites* » par point

Permet l'affichage de systèmes de particules pour effectuer des étincelles, des explosions, de la pluie, de la neige, etc.

- Textures 3-D volumétriques

Permet un système flexible pour les textures de certaines géométries, l'atténuation de l'éclairage en fonction de la distance et des effets atmosphériques.

- Support pour des primitives d'ordre plus grand que les versions précédentes

Permet d'améliorer l'apparence du contenu 3D en créant des formes plus agréables, qui peuvent aussi être importées plus facilement des logiciels de design 3 D.

- Amélioration de la bibliothèque D3DX

D3DX contient maintenant plusieurs nouvelles fonctions. Il s'agit d'une bibliothèque utilitaire placée au-dessus de Direct3D pour faciliter la tâche pour les fonctions fréquemment rencontrées par les développeurs. Plusieurs fonctionnalités reliées aux matrices s'y retrouvent.

DirectX 10

Développé principalement pour Windows Vista dont l'API graphique était maintenant basé sur DirectX. DirectX 10 implante beaucoup de nouveautés. Il s'agit en fait d'une refonte complète de l'API où plusieurs aspects du pipeline fixe ne sont plus disponibles.

À noter surtout :

- Un nouveau modèle de pilotes de périphériques (*drivers*) permettant à plusieurs applications d'utiliser les ressources du GPU simultanément.

- La disparition graduelle de certains aspects de la logique fixe de DirectX en faveur de la programmation du GPU via des langages comme HLSL et l'architecture FX.

DirectX 11

DX 11 est une version revue et redéfinie de DX 10. Beaucoup d'éléments de DX 10 s'y retrouvent avec peu ou pas de modifications. Mais beaucoup d'autres ont été révisés et beaucoup de nouveaux éléments ont été ajoutés. Voici certaines de ses caractéristiques principales :

- Direct3D 11 peut souvent fonctionner sur des cartes graphiques ne supportant pas DX11 grâce à un système de « **niveau de caractéristiques** » (*feature level*).
- Implantation de la **Tessellation** dans les étapes du pipeline (stages)
- Un nouveau type de shader, le *Hull Shader*, permettant de préparer ou de contrôler la tessellation.
- Un nouveau type de shader, le *Domain Shader* permettant de terminer la tessellation.
- Support pour le **Multithreading**. Cependant, le multithreading offre en pratique peu de gain dû à des limitations fondamentales de la librairie.
- Des ressources graphiques maintenant adressables donc plus facilement modifiables en cours de traitement — *Textures*, *Constant Buffers*, et *Samplers*
- De nouveaux types de ressources comme des tampons ou des textures.
- Le *Compute Shader* — un shader permettant de faire des calculs et pas nécessairement lié au pipeline graphique.
- Le *Geometry Shader*, déjà présent, permet maintenant une sortie de 1024 sommets.
- Textures : beaucoup d'améliorations ont été apportées aux formats et aux possibilités.
- Un support mémoire plus grand qui permet des ressources de plus de 4GB
- Shader Model 5
- **Draw Indirect** - DirectX 10 implémentait déjà **DrawAuto**, qui pouvait utiliser du contenu généré par le GPU et en faire le rendu sur le GPU.. DirectX 11 améliore DrawAuto de façon à ce qu'il puisse être utilisé par un *compute shader*. Une fonctionnalité qui ne sera pas abordée dans le cours, mais qui est un des ajouts les plus importants de DirectX 11

DirectX 12

DX 12 est la dernière version de DX à ce jour. Elle ne vise pas à remplacer DX 11, mais à offrir une abstraction de plus bas niveau pour permettre des gains de performances. La librairie offre beaucoup plus de contrôle aux développeurs en échange de beaucoup plus de travail.

- L'application est maintenant responsable de la gestion de mémoire graphique.
- Les différents états du pipeline graphique sont maintenant regroupés en 1 objet : le **Pipeline State Object** (PSO). Les PSOs peuvent être créés en dehors de l'application et chargés à l'initialisation. Ils permettent de réduire le coût du changement d'état dans le pipeline de rendu.
- Support du **Multithreading** amélioré.
- Gestion des manuelles des
- meilleurs supports des Multi-GPU.
- Shader Model 5.1 et plus

1. Le pipeline de rendu

L'infographie 3D a pour but de produire des images et des animations (c'est-à-dire des séquences d'images) au moyen d'un « système graphique ». Celui-ci est normalement composé du matériel et du logiciel nécessaires à la production des images qui nous intéressent.

C'est un domaine qui a aujourd'hui plus de trente ans d'existence et la terminologie ainsi que plusieurs concepts ont évolué de façon différente selon les plateformes, les bibliothèques, les objectifs et les auteurs.

Objectifs du chapitre

- ✓ Se familiariser avec les différentes étapes du pipeline de rendu.
- ✓ Se familiariser avec la terminologie tournant autour des sommets, des primitives, des objets 3D.
- ✓ Comprendre les différentes topologies de primitives.
- ✓ Comprendre les différentes opérations nécessaires au rendu d'un objet 3D.
- ✓ Se familiariser avec certains aspects d'optimisation tel le retrait des surfaces cachées.
- ✓ Comprendre le rôle d'un tri de profondeur

1. Le pipeline de rendu

Théorie

1.1 Le pipeline

Rendu (Rendition) (*rendering*)

Le rendu fait référence à l'ensemble des méthodes utilisées pour ajouter la couleur, l'ombrage, les dégradés de lumière, la texture et tout autre attribut de surface que nos objets 3D peuvent posséder. Le résultat du rendu est une image à l'écran.

Le terme **pipeline de rendu** se réfère à l'ensemble des transformations et des opérations appliquées aux données originales d'un objet 3D de façon à obtenir un résultat sur l'écran (ou ailleurs...). Nous décrirons ici un pipeline simplifié et plutôt classique (*voir figure 1.4*). Des variations existent et les nouvelles cartes graphiques (multi-GPU, **shaders** 4 et 5) introduisent de nouveaux modèles de pipelines. Plusieurs auteurs présentent un modèle de pipeline [Möller et Haines 1, chapitre 2] [Luna 1, chapitre 5], mais il s'agit plus d'un concept que d'un modèle concret.

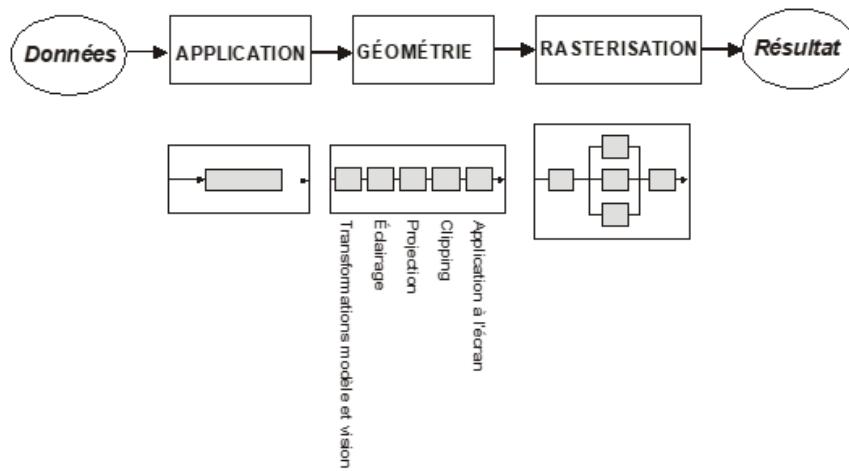


Figure 1.1 Le pipeline de rendu selon Möller et Haines.

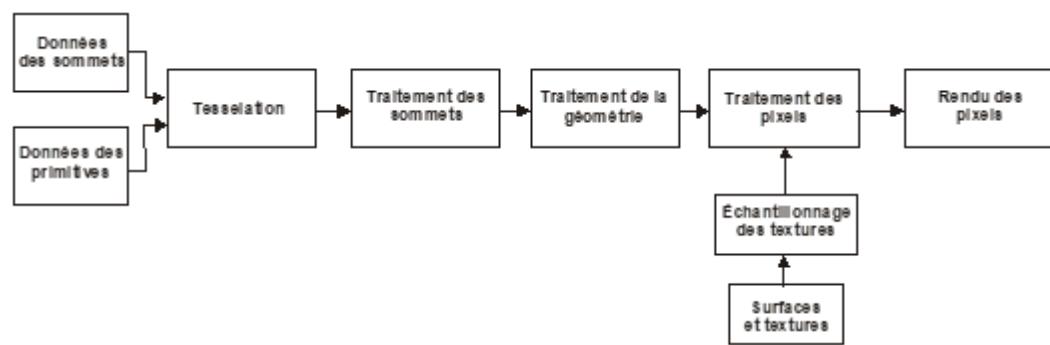


Figure 1.2 Le pipeline de rendu de DirectX 9 selon la documentation DirectX.

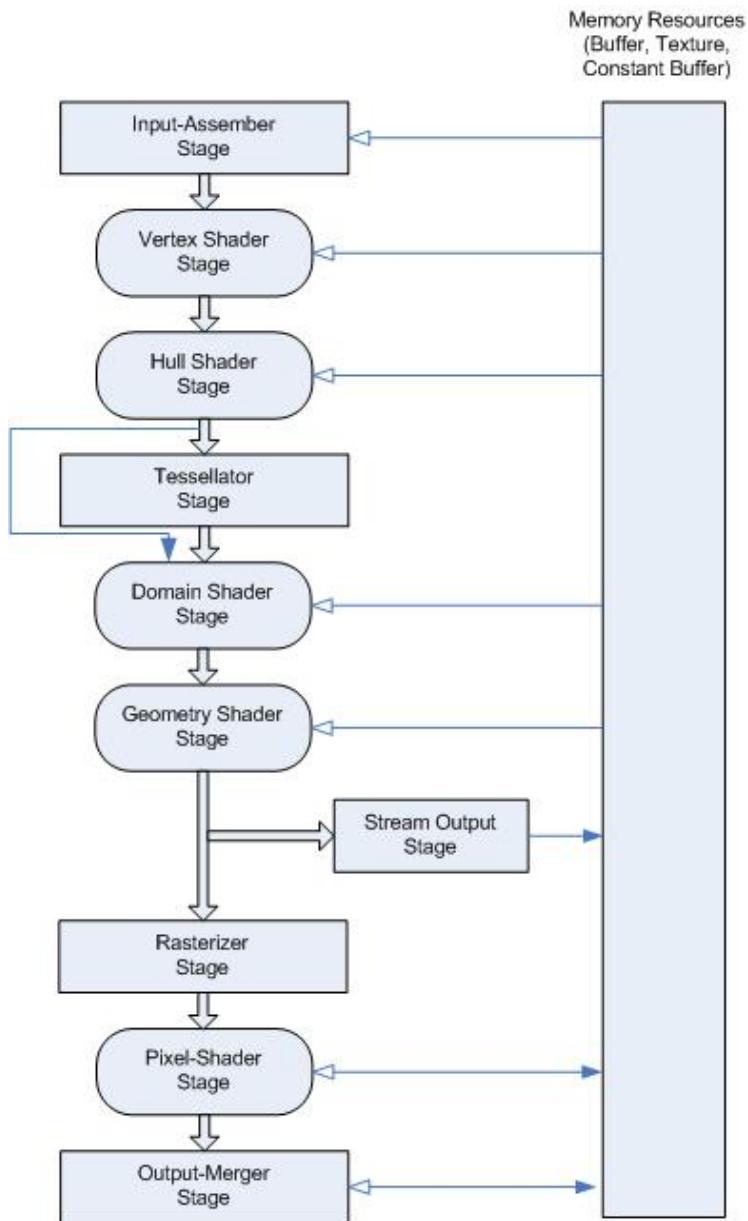
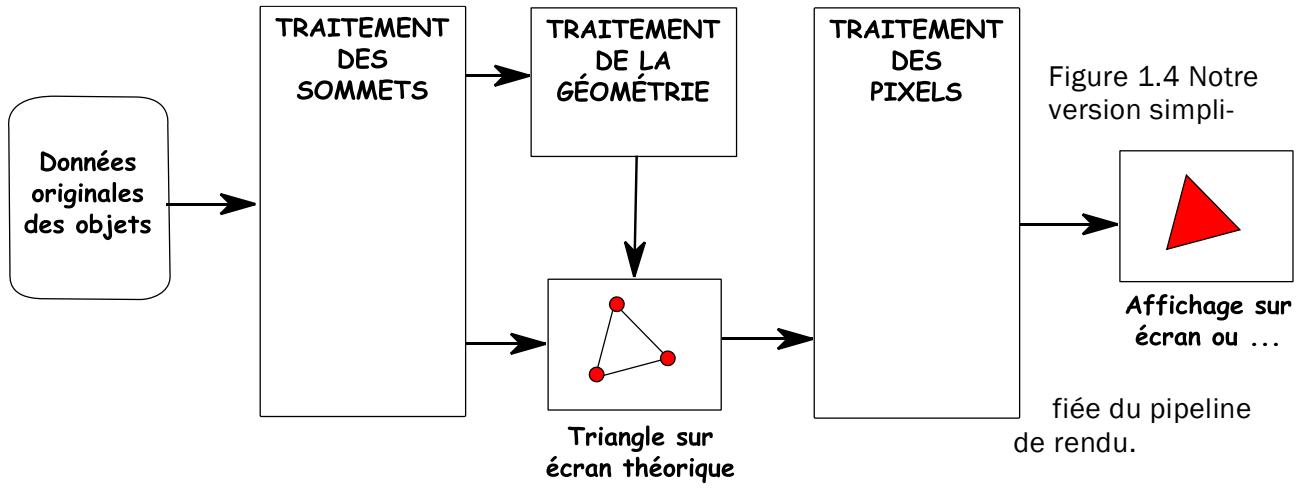


Figure 1.3 Le pipeline de rendu de DirectX 11 selon la documentation DirectX.



1.2 Les données originales des objets 3D

La façon la plus répandue aujourd’hui de **modéliser** (représenter) un objet 3D est de définir celui-ci au moyen de **polygones**, habituellement des triangles. Les polygones sont définis par programmation ou au moyen d’un logiciel de modélisation (par exemple 3DSMax) au moyen de leurs **sommets**.

Les sommets

Un sommet est généralement, comme en mathématiques, une position (un point) dans un espace 3D (pléonasme volontaire...). Mais en infographie 3D, les sommets deviennent des objets plus élaborés : en plus de la position, on y retrouvera presque toujours d’autres informations telles la normale, des coordonnées de texture, des informations de couleur, et même des données spécifiques à l’application.

Pour certains traitements, il serait même possible d’avoir des sommets *sans position...*

Il sera aussi possible de compléter ou de modifier la modélisation par programmation lors du traitement de la géométrie (shaders 4 et +).

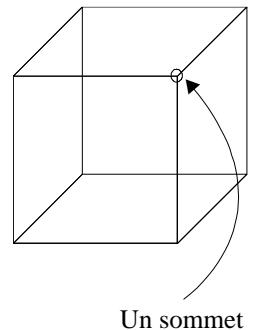
Primitive 3D

À l’origine, une **primitive** était un objet graphique tridimensionnel simple (bloc, sphère, etc.) qui pouvait être utilisé pour créer des objets plus complexes.

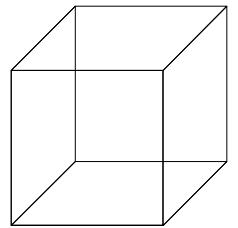
Aujourd’hui, une primitive correspond habituellement à une liste de sommets et une « méthode » (topologie) pour les rassembler et construire les polygones (par exemple la **liste de triangles**).

Une primitive est généralement dessinée au moyen d'**une seule** instruction de rendu (par exemple la fonction **Draw** de DirectX). À l’origine, les objets ainsi dessinés étaient assez simples, mais aujourd’hui ils peuvent facilement atteindre plusieurs millions de polygones. De façon générale, les polygones constituant une primitive ont les mêmes attributs de rendu soient texture, lissage, etc. (Ce n’est plus tout à fait vrai, mais c’est un bon concept de départ).

IMPORTANT :Certains auteurs limitent le terme primitive à l’élément graphique utilisé pour le dessin d’un objet 3D soient les **polygones** (triangles) et aussi à l’occasion les **points** et les **lignes**.



Un sommet



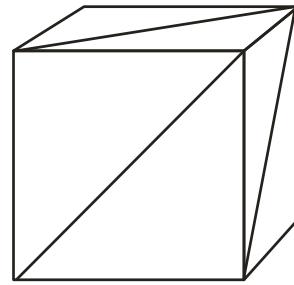
Une primitive

Représentations des primitives (topologies)

Une primitive devient donc un ensemble de polygones (triangles). Il existe plusieurs façons de définir ceux-ci. Une des façons les plus utilisées est de

faire une **liste de triangles** (*triangle list*), chacun de ceux-ci étant identifié par trois sommets :

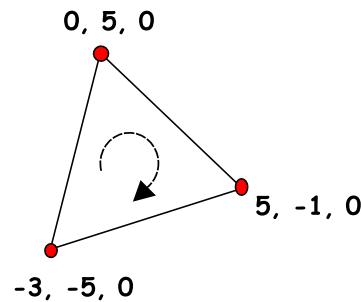
```
0.006556;0.026237;0.017418;    // premier triangle
0.005834;0.022446;0.022099;    // premier triangle
-0.008154;0.021845;0.024459;    // premier triangle
-0.011180;0.025823;0.021993;    // deuxième triangle
0.006733;0.040553;0.017223;    // deuxième triangle
-0.007337;0.045543;0.021333;    // deuxième triangle
0.010438;0.057050;0.012640;    // troisième triangle
-0.009031;0.062243;0.017385;    // troisième triangle
-0.012501;0.071133;0.011865;    // troisième triangle
... etc.
```



Topologie — Liste de triangles

Une **liste de triangles** est comme son nom l'indique une liste de triangles qui n'ont pas à être près l'un de l'autre (mais qui peuvent l'être).

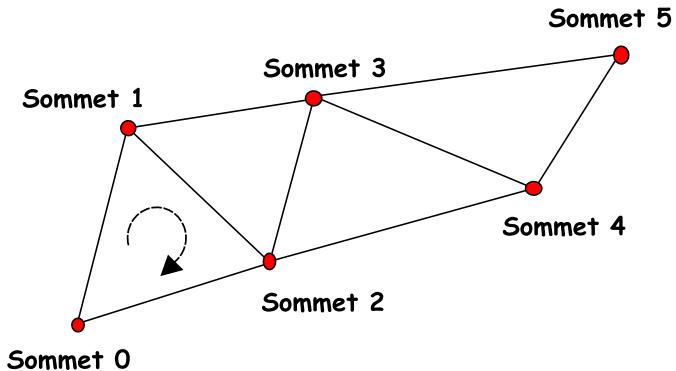
Les triangles sont généralement construits à partir d'une liste de sommets (positions et attributs) dont l'ordre permet de déterminer les triangles.



Attention, le sens où nous « tournons » un triangle est important. Il pourra nous servir avec certaines techniques de retrait des surfaces cachées (voir plus loin).

Topologie — Bande de triangles

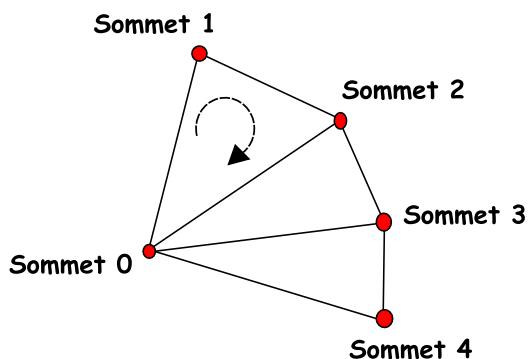
Une **bande de triangles** (*triangle strip*) est une liste de triangles interconnectés. Le premier triangle est construit au moyen des trois premiers sommets (comme avec une liste de triangles), et les triangles suivants sont construits à partir des deux derniers sommets du triangle précédent et d'un nouveau sommet.



En théorie, cette méthode est plus intéressante que la liste de triangles car elle semble nécessiter moins de sommets et moins de traitements de sommets. Elle fut longtemps la méthode privilégiée pour la construction de gros objets 3D. Pour plusieurs raisons, ce n'est plus le cas et la liste de triangles est habituellement la méthode de construction choisie par les logiciels de modélisation 3D.

Topologie — Éventail de triangles

Un **éventail de triangles** (*triangle fan*) est une liste de triangles interconnectés disposés en éventail. Le premier triangle est construit au moyen des trois premiers sommets (comme avec une liste de triangles), et les triangles suivants sont construits à partir du sommet 0, du dernier sommet du triangle précédent et d'un nouveau sommet.



Cette topologie, intéressante pour la « calotte » d'une sphère, n'a malheureusement pas beaucoup d'autres applications, mais si elle vous convient...

Topologie — Liste de points

Il s'agit d'une collection de sommets qui seront rendus comme de simples points. Avec les résolutions modernes, ces points sont plutôt petits, mais si vous en avez besoin...

Topologie — Liste de lignes

Il s'agit d'une liste de sommets dont chaque paire sera rendue comme une ligne. Attention, ici aussi la ligne est très mince...

Topologie — Bande de lignes (*line strip*)

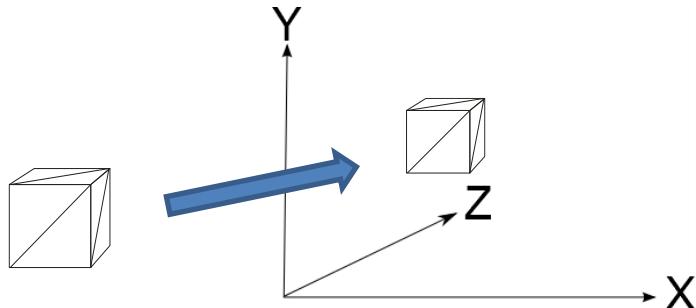
Il s'agit d'une liste de sommets dont la première paire sera rendue comme une ligne puis chaque nouveau sommet sera combiné au sommet précédent pour obtenir une ligne.

Topologies — Autres

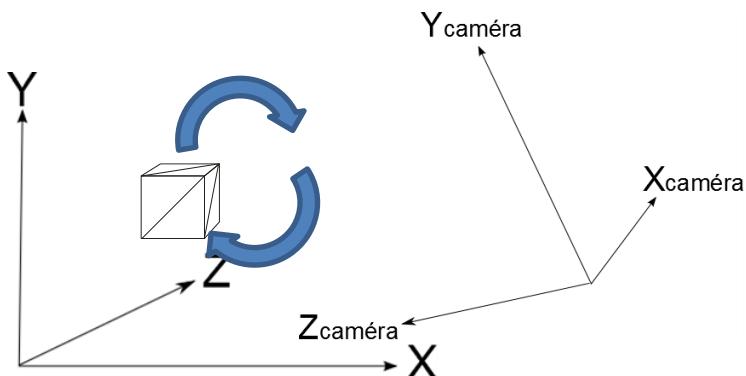
DirectX 11 offre des variantes des topologies « triangles » et « lignes » destinées entre autres à faciliter le travail que nous pourrions faire dans les « *geometry shaders* ».

1.3 Le traitement des sommets

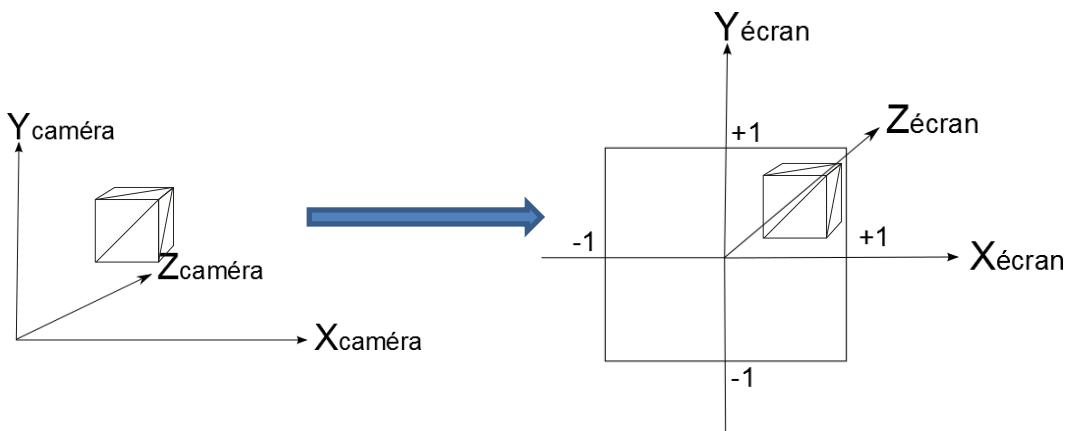
Le traitement des sommets est une opération permettant de transformer le plus rapidement possible un sommet de l'espace original de l'objet (espace objet) à l'espace de l'écran. On en profitera pour transformer aussi certains attributs de nos sommets. Dans presque tous les systèmes graphiques, cette transformation est modélisée ainsi :



1. L'objet est placé dans l'espace monde (transformation **Monde**)



2. Les sommets sont convertis pour l'espace caméra (transformation **Vision**)



3. Les sommets sont convertis pour l'espace écran (transformation **Projection**)

L'espace objet

Nos objets sont modélisés au moyen de sommets dont les coordonnées sont définies habituellement dans **l'espace objet** c'est-à-dire un espace propre à chaque objet.

L'espace monde (*world space*)

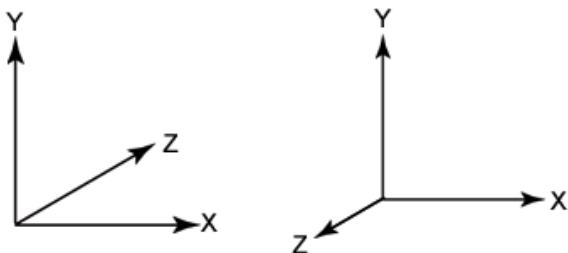
Nos objets seront ensuite positionnés dans un espace appelé **monde** (*world*). Ce sera l'espace de notre scène 3D, celui dont on veut faire le rendu.

Cet espace utilise des unités qui nous sont propres, mais qui devraient tout de même être choisies pour tirer profit de la précision des nombres, habituellement des éléments de type *float* en C++ et dans les *shaders*.

Système de coordonnées 3D

Les applications 3D utilisent habituellement un des deux types de coordonnées cartésiennes, « **main gauche** » ou « **main droite** ». Ces systèmes sont ainsi nommés en fonction de leur représentation au moyen des mains.

Notez bien que les explications sur la provenance des termes sont parfois contradictoires dans certains livres. La « vraie » définition (bien sûr...) est tout simplement que le pouce correspond à l'axe des X, l'index à celui des Y et le majeur à celui des Z. Il n'y a qu'une seule façon de placer ces doigts à angle droit pour chaque main, ce qui définit les deux systèmes.



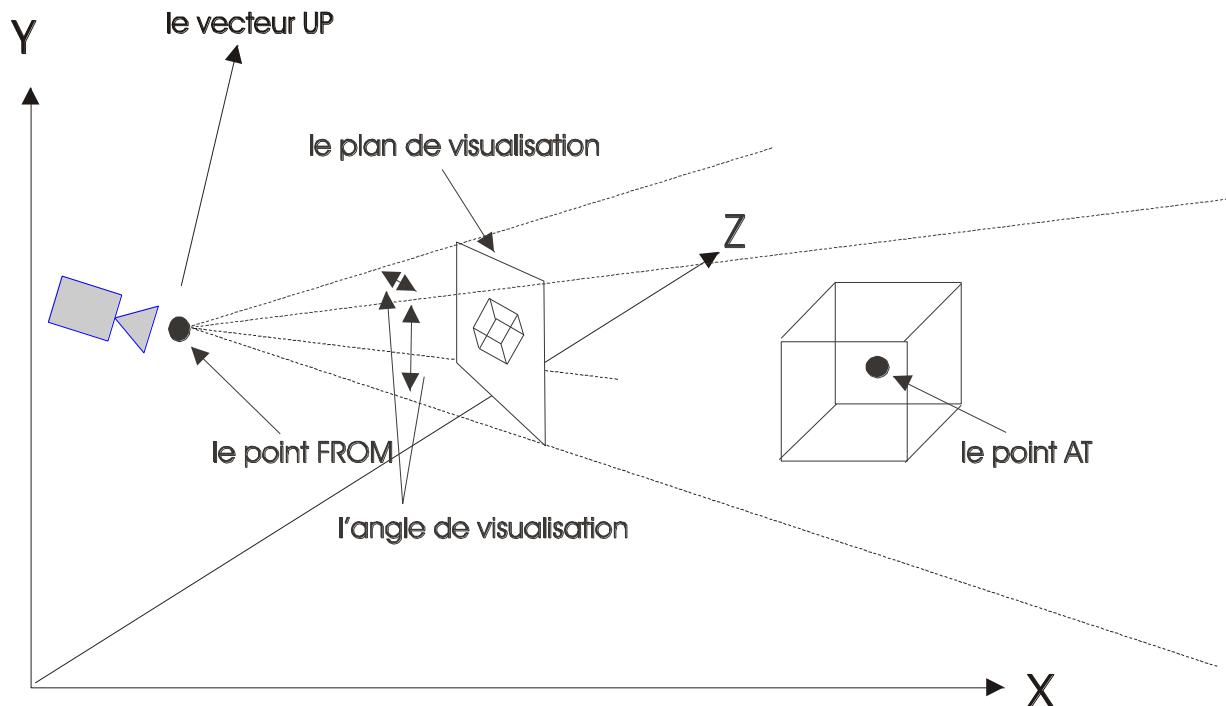


La documentation de Direct3D utilise presque toujours un système « main gauche » (OpenGL est presque toujours « main droite »), mais il est tout à fait possible de définir nos coordonnées en main-droite puisque ce n'est qu'une version différente de notre matrice de vision. Aucun des deux systèmes n'est plus avantageux puisque mathématiquement, c'est la même chose.

Le choix d'un système (main gauche ou main droite) se retrouvera dans l'organisation de nos objets dans le monde et aussi pour la définition de notre « caméra ».

L'espace final, c'est-à-dire l'espace de l'écran, ainsi que les coordonnées de pixels seront presque toujours exprimés selon un modèle « main gauche ».

L'espace vision (*view space*)



Certains termes associés à la vision et à la projection

L'espace vision est tout simplement un changement de base pour nous permettre de **simuler** l'utilisation d'une **caméra**. La caméra est un objet conceptuel.

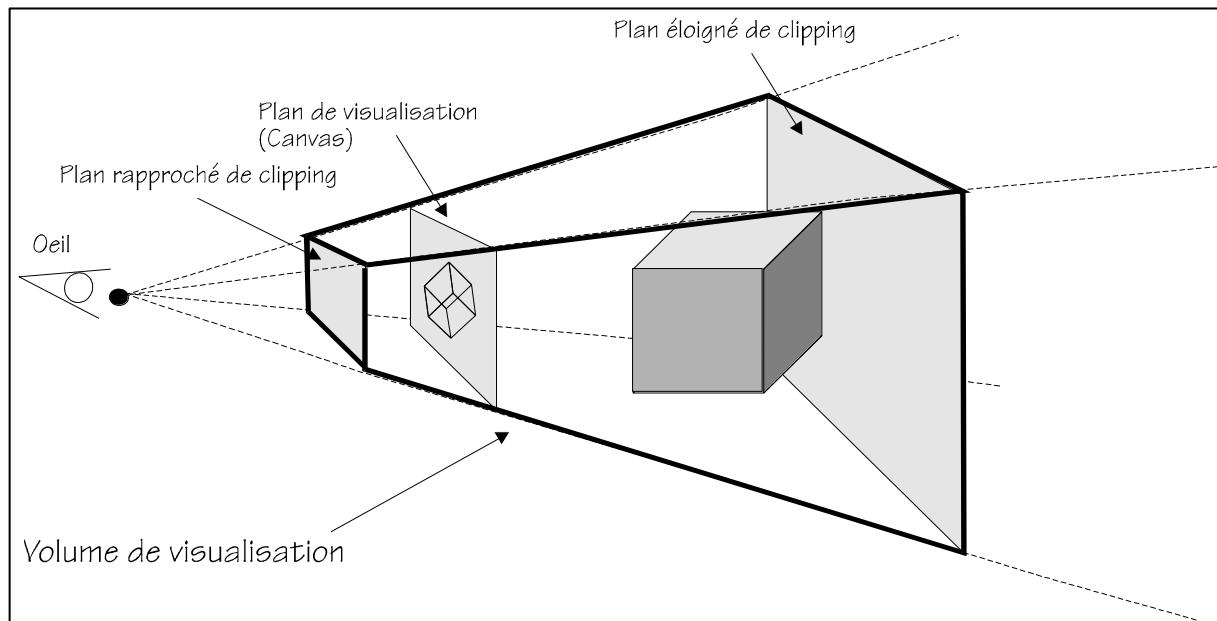
Les systèmes 3D ne disposent pas réellement d'une caméra ou d'un œil. Il s'agit plutôt d'un concept permettant de regrouper les informations de projection vers l'écran. Par exemple, la caméra possède souvent les attributs théoriques **From** et **At** (des points) qui déterminent le sens de la visualisation et l'attribut **Up** (un vecteur) qui détermine le sens qui sera en haut lors de l'affichage 2D sur l'écran. La caméra possède aussi un angle de champs. Cet angle détermine la façon dont nos objets sont projetés sur le plan de visualisation.

L'angle de visualisation agit comme un zoom sur une caméra. Si l'angle est petit, les objets se rapprochent rapidement (*zoom in*), par contre, s'il est grand, nous pouvons même donner l'impression de reculer (*zoom out*).

La caméra étant la plupart du temps un objet théorique correspondant en réalité à une de nos transformations de la vision. Il nous faudra la gérer au moyen de son propre système de coordonnées. Il suffit souvent d'effectuer des transformations pour les obtenir, mais certains noyaux 3D les conservent en permanence (surtout pour l'animation).

Volume de visualisation

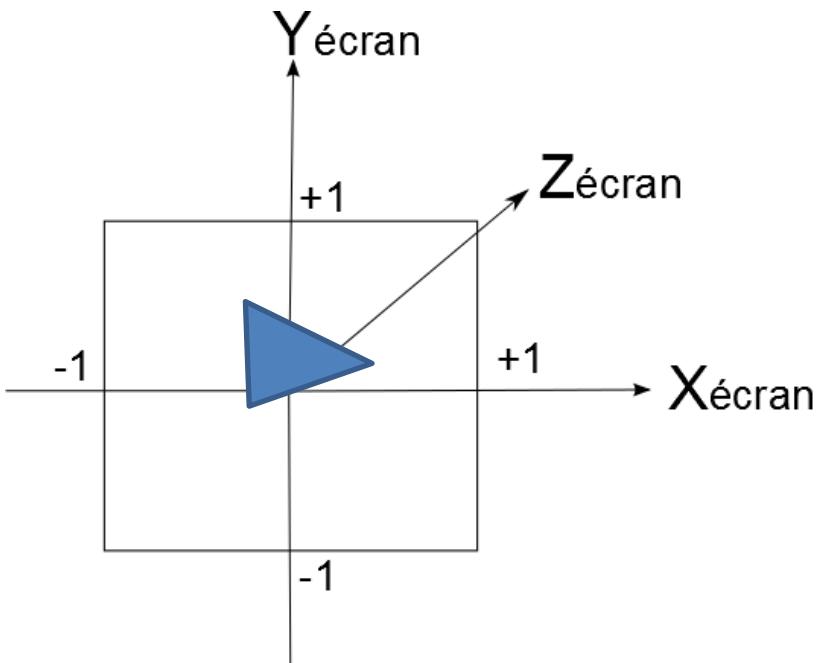
Ce terme fait référence au volume d'espace compris entre deux plans spécifiques appelés plan rapproché de *clipping* et plan éloigné de *clipping*. Le plan de visualisation se situe entre les deux, généralement assez près du plan rapproché de clipping.



L'espace écran (*screen space*)

L'espace écran est souvent appelé **espace projection** (*projection space*) puisque la transformation permettant de convertir nos coordonnées implique (entre autres) une projection.

Nos coordonnées 3D de l'espace vision sont converties en coordonnées « presque 2D » sur le plan de visualisation. Je dis « presque 2D » puisque le Z est souvent conservé pour une utilisation spéciale : le tri de profondeur Z.



Théorie

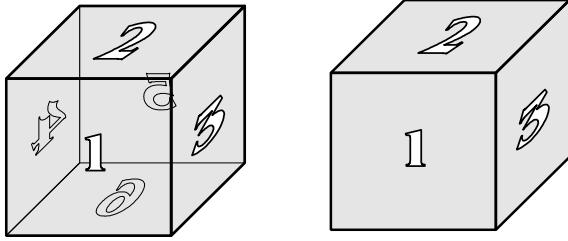
1.4 Le traitement de la géométrie

L'étape suivante, le **traitement de la géométrie** (automatique et/ou via un *geometry shader*) sera responsable d'assembler nos triangles et de les rejeter au besoin, le cas le plus courant étant un triangle à l'extérieur du volume de visualisation (*clipping*). Certains algorithmes de retrait des surfaces cachées pourront aussi y être activés, par exemple le « *backface culling* ».

Retrait des surfaces cachées

Ce terme regroupe une série de techniques 3D et 2D pour déterminer quelles surfaces sont totalement ou partiellement cachées soit par d'autres objets soit par l'orientation de l'objet par rapport à l'œil. Ces techniques peuvent conduire à des méthodes très sophistiquées. Il existe aussi des techniques plus simples comme le « pile ou face » des surfaces et le **tri de profondeur Z-Buffer**.

« Pile ou face » ou « vraie face » (*backface culling*)



C'est une méthode de **retrait des surfaces cachées** très simple qui n'affichera que le côté « face » des polygones. Le côté « pile » indique la surface d'un polygone n'étant jamais affiché, l'intérieur, par exemple.

Pour détecter si un polygone nous montre son côté « face » on regarde si les points à dessiner sont, par exemple, disposés de façon horaire. On aurait pu aussi choisir antihoraire.

Tri de profondeur — Algorithme du peintre

C'est une méthode simple de retrait des surfaces cachées. Elle consiste à trier toutes les facettes de notre scène selon leur distance de l'œil. C'est relativement simple en principe, mais les résultats ne sont pas toujours bons. De plus, il faut trier à chaque mouvement de l'œil. Elle est rarement utilisée pour le rendu, mais... elle servira à l'occasion pour certains effets.

Tri de profondeur — Algorithmes Z-Buffer et W-Buffer

Ce sont des techniques très efficaces. Elles sont aujourd'hui disponibles sur toutes les cartes graphiques 3D. Elles fonctionnent au niveau **du traitement des pixels**. Chaque pixel est accompagné de sa profondeur et on n'af-

fiche en fin de compte que le plus près. Notez que le W-Buffer est présentement un peu délaissé et que la plupart des cartes graphiques ne le supportent plus. Nous en reparlerons plus en détail dans une autre section.

Tri de profondeur — Algorithme BSP (*Binary Space Partitionning*)

C'est une vieille technique encore efficace, surtout avec des **objets**. Elle est plus simple à implémenter que *Z-Buffer* et théoriquement moins lourde en temps-machine (ce n'est plus vraiment le cas aujourd'hui si nous l'utilisons au niveau des polygones). Mais elle couvre moins de cas que *Z-Buffer*. Combinée avec la méthode « Pile ou face » ainsi qu'avec certaines règles de modélisation, nous pourrions toutefois l'utiliser et couvrir presque tous les cas.

Cette méthode et d'autres qui s'en inspirent (*quadtree* et *octree* par exemple) seront toutefois très utiles pour nous permettre de gérer la distance des objets par rapport à l'œil et ainsi d'implanter certaines techniques de **niveaux de détails** [GPG 6, Fleiz, Martin —Spatial Partitioning using an Adaptative Binary Tree].

2. Un modèle de base d'animation 3D

Dans ce chapitre, nous regarderons l'organisation et la structure d'un modèle de base d'animation qui nous sera particulièrement utile en 3D pour toute application nécessitant une animation en temps réel (applications multimédias, jeux, applications industrielles, etc.). Nous en profiterons pour examiner certains concepts particulièrement utiles et pour régler nos comptes avec certains mythes fréquemment véhiculés.

Objectifs du chapitre

- ✓ Faire certains choix dans le design d'un modèle d'animation.
- ✓ Faire un survol de certains problèmes pouvant se poser à nous.
- ✓ Présenter PetitMoteur3D.
- ✓ Se familiariser avec le démarrage d'un projet DirectX.
- ✓ Choisir un modèle d'animation.
- ✓ Comprendre certains aspects de l'optimisation d'une application 3D.

2. Un modèle de base d'animation 3D

Théorie

2.1 Le design d'un modèle d'animation

Regardons rapidement les éléments de design dont nous aurons besoin dans une application nécessitant une animation temps réel durant la majeure partie de son exécution. Ces éléments nous serviront de point de départ pour notre modèle.

Organisation du programme

Un découpage du programme en unités fonctionnelles bien identifiées permet de mettre chaque chose à sa place et de gagner un temps incroyable en développement. Chaque unité fonctionnelle doit avoir un rôle bien précis dans le programme de façon à ce que nous puissions nous y retrouver plus facilement, par exemple les fonctions d'initialisation des objets se retrouveront dans l'initialisation de la scène.

La performance

Un vieux proverbe informatique dit « Commencez par faire fonctionner votre programme, vous le rendrez plus rapide par la suite ». C'est vrai pour certains aspects de nos applications, mais la performance d'une application 3D est pour beaucoup une question de design. Les choix effectués au début du développement auront souvent plus d'impacts que les séances d'optimisation en fin de cycle.



PetitMoteur3D

Notre petit moteur 3D (nommé PetitMoteur3D) est comme son nom l'indique, un moteur de rendu 3D. Ce n'est peut-être pas le meilleur design pour un moteur de jeu même si plusieurs moteurs de jeu sont construits autour du « moteur 3D ». Vous trouverez toutefois beaucoup de points communs entre les choix que nous ferons et ceux des moteurs de jeu. Toutefois, PetitMoteur3D se veut d'abord et avant tout un modèle pédagogique donc la simplicité du code (hum !) et une certaine indépendance des éléments seront prioritaires sur la performance.

Dans ce chapitre, nous nous concentrerons sur l'architecture de base de l'application et nous commencerons les aspects 3D au prochain chapitre.

Atelier

2.2 Application de base

- Faites-vous une copie du dossier **Chapitre 2 — Départ** (voir les documents d'accompagnement). Renommez le dossier **PetitMoteur3D**.

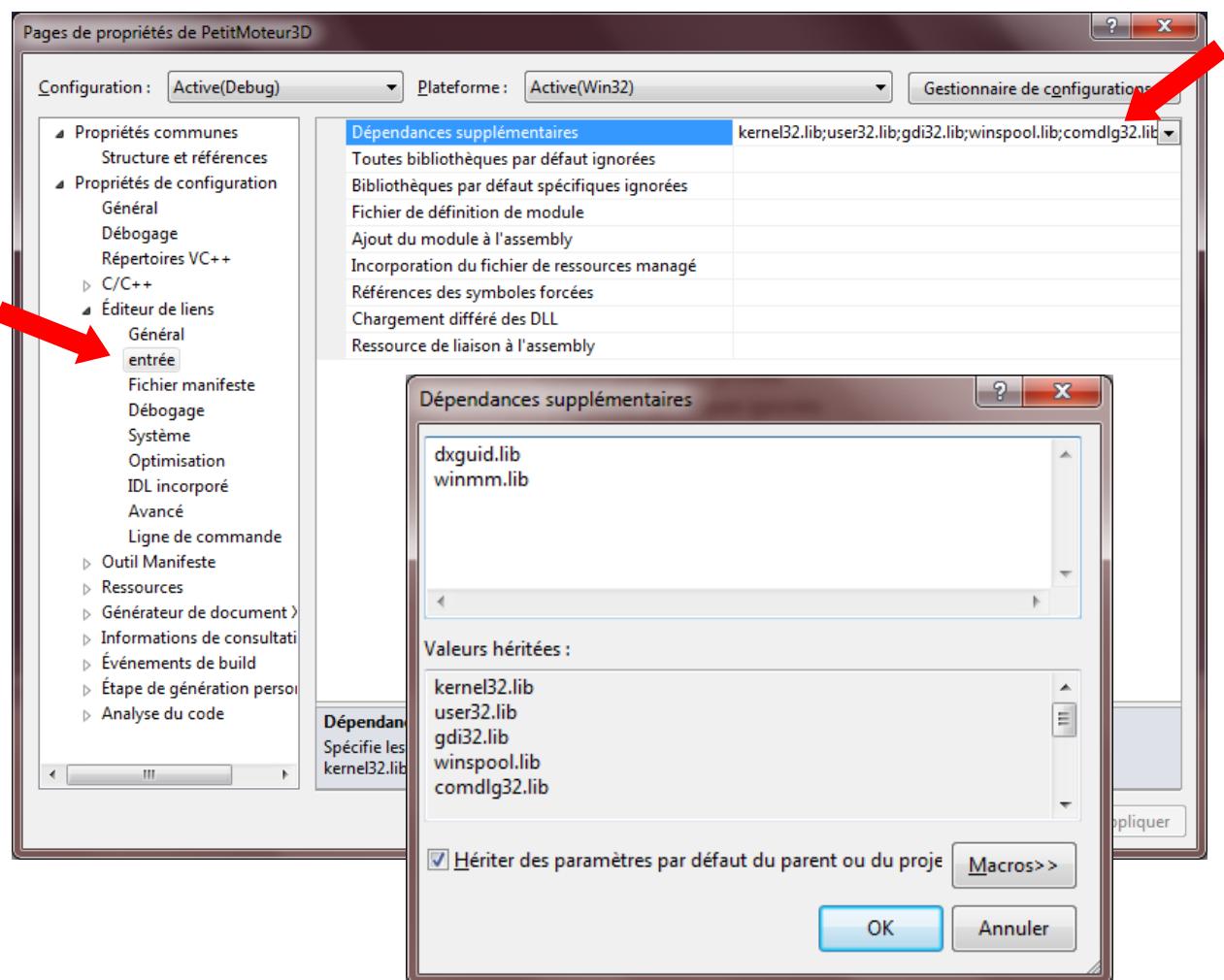
Il s'agit pour le moment d'une petite application Windows sans barre de boutons, sans barre de statut, sans support OLE, etc. Seulement un petit menu qui nous servira à l'occasion. On veut une application minimale...

- Dans le dialogue Pages de propriétés de **PetitMoteur3D** sous la rubrique **Éditeur de liens|Entrée**. Ajoutez les bibliothèques **dxguid.lib** dans le champ **Dépendances supplémentaires**.

DXGuid ?

Il s'agit des GUID pour DirectX. Un GUID (*globally unique identifier*) est une valeur d'identification dans le codage assure une représentation « unique mondialement ». La valeur est souvent représentée par 32 caractères correspondant à des valeurs hexadécimales, mais est en réalité conservée sous la forme d'un entier de 128 bits.

Le nombre total de clés (2^{128}) est tel que le risque de construire deux fois la même clé est minime. De plus, certaines techniques sont implantées pour renforcer l'unicité des clés.



Atelier

2.3 Configuration de l'environnement

Le SDK de DirectX est maintenant intégré au Windows SDK. Consultez la documentation pour l'utilisation de projet « pré DX11 ».

Atelier

2.4 Notre application

Ce que l'on a

pour le moment, PetitMoteur3D ne fait pas beaucoup de choses comme moteur 3D, mais :

- Nous avons réglé plusieurs aspects de base reliés à une application Windows
- Nous avons un objet de classe **CMoteurWindows**. Cette classe est spécifique à la programmation Windows avec DirectX (elle le deviendra !). Les aspects de programmation plus génériques seront placés dans la classe **CMoteur**, cette classe (en fait un *template*) est une classe dérivée de la classe **CSingleton** qui comme son nom l'indique reprend le *design pattern singleton* qui nous permettra d'implanter élégamment un certain nombre d'items système ne devant exister qu'en un seul exemplaire et devant être accessible un peu partout dans notre programme.
- Certaines fonctions sont déjà en place (**Run** par exemple).
- Les éléments spécifiques à la programmation Windows et DirectX seront implantés dans des fonctions portant le nom **xxxSpecific**, par exemple **RunSpecific**. Notre but étant de placer les éléments spécifiques dans des fonctions spécialisées et de bien placer les appels dans la fonction de la classe de base pour éviter les surcharges « sauvages ». Nous aurons la possibilité d'effectuer des surcharges de fonction génériques autres que les « *specific* », mais nous essaierons de l'éviter.

NOTE

À cause du sujet (la programmation 3D avec DirectX...), je ne mettrai pas d'efforts particuliers à définir un moteur 3D « 100 % portable » (*peu importe ce que ça veut dire*).

Par contre, une certaine distinction des aspects reliés à l'application, des aspects génériques de l'infographie et des aspects spécifiques à la plateforme nous permettra :

- de mieux organiser notre application ;
- de différencier les éléments spécifiques à l'application des éléments de bibliothèque ;
- de mieux cerner les aspects techniques de DirectX ;
- de faciliter la migration vers une nouvelle version de DirectX ;
- d'envisager plus facilement la réalisation d'une version de notre application pour une autre plateforme (*je n'ai pas dit porter* !)

```

int APIENTRY _tWinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPTSTR lpCmdLine,
                      int nCmdShow)
{
    // Pour ne pas avoir d'avertissement
    UNREFERENCED_PARAMETER(hPrevInstance) ;
    UNREFERENCED_PARAMETER(lpCmdLine) ;
    UNREFERENCED_PARAMETER(nCmdShow) ;

    try
    {
        // Création de l'objet Moteur
        CMoteurWindows& rMoteur =
        CMoteurWindows::GetInstance(); 1

        // Spécifiques à une application Windows
        rMoteur.SetWindowsAppInstance(hInstance); 2

        // Initialisation du moteur
        rMoteur.Initialisations(); 3

        // Boucle d'application
        rMoteur.Run(); 4

        return (int)1;
    }

    catch (const std::exception& E)
    {
        const int BufferSize = 128;
        wchar_t message[BufferSize] ;

        size_t numCharacterConverted;
        mbstowcs_s(&numCharacterConverted, message,
E.what(), BufferSize - 1) ;

        MessageBox(NULL, message, L"Erreur",
MB_ICONWARNING) ;

        return 99 ;
    }
}

```

1. On obtient une référence sur l'objet CMoteurWindows. Celui-ci est un singleton qui est en réalité la version spécifique de CMoteur. Nous travaillerons sur les deux niveaux (CMoteur et CMoteurWindows).
2. Malgré tous nos efforts, il sera parfois nécessaire d'implanter de nouvelles fonctions publiques dans notre descendant de CMoteur. Dans le cas présent, nos applications Windows utiliseront des fonctions qui nécessiteront l'instance Windows (le paramètre hInstance) comme paramètre. Nous limiterons évidemment ce genre de fonctions.
3. Appel de la fonction d'initialisation, j'y ai ajouté un « s » car il y aura beaucoup d'initialisations : l'application, la fenêtre, les objets, la scène, le dispositif de rendu, etc.
4. La fonction Run est la boucle de traitement de l'application. J'utiliserais aussi le terme « **boucle d'animation** », mais elle est responsable de l'animation, du son, de la logique d'affichage, etc. Le nom de la fonction (**Run**) provient de l'implantation de fonctions du même type dans plusieurs architectures d'applications (dont MFC), d'autres architectures implantent de façon similaire la boucle message dans une fonction appelée **Execute**.
5. Le traitement des exceptions est simplifié pour le moment, notre but sera de sortir en « catastrophe » si un problème imprévu se produit. Nous modifierons plus tard le traitement des exceptions pour obtenir des messages plus significatifs.

La fonction CMoteurWindows ::SetWindowsAppInstance

(MoteurWindows.cpp)

Cette fonction est un « mal nécessaire » pour initialiser une valeur dans l'objet CMoteurWindows. En effet, nous sommes dans une application Windows et nous aurons besoin d'utiliser certains paramètres Windows dont l'instance de l'application et plus tard la fenêtre principale. Plusieurs fonctions Win32 et quelques fonctions DirectX devront spécifier l'instance de l'application.

```
void CMoteurWindows::SetWindowsAppInstance(HINSTANCE hInstance)
{ // Stocke le handle d'instance de l'application,
  // plusieurs fonctions spécifiques en auront besoin

  hAppInstance = hInstance;
}
```

Rappel

hAppInstance est une variable de type **HINSTANCE**. HINSTANCE est un type de **HANDLE** que Windows utilise pour identifier l'occurrence présente de notre application en mémoire.

Un **HANDLE** est tout simplement un numéro de ressource Windows. Si Windows a besoin de changer un objet système d'emplacement en mémoire, il peut le faire. Les applications continuent de se servir du HANDLE plutôt que d'une adresse.

La fonction CMoteur::Initialisations

(Moteur.h)

1. Les initialisations spécifiques à la plateforme (Windows dans notre cas). Voir plus loin.
2. D'autres initialisations suivront dont le dispositif de rendu et l'initialisation de la scène (objets, paramètres...)
3. L'initialisation de l'animation est pour le moment en commentaire, nous l'implanterons un peu plus loin.

```
virtual int Initialisations()
{
  // Propre à la plateforme
  InitialisationsSpecific() ; 1

  // * Initialisation du dispositif de rendu
  // * Initialisation de la scène 2
  // * Initialisation des paramètres de l'animation
  et
  // préparation de la première image
  //InitAnimation() ; 3

  return 0 ;
}
```

La fonction CMoteurWindows::InitialisationsSpecific

(MoteurWindows.cpp)

```
int CMoteurWindows::InitialisationsSpecific()
{
    // Initialisations de l'application ; 1
    InitAppInstance();

    Show(); 2

    return 0;
}
```

1. **InitAppInstance** est une fonction qui enregistre et crée la fenêtre principale Windows. J'ai au départ créé mon application avec l'Assistant de nouveau projet de VS et j'ai transporté les éléments qui étaient dans les fonctions WinMain et InitInstance originales dans la fonction CMoteurWindows ::InitAppInstance.

Si vous voulez plus de détails sur les opérations effectuées dans cette fonction, consultez la documentation en ligne (MSDN) ou [PETZOLD PW] Chapitre 3.

2. La fonction **Show** permet de rendre visible la fenêtre principale et de mettre à jour certains aspects de l'interface. Pour l'instant, cette fonction est propre à l'implantation Windows,

La fonction CMoteur ::Run

(Moteur.h)

La boucle d'animation est implantée dans la boucle de l'application qui était dans la fonction **WinMain** (**tWinMain**), mais que j'ai déplacé dans la fonction **Run** de la classe CMoteur et dans la fonction **RunSpecific** de la classe CMoteurWindows

La boucle message Windows proposée pour la plupart des applications Windows ne nous convient pas. Il nous faudra la modifier pour qu'elle soit plus adaptée à une animation en temps réel, qu'elle traite spécifiquement certains messages et évidemment qu'elle intègre notre animation.

Les éléments reliés aux messages Windows seront relégués dans la fonction **CMoteurWindows ::RunSpecific**.

```
virtual void Run()
{
bool bBoucle=true;

    while (bBoucle)
    { 1
        // Propre à la plateforme
        // (Conditions d'arrêt, interface, messages)
        bBoucle = RunSpecific() ;

        // appeler la fonction d'animation 2
        if (bBoucle) bBoucle = Animation() ;
    }
}
```

À noter :

1. La possibilité pour la fonction **RunSpecific** de terminer l'application, tout simplement en retournant **false**.
2. L'appel de la fonction **Animation** à chaque tour de boucle. Et la possibilité pour celle-ci de terminer l'application, tout simplement en retournant **false**.

La fonction CMoteurWindows ::RunSpecific

(MoteurWindows.h)

Cette fonction s'occupe principalement du traitement des messages Windows. Certains de ces messages seront utiles pour notre application. Pour le moment, seulement le message WM_QUIT ainsi que certains messages de dialogues sont traités.

```
bool CMoteurWindows::RunSpecific()
{
MSG msg ;
bool bBoucle=true;

// Y-a-t'il un message Windows à traiter ?
if (::PeekMessage(&msg, NULL, 0,0, PM_REMOVE))
{
    // Est-ce un message de fermeture ?
    if (msg.message==WM_QUIT) bBoucle=false;

    // distribuer le message
    if (!::TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        ::TranslateMessage(&msg) ;
        ::DispatchMessage(&msg) ;
    }
}

return bBoucle ;
}
```

1

- 1 La boucle message originale proposée dans les applications Windows utilise **GetMessage** (API Win32) au lieu de **PeekMessage**.

Normalement, un programme peut utiliser GetMessage pour obtenir le message suivant dans la file de messages Windows. S'il n'y a pas de messages, la fonction GetMessage attend qu'un message soit présent pour revenir (c'est une fonction dormante). C'est très bien pour une application basée interface utilisateur, mais très contraignante pour notre type de programmation. La fonction PeekMessage, elle, retourne immédiatement le contrôle à notre programme même s'il n'y a pas de messages. Notre programme peut donc continuer son travail sans perdre trop de temps avec les messages Windows.

Ce que l'on veut par la suite

Les éléments de notre future application se retrouveront dans trois secteurs d'activités soient les initialisations, la boucle d'animation et la fin du programme.

Les **initialisations** seront les étapes les plus importantes de la plupart des applications 3D (les jeux en particulier). Il ne faut pas oublier que celles-ci

ne sont effectuées qu'une seule fois et qu'il vaut toujours la peine de préparer plein de choses avant de démarrer l'animation (voir l'axiome no 4 sur la performance).

Nos initialisations seront appelées à partir de la fonction **Initialisations**.

Les fonctions d'initialisation de Direct3D, du dispositif de rendu et de la scène seront implantées dans le chapitre suivant. Pour notre modèle de base, nous ne verrons que la fonction **InitAnimation**.

La **boucle d'animation** sera le point critique de notre application, c'est là qu'il faut réellement prendre soin de l'architecture de l'application et de la programmation impliquée. Ne pas oublier qu'en plus de l'affichage, il faut aussi implanter la logique de l'application et les liens avec l'interface utilisateur (plus : le son, les liens réseaux, etc.).

- La boucle d'animation est dans la fonction **Run** de la classe CMoteur.
- La fonction **Animation** correspond à un tour de boucle d'animation, elle appellera **AnimeScene** dont le rôle est de préparer les objets et de vérifier la logique puis la fonction **RenderScene** qui effectue le rendu de la scène sur un tampon d'arrière-plan..

La **fin du programme** est aussi un point important, il faudra effectuer un bon ménage des objets de l'environnement en plus des vérifications d'usage (est-ce qu'il faut sauver les fichiers ? ...). Notre ménage se fera principalement dans les destructeurs des classes CMoteur, CMoteurWindows et des autres classes qui apparaîtront. Nous y reviendrons au chapitre suivant.

2.5 La fonction InitAnimation

- Déclarer dans la classe CMoteur (dans Moteur.h) la fonction InitAnimation dans la section **protected** de la façon suivante :

```
virtual int InitAnimation()
{
    TempsSuivant = GetTimeSpecific() ;
    TempsCompteurPrecedent = TempsSuivant ;
    // première Image
    RenderScene() ;
    return true ;}
```

Notez l'appel de la fonction **RenderScene** pour préparer la première image. Une animation « optimisée » devrait avoir une image en réserve dès le début de l'animation (voir la discussion sur l'animation « optimisée » un peu plus loin dans ce chapitre).

- Déclarer la constante IMAGESPARSECONDE dans la section **namespace PM3D** du fichier **moteur.h** de la façon suivante :

```
namespace PM3D
{
    const int IMAGESPARSECONDE = 60 ;
    const double EcartTemps = 1.0 /
    static_cast<double>(IMAGESPARSECONDE) ;
    ...
}
```

- Déclarer les variables suivantes dans une section **protected** de la classe CMoteur.

```
protected:
// Variables pour le temps de l'animation
int64_t TempsSuivant ;
int64_t TempsCompteurPrecedent ;
```

4. Déclarer dans la classe CMoteur (dans Moteur.h) la fonction **RenderScene** dans la section **protected** de la façon suivante :

```
// Fonctions spécifiques au rendu et à la présentation de la scène
virtual bool RenderScene()
{
    return true ;
}
```

Pour l'instant la fonction ne fait rien du tout puisque nous n'avons encore rien à rendre.

5. Déclarer dans la classe CMoteur (dans Moteur.h) la fonction **GetTimeSpecific** dans la section **protected** de la façon suivante (elle devra être surchargée par l'implantation de la plateforme) :

```
virtual int64_t GetTimeSpecific()=0;
```

6. Déclarer dans la classe CMoteurWindows (dans MoteurWindows.h) la fonction **GetTimeSpecific** dans la section **protected** de la façon suivante :

```
virtual int64_t GetTimeSpecific();
```

7. Implanter dans la classe CMoteurWindows (dans MoteurWindows.cpp) la fonction **GetTimeSpecific** de la façon suivante :

```
/*
// FONCTION : GetTimeSpecific
//
// BUT : Fonction responsable d'obtenir l'heure système
en
// millièmes de seconde
//
int64_t CMoteurWindows::GetTimeSpecific() const
{
    return m_Horloge.GetTimeCount();
}

double CMoteurWindows::GetTimeIntervalsInSec(int64_t start,
int64_t stop) const
{
    return m_Horloge.GetTimeBetweenCounts(start, stop);
}
```

GetTimeCount

m_Horloge.GetTimeCount () utilise les fonctions de chronométrage à haute résolution de Windows. Le marquer de temps retourné à une résolution 1 microseconde ou moins.

GetTimeIntervalsInSec

Pour connaître l'intervalle de temps passé entre deux marqueurs de temps.

Notre horloge

Nous avons besoin d'une façon précise de mesurer le passage du temps dans notre application. Pour ce faire, nous allons créer une classe Horloge.

1. Déclarer la classe horloge dans Horloge.h de la façon suivante.

```
#pragma once

namespace PM3D
{

class Horloge
{
public:
    Horloge();

    int64_t GetTimeCount() const;
    double GetSecPerCount() const { return m_SecPerCount; }
    // retourne le temps en millisecondes entre deux count.
    double GetTimeBetweenCounts(int64_t start, int64_t stop) const;

private:
    double m_SecPerCount;
};

} // namespace PM3D
```

2. Implanter la classe Horloge dans le fichier Horloge.cpp.

```
#include "stdafx.h"
#include "Horloge.h"

namespace PM3D
{

Horloge::Horloge()
{
    LARGE_INTEGER counterFrequency;
    ::QueryPerformanceFrequency(&counterFrequency);
    m_SecPerCount = 1.0 / static_cast<double>(counterFrequency.QuadPart); 1

}

int64_t Horloge::GetTimeCount() const 2
{
    LARGE_INTEGER countNumber;
    ::QueryPerformanceCounter(&countNumber);
    return countNumber.QuadPart;
}

double Horloge::GetTimeBetweenCounts(int64_t start, int64_t stop) const 3
{
    return static_cast<double>(stop - start) * m_SecPerCount;
}

} // namespace PM3D
```

Pour la classe Horloge, nous allons utiliser le chronomètre à haute résolution (< 1us) de Windows. Ce chronomètre mesure le temps en unité de *count*. Pour transformer le nombre de *count* en secondes, nous devons obtenir la fréquence du chronomètre.

1

QueryPerformanceFrequency retourne le nombre de count par seconde de l'horloge. La fréquence renvoyée est fixe pour la durée de l'application et est sauvegardée dans la variable *m_SecPerCount*.

2

QueryPerformanceCounter retourne un marqueur de temps dans un entier 64 bits. Pour éviter les pertes de précisions reliées à la manipulation de nombre flottant, la fonction **GetTimeCount** retourne directement l'entier.

3

GetTimeBetweenCounts retourne le temps en seconde entre deux marqueurs de temps. Pour un maximum de précision, nous retournons l'intervalle de temps dans un double.

Exercice

2.6 La fonction Animation

1. Modifier dans la classe CMoteur (dans Moteur.h) la fonction **Animation** dans la section **protected** pour qu'elle ressemble maintenant à ceci :

```
//  
// FONCTION : Animation  
//  
// BUT : Fonction responsable de l'animation générale  
//       de la scène  
//  
// NOTES :  
//       D'autres animations (autres que 3D) pourront aussi  
//       être appelées ici  
//  
//       L'animation est optimisée pour les systèmes moins rapides  
//       c.-à-d. qu'elle effectue la présentation dès le début puis  
//       Construis la prochaine image.  
//  
virtual bool Animation()  
{  
    // méthode pour lire l'heure et calculer le  
    // temps écoulé  
    const int64_t TempsCompteurCourant = GetTimeSpecific() ;  
    const double TempsEcoule =  
        GetTimeIntervalsInSec(TempsCompteurPrecedent, TempsCompteurCourant) ;  
  
    // Est-il temps de rendre l'image ?  
    if (TempsEcoule > EcartTemps)  
    {  
        // Affichage optimisé  
        // pDispositif->Present() ; // On enlevera « // » plus tard  
  
        // On prépare la prochaine image  
        // AnimeScene(TempsEcoule) ;  
  
        // On rend l'image sur la surface de travail  
        // (tampon d'arrière plan)  
        RenderScene() ;  
  
        // Calcul du temps du prochain affichage  
        TempsCompteurPrecedent = TempsCompteurCourant ;  
    }  
    return true ;  
}
```

À noter

L'affichage de l'image, effectué par la fonction **Present** de notre futur dispositif de rendu est en commentaire.

L'appel de la fonction **AnimeScene** est aussi en commentaire puisque celle-ci ne sera implantée que dans le chapitre suivant.

TempsEcoule

Le temps écoulé est une fraction de seconde. Comme la plupart de nos mouvements seront définis en fonction de 1 seconde, il est facile de multiplier par TempsEcoule pour obtenir un mouvement partiel.

L'animation optimisée

Notre animation est « **optimisée** » puisque nous utilisons un modèle de type Afficher-Préparer au lieu de ce que l'on retrouve dans les tutoriels de DirectX qui sont de type Préparer-Afficher. La différence est importante puisque la plupart des applications professionnelles sont de type Afficher-Préparer. Voici de quoi il s'agit :

Dans toute animation par ordinateur, il faut d'abord préparer l'image à afficher. On vérifie selon la logique quels sont les objets à animer, on les déplace. On vérifie l'état de l'interface utilisateur. On démarre des sons. On effectue l'affichage en arrière-plan sur une surface de travail (*backbuffer*). Cette étape s'appelle la **préparation** et comprend l'**animation de la scène** et le **rendu de la scène** en arrière-plan.

Puis l'image est affichée par une fonction extrêmement rapide dont l'effet s'insère entre deux cycles de rafraîchissement de l'écran. Cette étape s'appelle **l'affichage**.

Voici notre problème :

Nous voulons un maximum d'images par seconde, mais il faut savoir que 15 images (et même 12) affichées à des intervalles de temps exacts donnent des résultats plus agréables pour l'utilisateur que 30 images à intervalles irréguliers. Pourquoi ?

Même si l'œil ne fonctionne pas en images/seconde (il fonctionne en continu), les experts considèrent que l'œil et le cerveau peuvent percevoir de 60 à 80 images par seconde, d'où les fréquences de rafraîchissement des écrans. C'est la théorie, mais en pratique, l'impression rétinienne (la rétention de l'image sur la rétine de l'œil soit de 1/20 à 1/25 de seconde selon les individus) fait en sorte que plus de 20 images/seconde sont inutiles pour 99 % de la population. Le cerveau assurant le lien entre les images. Par contre pour que ce principe fonctionne bien, il faut assurer la régularité des images c.-à-d. il faut que les images apparaissent à des intervalles exacts. C'est le cas dans les dessins animés et au cinéma. Beaucoup de dessins animés n'ont que 12 images/seconde. Le saviez-vous ?

Des intervalles irréguliers créent une dissonance au niveau du cerveau et l'utilisateur percevra les « hésitations » dans l'affichage même à une vitesse moyenne de plus de 30 images/seconde.

Le modèle Préparer-Afficher

Le modèle Préparer-Afficher présenté par plusieurs documents d'introduction à l'animation se présente ainsi :

- a) Un signal (horloge ou autre) est émis à intervalle précis (ex : 1/30 de sec.)

b) L'image est préparée en vérifiant la logique, en démarrant les sons, etc... ce qui prend généralement un intervalle de temps variable et impossible à prévoir exactement.

c) L'affichage a lieu (effectué par la carte vidéo – temps négligeable sur les systèmes modernes).

d) On retourne en a)

Donc l'affichage a lieu à un intervalle de temps irrégulier après le signal. Il en résulte des affichages irréguliers. Sur un système extrêmement rapide, ça ne paraît pas beaucoup sinon c'est désagréable et ce n'est pas professionnel.

Le modèle Afficher-Préparer

Le modèle Afficher-Préparer que nous utiliserons fonctionne de la façon suivante :

a) Une première image est préparée en vérifiant la logique, en démarrant les sons, etc. ce qui prend généralement un intervalle de temps variable et impossible à prévoir exactement. Mais ce n'est pas grave puisque la boucle d'animation n'est pas encore démarrée.

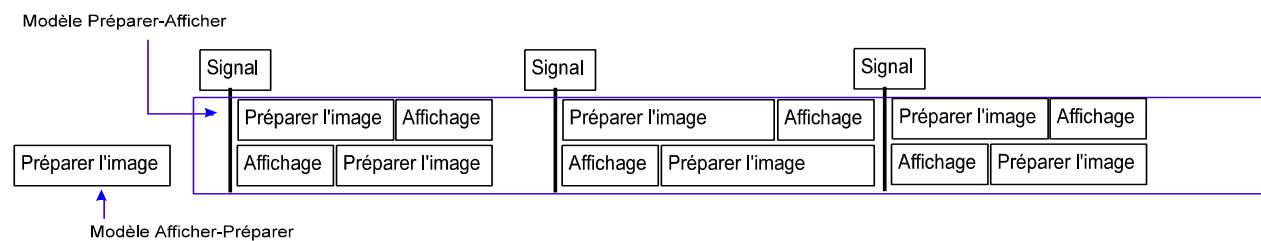
b) Un signal (horloge ou autre) est émis à intervalle précis (ex : 1/30 de sec.)

c) L'affichage a lieu (effectué par la carte vidéo – temps négligeable sur les systèmes modernes).

d) L'image suivante est préparée en vérifiant la logique, en démarrant les sons, etc. ce qui prend généralement un intervalle de temps variable et impossible à prévoir exactement.

e) On retourne en b)

L'affichage est donc toujours à intervalle régulier, à condition que les traitements reliés à la préparation soient assez rapides pour entrer dans le temps intermédiaire. Ce qui était vrai aussi pour le modèle Préparer-Afficher. Le graphique suivant illustre bien la différence entre les deux modèles.



On note que, dans le premier modèle, l'affichage est retardé en fonction de la longueur de la préparation tandis que, dans le deuxième, l'affichage a toujours lieu dès la réception du signal d'animation.

Théorie

2.10 Règles de base sur la performance des animations en 3D

Ces règles sont présentées de façon humoristique, mais si les retenez, j'aurai atteint mon objectif.

No 1. Règle la plus importante : « *The Ø second task* »

La tâche la plus rapide à exécuter est celle que l'on n'a pas à exécuter.

No 2. Design

Un bon design d'application finit toujours par être plus efficace qu'un mauvais, et ce, peu importe la plateforme, le langage et le programmeur.

No 3. Préparation

Un objet préparé à l'avance est plus rapide à traiter.

No 4. Trucs et astuces 1

On peut tricher, à condition que personne ne s'en aperçoive...

No 5. Base de programmation

Un programme qui ne fonctionne pas est toujours très lent.

No 6. Redondance 1

Ne pas refaire ce qui a déjà été fait.

No 7. Limites du monde 1 (corollaire du no 1)

Un objet invisible n'a pas à être affiché. *Discussion philosophique* : a-t-il besoin d'exister ?

No 8. Limites du monde 2 (corollaire du no 1 et suite philosophique du no 7)

Le monde n'a pas à être plus grand que ce qu'on en voit.

No 9. Redondance 2

Ne jamais calculer (ou recalculer) ce que l'on connaît déjà.

No 10. Trucs et astuces 2

De loin, on ne voit pas beaucoup les détails d'un objet. Donc, on peut tricher...

No 11. Trucs et astuces 3

De loin, on ne voit pas beaucoup les éclairages. Donc, on peut tricher...

No 12. Limites du monde 3 (corollaire du no 7)

Trop loin, un objet disparaît. Donc...

3. Premiers pas avec Direct3D

Maintenant que nous avons une application de base pour nos animations, nous pouvons commencer à y planter les éléments qui nous permettront de programmer la gestion une scène et surtout de programmer son rendu. Les premiers éléments de Direct3D y seront intégrés, nous y ferons la connaissance du dispositif 3D et des autres objets Direct3D qui seront au centre de nos préoccupations pour l'ensemble de nos travaux.

Objectifs du chapitre

- ✓ Comprendre le rôle des éléments Direct3D permettant le rendu.
- ✓ Se familiariser avec l'initialisation des éléments Direct3D.
- ✓ Se familiariser avec les classes de dispositifs de Direct3D soient les ID3D11Device et ID3D11DeviceContext.
- ✓ Créer et initialiser ces objets.
- ✓ Se donner des outils pour la gestion des erreurs.
- ✓ Mettre en place les principaux éléments de notre moteur.
- ✓ Se familiariser avec le nettoyage de la mémoire vidéo.

3. Premiers pas avec Direct3D

Nous pouvons identifier cinq éléments logiques nécessaires à une application 3D, soient:

1. Initialisation de l'environnement 3D
2. Initialisation de la scène
3. Animation de la scène
4. Rendu de la scène
5. Nettoyage des mémoires (vidéo et conventionnelle)

Initialisation de l'environnement 3D

Dans cette étape, nous initialiserons les objets DirectX nécessaires pour l'affichage d'une scène en 3D. Pour une application Direct3D, il sera nécessaire de créer et de configurer un certain nombre d'objets Direct3D soient un **dispositif de rendu** (*device*), un **contexte** (*device context* ou *immediate context*), une **chaîne d'échange** (*swap chain*) et une **vue de surface de rendu** (*render target view*).

Dans notre modèle, l'initialisation de l'environnement 3D sera effectuée par notre moteur 3D via une combinaison de fonctions génériques (dans CMoteur) et de fonctions spécifiques (dans CMoteurWindows).

Note:

J'utilise **dispositif** au lieu du terme anglais **device** parce que j'essaie (pas toujours avec succès...) d'avoir des noms de classes en français et parce que le terme **device** est déjà surutilisé.

Les objets Direct3D seront implantés une combinaison de classes génériques et spécifiques soient les classes **CDispositif** et **CDispositifDirect3D**. Je ne fais pas quatre classes pour les quatre objets pour les raisons suivantes:

- ils sont, jusqu'à un certain point, interdépendants;
- ils seront créés en même temps;
- ils seront souvent utilisés ensemble;
- leur utilisation est toujours spécifique;
- c'est plus simple.

Initialisation de la scène

Pour les besoins d'une petite application, nous n'utiliserons qu'une seule scène.

Une scène est en réalité constituée d'une ou de plusieurs « listes » d'objets 3D. Ces listes peuvent jouer plusieurs rôles: participer à la gestion des niveaux de détails, faciliter l'activation de certains paramètres, regrouper l'utilisation de certains effets, déterminer les éléments transparents, déterminer les éléments de l'interface, etc. Pour nos besoins, une ou deux listes suffiront, mais pour une « vraie » application (un jeu par exemple), il sera presque toujours nécessaire de construire une organisation de la scène plus complexe au moyen de listes, de « *containers* », de collections, d'arbres, de réseaux...

La création de l'objet scène sera pour le moment placée dans la fonction **InitScene** de CMoteur (toujours dans le fichier **Moteur.h**), où le chargement des fichiers nécessaires à l'application pourra prendre place. Dans notre cas, les structures nécessaires à la création des objets 3D seront créées et initialisées directement dans le moteur.

Dans notre modèle, la fonction InitScene sera appelée dans la fonction **Initialisations**.

Note de design

Dans une application plus complète, il serait intéressant d'avoir un gestionnaire de scènes qui permettrait le chargement et la gestion des scènes via des objets « scènes ». De plus les éléments de modélisation de la scène pourraient y prendre place.

Animation de la scène

L'animation de la scène deviendra plus tard le point le plus important de notre application 3D puisque c'est ici que les mouvements sont calculés et que la logique de l'application entre en jeu. Les liens entre l'animation de la scène, l'interface utilisateur, le module de « physique » et les besoins de l'application ne sont pas toujours faciles à définir. Le design de ces liens est souvent le défi principal d'une application 3D (dans un jeu par exemple).

Dans notre modèle, l'animation de la scène sera effectuée par **CMoteur::AnimeScene** (dans Moteur.h) et sera appelée par la fonction **Animation** comme nous l'avions entrevu dans le chapitre précédent.

Toutefois, dans l'exemple qui suivra, nous n'aurons pas encore de scène puisqu'elle sera implantée au prochain chapitre.

Rendu de la scène

Une fois que la géométrie a été mise à jour pour refléter l'animation désirée, nous pouvons effectuer le rendu de la scène.

Dans notre modèle, le rendu de la scène sera localisé dans la fonction **CMoteur::RenderScene** que l'on retrouvait dans la fonction InitAnimation et qui sera généralement appelée par la fonction CMoteur::Animation. Le « vrai » rendu sera effectué par les fonctions que nous retrouverons dans chaque classe d'objet 3D.

Le « nettoyage »

Notre application devra s'arrêter. Les objets créés dynamiquement devront bien sûr être libérés de la mémoire. L'arrêt d'une application 3D signifie aussi que nous devrons désallouer tous les objets placés sur la carte graphique utilisés par l'application et annuler les références et pointeurs qui s'y référaient.

La désaffectation des objets DirectX sera effectuée en appelant la fonction **IUnknown::Release** pour chaque objet. Puisque nous suivons les règles des objets COM, le compteur de référence pour la plupart des objets devrait devenir zéro et ces objets sont donc automatiquement retirés de la mémoire (graphique et/ou conventionnelle).

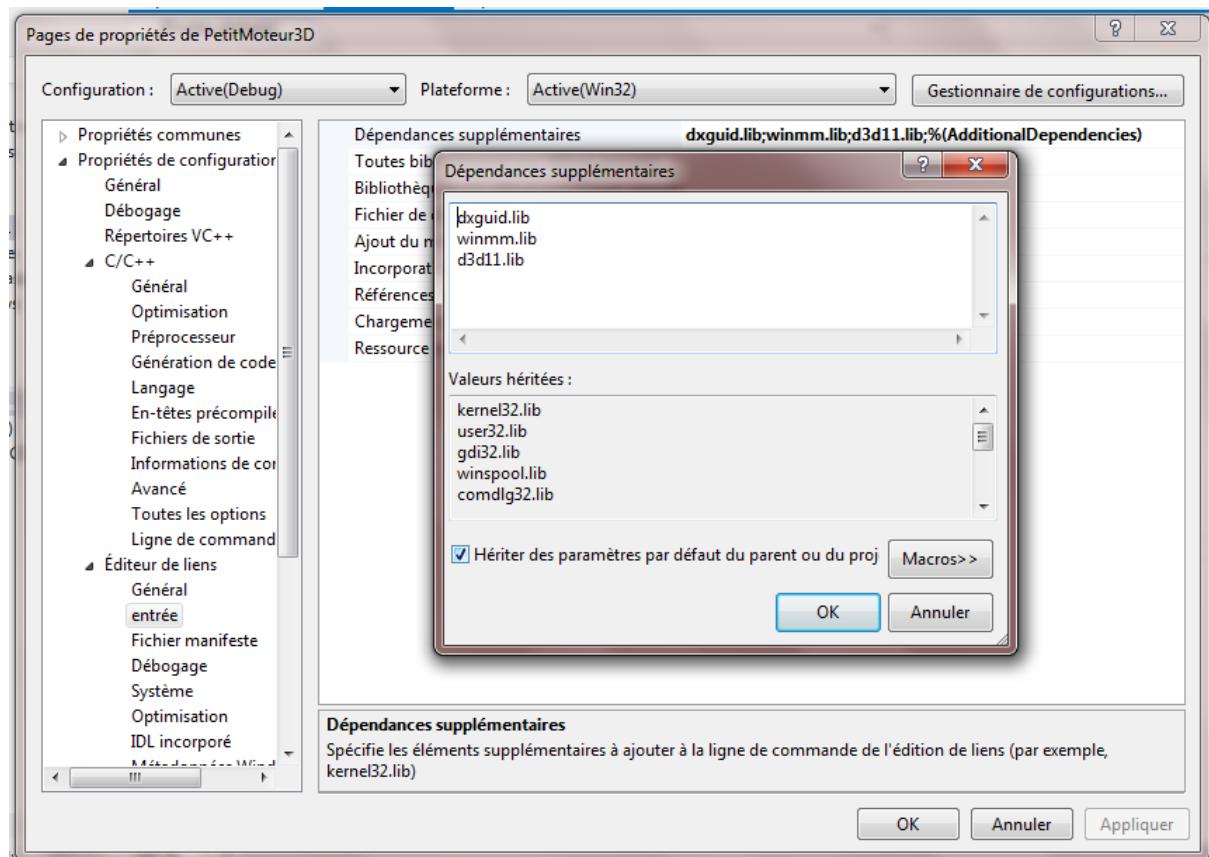
En plus de l'arrêt du programme, il existe des situations lors de l'exécution normale, par exemple lorsque l'utilisateur change la résolution ou le nombre de couleurs du bureau, où nous serions dans l'obligation de devoir détruire et recréer les objets DirectX en service. En conséquence, il est pratique d'avoir une fonction spécialisée pour le code de nettoyage de votre application, fonction qui peut s'appeler quand le besoin s'en fait sentir.

Dans notre modèle, le nettoyage s'appellera **CMoteur::Cleanup** et sera appelé par le destructeur de la classe CMoteur mais pourrait aussi être appelé en cas de problème.

3.1 Préparation de l'application

Nous devons d'abord préparer notre modèle d'application. Notre résultat sera une application 3D **fenêtrée**.

1. Prendre une copie du résultat du chapitre 2.
2. Dans le dialogue **Pages de propriétés** de PetitMoteur3D sous la rubrique **Éditeur de liens|Entrée**, ajouter la bibliothèque **d3d11.lib** aux bibliothèques dxguid.lib dans le champ **Dépendances supplémentaires**.



3. Ajouter les lignes suivantes dans le fichier **stdafx.h**, à la fin, si elles n'y sont pas déjà.

```
#define _XM_NO_INTRINSICS_
#include <d3d11.h>
#include <DirectXMath.h>
```

_XM_NO_INTRINSICS_ spécifie qu'aucun type intrinsèque ne sera utilisé et que la bibliothèque DirectX Math n'utilisera pas de types ou d'alignements spéciaux. Voir **DirectX Math Library Compiler Directives** dans la documentation du SDK.

Vous devriez compiler votre programme dès maintenant pour voir s'il ne vous manque rien. Attention, le programme ne fait rien pour l'instant.

Les fichiers stdafx.cpp et stdafx.h

Si vous n'êtes pas familier avec le développement d'applications avec Visual Studio et C++, les fichiers **stdafx.h** et **stdafx.cpp** peuvent vous sembler étranges. Il s'agit en fait d'un bloc de code (surtout des en-têtes) qui sera précompilé et conservé dans un fichier de travail. Le code précompilé est utile pendant le cycle de développement car il permet de réduire le temps de compilation, notamment si:

- Vous utilisez toujours un corps de code volumineux qui change peu fréquemment.
- Votre programme comprend plusieurs modules, qui utilisent tous un jeu standard de fichiers *include* et les mêmes options de compilation. Dans ce cas, tous les fichiers *include* peuvent être précompilés en un seul fichier en-tête précompilé.

La première compilation (celle qui crée le fichier d'en-tête précompilé) prend un peu plus de temps que les compilations suivantes. Les compilations ultérieures s'exécutent plus rapidement grâce au code précompilé inclus.

Voir: [MSDN] Création de fichiers d'en-tête précompilés.

Comme vous le savez sûrement... en C++ un fichier en-tête (*header file*) est un fichier source dont le texte est habituellement inclus dans d'autres fichiers sources (.H ou .CPP) par le précompilateur C++ au moyen de la directive **#include** ou de l'équivalent.

Dans le développement d'applications nécessitant plusieurs liens avec de « grosses bibliothèques », Win32 et DirectX par exemple, les lignes de codes inclus peuvent se compter par plusieurs dizaines de milliers.

Avec Visual Studio (en C++) et dans plusieurs autres environnements de travail, il est possible de réduire les temps de compilation en permettant à un certain nombre de ces fichiers en-têtes d'être précompilés. Le fichier résultant s'appelle un en-tête précompilé. Avec Visual Studio, il est stocké dans un fichier portant l'extension **.pch**, lui-même stocké dans un des dossiers **debug**.

3.2 Gestion des erreurs DirectX

Erreurs et débogage de l'application

Le SDK de DirectX nous offre deux fonctions pour faciliter le débogage des applications utilisant DirectX soient les fonctions **DXTrace** et **DXRrrorString**. Pour faciliter l'usage de DXTrace, trois macros pour utilisation en mode *Debug* sont aussi définies dans le SDK soient:

- **DXTRACE_ERR** qui effectue la sortie d'un message de débogage ainsi que du code d'erreur qui lui est associé dans la console de sortie de Visual Studio.
- **DXTRACE_ERR_MSGBOX** qui affiche une boîte de message (**MessageBox**) contenant l'information sur l'erreur.
- Attention! Lors de l'exécution de notre programme en mode plein écran, il est fort probable que le message ne sera pas visible.
- **DXTRACE_MSG** qui effectue la sortie d'un message de débogage dans la console de sortie de Visual Studio Outputs.

Erreurs d'environnement et erreurs d'installation

Les erreurs les plus importantes à traiter sont celles qui sont indépendantes de notre application. Deux exemples:

- Est-ce que la bonne version de DirectX est installée?
- Est-ce qu'un fichier de ressource (une texture par exemple) aurait disparu?

En effet, la plupart des fonctions de DirectX nous retourneront un code d'erreur nous permettant d'identifier assez facilement le problème. Ces erreurs doivent être traitées correctement lors de l'initialisation. Dans notre cas, un message d'erreur « technique » sera affiché puisque nous sommes en développement mais la version finale de votre application devra donner à l'utilisateur un message plus significatif et lui permettre de corriger la situation si possible.

Sauf exception, cette vérification des codes d'erreurs rentrés ne devrait pas faire partie de la logique d'animation puisque:

- Nous voulons de la vitesse.
- Les principales validations **devraient** avoir été faites lors de l'initialisation.

Une ou deux exceptions à cette règle se présenteront en cours de développement (par exemple lorsque le dispositif devient invalide!) et vous aurez évidemment la possibilité d'examiner les codes de retour pour fins de débogage.

Atelier

3.3 Fonctions de gestion des erreurs et des pointeurs COM

Pour faciliter l'utilisation des instructions DirectX pour lesquelles nous devons valider le code de retour ainsi que pour faciliter l'utilisation de certains types de pointeurs (par exemple pour les objets COM), nous implémenterons quelques petites fonctions (certaines sous forme de *template*) dans un petit fichier que l'on appellera **util.h**. Ces fonctions ont pour but d'alléger les codes d'initialisation et de nettoyage.

- Créer le fichier **util.h** et lui ajouter les lignes suivantes:

La fonction **DXEssayer** a pour but d'envoyer une exception si le code de retour de la fonction n'est pas S_OK, il serait possible de vérifier à quoi correspond le code d'erreur (nous le ferons plus tard), mais pour les cas d'initialisation, il est surtout utile de connaître l'instruction ayant échouée.

La fonction **DXValider** a pour but de vérifier si le pointeur pointe sur un objet. Généralement les fonctions DirectX qui retournent un pointeur renverront la valeur NULL s'il y a un problème. Mais il est tout de même recommandé de les initialiser à NULL. Cette fonction nous servira surtout pour les objets construits par des appels à des fonctions DirectX, mais nous l'utiliserons aussi pour d'autres pointeurs.

La fonction **DXRelacher** a pour but d'effectuer l'appel à la fonction IUnknown::Release pour les objets DirectX (qui sont aussi des objets COM) tout en validant le pointeur et en le réinitialisant à NULL.

```
// Essayer en envoyant le code d'erreur comme résultat
// Il ne faut pas oublier de "rattraper" le code...
template <class Type>
inline void DXEssayer(const Type& Resultat)
{
    if (Resultat != S_OK)
    {
        throw Resultat;
    }
}

// Plus pratique, essayer en envoyant un code spécifique
// comme résultat
template <class Type1, class Type2>
inline void DXEssayer(const Type1& Resultat, const Type2& unCode)
{
    if (Resultat != S_OK)
    {
        throw unCode;
    }
}

// Valider un pointeur
template <class Type>
inline void DXValider(const void* UnPointeur, const Type& unCode)
{
    if (UnPointeur == nullptr)
    {
        throw unCode;
    }
}

// Relâcher un objet COM (un objet DirectX dans notre cas)
template <class Type>
inline void DXRelacher(Type& UnPointeur)
{
    if (UnPointeur != nullptr)
    {
        UnPointeur->Release();
        UnPointeur = nullptr;
    }
}
```

2. Ajoutez un nouveau « catch » dans la fonction WinMain pour permettre une sortie rapide et un affichage des messages d'erreur:

```
catch (int codeErreur)
{
    wchar_t szErrMsg[MAX_LOADSTRING];// Un message d'erreur selon le code
    ::LoadString(hInstance, codeErreur, szErrMsg, MAX_LOADSTRING);
    ::MessageBox(nullptr, szErrMsg, L"Erreur", MB_ICONWARNING);

    return (int)99; // POURQUOI 99???
}
```

Les messages d'erreurs correspondant à ces exceptions devront être définis dans la « **string table** » de votre application Windows, nous y reviendrons quand l'occasion se présentera.

Atelier

3.4 Classes de dispositifs

Après avoir créé la fenêtre de l'application, nous pouvons commencer à initialiser les principaux objets DirectX. Pour une application Direct3D minimale, Nous aurons besoin pour nos opérations en 3D d'un objet représentant le dispositif de rendu et d'un objet représentant le contexte d'affichage. Toujours en conservant notre architecture d'application « simple », nous placerons les éléments génériques du dispositif dans la classe CDispositif et les éléments et fonctionnalités spécifiques à Direct3D 11 dans la classe CDispositifD3D11.

1. Ajoutez à votre projet les fichiers suivants:

Dispositif.h

```
#pragma once

namespace PM3D
{
// Constantes pour mode fenêtré ou plein écran
enum CDS_MODE
{
    CDS_FENETRE,
    CDS_PLEIN_ECRAN
};

// Classe : CDispositif
//
// BUT : Classe servant à construire un objet Dispositif
//       qui implantera les aspects "génériques" du dispositif de
//       rendu
//
class CDispositif
{
public:
    virtual ~CDispositif() = default;

    virtual void Present();
    virtual void PresentSpecific() = 0;
};

} // namespace PM3D
```

DispositifD3D11.h

```
#pragma once

#include "dispositif.h"

namespace PM3D
{
    //
    // Classe CDispositifD3D11
    //
    // BUT : Classe permettant d'implanter un dispositif de rendu
    //        Direct3D
    //
    class CDispositifD3D11 : public CDispositif
    {
    public:
        CDispositifD3D11(const CDS_MODE cdsMode, const HWND hWnd);
        virtual ~CDispositifD3D11();
    };
}
```

DispositifD3D11.cpp

```
#include "StdAfx.h"
#include "resource.h"
#include "DispositifD3D11.h"
#include "Util.h"

namespace PM3D
{
    // FONCTION : CDispositifD3D11, constructeur paramétré
    // BUT : Constructeur de notre classe spécifique de dispositif
    // PARAMÈTRES:
    //     cdsMode:CDS_FENETRE application fenêtrée
    //             CDS_PLEIN_ECRAN application plein écran
    //
    //     hWnd: Handle sur la fenêtre Windows de l'application,
    //           nécessaire pour la fonction de création du
    //           dispositif
    CDispositifD3D11::CDispositifD3D11(const CDS_MODE cdsMode,
                                       const HWND hWnd)
    {
        switch (cdsMode)
        {
        case CDS_FENETRE:
            break;
        }
    }
}
```

2. Ajoutez `#include "dispositif.h"` à la suite des autres `#include` dans le fichier Moteur.h
3. Ajoutez `#include "dispositifD3D11.h"` à la suite des autres `#include` dans le fichier MoteurWindows.h

Modification de CMoteur::Initialisations

Dans notre exercice, les initialisations sont effectuées dans la fonction CMoteur::Initialisations et l'initialisation de l'environnement 3D est effectuée au besoin dans le dispositif 3D.

1. Déclarez la variable **pDispositif** comme variable protégée (*protected*) dans la classe CMoteur:

```
// Le dispositif de rendu  
TClasseDispositif* pDispositif;
```



D'où vient **TClasseDispositif**? Pour simplifier l'utilisation d'un dispositif par des classes de travail, il est plus agréable (et probablement plus rapide) d'utiliser réellement un pointeur sur un objet CDispositifD3D11 plutôt que de passer par une conversion dynamique lors de l'exécution. Comme la classe CMoteur est un *template*, il est facile d'y insérer des éléments qui ne seront connus qu'à la compilation.

2. Donc, modifiez la déclaration de la classe CMoteur pour qu'elle ressemble maintenant à ceci:

```
template <class T, class TClasseDispositif> class CMoteur : public  
CSingleton<T>
```

3. Modifiez ensuite la déclaration de la classe CMoteurWindows pour qu'elle ressemble à ceci:

```
class CMoteurWindows : public CMoteur<CMoteurWindows, CDispositifD3D11>
```

4. Ajoutez un appel à la fonction **CreationDispositifSpecific** dans la fonction **CMoteur::Initialisations** (dans Moteur.h) et retirez les commentaires devant l'appel à InitAnimation. Nous aurons maintenant:

```
virtual int Initialisations()
{
    // Propre à la plateforme
    InitialisationsSpecific();

    // * Initialisation du dispositif de rendu
    pDispositif = CreationDispositifSpecific( CDS_FENETRE );

    // * Initialisation de la scène

    // * Initialisation des paramètres de l'animation et
    // préparation de la première image

    InitAnimation();

    return 0;
}
```

La création d'un dispositif sera **spécifique** dans notre petite architecture. Notez l'utilisation en paramètre de la constante CDS_FENETRE qui indique que nous voulons un dispositif fonctionnant dans une fenêtre et qui utilisera les valeurs actives de la carte graphique. Nous modifierons cette fonction pour la rendre plus polyvalente au chapitre 4 (et nous ajouterons le cas CDS_PLEIN_ECRAN).

5. Déclarez la fonction **CMoteur::CreationDispositifSpecific** dans la classe CMoteur (Moteur.h), comme toutes nos fonctions spécifiques, elle sera une fonction abstraite dans CMoteur:

```
virtual TClasseDispositif* CreationDispositifSpecific(const CDS_MODE
cdsMode)=0;
```

6. Déclarez la fonction **CMoteurWindows::CreationDispositif-Specific** dans la classe CMoteurWindows (MoteurWindows.h):

```
virtual CDispositifD3D11* CreationDispositifSpecific(const CDS_MODE cdsMode);
```

7. Écrire la fonction **CMoteurWindows::CreationDispositifSpecific** dans la classe CMoteurWindows (MoteurWindows.cpp):

```
// FONCTION : CreationDispositifSpecific
//
// BUT : Fonction responsable de créer le
//       dispositif selon certains paramètres
// NOTES:
//       CDS_MODE: CDS_FENETRE      application fenêtrée
//                  CDS_PLEIN_ECRAN application plein écran
//
CDispositifD3D11* CMoteurWindows::CreationDispositifSpecific(const CDS_MODE
cdsMode)
{
    return new CDispositifD3D11(cdsMode, hMainWnd);
}
```

Notez que le dispositif est un objet correspondant à notre classe CDispositifD3D11, qui est spécifique à notre plateforme et à notre choix de bibliothèque de développement.

3.5 Les objets Direct3D

La chaîne d'échange (*swap chain*)

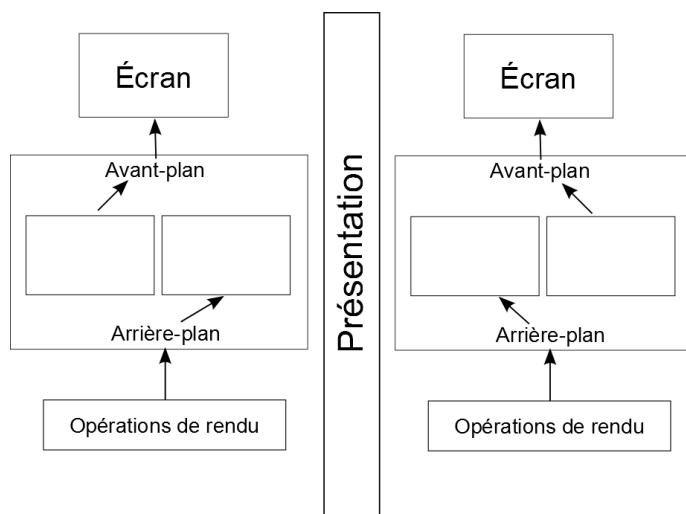
Pour éviter les problèmes d'affichage lors d'une animation (comme le *flickering*), il est recommandé de dessiner une image de notre animation (*frame*) sur une surface différente de celle utilisée pour l'écran. Cette surface, une texture, est appelée tampon d'arrière-plan (*back buffer*). Une fois notre scène dessinée sur ce tampon, il est présenté à l'écran d'un seul coup (en un seul cycle de rafraîchissement d'écran). Ainsi, l'utilisateur ne voit que des images complètes, il ne voit pas le processus de création de l'image.

Pour réaliser ceci, deux tampons (ou plus) sont utilisés par la carte graphique. Celui utilisé pour le rafraîchissement d'écran s'appelle le **tampon d'avant-plan** (*front buffer*), il est en lecture seulement donc ne peut être modifié. Celui utilisé pour le rendu s'appelle le **tampon d'arrière-plan** (*back buffer*). Une fois la scène affichée sur le tampon d'arrière-plan, les rôles sont inversés (*swapped* en anglais). Cet échange de rôle s'appelle **présentation**. D'où notre choix du terme **Present** pour la fonction qui effectuera cet échange. Il s'agit d'une opération extrêmement rapide puisque seul les pointeurs sur les tampons sont échangés.

Ces deux tampons forment ce que l'on appellera une **chaîne d'échange** (*swap chain*). En DirectX, la chaîne d'échange sera implantée dans l'interface **IDXGISwapChain**. Cette interface s'occupe de la gestion des tampons (stockage et méthodes) ainsi que de la présentation (**IDXGISwapChain::Present**).

L'utilisation de deux tampons (*front buffer* et *back buffer*) s'appelle *double buffering*. L'utilisation de trois tampons s'appelle *triple buffering*. Deux tampons sont généralement suffisants, mais...

La figure suivante illustre ce processus.



« FLICKERING »

Il s'agit d'un effet d'affichage déplaisant pour l'oeil, causé par des délais dans l'affichage d'une animation. Ce problème apparaît lorsque l'utilisateur « voit » les opérations réalisées dans une animation, par exemple l'effacement de l'écran avant de redessiner celui-ci.

Le dispositif (*device*)

Le dispositif est utilisé pour créer des ressources sur la carte graphique ou énumérer les caractéristiques de celle-ci. Les ressources sont des « objets » comme des objets 3D, des shaders, des textures, des tampons et bien d'autres.

Dans les versions, précédentes de DirectX le dispositif (*device*) était utilisé pour le rendu et pour la gestion des ressources.

Direct3D 11 utilise un « **contexte immédiat** » pour effectuer les opérations de rendu et le dispositif continue de s'occuper de la gestion des ressources.

Un dispositif est implanté au moyen d'un objet de classe **ID3D11Device**. En réalité, il sera créé au moyen de la fonction **D3D11CreateDevice**. Une fois que nous aurons créé le dispositif, nous pourrons créer la chaîne d'échange et accéder à la vue de surface de rendu. Le contexte, lui, est créé en même temps que le dispositif au moyen de la même fonction.

Comme nous désirons presque toujours une chaîne d'échange, une fonction, **D3D11CreateDeviceAndSwapChain**, permet de créer le dispositif, le contexte et la chaîne d'échange en un seul appel de fonction.

Chaque dispositif peut utiliser un ou plusieurs contexte. Habituellement, une application n'utilise qu'un seul dispositif et un seul contexte, mais il est possible d'avoir plusieurs dispositifs.

Le contexte (*device context*)

Le contexte est l'outil de contrôle du rendu de Direct3D. Il nous permet de manipuler les états des divers éléments du processus de rendu. Il est aussi responsable des opérations permettant d'effectuer le rendu dont les fonctions **Draw** et **DrawIndexed** qui seront parmi celles que nous utiliserons le plus souvent.

Un dispositif est implanté au moyen d'un objet de classe **ID3D11DeviceContext**.

Direct3D 11 nous offre deux types de contexte: un **contexte immédiat** pour effectuer un rendu directement via le pilote de périphérique et un **contexte « reporté » (deferred)** pour un rendu « théorique » qui pourra être exécuté plus tard (par exemple par un autre thread). Les contextes reporté sont une tentative de soutenir le rendu multithreadé. Cependant, D3D11 n'étant pas conçu pour le multithreading, les gains apportés par les contextes deférés sont négligeables.

Chaque dispositif n'a qu'**un seul** contexte immédiat. On pourrait par contre avoir plusieurs contextes reportés.

Ici, le terme « vue » a presque le même sens que **vue** dans le design pattern MVC.
(pour ceux qui con-

Vue de surface de rendu (*render target view*)

Dans les versions « pré DirectX 10 » de Direct3D, la **surface de rendu** (généralement le *backbuffer*) était définie par défaut dans le dispositif. À moins de vouloir effectuer des opérations spéciales, par exemple des « *post-effects* », la définition de défaut nous convenait.

Les GPUs offrent maintenant tellement de possibilités qu'une surface de défaut « cachée » serait trop limitative. Nous avons donc la possibilité voire l'obligation d'y accéder au moyen d'une **vue de ressource**. **Ressource** est le terme général pour un bloc mémoire réservé pour la carte graphique, généralement en mémoire graphique. Les ressources sont surtout des informations de **textures**, de **tampons** (buffers) et de **shaders** mais nous pourrions les utiliser pour autre chose.

La fonction **D3D11CreateDeviceAndSwapChain** a déjà créé le tampon d'arrière plan (la surface de rendu), mais certaines opérations nous demanderons d'y accéder directement.

La classe **ID3D11RenderTargetView** nous permet de définir une interface sur la surface de rendu. Nous pourrons donc y effectuer directement certaines opérations.

Vue de surface de rendu

J'utilise le terme **surface de rendu** plutôt que **cible de rendu** (*render target*) parce qu'il s'agit d'une texture et que le terme surface (*draw surface*) est utilisé dans la plupart des références génériques sur l'infographie. Même dans la documentation de Direct3D, l'utilisation de texture plutôt que surface est très récent.

Atelier

3.6 Crédit des objets Direct3D

Il nous faut maintenant créer les quatre objets Direct3D dont nous avons parlé plus tôt: un **dispositif de rendu** (*device*), un **contexte** (*immediate context*), une **chaîne de tampons d'affichage** (*swap chain*) et une **vue de surface de rendu** (*render target view*).

1. Dans la section *private* de la classe `CDispositifD3D11` (`dispositifD3D11.h`), déclarez quatre pointeurs pour nos quatre objets :

```
ID3D11Device* pD3DDevice;
ID3D11DeviceContext* pImmediateContext;
IDXGISwapChain* pSwapChain;
ID3D11RenderTargetView* pRenderTargetView;
```

2. Pour en simplifier l'accès, déclarez immédiatement quatre fonctions d'accès dans la section *public* de la classe `CDispositifD3D11`:

```
// Fonction d'accès aux membres protégés
ID3D11Device* GetD3DDevice() { return pD3DDevice; }
ID3D11DeviceContext* GetImmediateContext() { return pImmediateContext; }
IDXGISwapChain* GetSwapChain() { return pSwapChain; }
ID3D11RenderTargetView* GetRenderTargetView(){return pRenderTargetView; }
```

3. Puis nous modifions le constructeur de `CDispositifD3D11` pour qu'il nous crée nos quatres objets. Beaucoup d'éléments sont présentés ici et seront repris plus spécifiquement au chapitre 5, mais j'essaierai quand même d'en expliquer le maximum dès maintenant.

```
CDispositifD3D11::CDispositifD3D11(const CDS_MODE cdsMode,
                                    const HWND hWnd)
{
    int largeur;
    int hauteur;
    UINT createDeviceFlags = 0;

    #ifdef _DEBUG
        createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
    #endif

    D3D_FEATURE_LEVEL featureLevels[] =
    {
        D3D_FEATURE_LEVEL_11_1,
        D3D_FEATURE_LEVEL_11_0,
    };
    UINT numFeatureLevels = ARRAYSIZE( featureLevels );

    pImmediateContext = nullptr;
    pSwapChain = nullptr;
    pRenderTargetView = nullptr;

    DXGI_SWAP_CHAIN_DESC sd;
```

```

ZeroMemory( &sd, sizeof(sd) );

switch (cdsMode)
{
case CDS_FENETRE:
    RECT rc;
    GetClientRect( hWnd, &rc );
    largeur = rc.right - rc.left;
    hauteur = rc.bottom - rc.top;

    sd.BufferCount = 1;                                ← 4
    sd.BufferDesc.Width = largeur;
    sd.BufferDesc.Height = hauteur;
    sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    sd.BufferDesc.RefreshRate.Numerator = 60;
    sd.BufferDesc.RefreshRate.Denominator = 1;
    sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
    sd.OutputWindow = hWnd;
    sd.SampleDesc.Count = 1;
    sd.SampleDesc.Quality = 0;
    sd.Windowed = TRUE;

    break;
}

// Création dispositif, chaîne d'échange et contexte
DXEssayer( D3D11CreateDeviceAndSwapChain( 0,
                                             D3D_DRIVER_TYPE_HARDWARE,
                                             NULL,
                                             createDeviceFlags,
                                             featureLevels, numFeatureLevels,
                                             D3D11_SDK_VERSION,
                                             &sd,
                                             &pSwapChain,
                                             &pD3DDevice,
                                             NULL,
                                             &pImmediateContext ),
            DXE_ERREURCREATIONDEVICE ) ;                      ← 1

// Création d'un « render target view »
ID3D11Texture2D *pBackBuffer;
DXEssayer( pSwapChain->GetBuffer( 0,
                                    __uuidof( ID3D11Texture2D ),
                                    (LPVOID*)&pBackBuffer ),
            DXE_ERREUROBTENTIONBUFFER ) ;

DXEssayer( pD3DDevice->CreateRenderTargetView( pBackBuffer,
                                                NULL,
                                                &pRenderTargetView ),           ← 5
            DXE_ERREURCREATIONRENDERTARGET);

pBackBuffer->Release();

pImmediateContext->OMSetRenderTargets( 1, &pRenderTargetView, NULL ); ← 6

D3D11_VIEWPORT vp;
vp.Width = (FLOAT)largeur;
vp.Height = (FLOAT)hauteur;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
pImmediateContext->RSSetViewports( 1, &vp );          ← 7
}

```

1

Explications

Au cœur de nos initialisations se trouve la fonction **D3D11CreateDeviceAndSwapChain** qui va créer nos trois objets de base soient le **dispositif** (*device*), le **contexte** (*immediate context*), la **chaîne d'échange** (*swap chain*). Notre quatrième objet sera une vue sur le tampon d'arrière plan (créé dans la chaîne d'échange).

IMPORTANT: regardez la description de cette fonction dans la documentation de DirectX Graphics avant de poursuivre.

Paramètre 1 - « Adaptateur» (entrée)

Nous utilisons 0 pour désigner l'adaptateur de défaut donc celui qui est présentement en utilisation. C'est le choix à faire lorsque votre application est fenêtrée. L'adaptateur (*graphic adapter*) est en général la carte graphique. Mais certaines cartes graphiques offrent deux adaptateurs (ou plus) et certains ordinateurs ont deux cartes graphiques (ou plus). L'adaptateur de défaut n'est pas nécessairement le plus rapide.

Le choix d'un adaptateur n'est pas toujours évident à gérer sur les nouveaux systèmes (particulièrement lorsqu'il y a « collaboration » entre les divers adaptateurs présents. Nous y reviendrons au chapitre 5 .

Paramètre 2 – Type de pilote (entrée)

Nous spécifions ici le type de pilote que notre dispositif utilisera. Le choix idéal pour des applications 3D de haute performance sera **D3D_DRIVER_TYPE_HARDWARE**. Mais trois autres possibilités existent, chacune offrant des avantages parfois non négligeables.

Paramètre 3 – Logiciel (entrée)

Ce paramètre est toujours NULL (0) sauf si le type de pilote est **D3D_DRIVER_TYPE_SOFTWARE**.

2

Paramètre 4 – Fanions de création (entrée)

Normalement, nous utiliserons les options de défaut en initialisant une variable à 0. Dans nos cas, nous ajoutons la possibilité d'avoir aussi le fanion **D3D11_CREATE_DEVICE_DEBUG** si nous sommes en mode *debug*. L'option debug nous permet d'avoir les avertissements de DirectX dans le fenêtre de sortie de Visual Studio. Ces avertissements sont très utiles

3

Paramètres 5 et 6 – Niveau des caractéristiques souhaitées (entrée)

Le paramètre 5 est un pointeur sur un tableau de « niveaux de caractéristiques ». Le paramètre 6 indiquent combien de niveaux sont présents dans la liste.

Les niveaux de caractéristiques sont identifiés par la version de DirectX ainsi qu'un « numéro de *release* ». Il est possible d'en choisir un seul, par exemple D3D_FEATURE_LEVEL_11_0, mais j'ai préféré utiliser une liste. Les niveaux préférés devraient être en premier dans la liste.

Paramètre 7 – Numéro de version du SDK (entrée)

Ce paramètre, D3D11_SDK_VERSION, informe notre fonction de création de la version du SDK utilisée en développement. Nous informons ainsi Direct3D que nous désirons un objet correspondant à notre version du SDK. Si cet objet n'est pas disponible, la fonction échouera.

4

Paramètre 8 – Description de la chaîne d'échange (entrée)

La création de la chaîne d'échange nous demande un certains nombre de paramètres que nous inscrirons dans une structure de donnée de type DXGI_SWAPCHAIN_DESC. Plusieurs des champs de cette structure s'expliquent par eux même (vous pouvez tout de même consulter le SDK pour plus de détails). Mais notez tout de même le **format de pixel (BufferDesc.Format)** soit DXGI_FORMAT_R8G8B8A8_UNORM, l'utilisation du tampon d'arrière plan (**BackBufferUsage**) le champ **OutputWindow** qui désigne la fenêtre d'accueil et le champ **Windowed** (valeur **TRUE**);

Paramètre 9 – Pointeur sur la chaîne d'échange (sortie)

Ce paramètre est l'adresse d'un pointeur qui recevra l'adresse de l'objet **IDXGISwapChain** représentant la chaîne d'échange.

Paramètre 10 – Pointeur sur le dispositif (sortie)

Ce paramètre est l'adresse d'un pointeur qui recevra l'adresse de l'objet **ID3D11Device** représentant le dispositif.

Paramètre 11 – Pointeur sur une liste de caractéristiques (sortie)

Ce paramètre est l'adresse d'un pointeur qui recevra l'adresse d'un tableau contenant les niveaux de caractéristiques supportés par notre dispositif. Pour ne moment, cette information ne nous intéresse pas alors nous utilisons NULL (0).

Paramètre 12 – Pointeur sur le contexte (sortie)

Ce paramètre est l'adresse d'un pointeur qui recevra l'adresse de l'objet **ID3D11DeviceContext** représentant le contexte immédiat.

5

Nous devons par la suite créer une **vue de surface de rendu** (*render target view*) parce que nous voulons associer le tampon d'arrière-plan de notre chaîne d'échange à une surface de rendu.

Le tampon d'arrière-plan est **déjà** une surface de rendu. Nous voulons pouvoir y accéder via un objet de classe **ID3D11RenderTargetView** afin de faciliter certaines opérations, entre autres pour l'effacer ou pour le désigner comme cible pour le rendu. Ne pas oublier que nous pourrions avoir plusieurs cibles de rendu.

D'abord, nous obtenons un pointeur sur le tampon d'arrière-plan via la fonction **IDXGISwapChain::GetBuffer** puis nous utilisons la fonction **CreateRenderTargetView** de **ID3D11Device** pour créer la vue. Notez que le deuxième paramètre est à NULL, donc que nous utilisons la vue de surface de rendu de défaut. Nous aurions pu remplir une structure **D3D11_RENDERTARGETVIEW_DESC** pour raffiner les détails de notre vue.

6

Une fois notre vue créée, nous utilisons la fonction **OMSetRenderTargetTargets** de **ID3D11DeviceContext** pour associer notre vue au contexte immédiat. Nous désignons ainsi cette vue comme cible pour le rendu.

7

Une dernière chose avant que nous puissions effectuer un rendu soit l'initialisation du *viewport*. Si vous vous rappelez, avant le traitement des pixels, les coordonnées de notre « écran virtuel » étaient de -1 à 1 pour les X et les Y et de 0 à 1 pour les Z. Dans les versions « pré DX 10 », ces coordonnées étaient associés à la largeur et à la hauteur de la fenêtre automatiquement, ce qui pouvait causer des problèmes (rarement, mais quand même...). Avec DX 11, le *viewport* n'est pas initialisé par défaut. Il faut donc le faire nous même, cette méthode est plus souple et nous permet plus de latitude sur les **coordonnées de pixels**.

Problème de français

Viewport

Le *viewport* avec DirectX désigne principalement la zone d'affichage (de rendu) soit l'intérieur d'une fenêtre ou même l'écran au complet. Je conserve le terme *viewport* en attendant mieux...

IMPORTANT

Ne testez pas votre programme tout de suite, il manque certains éléments que nous ajouterons dans la section suivante -> La suite.

3.7 Crédation des objets Direct3D (suite)

Dans les pages précédentes, nous avons relativement bien décrit la construction des objets D3D nécessaires à notre première application, mais nous n'avons pas tout complété.

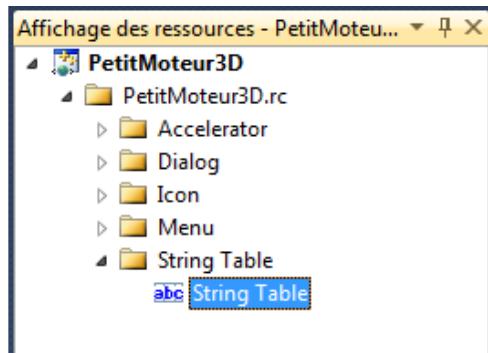
En effet lors de l'utilisation des fonctions de création, je les ai placées à l'intérieur d'une de nos fonctions de traitement des erreurs soit la fonction **DXEssayer**, par exemple:

```
// Crédation dispositif, chaine d'échange et contexte
DXEssayer(
    D3D11CreateDeviceAndSwapChain( 0,
        D3D_DRIVER_TYPE_HARDWARE,
        NULL,
        createDeviceFlags,
        featureLevels, numFeatureLevels,
        D3D11_SDK_VERSION,
        &sd,
        &pSwapChain,
        &pD3DDevice,
        NULL,
        &pImmediateContext ),
    DXE_ERREURCREATIONDEVICE ) ;
```

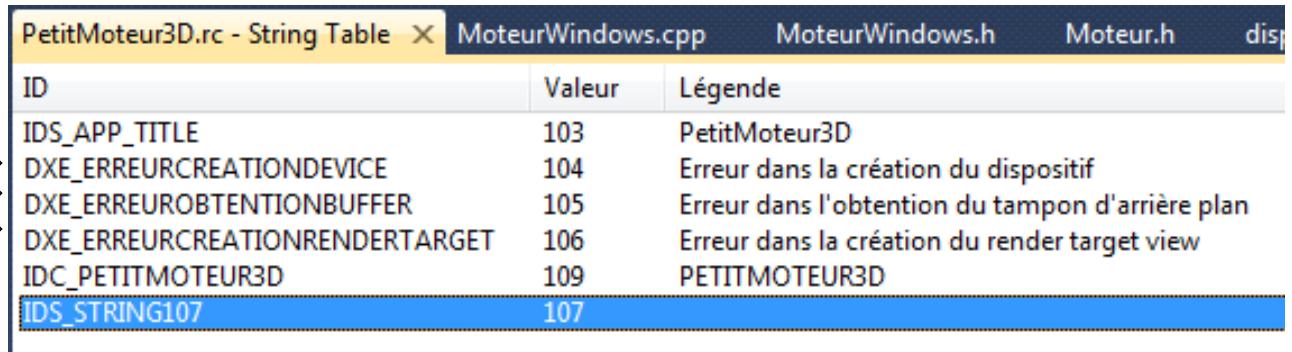
La fonction **DXEssayer** vérifie la valeur de retour de la fonction **D3D11CreateDeviceAndSwapChain** et effectue une « sortie en catastrophe » (avec un énoncé **throw**) si le code de retour n'est pas **S_OK**. Nous pourrions vérifier ce code de retour pour des raisons techniques. Mais pour des applications simples, il est presque toujours suffisant de savoir **quelle** fonction a échouée. L'erreur étant presque toujours soit une version du module *runtime* inadéquate soit, dans notre cas, des erreurs dans les paramètres de création.

Nous allons donc déclarer dans les ressources Windows de notre application, les messages d'erreurs utilisés dans le constructeur de **CDispositifD3D11**.

1. Dans la fenêtre Affichage des ressources, choisir **String Table** (double-clic)



2. La fenêtre d'édition des chaînes de caractères apparaît, elle nous permet d'y associer un identificateur de ressource à une chaîne. Ajoutez-y les trois chaînes utilisées dans le constructeur, comme suit:



ID	Valeur	Légende
IDS_APP_TITLE	103	PetitMoteur3D
DXE_ERREURCREATIONDEVICE	104	Erreur dans la création du dispositif
DXE_ERREUROBTENTIONBUFFER	105	Erreur dans l'obtention du tampon d'arrière plan
DXE_ERREURCREATIONRENDERTARGET	106	Erreur dans la création du render target view
IDC_PETITMOTEUR3D	109	PETITMOTEUR3D
IDS_STRING107	107	

Les valeurs (103, 104, etc) n'ont pas d'importance.

Nos initialisations sont terminées (pour l'instant).

3.8 La fonction d'animation

L'animation est une fonction très importante puisqu'elle est effectuée pour chaque image générée en sortie. Dans une application 3D plus complète que la nôtre, on y retrouvera entre autres:

- la vérification de l'état du clavier
- la vérification de l'état des autres périphériques d'entrée
- les changements d'états de l'application
- la logique associée à l'application

et

- les mouvements des objets graphiques en fonction de tout ça.
-

1. Modifier la fonction CMoteur::Animation. On en profite pour enlever les commentaires (//) devant l'appel de cette fonction ainsi que devant l'appel de la fonction **Present**.

```

...
// Affichage optimisé
pDispositif->Present(); // On enlevera «//» plus tard

// On prépare la prochaine image
// AnimeScene(TempsEcoule);

// On rend l'image sur la surface de travail
// (tampon d'arrière plan)
RenderScene();

// Calcul du temps du prochain affichage
TempsCompteurPrecedent = TempsCompteurCourant;
...

```

2. Déclarer et définir la fonction publique **Present** dans la classe CDispositif

```

virtual void Present()
{
    PresentSpecific();
};

```

3. Déclarer la fonction protégée **PresentSpecific** dans la classe CDispositif

```

virtual void PresentSpecific()=0;

```

4. Déclarer la fonction protégée **PresentSpecific** dans la classe **CDispositifD3D11**

```
virtual void PresentSpecific();
```

5. L'implémenter ainsi:

```
void CDispositifD3D11::PresentSpecific()
{
    pSwapChain->Present( 0, 0 );
}
```

La fonction **Present** de la classe **IDXGISwapChain** permet de présenter le contenu du prochain tampon de notre chaîne d'échange. Dans notre cas, puisque nous n'avons que deux tampons, le tampon d'arrière-plan devient le tampon d'avant plan et le tampon d'arrière-plan devient le tampon d'avant plan. Pour les curieux, regardez dans la documentation de DirectX Graphics ce qui se passe si notre chaîne d'échange a plus de deux tampons.

3.9 Rendu de la scène

La dernière partie de l'animation est le rendu de la scène. Nous aurons une vraie scène au chapitre 4 (un petit cube!), mais rien ne nous empêche de mettre en place nos fonctions de rendu, nous les complèterons au chapitre 4.

L'affichage des éléments graphiques se fera dans la fonction **RenderScene** et les opérations spécifiques à la plateforme dans les fonctions **Begin-RenderSceneSpecific** et **EndRenderSceneSpecific**.

1. Modifier la fonction **RenderScene** de la classe CMoteur (que nous avions implantée au chapitre 2):

```
// Fonctions de rendu et de présentation de la scène
virtual bool RenderScene()
{
    BeginRenderSceneSpecific();

    // Appeler les fonctions de dessin de chaque objet de la scène
    // à suivre...

    EndRenderSceneSpecific();
    return true;
}
```

2. Implémenter les fonctions protégées **BeginRenderSceneSpecific** et **EndRenderSceneSpecific** dans la classe CMoteur

```
virtual void BeginRenderSceneSpecific()=0;
virtual void EndRenderSceneSpecific()=0;
```

3. Déclarer les fonctions protégées **BeginRenderSceneSpecific** et **EndRenderSceneSpecific** dans la classe CMoteurWindows

```
virtual void BeginRenderSceneSpecific();
virtual void EndRenderSceneSpecific();
```

4. Les implémenter ainsi:

```
void CMoteurWindows::BeginRenderSceneSpecific()
{
    ID3D11DeviceContext* pImmediateContext = pDispositif->GetImmediateContext();
    ID3D11RenderTargetView* pRenderTargetView = pDispositif->GetRenderTargetView();

    // On efface la surface de rendu
    float Couleur[4] = { 0.0f, 0.5f, 0.0f, 1.0f }; // RGBA - Vert pour le moment
    pImmediateContext->ClearRenderTargetView( pRenderTargetView, Couleur );
}
```

```
void CMoteurWindows::EndRenderSceneSpecific()
{
}
```

La fonction **ClearRenderTargetView** permet d'effacer le *viewport* avec la couleur choisie.

3.10 Faire le ménage

Lorsque notre application se termine ou dans des situations particulières, nous devrons relâcher les objets DirectX utilisés par notre application. Nous utiliserons ici le destructeur de CMoteur et la fonction **CMoteur::Cleanup**.

1. Déclarez la fonction protégée **CMoteur::Cleanup**.

```
virtual void Cleanup()
{
    // détruire les objets
    // à suivre

    // Détruire le dispositif
    if (pDispositif)
    {
        delete pDispositif;
        pDispositif = nullptr;
    }
}
```

2. Modifiez le destructeur de CMoteur pour qu'il appelle la fonction CleanUp.

```
// Destructeur
~CMoteur()
{
    Cleanup();
}
```

3. Modifiez le destructeur de CDispositifD3D11

```
CDispositifD3D11::~CDispositifD3D11()
{
    if (pImmediateContext)
    {
        pImmediateContext->ClearState();
    }

    DXRelacher(pRenderTargetView);
    DXRelacher(pImmediateContext);
    DXRelacher(pSwapChain);
    DXRelacher(pD3DDevice);
}
```

Notre programme est maintenant prêt à être compilé et exécuté. Bien sûr, nous ne faisons presque rien mais avons mis en place tous les éléments de base de l'environnement

4. Première scène

Maintenant que nous avons une application Direct3D fonctionnelle, nous pouvons commencer à y afficher une scène. Celle-ci sera très simple, il nous suffira d'un bloc (cube) tournant sur lui-même.

Objectifs du chapitre

- ✓ Créer par programmation un objet simple, un bloc (un cube). Et le placer dans notre « scène ».
- ✓ Le copier en mémoire graphique (nous dirons souvent **en ressources**).
- ✓ Introduction aux shaders HLSL
- ✓ Mise en place des transformations nécessaires à la modélisation de la scène.
- ✓ Une « petite » animation de scène.
- ✓ Le rendu de notre scène.

4. Première scène

Atelier

4.1 Initialisation de la scène

Après la création de l'environnement 3D de base, nous pouvons maintenant mettre en place la scène 3D en initialisant les objets graphiques, en préparant les matériaux et l'éclairage. Notre exercice effectuera ces initialisations dans la fonction CMoteur::InitScene.

La fonction InitScene

1. Ajouter l'appel à la fonction **InitScene** dans la fonction CMoteur :: Initialisations.

```
virtual int Initialisations()
{
    // Propre à la plateforme
    InitialisationsSpecific();

    // * Initialisation du dispositif de rendu
    pDispositif = CreationDispositifSpecific(CDS_FENETRE);

    // * Initialisation de la scène
    InitScene();

    // * Initialisation des paramètres de l'animation et
    //   préparation de la première image
    InitAnimation();

    return 0;
}
```

2. Déclarer la fonction InitScene dans la section *protected* de la classe CMoteur.

```
virtual int InitScene()
{
    // Initialisation des objets 3D - création et/ou chargement
    if (!InitObjets()) return 1;

    return 0;
}
```

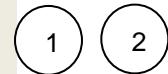
4.2 Le bloc est mis en scène

Dans notre exemple, les objets graphiques se limitent à un bloc. Néanmoins, il est toujours plus facile de travailler avec des objets graphiques ayant leur propre classe d'objet. Nous pourrons ainsi leur ajouter plus tard des attributs reliés à la logique de notre application (vitesse, interactions, hiérarchie, etc.). La création des objets graphiques est effectuée par la fonction **CMoteur::InitObjets**. Notre objet graphique sera de la classe **CBloc**, une classe qu'il nous faudra bien sûr définir correctement.

1. Déclarez la fonction **InitObjets** de la façon suivante (toujours dans une section *protected* de CMoteur):

```
bool InitObjets()
{
    // Puis, il est ajouté à la scène
    ListeScene.emplace_back(std::make_unique<CBloc>(2.0f, 2.0f, 2.0f,
pDispositif));

    return true;
}
```

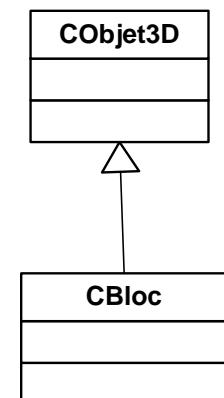


Attention ! Les choses se compliquent un peu. En (1), le bloc est créé et placé sur le dispositif, nous y reviendrons un peu plus loin. Mais en (2), le bloc est ajouté à une variable, **ListeScene**.

En effet, il deviendra bientôt essentiel de placer nos objets dans des « listes » représentant des scènes, des parties d'une scène, ou ce que l'on voudra... Donc même si nous n'avons qu'un seul objet pour le moment, j'effectue quand même une ébauche de gestion de scène en déclarant le vecteur **ListeScene** qui nous facilitera certaines tâches au moment de l'animation, du rendu et du nettoyage.

Toutefois pour rendre la liste de nos objets 3D plus polyvalente, nous allons construire une petite hiérarchie d'objets 3D et dériver **CBloc** d'une classe abstraite **CObject3D**.

Puis nous allons définir **ListeScene** comme contenant des **CObject3D*** (des pointeurs sur des objets **CObject3D**, donc nous pouvons y implanter le polymorphisme).



2. Dans la classe CMoteur, dans les variables protégées, définissez **ListeScene** ainsi :

```
// La seule scène  
std::vector<std::unique_ptr<CObjet3D>> ListeScene;
```

3. Ajoutez à votre projet le fichier suivant:

Objet3D.h

```
#pragma once  
#include "dispositif.h"  
  
using namespace DirectX;  
  
namespace PM3D  
{  
    // Classe : CObjet3D  
    //  
    // BUT : Classe de base de tous nos objets 3D  
    //  
    class CObjet3D  
    {  
        public:  
            virtual ~CObjet3D() = default;  
  
            virtual void Anime(float) {};  
            virtual void Draw() = 0;  
    };  
}  
} // namespace PM3D
```

Bien sûr, nous ajouterons d'autres éléments à cette classe par la suite.

4. Ajoutez à la suite des « #include » au début du fichier Moteur.h:

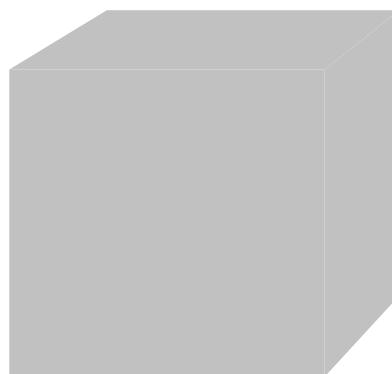
```
#include "Objet3D.h"  
#include "Bloc.h"
```

4.3 Le bloc est défini

Pour les besoins de notre exemple, nous allons construire un bloc. Ce n'est pas un gros objet 3D, mais c'est quand même beaucoup plus de travail qu'un simple triangle. De plus, comme c'est un objet réellement en 3D, nous pourrons y voir les liens entre les **points**, les **sommets** et les **polygones**.

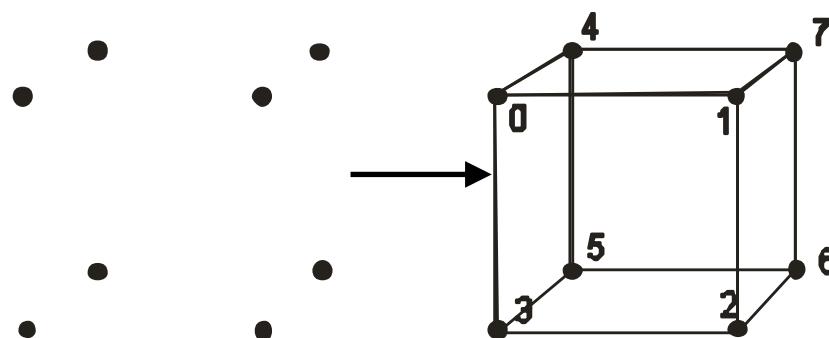
La classe CBloc est une **classe spécifique** c'est-à-dire qu'elle est conçue pour DirectX, mais nous ferons tout de même attention à ce que ses fonctions d'accès conservent la distinction générique spécifique que nous avons utilisée jusqu'à maintenant.

Une fois affiché, notre cube aura un peu cet aspect:

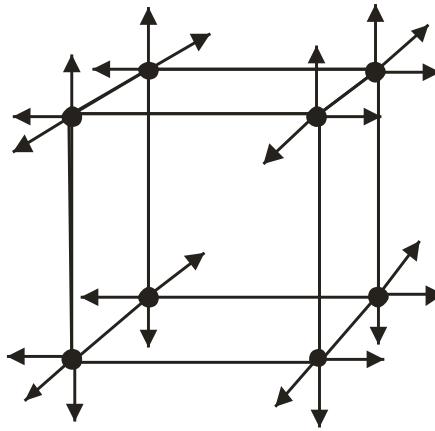


C'est normal, puisque nous n'aurons ni éclairage réel ni texture. Mais en réalité, notre cube sera préparé ainsi:

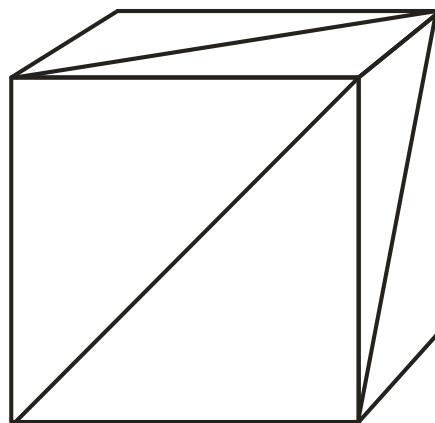
A. Huit (8) points de référence pour le positionnement des sommets



B. Ces huit points nous permettront de définir vingt-quatre (24) sommets. Nous aurons vingt-quatre sommets parce que nous désirons dès maintenant ajouter aux points leur normale. La normale étant l'élément de base pour le calcul des éclairages (plus de détails au chapitre 6).



C. Ces vingt-quatre sommets nous permettront de définir douze polygones (des triangles). Si nous utilisions une fonction telle **ID3D11DeviceContext::Draw**, nous aurions besoin de trente-six (36) sommets même si certains d'entre eux sont identiques. L'utilisation de la fonction **ID3D11DeviceContext::DrawIndexed** nous permettra de réutiliser les sommets qui apparaissent dans plus d'un polygone en utilisant leur numéro dans un index.



1. Définissez une nouvelle classe, la classe **CBloc**, qui aura pour fichiers BLOC.H pour l'interface et BLOC.CPP pour l'implémentation. La définition de cette classe se présente pour le moment ainsi:

Bloc.h

```
#pragma once
#include "Objet3D.h"
#include "DispositifD3D11.h"

namespace PM3D
{
    // Classe : CBloc
    // BUT : Classe de bloc
    class CBloc : public CObjet3D
    {
    public:
        CBloc(const float dx, const float dy, const float dz,
              CDispositifD3D11* pDispositif_);

        // Destructeur
        virtual ~CBloc(){}
    };
}
```

Bloc.cpp

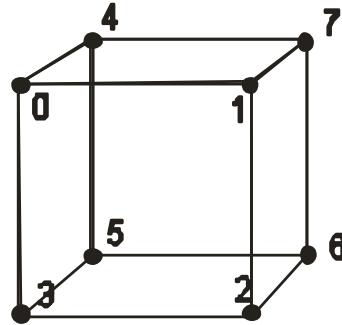
```
#include "stdafx.h"
#include "bloc.h"
#include "sommetbloc.h"
#include "MoteurWindows.h"

namespace PM3D
{
    // FONCTION : CBloc, constructeur paramétré
    // BUT : Constructeur d'une classe de bloc
    // PARAMÈTRES:
    //    dx, dy, dz:dimension en x, y, et z
    //    pDispositif: pointeur sur notre objet dispositif
    CBloc::CBloc(const float dx, const float dy, const float dz,
                  CDISPOSITIFD3D11* pDispositif_)
        : pDispositif(pDispositif_) // Prendre en note le dispositif
        //... sera complété plus loin
    }
}
```

2. Déclarez la variable *private* suivante:

```
CDISPOSITIFD3D11* pDispositif;
```

3. Modifiez le constructeur paramétré de CBloc ainsi pour y définir les points:



```
CBloc::CBloc(const float dx, const float dy, const float dz,
    CDispositifD3D11* pDispositif_)
    : pDispositif(pDispositif_) // Prendre en note le dispositif
{
    // Les points
    XMFBLOCK point[8] =
    {
        XMFBLOCK(-dx / 2, dy / 2, -dz / 2),
        XMFBLOCK(dx / 2, dy / 2, -dz / 2),
        XMFBLOCK(dx / 2, -dy / 2, -dz / 2),
        XMFBLOCK(-dx / 2, -dy / 2, -dz / 2),
        XMFBLOCK(-dx / 2, dy / 2, dz / 2),
        XMFBLOCK(-dx / 2, -dy / 2, dz / 2),
        XMFBLOCK(dx / 2, -dy / 2, dz / 2),
        XMFBLOCK(dx / 2, dy / 2, dz / 2)
    };
}
```

Vous noterez que le centre du bloc est en 0,0,0. Ce sera plus pratique pour lui faire faire une rotation sur lui-même.

4. Toujours dans le constructeur de CBloc, ajoutez, à la suite, le code pour le calcul des normales:

```
// Calculer les normales
XMFBLOCK n0(0.0f, 0.0f, -1.0f); // devant
XMFBLOCK n1(0.0f, 0.0f, 1.0f); // arrière
XMFBLOCK n2(0.0f, -1.0f, 0.0f); // dessous
XMFBLOCK n3(0.0f, 1.0f, 0.0f); // dessus
XMFBLOCK n4(-1.0f, 0.0f, 0.0f); // face gauche
XMFBLOCK n5(1.0f, 0.0f, 0.0f); // face droite
```

Notez que nos normales sont évidemment des vecteurs **normalisés**. Nous aurions pu utiliser des constantes pour définir celles-ci puisqu'elles seront toujours les mêmes pour tous les blocs. Mais j'ai préféré garder le code dans l'ordre de construction.

5. Toujours dans le constructeur de CBloc, vous ajoutez maintenant la construction des sommets:

```
CSommetBloc sommets[24];

// Le devant du bloc
sommets[0]=CSommetBloc(point[0],n0);
sommets[1]=CSommetBloc(point[1],n0);
sommets[2]=CSommetBloc(point[2],n0);
sommets[3]=CSommetBloc(point[3],n0);

// L'arrière du bloc
sommets[4]=CSommetBloc(point[4],n1);
sommets[5]=CSommetBloc(point[5],n1);
sommets[6]=CSommetBloc(point[6],n1);
sommets[7]=CSommetBloc(point[7],n1);

// Le dessous du bloc
sommets[8]=CSommetBloc(point[3],n2);
sommets[9]=CSommetBloc(point[2],n2);
sommets[10]=CSommetBloc(point[6],n2);
sommets[11]=CSommetBloc(point[5],n2);

// Le dessus du bloc
sommets[12]=CSommetBloc(point[0],n3);
sommets[13]=CSommetBloc(point[4],n3);
sommets[14]=CSommetBloc(point[7],n3);
sommets[15]=CSommetBloc(point[1],n3);
// La face gauche
sommets[16]=CSommetBloc(point[0],n4);
sommets[17]=CSommetBloc(point[3],n4);
sommets[18]=CSommetBloc(point[5],n4);
sommets[19]=CSommetBloc(point[4],n4);

// La face droite
sommets[20]=CSommetBloc(point[1],n5);
sommets[21]=CSommetBloc(point[7],n5);
sommets[22]=CSommetBloc(point[6],n5);
sommets[23]=CSommetBloc(point[2],n5);
```

6. Nous devons définir nous-mêmes l'organisation de nos sommets. Les sommets doivent répondre à certaines normes, mais doivent être définis par l'application.

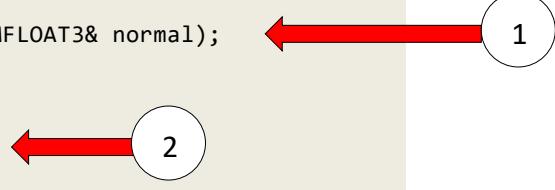
La classe de sommets CSommetBloc nous permettra de définir un type de sommet comprenant une **position** (un point) et une **normale**. Elle sera construite ainsi:

sommetsbloc.h

```
#pragma once
using namespace DirectX;
namespace PM3D
{
    class CSommetBloc
    {
    public:
        CSommetBloc() = default;
        CSommetBloc(const XMFLOAT3& position, const XMFLOAT3& normal);

    public:
        static UINT numElements;
        static D3D11_INPUT_ELEMENT_DESC layout[];

    private:
        XMFLOAT3 m_Position;
        XMFLOAT3 m_Normal;
    };
}
```


sommetsbloc.cpp

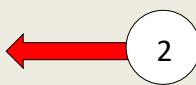
```
#include "StdAfx.h"
#include "sommetsbloc.h"

// Définir l'organisation de notre sommet
D3D11_INPUT_ELEMENT_DESC CSommetBloc::layout[] = 
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0 },

    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

UINT CSommetBloc::numElements = ARRAYSIZE(CSommetBloc::layout);

CSommetBloc::CSommetBloc(XMFLOAT3 _position, XMFLOAT3 _normal)
{
    position=_position;
    normal = _normal;
}
```


Explications

1

Chaque élément de type CSommetBloc contiendra une position et une normale. Dans les versions « pré DX 10 », l'ordre des éléments nous était imposé, ce n'est plus le cas. Il faudra par contre que les types et l'ordre respectent la déclaration d'organisation (le tableau **layout**).

Ici, nous utilisons le type **XMFLOAT3** pour désigner un tableau de trois éléments point flottant simple (le type float en C++). Ce type nous est fourni par DirectXMath (XM), mais **nous aurions pu utiliser toute autre définition compatible**.

2

Le tableau **layout** nous permettra à la section 4.7 de faire correspondre la déclaration de CSommetBloc à des éléments accessibles par les *shaders*. Ces éléments, appelés **sémantiques**, sont en réalité des noms réservés et associés à des registres spéciaux du GPU. Nous y reviendrons lorsque nous parlerons des shaders. Nous utilisons ici les sémantiques **POSITION** et **NORMAL**. Pour plus de détails, regardez la définition de **D3D11_INPUT_ELEMENT_DESC** dans le SDK.

7. Nous en sommes maintenant à la construction d'un index puisque nous utiliserons la fonction **DrawIndexed** pour le dessin de notre objet.

Cette fois nous utiliserons un tableau de constantes prédéfinies pour l'index. Dans le fichier Bloc.cpp, déclarez la constante **index_bloc**:

```
const uint16_t index_bloc[36] = {  
    0,1,2,           // devant  
    0,2,3,           // devant  
    5,6,7,           // arrière  
    5,7,4,           // arrière  
    8,9,10,          // dessous  
    8,10,11,         // dessous  
    13,14,15,        // dessus  
    13,15,12,        // dessus  
    19,16,17,        // gauche  
    19,17,18,        // gauche  
    20,21,22,        // droite  
    20,22,23        // droite  
};
```

Voilà, notre bloc est maintenant modélisé. J'ai préféré le modéliser dans des variables locales, mais nous aurions pu utiliser d'autres solutions et même construire les sommets directement sur la carte graphique.

Note: Il existe une fonction, **Draw**, qui n'utilise pas d'index. C'est plus simple dans certains cas. Mais comme la plupart des logiciels de 3D créent des objets 3D avec indexation, autant s'habituer tout de suite. De plus, l'indexation permet sur les cartes graphiques modernes un gain important en espace mémoire et en temps de traitement dans presque tous les cas.

Atelier

4.4 Copier le bloc en ressource

Nous devons maintenant copier nos sommets ainsi que l'index sur la carte graphique pour qu'ils soient accessibles au pipeline lors de l'exécution de la fonction **DrawIndexed**. Les sommets sont placés dans un tampon mémoire appelé **tampon des index** ou *vertex buffer*. Les indices sont placés dans un tampon appelé **tampon des indices** ou *index buffer*.

1. Création et initialisation du *vertex buffer*. Ajoutez le code suivant à la suite du constructeur de CBloc:

```
// Cr ation du vertex buffer et copie des sommets
ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

D3D11_BUFFER_DESC bd;
ZeroMemory(&bd, sizeof(bd));

bd. Usage = D3D11_USAGE_IMMUTABLE;
bd. ByteWidth = sizeof(sommets);
bd. BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd. CPUAccessFlags = 0;

D3D11_SUBRESOURCE_DATA InitData;
ZeroMemory(&InitData, sizeof(InitData));
InitData.pSysMem = sommets;

DXESSAYER(pD3DDevice->CreateBuffer(&bd, &InitData, &pVertexBuffer),
DXE_CREATIONVERTEXBUFFER);
```

Explications

La fonction **CreateBuffer** est utilis e pour cr er un tampon en m moire vid o (ou l' quivalent). Elle nous demande trois param tres soient:

Param tre 1 – Description du tampon (*Buffer Description*)

La description du tampon se fait dans une structure D3D11_BUFFER_DESC. **Usage** d termine la fa on dont le GPU et le CPU utilisent cette ressource. Le cas le plus fr quent est **D3D11_USAGE_DEFAULT** et **D3D11_USAGE_IMMUTABLE**, mais vous pouvez regarder dans le SDK pour bien comprendre les autres usages. **ByteWidth** est la dimension en octets de notre tampon. **BindFlags** indique le r ole de la ressource dans le pipeline, dans notre cas D3D11_BIND_VERTEX_BUFFER. **CPUAccessFlags** indique la fa on dont le CPU pourrait acc der 脿 ces donn es, dans notre cas 0 indique qu'il n'y aura pas d'acc s.

Paramètre 2 – Données initiales (*Init Data*)

À moins d'organisation spéciale de nos données, seul le champ pSysMem est utilisé. Il pointe sur nos données d'initialisation. Il est possible d'initialiser ou de modifier un tampon ailleurs que dans la fonction CreateBuffer, mais dans ce cas le paramètre 1 devra être initialisé différemment.

Par exemple avec

```
Usage = D3D11_USAGE_DYNAMIC  
CPUAccessFlags = D3D11_CPU_ACCESS_WRITE
```

Paramètre 3 – Le vertex buffer

En réalité, nous recevons un pointeur sur un objet **ID3D11Buffer** qui n'est pas réellement le tampon, mais plutôt une interface d'accès.

2. Déclarez la variable **pVertexBuffer** dans la section *private* de la classe CBloc:

```
ID3D11Buffer* pVertexBuffer;
```

3. Comme nous utilisons DXEssayer, ajoutez les trois lignes suivantes à la suite des #include dans le fichier **bloc.cpp**.

```
#include "resource.h"  
#include "util.h"
```

4. Déclarez dans le tableau des « strings » en ressource:

```
DXE_CREATIONVERTEXBUFFER Erreur dans la création du vertex buffer
```

5. Crédit et initialisation de l'*index buffer*. Ajoutez le code suivant à la suite du constructeur de CBloc:

```
// Crédit de l'index buffer et copie des indices  
ZeroMemory(&bd, sizeof(bd));  
  
bd. Usage = D3D11_USAGE_IMMUTABLE;  
bd. ByteWidth = sizeof(index_bloc);  
bd. BindFlags = D3D11_BIND_INDEX_BUFFER;  
bd. CPUAccessFlags = 0;  
  
ZeroMemory(&InitData, sizeof(InitData));  
InitData.pSysMem = index_bloc;  
  
DXEssayer(pDXDDevice->CreateBuffer(&bd, &InitData, &pIndexBuffer),  
DXE_CREATIONINDEXBUFFER);
```

6. Déclarez la variable **pIndexBuffer** dans la section *private* de la classe CBloc:

```
ID3D11Buffer* pIndexBuffer;
```

7. Déclarez dans le tableau des « strings » en ressource:

```
DXE_CREATIONINDEXBUFFER Erreur dans la création de l'index buffer
```

4.5 Les shaders

Avec DirectX 9 est arrivé l'un des aspects les plus intéressants des dernières années soit l'intégration du « High Level Shading Language » (HLSL) pour la conception des *shaders*. L'utilisation de ce langage permet aux concepteurs de shaders de se concentrer sur la programmation de la solution sans avoir à connaître tous les détails de l'implantation physique des shaders sur la carte graphique. Ces détails: nombre de registres, leur rôle, leur allocation, le support de certaines instructions, etc., ralentissaient le développement d'un shader puisque les développeurs non familiers avec le « langage d'assemblage » du GPU étaient nettement désavantagés.

Mais avant d'aller plus loin, qu'est-ce qu'un « shader » ?

Initialement utilisés par RenderMan, le logiciel conçu pour Pixar pour effectuer le rendu des images de leurs films, les shaders ont été ajoutés tout récemment (depuis 2000) dans les bibliothèques 3D (dans DirectX 8.0 et certaines extensions de OpenGL, ils font partie de la norme OpenGL 1.4), et ont très vite commencé à être supportés par les principaux constructeurs de cartes graphiques (par exemple nVidia et ATI).

Les shaders sont des petits programmes (en réalité de petites fonctions) destinés à être chargés et exécutés par la carte graphique lors des différentes étapes du rendu. Leur but est de s'occuper de certaines parties du pipeline et d'offrir au développeur des possibilités très élaborées de programmation. En réalité, presque tout est possible, à condition de l'écrire et de respecter les contraintes mémoire de la carte graphique. Nous pourrons choisir un shader pour certains rendus puis le changer pour d'autres.

Il existe deux types principaux de shaders: les *vertex shaders* et les *pixel shaders*. Un troisième type: les *geometry shaders* ont été introduit avec DirectX 10 et deux nouveaux avec DirectX 11: les *compute shaders* et la paire Hull-Domain (*Tessellation Shader*). Tous fonctionnent d'une façon similaire, bien qu'ils remplissent évidemment un rôle différent.

Les *vertex shaders* effectuent principalement les opérations « transformation, déplacement, attributs de matériaux et éclairage » du pipeline de visualisation, se plaçant donc presqu'au début de celui-ci. Les vertex shaders ne traitent **qu'un seul sommet à la fois**.

Les *geometry shaders* effectuent certains traitements sur une base de primitive. On y retrouvera des traitements comme la sélection de matériel, la détection des contours (silhouette). Ils peuvent générer de nouvelles primitives pour les systèmes de particules par exemple. Ils sont apparus avec la norme « shaders 4.0 ». Nous y reviendrons au chapitre 14.

Les *pixel shaders* viennent remplacer les parties texturage, filtrage et mélange du pipeline, ce qui va cette fois les placer à la fin du pipeline, juste avant l'étape finale de rendu. Son rôle sera de créer le pixel final qui sera écrit sur le tampon d'affichage, à partir des informations sur celui-ci telles

Certains auteurs proposent le terme « nuanceur » en français, mais la plupart des traductions françaises acceptent ou préconisent le terme **shader**

que les textures qu'il utilise, la couleur diffuse, la couleur spéculaire, et les coordonnées de textures. Les pixel shaders ne traitent qu'un pixel à la fois.

Et les *compute shaders*? Apparus avec la norme « Shaders 5.0 » ceux-ci ne s'intègrent pas directement au pipeline. Comme leur nom l'indique, ils servent à effectuer des calculs. Nous pouvons donc tirer profit de la puissance de traitement des GPUs. Les compute shaders sont écrits en HLSL aussi et utilisent une technologie appelée **DirectCompute** chez Microsoft. C'est à quelques détails près l'équivalent de C for Cuda (NVidia) ou de OpenCL (Khronos). Le grand avantage des compute shaders est qu'ils sont indépendants du pipeline de rendu. En effet, le couple vertex et pixel shader est étroitement lié à plusieurs fonctions fixes comme le rasterizer et l'output merger. Un compute shader nous permet d'utiliser directement la puissance de calcul du GPU sans passer par tous le pipeline de rendu.

Le HLSL

Pour simplifier l'utilisation des shaders avec DirectX, Microsoft a développé (conjointement avec un comité de design « indépendant ») un langage de création de shaders appelé HLSL. Le HLSL est aussi très fortement inspiré du Cg (C for Graphics) de NVidia qui (comme par hasard!) est un des principaux intervenants du comité de design du HLSL. Le but du HLSL est de fournir un langage de plus haut niveau que l'assembleur des shaders, le comité a choisi un langage ressemblant au langage C, mais incorporant les fonctionnalités avancées des GPU supportant les shaders, par exemple les traitements de matrices.

Note: *Le GLSL d'OpenGL a utilisé exactement les mêmes critères dans sa conception, c'est pourquoi tout ça finit par se ressembler beaucoup.*

En réalité, le code HLSL doit, comme le langage d'assemblage, être compilé pour être transféré sur la carte graphique. Dans le cas de DirectX11, le HLSL est compilé en assembleur D3D. Cette représentation intermédiaire sera ensuite compilée par le pilote d'affichage. Ainsi le même code peut être exécuté de façon transparente sur plusieurs architectures différentes (AMD, NVidia, Intel).

Il n'est pas nécessaire d'élaborer longtemps les avantages de langages tel le HLSL sur les langages d'assemblage des cartes graphiques: portabilité, adaptation facile aux nouvelles versions et aux nouvelles fonctionnalités des cartes et surtout utilisation d'un langage plus près du programmeur moderne.

4.6 Nos premiers shaders

Afin de pouvoir effectuer un rendu, toutes les applications DirectX « post DX 9 » doivent au moins utiliser un vertex shader et un pixel shader.

Dans cette section, nous construirons un *vertex shader* (VS) très simple et un pixel shader encore plus simple. Le but n'est pas d'obtenir quelque chose de très beau, mais plutôt de se familiariser avec un premier moyen d'intégrer les shaders dans nos applications DirectX.

La **compilation « runtime »** utilise un fichier texte contenant un shader (VS ou PS selon le cas), et le charge et le compile lors de l'exécution du programme. Le shader sera par la suite placé sur la carte graphique. Cette méthode, pas très rapide je vous l'accorde, présente l'immense avantage de pouvoir faire une bonne partie du développement de nos shaders sans avoir à reconstruire le projet à chaque fois. C'est particulièrement utile si on utilise un outil externe pour la mise au point des shaders (**RenderMonkey** ou **FXComposer** par exemple).

Le petit vertex shader – premiers éléments de HLSL

C'est une version minimale de vertex shader. On ne peut pas faire de code vide puisque le système attend un minimum de travail de la part des shaders.

Comme nous en avons déjà parlé, le vertex shader a comme rôle de transformer les sommets d'un polygone (généralement un triangle) de l'espace de la scène à l'espace de l'écran. Normalement, c'est là que l'on calcule les informations pour l'éclairage, les textures et autres, mais notre version sera plus limitée.

Voici notre premier vertex shader:

```
cbuffer param
{
    float4x4 matWorldViewProj;
}

float4 VS1(float4 Pos_in : POSITION) : SV_Position
{
    return mul(Pos_in, matWorldViewProj);
}
```

1. Créez un petit fichier : **VS1.vhl** et écrivez-y le code du shader. Placez le fichier dans votre répertoire projet (au même endroit que vos .cpp), ce n'est pas nécessaire de l'ajouter à votre projet dans Visual Studio, mais vous pouvez le faire pour en faciliter l'édition.

L'extension **.vhl** est l'extension recommandée pour les vertex shaders HLSL, mais comme vous le verrez bientôt, ça n'a pas réellement d'importance.

Un peu d'explications sur ce shader qui, même s'il est tout petit, contient quand même la base de tout bon vertex shader.

```
cbuffer param
{
    float4x4 matWorldViewProj;
}
```

La variable **matWorldViewProj** est comme son nom l'indique une matrice résultant de la multiplication de la matrice de transformation dans le monde (*world*), de la matrice de vision (*view*) et de la matrice de projection (*projection*). Avec HLSL 4.0 et 5.0, les constantes doivent être dans un bloc de type **cbuffer**. Nous devrons trouver un moyen de l'initialiser dans notre application...

float4x4 est un type HLSL correspondant à une matrice 4 X 4 (un type facilement représentable sur la carte graphique). Chaque élément de la matrice est une valeur point-flottant de 32 bits correspondant généralement au type **float** de C++ (le standard IEEE-754 avec à l'occasion certains ajustements).

→ Notez que **matWorldViewProj** sera souvent identifiée comme une **constante du shader** et non pas comme une variable. Pourquoi ?

Parce que **matWorldViewProj** ne changera pas lors de l'exécution du shader sur tous les sommets, elle demeurera donc constante. Toutefois, nous aurons la possibilité de la modifier avant la prochaine exécution du shader c'est-à-dire le prochain **Draw** (ou **DrawIndexed**).

```
float4 VS1(float4 Pos_in : POSITION) : SV_Position
```

La fonction **VS1** spécifie une valeur de retour de type **float4** (un vecteur de 4 éléments), le texte à la fin de la ligne « **:SV_Position** » correspond à une **sémantique** c'est-à-dire un mot réservé associé à une variable de travail du HLSL. Ces sémantiques correspondent habituellement à des entrées/sorties des shaders vers le pipeline, mais nous servirons à l'occasion à d'autres utilisations.

Vous remarquerez que le paramètre de la fonction, **Pos_in**, prend aussi sa valeur de la sémantique **POSITION**. On peut donc en déduire que la fonction VS1 reçoit la position d'un sommet en entrée et retourne une nouvelle valeur de celle-ci en sortie.

```
return mul(Pos_in, matWorldViewProj);
```

mul est une fonction HLSL qui effectue la multiplication du vecteur 1 x 4 **Pos_in** par la matrice **matWorldViewProj** et qui retourne un vecteur 1 x 4 (ou 4 x 1, il n'y a pas de différence en HLSL) donc qui transforme une coordonnée de l'espace de la scène à l'espace de l'écran.

À retenir : les sémantiques **POSITION** position d'un sommet en entrée
SV_Position position transformée (résultat)

NOTE: En réalité, depuis DX10 et HLSL 4.0, les sémantiques sont TOUTES définies par l'utilisateur sauf les sémantiques **SV_xxx** qui correspondent à certains registres prédéfinis.

Le petit pixel shader

Comme nous en avons déjà parlé, le pixel shader a comme rôle principal de transformer les pixels d'un polygone en leur donnant, par exemple, leur couleur finale. Les premiers pixels shaders que nous réaliserons ne s'occupent que de la couleur du pixel résultant, mais d'autres possibilités nous sont offertes (textures, transparence, effets, déformations...).

Voici notre premier pixel shader, il retourne la même couleur pour tous les pixels, ce n'est pas très beau, mais c'est un début:

```
float4 PS1() : SV_Target
{
    return(float4( 1.0f, 0.0f, 1.0f, 1.0f ));
}
```

1. Créez un petit fichier : **PS1.phl** et écrivez-y le code du shader. Placez le fichier dans votre répertoire projet, mais ne l'ajoutez pas à votre projet dans Visual Studio.

À retenir : la sémantique **SV_Target** valeur de sortie sur la surface de rendu

Atelier

4.7 Compilation et chargement des shaders

Note importante de design d'application

Pour les besoins de nos premiers pas avec le pipeline de DirectX, je placerai le chargement et la gestion des shaders directement dans l'objet qui les utilise. Mais, très rapidement, nous constaterons que plusieurs objets utiliseront les mêmes shaders (ou d'autres situations du même type). Il sera alors plus efficace d'avoir un « **gestionnaire de shaders** ». Nous en planterons un lorsque nous construirons nos autres gestionnaires de ressources (texture, maillages, etc.).

Dans cette section, nous effectuerons trois tâches soient:

- La compilation « *runtime* » du vertex shader, et son chargement en mémoire vidéo.
- La création d'un objet qui fera correspondre notre organisation de sommet aux entrées du vertex buffer (*Input Layout*). Cet objet (de type ID3D11InputLayout) doit être créé après le vertex buffer.
- La compilation « *runtime* » du pixel shader, et son chargement en mémoire vidéo.

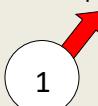
1. Initialisations des shaders -- insérez le code suivant à la suite du constructeur de CBloc :

```
InitShaders();
```

2. Déclarez la fonction **InitShaders** dans la section *private* – Puis implantez-la ainsi:

```
void CBloc::InitShaders()
{
    // Compilation et chargement du vertex shader
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    ID3DBlob* pVSBlob = nullptr;
    DXEasser(D3DCompileFromFile(L"vs1.vhl",
        nullptr, nullptr,
        "VS1",
        "vs_5_0",
        D3DCOMPILE_ENABLE_STRICTNESS,
        0,
        &pVSBlob, nullptr), DXE_FICHIER_VS);
```



1

```

DXEssayer(pD3DDevice->CreateVertexShader(pVSBlob->GetBufferPointer(),
    pVSBlob->GetBufferSize(),
    nullptr,
    &pVertexShader),
    DXE_CREATION_VS);
}

```



Explications

1

La fonction **D3DCompileFromFile** est une fonction utilitaire qui permet de compiler en mémoire un shader (VS, PS ou autre) ou un effet (que nous verrons plus loin). Regardez cette fonction dans la documentation du SDK. Nous noterons tout de même certains paramètres:

L" vs1.vhl"

Le nom du fichier contenant notre shader. Rappel: le L est parce que ce paramètre est Unicode.

"VS1"

Le nom du point d'entrée de notre vertex shader. Par défaut, ce nom est « main », ce qui deviendra rapidement limitatif.

"vs_5_0"

Le type de compilation désirée. **Vertex Shader version 5 .0** dans notre cas. Notez que nos premiers shaders se contenteraient bien de la version 2.0. Mais comme nous essayons d'être à la fine pointe...

D3DCOMPILE_ENABLE_STRICTNESS

Par défaut, le compilateur désactive les messages d'erreur et/ou d'avertissement reliés à des termes désuets dans le code HLSL. Nous avons activé cette vérification. Note: il sera peut-être nécessaire de retirer cette option lorsque nous utiliserons des shaders de version 3.0 ou moins créés par des logiciels comme FXComposer.

&pVSBlob

Un tampon qui contiendra le code compilé de notre shader. (**blob** = Binary Large OBject).

2

La fonction **ID3D11Device::CreateVertexShader** copie le vertex shader en mémoire vidéo (ou l'équivalent) à partir du blob et nous permet de créer un objet ID3D11VertexShader, en réalité un objet permettant d'identifier le shader et jusqu'à un certain point d'y accéder, mais ce n'est pas réellement le shader. Le relâchement (*release*) de cet objet libérera la mémoire vidéo associée (si elle n'est plus utilisée ailleurs).

3. Déclarez dans la section *private* de CBloc la variable:

```
ID3D11VertexShader* pVertexShader;
```

4. Déclarez dans le tableau des « strings » en ressource:

DXE_FICHIER_VS	Erreur compilation ou existence du fichier VS
DXE_CREATION_VS	Erreur création du VS

5. *Input Layout* – Nous reviendrons à plusieurs reprises sur les liens entre l’organisation des sommets et le vertex shader. Cette organisation sera utilisée dans la première étape du pipeline de DX11 soit l’étape « **Input Assembler** » (*IA stage*). Insérez le code suivant à la suite de la fonction InitShaders:

```
// Créer l'organisation des sommets
pVertexLayout = nullptr;
DXESSAYER(pD3DDevice->CreateInputLayout(CSommetBloc::layout,
    CSommetBloc::numElements,
    pVSBlob->GetBufferPointer(),
    pVSBlob->GetBufferSize(),
    &pVertexLayout),
    DXE_CREATIONLAYOUT);

pVSBlob->Release(); // On n'a plus besoin du blob
```

6. Déclarez la variable **pVertexLayout** de la façon suivante dans la section *private* de CBloc:

```
ID3D11InputLayout* pVertexLayout;
```

7. Déclarez dans le tableau des « strings » en ressource:

DXE_CREATIONLAYOUT	Erreur de création du layout
--------------------	------------------------------

8. Échange de données -- Notre vertex shader travaille avec une constante, la matrice **matWorldViewProj** que nous devrons initialiser avant chaque rendu. C'est une constante pour le pipeline, mais une variable pour notre application. Pour échanger avec le pipeline, nous utiliserons un tampon (un autre!) que nous déclarerons ainsi dans la section *private* de CBloc:

```
ID3D11Buffer* pConstantBuffer;
```

9. Puis, nous pourrons le créer. Ajoutez les lignes suivantes, toujours à la suite de la fonction InitShaders:

```
// Création d'un tampon pour les constantes du VS
D3D11_BUFFER_DESC bd;
ZeroMemory(&bd, sizeof(bd));

bd. Usage = D3D11_USAGE_DEFAULT;
bd. ByteWidth = sizeof(matWorld);
bd. BindFlags = D3D11_BIND_CONSTANT_BUFFER;
bd. CPUAccessFlags = 0;
pD3DDevice->CreateBuffer(&bd, nullptr, &pConstantBuffer);
```

10. Insérer le code suivant à la suite de la fonction InitShaders:

```
// Compilation et chargement du pixel shader
ID3DBlob* pPSBlob = nullptr;
DXEssayer(D3DCompileFromFile(L" ps1.phl",
    nullptr, nullptr,
    "PS1",
    "ps_5_0",
    D3DCOMPILE_ENABLE_STRICTNESS,
    0,
    &pPSBlob,
    nullptr), DXE_FICHIER_PS);

DXEssayer(pD3DDevice->CreatePixelShader(pPSBlob->GetBufferPointer(),
    pPSBlob->GetBufferSize(),
    nullptr,
    &pPixelShader),
    DXE_CREATION_PS);

pPSBlob->Release(); // On n'a plus besoin du blob
```

Ce code ressemble beaucoup à celui que nous avions utilisé pour le vertex shader.

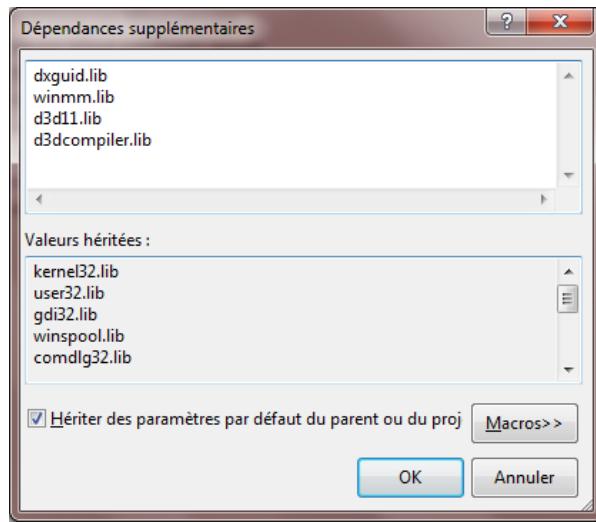
11. Déclarez dans la section *private* de CBloc la variable:

```
ID3D11PixelShader* pPixelShader;
```

12. Comme nous utiliserons souvent les compilations de shaders et qu'elles pourront avoir lieu ailleurs que dans CBloc, ajoutez la ligne suivante à la fin du fichier **stdafx.h** (à la suite des autres «#include»):

```
#include <d3dcompiler.h>
```

13. Insérez aussi **d3dcompiler.lib** dans les dépendances supplémentaires de l'éditeur de liens.



14. Déclarez dans le tableau des « strings » en ressource:

DXE_FICHIER_PS Erreur compilation ou existence du fichier PS

DXE_CREATION_PS Erreur création du PS

4.8 Mise en scène – les transformations

Les matrices de transformation

Les matrices reliées au monde, à la vue (caméra) et à la matrice de projection sont à la base du système de transformation le plus utilisé en 3D. Nous utiliserons ces matrices pour placer les objets dans la scène, ajuster la position et l'orientation de la caméra et déterminer la projection. Il s'agit d'un modèle de travail et il peut être simplifié au besoin.

La matrice de transformation dans le monde

Cette matrice porte souvent en anglais les noms **matWorld**, **World**, ou tout simplement **W**.

Cette matrice est un peu embêtante. En effet, chaque objet de notre scène aura besoin d'avoir sa matrice de transformation dans le monde. Donc la matrice « totale » ou WVP devra être recalculée chaque fois que nous voudrons faire le rendu d'un nouvel objet.

Cette matrice est utilisée pour placer les polygones dans le monde. Brièvement, les polygones peuvent être définis en coordonnées absolues ou en coordonnées relatives. La plupart des applications 3D utiliseront des coordonnées relatives puisqu'ainsi les manipulations d'objets sont grandement simplifiées.

Un gain intéressant pourrait être réalisé si les objets immobiles étaient déjà définis dans le monde. Dans ce cas un seul calcul de la matrice WVP serait nécessaire pour l'ensemble de ces objets. Toutefois, la présence d'objets complexes et l'utilisation des « éditeurs de scène » ne permettent pas toujours cette petite optimisation.

La matrice de vision

Cette matrice porte souvent en anglais les noms **matView**, **View**, ou tout simplement **V**.

Cette matrice définit la position et l'orientation de la caméra. Il est possible ici aussi de définir cette matrice en tenant compte de nos propres critères, mais si l'on désire appliquer une transformation « standard » comme projection, nous pouvons utiliser la fonction de la bibliothèque DirectXMath **XMMatrixLookAtLH**. Elle aussi pourra être modifiée en cours de route pour changer la position de la caméra par exemple. Pour les besoins de notre exemple, la caméra sera fixe.

La matrice de projection

Cette matrice porte souvent en anglais les noms **matProj**, **Proj**, **Projection**, ou tout simplement **P**.

Cette matrice définit comment la scène sera projetée sur la surface de rendu. Cette matrice est habituellement une projection en perspective,

mais nous pourrions la définir comme bon nous semble (attention aux résultats!). Si la projection en perspective nous convient, nous pouvons utiliser la fonction utilitaire de la bibliothèque DirectXMath **XMMatrixPerspectiveFovLH** qui nous permettra de créer plus facilement notre matrice de projection. Le **champ de vision** correspond à l'angle utilisé par la caméra, cet angle est exprimé en radians d'où l'utilisation de la constante XM_PI. Le **ratio d'aspect** est le rapport entre la largeur et la hauteur de notre surface destination.

Les matrices en résumé

matWorld, World, ou W	Matrice de transformation dans le monde
matView, View, ou V	Matrice de vision
matProj, Proj, Projection, ou P	Matrice de projection
matViewProj, ViewProj, ou VP V×P	
matWorldViewProj, WorldViewProj, ou WVP W×V×P	Matrice « totale »
et occasionnellement matWorldView, WorldView, ou WV W×V	

MAIS JAMAIS

WP W×P

4.9 Initialisation des matrices

Matrice de transformation dans le monde

- Définissez dans la section *private* de CBloc:

```
XMMATRIX matWorld;
```

- Puis initialisez dans le constructeur de CBloc, à la suite:

```
matWorld = XMMatrixIdentity();
```

La bibliothèque **DirectXMath** est une bibliothèque mathématique possédant deux intérêts principaux: elle est fournie avec DirectX et les formats des types sont compatibles avec les shaders (même si parfois, on doit faire de petites modifications). D'autres bibliothèques mathématiques existent, certaines plus facilement portables, mais attention à la conversion de données!

Plus de détails dans l'annexe **Mathématiques 3D**.

Matrice de vision et matrice de projection

La matrice de vision, à la base de la caméra ainsi que la matrice de projection seront définies dans la classe CMoteur. La **matrice de vision** sert aux mouvements de caméra, mais dans notre cas, elle sera fixe. La **matrice de projection**, comme son nom l'indique, permet de paramétriser la projection des sommets vers l'écran, elle est rarement modifiée lors de l'exécution d'applications 3D, mais pourquoi pas ?.

- Déclarez dans la classe CMoteur (Moteur.h) dans la section *protected*:

```
// Les matrices
XMMATRIX matView;
XMMATRIX matProj;
XMMATRIX matViewProj;
```

Vous remarquez que j'ai défini une matrice **matViewProj** qui contiendra le produit **matView * matProj**. Dans notre cas, comme ces deux matrices sont fixes, il est inutile de recalculer leur produit à chaque image. Lorsque votre caméra sera mobile, vous devrez recalculer ce produit à chaque changement de la matrice de vision ou de projection.

- Ajoutez les lignes suivantes à la suite de la fonction CMoteur::InitScene:

```
// Initialisation des matrices View et Proj
// Dans notre cas, ces matrices sont fixes
matView = XMMatrixLookAtLH( XMVectorSet( 0.0f, 0.0f, -10.0f, 1.0f ),
                           XMVectorSet( 0.0f, 0.0f, 0.0f, 1.0f ),
                           XMVectorSet( 0.0f, 1.0f, 0.0f, 1.0f ) );

float champDeVision = XM_PI/4;    // 45 degrés
float ratioDAspect = 1.0;        // horrible, il faudra corriger ça
```

```
float planRapproche = 2.0;
float planEloigne = 20.0;

matProj = XMMatrixPerspectiveFovLH(
    champDeVision,
    ratioDAspect,
    planRapproche,
    planEloigne);

// Calcul de VP à l'avance
matViewProj = matView * matProj;
```

3. Nous définissons aussi des fonctions d'accès aux matrices dans la section *public* de CMoteur. Ces fonctions nous permettront d'avoir accès à nos matrices à partir de n'importe quel objet via le fait que notre moteur est un singleton. Ces fonctions sont **GetMatView**, **GetMatProj** et **GetMatViewProj**:

```
const XMATRIX& GetMatView() const { return m_MatView; }
const XMATRIX& GetMatProj() const { return m_MatProj; }
const XMATRIX& GetMatViewProj() const { return m_MatViewProj; }
```

- Note:** Certains définissent les matrices Vision et Projection dans un objet, par exemple la **caméra**. C'est une excellente idée. Mais pour notre petite application, ce ne sera pas nécessaire.

4.10 L'animation de la scène

Dans notre cas, nous n'y retrouverons qu'une rotation de notre bloc.

1. Modifiez la fonction **CMoteur :: Animation** pour qu'elle appelle une nouvelle fonction: **AnimeScene**. Nous avions mis cet appel en commentaire, nous en profitons pour enlever les commentaires (//) devant l'appel de cette fonction.

```
...
// Affichage optimisé
pDispositif->Present();

TempsEcoule=static_cast<float>(TempsCourant-TempsPrecedent)
    * static_cast<float>(EchelleTemps);

// On prépare la prochaine image
// AnimeScene(TempsEcoule);

// On rend l'image sur la surface de travail (tampon d'arrière plan)
RenderScene();

...

```

Dans PetitMoteur3D, tous les objets sont « **animables** », mais dans une application plus élaborée, vous pourriez avoir différentes catégories d'objets pour éviter les appels de fonctions « vides ».

2. Déclarez et implantez la fonction **CMoteur::AnimeScene** dans la section *protected* de CMoteur:

```
for (auto& object3D : ListeScene)
{
    object3D->Anime(tempsEcoule);
}

return true;
```

3. Déclarer la fonction publique **Anime** dans la classe **CObjet3D**

```
virtual void Anime(float tempsEcoule){};
```

4. Déclarer la fonction **Anime** dans la section *public* de la classe **CBloc**:

```
virtual void Anime(float tempsEcoule);
```

5. Implanter la fonction **Anime** dans la classe **CBloc** de la façon suivante:

```
void CBloc :: Anime(float tempsEcoule)
{
    rotation = rotation + ((XM_PI * 2.0f) / 3.0f * tempsEcoule);

    // modifier la matrice de l'objet bloc
    matWorld = XMMatrixRotationX(rotation);
}
```

6. Définir la variable *rotation* dans la section *private* de CBloc:

```
float rotation;
```

7. L'initialiser à 0.0f dans le constructeur de CBloc:

```
rotation = 0.0f;
```

Dans notre exemple, nous désirons faire tourner le bloc sur l'axe des X à raison de 1 tour en 3 secondes. Le mouvement de l'objet est pour le moment fixé par la seule variable **rotation**, mais une application plus complète pourra modifier les paramètres d'animation d'un objet via une série de variables et d'accesseurs.

4.11 Le rendu de la scène

Lorsque la fonction **AnimeScene** a fini son travail, les matrices des objets et de la caméra devraient refléter la scène telle qu'on désire l'afficher. Il est maintenant temps de rendre ces informations sur la surface de rendu.

1. Modifiez la fonction **RenderScene** de la classe **CMoteur** (que nous avions implantée au chapitre 2):

```
virtual bool RenderScene()
{
    BeginRenderSceneSpecific();

    // Appeler les fonctions de dessin de chaque objet de la scène
    for (auto& object3D : ListeScene)
    {
        object3D->Draw();           ←
    }

    EndRenderSceneSpecific();
    return true;
}
```

2. Nous devons donc implémenter la fonction **Draw** dans la classe **CObject3D** (*public*). C'est d'ailleurs un des principaux objectifs de notre petite hiérarchie d'objet que de permettre le dessin polymorphe des objets.

```
virtual void Draw()=0;
```

3. Déclarez aussi la fonction **Draw** dans la section *public* de la classe **CBloc**:

```
void Draw();
```

4. L'implémentation de la fonction **Draw** dans la classe **CBloc** aura pour rôle d'effectuer le rendu de l'objet graphique au moyen de la fonction **DrawIndexed** de la classe **D3D11DeviceContext**. Toutefois avant d'effectuer l'appel à cette fonction, nous devons informer le pipeline de notre contexte de travail.

Voici donc la fonction CBloc::Draw:

```

void CBloc::Draw()
{
    // Obtenir le contexte
    ID3D11DeviceContext* pImmediateContext = pDispositif->GetImmediateContext(); 1

    // Choisir la topologie des primitives
    pImmediateContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    // Source des sommets
    const UINT stride = sizeof(CSommetBloc);
    const UINT offset = 0;
    pImmediateContext->IASetVertexBuffers( 0, 1, &pVertexBuffer, &stride, &offset ); 3

    // Source des index
    pImmediateContext->IASetIndexBuffer(pIndexBuffer, DXGI_FORMAT_R16_UINT, 0); 4

    // input layout des sommets
    pImmediateContext->IASetInputLayout(pVertexLayout); 5

    // Activer le VS
    pImmediateContext->VSSetShader(pVertexShader, nullptr, 0); 6

    // Initialiser et sélectionner les « constantes » du VS
    const XMATRIX viewProj = CMoteurWindows::GetInstance().GetMatViewProj();
    const XMATRIX matWorldViewProj = XMMatrixTranspose(matWorld * viewProj) ;
    pImmediateContext->UpdateSubresource( pConstantBuffer, 0, nullptr,
                                            &matWorldViewProj, 0, 0 ); 7

    pImmediateContext->VSSetConstantBuffers( 0, 1, &pConstantBuffer );

    // Activer le PS
    pImmediateContext->PSSetShader(pPixelShader, nullptr, 0); 8

    // **** Rendu de l'objet
    pImmediateContext->DrawIndexed( ARRSIZE(index_bloc), 0, 0 ); 9
}

```

Explications

1

Nous obtenons le contexte à partir de notre classe de dispositif (où il est implanté). J'ai choisi ici d'utiliser un « Get » parce qu'en théorie le contexte pourrait changer. En pratique, c'est plutôt rare pour un contexte immédiat. Vous pourriez donc prendre en note le contexte dans le constructeur de CBloc, ou de vos autres objets -> petite optimisation...

Les étapes suivantes sont très importantes dans votre compréhension du rendu d'une primitive avec **DrawIndexed**. Dans les exemples fournis avec DirectX et dans plusieurs exemples sur le Net, ces étapes ne sont pas faites à chaque rendu, mais une seule fois dans l'application. Ces exemples n'ont habituellement qu'un seul objet, mais nous construisons un moteur qui pourra en afficher plusieurs...

Topologie des primitives

2 Nous devons indiquer au pipeline quel type de primitives il devra traiter, donc leur topologie. Dans notre cas le mot clé est D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST.

Choix du vertex buffer

Nous devons indiquer quel tampon de sommets utiliser.

Choix de l'index buffer

Puisque nous utilisons DrawIndexed, nous devons indiquer quel tampon d'index utiliser. Nous devons aussi indiquer la taille d'un index. Les indices peuvent être soit DXGI_FORMAT_R16_UINT ou DXGI_FORMAT_R32_UINT. S'il l'objet à dessiner contient moins de 65536 sommets, il est plus performant d'utiliser le format DXGI_FORMAT_R16_UINT.

Format des sommets - Vertex Layout

5 Nous devons spécifier au pipeline l'organisation (le format) de nos sommets.

Choix du vertex shader

Nous devons indiquer quel vertex shader utiliser.

Constantes des shaders

7 Nous devons initialiser/modifier les constantes de nos shaders. N'oubliez pas qu'il s'agit de variables pour le « côté CPU » de notre application et qu'il s'agit de constantes pour le « côté GPU ». Dans notre cas, une seule variable soit la matrice « totale » **matWorldViewProj** qui est recalculée en multipliant la matrice matWorld de l'objet par la matrice viewProj du moteur. J'ai utilisé ici le fait que le moteur soit un singleton pour accéder à la matrice viewProj. Vous trouverez peut-être d'autres solutions plus élégantes. La fonction **IDeviceContext::UpdateSubresource** permet de modifier le tampon des constantes et la fonction **IDeviceContext::VSSetConstantBuffers** permet de choisir notre tampon de constante pour le VS.

Choix du pixel shader

Nous devons indiquer quel pixel shader utiliser.

Et finalement, nous pouvons donner l'ordre au pipeline de faire le rendu avec la fonction **IDeviceContext::DrawIndexed**.

```
// **** Rendu de l'objet  
pImmediateContext->DrawIndexed(ARRAYSIZE(index_bloc) 0, 0 );
```

9

Ses paramètres sont assez simples, le premier (36) nous indique combien d'entrées d'index seront traitées. Le deuxième nous indique que nous commençons au début du tampon d'index (déplacement de 0). Le troisième

nous indique que nos indices sont à partir du début du tampon des sommets (déplacement de 0). Pour ces deux derniers paramètres, c'est ce que nous utiliserons presque toujours, mais vous pourriez vous en servir pour dessiner partiellement un objet, y planter une méthode de niveaux de détails, ou...

4.12 Faire le ménage

Lorsque notre application se termine ou dans des situations particulières, nous devrons relâcher les objets DirectX utilisés par notre application. Nous utiliserons ici le destructeur de CMoteur et la fonction CMoteur::Cleanup.

1. Modifiez la fonction **CMoteur::Cleanup**. Ajoutez y les lignes suivantes (en encadré):

```
virtual void Cleanup()
{
    // détruire les objets
    ListeScene.clear();

    // Détruire le dispositif
    if (pDispositif)
    {
        delete pDispositif;
        pDispositif = nullptr;
    }
}
```

2. Modifiez le destructeur de CBloc pour que les objets DirectX soient « relâchés » lors de la destruction d'un bloc. Dans notre cas, l'ordre de « relâchement » des objets COM a peu d'importance, mais dans certains cas, certains objets seront construits « par-dessus » d'autres donc il est recommandé de relâcher ou de détruire les objets dans l'ordre inverse de leur création.

```
CBloc::~CBloc()
{
    DXRelacher(pPixelShader);
    DXRelacher(pConstantBuffer);
    DXRelacher(pVertexLayout);
    DXRelacher(pVertexShader);
    DXRelacher(pIndexBuffer);
    DXRelacher(pVertexBuffer);
}
```

La fonction **DXRelacher** appelle la fonction **Release** de l'objet COM concerné et met ensuite le pointeur à 0.

La fonction **Release** décrémente le compteur d'utilisation de l'objet COM. Lorsque le compteur est à 0, l'objet est réellement retiré de la mémoire CPU et/ou GPU selon le cas.

3. Modifier la déclaration du destructeur de CBloc dans le fichier bloc.h

```
virtual ~CBloc();
```

Notre programme est maintenant prêt à être compilé et exécuté. Bien sûr, le cube n'est pas très réaliste puisque l'éclairage est uniforme, mais nous avons mis en place tous les éléments de base.

5. Contrôle de l'affichage

Dans ce chapitre, nous réviserons l'initialisation du dispositif Direct3D afin d'obtenir une application dont l'affichage de départ sera plein-écran, mais elle pourra être alternée avec le mode fenêtré.

Jusqu'à maintenant, nous nous sommes contentés du dispositif de défaut avec des initialisations de base plutôt simples. Avec DirectX 11, c'est une bonne façon de procéder, mais nous explorerons un peu plus les possibilités d'interrogation du système pour valider nos choix ou pour offrir des choix à l'utilisateur.

Objectifs du chapitre

- ✓ Informations sur un adaptateur spécifique.
- ✓ Obtenir un mode d'affichage intéressant.
- ✓ Déterminer les dimensions d'affichage.
- ✓ Affichage « plein écran »/« fenêtré » sur demande.
- ✓ Corriger la projection.
- ✓ Implanter un tri de profondeur Z

5. Contrôle de l'affichage

Théorie

5.1 L'adaptateur graphique

Pour les besoins du présent document, les deux termes (**carte graphique** et **adaptateur graphique**) seront synonymes. Dans les programmes, nous favoriserons le terme adaptateur puisqu'il ressemble au terme anglais **adapter**.

Définition « officielle »

Adaptateur graphique est la traduction directe de **graphic adapter**, terme correspondant à **carte graphique** dans les textes français. Un adaptateur permet à un ordinateur d'effectuer des sorties graphiques sur un écran.

En pratique, c'est un peu plus compliqué que cela puisque:

- certaines cartes graphiques offrent deux adaptateurs (ou 4 ou plus). Dans certains cas, ces adaptateurs peuvent être configurés séparément;
- certains adaptateurs offrent deux sorties (ou plus);
- certains ordinateurs ont plusieurs cartes graphiques, parfois de marques différentes;
- plusieurs ordinateurs ayant deux cartes graphiques ont des cartes graphiques conçues pour travailler de façon « conjointe » ou « collaborative » sur un même écran. C'est le cas de la technologie **Optimus** de NVidia présente sur plusieurs ordinateurs portables (*laptop*) ou de l'équivalent chez AMD: **Dynamic Switchable Graphics**. On appellera cette combinaison de cartes un **système hybride**.
- Nous pourrons avec DirectX, définir des adaptateurs virtuels, nullement associés à des cartes graphiques;
- il existe d'autres configurations dont nous n'entendrons parler que lorsque nos utilisateurs nous en feront part...

Il nous faudrait donc, dans une application plus importante, interroger le système plus en détail et proposer des choix à l'utilisateur. Pourquoi des choix ? Parce qu'il est très difficile de déterminer de façon logicielle la meilleure configuration et ensuite parce que l'utilisateur aime avoir le choix et au besoin choisir une configuration moins belle, mais plus performante (ou le contraire).

Différence entre adaptateur et dispositif

L'**adaptateur** correspond au matériel disponible et le **dispositif** (*device*) correspond à une combinaison « carte graphique »/« pilote »/« mode d'affichage »/« paramètres ».

5.2 Obtenir l'information sur un adaptateur

Pour obtenir de l'information sur les adaptateurs et plus tard pour valider les modes et paramètres que nous voudrons avoir pour notre dispositif (en 5.5), nous implanterons de façon très sommaire une petite classe de travail: la classe **CInfoDispositif** qui comme son nom l'indique nous permettra de collecter de l'information sur tout ce que nous aurons besoin pour la création d'un dispositif.

1. Implantez la classe CInfoDispositif dans les fichiers **InfoDispositif.h** et **InfoDispositif.cpp**.

InfoDispositif.h

```
#pragma once

namespace PM3D
{

class CInfoDispositif
{
public:

    explicit CInfoDispositif(int NoAdaptateur);
}
}
```

Note:

Il est souvent utile de s'informer sur l'adaptateur courant.

InfoDispositif.cpp

```
#include "StdAfx.h"
#include "InfoDispositif.h"

namespace PM3D
{
```

2. Déclarez un constructeur paramétré pour la classe CInfoDispositif, le paramètre sera le numéro de l'adaptateur.

```
CInfoDispositif( int NoAdaptateur );
```

3. Pour plus de facilité, j'ai aussi déclaré dans le fichier InfoDispositif.h l'énumération suivante:

```
enum INFODISPO_TYPE
{
    ADAPTATEUR_COURANT
};
```

L'adaptateur 0 est toujours **l'adaptateur courant**.

- Déclarez un certain nombre de variables membres qui seront utiles pour prendre en note des informations sur l'adaptateur, les modes, etc. Déclarez-les dans la section *private*:

```
bool valide;
int largeur;
int hauteur;
int memoire;
wchar_t nomcarte[100];
```

Plusieurs autres informations sont disponibles via les structures de données **DXGI_OUTPUT_DESC**, **DXGI_ADAPTER_DESC** et **DXGI_MODE_DESC**, mais nous nous contenterons pour l'instant de ces quatre informations ainsi que d'un indicateur, **valide**, qui nous permettra de valider le numéro d'adaptateur.

- Déclarez et implantez les accesseurs pour les variables précédentes. Déclarez-les, évidemment, dans la section *public*:

```
bool EstValide() const {return valide;}
int GetLargeur() const {return largeur;}
int GetHauteur() const {return hauteur;}
int GetMemoire() const {return memoire;}
const wchar_t* GetNomCarte() const { return nomcarte; }
```

- Insérez ensuite les deux lignes suivantes à la suite des « #include » dans le fichier InfoDispositif.cpp. Nous allons utiliser certaines fonctions de UtilitairesDX:

```
#include "util.h"
using namespace UtilitairesDX;
```

- Implantez ensuite le code suivant:

```
// Pour obtenir les informations à partir d'un numéro d'adaptateur
// 0 = courant = ADAPTATEUR_COURANT
CInfoDispositif::CInfoDispositif( int NoAdaptateur )
{
    IDXGIFactory* pFactory = nullptr;
    IDXGIAdapter* pAdapter = nullptr;
    IDXGIOutput* pOutput = nullptr;

    valide = false;

    // Créer un IDXGIFactory.
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&pFactory);
```

1

```

if (FAILED(pFactory->EnumAdapters(NoAdaptateur, &pAdapter))) return; ← 2

// Obtenir les informations de la première sortie de l'adaptateur
// (Le moniteur)
if (FAILED(pAdapter->EnumOutputs(0, &pOutput))) return; ← 3

// Obtenir la description de l'état courant
DXGI_OUTPUT_DESC outDesc;
pOutput->GetDesc(&outDesc); ← 4

largeur = outDesc.DesktopCoordinates.right - outDesc.DesktopCoordinates.left;
hauteur = outDesc.DesktopCoordinates.bottom - outDesc.DesktopCoordinates.top;
valide = true;

DXGI_ADAPTER_DESC Desc;
pAdapter->GetDesc(&Desc); ← 5

// Mémoire dédiée (en megabytes).
memoire = (int)(Desc.DedicatedVideoMemory / 1024 / 1024);

// Nom de la carte video.
wcscpy_s(nomcarte, 100, Desc.Description);

// Faire le ménage pour éviter les « memory leaks »
DXRelacher(pOutput);
DXRelacher(pAdapter);
DXRelacher(pFactory);
}

```

Explications

1

L'interrogation des capacités graphiques se fait avec un objet de classe **IDXGIFactory**. DXGI est depuis DirectX 10 le moyen privilégié de s'informer sur certaines capacités du système. DXGI offre aussi d'autres possibilités. Voir « **DXGI Overview** » dans la documentation du SDK pour plus de détails.

2

Nous obtenons un pointeur sur un objet **IDXGIAdapter** (qui représentera notre adaptateur). Notez l'utilisation de la macro « **FAILED** » qui terminera la fonction. L'indicateur **valide** sera à *false*. La macro FAILED n'est pas aussi bien qu'un *template* mais comme elle est souvent présente dans les exemples de DirectX, j'ai choisi de l'utiliser quand même.

3

Nous obtenons ensuite un pointeur sur un objet **IDXGIOoutput** qui représentera la sortie principale de notre adaptateur, habituellement le moniteur. Nous pourrions obtenir de l'information sur d'autres sorties (par exemple, une sortie branchée à un projecteur). Nous nous servirons dans la prochaine section de cet objet pour valider ou choisir un mode d'affichage.

4

Nous obtenons une structure de données **DXGI_OUTPUT_DESC** qui comme son nom l'indique contient des informations sur la sortie. Dans notre cas, nous nous contenterons des dimensions.

5

Nous obtenons une nouvelle structure de données de type **DXGI_ADAPTER_DESC** qui contient des informations sur

l'adaptateur. Nous y noterons le nom de l'adaptateur et la quantité de mémoire vidéo dédiée.

Types de mémoire vidéo

Nous retrouverons trois types de « mémoire vidéo », les deux principales étant la **mémoire dédiée** et la **mémoire partagée**. Il pourra y avoir un impact sur les performances selon l'emplacement choisi pour nos ressources.

Mémoire vidéo dédiée (*Dedicated Video Memory*)

Il s'agit de la quantité de mémoire vidéo non partagée avec le CPU.

Elle est habituellement directement implantée sur la carte graphique et elle est contrôlée uniquement par le GPU.

Mémoire système partagée (*Shared System Memory*)

C'était au départ une solution économique pour les « petits » systèmes informatiques (par exemple les systèmes à base de Celeron d'il y a quelques années). Les cartes graphiques coûtaient moins cher puisqu'elles n'avaient pas (ou très peu) de mémoire. La mémoire étant prélevée sur celle du système.

Même si l'accès et le travail avec cette mémoire sont généralement moins rapides pour le GPU que la mémoire dédiée, certains ont eu la surprise, il y a quelques années, d'observer que certaines applications étaient plus rapides avec la mémoire partagée qu'avec la mémoire dédiée. Il s'agissait d'applications nécessitant beaucoup d'échanges entre les données du CPU et celles du GPU. C'est pourquoi la plupart des nouveaux systèmes disposent des deux types de mémoire.

Mémoire système dédiée (*Dedicated System Memory*)

Comme son nom l'indique, il s'agit de mémoire système non partagée. Cette mémoire est généralement réservée par le pilote de périphérique lors de son démarrage pour assurer un minimum de mémoire aux cartes graphiques. Elle n'est plus disponible pour le système par la suite, mais demeure disponible pour le GPU. Généralement, cette mémoire est à 0 sur les systèmes disposant de mémoire vidéo dédiée.

8. Insérez la ligne suivante à la suite des autres « **#include** » dans le fichier DispositifD3D11.cpp :

```
#include "InfoDispositif.h"
```

9. Insérez maintenant la ligne suivante dans le constructeur de CDispositifD3D11 juste avant l'énoncé **switch**:

```
...
ZeroMemory( &sd, sizeof(sd) );

// Obtenir les informations de l'adaptateur courant
CInfoDispositif Dispo0( ADAPTATEUR_COURANT );

switch (cdsMode)
{
    ...
}
```

Nous n'utiliserons pas Dispo0 pour l'instant, mais ses informations demeureront disponibles au cas où...

10. Dans le dialogue **Pages de propriétés** de PetitMoteur3D sous la rubrique **Éditeur de liens|Entrée**, ajoutez la bibliothèque **DXGI.lib** dans le champ **Dépendances supplémentaires**.
11. Vous pouvez générer votre application pour vérifier qu'il ne manque rien.

Théorie

5.3 Déterminer les dimensions d'affichage

L'époque où les meilleurs choix comme résolution d'écran (sur PC) étaient 640 X 480 ou 800 X 600 est maintenant loin derrière nous. La plupart des cartes graphiques offrent maintenant des résolutions beaucoup plus élevées soient 1024 X 768, 1280 X 1024 sans compter toutes les résolutions « exotiques » offertes par les nouveaux écrans (soit sur portable ou sur ordinateur de bureau). Par exemple, l'ordinateur portable sur lequel je travaille offre une résolution de 1366 X 768.

Vous pouvez choisir une résolution que vous jugez « intéressante », mais vous devez tout de même tenir compte de la disponibilité de cette résolution sur le système cible ainsi que de la mémoire vidéo requise par votre application (dédiée et/ou partagée). Heureusement, comme nous le verrons à la prochaine section, Direct3D nous offre de bonnes fonctionnalités pour interroger le système et aussi certaines options « passe-partout » dont nous profiterons pour les besoins de PetitMoteur3D.

Pour vous éclairer dans votre choix, voici un tableau des principales résolutions et des caractéristiques graphiques qui les accompagnent:

Largeur	Hauteur	Pixels	Écran Large	MégaOctets (1)	MégaOctets (2)
800	600	480000	Non	1,8	5,5
1024	600	614400	Oui ?	2,3	7,0
1024	768	786432	Non	3,0	9,0
1152	864	995328	Non	3,8	11,4
1280	800	1024000	Oui	3,9	11,7
1366	768	1049088	Oui	4,0	12,0
1440	900	1296000	Oui	4,9	14,8
1280	1024	1310720	Non	5,0	15,0
1600	900	1440000	Oui	5,5	16,5
1680	1050	1764000	Oui	6,7	20,2
1600	1200	1920000	Non	7,3	22,0
1920	1080	2073600	Oui	7,9	23,7
1920	1200	2304000	Oui	8,8	26,4

- (1) Nombre de Mo pour l'affichage seulement avec un format de pixels de 32 bits (4 octets)
- (2) Nombre de Mo pour tampons d'avant-plan, d'arrière-plan et de tri de profondeur Z avec un format de pixels de 32 bits (4 octets) pour l'affichage et aussi 32 bits pour le tampon Z.

Vous pouvez utiliser l'utilitaire DirectX DXCaps pour vérifier vous-même la disponibilité d'une résolution, mais en programmation, nous disposerons d'une ou deux astuces supplémentaires.

Vous constaterez qu'une résolution de 1366 X 768 demandera 12 mégaoctets. Même si ce nombre peut sembler petit en rapport avec les capacités mémoires des cartes graphiques modernes (souvent plus de 1 GO de mémoire dédiée), il n'est pas négligeable.

De plus, la résolution aura un impact direct sur la performance. Par exemple, pour une scène bien remplie, un affichage de 1366 X 768 effectuera environ 2 fois plus d'appels de *pixel shaders* qu'une résolution de 800 X 600.

Nos choix

Pour une application commerciale et particulièrement pour un jeu vidéo, il sera bon de sélectionner une résolution « optimale » comme point de départ et de proposer des choix à l'utilisateur par la suite. Heureusement pour nous, les nouvelles versions de DirectX nous offrent de meilleurs outils pour choisir et gérer les résolutions.

Pour les besoins de ce manuel, je choisirai une résolution de 1024 X 768. Ce n'est pas un affichage « écran large », mais pour nos besoins, qui impliqueront beaucoup d'alternance entre le mode « plein écran » et le mode « fenêtré », cela sera suffisant. Libre à vous de choisir une autre résolution si votre carte graphique n'implémente pas correctement celle-ci ou tout simplement si vous aimez mieux une résolution « écran large ».

Note

Avec DX 11, les cartes graphiques plus récentes implémentent les modes non compatibles avec votre résolution d'écran (mais supportés) en conservant un ratio de pixels de 1/1 et en complétant l'écran avec des pixels noirs (l'affichage effectif est centré).

Les cartes plus anciennes « étirent » souvent la résolution sur votre écran avec comme résultat un ratio d'aspect de pixels différent de 1/1.

Pour les applications « pré DX 10 », les résultats seront variables selon la carte et les pilotes (certaines avec ratio 1/1, d'autres « étirées »).

Intéressant aussi, avec DX 11, si une résolution n'est pas supportée, un choix « éclairé » automatique est fait pour obtenir la résolution la plus près. Par exemple, pour mon ordinateur portable, si je choisis une résolution de 1440 X 900 (non supportée), le pilote DX choisira 1366 X 768 comme résolution effective. Autrefois, la création du dispositif échouait...

Merveilleux ? Oui et non. Si votre application continue d'utiliser la résolution désirée au lieu de la résolution effective, il en résultera peut-être des déformations lors du rendu. Par exemple, 1440 X 900 nous donne un ratio de projection de 1,60 alors que 1366 X 768 nous donnerait 1,78. Négligeable ? Pas vraiment, vous trouverez que vos personnages semblent avoir pris du poids. 

Nous pourrons (nous devrons...) interroger DX sur cette résolution effective et en tenir compte pour notre projection. C'est ce que nous ferons dans la prochaine section (5.4).

Atelier

5.4 Obtenir l'affichage le plus intéressant

Dans cette section, nous avons deux tâches à accomplir soient:

- Écrire un nouveau constructeur pour `CInfoDispositif`, ce constructeur nous permettra d'obtenir de l'information sur le mode d'affichage disponible le plus près de nos spécifications.
- Écrire le code nécessaire dans `CDispositifD3D11` pour créer les objets du dispositif en fonction des informations effectives et pour démarrer l'application en mode plein écran.

Nouveau constructeur pour `CInfoDispositif`

1. Déclarez et implémentez le nouveau constructeur:

Dans `InfoDispositif.h`

```
CInfoDispositif(DXGI_MODE_DESC modeDesc );
```

Dans `InfoDispositif.cpp`

```
CInfoDispositif::CInfoDispositif(DXGI_MODE_DESC modeDesc )
{
}
```

modeDesc est une structure de donnée `DXGI_MODE_DESC` qui contient des informations sur le mode d'affichage soit:

- **Width** et **Height**: la **largeur** et la **hauteur**;
- **RefreshRate**: le **taux de rafraîchissement**. Nous sommes habitués à utiliser des valeurs entières comme taux de rafraîchissement, par exemple 60 Hz. Alors qu'en réalité un taux de « 60 Hz » correspond approximativement à $60,000 / 1,001$ Hz. En mode plein écran, si le taux de rafraîchissement réel ne correspond pas à celui que nous avons spécifié. DXGI effectuera un blit (une copie) au lieu d'un flip donc nous aurons une perte de performance. Le taux de rafraîchissement comprend un **numérateur** et un **dénominateur**;
- **Format**: le **format de pixel** désiré;
- **ScanlineOrdering**: l'**ordonnancement des scanlines**. Habituellement nous utiliserons 0 pour laisser le système choisir.

- **Scaling:** le mode de mise à l'échelle de notre affichage sur l'écran. Voir la note en fin de section 5.3.

2. Ajoutez le code suivant dans le constructeur:

```
// Énumération des adaptateurs
IDXGIFactory * pFactory = nullptr;
CreateDXGIFactory(__uuidof(IDXGIFactory) , (void**)&pFactory); 1

IDXGIAdapter * pAdapter;
std::vector <IDXGIAdapter*> vAdapters;

for ( UINT i = 0;
      pFactory->EnumAdapters(i, &pAdapter) != DXGI_ERROR_NOT_FOUND; ++i )
{
    vAdapters.push_back(pAdapter); 2
}

// On travaille presque toujours avec vAdapters[0]
// à moins d'avoir plusieurs cartes non-hybrides
*this = CInfoDispositif(0); 3

// Obtenir la sortie 0 - le moniteur principal
IDXGIOOutput* pOutput = nullptr;
vAdapters[0]->EnumOutputs(0,&pOutput); 4

// Obtenir le mode le plus intéressant
pOutput->FindClosestMatchingMode( &modeDesc, &mode, nullptr ); 5

// Faire le ménage pour éviter les « memory leaks »
DXRelacher(pOutput);

for (int i=0; i<vAdapters.size(); ++i)
{
    DXRelacher(vAdapters[i]);
}

DXRelacher (pFactory);
```

Explications

1

L'interrogation des capacités graphiques se fait avec un objet de classe **IDXGIFactory**. DXGI est depuis DirectX 10 le moyen privilégié de s'informer sur certaines capacités du système. DXGI offre aussi d'autres possibilités. Voir « **DXGI Overview** » dans la documentation du SDK pour plus de détails.

2

Nous obtenons les pointeurs sur les objets **IDXGIAdapter** (qui représentent les adaptateurs disponibles sur notre système). Nous prenons en note ces objets dans un vecteur. Même si nous utiliserons principalement l'adaptateur 0, nous implémenterons quand même l'énumération des adaptateurs, pour une prochaine réincarnation de **CInfoDispositif** peut-être ?

3

Nous travaillerons avec l'adaptateur 0. Lorsque nous initialiserons le dispositif, le contexte et la chaîne d'échange, nous ferons tout afin de « déclencher » la collaboration et d'utiliser l'adaptateur le plus avancé.

Nous utilisons le constructeur précédent, celui avec un numéro pour initialiser les variables membres présentées en 5.2. L'utilisation du « ***this** » dans un constructeur est discutable, mais...

4

Nous travaillons avec le moniteur principal (*output 0*). Nous aurions pu énumérer les moniteurs, mais pour les besoins de notre application, il n'y en a qu'un seul.

5

La fonction **IDXGIOOutput::FindClosestMatchingMode** est appelée. Cette fonction nous permet de remplir une nouvelle structure **DXGI_MODE_DESC** avec les informations trouvées. En programmation « pré DX10 », nous aurions eu besoin de plusieurs dizaines de lignes de code pour vérifier et trouver ces informations.

3. Déclarez la variable mode dans la section *private* de CInfoDispositif:

```
DXGI_MODE_DESC mode;
```

4. Déclarez l'accesseur suivant dans la section *public* de CInfoDispositif:

```
void GetDesc(DXGI_MODE_DESC& modeDesc){modeDesc = mode;}
```

Nouveaux paramètres pour les objets du dispositif

Ici, nous remplirons la section correspondant au cas « plein écran ». Nous verrons dans la section suivante (5.5) comment réagir à l'alternance « plein écran »/« fenêtré » de notre application.

1. Modifiez la ligne suivante dans la fonction CMoteur::Initialisations

```
// * Initialisation du dispositif de rendu  
pDispositif = CreationDispositifSpecific( CDS_PLEIN_ECRAN );  
au lieu de CDS_FENETRE
```

2. Nous remplissons une structure **DXGI_MODE_DESC**. Ces lignes sont insérées dans le constructeur paramétré de **CDispositifD3D11** à la suite de l'énoncé « `case CDS_PLEIN_ECRAN:` » que nous ajoutons dans le « `switch cdsMode` »

```

case CDS_PLEIN_ECRAN:
    largeur = 1024;
    hauteur = 768;

    DXGI_MODE_DESC desc;
    desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    desc.Height = hauteur;
    desc.Width = largeur;
    desc.RefreshRate.Numerator = 60;
    desc.RefreshRate.Denominator = 1;
    desc.ScanlineOrdering = DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
    desc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;

```

3. Puis nous créons un objet CInfoDispositif pour obtenir les informations sur le mode le plus conforme à nos spécifications (à la suite...).

```

CInfoDispositif DispoVoulu(desc);
DispoVoulu.GetDesc(desc);

```

4. Nous pouvons maintenant initialiser la structure **sd** (une structure de type DXGI_SWAP_CHAIN_DESC) avec les informations obtenues, nous en profitons pour modifier les variables **largeur** et **hauteur**.

```

largeur = desc.Width;
hauteur = desc.Height;

sd.BufferCount = 1;
sd.BufferDesc.Width = desc.Width;
sd.BufferDesc.Height = desc.Height;
sd.BufferDesc.Format = desc.Format;
sd.BufferDesc.RefreshRate.Numerator = desc.RefreshRate.Numerator;
sd.BufferDesc.RefreshRate.Denominator = desc.RefreshRate.Denominator;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = FALSE;           // Plein écran   NOTEZ!!!

```

Comparez avec l'initialisation utilisée pour le cas CDS_FENETRE. Beaucoup d'éléments sont presque les mêmes, les autres sont en fonction de nos nouvelles informations.

5. Vous pouvez générer et exécuter l'application, elle sera en plein écran et sur la plupart des cartes graphiques les pixels horizontaux seront étirés si vous avez un écran large. Nous corrigerais ça en 5.5.

Atelier

5.5 Affichage « plein écran »/« fenêtré » sur demande.

En premier lieu, la partie facile. DirectX étant intégré dans Windows (depuis Vista et DX10). L'utilisation de la combinaison de touches Alt-Retour permet de passer d'un mode à l'autre. Essayez.

Mais, remarquez que:

- le mode plein écran est étiré, donc: le ratio de pixel n'est pas 1/1;
- lorsqu'on passe en mode fenêtré, le ratio de pixel est 1/1, mais le format d'écran ne correspond pas réellement à l'espace que nous avons réservé pour nos surfaces de rendu (tampons avant et arrière). Il pourrait y avoir des pertes d'affichage (lorsque nous aurons autre chose qu'un cube...).
- Faire les ajustements avec le ratio d'aspect de la projection (voir section 5.6) sera difficile et ne réglera pas tous les problèmes.

1. Régler une partie de ces problèmes est relativement facile, il suffit d'ajouter la ligne suivante à notre initialisation de la structure **sd**:

```
// Permettre l'échange plein écran
sd.Flags = DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH;
```

Notez que l'échange était déjà possible, mais maintenant, presque tous les problèmes mentionnés plus haut sont réglés.

2. La documentation de DirectX mentionne un problème avec certaines cartes graphiques lorsque l'application se termine en mode plein écran. Je n'ai jamais rencontré ce problème, mais il est facile à résoudre: il suffit d'ajouter la ligne suivante au début du destructeur de CDispositifD3D11:

```
pSwapChain->SetFullscreenState(FALSE, NULL); // passer en mode fenêtré
```

Cette instruction fait passer l'application en mode fenêtré.

3. Le deuxième problème (passage au mode fenêtré) n'est pas complètement réglé, mais il est relativement facile à régler pour qui connaît un peu la programmation Windows. Insérez les lignes suivantes avant l'énoncé D3D11CreateDeviceAndSwapChain :

```

// régler le problème no 1 du passage en mode fenêtré
RECT rcClient, rcWindow;
POINT ptDiff;
GetClientRect(hWnd, &rcClient);
GetWindowRect(hWnd, &rcWindow);
ptDiff.x = (rcWindow.right - rcWindow.left) - rcClient.right;
ptDiff.y = (rcWindow.bottom - rcWindow.top) - rcClient.bottom;
MoveWindow( hWnd, rcWindow.left, rcWindow.top,
            largeur + ptDiff.x, hauteur + ptDiff.y, TRUE);

```

Je n'explique pas ces lignes ici puisque ce n'est pas le sujet. Ceux intéressés à en savoir plus pourront consulter la documentation en ligne de l'API de Windows ou [Petzold PW].

Un dernier problème subsiste: si l'utilisateur modifie les dimensions de la fenêtre en mode fenêtré. Il en résultera des déformations. Nous n'implanterons pas de solution à ce problème pour le moment, mais sachez qu'il existe plusieurs façons de contourner ou de régler ce problème:

- l'application peut **refuser** de passer en mode fenêtré. C'est ce que plusieurs jeux font. Pour nous, ce n'est pas une solution (en tout cas pas tout de suite), car nous voulons conserver le mode fenêtré pour fins de mise au point du programme;
- nous pourrions modifier les attributs de la fenêtre pour que l'utilisateur **ne puisse pas la redimensionner**. Voir l'API de Windows à ce sujet.
- **LA SOLUTION LONGUE** (et pénible...): détecter la modification de dimension (message WM_SIZE), calculer la nouvelle dimension, redimensionner les tampons avant et arrière ainsi que la vue de surface d'affichage (IDXGISwapChain::ResizeBuffers et IDXGISwapChain::ResizeTarget) puis ajuster le ratio d'aspect dans la matrice de projection. Et surtout, ne pas oublier de tout remettre en place si nous revenons en mode plein-écran.

Nouvelle version du constructeur de CDispositifD3D11

Les ajustements dont nous venons de parler se reflètent dans le constructeur de CDispositifD3D11, en voici donc la nouvelle version :

```

CDispositifD3D11::CDispositifD3D11(const CDS_MODE cdsMode,
                                     const HWND hWnd)
{
    UINT largeur;
    UINT hauteur;
    UINT createDeviceFlags = 0;

    #ifdef _DEBUG
        createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
    #endif
}

```

```

#endif

D3D_FEATURE_LEVEL featureLevels[] =
{
    D3D_FEATURE_LEVEL_11_0,
    D3D_FEATURE_LEVEL_10_1,
    D3D_FEATURE_LEVEL_10_0,
};

UINT numFeatureLevels = ARRAYSIZE( featureLevels );

pImmediateContext = NULL;
pSwapChain = NULL;
pRenderTargetView = NULL;

DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );

// Obtenir les informations de l'adaptateur de défaut
CInfoDispositif Dispo0( ADAPTATEUR_COURANT );

largeur = 1024;
hauteur = 768;

switch (cdsMode)
{
case CDS_FENETRE:
    sd.Windowed = TRUE;

    break;

case CDS_PLEIN_ECRAN:
    sd.Windowed = FALSE;
    break;

}

DXGI_MODE_DESC desc;
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.Height = hauteur;
desc.Width = largeur;
desc.RefreshRate.Numerator = 60;
desc.RefreshRate.Denominator = 1;
desc.ScanlineOrdering = DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
desc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;

CInfoDispositif DispoVoulu(desc);
DispoVoulu.GetDesc(desc);

largeur = desc.Width;
hauteur = desc.Height;

sd.BufferCount = 1;
sd.BufferDesc.Width = desc.Width;
sd.BufferDesc.Height = desc.Height;
sd.BufferDesc.Format = desc.Format;
sd.BufferDesc.RefreshRate.Numerator = desc.RefreshRate.Numerator;
sd.BufferDesc.RefreshRate.Denominator = desc.RefreshRate.Denominator;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;

```

```

// Permettre l'échange plein écran
sd.Flags = DXGI_SWAP_CHAIN_FLAG_ALLOW_MODE_SWITCH;

// régler le problème no 1 du passage en mode fenêtré
RECT rcClient, rcWindow;
POINT ptDiff;
GetClientRect(hWnd, &rcClient);
GetWindowRect(hWnd, &rcWindow);
ptDiff.x = (rcWindow.right - rcWindow.left) - rcClient.right;
ptDiff.y = (rcWindow.bottom - rcWindow.top) - rcClient.bottom;
MoveWindow(hWnd, rcWindow.left, rcWindow.top, largeur + ptDiff.x,
           hauteur + ptDiff.y, TRUE);

DXEssayer( D3D11CreateDeviceAndSwapChain( 0,
                                             D3D_DRIVER_TYPE_HARDWARE,
                                             NULL,
                                             createDeviceFlags,
                                             featureLevels, numFeatureLevels,
                                             D3D11_SDK_VERSION,
                                             &sd,
                                             &pSwapChain,
                                             &pD3DDevice,
                                             NULL,
                                             &pImmediateContext ),
            DXE_ERREURCREATIONDEVICE) ;

// Crédit d'un « render target view »
ID3D11Texture2D *pBackBuffer;
DXEssayer( pSwapChain->GetBuffer( 0, __uuidof( ID3D11Texture2D ),
(LPVOID*)&pBackBuffer ), DXE_ERREUROBTENTIONBUFFER ) ;

DXEssayer(pD3DDevice->CreateRenderTargetView( pBackBuffer, NULL,
&pRenderTargetView ), DXE_ERREURCREATIONRENDERTARGET);
pBackBuffer->Release();

pImmediateContext->OMSetRenderTargets( 1, &pRenderTargetView, 0 );

D3D11_VIEWPORT vp;
vp.Width = (FLOAT)largeur;
vp.Height = (FLOAT)hauteur;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
pImmediateContext->RSSetViewports( 1, &vp );

}

```

Atelier

5.6 Corriger la projection

Vous avez sans doute remarqué que notre « cube » est légèrement plus large que haut (exactement 1,33 fois ! Ou 1024/768... Donc ce n'est pas un cube). La raison en est simple, l'écran théorique où nous projetons nos primitives a des dimensions de -1 à 1 sur l'axe des X et de même pour l'axe des Y. C'est donc un écran « carré ».

Nous pouvons tenir compte de notre ratio d'aspect lors de l'initialisation de la matrice de projection. Au chapitre 4, dans la fonction CMoteur::InitScene nous avions initialisé cette matrice ainsi:

```
float champDeVision = XM_PI/4;    // 45 degrés
float ratioDAspect = 1.0;          // horrible, il faudra corriger ça
float planRapproche = 2.0;
float planEloigne = 20.0;

matProj = XMMatrixPerspectiveFovLH(
            champDeVision,
            ratioDAspect,
            planRapproche,
            planEloigne );
```

Nous avions un ratio d'aspect de 1.0 (et le commentaire qui allait avec...).

Nous effectuerons donc quelques petites modifications pour utiliser le bon ratio.

- D'abord dans la classe **CDispositif**, déclarez les variables suivantes dans la section *protected*:

```
uint32_t largeurEcran;
uint32_t hauteurEcran;
```

Pourquoi de type float ? Pour alléger les notations lorsque nous utiliserons ces variables lors des calculs (qui seront presque toujours en float).

- Puis dans le constructeur de **CDispositifD3D11**, ajoutez deux lignes après la récupération des valeurs de **largeur** et de **hauteur**:

```
largeur = desc.Width;
hauteur = desc.Height;
```

```
largeurEcran = largeur;
hauteurEcran = hauteur;
```

3. Dans la classe CDispositif, déclarez les deux accesseurs suivants (évidemment publics):

```
uint32_t GetLargeur() const {return largeurEcran;}  
uint32_t GetHauteur() const {return hauteurEcran;}
```

4. Et finalement corriger le calcul du ratio d'aspect dans la fonction CMoteur::InitScene:

```
const float ratioDAspect = static_cast<float>(pDispositif->GetLargeur())  
/ static_cast<float>(pDispositif->GetHauteur());
```

Essayez votre programme, le cube devrait cette fois être un « vrai cube ».

Théorie

5.7 Les tris de profondeur

Comme nous en avons déjà parlé, la méthode la plus utilisée aujourd’hui pour résoudre le problème des surfaces cachées est l’utilisation d’un **tampon de profondeur** (depth-buffer). Un tampon de profondeur, appelé souvent **z-buffer** ou **w-buffer** selon l’algorithme choisi, est un tableau qui contiendra de l’information sur la « profondeur » des pixels affichés sur la surface de rendu. Avec Direct3D, il s’agit d’une surface semblable aux surfaces de rendu c’est-à-dire une texture. Lorsque Direct3D effectue le rendu d’une scène 3D sur une surface, il peut alors utiliser le tampon de profondeur pour déterminer comment les pixels d’un polygone cacheront ceux d’un autre polygone.

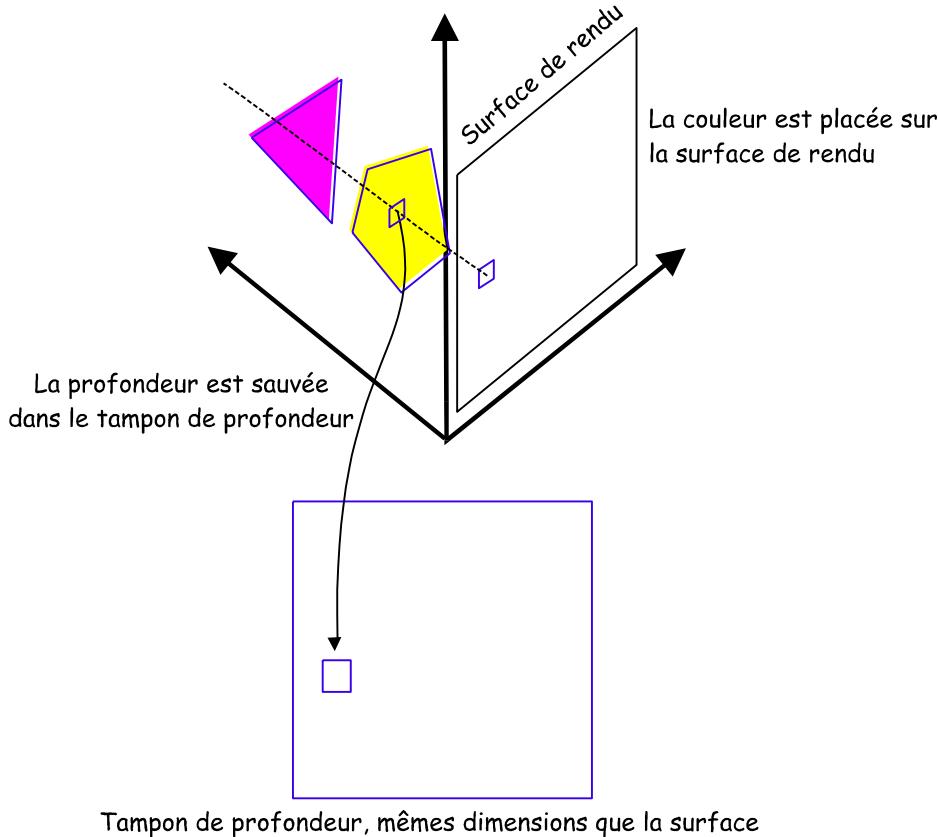
Lorsqu’une scène 3D est transformée en pixels sur la surface de rendu et qu’un tri de profondeur est activé, chaque point à rendre sur la surface est testé. Les valeurs placées dans le tampon de profondeur peuvent être le z d’un point (ou une approximation de celui-ci) ou la valeur w de sa coordonnée homogène (x, y, z, w). Tous deux résultant de la coordonnée d’un point dans l’espace de projection. Un tampon utilisant les valeurs z s’appelle un « *z-buffer* », et un tampon utilisant les valeurs w s’appelle un « *w-buffer* ». Chaque type de tampon a ses avantages et ses inconvénients comme nous le verrons plus tard.

Au début du rendu, les valeurs dans le tampon de profondeur sont initialisées à **la plus grande valeur Z** que puisse prendre les éléments de la scène une fois la projection effectuée c’est-à-dire 1.0 (**attention** au format...). La couleur de la surface de rendu est initialisée avec la couleur choisie pour le fond ou avec les pixels résultants de l’application d’une texture comme fond.

Lors du rendu de chaque pixel de chaque polygone dans la scène:

1. La couleur du nouveau pixel est calculée au moyen d’un pixel shader.
2. La valeur Z du nouveau pixel est testée pour savoir si elle est plus petite que la valeur actuelle du tampon. Si oui, la couleur calculée en 1 sera placée (ou mélangée) sur le tampon d’arrière-plan.

(Les pipelines pré-DX 10 faisaient l’inverse...).



Presque toutes les cartes graphiques avec accélération 3D implémentent les z-buffers, c'était d'ailleurs il y a quelque temps la seule fonctionnalité 3D de beaucoup de ces cartes, ce qui a rendu les z-buffers le type le plus répandu de tampons de profondeur. Mais les z-buffers ont leurs inconvénients. Dû au type d'opérations mathématiques impliquées, les valeurs z générées ne sont pas distribuées également sur l'intervalle du z-buffer (habituellement entre 0.0 et 1.0). La disposition des objets entre le plan de clipping rapproché et le plan de clipping éloigné affecte grandement la distribution des valeurs z. Souvent, plus de 90 % des objets sont situés à moins de 10 % du plan rapproché. Ce qui laisse très peu de latitude pour les autres objets. Ce qui pourra amener des déformations sur les objets distants, particulièrement avec des z-buffers de moins de 16 bits.

Un w-buffer, par contre, offre des valeurs distribuées plus régulièrement entre le plan rapproché et le plan éloigné. Notre application pourra alors supporter des écarts plus grands. Par contre, un w-buffer n'est pas parfait non plus et peut à l'occasion causer des déformations sur des objets situés très près (mais c'est aujourd'hui plus rare). Le principal inconvénient des w-buffers est que le support matériel n'est pas aussi répandu qu'avec les z-buffers (en fait, les w-buffers ne sont presque plus utilisés, pourquoi ?).

Optimisation des Z-Buffers

Les applications peuvent améliorer la performance du *z-buffer* (et des textures) en s'assurant que les objets de la scène sont rendus du plus près au plus loin. Par exemple en effectuant le rendu des personnages avant celui du décor. Les techniques utilisées pour l'application des textures sont encore plus efficaces puisqu'elles travaillent avec des « lignes de rendu » et non uniquement des points. Les *z-buffers* sont évidemment utiles et offrent une performance intéressante, mais leur utilisation est discutable si la scène comporte peu d'objets superposés et s'il est facile d'afficher les objets en partant du fond vers le devant.

Il est aussi possible d'améliorer la performance d'une application en déterminant la visibilité d'un objet avant de le rendre. Cette visibilité peut être déterminée par votre application selon vos critères au moyen de fonctions spécialisées. Si la fonction rapporte qu'un ensemble de polygones n'est pas visible, il ne sert à rien d'essayer de les rendre. Règle d'optimisation 3D no 1: rien n'est plus rapide que de ne rien faire...

Aussi, pour ce genre de test sur des ordinateurs très rapides, les opérations logicielles seront souvent plus rapides que celles effectuées par les cartes vidéo. Il vaut donc souvent la peine de tester nos fonctions avec les surfaces en mémoire système et en mémoire vidéo pour déterminer laquelle des deux méthodes est la plus efficace.

5.8 Implanter un tampon de profondeur

- Déclarez les deux variables suivantes dans la section *private* de la classe CDispositifD3D11:

```
// Pour le tampon de profondeur
ID3D11Texture2D* pDepthTexture;
ID3D11DepthStencilView* pDepthStencilView;
```

Explications

Le tampon est une texture. Un z-buffer est simplement un tableau reproduisant les mêmes dimensions que la surface de rendu. Ce tableau contient des valeurs de 0.0 à 1.0. 0.0 étant le plus près de la caméra et 1.0 étant le plus éloigné. Au début de chaque rendu, nous devrons initialiser le z-buffer à 1.0. Comme ça, les objets rencontrés seront affichés.

On devra l'associer à une « vue » pour que le système y fasse un « rendu ».

- Déclarez l'accesseur pour la variable pDepthStencilView dans la section *public* de la classe CDispositifD3D11:

```
ID3D11DepthStencilView* GetDepthStencilView() { return pDepthStencilView; }
```

- Ajoutez les deux messages suivants dans le tableau des chaînes de caractères (en ressource, les numéros n'ont pas d'importance):

DXE_ERREURCREATIONTEXTURE	115	Erreur de création de texture
DXE_ERREURCREATIONDEPTHSTENCILTARGET	116	Erreur de création du depth stencil view

- Déclarez la fonction **InitDepthBuffer** dans la section *private* de la classe CDispositifD3D11:

```
void InitDepthBuffer();
```

- Implémentez ainsi la fonction InitDepthBuffer:

```
void CDispositifD3D11::InitDepthBuffer()
{
    D3D11_TEXTURE2D_DESC depthTextureDesc;
    ZeroMemory( &depthTextureDesc, sizeof( depthTextureDesc ) );
    depthTextureDesc.Width = largeurEcran;
    depthTextureDesc.Height = hauteurEcran;
    depthTextureDesc.MipLevels = 1;
```

1

```

depthTextureDesc.ArraySize = 1;
depthTextureDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthTextureDesc.SampleDesc.Count = 1;
depthTextureDesc.SampleDesc.Quality = 0;
depthTextureDesc.Usage = D3D11_USAGE_DEFAULT;
depthTextureDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
depthTextureDesc.CPUAccessFlags = 0;
depthTextureDesc.MiscFlags = 0;

DXESSAYER( pD3DDevice->CreateTexture2D( &depthTextureDesc,
NULL, &pDepthTexture),
DXE_ERREURCREATIONTEXTURE );

// Création de la vue du tampon de profondeur (ou de stencil)
D3D11_DEPTH_STENCIL_VIEW_DESC descDSView;
ZeroMemory( &descDSView, sizeof(descDSView) );
descDSView.Format = depthTextureDesc.Format;
descDSView.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
descDSView.Texture2D.MipSlice = 0;

DXESSAYER( pD3DDevice->CreateDepthStencilView( pDepthTexture,
&descDSView,
&pDepthStencilView),
DXE_ERREURCREATIONDEPTHSTENCILTARGET );
}

```

Un **stencil** est parfois combiné au tampon de profondeur. Il s'agit de 8 bits permettant certains effets comme masquer des zones de l'écran, *dissoudre* des pixels, etc.

1

Explications

Pour créer la texture 2D qui nous servira de tampon de profondeur, nous devons remplir une structure de type D3D11_TEXTURE2D_DESC. Les dimensions du tampon de profondeur doivent correspondre à celles du tampon d'arrière-plan. Le paramètre D3D11_BIND_DEPTH_STENCIL indique que nous désirons utiliser la texture comme tampon de profondeur (ou de stencil, mais ça, c'est une autre histoire), et nous donnons à chaque élément du tampon un format de DXGI_FORMAT_D24_UNORM_S8_UINT, soit 24 bits pour la profondeur et 8 bits pour le stencil (inutilisés pour l'instant).

2

Puis, nous créons une vue « depth/stencil ». Nous remplissons à cet effet une structure de type D3D11_DEPTH_STENCIL_VIEW_DESC avec la définition des éléments correspondants à notre texture.

3

Puis nous pouvons appeler la fonction **ID3D11Device::CreateDepthStencilView**, qui prendra comme paramètres d'entrée la texture et la description de la vue.

- Effectuez l'appel de la fonction InitDepthBuffer, juste avant la ligne `pImmediateContext->OMSetRenderTargets...` (dans le constructeur de CDispositifD3D11)

```
InitDepthBuffer();
```

7. Modifiez l'énoncé `pImmediateContext->OMSetRenderTargets...` pour qu'il utilise la vue « depth/stencil ».

```
pImmediateContext->OMSetRenderTargets( 1,  
                                         &pRenderTargetView,  
                                         pDepthStencilView );
```

8. Ne pas oublier de libérer nos objets dans le destructeur de `CDispositifD3D11`:

```
DXRelacher (pDepthStencilView );  
DXRelacher (pDepthTexture);
```

Insérez ces lignes AVANT les autres énoncés « DXRelacher ». Même si nos éléments ne sont pas toujours interdépendants, j'ai pris comme habitude de relâcher les éléments dans l'ordre inverse de leur création.

9. C'est tout pour les initialisations. Il ne nous reste plus qu'à réinitialiser le tampon de profondeur avec des valeurs de 1.0 pour chaque pixel au début du rendu. Ajoutez les lignes suivantes à la fin de la fonction **CMoteurWindows::BeginRenderSceneSpecific**:

```
// On réinitialise le tampon de profondeur  
ID3D11DepthStencilView* pDepthStencilView = pDispositif-  
>GetDepthStencilView();  
pImmediateContext->ClearDepthStencilView( pDepthStencilView,  
D3D11_CLEAR_DEPTH, 1.0f, 0 );
```

Essayez votre programme, vous ne verrez pas beaucoup de changement puisque nous n'avons qu'un seul objet, mais si nous avions deux cubes...

Note:

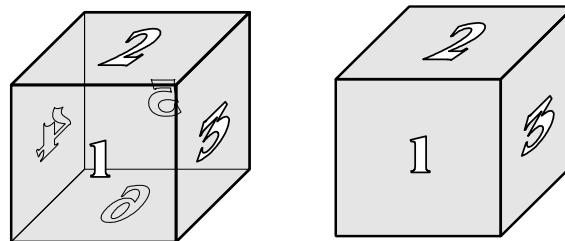
Il est possible de modifier la façon dont le test de profondeur est effectué.

1. Il faut pour cela remplir une structure de type `D3D11_DEPTH_STENCIL_DESC`.
2. Puis utiliser la fonction **ID3D11Device::CreateDepthStencilState** pour créer un état. Cet état sera conservé pour être utilisé au besoin lors du rendu. Attention, ces états ne sont pas dynamiques.
3. Lors du rendu (ou lors des initialisations ou...) vous utiliserez la fonction **ID3D11DeviceContext::OMSetDepthStencilState** pour activer

l'état. Le OM désigne le « **output-merger stage** » du pipeline de Direct3D. Les opérations de cette étape du pipeline sont bien décrites dans la documentation en ligne de DirectX Graphics.

Atelier

5.9 Implanter le « *backface-culling* »



Le « *backface culling* » est une technique quasi indispensable pour tous les objets non transparents. Elle permet au début de l'étape de rastérisation (*Rasterizer stage*) d'éliminer certains triangles **avant** de les rastériser.

Vous noterez que les objets solides présentent en moyenne 50 % de leurs polygones à la caméra, donc 50 % des polygones seront cachés. En évitant la rastérisation (et la suite du pipeline) pour ces polygones, nous obtenons un gain important de temps.

- Déclarer une variable dans la section *protected* de CDispositifD3D pour déterminer l'état de l'étape de rastérisation (*Rasterizer Stage State*) :

```
// Variables d'état
ID3D11RasterizerState* mSolidCullBackRS;
```

- Puis, insérez le code suivant à la fin du constructeur de CDispositifD3D11:

```
D3D11_RASTERIZER_DESC rsDesc;
ZeroMemory(&rsDesc, sizeof(D3D11_RASTERIZER_DESC));
rsDesc.FillMode = D3D11_FILL_SOLID;
rsDesc.CullMode = D3D11_CULL_BACK;
rsDesc.FrontCounterClockwise = false;
pD3DDevice->CreateRasterizerState(&rsDesc, &mSolidCullBackRS);

pImmediateContext->RSSetState(mSolidCullBackRS);
```

FILL_MODE est le mode de rastérisation, vous avez le choix entre D3D11_FILL_WIREFRAME et D3D11_FILL_SOLID.

CullMode est le mode de culling, nous éliminons les polygones « côté avant » (D3D11_CULL_FRONT) ou les polygones « côté arrière » (D3D11_CULL_BACK) ou nous n'en éliminons pas (D3D11_CULL_NONE).

FrontCounterClockWise indique si oui ou non le côté avant est « tournée » de façon antihoraire. Comme nous avons tourné nos triangles de façon horaire, nous indiquons **false**.

Notez que les états ne sont pas dynamiques et que vous **devrez** avoir un état différent pour chaque ensemble de paramètres que vous désirerez utiliser dans votre application. J'ai choisi un état dans l'initialisation, mais cet état pourrait (devrait ?) être choisi lors du rendu de chaque objet, ainsi un objet pourrait changer les paramètres de fonctionnement d'une étape du pipeline. Un **gestionnaire d'états** serait encore mieux...

3. Ne pas oublier de relâcher notre état dans le destructeur de CDispositifD3D11:

```
DXRelacher (mSolidCullBackRS);
```


6. Éclairage

La prochaine étape dans notre apprentissage de Direct3D sera d'ajouter de l'éclairage: pour simuler le soleil, les lumières intérieures, ou tout autre type d'éclairage. Sans éclairage, les objets sont uniformes ou noirs. C'est pourquoi nous avions tout de même utilisé un éclairage ambiant jusqu'à maintenant. La mise en place des sources d'éclairage dans nos scènes 3D est une tâche simple avec Direct3D, mais elle demande tout de même une bonne compréhension de la façon dont fonctionnent les différents modèles d'éclairage

objectifs du chapitre

- ✓ Comprendre les différents éléments impliqués dans l'éclairage d'une scène.
- ✓ Mettre en place un modèle d'éclairage simplifié.
- ✓ Développer des shaders plus complets.
- ✓ Se familiariser avec l'architecture « **effet** ».
- ✓ Comprendre l'échange d'information entre l'application et les shaders.

6. Éclairage

Si vous observez autour de vous, vous pourrez constater que la lumière provient de différentes sources. La lumière du soleil entre par la fenêtre, une lampe est allumée sur votre bureau, votre écran d'ordinateur émet une certaine lumière, il y a des fluorescents dans le corridor. En plus de toutes ces sources, la lumière est réfléchie par les différentes surfaces de la pièce, comme les miroirs ou les objets chromés, mais aussi par des surfaces plus mates comme le dessus de votre bureau.

Si nous voulions reproduire toute la luminosité d'une pièce, il nous faudrait des millions de calculs. Parce que la lumière se réfléchit sur les centaines de surfaces de la pièce, reproduire tous les rayons lumineux se croisant et se recroisant dans la pièce est une tâche quasi impossible pour une animation demandant un nombre réaliste d'images à la seconde. Beaucoup d'applications *tricheront* lorsque viendra le temps d'appliquer des éclairages. Au lieu d'utiliser de « vrais » éclairages 3D, elles utiliseront des éclairages déjà appliqués sur les textures (les bitmaps appliqués sur les murs, plafonds, planchers et autres objets). C'est une excellente solution pour les éclairages passifs qui ont peu d'impacts sur la scène. N'oubliez pas, pourquoi calculer ce qu'on peut préparer à l'avance ?

Néanmoins, nous sommes tout de même intéressés à implanter une certaine forme de « vrai » éclairage (on parlera souvent d'éclairage dynamique) dans nos applications. Les modèles d'éclairage utilisés en programmation 3D nous permettent certaines approximations de la façon dont les éclairages fonctionnent dans le monde réel.

6.1 Les modèles d'illumination et les types de sources de lumière

L'illumination est l'interaction des sources d'éclairages avec les matériaux éclairés. Plusieurs aspects de la lumière sont encore mystérieux. En physique et en photométrie, la lumière est souvent décrite comme une onde, mais paradoxalement elle est aussi décrite comme un flot de particules (les photons). Les deux explications ont leurs avantages et leurs inconvénients.

Pour les besoins du rendu en 3D, la lumière est définie en termes de sources de lumière (ou sources lumineuses ou tout simplement lumières). Et pour chacune de ces sources, nous aurons des caractéristiques telles position, direction du flux lumineux, couleur et intensité.

Lorsqu'une lumière du monde réel affecte une surface, elle peut:

- être absorbée (partiellement ou totalement),
- être réfléchie (partiellement ou totalement),
- être disséminée (réfléchie dans plusieurs directions ou **diffusée**),
- être réfractée (passée au travers de l'objet, mais en changeant de direction),
- être transmise (passée au travers de l'objet inchangée).

Il n'est pas possible en 3D (surtout en temps réel) de reproduire fidèlement ces possibilités pour tous les types de surfaces (matériaux). Donc le traitement graphique de la lumière et des surfaces qu'elle affectera ne sera en fait qu'une suite d'algorithmes implantant des modèles simples (relativement).

Les modèles d'**illumination globale** (*global illumination*) tentent d'implanter un modèle physique pour les interactions entre la lumière et les surfaces, ces modèles sont relativement simples, mais poussés à l'extrême peuvent rapidement générer des temps de calcul incompatibles avec une application temps réel (un jeu par exemple). Le **tracé de rayons** (*ray tracing*) amène la précision de l'éclairage encore plus loin en permettant d'implanter réellement le reflet, la réfraction et même la dissémination. Les méthodes de radiosité (*radiosity methods*) tentent de calculer les valeurs d'éclairage pour l'ensemble de la scène, diminuant de beaucoup les calculs nécessaires lors du déplacement du point de vue.

Lambert, de **Gouraud** et de **Phong** sont des modèles qui ont longtemps été utilisés en infographique. Aujourd'hui, des modèles plus complexes et plus physiquement réalisistes sont utilisés (Cook-Tor-

Les formules données dans les sections qui suivent sont des versions plus ou moins simplifiées de celles des travaux de Phong [Phong 1], mais elles nous seront particulièrement utiles pour mieux comprendre le fonctionnement de l'éclairage et pour le calculer nous-mêmes dans nos shaders.

L'illumination « émise »

La forme la plus simple d'illumination est de donner à chaque objet sa propre propriété d'illumination c'est-à-dire une valeur fixe indépendante de la position des sources de lumière et indépendante de la position de l'observateur. Le calcul de l'intensité résultante pourra être:

$$(formule 6.1.1) I_E = O_E$$

où I_E est l'intensité résultante, et O_E est la valeur fixe d'illumination de l'objet. Utilisée toute seule, cette valeur ne donne pas des résultats très réalistes, mais elle pourra nous être très utile dans certains cas pour compléter des valeurs d'illumination extérieure trop faibles. Cette forme d'illumination deviendra la valeur « émise » du matériau (*emissive*) d'où le E .

L'illumination ambiante et la source ambiante

Si nous considérons maintenant une source externe de lumière n'ayant ni position ni direction et communiquant à tous les éléments de la scène une lumière de couleur et d'intensité constantes (peut-être le résultat de tous les reflets et disséminations), nous obtenons une **lumière ambiante** (*ambient light*). La formule de celle-ci peut être exprimée:

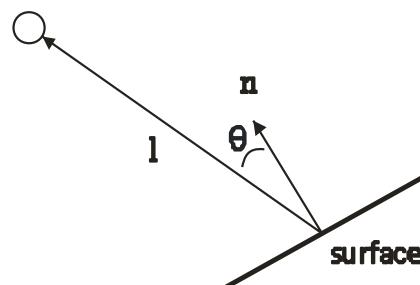
$$(formule 6.1.2) I_A = L_A O_A$$

où I_A est l'intensité ambiante résultante, L_A est l'intensité de la lumière ambiante, et O_A est le coefficient ambiant de l'objet (presque toujours une valeur de 0 à 1).

L'illumination diffuse

Pour des surfaces mattes qui disséminent la lumière dans toutes les directions (nous appellerons ceci **reflet diffus** (*diffuse reflection*) ou à l'occasion **reflet de Lambert**), l'intensité résultante dépend seulement de l'angle de la normale de la surface avec la direction de la source de lumière, elle ne dépend pas de la position de l'observateur.

source de lumière



Cette intensité peut être exprimée ainsi:

$$(formule 6.1.3) I_D = L_D O_D \cos(\theta)$$

où I_D est l'intensité diffuse résultante, L_D est l'intensité diffuse de la source de lumière, O_D est le coefficient diffus de l'objet (presque toujours une valeur de 0 à 1) et θ est l'angle entre \mathbf{l} , le vecteur vers la source de lumière frappant la surface et \mathbf{n} , la normale de la surface. Comme le démontre la figure ci-contre.

Notez que l'angle entre l'observateur (la caméra) et la surface de l'objet n'a pas d'importance parce que lorsque cet angle grandit, la quantité de lumière réfléchie diminue, mais l'aire de la surface observée augmente. Pour que notre formule fonctionne θ doit être entre zéro et quatre-vingt-dix degrés sinon la lumière n'illumine pas l'objet. Nous pourrions aussi vérifier ce critère. De plus, si nos deux vecteurs (\mathbf{l} et \mathbf{n}) sont normalisés (0 à 1) alors notre formule peut devenir:

$$(formule 6.1.4) I_D = L_D O_D (\mathbf{l} \cdot \mathbf{n})$$

Notez que cette équation doit être calculée en coordonnées du monde (ou occasionnellement de la caméra), **avant** la projection.

La combinaison de l'éclairage diffus et ambiant nous donne:

$$(formule 6.1.5a) I_T = I_A + I_D$$

ou

$$(formule 6.1.5b) I_T = L_A O_A + L_D O_D (\mathbf{l} \cdot \mathbf{n})$$

Si nous voulons améliorer la qualité de notre éclairage, nous pouvons maintenant tenir compte des effets de l'atténuation. Comme point de départ, nous pouvons tenir compte que l'éclairage diminue en fonction de la distance. Il nous suffit alors d'ajouter un facteur d'atténuation, **fatt**, qui tienne compte de la distance:

$$(formule 6.1.6) I_T = L_A O_A + fatt L_D O_D (\mathbf{l} \cdot \mathbf{n})$$

et nous pourrions très simplement calculer le facteur d'atténuation ainsi:

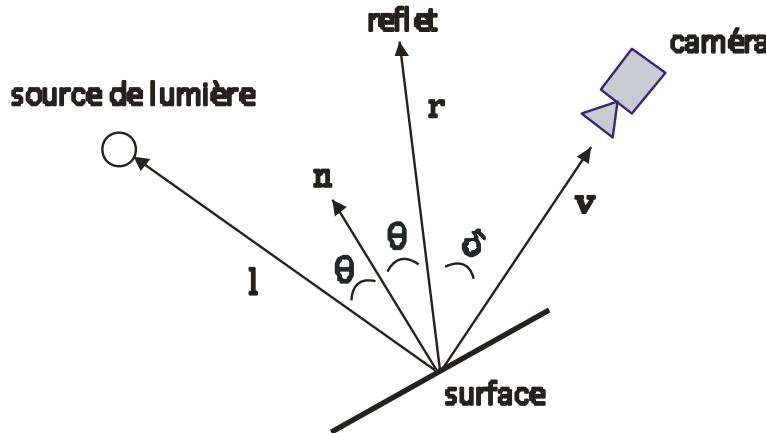
$$(formule 6.1.7) fatt = 1/dl$$

où **dl** est la distance de la source lumineuse.

L'illumination spéculaire (ou réflexion spéculaire)

La réflexion spéculaire est ce qui est observé quand une lumière vive est dirigée vers une surface « lustrée ». La couleur du reflet est surtout celle de la source de lumière donc le blanc est la couleur la plus naturelle pour les reflets spéculaires, mais comme nous utilisons un modèle mathématique, nous pourrions choisir la couleur du reflet spéculaire sur le matériau aussi. Le résultat pourrait n'être pas très naturel...

Si notre objet était un miroir parfait, alors nous ne verrions que le reflet de la lumière provenant à un angle égal à l'angle de vision, comme dans la figure suivante.



Donc l'illumination spéculaire variera avec l'angle de vision.

Plusieurs modèles d'illumination globale (Gouraud, Phong) s'appliquent à des miroirs imparfaits qui ne reflètent pas uniquement selon le vecteur reflet, mais aussi en tenant compte de l'angle δ entre l'observateur v et le vecteur reflet r . L'intensité du reflet visible est souvent en fonction de $\cos(\delta)^p$. Ce qui pourrait être ajouté à notre formule ainsi:

$$(formule 6.1.8) I_T = L_A O_A + f_{att} (L_D O_D (l \cdot n) + L_S O_S (r \cdot v)^p)$$

p est un coefficient de « spécularité », une « puissance » qui nous permet d'obtenir des résultats plus esthétiques. Le choix de cette puissance est arbitraire.

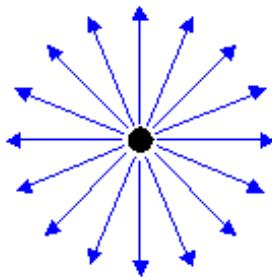
Nous reprendrons certaines de ces explications lors de la rédaction de notre shader.

La couleur de la lumière

La couleur d'une source lumineuse sera une combinaison des intensités de chaque couleur de base (dans notre cas le modèle RGB), avec les propriétés du matériau de chaque objet (dans notre cas selon le modèle RGBA). L'approximation résultante est nettement suffisante pour les besoins d'une application temps réel.

Les sources de lumières de type point

Cette source de lumière a une **position**, et émet des rayons lumineux dans toutes les directions depuis le point qui la représente. On l'appelle à l'occasion **source ponctuelle** de lumière.



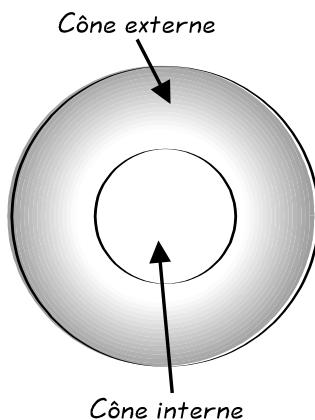
Une ampoule électrique (sans abat-jour...) est un bon exemple d'une lumière de type point. Ces lumières sont souvent affectées par la variation de lumière sur la distance (atténuation) et par la portée de ses rayons.

L'illumination de l'objet pourra varier selon sa distance et son angle par rapport à la source de lumière.

Pour réaliser ce type d'éclairage, nous utiliserons la position de la lumière et la position des éléments à éclairer pour calculer une **distance** et un **vecteur direction**. Puis, la source lumineuse communiquera à tous les points de la scène une lumière de couleur constante et d'intensité variable en fonction de la formule d'illumination choisie.

Les sources de lumière divergentes ou « spot »

Une source de lumière divergente a une **position** et émet des rayons de lumière dans des directions divergentes. L'ensemble des rayons lumineux forme un cône de lumière composé d'un **cône interne**, où l'intensité est constante, imbriqué dans un **cône externe**, où l'intensité décroît radialement et à une certaine vitesse jusqu'à atteindre 0 à l'extérieur du cône.



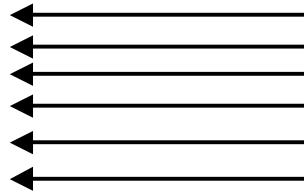
Cette source de lumière communique à tous les points qui se trouvent dans son champ une lumière de couleur constante et d'intensité variable, celle-ci étant fonction de:

- la **position** du point dans le cône de lumière;
- la **distance** qui sépare la source de lumière du point qu'elle illumine;
- la **portée maximale** de la source de lumière;
- l'orientation relative du point et de la source de lumière.

Les sources de lumière directionnelles

Si nous plaçons une source de lumière de type point à une grande distance de l'objet illuminé (comme le soleil par exemple) alors le vecteur **I** représentant un rayon provenant de notre source lumineuse sera constant et la distance de la source pourra être considérée comme infinie ou n'ayant pas de position.

Une source de lumière directionnelle n'a pas de position, donc la distance et l'atténuation n'entrent pas dans le calcul. Elle émet des rayons de lumière tous parallèles entre eux dans une direction donnée. Elle communique à tous les points de la scène une lumière de couleur constante et d'intensité variable, celle-ci étant uniquement fonction de l'orientation relative du point et de la source de lumière (l'angle entre **I** et **n**).



Sources lumineuses multiples

L'utilisation de plusieurs sources lumineuses est *presque* simple. Il suffit souvent d'additionner les résultats de l'illumination spéculaire et diffuse de chaque source.

Mais il faudra quelquefois s'assurer qu'un maximum n'est pas dépassé (le **plafond de couleur**). De plus, certains modèles tiennent compte de phénomènes d'éclairage plus complexes, alors l'addition ne sera pas suffisante.

Les matériaux

Un matériau est un ensemble d'attributs associé à un objet (ou à un sous-objet...) permettant de décrire la façon dont les polygones réfléchissent ou émettent de la lumière en 3D. C'est un moyen intéressant de définir certaines propriétés de luminosité sur un objet plutôt qu'uniquement dans nos sources d'éclairage. L'intérêt principal est de pouvoir utiliser un seul « effet d'éclairage » et d'obtenir des résultats différents selon les propriétés du matériau.

Ces attributs nous permettront de répondre aux trois questions suivantes:

- Comment les polygones reflètent la lumière diffuse et ambiante ?
- Quels sont leurs attributs de spécularité ?
- Est-ce que les polygones semblent émettre de la lumière ?

À titre d'exemple, voici ce que le pipeline fixe de Direct3D (version DX 9) utilisait:

```
// Initialiser le RGBA pour la réflexion ambiante.  
materiau.Ambient.r = 0.5f;  
materiau.Ambient.g = 0.0f;  
materiau.Ambient.b = 0.5f;  
materiau.Ambient.a = 1.0f;  
  
// Initialiser le RGBA pour la réflexion diffuse.  
materiau.Diffuse.r = 0.5f;  
materiau.Diffuse.g = 0.0f;  
materiau.Diffuse.b = 0.5f;  
materiau.Diffuse.a = 1.0f;  
  
// Initialiser la couleur et l'intensité de la spécularité  
materiau.Specular.r = 1.0f;  
materiau.Specular.g = 1.0f;  
materiau.Specular.b = 1.0f;  
materiau.Specular.a = 1.0f;  
materiau.Power = 50.0f;  
  
// Initialiser le RGBA pour la couleur "émissive".  
materiau.Emissive.r = 0.0f;  
materiau.Emissive.g = 0.0f;  
materiau.Emissive.b = 0.0f;  
materiau.Emissive.a = 0.0f;
```

Théorie

6.2 Éclairage simplifié avec spécularité

Une petite version de nuancement Phong (*Phong Shading*)

Pour les besoins de ce chapitre, nous développerons une version simplifiée du nuancement Phong et utiliserons une source de type point. Bui-Tong Phong a élaboré ces formules dans son livre « **Illumination for computer generated pictures** » paru en 1975 ! Nous étions alors très loin des cartes graphiques modernes.

Note historique:

Beaucoup de références, particulièrement celles destinées à des logiciels 3D, définissent le nuancement Phong comme étant « par pixel » et le nuancement Gouraud comme étant « par sommet ». Ce n'est pas tout à fait vrai...

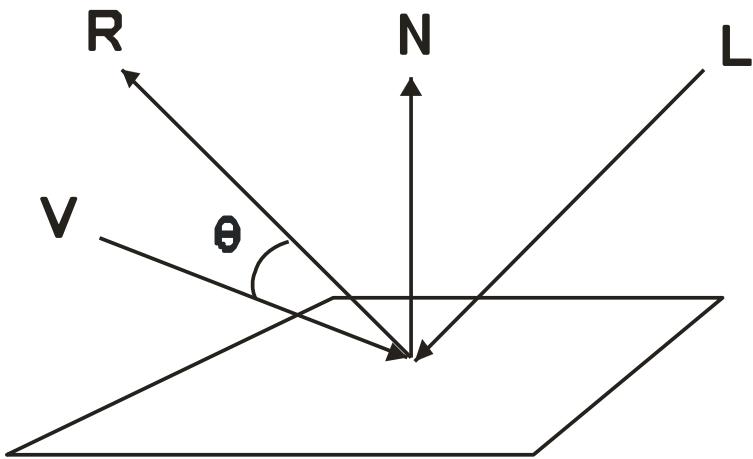
Effectivement, Gouraud a écrit ses techniques en 1971 et les a basés sur les sommets. Phong s'en est inspiré, y a ajouté la spécularité, et a proposé de corriger le « problème d'interpolation » (que nous verrons plus loin) en reportant certains calculs « au niveau des pixels ».

Au début des années 1970, tous les calculs étaient faits par le CPU, on ne parlait évidemment pas d'animation, les cartes graphiques étaient inexistantes hors de terminaux spécialisés, ceux-ci étaient monochromes et ne fonctionnaient que sur certains ordinateurs (*mainframe* ou *mini*), les résolutions graphiques étaient limitées, souvent moins de 300X200 (mais les terminaux Tektronix 4010 disposaient quand même d'une résolution monochrome de 1024 X 780). Donc la différence entre un traitement par pixel ou par sommet n'était pas aussi « physique » qu'aujourd'hui. Même en 1975, Phong n'a pas pu constater sur ordinateur les résultats de ses travaux.

Nous implanterons l'illumination Phong comme une technique mixte (sommets et pixels).

Retour sur la théorie

Pour obtenir la spécularité avec notre éclairage, nous devrons tenir compte de la **position de la caméra**, la **position de notre source d'éclairage** (ce sera un éclairage de type **point**). Dans le modèle développé par Phong, deux vecteurs interviennent dans le calcul de la composante de spécularité de l'illumination soient le vecteur V qui définit la position de la caméra, et le vecteur R qui correspond à la réflexion de la lumière sur la surface.



Nous identifierons l'angle entre V et R par θ . Plus V se rapproche de R et plus grand est l'effet de spécularité (100 %). Donc, nous pourrons utiliser $\cos(\theta)$ pour quantifier la valeur spéculaire. De plus, Phong introduit un paramètre p pour spécifier la puissance de la spécularité du matériel. Donc, la valeur de la spécularité deviendra:

$$(formule 6.2.1) S = (\cos \theta)^p$$

Notez que plus p est grand et plus S diminuera puisque P est un exposant et que $\cos(\theta)$ sera < 1 .

Tout comme dans les sections précédentes, nous pourrons profiter de certaines fonctionnalités du HLSL et des GPUs en utilisant une propriété du produit scalaire (*dot product*). Celui-ci est normalement défini comme suit:

$$(formule 6.2.2) R \cdot V = ||R|| * ||V|| * \cos \theta$$

Mais si les vecteurs de réflexion ainsi que la direction de la vision sont normalisés alors la formule peut être simplifiée en:

$$(formule 6.2.3) R \cdot V = \cos \theta$$

Puisque $R \cdot V$ est équivalent à $\cos \theta$, nous pourrons utiliser le produit scalaire (*dot product*) dans nos calculs. Donc la réflexion spéculaire peut maintenant s'exprimer:

$$(formule 6.2.4) S = (R \cdot V)^p$$

Il est possible de calculer le vecteur de réflexion R au moyen de la formule suivante:

$$(formule 6.2.5) R = 2 * (N \cdot L) * N - L$$

La formule d'illumination selon Phong devient donc:

$$(formule 6.2.6) \quad I = A_{\text{Éclairage}} * A_{\text{Matériaux}} \quad \text{composant ambient}$$
$$+ D_{\text{Éclairage}} * D_{\text{Matériaux}} * N \cdot L \quad \text{composant diffus}$$
$$+ S_{\text{Éclairage}} * S_{\text{Matériaux}} * (R \cdot V)^p \quad \text{composant spéculaire}$$

Le matériau pouvant être remplacé par la valeur d'un texel ou une combinaison des deux.

Notre version simplifiée de Phong

Nous utiliserons le blanc (100 %) comme couleur spéculaire (éclairage et matériau).

(formule 6.2.7)

$$I = A_{\text{Éclairage}} * A_{\text{Matériaux}} + D_{\text{Éclairage}} * D_{\text{Matériaux}} * N \cdot L + (R \cdot V)^p$$

Voilà pour la théorie!

6.3 Écriture des shaders

Par où commence-t-on ?

Par identifier les paramètres et constantes de nos shaders.

Comme nous le constaterons souvent, nous devrons écrire une paire de shaders soient un vertex shader, et un pixel shader. Nous pouvons classer les paramètres et constantes en trois catégories:

- Les constantes, initialisées avant le rendu. (Une fois par **rendu**)
- Les paramètres d'entrée et de sortie du **vertex shader** (une fois par **sommet**)
- Les paramètres d'entrée du pixel shader, la sortie étant la couleur... (Une fois par **pixel**)

a. Les constantes.

Nous avons besoin naturellement de:

```
float4x4 matWorldViewProj; // la matrice totale
```

Et pour notre algorithme:

```
float4x4 matWorld; // matrice de transformation dans le monde
float4 vLumiere; // la position de la source d'éclairage (Point)
float4 vCamera; // la position de la caméra
float4 vAEcl; // la valeur ambiante de l'éclairage
float4 vAMat; // la valeur ambiante du matériau
float4 vDEcl; // la valeur diffuse de l'éclairage
float4 vDMat; // la valeur diffuse du matériau
```

b. Les paramètres d'entrée et de sortie du vertex shader.

Outre les constantes, les paramètres d'entrée du vertex shader sont fournis par le pipeline via les informations que nous avons placées dans les sommets. Pour le moment, nous n'avons besoin que de la position (sémantique POSITION) et de la normale (sémantique NORMAL).

Comme le nuancement Phong est surtout orienté pour un traitement au niveau des pixels, notre vertex shader aura surtout comme travail de transmettre certaines valeurs au pixel shader (la normale des sommets par exemple). Mais il est **important** par souci d'optimisation de bien s'assurer de préparer un maximum d'information dans le vertex shader plutôt que de laisser le pixel shader le faire. Nous voulons transférer au pixel shader via le pipeline les informations suivantes:

- La position du sommet
- La normale

Les premières versions des shaders permettaient certaines libertés dans l'échange de données entre le VS et le PS.

Ce n'est plus le cas. Il faut maintenant que la « signature » de sortie due VS soit identique à la signature d'entrée.

- Le vecteur direction de la lumière
- Le vecteur direction de la caméra

Pour la position, pas de problème, nous pouvons réutiliser la sémantique `SV_Position`. Mais pour les autres valeurs, nous devrons utiliser des sémantiques libres, par exemple des sémantiques de texture. Nous en profitons pour déclarer une structure de sortie qui nous servira de structure d'entrée pour le pixel shader:

```
struct VS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : TEXCOORD0;
    float3 vDirLum : TEXCOORD1;
    float3 vDirCam : TEXCOORD2;
};
```

Notez les sémantiques `TEXCOORD0`, `TEXCOORD1` et `TEXCOORD2` associées à des éléments qui ne sont pas des textures. C'est un petit truc qui permet le transfert de **données interpolables** vers le pixel shader.

Notre vertex shader ressemblera donc à ceci (pour le moment) :

```
VS_Sortie MiniPhongVS(float4 Pos : POSITION, float3 Normale : NORMAL)
{
    VS_Sortie sortie = (VS_Sortie)0;

    ...

    return sortie;
}
```

c. Les paramètres d'entrée du pixel shader.

Outre les constantes, les paramètres d'entrée du pixel shader sont fournis par le pipeline via les informations que nous avions placées dans les sémantiques `TEXCOORD0`, `TEXCOORD1` et `TEXCOORD2`. Ne pas oublier que les données reçues correspondent à l'interpolation linéaire (bilinéaire pour être juste) de celles placées dans les sémantiques par le vertex shader.

`TEXCOORD0` correspond à la normale du pixel

`TEXCOORD1` correspond au vecteur direction de la lumière

`TEXCOORD2` correspond au vecteur direction de la caméra

Notre pixel shader ressemblera donc à ceci (pour le moment) :

```
float4 MiniPhongPS( VS_Sortie vs ) : SV_Target
{
    float4 couleur = 0;
    return couleur;
}
```

Le vertex shader

Le rôle principal du vertex shader (dans notre cas) est de convertir certaines informations exprimées en coordonnées de l'objet vers des coordonnées du monde 3D. C'est le cas pour la position et la normale. Les deux vecteurs, lumière et caméra, sont calculés selon la position du sommet dans le monde. Voici donc notre nouvelle version due VS:

```
cbuffer param
{
    float4x4 matWorldViewProj;    // la matrice totale
    float4x4 matWorld;          // matrice de transformation dans le monde
    float4 vLumiere;            // la position de la source d'éclairage (Point)
    float4 vCamera;             // la position de la caméra
    float4 vAEcl;                // la valeur ambiante de l'éclairage
    float4 vAMat;                // la valeur ambiante du matériau
    float4 vDEcl;                // la valeur diffuse de l'éclairage
    float4 vDMat;                // la valeur diffuse du matériau
}

struct VS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : TEXCOORD0;
    float3 vDirLum : TEXCOORD1;
    float3 vDirCam : TEXCOORD2;
};

VS_Sortie MiniPhongVS(float4 Pos : POSITION, float3 Normale : NORMAL)
{
    VS_Sortie sortie = (VS_Sortie)0;

    sortie.Pos = mul(Pos, matWorldViewProj);
    sortie.Norm = mul(float4(Normale, 0.0f), matWorld).xyz;

    float3 PosWorld = mul(Pos, matWorld).xyz;

    sortie.vDirLum = vLumiere.xyz - PosWorld;
    sortie.vDirCam = vCamera.xyz - PosWorld;

    return sortie;
}
```

ATTENTION!

Pour les besoins du développement et de nos exemples, nous plaçons nos shaders dans le dossier projet. Mais si vous exécutez le .EXE de votre programme directement, sans VS, il ne trouvera pas vos shaders. Solution: un ou plusieurs dossiers spéciaux pour les fichiers « runtime »



Copiez ce code dans un nouveau fichier appelé **MiniPhong.vhl** et sauvegardez-le dans votre projet (au même endroit que vos .cpp).

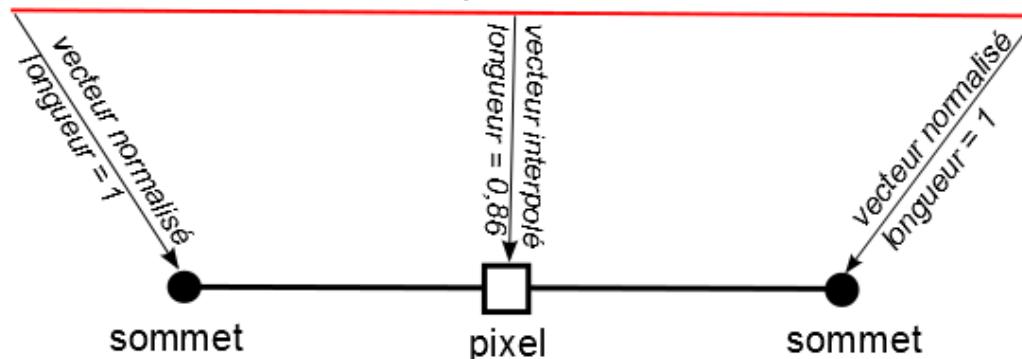


Les deux vecteurs direction ($vDirLum$ et $vDirCam$) ne sont pas normalisés, ils le seront dans le pixel shader. C'est un bon sujet de discussion:

Certains normalisent leurs vecteurs dans le VS plutôt que dans le PS, ce qui sauve un bon nombre de calculs, mais introduit une « légère » erreur dans les vecteurs utilisés par le PS. En effet, l'interpolation des valeurs placées dans les paramètres de sortie ne garantit absolument pas l'obtention de vecteurs normalisés. Par contre, dans certaines situations, l'erreur sera négligeable et vous pourrez tenir compte de cette condition pour accélérer votre effet.

C'est cette erreur que Phong a notée et c'est la raison pour laquelle il a reporté la normalisation et certains autres calculs au niveau des pixels. Certains auteurs effectuent le calcul des vecteurs direction dans le pixel shader. Ce ne sont pas de gros calculs, mais si nous pouvons les limiter.

Ligne d'interpolation linéaire



Puisque nous parlons de sommets (vertex) et de pixels, qu'est-ce qu'un **voxel** ?

Dans la figure ci-haut, l'erreur est volontairement exagérée. Les erreurs d'interpolation sont habituellement moindres pour de petits polygones d'où l'importance d'un certain niveau de **tessellation**.

Suite de la discussion: les résultats seraient-ils meilleurs si les vecteurs étaient aussi normalisés dans le VS avant d'être transmis au pipeline ?

Le pixel shader

- Première étape: Normaliser les vecteurs reçus en paramètres.
- Deuxième étape: calculer la composante diffuse (Section 6.1 et formule 6.2.7)
- Troisième étape: calculer R (formule 6.2.5)
- Quatrième étape: Calculez la spéculaire, nous utilisons ici une constante de 4 pour la puissance, mais celle-ci pourrait évidemment être un paramètre de l'effet (constante).
- Cinquième étape: additionner le tout selon la formule 6.2.7

Voici donc notre pixel shader:

```
cbuffer param
{
    float4x4 matWorldViewProj;    // la matrice totale
    float4x4 matWorld;          // matrice de transformation dans le monde
    float4 vLumiere;            // la position de la source d'éclairage (Point)
    float4 vCamera;             // la position de la caméra
    float4 vAEcl;                // la valeur ambiante de l'éclairage
    float4 vAMat;                // la valeur ambiante du matériau
    float4 vDEcl;                // la valeur diffuse de l'éclairage
    float4 vDMat;                // la valeur diffuse du matériau
}

struct VS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : TEXCOORD0;
    float3 vDirLum : TEXCOORD1;
    float3 vDirCam : TEXCOORD2;
};

float4 MiniPhongPS( VS_Sortie vs ) : SV_Target0
{
    float3 couleur;

    // Normaliser les paramètres
    float3 N = normalize(vs.Norm);
    float3 L = normalize(vs.vDirLum);
    float3 V = normalize(vs.vDirCam);

    // Valeur de la composante diffuse
    float3 diff = saturate(dot(N, L));

    // R = 2 * (N.L) * N - L
    float3 R = normalize(2 * diff.xyz * N - L);

    // Puissance de 4 - pour l'exemple
    float S = pow(saturate(dot(R, V)), 4);

    // I = A + D * N.L + (R.V)n
    couleur = vAEcl.rgb * vAMat.rgb +
              vDEcl.rgb * vDMat.rgb * diff;
    couleur += S;

    return float4(couleur, 1.0f);
}
```

Vous avez sans doute remarqué que j'utilise pour le pixel shader le même tampon de constantes que le vertex shader. C'est un truc pour nous simplifier la vie, surtout dans le cas où certaines constantes seraient utilisées dans les deux shaders (ce qui se produira souvent). Nous n'aurons ainsi qu'une seule initialisation de tampon de constantes.

Plus loin, nous aborderons l'utilisation de l'architecture « **Effects** » qui permet d'utiliser dans un même fichier plusieurs shaders (plusieurs VS, plusieurs GS, plusieurs PS). Dans ce cas, un seul tampon sera la meilleure solution.

→ Copiez ce code dans un nouveau fichier appelé **MiniPhong.phl** et sauvegardez-le dans votre projet (au même endroit que vos .cpp).

Atelier

6.4 Nos shaders dans le programme

L'échange d'information avec les shaders de type vertex et pixel est presque toujours à sens unique (ce ne sera pas le cas des compute shaders). Comme nous l'avons vu au chapitre 4, le moyen privilégié de communiquer avec les shaders est via un tampon de « constantes ».

Rappel

Avec HLSL 4.0 et 5.0, les constantes doivent être dans un bloc de type **cbuffer**.

Les constantes du shader sont toutefois des variables pour notre application. Les constantes ne changent pas pour toute la durée de l'exécution du pipeline.

1. Au chapitre 4, nous n'avions besoin que d'une seule constante, matWorldViewProj, une variable XMATRIX dans notre application. Mais cette fois-ci nous aurons besoin de plus que cela. Nous définirons donc une petite structure de données pour accueillir nos paramètres.

```
struct ShadersParams
{
    XMATRIX matWorldViewProj; // la matrice totale
    XMATRIX matWorld;        // matrice de transformation dans le monde
    XMFLOAT3 vLumiere;       // la position de la source d'éclairage (Point)
    XMFLOAT3 vCamera;        // la position de la caméra
    XMFLOAT3 vAEcl;          // la valeur ambiante de l'éclairage
    XMFLOAT3 vAMat;          // la valeur ambiante du matériau
    XMFLOAT3 vDEcl;          // la valeur diffuse de l'éclairage
    XMFLOAT3 vDMat;          // la valeur diffuse du matériau
};
```

Ajoutez cette structure à votre projet. Je vous suggère de la placer dans le fichier bloc.cpp puisque nous ne l'utiliserons pas ailleurs.

Les éléments de la structure correspondent directement aux constantes que nous avions placées dans nos shaders:

```
cbuffer param
{
    float4x4 matWorldViewProj; // la matrice totale
    float4x4 matWorld;        // matrice de transformation dans le monde
    float4 vLumiere;          // la position de la source d'éclairage (Point)
    float4 vCamera;           // la position de la caméra
    float4 vAEcl;              // la valeur ambiante de l'éclairage
    float4 vAMat;              // la valeur ambiante du matériau
    float4 vDEcl;              // la valeur diffuse de l'éclairage
    float4 vDMat;              // la valeur diffuse du matériau
}
```

ATTENTION! Certains problèmes d'alignement pourraient se produire. Vérifiez toujours si vos shaders reçoivent correctement les paramètres, par exemple avec le débogueur de shaders.

2. **Initialisation du tampon.** Dans mon cas, le tampon sera une variable locale de la fonction CBloc::Draw. C'est plus simple pour les besoins de nos démonstrations, mais il serait avantageux d'avoir une variable de classe et nous n'aurions rien à changer dans la fonction Draw puisque

nous aurions pu modifier les valeurs à des endroits stratégiques comme lors de l'initialisation (pour les couleurs et propriétés des matériaux) ou dans la fonction Anime, pour les matrices matWorldViewProj et matWorld. J'ai aussi, pour faire plus simple, codé directement la position de la source lumineuse et la position de la caméra. Ces informations devraient être obtenues du « système graphique ».

Dans la fonction **CBloc::Draw**, remplacez le code suivant:

```
// Initialiser et sélectionner les « constantes » du VS
XMMATRIX viewProj = CMoteurWindows::GetInstance().GetMatViewProj();
XMMATRIX matWorldViewProj = XMMatrixTranspose(matWorld * viewProj) ;
pImmediateContext->UpdateSubresource( pConstantBuffer, 0, NULL,
                                         &matWorldViewProj, 0, 0 );

pImmediateContext->VSSetConstantBuffers( 0, 1, &pConstantBuffer );

// Activer le PS
pImmediateContext->PSSetShader( pPixelShader, NULL, 0 );

// **** Rendu de l'objet
pImmediateContext->DrawIndexed( 36, 0, 0 );
```

Par celui-ci:

```
// Initialiser et sélectionner les « constantes » du VS
```

1

2

3

```
ShadersParams sp;
XMMATRIX viewProj = CMoteurWindows::GetInstance().GetMatViewProj();

sp.matWorldViewProj = XMMatrixTranspose(matWorld * viewProj );
sp.matWorld = XMMatrixTranspose(matWorld);

sp.vLumiere = XMVectorSet( -10.0f, 10.0f, -10.0f, 1.0f );
sp.vCamera = XMVectorSet( 0.0f, 0.0f, -10.0f, 1.0f );
sp.vAEcl = XMVectorSet( 0.2f, 0.2f, 0.2f, 1.0f );
sp.vAMat = XMVectorSet( 1.0f, 0.0f, 0.0f, 1.0f );
sp.vDEcl = XMVectorSet( 1.0f, 1.0f, 1.0f, 1.0f );
sp.vDMat = XMVectorSet( 1.0f, 0.0f, 0.0f, 1.0f );

pImmediateContext->UpdateSubresource(pConstantBuffer,0, nullptr ,&sp, 0,
0 );

pImmediateContext->VSSetConstantBuffers( 0, 1, &pConstantBuffer );

// Pas de Geometry Shader
pImmediateContext->GSSetShader(nullptr, nullp
```

2

3

```
pImmediateContext->PSSetShader( pPixelShader, nullptr, 0 );
pImmediateContext->PSSetConstantBuffers( 0, 1, &pConstantBuffer );

// **** Rendu de l'objet
pImmediateContext->DrawIndexed( (ARRAYSIZE(index_bloc)), 0, 0 );
```

Explications

1

Les initialisations sont semblables à celles du chapitre 4, sauf pour le nombre de paramètres.

2

Le **geometry shader** est spécifié à NULL (ou 0). Pour notre application, ce n'est pas nécessaire puisque nous n'avons qu'un seul objet, mais si nous en avions plusieurs avec plusieurs variantes de shaders, ce serait indispensable.

3

Notre pixel shader utilise le même tampon de constantes que le vertex shader. Mais nous aurions pu en avoir un différent, les paramètres auraient aussi été différents.

Nous avons deux autres petites modifications à apporter à la fonction **InitShaders** soient pour modifier les dimensions du tampon de constantes et évidemment pour utiliser MiniPhong.vhl et MiniPhong.phl comme shaders.

3. Dans la fonction InitShaders, modifiez la ligne suivante:

```
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(XMMATRIX);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    pD3DDevice->CreateBuffer( &bd, nullptr, &pConstantBuffer );
```

ainsi

```
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(ShadersParams);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    pD3DDevice->CreateBuffer( &bd, nullptr, &pConstantBuffer );
```

4. Modifiez la compilation du vertex shader pour qu'elle ressemble à ceci:

```
DXEssayer( D3DCompileFromFile( L"MiniPhong.vhl",
                                NULL, NULL,
                                "MiniPhongVS",
                                "vs_5_0",
                                D3DCOMPILE_ENABLE_STRICTNESS,
                                0,
                                &pVSBlob ,
                                NULL), DXE_FICHIER_VS);
```

5. Modifiez la compilation du pixel shader pour qu'elle ressemble à ceci:

```
DXEasser( D3DCompileFromFile( L"MiniPhong.phl",
                               NULL, NULL,
                               "MiniPhongPS",
                               "ps_5_0",
                               D3DCOMPILE_ENABLE_STRICTNESS,
                               0,
                               &pPSBlob,
                               NULL), DXE_FICHIER_PS);
```

Générez et exéutez votre programme pour vérifier que tout est en place.

Atelier

6.5 Les effets

Comme nous l'avons vu précédemment, les shaders permettent de développer de nouvelles fonctions graphiques destinées à être exécutées sur la carte graphique. Une amélioration importante aux shaders HLSL est l'architecture « **Effects** » qui permet d'améliorer l'écriture des shaders en écrivant un fichier d'effets pouvant contenir plusieurs shaders et leurs constantes, mais aussi des paramètres d'exécution, d'environnement, un support plus agréable du multipasse et même plus d'options pour lier notre fichier d'effets à des applications telles **FXComposer**, **RenderMonkey** et même avec certains outils de modélisation 3D comme 3DStudio Max.

À cause de ces avantages, l'architecture « Effects » sera idéale pour nous permettre d'expérimenter un peu plus avec les shaders. Nous connaissons déjà les vertex shaders, les pixel shaders, les geometry shaders (que nous n'avons pas encore utilisés) et les tampons de constantes comme paramètres pour nos shaders. Il sera possible avec « Effects » de placer la plupart de ces déclarations dans un seul fichier (extension suggérée: fx). Comme pour les shaders, le fichier d'effet peut être compilé lors de l'exécution ou de façon séparée (nous y reviendrons dans un autre chapitre).

Les fichiers effets contiennent au moins une **technique**. Une technique consiste en une ou plusieurs **passes** et son but est d'obtenir un effet de rendu, par exemple l'éclairage d'un objet avec spécularité (MiniPhong). Pour chaque passe, nos objets seront rendus d'une façon différente et le résultat de chaque passe sera possiblement combiné pour obtenir le résultat escompté par notre effet.

Une passe consiste généralement en un vertex shader, un geometry shader (optionnel), un pixel shader la possibilité de modifier certains états de rendu du pipeline (*render states*).

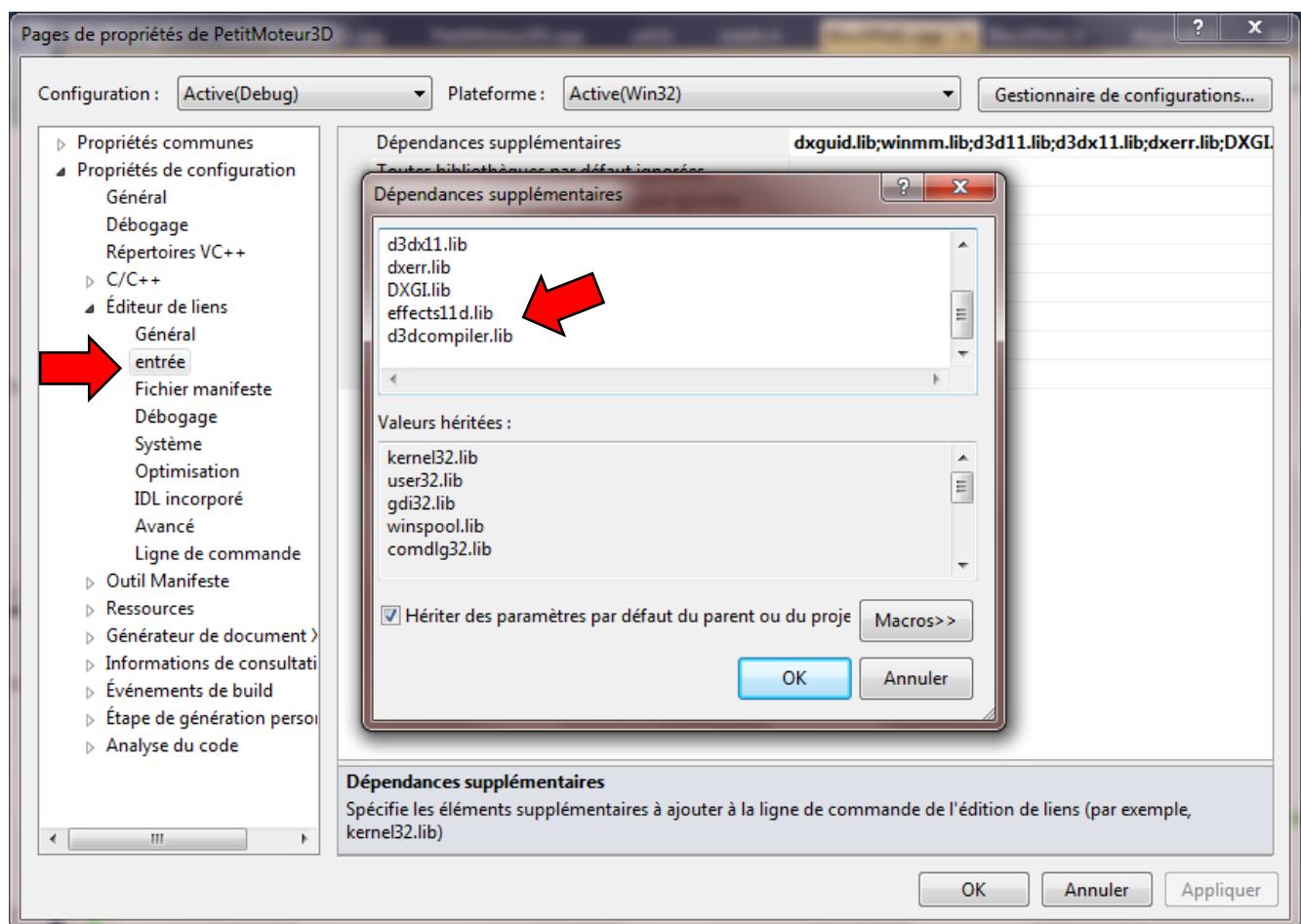
Dans cette section, nous regardons comment mettre en place dans notre application nos deux shaders précédents sous la forme d'un effet.

Effects 11

Avant DirectX 11, les éléments de l'architecture « **Effects** » faisaient partie de la bibliothèque D3DX. Mais **Effects 11** est distribué en version code source. Ainsi, lorsque nous créerons des applications utilisant les effets, nous pourrons utiliser directement les fichiers sources ou, comme moi, les utiliser à partir de la bibliothèque statique (.lib) créée par les projets fournis avec le SDK.

1. La bibliothèque **Effect 11** est disponible sur Internet en suivant le lien <http://go.microsoft.com/fwlink/?LinkId=271568>. Vous y trouverez une solution pour VS 2015 (et pour VS versions précédentes).

2. Générez la solution en configuration **DEBUG**. Dans le dossier **Debug**, renommez le fichier **Effects11.lib** en **Effects11d.lib** (le d est pour debug).
3. Si vous le désirez, vous pouvez aussi générer la solution en configuration **RELEASE**. Mais dans ce cas, conservez intact le nom de la bibliothèque soit **Effects11.lib**.
4. Vous pourriez utiliser des liens vers les fichiers précédemment créés, mais j'ai préféré les copier dans mon répertoire projet (PetitMoteur3D) au même endroit que mes fichiers sources.
5. Copiez aussi les deux fichiers « .h » du dossier **\Effects11\inc** dans votre répertoire projet.
6. Ajoutez le lien vers **Effects11d.lib** (ou **Effects11.lib** si vous êtes en configuration **Release**) au besoin ajoutez aussi un lien vers **D3DCompiler.lib** (utilisé par Effects) dans la section **Entrée|Dépendances supplémentaires** de l'éditeur de liens dans les propriétés de PetitMoteur3D.



Notre effet MiniPhong

7. Copiez le code suivant dans le fichier **MiniPhong.fx**

```

cbuffer param
{
    float4x4 matWorldViewProj;    // la matrice totale
    float4x4 matWorld;          // matrice de transformation dans le monde
    float4 vLumiere;            // la position de la source d'éclairage (Point)
    float4 vCamera;             // la position de la caméra
    float4 vAEcl;               // la valeur ambiante de l'éclairage
    float4 vAMat;               // la valeur ambiante du matériau
    float4 vDEcl;               // la valeur diffuse de l'éclairage
    float4 vDMat;               // la valeur diffuse du matériau
}

struct VS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : TEXCOORD0;
    float3 vDirLum : TEXCOORD1;
    float3 vDirCam : TEXCOORD2;
};

VS_Sortie MiniPhongVS(float4 Pos : POSITION, float3 Normale : NORMAL)
{
    VS_Sortie sortie = (VS_Sortie)0;

    sortie.Pos = mul(Pos, matWorldViewProj);
    sortie.Norm = mul(float4(Normale, 0.0f), matWorld).xyz;

    float3 PosWorld = mul(Pos, matWorld).xyz;

    sortie.vDirLum = vLumiere.xyz - PosWorld;
    sortie.vDirCam = vCamera.xyz - PosWorld;

    return sortie;
}

float4 MiniPhongPS(VS_Sortie vs) : SV_Target
{
    float3 couleur;

    // Normaliser les paramètres
    float3 N = normalize(vs.Norm);
    float3 L = normalize(vs.vDirLum);
    float3 V = normalize(vs.vDirCam);

    // Valeur de la composante diffuse
    float3 diff = saturate(dot(N, L));

    // R = 2 * (N.L) * N - L
    float3 R = normalize(2 * diff * N - L);

    // Puissance de 4 - pour l'exemple
    float S = pow(saturate(dot(R, V)), 4.0f);

    // I = A + D * N.L + (R.V)n
    couleur = vAEcl.rgb * vAMat.rgb +
              vDEcl.rgb * vDMat.rgb * diff;
    couleur += S;
}

```

```

        return float4(couleur, 1.0f);
    }

technique11 MiniPhong
{
    pass pass0
    {
        SetVertexShader(CompileShader(vs_5_0, MiniPhongVS()));
        SetPixelShader(CompileShader(ps_5_0, MiniPhongPS()));
        SetGeometryShader(NULL);
    }
}

```

Explications

- Le tampon de constantes est le même que précédemment. Sauf qu'on ne l'écrit qu'une seule fois.
- La structure VS_Sortie est la même que précédemment. Sauf qu'on ne l'écrit qu'une seule fois.
- Le vertex shader n'a pas changé.
- Le pixel shader n'a pas changé.
- Par contre, quelques lignes ont été ajoutées pour décrire la technique (une seule dans notre cas) MiniPhong. Cette technique n'a qu'une seule passe, identifiée **pass0**. Les lignes suivantes se passent d'explication puisqu'elles reprennent ce que nous faisions lors
 1. de l'initialisation des shaders (CompileShader);
 2. du rendu (SetVertexShader ou SetPixelShader ou SetGeometryShader)

Une classe de bloc avec effet

8. Copiez la classe CBloc pour obtenir une classe **CBlocEffet1**. Évidemment, renommez les références de classe de CBloc en CBlocEffet1 (par exemple CBloc::Draw en CBlocEffet1::Draw. Vous pourriez faire les modifications directement dans la classe CBloc, mais moi j'ai préféré la garder intacte... Nous aurions aussi pu faire un descendant de CBloc, mais pour les besoins de l'exemple, je préfère une classe « autonome ».

Vous aurez deviné que nous aurons sûrement une classe **CBlocEffet2** plus loin....

9. Retirez-y la définition de la constante **index_bloc**.

10. Mais ajoutez la ligne suivante à la suite des autres « #include »:

```
#include "bloc.h"
```

Pour l'index du bloc.

12. Modifiez la fonction **InitObjets** de CMoteur ainsi:

```
bool InitObjets()
{
    // Puis, il est ajouté à la scène
    ListeScene.emplace_back(std::make_unique<CBlocEffet1>(2.0f, 2.0f,
2.0f, pDispositif));

    return true; }
```

13. Ajoutez la ligne suivante à la suite des autres « #include » dans le fichier **Moteur.h**:

```
#include "bloceffet1.h"
```

Initialisation de l'effet

14. Modifiez le constructeur de CBlocEffet1 pour qu'il utilise une nouvelle fonction: **InitEffet** au lieu de InitShaders.

```
// Initialisation des shaders
InitShaders();
```

devient:

```
// Initialisation de l'effet
InitEffet();
```

15. Déclarez **InitEffet** dans la section *private* de CBlocEffet1.

16. Implantez la fonction InitEffet ainsi:

```
void CBlocEffet1::InitEffet()
{
    // Compilation et chargement du vertex shader
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    // Création d'un tampon pour les constantes du VS
    D3D11_BUFFER_DESC bd;
    ZeroMemory( &bd, sizeof(bd) );

    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(ShadersParams);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    pD3DDevice->CreateBuffer( &bd, NULL, &pConstantBuffer );
```

1

```

// Pour l'effet
ID3DBlob* pFXBlob = NULL;

DXESSAYER( D3DCompileFromFile( L"MiniPhong.fx", 0, 0, 0,
                                "fx_5_0", 0, 0,
                                &pFXBlob, 0),
            DXE_ERREURCREATION_FX);

D3DX11CreateEffectFromMemory( pFXBlob->GetBufferPointer(),
                               pFXBlob->GetBufferSize(),
                               0,
                               pD3DDevice,
                               &pEffet);

pFXBlob->Release();

pTechnique = pEffet->GetTechniqueByIndex(0);           ← 4
pPasse = pTechnique->GetPassByIndex(0);

// Créer l'organisation des sommets pour le VS de notre effet
D3DX11_PASS_SHADER_DESC effectVSDesc;
pPasse->GetVertexShaderDesc(&effectVSDesc);           ← 5

D3DX11_EFFECT_SHADER_DESC effectVSDesc2;
effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
                                              &effectVSDesc2);           ← 6

const void *vsCodePtr = effectVSDesc2.pBytecode;
unsigned vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout = NULL;
DXESSAYER( pD3DDevice->CreateInputLayout( CSommetBloc::layout,           ← 7
                                             CSommetBloc::numElements,
                                             vsCodePtr,
                                             vsCodeLen,
                                             &pVertexLayout ),
            DXE_CREATIONLAYOUT);
}

```

Explications

1 Le tampon de constantes est identique à ce que l'on faisait précédemment.

2 La compilation d'un effet utilise elle aussi la fonction **D3DCompileFromFile** (comme pour les shaders utilisés précédemment). Notez toutefois que le profil (paramètre 5) est maintenant "fx_5_0" et que le nom de la fonction d'entrée est maintenant 0 (soit un chaîne de longueur 0).

3 La fonction **D3DX11CreateEffectFromMemory** nous permet de créer un objet de classe **ID3DX11Effect**. En réalité, tout comme pour nos shaders précédents, cette fonction s'occupe de placer notre effet (ses shaders...) en mémoire vidéo. L'objet **ID3DX11Effect** nous permettra d'accéder plus facilement aux éléments de notre effet et d'automatiser certains traitements.

4

Un effet est composé de techniques (une seule dans notre cas), lesquelles sont composées de passes (une seule aussi pour nous) d'où le 0 comme numéro de technique et le 0 comme numéro de passe. Nous aurions pu accéder à notre technique via son nom (**GetTechniqueByName**).

Étant donné la situation (une technique, une passe). Nous prenons immédiatement en note un pointeur sur un objet **ID3DX11EffectTechnique** et un pointeur sur un objet **ID3DX11EffectPass**. C'est plus simple pour le moment.

Nous devons, comme pour nos shaders précédents, identifier l'organisation des sommets pour le vertex shader. C'est un peu plus complexe que précédemment puisque le vertex shader est « caché » dans notre effet.

5

- Nous obtenons d'abord la « **description** » de la passe.

6

- Puis nous obtenons la description du vertex shader. Celle-ci nous permet d'obtenir un pointeur sur la fonction (pour obtenir sa signature) et la longueur de code.

7

- Nous pouvons ensuite créer l'organisation de sommets

Nous retrouvons dans DirectX 11 beaucoup de fonctions **GetDesc** ou **GetXXXDesc** qui nous permettent d'accéder dynamiquement à des éléments situés habituellement en mémoire vidéo via des objets de référence

17. Déclarez ensuite dans la section *private* de la classe CBlocEffet1:

```
// Pour les effets
ID3DX11Effect* pEffet;
ID3DX11EffectTechnique* pTechnique;
ID3DX11EffectPass* pPasse;
```

18. Ajoutez à la suite des autres #include (dans le fichier BlocEffet1.h)

```
#include "d3dx11effect.h"
```

19. Ajoutez aussi le message d'erreur **DXE_ERREURCREATION_FX** "Erreur de création du fichier FX" à la table des chaînes de caractères (dans nos ressources).

Rendu avec effet

Le rendu est très semblable à celui utilisé précédemment, mais l'accès aux constantes est un peu différent et l'activation des shaders est effectuée par notre effet (en réalité par la passe active).

20. Modifiez la fonction CBlocEffet1::Draw pour qu'elle ressemble à ceci:

```
void CBlocEffet1::Draw()
{
    // Obtenir le contexte
    ID3D11DeviceContext* pImmediateContext = pDispositif->GetImmediateContext();

    // Choisir la topologie des primitives
    pImmediateContext->IASetPrimitiveTopology( D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

    // Source des sommets
    const UINT stride = sizeof( CSommetBloc );
    const UINT offset = 0;
    pImmediateContext->IASetVertexBuffers( 0, 1, &pVertexBuffer, &stride, &offset );

    // Source des index
    pImmediateContext->IASetIndexBuffer(pIndexBuffer, DXGI_FORMAT_R16_UINT, 0);

    // input layout des sommets
    pImmediateContext->IASetInputLayout( pVertexLayout );

    // Initialiser et sélectionner les « constantes » de l'effet
    ShadersParams sp;
    XMATRIX viewProj = CMoteurWindows::GetInstance().GetMatViewProj();

    sp.matWorldViewProj = XMMatrixTranspose(matWorld * viewProj );
    sp.matWorld = XMMatrixTranspose(matWorld);

    sp.vLumiere = XMVectorSet( -10.0f, 10.0f, -10.0f, 1.0f );
    sp.vCamera = XMVectorSet( 0.0f, 0.0f, -10.0f, 1.0f );
    sp.vAEcl = XMVectorSet( 0.2f, 0.2f, 0.2f, 1.0f );
    sp.vAMat = XMVectorSet( 1.0f, 0.0f, 0.0f, 1.0f );
    sp.vDEcl = XMVectorSet( 1.0f, 1.0f, 1.0f, 1.0f );
    sp.vDMat = XMVectorSet( 1.0f, 0.0f, 0.0f, 1.0f );
    pImmediateContext->UpdateSubresource( pConstantBuffer, 0, nullptr, &sp, 0, 0 );

    // Nous n'avons qu'un seul CBuffer
    ID3DX11EffectConstantBuffer* pCB = pEffet->GetConstantBufferByName("param");
    pCB->SetConstantBuffer(pConstantBuffer); 1

    // **** Rendu de l'objet
    pPasse->Apply(0, pImmediateContext); 2

    pImmediateContext->DrawIndexed( ARRSIZE(index_bloc), 0, 0 ); 3
}
```

Explications

Le début de la fonction n'a presque pas changé.

1

Nous obtenons un manipulateur sur le tampon des constantes.
Puis nous activons celui-ci.

- 2 Nous demandons à notre passe d'effectuer les sélections de shaders, et d'état. La passe fera donc ce que nous faisons auparavant avec des instructions telles VSSetShader, VSSetConstantBuffers et autres.
- 3 Le « DrawIndexed » n'a pas changé.

6.6 Le compilateur d'effets et de shaders

La méthode de compilation des effets et des shaders utilisée dans les sections précédentes ne nous permettait pas de corriger facilement les erreurs de compilation de nos shaders. Le compilateur FXC (*Effects compiler*) utilisé en mode commande nous permettra plus facilement de visualiser les erreurs de compilation de nos shaders. En réalité, nous aurions pu utiliser certains paramètres avec D3DX11CompileFromFile pour obtenir cette liste d'erreurs, mais c'est plus facile avec FXC.

La compilation externe des shaders avec FXC permet d'obtenir un shader binaire qui pourra être chargé par l'application. Plusieurs entreprises privilégient cette façon de faire puisqu'ainsi le code des shaders n'est pas distribué et ne peut donc être lu ni manipulé facilement.

L'utilitaire FXC permet la compilation des shaders et des effets, mais aussi la génération de code « assembler » correspondant au modèle cible (par exemple PS_5_0). Ce code permettra à ceux désirant devenir experts en GPU de se familiariser avec un jeu d'instruction parfois bien révélateur du fonctionnement interne du GPU.

La compilation avec FXC utilise elle aussi un fichier texte contenant un shader (VS ou PS ou GS selon le cas) ou des effets, elle le compile et le sauvegarde sous une forme binaire que nous pourrons charger dans notre application pour finalement la placer sur la carte graphique. Voir la section **Offline Compiling** de la documentation du SDK pour plus de détails.

À titre de référence pour plus tard, voici une petite fonction de lecture convenant autant aux effets qu'aux autres shaders:

```
HRESULT LireEffetOuShader(wchar_t* nomEffetOuShader, ID3DBlob** ppBuffer)
{
    HANDLE hFile;
    DWORD dwNombreDOctets;
    BYTE tampon[8192]; // Le shader est pour l'instant limité à 8k

    hFile = CreateFile(nomEffetOuShader,
                       GENERIC_READ,
                       0,
                       NULL,
                       OPEN_EXISTING,
                       FILE_ATTRIBUTE_NORMAL,
                       NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        return -1;
    }

    // Lecture du fichier
    ReadFile(hFile, tampon, sizeof(tampon), &dwNombreDOctets, NULL);

    // Création du ID3DXBuffer
    D3DCreateBlob( dwNombreDOctets, ppBuffer );
}
```

```
// Copie du code binaire dans le tampon  
memcpy( (*ppBuffer)->GetBufferPointer() , tampon, dwNombreDOctets);  
  
    return 0;  
}
```

Vous auriez pu utiliser vos fonctions ou classes de lecture préférées. J'ai pour ma part utilisé les fonctions de l'API de Windows. Ce n'est pas « portable », mais c'est évidemment plus complet.

7. Les textures

Les textures sont un moyen simple et très efficace d'ajouter du réalisme à nos objets 3D sans modifier leurs caractéristiques logiques. Les textures ne sont au départ que des images, mais elles peuvent être utilisées pour améliorer une primitive avec des motifs qui donneront l'illusion de vraies textures.

Objectifs du chapitre

- ✓ Connaître les différentes opérations pour le chargement de textures en mémoire vidéo et leur identification dans notre application.
- ✓ Comprendre comment associer les coordonnées de texture à notre objet.
- ✓ Comprendre les différents modes d'adressage des textures.
- ✓ Comprendre l'échantillonnage et le filtrage des textures.
- ✓ Mettre en place les états d'échantillonnage.
- ✓ Régler les problèmes de magnification ou de minification.

7. Les textures

Théorie

7.1 Les textures

Une texture est un **tableau de pixels**, ou plus spécifiquement un **tableau de texels** (*TEXture EElement*). Le terme *bitmap* sera aussi très utilisé dans les références (ne pas confondre avec le format de fichier .BMP). Les textures seront des **ressources** en mémoire vidéo.

Ce qui nous permettra d'utiliser n'importe quelle image et de l'appliquer à une primitive 3D. Les textures sont un moyen simple et très efficace d'ajouter du réalisme à nos objets 3D sans modifier leurs caractéristiques logiques. Les textures peuvent être utilisées pour améliorer une primitive avec des motifs qui donneront l'illusion de vraies textures.

Pour ajouter des textures à nos objets 3D, nous devrons effectuer les opérations suivantes:

1. Préparer l'utilisation des textures

Dans cette étape, nous devons préparer nos objets à recevoir des textures

2. Créer une surface de texture

Les textures doivent être chargées (d'une ressource ou d'un fichier) au moyen de fonctions spécialisées.

3. Faire le rendu de la primitive texturée

Direct3D et les shaders supportent beaucoup d'éléments reliés aux textures incluant certaines techniques et stratégies avancées.

Format des textures

Les texels d'une texture ont un format définissant de 1 à 4 composants, ce format est défini dans l'énumération DXGI_FORMAT. On parlera de **format de texel** ou de **format de pixel**.

Les textures sont organisées en fonction de leur format de tableau(1D, 2D ou 3D). Les textures seront créées comme une ressource de dimension connue, mais le format de pixel peut être spécifié à la création ou par l'utilisation d'une vue lorsque la texture sera associée au pipeline.

Pour les besoins de ce chapitre, nous utiliserons des textures 2D. Voir la documentation du SDK à la rubrique **Introduction To Textures in Direct3D 11** pour plus de détails.

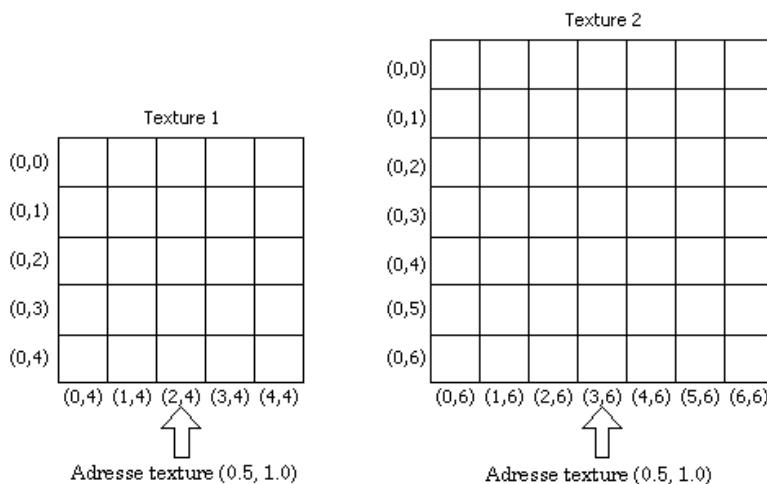
7.2 Coordonnées d'application des textures

La plupart des textures, comme des bitmaps, sont des tableaux à deux dimensions contenant des valeurs représentant des couleurs (il existe des exceptions). Les éléments individuels des textures s'appellent des **texels** (combinaison de texture et d'élément). Chaque texel a son adresse dans la texture. Les adresses sont exprimées sous forme de numéro de colonne et de numéro de ligne respectivement appelés *u* et *v*.

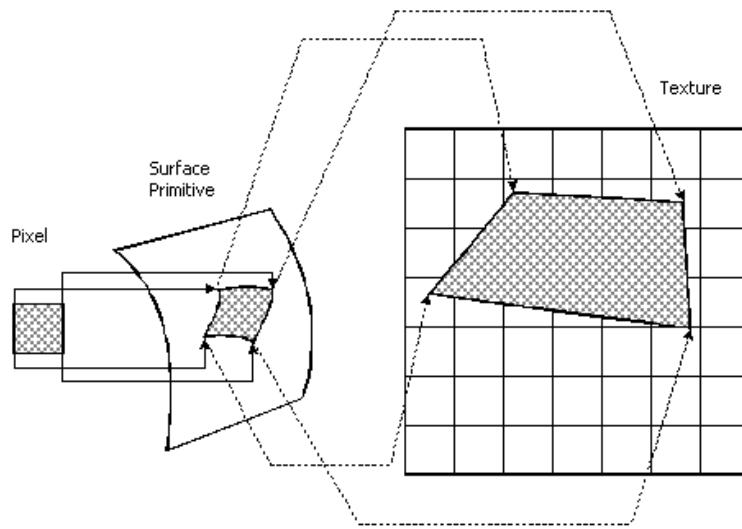
Les coordonnées de texture sont dans **l'espace texture**. C'est-à-dire qu'elles sont relatives à la position 0,0 de la texture. Quand une texture est appliquée à une primitive, les adresses de texels sont converties en coordonnées d'objet. Puis elles doivent être converties en coordonnées d'écran ou position de pixel. Direct3D convertira directement les texels de l'espace texture en pixels à l'écran, il n'y a pas d'étape intermédiaire. La projection est en réalité une projection inversée: pour chaque pixel déterminé sur la surface de rendu, le texel correspondant est trouvé. La couleur résultante est échantillonnée par un processus appelé le filtrage des textures.

Chaque texel dans une texture peut être identifié par ses coordonnées de texel. Néanmoins, pour appliquer les texels aux primitives, Direct3D requiert un intervalle d'adresses uniforme pour tous les texels de toutes les textures. Donc, nous utiliserons un intervalle de 0.0 à 1.0 pour représenter les adresses dans la texture.

Ceci a comme résultat que deux adresses de texture identiques peuvent correspondre à deux coordonnées de texel différentes si les textures n'ont pas les mêmes dimensions. Dans l'illustration de la figure suivante, l'adresse de texture utilisée est (0.5, 1.0). Néanmoins, parce que les textures sont de tailles différentes, l'adresse correspond à des coordonnées différentes.

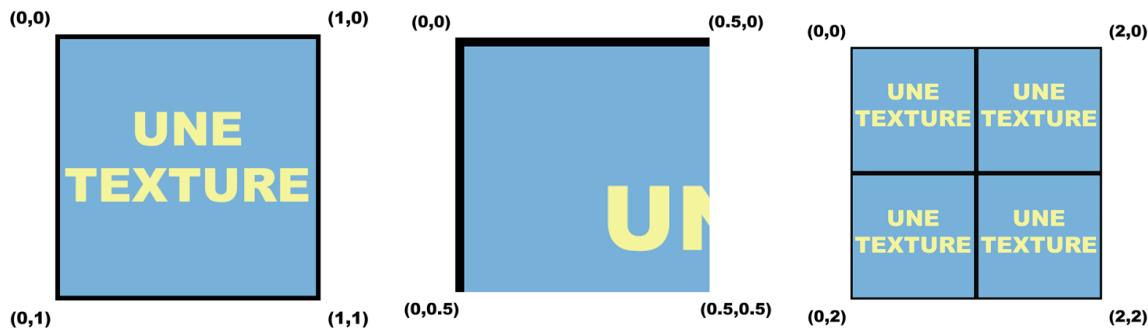


Une vision simplifiée du processus de projection des texels est illustrée à la figure suivante:



Pour les besoins de l'exemple, nous représentons le pixel résultant comme un carré de couleur. Les adresses théoriques des quatre coins du pixel sont projetées sur la primitive 3D dans l'espace de l'objet, la forme du pixel est déformée à cause de la forme de la primitive dans l'espace et à cause de l'angle de vision. Les coins sur la surface de la primitive sont alors projetés dans l'espace de la texture. Cette projection fait une autre déformation, c'est normal. La couleur qui sera donnée au pixel sera calculée en fonction des pixels dans la région obtenue. La méthode utilisée dans le calcul peut varier. Vous pouvez décider de cette méthode en modifiant la méthode de filtrage de texture.

Notre application peut assigner des coordonnées de texture au sommet de notre primitive. C'est une des caractéristiques de la structure SOMMET. Cette possibilité nous permet de décider quelle portion d'une texture est projetée sur la primitive. Par exemple, supposons que nous voulions appliquer une texture sur une primitive rectangulaire. Si nous désirons appliquer toute la texture sur l'ensemble de la primitive, les coordonnées de texture des sommets seront $(0.0,0.0)$, $(1.0,0.0)$, $(1.0,1.0)$ et $(0.0,1.0)$.



Supposons une autre primitive dont la hauteur serait la moitié de la précédente. Nous avons le choix entre continuer d'appliquer l'ensemble de la texture ou choisir de n'appliquer que la moitié de la texture.

Si nous décidons d'appliquer l'ensemble de la texture, des déformations se produiront alors le choix de la méthode de filtrage peut devenir important.

Par contre, si nous décidons d'assigner la moitié du bas de la texture sur la primitive, il suffit alors de modifier les coordonnées de texture des sommets pour $(0.0,0.0)$, $(1.0,0.0)$, $(1.0,0.5)$ et $(0.0,0.5)$. Direct3D pourra alors appliquer la moitié du bas de la texture à la primitive.

Il est aussi possible pour des coordonnées de texture d'être plus grandes que 1.0. Dans ce cas, il faudra décider du mode d'adressage de la texture pour que Direct3D sache quoi faire de ces coordonnées.

Atelier

7.3 Établir les coordonnées d'application de la texture

Notre application peut assigner les coordonnées d'application des textures directement au sommet des primitives en fournissant les valeurs correctes lors de l'initialisation des sommets (un type que nous avons nous-mêmes déclaré au chapitre 4).

1. Modifiez classe CSommetBloc (dans le fichier sommetbloc.h) pour qu'elle contienne maintenant des références aux coordonnées de textures.

```
class CSommetBloc
{
public:
    CSommetBloc() = default;
    CSommetBloc(const XMFLOAT3& position, const XMFLOAT3& normal, const
    XMFLOAT2& coordTex = XMFLOAT2(0.0f, 0.0f));

    static UINT numElements;
    static D3D11_INPUT_ELEMENT_DESC layout[];

private:
    XMFLOAT3 m_Position;
    XMFLOAT3 m_Normal;
    XMFLOAT2 m_CoordTex; ←
};
```

Notez l'ajout de la variable membre **coordTex**, qui correspondra à nos coordonnées d'application de texture. Notez aussi la modification au prototype du constructeur pour y inclure ces coordonnées.

2. Modifiez le constructeur de CSommetBloc (dans le fichier sommetbloc.cpp) pour qu'il ressemble à ceci:

```
CSommetBloc::CSommetBloc(const XMFLOAT3& position, const XMFLOAT3& normal,
const XMFLOAT2& coordTex /*= XMFLOAT2(0.0f, 0.0f)*/)
    : m_Position(position)
    , m_Normal(normal)
    , m_CoordTex(coordTex)
{}
```

Ici, je triche un peu...
Nous travaillerons avec
CBlocEffet1, mais je ne
veux pas d'erreurs dans la
version précédente soit
CBloc.

3. Modifiez la déclaration du tableau **layout** (dans le fichier sommet-bloc.cpp) de la façon suivante:

```
// Definir l'organisation de notre sommet
D3D11_INPUT_ELEMENT_DESC CSommetBloc::layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0},
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
D3D11_INPUT_PER_VERTEX_DATA, 0},
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
D3D11_INPUT_PER_VERTEX_DATA, 0}
};
```

4. Modifiez le code du constructeur de la classe CBlocEffet1 pour qu'il tienne compte des coordonnées de texture. Voici la nouvelle version des lignes qui ont été modifiées:

```
CSommetBloc sommets[24] =
{
    // Le devant du bloc
    CSommetBloc(point[0], n0, XMFLOAT2(0.0f, 0.0f)),
    CSommetBloc(point[1], n0, XMFLOAT2(1.0f, 0.0f)),
    CSommetBloc(point[2], n0, XMFLOAT2(1.0f, 1.0f)),
    CSommetBloc(point[3], n0, XMFLOAT2(0.0f, 1.0f)),

    // L'arrière du bloc
    CSommetBloc(point[4], n1, XMFLOAT2(0.0f, 1.0f)),
    CSommetBloc(point[5], n1, XMFLOAT2(0.0f, 0.0f)),
    CSommetBloc(point[6], n1, XMFLOAT2(1.0f, 0.0f)),
    CSommetBloc(point[7], n1, XMFLOAT2(1.0f, 1.0f)),

    // Le dessous du bloc
    CSommetBloc(point[3], n2, XMFLOAT2(0.0f, 0.0f)),
    CSommetBloc(point[2], n2, XMFLOAT2(1.0f, 0.0f)),
    CSommetBloc(point[6], n2, XMFLOAT2(1.0f, 1.0f)),
    CSommetBloc(point[5], n2, XMFLOAT2(0.0f, 1.0f)),

    // Le dessus du bloc
    CSommetBloc(point[0], n3, XMFLOAT2(0.0f, 1.0f)),
    CSommetBloc(point[4], n3, XMFLOAT2(0.0f, 0.0f)),
    CSommetBloc(point[7], n3, XMFLOAT2(1.0f, 0.0f)),
    CSommetBloc(point[1], n3, XMFLOAT2(1.0f, 1.0f)),

    // La face gauche
    CSommetBloc(point[0], n4, XMFLOAT2(0.0f, 0.0f)),
    CSommetBloc(point[3], n4, XMFLOAT2(1.0f, 0.0f)),
    CSommetBloc(point[5], n4, XMFLOAT2(1.0f, 1.0f)),
    CSommetBloc(point[4], n4, XMFLOAT2(0.0f, 1.0f)),

    // La face droite
    CSommetBloc(point[1], n5, XMFLOAT2(0.0f, 0.0f)),
    CSommetBloc(point[7], n5, XMFLOAT2(1.0f, 0.0f)),
    CSommetBloc(point[6], n5, XMFLOAT2(1.0f, 1.0f)),
    CSommetBloc(point[2], n5, XMFLOAT2(0.0f, 1.0f))
};
```

Notez l'organisation de la première face, le sommet 0 (en haut à gauche) correspond à la coordonnée de texture (0,0), le sommet 1 (en haut à droite) correspond à la coordonnée (1,0), etc.

Les autres faces sont disposées de façon plus aléatoire, à vous de les retourner (au besoin).

Pour chaque déclaration de CSommetBloc, le dernier paramètre représente les coordonnées d'application de texture du sommet. Dans notre cas, nous

appliquerons une texture carrée sur des surfaces carrées donc nous utiliserons toute la texture sur chaque face du bloc ce qui justifie les coordonnées: $(0, 0)$, $(1, 0)$, $(1, 1)$, et $(0, 1)$.

7.4 Le chargement et la gestion des textures

Il est facile de constater que les textures peuvent utiliser une quantité assez importante de mémoire vidéo. Par exemple, pour une application utilisant 10 textures de 512 X 512 texels de 32 bits, chaque texture occupe très exactement 1MB de mémoire donc un total de 10 MB. 3,3 MB de mémoire supplémentaire sera nécessaire si les textures sont des *mipmap*.

En partie à cause de l'espace mémoire important associé aux textures, il faudra en assurer une gestion efficace lors du chargement et de l'utilisation. Par exemple, une même texture **ne devrait pas** être chargée plusieurs fois.

Nouvelles classes nécessaires pour la gestion des textures

La classe CTexture

Il s'agit d'une classe **spécifique** dont le rôle est d'encapsuler les éléments spécifiques de l'implantation soient:

Données membres

- Le nom de fichier de la texture.
- Le pointeur vers un objet texture en ressource (**ID3D11ShaderResourceView***).

Fonctions membres

- Un constructeur paramètré pour le chargement de la texture.
- Un accesseur pour le nom de fichier de la texture (**GetFilename**).
- Un accesseur pour l'objet texture (**GetD3DTexture**).
- Un destructeur.

1. Ajoutez au projet le fichier Texture.h qui contiendra l'interface de notre classe, insérez dans le fichier les lignes suivantes:

```
#pragma once

namespace PM3D
{
class CTexture
{
public:
    CTexture();
    ~CTexture();
};
}
```

2. Ajoutez au projet le fichier Texture.cpp qui contiendra l'implémentation de notre classe, c'est-à-dire ce qui suit:

```
#include "StdAfx.h"
#include "strsafe.h"
#include "dispositifd3d11.h"
#include "Texture.h"
#include "resource.h"
#include "util.h"
#include "DDSTextureLoader.h" ←

using namespace UtilitairesDX;
using namespace DirectX;

namespace PM3D
{
    CTexture::CTexture()
    {
    }

    CTexture::~CTexture()
    {
    }
}
```

Notez que nous utiliserons la fonction utilitaire CreateDDSTextureFromFile. Il faut donc inclure le fichier **DDSTextureLoader.h**.

3. Copiez dans votre projet les fichiers **DDSTextureLoader.h** et **DDSTextureLoader.cpp** à partir des utilitaires DirectXTex (fournis avec mes fichiers sources, mais aussi disponibles sur <http://go.microsoft.com/fwlink/?LinkId=248926>).
4. Ajoutez **D3DSTextureLoader.cpp** à votre projet. Ajoutez aussi l'énoncé suivant au début des #include du fichier D3DSTextureLoader.cpp

```
#include "stdafx.h"
```

Nous pouvons ainsi utiliser les entêtes précompilés, mais surtout nous évitons des problèmes de définition de plateforme.

5. Déclarez dans la classe CTexture les variables correspondant au nom de fichier et au pointeur sur l'objet texture:

```
private:
    std::wstring m_Filename;
    ID3D11ShaderResourceView* m_Texture;
```

Notez que les types des variables sont spécifiques à notre implantation sous Windows et DirectX.

6. Déclarez dans la classe CTexture (Texture.h) le constructeur paramétré:

```
CTexture(const std::wstring& filename, CDispositifD3D11* pDispositif);
```

7. Ajoutez au fichier Texture.cpp le constructeur paramétré:

```
CTexture::CTexture(const std::wstring& filename, CDispositifD3D11*  
pDispositif)  
: m_Filename(filename)  
, m_Texture(nullptr)  
{  
    ID3D11Device* pDevice = pDispositif->GetD3DDevice();  
  
    // Charger la texture en ressource  
    DXEssayer(CreateDDSTextureFromFile(pDevice,  
        m_Filename.c_str(),  
        nullptr,  
        &m_Texture), DXE_FICHIERTEXTUREINTROUVABLE);  
}
```

Notez l'utilisation de **wstring** pour le support unicode.

8. Déclarez dans la section *public* de la classe CTexture la fonction **GetFilename**:

```
const std::wstring& GetFilename() const { return m_Filename; }
```

9. Déclarez dans la section *public* de la classe CTexture la fonction **GetD3DTexture**:

```
ID3D11ShaderResourceView* GetD3DTexture() { return m_Texture; }
```

10. Modifiez le destructeur (dans Texture.cpp) pour qu'il ressemble à ceci:

```
CTexture::~CTexture()  
{  
    DXRelacher(m_Texture);  
}
```

La classe CGestionnaireDeTextures

Nous planterons ici un petit gestionnaire de textures dont les objectifs seront:

- le stockage unique de chaque texture,
- l'obtention d'un pointeur sur la texture via son *nom de fichier*.
- La libération des ressources associées aux textures à la fin du programme ou au besoin.

Nous utiliserons ici une classe de gestionnaire de ressource. C'est un choix que vous retrouverez dans beaucoup d'architectures 3D.

Notez que vous pouvez utiliser d'autres modèles, les fabriques par exemple, pour faciliter la création et la gestion des

Données membres

- Un vecteur (**std::vector**) pour le stockage des textures. En réalité nous ne stockerons que des pointeurs vers des textures.

Fonctions membres

- Une fonction, **GetNewTexture**, pour obtenir un pointeur sur un objet texture via son nom de fichier. Cette fonction créera une nouvelle texture (qui se chargera) si celle-ci n'est pas déjà dans le vecteur.
- Une fonction, **GetTexture**, pour obtenir un pointeur sur un objet texture via son nom de fichier. Cette fonction retournera un pointeur nul (0) si la texture n'est pas déjà dans la liste.

9. Ajoutez au projet le fichier **GestionnaireDeTextures.h** qui contiendra l'interface de notre classe, insérez dans le fichier les lignes suivantes:

```
#pragma once
#include "texture.h"

namespace PM3D
{

    class CDispositifD3D11;

    class CGestionnaireDeTextures
    {
    public:
        CTexture* const GetNewTexture(const std::wstring& filename,
CDispositifD3D11* pDispositif);
        CTexture* const GetTexture(const std::wstring& filename);

    private:
        // Le tableau de textures
        std::vector<std::unique_ptr<CTexture>> ListeTextures;
    };
}

} // namespace PM3D
```

10. Ajoutez au projet le fichier **GestionnaireDeTextures.cpp** qui contiendra l'implémentation de notre classe, c'est-à-dire ce qui suit:

```
#include "StdAfx.h"
#include "dispositifD3D11.h"
#include "GestionnaireDeTextures.h"

namespace PM3D
{

    CTexture* const CGestionnaireDeTextures::GetNewTexture(const std::wstring&
filename, CDispositifD3D11* pDispositif)
{
    // On vérifie si la texture est déjà dans notre liste
    CTexture* pTexture = GetTexture(filename);
```

```

// Si non, on la crée
if (!pTexture)
{
    auto texture = std::make_unique<CTexture>(filename, pDispositif);
    pTexture = texture.get();
    // Puis, il est ajouté à la scène
    ListeTextures.push_back(std::move(texture));
}

assert(pTexture);
return pTexture;
}

CTexture* const CGestionnaireDeTextures::GetTexture(const std::wstring&
filename)
{
    CTexture* pTexture = nullptr;

    for (auto& texture : ListeTextures)
    {
        if (texture->GetFilename() == filename)
        {
            pTexture = texture.get();
            break;
        }
    }

    return pTexture;
}
} // namespace PM3D

```

10. Ajoutez aux messages d'erreurs le message suivant:

DXE_FICHIERTEXTUREINTROUVABLE

Fichier texture introuvable

Atelier

7.5 Les textures dans nos shaders

Le traitement des textures dans les shaders est pour le moment limité à trois éléments:

- Ajouter un lien vers la ressource « texture », ce lien nous servira dans le pixel shader. Ce lien sera accompagné d'un « état de sampling » (état d'échantillonnage en français) (*sampler state*). Voir la section 7.7 pour plus d'explications.
- Ajuster le vertex shader pour qu'il tienne compte des coordonnées d'application de texture. Celles-ci devront aussi être transmises au reste du pipeline donc nous devrons modifier la structure de sortie du VS (qui est la structure d'entrée du PS).
- Modifier le pixel shader pour qu'il utilise la couleur de la texture comme base pour le calcul de la couleur résultante.

1. Dans MiniPhong.fx, juste avant le pixel shader, ajoutez les lignes suivantes:

```
Texture2D textureEntree; // la texture
SamplerState SampleState; // l'état de sampling
```

2. Toujours dans le fichier MiniPhong.fx, modifiez le VS de la façon suivante:

```
VS_Sortie MiniPhongVS(float4 Pos : POSITION, float3 Normale : NORMAL,
                      float2 coordTex: TEXCOORD)
{
    VS_Sortie sortie = (VS_Sortie)0;                                Ajoutez!
    sortie.Pos = mul(Pos, matWorldViewProj);
    sortie.Norm = mul(float4(Normale, 0.0f), matWorld).xyz;

    float3 PosWorld = mul(Pos, matWorld).xyz;

    sortie.vDirLum = vLumiere.xyz - PosWorld;
    sortie.vDirCam = vCamera.xyz - PosWorld;
    // Coordonnées d'application de texture
    sortie.coordTex = coordTex;                                     Ajoutez!
    return sortie;
}
```

3. Toujours dans le fichier MiniPhong.fx, modifiez la structure VS_Sortie de la façon suivante:

```
struct VS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : TEXCOORD0;
    float3 vDirLum : TEXCOORD1;
    float3 vDirCam : TEXCOORD2;
    float2 coordTex : TEXCOORD3; // Ajoutez!
};
```

Notez qu'on utilise TEXCOORD3 comme sémantique parce que 0,1 et 2 sont déjà prises.

4. Toujours dans le fichier MiniPhong.fx, modifiez le PS de la façon suivante:

```
float4 MiniPhongPS( VS_Sortie vs ) : SV_Target
{
    float3 couleur;

    // Normaliser les paramètres
    float3 N = normalize(vs.Norm);
    float3 L = normalize(vs.vDirLum);
    float3 V = normalize(vs.vDirCam);

    // Valeur de la composante diffuse
    float3 diff = saturate(dot(N, L));

    // R = 2 * (N.L) * N - L
    float3 R = normalize(2 * diff * N - L);

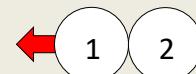
    // Puissance de 4 - pour l'exemple
    float S = pow(saturate(dot(R, V)), 4.0f);

    // Échantillonner la couleur du pixel à partir de la texture
    float3 couleurTexture = textureEntree.Sample(SampleState,
        vs.coordTex).rgb; // 1

    // I = A + D * N.L + (R.V)
    couleur = couleurTexture * vAEcl.rgb * vAMat.rgb +
        couleurTexture * vDEcl.rgb * vDMat.rgb * diff; // 2

    couleur += S;

    return float4(couleur, 1.0f);
}
```



Explications

couleurTexture est la couleur qui sera obtenue de la texture.

- 2 la fonction **sample** utilise les coordonnées de texture (interpolées) et l'état d'échantillonnage (ou de *sampling*) pour déterminer la couleur extraite de la texture. Nous parlerons plus en détail de l'échantillonnage en 7.7.
- 3 la couleur de la texture sert de couleur de base pour la valeur ambiante et pour la valeur diffuse. Notez que nous pourrions simplifier notre effet en « oubliant » vAMat et VDMat puisque ceux-ci seront presque toujours à (1.0, 1.0, 1.0, 1.0) pour une utilisation de la texture originale. Je les conserve toutefois, ça nous permettra au besoin de changer un peu le matériau de certains objets sans changer les effets.

7.6 Associer une texture à un objet

Implanter le chargement de la texture (des textures...)

1. Déclarez la variable suivante dans la section *protected* de la classe CMoteur:

```
// Le gestionnaire de texture  
CGestionnaireDeTextures TexturesManager;
```

2. Ajoutez aussi la ligne suivante à la suite des autres `#include` au début du fichier Moteur.h:

```
#include "GestionnaireDeTextures.h"
```

3. Ajoutez la fonction publique **GetTextureManager** à la classe CMoteur, elle nous permettra d'accéder au gestionnaire de texture de n'importe quel objet de notre projet:

```
CGestionnaireDeTextures& GetTextureManager(){ return TexturesManager; }
```

4. Ajoutez la variable **pTextureD3D** et la variable **pSampleState** à la classe CBlocEffet1

```
ID3D11ShaderResourceView* pTextureD3D;  
ID3D11SamplerState* pSampleState;
```

5. Les initialiser à 0 dans le constructeur de CBlocEffet1 (dans le constructeur paramètré, pas celui de défaut).

6. Ajoutez la ligne suivante à la suite des autres « `#include` » au début du fichier BlocEffet1.h:

```
#include "Texture.h"
```

7. Ajoutez la fonction **SetTexture** dans la section *public* de la classe CBlocEffet1:

```
void SetTexture(CTexture* pTexture);
```

8. Implanter la fonction SetTexture:

```
void CBlocEffet1::SetTexture(CTexture* pTexture)
{
    pTextureD3D = pTexture->GetD3DTexture();
}
```

L'état d'échantillonnage

10. Initialisez l'état d'échantillonnage en ajoutant les lignes suivantes à la fin de la fonction CBlocEffet1::InitEffet:

```
// Initialisation des paramètres de sampling de la texture
D3D11_SAMPLER_DESC samplerDesc;

samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_POINT;
samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 1;
samplerDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
samplerDesc.BorderColor[0] = 0;
samplerDesc.BorderColor[1] = 0;
samplerDesc.BorderColor[2] = 0;
samplerDesc.BorderColor[3] = 0;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;

// Création de l'état de sampling
pD3DDevice->CreateSamplerState(&samplerDesc, &pSampleState);
```

Les paramètres de l'état d'échantillonnage donc nos paramètres d'échantillonnage seront décrits plus en détail plus loin.

Activation de la texture lors du rendu

11. Ajoutez l'activation de la texture et de l'état d'échantillonnage dans la fonction CBlocEffet1::Draw, juste avant le rendu de l'objet :

```
// Activation de la texture
ID3DX11EffectShaderResourceVariable* variableTexture;

variableTexture = pEffet->GetVariableByName("textureEntree")-
>AsShaderResource();
variableTexture->SetResource(pTextureD3D);

// Le sampler state
ID3DX11EffectSamplerVariable* variableSampler;
variableSampler = pEffet->GetVariableByName("SampleState")->AsSampler();
variableSampler->SetSampler(0, pSampleState);

// *** Rendu de l'objet
...
```

12. **Modifiez** la fonction CMoteur::InitObjets de façon à ce que notre bloc utilise la texture "UneTexture.dds":

```
bool InitObjets()
{
CBlocEffet1* pBloc;

// Création d'un cube de 2 X 2 X 2 unités
// Le bloc est créé dans notre programme et sur le dispositif
pBloc = new CBlocEffet1( 2, 2, 2, pDispositif );

// Lui assigner une texture
pBloc-
>SetTexture(TexturesManager.GetNewTexture(L"UneTexture.dds",pDispositif));

// Puis, il est ajouté à la scène
ListeScene.push_back(pBloc);

return true;
}
```

Essayez d'exécuter votre programme... mais auparavant copiez dans votre répertoire projet un fichier dds avec l'image de votre choix et renommez le "UneTexture.dds". Utilisez l'outil « **DirectX Texture Tools** » du SDK pour ouvrir une texture par exemple "UneTexture.bmp". Modifiez le format pour A8R8G8B8. Sauvegardez la texture sous le nom « **UneTexture.dds** ».

Utilisez des textures carrées chaque fois que c'est possible (les calculs en sont simplifiés). Les textures de dimension 256×256 sont souvent les plus rapides. Si votre application utilise quatre (4) textures de 128×128, par exemple, vous avez avantage à les combiner pour les placer sur une texture unique de 256×256. Mais essayez tout de même de conserver vos textures les plus petites possible.

Théorie

7.7 L'échantillonnage

L'échantillonnage des textures est un processus consistant à obtenir une couleur à partir d'une texture et de coordonnées de référence. L'échantillonnage le plus simple est effectué « par point » (*point sampling*) et consiste à retourner directement le texel correspondant à la coordonnée logique utilisée. C'est ce que nous avions choisi avec l'option de filtre D3D11_FILTER_MIN_MAG_MIP_POINT.

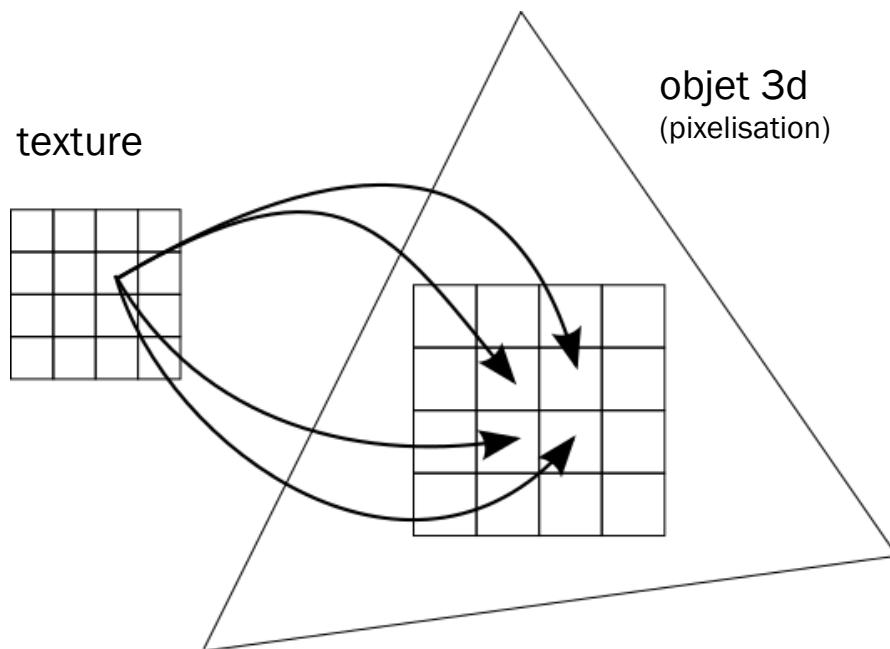
Exemple

- Nous avons une texture de 128 par 128
- Les coordonnées logiques interpolées sont (0.5, 0.5)
- Le texel choisi sera (63, 63) soit (128*0.5 - 1, 128*0.5 - 1)

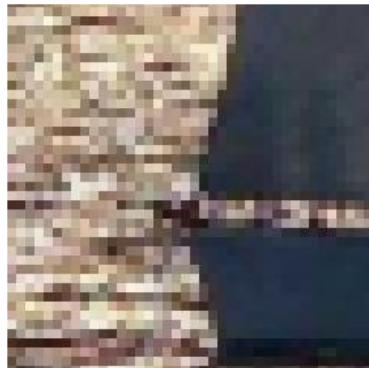
D'autres méthodes d'échantillonnages (filtres) existent, mais nous regarderons auparavant les problèmes potentiels rencontrés lors d'un échantillonnage par point.

Problème #1 – La magnification

La **magnification** est le phénomène qui se produit lorsqu'un texel doit être agrandi sur plusieurs pixels adjacents. En réalité, il s'agit plutôt de l'échantillonnage des pixels qui correspondent au même texel.



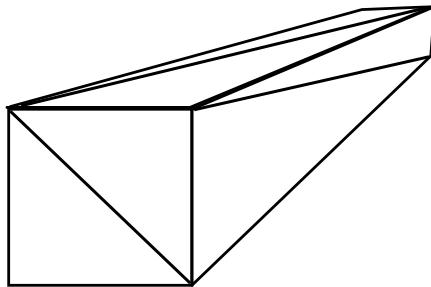
Le résultat donne souvent quelque chose du genre:



Ce phénomène était très fréquent dans les premiers jeux vidéo puisque les textures étaient limitées à des dimensions telles 64 X 64, 32 X 32 ou même 16 X 16.

Aujourd'hui, les cartes modernes DX11 (ou shaders 5.0) nous permettent des dimensions de 16384 X 16384 et ne sont pas limitées à des textures carrées ni à des textures « puissance de 2 » (attention! Certaines cartes graphiques le sont toujours en mode DX9 ou DX10).

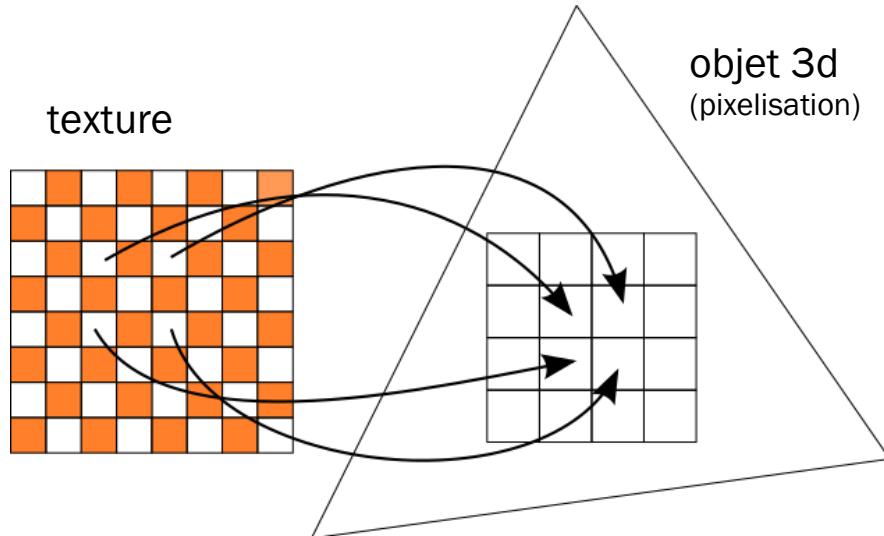
Le problème peut quand même se produire aujourd'hui puisqu'il n'est pas toujours facile de choisir la bonne dimension pour nos textures. De plus un polygone peut être « étiré en perspective » comme ceci:



Dans ce cas, nous retrouverons de la magnification à l'avant-plan et de la **minification** (voir plus loin) à l'arrière-plan.

Problème #2 – La minification

La **minification** est le phénomène qui se produit lorsqu'un texel doit être choisi parmi plusieurs texels adjacents. En réalité, il s'agit plutôt de l'échantillonnage des pixels qui fait en sorte que certains texels sont ignorés.



Dans la figure ci-haut, on voit que les texels de couleur ont été oubliés au profit des texels blancs. Ce n'est qu'une partie du problème puisqu'un léger mouvement de notre objet occasionnera de nouvelles coordonnées logiques qui nous donneront des texels différents. C'est ce que vous pouvez observer sur notre cube (si vous avez choisi l'image du voilier).

L'effet de scintillement de l'eau n'est pas volontaire. 😊

Solutions

Il n'y a pas qu'une seule solution, mais plusieurs que vous pourrez combiner dans certains cas.

Solution 1 — Astuce de base — Règle du 50 %-200 %

Si le rendu de notre polygone nous donne une dimension supérieure à 50 % et inférieure à 200 % de la dimension de la texture, le résultat de l'échantillonnage, même de type point, est intéressant et présente peu d'artéfacts (effets indésirables).

Mais il n'est pas facile de respecter cette règle puisque les objets et/ou la caméra se déplacent et font en sorte que les polygones peuvent sortir de la règle du 50 %-200 % assez facilement.

Solution 2 — Les mipmaps

Le terme **mipmaps** désigne une série de textures de différentes dimensions utilisées pour un même rendu. En fonction du ratio entre le rendu du polygone et la texture, le niveau de mipmap (la dimension) le plus approprié est sélectionné afin d'appliquer la règle 50 %-200 %.

Les mipmaps permettent de limiter les effets de scintillement sur les textures minifiées, elles permettent aussi de préparer à l'avance les différentes versions de notre texture avec des logiciels offrant des options de dimensionnement (filtres et autres) difficilement utilisables en temps réel.

Par contre un mipmap occupe plus d'espace en mémoire vidéo. Dans certains cas, le changement de niveau de mipmap peut être perçu par l'utilisateur. Les **filtres** (voir plus loin) permettent de régler ce problème dans presque tous les cas, mais...



Solution 3 – Le filtrage

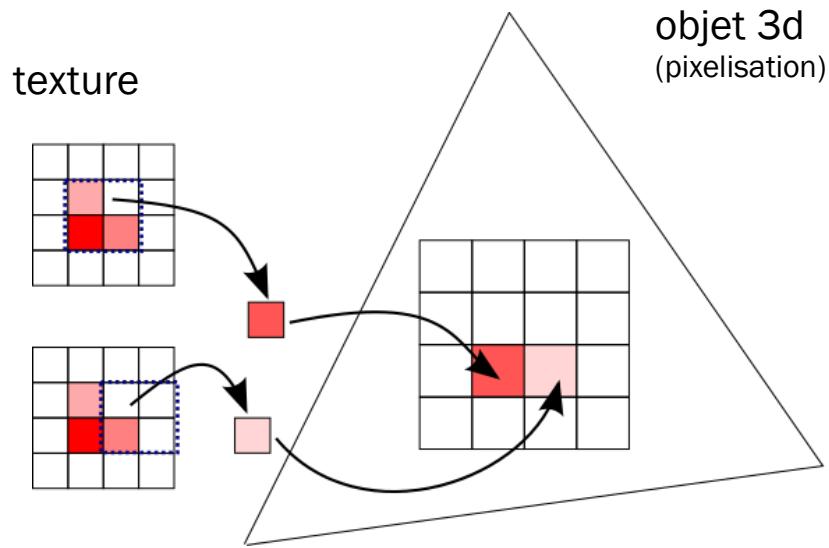
Différents types de filtrage existent pour améliorer les résultats de nos rendus de textures. Les filtres présents sur les cartes graphiques combinent habituellement un **échantillonnage** de texels suivi d'une **interpolation** permettant d'obtenir un seul pixel résultant.

Filtrage linéaire (bilinéaire)

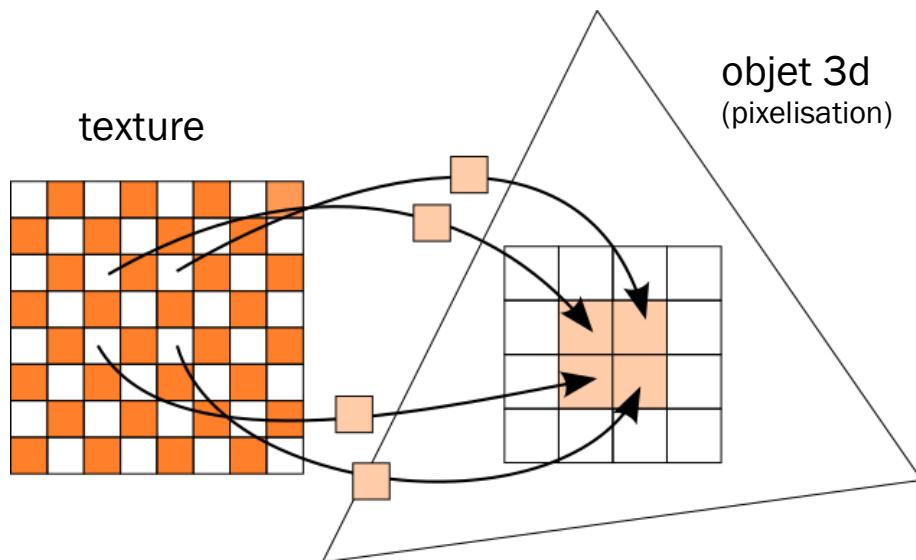
Le premier type présent sur la plupart des cartes graphiques est le **filtrage linéaire**. Lorsque nous travaillons avec des textures, il s'agit en réalité de **filtrage bilinéaire** (sur deux axes).

Au moyen d'une coordonnée d'application de texture (interpolée puisque nous sommes dans le pixel shader), le filtre détermine les quatre texels les plus près de la coordonnée puis utilise une interpolation bilinéaire [Möller et Haines 1 pages 158-159] afin d'obtenir une couleur résultante. Tant que nos polygones respectent la règle du 50 %-200 %, le résultat est très bien. C'est pourquoi le mipmapping est presque toujours requis.

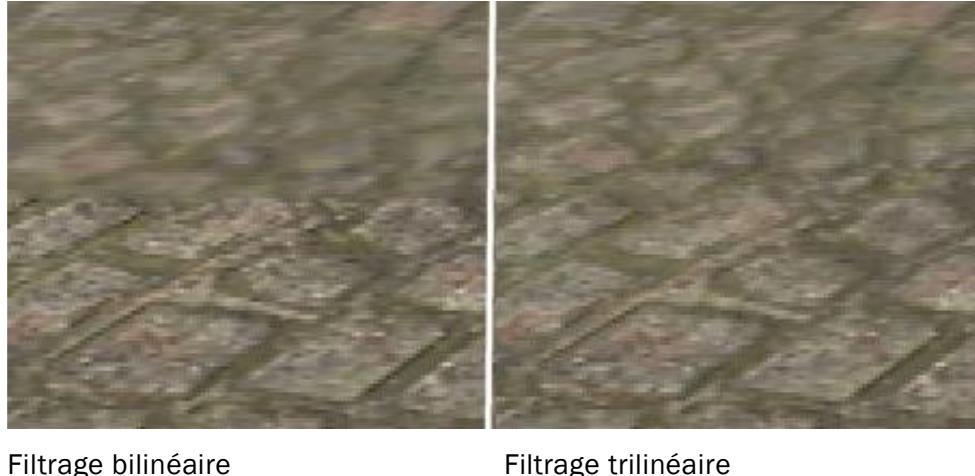
La figure suivante illustre l'utilisation du filtrage bilinéaire pour un problème de magnification. Notez que les coordonnées de texture identifient le même texel de base pour les deux pixels résultants, mais n'utilisent pas les mêmes texels pour le mélange des couleurs. Si nous avions obtenu les mêmes texels, par exemple pour certaines situations de magnification hors du 50 %-200 %, l'interpolation aurait tout de même obtenu un mélange de couleurs légèrement différent.



La figure suivante reprend notre problème de minification, mais cette fois avec filtrage bilinéaire et en utilisant les quatre texels les plus près, notez que le résultat est une couleur intermédiaire (la même dans mon cas, mais elle aurait pu varier un peu). Le résultat semble très différent de la texture originale, mais en fait c'est la même chose (ou presque). Essayez de réduire une image avec un « bon » logiciel de traitement d'image et vous obtiendrez un résultat semblable.



Certains artefacts peuvent tout de même se produire avec le filtrage bilinéaire, entre autres lorsqu'un objet est très étiré en perspective ou quand une partie de notre objet change de niveau de mipmap. Dans ce dernier cas, une démarcation apparaîtra puisque le « flou » associé à la texture plus petite peut être différent de celui de la plus grande.



Filtrage bilinéaire

Filtrage trilinéaire

Filtrage trilinéaire

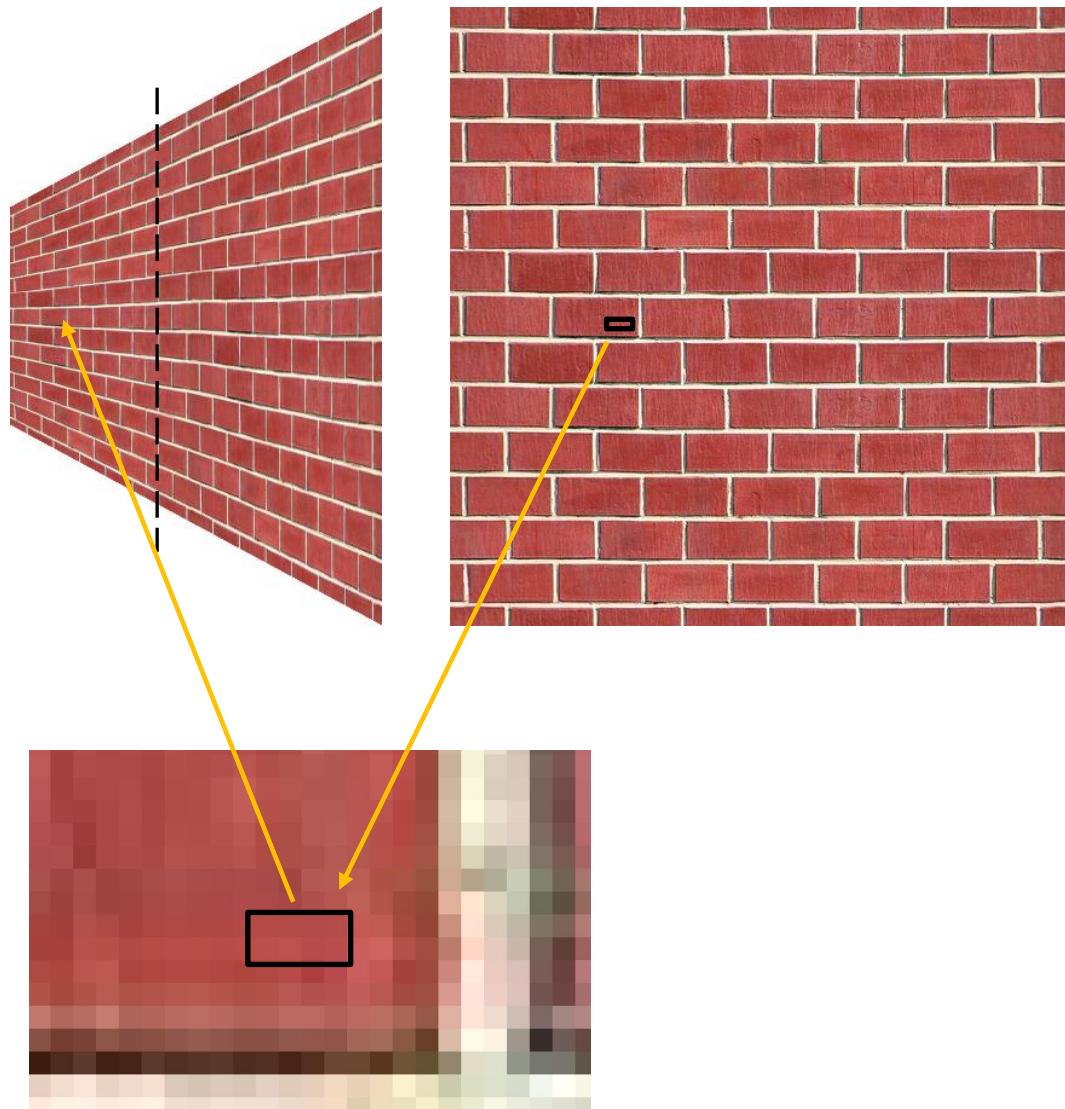
Le **filtrage trilinéaire** est en fait une extension du filtrage bilinéaire qui utilisera le mipmap comme troisième axe. En pratique, l'interpolation bilinéaire décrite plus haut est effectuée sur les deux niveaux de mipmap les plus près ce qui nous donne deux texels. Ces deux texels sont interpolés de nouveau pour obtenir le texel final.

Notez que les calculs sont de l'ordre de 2 fois plus important que lors du filtrage bilinéaire.

Filtrage anisotropique

Les techniques de filtrage vues jusqu'à maintenant étaient « **isotropiques** », c'est-à-dire que l'échantillonnage utilisé pour le filtre était identique dans toutes les directions. Nous avons parlé plus haut des problèmes résultants de l'affichage en perspective d'un objet. Dans ce cas, l'échantillonnage isotropique ne correspond plus à la vue qu'un utilisateur a d'un objet.

Le **filtrage anisotropique** ajoute un facteur permettant de tenir compte du fait que la texture visualisée est moins large (ou moins haute) que la texture originale. Les cartes graphiques implantent généralement un filtrage anisotropique simplifié où un facteur « largeur/hauteur » est calculé et permet d'ajuster l'échantillonnage en conséquence. Plus de texels (2X, 4X, 8X par exemple) sont sélectionnés pour l'élément (largeur ou hauteur) défavorisé. Beaucoup de nouvelles cartes graphiques permettent jusqu'à 16X (au prix des calculs correspondants). Puis l'interpolation bilinéaire (ou trilinéaire) est effectuée.



Dans la figure ci-haut, un filtrage anisotropique 2X (sur la largeur) est utilisé. Donc 8 texels sont utilisés sur chaque niveau de mip (dans le cas du filtrage trilinéaire).

Notez que les calculs sont de l'ordre de 2 fois (4 fois, etc) plus importants que lors du filtrage trilinéaire.

Autres techniques de filtrage

D'autres techniques de filtrage existent (cubique, bicubique par exemple). Certaines utilisent d'autres méthodes d'échantillonnage, d'autres utilisent d'autres formules d'interpolation.

Il est possible de les implanter nous-mêmes dans les shaders tout comme nous aurions pu écrire notre propre version des filtrages précédents.

7.8 L'échantillonnage dans notre application

Dans notre cas, l'échantillonnage est déjà implanté dans notre application. En effet, l'utilisation des textures demande l'utilisation de la fonction **Texture2D::Sample** (dans notre cas). Cette fonction HLSL (shaders 4 et +) remplace la fonction **tex2d** des versions précédentes. Notez que **tex2d** est encore disponible en utilisant certaines options de compilation des shaders (*legacy compile option*).

Notre programme n'utilise pour le moment un échantillonnage par point donc aucun filtre, **D3D11_FILTER_MIN_MAG_MIP_POINT**. C'est ce que vous devrez utiliser si vous voulez utiliser votre propre algorithme de filtrage.

Filtrage linéaire (bilinéaire)

- Pour obtenir le filtrage linéaire, il suffit de modifier la ligne suivante dans la fonction **CBlocEffet1::InitEffet**:

```
samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_POINT;
```

pour

```
samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
```

IMPORTANT

Regardez dans la documentation du SDK les paramètres de la structure **D3D11_SAMPLER_DESC**

Essayez l'application. Vous noterez une grande amélioration (diminution du scintillement) et la prochaine étape n'améliorera pas beaucoup la qualité de l'affichage.

Filtrage trilinéaire

Le filtrage trilinéaire est déjà défini dans notre état de *sampling* avec **D3D11_FILTER_MIN_POINT_MAG_MIP_LINEAR** (nous aurions dû utiliser **D3D11_FILTER_MIN_MAG_LINEAR_MIP_POINT** pour définir un vrai filtrage bilinéaire).

- Pour obtenir le filtrage trilinéaire, il nous faut d'abord une texture 3D c'est-à-dire une texture comprenant plusieurs niveaux de mipsmaps. Il est possible de créer ces niveaux de mipsmaps au moment du chargement de la texture, mais il est plus logique (et plus rapide) de créer notre mipmap à l'avance. Pour ce faire, j'ai utilisé de nouveau l'outil « **DirectX Texture Tools** » du SDK. Cet outil génère les mipsmaps de façon simple. D'autres outils graphiques vous permettraient de mieux paramétrier la création des mipsmaps. Mais c'est suffisant pour nos besoins du moment. Sauvegardez le mipmap toujours sous le nom « **UneTexture.dds** ».

Essayez l'application. Vous ne noterez pas une grande amélioration par rapport au filtrage bilinéaire puisque nous avions déjà assez bien réglé le problème de minification.

Filtrage anisotropique

1. Pour obtenir le filtrage anisotropique, il suffit de modifier la ligne suivante dans la fonction CBlocEffet1::InitEffet:

```
samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
```

pour

```
samplerDesc.Filter = D3D11_FILTER_ANISOTROPIC;
```

2. Toujours dans la fonction CBlocEffet1::InitEffet, modifiez la ligne suivante:

```
samplerDesc.MaxAnisotropy = 1;
```

pour

```
samplerDesc.MaxAnisotropy = 4;
```

J'utilise 4 pour limiter les calculs. On utilise généralement des puissances de 2.

Essayez l'application. Vous ne noterez pas une grande amélioration par rapport aux deux filtrages précédents. Toutefois les derniers scintillements devraient avoir disparu.

8. Objets 3D – Les maillages

Ce chapitre couvre l'intégration d'objets 3D plus évolués dans nos applications. Plusieurs techniques permettent la création d'objets plus intéressants que le cube que nous utilisions jusqu'à maintenant. Il est possible de créer au moyen d'algorithmes des sphères, des cylindres, des cônes, mais la création d'objets plus complexes demande des algorithmes trop lourds pour nos objectifs actuels...

Objectifs du chapitre

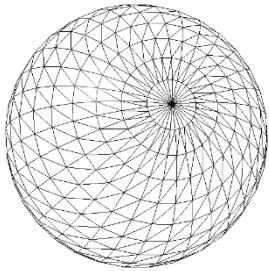
- ✓ Déterminer un format d'objet 3D (maillage ou *mesh*) qui nous convient pour notre programmation
- ✓ Utiliser des maillages en ressources (fichiers ou autres).
- ✓ Choisir un ou plusieurs formats pour nos objets.
- ✓ Comprendre le chargement et la conversion de maillages.
- ✓ Construire un maillage en mémoire système et graphique.
- ✓ Effectuer le rendu du maillage.

8. Maillages et objets 3D

Théorie

8.1 Les maillages

Heureusement pour nous, il existe des logiciels de modélisation 3D (3D Studio Max, Maya, Strata, Rhino, MilkShape, Blender, etc.) qui permettent de créer des objets 3D de façon plus agréable. Ces logiciels ont même la possibilité de créer des scènes entières, de les éclairer, d'y ajouter des effets spéciaux et même de construire certaines animations (de type cinéma, pas interactif).



Un objet 3D exporté par un de ces logiciels portera le nom de **maillage**. L'usage a depuis longtemps associé le terme anglais **mesh** (maillage) à un objet 3D. À l'origine, et même aujourd'hui, les maillages étaient tout simplement une façon de décrire un assemblage de primitives (triangles ou autres). Ce terme provient du fait que lorsque l'on regarde un objet 3D dans sa représentation **fil de fer** (*wireframe*), l'objet ressemble à un grand filet ou à une cotte de mailles. La représentation fil de fer ne dessine que les arêtes des polygones et ne fait pas de remplissage des surfaces, souvent, elle ne fait même pas de retrait des surfaces cachées. Cette représentation est tout ce que les premiers logiciels 3D étaient capables de faire.

L'utilisation d'objets 3D est une tâche complexe qu'on peut toutefois décrire de façon assez simple, par exemple pour un objet 3D sans animation:

- il suffit de lire un fichier. Il faut bien sûr en connaître le format et écrire l'algorithme de lecture (!!);
- en cours de lecture, nous devons générer des sommets et des polygones pour recréer l'objet. Nous devons aussi conserver l'organisation des matériaux et des textures;
- nous devons écrire la fonction de rendu selon les paramètres construits précédemment.

Formats

Chaque logiciel de modélisation 3D possède son propre format (format propriétaire). Mais la plupart d'entre eux peuvent importer ou exporter en d'autres formats. L'un des formats parmi les plus répandus est encore le format **.3ds**, originaire de **3D Studio version DOS** (années 1990, de Autodesk). 3D Studio fut longtemps le logiciel 3D le plus utilisé et ses formats sont donc bien connus. Dans ce chapitre, nous utiliserons le format **.obj** pour importer les éléments qui nous intéressent.

8.2 Construire notre format d'objet

Il ne s'agit pas ici de construire une nouvelle spécification pour des formats de fichier de maillages, mais plutôt de se construire un format de travail que nous pourrons utiliser dans notre application. Dans notre cas, ce format se retrouvera dans la classe **CObjetMesh** (voir section 8.4).

Nous avons déjà certains éléments que nous avons utilisés pour le bloc (version du chapitre 7) soient:

1. Un « vertex buffer » en mémoire graphique et un objet pour y accéder;
2. Un « index buffer » en mémoire graphique et un objet pour y accéder;
3. Une matrice de transformation dans le monde;
4. Une texture en mémoire graphique et un objet pour y accéder;
5. Un effet en mémoire graphique (nos shaders) et un objet pour y accéder;
6. La possibilité de spécifier une valeur diffuse et une valeur ambiante pour notre objet (une ébauche de matériau).

Les objets de type **mesh** ont comme principale caractéristique de pouvoir être composés de sous-objets (sous-ensemble, *submesh* ou *subset* en anglais). Chaque sous-objet possède des sommets, des indices, et habituellement (mais pas toujours) un matériau et une texture.

Voici donc ce que nous planterons dans notre classe de maillage:

Pour le maillage au complet:

1. Un « vertex buffer » en mémoire graphique et un objet pour y accéder (**pVertexBuffer**). Le *vertex buffer* étant partagé par tous les sous-objets. Nous y reviendrons lors de l'écriture de la fonction **Draw**;
2. Un « index buffer » en mémoire graphique et un objet pour y accéder (**pIndexBuffer**). L'*index buffer* étant partagé par tous les sous-objets. Nous y reviendrons aussi lors de l'écriture de la fonction **Draw**;
3. Une matrice de transformation dans le monde (**matWorld**);
4. Le nombre de sous-objets (**NombreSubmesh**);
5. Un vecteur de matériaux (**Material**);
6. Un effet en mémoire graphique (nos shaders) et un objet pour y accéder (référencé par **pEffet**);

Pour chaque sous-objet:

7. Un indice de départ dans l'index buffer. L'indice du sous-objet suivant servant de fin. Les indices de tous les sous-objets se retrouvent dans un vecteur (**SubmeshIndex**). Un indice supplémentaire est inséré pour la fin.
8. Un numéro de matériau (un indice). Cet indice sera placé dans un vecteur (**SubmeshMaterialIndex**);
9. La possibilité d'utiliser les valeurs du matériau lors du rendu de l'objet (via les paramètres de l'effet).

Pour chaque matériau:

10. Le nom du fichier texture. *Utilisé seulement lors du chargement.*
11. Un pointeur sur un objet permettant de référencer la texture en mémoire graphique (**pTextureD3D**) ;
12. Le nom du matériau. Ce nom sert de clé lors l'association des matériaux aux sous-objets. Certains privilégient l'utilisation d'une clé numérique, mais comme beaucoup de formats de maillages (comme les .OBJ) utilisent une chaîne de caractères, c'est ce que nous ferons. *Utilisé principalement lors du transfert du modèle dans notre objet.*
13. Des valeurs de matériau soient les attributs **Ambient**, **Diffuse** et **Specular**.
14. Un puissance pour la spécularité (voir **Illumination spéculaire** au chapitre 6).
15. Un indicateur de transparence. Cet indicateur ne sera pas utilisé dans ce chapitre. Il pourrait nous servir à ordonner nos sous-objets et à dessiner les sous-objets transparents (ou partiellement transparents) après les sous-objets opaques.

Bien sûr, d'autres possibilités s'offrent à nous. Des suggestions:

- chaque sous-objet pourrait utiliser un effet ou une technique différente.
- chaque sous-objet pourrait avoir sa matrice de transformation dans le monde. On pourrait ainsi animer les roues d'un véhicule. (Les personnages utilisent une autre méthode).

Les applications professionnelles (dont les jeux vidéo) privilégient souvent l'utilisation de plusieurs classes de maillages spécialisées plutôt qu'une « super-classe » pouvant tout faire. Il est ainsi plus facile d'optimiser.

8.3 Une classe pour nos objets *mesh*

Pour mettre en place les éléments que nous venons de déterminer, nous allons construire une petite classe. Cette classe jouera un rôle semblable à la classe CBloc c'est-à-dire qu'elle sera utilisée pour créer l'objet et pour le dessiner.

Les éléments allant sur la carte graphique sont aussi créés et stockés.

Elle aussi sera une classe dérivée de CObjet3D. Notez que tout comme le bloc, c'est une classe spécifique (Windows et DirectX), mais avec des interfaces génériques. Pour les besoins de mes exemples, je n'ai pas créé de classes génériques d'objets, mais libre à vous de le faire...

La classe

Ajouter la classe CObjetMesh à notre projet, vous pouvez reprendre le projet tel que nous l'avions laissé à la fin du chapitre précédent.

Le constructeur

1. Le constructeur devra transférer le maillage (déjà chargés par un objet de classe **CChargeurOBJ**, nous y reviendrons en section 87). Tout comme pour le bloc, nous utiliserons un constructeur paramétré et le « chargeur » ainsi que le dispositif seront passés comme paramètres. Voici le point de départ de la classe CObjetMesh (en partie générée à la page précédente) que nous retrouverons dans ObjetMesh.h :

```
#pragma once
#include "objet3d.h"
#include "dispositifD3D11.h"
#include "d3dx11effect.h"
#include "chargeur.h"

#include <vector>
using namespace std;

namespace PM3D
{
    class CObjetMesh: public CObjet3D
    {
    public:
        CObjetMesh(IChargeur& chargeur, CDispositifD3D11* pDispositif);
        virtual ~CObjetMesh(void);
    private:
        CObjetMesh();
    };
}
```

Ces "include" seront nécessaires pour les éléments que nous ajouterons à notre objet.

2. Et voici l'implantation de la programmation dans le fichier **Objet-Mesh.cpp**:

```
#include "StdAfx.h"

#include <string>

#include "ObjetMesh.h"
#include "moteurWindows.h"
#include "util.h"
#include "resource.h"

using namespace UtilitairesDX;

namespace PM3D
{
    CObjetMesh::CObjetMesh(IChargeur& chargeur, CDispositifD3D11* _pDispositif)
    {
    }

    CObjetMesh::~CObjetMesh(void)
    {
    }
}
```

3. Déclarez les variables membres suivantes dans la classe CObjetMesh:

```
private:
    // **** Données membres
    XMATRIX matWorld;           // Matrice de transformation dans le monde
    float rotation;

    // Pour le dessin
    CDispositifD3D11* pDispositif; // On prend en note le dispositif

    ID3D11Buffer* pVertexBuffer;
    ID3D11Buffer* pIndexBuffer;

    // Les sous-objets
    int NombreSubmesh;          // Nombre de sous-objets dans le mesh
    vector<int> SubmeshMaterialIndex; // Index des matériaux
    vector<int> SubmeshIndex;      // Index des sous-objets

    vector<CMaterial> Material; // Vecteur des matériaux

    // Pour les effets et shaders
    ID3D11SamplerState* pSampleState;
    ID3D11Buffer* pConstantBuffer;
    ID3DX11Effect* pEffet;
    ID3DX11EffectTechnique* pTechnique;
    ID3DX11EffectPass* pPasse;
    ID3D11InputLayout* pVertexLayout;
```

Certains éléments ressemblent à ce que nous avions placé dans la classe CBloc, entre autres les variables membres **matWorld** et **pDispositif**. Il faudra peut-être un jour définir une classe de base qui regroupera ces attributs...(*à suivre*).

4. Déclarez aussi dans la classe **CObjetMesh** les deux sous-classes suivantes:

```
private:  
    class CSommetMesh  
    {  
public:  
    CSommetMesh();  
    CSommetMesh(XMFLOAT3 _position, XMFLOAT3 _normal,  
                XMFLOAT2 _coordTex = XMFLOAT2(0,0));  
  
public:  
    static UINT numElements;  
    static D3D11_INPUT_ELEMENT_DESC layout[];  
  
    XMFLOAT3 position;  
    XMFLOAT3 normal;  
    XMFLOAT2 coordTex;  
};  
  
class CMaterial  
{  
public:  
    string NomFichierTexture;  
    string NomMateriau;  
    ID3D11ShaderResourceView* pTextureD3D;  
  
    XMFLOAT4 Ambient;  
    XMFLOAT4 Diffuse;  
    XMFLOAT4 Specular;  
    float Puissance;  
    bool transparent;  
  
    CMaterial()  
    {  
        Ambient = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);  
        Diffuse = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);  
        Specular = XMFLOAT4(0.0f, 0.0f, 0.0f, 1.0f);  
        transparent = false;  
        Puissance = 0;  
        pTextureD3D = NULL;  
        NomFichierTexture = "";  
    }  
};
```

La classe **CSommetMesh** est une « reprise » locale de la classe **CSommetBloc** utilisée précédemment. Je préférerais ici avoir une classe locale plus facile à consulter et à modifier.

La classe **CMaterial**, comme son nom l'indique, sert à définir les matériaux. Elle gagnera peut-être une envergure plus globale lorsque nous aurons plusieurs types d'objets utilisant les matériaux. Vous noterez que

la classe CMaterial nous sert principalement de structure de données pour faciliter l'accès aux éléments du matériau (d'où les variables publiques).

- Ajoutez les deux lignes suivantes dans la section PM3D de Objet-Mesh.cpp. Nous initialisons les variables statiques de CSommetMesh:

```
// Definir l'organisation de notre sommet
D3D11_INPUT_ELEMENT_DESC CObjetMesh::CSommetMesh::layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
D3D11_INPUT_PER_VERTEX_DATA, 0}
};

UINT CObjetMesh::CSommetMesh::numElements;
```

- Modifiez le constructeur de **CObjetMesh** pour qu'il contienne le code suivant:

```
CObjetMesh::CObjetMesh(const IChargeur& chargeur, CDispositifD3D11*
_pDispositif)
    : pDispositif(_pDispositif) // prendre en note le dispositif
{
    // Placer l'objet sur la carte graphique
    TransfertObjet(chargeur);

    // Initialisation de l'effet
    InitEffet();

    matWorld = XMMatrixIdentity();

    rotation = 0.0f;
}
```

Le « chargeur » a déjà lu et interprété le fichier, il faut le transférer dans notre objet.

La fonction TransfertObjet

- Déclarez la fonction TransfertObjet de la façon suivante dans la section *private* des fonctions de la classe CObjetMesh:

```
void TransfertObjet(IChargeur& chargeur);
```

- Puis définissez-la ainsi:

```
void CObjetMesh::TransfertObjet(const IChargeur& chargeur)
{
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    // 1. SOMMETS a) Créations des sommets dans un tableau temporaire
{
```

```

const size_t nombreSommets = chargeur.GetNombreSommets();
std::unique_ptr<CSommetMesh[]> ts(new CSommetMesh[nombreSommets]);

for (uint32_t i = 0; i < nombreSommets; ++i)
{
    ts[i].position = chargeur.GetPosition(i);
    ts[i].normal = chargeur.GetNormale(i);
    ts[i].coordTex = chargeur.GetCoordTex(i);
}

// 1. SOMMETS b) Cr ation du vertex buffer et copie des sommets
D3D11_BUFFER_DESC bd;
ZeroMemory(&bd, sizeof(bd));

bd.Usage = D3D11_USAGE_IMMUTABLE;
bd.ByteWidth = static_cast<uint32_t>(sizeof(CSommetMesh) * nombreSommets);
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;

D3D11_SUBRESOURCE_DATA InitData;
ZeroMemory(&InitData, sizeof(InitData));
InitData.pSysMem = ts.get();
pVertexBuffer = nullptr;

DXESSAYER(pD3DDevice->CreateBuffer(&bd, &InitData, &pVertexBuffer),
DXE_CREATIONVERTEXBUFFER);
}

// 2. INDEX - Cr ation de l'index buffer et copie des indices
//           Les indices  tant habituellement des entiers, j'ai
//           pris directement ceux du chargeur, mais attention au
//           format si vous avez autre chose que DXGI_FORMAT_R32_UINT
{
D3D11_BUFFER_DESC bd;
ZeroMemory(&bd, sizeof(bd));

bd.Usage = D3D11_USAGE_IMMUTABLE;
bd.ByteWidth = static_cast<uint32_t>(sizeof(uint32_t) * chargeur.GetNombreIndex());
bd.BindFlags = D3D11_BIND_INDEX_BUFFER;
bd.CPUAccessFlags = 0;

D3D11_SUBRESOURCE_DATA InitData;
ZeroMemory(&InitData, sizeof(InitData));
InitData.pSysMem = chargeur.GetIndexData();
pIndexBuffer = nullptr;

DXESSAYER(pD3DDevice->CreateBuffer(&bd, &InitData, &pIndexBuffer),
DXE_CREATIONINDEXBUFFER);
}

// 3. Les sous-objets
NombreSubset = chargeur.GetNombreSubset();

//   D but de chaque sous-objet et un pour la fin
SubsetIndex.reserve(NombreSubset);
chargeur.CopieSubsetIndex(SubsetIndex);

// 4. MATERIAUX
// 4a) Cr er un mat riaux de d faut en index 0
//       Vous pourriez changer les valeurs, j'ai conserv 
//       celles du constructeur
Material.reserve(chargeur.GetNombreMaterial() + 1);

```

2

3

4

```

        Material.emplace_back(CMaterial());

        // 4b) Copie des matériaux dans la version locale
        for (int32_t i = 0; i < chargeur.GetNombreMaterial(); ++i)
        {
            CMaterial mat;

            chargeur.GetMaterial(i, mat.NomFichierTexture,
                mat.NomMateriau,
                mat.Ambient,
                mat.Diffuse,
                mat.Specular,
                mat.Puissance);

            Material.push_back(mat);
        }

        // 4c) Trouver l'index du materiau pour chaque sous-ensemble
        SubsetMaterialIndex.reserve(chargeur.GetNombreSubset());
        for (int32_t i = 0; i < chargeur.GetNombreSubset(); ++i)
        {
            int32_t index;
            for (index = 0; index < Material.size(); ++index)
            {
                if (Material[index].NomMateriau == chargeur.GetMaterialName(i))
break;
            }

            if (index >= Material.size()) index = 0; // valeur de défaut
            SubsetMaterialIndex.push_back(index);
        }

        // 4d) Chargement des textures
        CGestionnaireDeTextures& TexturesManager =
CMoteurWindows::GetInstance().GetTextureManager();

        for (uint32_t i = 0; i < Material.size(); ++i)
        {
            if (Material[i].NomFichierTexture.length() > 0)
            {
                const std::wstring ws(Material[i].NomFichierTexture.begin(),
Material[i].NomFichierTexture.end());
                Material[i].pTextureD3D = TexturesManager.GetNewTexture(ws.c_str(),
pDispositif)->GetD3DTexture();
            }
        }
    }
}

```

5

6

7

Notes:

1

La création et le chargement du vertex buffer ressemble beaucoup à celui du bloc sauf qu'au lieu de créer les éléments individuellement, nous utilisons une boucle et les fonctions de la classe **IChargeur** : **GetPosition**, **GetNormale** et **GetCoordTex**. IChargeur étant la classe de base de la classe CChargeurOBJ. IChargeur est une **interface** de transfert vers la classe CObjet-Mesh. Notez aussi l'utilisation de sommets de classe CSommet-Mesh au lieu de CSommetBloc.

2 La création et le chargement de l'*index buffer*. Nous devons copier les indices dans un tableau temporaire, car l'organisation d'un fichier .obj est très différente de celle utilisée par les cartes graphiques. Nous construisons aussi le vecteur des indices de départ.

3 Nous prenons en note le nombre de sous-objets.

4 Un matériau de défaut est créé et placé en position 0 du vecteur des matériaux. Les sous-objets devraient tous avoir des matériaux, mais...

5 Nous créons les matériaux locaux.

6 Association des matériaux aux sous-objets.

7 Tout comme pour le bloc, il faut charger les textures en mémoire graphique. Le gestionnaire de textures s'assure qu'il n'y a pas de doublons. Notez qu'une texture par matériau peut être chargée, mais il est possible qu'un matériau n'ait pas de texture.

La fonction InitEffet

- Déclarez et définissez la fonction **InitEffet** dans la section *private*. La fonction InitEffet est presque identique à celle que nous avons utilisée au chapitre précédent dans la classe CBlocEffet1. La seule différence est l'utilisation de la classe CSommetMesh pour représenter nos sommets.

Déclaration

```
void InitEffet();
```

Définition

```
void CObjectMesh::InitEffet()
{
    // Compilation et chargement du vertex shader
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    // Création d'un tampon pour les constantes du VS
    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));

    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(ShadersParams);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    DXEssayer(pDXDDevice->CreateBuffer(&bd, nullptr, &pConstantBuffer));

    // Pour l'effet
    ID3DBlob* pFXBlob = nullptr;

    DXEssayer(D3DCompileFromFile(L"MiniPhong.fx", 0, 0, 0,
        "fx_5_0", 0, 0, &pFXBlob, 0),
        DXE_ERREURCREATION_FX);

    D3DX11CreateEffectFromMemory(pFXBlob->GetBufferPointer(), pFXBlob-
    >GetBufferSize(), 0, pD3DDevice, &pEffet);

    pFXBlob->Release();

    pTechnique = pEffet->GetTechniqueByIndex(0);
    pPasse = pTechnique->GetPassByIndex(0);

    // Crée l'organisation des sommets pour le VS de notre effet
    D3DX11_PASS_SHADER_DESC effectVSDesc;
    pPasse->GetVertexShaderDesc(&effectVSDesc);

    D3DX11_EFFECT_SHADER_DESC effectVSDesc2;
    effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
&effectVSDesc2);

    const void *vsCodePtr = effectVSDesc2.pBytecode;
    const uint32_t vsCodeLen = effectVSDesc2.BytecodeLength;

    pVertexLayout = nullptr;

    CSommetMesh::numElements = ARRAYSIZE(CSommetMesh::layout);
    DXEssayer(pDXDDevice->CreateInputLayout(CSommetMesh::layout,
        CSommetMesh::numElements,
        vsCodePtr,
```



```

    vsCodeLen,
    &pVertexLayout),
    DXE_CREATIONLAYOUT);

// Initialisation des paramètres de sampling de la texture
D3D11_SAMPLER_DESC samplerDesc;

samplerDesc.Filter = D3D11_FILTER_ANISOTROPIC;
samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 4;
samplerDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
samplerDesc.BorderColor[0] = 0;
samplerDesc.BorderColor[1] = 0;
samplerDesc.BorderColor[2] = 0;
samplerDesc.BorderColor[3] = 0;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;

// Création de l'état de sampling
pD3DDevice->CreateSamplerState(&samplerDesc, &pSampleState);
}

```

2. Déclarez la structure **ShadersParam** dans la section **private** de la classe CObjetMesh

```

struct ShadersParams
{
    XMATRIX matWorldViewProj; // la matrice totale
    XMATRIX matWorld;        // matrice de transformation dans le monde
    XMVECTOR vLumiere;       // la position de la source d'éclairage (Point)
    XMVECTOR vCamera;         // la position de la caméra
    XMVECTOR vAEcl;           // la valeur ambiante de l'éclairage
    XMVECTOR vAMat;           // la valeur ambiante du matériau
    XMVECTOR vDEcl;           // la valeur diffuse de l'éclairage
    XMVECTOR vDMat;           // la valeur diffuse du matériau
};

```

La fonction Draw

La fonction **Draw** joue évidemment le même rôle que la fonction `Draw` de `CBloc`, mais nous noterons beaucoup de différences:

1. Ajoutez la déclaration de la fonction `Draw` dans la section *publique* de la classe `CObjetMesh`:

```
void Draw();
```

2. Implanterez-la de la façon suivante:

```
void CObjetMesh::Draw()
{
    // Obtenir le contexte
    ID3D11DeviceContext* pImmediateContext = pDispositif->GetImmediateContext();

    // Choisir la topologie des primitives
    pImmediateContext->IASetPrimitiveTopology( D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

    // input layout des sommets
    pImmediateContext->IASetInputLayout( pVertexLayout );

    // Index buffer
    pImmediateContext->IASetIndexBuffer( pIndexBuffer, DXGI_FORMAT_R32_UINT, 0 );

    // Vertex buffer
    UINT stride = sizeof( CSommetMesh );
    UINT offset = 0;
    pImmediateContext->IASetVertexBuffers( 0, 1, &pVertexBuffer, &stride, &offset );

    // Initialiser et sélectionner les « constantes » de l'effet
    ShadersParams sp;
    XMATRIX viewProj = CMoteurWindows::GetInstance().GetMatViewProj();

    sp.matWorldViewProj = XMMatrixTranspose(matWorld * viewProj );
    sp.matWorld = XMMatrixTranspose(matWorld);

    sp.vLumiere = XMVectorSet( -10.0f, 10.0f, -15.0f, 1.0f );
    sp.vCamera = XMVectorSet( 0.0f, 3.0f, -5.0f, 1.0f );
    sp.vAEcl = XMVectorSet( 0.2f, 0.2f, 0.2f, 1.0f );
    sp.vDEcl = XMVectorSet( 1.0f, 1.0f, 1.0f, 1.0f );
    sp.vSEcl = XMVectorSet( 0.6f, 0.6f, 0.6f, 1.0f );

    // Le sampler state
    ID3DX11EffectSamplerVariable* variableSampler;
    variableSampler = pEffet->GetVariableByName("SampleState")->AsSampler();
    variableSampler->SetSampler(0, pSampleState);
}
```

1

2

3

```

// Dessiner les sous-objets non-transparents
for(int i = 0; i < NombreSubmesh; ++i)
{
    4a int indexStart = SubmeshIndex[i];
    int indexDrawAmount = SubmeshIndex[i+1] - SubmeshIndex[i];
    4b if (indexDrawAmount)
    {
        4c     sp.vAMat = XMLoadFloat4(&Material[SubmeshMaterialIndex[i]].Ambient);
        sp.vDMat = XMLoadFloat4(&Material[SubmeshMaterialIndex[i]].Diffuse);
        sp.vSMat = XMLoadFloat4(&Material[SubmeshMaterialIndex[i]].Specular);
        sp.puissance = Material[SubmeshMaterialIndex[i]].Puissance;

        // Activation de la texture ou non
        if (Material[SubmeshMaterialIndex[i]].pTextureD3D != nullptr)
        {
            ID3DX11EffectShaderResourceVariable* variableTexture;
            variableTexture =
                pEffet->GetVariableByName("textureEntree")->AsShaderResource();

            variableTexture->SetResource(Material[SubmeshMaterialIndex[i]].pTextureD3D);
            sp.bTex = 1;
        }
        else
        {
            sp.bTex = 0;
        }
    4e // IMPORTANT pour ajuster les param.
    pPasse->Apply(0, pImmediateContext);

    // Nous n'avons qu'un seul CBuffer
    4f ID3DX11EffectConstantBuffer* pCB = pEffet->GetConstantBufferByName("param");
    pCB->SetConstantBuffer(pConstantBuffer);
    pImmediateContext->UpdateSubresource( pConstantBuffer, 0, nullptr, &sp, 0, 0 );
    4g pImmediateContext->DrawIndexed( indexDrawAmount, indexStart, 0 );
    }
}
}

```

1

À l'exception de l'utilisation de sommets de classe CSommetMesh, la première partie de **Draw** est identique à ce que nous avions fait dans la fonction Draw de CBlocEffet1 soit :

- le choix de la topologie (étape **Input Assembler**)
- le choix de l'organisation des sommets, le *layout* (étape **Input Assembler**)
- la sélection de l'index buffer (étape **Input Assembler**)
- la sélection du vertex buffer (étape **Input Assembler**)
- l'initialisation dans les constantes de l'effet des matrices **matWorld** et **matWorldViewProj**

- 2 La suite de l'initialisation des constantes de l'effet comprend maintenant un paramètre pour la valeur spéculaire de la source d'éclairage (**sp.vSEcl**). Ce paramètre est initialisé pour l'instant à 60 %, mais vous pourrez « jouer » avec cette valeur lorsque vous aurez des objets avec spécularité. Notez que j'ai modifié la position de la source d'éclairage par rapport à l'exemple du chapitre précédent. La position de la caméra est aussi différente, il faudra utiliser la même lors de l'initialisation de la scène (en section 8.8).
- 3 Ces lignes sont identiques à celles de la fonction Draw de CBlocEffet1.
- 4 Le quatrième bloc de code est la vraie nouveauté de notre fonction Draw. Il s'agit d'une boucle permettant de dessiner chacun des sous-objets de notre maillage. Plusieurs opérations ressemblent à ce que nous avions fait pour le bloc, mais comme le contexte de travail est différent, nous les réviserons ici.
 - 4a Nous prenons en note l'indice de départ (dans l'index buffer) du sous-objet présentement traité (**indexStart**). Nous avons aussi besoin du nombre d'indices nécessaires pour le dessin de ce sous-objet (**indexDrawAmount**). Ces valeurs seront essentielles lors de l'appel à **DrawIndexed** (voir 4 g).
 - 4b Un test pour **indexDrawAmount**. Avec certains formats de modèles 3D (comme .OBJ), il est possible que celui-ci soit à 0. Nous aurions pu éliminer ces sous-objets lors du transfert vers notre format, mais plusieurs trouvent des utilisations à ces objets « vides » par exemple l'initialisation de certains paramètres des effets, l'identification de groupes des sous-objets, etc.
 - 4c L'initialisation des constantes de l'effet pour le sous-objet courant.
 - 4d Si nous avons une texture pour le sous-objet courant, nous la spécifions en paramètre. Nous utilisons le paramètre **sp.bTex** comme booléen pour indiquer à l'effet si nous avons une texture ou non.
 - 4e L'instruction **Apply** de ID3DX11EffectPass permet de modifier les constantes pendant une « passe », en réalité pendant une séquence de rendu. Si vous « oubliez » cette instruction, les valeurs précédentes des constantes ainsi que le nom de la texture seront conservées.
 - 4f Tout comme avec CBlocEffet1, nous passons les constantes à la carte graphique.
 - 4g **DrawIndexed** qui cette fois utilise un point de départ différent pour chaque sous-objet ainsi que le nombre d'indices impliqués.

3. Il nous faut maintenant modifier la structure de données qui contient les paramètres de l'effet:

```

struct ShadersParams // toujours un multiple de 16 pour les constantes
{
    XMATRIX matWorldViewProj; // la matrice totale
    XMATRIX matWorld; // matrice de transformation dans le monde
    XMVECTOR vLumiere; // la position de la source d'éclairage (Point)
    XMVECTOR vCamera; // la position de la caméra
    XMVECTOR vAEcl; // la valeur ambiante de l'éclairage
    XMVECTOR vAMat; // la valeur ambiante du matériau
    XMVECTOR vDEcl; // la valeur diffuse de l'éclairage
    XMVECTOR vDMat; // la valeur diffuse du matériau
    XMVECTOR vSEcl; // la valeur spéculaire de l'éclairage
    XMVECTOR vSMat; // la valeur spéculaire du matériau
    float puissance; // la puissance de spécularité
    int bTex; // Texture ou materiau
    XMFLOAT2 remplissage;
};
```

Notez les modifications. Les paramètres pour les valeurs spéculaires de l'éclairage et du matériau, la puissance de spécularité et notre indicateur de texture. Le dernier paramètre, remplissage, est une « nouveauté » dans notre cas, les structures utilisées comme paramètres doivent avoir des dimensions multiples de 16.

La fonction Anime

Peu de modifications à cette fonction, j'ai seulement utilisé une rotation autour de l'axe des Z. Vous verrez pourquoi lors de l'initialisation de la scène en 8.8.

```

void CObjetMesh::Anime(float tempsEcoule)
{
    rotation = rotation + ( (XM_PI * 2.0f) / 10.0f * tempsEcoule );

    // modifier la matrice de l'objet bloc
    matWorld = XMMatrixRotationZ( rotation );
}
```

Faire le ménage

On devra s'organiser pour que l'objet CObjetMesh ne laisse pas d'espace inutilisé dans les tampons de DirectX et dans les tampons de la carte graphique. Pour ce faire, chaque objet créé dynamiquement ou créé par DirectX devra être détruit ou relâché. Nous vidons aussi les vecteurs utilisés pour les index de sous-objets et pour les matériaux.

Nous implantons le ménage dans le destructeur de CObjetMesh:

```
CObjetMesh::~CObjetMesh()
{
    DXRelacher(pConstantBuffer);
    DXRelacher(pSampleState);

    DXRelacher(pEffet);
    DXRelacher(pVertexLayout);
    DXRelacher(pIndexBuffer);
    DXRelacher(pVertexBuffer);
}
```

Notre classe est maintenant complète, mais il nous reste à modifier l'effet et à construire la classe de chargement du fichier modèle.

8.4 Modifications à notre effet

Notre effet, MiniPhong, devra être modifié pour tenir compte des nouveaux éléments que nous voulons implanter soient les valeurs de spécularité et la présence ou non d'une texture. Voici donc la nouvelle version, les modifications sont indiquées:

```

cbuffer param
{
    float4x4 matWorldViewProj;    // la matrice totale
    float4x4 matWorld;          // matrice de transformation dans le monde
    float4 vLumiere;            // la position de la source d'éclairage (Point)
    float4 vCamera;             // la position de la caméra
    float4 vAEcl;               // la valeur ambiante de l'éclairage
    float4 vAMat;               // la valeur ambiante du matériau
    float4 vDEcl;               // la valeur diffuse de l'éclairage
    float4 vDMat;               // la valeur diffuse du matériau
    float4 vSEcl;               // la valeur spéculaire de l'éclairage
    float4 vSMat;               // la valeur spéculaire du matériau
    float puissance;           // la puissance de spécularité
    int bTex;                  // Booléen pour la présence de texture
    float2 remplissage;
}

struct VS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : TEXCOORD0;
    float3 vDirLum : TEXCOORD1;
    float3 vDirCam : TEXCOORD2;
    float2 coordTex : TEXCOORD3;
};

VS_Sortie MiniPhongVS(float4 Pos : POSITION, float3 Normale : NORMAL, float2 coordTex: TEXCOORD)
{
    VS_Sortie sortie = (VS_Sortie)0;

    sortie.Pos = mul(Pos, matWorldViewProj);
    sortie.Norm = mul(float4(Normale, 0.0f), matWorld).xyz;

    float3 PosWorld = mul(Pos, matWorld).xyz;

    sortie.vDirLum = vLumiere.xyz - PosWorld;
    sortie.vDirCam = vCamera.xyz - PosWorld;

    // Coordonnées d'application de texture
    sortie.coordTex = coordTex;

    return sortie;
}

Texture2D textureEntree; // la texture
SamplerState SampleState; // l'état de sampling

```

Ajoutez ces lignes

```

float4 MiniPhongPS( VS_Sortie vs ) : SV_Target
{
    Float3 couleur;

    // Normaliser les paramètres
    float3 N = normalize(vs.Norm);
    float3 L = normalize(vs.vDirLum);
    float3 V = normalize(vs.vDirCam);

    // Valeur de la composante diffuse
    float3 diff = saturate(dot(N, L));

    // R = 2 * (N.L) * N - L
    float3 R = normalize(2 * diff * N - L);

    // Calcul de la spécularité
    float3 S = pow(saturate(dot(R, V)), puissance);

    float3 couleurTexture;

    if (bTex > 0)
    {
        // Échantillonner la couleur du pixel à partir de la texture
        couleurTexture = textureEntree.Sample(SampleState, vs.coordTex).rgb;

        // I = A + D * N.L + (R.V)n
        couleur = couleurTexture * vAEcl.rgb +
                  couleurTexture * vDEcl.rgb * diff +
                  vSEcl.rgb * vSMat.rgb * S;
    }
    else
    {
        couleur = vAEcl.rgb * vAMat.rgb + vDEcl.rgb * vDMat.rgb * diff +
                  vSEcl.rgb * vSMat.rgb * S;
    }

    float4(couleur, 1.0f);
}

technique11 MiniPhong
{
    pass pass0
    {
        SetVertexShader(CompileShader(vs_5_0, MiniPhongVS()));
        SetPixelShader(CompileShader(ps_5_0, MiniPhongPS()));
        SetGeometryShader(NULL);
    }
}

```

Le calcul utilise maintenant la puissance

Atelier

8.5 Des classes pour le chargement de maillages

Notre classe **CObjetMesh** se veut indépendante du format de fichier utilisé pour le stockage du maillage. Le lien entre les deux sera effectué par une classe de chargement dont l'interface est spécifiée par la classe **IChargeur**.

Notre but ici est de ne pas modifier la classe CObjetMesh si nous utilisons un autre format de maillage. La classe IChargeur a donc été influencée par la classe CObjetMesh pour le choix des fonctions d'accès.

La classe-interface IChargeur

Cette classe est définie dans le fichier **chargeur.h**. Il s'agit d'une classe abstraite dont **toutes** les fonctions devront être implantées dans la classe descendante.

Nous en profitons aussi pour déclarer la classe **CParametresChangement** qui n'est qu'une structure de données permettant de regrouper et d'identifier nos paramètres plus facilement.

Le fichier chargeur.h

```
#pragma once

using namespace DirectX;

namespace PM3D
{

class CParametresChangement
{
public:
    CParametresChangement()
    {
        bInverserCulling = false;
        bMainGauche = false;
    }

    std::string NomFichier;
    std::string NomChemin;
    bool bInverserCulling;
    bool bMainGauche;
};

class IChargeur
{
public:
    virtual ~IChargeur() = default;

    virtual void Chargement(const CParametresChangement& param) = 0;

    virtual size_t GetNombreSommets() const = 0;
    virtual size_t GetNombreIndex() const = 0;
    virtual const void* GetIndexData() const = 0;
    virtual int GetNombreSubset() const = 0;
    virtual size_t GetNombreMaterial() const = 0;
    virtual void GetMaterial(int _i,
                           std::string& _NomFichierTexture,
                           std::string& _NomMateriau,
                           XMFLOAT4& _Ambient,
                           XMFLOAT4& _Diffuse,
                           XMFLOAT4& _Specular,
                           float& _Puissance) const = 0;

    virtual const std::string& GetMaterialName(int i) const = 0;

    virtual void CopieSubsetIndex(std::vector<int>& dest) const = 0;

    virtual const XMFLOAT3& GetPosition(int NoSommet) const = 0;
    virtual const XMFLOAT2& GetCoordTex(int NoSommet) const = 0;
    virtual const XMFLOAT3& GetNormale(int NoSommet) const = 0;

};

} // namespace PM3D
```

8.6 Travailler avec un objet 3D



Dans notre programme, nous allons simplement remplacer le bloc par un objet 3D en provenance d'un fichier **OBJ** en l'occurrence un modèle de voiture Lexus. Ce fichier est relativement petit et s'affiche donc rapidement.

1. Dans votre dossier projet, créez un dossier appelé « **modeles** ».
2. Copiez-y le dossier **LEXUS** (présent dans les documents d'accompagnement du chapitre 8 dans le dossier « **modeles** »)
3. N'oubliez pas d'inclure les fichiers `ObjetMesh.h` et `ChargeurOBJ.h` à la suite des autres *include* dans le fichier `Moteur.h`:

```
#include "ObjetMesh.h"
#include "ChargeurOBJ.h"
```

3. Modifiez la fonction **CMoteur::InitObjects** (dans le fichier `Moteur.h`) pour qu'elle crée maintenant le maillage au lieu du bloc:

```
bool InitObjects()
{
    CObjetMesh* pMesh;

    // Constructeur avec format binaire
    std::unique_ptr<CObjetMesh> pMesh =
        std::make_unique<CObjetMesh>(".\\modeles\\jin\\jin.OMB", pDispositif);
    // Puis, il est ajouté à la scène
    ListeScene.push_back(std::move(pMesh));
    return true;
}
```

Note

La classe `CObjetMesh` charge des textures de format DDS. Il est relativement facile de modifier les fichiers `.OBJ`, `.X` et `.BLEND` pour utiliser ce format. Si vous utilisez d'autres formats (ex: `3DS`) vous devrez planter un outil de lecture de texture spécifique.

Une petite note sur quelques paramètres de chargement. Les fichiers maillages sont définis « main droite » ou « main gauche » selon les logiciels utilisés et/ou les choix de l'utilisateur. Comme nous avons jusqu'à maintenant travaillé en « main gauche », il faudra ajuster certaines valeurs si le modèle est main droite, d'où l'indicateur **bMainGauche**. De plus, nous avions implanté un « *backface culling* » de type BACK (antihoraire) alors que certains modèles sont de type horaire (FRONT). Notez aussi que certains modèles n'ont aucun culling, les sommets sont tournés dans un sens ou dans l'autre. Vous devriez éviter ce genre de modèles puisqu'il faut alors désactiver le culling pour celui-ci.

4. Modifiez la fonction **CMoteur::InitScene** pour modifier la position de la caméra et transformer notre système visuel en système « main droite ».

J'en ai aussi profité pour modifier un peu les plans rapprochés et éloigné pour plus tard...

```

virtual int InitScene()
{
    // Initialisation des objets 3D - création et/ou chargement
    if (!InitObjets()) return 1;

    // Initialisation des matrices View et Proj
    // Dans notre cas, ces matrices sont fixes
    matView = XMMatrixLookAtRH( XMVectorSet( 0.0f, -150.0f, 50.0f, 1.0f ),
                                XMVectorSet( 0.0f, 0.0f, 0.0f, 1.0f ),
                                XMVectorSet( 0.0f, 1.0f, 0.0f, 1.0f ) );

    float champDeVision = XM_PI/4;      // 45 degrés
    float ratioDAspect = pDispositif->GetLargeur()/pDispositif->GetHauteur();

    float planRapproche = 0.05;
    float planEloigne = 400.0;

    matProj = XMMatrixPerspectiveFovRH(
                champDeVision,
                ratioDAspect,
                planRapproche,
                planEloigne );

    // Calcul de VP à l'avance
    matViewProj = matView * matProj;

    return 0;
}

```

5. Vous pouvez aussi effacer la scène en noir, pour un affichage plus « dramatique », dans la fonction **CMoteurWindows :: BeginRenderSceneSpecific**:

```

// On efface la surface de rendu
float Couleur[4] = { 0.0f, 0.0f, 0.0f, 1.0f }; // RGBA - NOIR

```

Compilation et exécution

Vous pouvez compiler et exécuter le programme pour vérifier que l'objet3D est bien affiché et bien animé.

8.7 Solution à la lecture très lente

Vous avez sûrement remarqué que le chargement du fichier est long. Le format OBJ est textuel et beaucoup d'options sont disponibles donc le traitement demeure un peu long. D'autres formats seraient plus rapides.

Nous pourrions optimiser un peu le transfert, mais la vitesse résultante serait encore un peu lente, surtout si nous avions plusieurs objets à charger.

Des bibliothèques de chargement de modèles existent, entre autres **Assimp** (<http://assimp.sourceforge.net>), qui supporte beaucoup de formats de modèles. Le chargement du format OBJ avec Assimp est un peu plus rapide que le mien () mais est quand même beaucoup plus lente que ce qu'accomplissent les jeux vidéos.

La solution est d'utiliser un format binaire **correspondant directement** à notre classe de maillage donc pas d'analyse, pas de conversion, pas de format texte. Le résultat sera une vitesse de chargement très intéressante.

9. Affichage 2D

Ce qu'on appelle affichage 2D est en réalité une version moderne de l'affichage « raster » c'est-à-dire de l'affichage d'images. Ce chapitre couvre l'utilisation de quelques un des aspects les plus fréquents de la « 2D » soient les sprites et les panneaux (« billboard »). Nous aborderons aussi différentes techniques permettant d'optimiser l'affichage des sprites et des panneaux.

Objectifs du chapitre

- ✓ Comprendre ce qu'est un sprite.
- ✓ Explorer différentes méthodes pour leur implantation.
- ✓ Régler les problèmes liés au choix des unités.
- ✓ Régler les problèmes liés à la transparence.
- ✓ Comprendre quand désactiver le Z-buffer.
- ✓ Comprendre ce qu'est un panneau.
- ✓ Comprendre le processus d'orientation linéaire-Z
- ✓ Se familiariser avec les autres orientations de panneau.

9. Affichage 2D

Nous employons ici le terme affichage 2D. Les éléments qui nous intéressent ici couvrent les cas suivants :

- l'affichage d'images sur la surface principale (sur l'écran), les **sprites**;
- l'affichage d'images sur des surfaces 3D (des panneaux-*billboard*);
- et l'écriture de texte sur une surface.

Le premier cas est principalement utilisé en 3D pour afficher des éléments de contrôle, de l'interface, ou des images qui vont par-dessus l'affichage 3D, mais il peut aussi servir à créer certains effets spéciaux. Cette méthode est aussi celle à utiliser pour l'écriture d'applications 2D.

Le deuxième cas correspond à une utilisation plus avancée des textures. Dans ce cas, au lieu d'utiliser l'écran comme destination, nous utilisons une texture puis nous appliquons celle-ci à un rectangle (ou un autre objet).

L'affichage du texte pourra quant à lui être effectué sur l'écran ou sur une texture.

L'affichage 2D peut aussi être utilisé pour réaliser une application 2D complète, sans 3D, comme on en retrouve dans certains jeux.

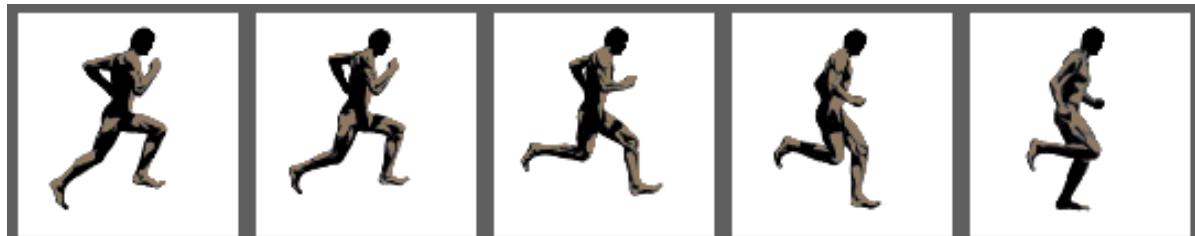
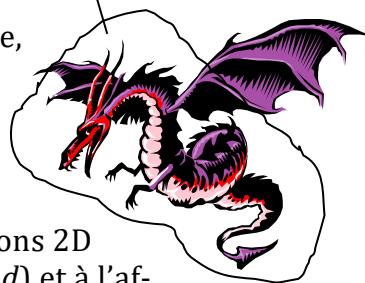
Théorie

9.1 Les sprites

UN SPRITE?
QU'EST-CE QUE C'EST?

Un **sprite** est une image de forme irrégulière (en infographie, tout ce qui n'est pas rectangulaire...) qui peut être placée n'importe où sur une autre image. En pratique, un sprite est un bitmap (rectangulaire) utilisant une couleur de transparence. La couleur de transparence sera appelée **clé de couleur**.

Le sprite est l'élément principal de l'affichage 2D. Les applications 2D les plus simples se résument à l'affichage d'un fond (*background*) et à l'affichage d'un ou plusieurs sprites sur ce fond. L'utilisation d'une clé de couleur permet à cette couleur prédéterminée d'être considérée comme transparente, ce qui donne au sprite la possibilité d'être de forme irrégulière.



L'utilisation d'une série de sprites pour représenter un objet permet une simulation d'animation si on les affiche en séquence à une vitesse raisonnable prédéterminée. C'est le même principe que le dessin animé. Notre modèle d'animation est prêt à recevoir ce type d'animation.

Théorie

IMPORTANT

Attention à la quantité de mémoire vidéo disponible. Un affichage de 800 X 600 et une texture de 1024 X 1024 en mode 32 bits par pixel demanderont presque 6 MB de mémoire vidéo (surface principale, backbuffer, et texture).

Même si DirectX peut s'occuper de la gestion des textures en les ôtant de la carte s'il a besoin d'espace, il peut en résulter une perte de performance.

9.2 Sprites et textures

Avec Direct3D, tous les objets sont en 3D. Ce qui veut dire que l'on peut afficher une image selon n'importe quel angle sur un panneau (*billboard*). Il s'agit en fait de textures appliquées aux polygones d'un objet 3D (le panneau). L'affichage 2D se résume alors à disposer un panneau à l'écran de façon à y placer le sprite qui nous intéresse. Ça vous semble peu évident ? En effet, il n'est pas toujours facile de déterminer la position et les dimensions à donner au panneau, surtout dans une application 3D où la position de la caméra et les informations de projection peuvent changer en tout temps.

Comme nous l'avons mentionné plus tôt, nos sprites seront quant à eux chargés au préalable dans des objets textures comme nous l'avons fait jusqu'à maintenant. Nous devrons ensuite les afficher à l'endroit que nous choisirons sur la surface de rendu.

Plusieurs façons de faire sont possibles pour l'affichage de sprites dont l'usage de Direct2D, un nouvel API disponible depuis DirectX 10. Mais Direct2D est un peu lourd à initialiser (moins que Direct3D, mais tout de

même) et n'offre pas réellement d'avantage sur les méthodes que nous allons présenter.

Méthode 1- Principe de base

Pour un simple sprite, sans animation et affiché avec des coordonnées exprimées en pixels, la méthode la plus simple se résume à :

1. Définir un objet 3D correspondant à un rectangle;
2. Lui donner une dimension originale de 1 X 1. Pourquoi ? Parce que cela simplifiera le dimensionnement;
3. Lui associer une texture;
4. Écrire des shaders (VS et PS) ou un effet dont le seul but est d'afficher la texture;
5. Définir une matrice équivalente à WVP, cette matrice nous permettra de placer le sprite, de le dimensionner, et même d'y appliquer d'autres transformations, par exemple des rotations.
6. L'affichage devra aussi tenir compte de certains éléments comme la transparence.

Cette méthode, relativement simple, nous servira de point de départ pour explorer les éléments dont nous aurons besoin par la suite.

Elle présente toutefois une lacune assez importante du côté du stockage et de la performance si nous avons plusieurs sprites. La méthode 2 nous permettra d'améliorer de beaucoup la performance.

Méthode 2 - L'afficheur de sprites

Les problèmes de la méthode 1 sont :

1. Un vertex buffer et son format de sommets (*vertex layout*) pour chaque sprite alors que tous nos sprites utilisent un rectangle de 1 X 1 et le même format de sommets.
2. Un tampon de constantes, un objet Effet, un objet Technique, pour chaque sprite alors que tous nos sprites utilisent le même effet.
3. Un état de sampling (*Sampler*) pour chaque sprite alors qu'ils utilisent tous les mêmes paramètres.
4. Ces objets sont créés pour chaque sprite, donc perte de performance au moment de l'initialisation.
5. Lors du rendu, tous ces éléments doivent être passés au pipeline. Ce qui est « refait » pour chaque sprite alors que seulement la texture et la matrice de positionnement-dimensionnement sont différentes.

Les quatre premiers problèmes pourraient être réglés par des **membres statiques** dans notre classe de sprite, ceux-ci n'existant qu'une seule fois et n'étant initialisés qu'une seule fois. Mais le problème #5 est de loin le plus important puisqu'il pourrait ralentir beaucoup notre application si nous avions beaucoup de sprites, par exemple pour un système de particules.

La solution est d'avoir un objet « Afficheur de sprites » qui s'occupe une seule fois de l'initialisation et qui conserve les sprites de façon simple en ne retenant que la texture et la matrice de positionnement-dimensionnement. Le rendu de tous les sprites sera alors effectué en une seule fonction qui s'occupera des paramètres et du rendu de chaque rectangle. C'est ce que nous ferons dans un deuxième temps.

9.3 Méthode 1- Implantation

Même si nous désirons utiliser un afficheur de sprite, il est plus facile de, temporairement, définir un objet sprite qui nous permettra d'explorer les éléments que nous aurons besoin et de mettre au point certaines fonctionnalités de notre affichage de sprite.

La classe CSpriteTemp

Nos sprites se retrouveront (temporairement) implantés au moyen de cette classe.

1. Définir la classe CSpriteTemp
2. Modifiez le fichier **spriteTemp.h** pour contienne les #include dont nous aurons besoin, pour que la classe soit dans l'espace de nom PM3D et pour que le constructeur soit paramétré.

```
#pragma once

#include <string>
#include "d3dx11effect.h"
#include "Objet3D.h"
#include "DispositifD3D11.h"
#include "Texture.h"

namespace PM3D
{

class CSpriteTemp : public CObjet3D
{
public:
    CSpriteTemp(const std::string& NomTexture, CDispositifD3D11*
pDispositif);

    virtual ~CSpriteTemp();
};

} // namespace PM3D
```

3. Modifiez aussi le fichier **spriteTemp.cpp** pour que la classe soit dans l'espace de nom PM3D et pour que le constructeur soit comme dans la déclaration :

```
#include "StdAfx.h"
#include "SpriteTemp.h"
#include "resource.h"
#include "MoteurWindows.h"
#include "util.h"

namespace PM3D
{
CSpriteTemp::CSpriteTemp(const std::string& NomTexture, CDispositifD3D11*
pDispositif)
{
}
```

```
CSpriteTemp::~CSpriteTemp()
{
}

}
```

4. Nous déclarerons la classe **CSommetSprite** (nos sommets) un peu avant la classe CSpriteTemp dans le fichier **spritetemp.h**:

```
class CSommetSprite
{
public:
    CSommetSprite() = default;
    CSommetSprite(const XMFLOAT3& position, const XMFLOAT2& coordTex)
        : m_Position(position)
        , m_CoordTex(coordTex)
    {
    }

public:
    static UINT numElements;
    static D3D11_INPUT_ELEMENT_DESC layout[];

    XMFLOAT3 m_Position;
    XMFLOAT2 m_CoordTex;
};
```

Notez que nous n'avons pas de normale.

5. Nous pouvons aussi dès maintenant définir et initialiser les variables statiques **layout** et **numElements** dans le fichier **spritetemp.cpp** :

```
// Definir l'organisation de notre sommet
D3D11_INPUT_ELEMENT_DESC CSommetSprite::layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0},
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT,      0, 12,
D3D11_INPUT_PER_VERTEX_DATA, 0}
};

UINT CSommetSprite::numElements = ARRAYSIZE(layout);
```

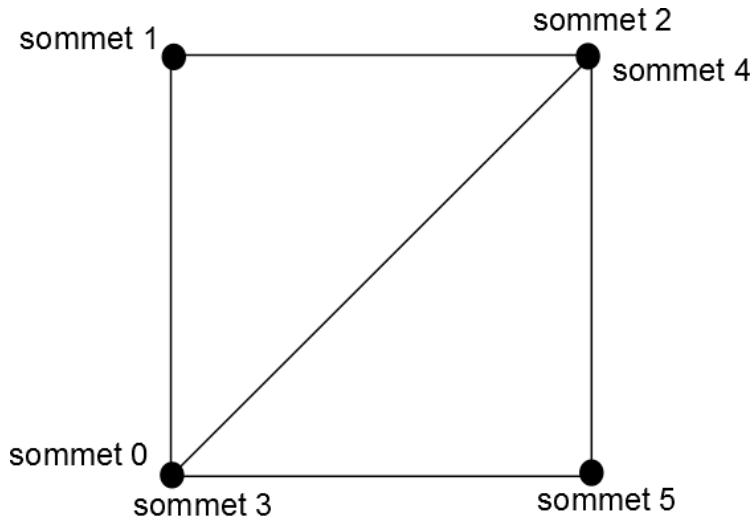
6. Nous voulons définir un objet utilisant un rectangle. Un rectangle étant un objet simple, nous n'utiliserons pas d'index buffer et nous utiliserons la fonction **ID3D11DeviceContext::Draw** au lieu de **ID3D11DeviceContext::DrawIndexed**. Nous définissons donc un tableau statique de 6 objets CSommetSprite comme suit dans la section *private* de la classe CSpriteTemp:

```
private:
    static CSommetSprite sommets[6];
```

Que nous définissons et initialisons ainsi dans le fichier **sprite-temp.cpp**:

```
CSommetSprite CSpriteTemp::sommets[6] = {
    CSommetSprite(XMFLOAT3(0.0f, 0.0f, 0.0f), XMFLOAT2(0.0f, 1.0f)),
    CSommetSprite(XMFLOAT3(0.0f, 1.0f, 0.0f), XMFLOAT2(0.0f, 0.0f)),
    CSommetSprite(XMFLOAT3(1.0f, 1.0f, 0.0f), XMFLOAT2(1.0f, 0.0f)),
    CSommetSprite(XMFLOAT3(0.0f, 0.0f, 0.0f), XMFLOAT2(0.0f, 1.0f)),
    CSommetSprite(XMFLOAT3(1.0f, 1.0f, 0.0f), XMFLOAT2(1.0f, 0.0f)),
    CSommetSprite(XMFLOAT3(1.0f, 0.0f, 0.0f), XMFLOAT2(1.0f, 1.0f))
};
```

Nos sommets:



7. Nous déclarons ensuite un pointeur sur le vertex buffer et un pointeur sur le dispositif dans la section *private* de la classe CSpriteTemp:

```
ID3D11Buffer* pVertexBuffer;
CDispositifD3D11* pDispositif;
```

8. Nous pouvons maintenant ajouter ces lignes au constructeur de CSpriteTemp:

```
CSpriteTemp::CSpriteTemp(string NomTexture, CDispositifD3D11* _pDispositif)
: pDispositif(_pDispositif)
, pVertexBuffer(nullptr)
, pConstantBuffer(nullptr)
, pEffet(nullptr)
, pTechnique(nullptr)
, pPasse(nullptr)
, pVertexLayout(nullptr)
, pSampleState(nullptr)
{
    // Création du vertex buffer et copie des sommets
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));
```

```
bd.Usage = D3D11_USAGE_IMMUTABLE;
bd.ByteWidth = sizeof(sommets);
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;

D3D11_SUBRESOURCE_DATA InitData;
ZeroMemory(&InitData, sizeof(InitData));
InitData.pSysMem = sommets;

DXESSAYER(pD3DDevice->CreateBuffer(&bd, &InitData, &pVertexBuffer),
DXE_CREATIONVERTEXBUFFER);

// Initialisation de l'effet
InitEffet();

... Suite
}
```

Vous noterez que ce code est très semblable à celui utilisé pour la création du vertex buffer du bloc sauf qu'il n'y a que 6 sommets et que ceux-ci sont déjà initialisés. J'ai ajouté l'appel de InitEffet qui est une de nos prochaines étapes.

9. À la suite du constructeur, nous pouvons charger notre texture:

```
// Initialisation de la texture
CGestionnaireDeTextures& TexturesManager =
    CMoteurWindows::GetInstance().GetTextureManager();

std::wstring ws(NomTexture.begin(), NomTexture.end());

std::unique_ptr<CSprite> pSprite = std::make_unique<CSprite>();
pSprite->pTextureD3D =
    TexturesManager.GetNewTexture(ws.c_str(), pDispositif)-
>GetD3DTexture();
```

10. Déclarez la fonction InitEffet dans la classe CSpriteTemp:

```
private:
    virtual void InitEffet();
```

Pourquoi *virtual*? Parce que nous voudrons peut-être faire une classe dérivée de CSpriteTemp...

11. Définissez la ainsi:

```
void CSpriteTemp::InitEffet()
{
    // Compilation et chargement du vertex shader
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    // Création d'un tampon pour les constantes de l'effet
    D3D11_BUFFER_DESC bd;
    ZeroMemory( &bd, sizeof(bd) );

    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(ShadersParams);
    bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
    bd.CPUAccessFlags = 0;
    pD3DDevice->CreateBuffer( &bd, nullptr, &pConstantBuffer );

    // Pour l'effet
    ID3DBlob* pFXBlob = nullptr;
```



```
DXEssayer( D3DCompileFromFile( L"Sprite1.fx", 0, 0, 0,
                                "fx_5_0", 0, 0,
                                &pFXBlob, 0),
            DXE_ERREURCREATION_FX);

D3DX11CreateEffectFromMemory( pFXBlob->GetBufferPointer(),
                             pFXBlob->GetBufferSize(), 0, pD3DDevice, &pEffet);

pFXBlob->Release();

pTechnique = pEffet->GetTechniqueByIndex(0);
pPasse = pTechnique->GetPassByIndex(0);

// Crée l'organisation des sommets pour le VS de notre effet
D3DX11_PASS_SHADER_DESC effectVSDesc;
pPasse->GetVertexShaderDesc(&effectVSDesc);

D3DX11_EFFECT_SHADER_DESC effectVSDesc2;
```

```

effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
                                              &effectVSDesc2);

const void *vsCodePtr = effectVSDesc2.pBytecode;
const unsigned vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout = nullptr;
DXESSAYER( pD3DDevice->CreateInputLayout(
    CSommetSprite::layout,
    CSommetSprite::numElements,
    vsCodePtr,
    vsCodeLen,
    &pVertexLayout ),
    DXE_CREATIONLAYOUT);

// Initialisation des paramètres de sampling de la texture
D3D11_SAMPLER_DESC samplerDesc;

samplerDesc.Filter = D3D11_FILTER_ANISOTROPIC;
samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 4;
samplerDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
samplerDesc.BorderColor[0] = 0;
samplerDesc.BorderColor[1] = 0;
samplerDesc.BorderColor[2] = 0;
samplerDesc.BorderColor[3] = 0;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;

// Création de l'état de sampling
pD3DDevice->CreateSamplerState(&samplerDesc, &pSampleState);
}

```

Il s'agit pour le moment du même code que pour le CBlocEffet1, sauf pour le nom de l'effet et pour l'utilisation de CSommetSprite. Nous y ferons d'autres modifications plus loin, mais pour la mise au point de notre classe, elle conviendra.

12. Il nous manquait les déclarations de variables suivantes:

```

ID3D11Buffer* pConstantBuffer;
ID3DX11Effect* pEffect;
ID3DX11EffectTechnique* pTechnique;
ID3DX11EffectPass* pPass;
ID3D11InputLayout* pVertexLayout;
ID3D11ShaderResourceView* pTextureD3D;
ID3D11SamplerState* pSampleState;

```

Le rôle de ces variables est amplement expliqué aux chapitres 6 et 7.

13. Ajoutez aussi la définition (locale) de ShadersParam dans le fichier spritetemp.cpp:

```
struct ShadersParams
{
    XMATRIX matWVP; // la matrice totale
};
```

14. Déclarez les fonctions publiques Anime et Draw de la façon suivante:

```
virtual void Anime(float tempsEcoule){};
virtual void Draw();
```

La fonction Anime n'est là que parce que CSpriteTemp est dérivée de CObjet3D. Mais peut-être voudrons-nous l'animer plus tard.

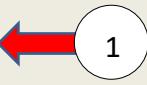
14. Définissez la fonction Draw ainsi:

```
void CSpriteTemp::Draw()
{
    // Obtenir le contexte
    ID3D11DeviceContext* pImmediateContext = pDispositif-
>GetImmediateContext();

    // Choisir la topologie des primitives
    pImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

    // Source des sommets
    UINT stride = sizeof( CSommetSprite );
    const UINT offset = 0;
    pImmediateContext->IASetVertexBuffers( 0, 1, &pVertexBuffer, &stride,
&offset );

    // input layout des sommets
    pImmediateContext->IASetInputLayout( pVertexLayout );

    // Initialiser et sélectionner les « constantes » de l'effet
    ShadersParams sp;
    sp.matWVP = XMMatrixIdentity(); 
    pImmediateContext->UpdateSubresource( pConstantBuffer, 0, nullptr, &sp,
0, 0 );

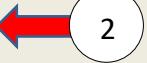
    // Nous n'avons qu'un seul CBuffer
    ID3DX11EffectConstantBuffer* pCB = pEffet-
>GetConstantBufferByName("param");
    pCB->SetConstantBuffer(pConstantBuffer);

    // Activation de la texture
    ID3DX11EffectShaderResourceVariable* variableTexture;
    variableTexture = pEffet->GetVariableByName("textureEntree")-
>AsShaderResource();
    variableTexture->SetResource(pTextureD3D);
```

```

// Le sampler state
ID3DX11EffectSamplerVariable* variableSampler;
variableSampler = pEffet->GetVariableByName("SampleState")->AsSampler();
variableSampler->SetSampler(0, pSampleState);

pPasse->Apply(0, pImmediateContext);

// **** Rendu de l'objet
pImmediateContext->Draw( 6, 0 );
}

```

À noter dans le code précédent

Le code ressemble beaucoup à celui utilisé pour CBlocEffet1, mais notez qu'il **n'y a pas d'index buffer** et que nous utilisons CSommetSprite comme type de sommets.

-  La matrice de position-dimension est **temporairement** la matrice identité.
-  Nous utilisons la fonction ID3D11DeviceContext::Draw pour effectuer le rendu.

15. Ajoutez au destructeur de CSpriteTemp :

```

DXRelacher(pConstantBuffer);
DXRelacher(pSampleState);

DXRelacher(pEffet);
DXRelacher(pVertexLayout);
DXRelacher(pVertexBuffer);

```

9.5 Méthode 1- L'effet

L'effet est une version très simple de vertex shader et de pixel shader. Il n'y a aucune normale, aucun éclairage. Notre but est surtout de positionner notre rectangle avec matWVP et d'utiliser les coordonnées de texture pour le sprite.

```

cbuffer param
{
    float4x4 matWVP;    // la matrice de travail
}

struct VS_Sortie
{
    float4 Pos : SV_Position;
    float2 coordTex : TEXCOORD0;
};

VS_Sortie Sprite1VS(float4 Pos : POSITION, float2 coordTex : TEXCOORD)
{
    VS_Sortie sortie = (VS_Sortie)0;

    sortie.Pos = mul(Pos, matWVP);

    // Coordonnées d'application de texture
    sortie.coordTex = coordTex;

    return sortie;
}

Texture2D textureEntree; // la texture
SamplerState SampleState; // l'état de sampling

float4 Sprite1PS(VS_Sortie vs) : SV_Target
{
    float4 couleurTexture;

    couleurTexture = textureEntree.Sample(SampleState, vs.coordTex);
    return couleurTexture;
}

technique11 AfficheSprite
{
    pass pass0
    {
        SetVertexShader(CompileShader(vs_5_0, Sprite1VS()));
        SetPixelShader(CompileShader(ps_5_0, Sprite1PS()));
        SetGeometryShader(NULL);
    }
}

```

Atelier

9.5 Méthode 1- L'affichage du sprite

L'affichage d'un sprite sur la surface de rendu est l'opération clé pour la réalisation d'applications 2D. En 3D, cette opération est principalement utilisée pour afficher des éléments fixes à l'environnement utilisateur soient des panneaux de contrôle, des boîtes d'information, des icônes illustrant certains états, etc. Pour les besoins de notre exemple, nous afficherons plutôt un arbre par-dessus notre personnage, ça n'a aucune utilité réelle, mais ça illustre bien l'intérêt des sprites.

1. Copiez dans votre répertoire projet le fichier **tree02s.dds** que vous retrouverez dans le sous-répertoire SAMPLES\MEDIA\TREES du dossier DIRECTX.



Le format **dds** (*DirectDraw Surface*) est un format 32 bits pouvant contenir **à l'avance** des informations sur la transparence partielle ou totale des pixels de notre image. Le fichier **tree02s.dds** contient ce type d'information. Il est important de noter que les images 24 bits (.BMP et autres) ne contiennent pas d'information sur la transparence. Les clés de couleurs (couleur de transparence) peuvent être interprétées au chargement des textures à condition que le format de texture le supporte, c.-à-d. que nous ayons créé une texture avec un format contenant la valeur alpha. La valeur alpha est notre seul moyen de réaliser des transparencies (partielles ou totales) une fois la texture chargée. Donc, un format ne possédant pas de A ne pourra pas facilement être utilisé comme sprite (mais...). Par contre, même si notre sprite est de format 32 bits, nous pouvons le transposer dans notre affichage 16 bits.

Le format DDS a été introduit avec DirectX 7.0 alors que DirectDraw existait encore d'où son nom qui a tout de même été conservé après la disparition de DirectDraw.

Il est possible de charger une image 24 bits (pas de transparence) dans un format 32 bits. Il nous faudrait alors spécifier une clé de couleur pour

simuler la transparence. Les pixels correspondants à cette couleur seraient remplacés lors du chargement par des pixels noirs transparents (x00000000).

2. Nous allons maintenant ajouter un sprite à la scène. Ce n'est pas nécessairement la meilleure méthode si nous désirons implanter une interface, mais dans mon cas je ne m'en sers que pour afficher des éléments « décoratifs ». Ces ajouts ont lieu dans la fonction **CMoteur::InitObjets**, avant l'énoncé **return**

```
bool InitObjets()
{
    ...
    ...

    // Création d'un objet sprite
    auto pSprite = std::make_unique<CSpriteTemp>("tree02s.dds",
pDispositif);

    // Puis, il est ajouté à la scène
    ListeScene.push_back(std::move(pSprite));

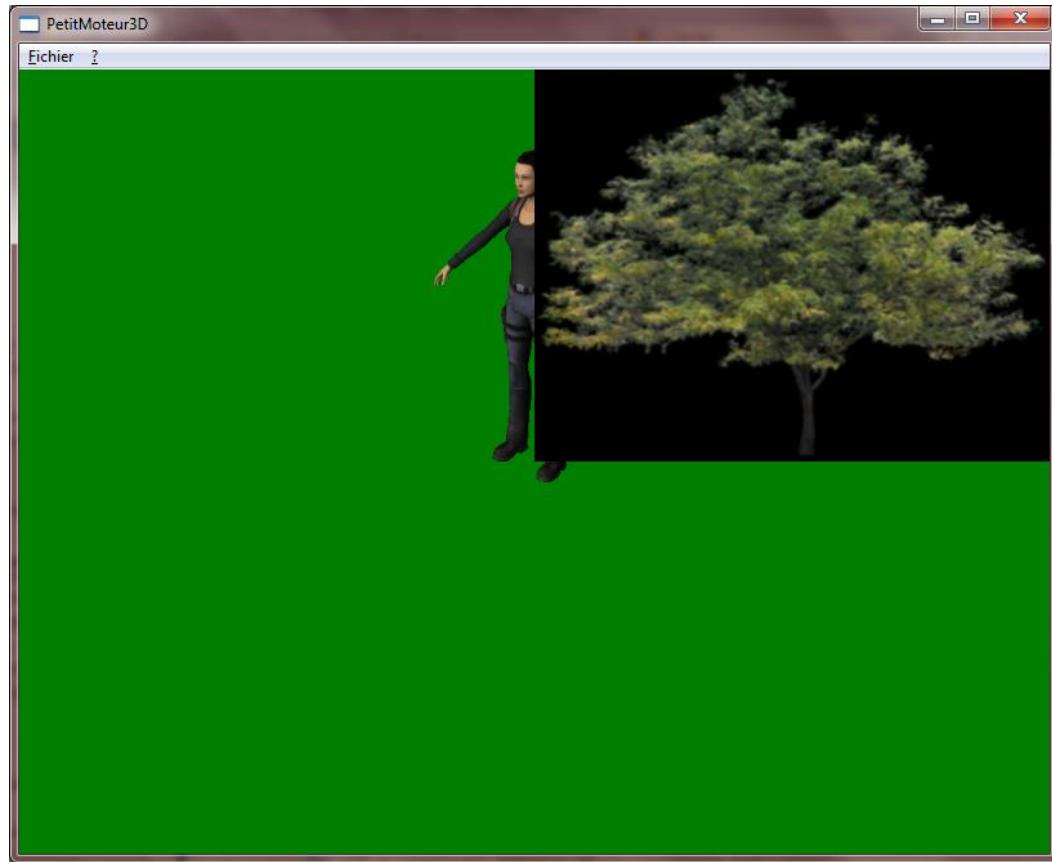
    return true;
}
```

L'ordre des fonctions d'initialisation utilisé dans **InitObjets** n'a pas toujours d'importance. Mais dans notre cas, il est important d'effectuer l'affichage des sprites après l'affichage des objets 3D. L'affichage des sprites est effectué sur la surface de rendu, mais a aussi comme effet de mettre la valeur 0 dans le Z-Buffer et ce, même pour les couleurs transparentes. En réalité, les couleurs avec transparences sont ajoutées aux couleurs déjà existantes pour produire une couleur résultat, le contraire serait difficile à obtenir même en désactivant le ZBuffer.

4. N'oubliez pas le #include à la suite des autres dans le fichier **moteur.h** !

```
#include "SpriteTemp.h"
```

Essayez votre programme, vous devriez obtenir un affichage ressemblant à ceci (j'ai ramené le fond vert pour plus de visibilité) :



Que pouvons-nous en déduire ?

Deux choses :

1. La matrice de position-dimension étant une matrice identité, notre sprite s'affiche sur le rectangle $\{ (0,0), (0,1), (1,1), (1,0) \}$ de l'espace-écran.
Rappel : l'écran utilise des dimensions de -1 à 1 sur les axes X et Y.
2. La transparence n'est pas activée.

Atelier

9.6 Méthode 1- La transparence (mélange alpha)

Dans la plupart des systèmes graphiques modernes, la transparence (à divers degrés) est implémentée au moyen du **mélange alpha** [Möller et Haines 1 section 5.7] c'est-à-dire de l'utilisation de la valeur alpha de l'objet transparent au moyen d'une formule ressemblant à celle-ci :

```
Resultat = Source.a * Source + ( 1 - Source.a) * Destination
```

où **Resultat**, **Source** et **Destination** sont des couleurs

C'est le cas avec DirectX 11 et avec le HLSL. Il nous faut toutefois définir à l'avance des « États de mélange » qui peuvent être assez variés selon le résultat que nous voulons obtenir, le mélange alpha n'étant pas la seule option. Dans notre cas, seulement deux états de mélanges (**blending states**) seront nécessaires.

1. Pour ce faire nous allons ajouter une fonction, **InitBlendStates**, à la classe **CDispositifD3D11**. Nous la déclarons ainsi dans la section *private* de la classe **CDispositifD3D11** :

```
void InitBlendStates();
```

2. Puis nous la définissons ainsi (dans **DispositifD3D11.cpp**) :

```
void CDispositifD3D11::InitBlendStates()
{
    D3D11_BLEND_DESC blendDesc;

    // Effacer la description
    ZeroMemory(&blendDesc, sizeof(D3D11_BLEND_DESC));

    // On initialise la description pour un mélange alpha classique
    blendDesc.RenderTarget[0].BlendEnable = TRUE;
    blendDesc.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_ALPHA;
    blendDesc.RenderTarget[0].DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
    blendDesc.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
    blendDesc.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
    blendDesc.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
    blendDesc.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_ADD;
    blendDesc.RenderTarget[0].RenderTargetWriteMask =
                                                D3D11_COLOR_WRITE_ENABLE_ALL;

    // On crée l'état alphaBlendEnable
    DXEssayer(pDXDDevice->CreateBlendState(&blendDesc, &alphaBlendEnable),
               DXE_ERREURCREATION_BLENDSTATE);
```

1

```

// Seul le booleen BlendEnable nécessite d'être modifié
blendDesc.RenderTarget[0].BlendEnable = FALSE;

// On crée l'état alphaBlendDisable
DXESSAYER( pD3DDevice->CreateBlendState(&blendDesc,&alphaBlendDisable),
            DXE_ERREURCREATION_BLENDSTATE);
}

```

2

Explications

1

BlendEnable active (ou non) le mélange. S'il est à FALSE, il n'y a pas de mélange et les autres valeurs n'ont pas d'importance.

SrcBlend est l'opération que nous voulons effectuer sur le pixel source. Nous utilisons SRC_ALPHA qui correspond à la couleur source multipliée par sa valeur alpha.

DestBlend est pour le pixel destination (celui déjà dans le tampon de rendu). Nous utilisons INV_SRC_ALPHA qui correspond à la couleur destination multipliée par 1 moins la valeur alpha de la source. Par exemple si la valeur alpha de la source est 0.40 alors 0.60 sera utilisé pour multiplier la valeur destination.

BlendOp est l'opération de mélange. Dans notre cas, D3D11_BLEND_OP_ADD indique qu'on veut additionner la valeur source et la valeur destination pour obtenir le pixel résultant.

Les trois paramètres suivants permettent de construire la valeur alpha du résultat.

Le dernier paramètre, **RenderTargetWriteMask**, nous permet de filtrer certaines valeurs RGBA. Dans notre cas, nous ne filtrons rien (D3D11_COLOR_WRITE_ENABLE_ALL).

2

Seulement la valeur BlendEnable a besoin d'être changée pour créer un état sans mélange.

3. Déclarez les variables suivantes dans la section *private* de la classe CDispositifD3D11 :

```

// Pour le mélange alpha (transparence)
ID3D11BlendState* alphaBlendEnable;
ID3D11BlendState* alphaBlendDisable;

```

4. Ajoutez le texte de référence pour dans le **string table** pour le message DXE_ERREURCREATION_BLENDSTATE

5. Nous insérons l'appel à la fonction **InitBlendStates** dans le constructeur de **CDispositifD3D11**, à la suite de l'appel à **InitDepthBuffer** :

```

...
// Initialiser le tampon de profondeur
InitDepthBuffer();

// Initialiser les états de mélange (blending states)
InitBlendStates();

...

```

6. Déclarez maintenant deux fonctions *publiques* dans la classe **CDispositifD3D11** :

```

void ActiverMelangeAlpha();
void DesactiverMelangeAlpha();

```

7. Définir la fonction **ActiverMelangeApha** ainsi :

```

void CDispositifD3D11::ActiverMelangeAlpha()
{
    // Activer le mélange - alpha blending.
    pImmediateContext->OMSetBlendState(alphaBlendEnable, nullptr,
0xffffffff);
}

```

8. Définir la fonction **DesactiverMelangeApha** ainsi :

```

void CDispositifD3D11::DesactiverMelangeAlpha()
{
    // Désactiver le mélange - alpha blending.
    pImmediateContext->OMSetBlendState(alphaBlendDisable, nullptr,
0xffffffff);
}

```

9. Modifiez maintenant la fonction **Draw** de **CSpriteTemp** pour qu'elle active le mélange alpha avant de faire le rendu et le désactive après :

```

...
// **** Rendu de l'objet
pDispositif->ActiverMelangeAlpha();
pImmediateContext->Draw( 6, 0 );
pDispositif->DesactiverMelangeAlpha();
}

```

Vous pouvez essayer votre application, l'arbre devrait maintenant être pourvu de transparence.

Note : Nous aurions pu implanter le mélange alpha directement dans l'effet. Il est possible de modifier certains états dans les techniques et dans les passes. J'ai préféré le faire dans le programme, car c'est un peu plus polyvalent comme approche.

Atelier

9.7 Méthode 1 - Position et dimensions d'affichage

Comme nous l'avons vu en 9.4, l'utilisation d'une matrice identité comme matrice de positionnement et de dimensionnement nous donnait un sprite occupant le premier quadrant de l'écran.

Nous désirons maintenant :

- positionner notre sprite à une position exprimée en pixels par rapport au coin supérieur gauche de l'écran;
- dimensionner au besoin notre sprite, sinon utiliser par défaut la grandeur de la texture.

Dimensions réelles de la texture

Pour obtenir les dimensions réelles de la texture, il existe plusieurs moyens. Le plus rapide étant évidemment de la connaître à l'avance... Dans notre cas, la méthode utilisée pour le chargement des textures (la fonction D3DX11CreateShaderResourceViewFromFile) ne nous permet pas directement d'obtenir ces dimensions. Nous allons donc interroger le « resource view » au moyen de quelques instructions pour obtenir les informations sur la texture. Cette procédure est intéressante parce qu'elle nous permettrait d'interroger le système sur des éléments chargés, par exemple, par d'autres modules et auquel nous n'aurions pas accès directement. Dans notre cas, elle nous évite tout simplement d'avoir à réviser notre classe CTexture.

1. Dans le constructeur de CSpriteTemp, à la suite du chargement de la texture, ajoutez les lignes de l'encadré suivant :

```

if (NomTexture != "")
{
    std ::wstring ws(NomTexture.begin(), NomTexture.end());

    std::unique_ptr<CSprite> pSprite = std ::make_unique<CSprite>();
    pSprite-> pTextureD3D =
        TexturesManager.GetNewTexture(ws.c_str(), pDispositif)->GetD3DTexture();

    // Obtenir la dimension de la texture
    ID3D11Resource* pResource;
    ID3D11Texture2D *pTextureInterface = 0;
    pSprite-> pTextureD3D->GetResource(&pResource);
    pResource->QueryInterface<ID3D11Texture2D>(&pTextureInterface);
    D3D11_TEXTURE2D_DESC desc;
    pTextureInterface->GetDesc(&desc);

    DXRelacher(pResource);
    DXRelacher(pTextureInterface);
}

```

```

    dimX = float(desc.Width);
    dimY = float(desc.Height) ;
}

```

2. Déclarez les variables **dimX** et **dimY** dans la section *private* de la classe CSpriteTemp :

```

float dimX ;      //Dimensions de la texture
float dimY ;

```

Position et dimensions d'affichage

3. Déclarez la fonction **SetPosDim** dans la section *public* de la classe CSpriteTemp :

```
void SetPosDim( int _x, int _y, int _ dx=0, int _ dy=0 );
```

_dx et _dy ont des valeurs de défaut de 0 pour nous permettre d'utiliser les dimensions réelles de la texture au besoin.

4. Définissez-la ainsi :

```

void CSpriteTemp::SetPosDim( int _x, int _y, int _dx, int _dy )
{
float x, y, dx, dy;
float posX, posY;
float facteurX, facteurY;

// Dimensions en pixel
if (_dx==0 && _dy==0)
{
    // Dimensions par défaut
    dx = dimX;
    dy = dimY;
}
else
{
    dx = float(_dx);
    dy = float(_dy);
}

// Dimensions en facteur
facteurX = dx*2.0f/pDispositif->GetLargeur();
facteurY = dy* 2.0f/pDispositif->GetHauteur();

// Position en coordonnées logiques
// 0,0 pixel = -1,1
x = float(_x);
y = float(_y);

posX = x* 2.0f/pDispositif->GetLargeur() - 1.0f;
posY = 1.0f - y*2.0f/pDispositif->GetHauteur();

```

La largeur en pixels, par exemple 1024 correspond à deux unités lors de l'affichage.

La position en X en pixels, se retrouve distribuée de -1 à 1 lors de l'affichage.
La position en Y en pixels, se retrouve distribuée de 1 à -1 lors de l'affichage.

```
    matPosDim = XMMatrixScaling(facteurX, facteurY, 1.0f) *
                XMMatrixTranslation(posX, posY, 0.0f) ;
}
```

5. Déclarez la variable matPosDim qui nous servira maintenant de matrice de position-dimension (paramètre de l'effet).

```
XMMATRIX matPosDim;
```

6. Initialisez-la à la matrice Identité dans le constructeur de CSpriteTemp :

```
matPosDim = XMMatrixIdentity() ;
```

7. Dans la fonction **Draw** de CSpriteItem, modifiez linitialisation de la constante sp.matWVP :

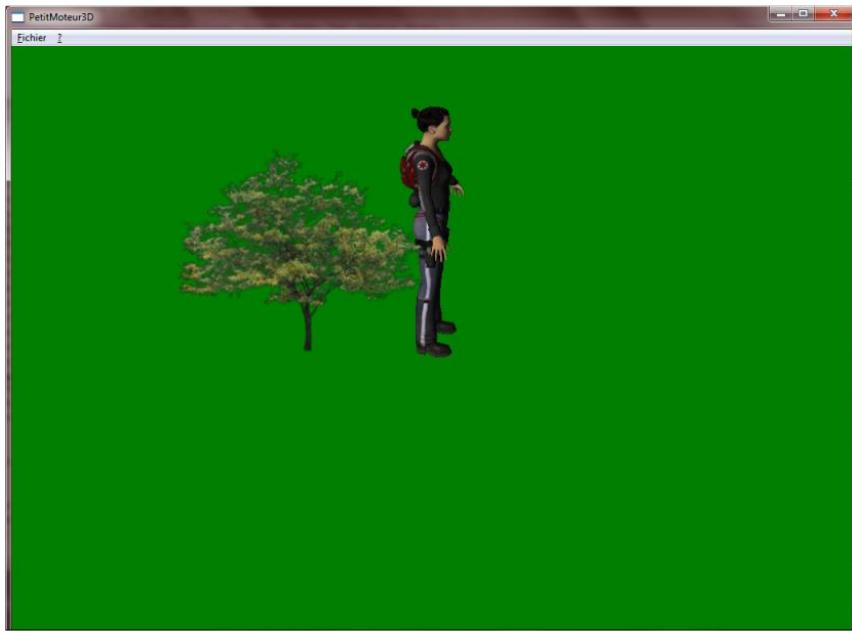
```
//Initialiser et sélectionner les « constantes » de l'effet
ShadersParams sp ;

//sp.matWVP = XMMatrixIdentity() ; //retirez cette ligne
sp.matWVP = XMMatrixTranspose (matPosDim) ;
```

8. Dans la fonction InitObjets de CMoteur, ajoutez un appel à la fonction SetPosDim immédiatement après la création de l'objet CSpriteTemp :

```
//Création d'un objet sprite
pSprite = new CSpriteTemp ("tree02s.dds", pDispositif) ;
pSprite->SetPosDim(200 400) ;
```

Vous devriez maintenant obtenir quelque chose du genre :



Atelier

9.8 Méthode 2 - L'afficheur de sprites

Maintenant que nous avons réglé les problèmes techniques liés à l'affichage d'un sprite, nous pouvons implanter notre « Afficheur de sprites ».

La classe CAfficheurSprite

1. Presque tous les éléments dont nous avons besoin sont dans la classe CSpriteTemp, nous commençons donc par en faire une copie en copiant les fichiers **SpriteTemp.h** dans **AfficheurSprite.h** et **SpriteTemp.cpp** dans **AfficheurSprite.cpp**.
2. Puis nous ajoutons ces fichiers à notre projet (ajouter un élément existant). Vous pouvez retirer SpriteTemp.h et SpriteTemp.cpp du projet, mais ne les détruisez pas, nous les conserverons pour références.
3. Modifications de base :
 - remplacer le `#include "SpriteTemp.h"` dans le fichier AfficheurSprite.cpp par `#include "AfficheurSprite.h"`
 - remplacer les occurrences du mot **CSpriteTemp** par **CAfficheurSprite** dans les fichiers **AfficheurSprite.h** et **AfficheurSprite.cpp**.
4. Modifiez le constructeur de CAfficheurSprite dans le fichier AfficheurSprite.h et AfficheurSprite.cpp pour qu'il ne fasse plus référence à une texture :

```
CAfficheurSprite(CDispositifD3D11* _pDispositif);
```

5. Déclarez la classe **CSprite** dans le fichier CAfficheurSprite.h. Je la déclare comme une sous-classe *private* de CAfficheurSprite puisque je ne m'en sers que comme structure de données :

```
class CSprite
{
public :
    ID3D11ShaderResourceView * pTextureD3D;

    XMATRIX matPosDim;
    bool bPanneau;
    CSprite()
        : bPanneau(false)
        , pTextureD3D(nullptr)
    {
    }
};
```

Le booléen **bPanneau** est ajoutée pour considérations futures (voir la section sur les Panneaux – *billboards*)

6. Retirez les variables **matPosDim**, **pTextureD3D**, **dimX** et **dimY** de la classe CAfficheurSprite.
7. Nos sprites seront conservés dans un vecteur. Ajoutez l'énoncé **#include <vector>** à la suite des autres includes du fichier AfficheurSprite.h. Puis déclarez la variable suivante dans la section *private* de CAfficheurSprite :

```
// Tous nos sprites
std ::vector<std ::unique_ptr<CSprite>> tabSprites;
```

8. Déclarez la fonction AjouterSprite dans la section *public* de la classe CAfficheurSprite :

```
void AjouterSprite(string NomTexture, int _x, int _y, int _dx=0, int _dy=0);
```

Cette fonction permettra d'ajouter un sprite à notre liste. Elle s'occupera de le charger (au besoin) et d'initialiser les valeurs de position-dimensions (en remplacement de SetPosDim...).

9. Définissez-la ainsi :

```
void CAfficheurSprite::AjouterSprite(const std ::string& NomTexture,
    int _x, int _y,
    int _dx, int _dy)
{
    float x, y, dx, dy;
    float posX, posY;
    float facteurX, facteurY;

    // Initialisation de la texture
```

```

CGestionnaireDeTextures& TexturesManager =
    CMoteurWindows::GetInstance().GetTextureManager();

std::wstring ws(NomTexture.begin(), NomTexture.end());

std::unique_ptr<CSprite> pSprite = std ::make_unique<CSprite>();
pSprite-> pTextureD3D =
    TexturesManager.GetNewTexture(ws.c_str(), pDispositif)-
>GetD3DTexture();

// Obtenir les dimensions de la texture si _dx et _dy sont à 0;
if (_dx == 0 && _dy == 0)
{
    ID3D11Resource* pResource;
    ID3D11Texture2D *pTextureInterface = 0;
    pSprite-> pTextureD3D->GetResource(&pResource);
    pResource->QueryInterface<ID3D11Texture2D>(&pTextureInterface);
    D3D11_TEXTURE2D_DESC desc;
    pTextureInterface->GetDesc(&desc);

    DXRelacher(pResource);
    DXRelacher(pTextureInterface);

    dx = float(desc.Width);
    dy = float(desc.Height);
}
else
{
    dx = float(_dx);
    dy = float(_dy);
}

// Dimension en facteur
facteurX = dx * 2.0f / pDispositif->GetLargeur();
facteurY = dy * 2.0f / pDispositif->GetHauteur();

// Position en coordonnées logiques
// 0,0 pixel = -1,1
x = float(_x);
y = float(_y);

posX = x * 2.0f / pDispositif->GetLargeur() - 1.0f;
posY = 1.0f - y * 2.0f / pDispositif->GetHauteur();

pSprite->matPosDim = XMMatrixScaling(facteurX, facteurY, 1.0f) *
    XMMatrixTranslation(posX, posY, 0.0f);

// On l'ajoute à notre vecteur
tabSprites.push_back(std ::move(pSprite));
}

```

Comme vous pouvez le constater, le début de la fonction a été emprunté au constructeur de CSpriteTemp et le reste est une reprise de la fonction SetPosDim de CSpriteTemp. Seulement la dernière ligne a été ajoutée. Notez l'utilisation d'un pointeur pour la création du sprite.

10. Retirez de la classe CAfficheurSprite la fonction SetPosDim.

11. Modifiez le constructeur de CAfficheurSprite pour qu'il n'initialise plus matPosDim et pour qu'il ne s'occupe plus du chargement de la texture. Il devrait maintenant ressembler à ceci :

```
CAfficheurSprite::CAfficheurSprite( CDispositifD3D11* _pDispositif)
    : pDispositif(_pDispositif)
    , pVertexBuffer(nullptr)
    , pConstantBuffer(nullptr)
    , pEffet(nullptr)
    , pTechnique(nullptr)
    , pPasse(nullptr)
    , pVertexLayout(nullptr)
    , pSampleState(nullptr)
{
    // Création du vertex buffer et copie des sommets
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));

    bd.Usage = D3D11_USAGE_IMMUTABLE;
    bd.ByteWidth = sizeof(sommets);
    bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    bd.CPUAccessFlags = 0;

    D3D11_SUBRESOURCE_DATA InitData;
    ZeroMemory(&InitData, sizeof(InitData));
    InitData.pSysMem = sommets;

    DXEssayer(pDXDevice->CreateBuffer(&bd, &InitData, &pVertexBuffer),
        DXE_CREATIONVERTEXBUFFER);

    // Initialisation de l'effet
    InitEffet();
}
```

13. La fonction **Draw** n'est en réalité que modifiée un peu pour afficher tous les sprites dans une boucle (un peu comme nous le faisions pour le mesh), mais certaines lignes ont du changer de place, en voici donc la nouvelle version :

```
void CAfficheurSprite::Draw()
{
    // Obtenir le contexte
    ID3D11DeviceContext* pImmediateContext =
        pDispositif->GetImmediateContext();

    // Choisir la topologie des primitives
    pImmediateContext->IASetPrimitiveTopology(
        D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

    // Source des sommets
    UINT stride = sizeof( CSommetSprite );
    const UINT offset = 0;
    pImmediateContext->IASetVertexBuffers( 0, 1, &pVertexBuffer, &stride,
        &offset );
```

```

// input layout des sommets
pImmediateContext->IASetInputLayout( pVertexLayout );

// Le sampler state
ID3DX11EffectSamplerVariable* variableSampler;
variableSampler = pEffet->GetVariableByName("SampleState")->AsSampler();
variableSampler->SetSampler(0, pSampleState);

ID3DX11EffectConstantBuffer* pCB =
    pEffet->GetConstantBufferByName("param");
ID3DX11EffectShaderResourceVariable* variableTexture;
variableTexture =
    pEffet->GetVariableByName("textureEntree")->AsShaderResource();

pDispositif->ActiverMelangeAlpha();

// Faire le rendu de tous nos sprites
for (int i=0;i<tabSprites.size();++i)
{
    // Initialiser et sélectionner les « constantes » de l'effet
    ShadersParams sp;
    sp.matWVP = XMMatrixTranspose(tabSprites[i]->matPosDim);
    pImmediateContext->UpdateSubresource( pConstantBuffer, 0, nullptr,
                                            &sp, 0, 0 );

    pCB->SetConstantBuffer(pConstantBuffer);

    // Activation de la texture
    variableTexture->SetResource(tabSprites[i]->pTextureD3D);

    pPasse->Apply(0, pImmediateContext);

    // **** Rendu de l'objet
    pImmediateContext->Draw( 6, 0 );
}

pDispositif->DesactiverMelangeAlpha();
}

```

L'utilisation de l'afficheur

1. Il nous faut d'abord remplacer le #include "spritetemp.h" au début du fichier moteur.h pour le remplacer par #include "afficheursprite.h".
2. Puis nous pouvons modifier la fonction InitObjets de CMoteur pour qu'elle utilise l'afficheur de sprites et trois sprites :

```

bool InitObjets()
{
    // Constructeur avec format binaire
    std ::unique_ptr<COBJETMESH> pMesh =
std ::make_unique<COBJETMESH>("./modele\\jin\\jin.OMB", pDispositif);
    // Puis, il est ajouté à la scène
    ListeScene.push_back(std ::move(pMesh));

    // Création de l'afficheur de sprites et ajout des sprites
    std ::unique_ptr<CAfficheurSprite> pAfficheurSprite =
std ::make_unique<CAfficheurSprite>(pDispositif);

```

```
pAfficheurSprite->AjouterSprite("tree02s.dds", 200,400);
pAfficheurSprite->AjouterSprite("tree02s.dds", 500,500, 100, 100);
pAfficheurSprite->AjouterSprite("tree02s.dds", 800,200, 100, 100);

ListeScene.push_back(std::move(pAfficheurSprite));

return true;

}
```

J'ai affiché trois fois la même texture, à des endroits différents. La méthode 2 présente un avantage marqué sur la création des sprites et un avantage de performance non négligeable pour le rendu.

Pourrait-on faire mieux ?

Eh oui! Deux améliorations pourraient être apportées :

1. Les textures pourraient être référencées dans les shaders (l'effet) par un tableau de textures (*textures array*), donc la ressource correspondant à notre texture n'aurait pas besoin d'être modifiée à chaque tour de boucle puisque nous pourrions le faire une seule fois avant la boucle et utiliser un indice comme paramètre supplémentaire. C'est particulièrement rapide si nous avons beaucoup de sprites utilisant un nombre limité de textures.
2. Les paramètres de tous nos sprites pourraient être placées en mémoire graphique en une seule étape et l'utilisation de **ID3D11DeviceContext::DrawInstanced** (au lieu de Draw) serait alors possible. La carte graphique s'occuperait alors de la boucle...

9.9 Désactiver/Activer le Z-buffer

Certaines situations vous demanderont peut-être d'activer ou de désactiver le Z-buffer au besoin.

Pour obtenir cet effet, il est nécessaire de créer une seconde vue de type **ID3D11DepthStencilView**, elle est identique à celle que vous utilisez jusqu'à maintenant sauf que son paramètre **DepthEnable** est à **false**. La fonction **ID3D11DeviceContext::OMSetDepthStencilState** vous permet de modifier la vue et donc d'activer ou de désactiver le Z-buffer.

Le principe d'initialisation et d'utilisation est similaire à ce que nous avons fait pour le mélange alpha.

Atelier

9.10 Affichage sur un panneau (*billboarding*)

Les panneaux sont en réalité des sprites, très semblables à ceux que nous avons implantés dans les sections précédentes. La principale différence est qu'ils sont placés dans l'espace de la scène au lieu de l'espace-écran. Plusieurs utilisations sont possibles pour les panneaux :

- systèmes de particules : les particules sont des panneaux généralement assez petits qui sont superposés avec différents degrés de transparence pour produire des effets difficiles à obtenir avec des objets 3D comme le feu, la fumée, la poussière, le feuillage, etc.
- imposteurs : les imposteurs sont des panneaux qui remplacent des objets 3D éloignés ou difficiles à rendre (comme les arbres d'une forêt).
- Effets cinématiques, les panneaux sont souvent utilisés pour y « faire jouer » une animation par exemple : une explosion ou un écran télé.

L'affichage sur un panneau (*billboard*) n'est pas réellement de l'affichage 2D puisque notre panneau est un objet 3D disposé dans la scène. Mais comme on pourrait y appliquer des techniques d'animation 2D, nous le traiterons dans ce chapitre. Un panneau permettant l'affichage de sprite est en réalité un peu plus simple qu'un panneau 3D puisque nous n'y mettrons pas d'éclairage donc pas de matériaux ni de normales. Par contre son affichage devra tenir compte de plusieurs facteurs importants :

- fixe ou orientable : un panneau fixe est un objet 3D dont l'orientation originale n'est pas modifiée lors de l'animation de la scène. Un panneau orientable modifie son orientation pour toujours faire face à la caméra (façons possibles : linéaire-Z, linéaire-XZ, linéaire-YZ, linéaire-XYZ). Certaines applications utilisent des panneaux semi-orientables (orientables selon certains angles seulement).
- position dans le monde, position de la caméra et matrice de projection : les panneaux étant placés dans le monde, ils devront utiliser WVP pour être affichés au bon endroit.
- Animation ou non : une méthode de modifier l'affichage devra être conçue pour les panneaux animables (avec cinématique par exemple).

Panneau orientable linéaire-Z

Ce type de panneaux est très intéressant pour nous parce qu'il ressemble beaucoup aux sprites. En réalité notre classe CAfficheurSprite est presque

prête à afficher ce genre de panneaux. La classe CSprite, elle, a déjà prévu une partie du travail avec la déclaration du booléen bPanneau.

Certaines choses doivent être implantées pour que nous puissions utiliser les panneaux orientables linéaire-Z :

1. La dimension en unités-monde du panneau ;
2. La position du panneau dans le monde ;
3. Le calcul de la matrice de position-dimension en fonction de la position du panneau, de la position de la caméra et de la matrice de projection ;
4. Le calcul à nouveau de la matrice de position-dimension lorsque la caméra bouge.

Pour les points 1 et 2, il nous suffit de créer la classe **CPanneau** que nous dériverons de CSprite.

1. Déclarez la classe CPanneau comme une sous-classe *private* de CAfficheurSprite :

```
class CPanneau : public CSprite
{
public :
    XMFLOAT3 position;
    XMFLOAT2 dimension;

    CPanneau()
    {
        bPanneau = true;
    }
};
```

2. Déclarez la fonction AjouterPanneau dans la section *public* de la classe CAfficheurSprite :

```
void AjouterPanneau(const std ::string& NomTexture, const XMFLOAT3&
_position,
                     float _dx=0.0f, float _dy=0.0f);
```

3. Définissez-la ensuite :

```
void CAfficheurSprite::AjouterPanneau(const std ::string& NomTexture,
                                         const XMFLOAT3& _position,
                                         float _dx, float _dy)
{
    // Initialisation de la texture
    CGestionnaireDeTextures& TexturesManager =
        CMoteurWindows::GetInstance().GetTextureManager();

    std::wstring ws(NomTexture.begin(), NomTexture.end());

    std::unique_ptr<CPanneau> pPanneau = std::make_unique<CPanneau>();
```

```

pPanneau->pTextureD3D =
    TexturesManager.GetNewTexture(ws.c_str(), pDispositif)-
>GetD3DTexture();

// Obtenir la dimension de la texture si _dx et _dy sont à 0;
if (_dx == 0.0f && _dy == 0.0f)
{
    ID3D11Resource* pResource;
    ID3D11Texture2D *pTextureInterface = 0;
    pPanneau-> pTextureD3D->GetResource(&pResource);
    pResource->QueryInterface<ID3D11Texture2D>(&pTextureInterface);
    D3D11_TEXTURE2D_DESC desc;
    pTextureInterface->GetDesc(&desc);

    DXRelacher(pResource);
    DXRelacher(pTextureInterface);

    pPanneau->dimension.x = float(desc.Width);
    pPanneau->dimension.y = float(desc.Height);

    // Dimension en facteur
    pPanneau->dimension.x = pPanneau->dimension.x * 2.0f / pDispositif-
>GetLargeur();
    pPanneau->dimension.y = pPanneau->dimension.y * 2.0f / pDispositif-
>GetHauteur();
}
else
{
    pPanneau->dimension.x = float(_dx);
    pPanneau->dimension.y = float(_dy);
}

// Position en coordonnées du monde
const XMATRIX& viewProj =
CMoteurWindows::GetInstance().GetMatViewProj();
pPanneau->position = _position;

pPanneau->matPosDim = XMMatrixScaling(pPanneau->dimension.x,
    pPanneau->dimension.y, 1.0f) *
XMMatrixTranslation(pPanneau->position.x,
    pPanneau->position.y, pPanneau->position.z) *
viewProj;

// On l'ajoute à notre vecteur
tabSprites.push_back(std ::move(pPanneau));
}

```

Un peu d'explications

Il s'agit d'une quasi-reprise de la fonction AjouterSprite.

En entrée, la position est en **coordonnées du monde**.

En entrée, la dimension est aussi en coordonnées du monde, mais si nous utilisons 0.0f comme valeurs, les dimensions de la texture (en pixels) sont converties en unités du monde. C'est pratique pour tester notre affichage ou si la texture est très petite (comme pour des particules), mais l'affichage nous donnera des résultats différents selon les dimensions de notre application.

Nous devons interroger le moteur pour obtenir la matrice VP.

4. Copiez le fichier « **grass_v1_basic_tex.dds** » dans votre dossier projet à partir du dossier **Samples\Media\IslandScene**.
5. Ajoutez les lignes suivantes à la fonction **CMoteur::InitObjets** (j'ai temporairement placé les sprites en commentaires) :

```
// ajout de panneaux
pAfficheurSprite->AjouterPanneau("grass_v1_basic_tex.dds",
    XMFLOAT3(1.0f, 0.0f, 1.0f));
pAfficheurSprite->AjouterPanneau("grass_v1_basic_tex.dds",
    XMFLOAT3(0.0f, 0.0f, -1.0f));
pAfficheurSprite->AjouterPanneau("grass_v1_basic_tex.dds",
    XMFLOAT3(-1.0f, 0.0f, 0.5f));
pAfficheurSprite->AjouterPanneau("grass_v1_basic_tex.dds",
    XMFLOAT3(-0.5f, 0.0f, 1.0f));
pAfficheurSprite->AjouterPanneau("grass_v1_basic_tex.dds",
    XMFLOAT3(-2.0f, 0.0f, 2.0f));

//pAfficheurSprite->AjouterSprite("tree02s.dds", 200,400);
//pAfficheurSprite->AjouterSprite("tree02s.dds", 500,500, 100, 100);
//pAfficheurSprite->AjouterSprite("tree02s.dds", 800,200, 100, 100);
```

6. Essayez votre application. Les panneaux ne s'affichent pas ! Pourquoi ?
7. La raison est simple, nous effectuons une création d'objets dont le positionnement demande de connaître la matrice VP, or celle-ci n'est pas encore initialisée puisque nous créons les objets en premier lors de l'initialisation de la scène. Ce n'était pas important jusqu'à maintenant.

Pour éviter que ce problème nous dérange encore, faites la modification suivante à la fonction **CMoteur::InitScene** :

```
virtual int InitScene()
{
    // Initialisation des matrices View et Proj
    // Dans notre cas, ces matrices sont fixes
    matView = XMMatrixLookAtLH( XMVectorSet( 0.0f, 3.0f,-5.0f, 1.0f ),
                                XMVectorSet( 0.0f, 0.0f, 0.0f, 1.0f ),
                                XMVectorSet( 0.0f, 1.0f, 0.0f, 1.0f ) );

    float champDeVision = XM_PI/4;    // 45 degrés
    float ratioDAspect = pDispositif->GetLargeur()/pDispositif->GetHauteur();

    float planRapproche = 1.0;
    float planEloigne = 30.0;

    matProj = XMMatrixPerspectiveFovLH( champDeVision, ratioDAspect,
                                         planRapproche, planEloigne );

    // Calcul de VP à l'avance
    matViewProj = matView * matProj;

    // Initialisation des objets 3D - création et/ou chargement
    if (!InitObjets()) return 1;
```

Ces deux lignes étaient au début de la fonction

```
    return 0;  
}
```

8. Essayez votre application. Les panneaux s'affichent bien.

Notez que les sprites devraient toujours être affichés **après** les panneaux puisque ces derniers sont de vrais objets 3D.

Est-ce complet ?

Non. Il nous manque le rafraîchissement du calcul de la position du panneau lorsque la matrice de vision (la caméra) ou la matrice de projection sont modifiées.

Dans mon cas, la matrice de vision et la matrice de projection ne sont pas modifiées donc il n'y a rien à ajouter 😊 .

Dans une application où il y a beaucoup de mouvements de caméra, il vaudrait peut-être mieux avoir une classe spéciale (CAfficheurPanneau) dont la fonction Draw utiliserait VP pour calculer WVP juste avant le rendu. D'autres solutions sont possibles.

Pourrait-on faire mieux ?

Oui, nous pourrions évidemment utiliser les améliorations suggérées pour l'afficheur de sprite soient les tableaux de texture et l'instanciation.

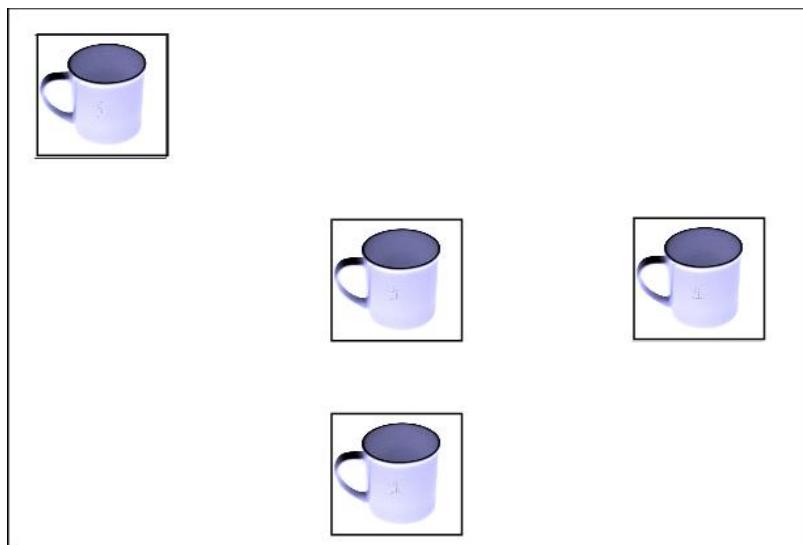
9.12 Orientation linéaire

Nous avons implanté un panneau orientable linéaire-Z dans la section précédente, mais qu'est-ce que c'est exactement ? Et pourquoi parlons nous aussi de linéaire-XZ, linéaire-YZ ou linéaire-XYZ ?

Linéaire-Z

Un panneau linéaire-Z est un peu plus facile à implanter que les autres techniques puis qu'il ne demande que la position dans le monde. Le panneau utilise seulement la distance Z et il est toujours parallèle à l'écran.

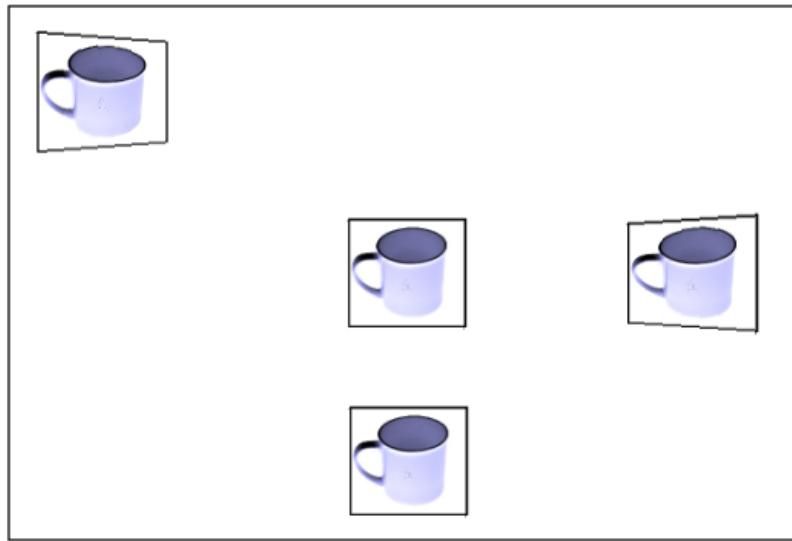
Donc, pour un même Z, les panneaux ont des dimensions identiques à l'écran :



Linéaire-XZ et Linéaire-YZ

Le panneau linéaire-YZ est un peu plus fréquent que le XZ, mais tous deux fonctionnent de la même façon. Il nous demande de plus de calculer une rotation sur l'axe des Y pour que l'orientation du panneau soit plus réaliste. Cette rotation est relativement facile à obtenir en calculant un vecteur position, en le transformant en vecteur 2D avec x et z seulement puis en le normalisant, la valeur x est le cosinus de l'angle à appliquer et la valeur z est le sinus. Nous pouvons alors construire facilement une matrice de rotation. Pour linéaire-XZ, c'est le même principe, mais avec y comme cosinus et z comme sinus.

Donc, pour un même Z, les panneaux seront disposés ainsi à l'écran :

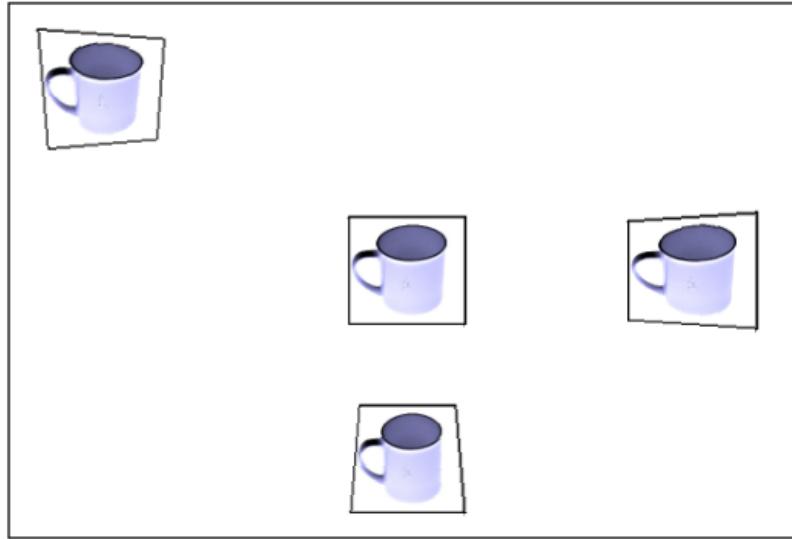


Notez que les proportions peuvent varier selon le point de rotation choisi.

Linéaire-XYZ

Le panneau linéaire-XYZ tient compte de deux rotations soient nos rotations YZ et XZ (ou d'une rotation 3D directement calculée pour les experts en math).

Donc, pour un même Z, les panneaux seront disposés ainsi à l'écran :



9.13 Le texte

Il existe différentes approches à l'affichage de texte sur des surfaces 3D. Nous les verrons plus loin.

IMPORTANT

Selon l'approche choisie, le résultat sera **toujours** une des deux options suivantes :

- Le texte complet a été dessiné d'une façon ou d'une autre sur un **sprite** puis le sprite est positionné sur le *backbuffer*.
- Le texte est affiché sur le *backbuffer* à partir d'un sprite contenant tous les caractères (et leurs caractéristiques).

Certaines bibliothèques, comme les versions précédentes de DirectX, offrent un choix entre ces deux options et permettaient dans certains cas de les combiner.

Pourquoi utiliser des sprites ?

Il faut comprendre que l'écriture de texte de type vectoriel au moyen de polices de caractères (**TrueType**, **OpenType** ou autres) est présentement incompatible avec des affichages rapides (plus de 30 images/seconde). Elle le sera sûrement un jour.

Donc, la plupart des systèmes d'écriture utilisent des méthodes du même type que celui décrit plus haut, c'est le cas du « vieux » GDI et de GDI++. Le calcul et le positionnement des caractères sont par contre presque toujours réalisés au niveau du CPU.

Dans le cas de texte dans un contexte d'animation 3D, il n'est pas très intéressant de « réécrire » le texte à chaque affichage. Celui-ci ne changeant sûrement pas aussi rapidement, il suffira de l'écrire lorsque le texte sera modifié.

L'utilisation d'un sprite pour le texte ou à la rigueur d'un ensemble de sprites (1 pour chaque lettre) est nettement plus intéressante.

DirectX et le texte

Les versions précédentes de DirectX (pré DX 10) utilisaient une classe d'affichage de texte implantée dans la bibliothèque D3DX.

Mais la version 10 a vu cette classe disparaître, DirectX se préparait à l'affichage de texte via le GPU. C'est ce qui est arrivé avec DirectWrite et Direct2D, apparu avec la version 11 du SDK, mais conçu pour DX 10.

En théorie, l'affichage avec DirectWrite devrait être très rapide et régler beaucoup des problèmes de performance liés au texte, mais voilà, il est difficile à faire coexister avec DX 11. Certains s'y sont essayés avec des résultats plus ou moins probants. Peut-être que la prochaine version du SDK nous amènera un DirectWrite 11 ?

Les extensions DirectXTK utilisent un texte similaire à celui que nous allons planter ici.

Solution temporaire

La meilleure solution (la plus rapide) serait de définir une feuille de caractères en mémoire graphique (*font sheet*). Mais cette solution implique d'identifier la position et les « extents » de chaque caractère sur la feuille et d'écrire un shader pour le faire. Cette solution est malheureusement un peu longue à mettre en place et encore plus complexe en français. Vous pouvez lire plus d'information à ce sujet sur le site de Frank Luna : <http://www.d3dcoder.net/Data/Resources/SpritesAndText.pdf>.

Quant à nous, nous planterons dans cette section une solution « temporaire » soit l'utilisation de GDI+ pour réaliser nos affichages. Cette méthode correspond presque à ce qui était fait en DirectX 9 (ID3DXFont) ou avec **Freetype**. Je dis « presque » parce que notre solution est moins complète, mais elle est plus rapide et complètement programmable.

Avantages :

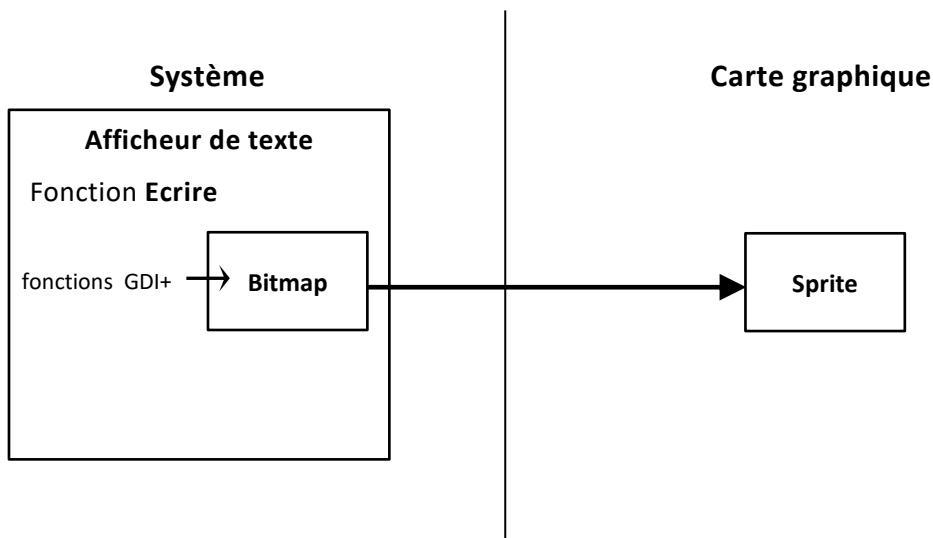
- GDI+ est très complet, texte et graphiques (lignes, rectangles, ellipses, etc).
- La méthode utilisée pourra servir d'interface avec d'autres bibliothèques.
- Nous pourrons expérimenter avec les échanges mémoires système-mémoire vidéo fréquents et parfois volumineux.

Désavantages :

- GDI+ est considéré comme « *legacy graphics* » c'est-à-dire qu'il sera remplacé bientôt (quand ?).
- GDI+ travaille au niveau du CPU et de la mémoire système, il faudra transférer le résultat vers la carte graphique.
- GDI+ est théoriquement plus lent. En pratique, nos affichages de texte seront plutôt limités et la partie affichage du texte est négligeable par rapport au transfert du sprite contenant le résultat d'affichage.

9.14 Principe de travail avec GDI+

1. Initialisation de GDI+ avant de créer des zones de texte.
2. Pour chaque zone de texte, nous créeros un « **Afficheur de texte** », celui-ci servira à initialiser un **sprite** contenant le résultat de l'affichage du texte, il servira aussi à modifier le texte et à transférer le sprite vers la carte graphique.
3. Les zones de texte seront modifiées **au besoin seulement**, limitant ainsi les échanges avec la carte graphique. La fonction Ecrire de notre afficheur de texte écrit le nouveau texte sur le bitmap qui est par la suite copié sur le sprite en mémoire graphique. IMPORTANT : le bitmap et le sprite devraient avoir la dimension minimale pour recevoir le texte, trop petits, et le texte ne sera pas affiché au complet, trop grands, et le transfert entre le bitmap et le sprite sera inutilement plus long.
4. Les zones de textes sont traitées comme des sprites et seront affichées par notre **afficheur de sprites**.



5. GDI+ doit être fermé à la fin de l'application.

Atelier

9.15 La classe CAfficheurTexte

La classe CAfficheurTexte nous permettra d'utiliser GDI+ pour réaliser des affichages (surtout du texte) sur un bitmap. Celui-ci sera au besoin transféré sur un sprite (une texture) en mémoire graphique.

Voici la déclaration de cette classe, les variables membres et les fonctions seront expliquées plus loin :

AfficheurTexte.h

```
#pragma once

#include <string.h>
#include "dispositifD3D11.h"
#include <Gdiplus.h>
#include <string>
#pragma comment(lib, "gdiplus.lib")

namespace PM3D
{
    class CAfficheurTexte
    {
        public :
            CAfficheurTexte(CDispositifD3D11* pDispositif, int largeur, int hauteur,
                Gdiplus ::Font* pPolice);
            ~CAfficheurTexte();
            void Ecrire(const std::wstring& s);
            ID3D11ShaderResourceView* GetTextureView() { return pTextureView; }

            static void Init();
            static void Close();

        private:
            UINT TexWidth;
            UINT TexHeight;

            ID3D11Texture2D *pTexture;
            IDXGISurface1* pSurface;
            ID3D11ShaderResourceView* pTextureView;
            CDispositifD3D11* pDispo;

            Gdiplus ::Font* pFont;
            std::unique_ptr<Gdiplus::Bitmap> pCharBitmap;
            std::unique_ptr<Gdiplus::Graphics> pCharGraphics;
            std::unique_ptr<Gdiplus::SolidBrush> pBlackBrush;

            // Variables statiques pour GDI+
            static ULONG_PTR token;
    };
}

} // namespace PM3D
```

Fonctions statiques

Nous avons deux fonctions **statiques**, celles-ci nous permettant d'initialiser GDI+ avant de créer des objets CAfficheurTexte (fonction **Init**) et de fermer GDI+ à la fin de l'application (fonction **Close**). Nous pourrons ainsi utiliser GDI+ pour initialiser des ressources telles polices (*font*), crayons (*pen*) et brosses (*brush*). Ces ressources pourront être passées en paramètres aux objets CAfficheurTexte et au besoin être utilisées par plusieurs objets.

La variable statique token

Déclarez dans le fichier AfficheurTexte.cpp la ligne suivante :

```
ULONG_PTR CAfficheurTexte::token = 0;
```

Cette variable sert lors de l'initialisation de GDI+ et permet un accès multitâche à GDI+.

La fonction Init

```
void CAfficheurTexte::Init()
{
    Gdiplus::GdiplusStartupInput startupInput(0, TRUE, TRUE);
    Gdiplus::GdiplusStartupOutput startupOutput;

    GdiplusStartup(&token, &startupInput, &startupOutput);
}
```

La fonction Close

```
void CAfficheurTexte::Close()
{
    Gdiplus::GdiplusShutdown(token);
}
```

Le constructeur

```

CAfficheurTexte::CAfficheurTexte(CDispositifD3D11* pDispositif, int largeur,
int hauteur, Gdiplus ::Font* pPolice)
    : pDispo(pDispositif)
    , TexWidth(largeur)
    , TexHeight(hauteur)
    , pFont(pPolice)
    , pTexture(nullptr)
    , pSurface(nullptr)
    , pTextureView(nullptr)
    , pCharBitmap(nullptr)
    , pCharGraphics(nullptr)
    , pBlackBrush(nullptr)
{
    // Créer le bitmap et un objet GRAPHICS (un dessinateur)
    pCharBitmap = std ::make_unique<Gdiplus ::Bitmap>(TexWidth, TexHeight,
PixelFormat32bppARGB);
    pCharGraphics = std::make_unique<Gdiplus::Graphics>(pCharBitmap.get());

    // Paramètres de l'objet Graphics
    pCharGraphics->SetCompositingMode(Gdiplus ::CompositingModeSourceOver);
    pCharGraphics-
    >SetCompositingQuality(Gdiplus ::CompositingQualityHighSpeed);
    pCharGraphics-
    >SetInterpolationMode(Gdiplus::InterpolationModeHighQuality);
    pCharGraphics->SetPixelOffsetMode(Gdiplus::PixelOffsetModeHighSpeed);
    pCharGraphics->SetSmoothingMode(Gdiplus::SmoothingModeNone);
    pCharGraphics->SetPageUnit(Gdiplus::UnitPixel);
    Gdiplus::TextRenderingHint hint = Gdiplus::TextRenderingHintAntiAlias;
//TextRenderingHintSystemDefault;
    pCharGraphics->SetTextRenderingHint(hint);

    // Un brosse noire pour le remplissage
    // Notez que la brosse aurait pu être passée
    // en paramètre pour plus de flexibilité
    pBlackBrush = std::make_unique<Gdiplus::SolidBrush>(Gdiplus ::Color(255,
0, 0, 0));

    // On efface le bitmap (notez le NOIR TRANSPARENT...)
    pCharGraphics->Clear(Gdiplus ::Color(0, 0, 0, 0));

    // Nous pourrions ici écrire une valeur initiale sur le bitmap
    // std ::wstring s=L"Valeur initiale";
    // pCharGraphics->DrawString( s.c_str(), s.size(), pFont, PointF( 0.0f,
0.0f ), pBlackBrush.get() );

    // Accéder aux bits du bitmap
    Gdiplus::BitmapData bmData;
    pCharBitmap->LockBits(&Gdiplus::Rect(0, 0, TexWidth, TexHeight),
Gdiplus ::ImageLockModeRead, PixelFormat32bppARGB, &bmData);

    // Création d'une texture de même dimension sur la carte graphique
    D3D11_TEXTURE2D_DESC texDesc;
    texDesc.Width = TexWidth;
    texDesc.Height = TexHeight;
    texDesc.MipLevels = 1;
    texDesc.ArraySize = 1;
    texDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
    texDesc.SampleDesc.Count = 1;
    texDesc.SampleDesc.Quality = 0;
}

```

```

texDesc.Usage = D3D11_USAGE_DEFAULT;
texDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
texDesc.CPUAccessFlags = 0;
texDesc.MiscFlags = 0;

D3D11_SUBRESOURCE_DATA data;
data.pSysMem = bmData.Scan0;
data.SysMemPitch = TexWidth * 4;
data.SysMemSlicePitch = 0;

// Création de la texture à partir des données du bitmap
DXEssayer(pDispo->GetD3DDevice()->CreateTexture2D(&texDesc, &data,
&pTexture));

// Création d'un « resourve view » pour accéder facilement à la texture
// (comme pour les sprites)
D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc;
srvDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
srvDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MipLevels = 1;
srvDesc.Texture2D.MostDetailedMip = 0;

DXEssayer(pDispositif->GetD3DDevice()->CreateShaderResourceView(pTexture,
&srvDesc, &pTextureView));

pCharBitmap->UnlockBits(&bmData);
}

```

1. La texture et sa vue sont créées ici. La texture n'est pas chargée, mais plutôt créée dynamiquement, il faudra donc la relâcher dans notre destructeur. La vue, elle, sera relâchée par l'afficheur de sprites.
2. La texture est initialisée à partir des données du bitmap. Pour le moment, celui-ci est vide (noir transparent), mais nous pourrions lui donner une valeur initiale.
3. Pour l'initialisation de l'objet Graphics et pour plus de détails sur les possibilités de GDI+, consultez MSDN.

Le destructeur

```

CAfficheurTexte::~CAfficheurTexte()
{
    DXRelacher(pTexture);
}

```

Nous nous contentons de relâcher ou de détruire les éléments créés dynamiquement dans le constructeur.

La fonction Ecrire

Cette fonction est la raison d'être de la classe CAfficheurTexte. Elle effectue les opérations suivantes :

1. Effacer le bitmap
2. Y écrire le nouveau texte
3. Transférer le bitmap vers la texture en mémoire graphique

```
void CAfficheurTexte::Ecrire(const std ::wstring& s)
{
    // Effacer
    pCharGraphics->Clear(Gdiplus ::Color(0, 0, 0, 0));

    // Écrire le nouveau texte
    pCharGraphics->DrawString(s.c_str(), static_cast<int>(s.size()), pFont,
Gdiplus::PointF(0.0f, 0.0f), pBlackBrush.get());

    // Transférer
    Gdiplus::BitmapData bmData;
    pCharBitmap->LockBits(&Gdiplus::Rect(0, 0, TexWidth, TexHeight),
Gdiplus::ImageLockModeRead, PixelFormat32bppARGB, &bmData);

    pDispo->GetImmediateContext()->UpdateSubresource(pTexture, 0, 0,
bmData.Scan0, TexWidth * 4, 0);

    pCharBitmap->UnlockBits(&bmData);
}
```

9.16 Utilisation de CAfficheurTexte

Ajout de la fonction CAfficheurSprite ::AjouterSpriteTexte

Notre texte sera sur une texture, donc il pourra fonctionner tout comme les sprites que nous avons implantés au début du chapitre. À la différence qu'il n'est pas chargé à partir d'un fichier, mais construit dynamiquement. Nous avons donc besoin d'une fonction pour ajouter notre vue à notre afficheur de sprites.

1. Déclarez la fonction AjouterSpriteTexte dans la classe CAfficheurSprite

```
void AjouterSpriteTexte(ID3D11ShaderResourceView* pTexture, int _x, int _y);
```

2. Définissez-la ainsi dans AfficheurSprite.h, il s'agit d'une version adaptée de la fonction AjouterSprite :

```
void CAfficheurSprite::AjouterSpriteTexte(
    ID3D11ShaderResourceView* pTexture, int _x, int _y)
{
    std ::unique_ptr<CSprite> pSprite = std ::make_unique<CSprite>();
    pSprite-> pTextureD3D = pTexture;

    // Obtenir la dimension de la texture;
    ID3D11Resource* pResource;
    ID3D11Texture2D *pTextureInterface = 0;
    pSprite-> pTextureD3D->GetResource(&pResource);
    pResource->QueryInterface<ID3D11Texture2D>(&pTextureInterface);
    D3D11_TEXTURE2D_DESC desc;
    pTextureInterface->GetDesc(&desc);

    DXRelacher(pResource);
    DXRelacher(pTextureInterface);

    const float dx = float(desc.Width);
    const float dy = float(desc.Height);

    // Dimension en facteur
    const float facteurX = dx * 2.0f / pDispositif->GetLargeur();
    const float facteurY = dy * 2.0f / pDispositif->GetHauteur();

    // Position en coordonnées logiques, 0,0 pixel = -1,1
    const float x = float(_x);
    const float y = float(_y);

    const float posX = x * 2.0f / pDispositif->GetLargeur() - 1.0f;
    const float posY = 1.0f - y * 2.0f / pDispositif->GetHauteur();

    pSprite->matPosDim = XMMatrixScaling(facteurX, facteurY, 1.0f) *
        XMMatrixTranslation(posX, posY, 0.0f);

    // On l'ajoute à notre vecteur
    tabSprites.push_back(std ::move(pSprite));
}
```

Exemple d'utilisation (dans CMoteur)

Initialisation

J'ai quant à moi placé l'initialisation dans la fonction **InitObjet**, juste avant le «push_back» de l'afficheur de sprite.

```
CAfficheurTexte::Init();
const Gdiplus::FontFamily oFamily(L"Arial", nullptr);
pPolice = std::make_unique<Gdiplus::Font>(&oFamily, 16.0f,
Gdiplus::FontStyleBold, Gdiplus::UnitPixel);
pTexte1 = std::make_unique<CAfficheurTexte>(pDispositif, 256, 256,
pPolice.get());

pAfficheurSprite->AjouterSpriteTexte(pTexte1->GetTextureView(), 0, 257);
```

Déclarations

Déclarez les variables suivantes (dans la section protected de CMoteur) :

```
// Pour le texte
std ::unique_ptr<CAfficheurTexte> pTexte1;
std::wstring str;

std ::unique_ptr<Gdiplus ::Font> pPolice;
```

Utilisation

Vous pouvez ensuite utiliser votre objet CAfficheurTexte (ou vos objets) pour « écrire » du texte au besoin, il n'est pas du tout utile de réécrire le texte à chaque image. Dans mon cas, je n'ai ajouté qu'un seul texte (pTexte1) et je l'utilise dans InitObjets ainsi :

```
pTexte1->Ecrire(L"Test de texte");
```

Mais comme pTexte1 appartient au moteur, vous pourriez l'utiliser n'importe où.

9.17 L'instanciation (intro)

Le rendu d'un grand nombre de petits objets, chacun créé à partir d'un petit nombre de polygones est une opération relativement lente. Je m'explique. Le rendu en lui-même est très court, mais les opérations de préparation du rendu sont les mêmes que pour des objets plus gros. C'est donc le ratio temps CPU/temps GPU qui est un peu « débalancé » par ce type de rendu.

Lorsque ces objets sont identiques (ou possèdent suffisamment de similitudes), il est possible d'utiliser une technique appelée **instanciation**. À l'origine, l'instanciation était principalement réalisée au niveau du CPU, optimisant quand même les opérations de préparation des objets, mais depuis quelques années, l'instanciation est disponible au niveau du GPU.

Cette technique nous permet de rendre plusieurs copies (instances) d'un même objet avec quelques modifications mineures entre les copies, par exemple la position, l'animation, la texture. C'est une technique très rapide, car la préparation et le « **draw** » ne sont effectués qu'une seule fois, libérant ainsi le CPU.

Instancier

L'instanciation est relativement facile à implanter. Nous n'avons qu'à aviser le pipeline (le GPU) du nombre de copies de notre objet à dessiner. Et le GPU les dessinera toutes, reprenant le pipeline pour chacune d'entre elles (pour chaque instance). Cette boucle nous permet ainsi de modifier certains paramètres pour chaque objet.

Données de l'instance

En plus d'aviser la carte graphique du nombre d'instances. Nous voudrons aussi fournir de l'information pour chaque instance comme une nouvelle matrice World dans nos cas. Nous pourrions aussi changer les éléments nécessaires à certains rendus comme les textures par exemple.

Nous pouvons modifier ces informations de différentes façons au moyen des tampons de constantes (*constant buffer*, comme nous en avons déjà utilisé), de tampons d'instance (*instance buffer*, presque essentielle) ou d'informations calculées dans les shaders. Nous pouvons évidemment utiliser une combinaison des trois méthodes.

10. Introduction aux dispositifs d'entrée

Extrait du SDK de DirectX: « L'API DirectInput est utilisé pour traiter des entrées en provenance d'un joystick ou d'un autre dispositif de jeu (par exemple: gamepad). L'utilisation de DirectInput pour le clavier et la souris n'est pas recommandée, l'utilisation des messages Windows devrait être utilisée à la place. L'API XInput a été introduit pour permettre le support du contrôleur Xbox 360 sous Windows ».

Dans ce chapitre, nous commencerons par regarder DirectInput (8) pour le clavier et pour la souris. DirectInput disparaîtra au profit de XInput dans les prochaines versions du SDK, mais pour le moment il existe encore beaucoup de composantes de jeu qui ne supportent pas XInput.

Objectifs du chapitre

- ✓ Implanter une classe pour l'accès aux données du clavier via DirectInput.
- ✓ Implanter une classe pour l'accès aux données de la souris via DirectInput.

10. Introduction aux dispositifs d'entrée

Théorie

10.1 DirectX

DirectInput est une composante de DirectX permettant l'accès aux périphériques d'entrée soient le clavier, la souris, le joystick, etc. En plus de permettre de manipuler des périphériques pour lesquels il n'existe aucun moyen d'accès dans l'API de Windows, DirectInput permet un accès « plus rapide » à tous les périphériques en communiquant directement avec les pilotes de ces périphériques plutôt que d'attendre et de gérer les messages de Windows.

DirectInput permet à une application de récupérer de l'information d'un périphérique d'entrée même quand l'application n'est pas au premier plan. Cette forme d'accès est évidemment très utile pour l'élaboration d'applications interactives en temps réel ou pour des applications de jeux (qui sont aussi une forme d'application interactive en temps réel).

Toutefois, DirectInput n'offre pas toujours un avantage réel pour la saisie de caractères au clavier ou pour les entrées de la souris, deux catégories d'opérations déjà bien implantées dans l'API de Windows. Lors de l'écriture de nos applications, il faudra alors bien déterminer la façon dont nous voulons interagir avec ces deux périphériques.

La version 8.0 de DirectInput n'est pas une refonte des versions précédentes même si elle introduit de nouvelles fonctions. Vous trouverez donc l'initialisation et les opérations qui s'en suivent un peu moins agréables que celles des autres versions de DirectX. Mais comme elles demeurent assez simples, vous n'aurez aucun problème à les utiliser.

10.2 Les objets DirectInput

L'objet DirectInput

L'objet DirectInput est un objet de type `IDirectInput8` qui représente le module DirectX DirectInput (une série de DLLs et de pilotes). L'objet DirectInput sert principalement à créer des dispositifs représentant des périphériques d'entrée comme le clavier, la souris, le joystick. L'objet DirectInput permet aussi de s'informer sur les possibilités matérielles de notre système afin de nous aider dans notre processus de création du dispositif.

L'objet DirectInputDevice – le dispositif DirectInput

Chaque objet DirectInputDevice est un objet de type `IDirectInputDevice8` qui représente un périphérique d'entrée comme le clavier, la souris, le joystick. Pour DirectInput, le mot *joystick* rassemble tous les périphériques autres que le clavier ou la souris. Un périphérique combinant différents types de dispositifs comme un clavier incorporant un *touchpad*, pourra être représenté par plus d'un objet DirectInputDevice.

Nous appellerons aussi les objets DirectInputDevice des dispositifs, mais la plupart du temps nous les identifierons par leur vrai nom de périphérique: clavier, souris, joystick...

Atelier

10.3 Une classe pour la gestion de DirectInput

Dans cette section, nous construirons une petite classe pour la gestion des objets DirectInput. Mais attention, cette classe n'est pas une classe de type « *wrapper class* », elle n'est pas non plus une très jolie classe, elle n'est qu'un moyen de simplifier l'organisation des opérations d'initialisation et de finalisation (ménage).

Les « *wrapper classes* », les « *classes d'emballage* » sont des classes dont le but est d'implanter en POO des API qui ne sont pas orientés objet. Par exemple, une librairie comme MFC est constituée d'une majorité de « *wrapper classes* ». Dans une « *wrapper class* », les opérations d'initialisation ainsi que la plupart des fonctions concernées sont redéfinies dans la classe. Ce type de classes est très intéressant, mais ce n'est pas ce que nous allons construire ici à cause des raisons suivantes:

- a) DirectX est déjà presque entièrement orienté objet. En effet, les objets COM ne sont pas des objets au sens C++ du terme, mais presque...
- b) Notre but dans ces notes est d'expliquer le fonctionnement des éléments de DirectX, pas de les cacher.
- c) Plus important encore, l'écriture et l'entretien de « *wrapper classes* » peuvent devenir très laborieux si vous désirez encapsuler toute la fonctionnalité de DirectX. Pire encore, une nouvelle version de DirectX vous demandera peut-être une révision complète de vos classes.

1. Ajoutez au projet la classe CDIManipulateur

2. Déclarez-y les variables suivantes dans la section *private*:

```
IDirectInput8* pDirectInput;
IDirectInputDevice8* pClavier;
IDirectInputDevice8* pSouris;
IDirectInputDevice8* pJoystick;

static bool bDejaInit;
```

3. Dans le constructeur de CDIManipulateur, ajoutez les lignes suivantes:

```
pDirectInput = nullptr;
pClavier = nullptr;
pSouris = nullptr;
pJoystick = nullptr;
```

4. Ajoutez la déclaration de la variable statique bDejaInit dans le fichier CDIManipulateur.CPP, au début, mais après les #include :

```
bool CDIManipulateur::bDejaInit = false;
```

5. Déclarez et implantez la fonction Init dans la section **public** de la classe CDIManipulateur :

```
bool CDIManipulateur::Init(HINSTANCE hInstance, HWND hWnd)
{
    // Un seul objet DirectInput est permis
    if (!bDejaInit)
    {
        // Objet DirectInput
        DXEssayer( DirectX8Create( hInstance,
                                    DIRECTINPUT_VERSION,
                                    IID_IDirectInput8,
                                    (void**)&pDirectInput,
                                    NULL), ERREUR_CREATION_DIRECTINPUT );

        // Objet Clavier
        // Objet Souris
        // Objet Joystick

        bDejaInit = true;
    }

    return true;
}
```

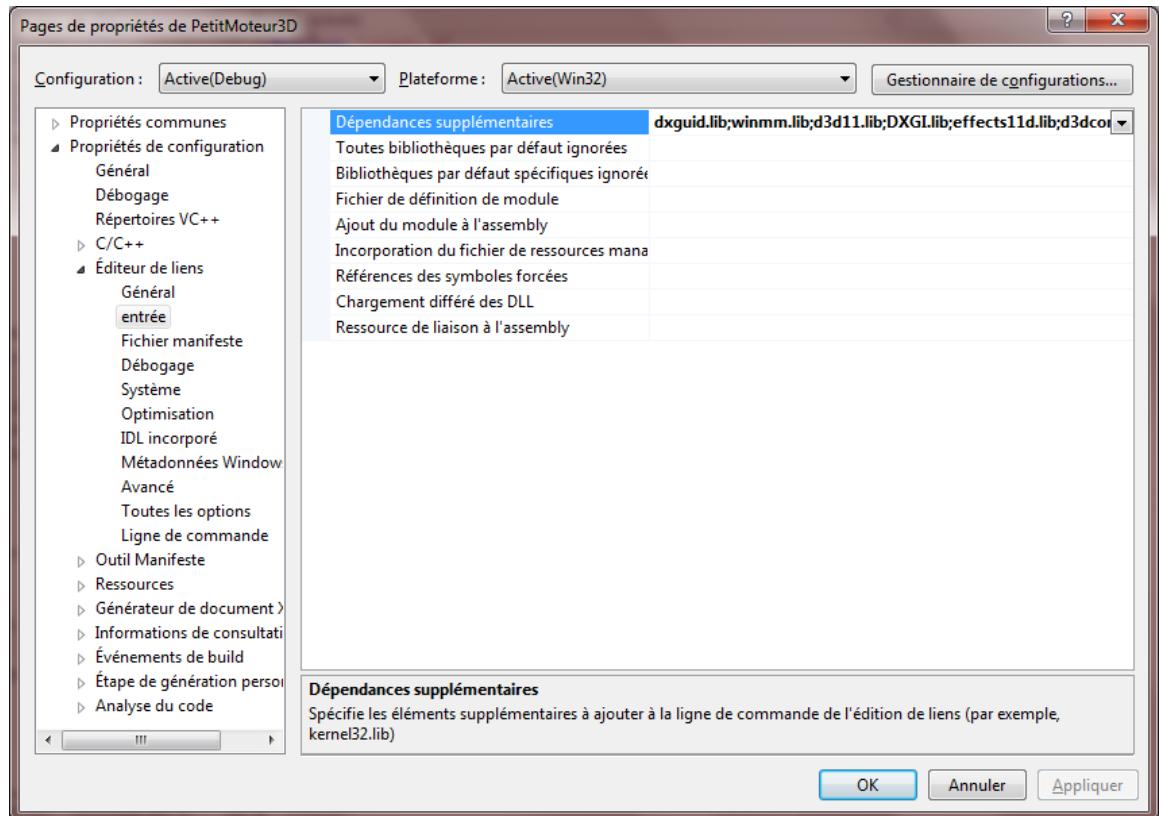
6. Ajoutez la ligne suivante au fichier stdafx.h, après les lignes de Direct3D :

```
#include <dinput.h>
```

7. Ajoutez les lignes suivantes au fichier DIManipulateur.CPP, au début, mais après les autres #include :

```
#include "util.h"
```

8. Ajoutez la bibliothèque **dinput8.lib** dans Projet|Propriétés à la rubrique Éditeur de lien/Entrée.



9. Dans le destructeur de CDIManipulateur, ajoutez les lignes suivantes :

```

if (pClavier)
{
    pClavier->Unacquire();
    pClavier->Release();
    pClavier = nullptr;
}

if (pSouris)
{
    pSouris->Unacquire();
    pSouris->Release();
    pSouris = nullptr;
}

if (pJoystick)
{
    pJoystick->Unacquire();
    pJoystick->Release();
    pJoystick = nullptr;
}

if (pDirectInput)
{
    pDirectInput->Release();
    pClavier = nullptr;
}

```

L'instruction **Unacquire** sera expliquée un peu plus loin. Le reste du code s'assure que les objets sont bien relâchés.

10. Déclarez les chaînes de caractères associées aux messages d'erreur pour les messages

ERREUR_CREATION_DIRECTINPUT,
ERREUR_CREATION_CLAVIER et
ERREUR_CREATION_FORMAT_CLAVIER.

PetitMoteur3D.rc (String Table)*			Util.h	DispositifD3D.cpp	DispositifD3D.h	Objet3D.h
ID	Valeur	Légende				
IDS_APP_TITLE	103	PetitMoteur3D				
DXE_ERREURCHOIXDISPO...	104	Il n'y a pas de dispositif correspondant à nos choix				
DXE_FICHIERTEXTUREINT...	105	Le fichier texture est introuvable				
DXE_FICHIEROBJETXINTR...	106	Le fichier X spécifié est introuvable				
IDC_PETITMOTEUR3D	109	PETITMOTEUR3D				
DXE_MAUVAISRUNTIME	110	Mauvaise version du RunTime de DirectX				
DXE_ADAPTEURNONDISP...	111	La carte graphique (adapter) n'est pas accessible				
DXE_ERREURCREATIOND...	112	Problème de création du dispositif				
ERREUR_CREATION_DIRE...	107	Erreur de création de DirectInput				
ERREUR_CREATION_CLAV...	108	Erreur de création du clavier				
ERREUR_CREATION_FORM...	113	Erreur dans le format du clavier				
IDS_STRING114	114					

11. Ajoutez la ligne suivante au début du fichier **DIManipulateur.CPP** (après les autres #include):

```
#include "resource.h"
```

Compilez votre programme pour vous assurer de ne pas avoir d'erreur de syntaxe. La prochaine section vous montrera comment compléter la classe pour le clavier.

10.4 Acquisition du clavier

Création du dispositif clavier

Ajoutez à la fonction Init de la classe CDIManipulateur les lignes suivantes:

```
// Objet Clavier
DXEssayer (pDirectInput ->CreateDevice( GUID_SysKeyboard,
                                         &pClavier,
                                         NULL),
            ERREUR_CREATION_CLAVIER);

DXEssayer(pClavier->SetDataFormat( &c_dfDIKeyboard),
          ERREUR_CREATION_FORMATCLAVIER);

pClavier->SetCooperativeLevel( hWnd,
                                 DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);
```

pClavier->Acquire();

Démarre le processus d'acquisition du clavier. Cette fonction DOIT être appelée AVANT d'utiliser les fonctions de réceptions des données.

L'accès au dispositif est limité à l'avant-plan et n'est pas exclusif. D'autres applications peuvent recevoir les signaux du clavier.

GUID_SysKeyboard qui est une valeur prédéfinie pour le clavier du système

c_dfDIKeyboard est une variable globale fournie par DirectInput. Cette variable fournit un format de données prédéterminé correspondant au clavier.

Obtenir l'information du clavier

Le principe utilisé pour l'obtention de l'information du clavier est assez simple: il suffit d'obtenir un « cliché » de l'état du clavier au moment où l'on désire vérifier le statut de certaines touches. Le cliché sera fait avec la fonction IDirectInput8::GetDeviceState.

Une fois l'état du clavier copié dans un tableau (notre « cliché »), l'application peut vérifier le statut des touches. Les touches sont identifiées par des constantes correspondant à des numéros, vous pouvez consulter ces constantes dans la documentation de DirectX sous la rubrique **Keyboard Device Constants**.

Dans notre cas, nous allons intégrer la prise du cliché et la vérification des touches dans la classe CDIManipulateur.

1. Ajoutez la variable *private* suivante à la classe CDIManipulateur:

```
char tamponClavier[256];
```

2. Déclarez et implantez la fonction **StatutClavier** à la classe **CDIManipulateur** dans la section **public**:

```
void CDIManipulateur::StatutClavier()
{
    pClavier->GetDeviceState(sizeof(tamponClavier), (void*)& tamponClavier);
}
```

Cette fonction sera appelée à chaque fois que nous voudrons obtenir notre « cliché » de l'état des touches du clavier. Elle sera appelée avant la vérification des touches.

3. Déclarez et implantez la fonction **ToucheAppuyee** à la classe **CDIManipulateur** dans la section **public**:

```
bool CDIManipulateur::ToucheAppuyee(UINT touche)
{
    return (tamponClavier[touche] & 0x80);
}
```

10.5 Travailler avec le clavier

Nous sommes maintenant prêts à utiliser la classe CDIManipulateur dans notre application. Pour les besoins de l'exercice, nous utiliserons le programme à l'état où il était au chapitre 9 c'est-à-dire avec notre personnage qui tournait:

1. Il nous faut d'abord déclarer une variable de classe CDIManipulateur dans la section **protected** de CMoteur (dans le fichier MOTEUR.H):

```
CDIManipulateur GestionnaireDeSaisie;
```

C'est pour le moment une variable spécifique puisque nous ne faisons qu'illustrer le fonctionnement de DirectInput.

Nous ne devrions avoir qu'une seule variable CDIManipulateur dans notre application, mais comme Moteur est un singleton...

2. Et ajoutez la fonction suivante dans la section **public** de CMoteur:

```
CDIManipulateur& GetGestionnaireDeSaisie() {return GestionnaireDeSaisie;}
```

3. Inclure la ligne suivante à la suite des autres #include dans le fichier Moteur.H:

```
#include "DIManipulateur.h"
```

4. Dans la fonction InitialisationsSpecific de CMoteurWindows, ajoutez les lignes suivantes après l'énoncé Show() :

```
int CMoteurWindows::InitialisationsSpecific()
{
    // Initialisations de l'application;
    InitAppInstance();
    Show();

    // Initialisation de DirectInput
    GestionnaireDeSaisie.Init(hAppInstance, hMainWnd);

    return 0;
}
```

DirectInput est maintenant prêt à être utilisé.

Dans notre cas, nous nous contenterons de faire tourner le personnage vers la gauche ou vers la droite au moyen des touches \leftarrow et \rightarrow .

5. Il nous faut d'abord modifier CMoteur::AnimeScene pour qu'elle saisisse le statut du clavier:

```
bool AnimeScene(float tempsEcoule)
{
    // Prendre en note le statut du clavier
    GestionnaireDeSaisie.StatutClavier();

    ... le reste de la fonction est inchangé
```

6. Modifiez la fonction Anime de la classe CObjetMesh pour qu'elle ressemble à ceci:

```
void CObjetMesh::Anime(float tempsEcoule)
{
    // Pour les mouvements, nous utilisons le gestionnaire de saisie
    CMoteurWindows& rMoteur = CMoteurWindows::GetInstance();
    CDIManipulateur& rGestionnaireDeSaisie =
    rMoteur.GetGestionnaireDeSaisie();

    // Vérifier l'état de la touche gauche
    if ( rGestionnaireDeSaisie.ToucheAppuyee(DIK_LEFT) )
    {
        rotation = rotation + ( (XM_PI * 2.0f) / 7.0f * tempsEcoule );

        // modifier la matrice de l'objet X
        matWorld = XMMatrixRotationY( rotation );
    }

    // Vérifier l'état de la touche droite
    if ( rGestionnaireDeSaisie.ToucheAppuyee(DIK_RIGHT) )
    {
        rotation = rotation - ( (XM_PI * 2.0f) / 7.0f * tempsEcoule );

        // modifier la matrice de l'objet X
        matWorld = XMMatrixRotationY( rotation );
    }
}
```

Le ménage des objets

Le ménage des objets DirectX a déjà été fait dans le destructeur de la classe CDIManipulateur. Notez seulement l'utilisation de la fonction **Unacquaire** pour relâcher l'acquisition du clavier.

Vous pouvez compiler et exécuter l'application.

10.6 Travailler avec la souris

Travailler avec la souris n'est pas toujours facile. En effet, nous sommes habitués à associer souris et curseur de souris alors que le curseur n'est en fait qu'un élément logique dessiné par l'application (ou par Windows) en fonction des informations reçues de la souris. DirectX nous propose deux modes pour le traitement des informations en provenance de la souris.

Le premier, le mode « tampon » (*buffered*) est un peu plus lourd à implanter et nous permet d'obtenir des événements de souris du type de ceux gérés par Windows. Dans le cadre de cette introduction, nous n'utiliserons pas ce mode.

Le second mode, le mode immédiat, fonctionne selon le même principe que le clavier et il permet d'aller, au moment opportun, lire le statut de la souris. Ce statut contient des coordonnées (relatives ou absolues) et l'état des boutons de la souris. Les coordonnées relatives sont uniquement des valeurs plus petites ou plus grandes que 0 indiquant le sens du déplacement des axes de la souris (X et Y). Les coordonnées absolues sont la somme des déplacements relatifs reçus par DirectX. Par défaut, la souris fonctionne en coordonnées relatives et ce sont elles que nous allons utiliser dans l'exercice.

Création du dispositif souris

- Ajoutez à la fonction **Init** de la classe CDIManipulateur les lignes suivantes:

```
// Objet Souris
DXEssayer(pDirectInput ->CreateDevice( GUID_SysMouse, &pSouris, NULL),
ERREUR_CREATION_SOURIS);

DXEssayer( pSouris->SetDataFormat( &c_dfDIMouse), ERREUR_CREATION_FORMAT_SOURIS);

pSouris->SetCooperativeLevel( hWnd, DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);

pSouris->Acquire();
```

- Déclarez les messages d'erreur dans le « string table » de vos ressources.

ERREUR_CREATION_FORMAT_SOURIS	114	Erreur dans le format de données de la souris
ERREUR_CREATION_SOURIS	115	Erreur de création du dispositif souris

(Les numéros de ressources n'ont pas d'importance)

Obtenir l'information de la souris

- Ajoutez la fonction suivante à la section **public** de CDIManipulateur

```
void SaisirEtatSouris();
```

- Implantez-la ainsi:

```
void CDIManipulateur::SaisirEtatSouris()
{
    pSouris->GetDeviceState(sizeof(mouseState), (void*)&mouseState);
}
```

- N'oubliez pas de déclarer la variable **mouseState** dans la section **protected**:

```
DIMOUSESTATE mouseState;
```

- Ajoutez la ligne suivante à la fonction CMoteur::AnimeScene (après le statut du clavier, mais avant la boucle qui appelle le Anime de chaque objet)

```
// Prendre en note l'état de la souris
GestionnaireDeSaisie.SaisirEtatSouris();
```

- Déclarez dans la section public de CDIManipulateur la fonction suivante:

```
const DIMOUSESTATE& EtatSouris() { return mouseState; }
```

- Ajoutez à la suite de la fonction Anime de CObjetMesh:

```
// ***** POUR LA SOURIS *****
// Vérifier si déplacement vers la gauche
if ( (rGestionnaireDeSaisie.EtatSouris().rgbButtons[0]&0x80) &&
    (rGestionnaireDeSaisie.EtatSouris().lX < 0))
{
    rotation = rotation + ( (XM_PI * 2.0f) / 4.0f * tempsEcoule );

    // modifier la matrice de l'objet X
    matWorld = XMMatrixRotationY( rotation );
}

// Vérifier si déplacement vers la droite
if ( (rGestionnaireDeSaisie.EtatSouris().rgbButtons[0]&0x80) &&
    (rGestionnaireDeSaisie.EtatSouris().lX > 0))
```

```
{  
    rotation = rotation - ( (XM_PI * 2.0f) / 4.0f * tempsEcoule );  
  
    // modifier la matrice de l'objet X  
    matWorld = XMMatrixRotationY( rotation );  
}
```

C'est tout ! Vous pouvez compiler votre programme et l'exécuter.

Notes:

Si nous utilisions la souris en mode exclusif, DirectInput supprimerait les messages de souris et Windows serait alors incapable d'afficher le curseur standard.

Il n'est pas toujours utile d'utiliser DirectInput pour les entrées de souris. Au contraire, je vous suggère d'essayer d'utiliser les messages de souris standards de Windows soient:

WM_LBUTTONDOWN,
WM_LBUTTONUP,
WM_LBUTTONDOWNDBLCLK,
WM_MOUSEMOVE,
WM_RBUTTONDOWN et autres...

Comme notre application est construite par-dessus une fenêtre, ces messages sont disponibles et donnent des coordonnées en pixels, coordonnées plus intéressantes que les informations transmises par le pilote de la souris.

11. Les ombres

Le dictionnaire définit une **ombre** comme étant « une zone sombre résultant de l'interception de la lumière ou de l'absence de lumière ». C'est un phénomène normal qui est cependant assez difficile à reproduire en 3D.

Objectifs du chapitre

- ✓ Se familiariser avec les différentes familles d'algorithmes pour l'implantation des ombres en 3D.
- ✓ Étudier plus en détail le « shadow mapping ».
- ✓ Implanter une solution de « shadow mapping ».
- ✓ Se familiariser avec l'implantation de solutions.

11. Les ombres

Un des aspects les plus importants du réalisme 3D est la variation d'éclairage appliquée aux objets selon leur position par rapport aux sources d'éclairage. Ces aspects sont souvent déjà disponibles dans les bibliothèques modernes (par exemple: *Gouraud shading*). Un complément très important aujourd'hui est le dessin d'ombres. Les ombres ajoutent beaucoup au modèle 3D. On voit très bien cet apport d'information dans cet exemple tiré de [SX2_1].

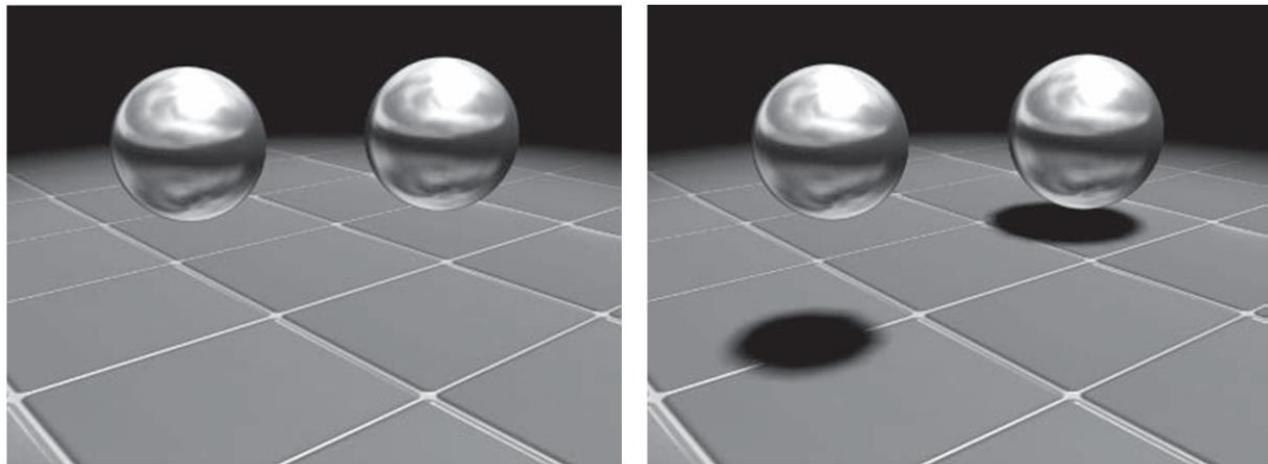


Figure 11.1 - Le rôle de l'ombre

11.1 Différentes façons de faire de l'ombre

Dans le monde réel, l'ombre est une zone dont l'illumination par une source de lumière est réduite par l'interférence d'un objet opaque. C'est une zone plus sombre causée par la position d'un objet par rapport à la lumière.

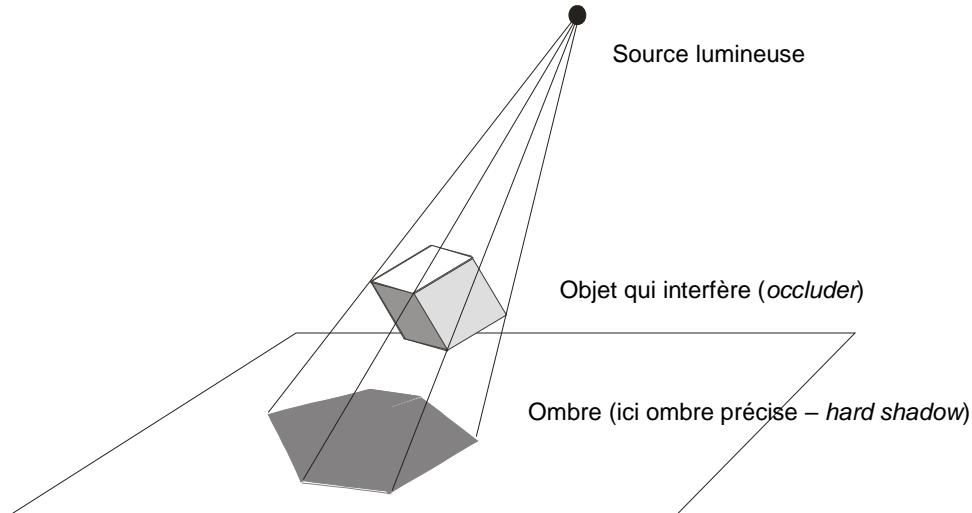


Figure 12.1 - Ombre

Ombres précises — ombres douces

Si l'ombre est produite par une source pure de type point (ou directionnelle, ou spot), on obtient une ombre dont la bordure est bien délimitée : la zone d'ombre est précise (*hard shadow*).

Si l'on considère une source moins pure, plus large ou un peu de réfraction dans l'environnement, on obtient une ombre dont la bordure est plus ou moins dégradée : la zone d'ombre est douce (*soft shadow*). La section de la bordure où l'on observe ce dégradé s'appelle la **pénombre** (penumbra) alors que la section de la zone pleinement à l'ombre s'appellera tout simplement **ombre** (umbra).

Les ombres douces sont plus réalistes que les précises, mais demanderont plus de travail (du programmeur et du système).

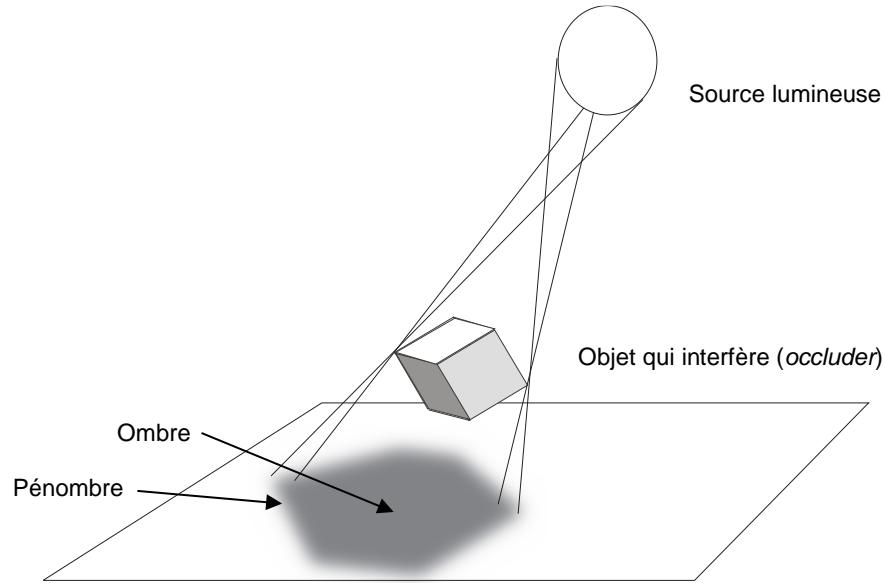
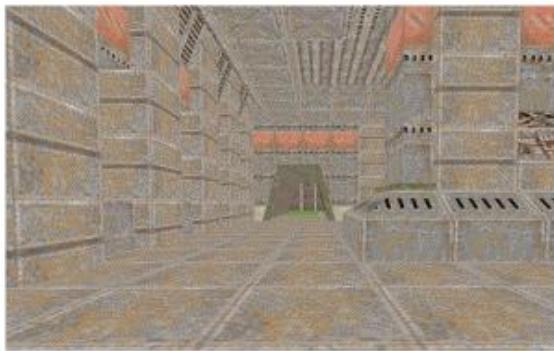


Figure 11.2 - Ombre douce

Première famille d'algorithmes – Les « cartes » d'éclairage ou d'ombrage

Les cartes d'éclairage (*lightmap*) ou cartes d'ombrage (*shadowmap*) ne sont pas à vrai dire des techniques d'ombres en temps réel, mais elles sont tellement utilisées qu'il convient d'en glisser un mot ici. Ce sont deux techniques similaires où les objets de la scène sont créés avec une texture supplémentaire préparée à l'avance par nos artistes au moyen de modules (*plugin*) d'éclairage ou d'ombrage ou dans certains cas de logiciels spécialisés (Ex : **Lightwave**, qui à l'origine se spécialisait dans l'éclairage). Les valeurs sont soit additionnées aux couleurs de la scène (*lightmap*) ou soustraites (*shadowmap*). Ces techniques, faciles à planter, permettent un grand niveau de réalisme pour le rendu de scènes fixes possédant plusieurs sources d'éclairage.



Une scène de Quake 2

La scène sans éclairage, remarquez qu'il n'y a pas non plus de ni nuancement, celui-ci sera précalculé dans l'éclairage



L'éclairage seulement, remarquez qu'il n'y a pas que des ombres, mais aussi des couleurs supplémentaires (matériaux)



La scène finale

Figure 11.3 - Éclairage avec carte d'éclairage

Deuxième famille d'algorithmes – La fausse ombre

Ce type d'ombre, où l'ombre est un objet qui suit l'objet principal, n'est plus très utilisé aujourd'hui. Ses limitations principales sont qu'elle nécessite un sol plat et que l'ombre générée reflète difficilement un objet complexe (à moins de stocker des dizaines d'images). À conserver toutefois comme truc pour des objets éloignés ou fixes si l'éclairage ne change pas et si le sol est plat.

Troisième famille d'algorithmes – L'ombre projetée

Un peu plus près des techniques modernes, cet algorithme relativement simple est rarement utilisé aujourd'hui. Il demande lui aussi un sol plat pour le rendu de l'ombre et permet difficilement les ombres douces (c'est discutable).

Quatrième famille d'algorithmes — *Shadow mapping* — ombre Z-Buffer

L'ombre à base de Z-Buffer (shadow Z-Buffer) est aussi appelée très fréquemment ***shadow mapping***. C'est une des techniques les plus utilisées dans les jeux vidéo. Le principe est relativement simple :

- Un rendu est effectué du « point de vue de la lumière ».
- L'ombre crée des « surfaces » cachées (des pixels cachés).
- Les valeurs Z permettent de conserver les valeurs qui ne sont pas à l'ombre.
- Lors du vrai rendu, chaque pixel est projeté vers le « *shadow map* » pour savoir s'il est dans l'ombre ou non.

Cette opération relativement lourde (elle demande un deuxième rendu) est quand même accélérée par le fait qu'il n'est pas nécessaire de calculer la couleur de chaque pixel, seulement sa valeur Z. De plus, les calculs sur la carte graphique (VS et PS) permettent aussi d'accélérer grandement cette technique. Notez que la plupart des techniques de volume d'ombre (*shadow volume*) ne sont pas plus rapides.

Mais cette technique a au départ deux problèmes :

1. Il faut « tricher » un peu sur la position des objets lors du calcul de l'ombre de chaque pixel, car l'ombre commence au début de l'objet qui crée l'ombre. On appelle ce problème potentiel **auto-ombrage**.
2. Les pixels générés lors du vrai rendu n'ont sûrement pas la même résolution que ceux calculés par l'éclairage. Donc soit des erreurs, soit une ombre *zigzagante*. C'est beaucoup moins vrai aujourd'hui avec les Z-Buffers à haute résolution et grand degré de profondeur ainsi qu'avec les algorithmes de filtrage.

Ce sont des algorithmes très efficaces qui peuvent produire des ombres très précises. Certains de ces algorithmes sont aussi adaptés pour tenir compte des éclairages multiples (comme les volumes d'ombre).

Cinquième famille d'algorithmes — Les volumes d'ombre

L'idée de base derrière les algorithmes de volumes d'ombre est d'utiliser un compteur qui note le nombre de « volumes d'ombre » dans lesquels notre point se retrouve. On incrémente le compteur lorsqu'il existe un polygone « face » placé devant notre point, on décrémente le compteur lorsqu'il existe un polygone « pile » placé devant notre point. Si le compteur est à zéro alors, le point n'est pas dans l'ombre (attention aux exceptions...).

L'algorithme dépend de trois éléments importants :

- Un moyen de générer la silhouette d'un objet

- Un moyen de construire et/ou dessiner les polygones du volume d'ombre
- Une technique pour le rendu de l'ombre

Le principe de l'algorithme

1. La scène est d'abord rendue sans ombrage, mais avec la vraie couleur et surtout les informations de profondeur.
2. La silhouette d'un objet peut être déterminée en trouvant les frontières entre un polygone « face » et un polygone « pile » adjacent (les listes d'adjacences peuvent être utiles ici).

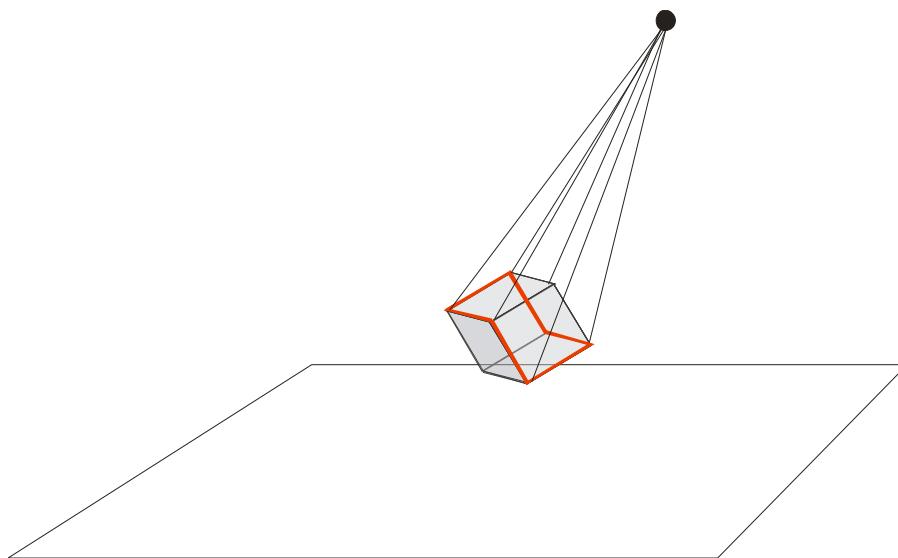


Figure 12.4 - Trouver la silhouette

3. Une fois la silhouette créée, le volume d'ombre peut être créé. De nouveaux polygones sont créés à partir des sommets de la silhouette et de leur projection au moyen d'un vecteur respectant la direction de la source lumineuse. Ces polygones sont théoriquement infinis, mais en pratique, nous les limiterons à des valeurs correspondant à notre volume de vision. L'important est que le volume ainsi créé soit « fermé » c.-à-d. qu'il n'existe pas de moyens d'entrer ou de sortir du volume sans passer par ses faces.

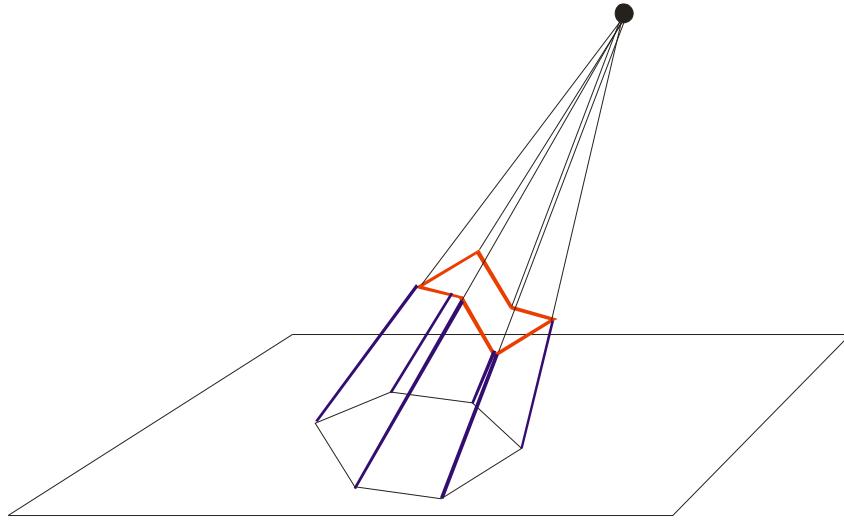


Figure 11,5 – Calculer le volume – Nouveaux polygones

4. Les volumes sont alors rendus en deux passes dans un tampon différent (par exemple un « *stencil buffer* »). Une première passe fait le rendu des polygones « faces ». La deuxième fait le rendu des polygones « piles ». Un tampon comme le *stencil buffer* ne contiendra pas des pixels, mais des valeurs. Dans notre cas, nous y additionnerons 1 chaque fois qu'un point d'un polygone face aura une profondeur z plus petite que celle de l'objet réel rendu en 1, de même nous soustrairons 1 chaque fois qu'un point d'un polygone pile aura une profondeur z plus petite que celle de l'objet réel.
5. Les pixels situés dans la zone d'ombre auront une valeur positive dans le tampon. Les bits placés dans le tampon peuvent ensuite être utilisés pour effectuer le rendu d'une couleur sombre semi-transparente par-dessus le rendu déjà effectué en 1. Le nombre dans le tampon peut aussi être utilisé pour obtenir des ombres de différents degrés.

Ce processus (2 à 4) doit être répété pour toutes les sources de lumière.

Cet algorithme possède lui aussi sa part de problèmes :

- Si la caméra est dans une zone d'ombre (ou 2...), le calcul ne se fait pas toujours bien.
- La création d'ombres douces est moins facile qu'avec la méthode précédente.
- La création des volumes d'ombre est assez lourde pour de gros objets et peut créer des problèmes avec les algorithmes de « *skinning* » (animation de personnages)

11.2 Le *shadow mapping*

Le petit exemple décrit ici se veut une adaptation HLSL et DX 11 du modèle de *shadow mapping* présenté par Michal Valient dans « **Shadow mapping with Direct3D 9** », un des chapitres de **ShaderX2 – Introductions and tutorials**. L'article original est probablement encore disponible sur Internet ([Gamedev](#) ou [Gamasutra](#)).

Présentation de l'algorithme

Cette implantation de *shadow mapping* est une amélioration de l'algorithme présenté par Lance Williams en 1978. Même si nous en avons déjà parlé précédemment, voici les éléments clés de la technique :

1. Un rendu de l'image est effectué de la position de la source lumineuse et les informations de distance de chaque pixel sont conservées dans un tampon (habituellement une texture) appelé « *shadow map* ».
2. Un rendu de l'image est effectué de la position de la caméra. Le *shadow map* est projeté sur la géométrie, ainsi pour chaque pixel dont nous faisons le rendu, nous pouvons calculer sa distance de la source de lumière et ainsi comparer avec la valeur reçue du *shadow map*. Si la valeur est plus élevée, le pixel est à l'ombre et nous pouvons restreindre son éclairage (par exemple en ne conservant que la composante ambiante).

L'avantage principal est qu'aucune nouvelle géométrie n'est nécessaire. Mais plusieurs problèmes peuvent se produire si le *shadow map* n'a pas une résolution « comparable » au rendu de l'image finale.

La technique présentée par Michal Valient n'utilise pas le z-buffer comme *shadow map*, mais plutôt une texture possédant des valeurs 32-bits. La distance est calculée dans l'espace de la scène puis ajustée linéairement dans l'intervalle [0... 1] selon l'équation:

$$Z_{\text{Calculé}} = \frac{(\text{Distance} - Z_{\text{PlanRapproché}})}{(Z_{\text{PlanÉloigné}} - Z_{\text{PlanRapproché}})}$$

L'utilisation d'une texture n'est pas plus rapide que le z-buffer, mais permet d'explorer cette technique et de choisir plus facilement le format du *shadow map*. De toute façon, le z-buffer est en quelque sorte une texture.

Le problème de biais de profondeur du *shadow map* (Depth Bias)

Le shadow map étant une texture qui sera projetée d'une façon ou d'une autre, des problèmes de minification ou de magnification peuvent se produire si sa résolution n'est pas « comparable » à celle de la scène. L'erreur principale est l'auto-ombrage. Une des façons de réduire ce problème est d'ajouter une valeur (le biais) à la distance. Cette valeur est souvent trouvée par essais et erreurs...

Michal Valient utilise une autre façon : comme ses objets sont tous des solides, le rendu 1 est effectué en utilisant seulement les polygones « piles » donc les polygones « faces » seront toujours plus près de la source lumineuse. Il restera probablement certaines « erreurs », mais ...

Filtrage des valeurs du *shadow map*

L'algorithme utilise aussi une technique d'interpolation linéaire inspirée de PCF (*percentage closer filtering*). La version de PCF utilisée permet l'élimination de certaines erreurs de minification-magnification et implémente aussi une forme simple d'ombre douce.

11.3 Programmation du *shadow map*

1. La création de la texture

La texture que nous utiliserons aura une dimension de 512 X 512, c'est une dimension appropriée pour les rendus utilisés dans les jeux aujourd'hui. Une dimension trop petite nous offrirait des ombres trop « dentelées ». Une dimension trop grande serait évidemment plus longue à remplir. Vous pourrez expérimenter avec plusieurs dimensions en faisant varier la constante SHADOWMAP_DIM.

Le projet sera démarré à partir du projet du chapitre 9.

Note: pour les besoins de l'exercice, et comme nous n'avons qu'un seul objet, nous implanterons les shaders dans la classe CObjetMesh. Cet effet pourrait être attribué à un groupe d'objets ou ...

Nous n'aurons qu'un seul effet, mais évidemment, l'ombre pourrait être réalisée au moyen de deux effets soient un pour les « objets qui font de l'ombre » et un pour « les objets qui peuvent recevoir de l'ombre » (*shadow caster* et *shadow receiver*).

1. Déclarez la constante dans la section *protected* de la classe CObjetMesh:

```
static const int SHADOWMAP_DIM = 512;
```

2. Déclarez les pointeurs suivants dans la section *private* de la classe CObjetMesh:

```
ID3D11InputLayout* pVertexLayoutShadow;  
ID3D11Texture2D* pDepthTexture; // texture de profondeur  
ID3D11DepthStencilView* pDepthStencilView;  
ID3D11ShaderResourceView* pDepthShaderResourceView;
```

Nous aurons besoin de quatre nouveaux objets. Nous avons besoin d'un layout pour nos sommets d'ombres. Ensuite, nous avons la texture que nos vues DepthStencilView et ShaderSourceView utiliseront. Finalement, nous avons nos 2 vues pour le rendu.

3. Ajoutez le code de création de ces objets à CObjetMesh::InitEffet:

```
InitDepthBuffer();
```

4. Pour simplifier le code, j'ai placé l'initialisation du tampon de profondeur dans sa propre fonction, comme nous avions fait pour le dispositif :

```
void CObjetMesh::InitDepthBuffer()
{
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    D3D11_TEXTURE2D_DESC depthTextureDesc;
    ZeroMemory(&depthTextureDesc, sizeof(depthTextureDesc));
    depthTextureDesc.Width = 512;
    depthTextureDesc.Height = 512;
    depthTextureDesc.MipLevels = 1;
    depthTextureDesc.ArraySize = 1;
    depthTextureDesc.Format = DXGI_FORMAT_R24G8_TYPELESS;
    depthTextureDesc.SampleDesc.Count = 1;
    depthTextureDesc.SampleDesc.Quality = 0;
    depthTextureDesc.Usage = D3D11_USAGE_DEFAULT;
    depthTextureDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_DEPTH_STENCIL;
    depthTextureDesc.CPUAccessFlags = 0;
    depthTextureDesc.MiscFlags = 0;

    DXEssayer(
        pD3DDevice->CreateTexture2D(&depthTextureDesc, nullptr,
&pDepthTexture),
        DXE_ERREURCREATIONTEXTURE);

    // Création de la vue du tampon de profondeur (ou de stencil)
    D3D11_DEPTH_STENCIL_VIEW_DESC descDSView;
    ZeroMemory(&descDSView, sizeof(descDSView));
    descDSView.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
    descDSView.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
    descDSView.Texture2D.MipSlice = 0;
    DXEssayer(
        pD3DDevice->CreateDepthStencilView(pDepthTexture, &descDSView,
&pDepthStencilView),
        DXE_ERREURCREATIONDEPTHSTENCILTARGET);

    // Création d'une shader resource view pour lire le tampon de profondeur
    // dans le shader.
    D3D11_SHADER_RESOURCE_VIEW_DESC sr_desc;
    sr_desc.Format = DXGI_FORMAT_R24_UNORM_X8_TYPELESS;
    sr_desc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
    sr_desc.Texture2D.MostDetailedMip = 0;
    sr_desc.Texture2D.MipLevels = 1;

    DXEssayer(
        pD3DDevice->CreateShaderResourceView(pDepthTexture, &sr_desc,
&pDepthShaderResourceView),
        DXE_ERREURCREATIONSRV); }
```

5. Que nous déclarons dans la section *private* de CObjetMesh

```
void InitDepthBuffer();
```

6. N'oubliez pas d'ajouter aussi le code de destruction de ces objets au destructeur de CObjetMesh:

```
DXRelacher(pDepthStencilView);
DXRelacher(pDepthShaderResourceView);
DXRelacher(pDepthTexture);
DXRelacher(pVertexLayoutShadow);
```

2. Autres préparatifs

- Pour le calcul de la matrice WVP qui permettra de projeter les valeurs sur la surface théorique du shadow map, nous déclarons trois matrices dans la section *protected* de la classe CObjetMesh:

```
XMMATRIX mVLight;
XMMATRIX mPLight;
XMMATRIX mVPLight;
```

- Nous initialisons ces matrices dans une nouvelle fonction de la classe CObjetMesh: **InitMatricesShadowMap**. Si la position de la lumière devait bouger, il faudrait recalculer mVLight.

```
void CObjetMesh::InitMatricesShadowMap()
{
    // Matrice de la vision vu par la lumière - Le point T0 est encore 0,0,0
    mVLight = XMMatrixLookAtLH( XMVectorSet( -5.0f, 5.0f, -5.0f, 1.0f ),
                                XMVectorSet( 0.0f, 0.0f, 0.0f, 1.0f ),
                                XMVectorSet( 0.0f, 1.0f, 0.0f, 1.0f ) );
    float champDeVision;
    float ratioDAspect;
    float planRapproche;
    float planEloigne;

    champDeVision = XM_PI/4; // 45 degrés
    ratioDAspect = 1.0f; // 512/512

    planRapproche = 2.0; // Pas besoin d'être trop près
    planEloigne = 100.0; // Suffisemment pour avoir tous les objets
    mPLight = XMMatrixPerspectiveFovLH( champDeVision,
                                         ratioDAspect,
                                         planRapproche,
                                         planEloigne );

    mVPLight = mVLight * mPLight;
}
```

- La fonction **InitMatricesShadowMap** sera appelée dans InitEffet, à la fin.
- Nous en profitons pour ajouter la matrice **matWorldViewProjLight** à la structure ShadersParams, cette matrice sera la matrice WVP telle que « vue » par la source de lumière :

```
struct ShadersParams // toujours un multiple de 16 pour les constantes
{
    XMMATRIX matWorldViewProj; // la matrice totale
    XMMATRIX matWorldViewProjLight; // WVP pour lumiere
    XMMATRIX matWorld; // matrice de transformation dans le monde
    ...
};
```

4. Dans la fonction InitEffet de CObjetMesh, remplacez le code suivant :

```
...
// Créer l'organisation des sommets pour le VS de notre effet
D3DX11_PASS_SHADER_DESC effectVSDesc;
pPasse->GetVertexShaderDesc(&effectVSDesc);

D3DX11_EFFECT_SHADER_DESC effectVSDesc2;
effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
                                              &effectVSDesc2);

const void *vsCodePtr = effectVSDesc2.pBytecode;
unsigned vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout = nullptr;

CSommetMesh::numElements = ARRSIZE(CSommetMesh::layout);
DXESSAYER( pD3DDevice->CreateInputLayout(CSommetMesh::layout,
                                             CSommetMesh::numElements,
                                             vsCodePtr,
                                             vsCodeLen,
                                             &pVertexLayout ),
DXE_CREATIONLAYOUT);
```

...

par celui-ci :

```
...
// Créer l'organisation des sommets pour les VS de notre effet
D3DX11_PASS_SHADER_DESC effectVSDesc;
D3DX11_EFFECT_SHADER_DESC effectVSDesc2;
const void *vsCodePtr;
unsigned vsCodeLen;
CSommetMesh::numElements = ARRSIZE(CSommetMesh::layout);

// 1 pour le shadowmap
pTechnique = pEffet->GetTechniqueByName("ShadowMap");
pPasse = pTechnique->GetPassByIndex(0);
pPasse->GetVertexShaderDesc(&effectVSDesc);
effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
                                              &effectVSDesc2);

vsCodePtr = effectVSDesc2.pBytecode;
vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout = nullptr;

DXESSAYER( pD3DDevice->CreateInputLayout(CSommetMesh::layout,
                                             CSommetMesh::numElements,
                                             vsCodePtr,
                                             vsCodeLen,
                                             &pVertexLayoutShadow ),
DXE_CREATIONLAYOUT);

// 2 pour miniphong
pTechnique = pEffet->GetTechniqueByName("MiniPhong");
pPasse = pTechnique->GetPassByIndex(0);
```

```
pPasse->GetVertexShaderDesc(&effectVSDesc);
effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
                                              &effectVSDesc2);

vsCodePtr = effectVSDesc2.pBytecode;
vsCodeLen = effectVSDesc2.BytecodeLength;

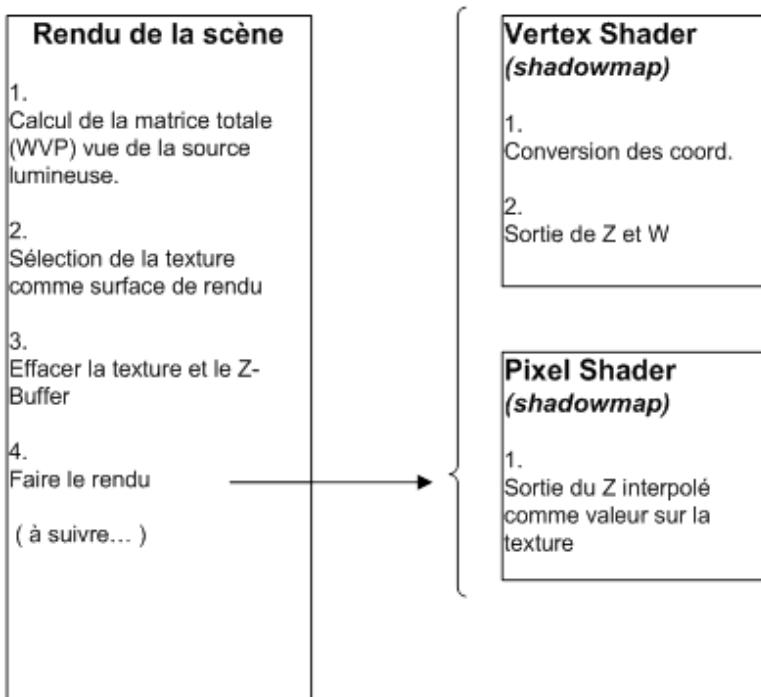
pVertexLayout = nullptr;

DXEssayer( pD3DDevice->CreateInputLayout( CSommetMesh::layout,
                                             CSommetMesh::numElements,
                                             vsCodePtr,
                                             vsCodeLen,
                                             &pVertexLayout ),
            DXE_CREATIONLAYOUT);

...
```

3. Rendu de la scène – Préparation et rendu du *shadow map*

(dans la fonction **Draw** de CObjetMesh)



Pour simplifier un peu le travail, je séparerai la fonction Draw en trois parties :

1. Code identique pour les deux rendus
2. Code pour le rendu du « shadow map »
3. Code pour l'affichage de l'objet avec ombre (voir en 5.)

Code identique pour les deux rendus

```

void CObjetMesh::Draw()
{
    // ***** OMBRES ---- Valide pour les deux rendus
    // Obtenir le contexte
    ID3D11DeviceContext* pImmediateContext = pDispositif-
>GetImmediateContext();

    // Choisir la topologie des primitives
    pImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

    // Index buffer
    pImmediateContext->IASetIndexBuffer( pIndexBuffer, DXGI_FORMAT_R32_UINT,
0);

    // Vertex buffer
    UINT stride = sizeof( CSommetMesh );
    UINT offset = 0;
    pImmediateContext->IASetVertexBuffers( 0, 1, &pVertexBuffer, &stride,
&offset );
}

```

Code pour le rendu du « shadow map »

```
...  
// ***** OMBRES ---- Premier Rendu - Cr閘ation du Shadow Map  
// Utiliser la surface de la texture comme surface de rendu  
pImmediateContext->OMSetRenderTargets(0, nullptr,  
pDepthStencilView);  
  
// Effacer le shadow map  
pImmediateContext-  
>ClearDepthStencilView(pDepthStencilView,D3D11_CLEAR_DEPTH,1.0f,0 );  
  
// Modifier les dimension du viewport  
pDispositif->SetViewPortDimension(512,512);  
  
// Choix de la technique  
pTechnique = pEffet->GetTechniqueByName(" ShadowMap ");  
pPasse = pTechnique->GetPassByIndex(0);  
  
// input layout des sommets  
pImmediateContext->IASetInputLayout( pVertexLayoutShadow );  
  
// Initialiser et s閑lectionner les « constantes » de l'effet  
ShadersParams sp;  
sp.matWorldViewProjLight = XMMatrixTranspose(matWorld * mVPLight );  
  
// Nous n'avons qu'un seul CBuffer  
ID3DX11EffectConstantBuffer* pCB =  
    pEffet->GetConstantBufferByName(" param ");  
pCB->SetConstantBuffer(pConstantBuffer);  
pImmediateContext->UpdateSubresource( pConstantBuffer, 0, nullptr, &sp,  
0, 0 );  
  
// Dessiner les subsets non-transparents  
for(int i = 0; i < NombreSubset; ++i)  
{  
    const int indexStart = SubsetIndex[i];  
    const int indexDrawAmount = SubsetIndex[i+1] - SubsetIndex[i];  
    if (indexDrawAmount)  
    {  
        // IMPORTANT pour ajuster les param.  
        pPasse->Apply(0, pImmediateContext);  
  
        pImmediateContext->DrawIndexed( indexDrawAmount, indexStart, 0 );  
    }  
}  
...
```

Une nouvelle fonction pour CDispositifD3D11

Calcul de la matrice WVP pour le shadow map

Les instructions de dessin ont été « épurées » puisqu'il n'y a pas de texture

4. Les shaders pour la création du *shadow map*

Les deux shaders pour le *shadow map* ont pour but de construire une texture qui contiendra les valeurs Z des objets tels que vus par la source d'éclairage. Nous ne traiterons que les surfaces « piles » pour éviter l'auto-ombrage. Nous placerons nos shaders dans le fichier **MiniPhongSM.fx**.

Les constantes et la structure de sortie du vertex shader:

```
cbuffer param
{
    float4x4 matWorldViewProj;    // la matrice totale
    float4x4 matWorldViewProjLight; // Matrice WVP pour lumière
    float4x4 matWorld;           // matrice de transformation dans le monde
    float4 vLumiere;             // la position de la source d'éclairage (Point)
    float4 vCamera;              // la position de la caméra
    float4 vAEcl;                // la valeur ambiante de l'éclairage
    float4 vAMat;                // la valeur ambiante du matériau
    float4 vDEcl;                // la valeur diffuse de l'éclairage
    float4 vDMat;                // la valeur diffuse du matériau
    float4 vSEcl;                // la valeur spéculaire de l'éclairage
    float4 vSMat;                // la valeur spéculaire du matériau
    float puissance;
    int bTex;                   // Booléen pour la présence de texture
    float2 remplissage;
}

struct ShadowMapVS_SORTIE
{
    float4 Pos : SV_POSITION;
    float3 Profondeur: TEXCOORD0;
};
```

Le vertex shader:

```
//------------------------------------------------------------------------------
// Vertex Shader pour construire le shadow map
//------------------------------------------------------------------------------

ShadowMapVS_SORTIE ShadowMapVS( float4 Pos : POSITION )
{
    ShadowMapVS_SORTIE Out = (ShadowMapVS_SORTIE)0;

    // Calcul des coordonnées
    Out.Pos = mul( Pos, matWorldViewProjLight); // WVP de la lumiere

    // Obtenir la profondeur et normaliser avec w
    Out.Profondeur.x = Out.Pos.z / Out.Pos.w ;

    return Out;
}
```

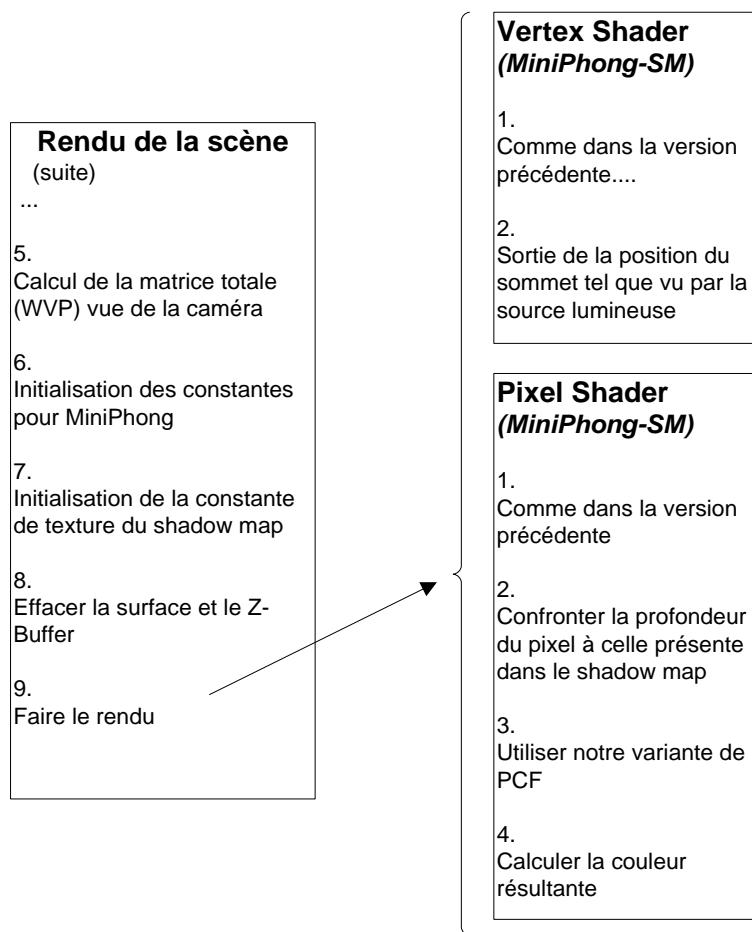
La technique:

```
RasterizerState rsCullFront
{
    CullMode = Front;
};

//-----
// Technique pour le shadow map
//-----
technique11 ShadowMap
{
    pass p0
    {
        VertexShader = compile vs_5_0 ShadowMapVS();
        SetRasterizerState( rsCullFront );           ← Les surfaces
                                                « piles »
        SetPixelShader(NULL);
        SetGeometryShader(NULL);
    }
}
```

INTÉRESSANT : Nous pouvons déclarer des états dans nos effets.

5. Rendu de la scène – Préparation et rendu avec MiniPhong



Code pour l'affichage de l'objet avec ombre

```

// ***** OMBRES ---- Deuxième Rendu - Affichage de l'objet avec ombres
// Ramener la surface de rendu
ID3D11RenderTargetView* tabRTV[1];
tabRTV[0] = pDispositif->GetRenderTargetView();
pImmediateContext->OMSetRenderTargets( 1,
                                         tabRTV,
                                         pDispositif->GetDepthStencilView());

// Dimension du viewport - défaut
pDispositif->ResetViewPortDimension();

// Choix de la technique
pTechnique = pEffet->GetTechniqueByName(" MiniPhong ");
pPasse = pTechnique->GetPassByIndex(0);

// Initialiser et sélectionner les « constantes » de l'effet
XMMATRIX viewProj = CMoteurWindows::GetInstance().GetMatViewProj();

sp.matWorldViewProj = XMMatrixTranspose(matWorld * viewProj );
sp.matWorld = XMMatrixTranspose(matWorld);

sp.vLumiere = XMVectorSet( -10.0f, 10.0f, -15.0f, 1.0f );
sp.vCamera = XMVectorSet( 0.0f, 3.0f, -5.0f, 1.0f );

```

Une nouvelle fonction pour
CDispositifD3D11

```

        sp.vAEcl = XMVectorSet( 0.2f, 0.2f, 0.2f, 1.0f ) ;
        sp.vDEcl = XMVectorSet( 1.0f, 1.0f, 1.0f, 1.0f );
        sp.vSEcl = XMVectorSet( 0.6f, 0.6f, 0.6f, 1.0f );

        // Le sampler state
        ID3DX11EffectSamplerVariable* variableSampler;
        variableSampler = pEffet->GetVariableByName("SampleState")->AsSampler();
        variableSampler->SetSampler(0, pSampleState);

        // input layout des sommets
        pImmediateContext->IASetInputLayout( pVertexLayout );

        ID3DX11EffectShaderResourceVariable* pShadowMap;
        pShadowMap = pEffet->GetVariableByName("ShadowTexture")-
>AsShaderResource();
        pShadowMap->SetResource(pDepthShaderResourceView);

        pDispositif->SetNormalRSState();   Une nouvelle fonction pour CDispositifD3D11

        // Dessiner les subsets non-transparents
        for(int i = 0; i < NombreSubset; ++i)
        {
            int indexStart = SubsetIndex[i];
            int indexDrawAmount = SubsetIndex[i+1] - SubsetIndex[i];
            if (indexDrawAmount)
            {
                sp.vAMat = XMLoadFloat4(&Material[SubsetMaterialIndex[i]].Ambient);
                sp.vDMat = XMLoadFloat4(&Material[SubsetMaterialIndex[i]].Diffuse);
                sp.vSMat = XMLoadFloat4(&Material[SubsetMaterialIndex[i]].Specular);
                sp.puissance = Material[SubsetMaterialIndex[i]].Puissance;

                // Activation de la texture ou non
                if (Material[SubsetMaterialIndex[i]].pTextureD3D != nullptr)
                {
                    ID3DX11EffectShaderResourceVariable* variableTexture;
                    variableTexture =
                        pEffet->GetVariableByName(`textureEntree`)->AsShaderResource();
                    variableTexture-
>SetResource(Material[SubsetMaterialIndex[i]].pTextureD3D);
                    sp.bTex = 1;
                }
                else
                {
                    sp.bTex = 0;
                }

                // IMPORTANT pour ajuster les param.
                pPasse->Apply(0, pImmediateContext);

                pCB->SetConstantBuffer(pConstantBuffer);
                pImmediateContext->UpdateSubresource( pConstantBuffer, 0, nullptr,
&sp, 0, 0 );

                pImmediateContext->DrawIndexed( indexDrawAmount, indexStart, 0 );
            }
        }
    }
}

```

Notez bien !

La principale différence avec les chapitres précédents est l'utilisation de la texture créée par la technique de *shadow mapping*

Nous en profiterons aussi pour ajouter à la classe CDispositifD3D11 trois fonctions publiques soient:

```
void SetViewPortDimension(float largeur_in, float hauteur_in);
void ResetViewPortDimension();
void SetNormalRSState();
```

La première fonction, **SetViewPortDimension**, permet de modifier les paramètres d'utilisation de la cible de rendu. C'est essentiel de l'utiliser, car nous effectuons un rendu sur une surface de 512 X 512.

La première fonction, **ResetViewPortDimension**, permet de restaurer les paramètres d'utilisation de la cible de rendu de défaut.

La troisième fonction permet de restaurer (ou d'instaurer...) un état pour le « Rasterizer Stage ». C'est très utile, car il est fréquent que les effets modifient l'état de cette étape du pipeline. Plus tard, il sera utile d'avoir une fonction similaire pour d'autres étapes du pipeline. Un « super » gestionnaire d'états permettrait aussi de régler ce problème.

Voici donc la définition de ces trois fonctions :

```
void CDispositifD3D11::SetViewPortDimension(float largeur_in, float
hauteur_in)
{
    D3D11_VIEWPORT vp;
    vp.Width = largeur_in;
    vp.Height = hauteur_in;
    vp.MinDepth = 0.0f;
    vp.MaxDepth = 1.0f;
    vp.TopLeftX = 0;
    vp.TopLeftY = 0;
    pImmediateContext->RSSetViewports( 1, &vp );
}

void CDispositifD3D11::ResetViewPortDimension()
{
    D3D11_VIEWPORT vp;
    vp.Width = largeurEcran;
    vp.Height = hauteurEcran;
    vp.MinDepth = 0.0f;
    vp.MaxDepth = 1.0f;
    vp.TopLeftX = 0;
    vp.TopLeftY = 0;
    pImmediateContext->RSSetViewports( 1, &vp );
}

void CDispositifD3D11::SetNormalRSState()
{
    pImmediateContext->RSSetState(mSolidCullBackRS);
}
```

6. Les shaders MiniPhong avec *shadow mapping*

Ces deux shaders sont une adaptation de ceux des chapitres précédents. Un ajout est fait pour déterminer par interpolation PCF une valeur de 0,0 à 1,0 à partir des informations que nous avions placées dans la texture *shadow map*. Cette valeur sera multipliée par les composantes diffuses et spéculaires de façon à « faire de l'ombre ».

Les deux textures et leurs « *sampler*s » :

```
Texture2D textureEntree; // la texture
SamplerState SampleState; // l'état de sampling

Texture2D ShadowTexture; // La texture du shadow map
SamplerState ShadowMapSampler
{
    Filter = MIN_MAG_MIP_POINT;
    AddressU = Clamp;
    AddressV = Clamp;
};
```

Nous pouvons aussi déclarer des états d'échantillonnage dans nos effets.

La structure de sortie du VS:

```
struct VS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : TEXCOORD0;
    float3 vDirLum : TEXCOORD1;
    float3 vDirCam : TEXCOORD2;
    float2 coordTex : TEXCOORD3;
    float4 PosInMap : TEXCOORD4;
};
```

Le Vertex Shader:

```
//-----
// Vertex Shader miniphong avec shadow map
//-----
VS_Sortie MiniPhongVS(float4 Pos : POSITION,
                      float3 Normale : NORMAL, float2 coordTex: TEXCOORD0)
{
    VS_Sortie sortie = (VS_Sortie)0;

    sortie.Pos = mul(Pos, matWorldViewProj);
    sortie.Norm = mul(float4(Normale, 0.0f), matWorld).xyz;

    float3 PosWorld = mul(Pos, matWorld).xyz;

    sortie.vDirLum = vLumiere.xyz - PosWorld;
    sortie.vDirCam = vCamera.xyz - PosWorld;

    // Coordonnées d'application de texture
```

```

sortie.coordTex = coordTex;

// Valeurs pour shadow map
// Coordonnées
sortie.PosInMap = mul(Pos, matWorldViewProjLight);

return sortie;
}

```

La position est convertie en fonction de la source de lumière ... et transmise au pixel shader

Le Pixel Shader:

```

#define DIMTEX 512

float4 MiniPhongPS( VS_Sortie vs ) : SV_Target
{
float4 couleur;

// Normaliser les paramètres
float3 N = normalize(vs.Norm);
float3 L = normalize(vs.vDirLum);
float3 V = normalize(vs.vDirCam);

// Valeur de la composante diffuse
Float3 diff = saturate(dot(N, L));

// R = 2 * (N.L) * N - L
float3 R = normalize(2 * diff * N - L);

// Calcul de la spécularité
float4 S = pow(saturate(dot(R, V)), puissance);

float3 couleurTexture;

if (bTex > 0)
{
    // Échantillonner la couleur du pixel à partir de la texture
    couleurTexture = textureEntree.Sample(SampleState, vs.coordTex).rgb;
}
else
{
    couleurTexture = vDMat.rgb;
}

// Calculer la valeur du shadowmapping
// Calculer les coordonnées de texture (ShadowMap)
float2 ShadowCoord = 0.5 * vs.PosInMap.xy / vs.PosInMap.w + float2( 0.5,
0.5 );
ShadowCoord.y = 1.0f - ShadowCoord.y;

float2 PixelActuel = DIMTEX * ShadowCoord; // Pour une texture de 512 X
512

// Valeur de l'interpolation linéaire
float2 lerps = frac( PixelActuel );

// Lire les valeurs du tableau, avec les vérifications de profondeur
float Profondeur = vs.PosInMap.z / vs.PosInMap.w ;
float3 kernel[9];

float echelle = 1.0/DIMTEX;

```

Les coordonnées de texture sont calculées en fonction de 0.0 à 1.0

y doit être inversé !

Le fractionnement permet de bien interpoler les 3 valeurs (voir article)

```

float2 coord[9];
coord[0] = ShadowCoord + float2(-echelle,-echelle);
coord[1] = ShadowCoord + float2(-echelle, 0.0    );
coord[2] = ShadowCoord + float2(-echelle, echelle);
coord[3] = ShadowCoord + float2( 0.0    ,-echelle);
coord[4] = ShadowCoord + float2( 0.0    , 0.0    );
coord[5] = ShadowCoord + float2( 0.0    , echelle);
coord[6] = ShadowCoord + float2( echelle,-echelle);
coord[7] = ShadowCoord + float2( echelle, 0.0    );
coord[8] = ShadowCoord + float2( echelle, echelle);

// Colonne 1
kernel[0].x=(ShadowTexture.Sample(ShadowMapSampler, coord[0])<
Profondeur)? 0.0f: 1.0f;
kernel[0].y=(ShadowTexture.Sample(ShadowMapSampler, coord[1])<
Profondeur)? 0.0f: 1.0f;
kernel[0].z=(ShadowTexture.Sample(ShadowMapSampler, coord[2])<
Profondeur)? 0.0f: 1.0f;
// Colonne 2
kernel[1].x=(ShadowTexture.Sample(ShadowMapSampler, coord[3])<
Profondeur)? 0.0f: 1.0f;
kernel[1].y=(ShadowTexture.Sample(ShadowMapSampler, coord[4])<
Profondeur)? 0.0f: 1.0f;
kernel[1].z=(ShadowTexture.Sample(ShadowMapSampler, coord[5])<
Profondeur)? 0.0f: 1.0f;
// Colonne 3
kernel[2].x=(ShadowTexture.Sample(ShadowMapSampler, coord[6])<
Profondeur)? 0.0f: 1.0f;
kernel[2].y=(ShadowTexture.Sample(ShadowMapSampler, coord[7])<
Profondeur)? 0.0f: 1.0f;
kernel[2].z=(ShadowTexture.Sample(ShadowMapSampler, coord[8])<
Profondeur)? 0.0f: 1.0f;

// Les interpolations linéaires
// Interpoler colonnes 1 et 2
float3 col12 = lerp( kernel[0], kernel[1], lerps.x );
// Interpoler colonnes 2 et 3
float3 col23 = lerp( kernel[1], kernel[2], lerps.x );
// Interpoler ligne 1 et colonne 1
float4 lc;
    lc.x = lerp( col12.x, col12.y, lerps.y );
// Interpoler ligne 2 et colonne 1
    lc.y = lerp( col12.y, col12.z, lerps.y );
// Interpoler ligne 1 et colonne 2
    lc.z = lerp( col23.x, col23.y, lerps.y );
// Interpoler ligne 2 et colonne 1
    lc.w = lerp( col23.y, col23.z, lerps.y );

// Faire la moyenne
float ValeurOmbre = (lc.x + lc.y + lc.z + lc.w) / 4.0;

// I = A + D * N.L + (R.V)n
couleur = (couleurTexture * vAEcl.rgb +
            couleurTexture * vDEcl.rgb * diff.rgb +
            vSEcl.rgb * vSMat.rgb * S) * ValeurOmbre;

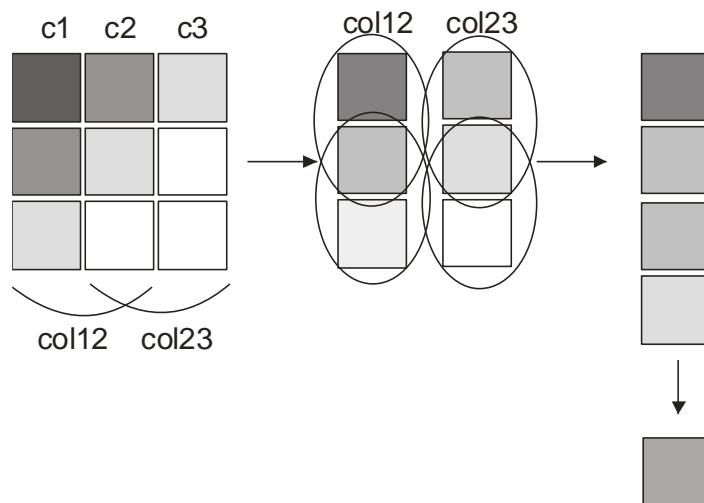
return float4(couleur, 1.0f); }

```

L'effet

Identique aux chapitres précédents

Voici un petit graphique des interpolations linéaires effectuées. Une zone 3 X 3 entourant notre position de pixel est utilisée comme base pour nos interpolations linéaires. Elle est d'abord réduite en deux colonnes, puis en quatre valeurs dont nous prendrons la moyenne.



Pour plus de détails, vous pouvez vous référer à l'article de Michal Valient [SX2_1].

12. Les « post-effects »

Les « post-effects », appelés aussi « post-processing effects » sont des effets visuels appliqués à l'image résultante de notre scène (avant les éléments d'interface !). Ces effets sont à l'origine des effets de traitement d'image (comme avec Photoshop), mais nous pouvons aller plus loin en conservant certaines informations, par exemple les valeurs z des pixels. Des milliers d'effets utilisés dans les jeux vidéo sont à base de post-processing.

Objectifs du chapitre

- ✓ Se perfectionner avec les surfaces de rendu.
- ✓ Développer de petits shaders destinés au traitement d'image

12. Les « post-effects »

NOTE : Beaucoup des trucs utilisés ici ont été entrevus au chapitre 11 – Les ombres.

Théorie

12.1 Le principe de base

1 — Afficher la scène sur une texture

Pour afficher la scène sur une texture, il faut d'abord créer celle-ci de dimension et de format compatibles avec l'affichage normal de la scène. Cette texture devra, de plus, avoir la possibilité de servir de **render target**. (Comme c'était le cas pour la carte d'ombre – *shadow map* – au chapitre précédent)

Puis, il faudra modifier le code d'affichage de la scène pour utiliser la texture comme surface de rendu **avant** l'affichage de la scène et revenir à la surface de rendu standard après l'affichage.

2 — Utiliser un panneau de post-effect

Un panneau de post-effect est tout simplement un « **quad** » (deux triangles) ajusté aux dimensions de notre fenêtre. C'est un objet qui s'affichera après la scène, mais pas sur la texture.

En pratique, le panneau mettra en action des shaders qui utiliseront la texture créée en 1 (et peut-être d'autres informations) pour réaliser des effets d'affichage.

3 — L'effet NUL

Cet effet est notre point de départ, c'est un effet (une technique avec les FX) qui se contente d'afficher la texture telle quelle à l'écran. Deux intérêts à cet effet : c'est un bon point de départ pour l'écriture d'autres effets et c'est un état « normal » que nous pouvons utiliser plutôt que de désactiver le code de traitement des post-effects. Note de performance : l'utilisation de l'effet « Nul » n'est pas négligeable, mais est généralement plus rapide que les autres post-effects, il peut donc être rentable de « désactiver » les post-effects (par exemple en utilisant plusieurs fonctions Draw...).

4 — Écrire d'autres effets

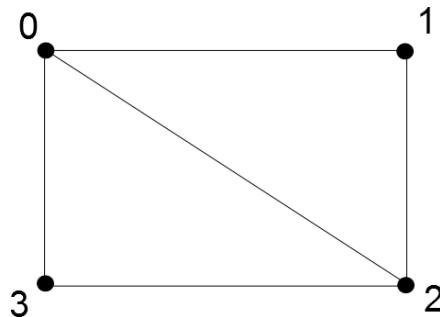
Pour les besoins de l'exercice, nous nous contenterons d'un effet plutôt simple à implanter, une version simple de « *radial-blur* ».

Atelier

12.2 La classe CPanneauPE

Le projet est construit par-dessus le projet du chapitre précédent.

Nous implanterons les shaders et l'organisation de nos effets dans la classe CPanneauPE. Beaucoup des éléments de cette classe sont semblables à ce que nous avions fait pour l'afficheur de sprites au chapitre 9. **Attention**, pour les besoins d'un panneau de post-effect, le sommet 0 sera en haut à gauche de l'écran :



PanneauPE.h

```

#pragma once
#include "objet3d.h"

namespace PM3D
{
class CDispositifD3D11;

class CSommetPanneauPE
{
public:
    CSommetPanneauPE() = default;
    CSommetPanneauPE(const XMFLOAT3& position, const XMFLOAT2& coordTex)
        : m_position(position)
        , m_coordTex(coordTex)
    { }

public:
    static UINT numElements;
    static D3D11_INPUT_ELEMENT_DESC layout[];

    XMFLOAT3 m_position;
    XMFLOAT2 m_coordTex;
};

// Classe : CPanneauPE
// BUT : Classe pour les post-effects
class CPanneauPE : public CObjet3D
{
public:
    explicit CPanneauPE(CDispositifD3D11* pDispositif_in);
    virtual ~CPanneauPE();

    virtual void Draw() override;
}
  
```

```

    void DebutPostEffect();
    void FinPostEffect();

private:
    void InitEffet();

    static CSommetPanneauPE sommets[6];
    ID3D11Buffer* pVertexBuffer;
    CDispositifD3D11* pDispositif;

    // Pour les effets
    ID3DX11Effect* pEffet;
    ID3DX11EffectTechnique* pTechnique;
    ID3DX11EffectPass* pPasse;
    ID3D11InputLayout* pVertexLayout;
    ID3D11SamplerState* pSampleState;
};

} // namespace PM3D

```

PanneauPE.cpp

```

#include "StdAfx.h"
#include "PanneauPE.h"
#include "util.h"
#include "resource.h"
using namespace UtilitairesDX;

namespace PM3D
{
// Definir l'organisation de notre sommet
D3D11_INPUT_ELEMENT_DESC CSommetPanneauPE::layout[] =
{
    {"POSITION", 0,DXGI_FORMAT_R32G32B32_FLOAT, 0,
     0,D3D11_INPUT_PER_VERTEX_DATA, 0 },
    {"TEXCOORD", 0,DXGI_FORMAT_R32G32_FLOAT,
     0,12,D3D11_INPUT_PER_VERTEX_DATA, 0}
};
UINT CSommetPanneauPE::numElements = ARRAYSIZE(layout);

CSommetPanneauPE CPanneauPE::sommets[6] = {
    CSommetPanneauPE(XMFLOAT3(-1.0f, -1.0f, 0.0f), XMFLOAT2(0.0f, 1.0f)),
    CSommetPanneauPE(XMFLOAT3(-1.0f, 1.0f, 0.0f), XMFLOAT2(0.0f, 0.0f)),
    CSommetPanneauPE(XMFLOAT3( 1.0f, 1.0f, 0.0f), XMFLOAT2(1.0f, 0.0f)),
    CSommetPanneauPE(XMFLOAT3(-1.0f, -1.0f, 0.0f), XMFLOAT2(0.0f, 1.0f)),
    CSommetPanneauPE(XMFLOAT3( 1.0f, 1.0f, 0.0f), XMFLOAT2(1.0f, 0.0f)),
    CSommetPanneauPE(XMFLOAT3( 1.0f, -1.0f, 0.0f), XMFLOAT2(1.0f, 1.0f))
};

CPanneauPE::~CPanneauPE()
{
    DXRelacher(pSampleState);

    DXRelacher(pEffet);
    DXRelacher(pVertexLayout);
    DXRelacher(pVertexBuffer);
}

```

```

// FONCTION : CPanneauPE, constructeur paramétré
// BUT : Constructeur d'une classe de PanneauPE
// PARAMÈTRE:
// pDispositif: pointeur sur notre objet dispositif
CPanneauPE::CPanneauPE(CDispositifD3D11* pDispositif_in)
    : pDispositif(pDispositif_in)
    , pVertexBuffer(nullptr)
    , pEffet(nullptr)
    , pTechnique(nullptr)
    , pPasse(nullptr)
    , pVertexLayout{nullptr, nullptr}
    , pSampleState(nullptr)
{
    // Création du vertex buffer et copie des sommets
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    D3D11_BUFFER_DESC bd;
    ZeroMemory(&bd, sizeof(bd));

    bd.Usage = D3D11_USAGE_IMMUTABLE;
    bd.ByteWidth = sizeof(sommets);
    bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    bd.CPUAccessFlags = 0;

    D3D11_SUBRESOURCE_DATA InitData;
    ZeroMemory(&InitData, sizeof(InitData));
    InitData.pSysMem = sommets;

    DXEssayer(pD3DDevice->CreateBuffer(&bd, &InitData, &pVertexBuffer),
              DXE_CREATIONVERTEXBUFFER);

    // Initialisation de l'effet
    InitEffet();
}

void CPanneauPE::InitEffet()
{
    // Compilation et chargement du vertex shader
    ID3D11Device* pD3DDevice = pDispositif->GetD3DDevice();

    // Pour l'effet
    ID3DBlob* pFXBlob = nullptr;

    DXEssayer( D3DCompileFromFile( L"PostEffect.fx", 0, 0, 0,
                                    "fx_5_0", 0, 0,
                                    &pFXBlob, 0),
               DXE_ERREURCREATION_FX);

    D3DX11CreateEffectFromMemory(
        pFXBlob->GetBufferPointer(), pFXBlob->GetBufferSize(), 0,
        pD3DDevice, &pEffet);

    pFXBlob->Release();

    pTechnique = pEffet->GetTechniqueByIndex(0);
    pPasse = pTechnique->GetPassByIndex(0);

    // Créer l'organisation des sommets pour le VS de notre effet
    D3DX11_PASS_SHADER_DESC effectVSDesc;
    D3DX11_EFFECT_SHADER_DESC effectVSDesc2;
}

```

```

pPasse->GetVertexShaderDesc(&effectVSDesc);

effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
&effectVSDesc2);

const void *vsCodePtr = effectVSDesc2.pBytecode;
unsigned vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout = nullptr;
DXESSAYER( pD3DDevice->CreateInputLayout( CSommetPanneauPE::layout,
CSommetPanneauPE::numElements,
vsCodePtr,
vsCodeLen,
&pVertexLayout ),
DXE_CREATIONLAYOUT);

// Initialisation des paramètres de sampling de la texture
// Pas nécessaire d'être compliqué puisque l'affichage sera
// en 1 pour 1 et à plat
D3D11_SAMPLER_DESC samplerDesc;

samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_POINT;
samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MaxAnisotropy = 0;
samplerDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
samplerDesc.BorderColor[0] = 0;
samplerDesc.BorderColor[1] = 0;
samplerDesc.BorderColor[2] = 0;
samplerDesc.BorderColor[3] = 0;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;

// Création de l'état de sampling
pD3DDevice->CreateSamplerState(&samplerDesc, &pSampleState);
}

}

```

Notez que pour l'instant, il n'y a qu'une seule technique. Nous devrons modifier le code lorsqu'il y en aura plusieurs.

Déclarer et créer le panneau

Le panneau devra être accessible dans le moteur.

1. Déclarez un pointeur sur le panneau dans la classe CMoteur (n'oubliez pas l'énoncé `#include "PanneauPE.h"`) :

```
std::unique_ptr<CPanneauPE> pPanneauPE;
```

2. Créez le panneau dans la fonction InitObjets de CMoteur (à la fin...) :

```
pPanneauPE = std::make_unique<CPanneauPE>(pDispositif);
```

Il ne faut pas l'ajouter à la scène.

Note : Dans mon cas, je n'ai pas d'interface utilisateur, seulement une scène. Je m'assure seulement que le panneau **ne soit pas** dans la scène.

12.3 La texture pour l'affichage de la scène

Pour simplifier le tout, la texture sera déclarée dans la classe CPanneauPE.

1. Déclarez les pointeurs suivants dans la section *private* de la classe CPanneauPE :

```
// Texture de rendu pour effets
ID3D11Texture2D* pTextureScene;
ID3D11RenderTargetView* pRenderTargetView;
ID3D11ShaderResourceView* pResourceView;
ID3D11Texture2D* pDepthTexture;
ID3D11DepthStencilView* pDepthStencilView ;
```

Rappels et spécifications :

pTextureScene est un pointeur sur la texture utilisée pour le rendu, il s'agit en fait d'un objet **ID3D11Texture2D**, une sorte de « référence » puisque la **vraie** texture est en mémoire vidéo.

pRenderTargetView est un pointeur sur une **vue** de notre texture qui sera utilisée pour le rendu de la scène sur la texture. Il s'agit d'un objet **ID3D11RenderTargetView**, ici aussi une sorte de « référence » puisque le **vrai objet** est en mémoire vidéo.

pResourceView est un pointeur sur une **vue** de notre texture qui sera utilisée par la suite par les shaders de post-effect pour réaliser le vrai rendu. Il s'agit d'un objet **ID3D11ShaderResourceView**, le **vrai objet** est en mémoire vidéo.

pDepthTexture est un pointeur sur la texture utilisée pour le tampon de profondeur utilisé pour le rendu sur la texture, il s'agit ici aussi d'un objet **ID3D11Texture2D**, la **vraie** texture est en mémoire vidéo.

pDepthStencilView est un pointeur sur une **vue** de la texture précédente qui sera utilisée pour gérer le tampon de profondeur. Il s'agit d'un objet **ID3D11DepthStencilView**, ici aussi une sorte de « référence » puisque le **vrai objet** est en mémoire vidéo.

2. Ajoutez le code de création de la texture de rendu et des autres objets à la suite de InitEffet.

```

// **** POUR LE POST EFFECT ****
D3D11_TEXTURE2D_DESC textureDesc;
D3D11_RENDER_TARGET_VIEW_DESC renderTargetViewDesc;
D3D11_SHADER_RESOURCE_VIEW_DESC shaderResourceViewDesc;

// Description de la texture
ZeroMemory(&textureDesc, sizeof(textureDesc));

// Cette texture sera utilisée comme cible de rendu et
// comme ressource de shader
textureDesc.Width = pDispositif->GetLargeur();
textureDesc.Height = pDispositif->GetHauteur();
textureDesc.MipLevels = 1;
textureDesc.ArraySize = 1;
textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
textureDesc.SampleDesc.Count = 1;
textureDesc.Usage = D3D11_USAGE_DEFAULT;
textureDesc.BindFlags =
D3D11_BIND_RENDER_TARGET|D3D11_BIND_SHADER_RESOURCE;
textureDesc.CPUAccessFlags = 0;
textureDesc.MiscFlags = 0;

// Création de la texture
pD3DDevice->CreateTexture2D(&textureDesc, nullptr, & pTextureScene);

// VUE - Cible de rendu
renderTargetViewDesc.Format = textureDesc.Format;
renderTargetViewDesc.ViewDimension = D3D11_RTV_DIMENSION_TEXTURE2D;
renderTargetViewDesc.Texture2D.MipSlice = 0;

// Création de la vue.
pD3DDevice->CreateRenderTargetView(pTextureScene,
                                    &renderTargetViewDesc,
                                    &pRenderTargetView);

// VUE - Ressource de shader
shaderResourceViewDesc.Format = textureDesc.Format;
shaderResourceViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
shaderResourceViewDesc.Texture2D.MostDetailedMip = 0;
shaderResourceViewDesc.Texture2D.MipLevels = 1;

// Création de la vue.
pD3DDevice->CreateShaderResourceView( pTextureScene,
                                         &shaderResourceViewDesc,
                                         &pResourceView);

// Au tour du tampon de profondeur
D3D11_TEXTURE2D_DESC depthTextureDesc;
ZeroMemory( &depthTextureDesc, sizeof( depthTextureDesc ) );
depthTextureDesc.Width = pDispositif->GetLargeur();
depthTextureDesc.Height = pDispositif->GetHauteur();
depthTextureDesc.MipLevels = 1;
depthTextureDesc.ArraySize = 1;
depthTextureDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthTextureDesc.SampleDesc.Count = 1;
depthTextureDesc.SampleDesc.Quality = 0;
depthTextureDesc.Usage = D3D11_USAGE_DEFAULT;
depthTextureDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;

```

```

depthTextureDesc.CPUAccessFlags = 0;
depthTextureDesc.MiscFlags = 0;

DXEssayer( pD3DDevice->CreateTexture2D( &depthTextureDesc, NULL,
&pDepthTexture ),
DXE_ERREURCREATIONTEXTURE );

// Cration de la vue du tampon de profondeur (ou de stencil)
D3D11_DEPTH_STENCIL_VIEW_DESC descDSView;
ZeroMemory( &descDSView, sizeof( descDSView ) );
descDSView.Format = depthTextureDesc.Format;
descDSView.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
descDSView.Texture2D.MipSlice = 0;
DXEssayer( pD3DDevice->CreateDepthStencilView( pDepthTexture,
&descDSView,
&pDepthStencilView ),
DXE_ERREURCREATIONDEPTHSTENCILTARGET );

```

Remarquez le format : **DXGI_FORMAT_R8G8B8A8_UNORM**. J'utilise une texture avec composante alpha, au cas où nous voudrions nous en servir.

Remarquez aussi l'utilisation des fonctions **GetHauteur** et **GetLargeur** de la classe de dispositif. Si vous ne disposez pas de ces fonctions dans votre classe de dispositif, ajoutez-les, elles vous seront souvent utiles.

Les autres éléments sont créés de façon similaire à ce que nous avions fait pour le *shadow map* au chapitre précédent.

3. N'oubliez pas d'ajouter aussi le code de destruction de tous ces objets au destructeur de CPanneauPE:

```

DXRelacher(pResourceView);
DXRelacher(pRenderTargetView);
DXRelacher(pTextureScene);
DXRelacher(pDepthStencilView);
DXRelacher(pDepthTexture);

```

Atelier

12.4 L'affichage de la scène sur la texture

Fonction pour activer l'affichage sur la texture

Nous avons besoin d'activer l'affichage sur la texture, j'ai choisi d'implanter cette fonction dans la classe CPanneauPE (pour que la plupart des éléments principaux de l'exercice se retrouvent au même endroit).

1. Déclarez la fonction **DebutPostEffect()** dans la section *public* de la classe CPanneauPE:

```
void DebutPostEffect();
```

2. Puis définissez ainsi la fonction **DebutPostEffect()**

```
void CPanneauPE::DebutPostEffect()
{
    // Prendre en note l'ancienne surface de rendu
    pOldRenderTargetView = pDispositif->GetRenderTargetView();

    // Prendre en note l'ancienne surface de tampon Z
    pOldDepthStencilView = pDispositif->GetDepthStencilView();

    // Utiliser la texture comme surface de rendu et le tampon de profondeur
    // associé
    pDispositif->SetRenderTargetView(pRenderTargetView, pDepthStencilView);
}
```

3. Déclarez les deux variables **pOldRenderTargetView** et **pOldDepthStencilView** dans la section **protected** de la classe CPanneauPE:

```
ID3D11RenderTargetView* pOldRenderTargetView;
ID3D11DepthStencilView* pOldDepthStencilView;
```

4. La fonction de **CDispositifD3D11 SetRenderTargetView** n'existe pas encore!!! Il faut donc l'ajouter à la classe **CDispositifD3D11**. La voici:

```
void CDispositifD3D11::SetRenderTargetView(ID3D11RenderTargetView*
pRenderTargetView_in,
                                         ID3D11DepthStencilView*
pDepthStencilView_in)
{
    pRenderTargetView = pRenderTargetView_in;
    pDepthStencilView = pDepthStencilView_in;

    ID3D11RenderTargetView* tabRTV[1];
    tabRTV[0] = pRenderTargetView;
    pImmediateContext->OMSetRenderTargets( 1,
                                            tabRTV,
                                            pDepthStencilView);
}
```

Fonction pour désactiver l'affichage sur la texture

1. Déclarez la fonction **FinPostEffect()** dans la section public de la classe **CPanneauPE**:

```
void FinPostEffect();
```

2. Puis définissez ainsi la fonction **FinPostEffect ()**

```
void CPanneauPE::FinPostEffect()
{
    // Restaurer l'ancienne surface de rendu et le tampon de profondeur
    // associé
    pDispositif->SetRenderTargetView(pOldRenderTargetView,
                                      pOldDepthStencilView);
}
```

Activer l'affichage sur la texture

1. Modifiez la fonction RenderScene de CMoteur

```
virtual bool RenderScene()
{
    // Scene sur surface de rendu « normale »
    BeginRenderSceneSpecific();

    pPanneauPE->DebutPostEffect();

    // Scene sur surface de rendu POST-EFFECT
    BeginRenderSceneSpecific();

    // Appeler les fonctions de dessin de chaque objet de la scène
    for (auto& object3D : ListeScene)
    {
        Object3D->Draw();
    }

    EndRenderSceneSpecific();

    pPanneauPE->FinPostEffect();

    pPanneauPE->Draw();

    EndRenderSceneSpecific();

    return true;
}
```

La fonction Draw n'est pas encore en place...

12.5 Utiliser le panneau de post-effect

En pratique, nous utilisons déjà le panneau (voir la fonction **RenderScene** à la page précédente).

Mais il nous reste à initialiser les effets (les shaders) et à les utiliser dans la fonction **Draw**. Celle-ci sera particulièrement au centre de nos préoccupations pour cette section.

Initialisation des effets

1. Notez que dans la fonction **CPanneauPE::InitEffects()**, nous utilisons déjà le fichier d'effets "posteffect.fx". Nous reviendrons à nos shaders en 12.6.

```
...
DXEssayer( D3DX11CompileFromFile( L"PostEffect.fx", 0, 0, 0,
                                    "fx_5_0", 0, 0, 0,
                                    &pFXBlob, NULL, 0),
            DXE_ERREURCREATION_FX);
...
```

La fonction **Draw**

1. Nous ferons ici plusieurs modifications à la fonction **Draw**. Je vous présente d'abord le résultat! (pour les plus rapides...). Puis j'expliquerai certains éléments.

```
void CPanneauPE::Draw()
{
    // Obtenir le contexte
    ID3D11DeviceContext* pImmediateContext = pDispositif-
>GetImmediateContext();

    // Choisir la topologie des primitives
    pImmediateContext->IASetPrimitiveTopology(
        D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );

    // Source des sommets
    UINT stride = sizeof( CSommetPanneauPE );
    UINT offset = 0;
    pImmediateContext->IASetVertexBuffers( 0, 1, &pVertexBuffer, &stride,
                                            &offset );

    // Choix de la technique
    pTechnique = pEffet->GetTechniqueByIndex(0);
    pPasse = pTechnique->GetPassByIndex(0);

    // input layout des sommets
    pImmediateContext->IASetInputLayout( pVertexLayout );

    // Le sampler state
    ID3DX11EffectSamplerVariable* variableSampler;
    variableSampler = pEffet->GetVariableByName("SampleState")->AsSampler();
```

```
variableSampler->SetSampler(0, pSampleState);

ID3DX11EffectShaderResourceVariable* variableTexture;
variableTexture = pEffet->GetVariableByName("textureEntree")-
>AsShaderResource();

// Activation de la texture
variableTexture->SetResource(pResourceView);

pPasse->Apply(0, pImmediateContext);

// **** Rendu de l'objet
pImmediateContext->Draw( 6, 0 );

}
```

Pour l'instant, le seul paramètre de l'effet est la texture où nous avons effectué le rendu de la scène (**pResourceView**).

La technique utilisée est la technique « **Nul** » qui reproduit exactement la texture à l'écran, nous associerons cette technique à l'index 0. En réalité, c'est la possibilité de conserver un post-effect « sans effet ». Nous parlerons de cette technique en 12,6 (étape suivante).

12.6 L'effet « Nul »

L'effet « Nul » (la technique « Nul ») est en réalité un effet qui reproduit tel quel la texture réalisée par les étapes précédentes. Cet effet a pour résultat de nous donner le même affichage que si nous n'avions pas utilisé de panneau de post-effect!

Cet effet a deux avantages:

1. Il nous permet de ne pas avoir à désactiver nos shaders et l'affichage du panneau post-effect. Cette désactivation peut être dans certains cas un peu compliquée à implanter. Mais du côté performance-vitesse, pensez sérieusement à un moyen de les désactiver...
2. Cette technique nous servira de modèle de base pour l'implantation de toutes les autres techniques de post-effect. La plupart d'entre elles conserveront même le vertex shader.

Voici le code de **PostEffect.fx**, le fichier effet ne contenant pour le moment que la technique « Nul ».

```
struct VS_Sortie
{
    float4 Pos : SV_POSITION;
    float2 CoordTex : TEXCOORD0;
};

Texture2D textureEntree; // la texture
SamplerState SampleState; // l'état de sampling
```

```
VS_Sortie NulVS(float4 Pos : POSITION, float2 CoordTex : TEXCOORD0 )
{
    VS_Sortie sortie = (VS_Sortie)0;
```

```
    sortie.Pos = Pos;
    sortie.CoordTex = CoordTex;

    return sortie;
}

float4 NulPS( VS_Sortie vs ) : SV_Target
{
    float4 couleur;
```

```
    couleur = textureEntree.Sample(SampleState, vs.CoordTex);

    return couleur;
}
```

```
technique11 Nul
{
    pass p0
    {
        VertexShader = compile vs_5_0 NulVS();
        PixelShader = compile ps_5_0 NulPS();
        SetGeometryShader(NULL);
    }
}
```

- 1 La texture utilisée dans le rendu de la scène. Notez le « sampler » qui lui est associé (voir l'initialisation dans InitEffet). Même si dans notre cas, l'affichage sera effectué dans une proportion de 1 pour 1, j'ai conservé la possibilité d'utiliser des filtres de magnification, de minification et même de « mipmapping » (mais ça ne sert vraiment pas souvent...). J'ai aussi conservé le « wrap » pour l'adressage.
- 2 Le vertex shader (le VS), notez qu'il ne fait que reproduire les positions des sommets et les coordonnées d'application de texture. Nous n'avons pas besoin de matrices pour les transformations W, V et P parce que notre rectangle est déjà défini en coordonnées écran.
- 3 Le pixel shader est presque aussi simple, il se sert des coordonnées d'application de texture pour reproduire la texture à l'écran. D'autres effets pourront utiliser d'autres paramètres (par exemple la position).
- 4 La technique « Nul », une seule passe composée des shaders NulVS et NulPS.

Vous pouvez compiler et exécuter le tout, le résultat devrait être comme si nous n'avions pas utilisé de post-effect.

12.7 L'effet « RadialBlur »

L'effet « **Radial Blur** » est un effet relativement simple présenté pour une première fois dans le volume ShaderX 2 (série de Wolfgang Engel), un des effets « **Meshuggah** » présentés par Carsten Wenzel (de Crytek à l'époque). Nous en ferons ici une version simplifiée.

Le Pixel shader

```
float4 RadialBlurPS(VS_Sortie vs) : SV_Target
{
    float4 couleur;
    float4 ct;
    float2 tc = vs.CoordTex;
    float d, dx, dy;

    couleur = 0;
    float x = tc.x*2 - 1.0;
    float y = tc.y*2 - 1.0;
    dx = sqrt(x*x); // Distance du centre
    dy = sqrt(y*y); // Distance du centre

    dx = x * (distance*dx); // Le dégradé (blur) est en fonction de la
    dy = y * (distance*dy); // distance du centre et de la variable
    distance.

    x = x - (dx*10); // Vous pouvez jouer avec le nombre d'itérations
    y = y - (dy*10);

    tc.x = (x+1.0)/2.0;
    tc.y = (y+1.0)/2.0;

    for (int i = 0; i<10; i++) // Vous pouvez jouer avec le nombre
    d'itérations
    {
        ct = textureEntree.Sample(SampleState, tc);
        couleur = couleur * 0.6 + ct * 0.4; // Vous pouvez « jouer » avec les
        %

        x = x + dx;
        y = y + dy;

        tc.x = (x+1.0)/2.0;
        tc.y = (y+1.0)/2.0;
    }

    return couleur;
}
```

La technique

```
technique11 RadialBlur
{
    pass p0
    {
        VertexShader = compile vs_5_0 NulVS();
        PixelShader = compile ps_5_0 RadialBlurPS();
        SetGeometryShader(NULL);
    }
}
```

Cette technique est notre deuxième technique (numéro 1) et doit être insérée après la technique « Nul » si nous voulons utiliser les index pour y accéder.

Le paramètre distance

Ce paramètre (**constante** pour les shaders) permet de jouer un peu sur la longueur du dégradé.

Notez que nous n'utilisons pas de « *constant buffer* », mais un autre moyen d'accéder à nos constantes. Si vous avez plus de deux ou trois constantes, utilisez les « *constants buffers* ».

1. Déclarez dans « PostEffect.fx » :

```
float distance ;
```

Ajustements pour plusieurs effets (techniques)

Idéalement, du côté design, nous devrions penser à implanter un **gestionnaire d'effets et techniques** qui nous permettrait d'éviter de faire du « copier-coller » à chaque fois que nous voulons un nouvel effet.

Un bon point de départ est d'utiliser les indices (index) des techniques et passes plutôt que leur nom pour identifier celles-ci.

Puis, pour le moment, nous effectuerons l'initialisation des éléments de chaque technique dans la fonction **InitEffet**, mais cette fois nous utiliserons un certain nombre de tableaux pour nos initialisations.

1. Modifiez la déclaration du format de sommet (*vertex layout*) dans la section *protected* de la classe CPanneauPE.

```
static const int NOMBRE_TECHNIQUES = 2;
ID3D11InputLayout* pVertexLayout[NOMBRE_TECHNIQUES];
```

Le format de sommet est la seule chose qui n'est pas vraiment gérée par l'architecture effet. Il nous faut donc définir un *vertex layout* pour chaque vertex shader différent que nous aurons dans nos effets. Ici, nous aurions pu tricher et ne pas déclarer de nouveau *vertex layout* puisque

l'effet « Radial Blur » utilise le même *vertex shader* que l'effet Nul mais je préfère préparer pour améliorations futures.

2. Dans la fonction **InitEffet**, modifier les lignes suivantes:

```
...
pTechnique = pEffet->GetTechniqueByIndex(0);
pPasse = pTechnique->GetPassByIndex(0);

// Créer l'organisation des sommets pour le VS de notre effet
D3DX11_PASS_SHADER_DESC effectVSDesc;
D3DX11_EFFECT_SHADER_DESC effectVSDesc2;

pPasse->GetVertexShaderDesc(&effectVSDesc);
effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
&effectVSDesc2);

const void *vsCodePtr = effectVSDesc2.pBytecode;
unsigned vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout = NULL;
DXESSAYER( pD3DDevice->CreateInputLayout( CSommetPanneauPE::layout,
                                             CSommetPanneauPE::numElements,
                                             vsCodePtr,
                                             vsCodeLen,
                                             &pVertexLayout ),
            DXE_CREATIONLAYOUT );
...
...
```

pour qu'elles tiennent compte de notre tableau et aussi pour implémenter l'effet "Radial Blur" (index numéro 1).

```
...
pTechnique = pEffet->GetTechniqueByIndex(0);
pPasse = pTechnique->GetPassByIndex(0);

// Créer l'organisation des sommets pour le VS de notre effet
D3DX11_PASS_SHADER_DESC effectVSDesc;
D3DX11_EFFECT_SHADER_DESC effectVSDesc2;

pPasse->GetVertexShaderDesc(&effectVSDesc);
effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
&effectVSDesc2);

const void *vsCodePtr = effectVSDesc2.pBytecode;
unsigned vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout[0] = nullptr;
DXESSAYER( pD3DDevice->CreateInputLayout( CSommetPanneauPE::layout,
                                             CSommetPanneauPE::numElements,
                                             vsCodePtr,
                                             vsCodeLen,
                                             &pVertexLayout[0] ),
            DXE_CREATIONLAYOUT );

pTechnique = pEffet->GetTechniqueByIndex(1);
pPasse = pTechnique->GetPassByIndex(0);

pPasse->GetVertexShaderDesc(&effectVSDesc);
effectVSDesc.pShaderVariable->GetShaderDesc(effectVSDesc.ShaderIndex,
&effectVSDesc2);
```

```

const void *vsCodePtr2 = effectVSDesc2.pBytecode;
vsCodeLen = effectVSDesc2.BytecodeLength;

pVertexLayout[1] = NULL;
DXESSAYER( pD3DDevice->CreateInputLayout( CSommetPanneauPE::layout,
                                             CSommetPanneauPE::numElements,
                                             vsCodePtr2,
                                             vsCodeLen,
                                             &pVertexLayout[1] ),
            DXE_CREATIONLAYOUT);

```

...

3. Modifiez le destructeur pour qu'il relâche tous les vertex layout. Remplacez

DXRelacher(pVertexLayout);

par

```
for (int i = 0; i < NOMBRE_TECHNIQUES; i++) DXRelacher(pVertexLayout[i]);
```

La fonction Draw

1. Elle ressemble beaucoup à ce que nous avions, mais nous allons maintenant utiliser le numéro d'index pour la technique "Radial Blur" (numéro 1)

```

// Choix de la technique
pTechnique = pEffet->GetTechniqueByIndex(1);
pPasse = pTechnique->GetPassByIndex(0);

// input layout des sommets
pImmediateContext->IASetInputLayout( pVertexLayout[1] );

```

2. On ajoute ensuite le code pour l'initialisation de la constante **distance**, n'importe où avant l'énoncé **Apply**.

```

// La « constante » distance
ID3DX11EffectScalarVariable* distance;
distance = pEffet->GetVariableByName("distance")->AsScalar();
distance->SetFloat((float)0.10f);

```

Notez que si la distance est à 0, il n'y a pas de dégradé. Par contre, plus de 10 % (0.10) commencent à provoquer des artéfacts (à cause de nos formules). J'ai choisi de modifier la distance à chaque image puis que, même si nous utilisons pour le moment une constante, la distance pourrait être modifiée dans la fonction **Anime** pour provoquer un effet variable (comme une accélération ou un « turbo »).

3. Compilez et exécutez.

13. Les « *geometry shaders* »

Les geometry shaders sont des petits programmes (comme nos autres shaders) qui s'insèrent entre les vertex shader et le rasterizer stage dans notre pipeline (si nous n'avons pas utilisé les stages de tessellation). Ils nous permettent de modifier la géométrie de notre objet soit en modifiant les sommets, soit en les retirant ou soient en en créant de nouveaux.

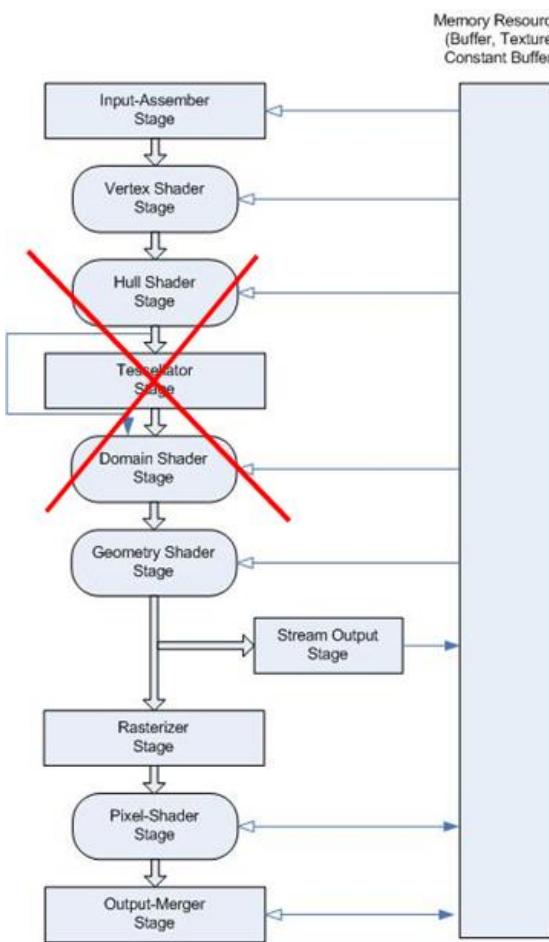
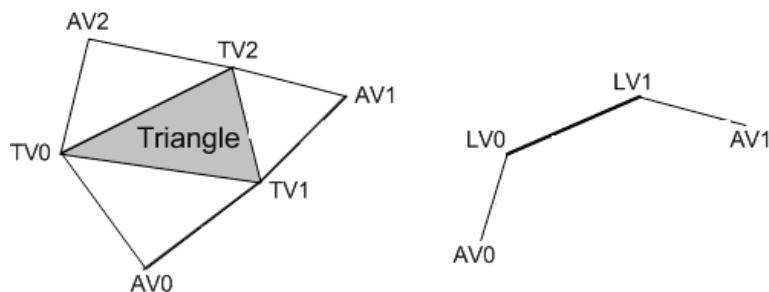
Objectifs du chapitre

- ✓ Se familiariser avec les geometry shaders
- ✓ Programmation des geometry shaders
- ✓ Intégration des geometry shaders

13. Les « *geometry shaders* »

L'étape de ***geometry-shader*** (GS) exécute de petits programmes qui reçoivent des sommets en entrée et génèrent des sommets en sortie.

Les sommets en entrée sont toutefois regroupés soient trois sommets pour un triangle, deux pour une ligne et un seul pour un point. Les geometry shaders peuvent aussi recevoir en entrée des données pour les primitives adjacentes (deux pour les lignes et trois pour les triangles).



Il est donc possible de faire des traitements tenant compte des propriétés de la primitive (et des primitives adjacentes) et même de faire des sorties utilisant une autre topologie que l'entrée. Un exemple intéressant utilise des points en entrée et génère des quads (deux triangles) en sortie, les quads permettant d'afficher un système de particules.

Le geometry shader effectue ses sorties un sommet à la fois en ajoutant ainsi les sommets à un flux de sortie (*output stream*). La topologie du flux est déterminée par la déclaration utilisée soit **Point-Stream**, **LineStream**, ou **TriangleStream**. Ce sont des objets (en réalité des *templates*) qui peuvent être déclarés et utilisés dans le geometry shader. La topologie est donc en fonction du template et le format de sommets en fonction du type utilisé.

D'autres types d'algorithmes pouvant être implantés :

- Génération de cheveux, de fourrure
- Shadow Volume
- Rendu monopasse sur un Cubemap
- Ajustement de matériel

13.1 La programmation des geometry shaders

La programmation d'un geometry shader ressemble beaucoup à la programmation d'un vertex shader mais nous noterons tout de même plusieurs nouveautés. Un geometry shader ressemble à ceci :

```
[maxvertexcount(N)] 1
void ShaderName ( PrimitiveType InputVertexType InputName[NumElements],
                  inout StreamOutputObject<OutputVertexType> OutputName)
{
    // à programmer
}
```

Par exemple :

```
struct GSPOS_INPUT
{
    float4 Pos : SV_POSITION;
    float3 Norm : TEXCOORD0;
    float2 Tex : TEXCOORD1;
};

[maxvertexcount(12)] 1
void GS( triangle GSPOS_INPUT input[3],
         inout TriangleStream<GSPOS_INPUT> TriStream ) 2
{
```

Nous devons spécifier le nombre de sommets qui seront « sortis » par le geometry shader pour cet appel. Nous devons le spécifier au moyen de l'énoncé **maxvertexcount**. Par exemple

```
[maxvertexcount(12)]
```

où 12 est le nombre maximum de sorties effectuées par le geometry shader en un seul appel (notez qu'il peut y en avoir moins).

maxvertexcount devrait toujours être le plus petit possible (pour des raisons de performance). Au besoin, vous pouvez faire deux shaders si vous avez *occasionnellement* besoin de plus de sommets en sortie.

Le geometry shader utilise (sauf exception) deux paramètres, un d'entrée et un de sortie.

Le **paramètre d'entrée** est toujours un tableau de sommets défini selon la topologie (un sommet pour un point, deux pour une ligne, trois pour un

triangle, quatre pour une ligne avec primitives adjacentes, et six pour un triangle avec primitives adjacentes).

Le type des sommets doit avoir le même type que celui retourné par le vertex shader. Le tableau d'entrée doit être préfixé par un type de primitive soient **point**, **line**, **triangle**, **lineadj**, ou **triangleadj**.

(2)

Le **paramètre de sortie** a la spécification **inout**. De plus, il s'agit toujours d'un type de flux. Les flux que nous utiliserons nous permettront de stocker la liste des sommets qui constitueront notre nouvelle géométrie. Les types de flux disponibles sont:

- PointStream<OutputVertexType>
- LineStream<OutputVertexType>
- TriangleStream<OutputVertexType>

Les sommets sont ajoutés au flux au moyen de la fonction **Append**:

```
void StreamOutputObject<OutputVertexType>::Append(OutputVertexType v);
```

Les sorties du geometry shader doivent former des primitives, le type étant en fonction du type de flux (**PointStream**, **LineStream**, **TriangleStream**). **IMPORTANT:** pour des lignes et des triangles la primitive de sortie est **toujours** une bande (**strip**). Ce qui nous demandera souvent une bonne « gymnastique » de création de sommets.

Dans les cas où nos algorithmes nécessitent des listes, les listes de triangles ou de lignes peuvent tout de même être simulées en utilisant la fonction **RestartStrip**:

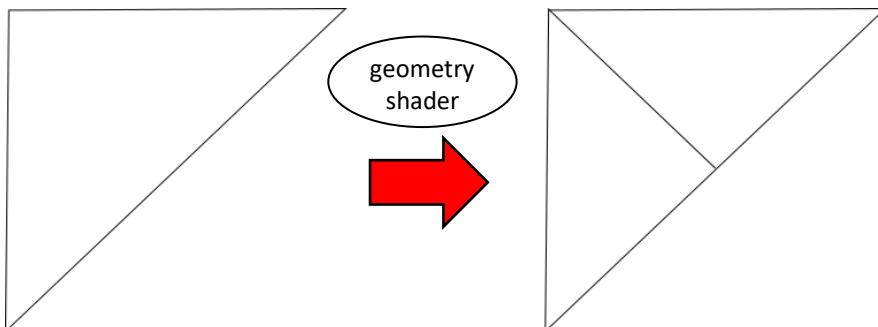
```
void StreamOutputObject<OutputVertexType>::RestartStrip();
```

Par exemple, pour des triangles, vous pouvez appeler cette fonction à chaque trois sommets et ainsi, vous aurez l'équivalent d'une liste de triangles.

13.2 Une petite tessellation

La tessellation est le nom donné à certaines techniques qui ont pour but de subdiviser des polygones. Cette subdivision permet d'ajouter plus de détails et d'éviter certains artefacts reliés aux objets trop grands.

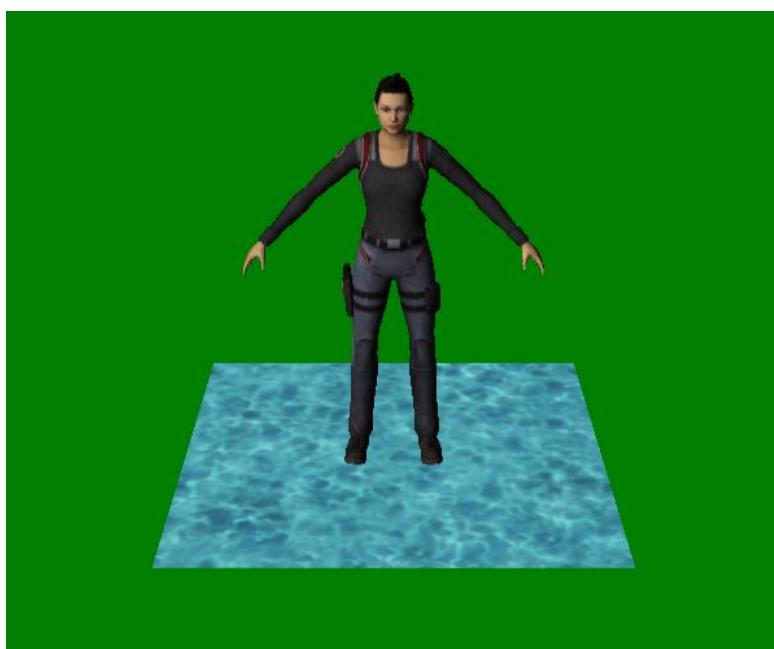
Nous planterons ici (comme exemple d'utilisation des geometry shaders) une tessellation minimale soit celle qui consiste à subdiviser un triangle en deux :



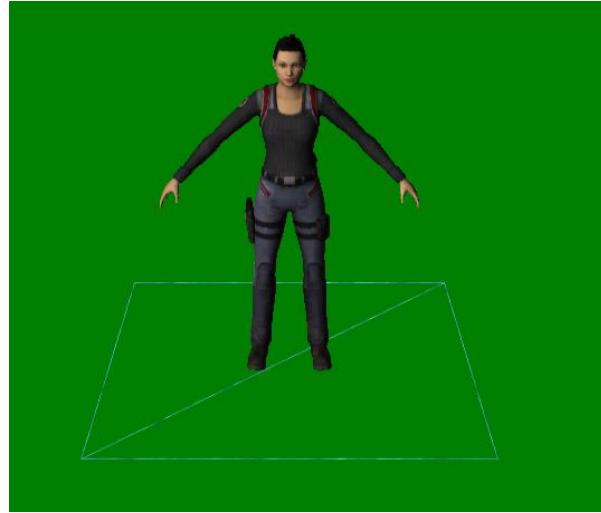
Dans notre cas, la « diagonale » sera utilisée pour la subdivision puisqu'elle sera facile à identifier, mais d'autres techniques existent.

Pour débuter

Pour simplifier l'exercice, nous partirons du projet « Chapitre 13 – départ » qui reprend l'atelier du chapitre 10 mais ajoute un petit panneau sous le personnage. Pour le moment, le panneau utilise une texture d'eau.



Pour mieux visualiser les effets de la tessellation, le projet implante la touche F2 qui permet d'afficher le panneau en mode fil de fer (**wireframe**).



À faire

Nous allons donc découper chaque triangle en deux en lui ajoutant un sommet supplémentaire et en construisant les deux triangles résultants.

```
PointSupp = sommet[0].pos + (sommet[2].pos - sommet[0].pos)/2;
```

Nouveau triangle 1 = sommet[0], sommet[1], SommetSuppl

Nouveau triangle 2 = sommet[1], SommetSuppl, sommet[2]

Vous noterez plus loin que nous utiliserons la topologie **TriangleStrip** pour la sortie du geometry shader, c'est la sortie de défaut pour les triangles (en réalité, c'est la seule sortie possible...).

Le panneau est défini dans la classe **CPanneauGS** et utilise un effet appelé « **EssaiGS.FX** ». C'est cet effet que nous devrons modifier pour qu'il implante notre futur geometry shader.

Le geometry shader

```
[maxvertexcount(4)] 1
void EssaiGS( triangle VS_Sortie input[3],
              inout TriangleStream<VS_Sortie> TriStream ) 2
{
    VS_Sortie output;

    // Calculer la normale
    float3 CoteA = input[1].Pos - input[0].Pos;
    float3 CoteB = input[2].Pos - input[0].Pos;
    float3 Normale = normalize( cross(CoteA, CoteB) ); 3
    float4 PointSupp = input[0].Pos + (input[2].Pos - input[0].Pos)/2; 4

    // Points 0 et 1
    for( int i=0; i<2; i++ )
    {
        output.Pos = mul( input[i].Pos, matWVP ); 5
        output.Norm = Normale;
        output.coordTex = input[i].coordTex;
        TriStream.Append( output );
    }

    // Point supplémentaire
    output.Pos = mul( PointSupp, matWVP );
    output.Norm = Normale;
    output.coordTex = input[0].coordTex +
                      (input[2].coordTex - input[0].coordTex)/2; 6
    TriStream.Append( output );

    // Point 2
    output.Pos = mul( input[2].Pos, matWVP );
    output.Norm = Normale;
    output.coordTex = input[2].coordTex;
    TriStream.Append( output );
}

}
```

Notez:

- 1 Le nombre de sommets en sortie est de 4 seulement.
- 2 L'entrée et la sortie utilisent des sommets de type VS_Sortie mais nous aurions pu utiliser des types différents.
- 3 Nous calculons la normale même si nous ne l'utilisons pas dans le pixel shader. Ce sera indispensable pour des shaders utilisant l'éclairage.
- 4 Le point supplémentaire est calculé au centre de la diagonale (dans notre cas, de sommet[0] à sommet[2]).
- 5 La position est multipliée par la matrice matWVP, il **faudra retirer** ce calcul du vertex shader.
- 6 Les coordonnées de texture du nouveau sommet sont calculées.

Petite modification au vertex shader

```
VS_Sortie EssaiVS(float4 Pos : POSITION, float2 coordTex: TEXCOORD)
{
    VS_Sortie sortie = (VS_Sortie)0;
    sortie.Pos = Pos; // On ne fait rien...

    // Coordonnées d'application de texture
    sortie.coordTex = coordTex;

    return sortie;
}
```

La technique

```
technique11 TechniqueGS
{
    pass pass0
    {
        SetVertexShader(CompileShader(vs_5_0, EssaiVS()));
        SetPixelShader(CompileShader(ps_5_0, EssaiPS()));
        SetGeometryShader(CompileShader(gs_5_0, EssaiGS()));
    }
}
```

Exercice

13.3 En liste de triangle

Plusieurs algorithmes de tessellation fonctionnent difficilement avec les « triangles strips ». Pour l'entrée du geometry shader, ce n'est pas un problème puisque nous recevons toujours trois sommets peu importe la topologie originale (*Strip* ou *List*). Pour la sortie, la « gymnastique » d'organisation des strips est parfois presque incompatible avec l'algorithme utilisé pour créer les nouveaux triangles. Il faudra alors utiliser la fonction **RestartStrip** pour simuler une liste de triangles.

Exercice

Modifiez le geometry shader de l'atelier 13.2 pour qu'il utilise des *triangles lists*.

14. Les shaders de tessellation

La tessellation consiste à décomposer des polygones en plusieurs polygones plus petits. Par exemple, si nous coupons un triangle en deux, nous obtenons une tessellation. Toutefois, la tessellation n'améliore le réalisme que si les nouveaux triangles sont utilisés pour ajouter de nouvelles informations.

Dans le chapitre précédent, nous avons utilisé le geometry shader pour effectuer une tessellation simple. Les geometry shaders peuvent être utilisées pour ajouter des primitives, mais ne sont pas nécessairement très performantes pour l'ajout de beaucoup de polygones. Tout dépendra de l'algorithme désiré.

Objectifs du chapitre

- ✓ Se familiariser avec les shaders « hull » et « domain »
- ✓ Programmation des shaders pour la tessellation
- ✓ Intégration des shaders « hull » et « domain »

14. Les shaders de tessellation

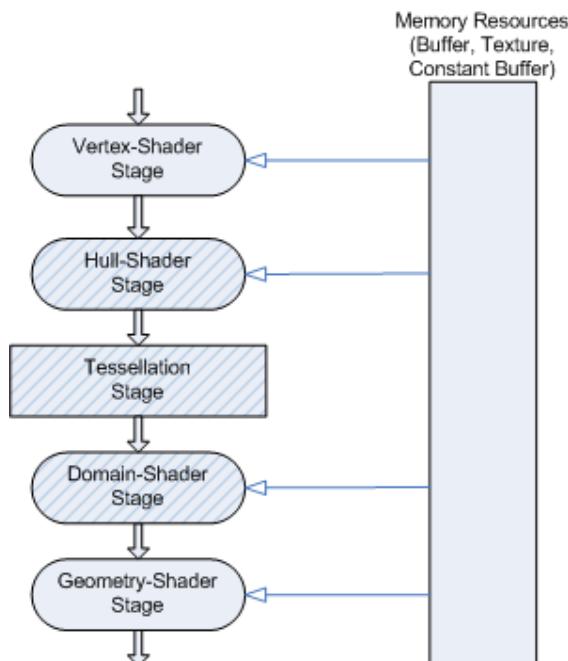
Dans les versions précédentes de DirectX, la tessellation devait être effectuée par le CPU (ou à la rigueur par un Geometry Shader). Cette méthode demeure intéressante si la tessellation est plus ou moins « statique », mais dans le cas de tessellation plus dynamique, par exemple pour des terrains, le transfert et le rafraîchissement des sommets peuvent facilement créer un ralentissement. Avec DirectX 11 et les shaders de modèle 5, il est possible de placer un objet de base en mémoire vidéo puis de faire exécuter notre algorithme de subdivision sur la carte graphique.

Théorie

14.1 Les étapes de tessellation

La tessellation est en réalité composée de trois étapes:

- Étape **Hull Shader** – C'est une étape programmable qui produira une nouvelle « **patch** » (voir plus loin) et ses constantes pour chaque « patch » reçue en entrée.
- Étape **Tessellator** – C'est une étape fixe qui crée des échantillons de domaine représentant la patch géométrique et qui génère un ensemble d'objets plus petits (triangles, points, ou lignes) pour connecter ces échantillons.
- Étape **Domain Shader** – Une étape programmable qui calcule la position des sommets pour chaque échantillon du domaine.



Qu'est-ce qu'une patch ?

Une « **patch** » est un nouveau type de primitive utilisé pour la tessellation et qui peut avoir de 1 à 32 **points de contrôle** qui seront utilisés pour la génération d'une nouvelle patch. La similarité avec les courbes de Bézier et les surfaces de Bézier est volontaire. De nouveaux types de primitives ont donc été ajoutés et devront être utilisés par la fonction **IASetPrimitiveTopology (...)** pour nous permettre d'utiliser la tessellation. Les nouveaux types sont:

```
D3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST
D3D11_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST
...
D3D11_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST
```

L'étape Hull Shader (HS)

Le **hull shader**, qui sera appelé pour chaque patch d'une façon décrite plus loin, transformera les points de contrôle d'entrée (surface avec peu de détails) en une nouvelle liste de points de contrôle définissant une nouvelle patch plus détaillée. Nous y ferons aussi des calculs permettant de fournir des données aux étapes suivantes (TS et DS). Ces données deviendront des « constantes » disponibles aux étapes suivantes.



Un hull shader a les propriétés suivantes:

- L'entrée du shader est de 1 à 32 points de contrôle.
- La sortie du shader est de 1 à 32 points de contrôles, peu importe le nombre de **facteurs de tessellation** (plus de détails à l'étape tessellation). Les points de contrôle de sortie et les constantes seront utilisés par le **domain shader**. Les facteurs de tessellation seront utilisés par l'étape tessellation et par le domain shader.
- Les facteurs de tessellation détermineront comment effectuer la division de chaque patch.
- Le hull shader est aussi responsable de déclarer l'état requis par l'étape tessellation. Il s'agit d'information sur le nombre de points

de contrôle, le type de surface et le type de partitionnement à utiliser lors de la tessellation. Cette information apparaîtra sous la forme des déclarations précédant le code du shader.

- Si le hull shader utilise un facteur de tessellation ≤ 0 ou NaN, la patch sera éliminée. Avec comme résultat que l'étape de tessellation ne sera pas exécutée (sauf exception), que le domain shader ne sera pas exécuté et qu'il n'y aura pas de sortie visible pour cette patch..

De façon plus intéressante, un hull shader deviendra en réalité deux phases: une phase de génération de points de contrôle et une phase de construction de constantes, ces deux phases seront exécutées en **parallèle** par le GPU. Le compilateur HLSL s'occupe lui-même de l'extraction du parallelisme.

- La phase de points de contrôle : cette phase est appellée une fois par point de contrôle de sortie..
- La phase des constantes: cette phase est appelée une fois par patch. Cette phase a un accès lecture seulement aux points de contrôle.

L'étape Tessellation

Le **tessellateur** est une étape fixe du pipeline, elle sera active seulement si nous avons un hull shader. Son but est de subdiviser un **domaine** (quad, triangle ou ligne) en plus petits objets donc d'effectuer une tessellation.

La tessellation est effectuée selon des coordonnées « normalisées » c'est-à-dire qu'un quad sera de dimension théorique 1 X 1 pour le tessellateur.

Le tessellateur travaille sur chaque patch en utilisant les facteurs de tessellation et le type de partitionnement déterminés dans le hull shader. Le tessellateur détermine ainsi des coordonnées uv « normalisées » pour chaque sommet de la patch résultant de la tessellation (une valeur w peut aussi être obtenue).

Le tessellateur travaille en deux phases:

- La première phase traite les facteurs de tessellation, elle ajuste les problèmes suivant: arrondissement des nombres, très petits facteurs et autres problèmes de réduction et de combinaison des facteurs au moyen de fonctions point flottant 32 bits.
- La deuxième phase génère les nouveaux points selon le type de partitionnement choisi. C'est la tâche principale du tessellateur. Pour ce faire, il utilise des fractions 16 bits en point fixe, ce qui permet des calculs très rapides tout en conservant une précision acceptable. Par exemple, pour une patch de 64 unités, nous pouvons avoir des points à une résolution de 0.002.

Type de partitionnement	intervalle
Fractional_odd	[1...63]
Fractional_even	[2..64]
Integer	[1..64]
Pow2	[1..64]

L'étape Domain Shader

Cette étape permet de compléter les informations des nouveaux sommets à partir des informations de la patch et des valeurs uv reçus du tessellateur. Le domain shader est appelé une fois pour chaque sommet créé par le tessellateur. Il a des accès lecture seulement aux coordonnées uv, aux informations de la patch issue du hull shader et aux constantes du hull shader.

La sortie du domaine shader est un sommet qui sera utilisé par l'étape rasterisation pour créer les polygones qui généreront les pixels.

14.2 Un exemple

Nous ferons ici un exemple simple soit le découpage de notre panneau (un peu comme au chapitre précédent, mais avec un peu plus de tessellation).

Pour débuter

Pour simplifier l'exercice, **nous partirons du projet « Chapitre 14 — départ »** qui reprend l'atelier du chapitre 10, mais ajoute un petit panneau sous le personnage. Il s'agit d'une reprise de « Chapitre 13 — Départ » où j'ai copié la classe de panneau pour en faire une nouvelle (**CPanneauTess**) qui nous servira ici. Si vous vous rappelez, l'utilisation de F2 affichait le panneau en mode fil de fer.

IMPORTANT

N'exécutez pas le programme avant d'en avoir la confirmation. Les résultats intermédiaires sont imprévisibles.

1. Copiez ce projet et renommez-le « **Chapitre 14 — Atelier** ».

Modifications au panneau

2. Notre panneau sera maintenant composé de seulement quatre sommets (nos points de contrôle) disposés à plat sur le plan XZ. Modifiez les sommets ainsi::

```
CSommetPanneauTess CPanneauTess::sommets[4] = {
    CSommetPanneauTess(XMFLOAT3(-1.0f, 0.0f, -1.0f), XMFLOAT2(0.0f, 1.0f)),
    CSommetPanneauTess(XMFLOAT3(-1.0f, 0.0f, 1.0f), XMFLOAT2(0.0f, 0.0f)),
    CSommetPanneauTess(XMFLOAT3( 1.0f, 0.0f, 1.0f), XMFLOAT2(1.0f, 0.0f)),
    CSommetPanneauTess(XMFLOAT3( 1.0f, 0.0f, -1.0f), XMFLOAT2(1.0f, 1.0f)) };
```

3. Modifiez aussi la déclaration de **sommets**:

```
static CSommetPanneauTess sommets[4];
```

4. Puisque nos sommets sont maintenant sur le plan XZ, modifiez l'initialisation de matWorld dans le constructeur de CPanneauTess:

```
matWorld = XMMatrixIdentity();
```

matWorld pourrait nous servir à animer le panneau dans le monde. Dans notre cas, nous ne ferons rien.

5. Modifiez aussi le nombre d'éléments à placer dans le vertex buffer:

```
bd.ByteWidth = sizeof( CSommetPanneauTess ) * 4;
```

Modification à la fonction Draw

6. La première chose à faire pour utiliser la tessellation est de modifier le format de nos primitives dans la fonction Draw. Dans notre cas, nous aurons quatre points de contrôle par patch (une seule patch) donc:

```
// 1. Choisir la topologie des primitives
pImmediateContext->IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST );
```

7. Modifiez aussi l'appel à la fonction Draw

```
// **** Rendu de l'objet
pImmediateContext->Draw( 4, 0 );
```

 Vous pouvez compiler et exécuter, mais le panneau ne s'affiche pas! Les shaders de tessellation ne sont pas encore là.

Modifications au vertex shader

Comme nous construirons de nouvelles primitives, le vertex shader est pour le moment limité à la transformation de nos sommets dans le monde. La transformation ViewProj sera appliquée dans le domain shader aux nouveaux sommets.

8. Modifiez le vertex shader pour qu'il ressemble à ceci:

```
VS_Sortie EssaiVS(float4 Pos : POSITION, float2 coordTex: TEXCOORD)
{
    VS_Sortie sortie = (VS_Sortie)0;

    sortie.Pos = mul(Pos, matW );

    // Coordonnées d'application de texture
    sortie.coordTex = coordTex;

    return sortie;
}
```

9. Modifiez aussi la structure de donnée VS_Sortie ainsi:

```
struct VS_Sortie
{
    float4 Pos : POSITION;      // Ne PAS utiliser SV_Position
    float3 Norm : NORMAL;
    float2 coordTex : TEXCOORD0;
};
```

Construction du hull shader

Nous allons maintenant coder le **hull shader**. Ce shader est composé de deux parties (phases): la partie des constantes et le hull shader lui-même.

L'**entrée** du hull shader sera composé de quatre sommets qui proviennent évidemment de la sortie du vertex shader (VS_Sortie).

La première **sortie** du hull shader sera un objet de type HS_CONSTANTES_Sortie pour les informations appartenant à la patch. Nous n'utiliserons ici que le strict minimum nécessaire à la tessellation soit les facteurs de tessellation de nos quads.

10. Dans le fichier **EssaiTess.FX**, ajoutez la déclaration suivante, avant le pixel shader:

```
struct HS_CONSTANTES_Sortie
{
    float Bords[4] : SV_TessFactor;
    float Interieurs[2] : SV_InsideTessFactor;
};
```

Nous aurons quatre facteurs de tessellation pour les bords du quad (4 bords). Nous utiliserons aussi deux facteurs de tessellation « intérieurs » qui représentent la division horizontale et verticale du centre du quad.

La deuxième **sortie** du hull shader sera un type de sommet de contrôle qui pourrait être différent de l'entrée, mais dans notre cas, il sera identique.

11. Dans le fichier **EssaiTess.FX**, ajoutez la déclaration suivante, avant le pixel shader :

```
struct HS_Sortie
{
    float4 Pos : POSITION;
    float3 Norm : NORMAL;
    float2 coordTex : TEXCOORD0;
};
```

12. Ajoutez le code de la partie « génération des constantes » du hull shader :

```
HS_CONSTANTES_Sortie ConstantHS( InputPatch<VS_Sortie, 4> ip, uint PatchID : SV_PrimitiveID )
{
    HS_CONSTANTES_Sortie Sortie;
    Sortie.Bords[0] = Sortie.Bords[1] = Sortie.Bords[2]
        = Sortie.Bords[3] = TessellationFactor; ← ??
    Sortie.Interieurs[0] = Sortie.Interieurs[1] = TessellationFactor;

    return Sortie;
}
```

? → La constante **TessellationFactor** n'est pas encore déclarée ni utilisée dans la fonction Draw.

Cette fonction effectuera la sortie d'une structure de type HS_CONSTANTES_Sortie et reçoit en entrée quatre sommets de contrôle (de format VS_Sortie). Nous recevons aussi comme paramètre d'entrée la valeur PatchID qui est un identifiant de patch généré par l'étape Input Assembler (IA). Cette fonction n'est appelée qu'une seule fois par patch. Nous aurions pu ajouter comme constantes des éléments associés à notre algorithme comme la distance de la caméra... Dans notre cas, les facteurs de tessellation sont constants, mais ils pourraient dépendre d'un calcul et même être ≤ 0 (dans ce cas, la patch serait éliminée du pipeline).

13. Déclarez **TessellationFactor** dans la déclaration des constantes:

```
cbuffer param
{
    float4x4 matW;
    float4x4 matVP; // la matrice vue proj
    float TessellationFactor;
    float3 remplissage;
}
```

Déclarez aussi **remplissage** (rappel: les tampons doivent être d'une dimension multiple de 16).

14. Codez le hull shader ainsi (à la suite dans **EssaiTess.FX**):

```
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_ccw")]
[outputcontrolpoints(4)]
[patchconstantfunc("ConstantHS")]
HS_Sortie EssaiHS( InputPatch<VS_Sortie, 4> p, uint i :
SV_OutputControlPointID, uint PatchID : SV_PrimitiveID )
{
    HS_Sortie Sortie;
    Sortie.Pos = p[i].Pos;
    Sortie.Norm = p[i].Norm;
    Sortie.coordTex = p[i].coordTex;

    return Sortie;
}
```

Ce code est appelé une fois par point de contrôle (sommel) de sortie, mais nous avons accès à toutes les données des patchs d'entrée et de sortie. Nous recevons en **entrée** les quatre sommets de contrôle (de format **VS_Sortie**), la valeur **PatchID** et le **i** est le numéro du point de contrôle de sortie présentement traité.

Les déclarations en préfixe sont :

domain (quad, tri, isoline) : Spécifie avec quoi nous travaillons, des quads, des triangles ou des lignes. Dans notre exemple, nous devons choisir **quad**.

partitioning (integer, fractional_even, fractional_odd, pow2) : Cet attribut indique au Tessellateur (Etape Tessellation) comment interpréter nos facteurs de tessellation. **Integer** veut dire que les facteurs seront interprétés comme des valeurs entières. Les nouveaux sommets ne seront pas nécessairement distribués de belle façon. Les paramètres **fractional_even** et **fractional_odd** créeront des sommets intermédiaires donc le résultat sera plus « joli ». **Pow2** se comporte d'une façon similaire à **integer**..

outputtopology(triangle_cw, triangle_ccw, line) : Cet attribut indique au Tessellateur comment devraient être organisées les primitives de sortie. **triangle_cw** construit des triangles en sens horaire, **triangle_ccw** construit des triangles en sens antihoraire et **line** construit des lignes. (l'orientation sur le plan X-Z est « inversée »).

outputcontrolpoints (1..32) : Le nombre de points de contrôle de sortie et le nombre de fois que le hull shader sera appelé. Pour notre exemple, nous conserverons les mêmes points de contrôle qu'en entrée.

patchconstantfunc (string) : Le nom de la fonction utilisée pour générer les constantes. Dans notre cas, c'est ConstantHS.
Important : notez que les constantes ne sont pas accessibles au hull shader lui-même.

Construction du domain shader

Nous pouvons maintenant coder le **domain shader**. Son rôle est souvent décrit comme celui d'un « vertex shader après la tessellation ». Dans ce shader, nous pourrons manipuler tous les sommets créés par le Tessellateur.

15.Dans le fichier **EssaiTess.FX**, ajoutez la déclaration suivante, avant le pixel shader :

```
struct DS_Sortie
{
    float4 Pos : SV_Position;
    float3 Norm : Normale;
    float2 coordTex : TEXCOORD0;
};
```

Vous constatez qu'il s'agit toujours de la même structure, j'en crée une nouvelle pour considération future...

16. Ajoutez maintenant le domain shader:

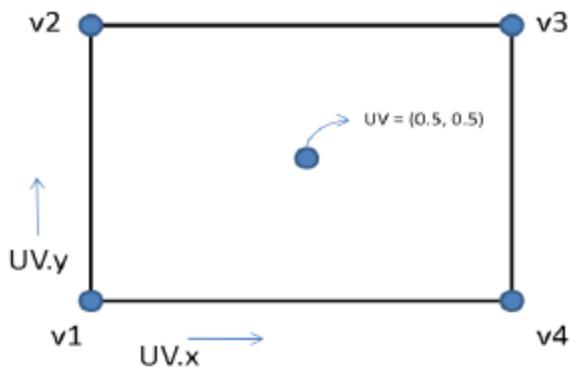
```
[domain("quad")]
DS_Sortie EssaiDS( HS_CONSTANTES_sortie entree, float2 UV :
SV_DomainLocation, const OutputPatch<HS_Sortie, 4> quad )
{
    DS_Sortie Sortie;
    float3 verticalPos1 = lerp(quad[0].Pos,quad[1].Pos,UV.y);
    float3 verticalPos2 = lerp(quad[3].Pos,quad[2].Pos,UV.y);
    float3 finalPos = lerp(verticalPos1,verticalPos2,UV.x);
    Sortie.Pos = mul( float4(finalPos,1), matVP );

    float3 coordTex1 = lerp(quad[0].coordTex,quad[1].coordTex,UV.y);
    float3 coordTex2 = lerp(quad[3].coordTex,quad[2].coordTex,UV.y);
    Sortie.coordTex = lerp(coordTex1, coordTex2,UV.x);

    return Sortie;
}
```

Le domain shader reçoit en entrée une structure HS_CONSTANTES_Sortie qui pourrait contenir des informations « par patch » reçues du hull shader. Nous devons aussi spécifier le même type d'attribut d'entrée que nous avions choisi pour le hull shader soit « quad ». L'appel au domain shader reçoit aussi les points de contrôle de la patch et un float2 UV contenant les données paramétriques du nouveau sommet dans l'**« espace patch »**.

Ces données permettent de mieux calculer les informations du nouveau sommet, par exemple $UV = (0.5, 0.5)$ désigne un point au milieu du quad:



ANNEXE 1. Bases mathématiques

ANNEXE 1. Bases mathématiques

Théorie

A.1 Les transformations sur le plan

Sur le plan, les formules sont plus simples et les opérations plus faciles à visualiser qu'en 3D. Les transformations 2D demeurent des opérations essentielles dans le monde de la 3D, leurs applications sont légions: rotation de textures, animation de textures, animation des sprites, calcul des couleurs, certaines étapes de l'anticrénelage (*antialiasing*) et la conception de plusieurs types d'effets spéciaux.

Les vecteurs, les points et les sommets

En notation mathématique, un vecteur est une coordonnée cartésienne qui détermine un certain déplacement par rapport à l'origine. En 2D, on identifiera un vecteur par (x, y) . Mais en pratique, le terme vecteur est souvent trop général. On emploiera souvent le terme **point** (ou position) pour identifier une coordonnée et le terme **sommet** pour identifier un point d'une figure. Notez que très rapidement le terme sommet prendra un sens élargi.

D'autres utilisations des vecteurs existent, entre autres pour identifier une translation de façon simple, pour identifier une direction de mouvement, pour spécifier où est le haut, pour spécifier la direction où regarde la caméra, etc.

Transformations

Les transformations seront pour nous des fonctions appliquées à des objets 3D, mais en réalité, une transformation est l'application d'une formule mathématique à chaque point (sommet) de l'objet. De plus les transformations serviront souvent à autre chose qu'au déplacement des objets.

Translation

Le déplacement d'un objet est appelé **translation**. Chaque sommet de l'objet est déplacé en appliquant la translation à ses coordonnées.

- P est un point de coordonnées (x , y)
- T est une translation exprimée par le vecteur (dx, dy)
- P' est le résultat de T(P) soit un point de coordonnées (x' , y')

$$x' = x + dx$$

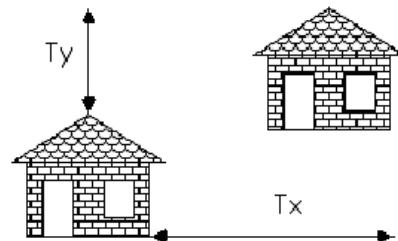
$$y' = y + dy$$

ou sous forme d'une fonction

$$P' = T(P)$$

ou par addition de vecteurs

$$P' = P + T$$



Rotation

Un objet peut subir une rotation. Celle-ci correspond toujours à la rotation d'un angle θ des points de l'objet autour de l'origine du système de coordonnées. La rotation d'un objet sur lui-même n'existe pas réellement.

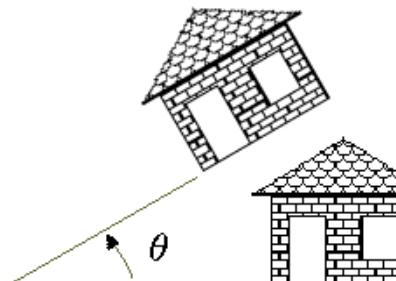
- P est un point de coordonnées (x , y)
- R est une rotation exprimée par l'angle θ
- P' est le résultat de R(P) soit un point de coordon:

$$x' = x * \cos\theta - y * \sin\theta$$

$$y' = x * \sin\theta + y * \cos\theta$$

ou sous forme d'une fonction

$$P' = R(P)$$



Homothétie

Changer les dimensions d'un objet est appelé homothétie (en anglais *scaling* ou changement d'échelle). Nous appliquons l'homothétie en multipliant les coordonnées x et y d'un point par des valeurs s_x et s_y .

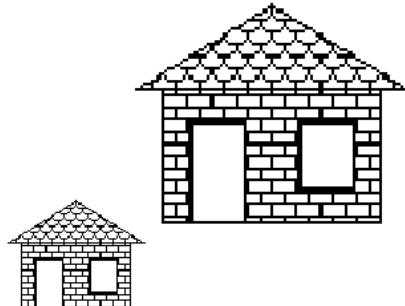
- P est un point de coordonnées (x, y)
- H est une homothétie exprimée par les valeurs s_x et s_y
- P' est le résultat de $H(P)$ soit un point de coordonnées (x', y')

$$x' = s_x * x$$

$$y' = s_y * y$$

ou sous forme d'une fonction

$$P' = H(P)$$



IMPORTANT

Nous devrons **éviter** de placer des homothéties dans les matrices de transformation, car celles-ci ne sont pas seulement appliquées aux sommets, mais aussi aux normales, aux vecteurs d'éclairage et à d'autres vecteurs qu'il ne faut pas « réduire » ou « agrandir » sous peine d'obtenir des « effets spéciaux » non désirés.

Coordonnées homogènes

Comme nous l'avons laissé entendre, nous utiliserons beaucoup d'autres transformations qui seront généralement des combinaisons de translations et de rotations (et parfois d'homothéties). Comment obtenir la formule de telles transformations ? Comment implanter ces transformations sous forme de programme ? Comment retenir les transformations appliquées à un objet ?

Heureusement pour nous, la solution existe déjà depuis plusieurs centaines d'années, il s'agit du calcul matriciel. Il nous suffit ici de savoir que les transformations qui nous intéressent peuvent être représentées sous forme de matrices et que les matrices possèdent plusieurs opérations permettant de les additionner et de les multiplier (par d'autres matrices ou par des vecteurs). En appliquant bien les règles, nous obtenons une matrice qui correspond à la transformation résultante. Exactement ce dont nous avons besoin.

Certaines transformations se représentaient déjà très bien sous forme matricielle, notez qu'il s'agit de multiplications de matrices (voir plus loin en 1.3) :

Rotation :

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \times \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

Homothétie :

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \times \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Par contre, il est impossible de représenter une translation au moyen d'une matrice 2×2 . Au lieu de renoncer, les mathématiciens de l'époque (et les physiciens) se sont acharnés et ont découvert qu'en prenant des matrices 3×3 , ils pouvaient y arriver. Attention, même s'il s'agit de matrices 3×3 , nous sommes encore en 2D.

Note

Les quaternions pourront aussi être utilisés pour certaines opérations au lieu des matrices

Nos formules matricielles deviennent donc :

Rotation :

$$\begin{bmatrix} x' & y' & w \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Homothétie :

$$\begin{bmatrix} x' & y' & w \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translation :

$$\begin{bmatrix} x' & y' & w \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{bmatrix}$$

IMPORTANT

Vous avez sûrement remarqué que nous multiplions un **vecteur ligne** par une matrice de transformation. C'est le modèle utilisé par DirectX et c'est aussi le modèle le plus fréquemment utilisé en développement de jeux vidéo. En infographie « traditionnelle » et avec OpenGL, la notation utilisée sera plutôt du type:

$$\begin{bmatrix} x' & y' & w \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Pour beaucoup de traitements, vous pourrez choisir le modèle que vous désirez (à condition de ne pas les mélanger...), mais avant de transmettre une matrice de transformation à une fonction graphique, vérifiez la notation qu'elle s'attend à recevoir. Il est évidemment plus « payant » de choisir une notation correspondant à notre système graphique plutôt que de programmer une transposition.

Théorie

A.2 Opérations sur les matrices

Le calcul matriciel est une branche très élaborée des mathématiques. Toutefois, pour débuter en infographie, nous commencerons par certaines opérations de base pour en ajouter d'autres par la suite.

D'abord, il existe plusieurs sortes de matrices, mais nous nous limiterons aux matrices carrées (3×3 et 4×4) ainsi qu'aux vecteurs lignes (1×3 et 1×4) et aux vecteurs colonnes (3×1 et 4×1).

Addition de matrices

Théoriquement, nous pouvons additionner deux matrices de mêmes dimensions. Mais en pratique, nous utiliserons surtout l'addition de vecteurs.

A, B et C sont des vecteurs 1×3

$$\begin{aligned} A &= [a_1 \ a_2 \ a_3] \\ B &= [b_1 \ b_2 \ b_3] \end{aligned}$$

$$C = A + B = [a_1 \ a_2 \ a_3] + [b_1 \ b_2 \ b_3] = [a_1 + b_1 \ a_2 + b_2 \ a_3 + b_3]$$

Multiplication de matrices

La multiplication de matrices aussi appelée produit matriciel est dénotée comme suit:

M1 * M2 ou M1 x M2 ou M1 M2

Cette opération est probablement l'une des plus intéressantes pour nous puisque la multiplication de deux matrices de transformation nous donne la transformation combinée c.-à-d. une transformation qui équivaut à l'exécution consécutive des deux transformations originales.

La multiplication de matrices obéit à des règles strictes et doit être exécutée de la façon suivante pour une matrice carrée multipliée par une matrice carrée:

$$\begin{aligned} A \times B &= \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \\ &= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix} \end{aligned}$$

Un cas courant est la multiplication d'un vecteur par une matrice carrée, comme suit:

$$A \times B = [a_1 \ a_2 \ a_3] \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$
$$A \times B = [a_1b_{11} + a_2b_{21} + a_3b_{31} \quad a_1b_{12} + a_2b_{22} + a_3b_{32} \quad a_1b_{13} + a_2b_{23} + a_3b_{33}]$$

Cette multiplication est utilisée pour toutes les transformations appliquées à des points.

*Exercices***A.3 Opérations en 2D**

1. Représenter sous forme homogène (par une matrice) et selon la notation choisie par DirectX les transformations suivantes:

- a) $T = [5 \ 7]$
- b) $T = [0 \ 1]$
- c) R où $\theta = 90^\circ$
- d) R où $\theta = 0^\circ$ (que remarquez-vous?)
- e) H où $s_x = 2$ et $s_y = 5$

2. Représenter sous forme homogène (par une matrice) et selon la notation choisie par DirectX la combinaison des opérations suivantes :

$$T1 = [2 \ 2] \quad R \text{ où } \theta = 90^\circ \text{ et } T2 = [4 \ 1]$$

3. Effectuer les multiplications de matrices suivantes :

$$\text{a)} \quad [4 \ 3 \ 1] \times \begin{bmatrix} 3 & 5 & 6 \\ 3 & 4 & 5 \\ 1 & 3 & 1 \end{bmatrix}$$

$$\text{b)} \quad \begin{bmatrix} 4 & 1 & 2 \\ 2 & 1 & 1 \\ 2 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 4 & 2 & 1 \\ 4 & 4 & 3 \\ 2 & 1 & 3 \end{bmatrix}$$

$$\text{c)} \quad \begin{bmatrix} 3 & 3 & 1 \\ 5 & 4 & 3 \\ 6 & 5 & 1 \end{bmatrix} \times \begin{bmatrix} 4 \\ 3 \\ 1 \end{bmatrix} \quad \text{Que remarquez-vous ? (comparez avec a)}$$

$$\text{d)} \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A.4 Les transformations en 3D

Dans nos applications 3D, nous utiliserons les transformations 3D pour effectuer les opérations suivantes:

- Exprimer la position d'un objet par rapport à un autre objet ou par rapport à l'origine de la scène.
- Tourner et changer les dimensions d'un objet.
- Changer les positions et les directions.

On se rappelle qu'en réalité, les transformations sont appliquées aux points (sommets) de l'objet. Ce qui veut dire qu'une transformation erronée pourra transformer notre objet en quelque chose de très différent de l'objet original.

Les transformations sont appliquées en multipliant le vecteur représentant le point par la matrice correspondant à la transformation. Par exemple, pour produire le nouveau point (x' , y' , z') à partir du point (x , y , z).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

Le calcul a lieu de la façon suivante:

$$x' = (x \times M_{11}) + (y \times M_{21}) + (z \times M_{31}) + (1 \times M_{41})$$

$$y' = (x \times M_{12}) + (y \times M_{22}) + (z \times M_{32}) + (1 \times M_{42})$$

$$z' = (x \times M_{13}) + (y \times M_{23}) + (z \times M_{33}) + (1 \times M_{43})$$

Ce qui est identique à la méthode démontrée dans les sections précédentes.

Les transformations les plus importantes sont évidemment les rotations et les translations. Mais il faut noter que d'autres transformations apparaîtront pour, entre autres, réaliser les projections. Il est possible de concaténer deux matrices pour produire le même effet que si nous les avions exécutées une à la suite de l'autre. La concaténation est réalisée au moyen de la multiplication de matrices.

Avec Direct3D, nous utiliserons souvent un tableau 4 X 4 déclaré par le type **XMMATRIX**. Ce type est utilisé dans plusieurs fonctions Direct3D. La bibliothèque DirectXMath (XM) possède aussi des extensions très utiles en programmation orientée objet. Voici un exemple d'homothétie qui double les dimensions sur les trois axes:

```
XMMATRIX homothetie1 = XMMATRIX(
    2.0f,    0.0f,    0.0f,    0.0f,
    0.0f,    2.0f,    0.0f,    0.0f,
    0.0f,    0.0f,    2.0f,    0.0f,
    0.0f,    0.0f,    0.0f,    1.0f
);
```

Translation

La transformation suivante effectue la translation du point (x, y, z) vers un nouveau point (x', y', z') .

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Nous pouvons créer une matrice de translation en la déclarant en C++ de la façon suivante (une translation de $(5, 2, 2)$):

```
XMMATRIX trans1 = XMMATRIX(
    1.0f,    0.0f,    0.0f,    0.0f,
    0.0f,    1.0f,    0.0f,    0.0f,
    0.0f,    0.0f,    1.0f,    0.0f,
    5.0f,    2.0f,    2.0f,    1.0f
);
```

DirectXMath contient aussi la fonction **XMMatrixTranslation** qui permet de construire une matrice de translation de la façon suivante:

```
D3DXMATRIX trans2;
trans2 = XMMatrixTranslation(5.0f, 2.0f, 2.0f);
```

Rotation

Les transformations décrites ici sont pour les systèmes de coordonnées **main-gauche** et selon une multiplication d'un vecteur par une matrice.

La transformation suivante effectue une rotation autour de l'axe des X, produisant un nouveau point (x' , y' , z').

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La transformation suivante effectue une rotation autour de l'axe des Y.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La transformation suivante effectue une rotation autour de l'axe des Z.

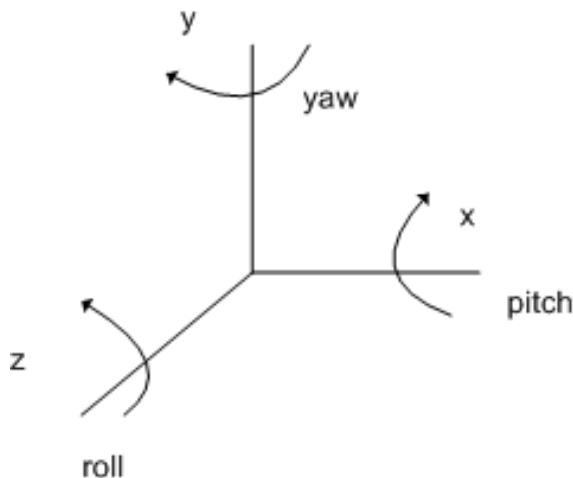
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dans ces exemples, la lettre grecque thêta (θ) corresponds à l'angle de rotation, en radians. Les angles sont mesurés dans le sens contraire des aiguilles d'une montre (antihoraire) en regardant de l'origine vers l'axe de rotation.

La bibliothèque DirectXMath nous fournit les fonctions **XMMatrixRotationX**, **XMMatrixRotationY**, et **XMMatrixRotationZ** qui nous permettent de créer des matrices de rotation plus facilement, mais il est aussi possible de les créer nous-mêmes.

Comme vous le constatez, il faut souvent deux ou trois rotations pour obtenir certains mouvements. Ce n'est pas toujours un problème puisque l'on fait souvent correspondre l'un des axes avec la direction du mouvement, mais cela peut dans plusieurs cas augmenter le nombre d'opérations à effectuer ou rendre certaines opérations difficiles à définir.

Rotations avec les angles d'Euler



Les angles d'Euler sont le nom donné à un ensemble d'angles qui spécifient chacun une rotation autour des axes X, Y et Z qui, combinés, définissent une rotation arbitraire dans l'espace. Il s'agit d'une représentation fréquemment utilisée, car elle se manipule très simplement à l'aide d'outils tels les matrices. C'est aussi une représentation très pratique pour l'animation de certains objets via une interaction de l'utilisateur. Il suffit en effet de faire correspondre des touches ou des directions de souris à chaque axe et de faire varier les valeurs des angles en fonctions de ces interactions.

Ces angles sont spécifiés sous la forme d'un vecteur [x y z] et peuvent être stockés comme une structure de type vecteur ce qui est économique en termes d'espace de stockage.

Voici quelques exemples d'angles d'Euler :

- [0 0 0] Rotation nulle (générera la matrice identité).
- [90 0 0] Rotation de 90° autour de l'axe X
- [0 90 0] Rotation de 90° autour de l'axe Y
- [10 160 90] Rotation de 10° autour de l'axe X, 160 autour de Y et 90° autour de Z.

Le problème de cette technique est que pour calculer la position finale d'un objet dans l'espace après application de la rotation, il faut appliquer successivement (dans un ordre prédéterminé) les rotations autour de chaque axe. Cette opération est relativement coûteuse en nombre d'opérations de calcul puisqu'elle implique plusieurs opérations trigonométriques et elle peut donc entraîner plus d'erreurs de précision. Elle rend souvent les interpolations entre angles plus difficiles.

Un autre problème potentiel de cette méthode est la perte d'un degré de liberté ou *Gimbal Lock*. Comme la position finale d'un objet dépend de l'ordre des rotations suivant les trois axes, il arrive parfois que l'une des

rotations autour d'un axe se confond avec un autre axe de rotation ce qui entraîne l'impossibilité de tourner l'objet selon le troisième axe (qui est ainsi perdu).

Par exemple, supposons que l'on oriente un objet à l'aide d'angles d'Euler en appliquant dans l'ordre les rotations selon les axes X, Y puis Z. Dans ce cas, la rotation selon X se fait correctement, puis viens la rotation selon Y qui est de 90° de telle sorte que l'axe Z ainsi généré vienne se confondre avec l'axe X autour duquel on a effectué la première rotation. Dans ce cas, la troisième rotation selon l'axe Z s'effectue en réalité autour du même axe que la première rotation, c'est-à-dire l'axe X ! On a bel et bien perdu la possibilité d'effectuer une rotation selon l'axe Z. C'est ici qu'interviendront les quaternions.

Homothétie

La transformation suivante applique une homothétie sur le point (x, y, z) , résultant en un nouveau point (x', y', z') .

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La bibliothèque DirectXMath nous fournit la fonction XMMatrixScaling pour créer des matrices d'homothétie.

Opérations sur les matrices

Il est important de noter ici que la bibliothèque DirectXMath nous fournit plusieurs surcharges d'opérateurs (*extensions*) permettant d'utiliser des notations plus intéressantes pour le travail avec les matrices et les vecteurs, entre autres les opérations d'addition de matrices (et de vecteurs), de multiplications de matrices (et de vecteurs) ou de matrices par des vecteurs ont été implantées.

Pour plus de détails, regardez la documentation XMMATRIX dans la documentation du SDK de DirectX 11.

Les quaternions

Les quaternions trouvent leur principale application pour la représentation de rotations dans l'espace.

Les quaternions sont formés d'une quantité (un scalaire) et d'un vecteur. Un quaternion q est donc un nombre complexe formé de 4 composantes :

$$q = a + ib + jc + kd$$

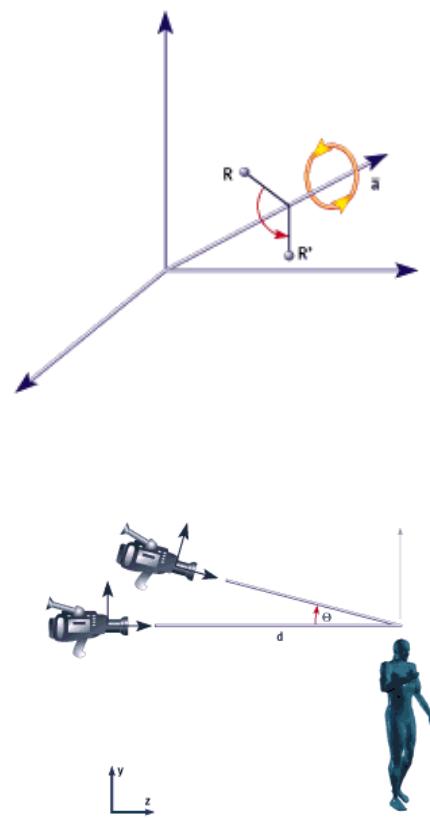
avec a, b, c, d des nombres réels et i, j, k des coefficients imaginaires.

Le principe de représentation des rotations utilisé avec les quaternions est à opposer totalement avec les angles d'Euler. Pour stocker une rotation dans un quaternion, on utilise non plus une série de rotations successives autour de chaque axe, mais un vecteur arbitraire et un angle quelconque de rotation autour de ce vecteur. La partie réelle du quaternion contenant l'angle, la partie imaginaire le vecteur 3D et le quaternion étant obligatoirement unitaire, c'est-à-dire de longueur 1.

Les quaternions furent découverts en 1843 par l'irlandais **William Rowan HAMILTON** (1805-1865). En faisant des calculs avec les nombres complexes, il s'intéressa à l'interprétation géométrique de l'addition et de la multiplication dans le plan à deux dimensions. Il se demanda si avec des nombres hypercomplexes, on ne pourrait pas obtenir des résultats analogues dans l'espace à trois dimensions. Un jour, il se rend subitement compte qu'on ne peut pas donner une structure multiplicative aux triplets de nombres réels, mais qu'on peut le faire pour les quadruplets (tout comme nos matrices 4×4). Les quaternions étaient nés.

Les quaternions offrent d'énormes avantages, le premier est la mémoire relativement faible qu'ils occupent à comparer avec celle que nécessitent les matrices (quatre fois moindre). Leur second avantage est la facilité qu'ils offrent pour la combinaison de rotations. Tout comme les matrices, les quaternions de rotation se combinent par multiplication, mais cette fois ce sont bien des rotations arbitraires qui sont combinées et le nombre d'opérations nécessaires à cette combinaison est nettement moindre que celui requis pour combiner des matrices. Une telle économie de calculs est extrêmement importante lorsque l'on travaille, par exemple, avec des animations sur des objets hiérarchiques (comme l'animation d'un squelette). De plus, la représentation *angle-plus-axe* ne souffre pas du problème du *Gimbal Lock*.

Un autre avantage des quaternions est qu'ils offrent un moyen simple et précis de réussir les interpolations entre deux rotations. Il existe en effet des techniques telles que SLERP (*Spherical Linear iNTERpolation*) qui permettent d'obtenir des interpolations de rotations extrêmement réalistes. De telles interpolations peuvent



être utilisées par exemple dans un jeu vidéo pour faire suivre un personnage par une caméra en réponse à des actions du joueur. Imaginons que le joueur se retourne d'un demi-tour en appuyant sur une touche, on calcule la rotation finale de la caméra pour la recentrer sur le personnage puis on effectue une interpolation afin que la caméra prenne sa place de manière fluide et progressive.

La bibliothèque DirectXMath nous offre aussi des opérations sur les quaternions dont entre autres la méthode SLERP.

Références

- [GPG 6] Dickheiser, Mike - **Game programming gems 6**
- [Luna 1] Luna, Frank - **Introduction to 3D Game Programming with DirectX 10**
- [Möller et Haines 1] Haines, Eric ; Möller, Tomas —**Real time rendering**
- [MSDN] Microsoft Development Network, en ligne au msdn.microsoft.com (en anglais, en français et dans plusieurs autres langues)
- [Petzold PW] Petzold, Charles – **Programming Windows**
- [Phong 1] Phong Bui-Tong - **Illumination for computer generated pictures.** 1975
- [SX2_1] Engel, Wolfgang, **ShaderX 2 Introductions and tutorials**, (Shadow mapping with Direct3D 9, Michal Valient)