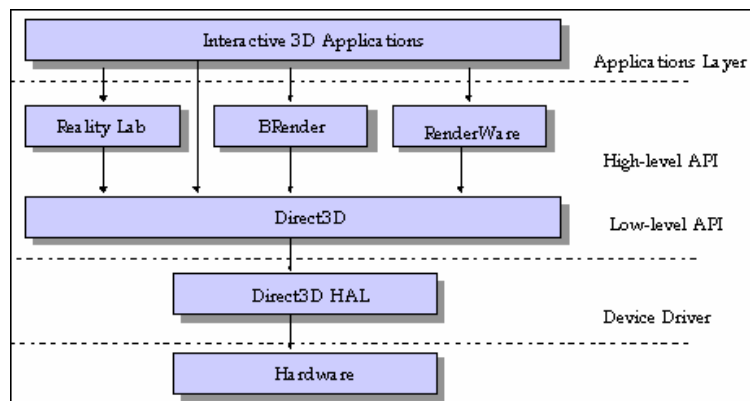

Pipeline graphique et GPUs

Daniel Etiemble
de@lri.fr

Les applications graphiques

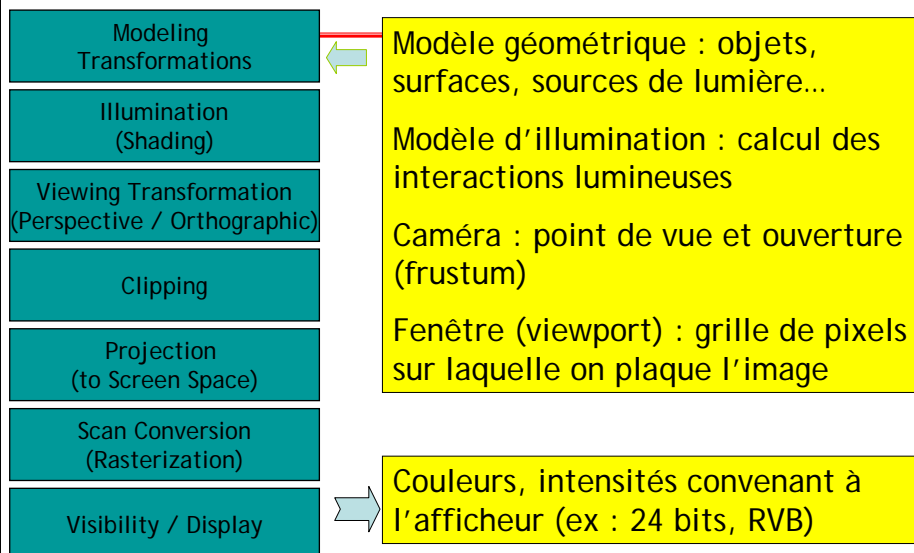
De l'application graphique au matériel : exemple Direct3D



OpenGL vs. DirectX

- Seulement graphique
- Interfaces standard C
- Machine d'états
- Multi-plateformes
- Usage académique
- Graphique, multimédia, etc.
- Interfaces C++
- Orienté objets
- Windows
- Jeux PC

Le pipeline graphique



Le pipeline graphique

Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display

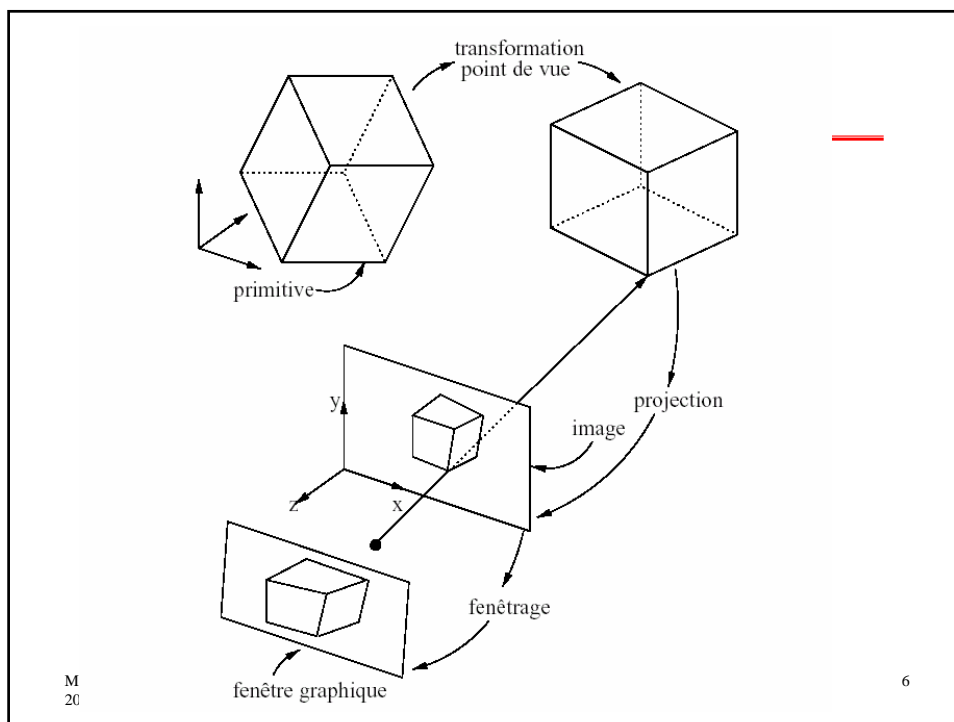
Chaque primitive passe successivement par toutes les étapes

Le pipeline peut être implémenté de diverses manières avec des étapes en matériel et d'autres en logiciel

A certaines étapes, on peut disposer d'outils de programmation (ex : programme de sommets ou programme de pixels)

Optimisations pour graphique et multimédia
D. Etiemble

5



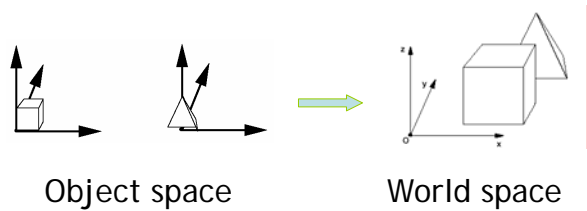
M
20

6

Changements de coordonnées

Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display

Passage du système de coordonnées local de chaque objet 3D (object space) vers un repère global (world space)



Optimisations pour graphique et multimédia
D. Etienne

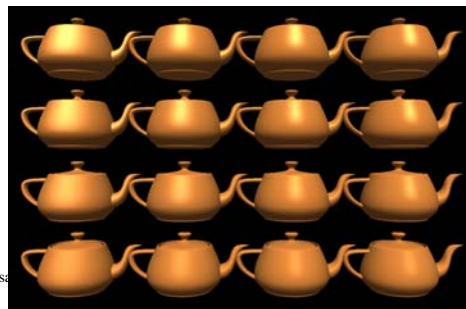
7

Illumination

Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display

Les primitives sont éclairées selon leur matériau, le type de surface et les sources de lumière.

Les modèles d'illumination sont locaux (pas d'ombres) car le calcul est effectué par primitive.



Optimisa

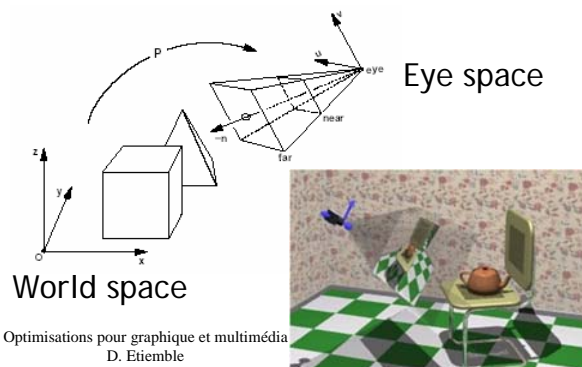
8

Transformation caméra

Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display

Passes des coordonnées dans l'espace à celles du point de vue (repère caméra ou « eye space »).

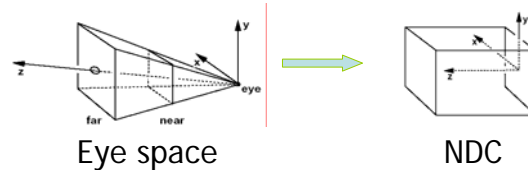
En général le repère est aligné selon z.



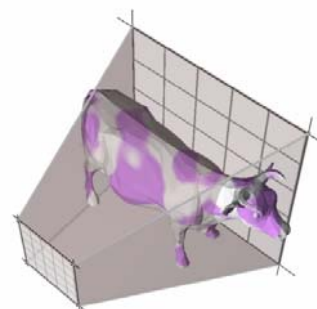
Clipping

Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display

Coordonnées normalisées (NDC : normalized device coordinates)



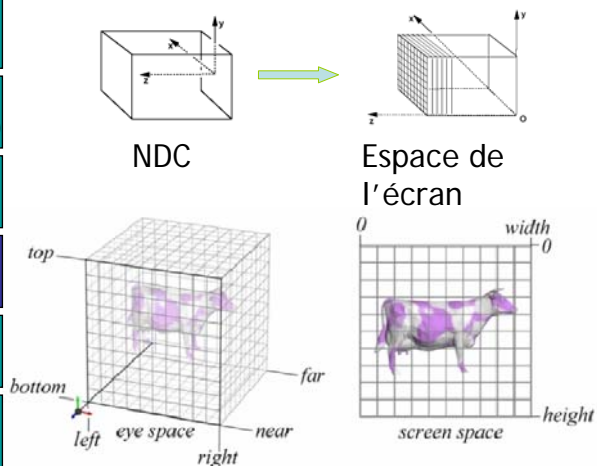
Les portions en dehors du volume de vue (frustum) sont coupées.



Projection

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

Les primitives 3D sont projetées sur l'image 2D (espace de l'écran)

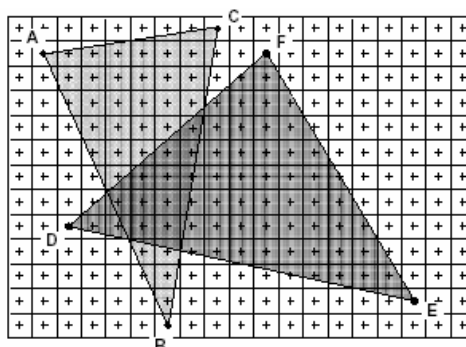


Rasterisation

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

Découpe la primitive 2D en pixels

Interpole les valeurs connues aux sommets : couleur, profondeur, ...



Visibilité, affichage

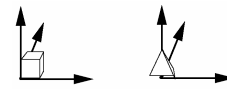
Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display

Calcul des primitives visibles : z-buffer.

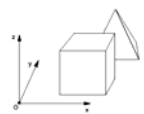
Remplissage du tampon de trames avec le bon format de couleur.

Systèmes de coordonnées

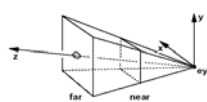
Modeling Transformations
Illumination (Shading)
Viewing Transformation (Perspective / Orthographic)
Clipping
Projection (to Screen Space)
Scan Conversion (Rasterization)
Visibility / Display



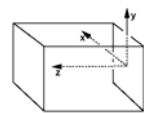
Object space



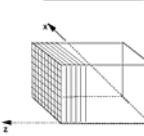
World space



Eye Space / Camera Space

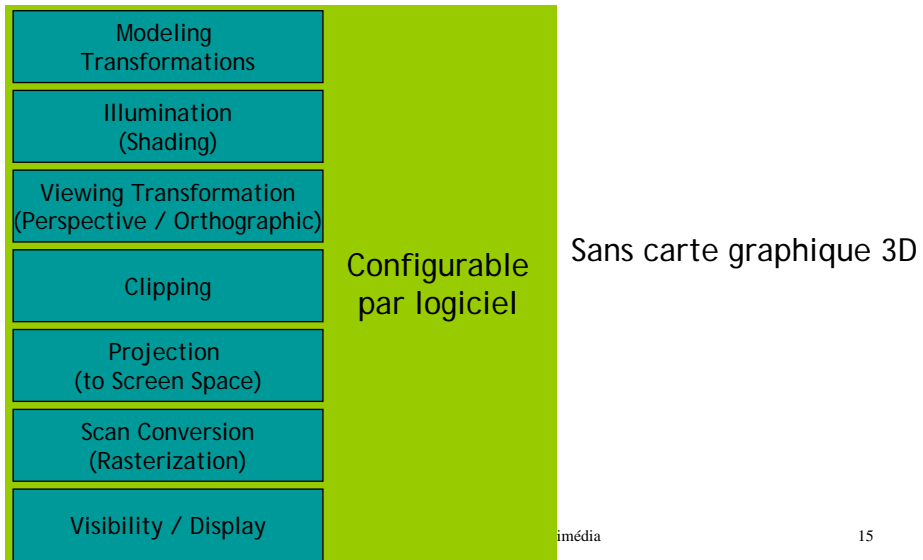


Clip Space (NDC)

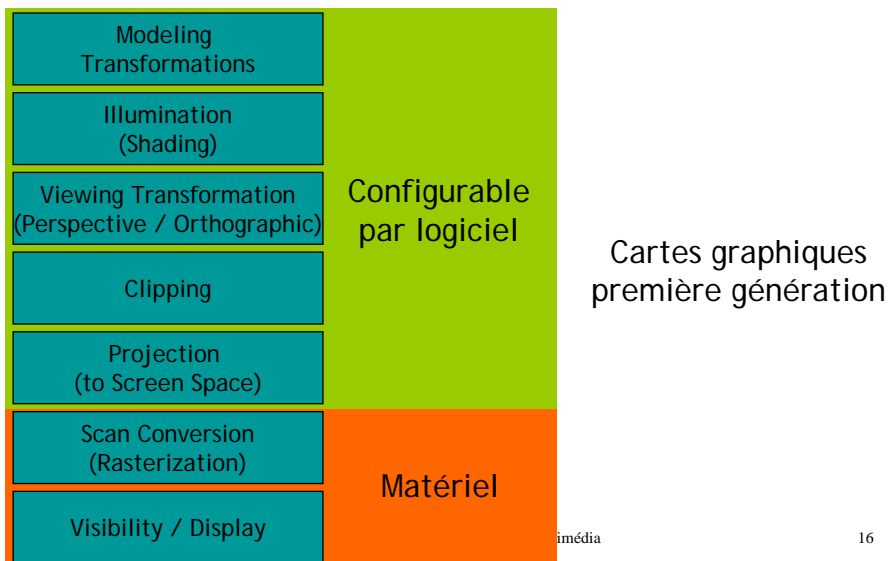


Screen Space

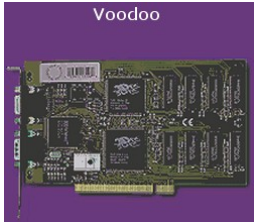
Le pipeline graphique



Le pipeline graphique

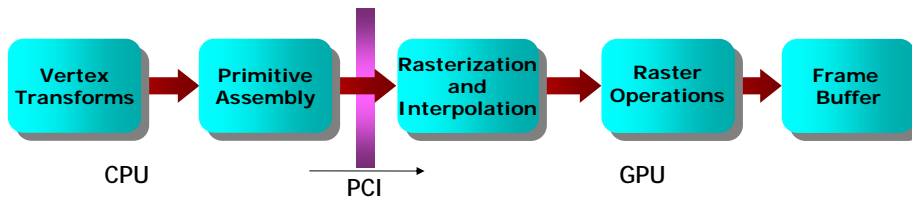


Generation I: 3dfx Voodoo (1996)



<http://acceleation.com/?ac.id.123.2>

- Une des premières vraies cartes de jeu 3D
- Ajoute à la carte video standard 2D
- Ne faisait pas les transformations de sommets : faites par le CPU
- **Faisait** le mappage des textures, le z-buffering.

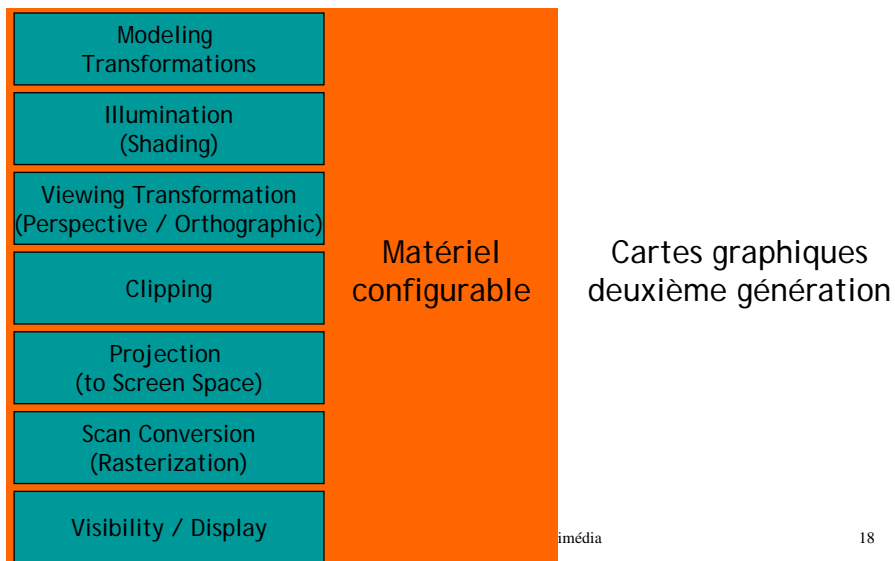


Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etienne

17

Le pipeline graphique



imédia

18

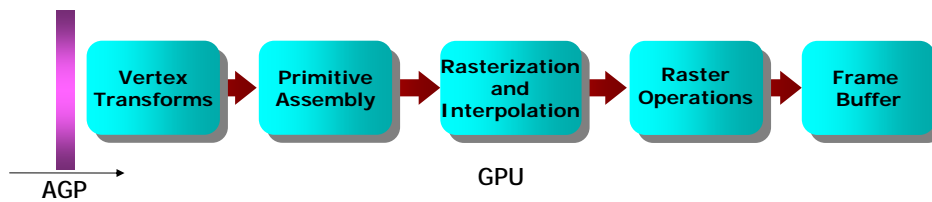
Generation II: GeForce/Radeon 7500 (1998)

GeForce 256



<http://accelenation.com/?ac.id.123.5>

- **Innovation principale** : fait passer les calculs de transformation et d'éclairage au GPU
- Permettait plusieurs textures : bump maps, light maps, et autres.
- Bus AGP au lieu du bus PCI

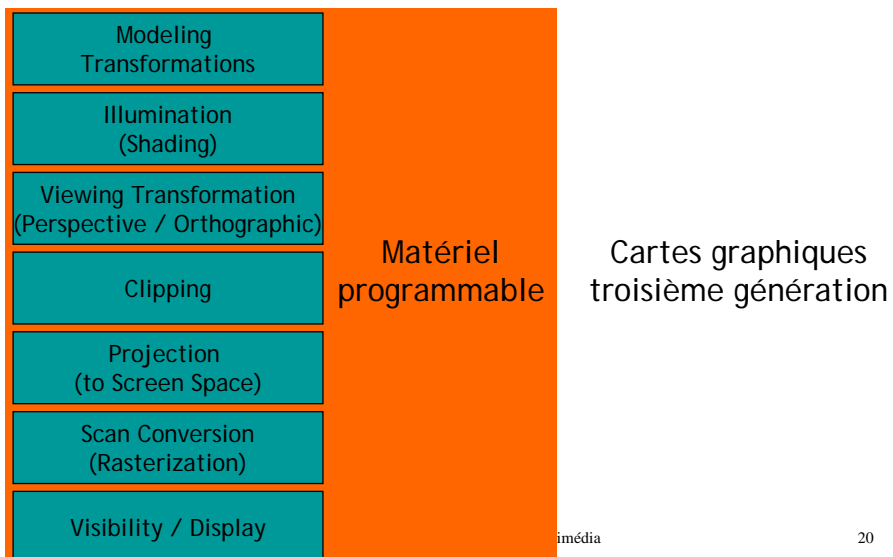


Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etienne

19

Le pipeline graphique



multimédia

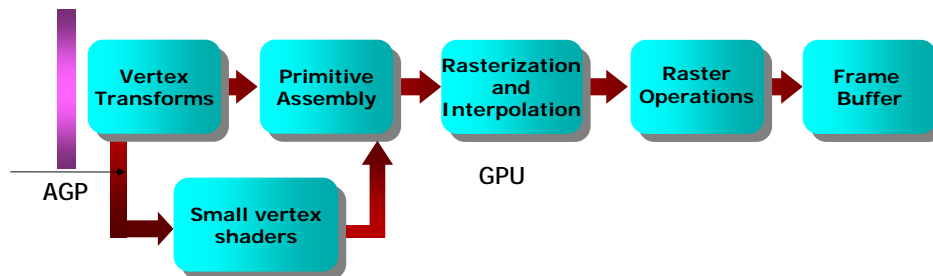
20

Generation III: GeForce3/Radeon 8500(2001)



<http://acceleration.com/?ac.id.123.7>

- Pour la première fois, permettait une programmation limitée au niveau du pipeline des sommets
- Permettait également les textures de volume et le “multi-sampling” pour l’antialiasing.



Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etienne

21

GeForce3

- 2001: GeForce3 (NV20)
 - Coeur 240 MHz /Mémoire 500 MHz
 - 57 millions transistors
 - 46-76 Gigaflops
 - Technologie “Vertex shader”
 - Technologie “Pixel shader”
 - Architecture mémoire “LightSpeed”

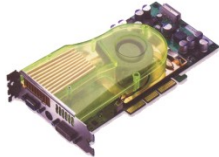
Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etienne

22

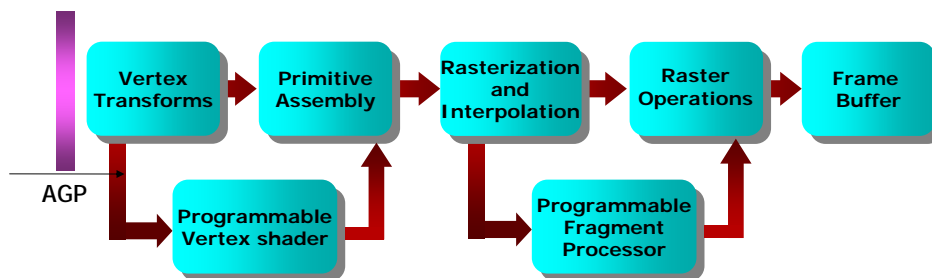
Generation IV: Radeon 9700/GeForce FX (2002)

GeForce FX



<http://acceleration.com/?ac.id.123.8>

- C'est la première génération des cartes graphiques totalement programmables
- Les différentes versions ont des limites différentes sur les possibilités des programmes de fragment et de sommets.



Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

23

GeForce FX

- Novembre 2002: Geforce FX (NV30)
 - 16 variantes
 - 125 millions transistors
 - 8 pixels/période d'horloge
 - 1 tmu/pipeline (16 textures/unité)
 - Interface mémoire 128 bits
 - Compatible avec mémoires 128 Mo/256 Mo

Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

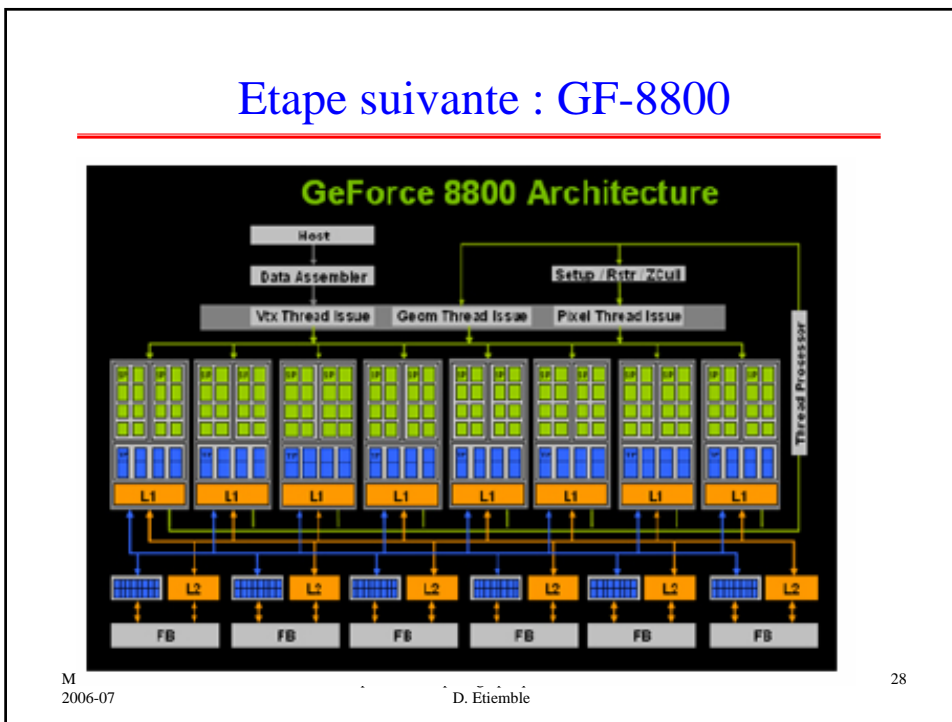
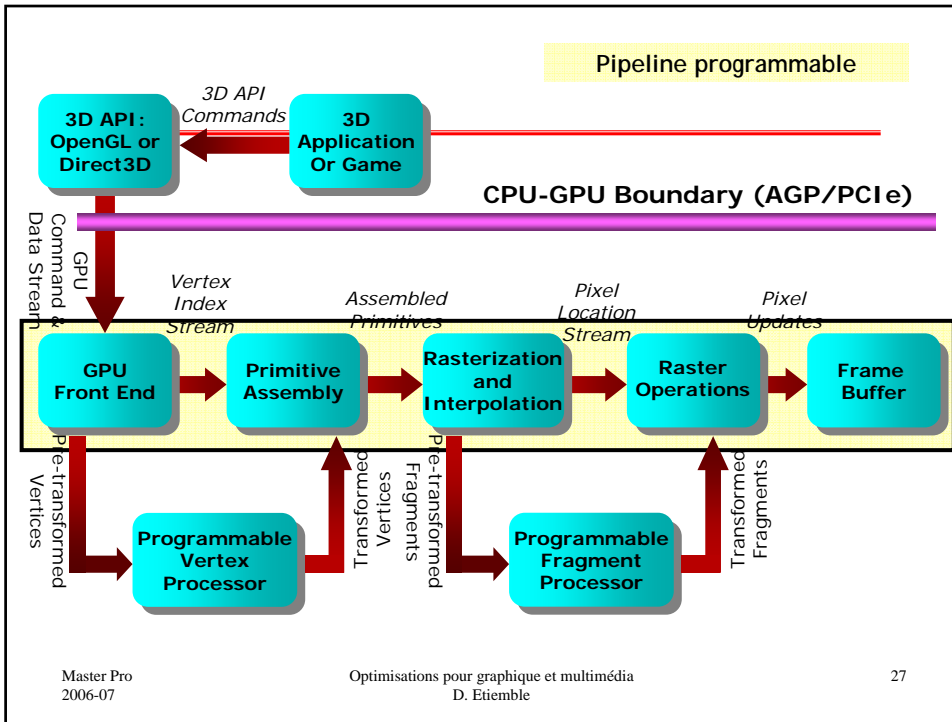
24

Séries GeForce 6

- Série GeForce 6 (NV40)
 - 6200; 6600 GT et Ultra; 6800 GT, Ultra, et Ultra Extreme
 - Fréquence horloge coeur 450 MHz
 - Fréquence horloge mémoire 600 MHz
 - Unités “vertex shader” à 6 MADD de 4 “floats”/ période d’horloge
 - Unités “pixel shader” à 16 MADD de 4 “floats”/ période d’horloge

Séries GeForce 6

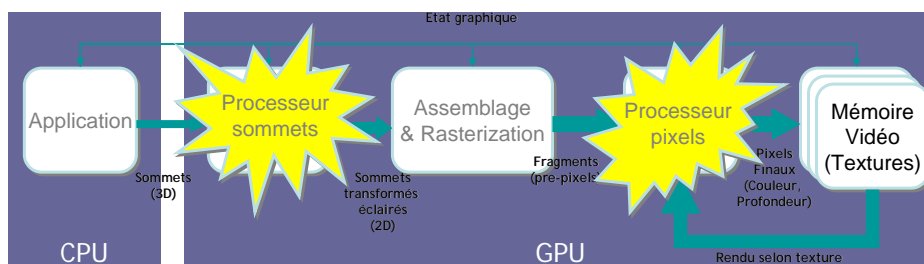
- Architecture superscalaire à 16 pipelines
- Moteur CineFX3.0
- Toutes les opérations effectuées en flottants 32 bits pour chaque composante
- 200 Gigaflops (contre 6,4 Gigaflops pour Itanium)



Goulets d'étranglement possibles de la performance GPU

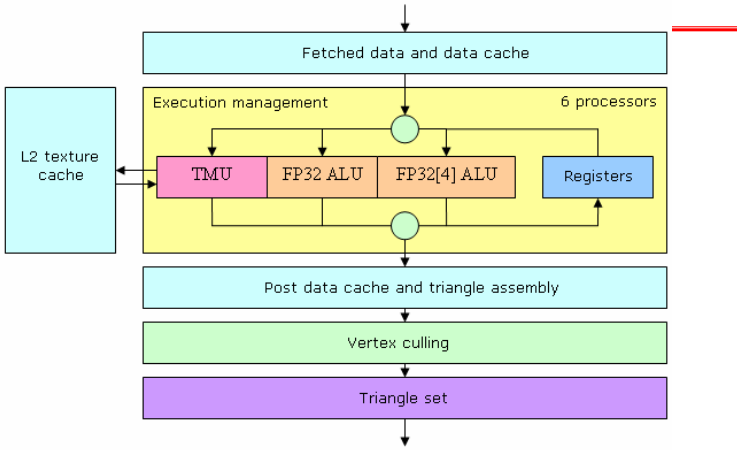
- Limite bus CPU
 - Pas capable d'envoyer suffisamment de sommets à la carte
- Limite "sommets"
 - Le "processeur" sommets est chargé à 100% alors que le "processeur" de fragments attend pour récupérer les données dès qu'elles sont prêtes
- Limite "pixel"
 - Le "processeur" pixels est chargé à 100% et le "processeur" de sommets attend avant d'envoyer plus de données

Pipeline OpenGL moderne



- Processeur sommets programmable
- Processeur Fragment (Pixel) programmable

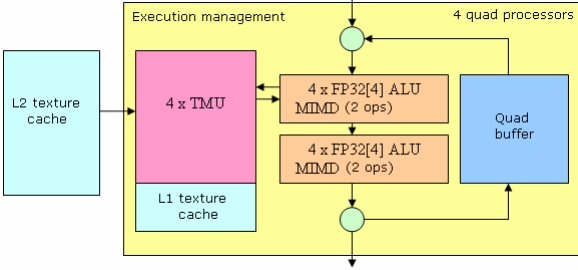
Processeur sommets NV40



Un processeur de sommets NV40 peut exécuter une opération vectorielle (jusqu'à 4 F32), une opération flottante scalaire F32, et faire un accès à la texture par cycle.

Processeurs de fragments NV40

Terminaison précoce du mini tampon z et vérifications du tampon z : les ensembles de 4 pixels (quads) sont passés aux unités de fragments

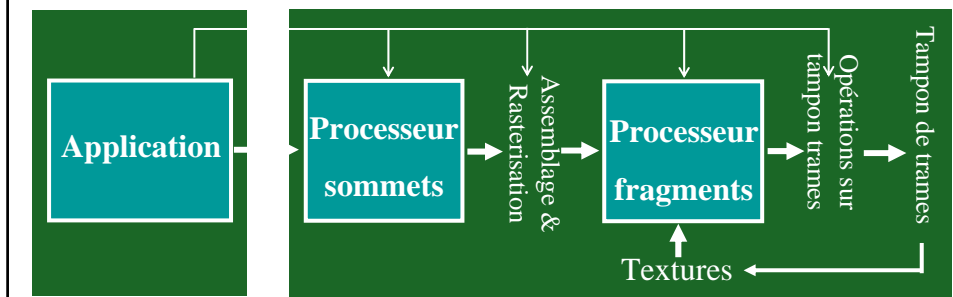


Les différences GPU - CPU

Le GPU est un processeur de flots

Plusieurs unités de traitement programmables

Connectés par des flots de données



Les différences GPU / CPU

Optimisé pour arithmétique vectorielle (4 x 32)

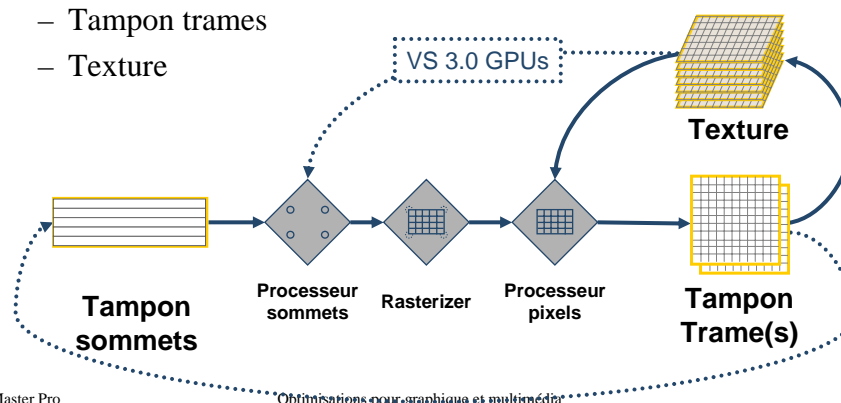
- Utile pour le graphique
- Manière simple d'obtenir un bon rapport performance/coût
- SIMD/MIMD

Modèle mémoire / CPU

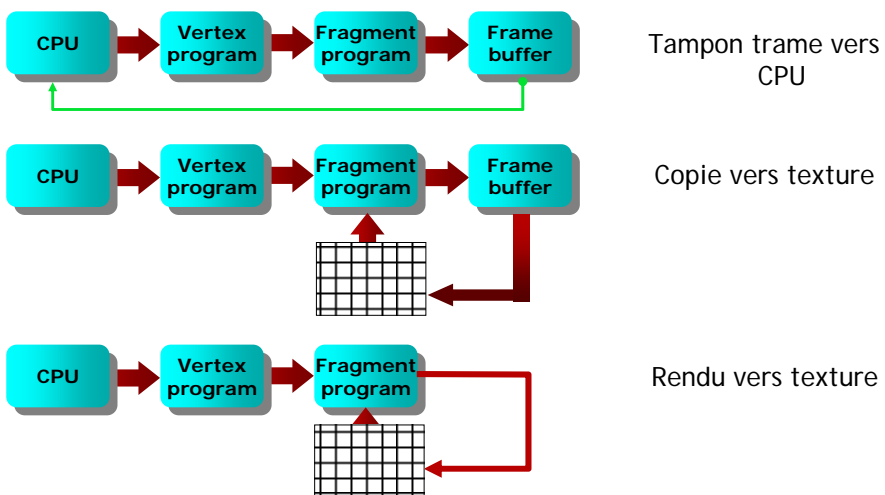
- **Modèle accès mémoire plus restreint**
 - Allocation/libération mémoire uniquement avant calculs
 - Accès mémoire limités durant les calculs (noyaux)
 - Registres
 - Lecture/écriture
 - Pas de mémoire locale
 - Mémoire globale
 - Lectures seulement durant calculs
 - Ecritures seulement à la fin des calculs (adresses pré-calculées)
 - Pas d'accès disque

Modèle mémoire GPU

- **Stockage des données GPU ?**
 - Tampon sommets
 - Tampon trames
 - Texture



Les différents transferts mémoire

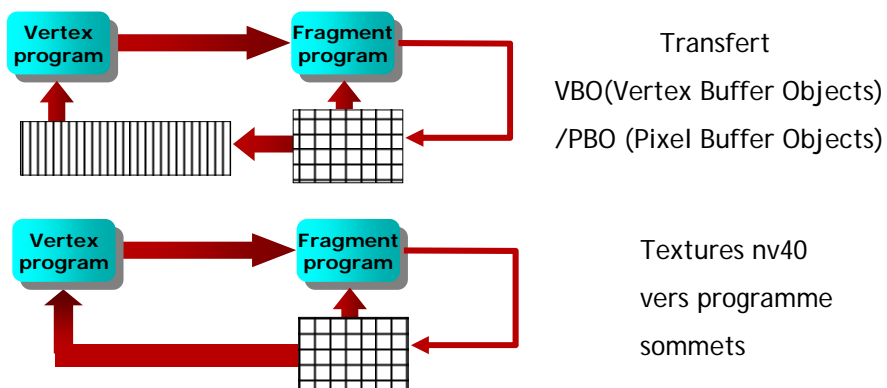


Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

37

Les différents transferts mémoire

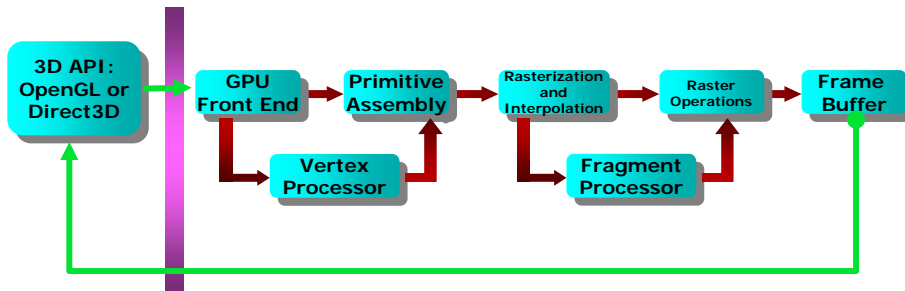


Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

38

Transferts données : Récupération par le CPU



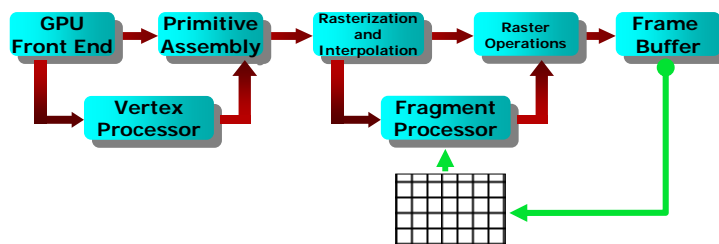
- Readbacks transfer data from the frame buffer to the CPU.
- ⊙ They are very general (any buffer can be transferred)
- ⊙ Partial buffers can be transferred
- ⊙ They are slow: reverse data transfer across PCI/AGP bus is very slow (PCIe is expected to be a lot better)
- ⊙ Data mismatch: readbacks return image data, but the CPU expects vertex data (or has to load image into texture)

Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

39

Transferts données : Copies dans textures



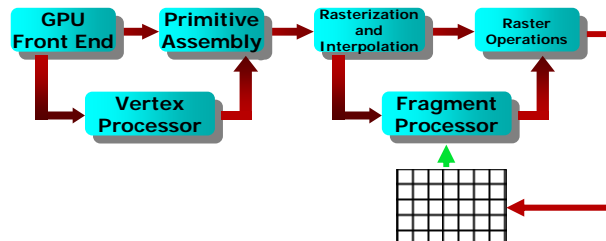
- Copy-to-texture transfers data from frame buffer to texture.
- ⊙ Transfer does not cross GPU-CPU boundary.
- ⊙ Partial buffers can be transferred
- ⊙ Not very flexible: depth and stencil buffers cannot be transferred in this way, and copy to texture is still somewhat slow.
- ⊙ Loss of precision in the copy.

Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

40

Transferts données : Rendu vers texture



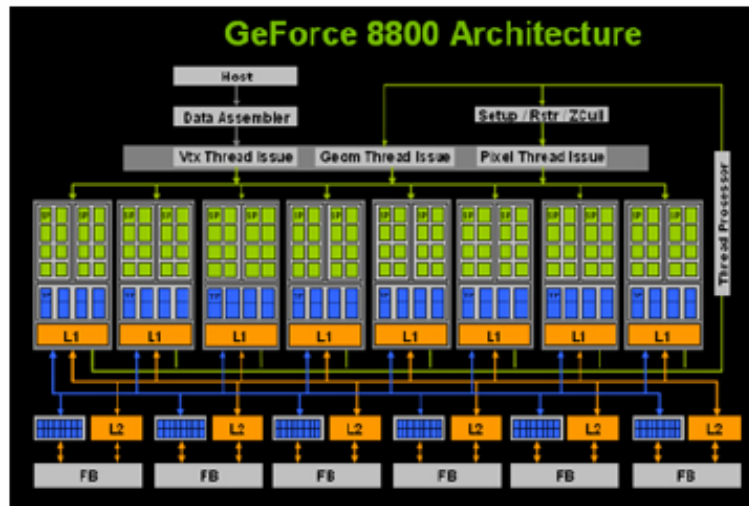
- Render-to-texture renders **directly** into a texture.
- ☺ Transfer does not cross GPU-CPU boundary.
- ☺ Fastest way to transfer data to fragment processor
- ⊗ Only works with depth and color buffers (not stencil).

Render-to-texture is the best method for reading data back after a computation.

Problèmes dans la conception d'architectures pour le graphique

- Rapports entre unités traitement sommets/fragments
 - Typiquement plus de “fragment shaders” que de “vertex shaders”
- Calcul sur des vecteurs ou des scalaires
 - Vecteurs de 4 scalaires
 - Vecteurs bien adaptés pour représenter les couleurs et les normales
 - Les scalaires gaspillent de la place lorsqu'ils sont passés comme des vecteurs
- Branchements et boucles
 - Très inefficaces dans les shaders
 - Il y a généralement une limite sur le nombre possible d'itérations
 - Pour l'efficacité, des programmes shader différents doivent être créés, chacun avec un nombre particulier d'itérations de boucle.
- Z-culling précoce
 - Les “fragment shaders peuvent être coûteux. On aimerait supprimer autant de fragments que possible avant de les “passer” au “fragment shader”
- En général, le matériel graphique est optimisé pour la sortie “tampon de trames”, pas pour réinjecter des données dans le pipeline.
- Il est optimisé pour des données structurées (lecture de textures), pas pour répartir des données (ne peut écrire dans n'importe quelle case du tampon de trames)

GeForce 8800



M
2006-07

D. Etiemble

43

Architecture GeForce 8800

- Architecture “shader” unifiée : pas d’unités séparées pour le “vertex shader” et le “fragment” shader
- 8 coeurs “shader” ayant chacun
 - 16 processeurs de flots (au total 128 processeurs de flot dans le GPU)
 - 4 unités d’adressage de texture
 - 8 unités de filtrage des texture
 - 1 cache L1

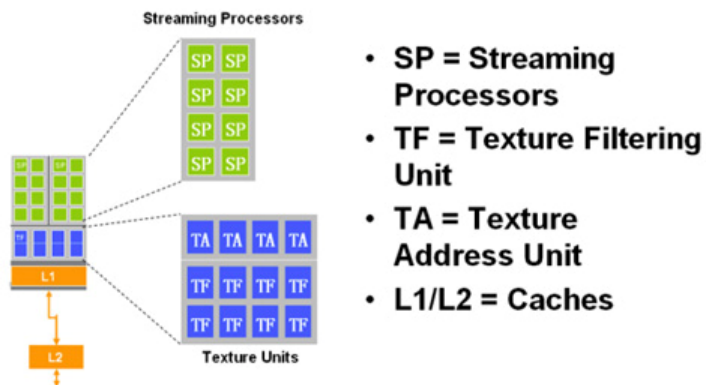
Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

44

8800 « Shader core »

Streaming Processors, Texture Units, and On-chip Caches



Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

45

Architecture unifiée (1)

Why unify?

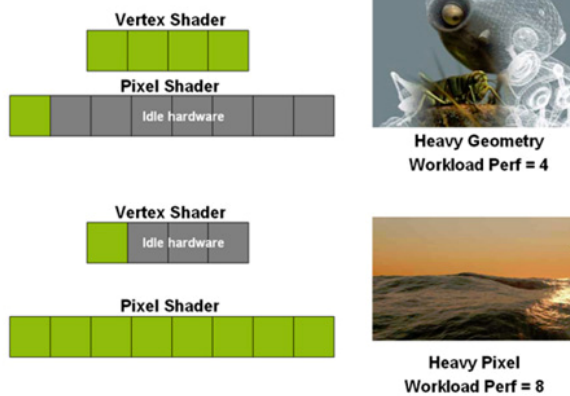


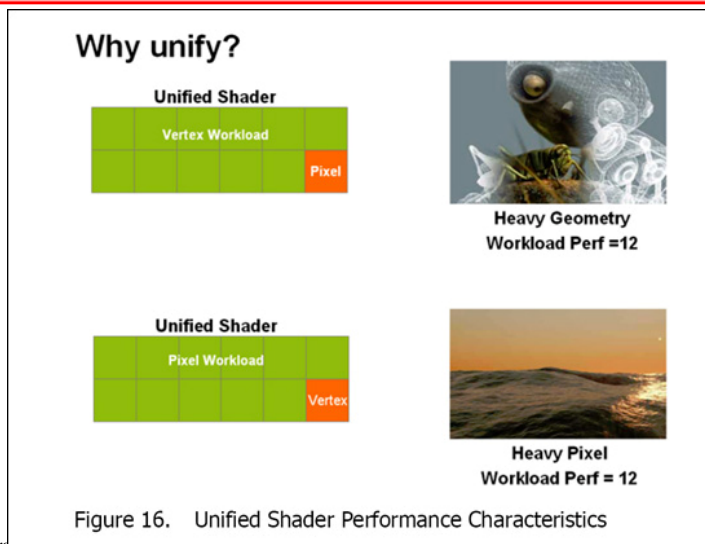
Figure 15. Fixed Shader Performance Characteristics

Master Pro
2006-07

Optimisations pour graphique et multimédia
D. Etiemble

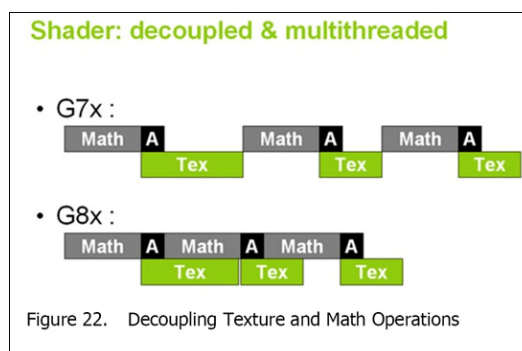
46

Architecture unifiée (2)



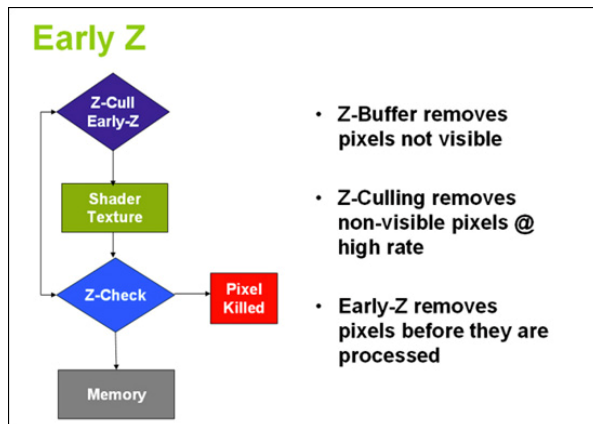
Architecture GeForce 8800

- Flots de données scalaires
 - Dans les processeurs de flot, les vecteurs sont convertis en scalaires
- Plus de parallélisme – moins de temps “inoccupé”
 - Opérations mathématiques s’exécutent en même temps que les accès aux textures



Architecture GeForce 8800

- Z-culling le plus tôt possible
 - Supprime des fragments avant qu'ils n'entrent dans le "fragment shader"

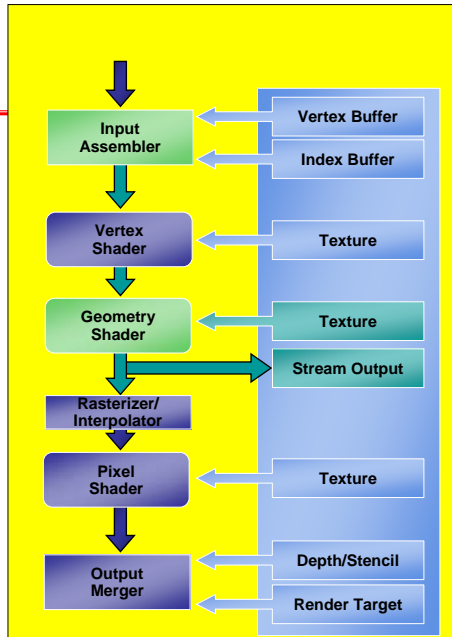
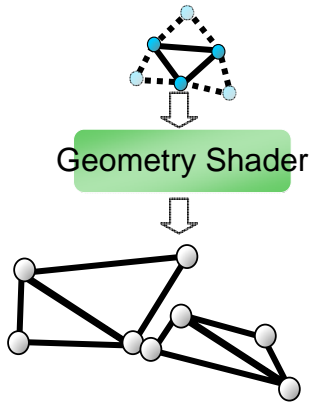


- Z-Buffer removes pixels not visible
- Z-Culling removes non-visible pixels @ high rate
- Early-Z removes pixels before they are processed

Architecture GeForce 8800

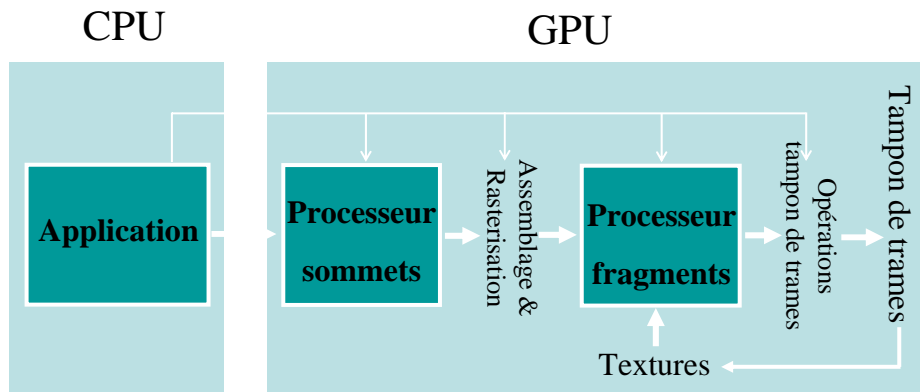
- "Geometry shader"
 - Traite des primitives complètes, pas seulement des sommets
 - Primitives d'entrée et de sortie
 - Des exemples de primitives sont les listes de point, les listes de triangles

Pipeline DirectX 10



Modèle de programmation des GPU

NVIDIA - 2002



Types de données

- Flottants 32 bits à travers le pipeline
 - Tampon de trames
 - Textures
 - Processeur fragments
 - Processeur de sommet
 - Interpolations
- Le processeur fragment dispose aussi de
 - Flottants 16 bits “half”
 - Virgule fixe 12 bits
- Les textures et le tampon de trames disposent aussi de
 - Grande gamme de formats fixes
 - Par exemple, le format classique 8 bits par pixel
 - Ces formats utilisent moins de bande passante que les flottants 32 bits

Fonctionnalités du processeur de sommets

- SIMD flottant vectoriel (4 éléments)
- Contrôle de flots dépendant des données
 - Instructions de branchement conditionnelles
 - Appel de fonctions, avec jusqu’à quatre appels imbriqués
 - Table de sauts (pour les instructions multi-voies)
- Codes conditions
- Nouvelles instructions arithmétiques (COS)
- 256 instructions par programme (beaucoup plus sans branchement)
- 16 registres vectoriels à 4 éléments temporaires
- 256 registres de paramètres
- 2 registres d’adresses (vecteurs de 4 éléments)
- 6 sorties de distance de clipping.

Le processeur de fragments (1)

- Jeu d'instructions
 - Instructions générales et orthogonales
 - Même syntaxe que le processeur de sommets
MUL R0, R1.xyz, R2.yxw;
 - Ensemble complet d'instructions arithmétiques
 - RCP, RSQ, COS, EXP, ...
- Instructions sur les textures
 - Les lectures de texture correspondent à une instruction (TEX, TXP, or TXD)
 - Permettent de calculer les coordonnées sur les textures, avec un niveau quelconque d'imbrication
 - Permettent plusieurs utilisations d'une unité de texture.
- Autres possibilités
 - Accès en lecture à une position dans la fenêtre
 - Accès en lecture écriture au Z fragment
 - Instructions de dérivation intégrées
 - Dérivées partielles par rapport au coordonnées x ou y de l'écran
 - Utile pour l'anti-aliasing
 - Instruction conditionnelle de suppression d'un fragment

Le processeur fragments (2)

- Possibilités
 - 1024 instructions
 - 512 paramètres uniformes ou constantes
 - Chaque constante compte comme une instruction
 - 16 unités de textures
 - Réutilisables à volonté
 - Entrées : 4 x 8 flottants 32 bits
 - Sortie "couleur" tampon de trames de 128 bits
 - 4 flottants 32 bits, 8 half 16 bits, etc.
- Limitations
 - Pas de branchement, mais possibilité d'utiliser les codes condition
 - Pas de lectures indexées à partir des registres
 - Utilisation de lecture de textures
 - Pas d'écriture mémoire

De l'assembleur à Cg

Assembleur

```
...  
FRC R2.y, C11.w;  
ADD R3.x, C11.w, -R2.y;  
MOV H4.y, R2.y;  
ADD H4.x, -H4.y, C4.w;  
MUL R3.xy, R3.xyww, C11.xyww;  
ADD R3.xy, R3.xyww, C11.z;  
TEX H5, R3, TEX2, 2D;  
ADD R3.x, R3.x, C11.x;  
TEX H6, R3, TEX2, 2D;  
...
```

Cg

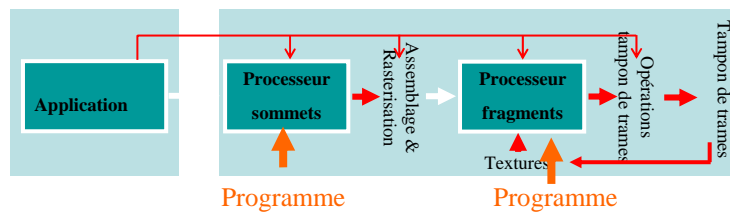
```
...  
L2weight = timeval - floor(timeval);  
L1weight = 1.0 - L2weight;  
ocoord1 = floor(timeval)/64.0 + 1.0/128.0;  
ocoord2 = ocoord1 + 1.0/64.0;  
L1offset = f2tex2D(tex2, float2(ocoord1, 1.0/128.0));  
L2offset = f2tex2D(tex2, float2(ocoord2, 1.0/128.0));  
...
```

Stratégie de conception de Cg

- Partir de C (et un peu de C++)
 - Minimise le nombre of décisions
 - Permet de prendre en compte les erreurs “connues” au lieu des “inconnues”
- Permettre de réduire le langage
- Ajouter les caractéristiques voulues pour les processeurs graphiques
 - Pour prendre en compte le modèle de programmation des GPU
 - Pour la haute performance
- Faire une synthèse des différents aspects

Différences CPU et GPU

1. Le processeur graphique est un processeur de flots
 - Plusieurs unités de traitement programmables
 - Connectées par les flots de données.



- Deux types de programmes distincts
 - Programme sommets et programme fragments
- Deux types d'entrées
 - Entrées variables (flot de données)
 - Entrées uniformes (état graphique)

Différences CPU et GPU

2. Plus de variantes dans les possibilités de base
 - La plupart des processeurs n'ont toujours pas de branchements
 - Les processeurs de sommets n'ont pas de support pour le mappage des textures
 - Certains processeurs ont des types de données supplémentaires

- **Le compilateur ne peut cacher les différences**
- **Le plus petit commun dénominateur est trop restrictif**
- **Profils de langage différents**
(liste des possibilités et types de données)
- **Convergence des profils espérée dans le futur**

Différence GPU et CPU

- Optimisés pour l'arithmétique sur des vecteurs de 4 éléments
 - Utile pour le graphique –couleurs, vecteurs, coordonnées
 - Bonne manière d'obtenir la performance/coût

- La philosophie C expose les types de données au matériel**
- Cg a des types de données et des opérations vectorielles : float2, float3, float4**
- Rend évident la manière d'obtenir une performance élevée**
- Cg a aussi des types de données matriciels : float3x3, float3x4, float4x4**

Quelques opérations vectorielles

```
//  
// Clamp components of 3-vector to [minval,maxval] range  
//  
float3 clamp(float3 a, float minval, float maxval) {  
    a = (a < minval.xxx) ? Minval.xxx : a;  
    a = (a > maxval.xxx) ? Maxval.xxx : a;  
    return a;  
}
```

Comparaisons
vectorielles élément par
élément et résultat
vectoriel.

? : par élément
des vecteurs

Duplication et/ou réarrangement
des éléments

Opérations vectorielles

- Swizzle – dupliquer/réarranger les éléments

```
a = b.xxyy;
```

- Masque d'écriture – écriture partielle

```
a.w = 1.0;
```

- Construction de vecteurs

```
- a = float4(1.0, 0.0, 0.0, 1.0);
```

Cg a des tableaux

- Déclarés comme en C
- Les tableaux sont distincts des types vectoriels
`float4 != float[4]`
- Les profils peuvent restreindre l'utilisation des tableaux

```
vout MyVertexProgram(float3 lightcolor[10],  
                    ...) {  
    ...  
}
```

Différences CPU et GPU

4. Pas de pointeurs
 - Les tableaux sont les types de données privilégiés de Cg
5. Pas de type de données entiers
 - Cg ajoute le type "bool" pour les opérations entières

LES TYPES DE DONNÉES CG

- Tous profils:
 - float
 - bool
- Tous profils avec recherche de texture :
 - sampler1D, sampler2D, sampler3D, samplerCUBE
- Profil du programme fragment NV:
 - half -- flottant demi-précision
 - fixed -- virgule fixe [-2,2)

Fonctions intégrées Cg

- Mappage de textures (dans les profils fragment)
- Math
 - Produit scalaire
 - Multiplication de matrices
 - Sin/cos/etc.
 - Normalisation
- Diverses
 - Dérivées partielles (quand elles sont supportées)

Exemple Cg (1)

```
// In:
// eye_space position = TEX7
// eye space T = (TEX4.x, TEX5.x, TEX6.x) denormalized
// eye space B = (TEX4.y, TEX5.y, TEX6.y) denormalized
// eye space N = (TEX4.z, TEX5.z, TEX6.z) denormalized

fragout frag program main(vf30 In) {

    float m = 30; // power
    float3 hiCol = float3( 1.0, 0.1, 0.1 ); // lit color
    float3 lowCol = float3( 0.3, 0.0, 0.0 ); // dark color
    float3 specCol = float3( 1.0, 1.0, 1.0 ); // specular color

    // Get eye-space eye vector.
    float3 e = normalize( -In.TEX7.xyz );

    // Get eye-space normal vector.
    float3 n = normalize( float3(In.TEX4.z, In.TEX5.z, In.TEX6.z ) );
```

Exemple Cg (2)

```
float edgeMask = (dot(e, n) > 0.4) ? 1 : 0;
float3 lpos = float3(3,3,3);
float3 l = normalize(lpos - In.TEX7.xyz);
float3 h = normalize(l + e);

float specMask = (pow(dot(h, n), m) > 0.5) ? 1 : 0;

float hiMask = (dot(l, n) > 0.4) ? 1 : 0;
float3 ocoll = edgeMask *
    (lerp(lowCol, hiCol, hiMask) + (specMask * specCol));

fragout O;
O.COL = float4(ocoll.x, ocoll.y, ocoll.z, 1);
return O;
}
```