



## M4104c - Architecture pour l'Imagerie numérique

Romain Raffin

IUT Aix-Marseille, dept. Informatique  
[romain.raffin@univ-amu.fr](mailto:romain.raffin@univ-amu.fr)

v2016

CM/TD/TP dispos sur : [www.iut-arles.up.univ-mrs.fr/raffin.r](http://www.iut-arles.up.univ-mrs.fr/raffin.r) rubrique « Info2 -  
Architecture pour l'IN »

# Sommaire

## ① Organisation

## ② Introduction

## ③ Matériel

Stockage

Carte Video

Signal

Écrans

Numérisation

## ④ Pipeline graphique

Programmation du pipeline graphique

Comment cela fonctionne ?

## ⑤ Description des géométries en OpenGL 2.0

Avant c'était bien statique ...

... mais ça c'était avant

Les VBO

Les FBO

## ⑥ Autres matériels

## ⑦ Culture

# Planning

- 4 séances de CM
- 2 séances de TD
- 4 séances de TP
- et 3 TPA...

# Planning

- 4 séances de CM (et un contrôle d'une heure)
- 2 séances de TD
- 4 séances de TP
- et 3 TPA... (et un projet-TP à rendre)

## Pourquoi une architecture spécifique ?

Besoins de :

- bande passante (flux de données) ;
- puissance de calcul ;
- capacités de stockage.

## Exemple de traitement d'une image fixe

Hypothèses : traitement d'une image numérique,  $1\mu s$  par pixel (env. 1 000 instructions à 1 GHz<sup>1</sup>).

résolution	nb. couleurs	nb. pixels	codage	taille mémoire	temps de traitement
640 × 480	256	307 200	8 bits	300 ko	0,31 s
1024 × 768	16M	768 432	24 bits	2,25 Mo	0,78 s
10000 × 10000	16M	$10000^6$	24 bits	286,1 Mo	100 s

---

$$1. \quad 1\text{GHz} = \frac{1}{10^9} \text{ impulsions par secondes}$$

## Exemple de traitement d'une image animée

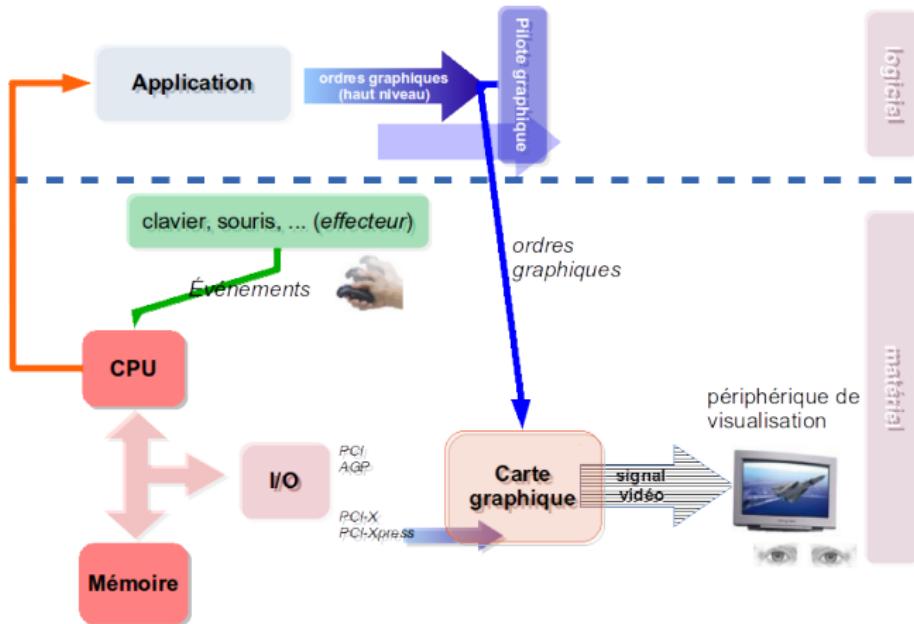
Hypothèses : traitement d'une image numérique,  $1\mu s$  par pixel (env. 1000 instructions à  $1\text{ GHz}^2$ ), cadence de 24 im/s.

résolution	nb. couleurs	nb. pixels/s	codage	taille mémoire	perte due au traitement
$640 \times 480$	256	7 372 800	8 bits	7 Mo/s	7,4 im/s
$1024 \times 768$	16M	18 874 368	24 bits	54 Mo/s	18,72 im/s

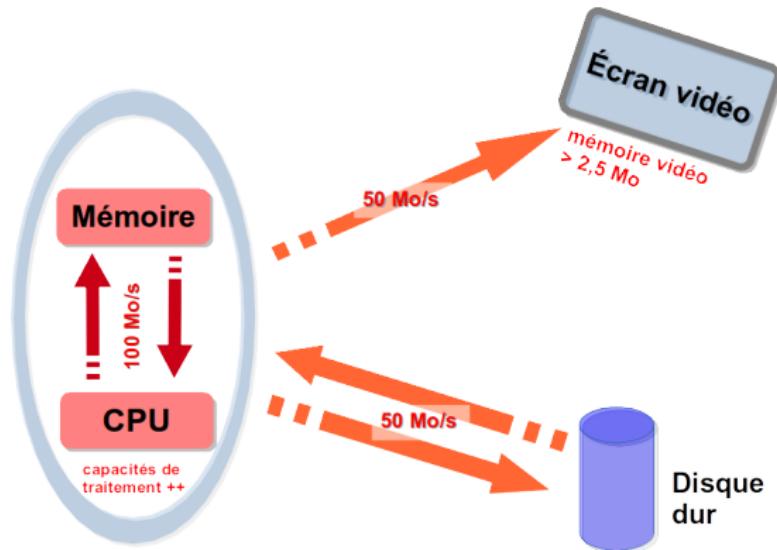
---

$$2. 1\text{GHz} = \frac{1}{10^9} \text{ impulsions par secondes}$$

# Traitement de l'Image



## Contraintes matérielles



# Plan

① Organisation

② Introduction

③ **Matériel**

Stockage

Carte Vidéo

Signal

Écrans

Numérisation

④ Pipeline graphique

Programmation du pipeline graphique

Comment cela fonctionne ?

⑤ Description des géométries en OpenGL 2.0

Avant c'était bien statique ...

... mais ça c'était avant

Les VBO

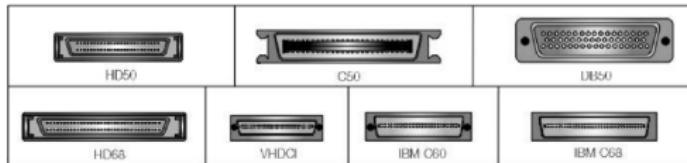
Les FBO

⑥ Autres matériels

⑦ Culture

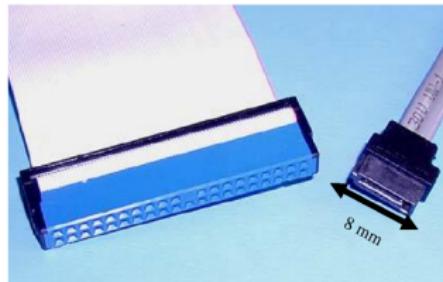
## SCSI (*Small Computer System Interface*)

- transmission en parallèle ou en différentiel, câble 80 broches ;
- avantages : plus rapide, 7 à 15 périphériques par carte contrôleur (*daisy chain*), existe en externe ;
- inconvénient : cher ;
- standards : SCSI 8 bits 10 Mo/s, Wide 16 bits 20 Mo/s, Ultra2Wide 80 Mo/s, Ultra160, Ultra320, Ultra640 ;
- dédié Image : AV, garanti un taux de transfert constant, une disponibilité (cache, correction d'erreurs, transferts de gros blocs).



## IDE (*Integrated Drive Electronics*)

- interface série (sATA) ou parallèle (pATA) ;
- moins chère mais moins rapide, ne permet le branchement que de 4 disques ;
- débit : 133 Mo/s (ultraDMA 133, 80 fils, CRC) ;
- évolution : Serial Ata, transmission en série, plus rapide (150 Mo/s puis 300 et 600 Mo/s), câble plus long (1 m), compatible avec le parallel-ATA, faible consommation, 7 câbles au lieu des 80 ;
- limité par les Bus PCI (33 à 66 MHz) et la puissance du processeur.



# Plan

① Organisation

② Introduction

③ Matériel

Stockage

Carte Vidéo

Signal

Écrans

Numérisation

④ Pipeline graphique

Programmation du pipeline graphique

Comment cela fonctionne ?

⑤ Description des géométries en OpenGL 2.0

Avant c'était bien statique ...

... mais ça c'était avant

Les VBO

Les FBO

⑥ Autres matériels

⑦ Culture

## Bus vidéo

4 étapes sont indispensables à la fabrication d'une image sur un écran d'ordinateur :

- ① transfert Bus Mémoire vers le Chipset Vidéo où elle est calculée (numérique) ;
- ② depuis le chipset vidéo jusqu'à la mémoire vidéo (stockage) ;
- ③ depuis la mémoire vidéo jusqu'au convertisseur digital-analogique (=RAM DAC), pour convertir les données numériques en signal pour l'écran ;
- ④ depuis le Digital Analog Converter jusqu'au moniteur (en analogique ou non).

## Bus et interfaces

les goulets d'étranglement dépendent du type de bus et de sa vitesse, de la carte mère et du chipset CM, et de la carte vidéo. Le bus le plus rapide est le PCI (par rapport à AGP, VL-bus, ISA, EISA et NuBus (Mac)). Le PCI est limitée à 66 MHz<sup>3</sup> ce qui ne suffit plus :

- AGP (Advanced Graphics Port), canal mémoire direct (DMA), 266 Mo/s - 2,11 Go/s, 1 seul périphérique,
- PCI Express, 250 Mo/s - 8 Go/s<sup>4</sup> - 16 Go/s<sup>5</sup>, PCI série, plusieurs périphériques qui peuvent communiquer sans passer par le CPU (comme PCI)



- 
3. sur un bus à 64 bit (528 Mo/s), mais existent également les PCI-X : 64 bits / 33 MHz / 1066 Mo/s et PCI-X 2.0 : 64 bits / 266 MHz/ 2133 Mo/s
4. PCI-Express 2.x, 16 lignes, 500 MHz
5. PCIe 3.x, 16 lignes, 1 GHz

# Évolution de cartes vidéo

Spécifications des principales cartes testées					
GPU	7600 GT	X1800 GTO	X1800 XL	6800 GT	
Fréquence GPU	560 MHz	500 MHz	500 MHz	350 MHz	
Fréquence mémoire	700 MHz	500 MHz	500 MHz	500 MHz	
Largeur bus mémoire	128 bits	2 x 256 bits	2 x 256 bits	256 bits	
Type de mémoire	GDDR3	GDDR3	GDDR3	GDDR3	
Quantité de mémoire	256 Mo	256 Mo	256 Mo	256 Mo	
Nombre de Pixel Pipelines	12	12	16	16	
Nombre d'unités de Texture Sampling	1	1	1	1	
Nombre de processeurs de vertex	5	8	8	6	
Fillrate théorique	6720 MPixels	6000 MPixels	8000 MPixels	5600 MPixels	
Vertex rate théorique	700 MTris	1000 MTris	1000 Mtris	525 MTris	
Bande passante mémoire	22,4 Go/s	32 Go/s	32 Go/s	32 Go/s	
Ratio fillrate / bande passante mémoire	300	187	250	175	
Nombre de transistors	? millions	321 millions	321 millions	222 millions	
Process	0.09µ	0.09µ	0.09µ	0.13µ	
Surface du die	120 mm <sup>2</sup>	288 mm <sup>2</sup>	288 mm <sup>2</sup>	287 mm <sup>2</sup>	
Génération	2005	2005	2005	2004	
Shader Model supporté	3.0	3.0	3.0	3.0	
Prix	200 \$	250 \$	300 \$ ?	220 €	

(a) 2006

	GTX Titan Black	GTX Titan Z	R9 295X2	R9 290X
GPU	GK110	2x GK110	2x Hawaii	Hawaii
Procédé de fabrication	28 nm	28 nm	28 nm	28 nm
Unités de calcul	2880	5760	5632	2816
Unités de texturing	240	480	352	176
ROPs	48	96	128	64
Fréq. GPU de base (MHz)	889	705	300	727
Fréq. GPU Turbo/Max (MHz)	≥ 980	≥ 876	1018	1000
Débit de triangles (Gtri/s)	≥ 7.35 (4.90*)	≥ 13.14 (8.76*)	8.14	4.00
Débit de pixels (Gpixels/s)	≥ 39	≥ 70	130	64
Débit de calcul FMA SP (Tflops)	≥ 5.64	≥ 10.09	11.47	5.63
Débit de calcul FMA DP (Tflops)	≥ 1.88	≥ 3.36	1.43	0.70
Débit de filtrage (Gtexels/s)	≥ 235	≥ 420	358	176
Fréq. mémoire (MHz)	1750	1750	1250	1250
Taille mémoire (Mo)	6144	6144	4096	4096
Bus mémoire (bits)	384	768	1024	512
BP mémoire (Go/s)	313	626	596	298
Direct3D	11.0	11.0	11.1	11.1
TDP	250W	375W	~550W	~285W
Prix	900 €	2 800 €	1 300 €	470 €

(b) 2014

## Consommation



(c) Jeu

(d) Idle

# Consommation

Si on veut comparer avec les CPU :



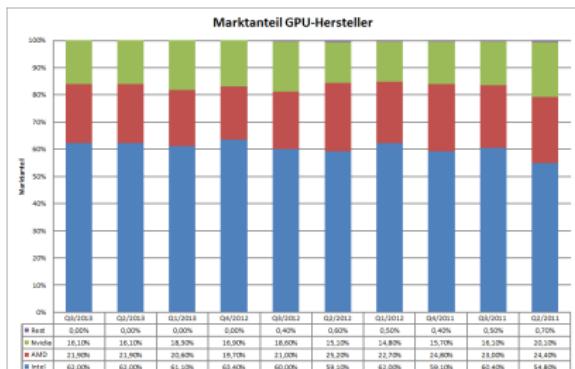
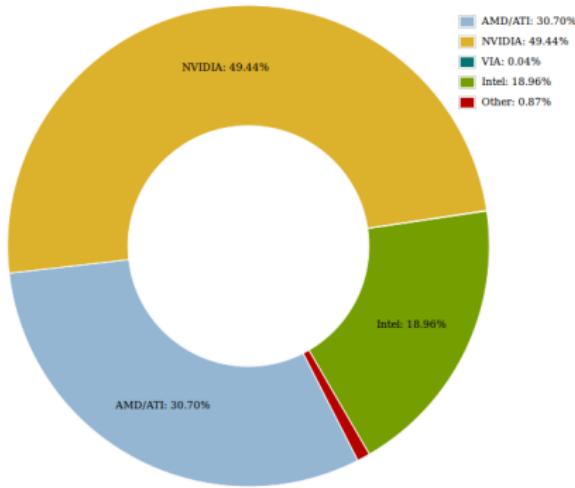
(e) Load

# Construction

Si on veut comparer avec les CPU :

CPU Specification Comparison						
	CPU	Node	Cores	GPU	Transistor Count (Schematic)	Die Size
<b>Server CPUs</b>						
Intel	<b>Haswell-EP 14-18C</b>	22nm	14-18	N/A	5.69B	662mm <sup>2</sup>
Intel	<b>Haswell-EP 10C-12C</b>	22nm	6-12	N/A	3.84B	492mm <sup>2</sup>
Intel	<b>Haswell-EP 6C-8C</b>	22nm	4-8	N/A	2.6B	354mm <sup>2</sup>
Intel	<b>Ivy Bridge-EP 12C-15C</b>	22nm	10-15	N/A	4.31B	541mm <sup>2</sup>
Intel	<b>Ivy Bridge-EP 10C</b>	22nm	6-10	N/A	2.89B	341mm <sup>2</sup>
<b>Consumer CPUs</b>						
Intel	<b>Haswell-E 8C</b>	22nm	8	N/A	2.6B	356mm <sup>2</sup>
Intel	<b>Haswell GT2 4C</b>	22nm	4	GT2	1.4B	177mm <sup>2</sup>
Intel	<b>Haswell ULT GT3 2C</b>	22nm	2	GT3	1.3B	181mm <sup>2</sup>
Intel	<b>Ivy Bridge-E 6C</b>	22nm	6	N/A	1.86B	257mm <sup>2</sup>
Intel	<b>Ivy Bridge 4C</b>	22nm	4	GT2	1.2B	160mm <sup>2</sup>
Intel	<b>Sandy Bridge-E 6C</b>	32nm	6	N/A	2.27B	435mm <sup>2</sup>
Intel	<b>Sandy Bridge 4C</b>	32nm	4	GT2	995M	216mm <sup>2</sup>
Intel	<b>Lynnfield 4C</b>	45nm	4	N/A	774M	296mm <sup>2</sup>
AMD	<b>Trinity 4C</b>	32nm	4	7660D	1.303B	246mm <sup>2</sup>
AMD	<b>Vishera 8C</b>	32nm	8	N/A	1.2B	315mm <sup>2</sup>

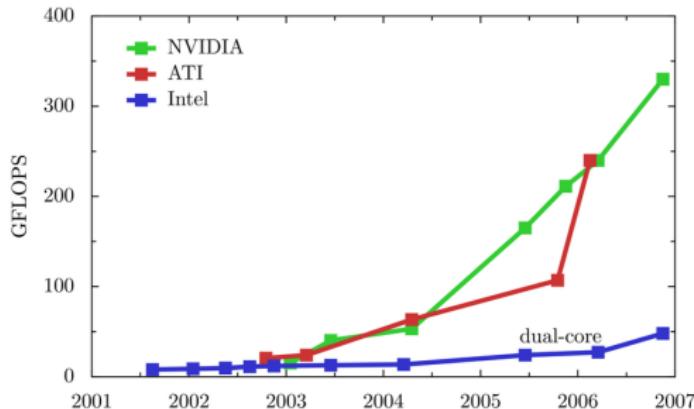
## Répartition du marché GPU



Sources : <http://www.videocardbenchmark.net/30dayshare.html> wikipedia.fr

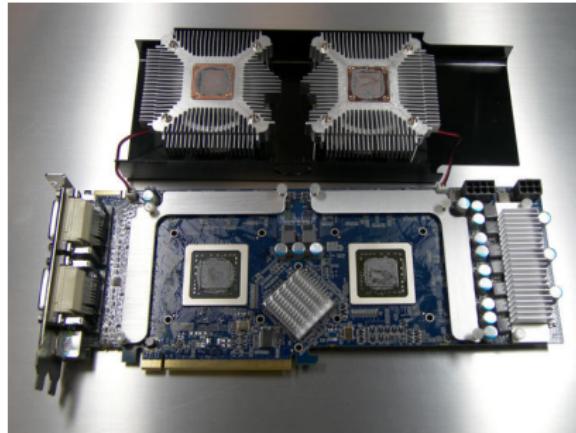
## Tendances

Sur le long terme les GPU gagnent :



Sources : SuperComputing 2006 <http://www.gpgpu.org/>

## Tendances : multi-GPU



Sources : <http://www.pcgameshardware.com>

# Plan

- ① Organisation
- ② Introduction
- ③ **Matériel**
  - Stockage
  - Carte Vidéo
  - Signal**
  - Écrans
  - Numérisation
- ④ Pipeline graphique
  - Programmation du pipeline graphique
  - Comment cela fonctionne ?
- ⑤ Description des géométries en OpenGL 2.0
  - Avant c'était bien statique ...
  - ... mais ça c'était avant
  - Les VBO
  - Les FBO
- ⑥ Autres matériels
- ⑦ Culture

## Types de signaux

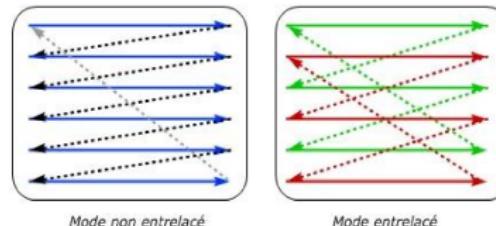
Signal analogique :

- composante RGB,
- YC (luminance-couleur),
- composite.

Signal numérique :

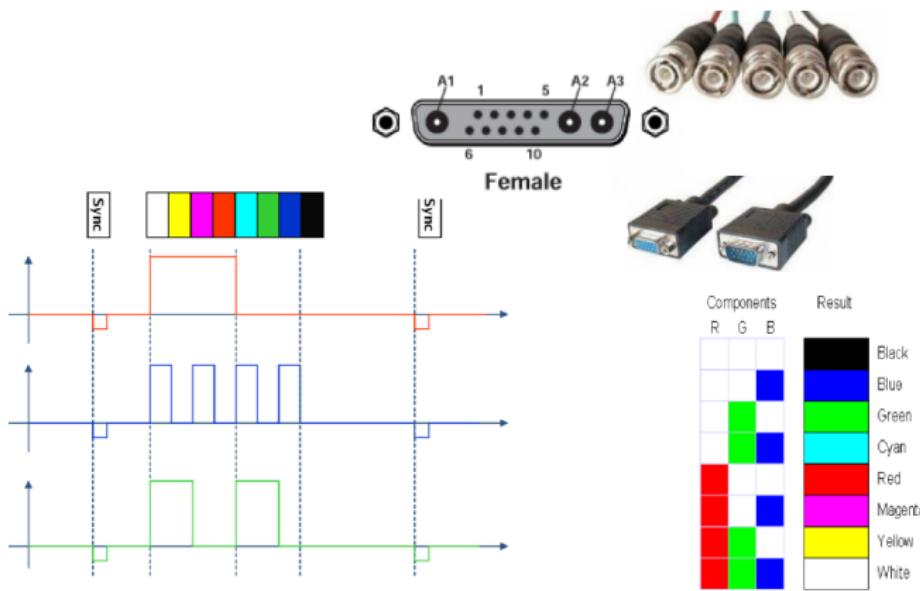
- DVI (Digital Visual Interface), bus parallèle/composantes
- SDI (Serial Digital Interface), bus série, YCrCb échantillonnés

Diffusion : progressif ou entrelacé



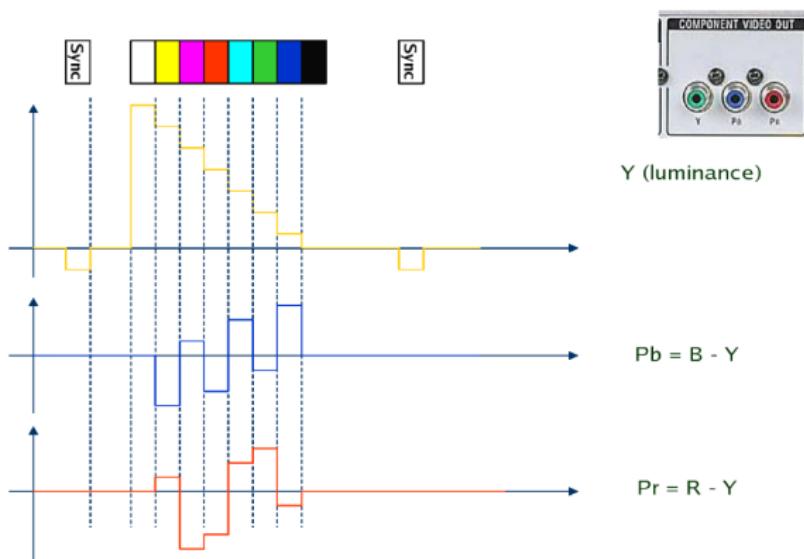
## Signal RGB

Les couleurs et les synchronisations horizontale et verticale sont transportées sur 3, 4 ou 5 conducteurs coaxiaux séparés.



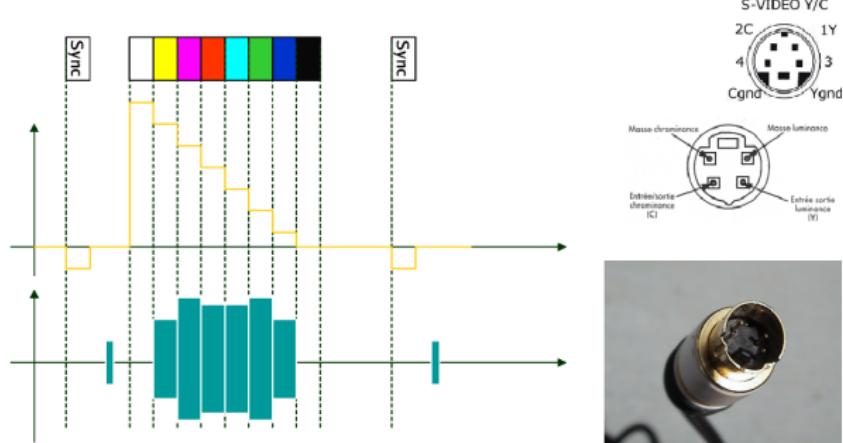
## Signal YPrPb

On fait la différence entre la luminosité (Y), la teinte de bleu (% Y, Pb), la teinte de rouge (%Y, Pr).



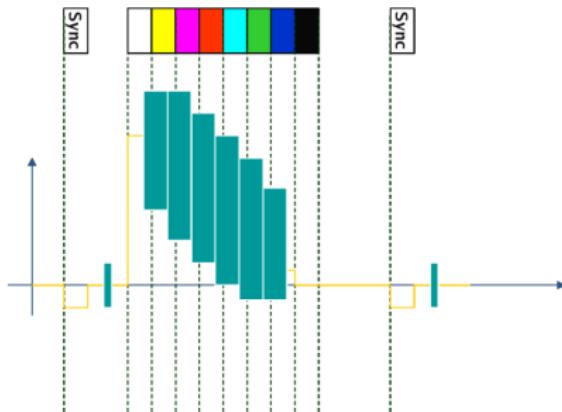
## Signal Y/C (SVideo)

La luminance et la chrominance sont transportées sur 2 conducteurs coaxiaux séparés (Pr et Pb mélangés). La qualité se dégrade...

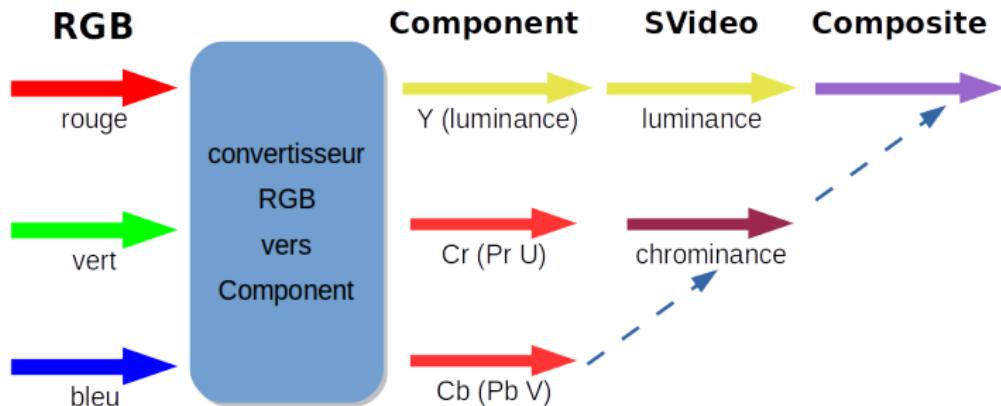


## Signal composite

La luminance et la chrominance sont transportées sur un seul conducteur coaxial. La qualité se dégrade encore...



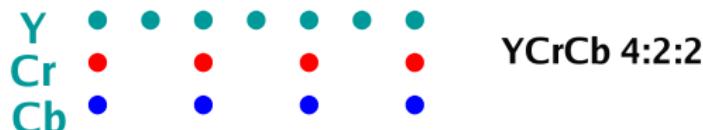
## Conversion de signaux



## Signal numérique SDI

Le signal vidéo YPrPb est échantillonné sur 10 bits :

- SDI PAL 13.5 Mhz SMPTE 259M ( $720 \times 576$ )
- SDI HDTV 74.25 MHz SMPTE 274M ( $1920 \times 1080$ )

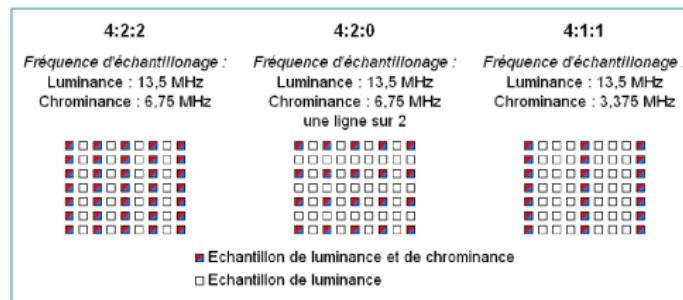


- transmis en série sur un câble coaxial
- utilisé en vidéo professionnelle (régie TV, production de films)

# SDI

La dénomination 4 :2 :2 indique avec quel multiple d'une fréquence unitaire seront échantillonnés les signaux de luminance et de chrominance. Cette fréquence unitaire commune aux systèmes 625 et 525 lignes est de 3,375 MHz. Le multiplexage des signaux Y, Cr et Cb aux fréquences d'échantillonnage de 13.5 MHz, 6.75 MHz et 6.75 MHz donne une fréquence d'horloge de 27 MHz pour chacun des 8 ou 10 bits.

Le taux de transfert total est donc de 270 Mbits/s en 10 bits ( $27 \times 10$ ) et de 216 Mbits/s en 8 bits ( $27 \times 8$ ).



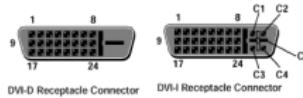
# DVI

Les données RGB sont transmises sur 3 ou 6 paires torsadées 2 liens de 3 canaux activables séparément 165 MPixels/sec sur 1 lien ou 330 MPixels/s sur 2 liens

- 1600x1200 60Hz UXGA (1 lien)
- 2048x1536 60Hz QXGA (2 liens)

Longueur maxi 10 mètres et trois types de DVI :

- DVI-D (Digital)
- DVI-A (Analogique)
- DVI-I (Integrated Digital/Analog)



Quelques propriétés supplémentaires :

- Codage TMDS (transition minimized differential signaling)
- Extended Display Identification Data (EDID)
- High bandwidth Digital

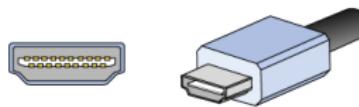
## HDMI

En termes de capacités, l'interface HDMI permet d'obtenir des débits de l'ordre de 5 Gb/s. Elle permet ainsi de transmettre :

- des signaux audio multi-canaux (jusqu'à 8 voies PCM 24 bits/192 kHz) avec une fréquence d'échantillonnage de 32 kHz, 44.1kHz, 48kHz ou 192kHz,
- des signaux vidéo en haute définition (jusqu'à 1920x1080) sur 3 canaux codés sur 24 bits (8 bits par canal).

L'interface HDMI supporte l'ensemble des formats vidéos actuels et propose 3 nouveaux formats afin de tenter une homogénéisation des équipements :

- SDTV : 720x480i en NTSC, 720x576i en PAL,
- EDTV : 640x480p en VGA, 720x480p en NTSC progressif, 720x576p et PAL progressif,
- HDTV : 1280x720p, 1920x1080i, 1980x1080p



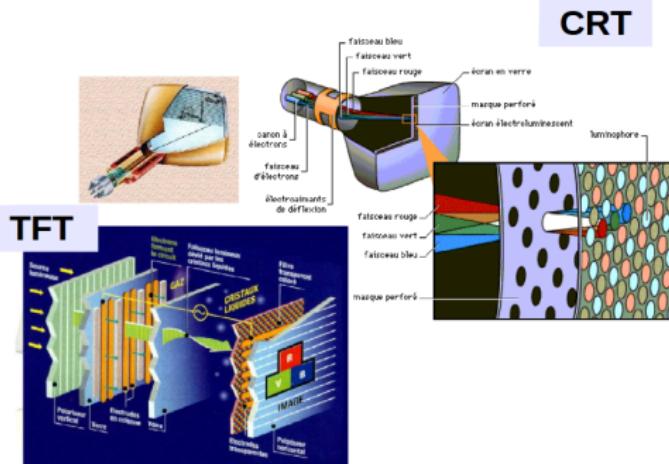
# Plan

- ① Organisation
- ② Introduction
- ③ **Matériel**
  - Stockage
  - Carte Vidéo
  - Signal
  - Écrans**
  - Numérisation
- ④ Pipeline graphique
  - Programmation du pipeline graphique
  - Comment cela fonctionne ?
- ⑤ Description des géométries en OpenGL 2.0
  - Avant c'était bien statique ...
  - ... mais ça c'était avant
  - Les VBO
  - Les FBO
- ⑥ Autres matériels
- ⑦ Culture

## De types divers

- CRT (*Cathode Ray Tube*)
- LCD (*Liquid Crystal Display*)
- matrice passive DSTN (*dual-scan twisted nematic*)
- matrice active TFT (*Thin Film Transistor*)
- plasma
- micro pointes ou à émission de champs

# LCD & TFT

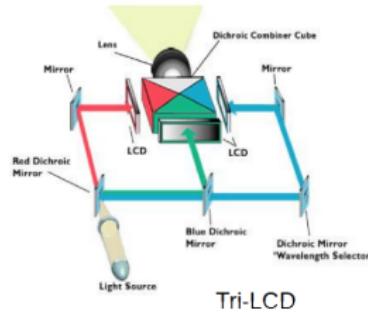


## LCD & TFT

Display Type	Viewing angle	Contrast Ratio	Response Speed	Brightness	Power Consumption	Life
DSTN LCD	49 - 100°	40:1	300ms	70 90	45 watts	60K hours
TFT LCD	> 140°	500:1	8ms	70 90	50 watts	60K hours
CRT	> 190°	300:1	n/a	220 270	180 watts	Years

## Et les projecteurs ?

- Tri-tubes (3 mini tubes CRT)
- LCD (*Liquid Crystal Display*) (passage de la lumière décomposée dans 3 LCD NdG)
- DLP (*Digital Light Processors*) ou DMD (*Digital microMirrors Display*) (activation de la luminosité d'un pixel par rotation d'un micro-miroir+filtre RGB tournant)



# Plan

① Organisation

② Introduction

③ Matériel

Stockage

Carte Vidéo

Signal

Écrans

Numérisation

④ Pipeline graphique

Programmation du pipeline graphique

Comment cela fonctionne ?

⑤ Description des géométries en OpenGL 2.0

Avant c'était bien statique ...

... mais ça c'était avant

Les VBO

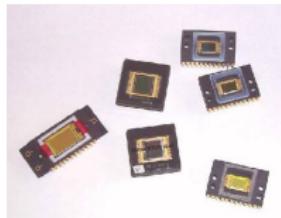
Les FBO

⑥ Autres matériels

⑦ Culture

## Numérisation 2D

- scanner : SCSI, parallèle, Usb. A plat, diapositive, à défilement : résolution  $1200 \times 2400$  dpi, A4, capteur CCD défilant, couleurs 24, 32 voire 48 bits,
- diapositive :  $1800 \times 1800$  dpi,  $19200 \times 19200$  par interpolation, surface de numérisation  $24 \times 36$  mm, à défilement : 300 dpi
- appareil photo : prise de vue par capteur CCD. Stockage avec ou sans compression sur SmartMédia, Compact Flash, Secure Digital, minidisque, mémoire SDRAM, ... Connecteur Usb.  $640 \times 480$  à  $2600 \times 1950$  et +. Sup. à 12Mpixels pour un équivalent à l'argentique (il existe des 100MP)
- caméra : technique idem à appareil photo, traite un flux de 25 im/s, résolution  $720 \times 576$  (DV) ou + (FullHD, UHD 4k -  $3840 \times 2160$ , UHD8k -  $7680 \times 4320$  pixels). Connexion FireWire, USB.



## Numérisation 3D

Point par point (à la main) :



Par scanner Laser ( $x, y, z + R, G, B$ ) :



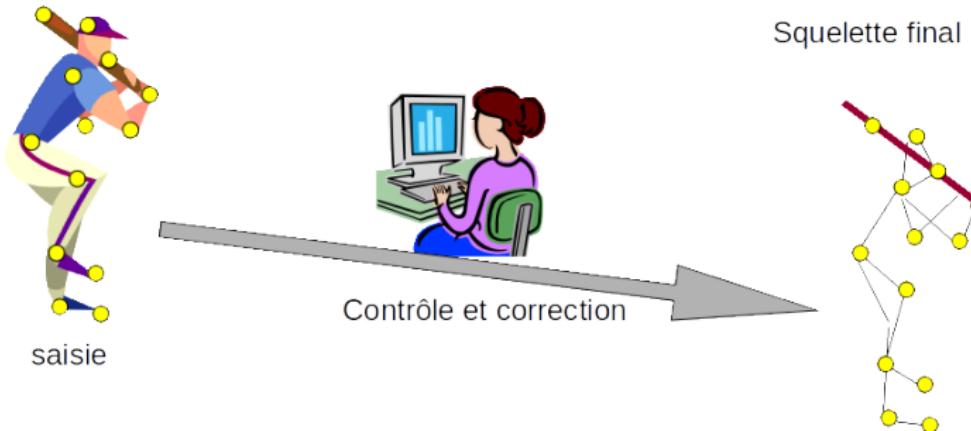
Par corrélation d'images 2D :



## Motion capture

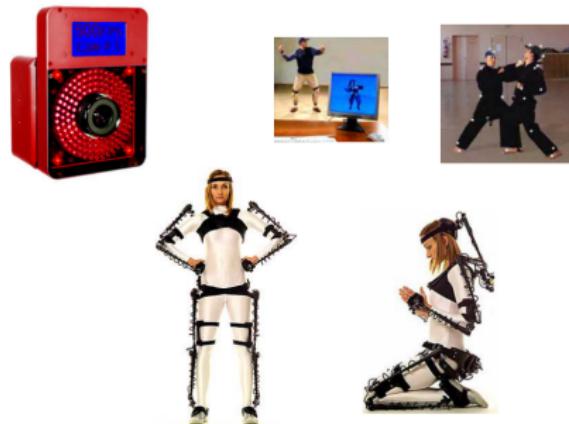
Utilisée par l'industrie du jeu, des nouvelles technologies. Plusieurs techniques sont utilisables : capteurs visibles, infrarouges, électromagnétiques. Elle nécessite beaucoup d'espace  $> 300 \text{ m}^2$ .

Principes :



# Périphériques de saisie

Principes :

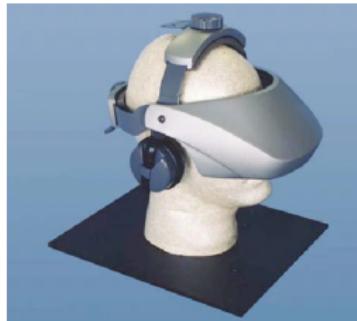


## Exemples de Mocap



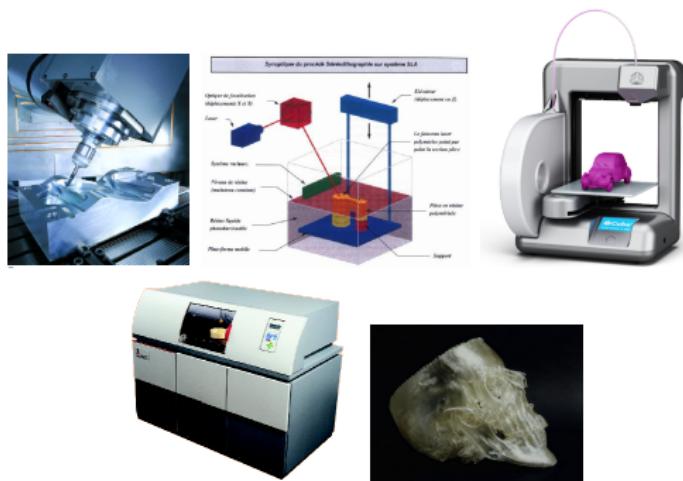
## Pérophériques de sortie 2D

- écran (analogique, TFT, plasma, ...),
- imprimante (laser, jet d'encre),
- HMD (*head mounted display*), HUD (*Head-Up Display*).



## Périphériques de sortie 3D

- robot de découpe : fraise sur 6 axes, choix du matériau, interaction de la machine,
- stéréolithographie,
- multi-jet : impression de plastique.



# Plan

① Organisation

② Introduction

③ Matériel

Stockage

Carte Video

Signal

Écrans

Numérisation

④ Pipeline graphique

Programmation du pipeline graphique

Comment cela fonctionne ?

⑤ Description des géométries en OpenGL 2.0

Avant c'était bien statique ...

... mais ça c'était avant

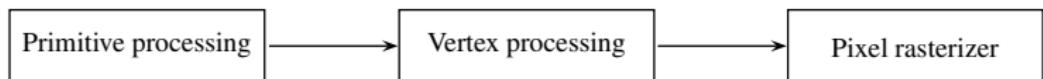
Les VBO

Les FBO

⑥ Autres matériels

⑦ Culture

## Pipeline habituel



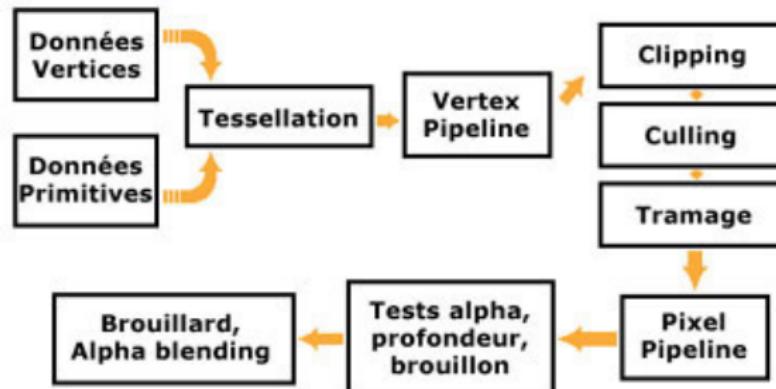
Depuis la version 1.4 l'API OpenGL permet la programmation du GPU grâce aux shaders. Les données en entrée du pipeline graphique sont celles que transmet le programme :

- position ( $x, y, z, w$ )
- couleur ( $r, g, b, a$ )
- coordonnées de texture ( $u, v$ )

On peut rajouter les normales aux points, aux faces, ou des attributs.

## Pipeline détaillé

Le pipeline graphique est en fait plus complexe, par opérations.

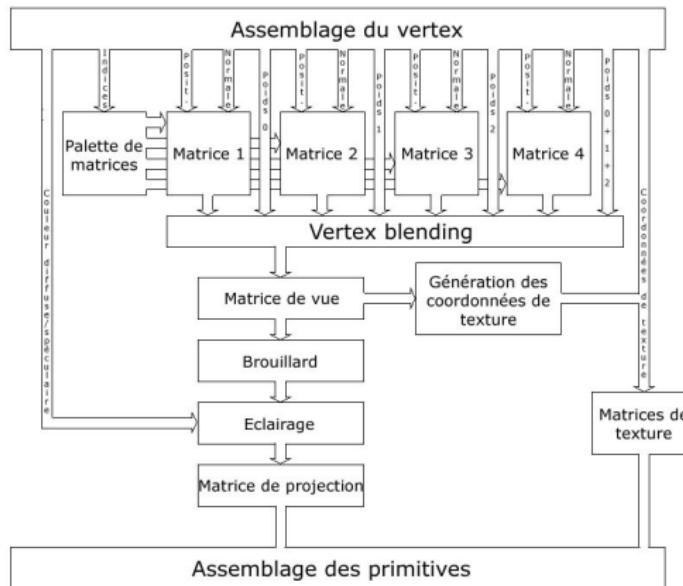


Le fragment en sortie de la projection 2D -> 3D (avant le Pixel Pipeline) contient pour chaque pixel :

- les coordonnées 2D entières,
- sa couleur,
- sa pseudo profondeur (profondeur relative dans le volume de vue).

# Architecture d'un GPU

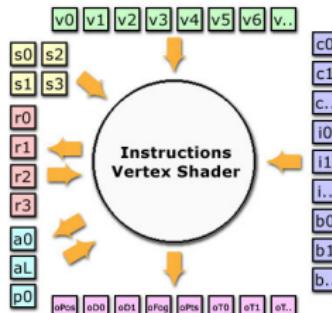
Le processeur graphique, une architecture différente...



# Architecture d'un GPU

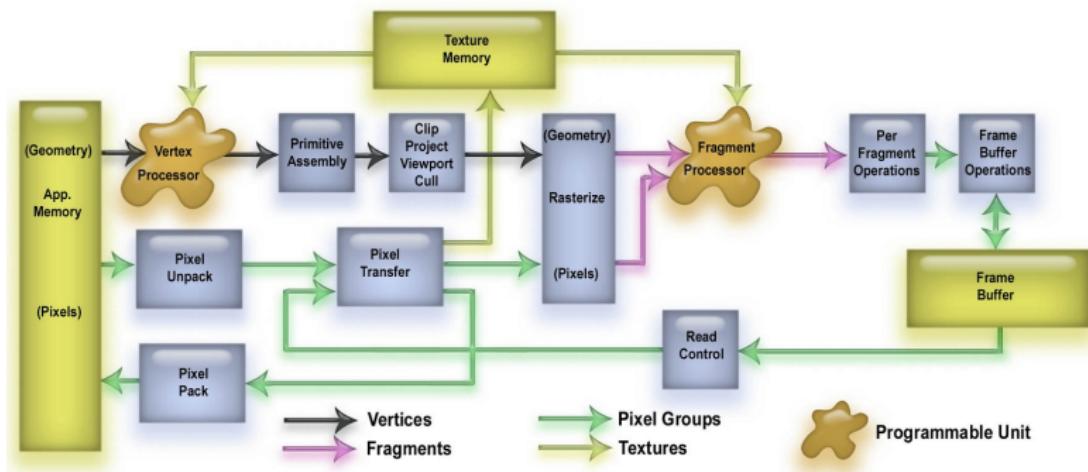
Le processeur graphique, une architecture différente...

- V0 - V... (15), registres d'entrée
- R0 - R... (12), registres de travail
- c0 - c... (256), constantes réelles
- i0 - i... (16), constantes entières
- b0 - b... (16), constantes booléennes
- s0 - s3, registres d'échantillonage, permet au vertex shader d'accéder aux textures
- a0, adresse d'un registre / aL : compteur /
- p0, branchements



## OpenGL2+

Le pipeline d'OpenGL 2.0 et + et donc relativement plus compliqué car on peut communiquer entre les unités programmables.



## Quels langages ?

On trouve comme d'habitude des implémentations propriétaires et des standards...

- Microsoft DirectX, se déclinant en 3 versions (de la plus ancienne à la plus récente) :
  - ASM, en assembleur comme son nom l'indique
  - HLSL, à l'aide d'un langage de haut niveau (C++ par ex.)
  - XNA (unification MsWindows/Xbox)

Le principe est de transformer les sources du shader en assembleur binaire, par le compilateur `fxc` si le shader est précompilé, par DirectX sinon. Le driver interprète cet assembleur en byte code spécifique au hardware.

## Quels langages ?

On trouve comme d'habitude des implémentations propriétaires et des standards...

- OpenGL
  - GLSL, mis en place par 3DLabs. Intégré à OpenGL2.0  
Le code GLSL du shader est fourni directement au driver de la carte graphique. Le compilateur est inclus au driver.
    - avantage : le compilateur est mis à jour avec le driver
    - inconvénient : l'obligation de développer un compilateur avec le driver

## Quels langages ?

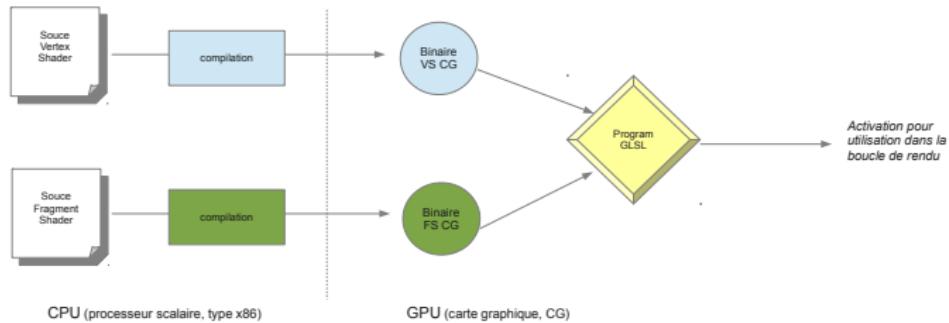
On trouve comme d'habitude des implémentations propriétaires et des standards...

- Nvidia
  - CG, cross-platform, langage spécifique
    - Transformation du langage en assembleur par le compilateur Cg. Assemblage du shader en format binaire par OpenGL ou DirectX. interprétation de cet assembleur en byte code spécifique au hardware.

## Introduction

Un shader OpenGL porte sur la géométrie (« Vertex Shader » ou VS) ou sur les fragments (« Fragment Shader » ou FS) qui sont des pixels avec une pseudo-profondeur (comme dans le Z-Buffer). Pour les utiliser, il faut construire un « Program » qui est l'union d'un VS et d'un FS.

Chaque shader est compilé par le pilote de la carte graphique, on peut donc faire cette étape avant le début de la boucle de rendu. La création d'un « Program » suivra donc les étapes suivantes :



# Plan

① Organisation

② Introduction

③ Matériel

Stockage

Carte Vidéo

Signal

Écrans

Numérisation

④ Pipeline graphique

Programmation du pipeline graphique

Comment cela fonctionne ?

⑤ Description des géométries en OpenGL 2.0

Avant c'était bien statique ...

... mais ça c'était avant

Les VBO

Les FBO

⑥ Autres matériels

⑦ Culture

## Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

- ➊ charger et lire les fichiers sources (attention à l'encodage, UTF-8 ou non peut entraîner des erreurs) ;

## Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

- ② créer des identifiants de shader (ces variables sont de type GLuint) :

```
VertexShader = glCreateShader(GL_VERTEX_SHADER);  
FragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

- ③ lier ces deux sources (chaîne de caractères) à un structure OpenGL :

```
glShaderSource(IDVertexShader, 1, &constVS, NULL);  
glShaderSource(IDFragmentShader, 1, &constFS, NULL);
```

## Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

- ④ compiler ces 2 sources :

```
glCompileShader(IDVertexShader);  
glCompileShader(IDFragmentShader);
```

Note : Comme le shader est compilé par la carte graphique, on peut le modifier et relancer le programme principal. Cela recompile forcément le shader.

## Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

- ⑤ créer un `ProgramObject`, qui contient les binaires du vertex et du fragment shader et qui remplacera le programme natif de la carte graphique :

```
IDProgramObject = glCreateProgram();
```

- ⑥ attacher les 2 shaders à ce programme (on peut les détacher et en attacher d'autre dynamiquement) :

```
glAttachShader(IDProgramObject, IDVertexShader);  
glAttachShader(IDProgramObject, IDFragmentShader);
```

- ⑦ lier le programme au moteur graphique :

```
glLinkProgram(IDProgramObject);
```

## Mise en place des shaders

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

- ⑧ et pour finir, utiliser ce programme pour toutes les actions OpenGL qui suivent :

```
glUseProgram(IDProgramObject);
```

Note : `glUseProgram(0)` pour désactiver ce programme.

# Le shader de base

## VERTEX SHADER

```
void main(void) {  
//renvoie la couleur de la face avant sur les sommets dans le pipeline  
    gl_FrontColor = gl_Color;  
  
//effectue les transfos de modélisation et de vue  
    gl_Position = ftransform();  
}
```

## FRAGMENT SHADER

```
void main (void) {  
  
//on prend la couleur dans le pipeline  
//et on l'utilise pour l'affichage  
    gl_FragColor = gl_Color;  
}
```

Attention, s'applique sommet/sommet, pixel/pixel.

## Qu'est-ce que l'on peut faire d'un shader ?

Mettre les pixels de la scène en niveau de gris :

### FRAGMENT SHADER

```
void main (void)
{
    //on récupère la couleur dans le pipeline
    vec4 vectcolor = gl_Color;
    float luminance = 0.2126*vectcolor.r + 0.7152*vectcolor.g
                    + 0.0722*vectcolor.b;

    //on calcule la luminance
    vectcolor = vec4(luminance, luminance, luminance, 0.0);

    //et on la renvoie pour l'affichage
    gl_FragColor = vectcolor;
}
```

## types GLSL

Comme en C++, on trouve les types :

- float
- bool
- int

Plus les types vectoriels manipulés par le GPU :

- Vecteurs 2-4D
  - vec2,3,4 de réels (floats)
  - bvec2,3,4 de booléens
  - ivec2,3,4 d'entiers
- Matrices carrées 2x2, 3x3 et 4x4
  - mat2, mat3, mat4

## Précautions à prendre

Attention !

```
vec2 a = vec2(1.0,2.0);
vec2 b = vec2(3.0,4.0);
vec4 c = vec4(a,b);           //donne c=vec4(1.0,2.0,3.0,4.0);
vec2 g = vec2(1.0,2.0);
float h = 3.0;                // et pas 3
vec3 j = vec3(g,h);
```

Un *tips&tricks* : le *swizzling*. On peut échanger les coordonnées dans la description, RGB vs XYZ ou indices :

```
vec3 v0 = vec3(0.0, 1.0, 2.0);
vec3 v1 = v0.zyx;              //v1 est (1.0, 2.0, 3.0)
vec3 v2.xzy = v1.zxy;          //v2 = v0 !
vec3 v3.rgb = v0.xyz;          //v3 = v0
```

## Textures et structures

On trouve également les types Texture, les « Samplers » :

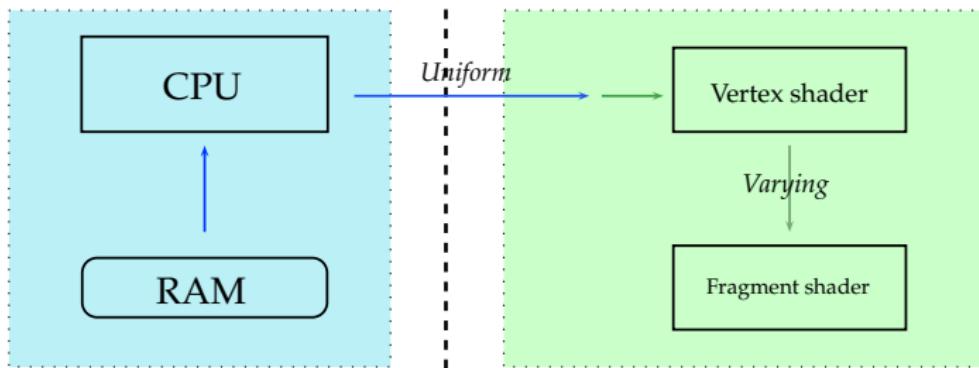
- sampler1D, texture 1D
- sampler2D, texture 2D
- sampler3D, texture 3D
- samplerCube, texture cube map
- sampler1DShadow, pour les shadow maps
- sampler2DShadow, pour les shadow maps

Et les structures comme en C :

```
struct lastructure {  
    vec3 monvecteur;  
    float monreel;  
} mastructure;
```

## Passage de valeurs

On peut aussi passer des données (`int`, `float`, `float *`, ...) entre le programme en CPU et les shaders du GPU. On peut également passer des données entre shaders (dans le sens du pipeline).



## Exemple de shader

### FRAGMENT SHADER

```
uniform vec3 lightDir;  
varying vec3 normal;  
void main()  
{  
    float intensity;  
    vec4 color;  
    intensity = dot(lightDir,normal);  
  
    if (intensity > 0.95)  
        color = vec4(1.0,0.5,0.5,1.0);  
    else if (intensity > 0.5)  
        color = vec4(0.6,0.3,0.3,1.0);  
    else if (intensity > 0.25)  
        color = vec4(0.4,0.2,0.2,1.0);  
    else  
        color = vec4(0.2,0.1,0.1,1.0);  
  
    gl_FragColor = color;  
}
```

### VERTEX SHADER

```
varying vec3 normal;  
  
void main()  
{  
    normal = gl_Normal;  
  
    gl_Position = ftransform();  
}
```



# Plus dur

```
#define KERNEL_SIZE 9

float kernel[KERNEL_SIZE];

uniform sampler2D colorMap;

//taille de l'image=512, attention en dur !2000
const float step_w = 1.0 / 512.0;
const float step_h = 1.0 / 512.0;

vec2 offset[KERNEL_SIZE];

void main(void)
{
    //le noyau de la convolution, i.e. le masque appliqué aux pixels
    //pour l'instant laplacien
    kernel[0] = 0.0; kernel[1] = 1.0; kernel[2] = 0.0;
    kernel[3] = 1.0; kernel[4] = -4.0; kernel[5] = 1.0;
    kernel[6] = 0.0; kernel[7] = 1.0; kernel[8] = 0.0;

    //comment sont définis les voisins du pixel courant
    offset[0] = vec2(-step_w, -step_h); offset[1] = vec2(0.0, -step_h); offset[2] = vec2(step_w, -step_h);
    offset[3] = vec2(-step_w, 0.0); offset[4] = vec2(0.0, 0.0); offset[5] = vec2(step_w, 0.0);
    offset[6] = vec2(-step_w, step_h); offset[7] = vec2(0.0, step_h); offset[8] = vec2(step_w, step_h);
```

>... suite après ...>

# Plus dur

```
int i = 0;
vec4 sum = vec4(0.0);

//on effectue les opérations que sur les pixels dans la moitié gauche de l'image
if(gl_TexCoord[0].s < 0.498) {
    for( i = 0; i < KERNEL_SIZE; i++ ) {
        vec4 tmp = texture2D(colorMap, gl_TexCoord[0].st + offset[i]);
        float luminosite = 0.299 * tmp.r + 0.587 * tmp.g + 0.114 * tmp.b;

        tmp.rgb = vec3(luminosite);
        sum += tmp * kernel[i];
    }
}
//sur la partie droite, on ne fait rien
else if( gl_TexCoord[0].s > 0.502 ) {
    sum = texture2D(colorMap, gl_TexCoord[0].xy);
}
else {
    sum = vec4(1.0, 0.0, 0.0, 1.0); //au milieu on met des pixels rouges
}

gl_FragColor = sum;
```

## Traitement d'une image

mots-clés : fragment shader, convolution et masque, offset de texture, uniform sampler2D.

Pour traiter une image avec un Fragment Shader, on a besoin de sommets qui passent dans un VS puis qui sont utilisés dans un FS. Le plus simple est donc d'utiliser un QUAD sur lequel on va plaquer une texture, et avec lequel on effectuera le Fragment Shader.

Le Vertex Shader contient juste la transformation des points 3D en points 2D et le passage de la texture au Fragment Shader :

```
void main( void )
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

## Traitement d'une image

mots-clés : fragment shader, convolution et masque, offset de texture, uniform sampler2D.

Pour traiter une image avec un Fragment Shader, on a besoin de sommets qui passent dans un VS puis qui sont utilisés dans un FS. Le plus simple est donc d'utiliser un QUAD sur lequel on va plaquer une texture, et avec lequel on effectuera le Fragment Shader.

La texture est récupérée dans le Fragment Shader dans un uniform sampler2D. On peut alors utiliser le point (`gl_TexCoord[0].st`) de la 1ère texture (" [0] ") du sampler2D pour avoir sa couleur.

On peut aussi utiliser des décalages pour tenir compte des points voisins (offset). Cela permet de programmer des masques de traitement d'images (convolution) très facilement.

## Exemple de Vertex Shader

Modifier le vertex shader : comme on récupère une position dans `gl_Vertex`, on peut la faire modifier par la carte graphique. par exemple, une translation de  $(1, 1, 0)$  :

### VERTEX SHADER

```
void main(void) {  
    //Translation  
    vec4 point = gl_Vertex;  
    point.x = point.x + 1.0;  
    point.y = point.y + 1.0;  
    gl_Position = gl_ModelViewProjectionMatrix * point;  
}
```

## Plus dur

Modifier le comportement vertex shader, en fonction de données qui sont transmises par le programme en C++. On utilise des variables de type **Uniform**.

Comme elles sont stockées sur la carte graphique, on doit, depuis le prog. C++, pouvoir les modifier. Il faut connaître l'endroit où la carte les stocke (grossso modo, la carte renvoie une référence d'un emplacement mémoire). Par exemple, on passe une variable `mytime` incrémentée dans le prog en C++ `RenderScene` et on s'en sert pour déformer les sommets dans le vertex shader.

## Plus dur

Modifier le comportement vertex shader, en fonction de données qui sont transmises par le programme en C++. On utilise des variables de type **Uniform**.

### VERTEX SHADER

```
uniform float cputime;  
  
void main(void) {  
    vec4 v = vec4(gl_Vertex);  
  
    //sin() est une fonction implémentée dans le GPU,  
    //cf GLSL référence  
  
    v.z = sin(5.0*v.x + cputime)*0.5;  
  
    gl_Position = gl_ModelViewProjectionMatrix * v;  
}
```

## Plus dur

Modifier le comportement vertex shader, en fonction de données qui sont transmises par le programme en C++. On utilise des variables de type **Uniform**.

### PROGRAMME C++

```
<variables en global>
GLint loc;
float mytime=0.0f;

<fonction RenderScene()>
    glRotatef(angle,1,1,0);
    glColor3f(1.0, 0.0, 0.0);

//on fait la liaison entre
//la valeur en GPU à l'adresse "loc"
//et la valeur dans "mytime"
    glUniform1fARB(loc, mytime);

    glutSolidTeapot(1);

    mytime += 0.01;

<fonction setShaders()>
    glUseProgram(IDProgramObject);

    //on récupère l'adresse de
    //la variable cputime du GPU, par son nom, dans loc
    loc = glGetUniformLocation(IDProgramObject,"cputime");
```

## « Éditeurs » GLSL

- Shader-maker, [http://cg.in.tu-clausthal.de/teaching/shader\\_maker/index.shtml](http://cg.in.tu-clausthal.de/teaching/shader_maker/index.shtml)
- GLSL Validate
- GLSL Parser Test
- Rendermonkey (ATI)
- ShaderDesigner (Typhoon Labs,  
<http://www.opengl.org/sdk/tools/ShaderDesigner/>)
- gDEBugger (Graphics Remedy)

## Exemple de Varying

Varying permet de passer une valeur d'un Vertex Shader à un Fragment Shader pour ce pixel.

### VERTEX

```
//les valeurs que l'on reçoit du CPU
uniform vec3 Uv3centre_deformation;
uniform vec3 Uv3vecteur_deformation;
uniform float Ufrayon_deformation;

//une valeur que l'on souhaite faire passer au Fragment shader
varying float VFactif;

void main(void) {
vec4 point = gl_Vertex;
VFactif = 0.0;

//on calcule la distance du centre de la déformation au vertex (point) courant
//distance est une fonction de GLSL (cf. mémo)

float d = distance(vec4(Uv3centre_deformation, 1.0), point);

//si la distance ci-dessus est inférieure au rayon d'action de la contrainte
//on déforme de manière linéaire (max=centre, min=rayon)
if (d < Ufrayon_deformation) {
    point.xyz = point.xyz + (1.0 - (d / Ufrayon_deformation))
        * Uv3vecteur_xyz;

    VFactif = 1.0;
}

//et on projette ce point
```

## Exemple de Varying

Varying permet de passer une valeur d'un Vertex Shader à un Fragment Shader pour ce pixel.

FRAGMENT

```
varying float VFactif;

void main (void) {
    if (VFactif == 1.0)
        //modif du pixel actif en rouge
        gl_FragColor = vec4(1.0,0.0,0.0,1.0);
    else
        //modif de tous les pixels (bleu)
        gl_FragColor = vec4(0.0,0.0,1.0,1.0);
}
```

## Geometry shader

Les Geometry Shader permettent de créer à la volée des géométries, qui retourneront dans le pipeline (transformations, projections, rasterisation).

```
void main(void) {  
    mat4x4 bezierBasis = mat4x4( 1, -3, 3, -1, 0, 3, -6, 3 , 0, 0, 3, -3 , 0, 0, 0, 1);  
    for(int i=0; i<64; i++) {  
        float t = i / (64.0-1.0);  
        vec4 tvec = vec4(1, t, t*t, t*t*t);  
        vec4 b = tvec*bezierBasis;  
        vec4 p = gl_PositionIn[0]*b.x + gl_PositionIn[1]*b.y + gl_PositionIn[2]*b.z + gl_PositionIn[3]*b.w;  
  
        gl_Position = p;  
        EmitVertex();  
    }  
  
    EndPrimitive();  
}
```

Le renvoi de géométrie se fait grâce à `EmitVertex()`, La primitive doit être préalablement choisie (triangle, point, ligne).

# Plan

- ① Organisation
- ② Introduction
- ③ Matériel
  - Stockage
  - Carte Video
  - Signal
  - Écrans
  - Numérisation
- ④ Pipeline graphique
  - Programmation du pipeline graphique
  - Comment cela fonctionne ?
- ⑤ Description des géométries en OpenGL 2.0
  - Avant c'était bien statique ...
  - ... mais ça c'était avant
  - Les VBO
  - Les FBO
- ⑥ Autres matériels
- ⑦ Culture

## Mode direct

... Et coûteux en temps et en débit.

```
//Dans la boucle de rendu
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 1.0f);
    glVertex3f(2.0f, 2.0f, 2.0f);
glEnd();
```

Cela implique qu'à chaque image (à chaque fois qu'OpenGL a besoin de retracer), on re-décrit toute la géométrie de la scène et qu'on la renvoie de la RAM du CPU vers la RAM du GPU. Si l'objet n'est pas modifié, c'est beaucoup de temps et d'énergie perdus !

## DisplayLists

Vous avez sans doute vus les **DisplayLists**, qui permettent la pré-compilation des objets en RAM. La carte graphique arrange les données, les prépare pour leur utilisation et repère leurs emplacements grâce à un identifiant.

```
id = glGenLists(1);

glNewList(id, GL_COMPILE);

for (int i=0; i < nbfaces; i++)
    glBegin(GL_TRIANGLES);
        glVertex3f(...);
        glVertex3f(...);
        glVertex3f(...);
    glEnd();

glEndList();
```

On effectue donc un pré-travail pour la carte graphique, il n'y a plus qu'à appeler la liste (`glCallList(id)`) pour l'envoyer au traitement du GPU.

Il reste 2 inconvénients : les transmissions CPU -> GPU et l'impossibilité de modifier ces géométries sans recréer la liste.

Évidemment si l'objet est modifié, il faut détruire et refaire la DisplayList et la recompiler. Le 2ème inconvénient peut être résolu en implémentant un VertexShader qui peut modifier une géométrie dans une DisplayList.  
C'est évidemment plus rapide, pas la peine de recréer la géométrie à chaque fois mais on peut peut-être faire mieux...

# Plan

- ① Organisation
- ② Introduction
- ③ Matériel
  - Stockage
  - Carte Video
  - Signal
  - Écrans
  - Numérisation
- ④ Pipeline graphique
  - Programmation du pipeline graphique
  - Comment cela fonctionne ?
- ⑤ Description des géométries en OpenGL 2.0
  - Avant c'était bien statique ...
  - ... mais ça c'était avant
  - Les VBO
  - Les FBO
- ⑥ Autres matériels
- ⑦ Culture

## Mode indirect

### Les géométries, en mieux : `VertexArray`

Depuis la version 1.2 d'OpenGL, on a la possibilité de préparer un peu mieux les données pour la carte graphique. Les `VertexArray` permettent de stocker la géométrie d'un objet, mais aussi sa (ses) couleur(s), ses normales, ses coordonnées de texture... dans des tableaux. Lors du rendu, on envoie ces tableaux à la carte graphique qui n'a plus qu'à lire, les données sont déjà organisées.

## Vertex Array

Comment fait-on ? On crée des tableaux qui vont contenir les données de l'objet (des tableaux de float), on met dedans toutes les données de l'objet chargé, et on les utilise plus tard.

```
float ColorArray[nbsommets*3] ;  
float NormalArray[nbsommets*3] ;  
float VertexArray[nbsommets*3] ;
```

Dans le rendu, on active le rendu depuis des buffers, pour chaque type de données à transmettre :

```
//attention, on va se servir de sommets
glEnableClientState(GL_VERTEX_ARRAY);
//attention, on va se servir de normales
glEnableClientState(GL_NORMAL_ARRAY);
//attention, on va se servir de couleurs
glEnableClientState(GL_COLOR_ARRAY);

//et voilà le buffer des couleurs, sur 3 float pour chaque couleur
glColorPointer(3, GL_FLOAT, 0, ColorArray);

//le buffer des normales, sur des floats (3 implicitement)
glNormalPointer(GL_FLOAT, 0, NormalArray);

//le buffer des sommets, 3 floats pour chaque coordonnées
glVertexPointer(3, GL_FLOAT, 0, VertexArray);

//et hop, on envoie au rendu de la CG
glDrawArrays(GL_TRIANGLES, 0, nbfaces * 3);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

# Plan

- ① Organisation
- ② Introduction
- ③ Matériel
  - Stockage
  - Carte Video
  - Signal
  - Écrans
  - Numérisation
- ④ Pipeline graphique
  - Programmation du pipeline graphique
  - Comment cela fonctionne ?
- ⑤ Description des géométries en OpenGL 2.0
  - Avant c'était bien statique ...
  - ... mais ça c'était avant
- ⑥ Les VBO
- ⑦ Les FBO
- ⑧ Autres matériels
- ⑨ Culture

## Le Graal : les VBO

Les géométries, en encore encore mieux : `VertexBufferObject`

Depuis OpenGL 1.5 on peut se servir des `VertexBufferObject`. L'intérêt de ces petites bêtes est que l'on peut stocker directement la géométrie sur la carte graphique.

Comme pour les `VertexArrays`, on utilise des tableaux pour stocker la géométrie d'un objet, on lie ensuite ces tableaux avec des zones mémoires de la carte graphique, on copie les données, et hop, c'est prêt ! (du coup on peut désallouer la mémoire en RAM du CPU puisque la géométrie est sur la carte graphique).

Il devient donc inutile de conserver la géométrie à la fois dans le CPU et le GPU, d'où un gain de mémoire en plus du gain de transfert.

## Le mode des VBO

On peut gérer les VBO sous 3 modes :

- **GL\_STREAM\_DRAW** lorsque les informations sur les sommets peuvent être mises à jour entre chaque rendu = VertexArrays. On envoie les tableaux à chaque image.
- **GL\_DYNAMIC\_DRAW** lorsque les informations sur les sommets peuvent être mises à jour entre chaque frame. On utilise ce mode pour laisser la carte graphique gérer les emplacements mémoires des données, pour le rendu multi-passe notamment.
- **GL\_STATIC\_DRAW** lorsque les informations sur les sommets ne sont pas mises à jour - ce qui redonne le fonctionnement d'une DisplayList.

Se rajoute à ces modes, des modes d'accès aux données : READ, WRITE, COPY, READ\_WRITE. Ce qui permet également de copier des parties d'un objet dans un autre, et même de faire des interactions avec le CPU.

note : attention aux différents modes, cela ne fonctionne pas sur toutes les cartes graphiques.

## Comment fait-on cela ?

Préparation des données :

- ➊ on crée les tableaux de géométrie comme pour les VertexArray

```
float lcolors[nbsommets*3];  
float lnormales[nbsommets*3];  
float lsommets[nbsommets*3];
```

- ➋ on crée les buffers GPU qui contiendront ces données (`glBindBuffer`, `glBufferData`)

# Création de VBO

```
//VBO pour les sommets
glGenBuffers((GLsizei) 1, &VBOSommets);
 glBindBuffer(GL_ARRAY_BUFFER, VBOSommets);
 glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float), lsommets, GL_STATIC_DRAW);

//VBO pour les couleurs
glGenBuffers((GLsizei) 1, &VBOCouleurs);
 glBindBuffer(GL_ARRAY_BUFFER, VBOCouleurs);
 glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float), lcolors, GL_STATIC_DRAW);

//VBO pour les normales
glGenBuffers((GLsizei) 1, &VBONormales);
 glBindBuffer(GL_ARRAY_BUFFER, VBONormales);
 glBufferData(GL_ARRAY_BUFFER, nbfaces * 9 * sizeof(float) , lnormales, GL_STATIC_DRAW);
```

# Rendu de VBO

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);

glBindBuffer( GL_ARRAY_BUFFER, VBOSommets);
glVertexPointer( 3, GL_FLOAT, 0, (char *) NULL );

glBindBuffer( GL_ARRAY_BUFFER, VBOCouleurs);
glColorPointer( 3, GL_FLOAT, 0, (char *) NULL );

glBindBuffer( GL_ARRAY_BUFFER, VBONormales);
glNormalPointer(GL_FLOAT, 0, (char *) NULL );

glDrawArrays(GL_TRIANGLES, 0, monobjet.nbfaces * 3);

glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

# Plan

- ① Organisation
- ② Introduction
- ③ Matériel
  - Stockage
  - Carte Video
  - Signal
  - Écrans
  - Numérisation
- ④ Pipeline graphique
  - Programmation du pipeline graphique
  - Comment cela fonctionne ?
- ⑤ Description des géométries en OpenGL 2.0
  - Avant c'était bien statique ...
  - ... mais ça c'était avant
  - Les VBO
  - Les FBO**
- ⑥ Autres matériels
- ⑦ Culture

## Les VBO

Et pour les images : le FBO

Un Frame Buffer Object (FBO) est l'espace de stockage qu'utilise OpenGL pour mettre, au fur et à mesure de son avancement, toutes les données issues d'algorithmes de l'espace image (lié à la résolution du rendu, en pixels). Par exemple les pixels de la scène, mais aussi le Z-buffer. Dans un fonctionnement statique, vous devez utiliser des stencils ou des masques pour interagir avec le Z-buffer, et vous ne pouvez pas modifier les pixels de l'image 2D rendue.

À partir d'OpenGL 2, on peut adresser ces espaces de stockages comme des textures (Texture Object) ou des buffers (RenderBuffer Object). Pour schématiser, les premiers sont faciles d'accès (par le programme principal ou un shader) et les seconds sont plus rapides (stockés en mémoire vive de la carte graphique, pas dans l'espace des textures).

## Création d'un FBO

Pour créer un FBO, c'est comme pour les textures et les VBO, il faut générer une référence sur la carte graphique :

- ➊ obtenir un identifiant

```
void glGenFramebuffersEXT(GLsizei quantité, GLuint* idfbo)
```

## Création d'un FBO

Pour créer un FBO, c'est comme pour les textures et les VBO, il faut générer une référence sur la carte graphique :

- ② Pour attacher une texture à ce FBO, on écrit :

```
glFramebufferTexture2DEXT(GLenum target,  
                           GLenum attachmentPoint,  
                           GLenum textureTarget,  
                           GLuint textureId,  
                           GLint level)
```

avec :

- `target = GL_FRAMEBUFFER_EXT,`
- `attachmentPoint` dans `GL_COLOR_ATTACHMENT0_EXT`, ..., `GL_COLOR_ATTACHMENTn_EXT`, `GL_DEPTH_ATTACHMENT_EXT`, `GL_STENCIL_ATTACHMENT_EXT` selon le FBO à récupérer.
- `textureTarget = GL_TEXTURE_2D,`
- `textureId = identifiant de votre texture`
- `level = ?`, comme vous voulez, c'est le niveau de mipmapping de la texture finale.

## Rendu d'un VBO

On active ensuite le FBO dans la boucle de rendu par :

```
void glBindFramebufferEXT(GLenum target, GLuint id)
```

avec target = GL\_FRAMEBUFFER\_EXT. Si on met target = 0, le FBO est désactivé.

Note : faire du rendu (off-screen) directement dans une image ou un fichier AVI -> c'est le pbuffer, et c'est différent.

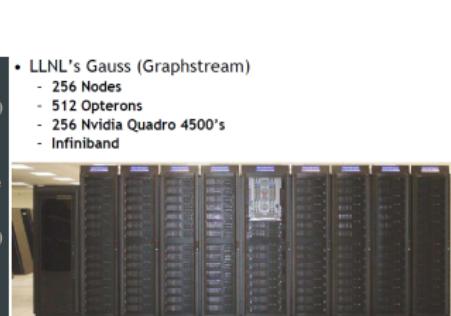
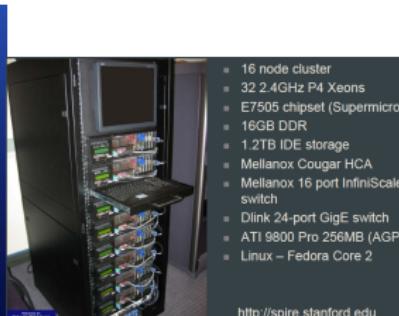
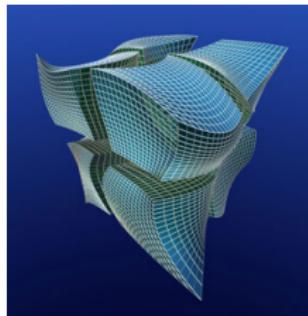
On peut imaginer diverses utilisation des FBO, par exemple :

- effectuer un flou d'une scène selon la profondeur (comme dans les déserts par exemple) -> c'est faire 2 FBO (couleurs et profondeur) + 1 fragment shader qui parcourt la texture de profondeur et modifie la texture de couleur par un masque de coefficients pris dans la profondeur (comme une convolution) ;
- appliquer des effets d'ombrage ou d'éclairage. On peut mettre une caméra à la place de la lumière, prendre le tampon de profondeur et l'utiliser pour faire un éclairage des sommets visibles et un ombrage des autres (cf shadowmap).

## Et la suite ...

### Convergence CPU/GPU

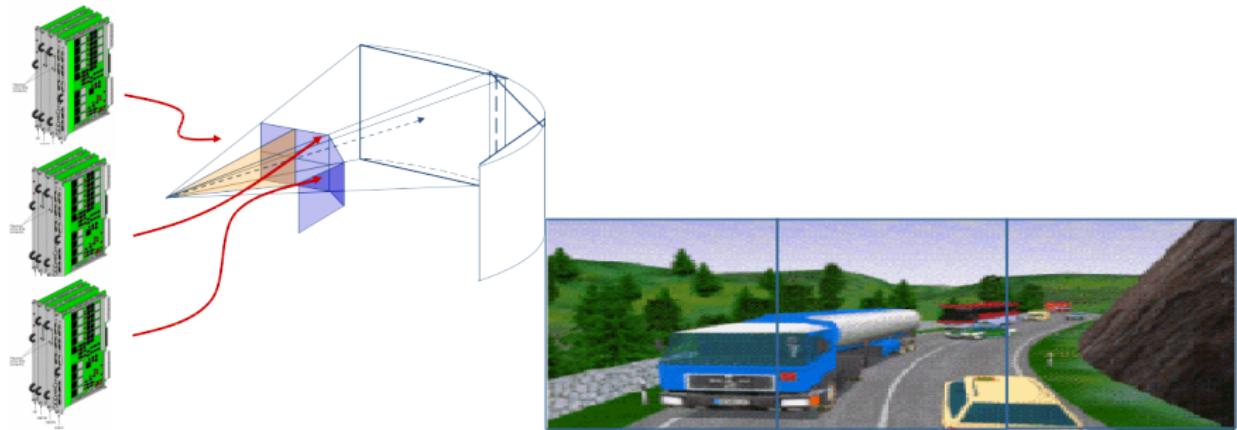
- Shader model 4
  - unification des unités de traitement, pas de différence vertex ou fragment
  - 32 unités de shader
- Geometry Shader
  - surfaces de haut-niveau (Bézier, Bspline, Nurbs, subdivision)
  - rendu automatique (tesselation)
- GPGPU
  - Compute Shader
  - programmation en calcul vectoriel massif sur carte graphique
  - cf [www.gpgpu.org](http://www.gpgpu.org)



## Le graphique en parallèle

Besoins de plusieurs canaux en parallèle :

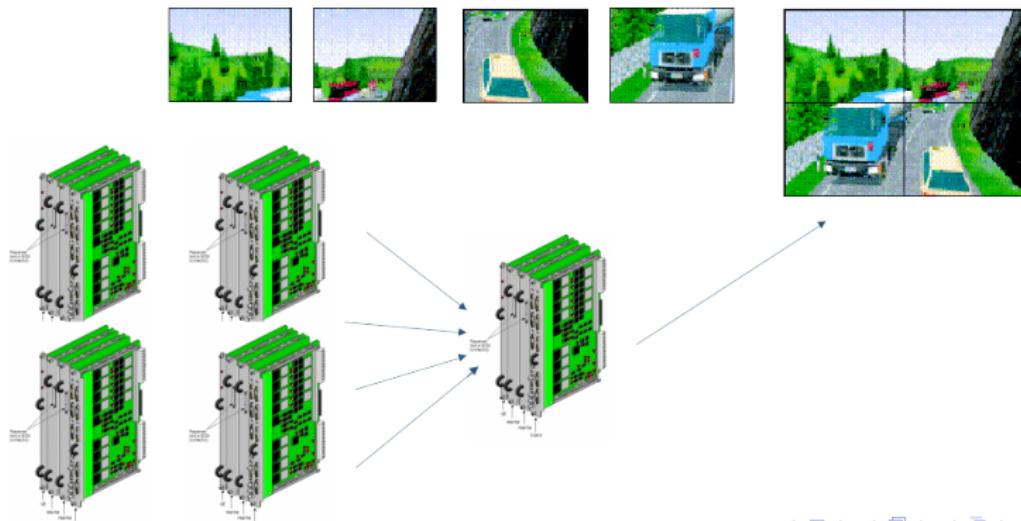
- larges vues, jointes
- stéréo,
- immersion (CAVE)



# Le graphique en parallèle

Besoin de performances :

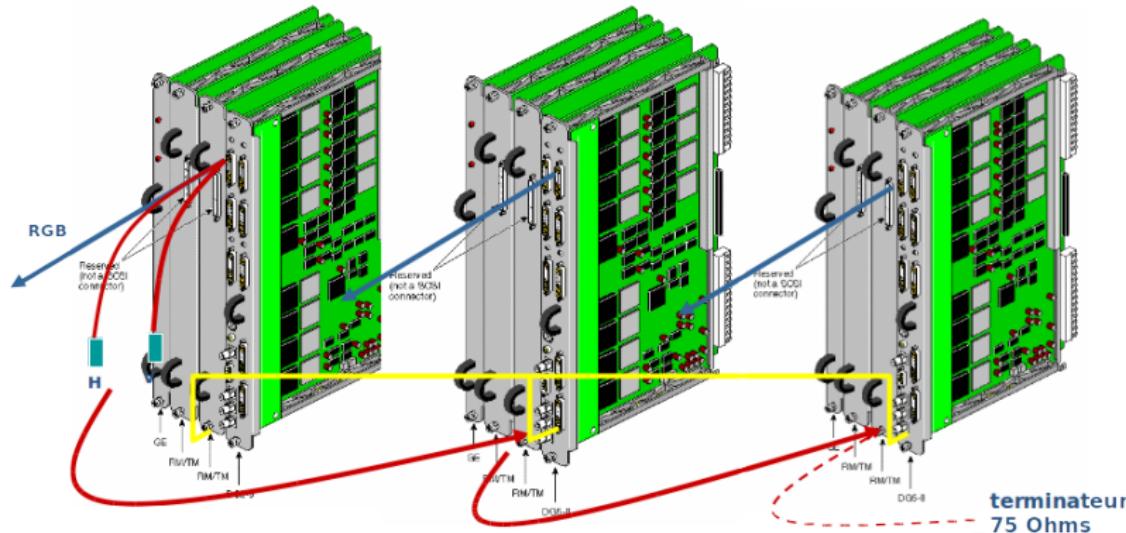
- gros volume de données à visualiser
- visualisation complexe
- « bonne » interactivité et « bonne » qualité de rendu



## Le graphique en parallèle

Deux types de synchronisation :

- synchronisation des lignes vidéo (*genlock*)
- synchronisation des images (*swap ready*)



# Réalité augmentée / virtuelle



# SGI - Silicon Graphics

InfinitePerformance :

- V-Brick : modules rack qui s'intègre dans une architecture Onyx 3000.  
Chaque V-Brick a 1 ou 2 pipeline InfinitePerformance, chacun capable de 17.7 million triangles /s et 484 megapixels /s.
- SGI graphics compositor : rack qui combine la sortie de 2 à 4 pipelines en un flot de sortie.



## Evans et Sutherland

2 chercheurs américains qui ont créé les 1ères cartes graphiques (vectorielles), les premières stations pour le graphisme, le premier crayon optique. C'est (depuis 1968) une société spécialisée dans la simulation, le data-mining et les cartes vidéo professionnelles (HUD, casques, CAO, ...)



## Intergraph

- 1970 : Vax/VMS pour le graphique
- 1985 : Processeur Clipper (Risc, Unix)
- 1992 : C5, architecture 64 bits, instruction parallèle
- développement avec Intel du pentium et pentium Pro
- a vendu la partie graphique à SGI, les cartes vidéo à 3Dlabs en 2000
- place importante sur les SIG (logiciel)

