Un livre de Wikilivres.

Les cartes graphiques

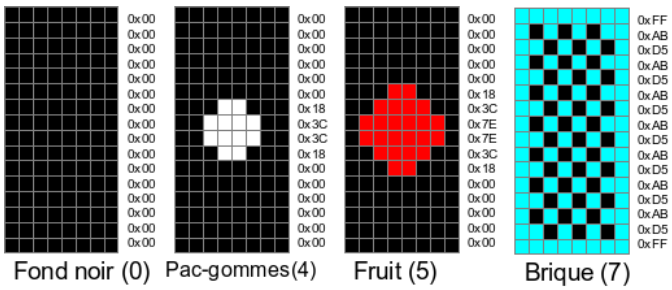
Une version à jour et éditable de ce livre est disponible sur Wikilivres,
une bibliothèque de livres pédagogiques, à l'URL :
http://fr.wikibooks.org/wiki/Les_cartes_graphiques

Les cartes d'affichage

Les cartes graphiques sont des cartes qui s'occupent de communiquer avec l'écran, pour y afficher des images. Au tout début de l'informatique, ces opérations étaient prises en charge par le processeur : celui-ci calculait l'image à afficher à l'écran, et l'envoyait pixel par pixel à l'écran, ceux-ci étant affichés immédiatement après. Cela demandait de synchroniser l'envoi des pixels avec le rafraichissement de l'écran. Pour simplifier la vie des programmeurs, les fabricants de matériel ont inventé des cartes d'affichage, ou cartes vidéo. Avec celles-ci, le processeur calcule l'image à envoyer à l'écran, et la transmet à la carte d'affichage. Celle-ci prend alors en charge son affichage à l'écran, déchargeant le processeur de cette tâche.

Cartes d'affichage en mode texte

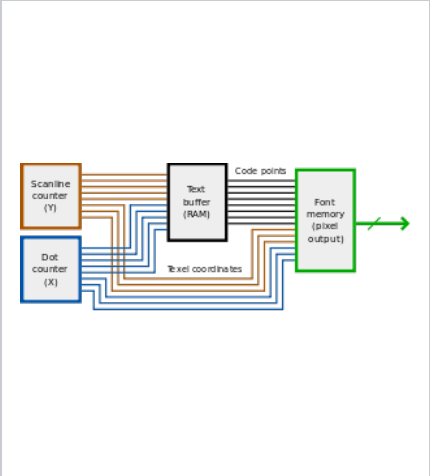
Les premières cartes graphiques fonctionnaient en **mode texte**, c'est à dire qu'elles traitaient des caractères et non des pixels. Il était impossible de modifier des pixels individuellement. Ce mode texte est toujours présent dans nos cartes graphiques actuelles et est encore utilisé par le BIOS ou par les écrans bleus de Windows. Les caractères sont des lettres, des chiffres, ou des symboles courants, même si des caractères spéciaux sont disponibles. Ceux-ci sont encodés dans un jeu de caractère spécifique (ASCII, ISO-8859, etc.). Ces caractères ont une taille fixe, que ce soit en largeur ou en hauteur. Par exemple, chaque caractère occupera 8 pixels de haut et 5 pixels de large. L'ensemble des caractères gérés par une carte graphique (y compris ceux crées par l'utilisateur) s'appelle la **table des caractères**. Certaines cartes graphiques permettent à l'utilisateur de créer ses propres caractères en modifiant cette table. La mémoire de caractères est une mémoire ROM/EEPROM. Dans cette mémoire, chaque caractère est représenté par une matrice de pixels.



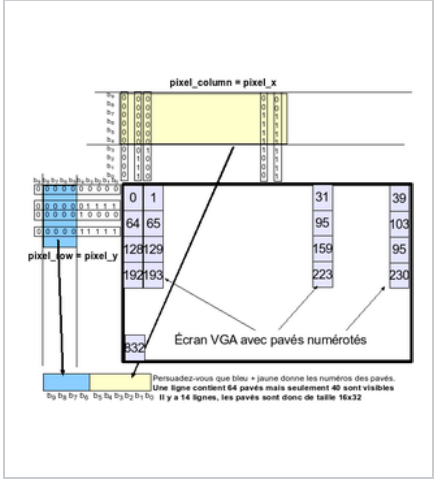
Ces caractères se voient attribuer des informations en plus de leur code ASCII. A la suite de leur code ASCII, on peut trouver un octet qui indique si le caractère clignote, sa couleur, sa luminosité, si le caractère doit être souligné, etc. Une gestion minimale de la couleur est parfois présente. La carte graphique contient un circuit chargé de gérer les **attributs des caractères** : l'ATC (Attribute Controller), aussi appelé le contrôleur d'attributs.

Le **text buffer** est une mémoire dans laquelle les caractères à afficher sont placés les uns à la suite des autres. Chaque caractère est alors stocké en mémoire sous la forme d'un code ASCII suivi d'un octet d'attributs. Le processeur va envoyer les caractères à afficher un par un, ceux-ci étant accumulés dans le text buffer au fur et à mesure de leur arrivée.

Vient ensuite le CRTC (Cathod Ray Tube Controller) ou **contrôleur de tube cathodique**. Celui-ci gère l'affichage sur l'écran proprement dit. Sur les vieux écrans CRT, les pixels sont affichés les uns après les autres, ligne par ligne, en commençant par le pixel en haut à gauche. Pour une carte en mode texte, ce contrôleur va envoyer ces pixels les uns après les autres, en utilisant la mémoire de caractères et le text buffer. Pour commencer, il se souvient du prochain pixel à afficher grâce à deux registres : un registre X pour la ligne, et un registre Y pour la colonne. Évidemment, nos deux registres de ligne et de colonne sont incrémentés régulièrement afin de passer au pixel suivant. Le CRTC va déduire à quel caractère cela correspond dans le text buffer avec quelques bits des registres X et Y, le lire et l'envoyer à la mémoire de caractères. Là, le code du caractère, et les bits restants des deux registres sont utilisés pour calculer l'adresse mémoire du pixel à afficher. Ce pixel est alors envoyé à l'écran.

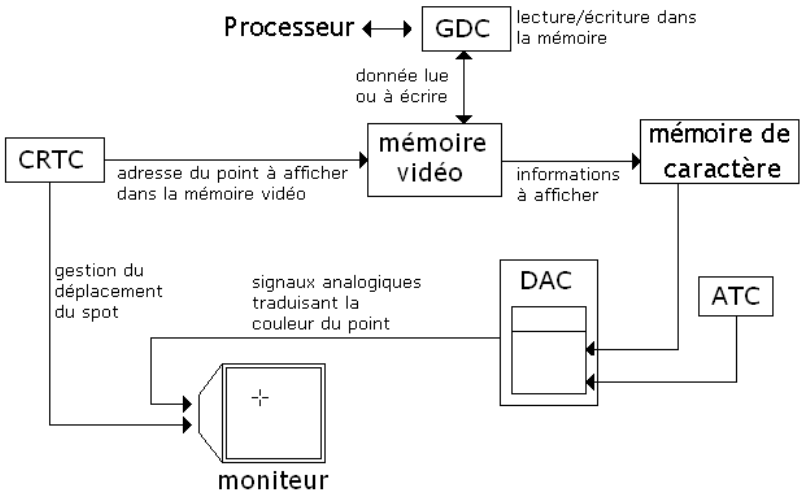


Adressage des pixels dans le text buffer grâce au CRTC.



Lien entre localisation des pixels et caractères sur l'écran.

Enfin, le pixel à afficher est envoyé à l'écran. Ceci dit, les écrans assez anciens fonctionnent en analogiques et non en binaire, ce qui demande de faire une conversion. C'est le rôle du **DAC**, un convertisseur qui traduit des données binaires en données analogiques. Sur les écrans récents, ce DAC n'existe pas : les données sont envoyées sous forme numérique à l'écran, via une interface DVI ou autre, et sont automatiquement gérées par l'écran.

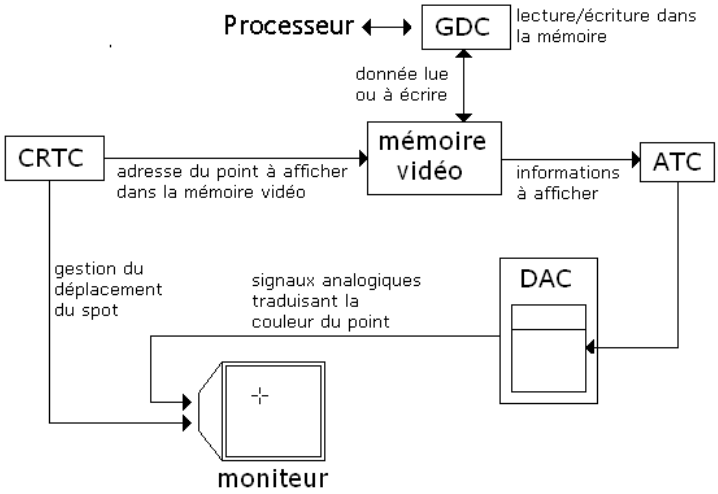


Cartes d'affichage en mode graphique

Les cartes en mode texte ont rapidement étés remplacées par des cartes graphiques capables de colorier chaque pixel de l'écran individuellement. Il s'agit d'une avancée énorme, qui permet beaucoup plus de flexibilité dans l'affichage. Ces cartes graphiques étaient conçues avec des composants assez similaires aux composants des cartes graphiques à mode texte. Divers composants sont toutefois modifiés.

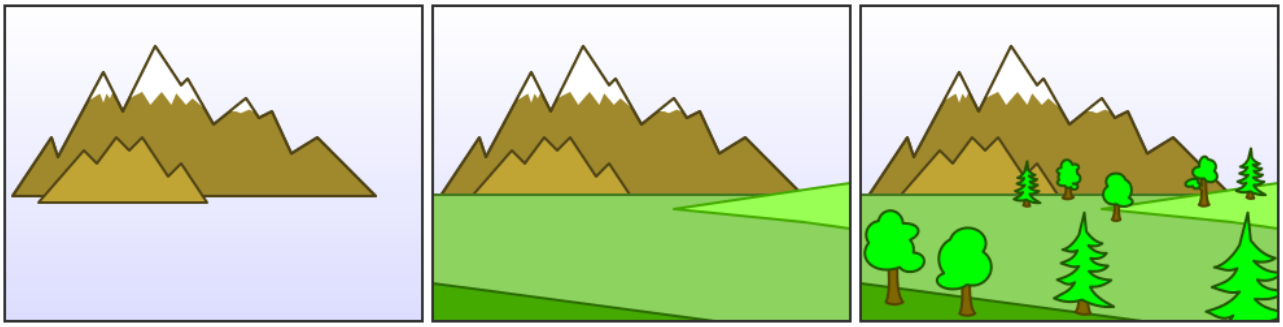
- La mémoire de caractères a évidemment disparu.
- La mémoire vidéo stocke une image à afficher à l'écran, et non des caractères : on appelle cette mémoire vidéo le **Frame Buffer**.
- Le CRTC peut gérer différentes résolutions, quelques registres permettant de configurer la résolution voulue.

La couleur fait son apparition. Toutefois, ces cartes graphiques codaient ces couleurs sur 4 bits, 8 bits, à la rigueur 16. Dans ces conditions, les couleurs n'étaient pas encodées au format RGB habituel : chaque couleur se voyait attribuer un numéro prédéterminé lors de la conception de la carte graphique. Vu que les écrans ne gèrent que des données au format RGB, ces cartes graphiques devaient effectuer la conversion en RGB avec un circuit, la **Color Look-up Table**. Dans le cas le plus simple, il s'agissait d'une ROM qui mémorisait la couleur RGB pour chaque numéro envoyé en adresse. Dans d'autres cas, cette mémoire était une RAM, ce qui permettait de modifier la palette au fur et à mesure : on pouvait ainsi changer les couleurs de la palette d'une application à l'autre sans aucun problème. Cette Color Look-up Table était alors fusionnée avec le DAC, et formait ce qu'on appelait le **RAMDAC**.


Processeur
GDC
lecture/écriture dans la mémoire
donnée lue ou à écrire
CRTC
adresse du point à afficher dans la mémoire vidéo
mémoire vidéo
informations à afficher
ATC
gestion du déplacement du spot
signaux analogiques traduisant la couleur du point
DAC
moniteur

Les cartes accélératrices 2D

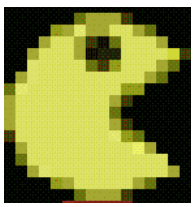
Avec l'arrivée des interfaces graphiques (celles des systèmes d'exploitation, notamment) et des jeux vidéo 2D, les cartes graphiques ont pu s'adapter. Les cartes graphiques 2D ont d'abord commencé par accélérer le tracé et coloriage de figures géométriques simples, tels les lignes, segments, cercles et ellipses, afin d'accélérer les premières interfaces graphiques. Par la suite, diverses techniques d'accélération de rendu 2D se sont fait jour. La base d'un rendu en 2D est de superposer des images 2D précalculées les unes au-dessus des autres. Par exemple, on peut avoir une image pour l'arrière plan (le décor), une image pour le monstre qui vous fonce dessus, une image pour le dessin de votre personnage, etc. Ces images sont appelées des **sprites**. Ces images sont superposées les unes au-dessus des autres, au bon endroit sur l'écran. Cette superposition se traduit par une copie des pixels de l'image aux bons endroits dans la mémoire, chaque sprite étant copié dans la portion de mémoire qui contient l'arrière plan. Le rendu des sprites doit s'effectuer de l'image la plus profonde (l'arrière-plan), vers les plus proches (les sprites qui se superposent sur les autres) : on parle d'**algorithme du peintre**. Ce genre de copie arrive aussi lorsqu'on doit scroller, ou qu'un objet 2D se déplace sur l'écran. Ces techniques ne sont pas utiles que pour les jeux vidéo, mais peuvent aussi servir à accélérer le rendu d'une interface graphique. Après tout, les lettres, les fenêtres d'une application ou le curseur de souris sont techniquement des sprites qui sont superposés les uns au-dessus des autres. C'est ainsi que les cartes graphiques actuelles supportent des techniques d'accélération du rendu des polices d'écriture, une accélération du scrolling ou encore un support matériel du curseur de la souris, toute dérivées des techniques d'accélération des sprites.



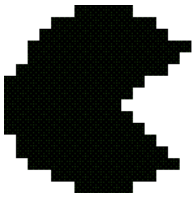
Blitter

Certaines cartes 2D ont introduit un composant pour accélérer ces copies : le **blitter**. Sans blitter, les copies étaient donc à la charge du processeur, qui devait déplacer lui-même les données en mémoire. Pour ce faire, un petit morceau de programme répétait en boucle une série d'instructions pour copier les données pixel par pixel. Le blitter est conçu pour ce genre de tâches, sauf qu'il n'utilise pas le processeur. Ceci dit, un blitter possède d'autres fonctionnalités. Il peut effectuer une opération bit à bit entre les données à copier et une données fournie par le programmeur.

Pour voir à quoi cela peut servir, reprenons notre exemple du jeu 2D, basé sur une superposition d'images. Les images des différents personnages sont souvent des images rectangulaires. Par exemple, l'image correspondant à notre bon vieux pacman ressemblerait à celle-ci. Évidemment, cette image s'interface mal avec l'arrière-plan. Avec un arrière-plan blanc, les parties noires de l'image du pacman se verraient à l'écran.



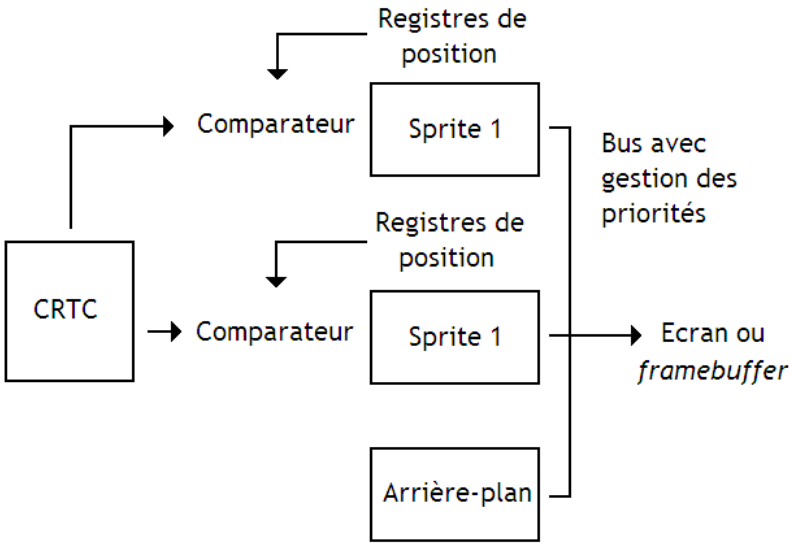
L'idéal serait de ne pas toucher à l'arrière-plan sur les pixels noirs de pacman, et de ne modifier l'arrière-plan que pour les pixels jaunes. Ceci est possible en fournissant un masque, une image qui indique quels pixels modifier lors d'un transfert, et quels sont ceux qui ne doivent pas changer. Grâce à ce masque, le blitter sait quels pixels modifier. Le blitter prend l'image du pacman, le morceau de l'arrière-plan auquel on superpose pacman, et le masque. Pour chaque pixel, il effectue l'opération suivante : ((arrière-plan) AND (masque)) OR (image de pacman). Au final, l'image finale est bel et bien celle qu'on attend.



Accélération matérielle des sprites

Avec d'autres cartes 2D, les sprites ne sont pas copiés sur un arrière-plan préxistant. À la place, c'est la carte graphique qui décidera d'afficher les pixels de l'arrière-plan ou du sprite pendant l'envoi des pixels à l'écran, lors du balayage

effectué par le CRTC. Pour cela, les sprites sont stockés dans des registres ou des RAM. Pour chaque RAM /sprite, on trouve trois registres permettant de mémoriser la position du sprite à l'écran : un pour sa coordonnée X, un autre pour sa coordonnée Y, et un autre pour sa profondeur (pour savoir celui qui est superposé au-dessus de tous les autres). Lorsque le CRTC demande à afficher le pixel à la position (X , Y), chaque triplet de registres de position est comparé à la position X,Y envoyée par le CRTC. Si aucun sprite ne correspond, les mémoires des sprites sont déconnectées du bus et le pixel affiché est celui de l'arrière-plan. Dans le cas contraire, la RAM du sprite est connectée sur le bus et son contenu est envoyé au RAMDAC. Si plusieurs sprites doivent s'afficher en même temps, le bus choisit celui dans la profondeur est la plus faible (celui superposé au-dessus de tous les autres).



Cette technique a autrefois été utilisée sur les anciennes bornes d'arcade, ainsi que sur certaines console de jeu bon assez anciennes. Mais de nos jours, elle est aussi présente dans les cartes graphiques actuelles dans un cadre particulièrement spécialisé : la prise en charge du curseur de la souris, ou le rendu de certaines polices d'écritures ! Les cartes graphiques contiennent un ou plusieurs sprites, qui représentent chacun un curseur de souris, et deux registres, qui stockent les coordonnées x et y du curseur. Ainsi, pas besoin de redessiner l'image à envoyer à l'écran à chaque fois que l'on bouge la souris : il suffit de modifier le contenu des deux registres, et la carte graphique place le curseur sur l'écran automatiquement. Pour en avoir la preuve, testez une nouvelle machine sur laquelle les drivers ne sont pas installés, et bougez le curseur : effet lag garantit !

Les cartes accélératrices 3D

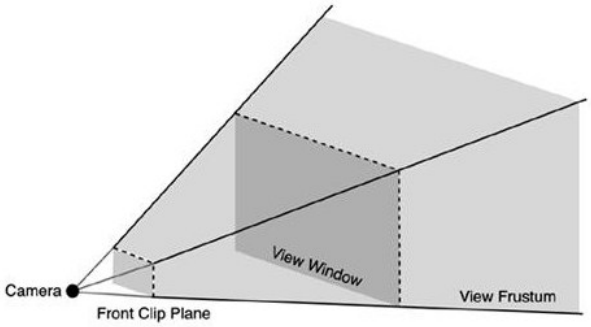
Le premier jeu à utiliser de la "vraie" 3D fût le jeu Quake, premier du nom. Et depuis sa sortie, presque tous les jeux vidéos un tant soit peu crédibles utilisent de la 3D. Face à la prolifération de ces jeux vidéos en 3D, les fabricants de cartes graphiques se sont adaptés et ont inventé des cartes capables d'accélérer les calculs effectués pour rendre une scène en 3D : les cartes accélératrices 3D.

Concepts de base du rendu 3D

Une **scène 3D** est composée d'un espace en trois dimensions, dans laquelle le moteur physique du jeu vidéo place des objets et les fait bouger. Cette scène est, en première approche, un simple parallélogramme. Un des coins de ce parallélogramme sert de système de coordonnées : il est à la position (0, 0, 0), et les axes partent de ce point en suivant les arêtes. Les objets seront placés à des coordonnées bien précises dans ce parallélogramme.

Dans toute scène 3D, on trouve une **caméra**, qui représente les yeux du joueur. Cette caméra est définie par :

- une position ;
- par la direction du regard (un vecteur) ;
- le champ de vision (un angle) ;
- un plan qui représente l'écran du joueur ;
- et un plan au-delà duquel on ne voit plus les objets.



Ces autres objets sont composés de formes de base, combinées les unes aux autres pour former des objets complexes. Ces formes géométriques peuvent être des triangles, des carrés, des courbes de Béziers, etc. Dans la majorité des jeux vidéos actuels, nos objets sont modélisés par un assemblage de triangles collés les uns aux autres. Ces triangles sont définis par leurs sommets, qui sont appelés des **vertices**. Chaque vertice possède trois coordonnées, qui indiquent où se situe le sommet dans la scène 3D : abscisse, ordonnée, profondeur.

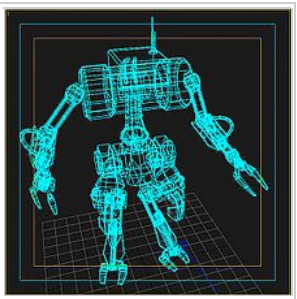


Illustration des vertices pour un modèle 3D complexe.

Pour rajouter de la couleur, ces objets sont recouverts par des **textures**, des images qui servent de papier peint à un objet. Un objet géométrique est recouvert par une ou plusieurs textures, qui permettent de le colorier ou de lui appliquer du relief.

Pipeline graphique

Depuis un bon moment, les jeux vidéos utilisent une technique de rendu spécifique : la rasterization. Celle-ci calcule une scène 3D intégralement, avant de faire des transformations pour n'afficher que ce qu'il faut à l'écran. Le calcul de l'image finale passe par diverses étapes bien séparées, l'ensemble étant appelé le **pipeline graphique**. Le cas le plus simple ne demandant que quatre étapes :

- une étape de **traitement des vertices** ;
- une étape de **rasterization**, qui va déterminer quelle partie de l'image 3D s'affiche à l'écran, et qui attribue chaque vertice à un pixel donné de l'écran ;
- une étape de **texturing** et de traitement des pixels ;
- une étape d'**enregistrement** des données calculées en mémoire.

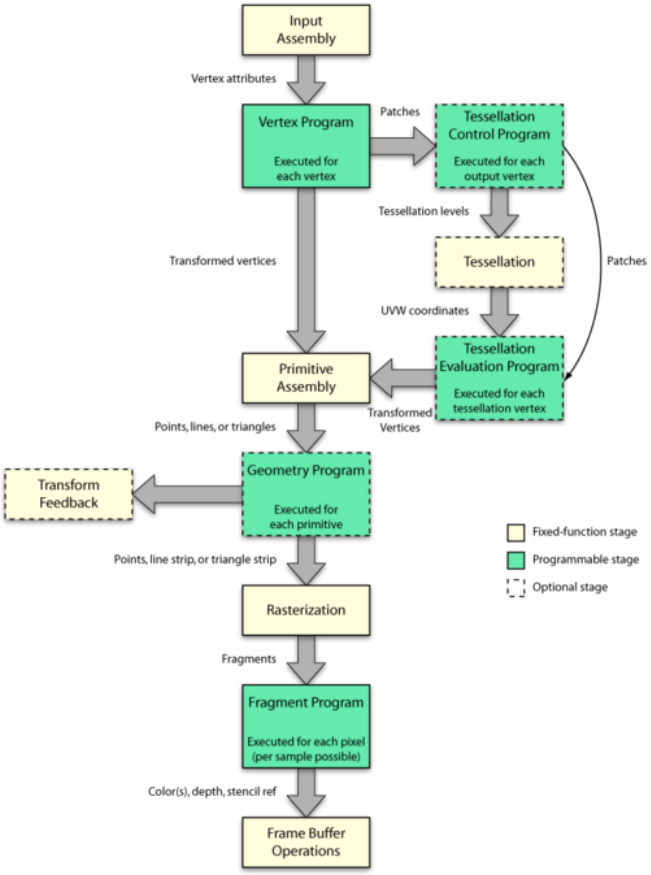
Dans certains cas, des traitements supplémentaires sont ajoutés. Par exemple, les cartes graphiques modernes supportent une étape en plus, qui permet de rajouter de la géométrie : l'étape de tesselation. Cela permet de déformer les objets ou d'augmenter leur réalisme.

Traitement des vertices

La première étape place les objets au bon endroit dans la scène 3D. Lors de la modélisation d'un objet, celui-ci est encastré dans un cube : un sommet du cube possède la coordonnée (0, 0, 0), et les vertices de l'objet sont définies à partir de celui-ci. Pour placer l'objet dans la scène, il faut tenir compte de sa localisation, calculée par le moteur physique : si le moteur physique a décrété que l'objet est à l'endroit de coordonnées (50, 250, 500), toutes les coordonnées des vertices de l'objet doivent être modifiées. Pendant cette étape, l'objet peut subir une translation, une rotation, ou un gonflement/dégonflement (on peut augmenter ou diminuer sa taille). C'est la première étape de calcul : l'étape de **transformation**.

Ensuite, les vertices sont éclairées dans une phase de **lightning**. Chaque vertice se voit attribuer une couleur, qui définit son niveau de luminosité : est-ce que la vertice est fortement éclairée ou est-elle dans l'ombre ?

Vient ensuite une phase de **traitement de la géométrie**, où les vertices sont assemblées en triangles, points, ou lignes, voire en polygones. Ces formes géométriques de base sont ensuite traitées telles quelles par la carte graphique. Sur les cartes graphiques récentes, cette étape peut être gérée par le programmeur : il peut programmer les divers traitements à effectuer lui-même.



Rasterization

Vient ensuite la traduction des formes (triangles) rendues dans une scène 3D en un affichage à l'écran. Cette étape de rasterization va projeter l'image visible sur notre caméra. Et cela nécessite de faire quelques calculs. Tout d'abord, la scène 3D va devoir passer par une phase de **clipping** : les triangles qui ne sont pas visibles depuis la caméras sont oubliés. Ensuite, ils passent par une phase de **culling**, qui élimine les pixels cachés par un objet géométrique. Enfin, chaque pixel de l'écran se voit attribuer un ou plusieurs triangle(s). Cela signifie que sur le pixel en question, c'est le triangle attribué au pixel qui s'affichera. C'est lors de cette phase de **rasterisation** que la perspective est gérée, en fonction de la position de la caméra.

Pixels et textures

À la suite de cela, les textures sont appliquées sur la géométrie. La carte graphique sait à quel triangle correspond chaque pixel et peut donc colorier le pixel en question en fonction de la couleur de la texture appliquée sur la géométrie. C'est la phase de **Texturing**. Sur les cartes graphiques récentes, cette étape peut être gérée par le programmeur : il peut programmer les divers traitements à effectuer lui-même. En plus de cela, les pixels de l'écran peuvent subir des traitements divers et variés avant d'être enregistrés et affichés à l'écran. Un effet de brouillard peut être ajouté, des tests de visibilité sont effectués, l'antialiasing est ajouté, etc.

Architecture d'une carte 3D

Avant l'invention des cartes graphiques, toutes ces étapes étaient réalisées par le processeur : il calculait l'image à afficher, et l'envoyait à une carte d'affichage 2D. Au fil du temps, de nombreux circuits furent ajoutés, afin de déporter un maximum de calculs vers la carte vidéo. Pour déléguer ses calculs à la carte 3D, les applications pourraient communiquer directement avec la carte graphique, sans prendre en compte toute contrainte de compatibilité. Pour éviter cela, les concepteurs de systèmes d'exploitations et de cartes graphiques ont inventé des API 3D, des bibliothèques qui fournissent des fonctions que l'application pourra exécuter au besoin. De nos jours, les plus connues sont DirectX, et OpenGL. Les fonctions de ces APIs vont préparer des données à envoyer à la carte graphique, avant que le pilote s'occupe des les communiquer à la carte graphique. Un driver de carte graphique gère la mémoire de la carte graphique : où placer les textures, les vertices, et les différents buffers de rendu. Le pilote de carte graphique est aussi chargé de traduire les shaders, écrits dans un langage de programmation comme le HLSL ou le GLSL, en code machine.

Processeur de commandes

Tout les traitements que la carte graphique doit effectuer, qu'il s'agisse de rendu 2D, de calculs 2D, du décodage matérielle d'un flux vidéo, ou de calculs généralistes, sont envoyés par un programme, le pilote de la carte graphique, sous la forme de commandes. L'envoi des données à la carte graphique ne se fait pas immédiatement : il arrive que la carte graphique n'ait pas fini de traiter les données de l'envoi précédent. Il faut alors faire patienter les données tant que la carte graphique est occupée. Les pilotes de la carte graphique vont les mettre en attente dans une file (une zone de mémoire dans laquelle on stocke des données dans l'ordre d'ajout) : le **tampon de commandes**. Ensuite, ces commandes

sont interprétées par un circuit spécialisé : le **processeur de commandes**. Celui-ci est chargé de piloter les circuits de la carte graphique.

Sur les cartes graphiques modernes, le processeur de commandes peut démarrer une commande avant que les précédentes soient terminées. Par exemple, il est possible d'exécuter une commande ne requérant que des calculs, en même temps qu'une commande qui ne fait que faire des copies en mémoire. Toutefois, cette parallélisation du processeur de commandes a un désavantage : celui-ci doit gérer les synchronisations entre commandes. Par exemple, imaginons que Direct X décide d'allouer et de libérer de la mémoire vidéo. Direct X et Open Gl ne savent pas quand le rendu de l'image précédente se termine. Comment éviter d'enlever une texture tant que les commandes qui l'utilisent ne sont pas terminées ? Ce problème ne se limite pas aux textures, mais vaut pour tout ce qui est placé en mémoire vidéo. De manière générale, Direct X et Open Gl doivent savoir quand une commande se termine. Un moyen pour éviter tout problème serait d'intégrer les données nécessaire à l'exécution d'une commande dans celle-ci : par exemple, on pourrait copier les textures nécessaires dans chacune des commandes. Mais cela gâche de la mémoire, et ralentit le rendu à cause des copies de textures. Les cartes graphiques récentes incorporent des **commandes de synchronisation** : les fences. Ces fences vont empêcher le démarrage d'une nouvelle commande tant que la carte graphique n'a pas fini de traiter toutes les commandes qui précèdent la fence. Pour gérer ces fences, le command buffer contient des registres, qui permettent au processeur de savoir où la carte graphique en est dans l'exécution de la commande.

Un autre problème provient du fait que les commandes se partagent souvent des données, et que de nombreuses commandes différentes peuvent s'exécuter en même temps. Or, si une commande veut modifier les données utilisées par une autre commande, il faut que l'ordre des commandes soit maintenu : la commande la plus récente ne doit pas modifier les données utilisées par une commande plus ancienne. Pour éviter cela, les cartes graphiques ont introduit des **instructions de sémaphore**, qui permettent à une commande de bloquer tant qu'une ressource (une texture) est utilisée par une autre commande.

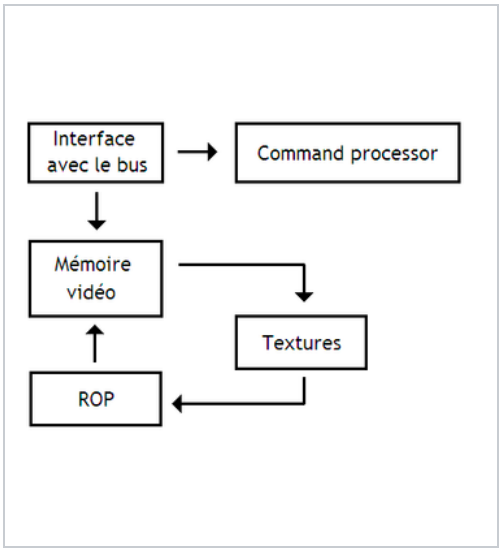
Circuits fixes

Toute carte graphique contient des circuits, aussi appelés unités, qui prennent en charge une étape du pipeline graphique. Entre les différentes unités, on trouve souvent des mémoires pour mettre en attente les vertices ou les pixels, au cas où une unité est trop occupée. Pour plus d'efficacité, ces cartes graphiques possédaient parfois plusieurs unités de traitement des vertices et des pixels, ou plusieurs ROP. Dans ce cas, ces unités multiples sont précédées par un circuit qui se charge de répartir les vertex ou pixels sur chaque unités. Généralement, ces unités sont alimentées en vertex/pixels les unes après les autres (principe du round-robin).

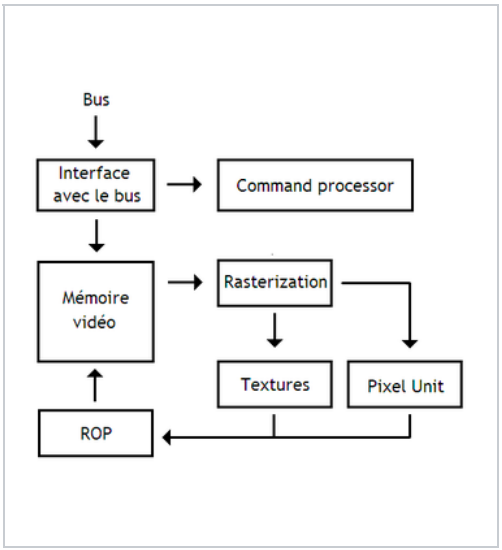
Les toutes premières cartes graphiques contenaient simplement des circuits pour gérer les textures, en plus de la mémoire RAM vidéo. Seules l'étape de texturing, quelques effets graphiques (brouillard) et l'étape d'enregistrement des pixels en mémoire étaient prises en charge par la carte graphique. Par la suite, ces cartes s'améliorèrent en ajoutant plusieurs circuits de gestion des textures, pour colorier plusieurs pixels à la fois. Cela permettait aussi d'utiliser plusieurs textures pour colorier un seul pixel : c'est ce qu'on appelle du multitexturing. Les cartes graphiques construites sur cette architecture sont très anciennes. On parle des cartes graphiques ATI rage, 3DFX Voodoo, Nvidia TNT, etc.

Les cartes suivantes ajoutèrent une gestion des étapes de rasterization directement en matériel. Les cartes ATI rage 2, les Invention de chez Rendition, et d'autres cartes graphiques supportaient ces étapes en hardware. De nos jours, ce genre d'architecture est commune chez certaines cartes graphiques intégrées dans les processeurs ou les cartes mères.

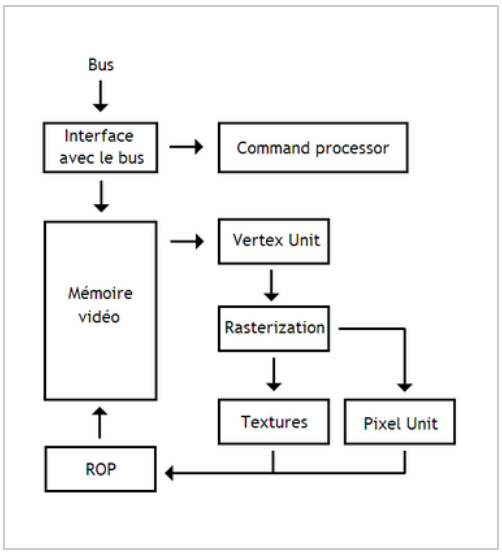
La première carte graphique capable de gérer la géométrie fût la Geforce 256, la toute première Geforce.



Carte 3D sans rasterization matérielle.



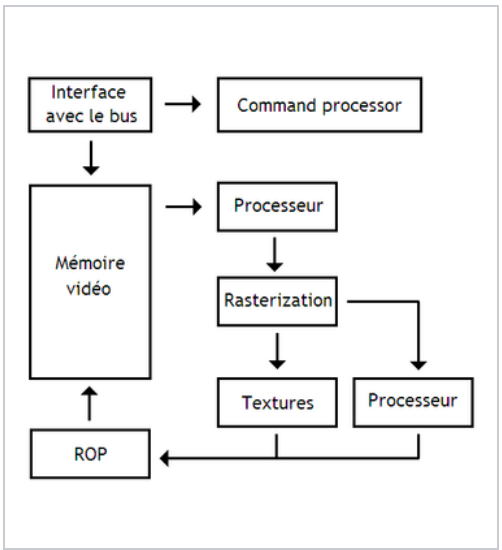
Carte 3D avec gestion de la géométrie.



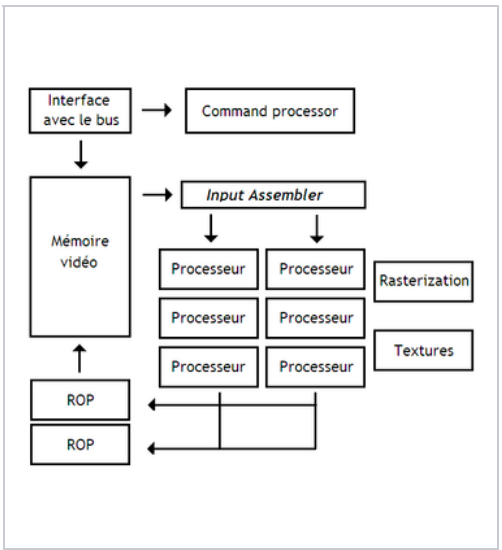
Carte 3D avec gestion de la géométrie.

Circuits programmables : vertex et pixels shaders

A partir de la Geforce 3 de Nvidia, les unités de traitement de la géométrie sont devenues programmables. Cela permet une grande flexibilité, à savoir que changer le comportement ne nécessite pas de re-câbler tout le circuit. Les unités de traitement de la géométrie deviennent donc des processeurs indépendants, capables d'exécuter des programmes appelés **Vertex Shaders**. Par la suite, l'étape de traitement des pixels est elle aussi devenue programmable. Des programmes capables de traiter des pixels, les **pixels shaders** ont fait leur apparition. Une seconde série d'unités a alors été ajoutée dans nos cartes graphiques : les processeurs de pixels shaders. Ces shaders sont écrits dans un langage de haut-niveau, le HLSL ou le GLSL, et sont traduits (compilés) par les pilotes de la carte graphique avant leur exécution. Au fil du temps, les spécifications de ces langages sont devenues de plus en plus riches et le matériel en a fait autant. Les premières cartes graphiques avaient des jeux d'instructions séparés pour les unités de vertex shader et les unités de pixel shader, et les processeurs étaient séparés. Pour donner un exemple, c'était le cas de la Geforce 6800. Cette séparation entre unités de texture et de vertices était motivée par le fait que les unités de vertice n'accédaient jamais à la mémoire, contrairement aux unités de traitement de pixels qui doivent accéder aux textures. Depuis DirectX 10, ce n'est plus le cas : le jeu d'instructions a été unifié entre les vertex shaders et les pixels shaders, ce qui fait qu'il n'y a plus de distinction entre processeurs de vertex shaders et de pixels shaders, chaque processeur pouvant traiter indifféremment l'un ou l'autre.



Carte 3D avec pixels et vertex shaders non-unifiés.

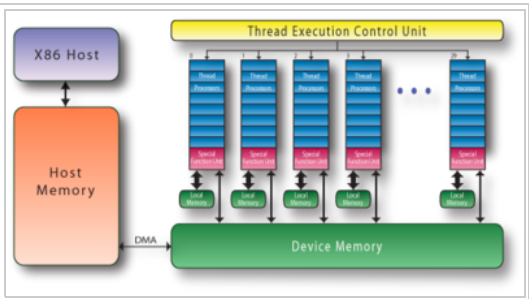


Carte 3D avec pixels et vertex shaders unfifiés.

Les processeurs de shaders contiennent un grand nombre d'unités de calcul. Il faut dire que la nature fortement parallèle des traitements de rendu 3D fait que chaque vertice ou pixel pouvant être traité indépendamment des autres, dans une unité de calcul séparée. Pour masquer les accès mémoire, les techniques d'exécution dans le désordre ou d'exécution superscalaire sont occultées au profit d'une gestion poussée du multithreading. Pour profiter au mieux des opportunités de parallélisme, une carte graphique contient de nombreux processeurs (ou tout du moins des cœurs de processeurs), qui eux-même contiennent plusieurs unités de calcul. Savoir combien de cœurs contient une carte graphique est cependant

compliqué, les services marketing gardant un certain flou sur le sujet. Il n'est pas rare que ceux-ci appellent cœurs ou processeurs de simples unités de calcul, histoire de gonfler les chiffres. Et on peut généraliser à la majorité de la terminologie utilisée par les fabricants, que ce soit pour les termes warps processor, ou autre, qui ne sont pas aisés à interpréter. D'ordinaire, ce qui est appelé processeur de thread sur une carte graphique correspond en réalité à une unité de calcul.

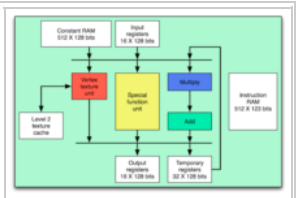
Comparaison entre l'architecture d'un processeur généraliste et d'un processeur de shaders.



Ce schéma illustre l'architecture d'un GPU en utilisant la terminologie NVIDIA. Comme on le voit, la carte graphique contient plusieurs coeurs de processeur distincts. Chacun d'entre eux contient plusieurs unités de calcul généralistes, appelées processeurs de threads, qui s'occupent de calculs simples (en bleu). D'autres calculs plus complexes sont pris en charge par une unité de calcul spécialisée (en rouge). Ces cœurs sont alimentés en instructions par un gestionnaire d'exécution, le Thread Execution Control Unit, qui répartit les différents shaders sur chaque cœur. Enfin, on voit que chaque cœur a accès à une mémoire locale dédiée, en plus d'une mémoire vidéo partagée entre tous les cœurs.

Les processeurs de shaders sont des processeurs SIMD ou VLIW, éventuellement des processeurs de flux. Les processeurs VLIW étaient autrefois utilisés sur les anciennes RADEON d'AMD, mais ne sont plus vraiment utilisées aujourd'hui, la mode étant aux processeurs SMID. Les plus récents fonctionnent comme des processeurs SIMD au niveau de l'unité de calcul, mais ce fonctionnement est masqué au niveau du jeu d'instruction. Chaque shader ne manipule qu'un seul pixel ou vertex. Mais ces instructions sont rassemblées en groupes de 16 à 32 instructions identiques qui exécutent la même instruction sur des pixels différents. En clair, ces processeurs vont découvrir à l'exécution qu'ils peuvent exécuter la même instruction sur des pixels différents et la fusionner en une instruction vectorielle : on parle de **Single Instruction Multiple Threads**. L'instruction vectorielle née de cette fusion est appelée un warp. Si un branchement ne donne pas le même résultat dans différents threads d'un même warp, le processeur se charge d'effectuer la prédication en interne : il utilise quelque chose qui fait le même travail que des instructions qui utilisent vector mask register.

La hiérarchie mémoire des GPUs est assez particulière. On y trouve souvent des caches dédiés aux textures ou aux vertices, et les GPUs récents contiennent aussi des caches L1 et L2 de faible taille. Plus rarement, on trouve des local store (scratchpad memory). Un processeur de shaders contient, en plus des registres généraux, des **registres de constantes** pour stocker les matrices servant aux différentes étapes de transformation ou d'éclairage. Les shaders peuvent écrire dans ces registres, au prix d'une perte de performance. Le choix du registre de constante à utiliser s'effectue en utilisant un **registre d'adresse de constante**.



Processeur de shader (vertex shader) d'une GeForce 6800. On voit clairement que celui-ci contient, outre les traditionnelles unités de calcul et registres temporaires, un "cache" d'instructions, des registres d'entrée et de sortie, ainsi que des registres de constante.

Le multi-GPU

Combiner plusieurs cartes graphiques dans un PC pour gagner en performances est la base des techniques dites de multi-GPU, tels le SLI et le Crossfire. Ces technologies sont surtout destinées aux jeux vidéos, même si les applications de réalité virtuelle, l'imagerie médicale haute précision ou les applications de conception par ordinateur peuvent en tirer profit. C'est ce genre de choses qui se cachent derrière les films d'animation : Pixar ou Disney ont vraiment besoin de rendre des images très complexes, avec beaucoup d'effets. Et ne parlons pas des effets spéciaux créés par ordinateur. Contrairement à ce qu'on pourrait penser, le multi-GPU n'est pas une technique récente. Pensez donc qu'en 1998, il était possible de combiner dans un même PC deux cartes graphiques Voodoo 2, de marque 3dfx (un ancien fabricant de cartes graphiques, aujourd'hui disparu). Autre exemple : dans les années 2006, le fabricant de cartes graphiques S3 avait introduit cette technologie pour ses cartes graphiques Chrome.

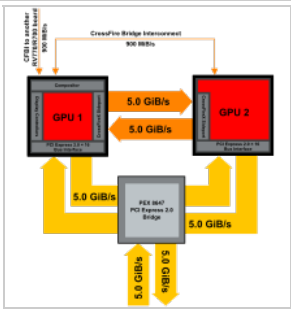


Illustration du multi-GPU où deux cartes graphiques communiquent via un lien indépendant du bus PCIExpress. On voit que le débit du lien entre les deux cartes graphique est ajouté au débit du bus PCIExpress.

Le multi-GPU peut se présenter sous plusieurs formes, la plus simple consistant à placer plusieurs GPU sur une même carte graphique. Mais il est aussi possible d'utiliser plusieurs cartes graphiques séparées, connectées à la carte mère via PCI-Express. Si les deux cartes ont besoin d'échanger des informations, les transferts passent par le bus PCI-Express ou par un connecteur qui relie les deux cartes (ce qui est souvent plus rapide). Il n'y a pas de différences de performances avec la solution utilisant des cartes séparées reliées avec un connecteur. Tout le problème des solutions multi-GPU est de répartir les calculs sur plusieurs cartes graphiques, ce qui est loin d'être chose facile. Il existe diverses techniques, chacune avec ses avantages et ses inconvénients, que nous allons aborder de suite.

Split Frame Rendering

Le **Split Frame Rendering** découpe l'image en morceaux, qui sont répartis sur des cartes graphiques différentes. Ce principe a été décliné en plusieurs versions, et nous allons les passer en revue. Nous pouvons commencer par faire la différence entre les méthodes de distribution statiques et dynamiques. Avec les méthodes statiques, la manière de découper l'image est toujours la même : celle-ci sera découpée en blocs, en lignes, en colonnes, etc; de la même façon quelque soit l'image. Avec les techniques dynamiques, le découpage s'adapte en fonction de la complexité de l'image. Nous allons commencer par aborder les méthodes statiques.

Historiquement, la première technique multi-GPU fût utilisée par les cartes graphiques Voodoo 2. Avec cette technique, chaque carte graphique calculait une ligne sur deux, la première carte rendait les lignes paires et l'autre les lignes impaires. On peut adapter la technique à un nombre arbitraire de GPU, en faisant calculer par chaque GPU une ligne sur 3, 4, 5, etc. Cette technique s'appelait le **Scan Line Interleave**. Cette technique avait un avantage certain quand la résolution des images était limitée par la quantité de mémoire vidéo, ce qui était le cas de la Voodoo 2, qui ne pouvait pas dépasser une résolution de 800 * 600. Avec le scan line interleave, les deux framebuffers des deux cartes étaient combinés en un seul framebuffer plus gros, capable de supporter des résolutions plus élevées. Cette technique a toutefois un gros défaut : l'utilisation de la mémoire vidéo n'est pas optimale. Comme vous le savez, la mémoire vidéo sert à stocker les objets géométriques de la scène à rendre, les textures, et d'autres choses encore. Avec le scan line interleave, chaque objet et texture est présent dans la mémoire vidéo de chaque carte graphique. Il faut dire que ces objets et textures sont assez grands : la carte graphique devant rendre une ligne sur deux, il est très rare qu'un objet doive être rendu totalement par une des cartes et pas l'autre. Avec d'autres techniques, cette consommation de mémoire peut être mieux gérée.

La technique du **Checker Board** découpe l'image non en lignes, mais en carrés de plusieurs pixels. Dans le cas le plus simple, les carrés ont une taille fixe, de 16 pixels de largeur par exemple. Si les carrés sont suffisamment gros, il arrive qu'ils puissent contenir totalement un objet géométrique. Dans ces conditions, une seule carte graphique devra calculer cet objet géométrique et charger ses données, qui ne seront donc pas dupliquées dans les deux cartes. Le gain en terme de mémoire peut être appréciable si les blocs sont suffisamment gros. Mais il arrive souvent qu'un objet soit à la frontière entre deux blocs : il doit donc être rendu par les deux cartes, et sera stocké dans les deux mémoires vidéos. Pour plus d'efficacité, on peut passer d'un découpage statique, où tous les carrés ont la même taille, à un découpage dynamique, dans lequel on découpe l'image en rectangles dont la longueur et la largeur varient. En faisant varier le mieux possible la taille et la longueur de ces rectangles, on peut faire en sorte qu'un maximum de rectangles contiennent totalement un objet géométrique. Le gain en terme de mémoire et de rendu peut être appréciable. Néanmoins, découper des blocs dynamiquement est très complexe, et le faire efficacement est un casse-tête pour les développeurs de drivers.

Il est aussi possible de simplement couper l'image en deux : la partie haute de l'image ira sur un GPU, et la partie basse sur l'autre. Cette technique peut être adaptée avec plusieurs GPU, en découpant l'image en autant de parties qu'il y a de GPU. Vu que de nombreux objets n'apparaissent que dans une portion de l'image, le drivers peut ainsi répartir les données de l'objet pour éviter toute duplication entre cartes graphiques. Cela demande du travail au driver, mais cela en vaut la peine, le gain en terme de mémoire étant appréciable. Le découpage de l'image peut reposer sur une technique statique : la moitié haute de l'image pour le premier GPU, et le bas pour l'autre. Ceci dit, quelques complications peuvent survenir dans certains jeux, les FPS notamment, où le bas de l'image est plus chargé que le haut. C'est en effet dans le bas de l'image qu'on trouve un sol, des murs, les ennemis, ou d'autres objets géométriques complexes texturés, alors que le haut représente le ciel ou un plafond, assez simple géométriquement et aux textures simples. Ainsi, le rendu de la partie haute sera plus rapide que celui du bas, et une des cartes 3D finira par attendre l'autre. Mieux répartir les calculs devient alors nécessaire. Pour cela, on peut choisir un découpage statique adapté, dans lequel la partie haute envoyée au premier GPU est plus grande que la partie basse. Cela peut aussi être fait dynamiquement : le découpage de l'image est alors choisi à l'exécution, et la balance entre partie haute et basse s'adapte aux circonstances. Comme cela, si vous voulez tirer une roquette sur une ennemi qui vient de prendre un jumper (vous ne jouez pas à UT ou Quake ?), vous ne subirez pas un gros coup de lag parce que le découpage statique était inadapté. Dans ce cas, c'est le driver qui gère ce découpage : il dispose

d'algorithmes plus ou moins complexes capables de déterminer assez précisément comment découper l'image au mieux. Mais il va de soit que ces algorithmes ne sont pas parfaits.

Alternate Frame Rendering

L'**alternate Frame Rendering** consiste à répartir des images complètes sur les différents GPUs. Dans sa forme la plus simple, un GPU calcule une image, et l'autre GPU calcule la suivante en parallèle. Les problèmes liés à la répartition des calculs entre cartes graphiques disparaissent alors. Cette technique est supportée par la majorité des cartes graphiques actuelles. Cette technique a été inventé par ATI, sur ses cartes graphiques Rage Fury, afin de faire concurrence à la Geforce 256. Évidemment, on retrouve un vieux problème présent dans certaines des techniques vues avant : chaque objet géométrique devra être présent dans la mémoire vidéo de chaque carte graphique, vu qu'elle devra l'afficher à l'écran. Il est donc impossible de répartir les différents objets dans les mémoires des cartes graphiques. Mais d'autres problèmes peuvent survenir.

Un des défauts de cette approche est le **micro-stuttering**. Dans des situations où le processeur est peu puissant, les temps entre deux images peuvent se mettre à varier très fortement, et d'une manière beaucoup moins imprévisible. Le nombre d'images par seconde se met à varier rapidement sur de petites périodes de temps. Alors certes, on ne parle que de quelques millisecondes, mais cela se voit à l'œil nu. Cela cause une impression de micro-saccades, que notre cerveau peut percevoir consciemment, même si le temps entre deux images est très faible. Suivant les joueurs, des différences de 10 à 20 millisecondes peuvent rendre une partie de jeu injouable. Pour diminuer l'ampleur de ce phénomène, les cartes graphiques récentes incorporent des circuits pour limiter la casse. Ceux-ci se basent sur un principe simple : pour égaliser le temps entre deux images, et éviter les variations, le mieux est d'empêcher des images de s'afficher trop tôt. Si une image a été calculée en très peu de temps, on retarde son affichage durant un moment. Le temps d'attente idéal est alors calculé en fonction de la moyenne du framerate mesuré précédemment.

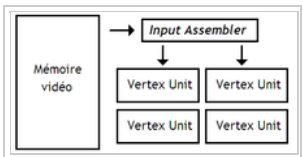
Ensuite, il arrive que deux images soient dépendantes les unes des autres : les informations nées lors du calcul d'une image peuvent devoir être réutilisées dans le calcul des images suivantes. Cela arrive quand des données géométriques traitées par la carte graphique sont enregistrées dans des textures (dans les Streams Out Buffers pour être précis), dans l'utilisation de fonctionnalités de DirectX ou d'Open GL qu'on appelle le Render To Texture, ainsi que dans quelques autres situations. Évidemment, avec l'AFR, cela pose quelques problèmes : les deux cartes doivent synchroniser leurs calculs pour éviter que l'image suivante rate des informations utiles, et soit affichée n'importe comment. Sans compter qu'en plus, les données doivent être transférées dans la mémoire du GPU qui calcule l'image suivante.

Les unités de gestion de la géométrie

Nous allons maintenant voir les circuits chargés de gérer la géométrie. Il existe deux grands types de circuits chargés de traiter la géométrie : l'input assembler charge les Sur les cartes graphiques assez anciennes, ce cache est souvent très petit, à peine 30 à à 50 sommets. Pour profiter le plus possible de ce cache, les concepteurs de jeux vidéo peuvent changer l'ordre des sommets en mémoire.s depuis la mémoire vidéo, et les circuits de traitement de vertices les traitent. Ceux-ci effectuent plusieurs traitements, qui peuvent être synthétisés en trois grandes étapes.

- La première étape de traitement de la géométrie consiste à placer les objets au bon endroit dans la scène 3D. Lors de la modélisation d'un objet, celui-ci est encastré dans un cube : un sommet du cube possède la coordonnée (0, 0, 0), et les vertices de l'objet sont définies à partir de celui-ci. Pour placer l'objet dans la scène, il faut tenir compte de sa localisation, calculée par le moteur physique : si le moteur physique a décrété que l'objet est à l'endroit de coordonnées (50, 250, 500), toutes les coordonnées des sommets de l'objet doivent être modifiées. Pendant cette étape, l'objet peut subir une translation, une rotation, ou un gonflement/dégonflement (on peut augmenter ou diminuer sa taille). C'est la première étape de calcul : l'**étape de transformation**.
- Ensuite, les sommets sont éclairées dans une **phase de lightning**. Chaque Sur les cartes graphiques assez anciennes, ce cache est souvent très petit, à peine 30 à à 50 sommets. Pour profiter le plus possible de ce cache, les concepteurs de jeux vidéo peuvent changer l'ordre des sommets en mémoire. se voit attribuer une couleur, qui définit son niveau de luminosité : est-ce que le sommet est fortement éclairée ou est-elle dans l'ombre ?
- Vient ensuite une **phase de traitement de la géométrie**, où les sommets sont assemblés en triangles, points, lignes, en polygones. Ces formes géométriques de base sont ensuite traitées telles quelles par la carte graphique. Sur les cartes graphiques récentes, cette étape peut être gérée par le programmeur : il peut programmer les divers traitements à effectuer lui-même.

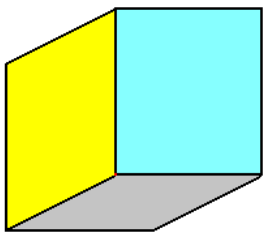
Input assembler



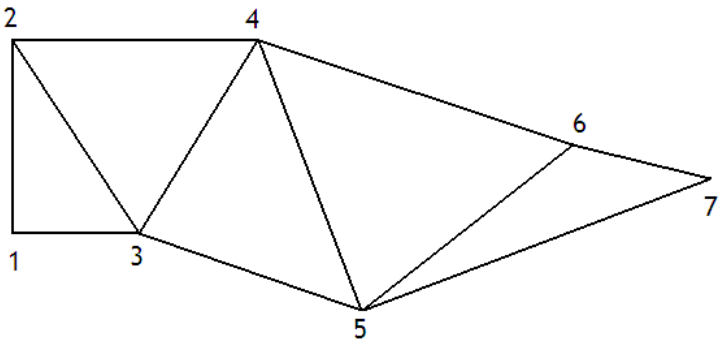
Input assembler

Avant leur traitement, les sommets sont stockées dans un tableau en mémoire vidéo : le **tampon de sommets**. L'input assembler va charger les sommets dans les unités de traitement des sommets. Pour ce faire, il a besoin d'informations mémorisées dans des registres, à savoir l'adresse du tampon de sommets en mémoire, sa taille et éventuellement du type des données qu'on lui envoie (sommets codées sur 32 bits, 64, 128, etc).

Cette étape de chargement est la source de quelques optimisations bienvenues, pour les cas où un sommet est réutilisée dans plusieurs triangles. Par exemple, prenez le cube de l'image ci-dessous. Le sommet rouge du cube appartient aux 3 faces grise, jaune et bleue, et sera présent en trois exemplaires dans le tampon de sommets : un pour la face bleue, un pour la jaune, et un pour la grise. Pour éviter ce gâchis, les concepteurs d'API et de cartes graphiques ont inventé des techniques pour limiter la consommation de mémoire.

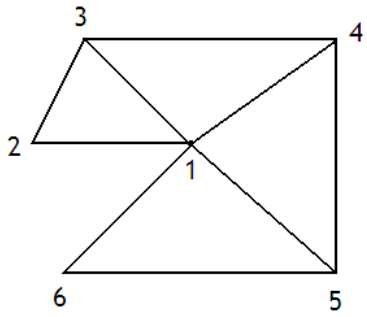


La technique des **triangles strip** permet d'optimiser le rendu de triangles placés en série, qui ont une arête et deux sommets en commun. L'optimisation consiste à ne stocker complètement que le premier triangle le plus à gauche, les autres triangles étant codés avec un seul sommet. Ce sommet est combiné avec les deux derniers sommets chargés par l'input assembler pour former un triangle. Pour gérer ces triangles strips, l'input assembler doit mémoriser dans un registre les deux derniers sommets utilisées. En mémoire, le gain est énorme : au lieu de trois sommets pour chaque triangle, on se retrouve avec un sommet pour chaque triangle, sauf le premier de la surface.



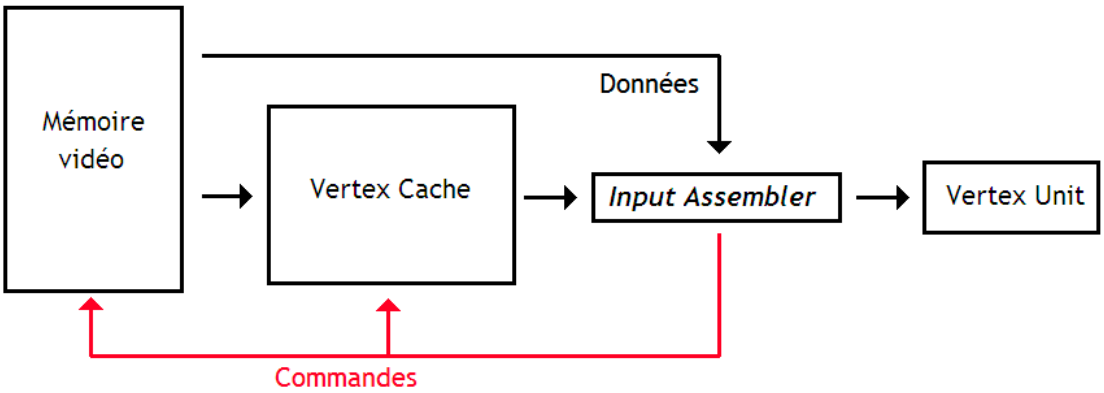
La technique des **triangles fan** fonctionne comme pour le triangles strip, sauf que le sommet n'est pas combiné avec les deux sommets précédents. Supposons que je crée un premier triangle avec les sommets v1, v2, v3. Avec la technique des triangles strips, les deux sommets réutilisés auraient été les sommets v2 et v3. Avec les triangles fans, les sommets réutilisés sont les sommets v1 et v3. Les triangles fans sont utiles pour créer des figures comme des cercles, des halos de

lumière, etc.



Enfin, nous arrivons à la dernière technique, qui permet de stocker chaque sommet en un seul exemplaire dans le tampon de sommets. Le tampon de sommets est couplé à un **tampon d'indices**. Ces indices servent à localiser le sommet dans le tampon de sommets. Pour charger un sommet, il suffit de fournir l'indice du sommet. Pour charger plusieurs fois le même sommet, il suffira de fournir le même indice à l'input assembler. On gagne de la mémoire vu qu'un indice prend moins de place qu'un sommet : à peu près 32 bits pour l'indice contre 128 à 256 bits par sommet.

Avec un tampon d'indices, un sommet peut être chargé plusieurs fois depuis la mémoire vidéo. Pour exploiter cette propriété, les cartes graphiques intercalent une mémoire ultra-rapide entre la mémoire vidéo et la sortie de l'input assembler : le **cache de sommets**. Chaque sommet est stocké dans ce cache avec son indice en guise de Tag. Sur les cartes graphiques assez anciennes, ce cache est souvent très petit, à peine 30 à à 50 sommets. Pour profiter le plus possible de ce cache, les concepteurs de jeux vidéo peuvent changer l'ordre des sommets en mémoire.



Transformation

Chaque sommet appartient à un objet, dont la surface est modélisée sous la forme d'un ensemble de points. Chaque point est localisé par rapport au centre de l'objet qui a les coordonnées (0, 0, 0). La première étape consiste à placer cet objet aux coordonnées (X, Y, Z) déterminées par le moteur physique : le centre de l'objet passe des coordonnées (0, 0, 0) aux coordonnées (X, Y, Z) et tous les sommets de l'objet doivent être mis à jour. De plus, l'objet a une certaine orientation : il faut aussi le faire tourner. Enfin, l'objet peut aussi subir une mise à l'échelle : on peut le gonfler ou le faire rapetisser, du moment que cela ne modifie pas sa forme, mais simplement sa taille. En clair, l'objet subit une translation, une rotation et une mise à l'échelle.

Ensuite, la carte graphique va effectuer un dernier changement de coordonnées. Au lieu de considérer un des bords de la scène 3D comme étant le point de coordonnées (0, 0, 0), il va passer dans le référentiel de la caméra. Après cette transformation, le point de coordonnées (0, 0, 0) sera la caméra. La direction de la vue du joueur sera alignée avec l'axe de la profondeur (l'axe Z).

Toutes ces transformations ne sont pas réalisées les unes après les autres. À la place, elles sont toutes effectuées en un seul passage. Pour réussir cet exploit, les concepteurs de cartes graphiques et de jeux vidéos utilisent ce qu'on appelle des matrices, des tableaux organisés en lignes et en colonnes avec un nombre dans chaque case. Le lien avec la 3D, c'est qu'appliquées sur le vecteur (X, Y, Z) des coordonnées d'un sommet, la multiplication par une matrice peut simuler des translations, des rotations, ou des mises à l'échelle. Il existe des matrices pour la translation, la mise à l'échelle, d'autres pour la rotation, etc. Et mieux : il existe des matrices dont le résultat correspond à plusieurs opérations simultanées : rotation ET translation, par exemple. Autant vous dire que le gain en terme de performances est assez sympathique.

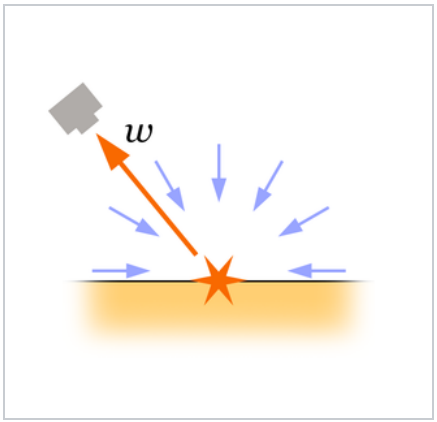
Mais les matrices qui le permettent sont des matrices avec 4 lignes et 4 colonnes. Et pour multiplier une matrice par un vecteur, il faut que le nombre de coordonnées dans le vecteur soit égal au nombre de colonnes. Pour résoudre ce petit problème, on ajoute une 4éme coordonnée, la coordonnée homogène. Pour faire simple, elle ne sert à rien, et est souvent mise à 1, par défaut.

Les anciennes cartes graphiques contenaient un circuit spécialisé dans ce genre de calculs, qui prenait un sommet et renvoyait le sommet transformé. Il était composé d'un gros paquet de multiplieurs et d'additionneurs flottants. Pour plus d'efficacité, certaines cartes graphiques comportaient plusieurs de ces circuits, afin de pouvoir traiter plusieurs sommets

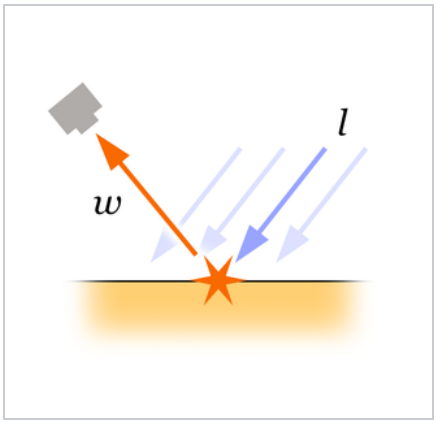
d'un même objet en même temps.

Eclairage

Seconde étape de traitement : l'éclairage. À la suite de cette étape d'éclairage, chaque sommet se voit attribuer une couleur, qui correspond à sa luminosité. Le calcul exact de cette couleur demande de calculer trois couleurs indépendantes, dont l'origine est différente, et qui ne proviennent pas des mêmes types de sources lumineuses. Par exemple, on peut simuler le soleil sans utiliser de source de lumière grâce à cette couleur. Il s'agit d'une source de **lumière ambiante**. Par simplicité, il est dit que celle-ci est égale en tout point de la scène 3D (d'où le terme lumière ambiante). Mais toute scène 3D contient aussi des sources de lumières, comme des lampes, des torches, etc. Celles-ci sont modélisées comme de simples points, qui ont une couleur bien précise (la couleur de la lumière émise) et émettent une intensité lumineuse codée par un entier. La lumière provenant de ces sources de lumière est appelée la **lumière directionnelle**.



Lumière ambiante.

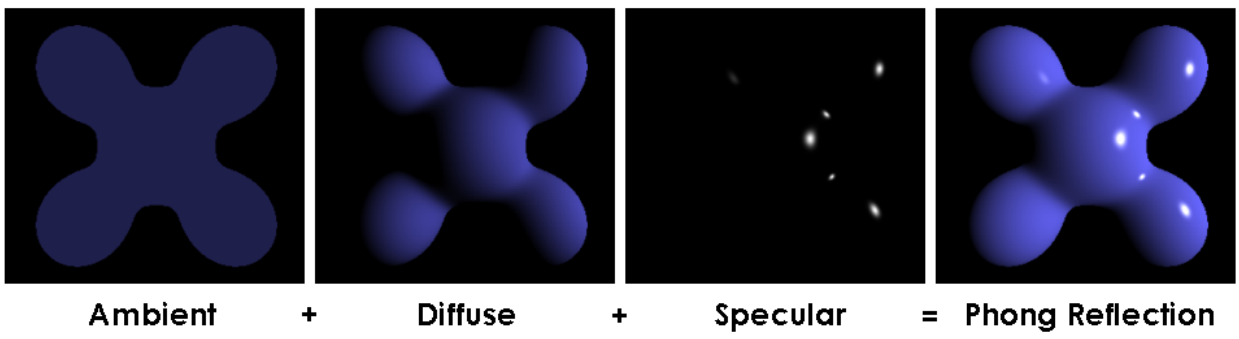


Lumière directionnelle.

- La **couleur ambiante** correspond à la lumière ambiante réfléchie par la surface. Celle-ci s'obtient simplement en multipliant la couleur ambiante de la surface par l'intensité de la lumière ambiante, deux constantes pré-calculées par les concepteurs du jeu vidéo ou du rendu 3D.
- les autres couleurs proviennent de la réflexion de a lumière directionnelle. Elles doivent être calculées par la carte graphique, généralement avec des algorithmes compliqués qui demandent de faire des calculs entre vecteurs.
  - La **couleur spéculaire** est la couleur de la lumière réfléchie via la réflexion de Snell-Descartes.
  - La **couleur diffuse** vient du fait que la surface d'un objet diffuse une partie de la lumière qui lui arrive dessus dans toutes les directions. Cette lumière « rebondit » sur la surface de l'objet et une partie s'éparpille dans un peu toutes les directions. La couleur diffuse ne dépend pas vraiment de l'orientation de la caméra par rapport à la surface. Elle dépend uniquement de l'angle entre le rayon de lumière et la verticale de la surface (sa normale).

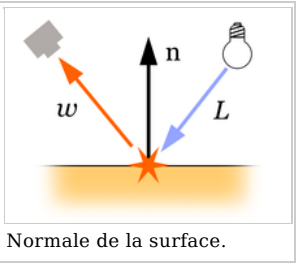
Illustration de la dispersion de la lumière diffuse par une surface, suivant sa rugosité.

Ces couleurs sont additionnées ensemble pour donner la couleur finale du sommet. Chaque composante rouge, bleu, ou verte de la couleur est traitée indépendamment des autres.



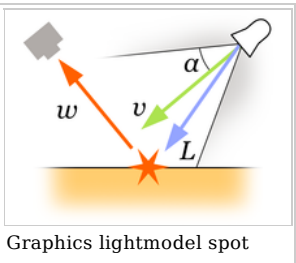
Vecteurs nécessaires pour faire les calculs

Les calculs de réflexion de la lumière demandent de connaitre l'orientation de la surface. Pour gérer cette orientation, le sommet est fourni avec une information qui indique comment est orientée la surface : la **normale**. Cette normale est un simple vecteur, perpendiculaire à la surface de l'objet, dont l'origine est le sommet. Autre paramètre d'une surface : son **coefficient de réflexion**. Il indique si la surface réfléchit beaucoup la lumière ou pas, et dans quelles proportions. Généralement, chaque point d'une surface a trois coefficient de réflexion fournis de base : un pour la couleur diffuse, un pour la couleur spéculaire, et un pour la couleur ambiante. Outre la normale et la brillance, il faut aussi connaitre l'angle entre la normale et le trajet surface-caméra (noté w dans le schéma ci-dessous).



Normale de la surface.

La carte graphique a aussi besoin de l'angle avec lequel arrive un rayon lumineux sur la surface de l'objet. Cet angle dépend de l'orientation de la lumière et du point de surface considéré. Par orientation de la lumière, il faut savoir que la majorité des sources de lumière émet de la lumière dans une direction privilégiée, la lumière émise diminuant avec l'angle comparé à cette direction. Il existe bien quelques sources de lumière qui émettent de manière égale dans toutes les directions, mais nous passons cette situation sous silence. La direction privilégiée est notée v dans le schéma du dessous. Le trajet entre la source de lumière fait un certain angle par rapport à la direction privilégiée. Il faut donc avoir une formule qui donne l'intensité de la lumière en fonction de cet angle, angle noté L dans le schéma du dessous.

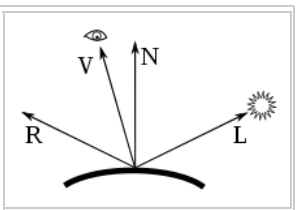


Graphics lightmodel spot

À partir de ces informations, la carte graphique calcule l'éclairage. Les anciennes cartes graphiques, entre la Geforce 256 et la Geforce FX contenaient des circuits câblés capables d'effectuer des calculs d'éclairage simples. Cette fonction de calcul de l'éclairage faisait partie intégrante d'un gros circuit nommé le T&L. Dans ce qui va suivre, nous allons voir l'algorithme d'éclairage de Phong, une version simplifiée de la méthode utilisée dans les circuits de T&L.

Calcul des couleurs spéculaire et diffuse

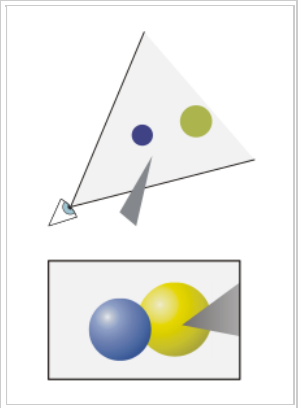
Sur la droite, vous voyez illustrés les vecteurs utiles dans le calcul de l'éclairage directionnel. De base, le vecteur L est dirigé vers la source de lumière, et sa norme est égale à l'intensité de la source de lumière (qu'on suppose connue). La normale est multipliée par la couleur diffuse du sommet, ce qui donne le vecteur de couleur diffuse. La couleur diffuse finale est calculée en effectuant le produit scalaire entre l'intensité de la source de lumière et le vecteur de couleur diffuse.



Vecteurs utilisés dans l'algorithme de Phong (et dans le calcul de l'éclairage, de manière générale).

La lumière réfléchie directement par la surface est émise dans la direction R. Sa couleur est simplement égale à l'intensité de la lumière multipliée par la couleur spéculaire. Mais la caméra n'est pas forcément alignée avec cette direction. Pour calculer la lumière spéculaire, il faut prendre en compte l'angle que fait la caméra et la lumière réfléchie. Dans le schéma de droite, c'est l'angle entre les vecteurs R et V, que nous appellerons angle A. Pour calculer la couleur dans la direction V, il faut multiplier la couleur sur le rayon R par le carré du cosinus de l'angle A.

Le rasterizeur


Rasterization.

A ce stade, les vertices ont étés converties en triangles, après une éventuelle phase de tesselation. Mais toutes les vertices ne s'afficheront pas à l'écran : une bonne partie n'est pas dans le champ de vision, une autre est caché par d'autres objets, etc. Dans un souci d'optimisation, ces vertices non-visibles doivent être éliminées. Une première optimisation consiste à ne pas afficher les triangles en-dehors du champ de vision de la caméra : c'est le **clipping**. Toutefois, un soin particulier doit être pris pour les triangles dont une partie seulement est dans le champ de vision : ceux-ci doivent être découpés en plusieurs triangles, tous présents intégralement dans le champ de vision. La seconde s'appelle le **Back-face Culling**. Celle-ci va simplement éliminer les triangles qui tournent le dos à la caméra. Ces triangles sont ceux qui sont placés sur les faces arrière d'une surface. On peut déterminer si un triangle tourne le dos à la caméra en effectuant des calculs avec sa normale. Enfin, chaque pixel de l'écran se voit attribuer un ou plusieurs triangle(s). Cela signifie que sur le pixel en question, c'est le triangle attribué au pixel qui s'affichera. C'est lors de cette phase que la perspective est gérée, en fonction de la position de la caméra.

Triangle setup

Une fois tous les triangles non-visibles éliminés, la carte graphique va attribuer les triangles restants à des pixels : c'est l'étape de Triangle Setup.

Fonction de contours

On peut voir un triangle comme une portion du plan délimitée par trois droites. À partir de chaque droite, on peut créer une **fonction de contours**, qui va prendre un pixel et va indiquer de quel côté de la droite se situe le pixel. La fonction de contours va, pour chaque point sur l'image, renvoyer un nombre entier :

- si le point est placé sur la droite, la fonction renvoie zéro ;
- si le point est placé d'un coté de la droite, cette fonction renvoie un nombre négatif ;
- et enfin, si le point est placé de l'autre coté, la fonction renvoie un nombre positif.

Comment calculer cette fonction ? Tout d'abord, nous allons dire que le point que nous voulons tester a pour coordonnées x et y sur l'écran. Ensuite, nous allons prendre un des sommets du triangle, de coordonnées X et Y. L'autre sommet, placé sur cette droite, sera de coordonnées X2 et Y2. La fonction est alors égale à :

$$(x-X)*(Y2-Y)-(y-Y)*(X2-X)$$

Si vous appliquez cette fonction sur chaque coté du triangle, vous allez voir une chose assez intéressante :

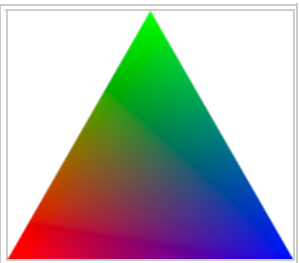
- à l'intérieur du triangle, les trois fonctions (une par côté) donneront un résultat positif ;
- à l'extérieur, une des trois fonctions donnera un résultat négatif.

Pour savoir si un pixel appartient à un triangle, il suffit de tester le résultat des fonctions de contours.

Triangle traversal

Dans sa version la plus naïve, tous les pixels de l'écran sont testés pour chaque triangle. Si le triangle est assez petit, une grande quantité de pixels seront testés inutilement. Pour éviter cela, diverses optimisations ont étés inventées. La première consiste à déterminer le plus petit rectangle possible qui contient le triangle, et à ne tester que les pixels de ce rectangle. De nos jours, les cartes graphiques actuelles se basent sur une amélioration de cette méthode. Le principe consiste à prendre ce plus petit rectangle, et à le découper en morceaux carrés. Tous les pixels d'un carré seront testés simultanément, dans des circuits séparés, ce qui est plus rapide que les traiter uns par uns.

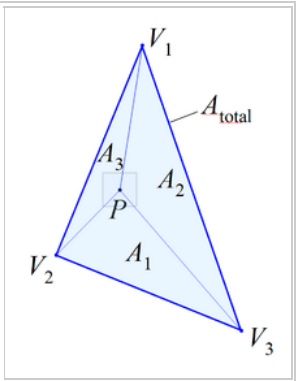
Interpolation des pixels


Interpolation des pixels.

Une fois l'étape de triangle setup terminée, on sait donc quels sont les pixels situés à l'intérieur d'un triangle donné. Mais il faut aussi remplir l'intérieur des triangles : les pixels dans le triangle doivent être coloriés, avoir une coordonnée de profondeur, etc. Pour cela, nous sommes obligés d'extrapoler la couleur et la profondeur à partir des données situées aux sommets. Cela va être fait par une étape d'interpolation, qui va calculer les informations à attribuer aux pixels qui ne sont pas pile-poil sur une vertice. Par exemple, si j'ai un sommet vert, un sommet rouge, et un sommet bleu, le triangle résultant doit être colorié comme indiqué dans le schéma de droite. Ce que l'étape de triangle setup va fournir, ce sont des informations qui précisent quelle est la couleur, la profondeur d'un pixel calculée à partir d'un triangle. Or, il est rare qu'on ne trouve qu'un seul triangle sur la trajectoire d'un pixel : c'est notamment le cas quand plusieurs objets sont l'un derrière l'autre. Si vous tracer une demi-droit dont l'origine est la caméra, et qui passe par le pixel, celle-ci intersecte la géométrie en plusieurs points : ces points sont appelés des **fragments**. Dans la suite, les fragments attribués à un même pixel sont combinés pour obtenir la couleur finale de ce pixel. Mais cela s'effectuera assez loin dans le pipeline graphique, et nous reviendrons dessus en temps voulu.

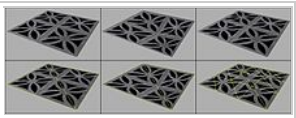
Pour calculer les couleurs et coordonnées de chaque fragment, on va utiliser les **coordonnées barycentriques**. Pour faire simple, ces coordonnées sont trois coordonnées notées u, v et w. Pour les déterminer, nous allons devoir relier le

fragment aux trois autres sommets du triangle, ce qui donne trois triangles. Les coordonnées barycentriques sont simplement proportionnelles aux aires de ces trois triangles. L'aire totale du triangle, ainsi que l'aire des trois sous-triangles, sont calculées par un petit calcul tout simple, que la carte graphique peut faire toute seule. Quand je dis proportionnelles, il faut savoir que ces trois aires sont divisées par l'aire totale du triangle, qui se ramène dans l'intervalle [0, 1]. Cela signifie que la somme de ces trois coordonnées vaut 1 : u + v + w = 1. En conséquence, on peut se passer d'une des trois coordonnées dans nos calculs, vu que w = 1 - (u + v). Ces trois coordonnées permettent de faire l'interpolation directement . Il suffit de multiplier la couleur/profondeur d'un sommet par la coordonnée barycentrique associée, et de faire la somme de ces produits. Si l'on note C1, C2, et C3 les couleurs des trois sommets, la couleur d'un pixel vaut : (C1 * u) + (C2 * v) + (C3 * w).

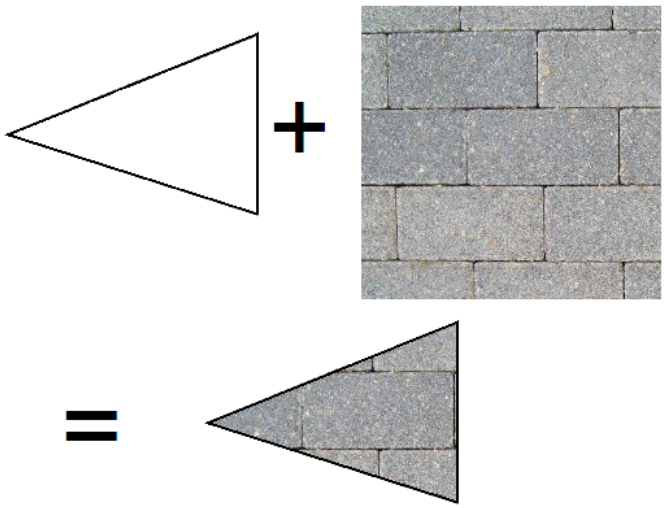


Coordonnées barycentriques.

Le problème : la **perspective** n'est pas prise en compte ! Intuitivement, on pouvait le deviner : la coordonnée de profondeur (z) n'était pas prise en compte dans le calcul de l'interpolation. Pour résumer, le problème vient du fait que l'interpolation de la cordonnée z est à l'origine de la mauvaise perspective : en interpolant 1/z, et en calculant z à partir de cette valeur interpolée, les problèmes disparaissent. Le problème : la perspective n'est pas prise en compte ! Intuitivement, on pouvait le deviner : la coordonnée de profondeur (z) n'était pas prise en compte dans le calcul de l'interpolation. Pour résumer, le problème vient du fait que l'interpolation de la cordonnée z est à l'origine de la mauvaise perspective : en interpolant 1/z, et en calculant z à partir de cette valeur interpolée, les problèmes disparaissent.


Correction de perspective.

Les unités de texture

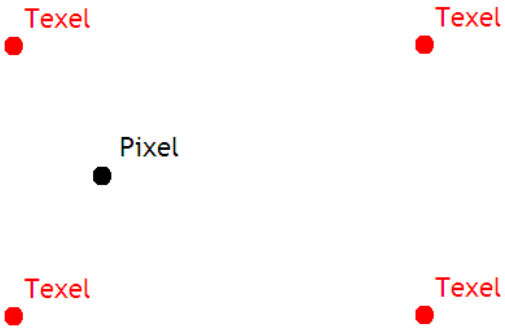


Les **textures** sont des images que l'on va plaquer sur la surface d'un objet, du papier peint en quelque sorte. Les cartes graphiques supportent divers formats de textures, qui indiquent comment les pixels de l'image sont stockés en mémoire : RGB, RGBA, niveaux de gris, etc. Une texture est donc composée de "pixels", comme toute image numérique. Pour bien faire la différence entre les pixels d'une texture, et les pixels de l'écran, les pixels d'une texture sont couramment appelés des texels.

Plaquer une texture sur un objet consiste à attribuer une vertice à chaque texel, ce qui est fait lorsque les créateurs de jeu vidéo conçoivent le modèle de l'objet. Chaque vertice contient donc des coordonnées de texture, qui indiquent quel texel appliquer sur la vertice. Ces coordonnes précisent la position du texel dans la texture. Par exemple, la coordonnée de texture peut dire : je veux le pixel qui est à la ligne 5, et la colonne 27 dans ma texture. Lors de la rasterization, ces coordonnées sont interpolées, et chaque pixel de l'écran se voit attribuer une coordonnée de texture, qui indique avec quel texel il doit être colorié. À partir de ces coordonnées de texture, le circuit de gestion des textures calcule l'adresse du texel qui correspond, et se charge de lire celui-ci. Sur les anciennes cartes graphiques, les textures disposaient de leur propre mémoire, séparée de la mémoire vidéo. Mais c'est du passé : de nos jours, les textures sont stockées dans la mémoire vidéo principale. Évidemment, l'algorithme de rasterization a une influence sur l'ordre dans lequel les pixels sont envoyés aux unités de texture. Et suivant l'algorithme, les texels lus seront proches ou dispersés en mémoire. Généralement, le meilleur algorithme est celui du tiled traversal.

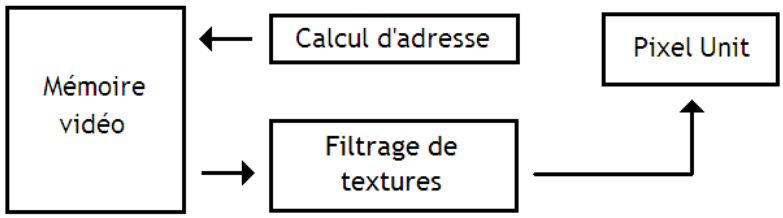
Filtrage

On pourrait croire que plaquer des textures sans autre forme de procès suffit à garantir des graphismes d'une qualité époustouflante. Mais les texels ne vont pas tomber tout pile sur un pixel de l'écran : la vertice correspondant au texel peut être un petit peu trop en haut, ou trop à gauche, etc.



Pour résoudre ce problème, on peut colorier avec le texel correspondant à la vertice la plus proche. Autant être franc, le résultat est assez dégueulasse. Pour améliorer la qualité de l'image, la carte graphique va effectuer un **filtrage de texture**. Ce filtrage consiste à choisir le texel à appliquer sur un pixel du mieux possible, par un calcul mathématique assez simple. Ce filtrage est réalisé par un circuit spécialisé : le texture sampler, lui-même composé :

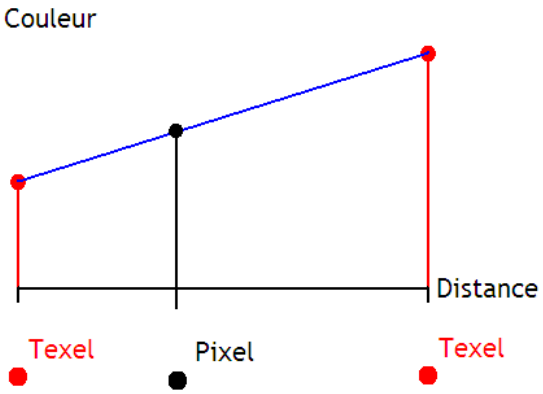
- d'un circuit qui calcule les adresses mémoire des texels à lire et les envoie à la mémoire ;
- d'un circuit qui va filtrer les textures.



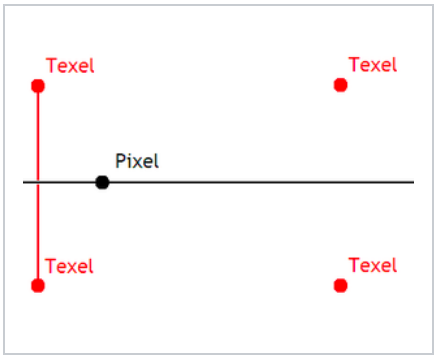
Filtrage bilinéaire

Le plus simple de ces filtrage est le filtrage bilinéaire, qui effectue une sorte de moyenne des quatre texels les plus proches du pixel à afficher. Plus précisément, ce filtrage va effectuer ce qu'on appelle des interpolations linéaires. Pour

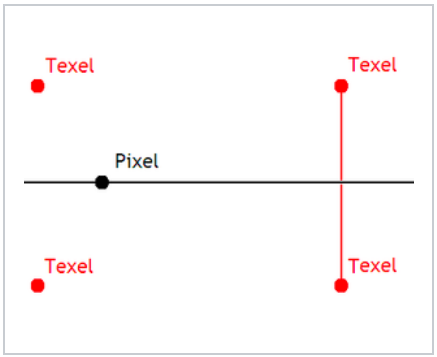
comprendre l'idée, nous allons prendre une situation très simple, où un pixel est aligné avec deux autres texels. Pour effectuer l'interpolation linéaire entre ces deux texels, nous allons faire une première supposition : la couleur varie entre les deux texels en suivant une fonction affine. On peut alors calculer la couleur du pixel par un petit calcul mathématique d'interpolation (une simple moyenne pondérée par la distance).



Seul problème, cela marche pour deux pixels, pas 4. Avec 4 pixels, nous allons devoir calculer la couleur de points intermédiaires. Le premier est celui qui se situe à l'intersection entre la droite formé par les deux texels de gauche, et la droite parallèle à l'abscisse qui passe par le pixel. Le second est celui qui se situe à l'intersection entre la droite formé par les deux texels de gauche, et la droite parallèle à abscisse qui passe par le pixel. La couleur de ces deux points se calcule par interpolation linéaire, et il suffit d'utiliser une troisième interpolation linéaire pour obtenir le résultat.

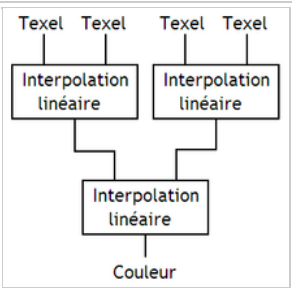


Première interpolation.



Deuxième interpolation.

Le circuit qui permet de faire ce genre de calcul est particulièrement simple. On trouve un circuit de chaque pour chaque composante de couleur de chaque texel : un pour le rouge, un pour le vert, un pour le bleu, et un pour la transparence. Chacun de ces circuit est composé de sous-circuits chargés d'effectuer une interpolation linéaire, reliés comme suit.

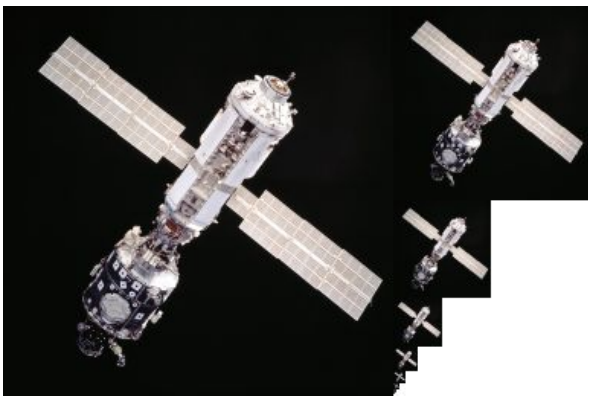


Unité de filtrage bilinéaire.

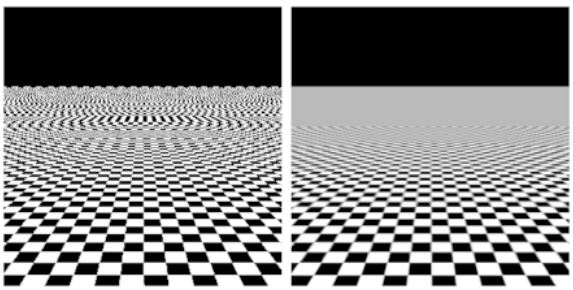
Mip-mapping

Si une texture est plaquée sur un objet lointain, une bonne partie des détails est invisible pour l'utilisateur. Par exemple, un objet assez lointain peut très bien ne prendre que quelques dizaines de pixels à l'écran. Dans ces conditions, plaquer une texture de 512 pixel de coté serait vraiment du gâchis en terme de performance : il faudrait charger tous les pixels de la texture, les traiter, et n'en garder que quelque uns. De plus, cela pourrait créer des artefacts visuels : les textures affichées ont tendance à pixeliser. Pour limiter la casse, les concepteurs de jeux vidéo utilisent souvent la technique du **mip-mapping**. Cette technique consiste simplement à utiliser plusieurs exemplaires d'une même texture, chaque exemplaire étant adapté à une certaine distance. Ce qui différenciera ces exemplaires, ce sera leur résolution. Par exemple, une texture sera stocké dans un exemplaire de 512 * 512 pixels, un autre de 256 * 256, un autre de 128 * 128 et ainsi de suite jusqu'à un dernier exemplaire de 32 * 32. Chaque exemplaire correspond à un **niveau de détail**, aussi

appelé Level Of Detail en anglais (abrévié en LOD). Le bon exemplaire sera choisit lors de l'application de la texture. Ainsi, les objets proches seront rendus avec la texture la plus grande (512 par 512 dans notre exemple). Au-delà d'une certaine distance, les textures 256 par 256 seront utilisées. Encore plus loin, les textures 128 par 128 seront utilisées, etc.



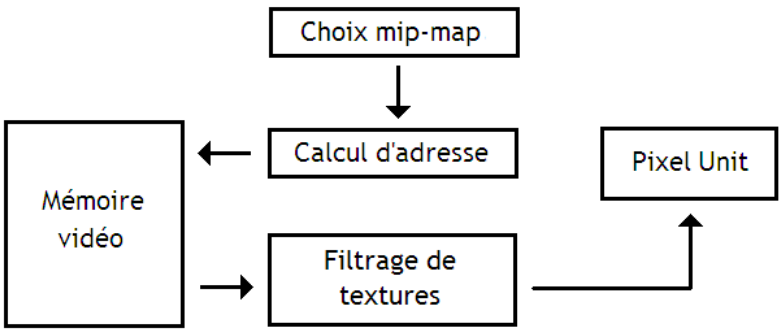
Exemples de mip-maps.



Exemple de mipmapping.

Évidemment, cette technique consomme de la mémoire vidéo, vu que chaque texture est dupliquée en plusieurs exemplaires. Dans le détail, la technique du mip-mapping prend au maximum 33% de mémoire en plus (sans compression). Cela vient du fait que chaque texture prend 4 fois de pixels que l'image immédiatement supérieure : 2 fois moins de pixels en largeur, et 2 fois moins en hauteur. Donc, si je pars d'une texture de base contenant X pixels, la totalité des mip-maps, texture de base comprise, prendra X + (X/4) + (X/16) + (X/256) + ... Un petit calcul de limite donne 4/3 * X, soit 33% de plus.

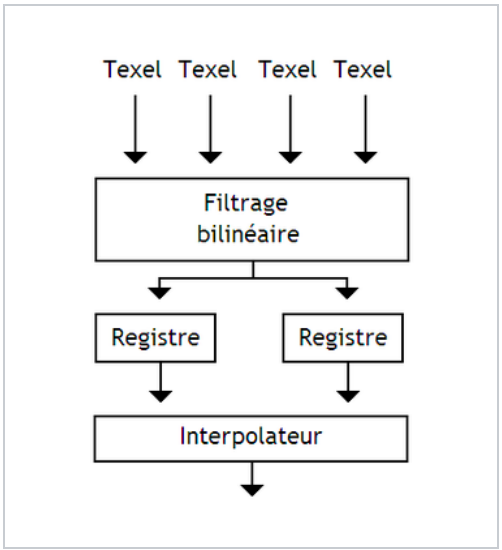
Pour choisir la bonne mipmap, les circuits de calcul d'adresse doivent connaitre les adresses des différents niveaux de détails, ainsi que des informations sur la profondeur de la texture. Pour faciliter les calculs d'adresse, les mip-maps d'une texture sont stockées les unes après les autres en mémoire (dans un tableau, comme diraient les programmeurs). Ainsi, pas besoin de se souvenir de la position en mémoire de chacune des mip-map : l'adresse de la plus grande, et quelques astuces arithmétiques suffisent.



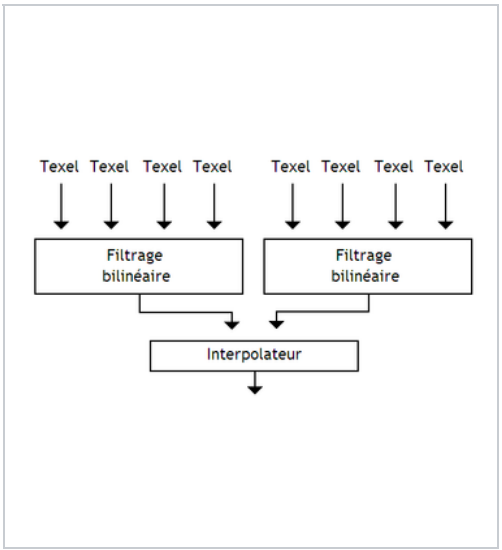
Filtrage trilinéaire

Avec le mip-mapping, des discontinuités apparaissent lorsqu'une texture est appliquée répétitivement sur une surface, comme quand on fabrique un carrelage à partir de carreaux tous identiques. Par exemple, pensez à une texture de sol : celle-ci est appliquée plusieurs fois sur toute la surface du sol. Au delà d'une certaine distance, le LOD utilisé change brutalement et passe par exemple de 512*512 à 256*256, ce qui est visible pour un joueur attentif. Le **filtrage trilinéaire** permet d'adoucir ces transitions. Il consiste à faire « une moyenne » pondérée entre les textures des niveaux de détails adjacents. Le filtrage trilinéaire demande d'effectuer deux filtrages bilinéaires : un sur la texture du niveau de détail adapté, et un autre sur la texture de niveau de détail inférieur. Les deux textures obtenues par filtrage vont ensuite subir une interpolation linéaire.

Le circuit qui s'occupe de calculer un filtrage trilinéaire est une amélioration du circuit utilisé pour le filtrage bilinéaire. Il est constitué d'un circuit effectuant un filtrage bilinéaire, de deux registres, d'un interpolateur linéaire, et de quelques circuits de gestion, non-représentés. Son fonctionnement est simple : ce circuit charge 4 texels d'une mip-map, les filtre, et stocke le tout dans un registre. Il recommence l'opération avec les 4 texels de la mip-map de niveau de détail inférieure, et stocke le résultat dans un autre registre. Enfin, le tout passe par un circuit qui interpole les couleurs finales en tenant compte des coefficients d'interpolation linéaire, mémorisés dans des registres. Il est possible de créer un circuit qui effectue les deux filtrages en parallèle. Seul problème : ce genre de circuit nécessite de charger 8 pixels simultanément. Qui plus est, ces 8 pixels ne sont pas consécutifs en mémoire. Utiliser ce genre de circuit nécessiterait d'adapter la mémoire et le cache, ce qui ne vaut généralement pas la peine.



Unité de filtrage trilineaire série.

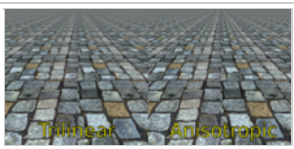


Unité de filtrage trilineaire parallèle.

Modifier le circuit de filtrage ne suffit pas. Comme je l'ai dit plus haut, la dernière étape d'interpolation linéaire utilise des coefficients, qui lui sont fournis par des registres. Seul problème : entre le temps où ceux-ci sont calculés par l'unité de mip-mapping, et le moment où les texels sont chargés depuis la mémoire, il se passe beaucoup de temps. Le problème, c'est que les unités de texture sont souvent pipelinées : elles peuvent démarrer une lecture de texture sans attendre que les précédentes soient terminées. A chaque cycle d'horloge, une nouvelle lecture de texels peut commencer. La mémoire vidéo est conçue pour supporter ce genre de chose. Cela a une conséquence : durant les 400 à 800 cycles d'attente entre le calcul des coefficients, et la disponibilité des texels, entre 400 et 800 coefficients sont produits : un par cycle. Autant vous dire que mémoriser 400 à 800 ensembles de coefficient prend beaucoup de registres.

Filtrage anisotropique

D'autres artefacts peuvent survenir lors de l'application d'une texture, la perspective pouvant déformer les textures et entraîner l'apparition de flou. Pour gommer ce flou de perspective, les chercheurs ont inventé le **filtrage anisotropique**. En fait, je devrais plutôt dire : LES filtrages anisotropique. Il en existe plusieurs. Certains sont des algorithmes qui ne sont pas utilisés dans les cartes graphiques actuelles. Ceux-ci prennent beaucoup trop de circuits, et sont trop gourmand en accès mémoires et en calculs pour être efficaces. Il semblerait que les cartes graphiques actuelles utiliseraient des variantes de l'algorithme TEXRAM, comme l'algorithme Fast Footprint Assembly. On pourrait aussi citer l'algorithme Talisman de Microsoft, qui serait implémenté depuis Direct X 6.0. Tous vont effectuer plusieurs filtrages bilinéaires sur des texels convenablement choisis, d'une manière qui change selon l'algorithme utilisé. De plus, ces texels se verront attribuer des coefficients afin de prendre en compte certains texels en priorité. Au niveau des circuits, l'utilisation de filtrage anisotropique ne change pas grand chose au niveau des circuits de filtrage.



Exemple de filtrage anisotrope.

Compression de textures

Certaines textures un peu spéciales peuvent aller jusqu'au mébioctet. Pour limiter la casse, les cartes graphiques peuvent compresser les textures. La carte graphique contient alors un circuit, capable de décompresser un ou plusieurs texels. Fait important : toute la texture n'est pas décompressée : seuls les texels lus depuis la mémoire le sont. Nos cartes graphiques supportent un grand nombre de formats de **compression de texture**, qui entrainent souvent une légère perte de qualité lors de la compression. Toutefois, cette perte peut être compensée en utilisant des textures à résolution plus grande. Nous allons voir quelque algorithmes de compression de textures. Il existe des formats de texture plus récents que ceux qui nous allons aborder, comme l'Ericsson Texture Compression ou l'Adaptive Scalable Texture Compression.

Palette

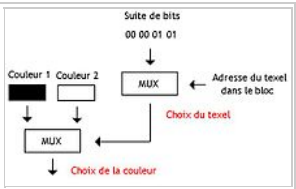
La première technique est celle de la **palette**, que l'on a entraperçue dans le chapitre sur les cartes graphiques 2D. Avec cette technique, chaque texture est fournie avec une table de correspondances entre numéro et couleurs : ce tableau s'appelle la palette. La texture ne contient aucune couleur, chaque pixel indiquant le numéro de sa couleur. Cependant, la table des couleurs contient un nombre limité de couleurs, ce qui fait que cette technique ne marche pas pour les textures qui utilisent beaucoup de couleurs différentes. Certains pixels se voient attribuer la couleur la plus proche qui est présente dans la palette, ce qui fait que la compression n'est pas sans pertes.

Vector quantization

De nos jours, la compression ne cherche pas à compresser des pixels individuels, mais ces blocs blocs de 16*16, 8*8 ou 4*4 texels. Le nombre de bit utilisé pour chaque texel peut varier, se limitant au strict minimum utile. La technique de **vector quantization** peut être vue comme une amélioration de la palette, qui travaille non pas sur des texels, mais sur des blocs de texels. À l'intérieur de la carte graphique, on trouve une table qui stocke tous les blocs possible de 2 * 2, 3 * 3 , ou 4 * 4 texels. Chaque de ces blocs se voit attribuer un numéro, et la texture sera composé d'une suite de ces numéros. Quelques anciennes cartes graphiques ATI, ainsi que quelques cartes utilisées dans l'embarqué utilisent ce genre de compression.

Block Truncation coding

La première technique de compression élaborée est celle du **Block Truncation Coding**. Cette méthode ne marche que pour les images en niveaux de gris, mais peut être amélioré pour gérer les images couleur. La majorité des algorithmes de compression de texture utilisés dans nos cartes graphiques sont une sorte d'amélioration de cet algorithme. Le BTC ne mémorise que deux niveaux de gris par bloc, que nous appellerons couleur 1 et couleur 2 : à l'intérieur du bloc, chaque pixel est obligatoirement colorié avec un de ces niveaux de gris. Pour chaque pixel dans le bloc, on utilise un bit mémoriser sa couleur : 0 pour couleur 1, et 1 pour couleur 2. Chaque bloc est donc mémorisé en mémoire par deux entiers, qui codent chacun une couleur, et une suite de bits pour les pixels proprement dit. Le circuit de décompression est alors vraiment très simple : il suffit d'utiliser deux multiplexeurs.

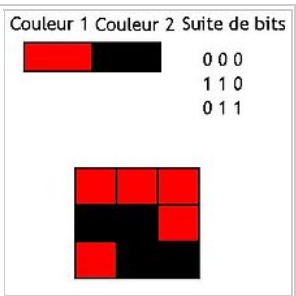


Block Truncation coding.

La technique du BTC peut être appliquée non pas du des niveaux de gris, mais pour chaque composante Rouge, Vert et Bleu d'un pixel. Dans ces conditions, chaque bloc sera séparé en trois sous-bloc : un sous-bloc pour la composante verte, un autre pour le rouge, et un dernier pour le bleu. Cela prend donc trois fois plus de place en mémoire que le BTC pur, mais cela permet de gérer les images couleur.

Color Cell Compression

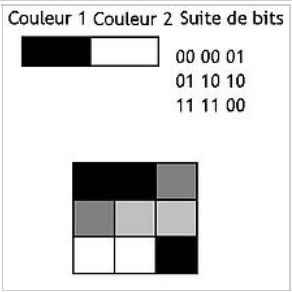
On peur améliorer le BTC pour qu'il gère des couleurs autre que des niveaux de gris : on obtient alors l'algorithme du **Color Cell Compression**, ou CCC. Ce CCC utilise deux couleurs RGBA codées sur 32 bits, au lieu de deux niveaux de gris. Le circuit de décompression est identique à celui utilisé pour le BTC.



Color Cell Compression.

S3TC / DXTC

Le format de compression de texture utilisé de base par Direct X s'appelle le DXTC. Il est décliné en plusieurs versions : DXTC1, DXTC2, etc. La première version du DXTC est une sorte d'amélioration du CCC : il ajoute une gestion minimale de transparence, et découpe la texture à compresser en carrés de 4 pixels de coté. La différence, c'est que la couleur finale d'un texel est un mélange des deux couleurs attribuée au bloc.



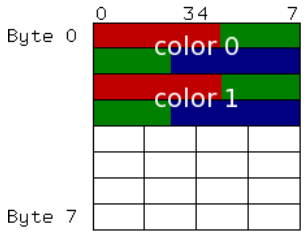
DXTC.

Pour indiquer comment faire ce mélange, on trouve deux bits de contrôle par texel. Si jamais la couleur 1 < couleur2, ces deux bits sont à interpréter comme suit :

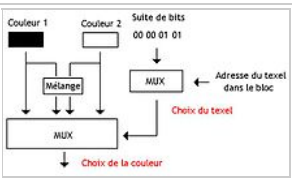
- 00 = Couleur1
- 01 = Couleur2
- 10 = (2 * Couleur1 + Couleur2) / 3
- 11 = (Couleur1 + 2 * Couleur2) / 3

Sinon, les deux bits sont à interpréter comme suit :

- 00 = Couleur1
- 01 = Couleur2
- 10 = (Couleur1 + Couleur2) / 2
- 11 = Transparent



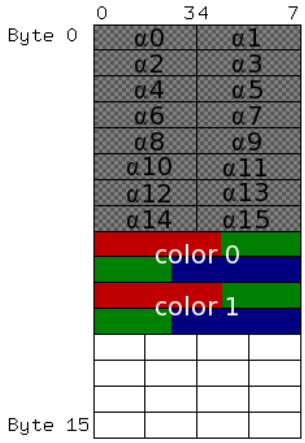
Le circuit de décompression du DXTC ressemble alors à ceci :



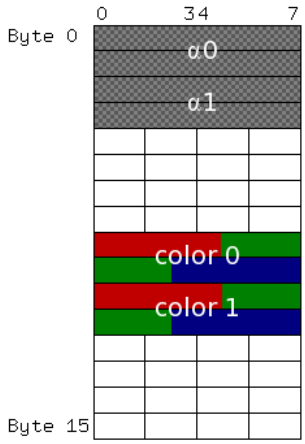
Circuit de décompression du DXTC.

DXTC 2, 3, et 4

Pour combler les limitations du DXT1, le format DXT2 a fait son apparition. Il a rapidement été remplacé par le DXT3, lui-même replacé par le DXT4 et par le DXT5. Dans le DXT3, la texture et toujours découpée en blocs de 16 texels. Seule différence : la transparence fait son apparition. Chacun de ces blocs de texels est encodé sur 128 bits. Les premiers 64 bits servent à stocker des informations de transparence : 4 bits par texel. Le tout est suivi d'un bloc de 64 bits identique au bloc du DXT1.



Dans ces deux formats, la méthode utilisée pour compresser les couleurs l'est aussi pour les valeurs de transparance. L'information de transparence est stockée par un en-tête contenant deux valeurs de transparence, le tout suivi d'une matrice qui attribue trois bits à chaque texel. En fonction de la valeur des trois bits, les deux valeurs de transparence sont combinées pour donner la valeur de transparence finale. Le tout est suivi d'un bloc de 64 bits identique à celui qu'on trouve dans le DXT1.



PVRTC

Passons maintenant à un format de compression de texture un peu moins connu, mais pourtant omniprésent dans notre vie quotidienne : le PVRTC. Ce format de texture est utilisé notamment dans les cartes graphiques de marque PowerVR. Vous ne connaissez peut-être pas cette marque, et c'est normal : elle ne crée par de cartes graphiques pour PC. Elle travaille surtout dans les cartes graphiques embarquées. Ses cartes se trouvent notamment dans l'ipad, l'iPhone, et bien d'autres smartphones actuels.

Avec le PVRTC, les textures sont encore une fois découpées en blocs de 4 texels par 4, mais la ressemblance avec le DXTC s’arrête là. Chacun de ces blocs est stocké en mémoire dans un bloc qui contient :

- une couleur codée sur 16 bits ;
- une couleur codée sur 15 bits ;
- 32 bits qui servent à indiquer comment mélanger les deux couleurs ;
- et un bit de modulation, qui permet de configurer l’interprétation des bits de mélange.

Les 32 bits qui indiquent comment mélanger les couleurs sont une collection de 2 paquets de 2 bits. Chacun de ces deux bits permet de préciser comment calculer la couleur d'un texel du bloc de 4*4.

Cache de textures

Les accès aux textures se font donc en mémoire vidéo, celle-ci étant relativement lente. Pour faciliter l'accès aux textures, les cartes 3D utilisent souvent un ou plusieurs cache, spécialisés dans le traitement des textures. Les cartes graphiques actuelles ont souvent une hiérarchie de caches.

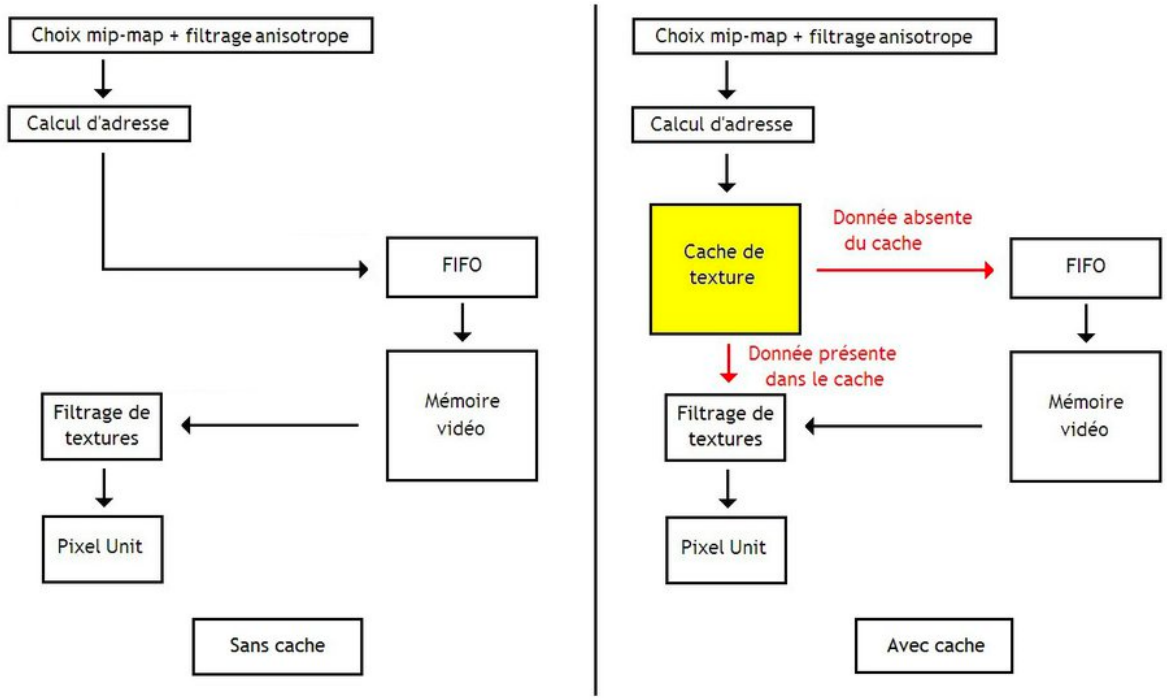
De base, les pixels d'une texture sont stockés les uns à la suite des autres, ligne par ligne. On pourrait croire que cette solution fonctionne bien pour échanger des données entre le cache de textures et la mémoire vidéo, mais elle entre en conflit avec le filtrage de texture. Comme on l'a vu précédemment, le filtrage de texture utilise souvent des carrés de texels. Dans ces conditions, mieux vaut découper la texture en carrés de N texels de coté, placés les uns à coté des autres en mémoire. Les performances sont les meilleurs possible quand chaque carré de texel permet de remplir exactement une ligne de cache.

D'ordinaire, les textures sont décompressées après lecture dans le cache. Il est possible de décompresser les textures avant de les placer dans le cache, mais ces textures décompressées prennent beaucoup plus de cache que les textures compressées. L'utilisation du cache est alors moins optimale.

Les jeux vidéos 3D récents utilisent des techniques dites de render-to-texture, qui permettent de calculer certaines données et à les écrire en mémoire vidéo pour une utilisation ultérieure. Si un seul cache de texture est présente dans la carte graphique, il n'y a pas de problèmes. Mais si il y en a plusieurs, le problème mentionné plus haut revient : les copies des autres caches doivent être invalidées. De plus, la mémoire cache qui a la bonne donnée doit fournir la bonne version de la donnée, quand les autres caches voudront la mettre à jour. Des techniques de cohérence de caches, similaires à celles utilisées sur les processeurs à mémoire partagée, sont alors utilisées.

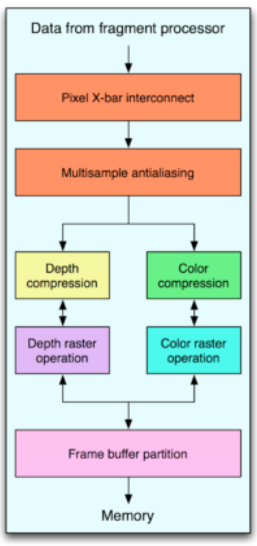
Prechargement

Pour améliorer la rapidité des accès, il est possible de préparer certaines lectures de texture à l'avance. En effet, un accès à une texture est composé d'un grand nombre de sous-étapes, comme déterminer le niveau de mip-map et calculer les adresses de texture, filtrer la texture, accéder à la mémoire vidéo, etc. Le but du prefetching est d'effectuer à l'avance les étapes avant l'accès mémoire pour certaines requêtes de texture. Ainsi, pas besoin d'attendre qu'une lecture termine pour commencer à déterminer les mip-maps ou calculer l'adresse de la prochaine lecture : les adresses des texels sont précalculées. Les adresses précalculées sont mises en attente dans une mémoire FIFO, en attendant que la mémoire vidéo soit libre. Ce prefetch peut s'implémenter de deux façons, suivant que la carte graphique utilise ou non un cache de texture.



Les Render Output Target

Pour rappel, nos fragments ne sont pas tout à fait des pixels. Il s'agit de données qui vont permettre, une fois combinées, d'obtenir la couleur finale d'un pixel. Ceux-ci contiennent diverses informations, comme leur position à l'écran, leur profondeur, leur couleur, ainsi que quelques autres informations potentiellement utiles. Une fois que nos fragments se sont vus appliquer une texture, il faut les enregistrer dans la mémoire, afin de les afficher. On pourrait croire qu'il s'agit là d'une opération très simple, mais ce n'est pas le cas. Il reste encore un paquet d'opérations à effectuer sur nos pixels : la profondeur des fragments doit être gérée, de même que la transparence, etc. Elles sont réalisées dans un circuit qu'on nomme le **Render Output Target**. Celui-ci est le tout dernier circuit, celui qui enregistre l'image finale dans la mémoire vidéo. Ce chapitre va aborder ce circuit dans les grandes lignes. Dans ce chapitre, nous allons voir celui-ci.



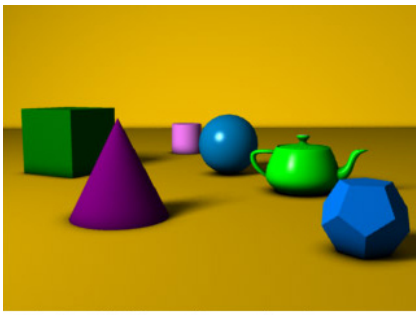
R.O.P des GeForce 6800.

Test de visibilité

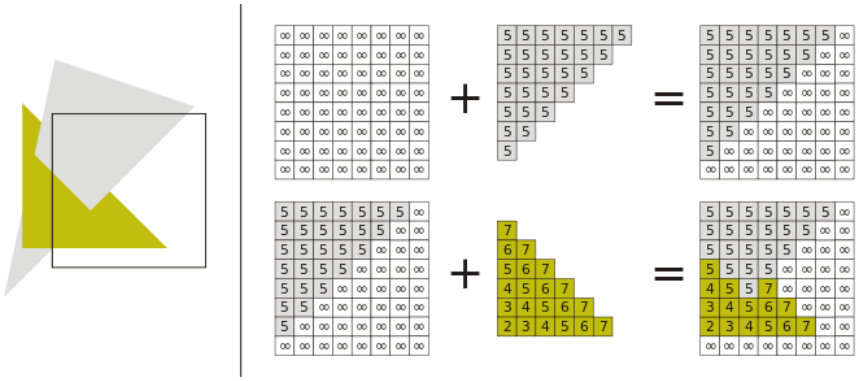
Pour commencer, il va falloir trier les fragments par leur profondeur, pour gérer les situations où un triangle en cache un autre (quand un objet en cache un autre, par exemple). Prenons un mur rouge qui cache un mur bleu. Dans ce cas, un pixel de l'écran sera associé à deux fragments : un pour le mur rouge, et un pour le bleu. De tous les fragments, un seul doit être choisit : celui du mur qui est devant. Pour cela, la profondeur d'un fragment dans le champ de vision est calculée à la rasterization. Cette profondeur est appelée la coordonnée z. Par convention, plus la coordonnée z est petite, plus l'objet est prêt de l'écran. Cette coordonnée z est un nombre, codé sur plusieurs bits.

Z-buffer

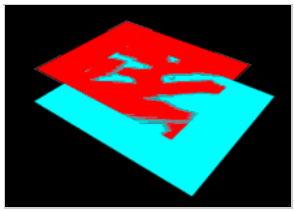
Pour savoir quels fragments sont à éliminer (car cachés par d'autres), notre carte graphique va utiliser ce qu'on appelle un **tampon de profondeur**. Il s'agit d'un tableau, stocké en mémoire vidéo, qui va mémoriser la coordonnée z de l'objet le plus proche déjà rendu pour chaque pixel.



Par défaut, ce tampon de profondeur est initialisé avec la valeur de profondeur maximale. Au fur et à mesure que les objets seront calculés, la coordonnée z stockée dans ce depth-buffer sera mise à jour, conservant ainsi la trace de l'objet le plus proche de la caméra. Si jamais un pixel à calculer a une coordonnée z plus grande que celle du tampon de profondeur, cela veut dire qu'il est situé derrière un objet déjà rendu, et il n'a pas à être calculé. Dans le cas contraire, le fragment reçu est plus près de la caméra, et il est rendu : sa coordonnée z va remplacer l'ancienne valeur z dans le tampon de profondeur.



Si deux objets sont suffisamment proches, le depth-buffer n'aura pas la précision suffisante pour discriminer les deux objets : pour lui, les deux objets seront à la même place. Conséquence : il faut bien choisir un des deux objets. Si l'objet choisi est le mauvais, des artefacts visuels apparaissent. Voici ce que cela donne :



Z-fighting

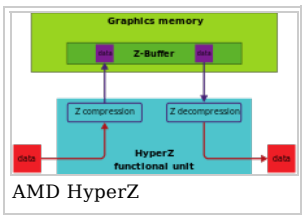
On peut préciser qu'il existe des variantes du depth-buffer, qui utilisent un codage de la coordonnée de profondeur assez différent. Ils se distinguent du depth-buffer par le fait que la coordonnée z n'est pas proportionnelle à la distance entre le fragment et la caméra. Avec eux, la précision est meilleure pour les fragments proches de la caméras, et plus faible pour les fragments éloignés. Mais il s'agit-là de détails assez mathématique que je me permets de passer sous silence.

Circuit de gestion de la profondeur

La profondeur est gérée par un circuit spécialisé. Celui-ci va :

- récupérer les coordonnées du fragment reçu à l'écran ;
- lire en mémoire la coordonnée z correspondante dans le depth-buffer ;
- comparer celle-ci avec la coordonnée z du fragment reçu ;
- et décider s'il faut mettre à jour le frame-buffer et le depth-buffer.

Comme vous le voyez, ce circuit va devoir effectuer des lectures et des écritures en mémoire vidéo. Or, la mémoire est déjà mise à rude épreuve avec les lectures de vertices et de textures. Diverses techniques existent pour limiter l'utilisation de la mémoire, en diminuant la quantité de mémoire vidéo utilisée et le nombre de lectures et écritures dans celle-ci.



AMD HyperZ

Une première solution consiste à compresser le tampon de profondeur. Évidemment, les données devront être compressées avant d'être stockée ou lue dans le tampon de profondeur. Pour donner un exemple, nous allons prendre la **z-compression** des cartes graphiques ATI radeon 9800. Cette technique de compression découpait des morceaux de 8 * 8 fragments, et les encodait avec un algorithme nommé DDPCM : Differential differential pulse code modulation. Ce découpage du tampon de profondeur en morceaux carrés est souvent utilisé dans la majorité des circuits de compression et de décompression de la profondeur. Toutefois, il arrive que certains de ces blocs ne soient pas compressés : tout dépend si la compression permet de gagner de la place ou pas . On trouve un bit au tout début de ce bloc qui indique s'il est compressé ou non.

Entre deux images, le depth-buffer doit être remis à zéro. La technique la moins performante consiste à réécrire tout son contenu avec la valeur maximale. Pour éviter cela, chaque bloc contient un bit : si ce bit est positionné à 1, alors le ROP va faire comme si le bloc avait été remis à zéro. Ainsi, au lieu de réécrire tout le bloc, il suffit simplement de récrire un bit par bloc. Le gain en nombre d'accès mémoire peut se révéler assez impressionnant.

Une dernière optimisation possible consiste à ajouter une mémoire cache qui stocke les derniers blocs de coordonnées z lues ou écrites depuis la mémoire. Comme cela, pas besoin de les recharger plusieurs fois : on charge un bloc une fois pour toute, et on le conserve pour gérer les fragments qui suivent.

Transparence

En plus de la profondeur, il faut aussi gérer la transparence, une sorte de couleur ajoutée aux composantes RGB qui indique si un pixel est plus ou moins transparent. Et là, c'est le drame : que se passe-il si un fragment transparent est placé devant un autre fragment ? Je vous le donne en mille : la couleur du pixel calculée avec l'aide du depth-buffer ne sera pas la bonne, vu que le pixel transparent ne cache pas totalement l'autre. Sur le principe, la couleur sera un mélange de la couleur du fragment transparent, et de la couleur du (ou des) fragments placé(s) derrière. Le calcul à effectuer est

très simple, et se limite en une simple moyenne pondérée par la transparence de la couleur des deux pixels.

Les pixels étant calculés uns par uns par les unités de texture et de shaders, le calcul des couleurs est effectué progressivement. Pour cela, la carte graphique doit mettre en attente les résultats temporaires des mélanges pour chaque pixel. C'est le rôle du **tampon de couleur**, une portion de la mémoire vidéo. A chaque fragment envoyé dans le ROP, celui-ci va lire la couleur dans le tampon de couleur, faire la moyenne pondérée avec le fragment recu et enregistrer le résultat.

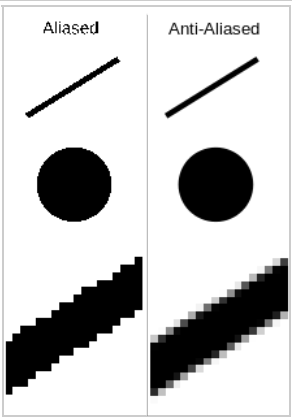
Certaines vielles cartes graphiques possédaient une "optimisation" assez intéressante : l'**alpha test**. Cette technique consistait à ne pas enregistrer en mémoire les fragments dont la couleur alpha était inférieure à un certain seuil. De nos jours, cette technologie est devenue obsolète.

Le ROP peut aussi ajouter des effets de brouillard dans notre scène 3D. Ce brouillard sera simplement modélisé par une couleur, la couleur de brouillard, qui est mélangée avec la couleur du pixel calculée par un simple calcul de moyenne. La carte graphique stocke une couleur de brouillard de base, sur laquelle elle effectuera un calcul pour déterminer la couleur de brouillard à appliquer au pixel. En dessous d'une certaine distance fogstart, la couleur de brouillard est nulle : il n'y a pas de brouillard. Au-delà d'une certaine distance fogend, l'objet est intégralement dans le brouillard : seul le brouillard est visible. Entre les deux, la couleur du brouillard et de l'objet devront toutes les deux être prises en compte dans les calculs. Les premières cartes graphiques calculaient une couleur de brouillard pour chaque vertice, dans les unités de vertices. Sur les cartes plus récentes la couleur de brouillard définitivement était calculée dans les ROP, en fonction de la coordonnée de profondeur du fragment.

Ces opérations de test et de blending sont effectuées par un circuit spécialisé qui travaille en parallèle du Depth-ROP : le Color ROP. Il va ainsi mélanger et tester nos couleurs pendant que le Depth-ROP effectue ses comparaisons entre coordonnées z. Et comme toujours, les lectures et écritures de couleurs peuvent saturer la mémoire vidéo. On peut diminuer la charge de la mémoire vidéo en ajoutant une mémoire cache, ou en compressant les couleurs. Il est à noter que sur certaines cartes graphiques, l'unité en charge de calculer les couleurs peut aussi servir à effectuer des comparaisons de profondeur. Ainsi, si tous les fragments sont opaques, on peut traiter deux fragments à la fois. C'était le cas sur la Geforce FX de Nvidia, ce qui permettait à cette carte graphique d'obtenir de très bonnes performances dans le jeu DOOM3.

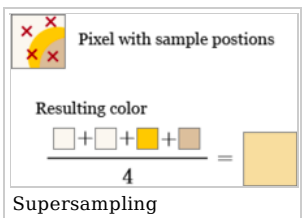
Antialiasing

Le ROP prend en charge l'anti-aliasing, une technologie qui permet d'adoucir les bords des objets. Le fait est que dans les jeux vidéos, les bords des objets sont souvent pixelisés, ce qui leur donne un effet d'escalier. Le filtre d'anti-aliasing rajoute une sorte de dégradé pour adoucir les bords des lignes.



Anti-aliasing demo

Types d'anti-aliasing



Supersampling

Il existe un grand nombre de techniques d'anti-aliasing différentes. Toutes ont des avantages et des inconvénients en terme de performances ou de qualité d'image. La première de ces technique, le SSAA - **Super Sampling Anti Aliasing** - calcule l'image à une résolution supérieure, avant de la réduire. Par exemple, si je veux rendre une image en 1280*1024, la carte graphique va calculer une image en 2560 * 2048, avant de la réduire. Pour effectuer la réduction de l'image, notre ROP va découper l'image en blocs de 4, 8, 16 pixels, et va effectuer un "mélange" des couleurs de tout le bloc. Ce "mélange" est en réalité une série d'interpolations linéaires, comme montré dans le chapitre sur le filtrage des textures, mais avec des couleurs de fragments. Si vous regardez les options de vos pilotes de carte graphique, vous verrez qu'il existe plusieurs réglages pour l'antia-alising : 2X, 4X, 8X, etc. Cette option signifie que l'image calculé par la carte graphique contiendra respectivement 2, 4, ou 8 fois plus de pixels que l'image originale. Cette technique filtre toute l'image, y compris l'intérieur des textures, mais augmente la consommation de mémoire vidéo et de processeur (on calcule 4 fois plus de pixels).

Pour réduire la consommation de mémoire induite par le SSAA, il est possible d'améliorer celui-ci pour faire en sorte qu'il ne filtre pas toute l'image, mais seulement les bords des objets, seuls endroit où l'effet d'escalier se fait sentir. On parle alors de **Multi-Sampling Anti-Aliasing, abrévié en MSAA**. Avec le MSAA, l'image à afficher est rendue dans une résolution supérieure, mais les fragments sont regroupés en carrés qui correspondent à un pixel. Avec le SSAA, chaque

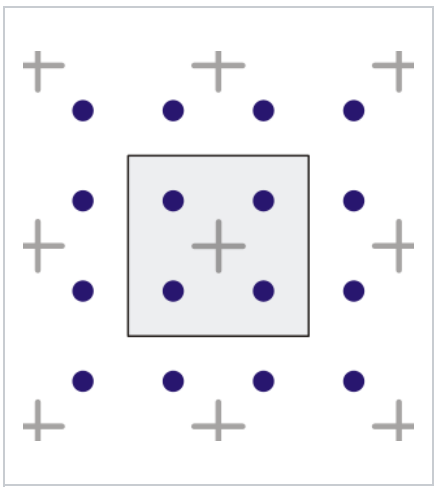
sous-pixel se verrait appliquer un morceau de texture. Avec le MSAA, les textures ne s'appliquent pas aux sous-pixels, mais à un bloc complet. La couleur finale dépend de la position du sous-pixel : est-il dans le triangle qui lui a donné naissance (à l'étape de rasterization), ou en dehors ? Si le sous-pixel est complétement dans le triangle, sa couleur sera celle de la texture. Si le sous-pixel est en-dehors du triangle, sa couleur est mise à zéro. Pour obtenir la couleur finale du pixel à afficher, le ROP va faire la moyenne des couleurs des sous-pixels du bloc. Niveau avantages, le MSAA n'utilise qu'un seul filtrage de texture par pixel, et non par sous-pixel comme avec le SSAA. Mais le MSAA ne filtre pas l'intérieur des textures, ce qui pose problème avec les textures transparentes. Pour résoudre ce problème, les fabricants de cartes graphiques ont crées diverses techniques pour appliquer l'anti-aliasing à l'intérieur des textures alpha.

Comme on l'a vu, le MSAA utilise une plus grande quantité de mémoire vidéo. Le **Fragment Anti-Aliasing**, ou FAA, cherche à diminuer la quantité de mémoire vidéo utilisée par le MSAA. Il fonctionne sur le même principe que le MSAA, à un détail prêt : il ne stocke pas les couleurs pour chaque sous-pixel, mais utilise à la place un masque. Dans le color-buffer, le MSAA stocke une couleur par sous-pixels, couleur qui peut prendre deux valeurs : soit la couleur calculée lors du filtrage de texture, soit la couleur noire (par défaut). A la place, le FAA stockera une couleur, et un petit groupe de quelques bits. Chacun de ces bits sera associé à un des sous-pixels du bloc, et indiquera sa couleur : 0 si le sous-pixel a la couleur noire (par défaut) et 1 si la couleur est à lire depuis le color-buffer. Le ROP utilisera ce masque pour déterminer la couleur du sous-pixel correspondant. Avec le FAA, la quantité de mémoire vidéo utilisée est fortement réduite, et la quantité de donnée à lire et écrire pour effectuer l'anti-aliasing diminue aussi fortement. Mais le FAA a un défaut : il se comporte assez mal sur certains objets géométriques, donnait naissance à des artefacts visuels.

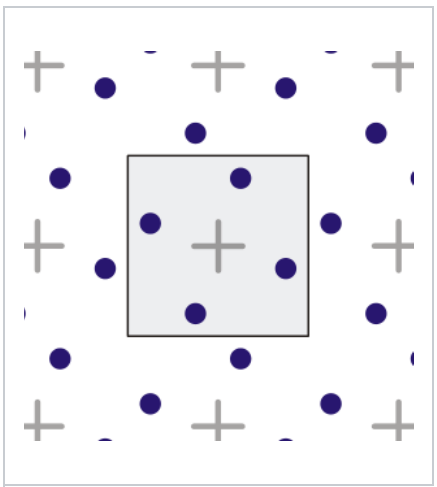
Position des sous-pixels

Un point important concernant la qualité de l'anti-aliasing concerne la position des sous-pixels sur l'écran. Comme vous l'avez vu dans le chapitre sur la rasterization, notre écran peut être vu comme une sorte de carré, dans lequel on peut repérer des points. Reste que l'on peut placer ces pixels n'importe où sur l'écran, et pas forcément à des positions que les pixels occupent réellement sur l'écran. Pour des pixels, il n'y a aucun intérêt à faire cela, sinon à dégrader l'image. Mais pour des sous-pixels, cela change tout. Toute la problématique peut se résumer en un phrase : où placer nos sous-pixels pour obtenir une meilleure qualité d'image possible.

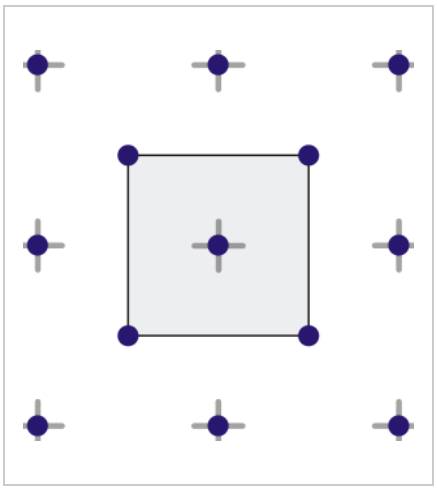
- La solution la plus simple consiste à placer nos sous-pixels à l'endroit qu'il occuperaient si l'image était réellement rendue avec la résolution simulée par l'anti-aliasing. Cette solution gère mal les lignes pentues, le pire cas étant les lignes penchées de 45 degrés par rapport à l'horizontale ou la verticale.
- Pour mieux gérer les bords penchés, on peut positionner nos sous-pixels comme suit. Les sous-pixels sont placés sur un carré penché (ou sur une ligne si l'on dispose seulement de deux sous-pixels). Des mesures expérimentales montrent que la qualité optimale semble être obtenue avec un angle de rotation de arctan(1/2) (26,6 degrés), et d'un facteur de rétrécissement de √5/2.
- D'autres dispositions sont possibles, notamment une disposition aléatoire ou de type Quincunx.



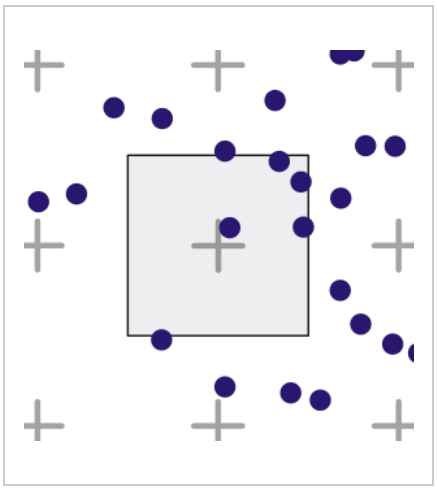
Antialiasing uniforme.



Antialiasing à grille tournée.



Antialiasing Quincunx.



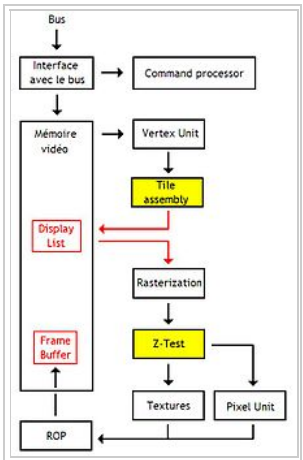
Antialiasing aléatoire.

L'élimination précoce des pixels cachés

Les cartes graphiques normales ne peuvent pas toujours éliminer les pixels cachés par des zones opaques de manière précoce. Vu que la détection des pixels masqués s'effectue dans les ROP, de nombreux pixels inutiles seront coloriés et éclairés par les pixels shaders. Or, la profondeur d'un pixel est connue dès la fin de l'étape de rasterisation, ce qui permet de détecter précocement si celui-ci sera ou non calculé. On peut penser que comparer les valeurs de profondeur en sortie du rasterizer serait une bonne chose, mais cela ne marcherait pas aussi bien que prévu. En effet, sur les cartes graphiques normales, l'ordre de soumission des triangles est aléatoire : un objet peut en cacher un autre sans que ces deux objets soient rendus consécutivement. Or, effectuer un test de profondeur précoce ne fonctionne que si les objets soumis à la carte graphique sont triés par leur valeur de profondeur, ce qui n'est jamais le cas. Et effectuer le tri des objets avant d'effectuer un test de profondeur serait nettement plus lent que d'effectuer le test de profondeur dans les ROP. Mais il existe des solutions alternatives, qui demandent d'adapter les cartes graphiques usuelles avec quelques optimisations, ou repenser totalement l'architecture des cartes graphiques. Dans ce chapitre, nous allons voir les deux solutions en détail.

Tiled rendering

La première solution demande d'utiliser une classe de carte 3D légèrement différente de celles vues précédemment. Sur ces architectures, l'écran/image à rendre est découpé en rectangles, rendus indépendamment, uns par uns. Ces rectangles sont appelés des tiles, d'où le nom d**architectures à tiles** donné à ce type de cartes graphiques. Sur celles-ci, l'élimination des pixels et triangles cachés s'effectue dès que la profondeur est disponible, c'est à dire à l'étape de rasterization. Sur ces architectures, le résultat des calculs géométriques est mémorisé en mémoire vidéo, avant d'être traité tile par tile. Chaque tile se voit attribuer la liste des triangles qu'elle contient : cette liste est appelée la Display List, et elle est enregistrée en mémoire vidéo. Par la suite, il suffit de rasterizer, placer les textures et exécuter les shaders chaque tile, avant d'envoyer le tout aux ROP. L'architecture globale d'une carte graphique à tiles change peu comparé à une carte à rasterization, si ce n'est que le rasterizer est modifié et qu'il est suivi d'une unité d'élimination des pixels cachés : l'Image Synthesis Processor, ou ISP.



Tiled rendering architecture.

Le rasterizer se voit ajouter un nouveau rôle : décider dans quelle tile se trouve un triangle. Pour cela, le rasterizer va calculer le rectangle qui contient un triangle (souvenez-vous le chapitre sur la rasterization), et va vérifier das quelle tile celui-ci est inclus : cela demande de faire quelques comparaison entre les sommets du rectangle et les sommets des tiles. L'Image Synthesis Processor remplace en quelque sorte le Z-Buffer et les circuits d'élimination des pixels cachés. Une architecture à tile a juste besoin d'un Z-Buffer pour la tile en cours de traitement, là où les cartes graphiques normales ont besoin d'un Z-buffer pour toute l'image. De plus, les tiles sont tellement petites que l'on peut stocker tout le Z-Buffer dans une mémoire tampon integrée dans l'ISP. Cette mémoire tampon réduit fortement les besoins en bande passante et en débit mémoire, ce qui rend inutile de nombreuses optimisations, comme la compression du Z-buffer.

Early-Z

Les architectures à base de Tiles ne sont pas la seule solution pour éviter le calcul des pixels cachés. Les concepteurs de cartes graphiques usuelles (sans tiled rendering) ont inventé des moyens pour détecter une partie des pixels qui ne seront pas visibles, avant que ceux-ci n'entrent dans l'unité de texture. Ces techniques sont des techniques d'**early-Z**. Mais ces techniques nuisent au rendu si les shaders peuvent bidouiller la profondeur ou la transparence d'un pixel. Pour éliminer tout problème, les drivers de la carte graphique doivent analyser les shaders et décider si le test de profondeur précoce peut être effectué ou non. Il existe plusieurs techniques d'early-Z, qui sont présentes depuis belle lurette dans nos cartes graphiques. Celles-ci peuvent être classées en deux catégories : le zmax, et le zmin. Il est parfaitement possible d'utiliser le zmax conjointement avec le zmin, ce qui donne des techniques hybrides.

Z-Max

Les deux techniques z-max et z-min découpent l'écran en tiles. Le Z-max consiste à vérifier si la tile à rendre est situé derrière des tiles déjà rendues pour la masquer cas échéant. Pour cela, il suffit de savoir quelle est la tile la plus profonde déjà rendue. Précisémment, il suffit de conserver la profondeur de cette tile et de faire les vérifications de profondeur. Le zmax consiste donc à vérifier si le triangle à rendre est situé derrière le pixel le plus profond de la tile. Ces techniques ont un gros défaut : il faut calculer la valeur maximale des pixels de la tile, ce qui demande de comparer les profondeurs de tous les pixels. Ce genre de chose s'effectue dans les ROPs, et demande parfois de lire les profondeurs depuis la mémoire vidéo...

La première technique de Z-Max est celle du **Hierarchical Z**. Dans les grandes lignes, cette technique consiste à conserver dans une mémoire cache (rarement en mémoire vidéo) une copie basse-résolution du tampon de profondeur, qui mémorise la valeur maximale de la profondeur pour chaque tile. Cette copie basse-résolution est mise à jour par les ROPs, en même temps que le Z-Buffer. Il existe d'autres techniques qui permettent d'éliminer ce genre de problèmes, comme le Depth Filter ou le Mid-texturing.

Z-Min

Avec le Z-min, on utilise la profondeur maximale des sommets du triangle dans les calculs. Cette valeur est comparée avec la valeur de profondeur minimale dans la tile. Si la profondeur du pixel à rendre est plus petite, cela veut dire que le pixel

n'est pas caché et qu'il n'y a pas besoin d'effectuer de test de profondeur dans les ROPs. Le calcul de la profondeur minimale de la tile est très simple : il suffit de mémoriser la plus petite valeur rencontrée et la mettre à jour à chaque rendu de pixel. Par besoin de lire toutes les profondeurs de la tile d'un seul coup, ou quoique ce soit d'autre, comme avec le zmax. Cette méthode est particulièrement adaptée aux rendus dans lesquels les recouvrements de triangles sont relativement rares. Il faut dire que cette méthode ne rejette pas beaucoup de pixels comparé à la technique du zmax. En contrepartie, elle n'utilise pas beaucoup de circuits comparé au zmax : c'est pour cela qu'elle est surtout utilisée dans les cartes graphiques pour mobiles.