Zeste de savoir

Les cartes graphiques

12 août 2019

Table des matières

I. Evolution dans le temps **4**

1. Les cartes d'affichage **6**
1.1. Mode texte . . . 6
1.1.1. Mémoire de caractères . . . 6
1.1.2. Contrôleur d'attribut . . . 6
1.1.3. Mémoires . . . 7
1.1.4. Le contrôleur graphique . . . 7
1.1.5. Contrôleur vidéo . . . 7
1.1.6. DAC . . . 8
1.1.7. Résumé . . . 8
1.2. Mode graphique . . . 9

2. Cartes accélératrices 2D **12**
2.1. Blitter . . . 12
2.1.1. Cas d'utilisation . . . 12
2.1.2. Opérations bit à bit . . . 13
2.2. Sprites et accélération matérielle du curseur de souris . . . 14
2.2.1. Sprites . . . 14
2.2.2. Curseur de souris matériel . . . 16

3. Les cartes accélératrices 3D **17**
3.1. Pipeline graphique . . . 17
3.1.1. Pipeline graphique . . . 17
3.2. Architecture d'une carte 3D . . . 19
3.2.1. Architecture de base . . . 19
3.2.2. Première génération . . . 20
3.2.3. Seconde génération . . . 21
3.2.4. Troisième génération . . . 22

II. Les composants d'une carte graphique **24**

4. Command Buffer et interface logicielle **27**
4.1. Le logiciel . . . 27
4.1.1. DirectX et OpenGL . . . 27
4.1.2. Pilotes de carte graphique . . . 27
4.2. Command Processor . . . 28
4.2.1. Commandes . . . 28
4.2.2. Parallélisme . . . 29

4.2.3. Synchronisation CPU . . . 29

5. Unités de gestion des sommets **31**
5.1. Input assembler . . . 31
5.1.1. Triangles strip . . . 31
5.1.2. Triangle fan . . . 32
5.1.3. Index buffers . . . 33
5.2. Transformation . . . 34
5.3. Eclairage . . . 35
5.3.1. Données vertices . . . 35
5.3.2. Calcul de l'éclairage . . . 36

6. Rasterization **37**
6.1. Triangle setup . . . 37
6.1.1. Fonction de contours . . . 37
6.1.2. Triangle traversal . . . 38
6.2. Interpolation des pixels . . . 38
6.2.1. Fragments . . . 39
6.2.2. Coordonnées barycentriques . . . 39
6.2.3. Perspective . . . 40

7. Unités de texture **41**
7.1. Filtrage . . . 41
7.1.1. Filtrage bilinéaire . . . 42
7.1.2. Mip-mapping . . . 45
7.1.3. Filtrage trilinéaire . . . 46
7.1.4. Filtrage anisotropique . . . 48
7.2. Compression . . . 48
7.2.1. Palette . . . 49
7.2.2. Compression par blocs . . . 49
7.3. Texture cache . . . 55
7.3.1. Stockage des textures en mémoire . . . 55
7.3.2. Multi-level texture cache . . . 56
7.3.3. Cohérence des caches . . . 56
7.4. Prefetching . . . 56

8. Les processeurs de Shaders **58**
8.1. Jeux d'instruction . . . 59
8.1.1. Processeurs SIMD . . . 59
8.1.2. Processeur VLIW . . . 62
8.1.3. Streams processors . . . 62
8.2. Microarchitecture . . . 64
8.2.1. Une forme limitée d'exécution dans le désordre . . . 64
8.2.2. Multi-threading matériel . . . 64
8.2.3. SIMT . . . 64

9. Render Output Target **66**
9.1. Test de visibilité . . . 66
9.1.1. Z-buffer . . . 67

9.1.2. Circuit de gestion de la profondeur . . . 68
9.2. Transparence . . . 69
9.2.1. Alpha Blending . . . 69
9.2.2. Color buffer . . . 70
9.2.3. Color ROP . . . 71
9.3. Anti-aliasing . . . 71
9.3.1. Types d'anti-aliasing . . . 71
9.3.2. Position des samples . . . 73

10. Elimination précoce des pixels cachés **76**
10.1. Tiled rendering . . . 76
10.2. Early-Z . . . 78
10.2.1. Z-Max et Z-Min . . . 79

III. Le multi-GPU **81**

11. Multi-GPU **83**
11.1. Split Frame Rendering . . . 84
11.1.1. Scan Line Interleave . . . 84
11.1.2. Checker Board . . . 85
11.1.3. Scan-lines contigues . . . 85
11.2. Alternate Frame Rendering . . . 86
11.2.1. Micro-stuttering . . . 86
11.2.2. Dépendances inter-frames . . . 87

Votre ordinateur contient forcément une carte graphique. Sans elle, rien ne s'afficherait à l'écran. Cette carte graphique est aussi celle qui est utilisée dans vos jeux vidéo, pour calculer les images à afficher à l'écran. Environ tous les deux ans, de nouvelles cartes graphiques sortent sur le marché, et sont vendues à prix d'or. Tous les nouveaux jeux vidéos ont besoin de plus en plus de puissance, et le marché des cartes graphiques marche à plein.

Lors de la présentation de ces nouvelles cartes graphiques, qui sont souvent faites par des sites comme Hardware.fr, Pcinpact, clubic, etc., n'avez-vous jamais tilté sur certains termes ? Du genre *shader core*, *stream processor*, *filtrage de textures*, etc. ? Vous vous êtes sûrement posés des questions sur certaines options dans vos drivers, comme la *vsync*, le *triple buffering*, etc. ? Savez-vous comment votre carte graphique fonctionne ? Ce qui se cache derrière vos jeux vidéos ? Comment les images qui vous sont affichées sont calculées ?

Si la réponse est non, alors vous êtes tombés au bon endroit. Ce tutoriel va vous expliquer ce que vos cartes graphiques ont dans le ventre. Nous allons commencer par une petite revue historique des premières cartes graphiques, qui datent de l'époque des tout premiers Windows, pour finir sur les cartes graphiques les plus récentes qui soient. Votre matériel n'aura plus le moindre secret pour vous !

Première partie

Evolution dans le temps

Entre les toutes premières cartes graphiques et les modèles le plus récent, il y a un monde : des années de progrès technologique les séparent. Le modèles les plus anciens ne permettaient même pas de gérer de la 3D, ni même de la 2D : ils savaient juste afficher du texte ou une image pré-calculée par le processeur sur un écran. Les modèles suivant ont ajouté une gestion matérielle de la 2D : ces cartes accélératrices 2D pouvaient tracer des lignes ou des figures géométriques sur une image, gérer l'arrière plan, afficher des *sprites*, etc. Ce n'est qu'après que les cartes graphiques ont étés dotées de capacités 3D. Dans cette partie, nous allons progresser dans l'ordre chronologique, des premières cartes au modèles 3D plus récents.

1. Les cartes d'affichage

1.1. Mode texte

Les premières cartes graphiques étaient très simples : elles fonctionnaient en mode texte. Avec ces cartes graphiques, l'écran était considéré comme un quadrillage, formé de lignes et de colonnes. À l'intersection de chaque ligne et de chaque colonne, on trouve un caractère – et non un pixel, comme sur les écrans normaux. Avec ces cartes graphiques, il était impossible de modifier des pixels individuellement : on ne pouvait gérer l'affichage qu'à partir de caractères.

Ce mode texte est toujours présent dans nos cartes graphiques actuelles. Il est encore utilisé par le BIOS au démarrage de l'ordinateur, ou par les écrans bleus de Windows.

1.1.1. Mémoire de caractères

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.). 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 une partie de cette table.

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.

La mémoire de caractères est très souvent une mémoire ROM. Dans les cas où la carte graphique permet à l'utilisateur de définir ses propres couleurs, cette ROM est couplée à une petite mémoire RAM. Au démarrage de la carte graphique, le contenu de la ROM est copié dans cette RAM, et c'est cette RAM qui est utilisée pour afficher les caractères. Dans cette mémoire, chaque caractère est représenté par une matrice de pixels.

1.1.2. Contrôleur d'attribut

Ces caractères se voient attribuer des informations en plus de leur code ASCII : à 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 : ATC (Attribute Controller), aussi appelé le contrôleur d'attributs.

1.1.3. Mémoires

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.

1.1.4. Le contrôleur graphique

Pour communiquer avec notre carte graphique, le processeur va envoyer les caractères à afficher un par un. Ces caractères sont alors accumulés dans le *text buffer* au fur et à mesure de leur arrivée. Pour communiquer de cette façon avec le processeur, la carte graphique incorpore un circuit qui lui permet de communiquer avec le bus qui la relie au processeur. Ce circuit varie fortement suivant le bus utilisé.

Cette communication s'effectue par des registres, reliés à un contrôleur d'entrée-sortie. Lorsque le processeur veut envoyer un ordre à la carte graphique, il écrit dans ces registres, et la carte graphique réagira en fonction des valeurs présentes dans ces registres.

Ces registres sont reliés à un **contrôleur graphique**, qui va lire le contenu de ces registres, et en déduire quoi faire : configurer la carte graphique, écrire un caractère en mémoire, ajouter un caractère défini par l'utilisateur dans la mémoire de caractères, etc.

1.1.5. Contrôleur vidéo

Vient ensuite le **CRTC** (*Cathod Ray Tube Controller*) ou contrôleur de tube cathodique. Celui-ci gère l'affichage sur l'écran proprement dit. Comme vous le savez, nos écrans d'ordinateurs fonctionnent par balayage : les pixels sont affichés les uns après les autres, en commençant par le pixel en haut à gauche, et en poursuivant ligne par ligne. Pour une carte en mode texte, ce contrôleur va donc aller chercher ces pixels les uns après les autres, en utilisant le contenu de la mémoire de caractères, et du *text buffer*.

L'affichage se faisait à une résolution qui était fixée une fois pour toute, souvent 80 pixels de long pour 25 pixels de hauteur. Il s'agit d'un standard, créé en même temps que la toute première carte graphique monochrome : la **Monochrome Display Adaptater**.

Pour commencer, il doit se souvenir à quel pixel il en est rendu. Pour cela, il contient deux registres : un pour indiquer la ligne du pixel courant sur l'écran, et l'autre la colonne. À partir du contenu de ces deux registres, il va déduire à quel caractère cela correspond dans le *text buffer*. Pour cela, une partie des bits de ces deux registres sera utilisée. Ce caractère sera alors lu depuis le *text buffer*, et envoyé 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.

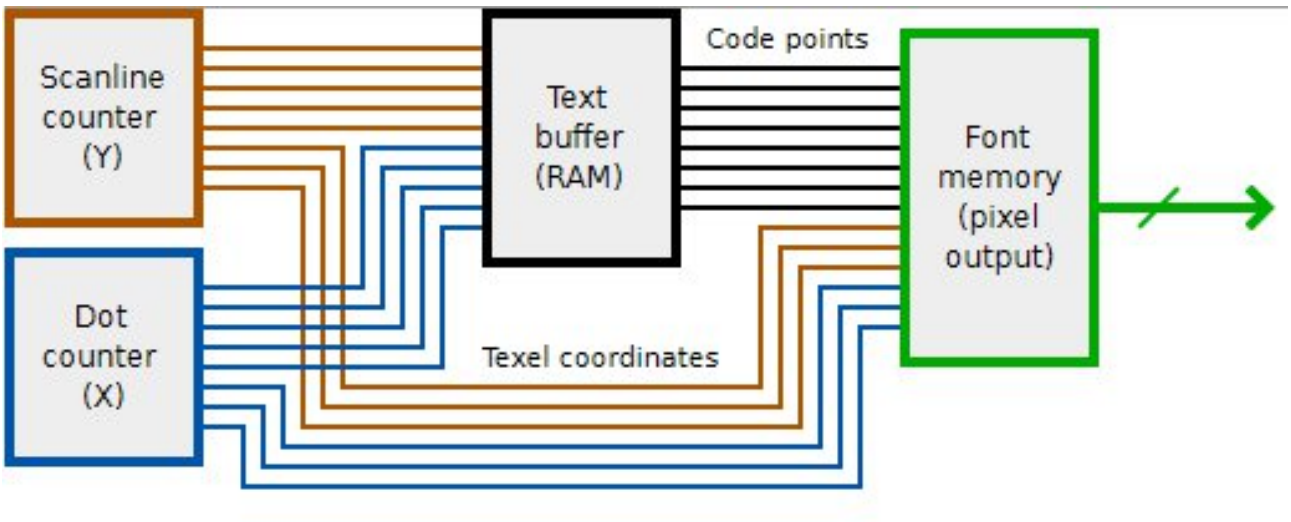

FIGURE 1.1. – Architecture simplifiée du CRTC

Évidemment, nos deux registres de ligne et de colonne sont incrémentés régulièrement afin de passer au pixel suivant.

1.1.6. DAC

Enfin, notre pixel étant disponible, celui-ci est envoyé à l'écran. Ceci dit, les écrans assez anciens fonctionnent en analogiques : ils ne comprennent par les données codées en binaire. Résultat : il faut faire la 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.

1.1.7. Résumé

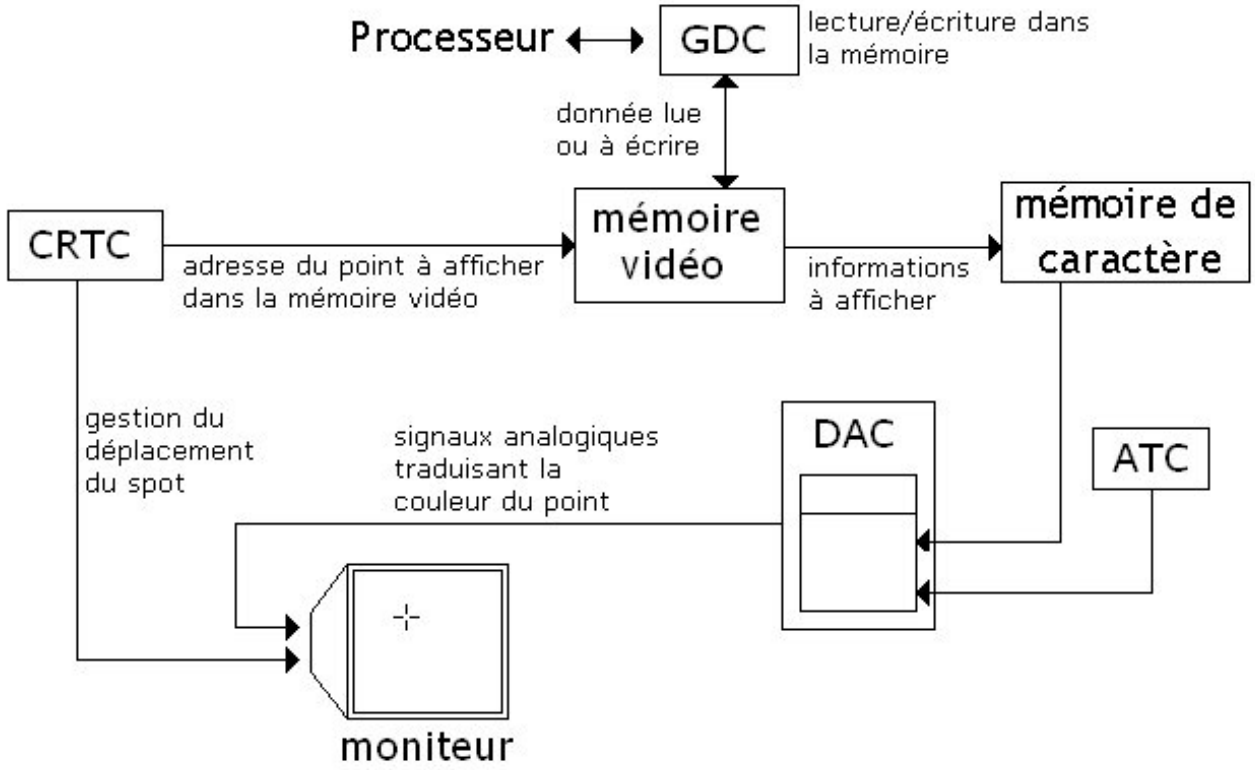

FIGURE 1.2. – Architecture d'une carte 2D en mode texte

1.2. Mode graphique

Nos cartes en mode texte sont assez rudimentaires. Aussi, celles-ci 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.

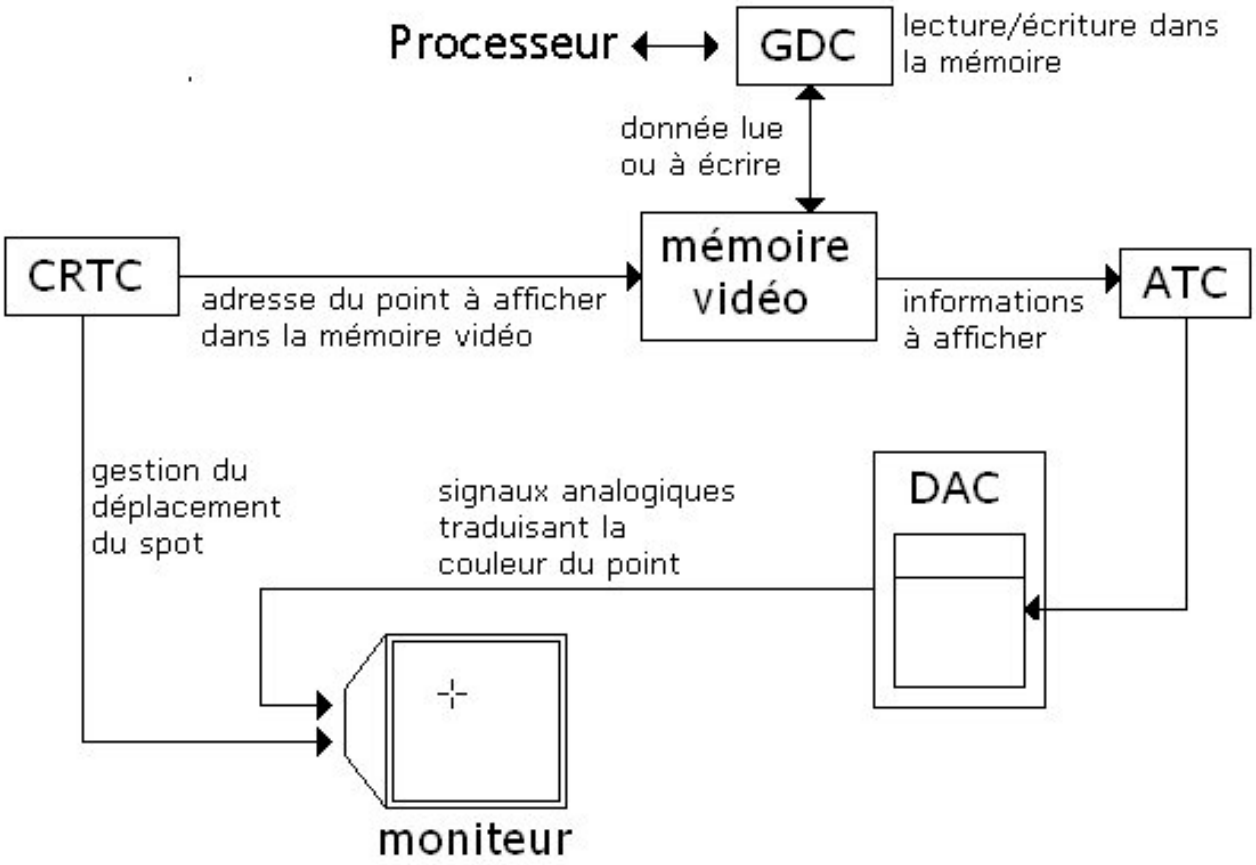

FIGURE 1.3. – Architecture d'une carte 2D en mode graphique

On remarque que la mémoire de caractères a disparu, ce qui est normal : la carte graphique ne gère plus les caractères. 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 devient programmable, et peu maintenant gérer différentes résolutions. On peut ainsi changer la résolution de l'écran. Pour cela, ce contrôleur contient des registres : pour configurer ce contrôleur, il suffit d'écrire les valeurs adéquates dans ceux-ci.

De plus, 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 était prédéterminée par la carte vidéo, et se voyait attribuer un numéro.

Seul problème : les écrans ne gèrent que des données au format RGB. C'est à dire que chaque pixel se voit attribuer une teinte de bleu, une teinte de rouge, et une teinte de vert, chacune codée par un entier : plus cet entier est grand, plus le pixel est lumineux (0 correspond au noir, et la valeur maximale de l'entier au blanc). Ces cartes graphiques devaient donc effectuer la conversion. Pour cela, ces cartes graphiques disposaient d'un circuit, la **Color Look-up Table**, qui traduisait ces numéros de couleur en couleur au format RGB.

Dans le cas le plus simple, il s'agissait d'une ROM qui mémorisait la couleur RGB pour chaque numéro : on obtenait la couleur au format RGB en envoyant notre numéro de couleur sur son bus d'adresse, et en récupérant la couleur RGB sur le bus après lecture. 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**.

Le DAC est toujours fidèle au poste. Seulement, celui-ci peut maintenant gérer les couleurs. À partir de données RGB, RGBA, etc., il peut maintenant fournir des signaux analogiques gérant plusieurs couleurs.

2. Cartes accélératrices 2D

Avec l'arrivée des jeux vidéo, les performances ont commencé à poser quelques problèmes. Les premiers jeux vidéos étaient tous des jeux 2D qui donnaient l'illusion de la 3D. Les premières cartes accélératrices ont permis un grand bond en avant : au lieu de laisser tout le boulot au processeur, ces cartes graphiques faisaient une partie des calculs.

Les cartes graphiques 2D ont d'abord commencé par accélérer le traitement de figures géométriques simples : lignes, segments, cercles, ellipses, etc. Dans certains cas, elles pouvaient colorier une de ces figures : remplir un disque de rouge, remplir un rectangle en orange, etc. Ces fonctions permettaient d'accélérer les premières interfaces graphiques des systèmes d'exploitation.

Les cartes graphiques actuelles supportent aussi d'autres technologies, comme des techniques d'accélération du rendu des polices d'écriture, une accélération du *scrolling*, des accélérateurs d'affichage pour les bibliothèques GDI (utilisées sous Windows), ou encore un support matériel du curseur de la souris. Si vous ne me croyez pas, lancez Windows ou Linux sans les pilotes de la carte graphique...

2.1. Blitter

Un peu plus tard, elles ont introduit un composant très utile pour l'époque : le ***blitter***. Dans sa version la plus simple, ce *blitter* sert à accélérer les copies de données d'un endroit de la mémoire vidéo à un autre.

2.1.1. Cas d'utilisation

Ce genre de copie arrive souvent lorsqu'on doit *scroller*, ou qu'un objet 2D se déplace sur l'écran. Déplacer une fenêtre sur votre bureau est un bon exemple : le contenu de ce qui était présent sur l'écran doit être déplacé vers le haut ou vers le bas. Dans la mémoire vidéo, cela correspond à une copie des pixels correspondant de leur ancienne position vers la nouvelle.

Cela a aussi des applications dans les jeux en 2D. La base d'un rendu en 2D, c'est de superposer des images 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 superposées sur l'arrière-plan au bon endroit sur l'écran, ce qui se traduit par une copie des pixels de l'image aux bons endroits dans la mémoire.

Créer un circuit juste pour faire des copies en mémoire et quelques opérations bit à bit peu sembler bizarre. Mais il y a de bonnes raisons à cela. Lorsque ce circuit a été inventé, le *screen buffer* des cartes graphiques était accessible par le processeur, qui pouvait lire et écrire directement dedans. Ces 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 s'en chargeait.

Ce morceau de programme répétait une série d'instructions en boucle pour copier les données pixel par pixel.

Répéter ces instructions nécessitait de recharger celles-ci depuis la mémoire à chaque utilisation, ce qui prenait du temps. A contrario, un *blitter* ne souffre pas de ce genre de problèmes. Il s'agit d'un circuit qui est conçu pour ce genre de tâches : il n'accède à la mémoire que pour transférer des données, sans besoin de charger la moindre instruction. Le gain en performance est assez appréciable.

2.1.2. Opérations bit à bit

Ceci dit, un *blitter* possède d'autres fonctionnalités. Il peut effectuer une opération bit à bit entre le bloc de données à copier et le bloc de destination. Le résultat de cette opération est ensuite enregistré en mémoire. Généralement, le *blitter* peut effectuer des négations, des `ET` bit à bit, des `OU` bit à bit, parfois des `XOR` bit à bit.

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 :

Figure 2.1. – Image de Pacman

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. Par exemple, le masque du pacman est celui-ci :

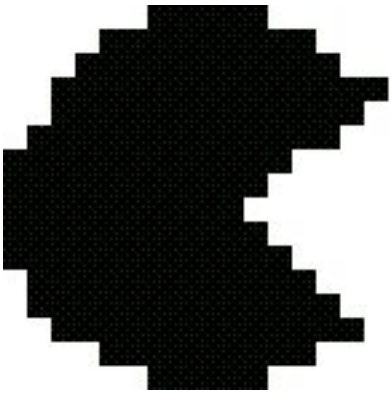

FIGURE 2.2. – Masque de Pacman

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.

2.2. Sprites et accélération matérielle du curseur de souris

Le *blitter* était concurrencé par une autre technique en vigueur : les *sprites* matériels. Avec cette technique, nul besoin de *blitter* pour superposer une image sur une autre. Ces *sprites* ne sont ni plus ni moins que les images à superposer à l'arrière-plan. Ceux-ci sont stockés dans de petites mémoires RAM intégrées dans la carte graphique.

2.2.1. Sprites

Avec la technique des *sprites*, ces *sprites* ne sont pas ajoutés sur l'arrière-plan : celui-ci n'est pas modifié. 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.

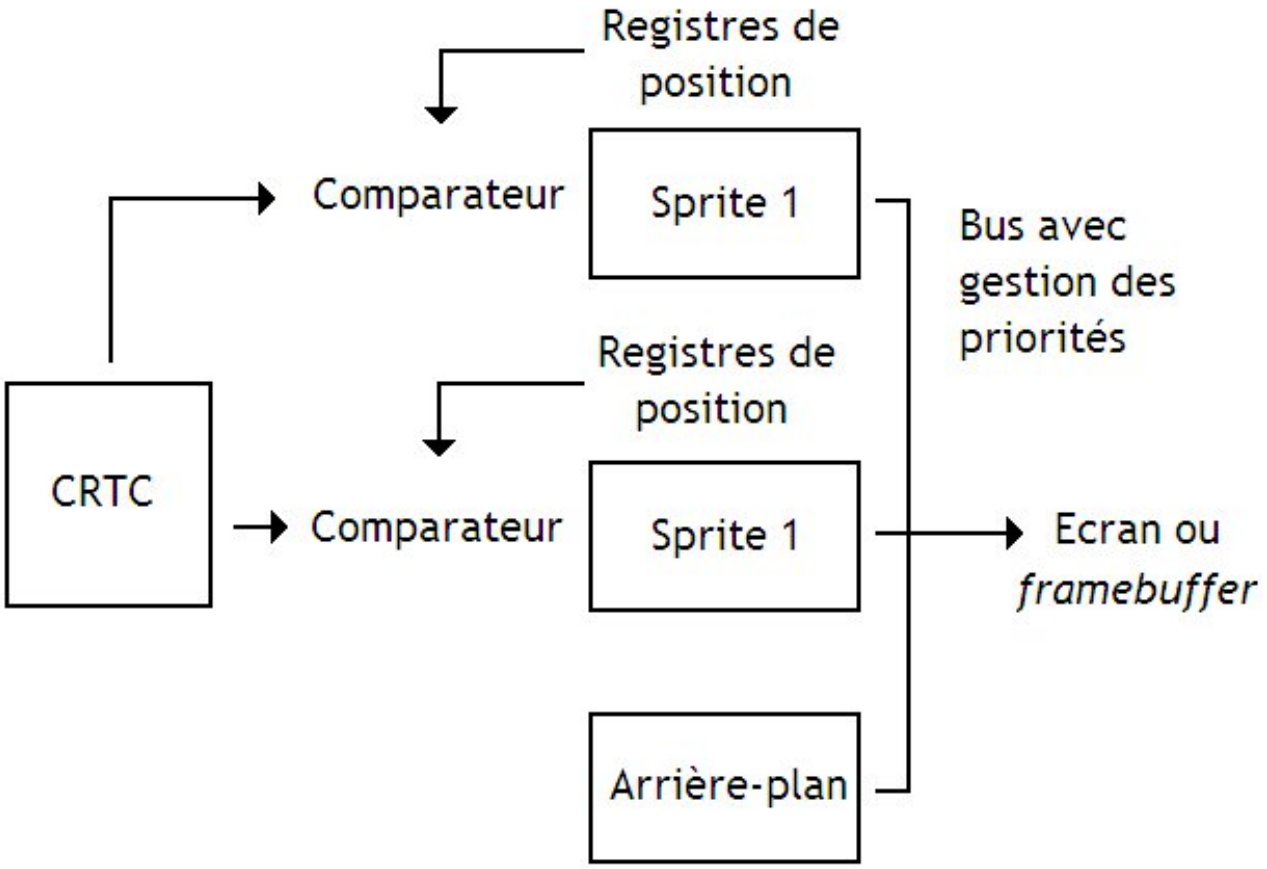

FIGURE 2.3. – Circuit de gestion matérielle des sprites

Notre carte contient donc des RAM pour stocker les *sprites*, plus une grosse RAM pour l'arrière-plan. Toutes ces RAMs sont connectées au RAMDAC ou au *framebuffer* via un bus, un ensemble de fils électriques sur lequel un seul composant peut envoyer des données à la fois.

Pour chacune des RAM associée au *sprite*, on trouve deux registres permettant de mémoriser la position du *sprite* à l'écran : un pour sa coordonnée X, et un autre pour sa coordonnée Y. Lorsque le CRTC demande à afficher le pixel à la position (X , Y), chacun des registres de position est alors comparé à la position envoyée par le CRTC. Si aucun *sprite* ne correspond, les mémoires des *sprites* sont déconnectées du bus. 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 *framebuffer* ou au RAMDAC.

Si plusieurs *sprites* doivent s'afficher en même temps, alors le bus décide de choisir un des *sprites*, et déconnecte tous les autres du bus. Pour cela, divers mécanismes d'arbitrage sont implantés dans le bus (une simple *daisy chain* peut suffire). Grosso-modo, certains registres seront prioritaires sur les autres, et ces priorités sont fixées une fois pour toute dans le matériel qui gère le bus. La gestion de recouvrements est donc gérée par le programmeur, qui décide dans quels registres placer les images à afficher suivant leur priorités.

2.2.2. Curseur de souris matériel

Cette technique des *sprites* est présente dans notre cartes graphique, mais dans un cadre particulièrement spécialisé : la prise en charge du curseur de la souris !

Les cartes graphiques contiennent un ou plusieurs *sprites*, qui représentent chacun un curseur de souris. La carte graphique permet de configurer la position de ce *sprite* grâce à 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.

Cette technique est présente dans toutes les cartes graphiques actuelles. Pour en avoir la preuve, testez une nouvelle machine sur laquelle les drivers ne sont pas installés, et bougez le curseur : effet *lag* garantit !

3. 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**.

3.1. Pipeline graphique

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 notre scène 3D, on trouve un objet spécial : la **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 géométriques, 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.

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.

3.1.1. 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, le cas le plus simple ne demandant que trois 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.

Chacune de ces étapes est elle-même découpée en plusieurs sous-étapes. Toutes ces sous-étapes doivent s'exécuter dans un ordre bien précis. L'ensemble de ces étapes et sous-étapes forme ce qu'on appelle le **pipeline graphique**.

3.1.1.1. Traitement des vertices

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 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.

3.1.1.2. 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* : les triangles qui, du point de vue de la caméra, sont situés derrière une surface géométrique, sont oubliés. Ceux-ci ne sont simplement pas visibles depuis la caméra, donc autant les virer.

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.

3.1.1.3. 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.

3.1.1.4. Bilan

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.

3.2. 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. Les toute premières cartes graphiques contenaient des circuits pour accélérer une partie des étapes vues au-dessus. Au fil du temps, de nombreux circuits furent ajoutés, afin de déporter un maximum de calculs du CPU vers la carte graphique.

3.2.1. Architecture de base

Néanmoins, toute carte graphique contient obligatoirement certains circuits :

— la mémoire vidéo ;
— les circuits de communication avec le bus ;
— le *command buffer*.

La carte graphique a besoin d'une mémoire RAM : la **mémoire vidéo**. Cette mémoire vidéo, est très proche des mémoires RAM qu'on trouve sous forme de barrettes dans nos PC, à quelques différences près : la mémoire vidéo peut supporter un grand nombre d'accès mémoire simultanés, et elle est optimisée pour accéder à des données proches en mémoire. Dans le cas le plus simple, elle sert simplement de *Framebuffer* : elle stocke l'image à afficher à l'écran. Au fil du temps, elle s'est vu ajouter d'autres fonctions : stocker les textures et les vertices de l'image à calculer, ainsi que divers résultats temporaires.

La carte graphique communique via un **bus**, un vulgaire tas de fils qui connectent la carte graphique à la carte mère. Les premières cartes graphiques utilisaient un bus nommé ISA, qui fût rapidement remplacé par le bus PCI, plus rapide. Viennent ensuite le bus AGP, puis le bus PCI-Express.

Ce bus est géré par un **contrôleur de bus**, un circuit qui se charge d'envoyer ou de réceptionner les données sur le bus. Il contient quelques registres dans lesquels le processeur pourra écrire ou lire, afin de lui envoyer des ordres du style : j'envoie une donnée, transmission terminée, je ne suis pas prêt à recevoir les données que tu veux m'envoyer, etc.

Les anciennes cartes graphique pouvaient lire ou écrire directement dans la mémoire RAM, grâce à certaines fonctionnalités du bus AGP. Mais généralement, les données sont copiées depuis la mémoire RAM vers la mémoire vidéo, en passant par le bus. Cette copie est effectuée par un circuit spécialisé : le **contrôleur DMA**, qui permet d'échanger des données entre mémoire vidéo et mémoire RAM sans devoir utiliser le processeur. Il est souvent intégré dans le contrôleur de bus.

Tout les traitements que la carte graphique doit effectuer sont envoyés par un programme, le pilote de la carte graphique, sous la forme de commandes. Ces commandes sont des ordres, que la carte graphique doit exécuter. Elles sont stockées temporairement dans une zone de mémoire, le **command buffer**, avant d'être interprétées par un circuit spécialisé : le **command processor**. Celui-ci est chargé de piloter les circuits de la carte graphique.

3.2.2. Première génération

Les toutes premières cartes graphiques contenaient simplement des circuits pour gérer les textures, ainsi qu'un *framebuffer*. 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.

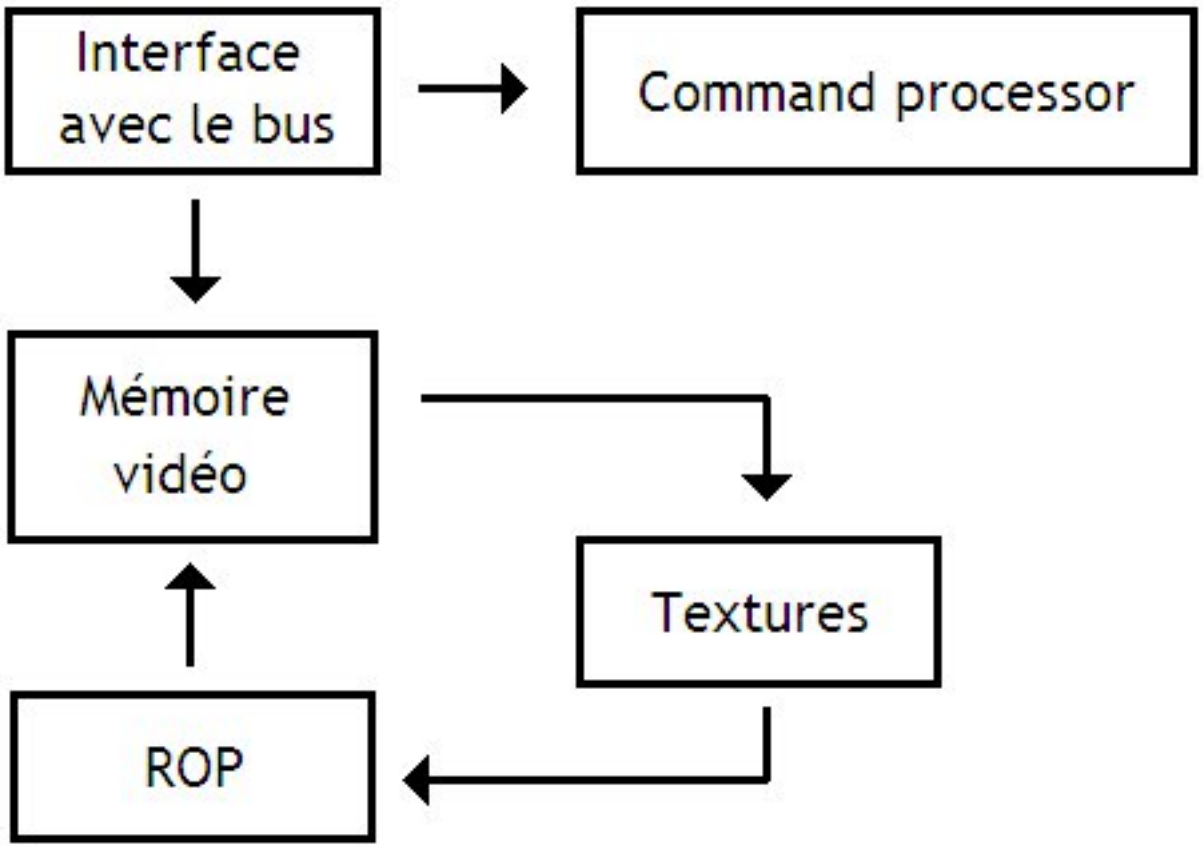

FIGURE 3.1. – Architecture d'une carte 3D de première génération

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.

3.2.3. Seconde génération

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

- d'un circuit pour manipuler les vertices et la géométrie ;
- d'un circuit pour appliquer des textures ;
- d'un circuit pour effectuer la *rasterization*, le *clipping* et le *culling* ;
- d'un circuit pour les opérations finales, comme le brouillard, la gestion de la transparence, de la profondeur, etc : le *raster operation pipeline*, ou ROP.

Les données circulent d'un circuit à l'autre dans un ordre bien précis. Cela permet d'enchaîner les sous-étapes de traitement du pipeline graphique dans l'ordre voulu. Entre ces différentes unités, on trouve souvent des mémoires pour mettre en attente les vertices ou les pixels, au cas où une unité est trop occupée : les unités précédentes peuvent continuer à faire leurs calculs et accumulent leurs résultats dans ces mémoires tampons, résultats utilisables quand l'unité sera libre.

Nous verrons ces circuits en détail dans la suite du tutoriel. Dans les grandes lignes, une carte graphique ressemble à ceci :

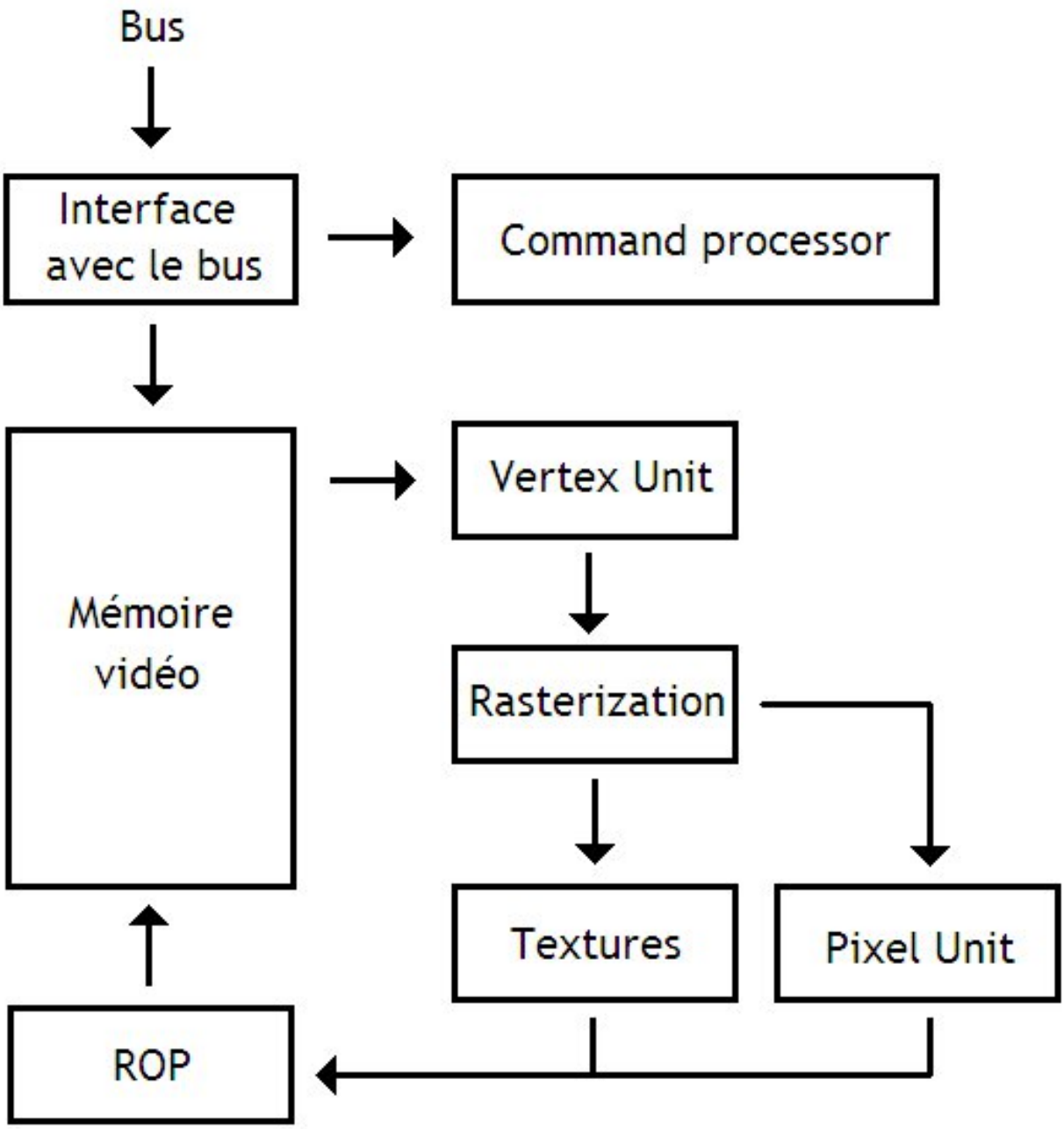

FIGURE 3.2. – Architecture d'une carte 3D de deuxième génération

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*).

3.2.4. Troisième génération

Enfin, les circuits qui gèrent les calculs sur les pixels et sur les vertices sont devenus des processeurs programmables. Au tout début, seuls les traitements sur les vertices étaient programmables. C'était le cas sur les NVIDIA's GeForce 3, GeForce4 Ti, Microsoft's Xbox, et les ATI's Radeon

8500. Puis, les cartes suivantes ont permis de programmer les traitements sur les pixels : certains processeurs s'occupaient des vertices, et d'autres des pixels. Mais sur les cartes graphiques récentes, les processeurs peuvent traiter indifféremment pixels et vertices.

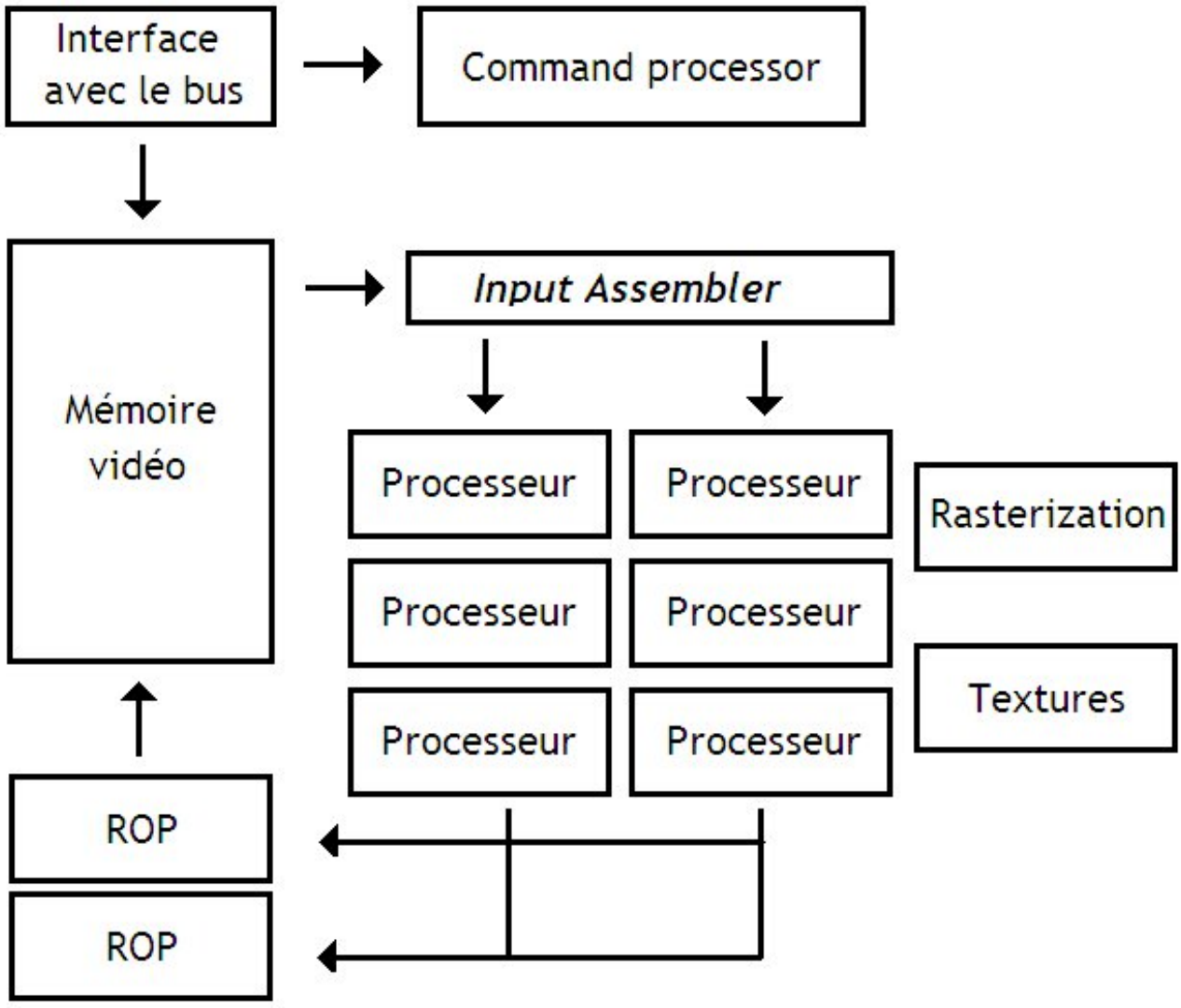

FIGURE 3.3. – Architecture d'une carte 3D de troisième génération

Pour l'avenir, on suppose que certains autres circuits deviendront programmables. Les circuits chargés de la *rasterization*, ainsi que les ROP sont de bons candidats. Par contre, les circuits de textures ne changeront pas avant un moment.

Deuxième partie

Les composants d’une carte graphique

Comme on l'a vu dans les chapitres précédents, une carte graphique est composé de plusieurs composants :

— une mémoire vidéo ;
— une interface avec le bus qui échange des information avec le processeur et la RAM ;
— un command buffer qui interprète les demandes en provenance du processeur et pilote les autres composants ;
— un circuit de gestion des sommets ;
— un circuit de rasterization (passage d'un monde en 3D à un écran 2D) ;
— un circuit de lecture des textures ;
— un circuit d'enregistrement de l'image finale en mémoire : le ROP.

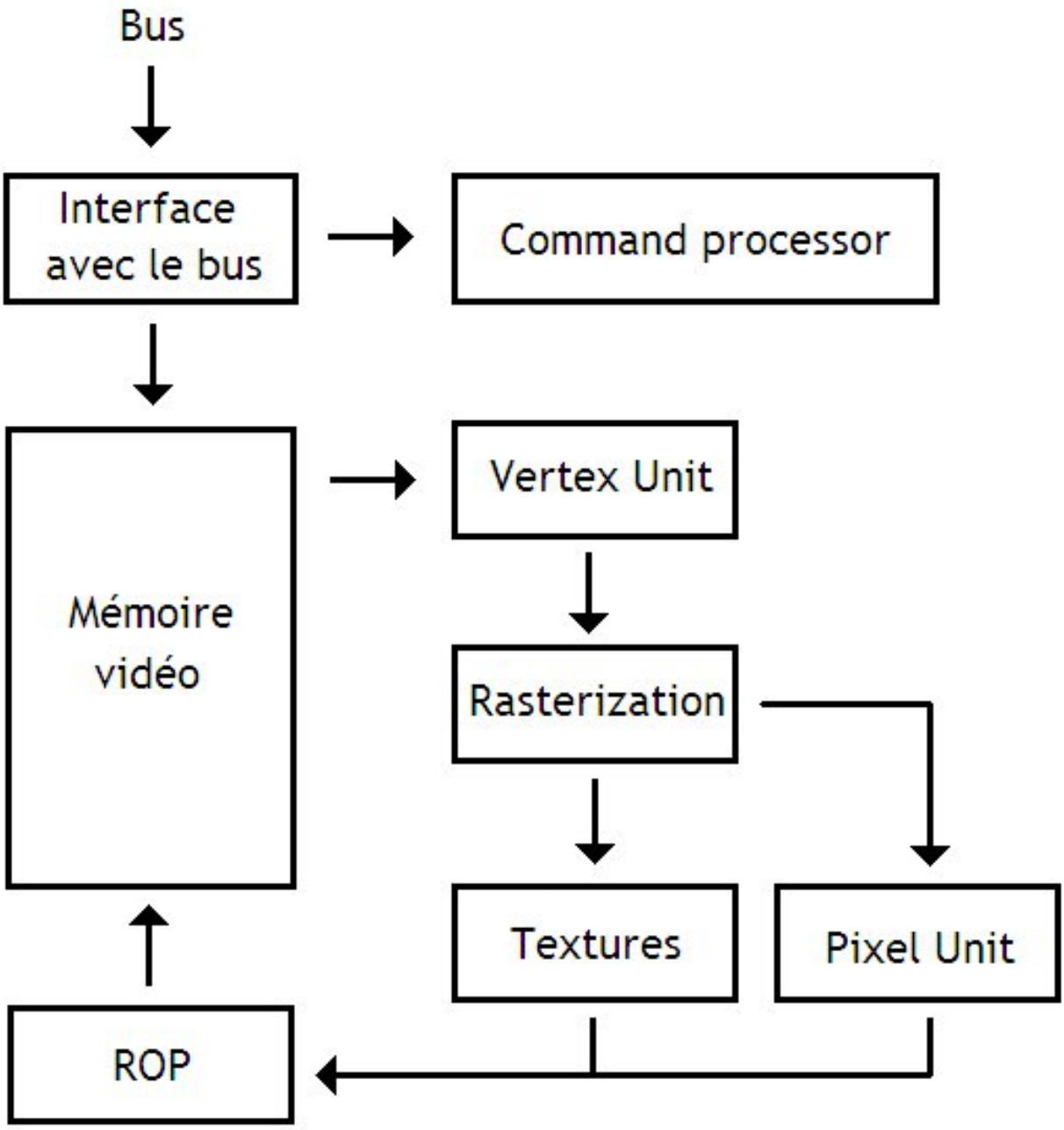

FIGURE 3.4. – Architecture d'une carte graphique

Sur les cartes modernes, on trouve aussi des processeurs de shaders, dont certains remplacent parfois les circuits de gestion des sommets.

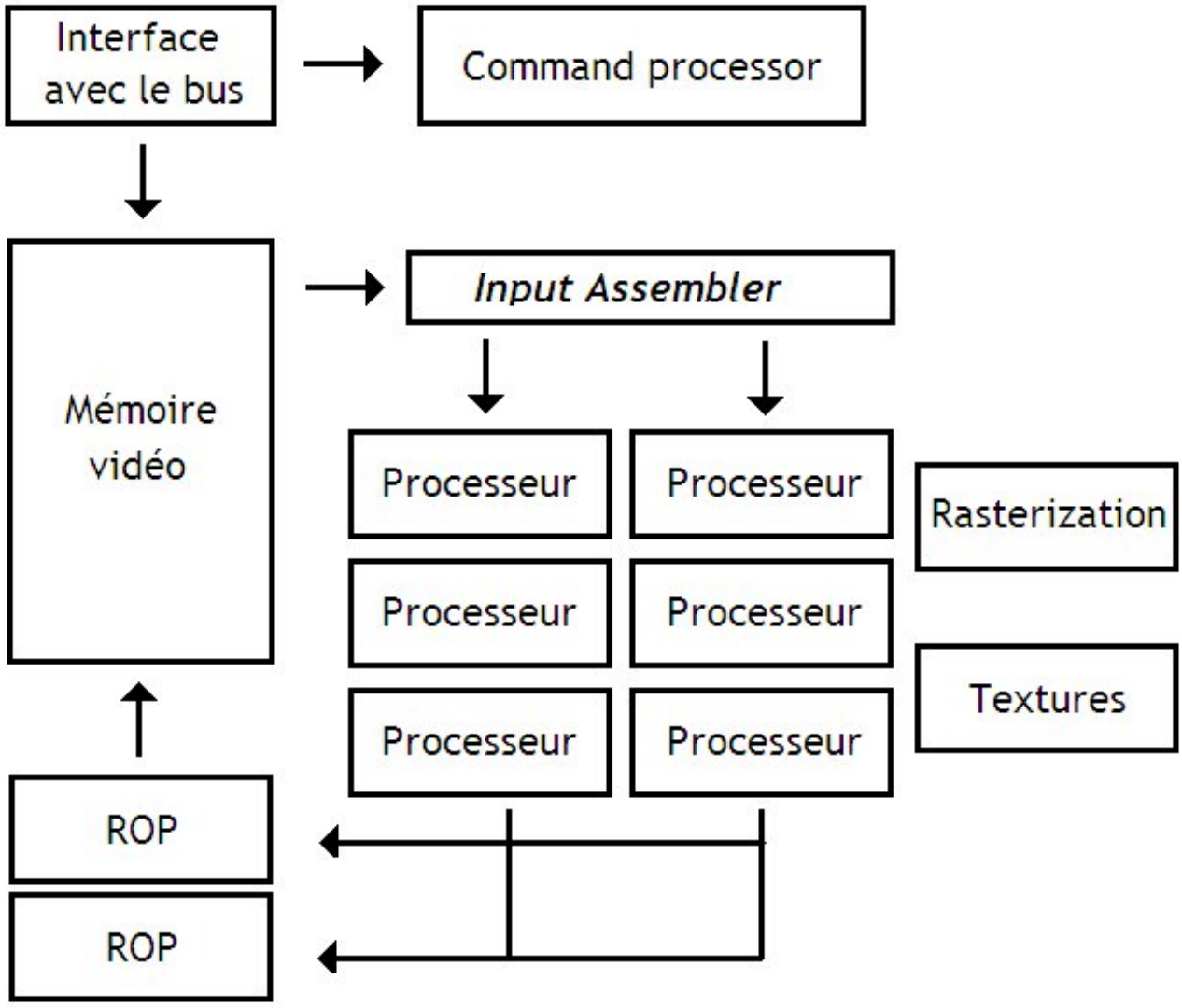

FIGURE 3.5. – Carte graphique moderne

Dans ce qui va suivre, nous allons voir chaque composant dans le détail.

4. Command Buffer et interface logicielle

Une carte graphique est un périphérique comme un autre, connecté sur la carte mère (sauf pour certaines cartes graphiques intégrées). Le processeur doit envoyer des informations à la carte graphique pour que celle-ci fasse son travail. Voyons tout ce qui se passe durant ces transferts.

4.1. Le logiciel

La carte graphique accélére les jeux vidéos, les applications de conception assistée par ordinateur (solidworks), ou de rendu d'images 3D (blender, maya, etc). Elle peut aussi accélérer le traitement de l'affichage 2D : essayez d'utiliser Windows sans pilote de carte graphique, vous verrez de quoi je parle. Bref, il y a forcément un programme, ou un logiciel qui utilise notre carte graphique et lui délègue ses calculs.

4.1.1. DirectX et OpenGL

Pour déléguer ses calculs à la carte 3D, l'application pourrait communiquer directement avec la carte graphique, en écrivant dans ses registres et dans sa mémoire vidéo. Seul problème : le programme ne pourra communiquer qu'avec un ou deux modèles de cartes, et la compatibilité sera presque inexistante.

Pour résoudre ce problème, les concepteurs de systèmes d'exploitations et de cartes graphiques ont inventé des **API 3D**, des bibliothèques qui fournissent des "sous-programmes" de base, des fonctions, que l'application pourra exécuter au besoin. De nos jours, les plus connues sont DirectX, et OpenGL.

4.1.2. Pilotes de carte graphique

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.

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 portion de la mémoire : le ***ring buffer***. Ce *ring buffer* est ce qu'on appelle une file, une zone de mémoire dans laquelle on stocke des données dans l'ordre d'ajout.

Si le *ring buffer* est plein, le driver n'accepte plus de demandes en provenance des applications. Un *ring buffer* plein est généralement mauvais signe : cela signifie que la carte graphique est trop lente pour traiter les demandes qui lui sont faites. Par contre, il arrive que le *ring buffer* soit vide : dans ce cas, c'est simplement que la carte graphique est trop rapide comparé au processeur, qui n'arrive alors pas à donner assez de commandes à la carte graphique pour l'occuper suffisamment.

4.2. Command Processor

Les commandes envoyées par le pilote de la carte graphique sont gérées par le **command processor**.

4.2.1. Commandes

Certaines de ces commandes vont demander à la carte graphique d'effectuer une opération 2D, d'autres une opération 3D, et d'autres une opération concernant l'accélération vidéo. Vous vous demandez à quoi peuvent bien ressembler ces commandes. Prenons les commandes de la carte graphique AMD Radeon X1800.

Voici les commandes 2D :

Commandes 2D	Fonction
PAINT	Peindre des rectangle d'une certaine couleur
PAINT_MULTI	Peindre des rectangles (pas les mêmes paramètres que PAINT)
BITBLT	Copie d'un bloc de mémoire dans un autre
BITBLT_MULTI	Plusieurs copies de blocs de mémoire dans d'autres
TRANS_BITBLT	Copie de blocs de mémoire avec un masque
NEXTCHAR	Afficher un caractère avec une certaine couleur
HOSTDATA_BLT	Écrire une chaine de caractère à l'écran ou copier une série d'image bitmap dans la mémoire vidéo
POLYLINE	Afficher des lignes reliées entre elles
POLYSCANLINES	Afficher des lignes
PLY_NEXTSCAN	Afficher plusieurs lignes simples
SET_SCISSORS	Utiliser les scissors ?
LOAD_PALETTE	Charger la palette pour affichage 2D

D'autres commandes servent à synchroniser le processeur et le GPU :

Commandes de synchronisation	Fonction

NOP	Ne rien faire
WAIT_SEMAPHORE	Attendre la synchronisation avec un sémaphore
WAIT_MEM	Attendre que la mémoire vidéo soit disponible et inoccupée par le CPU

D'autres commandes servent pour l'affichage 3D : afficher une image à partir de paquets de vertices, ou préparer le passage d'une image à une autre. Et enfin, certaines commandes servent pour l'accélération des vidéos.

4.2.2. Parallélisme

Sur les cartes graphiques modernes, le *command processor* 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 *command processor* a un désavantage : celui-ci doit gérer les synchronisations entre commandes.

4.2.3. Synchronisation CPU

Avec un *command buffer*, le processeur et la carte graphique ne fonctionnent pas au même rythme, et cela pose problèmes. Par exemple, Direct X et Open Gl décident d'allouer ou de libérer de la mémoire vidéo pour les textures ou les vertices. Or, Direct X et Open Gl ne savent pas quand le rendu de l'image se termine. Mais comment éviter d'enlever une texture de la mémoire tant que les commandes qui utilisent celle-ci 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.

4.2.3.1. Fences

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.

4.2.3.2. Sémaphores

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.

5. Unités de gestion des sommets

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 ;
— et les circuits de traitement de vertices.

5.1. Input assembler

Avant leur traitement, les vertices sont stockées dans un tableau en mémoire vidéo : le ***vertex buffer***. L'***input assembler*** va charger les vertices dans les unités de traitement des vertices. Pour ce faire, il a besoin d'informations mémorisées dans des registres :

— l'adresse du *vertex buffer* en mémoire ;
— sa taille ;
— du type des données qu'on lui envoie (vertices codées sur 32 bits, 64, 128, etc).

L'*input assembler* peut aussi gérer des lectures simultanées dans plusieurs *vertex buffers*, si la mémoire vidéo le permet.

5.1.1. Triangles strip

Il arrive qu'une vertice soit 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 *vertex buffer* : 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.

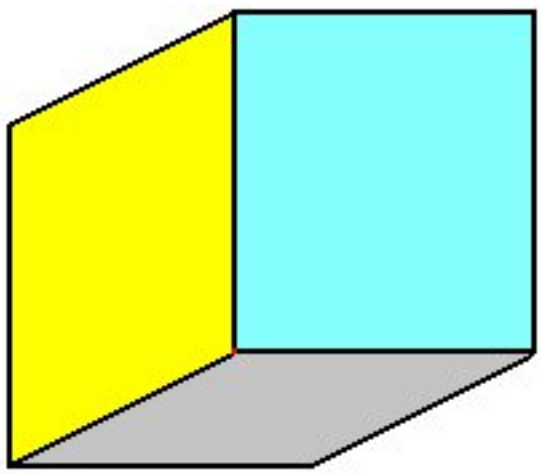

FIGURE 5.1. – Cube en 3D

La technique des *triangles strip* permet d'optimiser le rendu de triangles placés en série, qui ont une arête et deux sommets en commun.

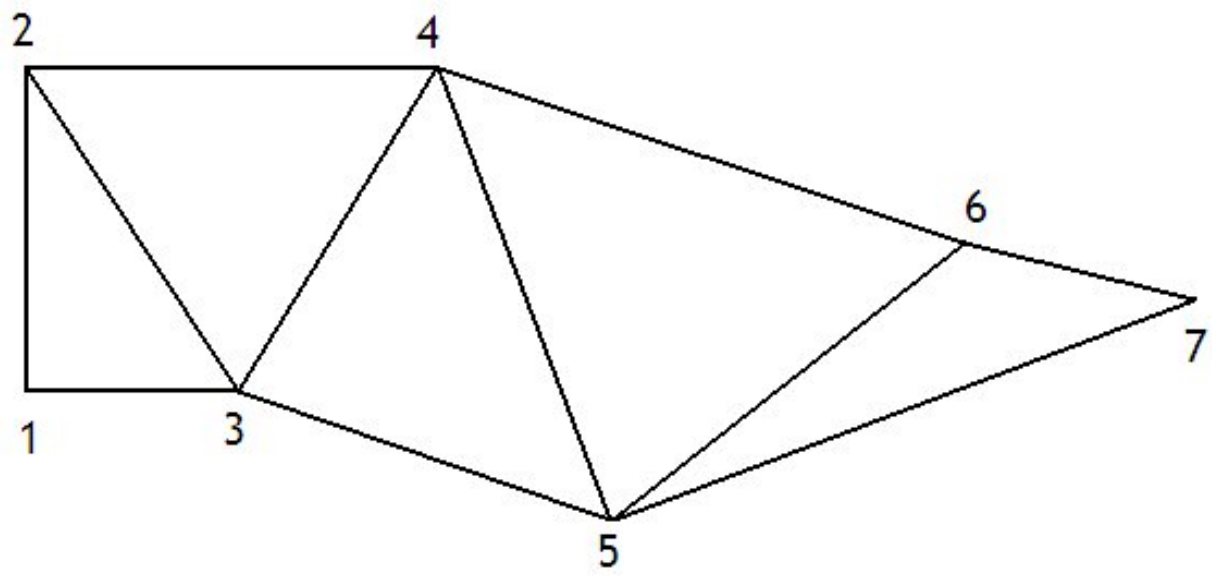

FIGURE 5.2. – Triangle Strip

L'optimisation consiste à ne stocker complètement que le premier triangle le plus à gauche, les autres triangles étant codés avec une seule vertice. Cette vertice est combinée avec les deux dernières vertices chargées par l'*input assembler* pour former un triangle. Pour gérer ces *triangles strips*, l'*input assembler* doit mémoriser dans un registre les deux dernières vertices utilisées. En mémoire, le gain est énorme : au lieu de trois vertices pour chaque triangle, on se retrouve avec une vertice pour chaque triangle, sauf le premier de la surface.

5.1.2. Triangle fan

La technique des *triangles fan* fonctionne comme pour le *triangles strip*, sauf que la vertice n'est pas combinée avec les deux vertices précédentes. Supposons que je crée un premier triangle avec les vertices v1, v2, v3. Avec la technique des *triangles strips*, les deux vertices réutilisées auraient été les vertices v2 et v3. Avec les *triangles fans*, les vertices réutilisées sont les vertices v1 et v3. Les *triangles fans* sont utiles pour créer des figures comme des cercles, des halos de lumière, etc.

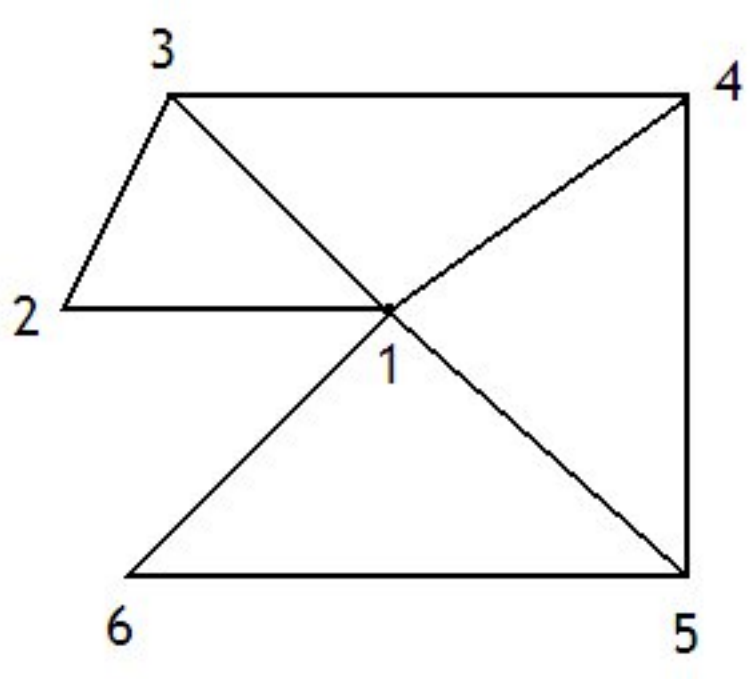

FIGURE 5.3. – Triangle Fan

5.1.3. Index buffers

Enfin, nous arrivons à la dernière technique, qui permet de stocker chaque vertice en un seul exemplaire dans le *vertex buffer*. Le *vertex buffer* est couplé à un *index buffer*, un tableau d'indices qui servent à localiser la vertice dans le *vertex buffer*. Quand on voudra charger une vertice depuis la mémoire vidéo, il suffira de fournir l'indice de la vertice. Pour charger plusieurs fois la même vertice, il suffira de fournir le même indice à l'*input assembler*.

Dit comme cela, on ne voit pas vraiment où se trouve le gain en mémoire. On se retrouve avec deux tableaux : un pour les indices, un pour les vertices. Le truc, c'est qu'un indice prend beaucoup moins de place qu'une vertice. Un indice tient au grand maximum sur 32 bits, là où une vertice peut prendre 128 à 256 bits facilement. Entre copier 7 fois la même vertice, et avoir 7 indices et une vertice, le gain en mémoire est du côté de la solution à base d'index.

5.1.3.1. Vertex cache

Avec un *index buffer*, une vertice peut être chargée 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 ***vertex cache***. Lorsqu'une vertice est lue depuis la mémoire, elle est placée dans ce cache. Lors des utilisations ultérieures, la carte graphique lira la vertice depuis le cache, au lieu d'accéder à la mémoire vidéo, plus lente.

Cette mémoire est transparente via à vis de l'*input assembler* : les accès mémoire sont interceptés par la mémoire cache, qui vérifie si l'adresse demandée correspond à une donnée dans le *vertex cache*. Pour cela, chaque vertice est stockée dans la mémoire cache avec son indice : si jamais l'indice intercepté est présent dans le cache, le cache renvoie la vertice associée.

Sur les cartes graphiques assez anciennes, cette mémoire cache est souvent très petite : elle contient à peine 30 à à 50 vertices. Pour profiter le plus possible de ce *vertex cache*, les concepteurs de jeux vidéo peuvent changer l'ordre des vertices en mémoire : ainsi, l'ordre de lecture des vertices peut profiter de manière optimale du cache.

5.2. Transformation

Une fois la vertice chargée depuis la mémoire, elle est envoyée dans une unité chargée de la traiter. Chaque vertice 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 toutes les vertices de l'objet doivent être mises 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'une vertice, la multiplication par une matrice peut simuler des translations, des rotations, ou des mises à l'échelle.

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.

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.

Les anciennes cartes graphiques contenaient un circuit spécialisé dans ce genre de calculs, qui prenait une vertice et renvoyait la vertice transformée. 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 vertices d'un même objet en même temps.

5.3. Eclairage

Seconde étape de traitement : l'éclairage. À la suite de cette étape d'éclairage, chaque vertice se voit attribuer une couleur, qui correspond à sa luminosité.

5.3.1. Données vertices

À partir de ces informations, la carte graphique va devoir calculer 4 couleurs, attribuées à chaque vertice. Chacune de ces couleurs est une couleur au format RGB. Ces 4 couleurs sont :

— la couleur diffuse ;
— la couleur spéculaire ;
— la couleur ambiante ;
— la couleur émissive.

La **couleur ambiante** correspond à la couleur réfléchie par la lumière ambiante. Par lumière ambiante, on parle de la lumière qui n'a pas vraiment de source de lumière précise, qui est égale en tout point de notre scène 3D (d'où le terme lumière ambiante). Par exemple, on peut simuler le soleil sans utiliser de source de lumière grâce à cette couleur.

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.

Vient ensuite la **couleur spéculaire**, la couleur due à la lumière réfléchie par l'objet. Lorsqu'un rayon lumineux touche une surface, une bonne partie de celui-ci est réfléchie suivant un angle bien particulier (réflexion de Snell-Descartes). Suivant l'angle entre la source de lumière, la position de la caméra, son orientation, et la surface, une partie plus ou moins importante de cette lumière réfléchie sera renvoyée vers la caméra.

Et enfin, on trouve la **couleur émissive** : c'est la couleur de la lumière produite par l'objet. Il arrive que certains objets émettent de la lumière, sans pour autant être des sources de lumière.

La carte graphique a aussi besoin de l'angle avec lequel arrive un rayon lumineux sur la surface de l'objet. Pour gérer l'orientation de la surface, la vertice est fournie 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 la vertice.

Autre paramètre d'une surface : sa **brillance**. Celle-ci indique si la surface brille beaucoup ou pas.

5.3.2. Calcul de l'éclairage

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.

Tout d'abord, la couleur émissive ne change pas : elle ne subit aucun calcul.

Ensuite, la couleur ambiante de notre objet est multipliée par l'intensité de la lumière ambiante.

La couleur diffuse est calculée en multipliant :

- la couleur diffuse de la vertice ;
- l'intensité de la source de lumière ;
- et le cosinus entre la normale, et le rayon de lumière qui touche la vertice.

Ce cosinus sert à gérer l'orientation de la surface comparée à la source de lumière : plus le rayon de lumière incident rase la surface de l'objet, moins celui-ci diffusera de lumière.

Ensuite, la lumière réfléchie, spéculaire est calculée. Pour cela, il suffit de multiplier :

- la couleur spéculaire ;
- l'intensité de la source lumineuse ;
- et un paramètre qui dépend de la direction du regard de la caméra, et de la direction du rayon réfléchi par la surface.

Ce rayon réfléchi correspond simplement à la direction qu'aurait un rayon de lumière une fois réfléchi par la surface de notre objet. Pensez à la loi de Snell-Descartes. Le paramètre en question se calcule avec cette formule : cos^brillance (couleur_spéculaire).

Enfin, les 4 couleurs calculées plus haut sont additionnées ensemble. Chaque composante rouge, bleu, ou verte de la couleur sera traitée indépendamment des autres.

6. 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.

6.1. 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***.

6.1.1. 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.

6.1.2. 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.

FIGURE 6.1. – Plus petit 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.

FIGURE 6.2. – tiled traversal

6.2. 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. Par exemple, si j'ai un sommet vert, un sommet rouge, et un sommet bleu, le triangle résultant doit être colorié comme ceci :

FIGURE 6.3. – Interpolation dans un triangle

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.

6.2.1. Fragments

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.

6.2.2. Coordonnées barycentriques

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 :

FIGURE 6.4. – Coordonnées barycentriques

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).

6.2.3. Perspective

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).

Le résultat obtenu est alors celui du milieu :

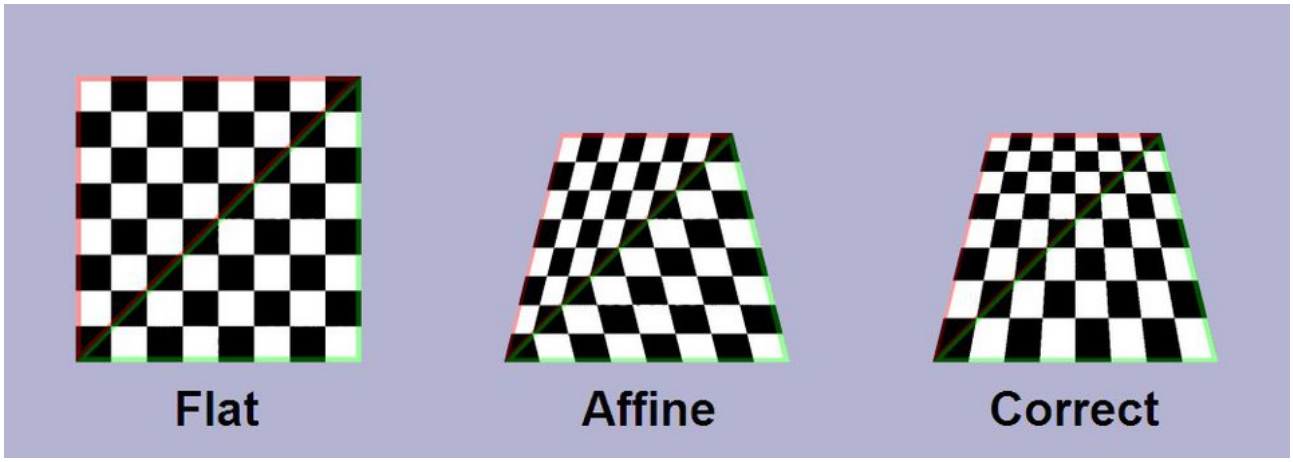

FIGURE 6.5. – Perspective

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.

7. 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.

FIGURE 7.1. – texture mapping

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*.

7.1. 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 la 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.

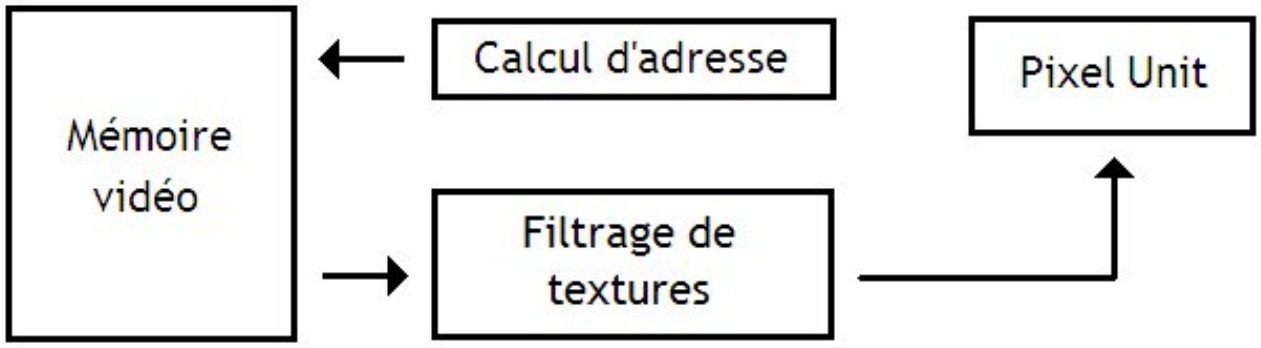

FIGURE 7.2. – Circuits de gestion des textures

7.1.1. 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.

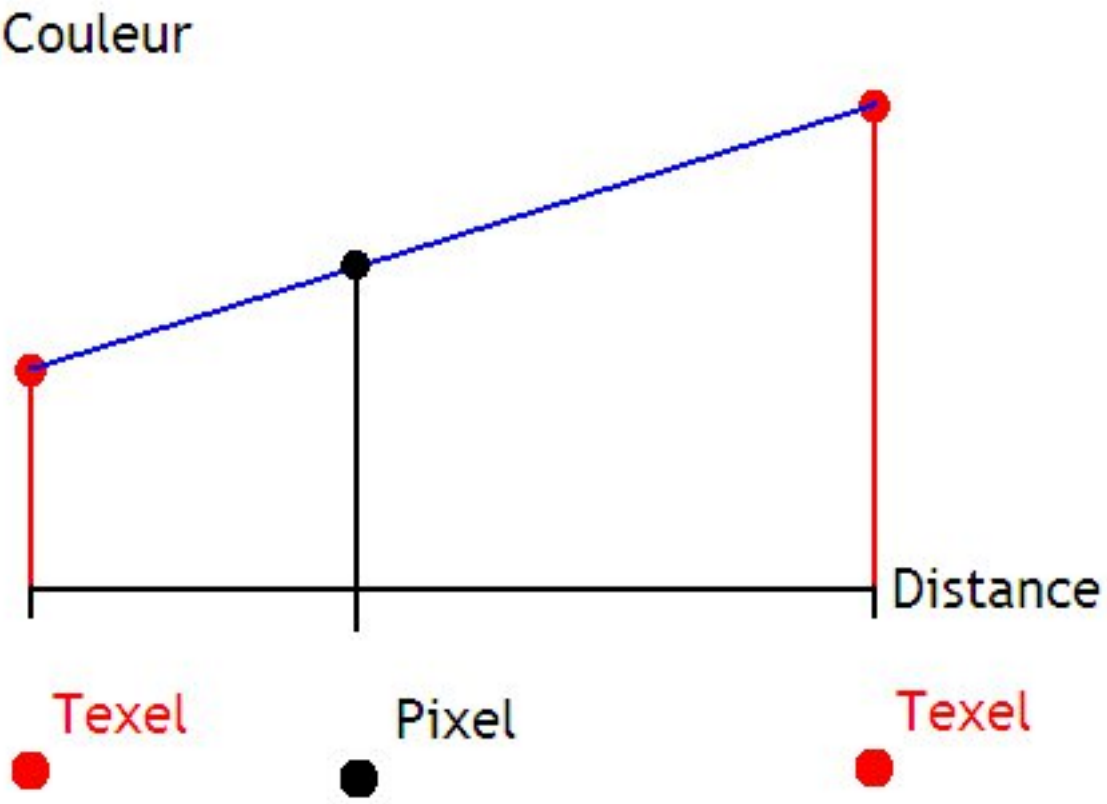

FIGURE 7.3. – Interpolation linéaire

Il suffit de :

— calculer la pente de la courbe ;
— multiplier cette pente par la distance entre le texel choisit et le pixel ;
— ajouter la couleur de base du texel choisit.

Seul problème, cela marche pour deux pixels, pas 4. Avec 4 pixels, nous allons devoir calculer la couleur de points intermédiaires :

— 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.

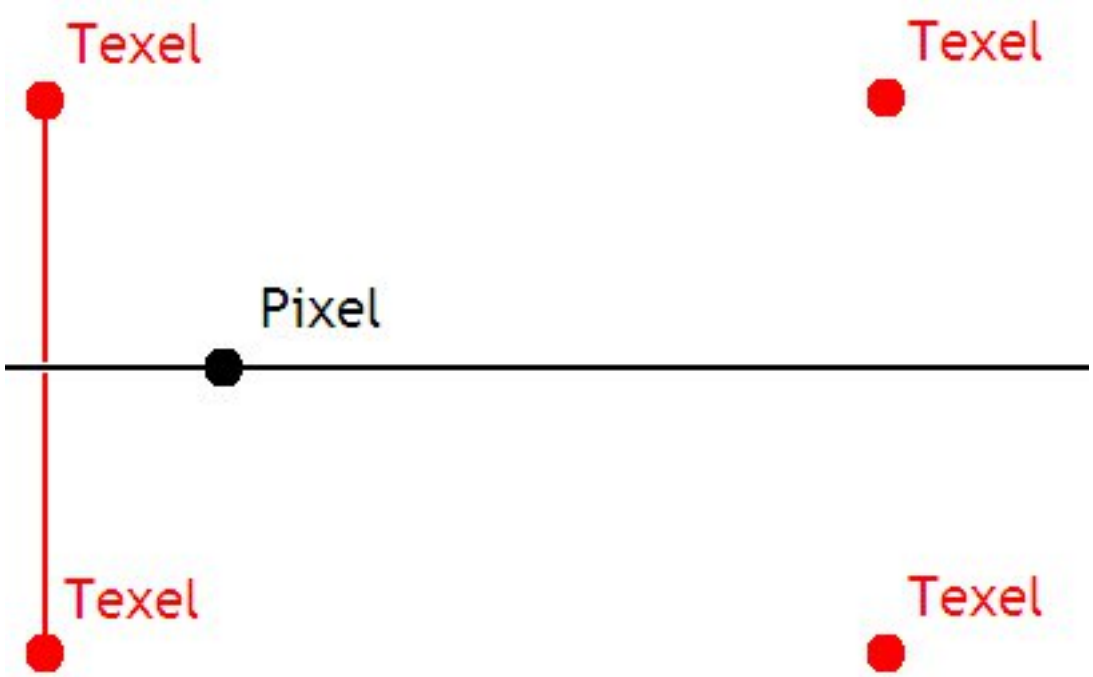

FIGURE 7.4. – Premier filtrage

— 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.

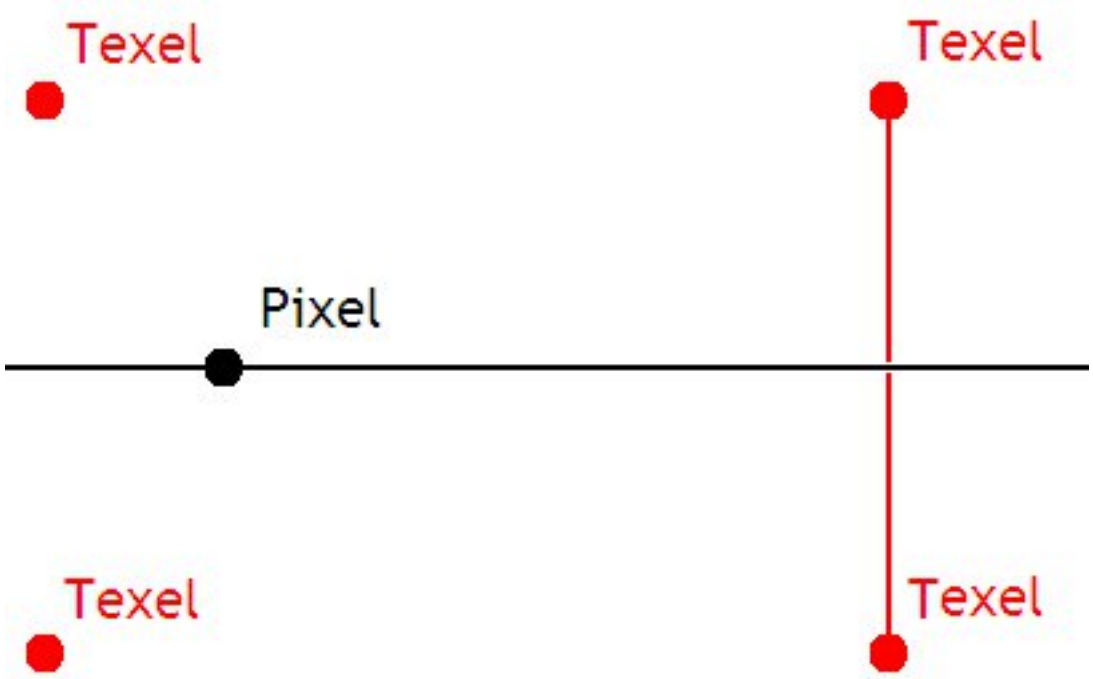

FIGURE 7.5. – Second filtrage

La couleur de ces deux points se calcule par interpolation linéaire, et il suffit d'utiliser une troisième interpolation linéaire pour obtenir le résultat.

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 :

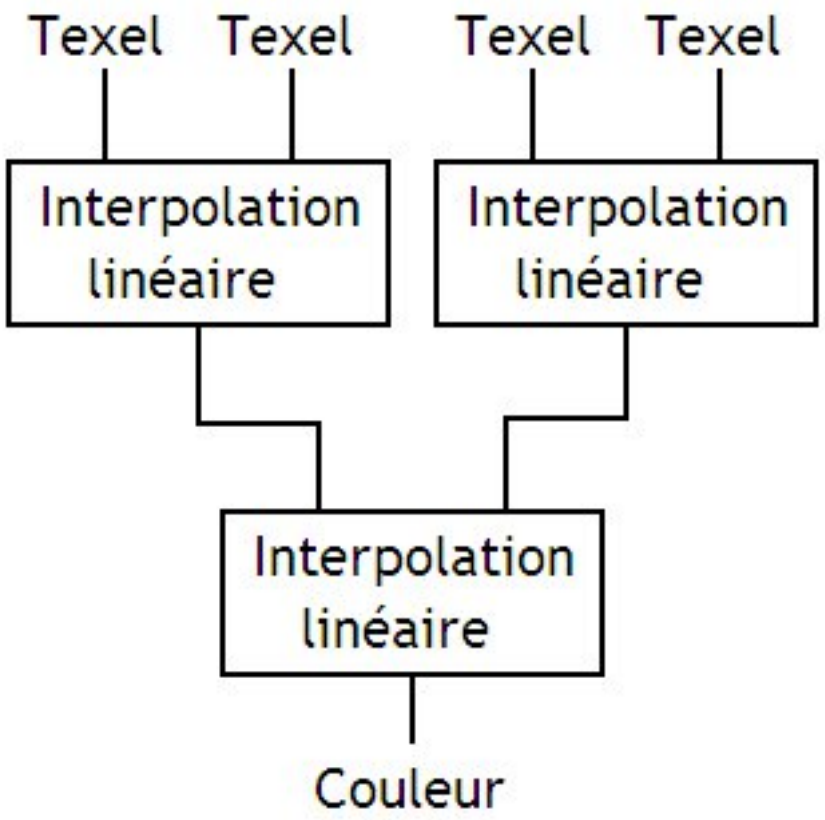

FIGURE 7.6. – Circuit de filtrage bilinéaire

7.1.2. Mip-mapping

Si une texture est plaquée sur un objet lointain, une bonne partie des détails de la texture est invisible pour l'utilisateur : 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, procéder comme 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.

Pour faire en sorte que la bonne mip-map soit choisie, les circuits chargés de calculer l'adresse de la texture doivent recevoir des informations supplémentaires pour choisir la mip-map à appliquer. Der plus, ils doivent être adaptés pour calculer l'adresse du texel correctement : celui doit être chargé depuis la bonne mip-map.

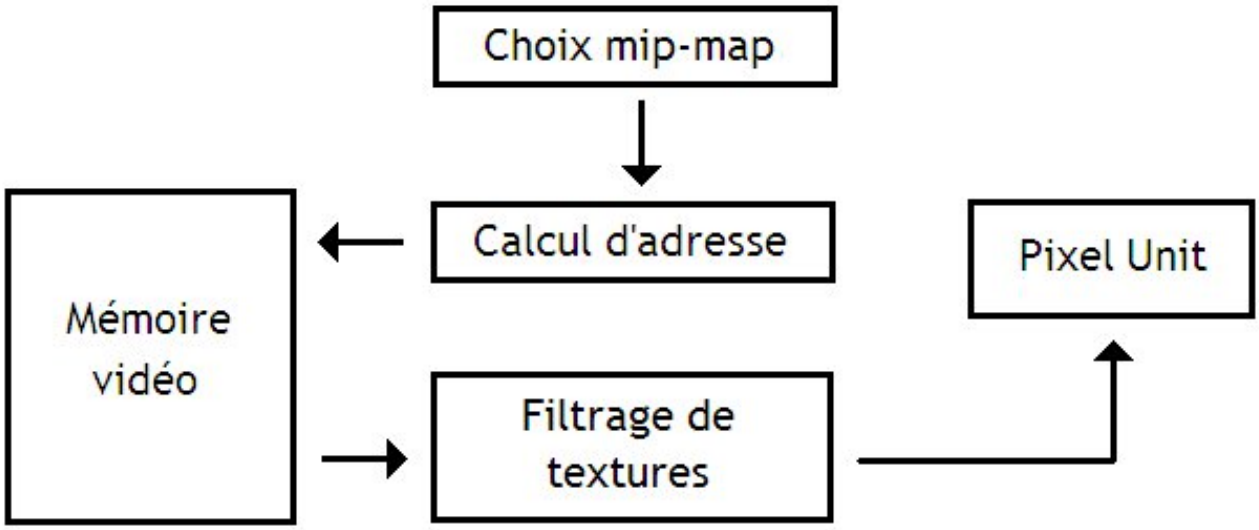

FIGURE 7.7. – Mip-mapping

Pour faciliter ces calculs, les mip-maps d'une texture sont stockées les unes après les autres en mémoire. 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.

Évidemment, cette technique consomme de la mémoire RAM : chaque texture est dupliquée en plusieurs exemplaires. On peut remarquer une chose : si je prend une texture à un niveau de détail donné, la texture de niveau de détail immédiatement inférieur sera 4 fois plus petite : 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/4*4*) + *(X/4*4*4*) + … *Cela donne 4/3* X. La technique du mip-mapping prendra donc au maximum 33% de mémoire en plus (sans compression).

7.1.3. Filtrage trilinéaire

Avec le mip-mapping, les textures sont un peu plus belles, mais cette technique a un défaut : 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. Son principe est simple : il consiste à faire « une moyenne » 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.

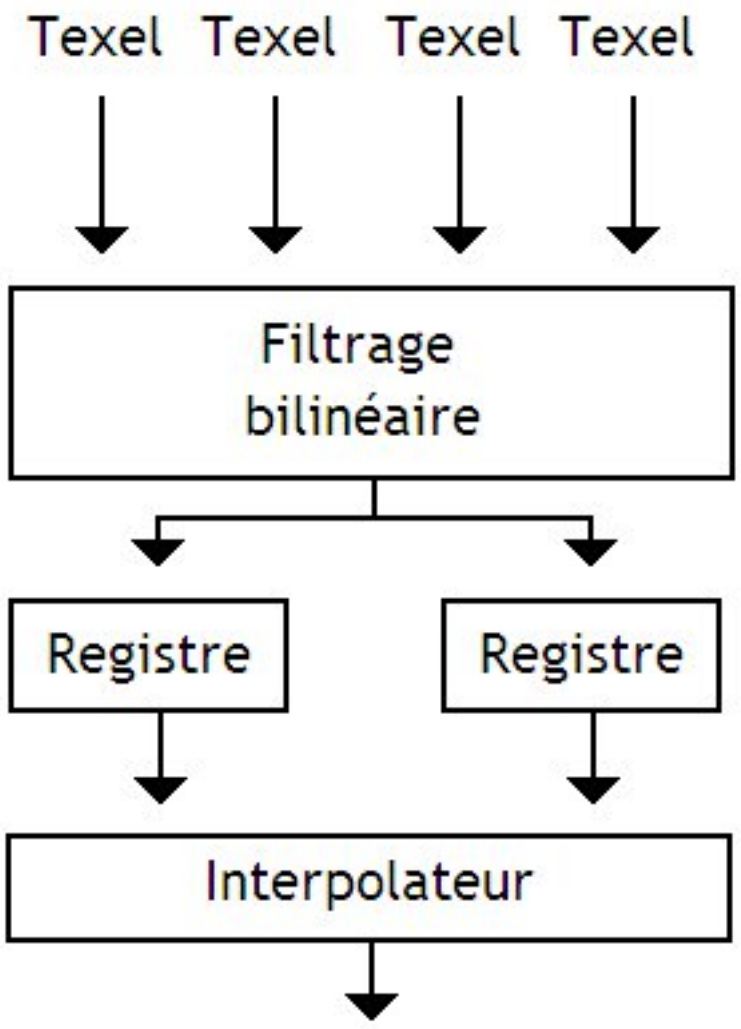

FIGURE 7.8. – Circuit de filtrage de textures

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.

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.

7.1.4. Filtrage anisotropique

Le filtrage trilinéaire permet de gommer les imperfections dues au mip-mapping. Mais d'autres artefacts peuvent survenir lors de l'application d'une texture : la perspective a tendance à déformer les textures, et peut entraîner l'apparition de flou dans certains cas. Pour gommer ce flou de perspective, les chercheurs ont inventé le filtrage anisotropique.

Dans tous les cas, le filtrage anisotropique va charger un grand nombre de texels, et effectuer des suites de filtrages bilinéaires sur ces texels chargés. Les texels chargés seront 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 rien au niveau des circuits de filtrage. Cela peut paraître bizarre, mais en réalité, le filtrage anisotropique ne fait que mieux choisir les texels sur lesquels utiliser le filtrage, et leur attribuer des coefficients. Tout se passe donc lors du choix des texels, à savoir : l'étape de calcul d'adresse.

Quand je parle de filtrage anisotropique, je mens un tout petit peu. 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.

7.2. Compression

Certaines textures un peu spéciales peuvent aller jusqu'au mébioctet, et quand on sait qu'une scène 3D normale peut dépasser la cinquantaine de textures, on est heureux de ne pas être une mémoire vidéo. Et c'est sans compter le filtrage, qui impose de lire plusieurs texels pour colorier un seul pixel : 4 d'un coup pour un filtrage bilinéaire, 8 pour le filtrage trilinéaire, et encore plus pour le filtrage anisotropique.

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. Nous allons en voir quelque uns. Tous ces formats sont des formats de compression dits avec pertes. Cela signifie qu'il y a une légère pertes de qualité lors de la compression. Toutefois, cette perte peut être compensée en utilisant des textures à résolution plus grande.

Comme je l'ai dit auparavant dans ce tutoriel, on peut approximativement considérer qu'une texture est une image. Il existe cependant des textures qui ne sont pas vraiment des images, et qui sont de simples tableaux de données manipulés par les pixels shaders. Et comprimer ces textures n'est pas la même chose que comprimer des textures images. Comme format, on pourrait le format 3dc d'ATI/AMD, qui sert à compresser els normals maps.

Pour les textures qui mémorisent des images, on pourrait penser utiliser des algorithmes comme le JPEG pour compresser les textures qui représentent des images. Seul problème : ces algorithmes codent des pixels sur un nombre de bits variable : impossible de calculer à l'avance la position d'un pixel en mémoire vidéo. On devrait parcourir la totalité de la texture pour lire un seul pixel ! Utiliser de tels algorithmes est donc impossible : il faut ruser...

7.2.1. 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 des couleurs, qui contient les couleurs utilisées dans la texture : ce tableau s'appelle la **palette**. La texture ne contient aucune couleur par elle-même : à la place de chaque couleur, la texture stockera l'indice de la couleur dans la table.

Cependant, la table des couleurs a une taille fixe (de même que les numéros utilisés pour encoder les couleurs). En conséquence, cette technique ne marche pas pour les textures qui utilisent beaucoup de couleurs différentes : on est obligé de sélectionner un nombre limité de couleurs. 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.

7.2.2. Compression par blocs

Mais il y a moyen de ruser. Dans la sous-partie sur le filtrage de texture, on a vu que les cartes graphiques lisent les textures par blocs de 16*16, 8*8 ou 4*4 texels. De nos jours, la compression ne cherche pas à compresser des pixels individuels, mais ces blocs de texels. Le nombre de bit utilisé pour chaque texel peut varier : certains pixels se voient attribuer plus de pixels que prévus, alors que d'autres sont moins prioritaires.

7.2.2.1. Vector quantization

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.

7.2.2.2. Block Truncation coding

La première technique de compression élaborée est celle du Block Truncation Coding. Cette méthode a toutefois un défaut : elle ne marche que pour les images en niveaux de gris. Mais on va voir que cet algorithme 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 du Block Truncation Coding.

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, il faut mémoriser quelle est la couleur du pixel : s'agit-il de la couleur 1 ou de la couleur 2 ? Pour cela, on utilise un bit par pixel, dont la valeur indique quelle couleur choisir : 0 pour couleur 1, et 1 pour couleur 2.

Chaque bloc est donc mémorisé en mémoire par :

— deux entiers, qui codent chacun une couleur ;
— une suite de bits.

La méthode de décompression est la suivante :

— sélectionner le bit attribué au pixel à lire ;
— selon la valeur de ce bit, choisir la couleur 1 ou couleur.

Le circuit de décompression est alors vraiment très simple : il suffit d'utiliser deux multiplexeurs.

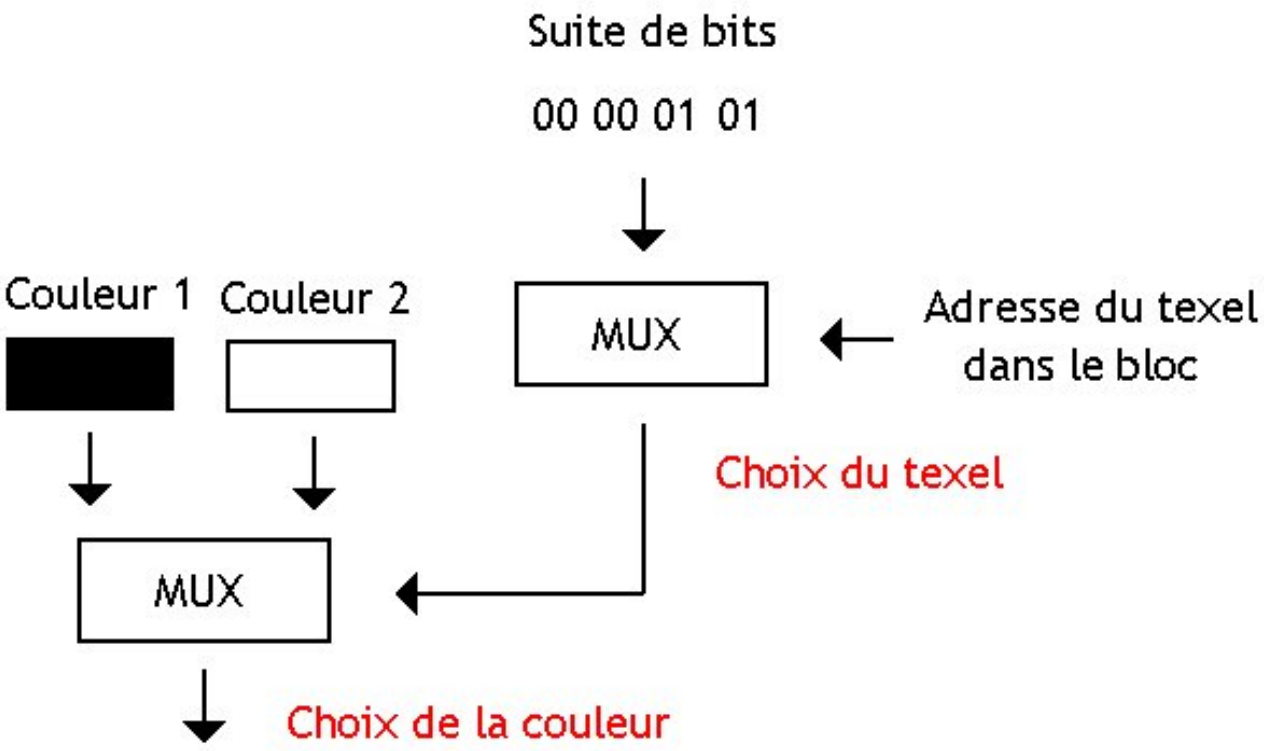

FIGURE 7.9. – Circuit de décompression

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.

7.2.2.3. 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 est très simple : au lieu d'utiliser deux niveaux de gris par bloc, on utilise deux couleurs RGBA codées sur 32 bits. Le circuit de décompression est identique à celui utilisé pour le BTC.

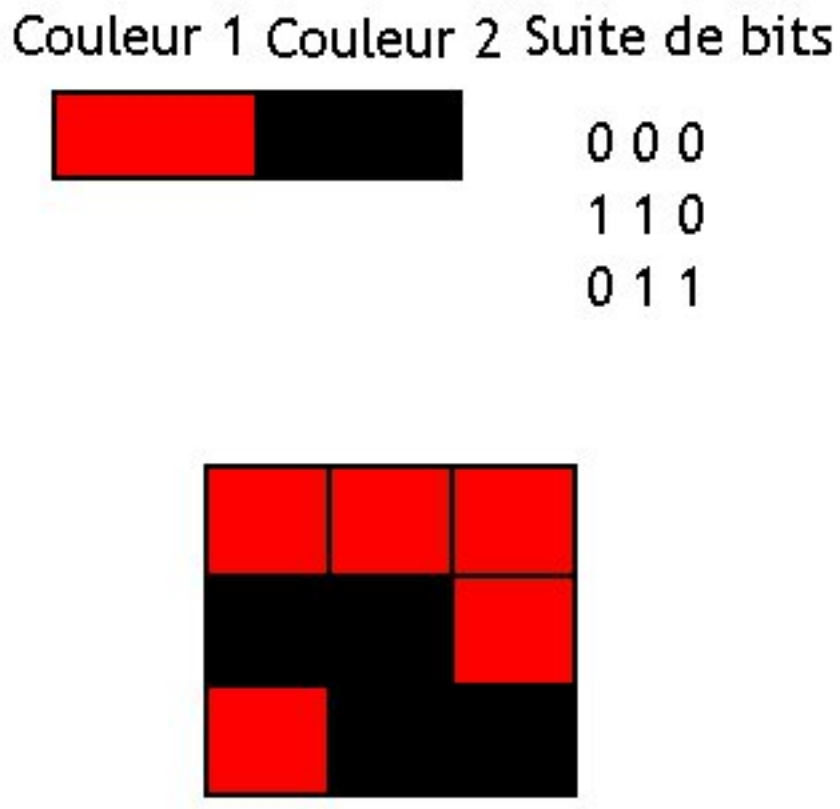

FIGURE 7.10. – CCC

7.2.2.4. 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. Pour indiquer comment faire ce mélange, on trouve deux bits de contrôle par texel.

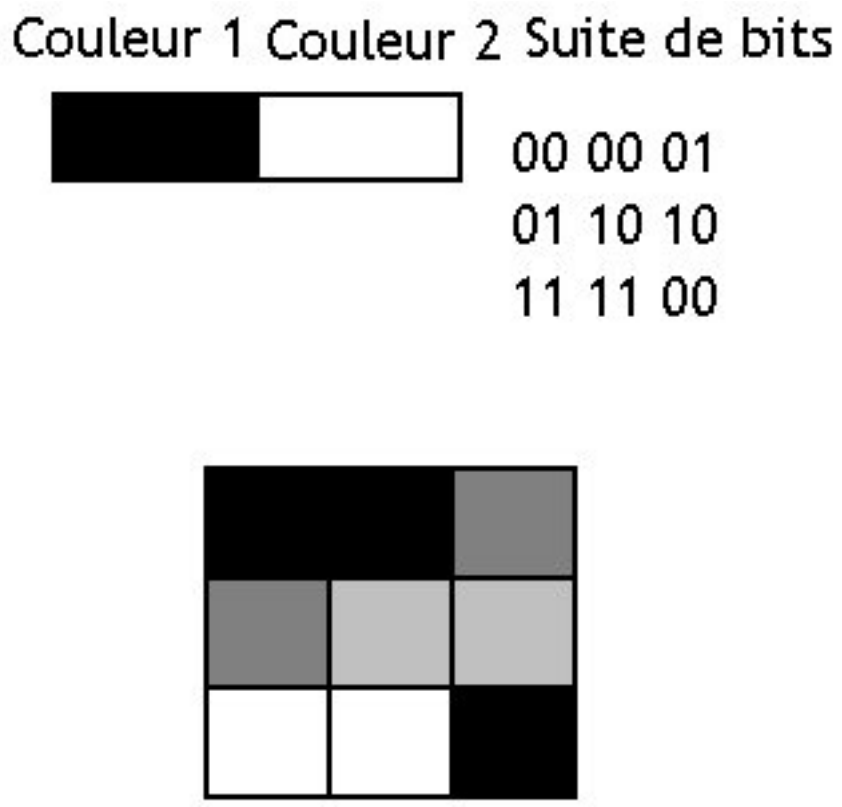

FIGURE 7.11. – DXTC

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 :

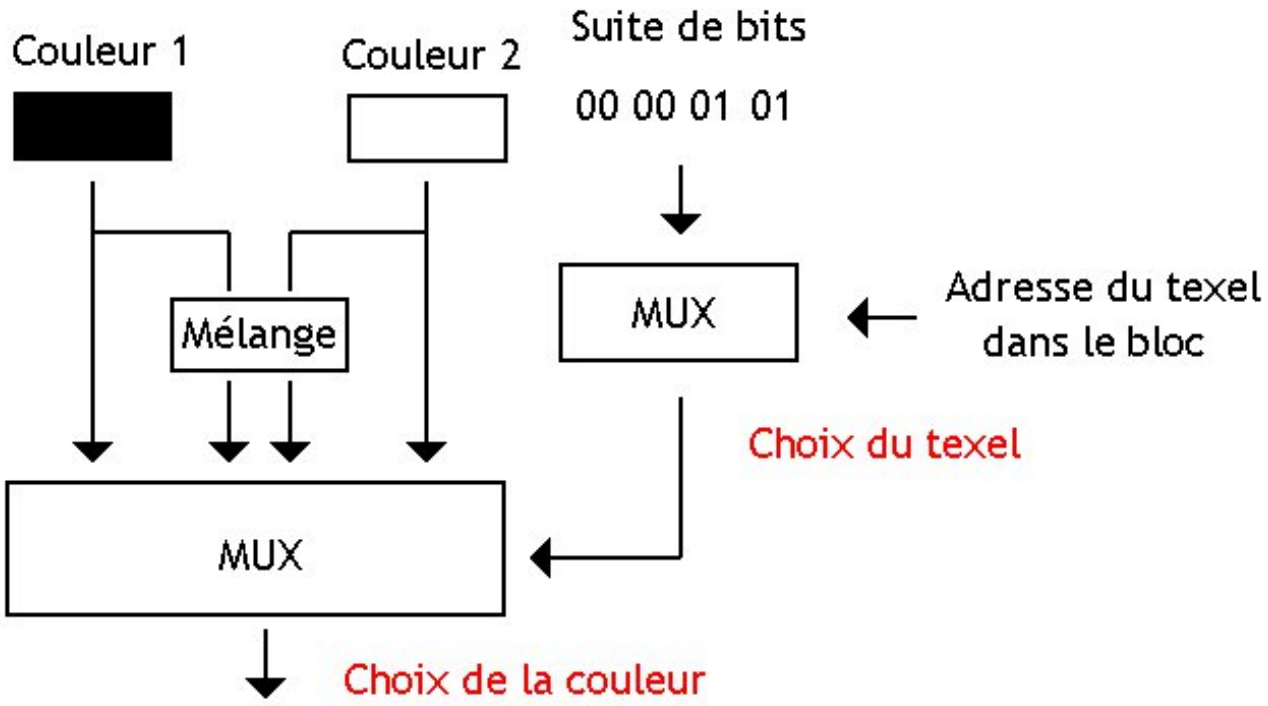

FIGURE 7.12. – Circuit de décompression du DXTC

7.2.2.5. 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. 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. Le DXT3 a rapidement été replacé par le DXT4 et par le DXT5.

Dans ces deux formats, 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 de ces 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. Pour être franc, tous les jeux vidéos actuels encodent une bonne partie de leurs textures en DXT5.

7.2.2.6. 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.

7.2.2.7. Méthodes plus récentes

Il existe des format de texture plus récents, comme l'*Ericsson Texture Compression* ou l'*Adaptive Scalable Texture Compression*.

7.3. Texture cache

Les accès aux textures se font donc en mémoire vidéo. Seul problème : notre mémoire vidéo est lente. Pour faciliter l'accès aux textures, les cartes 3D utilisent souvent une ou plusieurs mémoires caches ultra-rapides, spécialisées dans le traitement des textures. Lorsqu'un texel est lu pour la première fois, celui-ci est placée dans ce **cache de textures**. Lors des utilisations ultérieures, la carte graphique aura juste à lire le texel depuis ce cache au lieu de devoir accéder à la mémoire vidéo, ce qui est nettement plus rapide.

7.3.1. Stockage des textures en mémoire

Ce cache est composé de blocs de mémoire de taille fixe, les lignes de cache, qui servent d'unité de base pour les échanges entre mémoire et cache. 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 en réalité, 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.

7.3.2. Multi-level texture cache

Ceci dit, les cartes graphiques actuelles n'ont pas qu'un seul cache de textures. Toute les cartes graphiques actuelles disposent de deux caches de textures : un petit, et un gros. Les deux caches sont fortement différents. L'un est un gros cache, qui fait dans les 4 kibioctets, et l'autre est un petit cache, faisant souvent moins d'1 kibioctet.

7.3.3. Cohérence des caches

Dans la majorité des cas, le cache de textures est accessible uniquement en lecture, pas en écriture. Simple question de coût. Seulement, 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. La présence d'une mémoire cache en lecture seule peut alors poser des problèmes : la modification d'une texture via render-to-texture n'est pas propagée dans le cache, qui conserve l'ancienne donnée.

Une solution simple consiste à garder un cache en lecture seule, et à invalider les données mises à jour lors d'une écriture. Si la carte graphique écrit dans la mémoire, le cache vérifie si la donnée dans le cache est mise à jour, l'invalide si c'est le cas. Pour cela, notre cache contient un bit pour chaque ligne, qui indique si la donnée est invalide, qui est mis à jour lors des écritures.

Cette technique peut être adaptée dans le cas où plusieurs mémoires de textures séparées existent sur une même carte graphique : les écritures doivent invalider toutes les copies dans tous les caches de texture. Cela nécessite d'ajouter des circuits qui propagent l'invalidation dans tous les autres caches.

Autre solution : rendre le cache de texture accessible en écriture. Si un seul cache de texture est présente dans la carte graphique, il n'y a pas besoin de modifications supplémentaires. 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.

7.4. Prefetching

Avec l'organisation telle qu'on l'a vue, l'accès aux textures est lent : plusieurs centaines de cycles d'horloges si on lit depuis la mémoire vidéo, et environ 20 cycles si on lit dans le cache. Pour améliorer la rapidité des accès, il est possible d'utiliser un **prefetch de texture**. Avec ce prefetch, la carte graphique peut préparer certaines lectures de texture à l'avance.

Un accès à une texture est composé d'un grand nombre de sous-étapes. Par exemple, cette série d'étapes pourrait être :

— déterminer le niveau de mip-map ;
— effectuer des calculs pour le filtrage anisotropique ;
— calculer l'adresse effective en mémoire vidéo des texels à lire ;
— accéder à la mémoire ;
— filtrer les texels et les décompresser.

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 mise en attente dans une petite mémoire tampon, en attendant que la mémoire vidéo soit libre. Cette mémoire tampon est une mémoire FIFO, une mémoire dans laquelle les données sont stockées dans leur ordre d'arrivée. Lors d'une lecture, la donnée arrivée en dernier est renvoyée.

Ce prefetch peut s'implémenter de deux façons, suivant que la carte graphique utilise ou non un cache de texture.

FIGURE 7.13. – Prefetching

8. Les processeurs de Shaders

Au fur et à mesure que les procédés de fabrication devenaient de plus en plus étoffés, les cartes graphiques pouvaient incorporer un plus grand nombre de circuits. Les unités de traitement de la géométrie étaient autrefois câblées : elles ne pouvaient effectuer qu'un éclairage de type Phong, rien de plus. Alors certes, il était possible de configurer certains paramètres pour obtenir un éclairage proche de celui voulu, mais le type d'éclairage restait le même.

Par la suite, les unités de traitement de la géométrie sont devenues des unités programmables. Par programmable, on veut dire qu'il est possible de spécifier leur comportement via un programme informatique. Cela permet une grande flexibilité : changer le comportement ne nécessite pas de re-câbler tout le circuit (ce qui est souvent impossible) : il suffit simplement ce changer la suite d'instructions à exécuter. Nos unités de traitement de la géométrie deviennent donc des processeurs indépendants, capable d'effectuer un certain nombre d'instructions sur les données de nos Vertices.

Ces processeurs sont donc capable d'exécuter des programmes sur des vertices. Ces programmes sont appelés des **Vertex Shaders**. Ils sont souvent écrits dans un langage de haut-niveau, le HLSL ou le GLSL, et sont traduits (compilés) par les pilotes de la carte graphique, pour les rendre compatibles avec le processeur de *vertex shaders*. Au début, ces langages, ainsi que le matériel, supportaient uniquement des programmes simples. Au fil du temps, les spécifications de ces langages sont devenues de plus en plus riches à chaque version, et le matériel en a fait autant.

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*. Ils fonctionnent sur le même principe que les processeurs de *vertex shaders*.

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. Depuis DirectX 10, ce n'est plus le cas. Depuis, le jeu d'instructions a été unifié entre les *vertex shaders* et les *pixels shaders*.

Les premiers processeurs de *shaders* disposaient de peu d'instructions. On trouvait uniquement des instructions de calcul arithmétiques, dont certaines étaient assez complexes (logarithmes, racines carrées, etc). Depuis, d'autres versions de *vertex shaders* ont vu le jour. Pour résumer, les améliorations ont portées sur :

— le nombre de registres ;
— la taille de la mémoire qui stocke les *shaders* ;
— le support des branchements ;
— l'ajout d'instructions d'appel de fonction ;
— le support de fonctions imbriquées ;
— l'ajout d'instructions de lecture/écriture en mémoire centrale ;

— l'ajout d'instructions capables de traiter des nombres entiers ;
— l'ajout d'instructions bit à bit.

Un processeur de *shaders* contient deux types de registres :

— des registres généraux, qui peuvent mémoriser tout type de données ;
— des registres qui servent à stocker des constantes.

Ces derniers permettent de stocker les matrices servant aux différentes étapes de transformation, à stocker les positions des sources de lumière pour l'éclairage, etc. Ces constantes sont placées dans ces registres lors du chargement du *vertex shader* dans la mémoire vidéo : les constantes sont chargées un peu après. Toutefois, le *vertex shader* peut écrire dans ces registres, au prix d'une perte de performance particulièrement violente.

Le choix de la constante à utiliser dans une instruction s'effectue en utilisant un registre : le **registre d'adresse de constante**. Celui-ci va permettre de préciser quel est le registre de constante à sélectionner dans une instruction. Une instruction peut ainsi lire une constante depuis les registres constants, et l'utiliser dans ses calculs.

8.1. Jeux d'instruction

Sur tous les processeurs de traitement de vertices, il est possible de traiter plusieurs morceaux de vertices à la fois. Pour cela, les processeurs de traitement de vertices utilisent :

— soit des instructions spécialisées, qui peuvent manipuler un grand nombre de données simultanément ;
— soit incluent un grand nombre d'unités de calcul qui fonctionnent en parallèle.

Il existe deux techniques pour cela :

— les processeurs SIMD ;
— les processeurs VLIW ;
— les processeurs de flux.

8.1.1. Processeurs SIMD

Les instructions des processeurs SIMD sont des **instructions vectorielles** : elles travaillent sur des **vecteurs**. Ces vecteurs contiennent plusieurs nombres entiers ou nombres flottants placés les uns à côté des autres, et ont une taille fixe.

Une instruction de calcul vectoriel va traiter chacune des données du vecteur indépendamment des autres. Par exemple, une instruction d'addition vectorielle va additionner ensemble les données qui sont à la même place dans deux vecteurs, et placer le résultat dans un autre vecteur, à la même place. Quand on exécute une instruction sur un vecteur, les données présentes dans ce vecteur sont traitées simultanément.

8.1.1.1. Geforce 3

La première carte graphique commerciale destinée aux *gamers* à disposer d'une unité de vertex programmable est la Geforce 3. Celui-ci respectait le format de *vertex shader* 1.1. L'ensemble des informations à savoir sur cette unité est disponible dans l'article *"A user programmable vertex engine"*, disponible sur le net.

Le processeur de cette carte était capable de gérer un seul type de données : les nombres flottants de norme IEEE754. Toutes les informations concernant la coordonnée d'une vertice, voire ses différentes couleurs, doivent être encodées en utilisant ces flottants. De nos jours, les processeurs de vertices sont capables de gérer des nombres entiers, et les instructions qui vont avec.

Ce processeur est capable d'exécuter 17 instructions différentes.
Voici la liste de ces instructions :

OpCode	Nom	Description
MOV	Move	vector -> vector
MUL	Multiply	vector -> vector
ADD	Add	vector -> vector
MAD	Multiply and add	vector -> vector
DST	Distance	vector -> vector
MIN	Minimum	vector -> vector
MAX	Maximum	vector -> vector
SLT	Set on less than	vector -> vector
SGE	Set on greater or equal	vector -> vector
RCP	Reciprocal	scalar-> replicated scalar
RSQ	Reciprocal square root	scalar-> replicated scalar
DP3	3 term dot product	vector-> replicated scalar
DP4	4 term dot product	vector-> replicated scalar
LOG	Log base 2	miscellaneous
EXP	Exp base 2	miscellaneous
LIT	Phong lighting	miscellaneous
ARL	Address register load	miscellaneous

Comme on le voit, ces instructions sont presque toutes des instructions arithmétiques. On y trouve des multiplications, des additions, des exponentielles, des logarithmes, des racines carrées, etc. À coté, on trouve des comparaisons (SDE, SLT), une instruction MOV qui déplace le contenu d'un registre dans un autre, et une instruction de calcul d'adresse. Fait intéressant, toutes ces instructions peuvent s'exécuter en un seul cycle d'horloge.

On remarque que parmi toutes ces instructions arithmétiques, la division est absente. Il faut dire que la contrainte qui veut que toutes ces instructions s'exécutent en un cycle d'horloge pose quelques problèmes avec la division, qui est une opération plutôt lourde en hardware. À la place, on trouve l'instruction RCP, capable de calculer 1/x, avec x un flottant. Cela permet ainsi de simuler une division : pour obtenir Y/X, il suffit de calculer 1/X avec RCP, et de multiplier le résultat par Y.

Autre manque : les instructions de branchement. C'est un fait, ce processeur ne peut pas effectuer de branchements. À la place, il doit simuler ceux-ci en utilisant des instructions arithmétiques. C'est très complexe, et cela limite un peu les possibilités de programmation. À l'époque, ces branchements n'étaient pas utiles, sans compter que les environnements de programmation ne permettaient pas d'utiliser de branchements lors de l'écriture de *shaders*. De nos jours, les cartes graphiques récentes peuvent effectuer des branchements, ou du moins, des instructions similaires.

On remarque qu'il n'y a aucune instruction d'accès à la mémoire. Notre processeur ne peut pas aller chercher d'informations dans la mémoire vidéo. Le processeur de la Geforce 3 doit se contenter de ses registres. Depuis, la situation a changé : les cartes graphiques récentes peuvent aller lire certaines données depuis la mémoire vidéo.

8.1.1.2. Instructions à prédicats

Les instructions vectorielles sont très utiles quand on doit effectuer un traitement identique sur un ensemble de données identiques. Mais dès que le traitement à effectuer sur nos données varie suivant le résultat d'un test ou d'un branchement, les choses se gâtent. Mine de rien, avec une instruction vectorielle, on est obligé de calculer et de modifier tous les éléments d'un paquet : il est impossible de zapper certains éléments d'un paquet dans certaines conditions. Par exemple, imaginons que je veuille seulement additionner les éléments d'un paquet ensemble s'ils sont positifs : je ne peux pas le faire avec une instruction vectorielle "normale". Du moins, pas sans aide.

Pour résoudre ce problème, certains processeurs utilisent des **instructions à prédicats**. Pour faire simple, ces instructions sont des instructions "annulables". Elle ne modifient un élément d'un vecteur que si celui-ci remplit une condition. Pour cela, notre processeur de traitement de vertices contient un **Vector Mask Register**. Celui-ci permet de stocker des informations qui permettront de sélectionner certaines données et pas d'autres pour faire notre calcul. Il est mis à jour par des instructions de comparaison.

Ce *Vector Mask Register* va stocker des bits pour chaque flottant présent dans le vecteur à traiter. Si ce bit est à 1, notre instruction doit s'exécuter sur la donnée associée à ce bit. Sinon, notre instruction ne doit pas la modifier. On peut ainsi traiter seulement une partie des registres stockant des vecteurs SIMD.

8.1.2. Processeur VLIW

Autre solution : faire de nos *Streams Processors* des processeurs VLIW. Sur ces processeurs VLIW, nos instructions sont regroupées dans ce qu'on appelle des *Bundles*, des sortes de super-instructions. Ces *bundles* sont découpés en *slots*, en morceaux de taille bien précise, dans lesquels il va venir placer les instructions élémentaires à faire exécuter.

Instruction	**VLIW**	**à 3 slots**
Slot 1	Slot 2	Slot 3
Addition	Multiplication	Décalage à gauche

Chaque *slot* sera attribué à une unité de calcul bien précise. Par exemple, le premier *slot* sera attribué à la première ALU, la second à une autre ALU, le troisième à la FPU, etc. Ainsi, l'unité de calcul exécutant l'instruction sera précisée via la place de l'instruction élémentaire, le *slot* dans lequel elle se trouve.

Qui plus est, vu que chaque *slot* sera attribué à une unité de calcul différente, le compilateur peut se débrouiller pour que chaque instruction dans un *bundle* soit indépendante de toutes les autres instructions dans ce *bundle*. Lorsqu'on exécute un *bundle*, il sera décomposé par le séquenceur en petites instructions élémentaires qui seront chacune attribuée à l'unité de calcul précisée par le *slot* qu'elles occupent. Pour simplifier la tâche du décodage, on fait en sorte que chaque *slot* ait une taille fixe.

Dans la majorité des cas, ces unités VLIW sont capables de traiter deux instructions arithmétiques en parallèles : une qui sera appliquée aux couleurs R, G, et B, et une autre qui sera appliquée à la couleur de transparence. Cette possibilité s'appelle la **co-issue**.

8.1.3. Streams processors

De nos jours, les processeurs de *shaders* sont ce qu'on appelle des *Streams Processors*, des processeurs SIMD qui utilisent plusieurs "couches" de registres.

8.1.3.1. Hiérarchie de registres

On trouve d'abord les *Local Register Files*, directement connectés aux unités de calcul : c'estlà que les unités de calcul vont aller chercher les données à manipuler. Plus bas, ces *Local Register Files* sont reliés à un *Register File* plus gros, le *Global Register File*, lui-même relié à la mémoire.

FIGURE 8.1. – Hiérarchie de registres des *Streams processors*

Le *Global Register File* va servir d'intermédiaire entre la mémoire RAM et le *Local Register File*, un peu comme une mémoire cache. La différence entre ce *Global Register File* et un cache vient du fait que les caches sont souvent gérés par le matériel, tandis que ces *Register Files* sont gérés par le logiciel (le programmeur) : le processeur dispose d'instructions pour transférer des données entre les *Register Files* ou entre ceux-ci et la mémoire.

Le *Global Register File* va servir à stocker un ou plusieurs *Threads* destinés à être traités par notre *Stream Processor*. Il peut aussi servir à transférer des données entre les *Local Register Files*, où à stocker des données globales, utilisées par des *Clusters* d'ALU différents. Quand à nos *Local Register Files*, ils vont servir à stocker des morceaux de *Threads* en cours de traitement : tous les résultats temporaires vont aller dans ce *Local Register File*, afin d'être lus ou écrits le plus rapidement possible.

Pourquoi trouve-t-on plusieurs couches de registres ? Le fait est que les *Streams Processors* disposent de plusieurs centaines d'unités de calcul. Or, pour garder un *Register File* rapide et pratique, on est obligé de limiter le nombre d'unités de calcul connectées dessus, ainsi que le nombre de registres. La solution est donc de casser notre gros *Register File* en plusieurs plus petits, reliés à un *Register File* plus gros, capable de communiquer avec la mémoire.

8.1.3.2. Architecture d'un GPU

Sur ces processeurs, des programmes, nommés *Kernels*, sont appliqués entièrement à un tableau de donnée que l'on appelle un *Stream*. Dans nos cartes graphiques actuelles, ce *Stream* est découpé en morceaux qui seront chacun traités sur un *Stream Processor*. Chacun de ces morceaux est appelé un *Thread*. Vous remarquerez que le terme *Thread* est ici utilisé dans un sens différent de celui utilisé précédemment. Faites attention !

Le découpage du *Stream* en *Threads* se fait à l'exécution. En clair : on envoie à notre carte 3D des informations sur le tableau à manipuler et celle-ci se débrouille toute seule pour le découper en morceau et les répartir sur les processeurs disponibles.

Un GPU actuel est souvent composé de plusieurs de ces *Streams Processors*, placés ensembles sur une même puce, avec quelques autres circuits annexes, utilisés dans les taches de rendu 3D. Ces *Streams Processors* sont alors pilotés par un gros micro-contrôleur qui se charge de découper le *Stream* à traiter en *Threads*, avant de les répartir sur les différents *Streams Processors* de la puce.

8.2. Microarchitecture

Tous les pixels doivent accéder à une texture pour être coloriés, certains traitements devant être effectués ensuite par un *pixel shader*. Mais un accès à une texture, c'est long : une bonne centaine de cycles d'horloges lors d'un accès à une texture est un minimum si celle-ci est lue depuis la mémoire vidéo. Pour éviter que le processeur de *shaders* attende la mémoire, celui-ci dispose de techniques élaborées.

Autrefois, la séparation entre unités de texture et unités de vertice était motivée par un argument simple : les unités de vertice n'accédaient jamais à la mémoire, contrairement aux unités de traitement de pixels qui doivent accéder aux textures.

8.2.1. Une forme limitée d'exécution dans le désordre

L'unité de texture est située dans le processeur de *shaders*, avec toutes les autres unités de calcul. Ceci dit, les processeurs de *shaders* ont une particularité : l'unité de texture peut fonctionner en parallèle des autres unités. Ainsi, on peut poursuivre l'exécution du *shader* en parallèle de l'accès mémoire, à condition que les calculs soient indépendants de la donnée lue.

Dans ces conditions, un *shader* doit avoir une grande quantité d'instructions à exécuter : si un accès mémoire dure 200 cycles d'horloge, le processeur de *shader* doit disposer de 200 instructions à exécuter pour masquer totalement l'accès à la texture. De plus, le *shader* effectue souvent plusieurs accès mémoire assez rapprochés : si l'unité de texture ne peut pas gérer plusieurs lectures en parallèle, la lecture la plus récente est mise en attente et bloque toutes les instructions qui la suivent.

8.2.2. Multi-threading matériel

Trouver suffisamment d'instructions indépendantes d'une lecture dans un *shader* n'est donc pas une chose facile. Les améliorations au niveau du compilateur de *shaders* des drivers peuvent aider, mais la marge est vraiment limitée. Pour trouver des instructions indépendantes d'une lecture en mémoire, le mieux est encore d'aller chercher dans d'autres *shaders*...

Sans la technique qui va suivre, chaque *shader* correspond à un programme qui s'exécute sur toute une image. Avec les techniques de multi-threading matériel, chaque *shader* est dupliqué en plusieurs programmes indépendants, des des *threads*, qui traitent chacun un morceau de l'image. Un processeur de *shader* peut traiter plusieurs *threads*, et répartir les instructions de ces *threads* sur l'unité de calcul suivant les besoins : si un *thread* attend la mémoire, il laisse l'unité de calcul libre pour un autre.

8.2.3. SIMT

Les cartes graphiques récentes fonctionnent un tout petit peu différemment de ce qu'on a vu jusqu'à présent : ils fonctionnent comme des processeurs SIMD au niveau de l'unité de calcul, mais ce fonctionnement interne est masqué au niveau du jeu d'instruction.

Ces processeurs poussent la logique des *threads* jusqu'au bout : chaque *thread* ne manipule qu'un seul pixel ou vertex. Ces *threads* sont rassemblés en groupes de 16 à 32 *threads* qui exécutent la même instruction en même temps, mais 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 fusionner ces instructions en une seule instruction vectorielle : on parle de **SIMT**.

Chaque *threads* se voit attribuer un *Program Counter*, des registres, et un identifiant qui permet de l'identifier parmi tous les autres. Un circuit spécialisé fusionne les pixels des *threads* en vecteurs qu'il distribue aux unités de calcul. L'instruction vectorielle née de la fusion de plusieurs *threads* est appelée un **warp**. Sur certaines cartes graphiques récentes, le processeur peut démarrer l'exécution de plusieurs *warps* à la fois.

Il faut noter que 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 de prédication qui utilisent *vector mask register*. Dans ce cas, chaque *thread* est traité un par un par l'unité de calcul. Ce mécanisme se base sur une pile matérielle qui mémorise les *threads* à exécuter, dans un certain ordre.

9. 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 :

— position à l'écran
— profondeur
— couleur
— valeur de stencil
— transparence alpha

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.

Ces opérations sont les opérations suivantes :

— la gestion des profondeurs des fragments ;
— les mélanges de couleurs et la gestion de la transparence ;
— l'antialiasing ;
— et enfin la gestion du stencil buffer.

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.

9.1. 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. Reste à faire ce choix.

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.

Petite précision : il est assez rare qu'un objet soit caché seulement par un seul objet. En moyenne, un objet est caché par 3 à 4 objets dans un rendu 3d de jeu vidéo.

9.1.1. Z-buffer

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

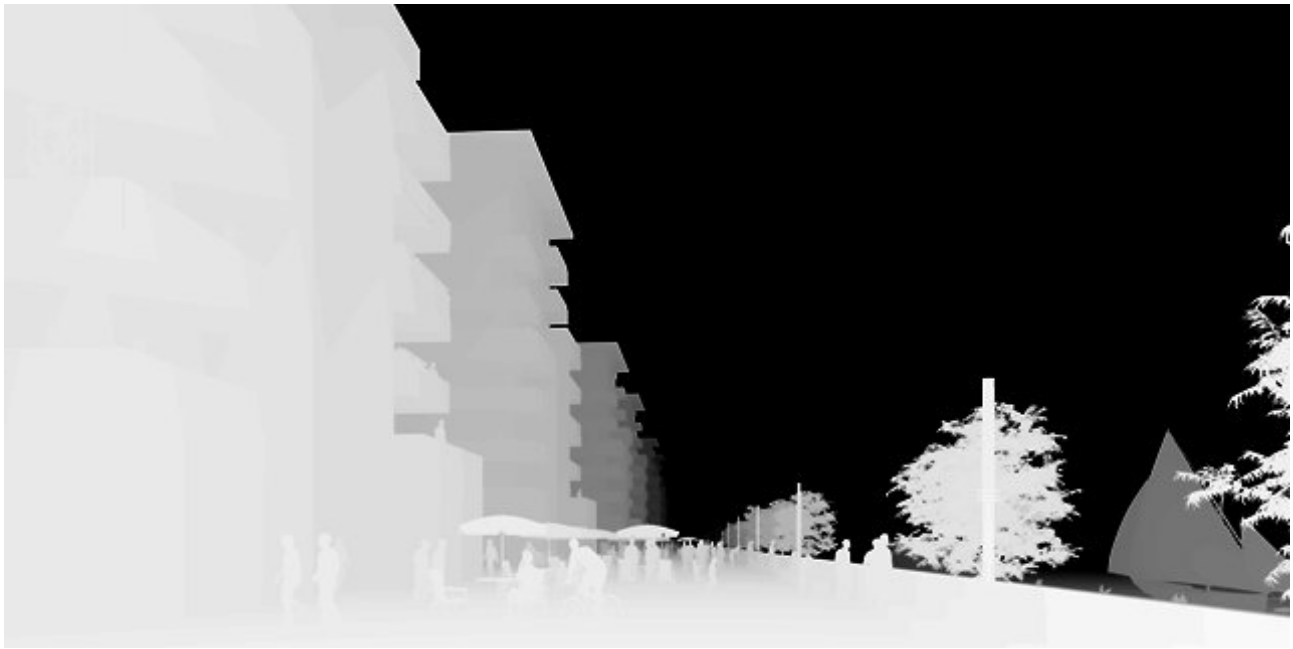

FIGURE 9.1. – Z-buffer correspondant à un rendu

Par défaut, ce depth-buffer est initialisé à une valeur de profondeur suffisamment grande. Par défaut, ce depth-buffer est remplit 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 depth-buffer, 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 depth-buffer.

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 :

http://zestedesavoir.com/media/galleries/605/3

FIGURE 9.2. – z-fighting

On peut préciser qu'il existe des variantes du depth-buffer, qui utilisent un codage de la coordonnée de profondeur assez différent. On peut notamment citer :

— l'Irregular Z-buffer ;
— le W-buffer.

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.

9.1.2. Circuit de gestion de la profondeur

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

— 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 ;
— le nombre de lectures et écritures dans celle-ci.

9.1.2.1. Z Compression

Une première solution consiste à compresser le depth-buffer. Cette compression est une compression sans-perte. Évidemment, les données devront être compressées avant d'être stockée ou lue dans le depth-buffer.

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 depth-buffer 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. C'est le circuit de compression qui décidé s'il faut compresser un bloc ou non.

9.1.2.2. Fast Z Clear

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.

9.1.2.3. Z-cache

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.

9.2. Transparence

En plus de la profondeur, il faut aussi gérer la **transparence**. Cette transparence est gérée comme une sorte de couleur ajoutée aux composantes RGB : elle est codée par un nombre, le **canal alpha**, qui indique su 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.

9.2.1. Alpha Blending

Mais comment calculer la couleur finale du pixel à partir de fragments contenant de la transparence ? 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.

Première précisions : A est la couleur de l'élément qui est devant, B est la couleur de l'élément qui est derrière, a et b sont les valeurs alpha respectives de A et B. Le calcul de la couleur finale s'effectue pour chaque composante R, G, B et alpha.

La valeur de transparence finale se calcule avec cette formule : $a_0 = a + b * (1 - a)$

Les composantes R, G, et B se calculent comme suit : $C = \frac{(A*a)+(B*b*(1-a)}{a_0}$

9.2.2. Color buffer

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 **color buffer**, une portion de la mémoire vidéo.

Calculer la couleur finale s'effectue simplement : à chaque fragment envoyé dans le ROP, celui-ci va :

— lire l'ancienne couleur, contenue dans le color buffer ;
— calculer la couleur finale en fonction de la couleur du fragment qui est devant, et de celle lue depuis le color buffer ;
— et enregistrer le résultat.

9.2.2.1. Alpha Test

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.

9.2.2.2. Effets de brouillard

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.

Le calcul de la couleur de brouillard dans l'intervalle mentionné plus haut peut s'effectuer de diverses façons :

— par un calcul linéaire : $\frac{fogend-z}{fogendd-fogstart}$
— avec une exponentielle : $\frac{1}{e^{(z*fogdensity)}}$

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.

9.2.3. Color ROP

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. Cela a une tête de déjà-vu...

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.

9.3. Anti-aliasing

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.

FIGURE 9.3. – Lettre A avec et sans anti-aliasing

9.3.1. Types d'anti-aliasing

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. Dans ce qui va suivre, nous allons voir ces différentes techniques.

9.3.1.1. SSAA - Super Sampling Anti Aliasing

Première technique : calculer l'image à une résolution supérieure, et la réduire avant de l'afficher. 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. On appelle cet anti-aliasing le **Super sampling anti-aliasing**, ou SSAA.

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.

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.

Niveau avantage, cette technique filtre toute l'image, y compris l'intérieur des textures. Niveau désavantage, le SSAA va augmenter la résolution des images à traiter, ce qui signifie augmentation de la consommation de la mémoire vidéo (frame-buffer, color buffer, z-buffer, etc) et du temps de calcul (on calcule 4 fois plus de pixels).

9.3.1.2. MSAA - MultiSampling Anti-Aliasing

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. Après tout, les bords des objets sont censés être les seuls endroits où l'effet d'escalier se fait sentir, alors pourquoi filtrer le reste ? Cette optimisation a donné naissance au **Multi-Sampling Anti-Aliasing**, abrévié en MSAA.

Avec le MSAA, l'image à afficher est rendue dans une résolution supérieure, qui contiennent 4, 9, 16, 25 fois plus de fragments. Ces fragments sont regroupés en blocs de 4, 9, 16, etc : chaque bloc correspond à un pixel à afficher. Nous allons appeler les fragments d'un même bloc des sous-pixels.

Contrairement au SSAA, les textures ne s'appliquent pas aux sous-pixels, mais à un bloc complet : tous les sous-pixels d'un bloc ont la même couleur. Avec le SSAA, chaque sous-pixel se verrait appliquer un morceau de texture indépendamment des autres, ce qui peut leur donner des couleurs différentes.

Le MSAA va jouer sur l'enregistrement de cette couleur dans le color-buffer. 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 du triangle ? Si le sous-pixel est complétement dans le triangle, sa couleur sera inchangée. 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 m"élanger les couleurs des sous-pixels du bloc (série d'interpolation linéaire, comme dit plus haut).

Niveau avantages, le MSAA n'utilise qu'un seul filtrage de texture par pixel, alors que le SSAA effectue un filtrage par sous-pixel. Niveau désavantage, il faut remarquer que 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.

9.3.1.3. FAA : Fragment Anti-Aliasing

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.

9.3.2. Position des samples

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. Chaque point peut être repéré par deux nombres flottants.

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.

9.3.2.1. Simple Grid

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.

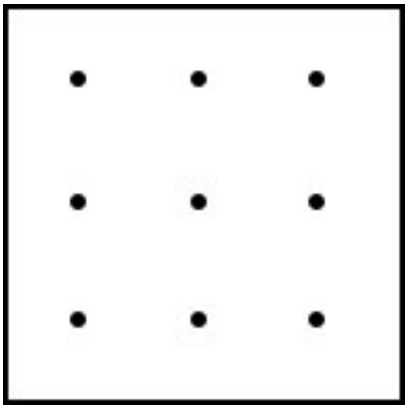

FIGURE 9.4. – Simple Grid

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.

9.3.2.2. Rotated Grid

Pour mieux gérer les bords penchés, on peut positionner nos sous-pixels comme ce ci :

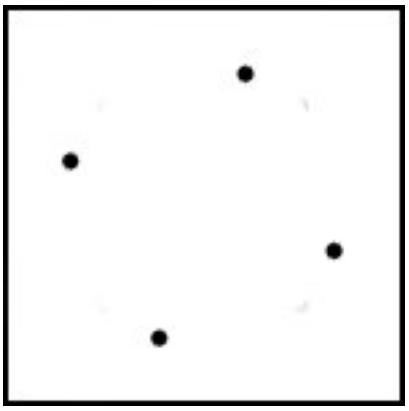

FIGURE 9.5. – Rotated Grid

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 $\sqrt{5}/2$.

9.3.2.3. Quincux

Le Quincux utilise cette grille :

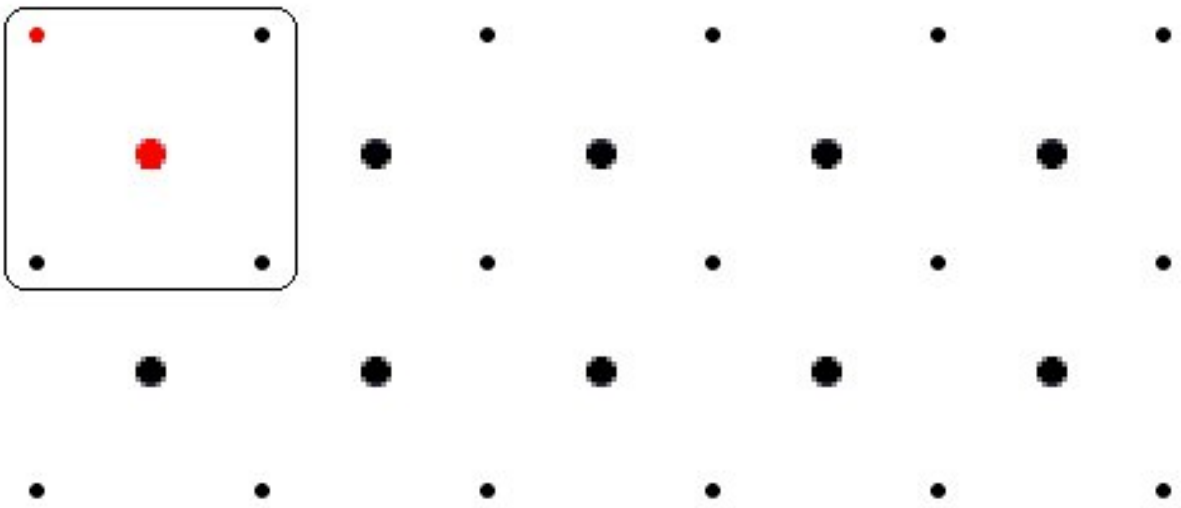

FIGURE 9.6. – Quincux

Avec cette grille, les sous-pixels sont partagés avec les pixels voisins. Une fois qu'on a calculé un sous-pixel dans un bloc, le résultat est réutilisable directement dans les calculs de couleur de l'autre bloc. Ainsi, on a besoin de calculer que deux sous-pixels pour un seul pixel : ceux indiqués en rouge sur le schéma au-dessus.

9.3.2.4. FLIPSQUAD

L'algorithme FLISQUAD réutilise l'astuce du Quincux, mais en utilisant une grille tournée, comme dans l'algorithme Rotated Grid.

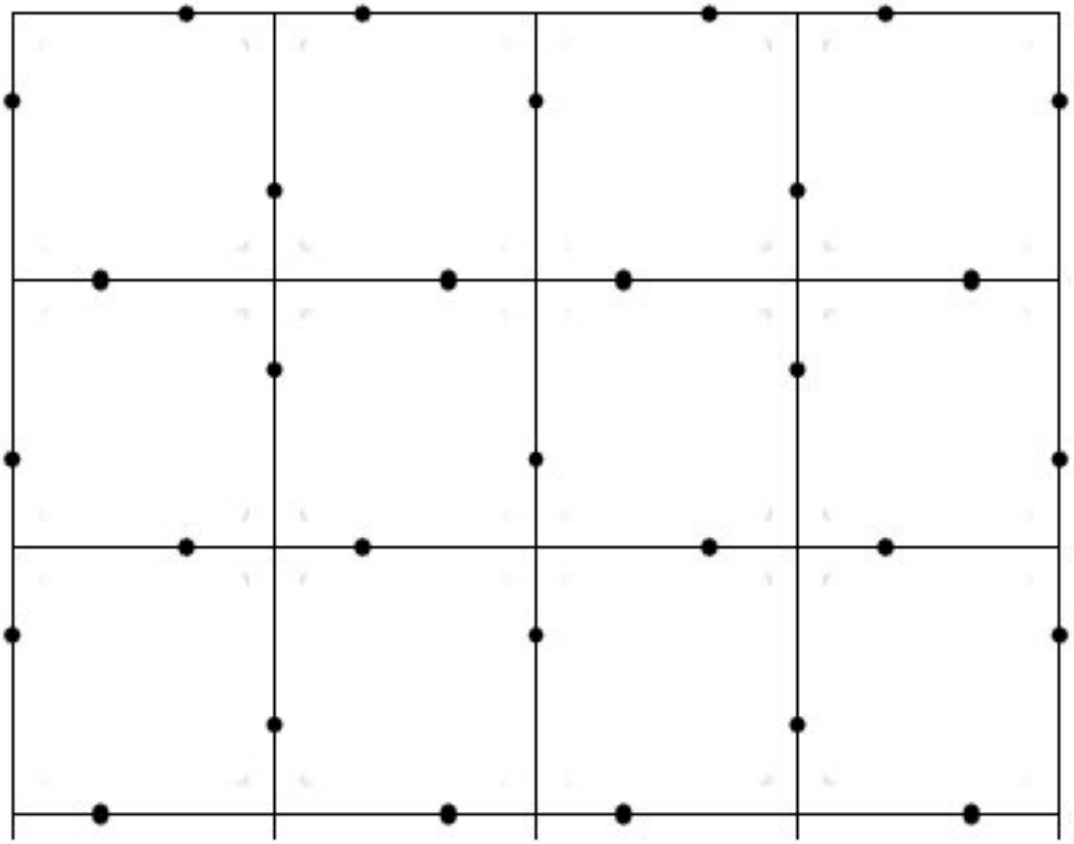

FIGURE 9.7. – FLISQUAD

9.3.2.5. Aléatoire

D'autres techniques placent les sous-pixels à des positions aléatoires dans le pixel. Et contrairement à ce qu'on pourrait croire, la qualité est souvent au rendez-vous.

En plus du Z-buffer et du color buffer, certaines cartes graphiques permettent le support matériel d'un troisième buffer, présent en mémoire vidéo : le **stencil buffer**. Celui-ci n'a pas vraiment d'utilité bien définie, et peut servir à beaucoup de choses diverses et variées. Dans la majorité des cas, celui-ci sert pour effectuer des calculs d'éclairage. D'ordinaire, ce buffer mémorise des entiers, un par pixel.

10. Elimination précoce des pixels cachés

Les cartes graphiques normales ne peuvent pas toujours éliminer les pixels cachés par des zones opaques de manière précoce. Si la détection des pixels masqués s'effectue dans les ROP, de nombreux pixels inutiles seront calculés par les pixels shaders, coloriés, éclairés, et j'en passe. Or, la profondeur d'un pixel est connue dès la fin de l'étape de rasterisation. On peut ainsi déterminer plus ou moins facilement si un fragment 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. Sur les cartes graphiques normales, l'ordre de soumission des triangles est relativement aléatoire : un objet peut en cacher un autre sans que ces deux objets soient rendus consécutivement.

Sur les cartes graphiques normales, on est obligé d'utiliser un Z-Buffer pour toute l'image : 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. Il faudrait donc trier ces objets pour obtenir un résultat correct, et obtenir le rendu souhaité. Effectuer le tri des objets avant d'effectuer un test de profondeur serait nettement plus lent que d'effectuer le test de profondeur après l'application des textures et shaders.

Mais il existe des solutions alternatives. On peut 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.

10.1. Tiled rendering

Dans ce tutoriel, nous avons abordé les cartes graphiques 3D usuelles. Mais il existe une classe de carte 3D légèrement différente, qui calcule les images à afficher d'une manière relativement différente. Sur ces architectures, l'écran/image à rendre est découpé en rectangles, rendus indépendamment, uns par uns : ces rectangles sont appelés des **tiles**. C'est pour cela que ces architectures sont nommées des **architectures à tiles**.

Le principe de ces architectures est de ne rendre que les pixels visible à l'écran, et de ne pas calculer les pixels cachés, situés trop loin, etc. On en déduit donc que 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 rendu d'une image s'effectue en plusieurs étapes :

1. la géométrie est calculée et le résultat est mémorisé dans une mémoire tampon ou en mémoire vidéo ;
2. 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 ;

3. par la suite, il suffit de rasterizer, placer les textures et exécuter les shaders chaque tile, avant d'envoyer le tout aux ROP.

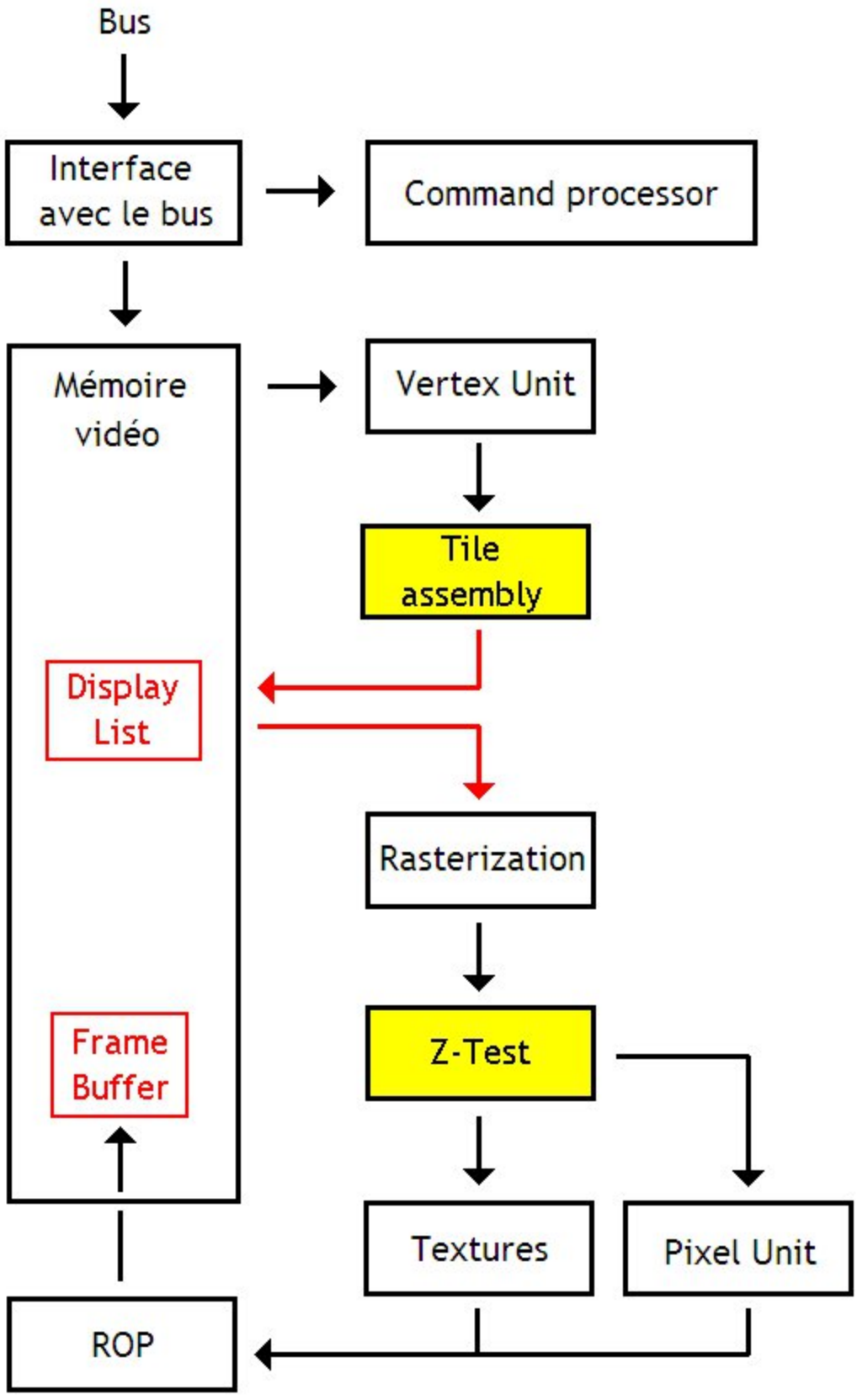

FIGURE 10.1. – Architecture tilée

Chaque étape est prise en charge par une unité spécialisée :

- la gestion de la géométrie est réalisée par les unités de vertex, qui ne changent pas ;
- la seconde étape est prise en charge par une nouvelle unité, qui remplace le rasterizer : le **Tile Accelerator** ;
- l'élimination des pixels cachés est réalisée par une nouvelle unité : l'**Image Synthesis Processor**, ou ISP ;
- on retrouve les unités de texture et les processeurs de shaders ;
- et l'écriture en mémoire vidéo est effectuée par les ROPs.

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. Sur une architecture normale, on devrait utiliser un z-buffer pour l'image entière. Une architecture à tile a juste besoin d'un Z-Buffer pour la tile en cours de traitement. De plus, les tiles sont tellement petites que l'on peut stocker tout le Z-Buffer dans une petite 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. Par exemple, le Z-Buffer n'a pas besoin d'être compressé. Idem pour le color buffer. En fait, une grande partie des accès à la mémoire vidéo disparait purement et simplement (même si une partie est remplacée par l'enregistrement des listes de triangles).

Les ROPs sont modifiés : ils ne gèrent que l'alpha blending et la la gestion de l'antialiasing. La gestion du Z-Buffer n'est plus prise en charge par les ROPs, mais est gérée par l'Image Synthesis Processor.

De plus, une mémoire tampon est généralement ajoutée aux ROP. En effet, les tiles étant relativement petites, on peut placer le résultat final dans une petite mémoire tampon au lieu d'écrire le tout dans la mémoire vidéo directement. Cette mémoire tapon permet d'accumuler le morceau d'image qui correspond à la tile, et de l'écrire d'un seul bloc en mémoire vidéo. Au lieu d'effectuer les écritures pixels par pixel, on peut y aller par blocs, ce qui est bien plus efficace.

10.2. 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**.

Avec ces techniques, la carte graphique doit faire en sorte de gérer les situations où 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. Selon le résultat de l'analyse des shaders, l'Early-Z est activé ou non, afin de garder un résultat correct à l'écran.

En somme, les techniques d'Early-Z sont assez conservatives : elles vont éliminer un pixel quand il est certain que celui-ci n'est pas affiché. Au moindre doute, les techniques d'Early-Z laissent passer le pixel. Utiliser une élimination précoce des pixels n'est donc pas sans dommages, si le test de profondeur n'est pas refait une fois les shaders calculés. Voilà qui est nettement moins performant que l'utilisation d'une Tiled Architecture...

10.2.1. Z-Max et Z-Min

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.

Les deux techniques se basent sur une même idée : l'écran est découpé en tiles. Dans chaque tile, la carte graphique vérifie si chaque pixel est affiché ou masqué. Parmi tous les pixels affichés, il y en aura un dont la profondeur sera plus élevée ou plus petite que les autres. L'unité d'Early-Z mémorise cette profondeur maximale ou minimale. Dans le cas du zmax, c'est la profondeur la plus grande qui est mémorisé, alors que le zmin mémorise la plus petite profondeur.

10.2.1.1. Z-Max

Le Z-max consiste à vérifier si la tile à rendre est situé derrière des tiles déjà rendues : si c'est le cas, la tile est masquée et on ne la calcule pas. Pour cela, il suffit de savoir quelle est la tile la plus profonde déjà rendue. Pour optimiser le tout, 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. Si c'est le cas, cela signifie que le triangle est masqué. Pour aller plus vite et pour éviter tout problème, cette vérification se fait en utilisant le pixel du triangle le plus proche (plus précisément, sa profondeur). Si cette profondeur du sommet le plus proche est supérieure à la profondeur maximale de la tile, le triangle est masqué et n'est donc pas calculé.

Ces techniques ont un gros défaut : il faut calculer la valeur maximale des pixels de la tile. Et pour cela, il n'y a qu'une seule solution : lire toutes les profondeur de la tile, et calculer la maximum. 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 une copie basse-résolution du tampon de profondeur. Cette copie basse-résolution mémorise simplement 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. La copie basse-résolution peut tenir dans une mémoire cache de la carte graphique, ou dans la mémoire vidéo, mais sa mise à jour demande que les ROPs déterminent la profondeur maximale d'une tile : cela bouffe du circuit !

Il existe d'autres techniques qui permettent d'éliminer ce genre de problèmes, comme le Depth Filter ou le Mid-texturing.

10.2.1.2. Z-Min

Avec le zmin, 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.

10.2.1.3. Hybrides

Il est parfaitement possible d'utiliser le zmax conjointement avec le zmin. On obtient alors des techniques hybrides, relativement puissantes. On pourrait citer l'exemple de l'adaptive tile depth filter.

Troisième partie

Le multi-GPU

Dans tout ce qui précède, nous nous sommes occupé d'ordinateurs où il n'y avait qu'une seule carte graphique. Mais certains d'entre vous savent qu'il est possible de mettre plusieurs cartes graphiques dans un ordinateur. Le chapitre qui va suivre va expliquer comment ce genre de prouesse est possible.

11. Multi-GPU

Combiner plusieurs cartes graphiques dans un PC pour gagner en performances. L'idée n'est pas mauvaise : si une seule carte graphique ne suffit pas, profiter des performances de plusieurs cartes graphiques peut aider à obtenir les performances voulues. Ces dernières années, les deux principaux fabricants de cartes graphiques non-intégrées ont sorti des techniques multi-GPU : le SLI et le Crossfire. Ces technologies sont destinées aux jeux vidéos, et permettent aux joueurs d'obtenir des performances excellentes avec des graphismes sublimes.

Toutefois, le multi-GPU ne sert pas qu'aux joueurs. Combiner la puissance de plusieurs cartes graphiques peut servir à bien d'autres applications. Par exemple, on pourrait citer les applications de réalité virtuelle, l'imagerie médicale haute précision, les applications de conception par ordinateur, etc. Utiliser plusieurs cartes graphiques est alors une nécessité.

Mine de rien, 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. Une seule carte graphique a tendance à rapidement rendre l'âme dans ces situations.

Le multi-GPU peut se présenter sous plusieurs formes. La plus simple consiste à utiliser plusieurs cartes graphiques séparées, reliées à la même carte mère. Nos cartes graphiques sont alors utilisées de concert, à condition que les drivers de l'ordinateur le supportent.

Dans certains cas, deux cartes graphiques sont simplement 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. Sur certaines cartes, on trouve un connecteur qui relie les deux cartes, sans passer par le bus PCI-Express : les échanges d'informations entre les cartes graphiques passent alors par ce connecteur spécialisé. Généralement, la solution la plus rapide est celle utilisant le connecteur. Si vous ne l'avez pas branché, faites-le pour obtenir de meilleures performances.

Dans d'autres cas, les deux, trois, quatre GPUs sont câblés sur une seule carte. Les communications entre GPUs s'effectuent alors via un bus ou un système d'interconnexion spécialisé, placé sur la carte. Il n'y a pas de différences de performances avec la solution utilisant des cartes séparées reliées avec un connecteur.

Contrairement à ce qu'on pourrait penser, le multi-GPU n'est pas une technique inventée par Nvidia ou AMD. Il s'agit d'une techniques très ancienne. 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.

Tout le problème des solutions multi-GPU est de répartir les calculs sur plusieurs cartes graphiques. Et c'est loin d'être chose facile. Il existe diverses techniques, chacune avec ses avantages et ses inconvénients. Cet article va vous présenter quelles sont les techniques de répartitions des calculs sur plusieurs GPUs. Vous apprendrez ainsi des choses très intéressantes,

et vous pourrez mieux configurer vos drivers de cartes graphiques ou vos jeux, pour gagner en performances.

11.1. Split Frame Rendering

Une des techniques les plus simples pour répartir nos calculs sur plusieurs cartes graphiques consiste à découper l'image en morceaux, qui sont répartis sur des cartes graphiques différentes. Ce principe s'appelle le **Split Frame Rendering**.

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. Dans 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. Dans les techniques dynamique, la taille des blocs, lignes, etc ; s'adapte en fonction de la complexité de l'image. Nous allons commencer par aborder les méthodes statiques.

11.1.1. Scan Line Interleave

Historiquement, la première technique multi-GPU fût utilisée par les cartes graphiques Voodoo 2. Cette technique s'appelait le *Scan Line Interleave*.

Comme vous le savez, l'image à rendre est composée de pixels, organisés en lignes et en colonnes. 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. Il suffit de faire calculer par chaque GPU une ligne sur 3, 4, 5, etc. Pour n GPUs, chaque GPU calculera une ligne sur n.

Cette technique avait un avantage certain pour l'époque. Avant, la résolution des images était limitée par la quantité de mémoire vidéo. Dans toutes les cartes graphiques, l'image à afficher est calculée, puis stockée dans une portion de la mémoire vidéo qu'on appelle le *framebuffer*. Celui-ci avait une taille limitée matériellement : une Voodoo 2 ne pouvait pas dépasser une résolution de 800 *600. Avec le* scan line interleave*, ce* framebuffer *était utilisé pour seulement une moitié de l'image. Tout se passait comme si 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.

11.1.2. Checker Board

Autre technique : ne pas découper l'image en lignes, mais en carrés de plusieurs pixels. Chaque carré sera alors attribué à une carte graphique bien précise. Dans le cas le plus simple, les carrés ont une taille fixe : par exemple, on peut découper une image en blocs de 16 pixels.

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 ce qui a rapport à cet objet géométrique. Elle seule aura donc besoin des données géométriques et des textures de l'objet. L'autre carte n'en aura pas besoin, et n'aura pas à charger ces données dans sa mémoire vidéo. 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.

11.1.3. Scan-lines contigues

Une technique permet de garder le gain en mémoire sans trop compliquer la tâche des développeurs de drivers. L'idée consiste à simplement couper l'image en deux, horizontalement. 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 : il suffit de découper l'image en autant de parties qu'il y a de GPUs, et attribuer chaque portion à un GPU.

Ainsi, le gain en terme de mémoire et de rendu est appréciable. De nombreux objets n'apparaissent que dans une portion de l'image, et pas dans l'autre. Le drivers peut ainsi répartir les données géométriques et les textures pour éviter toute duplication, comme dans la technique des carrés. Cela demande du travail au driver, mais cela en vaut la peine.

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. Et cette technique s'adapte aussi avec plus de deux portions en découpant l'image en N portions identiques.

Ceci dit, quelques complications peuvent survenir dans certains jeux : les FPS notamment. Dans ces jeux, plus ou moins réalistes, le bas de l'image est plus chargé que le haut. Dans le bas de l'image, on trouve un sol assez complexe, des murs, les ennemis, etc. Le haut représente souvent le ciel ou un plafond, assez simple géométriquement. Toute l'agitation a lieu vers le bas ou le milieu de l'écran, et le haut est presque vide. 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. Après tout, il vaut mieux que chaque carte doive faire 50% du travail, au lieu d'avoir une carte qui se tape 70% du boulot, et l'autre qui ne fait que 30% du rendu. 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.

11.2. Alternate Frame Rendering

Comme je l'ai dit, la répartition des différentes données entre les GPUs et la répartition des calculs pose de nombreux problèmes. Si elle est mal faite, les performances s'effondrent. Et même bien faite, on n'obtient pas un gain proportionnel au nombre de GPUs : des transferts entre les cartes graphiques sont toujours nécessaires, et la répartition ne peut pas être parfaite.

L'**alternate Frame Rendering**, ou AFR ne pose pas ce genre de problèmes. Cette technique consiste à répartir non pas des blocs ou lignes de pixels, mais des images complètes sur les différents GPUs. Au lieu de découper une image en morceaux et de répartir les morceaux sur les différents GPUs, ce sont des images complètes qui sont envoyées à chaque GPU.

Le problème mentionné plus haut disparaît alors. Plus besoin de réfléchir longuement pour savoir comment découper l'image et répartir les morceaux au mieux. Dans sa forme la plus simple, un GPU calcule une image, et l'autre GPU calcule la suivante en parallèle. Cette idée s'adapte aussi avec plus de deux GPUs.

Cette technique est supportée par la majorité des cartes graphiques actuelles. De même, la technique consistant à découper l'image en deux et à répartir les deux portions sur deux GPUs l'est aussi. Dans les drivers AMD et Nvidia, vous avez ainsi le choix entre AFR et SFR, le SFR étant la technique de découpage d'image en deux. Il faut toutefois noter que cette technique n'est pas nouvelle : c'est ATI qui a inventé cette technologie sur ses cartes graphiques Rage Fury, afin de faire concurrence à la Geforce 256, la toute première Geforce.

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.

11.2.1. Micro-stuttering

Un des défauts de cette approche, c'est le *micro-stuttering*. Lorsque nos images sont réparties sur nos cartes graphiques, elles le sont dès qu'une carte graphique est libre et qu'une image est disponible. Suivant le moment où une carte graphique devient libre, et le temps mis au calcul de l'image, on peut avoir des temps de latences entre images qui varient beaucoup.

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. Le phénomène est bien connu, surtout depuis la publication d'un article sur le site TheTechReport .

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.

11.2.2. Dépendances inter-frames

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.
