TELECOMLille1
ECOLE D'INGENIEURS

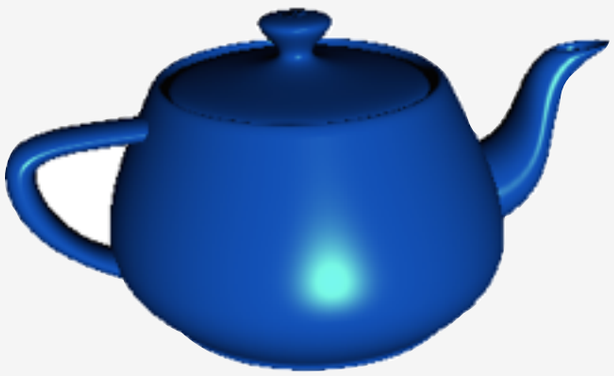
INTRODUCTION AUX CARTES GRAPHIQUES

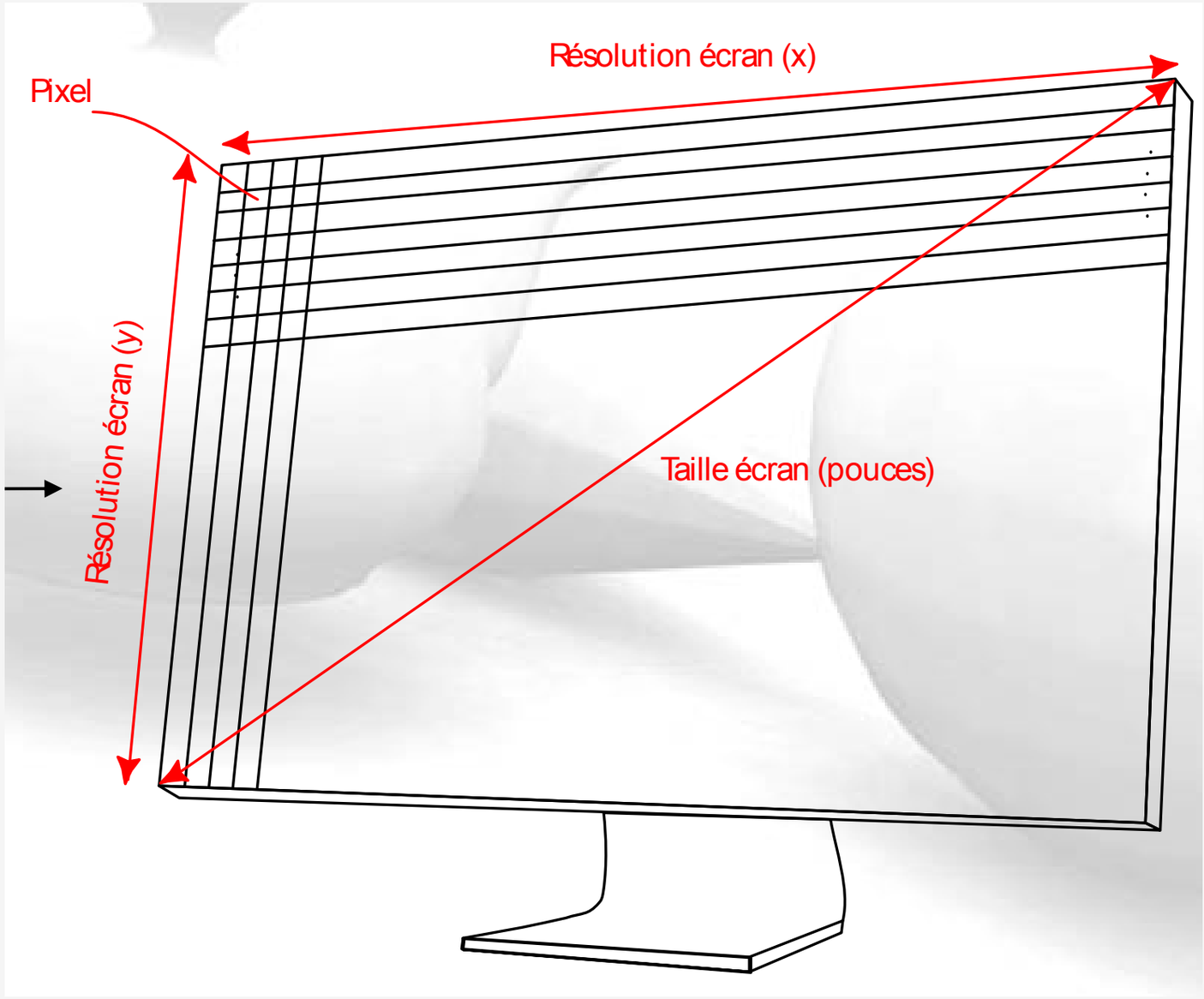
Novembre 2013

Thomas Pietrzak (supports adaptés de Géry Casiez)

ECRANS ET IMAGES

Carte graphique





CARACTÉRISTIQUES D'UN ÉCRAN

- Taille (in/cm) : diagonale de l'écran, ratio largeur / hauteur
- Résolution maximale : nb pixels en largeur x nb pixels en hauteur
- Nombre de couleurs
- Fréquence de rafraîchissement maximale (Hz)

DENSITÉ DE PIXELS

- Pixels per inch (PPI)

$$PPI = \frac{d_p}{d_i}$$

$$d_p = \sqrt{w_p^2 + h_p^2}$$

d_p : résolution en pixels de la diagonale

w_p et h_p : résolution en pixels de la largeur et de la hauteur

d_i : longueur de diagonale en pouces

- Ecran 20" avec résolution 1680x1050, PPI = 99.06
- PPI compris entre 20 et 250 PPI

L'ŒIL HUMAIN - RÉSOLUTION

- A partir de 300 PPI, l'œil humain ne peut plus distinguer les détails

  ⇒ qualité d'impression

- Pour un écran 17" format 4/3, => résolution de 4080x3060

  4K : 3840 × 2160

L'ŒIL HUMAIN - FRÉQUENCE

Fonctionnement de l'œil (persistance rétinienne)

- Une image perçue reste environ 0,1 sec sur la rétine

  ⇒ rémanence visuelle (dépend de l'intensité lumineuse)

- En enchaînant les images à une fréquence de 25Hz à 30Hz

  ⇒ effet d'animation

- **Au moins 72 Hz pour le confort visuel (pas d'effet « flicker »)**

L'ŒIL HUMAIN

Selon les sources l'œil humain peut discriminer

entre 100 000 et 10 millions de couleurs

MODES GRAPHIQUES

MODES GRAPHIQUES

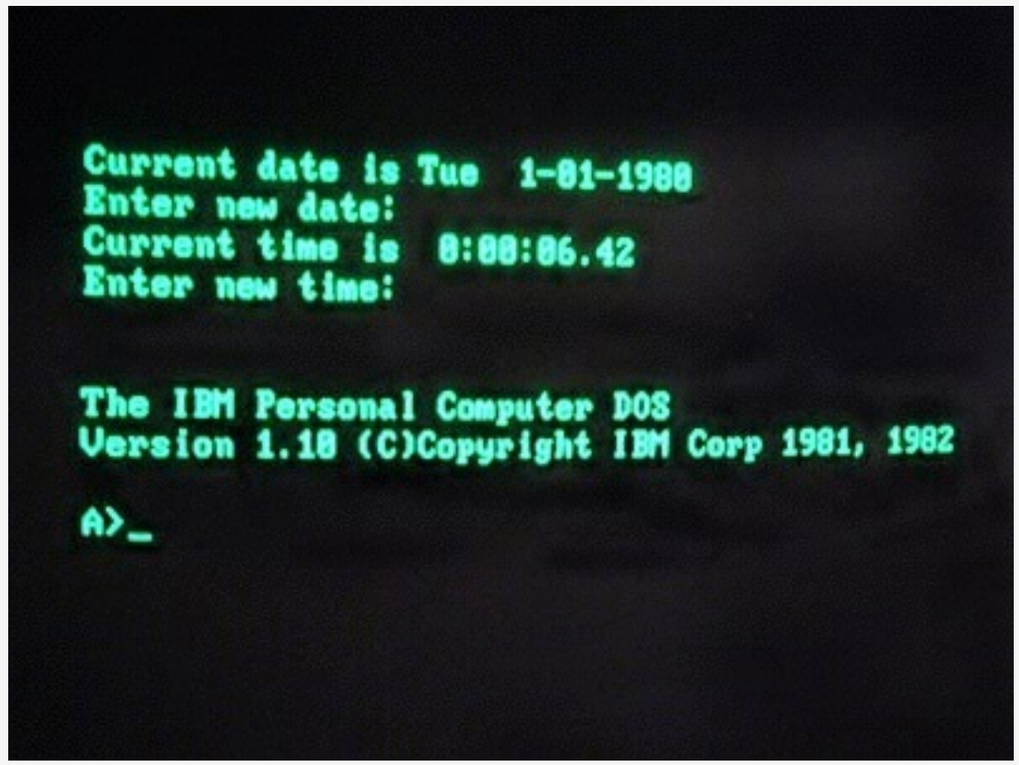
Un mode graphique décrit une configuration de la carte graphique :

- Résolution (taille)
- Nombre de couleurs par pixel
- Fréquence

MODES GRAPHIQUES (1/13)

MDA : Première carte graphique

- Par IBM en 1981
- MDAs: Monochrome Display Adapters
- Mode texte uniquement 80x25 caractères à 50Hz
- Port imprimante



MODES GRAPHIQUES (2/13)

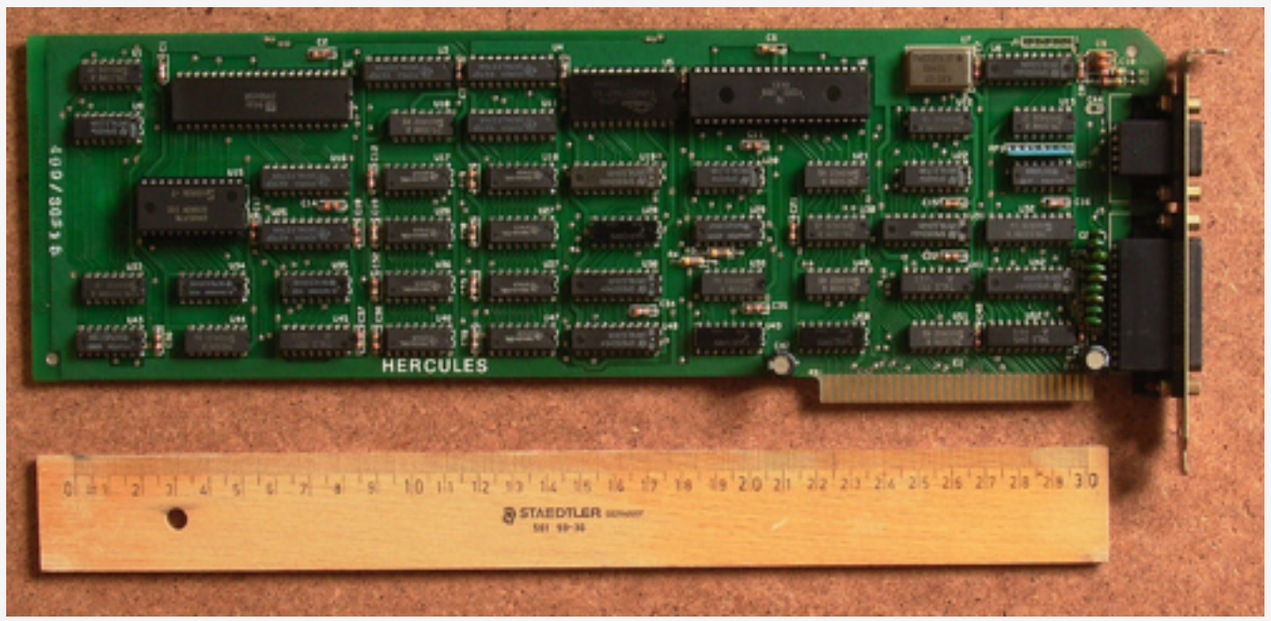
CGA

- En 1981 (avec arrive des Personal Computer)
- Mode Graphique Couleur 320x200 pixels x 4 couleurs
- Manque de résolution pour bureautique
- Essentiellement jeux vidéos

MODES GRAPHIQUES (3/13)

HGC

- Par Hercules (fondé en 1982)
- Hercules Graphics Card (compatible MDAs)
- Mode graphique monochrome 720x348



MODES GRAPHIQUES (4/13)

EGA (Enhanced Graphics Adapter)

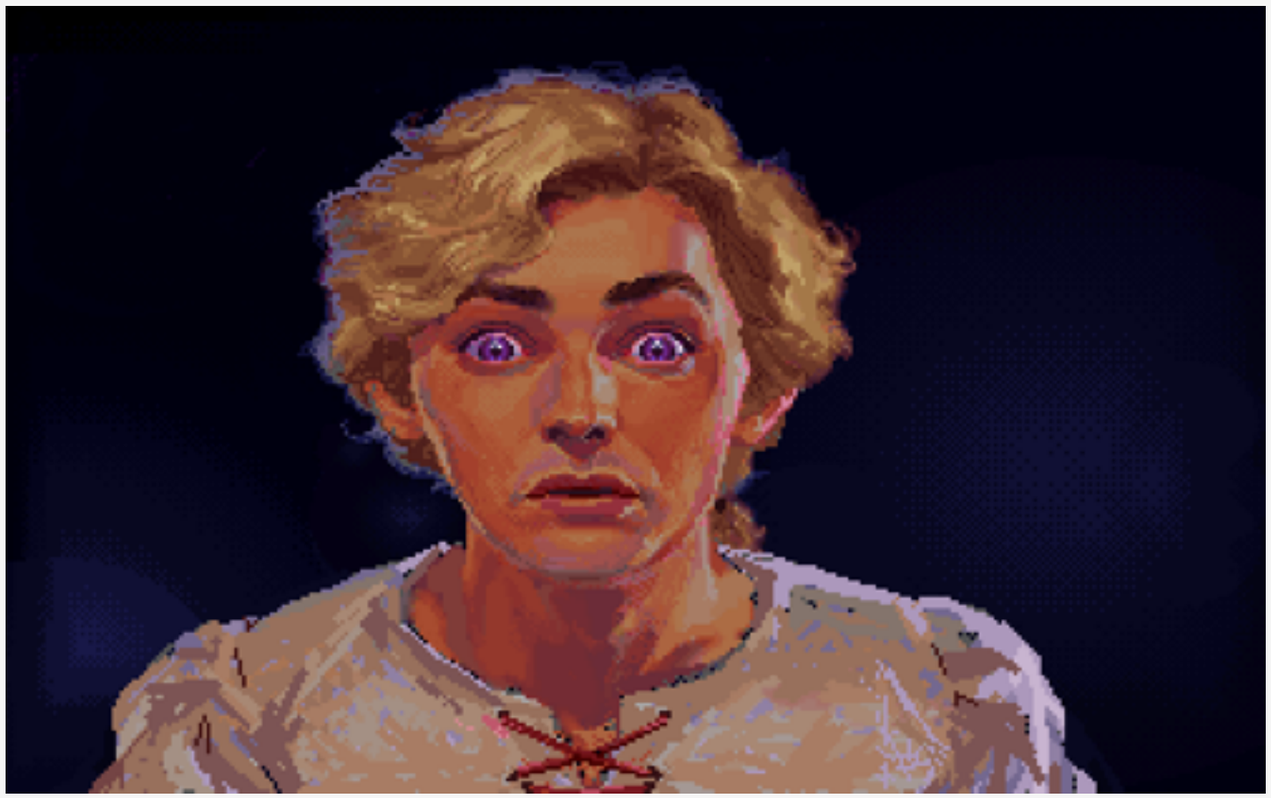
- Par IBM en 1984
- Résolution max : 640x350x16 couleurs
- Palette de 64 couleurs
- Fréquence de 60Hz

MODES GRAPHIQUES (5/13)

VGA (Video Graphic Array)

- Par IBM en 1987
- Résolution 640x480x16 – 720x400 (mode texte)
- Résolution 320x200x256 (mode MCGA)
- Palette de 262.144 couleurs

  6 bits par couleur
- 1er standard vidéo à
- fonctionner en analogique



MODES GRAPHIQUES (6/13)

Amélioration du VGA par IBM

- 8514/A (1987), première carte accélératrice
- 1024x768x256 43.5Hz entrelacé
- 640x480 60Hz non-entrelacé

MODES GRAPHIQUES (7/13)

Amélioration du VGA par IBM

- XGA (eXtended Graphics Array)
- Introduit en 1990
- successeur du 8514/A
- 512Kb/1Mb VRAM
- 1024x768x256 – 640x480x16bits

MODES GRAPHIQUES (8/13)

Amélioration du VGA par IBM

- XGA2 : XGA avec fréquence plus élevées
- 1024x768x16bits ⇒ 65535 couleurs
- 800x600x24bits ⇒16 millions de couleurs

MODES GRAPHIQUES (9/13)

SVGA (SuperVGA)

- Basé sur le VGA avec spécifications du constructeur
- Reconnu standard du mode 800x600
- Manque de standardisation

  ⇒ consortium des principaux constructeurs

- 'Video Electronics Standards Association' (VESA)

  ⇒ interface logiciel standard

MODES GRAPHIQUES (10/13)

Standard : XGA, SXGA, UXGA

- UVGA (UltraVGA) et XGA : standard 1024x768
- SXGA (SuperXGA) : standard 1280x1024
- UXGA (UltraXGA) : standard 1600x1200

MODES GRAPHIQUES (11/13)

Wide screen: WXGA, WSXGA, WUXGA

16/9ème 16/10ème

- WXGA (Wide eXtended Graphics Array)
  1280x800
- WSXGA (Wide Super eXtended Graphics Array)
  1600x1024
- WUXGA (Wide Ultra eXtended Graphics Array)
  1920x1200

MODES GRAPHIQUES (12/13)

Quad format : QXGA, QSXGA, QUXGA

- QXGA (Quad eXtended Graphics Array)

  2048x1536

- QSXGA (Quad Super eXtended Graphics Array)

  2560x2048

- QUXGA (Quad Ultra eXtended Graphics Array)

  3200x2400

MODES GRAPHIQUES (13/13)

4K

- 4K Ultra High Definition Television (UHD)

  3840x2160

- Digital Cinema Initiatives (DCI) 4K

  4096x2160

  4096x1714 (Cinemascope)

  3996x2160 (flat cropped)

RETINA

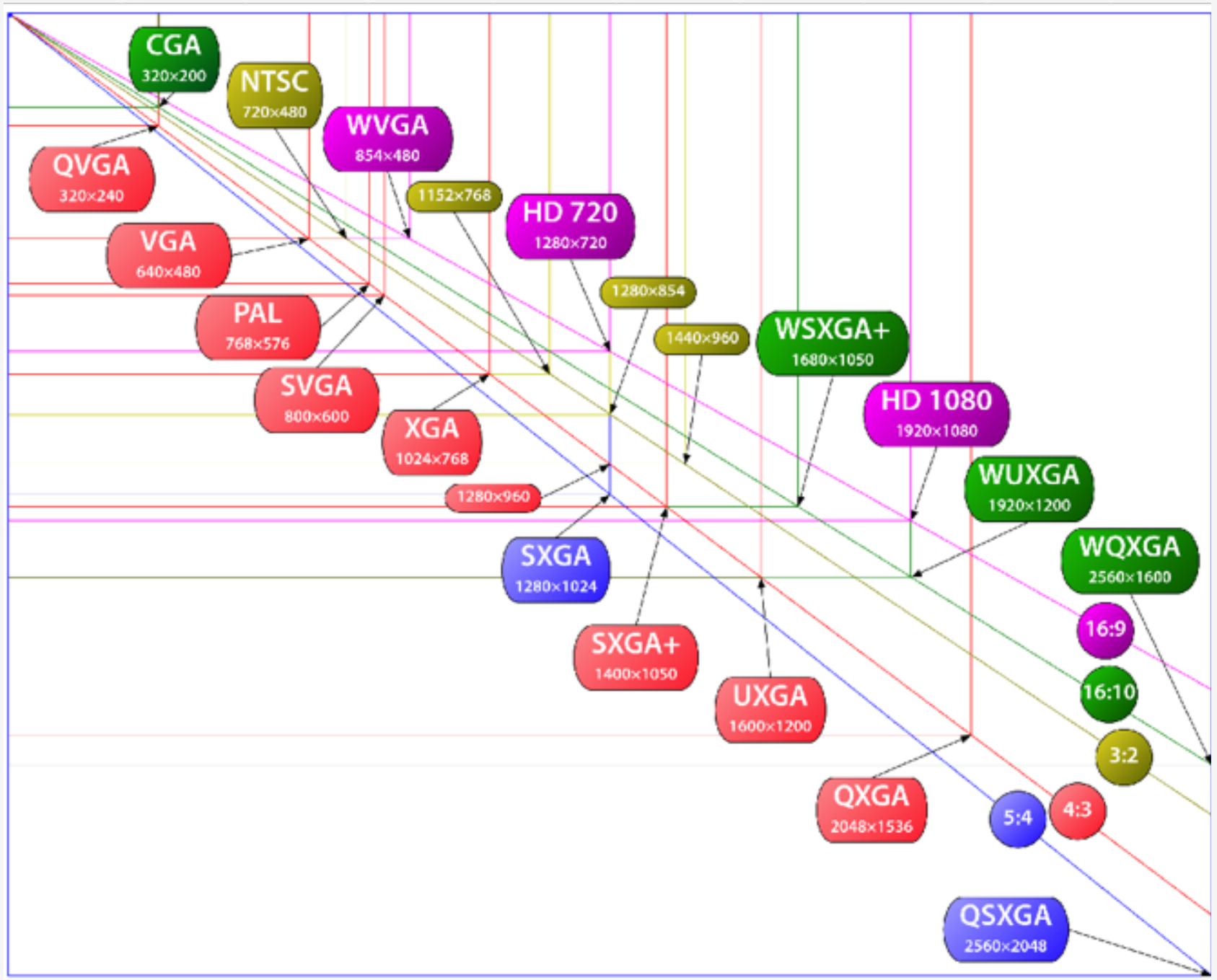
Densité de pixels plus haute que l'œil humain

À une distance donnée

- iPhone 4 : 960x640
- iPhone 5 : 1136x640
- iPad 3/4/Mini : 2048x1536
- MacBook Pro 15" 2880x1800
- MacBook Pro 13" 2560x1600

cf http://en.wikipedia.org/wiki/Retina_Display

MODES GRAPHIQUES



MODES GRAPHIQUES

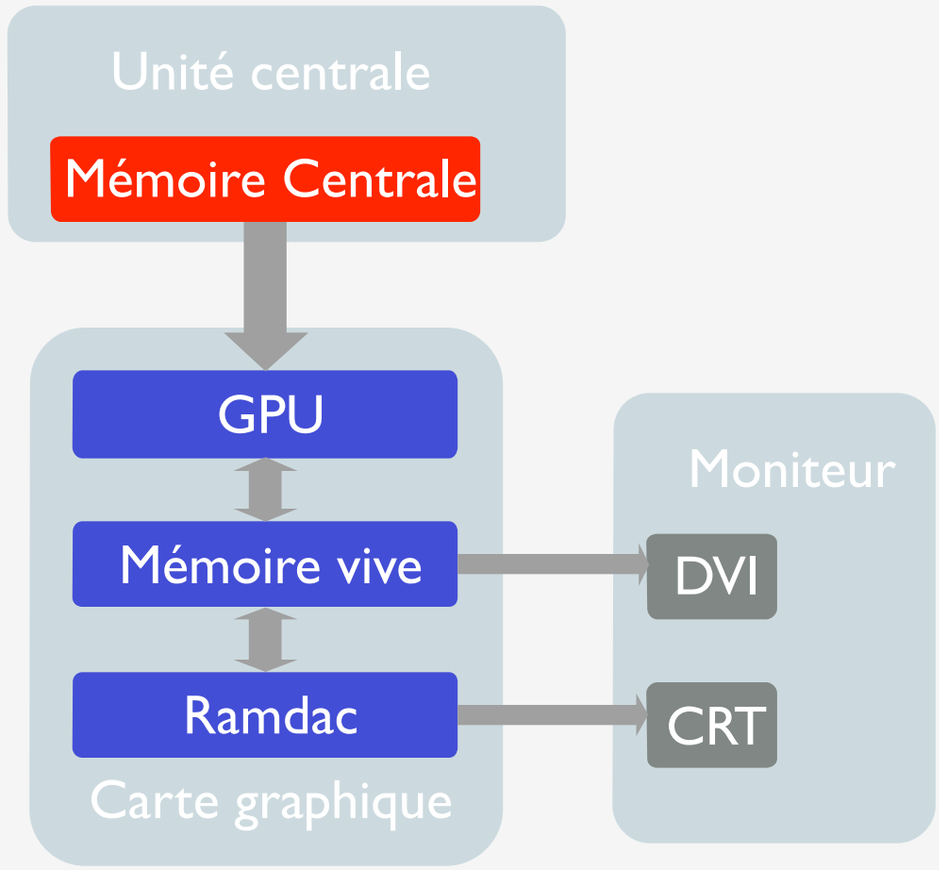
Computer Standard	Resolution	Ratio	Pixels
HXGA	4096×3072	4:3	12.6M
WHXGA	5120×3200	16:10	16.4M
HSXGA	5120×4096	5:4	21M
WHSXGA	6400×4096	25:16	26M
HUXGA	6400×4800	4:3	31M
WHUXGA	7680×4800	16:10	37M

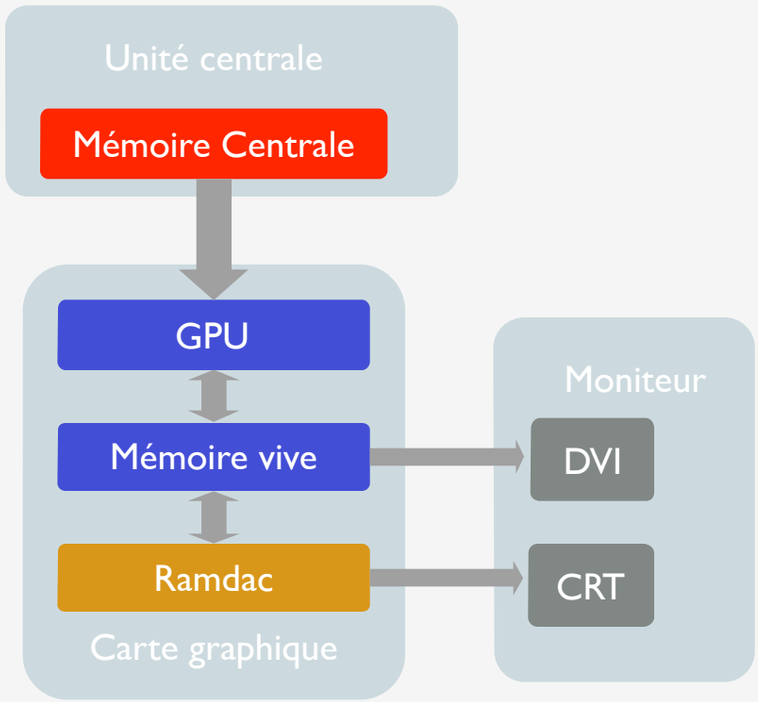
- Aucun moniteur ne supporte ces résolutions
- Les 4K arrivent !

ARCHITECTURE

VUE GLOBALE

- Ramdac
- DVI
- Bus
- Mémoire

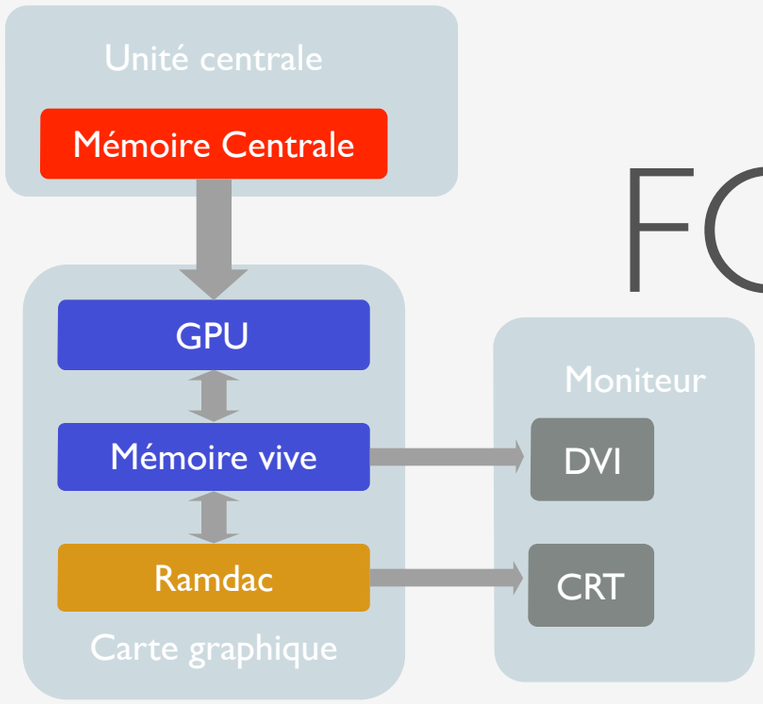




RAMDAC

Random Access Memory Digital to Analogic Converter

- Signal numérique ⇒ signal analogique pour périphérique CRT
- Fréquence interne : fréquence de rafraîchissement du moniteur
- Accès à la mémoire vidéo (framebuffer)
- Mémoire vive propre : stockage des palettes de conversion

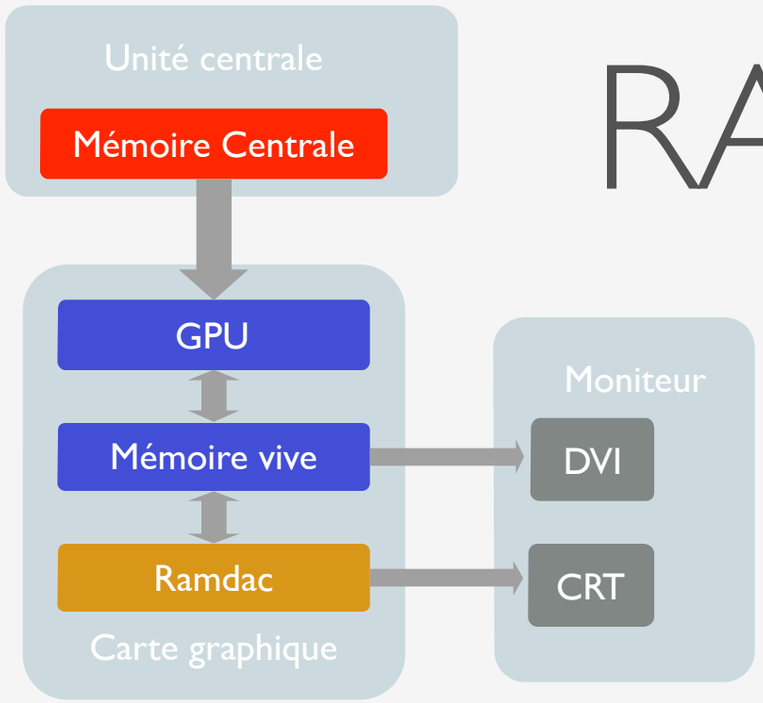


FORMATS RAMDAC

- True Color : 1 octet/composante ⇒ 24 bits
- HiColor : 2 modes

  2x5bits (rouge + bleu) + 6bits (vert) ⇒ 16 bits (65536 couleurs)

  5 bits par composante ⇒ 15 bits (32768 couleurs)
- RGBA True Color : + canal alpha ⇒ 32 bits
- Fausses couleurs : palette indexée (24bits/couleur)

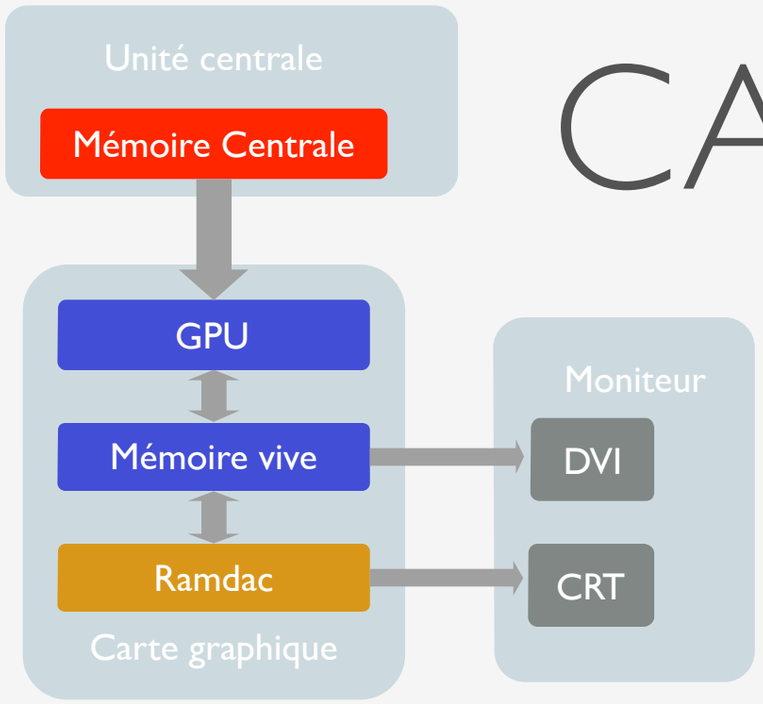


RAFRAÎCHISSEMENT RAMDAC

- Minimum : 50Hz en entrelacé

  entrelacement = affichage des lignes impaires puis paires
- Confort des yeux pour f > 72Hz
- Fréquence interne élevée (MHz)
- Accès mémoire très fréquent

  ⇒ Double buffering

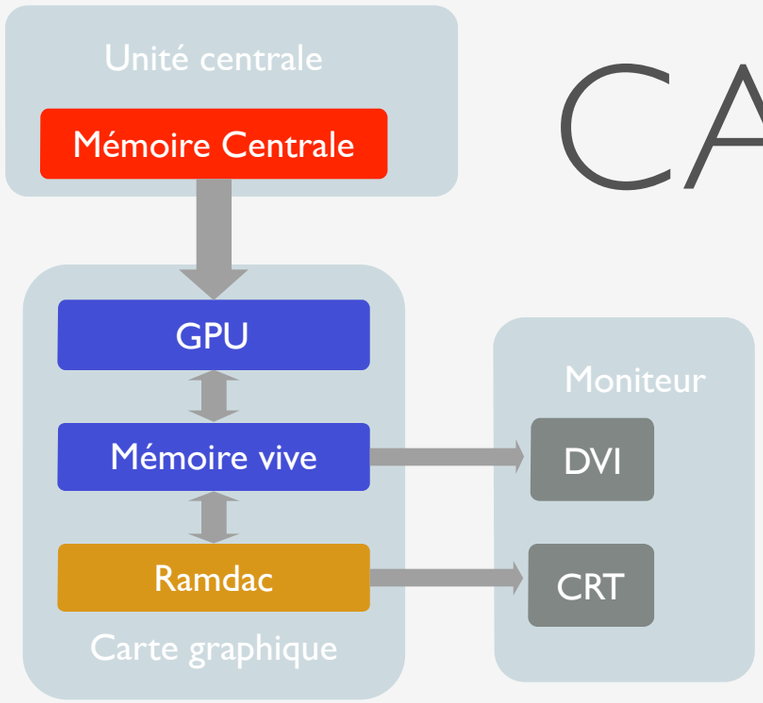


CALCUL THÉORIQUE RAMDAC

- Taille de l'écran : **largeur x hauteur**
- Fréquence visuelle (ex : 75 Hz) : **freq**
- Fréquence interne :

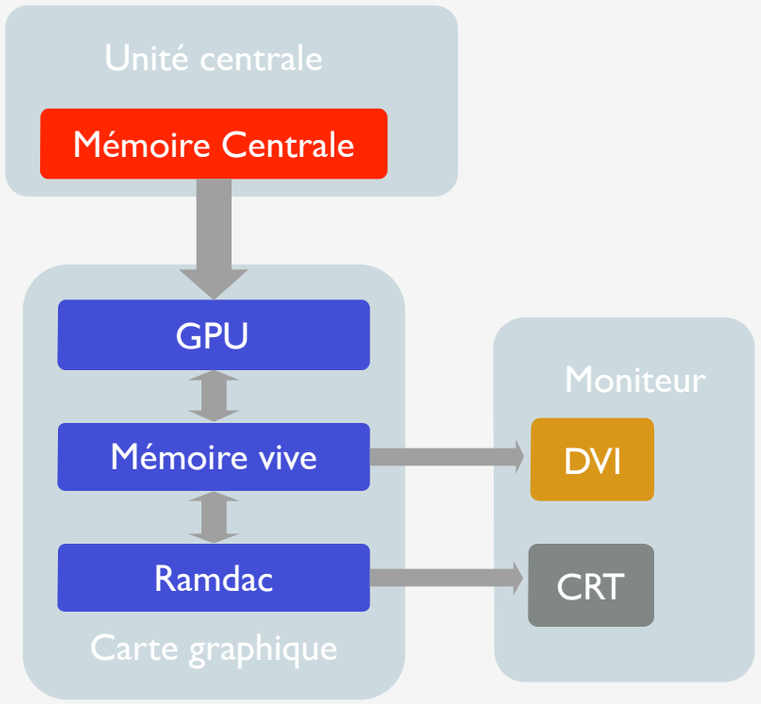
freq. int. = largeur x hauteur x freq x 1.32

- Facteur 1.32 : le canon à électrons perd environ 32% du temps pour les déplacements horizontaux et verticaux



CALCUL THÉORIQUE RAMDAC

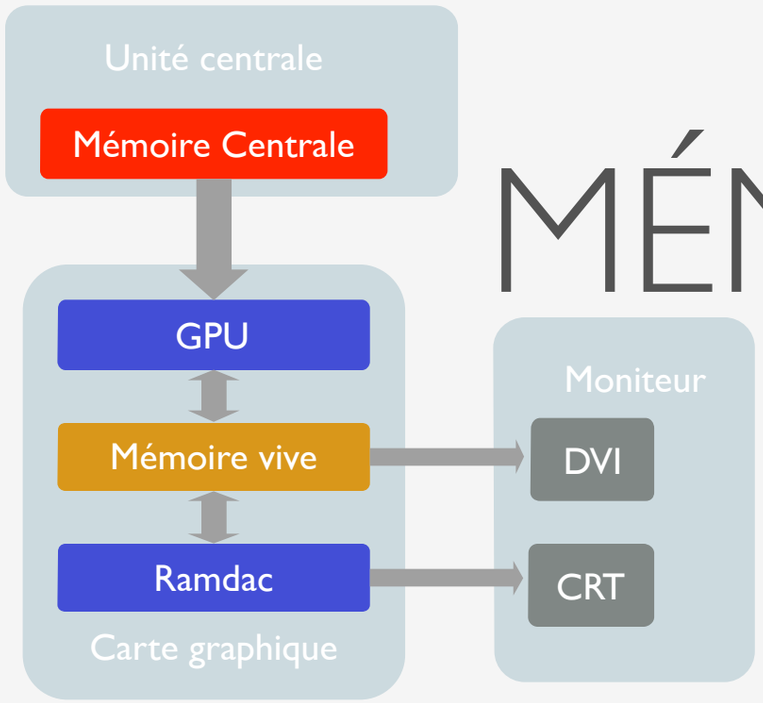
- Radeon 9800 ou GeForce FX 5900 : Bi-Ramdac 400Mhz
- 1600 x 1200 ⇒ 120 Hz – 304MHz
- 2048 x 1536 ⇒ 85 Hz – 352MHz
- Pour résolution de 1600x1200 à 85hz il faut un RAMDAC de 1600 x 1200 x 85 x 1.32 = 215 MHz



DVI

Digital Video Interface

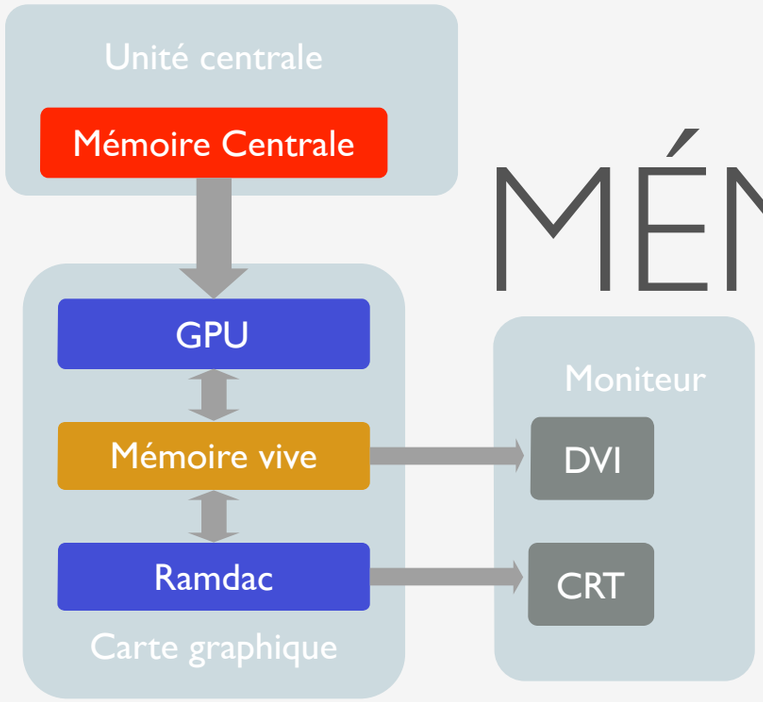
- Spécifié par Digital Display Working Group
- Périphériques numériques et analogiques
- 3 normes existantes :
  - DVI-A : signal analogique
  - DVI-D : signal numérique
  - DVI-I (I = integrated) : signal analogique ou numérique
- périphériques numériques (écran plat, …) ⇒ pas de RAMDAC



MÉMOIRE VIDÉO : TAILLE

- DRAM
- DRAM EDO
- VRAM
- WRAM
- MDRAM
- SDRAM
- SGRAM
- RAMBUS
- DDR-SDRAM

X	Y	bpp	nbCoul	Mémoire	Carte
320	200	1	2	7,8Ko	8Ko
320	200	8	256	64Ko	64Ko
640	480	4	16	150Ko	256Ko
800	600	4	16	235Ko	512Ko
640	480	8	256	300Ko	512Ko
1024	768	4	16	384Ko	512Ko
800	600	8	256	468,7Ko	512Ko
640	480	16	65536	600Ko	1Mo
1280	1024	4	16	640Ko	1Mo
1024	768	8	256	768Ko	1Mo
640	480	24	16M	900Ko	1Mo
800	600	16	65536	937Ko	1Mo
1280	1024	8	256	1,25Mo	2Mo
800	600	24	16M	1,3Mo	2Mo
1024	768	16	65536	1,5Mo	2Mo
1024	768	24	16M	2,25Mo	3Mo



MÉMOIRE VIDÉO : DÉBIT

- Résolution 1024x768, True color, 70Hz :

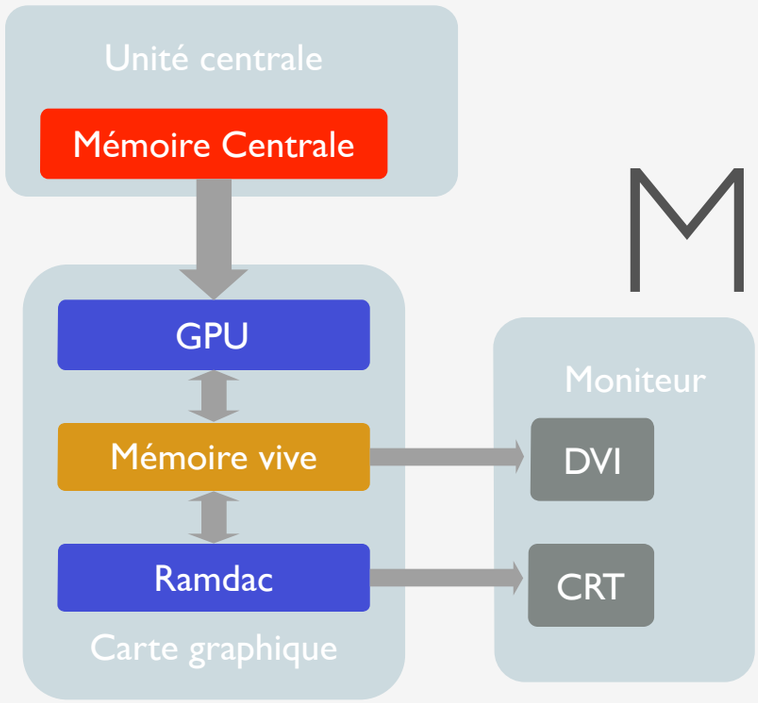
  débit de 1024x768x3x70 = 157 Mo/s

- QSXGA : résolution 2560x2048, True color, 60 Hz :

  débit de 2560x2048x3x60 = 943 Mo/s

- Accès multiple du GPU et du RAMDAC ⇒ latence

- Nombreux types de mémoire : DRAM, VRAM, SDRAM, ...



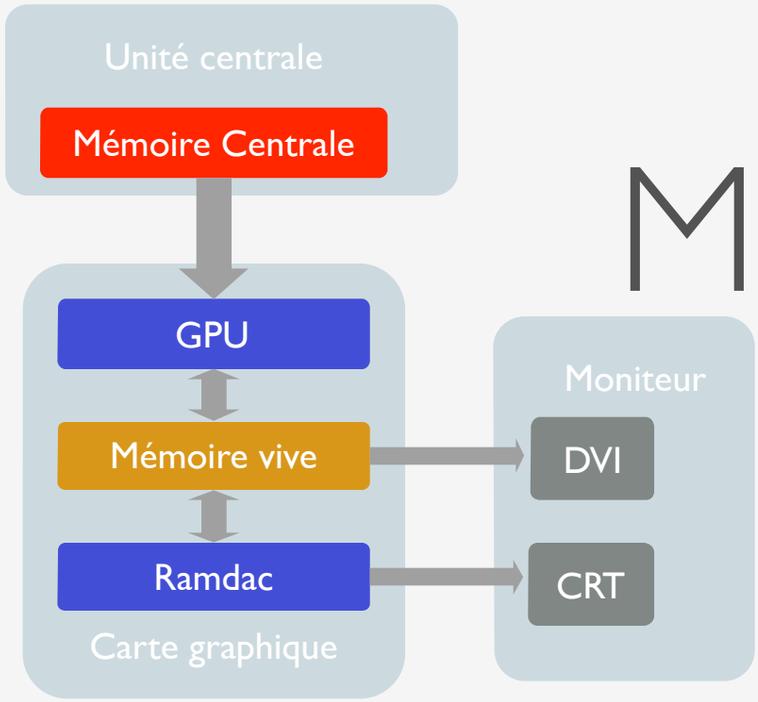
MÉMOIRE VIDÉO

DRAM (Dynamic RAM)

- Type FPM (Fast Page Mode)

  rafraîchit souvent ⇒ latence (+asynchrone) ⇒ accès 70 ns

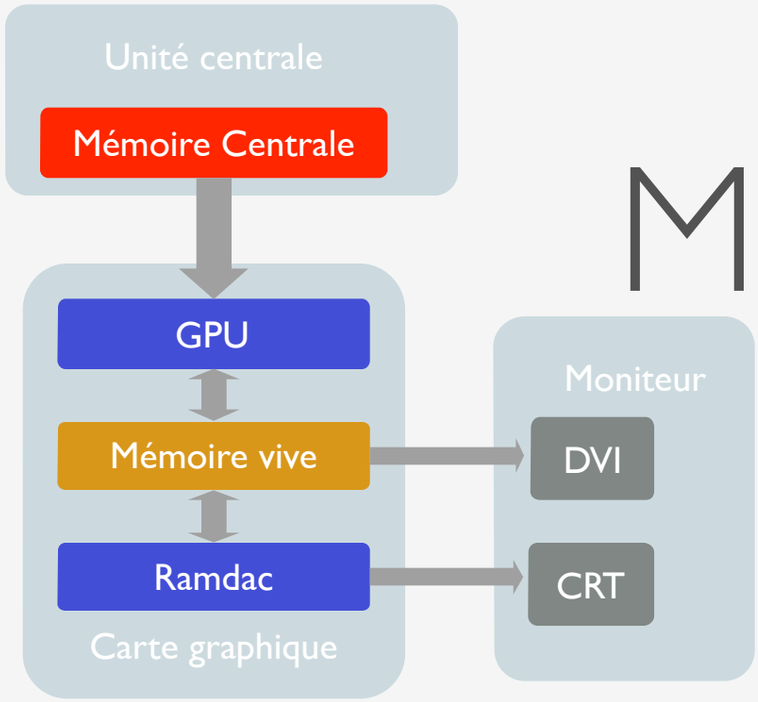
- 225 Mo/s (33Mhz-50MHz)
- Lente et dépassée



MÉMOIRE VIDÉO

DRAM EDO (DRAM Extended Data Out) :

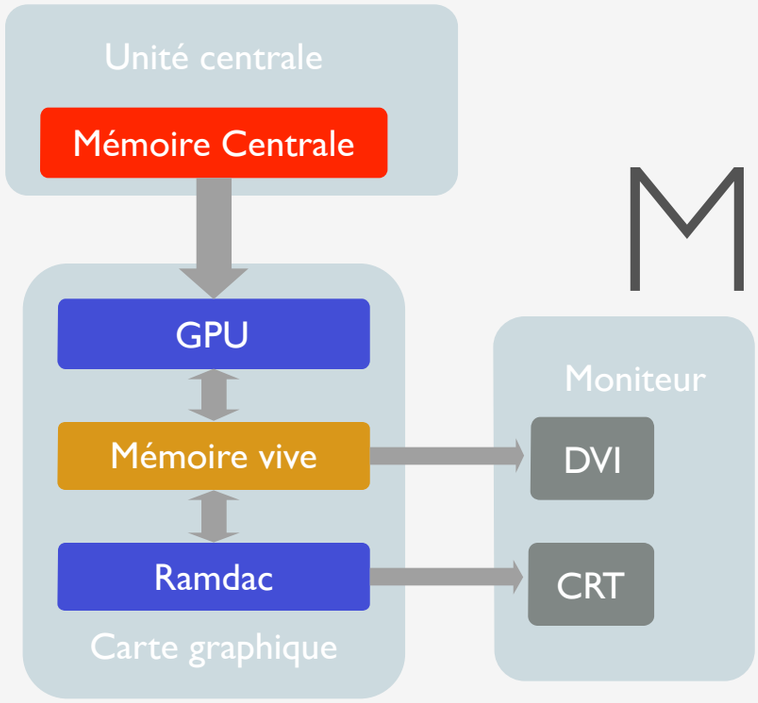
- 400 Mo/s (inadaptée pour f > 66 MHz)
- accès 50-60 ns (grâce à un cache)
- cf 3DFX (Diamond Monster), S3 (Virge 3D)



MÉMOIRE VIDÉO

VRAM (Video RAM)

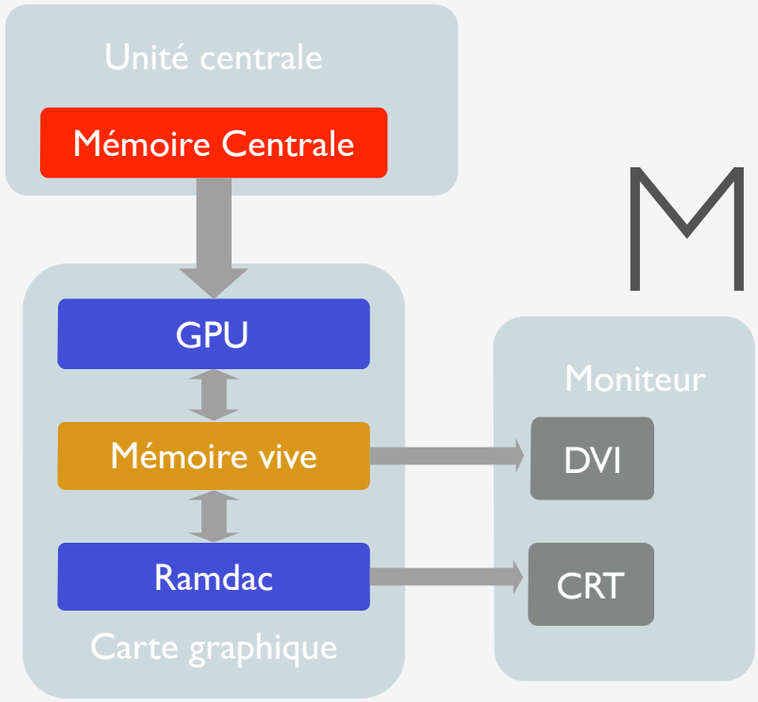
- 625 Mo/s (30% plus chère que DRAM)
- Jusqu'à 80 MHz - temps d'accès 20-25ns
- Dual Port : lecture et écriture en même temps



MÉMOIRE VIDÉO

WRAM (Windows RAM)

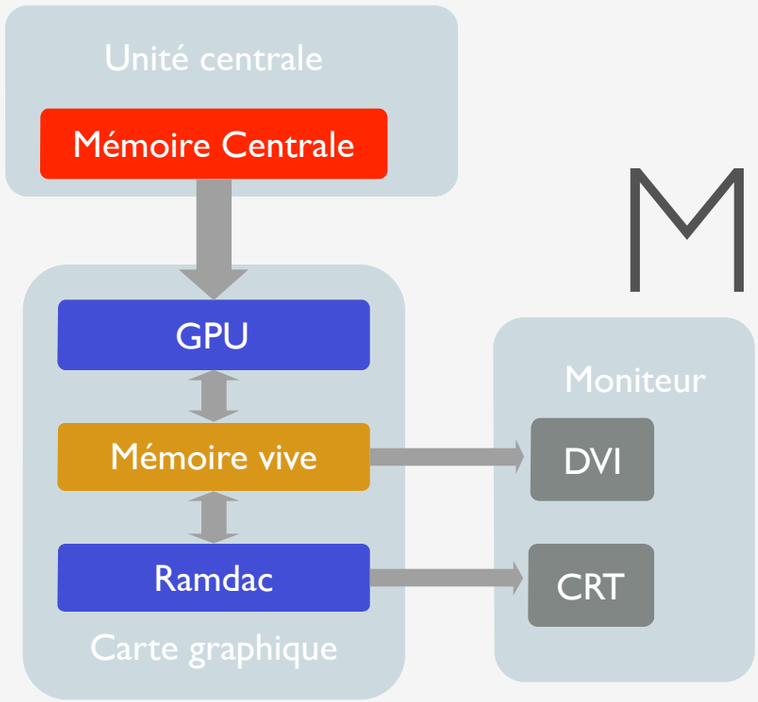
- 960 Mo/s
- VRAM avec circuit 32 bits (accès par blocs)
- Fonctionnalités 2D (transfert de mémoire, …)
- Matrox (Millénium II)



MÉMOIRE VIDÉO

MDRAM (Multi-bank DRAM) :

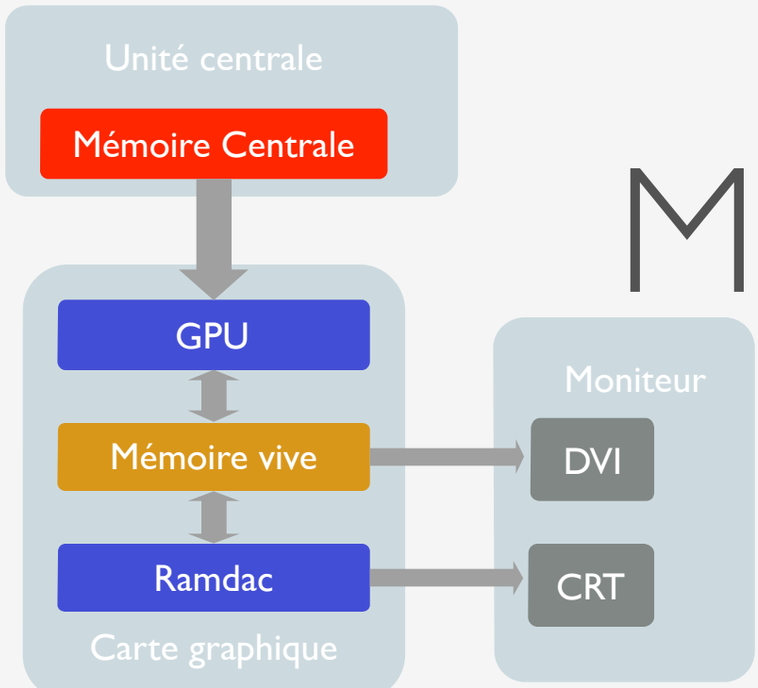
- Accès simultanés
- 500 Mo/s
- Cartes S3, Hercules Dynamite 128, Tseng Labs ET6000



MÉMOIRE VIDÉO

SDRAM (Synchronous RAM)

- apparue en 1997
- synchronisée avec le bus mémoire
- temps d'accès 10-12ns
- Fréquence 66-133Mhz
- 500, 800, 1.060 Mo/s ⇒ 66, 100, 133 Mhz



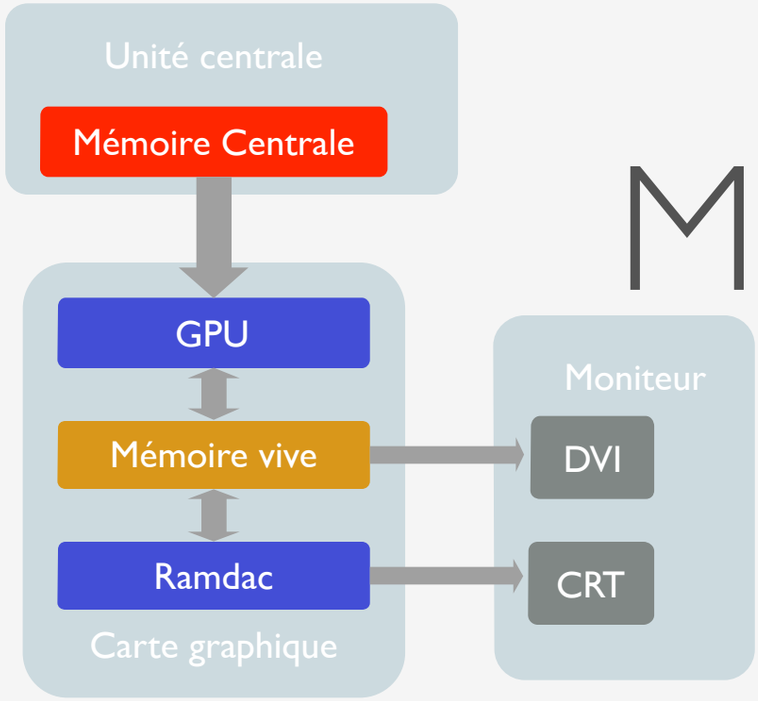
MÉMOIRE VIDÉO

SGDRAM (Synchronous Graphic RAM) :

- SDRAM dédié pour la vidéo
- Single Port
- Synchronisé au bus mémoire (<100MHz)
- Techniques :

  Masked write : modification en 1 cycle

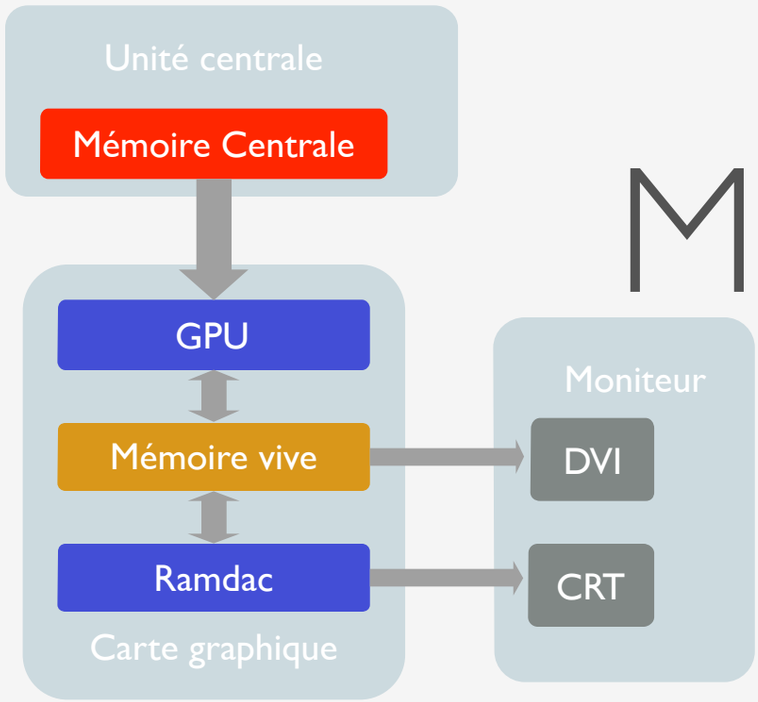
  Block write : récupération/modification par blocs entiers
- Cartes Matrox Mystique



MÉMOIRE VIDÉO

RAMBUS ou RDRAM

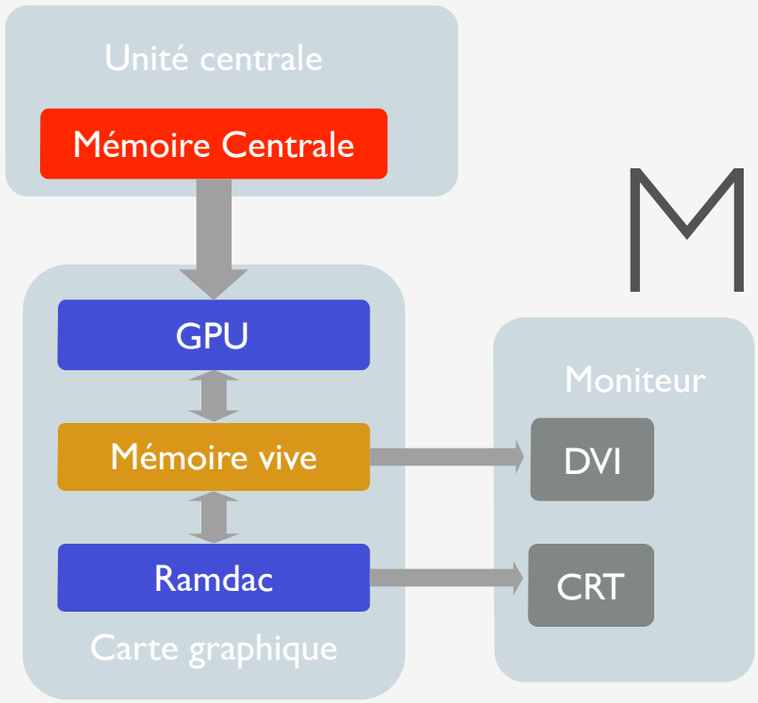
- Soutenue par Intel
- 300 MHz : 1.2 Go/s
- 400 MHz : 1.6 Go/s
- 800 MHz : 6,4 Go/s
- Énorme latence (45ns pour les 400MHz)



MÉMOIRE VIDÉO

DDR-SDRAM (Double Data Rate SDRAM)

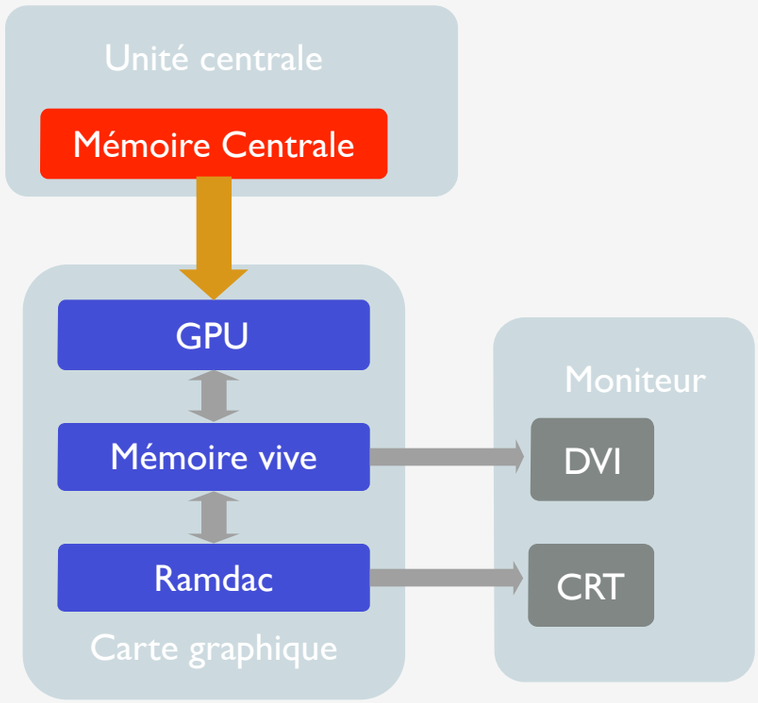
- 2 informations transférées par cycle (front montant/descendant)
- Temps d'accès : 6 ns
- Fréquence de 100 - 600 MHz
- Débit de 1.6 - 4.8 Go/s



MÉMOIRE VIDÉO

DDR-SDRAM (Double Data Rate SDRAM)

- DDR2 ⇒ 313MHz - 10Go/s
- DDR3 ⇒ 375MHz - 24 Go/s
- GDDR3 ⇒ 1.25GHz - 19.9Go/s
- DDR4 ⇒ 400MHz - 25.6Go/s
- GDDR4 ⇒ 1.1MHz - 17.6Go/s
- GDDR5 1.5MHz - 48Go/s



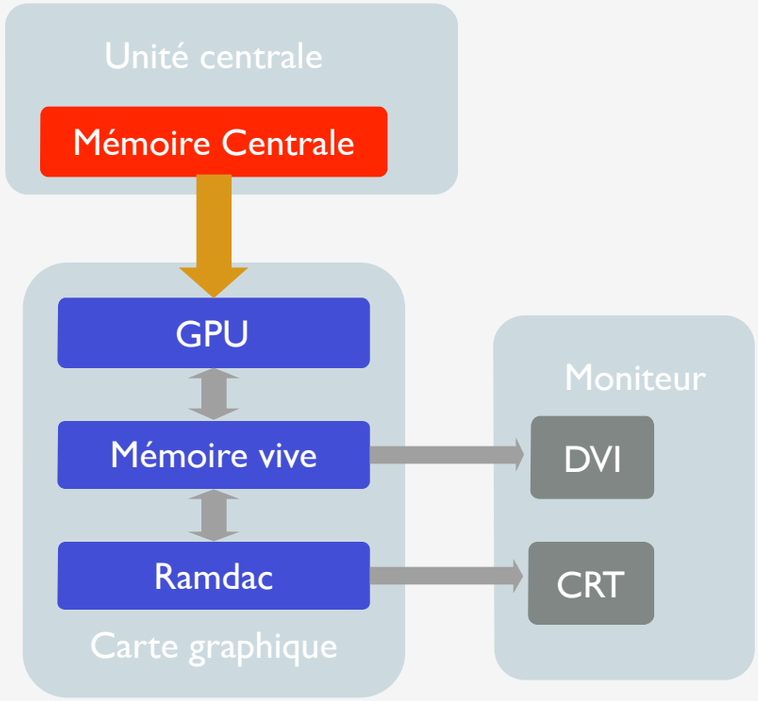
BUS

ISA (Industry Standard Architecture)

- 1981 : 8 bits à 4.7 MHz (8086)
- 1994 : 16 bits à 6 puis 8 MHz (80286)
- Débit maximal de 5.5Mo/s
- Goulot d'étranglement dès l'arrivée du 80386
- Propriétés :

  ⇒ Facilement utilisable et peu coûteux

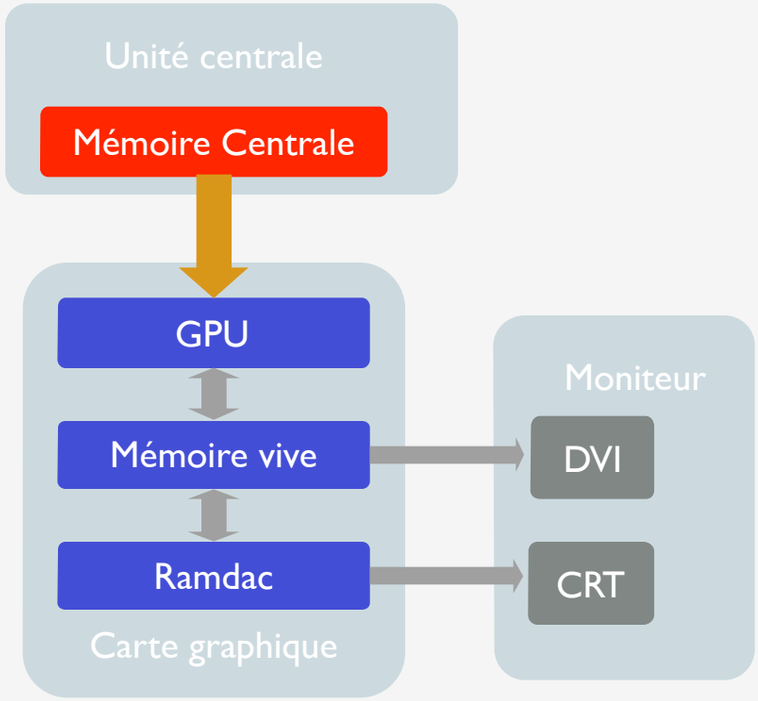
  ⇒ Bus très lent



BUS

MCA (Micro Channel Architecture)

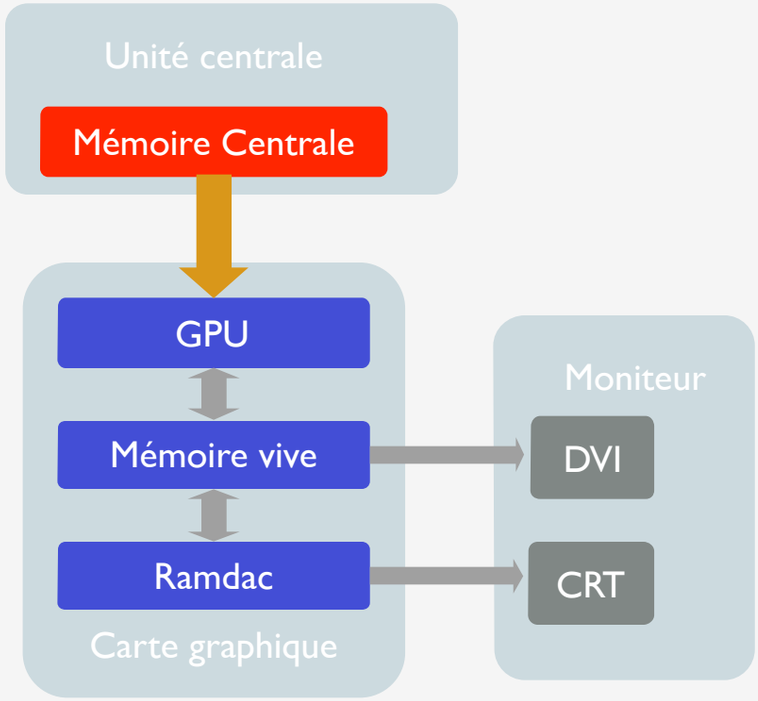
- Développé et utilisé par IBM (avec ses PS/2)
- Incompatibilité avec l'ISA ⇒ monopole d'IBM
- 1er bus à 32bits - 10 Mhz (40 Mo/s)



BUS

EISA (Enhanced ISA)

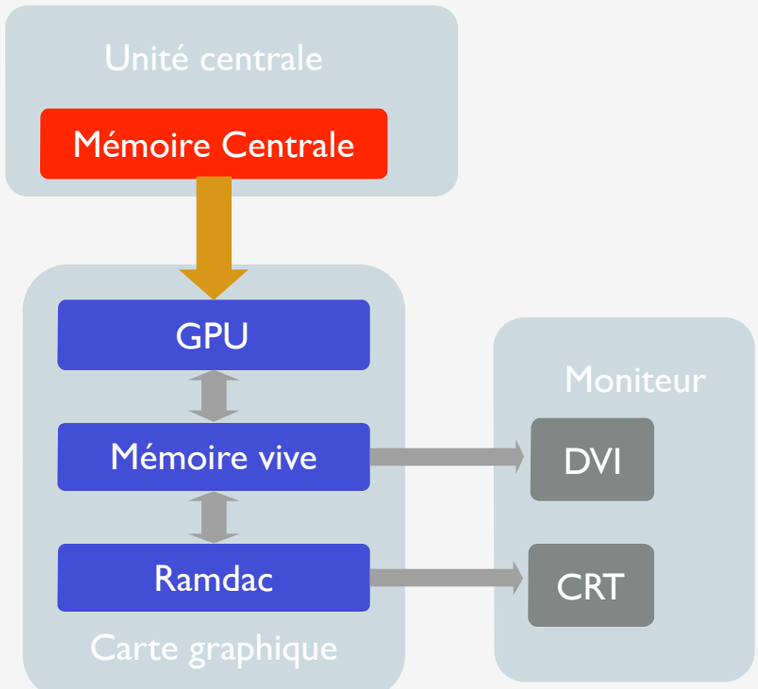
- Sorti en 1988
- Contre-attaque des constructeurs (Compaq, …) face au MCA
- Architecture 32bits, compatible ISA 8/16 bits
- 8.33 Mhz (compatibilité ISA) - 33 Mhz
- Utilisation : applications lourdes (serveurs de fichiers, …)



BUS

VLB (VESA Local Bus)

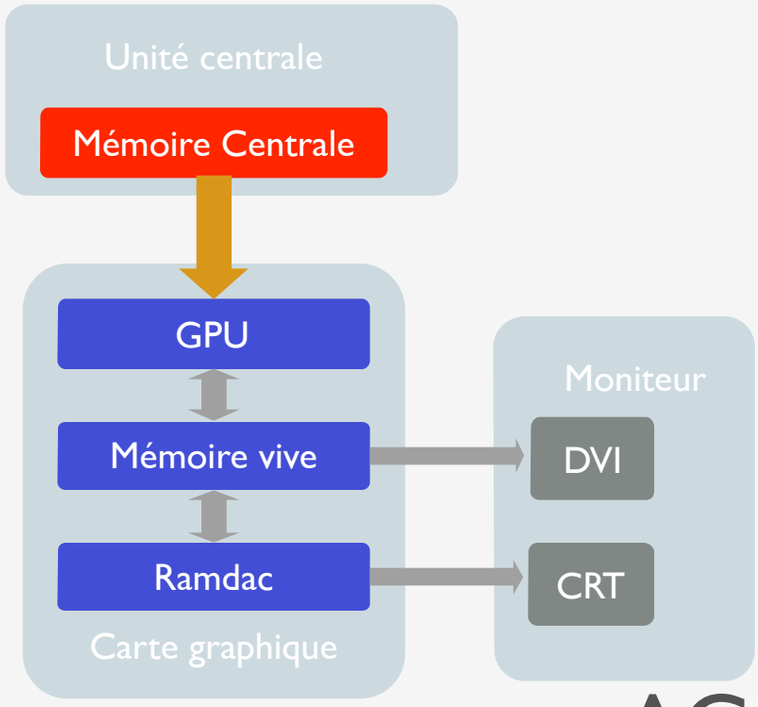
- Sorti en 1992
- Développé par VESA
- Extension du bus local du processeur (port ISA étendu)
- Spécifique aux cartes graphiques
- Bus de 32bits
- Fréquence externe du processeur : 33 à 50 Mhz
- Débit théorique : 130 Mo/s (40Mo/s en pratique)
- VLB 2.0 fonctionne en 64 bits



BUS

PCI (Peripheral Component Interconnect)

- Réponse d'Intel au VLB (1993)
- PCI 1.0 : 32bits - 33MHz ⇒ 132 Mo/s (le plus courant)
- PCI 2.0 : 64 bits (33, 66, 133 MHz) pour serveurs
- Fonctionnalités que ne possède pas le VLB :
  - Bus Master : transfert entre 2 périphériques PCI
  - Look Ahead : anticipation
  - Buffering : meilleur débit
- Incompatible avec ISA ⇒ carte-mère avec slots PCI et ISA
- Devenu un standard… et un goulet d'étranglement !



BUS

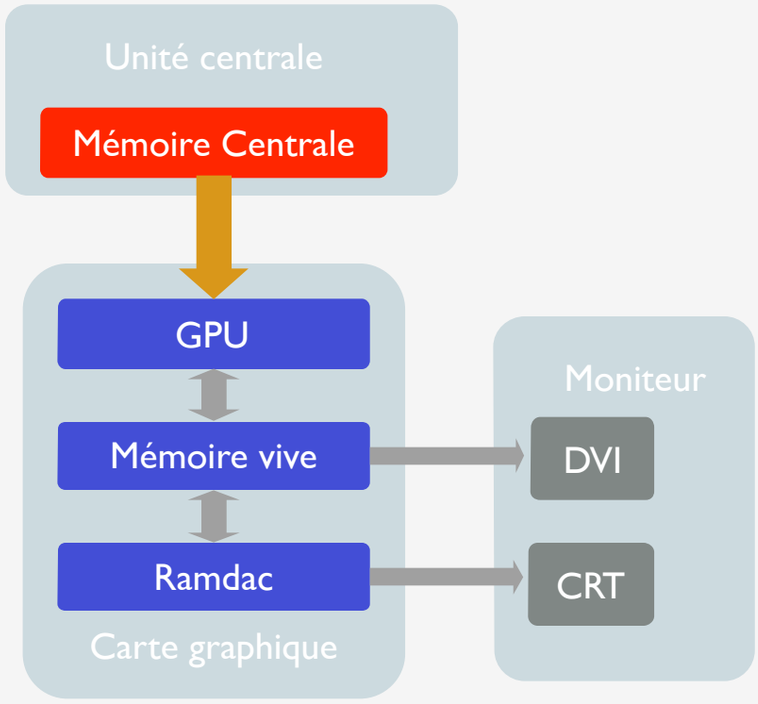
AGP (Accelerated Graphics Port)

- Sorti en 1997
- Bus spécifique aux cartes graphiques / archi 32bits
- Basé sur le PCI 2.1 ⇒ 66 MHz

- Version professionnelle (110 W au lieu de 50W)
- SBA (Side Board Addressing)

  Données chargées sur le bus AGP

  + utilisation du PCI pour signaux de contrôle



BUS

AGP

- Fonctionnalités :

  Bus mastering accès direct à la mémoire

  Pipelining : requêtes en série (≠ PCI avec attente d 'ACK)

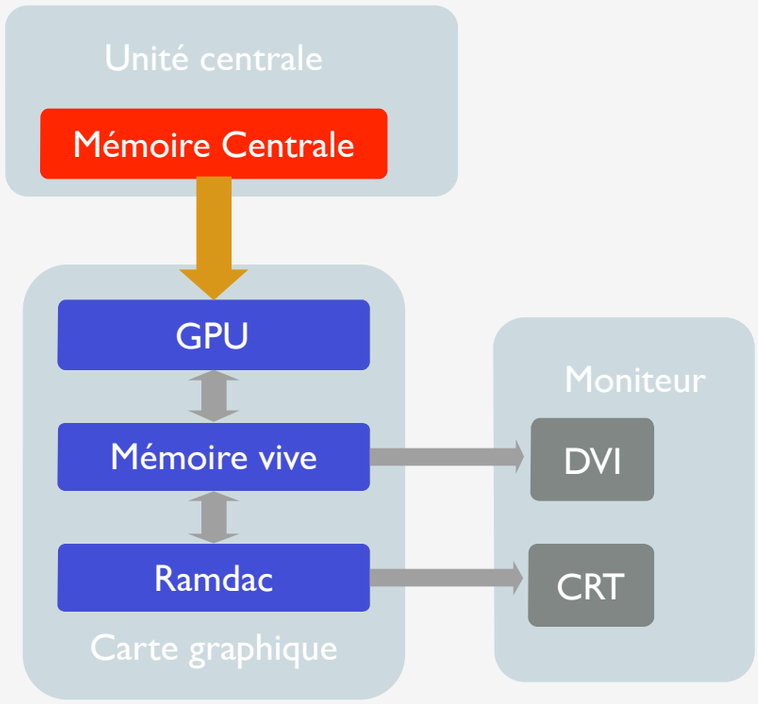
- 2 modes de fonctionnements :

  Mode PIPE (local texturing)

  DMA pour transfert vers mémoire locale
  2 cycles pour afficher

  Mode DIME (Direct Memory Execute):

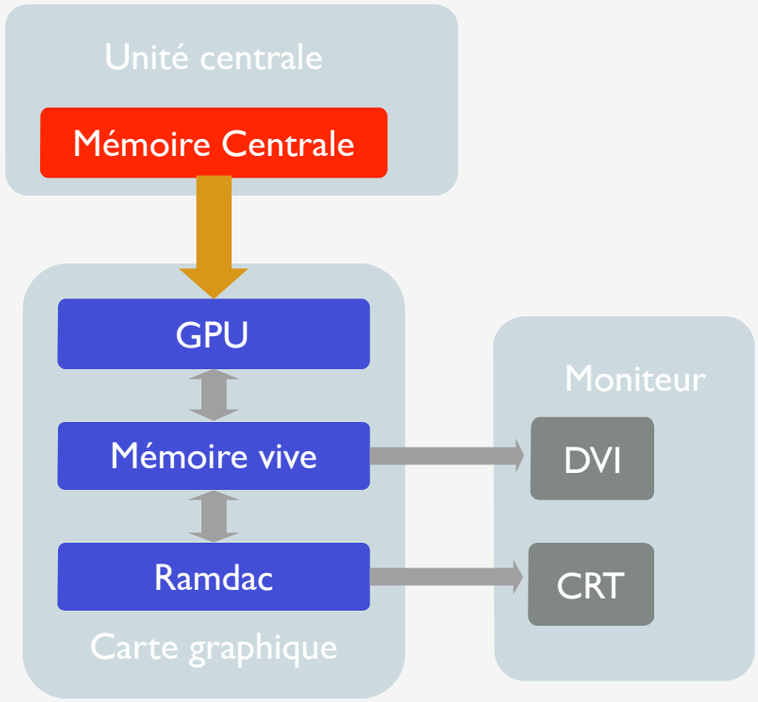
  Traitement des textures en mémoire centrale
  1 cycle pour afficher



BUS

AGP : débits

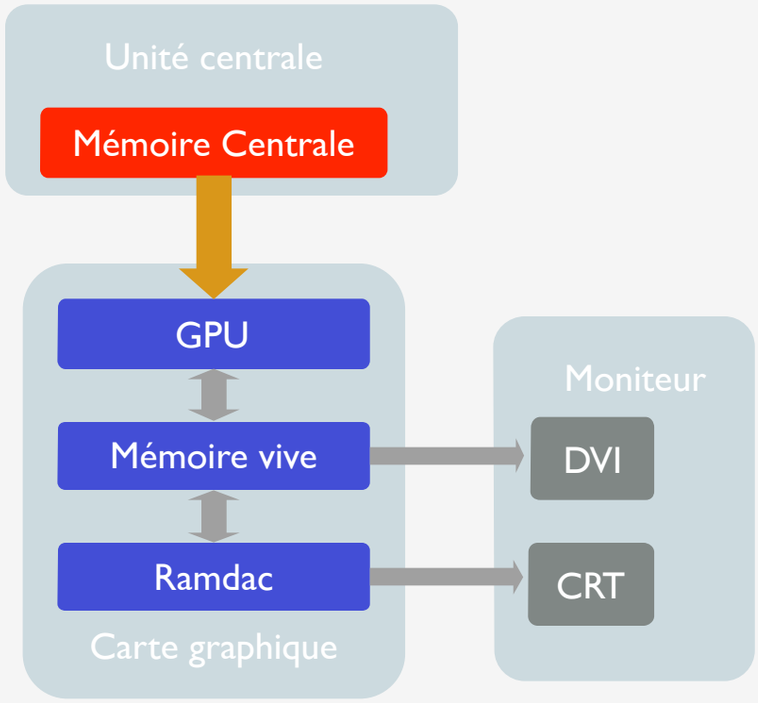
- 1x : 264Mo/s (codage sur les fronts montants)
- 2x : 528Mo/s (codage sur les deux fronts) :
  - PIPE: envoi donnée + contrôle multiplexé…
  - SBA : envoi donnée + contrôle dé-multiplexé
- 4x : 1Go/s (double les infos sur chaque front)
  - Mémoire suffisamment rapide (DDR ou RAMBUS)
  - Fast Write : accès direct mémoire vidéo
- 8x : 2Go/s (quadruple les infos sur chaque front)
  - Gain minuscule de 5% par rapport au 4x



BUS

PCI express (PCIe)

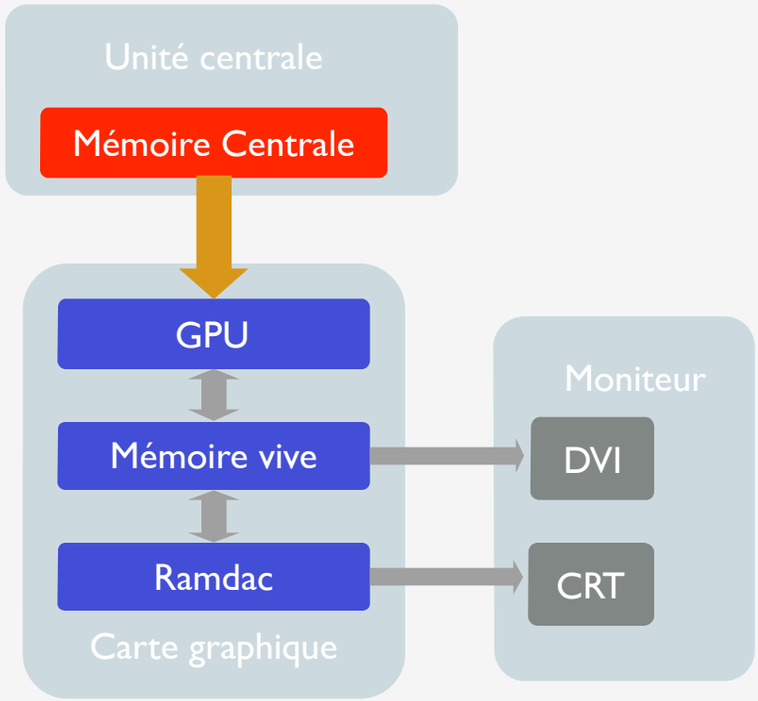
- Bande passante élevée (jusqu'à 8Go/s)
- Qualité de service :
    - Intégrité des données/détection d'erreurs
    - Priorité des données
    - Branchement à chaud (hot plug)
    - Gestion de l'alimentation



BUS

PCI express

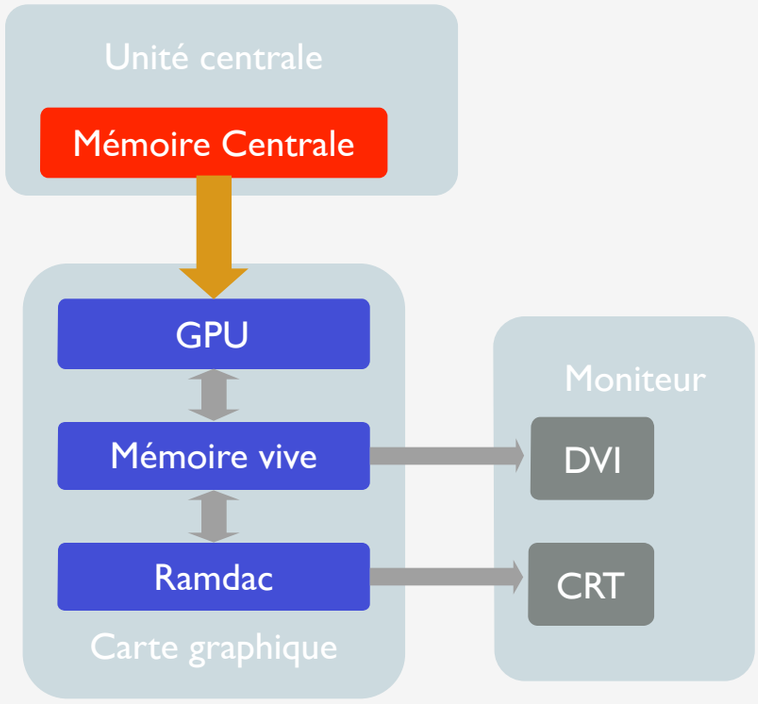
- Interface série décomposée en 3 niveaux :
  - Couche physique
  - Couche de données
  - Couche de transactions



BUS

PCI express : couche physique

- Bus constitué de 2 paires de signaux
  - Émission
  - Réception
- Bande passante peut-être augmentée linéairement
  - ajout de paires de lignes (x1 ⇒ x32)

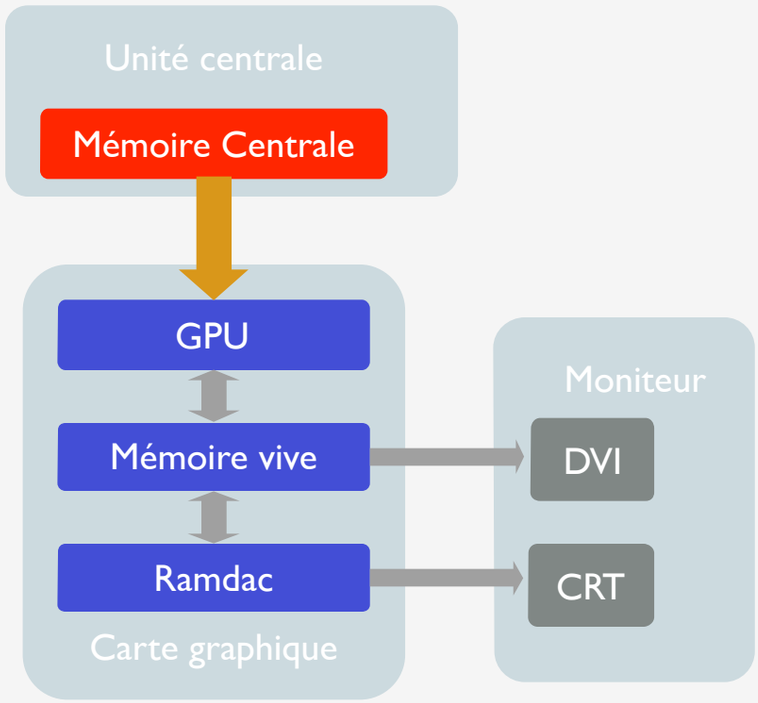


BUS

PCI express : couche de données

- Responsable de l'intégrité des données
- Protocole de contrôle de flux

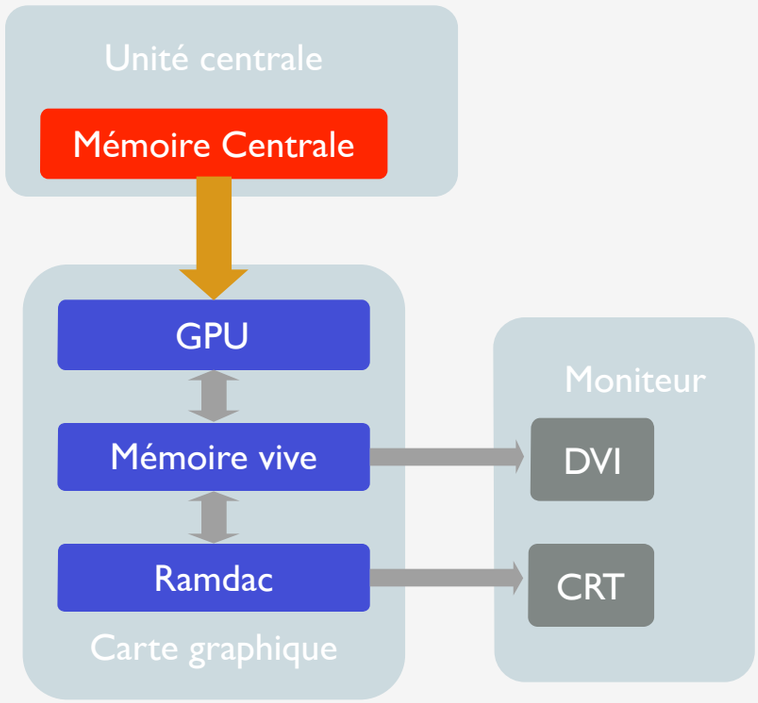
Assurer que le paquet n'est émis que lorsqu'un buffer de réception est disponible



BUS

PCI express : couche de transactions

- Paquets supportent adressage de 32bits / 64bits étendu
- Paquets peuvent avoir des attributs (priorité,…)



BUS

PCI express : encapsulation

3. Couche Physique

ACCÉLÉRATEUR GRAPHIQUE

ACCÉLÉRATEUR GRAPHIQUE (GPU)

- Accélérateur 2D

- Accélérateur 3D / Pipeline de rendu

- Les shaders : vers un pipeline programmable

GPU - ACCÉLÉRATEUR 2D

- Cartes à coprocesseur graphique (carte programmable)
- Début 90

    TIGA (TMS34020)

    P9000 (Carte Diamond Viper)

- Dépassé ensuite par les cartes accélératrices
- Principe d'architecture graphique programmable

    réutilisé + tard en 3D

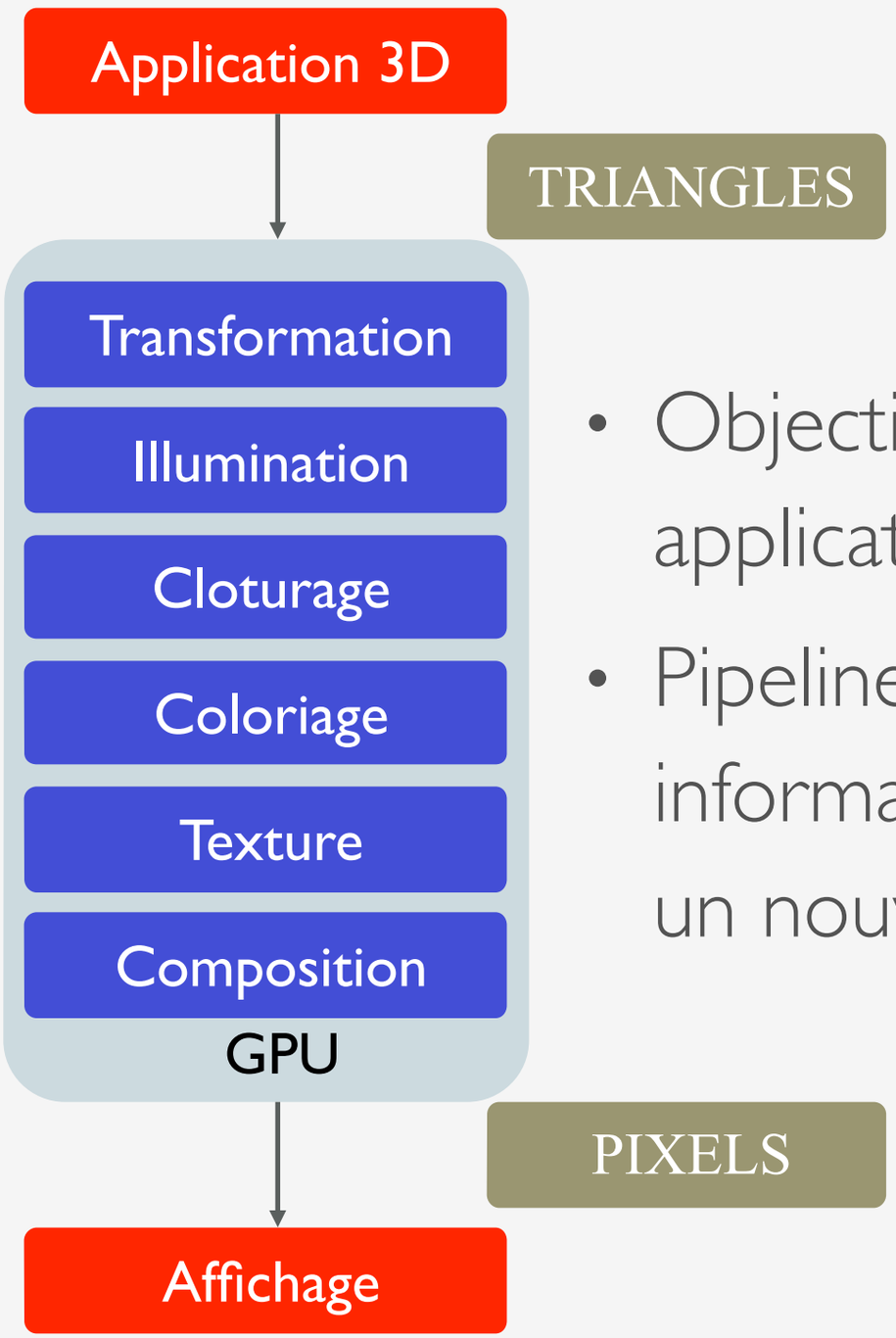
GPU - ACCÉLÉRATEUR 2D

- Accélération des traitements 2D

  ligne, rectangle, curseur souris, overlay, bitBLT

- Décompression MPEG2 (DVD)
- Tuner TV
- Acquisition vidéo
- Cartes 2D seules n'existent plus
- Carte 3D pure (3DFX voodoo)
- Maintenant, carte 2D/3D

GPU - PIPELINE DE RENDU



- Objectif : transformer les triangles 3D d'une application en pixels à l'écran
- Pipeline : succession d'étages qui utilisent les informations de l'étage précédent pour produire un nouveau résultat

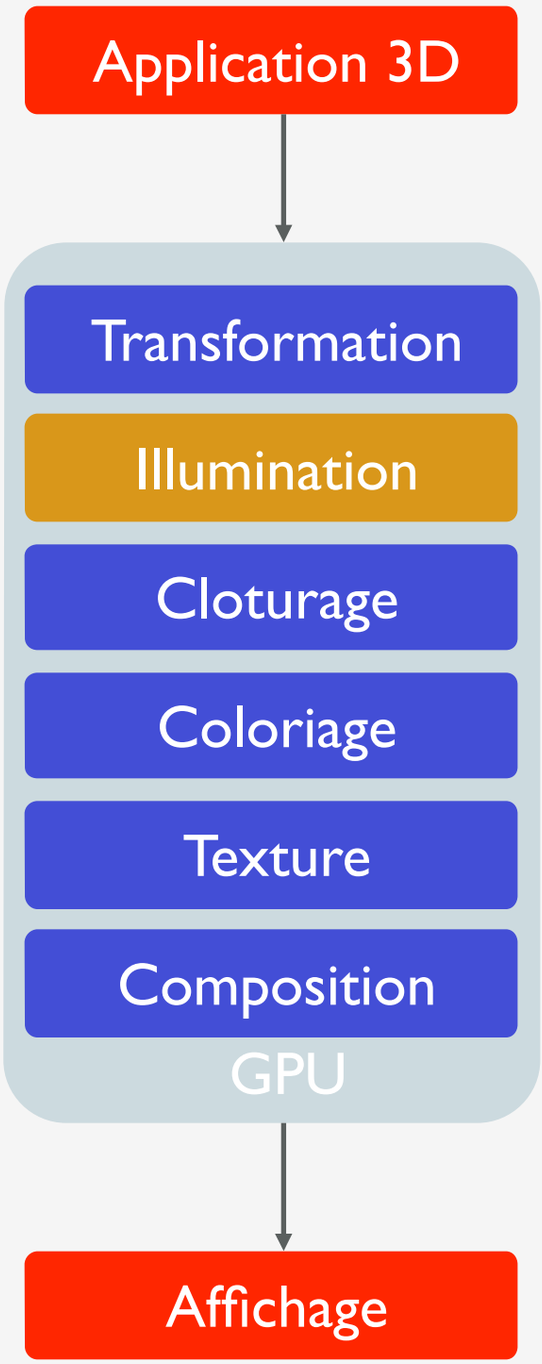
GPU - PIPELINE DE RENDU

Transformation

placer l'objet dans la scène / à la caméra

- opérations matricielles 3D
- translation, rotation, mise à l'échelle, projections

GPU - PIPELINE DE RENDU



Illumination = modèle de Phong

- Éclairement ambiant

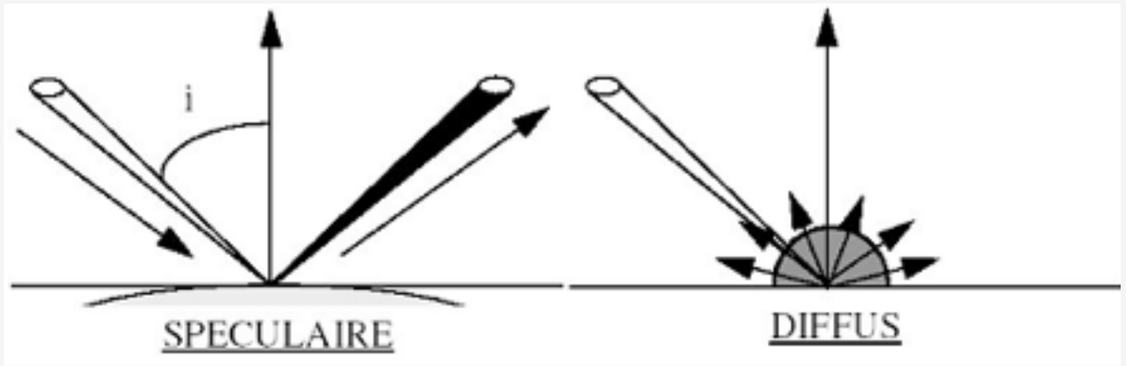
  propriété générale de l'environnement

- Éclairement diffus

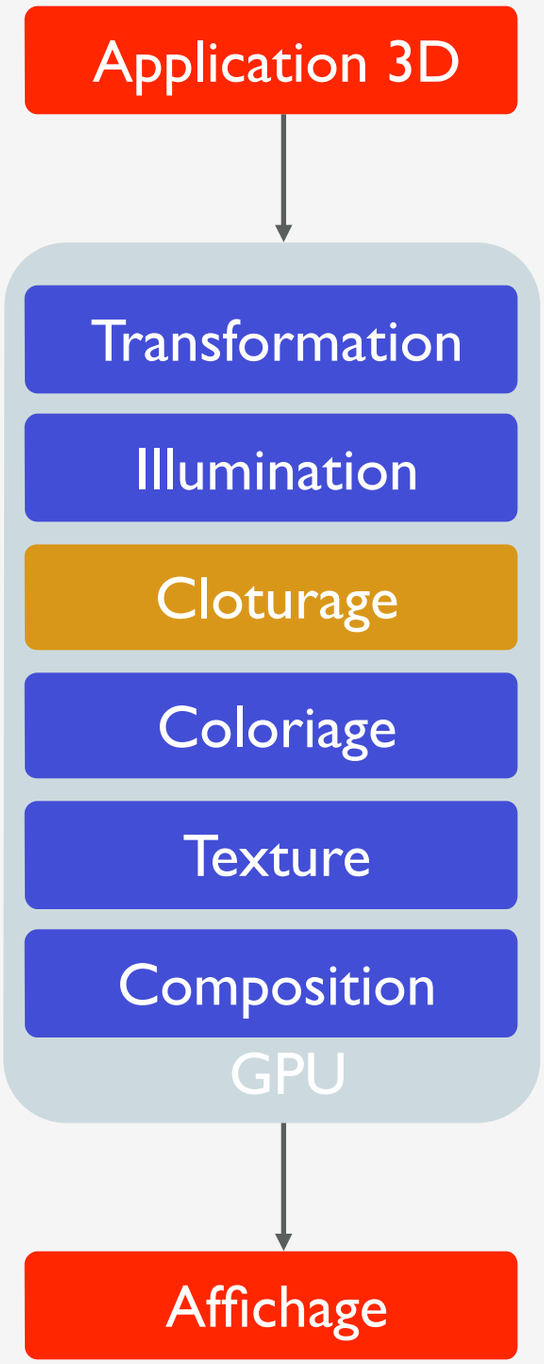
  propriété du matériau

- Éclairement spéculaire

  reflets



GPU - PIPELINE DE RENDU



Cloturage

Calculer le triangle qui sera affiché à l'écran

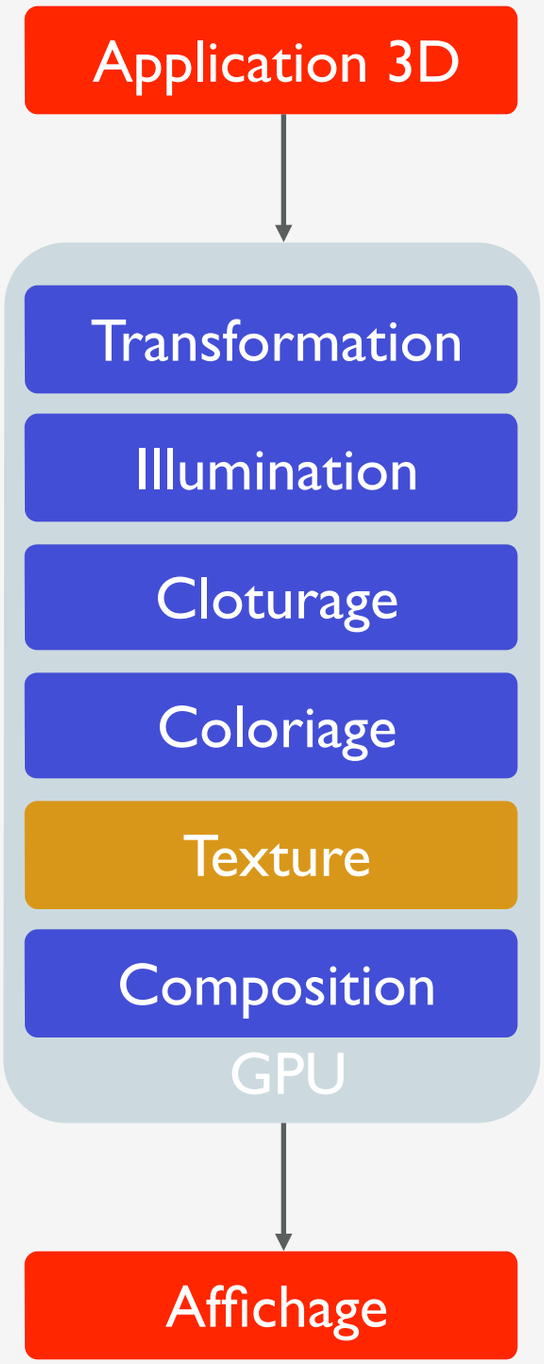
- Culling : orientation de la face
- Clipping : découpe dans le volume de vision
- Projection sur l'écran

GPU - PIPELINE DE RENDU

Coloriage

- Interpolation linéaire des attributs des sommets

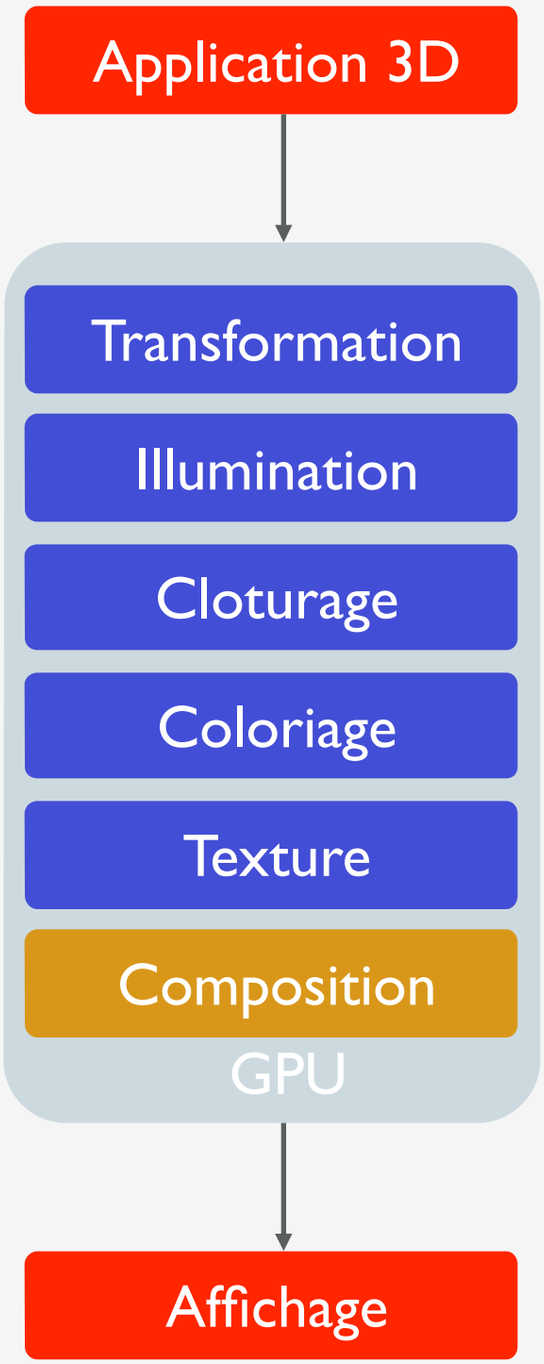
GPU - PIPELINE DE RENDU



Texture

- Combinaison
  - Couleur
  - Éclairement
  - Motif de la texture
- Filtrage des textures (bi-, tri-linéaire)
- Possibilité d'utiliser plusieurs textures
  - Mip-map
  - Light-map
  - Bump mapping

GPU - PIPELINE DE RENDU



Composition (rasterization)

- Alpha test
- Stencil test
- Z test
- Alpha blending

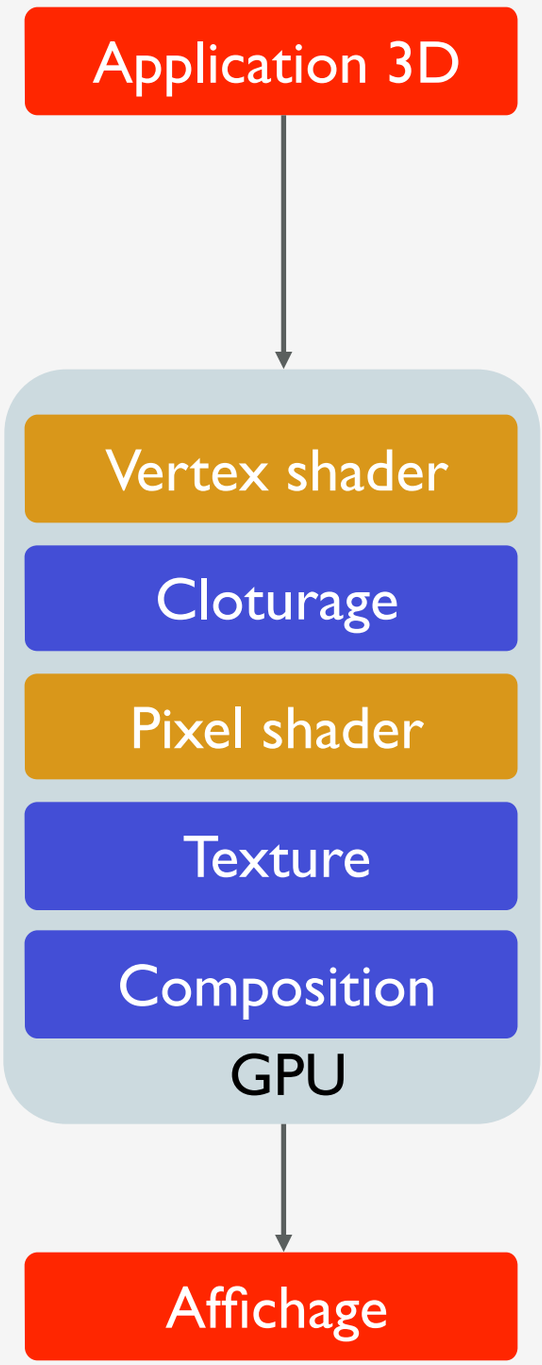
DOUBLE BUFFER

- Utilisation de deux buffers
  - Le premier sert à l'affichage (lecture) pendant que les informations sont écrites dans le second

STÉRÉOVISION

- Présenter des images différentes à chaque oeil
- Utilisation de 2 doubles buffers (2 pipelines)

PIPELINE PROGRAMMABLE: VERTEX ET PIXEL SHADERS



- Shaders : programmes qui peuvent remplacer des phases du pipeline de rendu.
- Vertex shader
  - Transformation des sommets
  - Éclairement/coordonnées de texture aux sommets
- Pixel shader
  - Attribution de la couleur
  - Coordonnées de texture aux fragments

LANGAGE SHADER

- Langage de haut niveau
  - Cg (nVidia, interfaçable avec Direct 3D et OpenGL)
  - HLSL (direct3D)
  - GLSL (OpenGL)
- Ce sont des programmes : le source doit être chargé en mémoire et compilé (grâce à des instructions OpenGL).
- Choix : GLSL (OpenGL 2.0).
- Syntaxe très similaire au C.

VERTEX SHADER

- Entrée : caractéristiques des sommets : position, couleur, coordonnée de texture, etc.
- Exemple : le vertex shader connait la position et la couleur avec le code suivant :

```
glBegin(GL_POLYGON);
glColor3f(0.2, 0.4, 0.6);
glVertex3f(-1.0,1.0,2.0);
…
glEnd();
```

- Dans un vertex shader, on peut faire ses propres calculs de transformation (PROJECTION et MODELVIEW disponibles), calculs de coordonnées de texture (on n'est plus limité au LINEAR_OBJECT, SPHERE_MAP,…), calculs d'éclairement (substitution à Phong).

PIPELINE

- Problème : toutes les opérations par défaut ne sont plus faites (substitution totale).

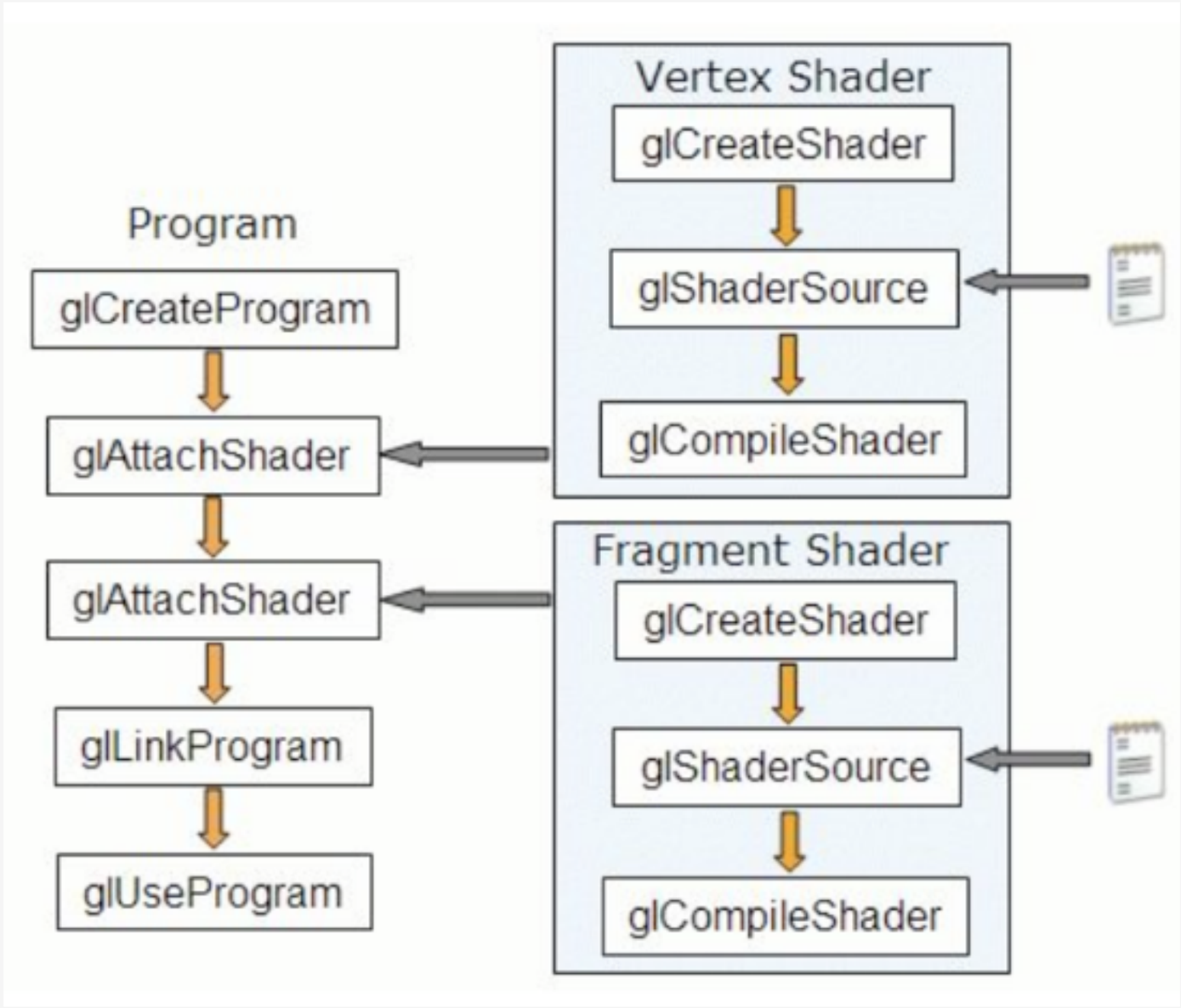
  Par exemple, si vous voulez simplement changer la génération des coordonnées de texture, il vous faut programmer les transformations, le calcul d'éclairement, ...

- Pas de connaissance des autres sommets.
- Cependant : accès aux états d'OpenGL (material, texture, …).

PIXEL SHADER

- Entrée les valeurs interpolées des sommets : position, couleur, coordonnée de texture, depth.
- Écrit la couleur du fragment courant.
- On peut y calculer ses propres couleurs, coordonnées de texture, normales (permet de faire un éclairement par pixel).
- Pas de connaissance des autres fragments (pixels voisins par exemple).
- Accès aux états d'OpenGL.
- Remarque : on manipule bien un fragment et non un pixel (par exemple, un pixel shader qui déplacerait un pixel n'a aucun sens).
- Pas de blending (opération faite après l'exécution du pixel shader).

INTÉGRATION DES PIXELS DANS OPENGL



EXEMPLE

```
void myShader() {
    GLchar *v_source,*f_source;
    GLuint v,f,p;
  // creation handle :
    v = glCreateShader(GL_VERTEX_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);
    // lecture du GLSL source :
    v_source = textFileRead("exemple_vertex.vert");
    f_source = textFileRead("exemple_pixel.frag");
  // affectation des sources aux handles
    glShaderSource(v, 1, (const char **)&v_source,NULL);
    glShaderSource(f, 1, (const char **)&f_source,NULL);
    // compilation des sources
    glCompileShader(v);
    glCompileShader(f);
  // mise en place dans le pipeline :
    p = glCreateProgram();
    glAttachShader(p,v);
    glAttachShader(p,f);
    glLinkProgram(p);

    glUseProgram(p); // si p=0, pipeline classique
}
```

GLSL (GL SHADING LANGUAGE)

EXEMPLE: VERTEX

- Fichier exemple_vertex.vert:

```
void main {
  gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

- On se contente ici de faire ce qui est fait par défaut par le pipeline OpenGL :

Pproj = PROJECTION * MODELVIEW * PLOCAL

- Les gl_ sont des "variables" GLSL prédéfinies
- gl_Position doit obligatoirement être affecté lors d'un vertex shader
- Types de qualifications de ces variables pré-définies :
  uniform mat4 gl_ModelViewmatrix;
  uniform mat4 gl_ProjectionMatrix;
  attribute vect4 gl_Vertex;

TYPES ET QUALIFICATIONS

- mat4 type matrice (16 floats); vect4 type vecteur (4 floats)
- Exemple :

```
void main() {
  vect4 a=vect4(0.1,0.2,0.3,0.4);
  float b=a.x;
  vect2 x=a.xy;
  float d=a.w;
  mat2 e=mat2(1.0,0.0,1.0,0.0);
  float e=mat2[1][1];
  vect2 g=e[1]; //2e colonne
}
```

- uniform : constant dans une primitive (`glBegin(...) ... glEnd()`). Lecture seule
- attribute : peut changer entre chaque sommet. Lecture seule (sauf spécial).
- varying : valeur interpolée (généralement en écriture pour vertex, et lecture pour pixel).

EXEMPLE: PIXEL

- Exemple fichier: exemple_pixel.frag

```
void main() {
  gl_FragColor = vec4(0.4, 0.4, 0.8, 1.0);
}
```

- La variable gl_FragColor doit être obligatoirement affectée dans un pixel shader.
- De type vec4, de qualification varying.

INTERAGIR AVEC UN SHADER

- Les shaders peuvent lire (et non écrire) des variables globales passées par OpenGL
- vertex shader GLSL :

```
void main(void) {
vec4 v = vec4(gl_Vertex);
   v.z = v.z+sin(5.0*sqrt(v.x*v.x+v.y*v.y) + time*0.5)*0.25;
   gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

MODIFICATION DE TIME

programme OpenGL :

```
int location_time; // handle pour la variable time
                   // du shader
float mon_temps; // a faire varier dans myIdle par exemple
...
void myShader () {
  ...
  glLinkProgram(p);
  glUseProgram(p);
  location_time = glGetUniformLocation(p,"time");
}
...
void myDisplay(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glUniform1f(location_time,mon_temps);
  glutSolidTeapot(1);

  glutSwapBuffers();

  mon_temps+=0.01;
}
```

UN EXEMPLE: ÉCLAIREMENT PAR VERTEX

RENDRE LE SPÉCULAIRE

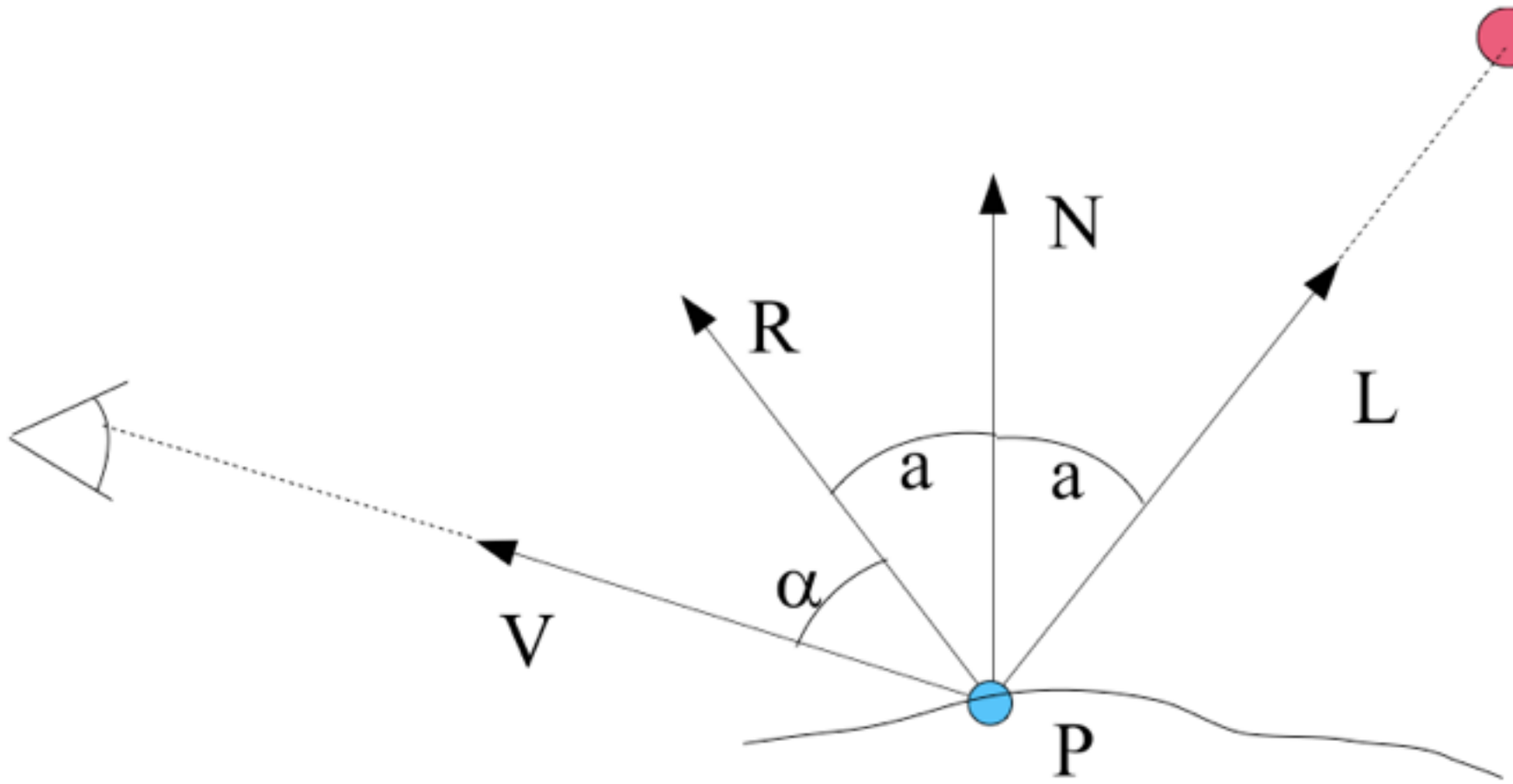
- Eclairement par défaut d'OpenGL

  Calcul d'éclairement en chacun des sommets
  puis interpolation des couleurs pour chacun des pixels

- Intérêt avec le pixel shader :

  Provoquer un éclairement par pixel
  pour nuancer le mauvais rendu du spéculaire d'OpenGL.

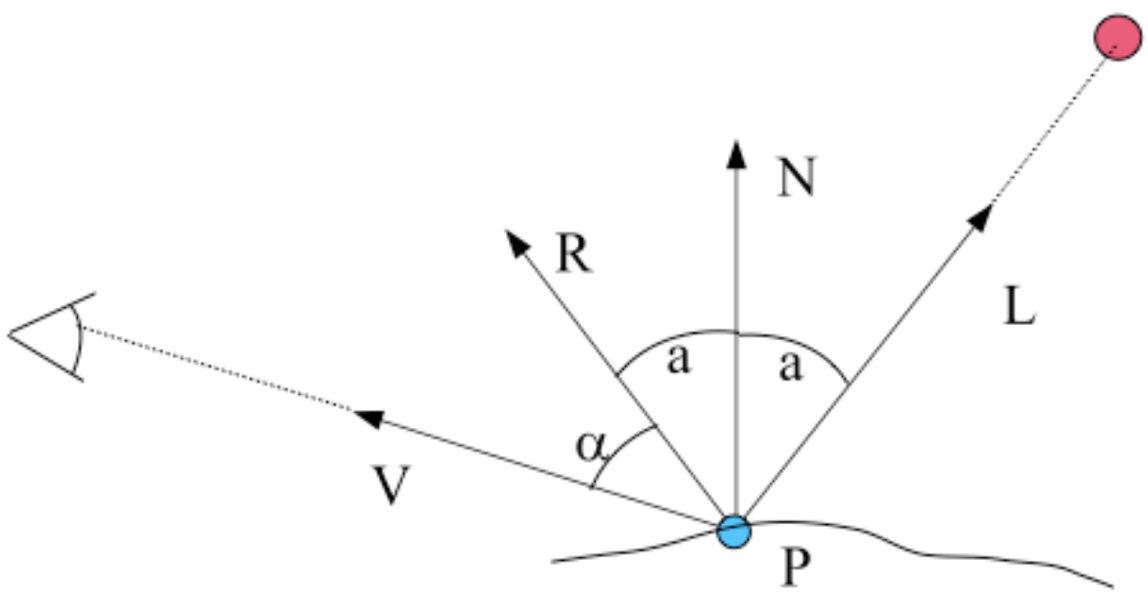
CALCUL DU SPÉCULAIRE EN OPENGL

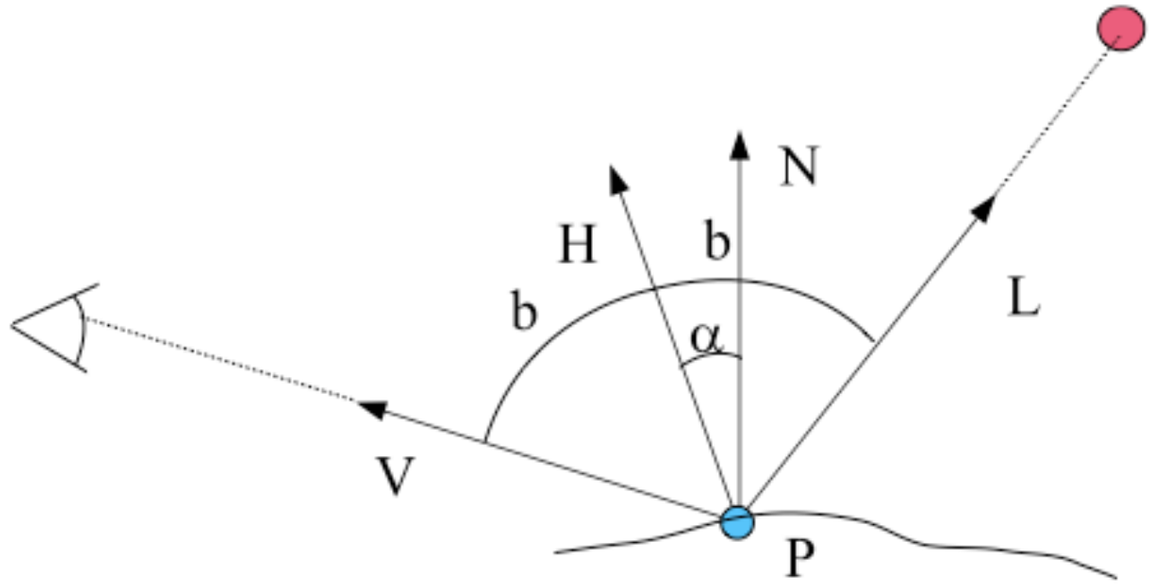


$$C_s = I_s * K_s(V \cdot R)^s$$

avec $R = 2(N \cdot L)N - L$

SIMPLIFIER LE CALCUL DE R





- On pose $H = \frac{1}{2}(L + V)$
- On prend $N \cdot H$ au lieu de $V \cdot R$ dans le calcul du spéculaire.
- Plus rapide et résultat similaire (les 2 calculs ne sont pas identiques).

PRINCIPE

- Pour chaque sommet on va calculer explicitement le N et le H
- En chacun des pixels on pourra récupérer les valeurs interpolées (moyennées) de N et de H.
- H et N seront donc qualifiés de varying.

VERTEX SHADER

```
varying vec3 N,H;

void main() {
    N = normalize(gl_NormalMatrix * gl_Normal);
    H = normalize(gl_LightSource[0].halfVector.xyz);

    gl_Position = ftransform();
}
```

- `normalize` est fournie par GLSL pour normer un vecteur
- `ftransform` est un raccourci de gl_ModelViewProjectionMatrix
- `gl_LightSource[0]` correspond à `LIGHT0`

  On a accès à ses paramètres, en particulier `.halfVector` : notre H

PIXEL SHADER

```
varying vec3 N,H;
void main() {
    vec3 N2,H2;
    vec4 couleur=vec4(0,0,0,0);
    float NdotH;
    /* interdiction d'écrire dans N */
    N2 = normalize(N);
    H2 = normalize(H);

    NdotH = max(dot(N2,H2),0.0);
    couleur = gl_FrontMaterial.specular *
    gl_LightSource[0].specular *
    pow(NdotH, gl_FrontMaterial.shininess);
    gl_FragColor = couleur;
}
```

- Il faut normer **N** et **H**
- `dot` est la fonction produit scalaire
- On accède au K_s par `gl_FrontMaterial.specular`, au I_s par `glLightSource[0].specular` et à la brillance **s** par `gl_FrntMaterial.shininess`

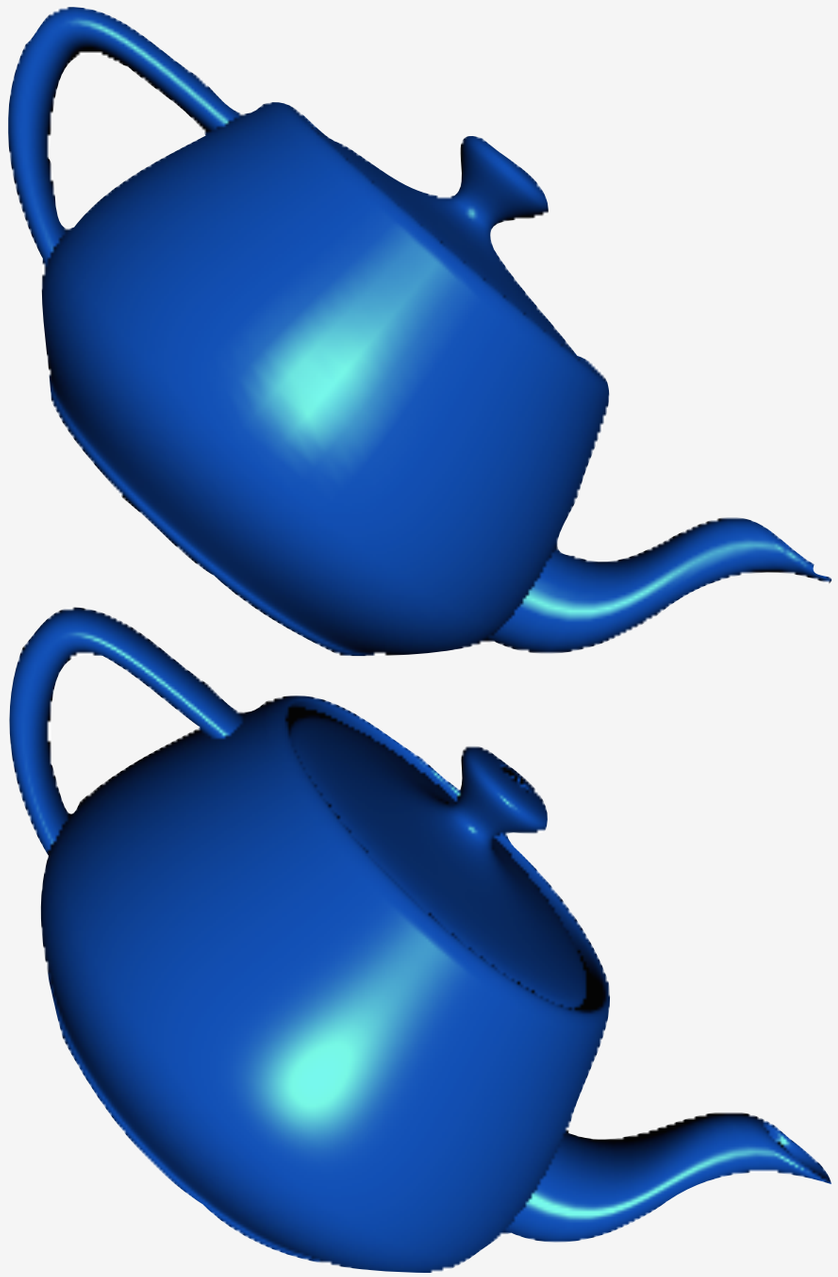
REMARQUES

- Attention : il n'y a que le spéculaire ici.

  Il faut ajouter le diffus et l'ambiant dans le vertex et le pixel shader

- Remarque : pour ces 2 derniers, il suffit de calculer la couleur aux sommets, puis d'interpoler cette couleur (on ajoute directement à couleur les 2 varying provenant du diffus et de l'ambiant).

SPÉCULAIRE PAR VERTEX VS PIXEL



EFFET CARTOON

GPGPU

General-Purpose Computing on Graphics Processing Units

GPU VS CPU

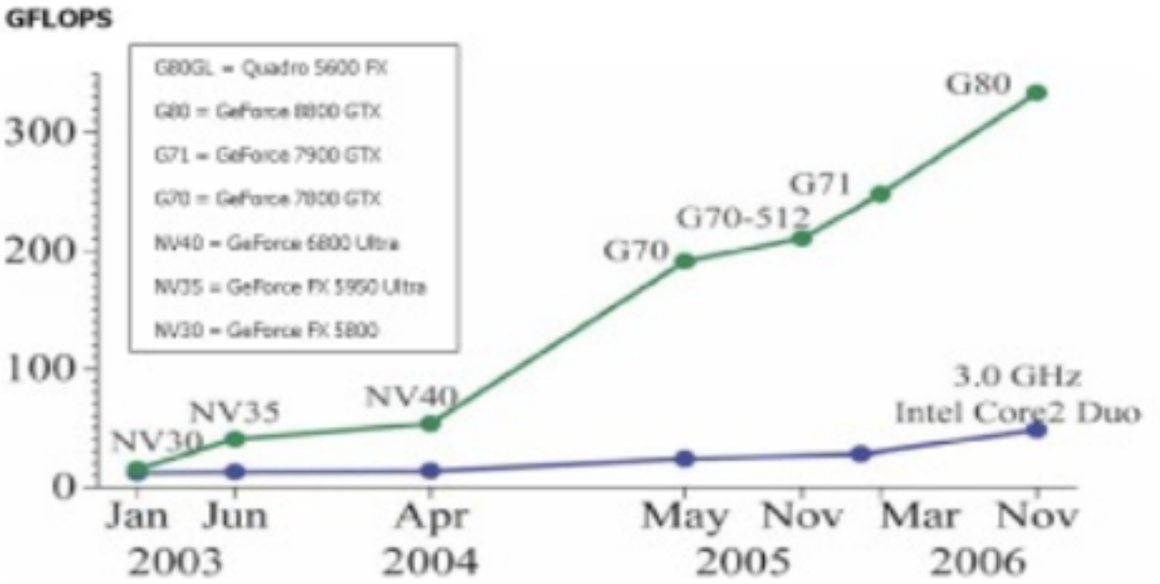
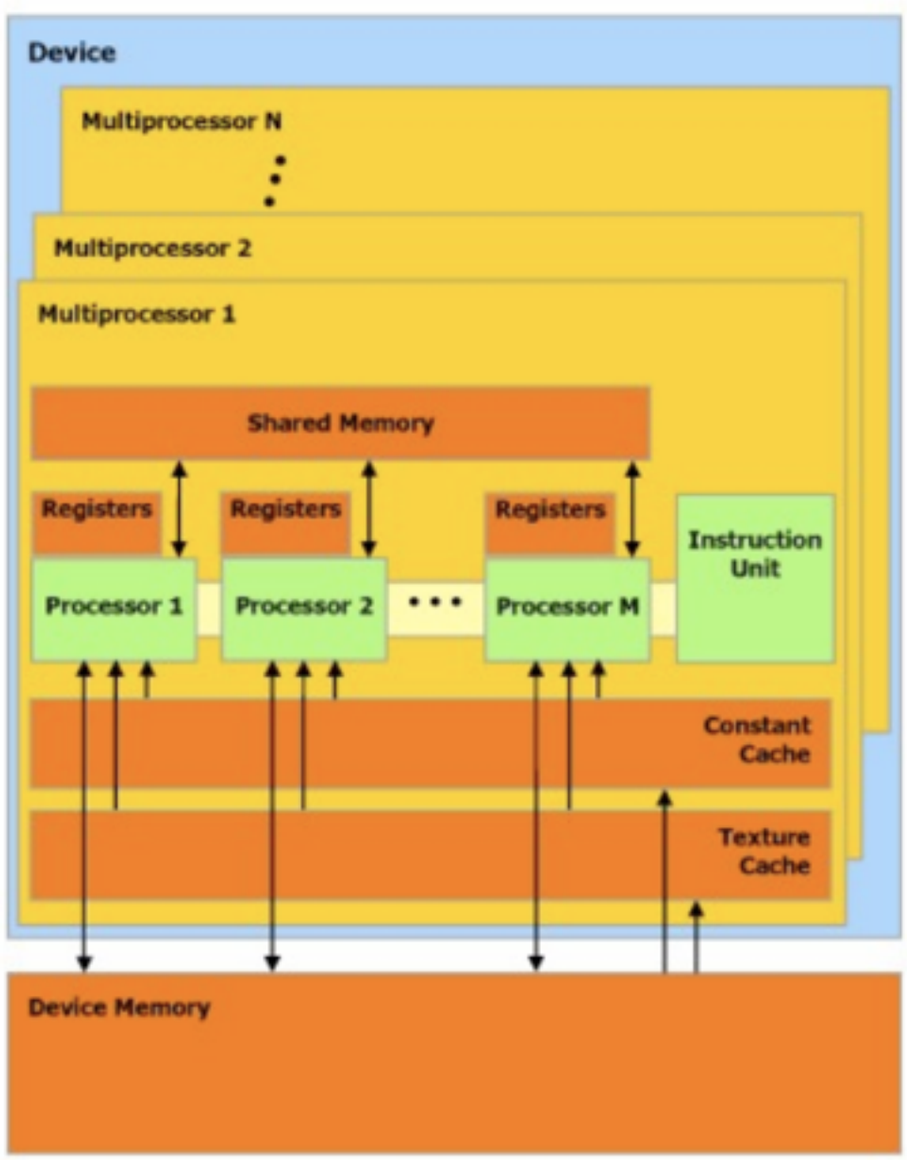


Figure 1-1. Floating-Point Operations per Second for the CPU and GPU

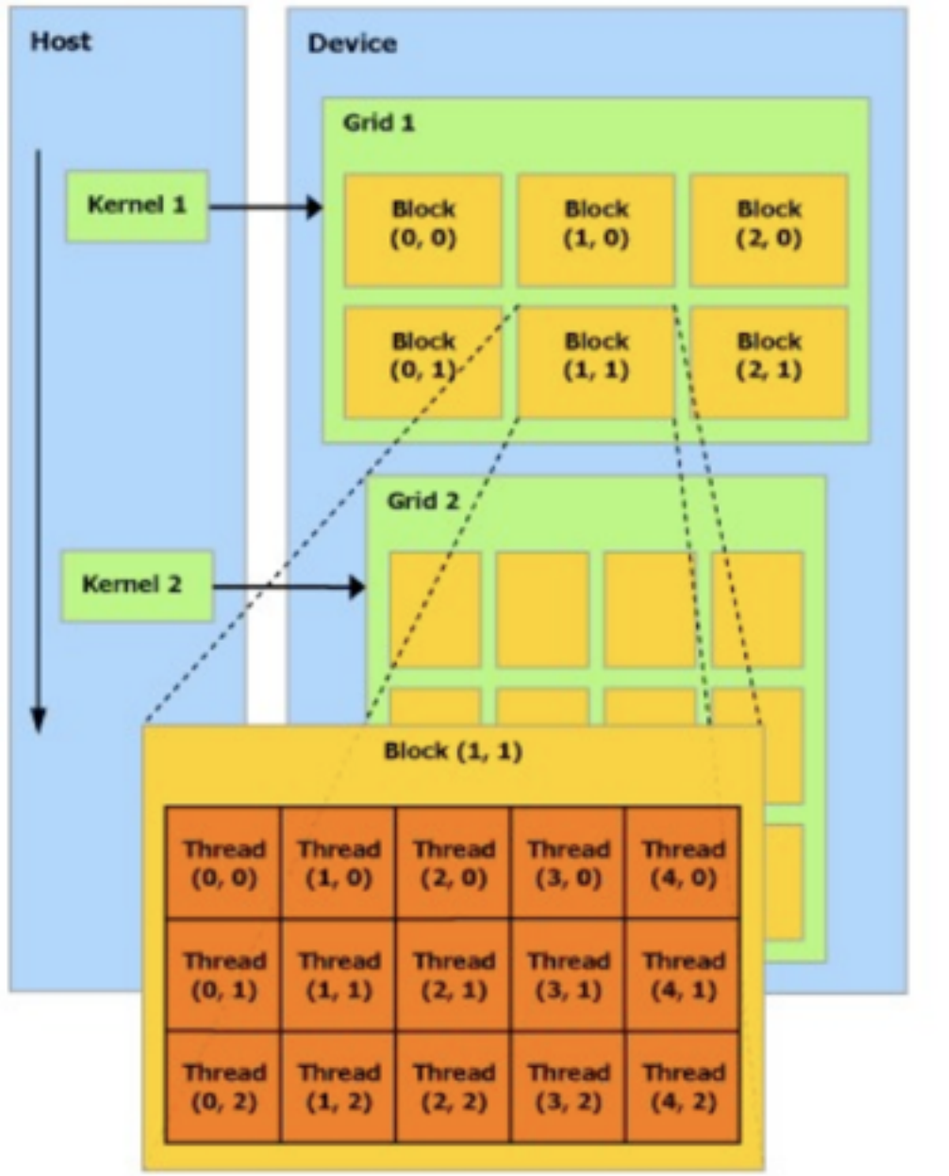
Figure 1-2. The GPU Devotes More Transistors to Data Processing

ARCHITECTURE G80 NVIDIA



A set of SIMD multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Figure 2-1. Thread Batching

OBJECTIF

Utiliser le GPU pour faire des calculs traditionnellement faits par le CPU

- Les GPU supportent maintenant les nombres à virgule flottante (sur 16 ou 32 bit)
- Support if then else

STREAM PROCESSING (SIMD)

- Principe : appliquer un ensemble un même jeu d'instructions relativement petit (kernel) à un flux
- Les kernels sont appliquées à chaque élément du flux
- Les éléments sont traités indépendamment

KERNEL ET FLUX

- Les vertex et les fragments sont les éléments du flux
- Les pixels et vertex shaders sont les kernels
- Le flux se présente sous forme de grille 2D

APPLICATIONS

- Algèbre linéaire
- Traitement d'image
- Simulation physique…

EXEMPLE – VERSION CPU

```
x = [10^8 éléments]
y = [10^8 éléments]
make array x by y

for each “x” {
    for each “y” {
        do_some_hard_work(x,y) //10^16 fois
    }
  }
```

EXEMPLE – VERSION GPU

```
x = [10^8 éléments]
y = [10^8 éléments]
make array x by y

do_some_hard_work(x,y) // 1 fois
```

API

- Avant : utilisation de shaders
- Maintenant : CUDA (2007) pour carte NVidia
- 2010 : OpenCL (Kronos group - Apple)

MULTIPLICATION D'UNE MATRICE PAR UNE CONSTANTE

CPU

```
void MatrixMulOnHost (float a, float **A, int size)
{
    for (int i = 0 ; i < size ; i++)
        for (int j = 0 ; j < size ; j++)
            A[i][j] = a * A[i][j];
}
```

MULTIPLICATION D'UNE MATRICE PAR UNE CONSTANTE

GPU avec CUDA

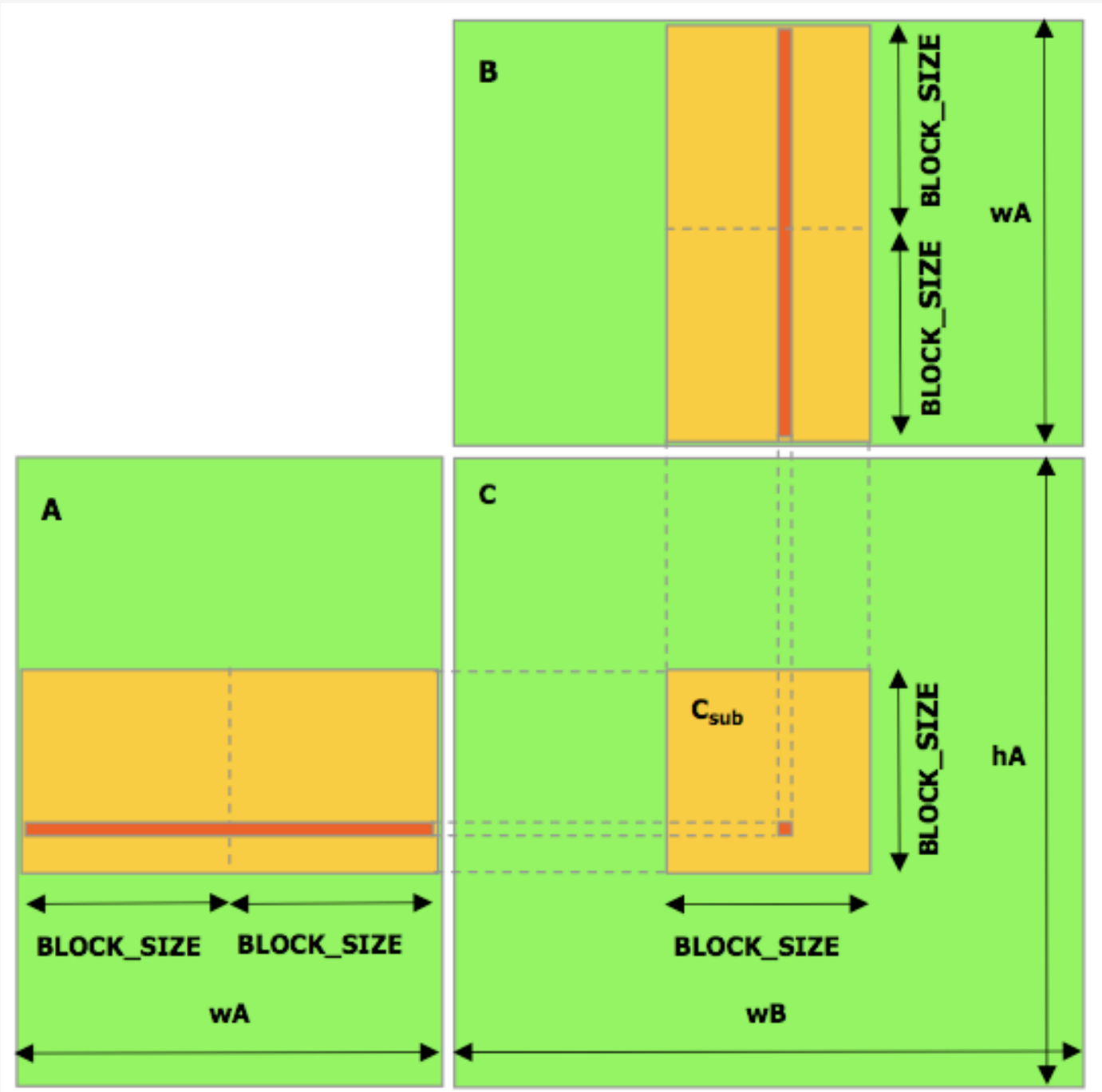
```
__global__
void MatrixMulKernel(float a, float **A)
{
    int row = threadIdx.y;
    int col = threadIdx.x;
    A[row][col] = a * A[row][col];
}

void MatrixMulOnDevice(float a, float *A, int size)
{
    int matrix_size = size * size * sizeof(float);
    float **d_A;
    cudaMalloc(&d_A, matrix_size);
    cudaMemcpy(d_A, A, matrix_size, cudaMemcpyHostToDevice);
    dim3 dimGrid(1, 1);
    dim3 dimBlock(size, size);
    MatrixMulKernel<<<dimGrid, dimBlock>>>(a, d_A, size);
    cudaMemcpy(A, d_A, matrix_size, cudaMemcpyDeviceToHost);
    cudaFree(d_A);
}
```

MULTIPLICATION DE MATRICES AVEC CUDA

- GPU divisé en thread blocs et threads
- Chaque thread bloc est responsable de calculer une sous matrice
- Chaque thread va calculer un élément de la matrice

MULTIPLICATION DE MATRICES AVEC CUDA



CONCLUSION - PERSPECTIVES

- Modes graphiques : assez peu de changement

  Augmentation de résolutions

- Amélioration des bus et mémoires

  Débit, fréquence de fonctionnement

- Efforts concentrés sur le GPU :

  Processeur de plus en plus performant

  Effets visuels coûteux en temps-réel (lancer de rayons).

  Calculs scientifiques complexes (simulation de tissus, cheveux, …).