IMN638 – CHAPITRE 2 MODULE 1

PROCESSEURS GRAPHIQUES ET PIPELINE

Automne 2013 – Université de Sherbrooke

Sommaire

Introduction

Ce chapitre porte sur l'étude détaillée des composantes bas niveau d'un système de rendu. On pense donc à l'étude du fonctionnement précis d'un **pipeline graphique**, l'étude des mécanismes et **systèmes matériels** impliqués dans le rendu 3D en temps réel, et l'étude des diverses **architectures graphiques** utilisées pour le rendu en temps réel.

Comprendre le fonctionnement interne d'un dispositif de rendu en temps réel est primordial pour pouvoir construire et maintenir des applications 3D interactives performantes, efficaces et durables. Dans cette optique, comprendre les technologies de base qui constituent ces dispositifs devrait devenir un réflexe pour tout programmeur spécialisé en rendu 3D.

CONCEPT DE PIPELINE

Concept de pipeline

Le **pipeline de rendu** est considéré comme étant le chemin directeur et central d'un système de rendu en temps réel. La fonction principale du pipeline est de générer, ou **rendre** une image bidimensionnelle à partir d'informations géométriques tridimensionnelle ou bidimensionnelle.

On parle souvent de *pipeline de rendu* en faisant références aux étapes qui le constituent sans nécessairement s'attarder au concept théorique d'un pipeline. Ceci étant dit, le choix d'une architecture en pipeline demeure un élément déterminant qui affecte toutes les étapes du rendu. Les notions propre à un pipeline deviennent aussi intéressantes à étudier dans la mesure où ces dernières permettent de mieux comprendre les contraintes et faiblesses d'un processus de rendu en temps réel.

En combinant les notions de pipeline à une compréhension matérielle des systèmes de rendu 3D, il devient facile d'analyser un logiciel afin de déterminer où l'effort d'optimisation doit être concentré afin de garantir des rendus complexes en temps interactif.

Concept de pipeline

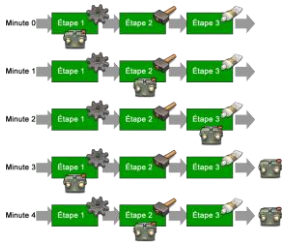
Le concept de pipeline n'est pas uniquement rattaché à l'infographie. En effet, plusieurs exemples de pipeline peuvent être trouvés dans la vie de tous les jours, notamment dans les chaînes d'assemblage en usine ou dans les canalisations de transport d'hydrocarbure.

En tout et pour tout, un pipeline, au sens théorique, est une chaîne de traitement ou de transport constituée de plusieurs étapes travaillant simultanément sur des instances séparées d'un produit similaire.

En prenant, par exemple, un pipeline de construction de voiture (une chaîne d'assemblage), la voiture à la première étape de construction se déplace à la deuxième étape lorsque la voiture de la deuxième étape est prête à passer à la troisième étape, et ainsi de suite. À tout moment, dans le pipeline de montage, plusieurs voitures sont en construction, même si seulement un nombre réduit d'entre elles sortent à chaque moment de la chaîne.

Concept de pipeline

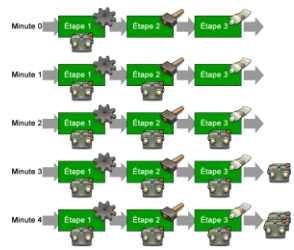
Dans un processus chaîné sans pipeline, le désavantage principal est que le système doit attendre qu'un élément ait traversé chaque étape avant d'y faire entrer un nouvel élément. Le temps de construction d'un nombre **n** d'éléments est donc équivalent à la somme des durées de chaque étapes multipliées par n.



Par exemple, si nous avons un système de construction possédant 3 étapes et que chaque étape prend exactement 1 minute à être réalisée, un item sera produit tous les 3 minutes.

Concept de pipeline

L'avantage principal d'un système en pipeline est que chaque étape, bien qu'effectuée séquentiellement, demeure effectuée parallèlement aux autres étapes. Autrement dit, il n'est pas nécessaire d'attendre qu'un élément soit sorti de la chaine avant d'en insérer un nouveau. Dès qu'une étape se libère et qu'un item peut être envoyé à cette étape, celle-ci se remet à fonctionner.



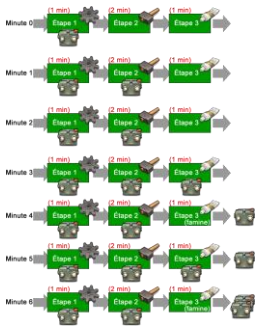
Si on reprend l'exemple précédent, chaque étape dure une minute . Après une période d'initialisation de durée équivalente à la somme de temps de chaque étape, le pipeline arrive à sortir un item à chaque minute.

Concept de pipeline

Dans un pipeline, l'étape la plus lente détermine la vitesse globale du pipeline. Par exemple, si un pipeline possède 3 étapes où la première prend 1 minute, la deuxième 2 minutes et la troisième 1 minute, la deuxième étape ne fournira pas suffisamment d'items à la troisième. Elle bloquera aussi la première étape qui devra attendre que l'item de la deuxième étape soit traité avant de pouvoir envoyer son propre item vers l'étape suivante.

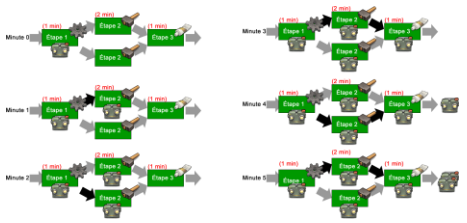
Dans un pipeline, une étape ne traitant aucun item car elle est en attente d'une étape précédente est dite **en famine**. Une étape ralentissant les étapes précédentes par sa lenteur est appelée un **goulot d'étranglement**.

Dans l'exemple ci-contre, l'étape 2 prends 2 minutes, ceci crée un goulot d'étranglement et place l'étape 3 en famine 50% du temps. Le système ne produit plus qu'un seul item tous les 2 minutes.



Concept de pipeline

Il existe différentes méthodes pour absorber ou éliminer l'effet des goulots d'étranglement. Une technique courante, et c'est celle utilisée dans un pipeline graphique, est de **paralléliser la ou les étapes les plus lentes** pour que plusieurs instances de ces mêmes étapes s'effectuent simultanément sur des instances différentes d'items.



Une autre méthode pour réduire les famines et les goulots d'étranglement est de placer des **« tampons »** pouvant accumuler temporairement des items en traitement et ainsi éviter la stagnation d'étapes plus rapide. Ceci ne fait cependant que reporter le problème plus loin dans le pipeline mais devient utile lorsqu'une étape peut changer de fonction si elle est en attente.

Concept de pipeline

Les considérations précédemment présentées pour un pipeline se traduisent telles qu'elles dans un contexte de pipeline graphique. Dans un pipeline graphique, les différentes étapes du pipeline traitent de l'information de géométrie ou de couleur. Le produit du pipeline est une couleur donnée pour un pixel particulier.

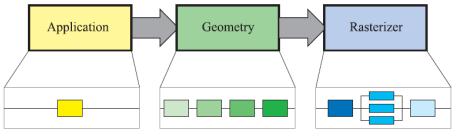
Au même titre que tout pipeline, un pipeline graphique possède des **goulots d'étranglement**, des **potentiels de famine** et certaines **étapes de traitement effectuées en parallèle**.

L'étape la plus lente du pipeline graphique déterminera la vitesse d'exécution du rendu, soit le nombre d'images par secondes produites par votre carte graphique. Ce **goulot d'étranglement peut se trouver à différents endroits dans le pipeline selon le type de scène rendue et l'architecture du processeur graphique**. Savoir isoler l'emplacement du goulot d'étranglement dans le pipeline permettra souvent d'augmenter conséquemment les performances de rendu.

ÉTUDE DÉTAILLÉE DU PIPELINE GRAPHIQUE

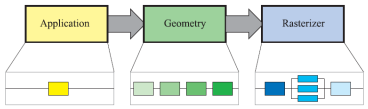
Étude détaillée du pipeline graphique

Le pipeline graphique d'une carte vidéo moderne possède en moyenne entre 110 et 150 étapes de traitement différentes (ce nombre d'étape peut monter à plusieurs milliers dans les systèmes de rendus dédiés). La plupart de ces étapes étant triviales et complètement transparentes au développeur, on divise habituellement le pipeline graphique en **étapes conceptuelles**, ces dernières englobant une série d'**étapes fonctionnelles**, qui elles englobent les **étapes réelles** du pipeline. Ces dernières sont abstraites car trop nombreuses et différentes d'une carte à l'autre pour être détaillée dans un temps raisonnable.



Étapes conceptuelles du pipeline graphique, respectivement : étape d'application, étape de géométrie, étape de rastérisation (RTR3, p.13, 2008)

Étapes conceptuelles du pipeline graphique



- **Application :** L'étape d'application s'exécute sur le processeur central de l'ordinateur. C'est dans cette étape que le développeur fourni à la carte graphique, via l'API de programmation et les pilotes vidéos, **l'information géométrique à rendre** et **la configuration des états de la carte vidéo**. Le pilote vidéo est une couche logicielle permettant le passage entre l'étape d'application et l'étape de géométrie.
- **Géométrie :** L'étape de géométrie est exécutée à l'intérieur du processeur graphique. Elle est constituée des manipulations et transformations géométriques effectuées sur chaque vertex envoyé à la carte graphique.
- **Rastérisation :** L'étape de rastérisation est aussi effectuée à l'intérieur du processeur graphique. Elle consiste à **convertir l'information vectorielle (vertex, normales, coordonnées de textures) en fragments et en pixels pour obtenir un rendu final**.

Étude détaillée du pipeline graphique

Les acétates qui suives effectueront une étude détaillée des différentes étapes du pipeline graphique. La théorie générale d'un pipeline graphique a déjà été vue dans le cours IMN428 (infographie), certaines étapes seront donc très brièvement résumées.

Certains étapes, plus complexes, seront abordées avec une plus grande précision. Notons que l'étape conceptuelle d'application a déjà été abordée en détail en IMN428 (infographie), nous débuterons donc directement à l'étape de géométrie.

Étude détaillée du pipeline graphique

L'étape de géométrie du pipeline set divise en 5 étapes fonctionnelles, soit :

- Transformation modèle-vue
- Nuance des vertex
- Projection
- Tronquage (clipping) de vision
- Conversion en coordonnées fenêtres

Étapes de fonctionnelles du pipeline géométrique. (Real-Time rendering 3, p.16, 2008)

16

Étude détaillée du pipeline graphique : étape de géométrie

Transformation modèle-vue : Cette étape consiste simplement à multiplier chaque vertex envoyé au processeur graphique par la matrice modèle-vue spécifiée au moment de l'entrée du vertex dans la carte vidéo. Notons que la matrice modèle-vue est la combinaison de la matrice « modèle », permettant de passer de l'espace objet à l'espace monde, et de la matrice « vue », représentant le mouvement de la caméra (permettant de passer de l'espace monde à l'espace de vision).

Nuance des vertex: Cette étape consiste à appliquer le calcul des informations par vertex qui seront interpolées lors de la rastérisation. Ces informations englobent, par exemple, l'éclairage, la couleur du vertex ou le calcul des coordonnées de texture.

17

Étude détaillée du pipeline graphique : étape de géométrie

Projection : La projection consiste à prendre les vertex en espace monde (transformés à l'étape de la transformation modèle-vue) et de les transformer, à l'aide de la matrice de projection, vers un espace normalisé cubique. Les limites de cet espace sont de (-1,-1,-1) à (1,1,1) sous OpenGL. Après la transformation de projection, les coordonnées des vertex sont dits être en **coordonnées normalisées d'affichage.**

La sortie de cette étape demeure une coordonnée homogène, ce n'est qu'à l'étape de conversion en coordonnées fenêtres que les composantes inutiles des vertex sont enlevées.

Notons que les trois étapes précédentes sont des étapes dites **programmables** car elles peuvent être remplacées par un nuanceur (shader). Plus d'information sur les nuanceurs seront données au chapitre suivant.

18

Étude détaillée du pipeline graphique : étape de géométrie

Tronquage de vision : Cette étape consiste à éliminer tous les éléments situés à l'extérieur du cube unitaire créé à l'étape de projection. En effet, la totalité des éléments que l'utilisateur pourra voir dans la scène se trouvent dans ce volume en particulier. Les éléments à l'extérieur n'ont pas besoin d'être rendu car ils se trouvent à l'extérieur du champ de vision. On tronque donc les éléments en question.

Plus important encore, notons que le tronquage de vision s'effectue à ce moment dans le pipeline graphique parce qu'il permet de rendre le problème de tronquage consistant. En effet, le tronquage est toujours effectué par rapport à un cube de vision unitaire, ce qui permet un calcul plus rapide du tronquage que s'il était effectué plus tôt dans le pipeline. (Les calculs sont optimisés du fait qu'on travaille toujours avec un volume unitaire aligné.)

Notons aussi qu'il ne peut être effectué avant dans une carte vidéo moderne puisque le nuanceur (shader) de vertex peut modifier la position des vertex, ce qui modifierait ou non la position d'un vertex dans le volume de vision normalisé, modifiant par le fait même le tronquage.

19

Étude détaillée du pipeline graphique : étape de géométrie

Conversion en coordonnées fenêtre : La conversion en coordonnées fenêtre consiste simplement à prendre le volume de vision et à faire correspondre chaque vertex du volume de projection normalisé à un pixel. Pour ce faire, on prend la coordonnée x et y normalisée qu'on associe au bon pixel en tenant compte des dimensions du canevas d'affichage. La coordonnée en z est ignorée pour cette étape mais sera utilisée pour l'inscription de la profondeur dans le tampon-z.

Un élément intéressant à étudier au niveau de la conversion en coordonnées fenêtres est la règle utilisée pour faire correspondre une position (x,y) normalisée à un pixel en particulier. En effet, la convention change d'une librairie graphique à l'autre, ce qui en fait un détail important à aborder dans un contexte de 3D en temps réel.

20

Étude détaillée du pipeline graphique : étape de géométrie

Conversion en coordonnées fenêtre pour OpenGL, Direct3D 10 et ses successeurs :

Dans le cas de Direct3D9, la coordonnée (0,0) d'un pixel correspond au centre de celui-ci. Un pixel va donc de la coordonnée (-0.5,-0.5) à la coordonnée (0.5,0.5).

Ceci signifie qu'une série de pixels allant de [0,9] englobera les vertex du volume de vision allant de la coordonnée [-0.5,9.5] dans notre système de rendu.

Dans le cas d'OpenGL, D3D10 et ses successeurs, la coordonnée (0,0) d'un pixel correspond au coin supérieur gauche de celui-ci. Un pixel va donc de la coordonnée (0,0) à la coordonnée (1,1).

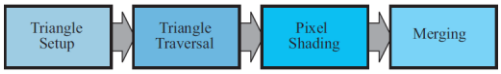
Ceci signifie qu'une série de pixels allant de [0,9] englobera les vertex du volume de vision allant de la coordonnée [0,10] dans notre système de rendu.

21

Étude détaillée du pipeline graphique : étape de rastérisation

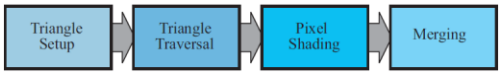
L'étape conceptuelle de **rastérisation** se divise quant à elle en quatre étapes fonctionnelles :

- Préparation des triangles
- Rastérisation des triangles
- Coloration des pixels
- Combinaison



Étapes fonctionnelles du pipeline de rastérisation. (Real-Time Rendering, p22, 2008)

Étude détaillée du pipeline graphique : étape de rastérisation



Préparation des triangles : Cette étape sert à générer des données d'interpolation et à pré-calculer des données utilisées dans l'étape qui suivra. Elle est complètement intégrée au matériel du processeur graphique et s'effectue automatiquement. Elle dépend énormément de la structure interne du processeur graphique et de l'organisation de ses tampons de mémoire temporaires. De ce fait, elle dépend donc beaucoup du modèle et du constructeur du processeur graphique. Le développeur n'a naturellement pas de contrôle sur cette étape et elle demeure transparente à celui-ci.

Rastérisation : La rastérisation des triangles consiste à prendre les triangles et à les convertis en fragments. Pour chaque fragment généré, on lui associe les données des vertexes correspondants après les avoir interpolées. (Coordonnées textures, normale, couleur, etc.) Ces informations seront utilisées à l'étape suivante.

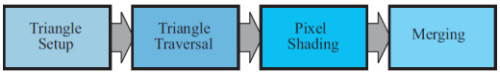
> **Note**: On parle ici de **fragments** plutôt que de **pixels** car l'écriture de ces derniers dans le tampon image n'est pas encore garantie.

Étude détaillée du pipeline graphique : étape de rastérisation

Coloration des pixels : Les deux étapes précédentes sont directement intégrées et effectuées dans les circuits imprimés dédiés du processeur graphique. Contrairement à ces dernières, la coloration des pixels est habituellement programmable et s'effectue à l'aide d'un **nuanceur de fragment (fragment shader)**. C'est à cette étape que s'effectue la coloration finale du pixel, en fonction des données fournies par la rastérisation, par les textures, etc.

> **Note:** Malgré le nom de l'étape, on parle encore ici de fragments plutôt que de pixels.

Étude détaillée du pipeline graphique : étape de rastérisation



Combinaison : La dernière et plus importante étape du pipeline de rastérisation est l'étape de combinaison. Dans cette étape se résolvent plusieurs éléments au niveau de la visibilité et de l'affichage, et s'inscrivent plusieurs informations dans les différents tampons de mémoire de la carte vidéo. On compte notamment :

- **Test pochoir** et écriture dans le **tampon pochoir** (stencil buffer).
- **Test de visibilité** et écriture dans le **tampon-z** (Z-buffer)
- **Mélange de couleur** et écriture dans le **tampon de couleur** (blending).
- Écriture dans le **canal de couleur de transparence** si applicable.

Si un fragment passe les tests pochoir et le test de visibilité du tampon-z, il est écrit dans le tampon de couleur. **C'est uniquement à ce moment que le fragment devient un pixel.**

Avant de poursuivre, regardons en détail les possibilités d'utilisation des éléments précédents.

25

PROCESSEURS GRAPHIQUES ET PIPELINE

Encart : Tampon-Z (Z-Buffer)

Comme il a été vu en infographie, le Tampon-Z sert à stocker l'information de profondeur de chaque pixel déjà rendu. Si un nouveau pixel à rendre échoue le test de profondeur, il ne sera simplement pas rendu. Si le pixel à rendre réussi ce même test, il sera rendu en fonction des paramètres de mélange fournis à la carte vidéo.

L'utilisation du Tampon-Z pour la 3D en temps réel peut être configurer de trois manière différentes :

1. Activation/Désactivation du test de profondeur
2. Activation/Désactivation de l'écriture dans le tampon-z
3. Modification de la condition de passation du test de profondeur (plus grand que, égal, jamais, toujours, etc.)

En jouant avec ces options de configuration, il est possible d'obtenir différents effets lors du rendu.

Notons que le tampon-z peut être reconfiguré de différentes façons à l'intérieur du rendu d'une même image.

26

PROCESSEURS GRAPHIQUES ET PIPELINE

Encart : Tampon-Z (Z-Buffer)

1 - Exemple d'utilisation avancée du tampon-z : On souhaite rendre un arrière-plan à notre scène. Tous les autres éléments de la scène doivent être rendus par-dessus la texture d'arrière plan.

Solution :

1. Réinitialiser le tampon-z.
2. Désactiver l'écriture dans le tampon-z et le test de profondeur.
3. Rendre la texture de fond.
4. Réactiver l'écriture dans le tampon-z et le test de profondeur
5. Rendre le reste de la scène.

Explication :

Le tampon-z reste initialisé à 0 même à l'écriture de l'arrière plan. Tous les éléments rendus par la suite, même avec le test et l'écriture activés sont donc rendus devant l'arrière plan. De plus, le fait de laisser le tampon-z initialisé à 0 permet certaines optimisations du test de profondeur qui ne seraient pas possible si l'arrière plan avait été précédemment rendu avec une profondeur élevée.

27

PROCESSEURS GRAPHIQUES ET PIPELINE

Encart : Tampon-Z (Z-Buffer)

2- Exemple d'utilisation avancée du tampon-z : On souhaite mélanger plusieurs particules transparentes superposées dans un ordre quelconque tout en s'assurant que les objets opaques situés devant les particules les cachent bien.

Solution :

1. Activer l'écriture dans le tampon-z et le test de profondeur
2. Rendre les objets opaques de la scène.
3. Désactiver l'écriture dans le tampon-z mais garder le test de profondeur activé.
4. Activer un mélanger additif des couleurs (couleur finale = couleur source + couleur destination)
5. Rendre les particules transparentes.

Explications :

Les particules se mélangent les unes aux autres indépendamment de leur profondeur puisque l'écriture dans le tampon-z est désactivé pour leur rendu. Elles sont néanmoins cachées par les éléments opaques puisque le test de profondeur demeure activé et que les éléments opaques ont été précédemment rendus dans le tampon-z.

28

Encart : Mélange de couleur et écriture dans le tampon de couleurs

Le mélange de couleur (blending) est une étape non-programmable du pipeline graphique qui demeure néanmoins extrêmement flexible et configurable. Globalement, les options de mélange de couleur permettent de configurer comment seront écrites les couleurs dans le tampon de couleur (frame buffer ou color buffer) par rapport aux couleurs qui y sont déjà présentes.

Avant de se pencher sur le mélange en soit, il est important de garder en mémoire qu'un pixel, s'il échoue le test de profondeur, ne sera jamais rendu. Le mélange de couleur, dans un tel cas, ne sera pas effectué. Autrement dit, si un pixel est déjà présent dans le tampon de couleur avec une certaine transparence, et qu'un autre pixel doit être inscrit derrière celui-ci, le pixel à inscrire échouera le test de profondeur et ne sera pas écrit, même si le pixel déjà présent est transparent.

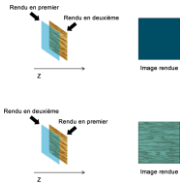
Dans cette optique, les éléments transparents d'une scène doivent donc toujours être rendu après les éléments opaque, en partant de l'élément transparent le plus loin vers l'élément transparent le plus proche pour éviter des problèmes dans l'ordre de rendu.

29

Encart : Mélange de couleur et écriture dans le tampon de couleurs

Par exemple, si notre test de profondeur est de type « plus petit que » (less-than), et qu'on rend un pixel bleu semi-transparent en premier avec une certaine profondeur, il sera combiné avec le fond (par exemple noir). Si on tente par la suite de rendre un pixel non-transparent derrière celui-ci, à une profondeur plus grande, ce pixel ne sera jamais rendu car il échouera le test de profondeur.

Pour remédier à ce problème, on inverse l'ordre de rendu pour que le pixel transparent soit rendu **après** le pixel opaque.



30

Encart : Mélange de couleur et écriture dans le tampon de couleurs

Lorsque le mélange de couleur est activé, le développeur possède un certain contrôle sur le type de mélange qu'il souhaite effectuer en spécifiant comment seront combinées les couleurs déjà dans le tampon de couleur (couleurs de destination) et celles à écrire dans celui-ci (couleurs source).

La formule générale pour déterminer quelle sera la couleur dans le tampon de couleur se définie tel que :

$$Couleur_s * Facteur_s + Couleur_d * Facteur_d = CouleurFinale$$

Où **Facteur** et **Couleur** sont des vecteurs à 4 dimensions. L'astérisque (*) dénote une multiplication point à point.

Encart : Mélange de couleur et écriture dans le tampon de couleurs

Pour des raisons de performance, les facteurs de mélange ne peuvent être arbitrairement définis. Une série de facteurs disponibles est habituellement préconfigurée dans le processeur graphique et accessible au développeur via les API graphique et le pilote vidéo. Des exemples de facteurs disponibles sont :

- (0,0,0,0)
- (1,1,1,1)
- (Src_R,Src_G,Src_B,Src_A)
- ($Dest_R$,$Dest_G$,$Dest_B$,$Dest_A$)
- (1-Src_R,1-Src_G,1-Src_B,1-Src_A)
- (Src_A,Src_A,Src_A,Src_A)
- etc.

Par défaut, lorsque les objets opaques sont rendus et que le mélange est désactivé, le facteur de la source est mis à (1,1,1,1) et le facteur de la destination à (0,0,0,0), de cette façon, si un pixel passe le test de profondeur, sa couleur remplace automatiquement la couleur précédemment inscrite dans le tampon de couleur

Encart : Tampon pochoir (Stencil Buffer)

Le tampon pochoir est un outil très puissant servant dans l'accomplissement d'une multitude d'effets en infographie temps réel.

Globalement, le tampon pochoir est un tampon similaire au tampon-z contenant des entiers positifs. Chaque fois qu'un pixel est prêt à être affiché, il est soumis à un test pochoir. Si le pixel échoue le test, il n'est pas envoyé au test de profondeur et n'est jamais affiché. Chaque fois qu'un pixel passe ou échoue le test pochoir, une opération de mise à jour sur le tampon pochoir peut être effectuée, selon le succès ou l'échec. Il est aussi possible de configurer le tampon pochoir de manière à ce qu'il soit modifié lorsque le pixel passe ou échoue le test de profondeur.

Le succès ou l'échec du test pochoir dépend du test pochoir effectué, de l'information contenue dans le tampon pochoir à ce moment et d'une variable de référence (numérique) configurée dans le code.

Rappelons que les tests et mises à jour du tampon pochoir sont effectués de manière indépendante, pixel par pixel, comme c'est le cas pour le tampon-z.

Encart : Tampon pochoir (Stencil Buffer)

Les tests pochoirs sont effectués en fonction d'une valeur de référence numérique fixée par le développeur via l'API graphique. Les tests sont effectués pour chaque pixel à rendre, au début de l'étape de combinaison.

Les tests possibles sont :

- Jamais (toujours échouer le test)
- Toujours (toujours passer le test)
- Plus petit (passe le test si la valeur dans le tampon pochoir est plus petite que la valeur de référence)
- Plus petit ou égal (passe le test si la valeur dans le tampon pochoir est plus petite ou égale à la valeur de référence)
- Égal (passe le test si la valeur dans le tampon pochoir est égale à la valeur de référence)
- Plus grand ou égal, Plus grand, Non égal, etc.

Notons que lors de la spécification du test, le développeur peut aussi spécifier un masque qui sera appliqué avec un « et » (and) binaire sur la valeur de référence et la valeur courante. Les valeurs comparées lors du test sont les valeurs sur lesquelles le masque a été appliqué. Le masque possède normalement une valeur de 0xffffffff par défaut et n'a donc pas d'effet lorsque laissé tel quel.

Encart : Tampon pochoir (Stencil Buffer)

En fonction du résultat du test, les pixels du tampon pochoir peuvent optionnellement être mis à jour. Les différentes opérations possibles à effectuer sur celui-ci sont :

- Conserver (la valeur du tampon pochoir reste la même)
- Zéro (la valeur du tampon pochoir est remise à 0)
- Remplacer (la valeur du tampon pochoir est mise à la valeur de référence)
- Incrémenter (la valeur du tampon pochoir est incrémentée pour le pixel courant)
- Décrémenter (la valeur du tampon pochoir est décrémentée pour le pixel courant)
- Inverser (une inversion binaire est effectuée pour la valeur du pixel)

Encart : Tampon pochoir (Stencil Buffer)

Finalement, notons que les opérations de l'acétate précédente peuvent être configurées différemment pour les différentes issues possible du test pochoir. Il existe trois issues possible pour le test pochoir. Remarquons que ces issues tiennent aussi compte du tampon-z, ce qui devient extrêmement utile pour certains algorithmes de rendu avancés.

Les opérations peuvent être configurées en fonction de 3 issues possibles :

- Si le test pochoir est passé et le test de profondeur est passé.
- Si le test pochoir est passé et le test de profondeur a échoué.
- Si le test pochoir a échoué.

Encart : Tampon pochoir (Stencil Buffer)

Exemple d'utilisation simple du tampon pochoir :

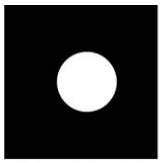
On souhaite effectuer le rendu d'une scène 3D uniquement dans un cercle au milieu de notre fenêtre de rendu et laisser le reste de l'image noir.

Étapes :

1. Réinitialiser le tampon-z et le tampon pochoir à 0.
2. Désactiver l'écriture au tampon-z.
3. Configurer le tampon pochoir pour toujours faire passer le test pochoir et incrémenter la valeur du tampon pochoir de 1 où le test passe.
4. Rendre un cercle rempli où on souhaite que le rendu final apparaisse (le tampon pochoir sera alors à 1 où le cercle a été rendu et 0 dans le reste de l'image)
5. Réactiver l'écriture du tampon-z.
6. Changer le test du tampon pochoir pour réussir seulement si la valeur du pochoir est égale à la valeur de référence 1.
7. Mettre l'opération de mise à jour du tampon pochoir à « conserver » pour éviter que sa valeur ne soit subséquemment modifiée.
8. Rendre la scène. (La scène s'affiche alors uniquement où le cercle avait été rendu.)

Encart : Tampon pochoir (Stencil Buffer)

Seulement les pixels de la scène où le pochoir avait préalablement été mis à 1 seront affichés, les autres pixels resterons noirs car ils ne seront pas affichés suite à l'échec du test pochoir.



Cercle rendu en premier qui incrémente le pochoir

Scène 3D rendue, seulement où le pochoir était incrémenté

Nous verrons des exemples d'utilisation beaucoup plus complexe du tampon pochoir dans les chapitres à venir.

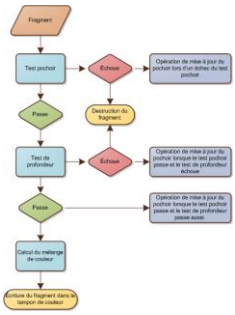
Étude détaillée du pipeline graphique : étape de rastérisation

Somme toute, on remarque que l'étape de combinaison fait appel à plusieurs types de tampons mémoire différents et soumet chaque fragment à différents tests avant qu'ils ne soient inscrits dans le tampon de couleur.

Dans l'ordre, les actions effectuées sont :

1. Test pochoir
2. Test de profondeur
3. Écriture dans le tampon-z
4. Écriture dans le pochoir
5. Calcul du mélange de couleur
6. Écriture dans le tampon de couleurs

Ces étapes effectuées, le pipeline a été complètement traversé et notre image peut maintenant être affichée.



Représentation schématique de l'étape fonctionnelle de combinaison des fragments.

GESTION DE LA MÉMOIRE VIDÉO

Gestion de la mémoire vidéo

Il y a à peine 15 ans, les cartes vidéos ne possédaient pas de mémoire dédiées au rendu. La totalité de l'information constituant la scène à rendre devait être transférée de la mémoire vive principale de l'ordinateur vers le processeur graphique à chaque image rendue. Dans un tel contexte, le transfert d'information entre la mémoire vive et le processeur graphique impliquait un passage par le bus mémoire, le contrôleur mémoire, le bus interne, le contrôleur d'entrée sortie puis le bus AGP (PCI n'existait pas à l'époque). Naturellement, le transfert était extrêmement lent, ce qui avait un impact négatif non négligeable sur la vitesse du rendu.

Pour remédier à ce problème, **les cartes vidéo modernes possèdent maintenant leur propre mémoire graphique embarquée**, ce qui **permet de stocker de l'information utile au rendu directement sur la carte vidéo**. Si cette information est réutilisée sur plusieurs images, on gagne en performance car le transfert mémoire n'a pas à être effectué.

Dans les systèmes de rendu modernes, le **goulot d'étranglement principal pour les formances de rendu demeure encore l'accès à la mémoire**, qui constitue une opération lente. Comprendre comment et où est stockée l'information devient important afin d'en optimiser le placement mémoire et, par le fait même, optimiser le rendu. Dans cette section, nous nous pencherons donc sur la question de l'allocation de la mémoire vidéo.

Mémoire vidéo et mémoire vive

En faisant abstraction des ports et divers systèmes de contrôle électroniques, une carte vidéo moderne est à toute fin pratique constituée de deux composantes principales:

• Des unités de mémoire vidéo

• Un processeur graphique

La mémoire vidéo se trouve donc généralement directement sur la carte, à proximité (d'un point de vu électronique), du processeur graphique. Elle sert à stocker de l'information utile et réutilisable dans le contexte du rendu, alors que le processeur graphique s'occupe de faire les calculs requis pour un rendu 3D.

Cette mémoire, dite **« VRAM » (pour Video Ram), s'accède relativement rapidement en lecture et en écriture à partir du processeur graphique**.

Mémoire vidéo et mémoire vive

Ceci étant dit, il faut cependant placer certains bémols quant à la vitesse de lecture et écriture de la mémoire vidéo.

Généralement, l'information de rendu **peut être stockée sur la mémoire vidéo ou sur la mémoire vive de l'ordinateur**. La carte vidéo peut donc lire et écrire sur la mémoire vidéo et sur la mémoire vive. En ordre de vitesse (plus rapide au plus lent), les opérations mémoires effectuées par la carte vidéos sont les suivantes :

1. Lecture de la mémoire vidéo.
2. Écriture dans la mémoire vidéo à partir du processeur graphique.
3. Lecture de la mémoire vive.
4. Écriture dans la mémoire vidéo à partir de la mémoire vive. (requiert l'écriture dans la mémoire vive pour ensuite transférer vers la mémoire vidéo)
5. Écriture dans la mémoire vive à partir de la mémoire vidéo. (requiert l'écriture dans la mémoire vidéo pour subséquemment transférer vers la mémoire vive)

Ceci étant dit, il faut garder en tête que l'information lue sur la carte vidéo a souvent préalablement été écrite dans cette dernière à partir de la mémoire vive. **La lecture de la mémoire vidéo est donc plus rapide que la lecture dans la mémoire vive seulement si l'information lue n'a pas besoin d'être mise à jour (réécrite) à partir de la mémoire vive trop souvent.**

Mémoire vidéo et mémoire vive

Un exemple classique de cas où l'ordre de performance précédent ne s'applique pas est le transfert d'une vidéo vers la carte graphique.

En effet, pour afficher une vidéo en utilisant un processeur graphique, une texture doit être remplie et affichée pour chaque image de la vidéo. La texture est donc très souvent réécrite.

Dans ce contexte, l'option de transférer la texture vers la mémoire vidéo à chaque image, pour ensuite la relire de la mémoire vidéo à l'affichage serait moins performante que de simplement laisser la texture en mémoire vive et indiquer au processeur graphique d'aller la lire en mémoire vive directement.

Il serait donc plus optimal d'indiquer au pilote, dans cet exemple, que la texture contenant les images de la vidéo devrait rester en mémoire vive pour augmenter les performances de rendu et diminuer les délais d'écriture en mémoire vidéo.

Cet exemple introduit des concepts importants en gestion de la mémoire vidéo, soit le dynamisme et le type d'accès de l'information utilisée pour le rendu.

Mémoire vidéo et mémoire vive

Le dynamisme et le type d'accès de l'information de rendu sont des concepts de gestion de mémoire permettant de *suggérer* au pilote vidéo où stocker l'information à l'aide d'indices sur son utilisation. En spécifiant correctement le dynamisme et le type d'accès à l'information, le pilote vidéo arrive à optimiser automatiquement le placement de l'information en mémoire afin de maximiser les performances d'accès.

Le dynamisme de l'information de rendu se divise en deux catégories :

• **Information volatile (ou dynamique) :** De l'information qui change très souvent (1 fois ou plus tous les 1 ou 2 images).

• **Information statique :** Information de rendue qui change peu souvent et reste constante sur plusieurs images rendues.

Les types d'accès en mémoire se divisent quant à eux en trois catégories :

• Lecture seule

• Écriture seule

• Lecture et écriture

Le type d'accès en mémoire représente le type d'accès aux données par l'application et non par le processeur graphique.

Mémoire vidéo et mémoire vive

Selon le type d'information et son dynamisme, le pilote de la carte vidéo décidera où placer l'information.

Une **information dynamique** sera habituellement placée dans la mémoire vive et lue par la carte vidéo. Ceci se justifie par le fait que lire la mémoire vive à partir de la carte vidéo demeure plus rapide que de devoir écrire dans la mémoire vidéo puis subséquemment la lire. Dans la mesure où l'information dynamique change souvent, il est préférable d'éviter l'écriture dans la mémoire vidéo.

Une **information statique** sera quant à elle habituellement placée en mémoire vidéo directement car elle n'aura pas besoin d'être accédée directement par l'application après y avoir été inscrite.

Mémoire vidéo et mémoire vive

Le dynamisme et le type d'accès en mémoire permettent différentes optimisations. Voici un exemple d'optimisation effectué dans le pilote vidéo :

1. **Données de vertex statiques envoyés en écriture/lecture :** Le pilote vidéo stocke les données en mémoire vidéo puisqu'elle est statique mais garde aussi une copie en mémoire vive. Lorsqu'elle doit être mise à jour, la copie en mémoire vive est utilisée puis transférée vers la copie en mémoire vidéo de manière asynchrone dans un moment opportun de la boucle de rendu.

Mémoire cache vidéo

Sans étudier en détail les algorithmes de mise en cache des processeurs graphiques, notons que ces derniers, au même titre que les processeurs centraux, **mettent en cache des intervalles de mémoire afin de limiter le nombre d'accès à la mémoire vidéo ou à la mémoire vive lors du rendu**.

Par exemple, si un processeur graphique doit accéder à un pixel d'une texture, il placera habituellement en cache plusieurs pixels environnant de cette même texture. Le processeur prend alors pour acquis que les données à proximité en mémoire seront-elles aussi utilisées dans un court délais.

Cette méthode de fonctionnement devient importante à considérer dans la mesure où **la cache du processeur vidéo est souvent libérée lorsque celui-ci change d'état de rendu**. La cache doit alors être reconstruite au fil du rendu. Il est donc **important de limiter les changements d'état inutiles afin de préserver la cohérence de la cache le plus longtemps possible**.

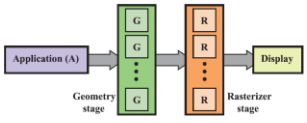
Par exemple, permuter deux textures a souvent pour effet de libérer une très grande partie de la cache.

ARCHITECTURES GRAPHIQUES

Architecture générale

Jusqu'à présent, nous avons étudié une représentation abstraite pouvant être perçu comme le dénominateur commun entre les différentes architectures graphiques existantes pour le rendu en temps réel. Cette représentation débute par une étape d'application, suivie d'une étape de géométrie puis terminant avec la rastérisation.

Pour garantir des performances adéquates, les étapes de **géométrie** et de **rastérisation** sont habituellement parallélisées grâce à une série d'unités de calculs fonctionnant en parallèle. Ces unités peuvent être perçues comme de petits processeurs indépendants contenus dans le processeur graphique.



Modèle simplifié d'une architecture de rendu (Real-Time Rendering 3, p.842, 2008)

Architecture générale

Chaque unité de calcul d'un processeur graphique fonctionnent indépendamment des autres et possède une très petite mémoire cache dédiée à l'unité. Elle a aussi accès à une mémoire cache beaucoup plus grande qui est commune à un groupe constitué de quelques dizaines d'unités de calcul. **Pour optimiser le rendu, il est important de s'assurer que les divers groupes d'unités de calcul travaillent sensiblement sur les mêmes données,** afin de garantir une meilleure cohérence de la cache et donc une meilleure performance.

Similairement, le processeur graphique tente aussi de **s'assurer que les divers groupes d'unité ont un travail équivalent**, afin d'éviter que certains groupes soient en surcharge de travail alors que d'autres sont en famine.

Dans cette optique, le **processeur graphique tente de diviser et répartir le travail entre les groupes de calcul de manière à optimiser l'utilisation de cache et la charge de travail entre chaque groupe**.

Bien entendu, **cette répartition possède un coût de performance non-négligeable**. L'effectuer à chaque étape du processus (avant la géométrie et avant la rastérisation) finirait par annuler le gain de performance qu'elle permet.

Architectures graphiques

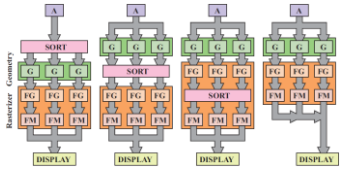
Placer l'étape de répartition à divers endroits dans le pipeline permet d'obtenir des gains de performances différents, et de diviser le travail différemment au niveau du processeur graphique et des unités de calcul. C'est donc **l'emplacement du dispositif de répartition dans le pipeline qui détermine le type d'architecture d'un processeur graphique**.

Molnar et al. ont présenté une classification des architectures de rendu en les classant en fonction de l'emplacement de la phase de répartition au sein du pipeline. On compte donc quatre familles d'architectures différentes :

- Architecture de répartition première.
- Architecture de répartition intermédiaire.
- Architecture de répartition de fin au niveau du fragment.
- Architecture de répartition de fin au niveau de l'image.

Architectures graphiques

Les quatre architectures sont représentées dans l'image suivante :



De gauche à droite : Répartition première, répartition intermédiaire, répartition de fin au niveau du fragment, répartition de fin au niveau de l'image. (Real-Time Rendering 3, p.843, 2008)

Notons que les unités de rastérisation sont ici divisées en 2 sous-unités. Soit la sous-unité de génération des fragments (FG), qui s'occupe de la rastérisation en soit, et la sous-unité de mélange des fragments (FM), qui s'occupe du mélange de couleurs (l'étape de combinaison).

Architecture de répartition première

Fonctionnement :

1. L'information utile au rendu est envoyée par l'application
2. On effectue le strict minimum de calcul requis sur les vertexes pour les emmener en espace image.
3. On divise l'image en tuiles et on associe un groupe de calcul à chaque tuile.
4. On envoie chaque triangle aux unités de calcul correspondant aux tuiles où il son affichés. On y effectue le reste de l'étape de géométrie et la rastérisation.
5. Lorsqu'une unité de calcul a fini de rendre sa tuile, elle renvoie cette dernière à une unité centrale qui s'occupe de positionner les tuiles dans l'image.

Architecture de répartition intermédiaire

Fonctionnement :

1. L'information utile au rendu est envoyée par l'application.
2. On effectue la totalité du traitement de la géométrie. (Offre une meilleure balance de charge pour cette partie)
3. On divise l'image en tuiles et on associe une unité de calcul de rastérisation à chaque tuile.
4. On envoie chaque triangle déjà transformé, en coordonnée fenêtre, aux unités de calcul de rastérisation correspondant aux tuiles où ils sont affichés.
5. Lorsqu'une unité de calcul a fini de rendre sa tuile, elle renvoie cette dernière à une unité centrale qui s'occupe de positionner les tuiles de manière à former une image complète.

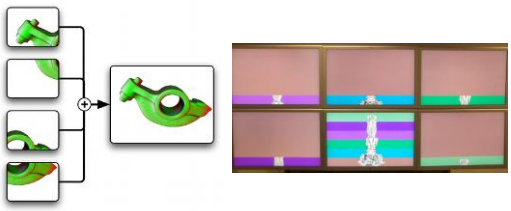
Architectures de répartition première et intermédiaire

Notons que les deux architectures précédentes, soit la **répartition première** et la **répartition intermédiaire** sont peu utilisées dans un contexte d'ordinateur simple avec une seule carte vidéo.

On utilise surtout ces architectures pour des cas où le rendu est distribué sur plusieurs cartes vidéos puisque la division en tuile se porte bien à une telle situation. Dans un tel cas, chaque carte se charge d'afficher un certain nombre de tuiles (une partie de l'image) et se divisent le travail de cette façon. Les tuiles sont ensuite envoyées à une des deux cartes qui s'occupe de former l'image.

Architectures de répartition première et intermédiaire

Représentation du rendu en tuiles des architectures de **répartition première** et **répartition intermédiaire.**



[Equalizergraphics.com, 2009]

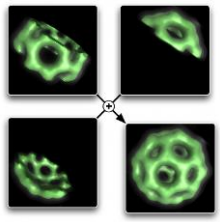
Architecture de répartition de fin au niveau du fragment

Fonctionnement :

1. On envoie les informations à rendre de l'application.
2. On effectue la totalité de la partie géométrique du pipeline.
3. On génère les fragments par rastérisation.
4. L'espace 3D du volume canonique de vision est divisé en différentes régions. On associe un groupe de calcul pour le « shading » et le mélange de chaque région.
5. Tous les fragments se trouvant dans une certaine région spatiale sont envoyés à l'unité de calcul associée à cette région. (Chaque unité de calcul possède son propre tampon de couleurs (frame buffer) et son propre tampon-z (z-buffer)).
6. Une fois tous les fragments générés, on combine les images produites à l'aide des tampons de couleur et des tampon-z générés dans chaque image. (On se sert de ceux-ci pour résoudre la visibilité si des fragments sont superposés.)

Architecture de répartition de fin au niveau du fragment

Représentation de la **répartition de fin au niveau du fragment** :



[Equalizergraphics.com, 2009]

Exemple réel d'une architecture de répartition de fin au niveau du fragment. L'espace de vision canonique est divisé en différents blocs qui sont rendus par différents processeurs graphiques puis recombinés dans un processeur graphique central (écran du bas au centre) [Equalizergraphics.com, 2009]

Ce type d'architecture est avantageux dans la mesure où il garanti une bonne balance de charge jusqu'à très tard dans le pipeline. Il demande cependant une bande passante très importante au niveau du groupe de calcul central où les images sont recombinées.

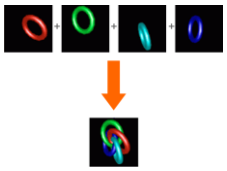
Architecture de répartition de fin au niveau de l'image

Fonctionnement :

1. On envoie les informations à rendre à l'unité de calcul la moins chargée. Chaque unité de calcul contient un pipeline complet (géométrie et rastérisation) complètement indépendant et non-communiquant avec les pipelines des autres unités de calcul. Les données lorsqu'elles entrent dans un pipeline restent dans celui-ci jusqu'à la production du pixel final.
2. On procède au rendu complet pour chaque unité de calcul.
3. Chaque unité de calcul possède son propre tampon image (frame buffer) et écrit dans celui-ci de manière désordonnée. Comme pour l'architecture précédente, chaque unité possède son propre tampon-z.
4. On combine les images en les superposant et en tenant compte de la profondeur via le tampon-z préservé de chaque image.

Architecture de répartition et fin au niveau de l'image

Représentation de la répartition de fin au niveau de l'image :



[VirtualGL, 2006]

Notons que la répartition de fin au niveau de l'image demande une quantité très élevée de mémoire car elle nécessite autant de tampon image (frame buffer) et de tampon-z (z-buffer) qu'il y a d'instance parallèle de rendu.

Architecture de répartition de fin au niveau de l'image et du fragment

Pour les architectures avec **répartition de fin**, l'avantage principal demeure la balance de charge sur la majorité du pipeline.

Un exemple de système populaire utilisant un rendu avec répartition de fin est la console de jeu vidéo PlayStation 3. Dans ce système, la distribution de la géométrie s'effectue juste avant le nuanceur de fragment (fragment shader), et une combinaison est effectuée juste avant le rendu. On a, dans ce cas, une **répartition de fin au niveau du fragment.**

Référence recommandée

[Livre] T. Akenine-Möller, E. Haines et N. Hoffman, **« Real-Time Rendering »,3rd edition**, *A.K. Peters, Ltd.*,Natick, MA, USA, 2008 (chapitre 2 et chapitre 18)