ENJMIN

ECOLE NATIONALE
DU JEU ET DES MEDIAS
INTERACTIFS NUMERIQUES

PIXAR's RenderMan

Shaders : une introduction

Cg

OpenGL

DIRECTX

Alexandre Topol

Sommaire

- Historique
  - L'évolition du pipeline 3D
  - L'état aujourd'hui
  - Fixed-function (FFP) VS programmable-function (PFP)
- Le FFP plus en détail
  - La séquence d'opérations
  - Opérations sur des flux (de sommets ou de fragments)
- Les Shaders
  - Définition des shaders
  - Les précurseurs
  - Présentation des langages de shaders
    - Cg
    - HLSL
    - GLSL
  - Comparaison entre les langages

Définition des shaders

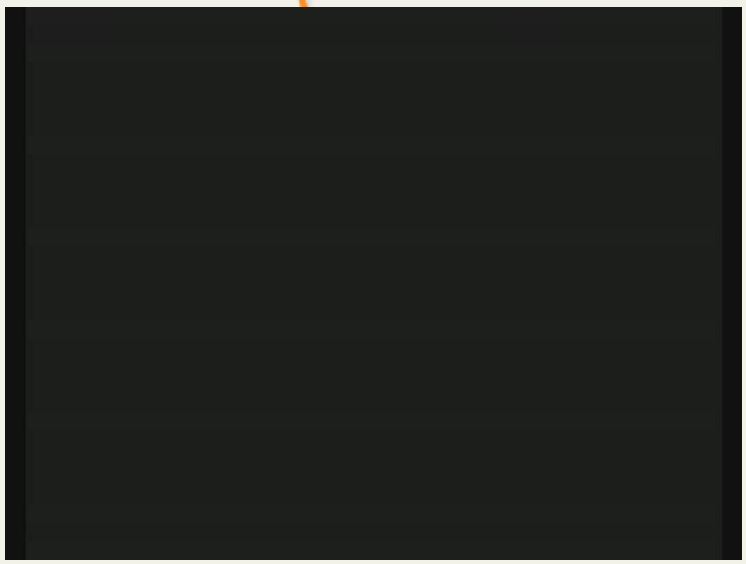
- Objectif : création d'effets visuels
- Technique récente (en 3D temps réel)
- Conséquence : choix du langage difficile
- Petit programme écrit dans un fichier et exécuté pour chaque primitive.

- 3 types de shaders :
  - Vertex shader : traite les sommets
  - Pixel shader : traite les fragments
  - Geometry shader : traite les géométries
- OpenGL ou DirectX se chargent d'inclure les shaders.

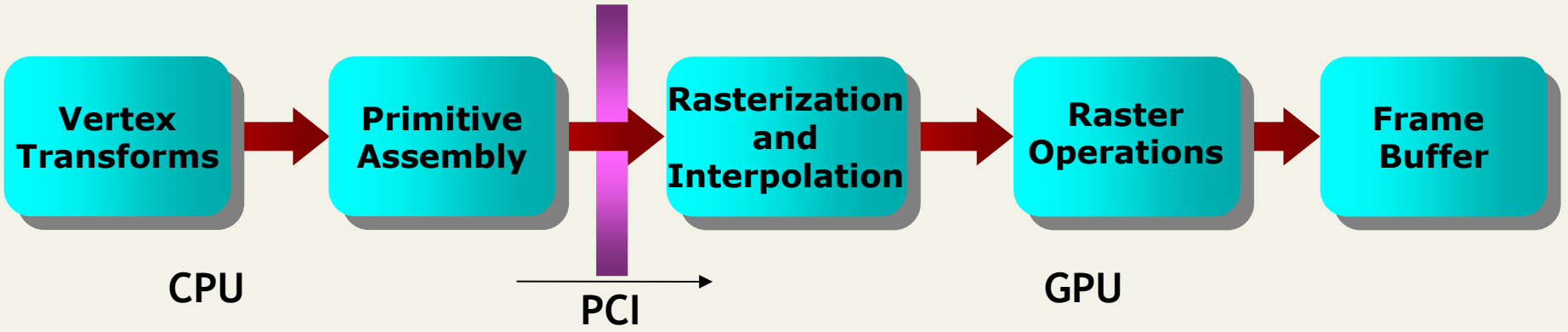
Définition des shaders

- Pourquoi programmer le GPU ?
  - Principalement pour les effets dans les jeux vidéos. Ils utilisent souvent l'éclairage dynamique, les ombres portées, ...
- Des minis programmes embarqués dans le GPU
  - Ce sont des programmes de 3 à 100 lignes de code (en moyenne 10)
- GPGPU (www.gpgpu.org) ⇒ CUDA, OpenCL
  - Utilisation généralisée, hors contexte graphique
    - ⇨ particle engines
    - ⇨ Illumination
    - ⇨ signal processing
    - ⇨ image compression
    - ⇨ computer vision
    - ⇨ sorting/searching

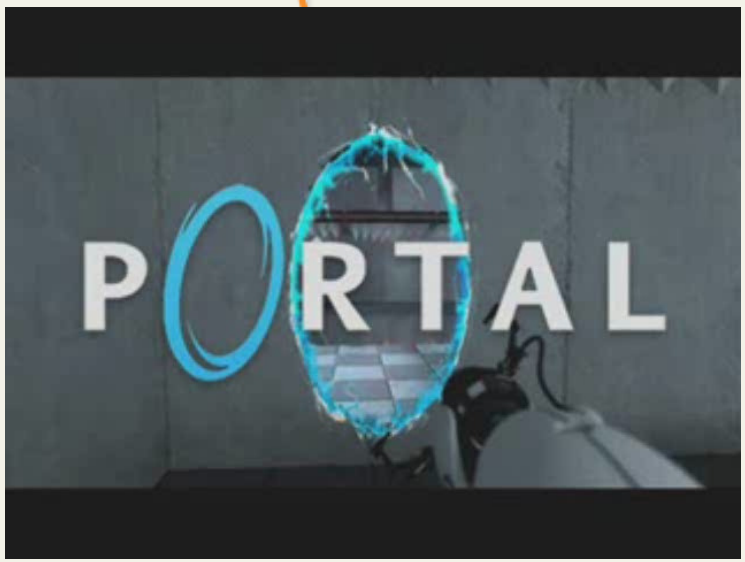
Generation I : 3dfx Voodoo (1996)



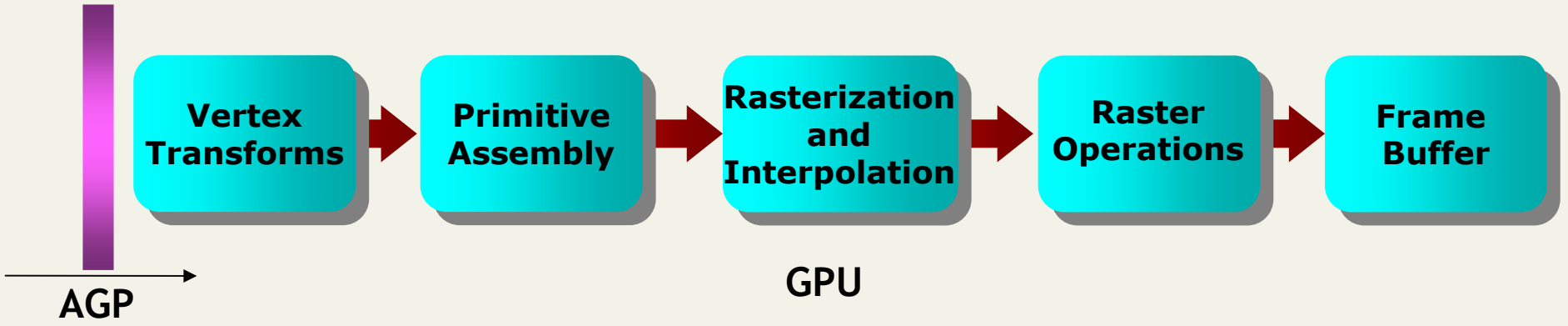
- La première carte 3D
- En ajout à la carte 2D
- Ne faisait pas de transformations sur les sommets
- Faisait texture mapping, z-buffer



Generation II : GeForce/Radeon 7500 (1998)



- Innovation principale : le GPU effectue les opérations de T&L
- Possibilité de faire du multi-texturing : bump map, light map, ...
- Bus AGP plus rapide à la place du PCI



La situation à la fin des années 90

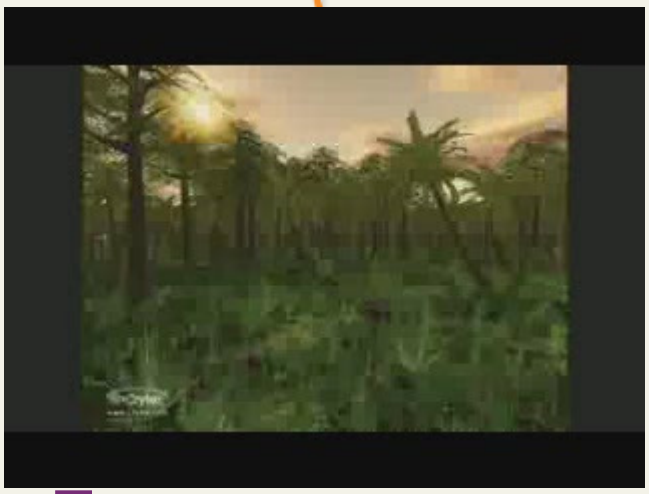
- Les + :
  - Moteur 3D standardisé sur les cartes graphiques
  - Accélération materielle
- Les -
  - *En tant que programmeurs nous ne pouvions que tourner des potars et positionner quelques cavaliers avant d'appuyer sur un bouton de rendu*
  - *De la boîte noire sortait une image*

- Ca permettait cependant de faire pas mal de choses, une fois les variables de la boîte noire bien connues.
- On pouvait (devenir un super opérateur de potars & cavaliers et) obtenir des supers effets (mais très peu et à quel prix !)

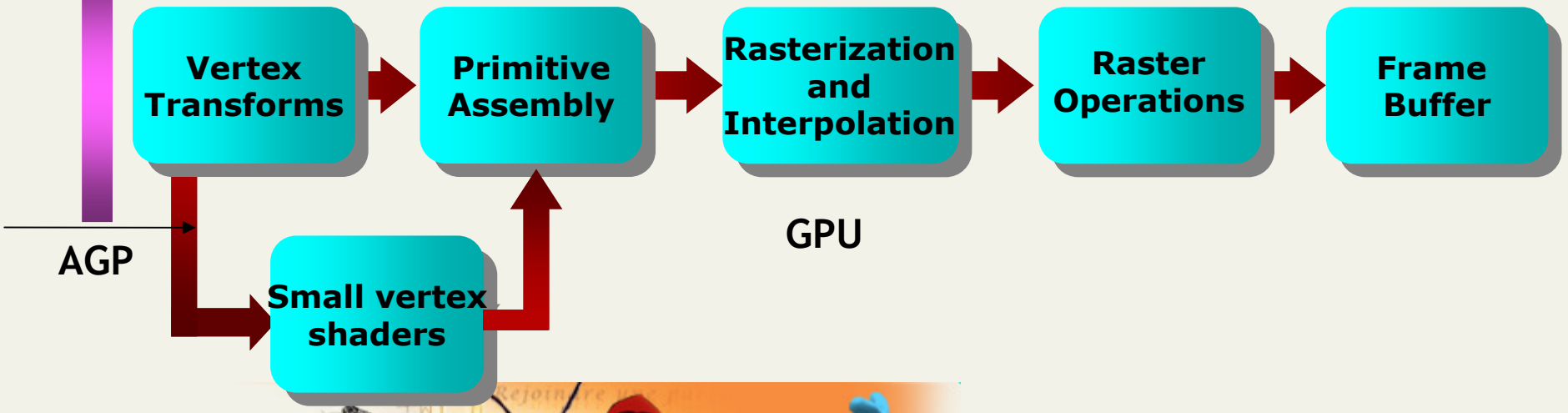
La situation à la fin des années 90

- L'industrie vidéo-ludique demande de la créativité graphique
  - Antinomique avec des effets prédéfinis
  - Besoin de nouveaux effets
- Les programmeurs graphiques devaient faire des rendus multipasse
- Et passer beaucoup de temps à tuner les potars et cavaliers pour obtenir de nouveaux effets :
  - Pour obtenir des effets en utilisant les matrices de transformation des textures
  - Pour combiner les textures avec des equations autres que celles proposées par le blending traditionnel

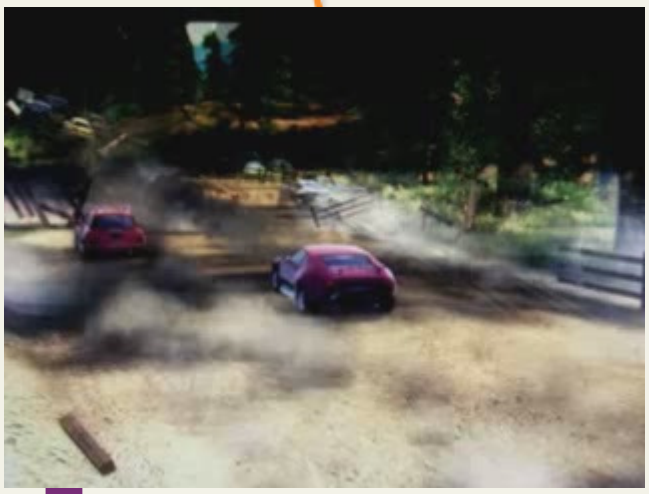
Generation III : GeForce3/Radeon 8500 (2001)



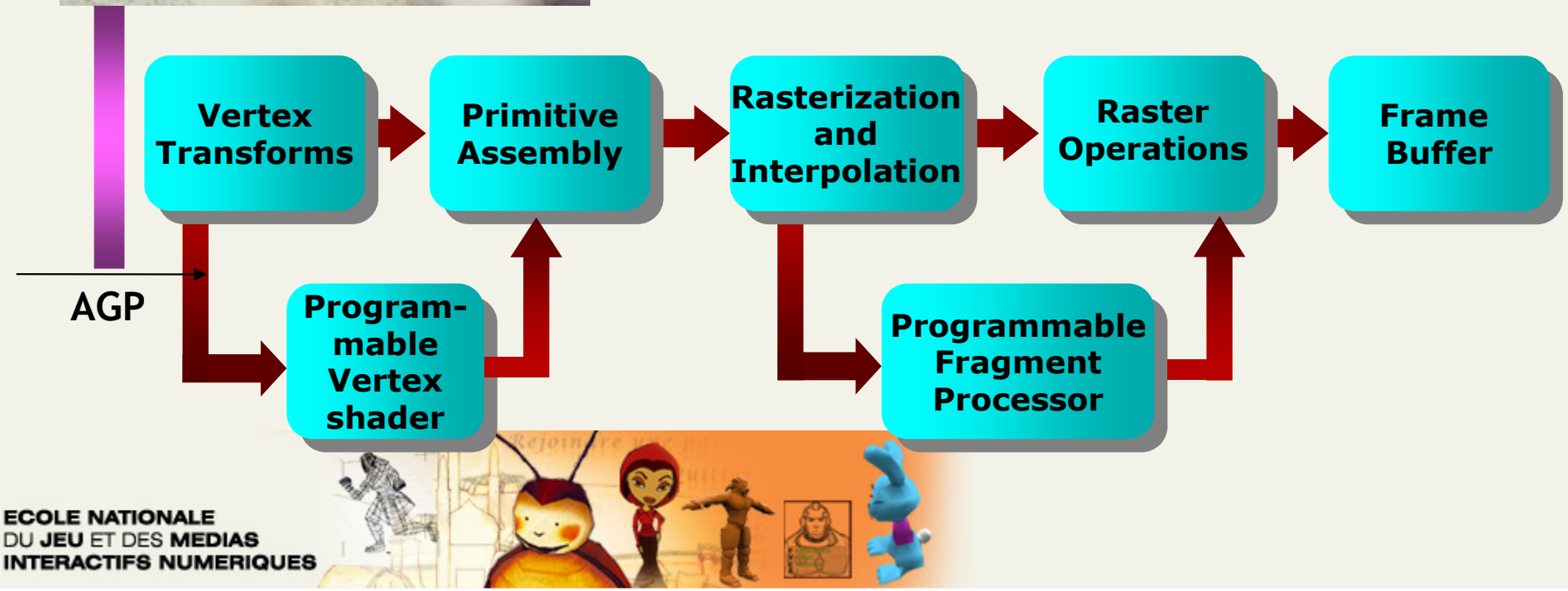
- Pour la première fois, possibilité d'un peu de programmation sur les sommets
- Gestion du mip-mapping et de l'over-sampling (pour l'antialiasing)



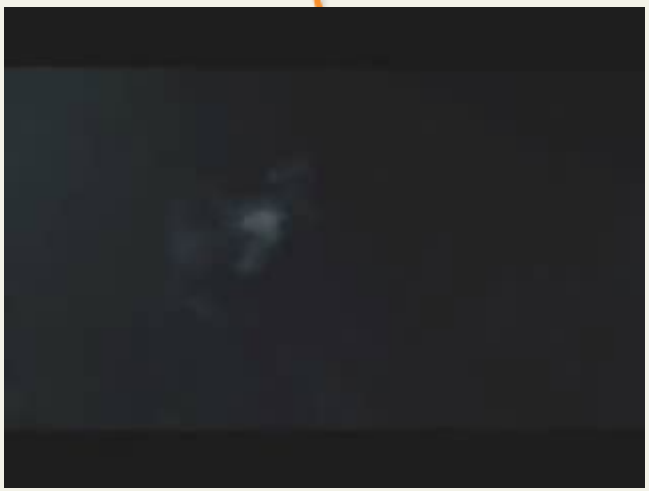
Generation IV : Radeon 9700/GeForce FX (2002)



- Première génération de cartes programmables
- Différentes versions ont des possibilités différentes pour les shaders (mémoires, traitements)



Generation IV.V : GeForce6/X800 (2004)

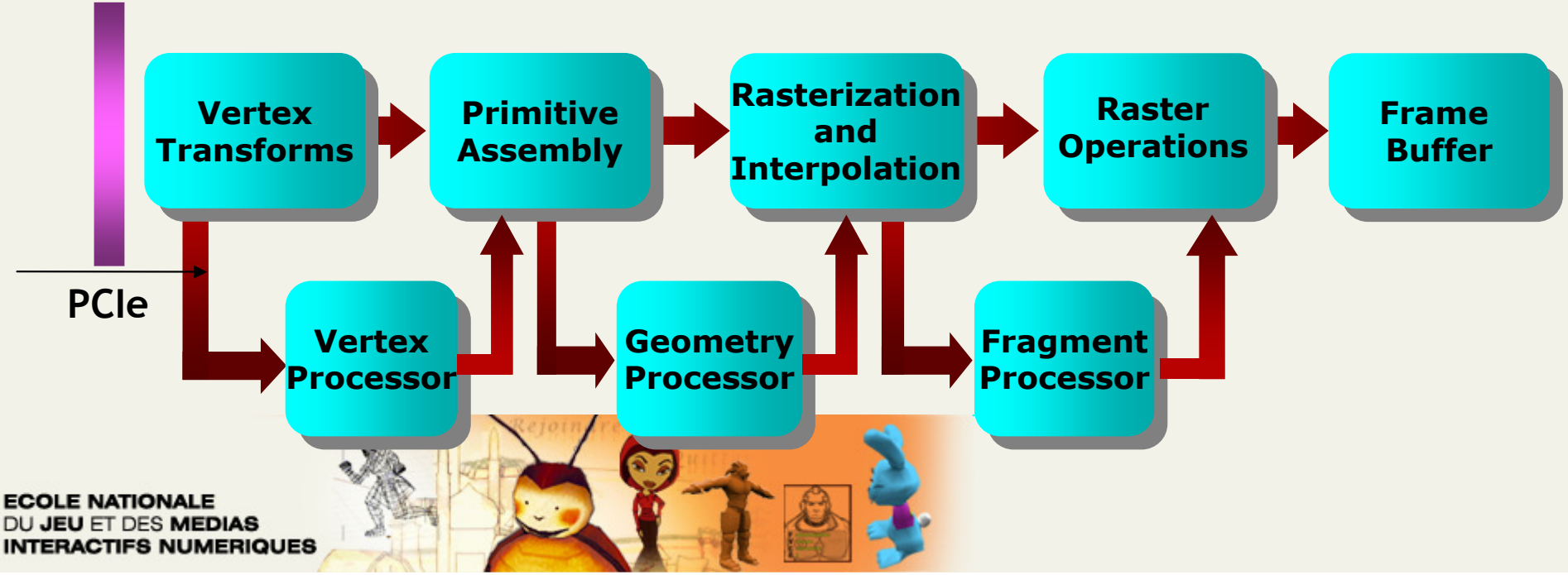


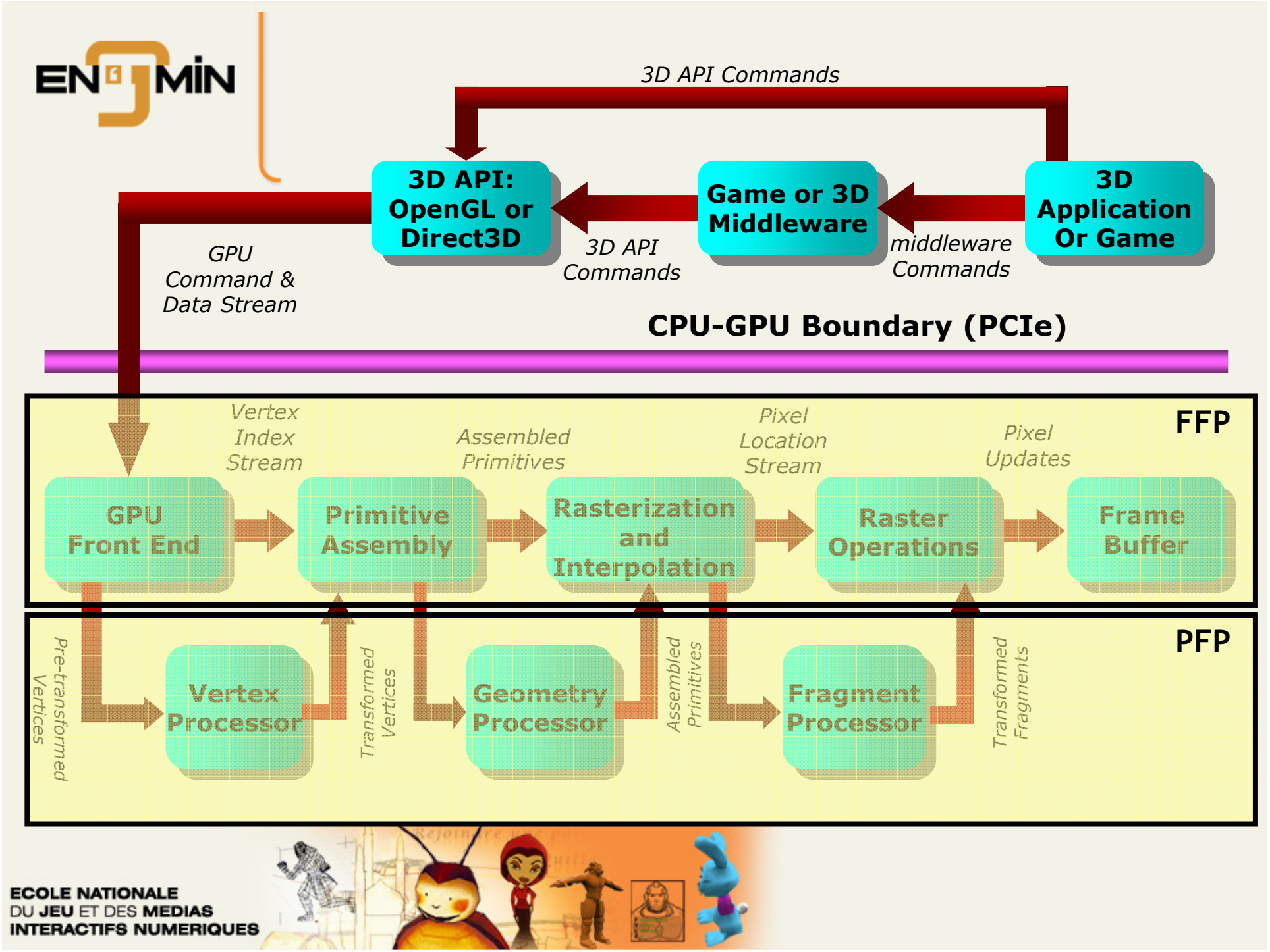
Pas vraiment un énorme changement mais ...

- Rendu simultané dans plusieurs buffers
- De vraie conditionnelles et boucles
- Plus grande précision pour les opérations (64bits de bout en bout)
- Bus PCIe
- Plus de mémoire / pour de plus longs programmes / pour de plus grandes textures

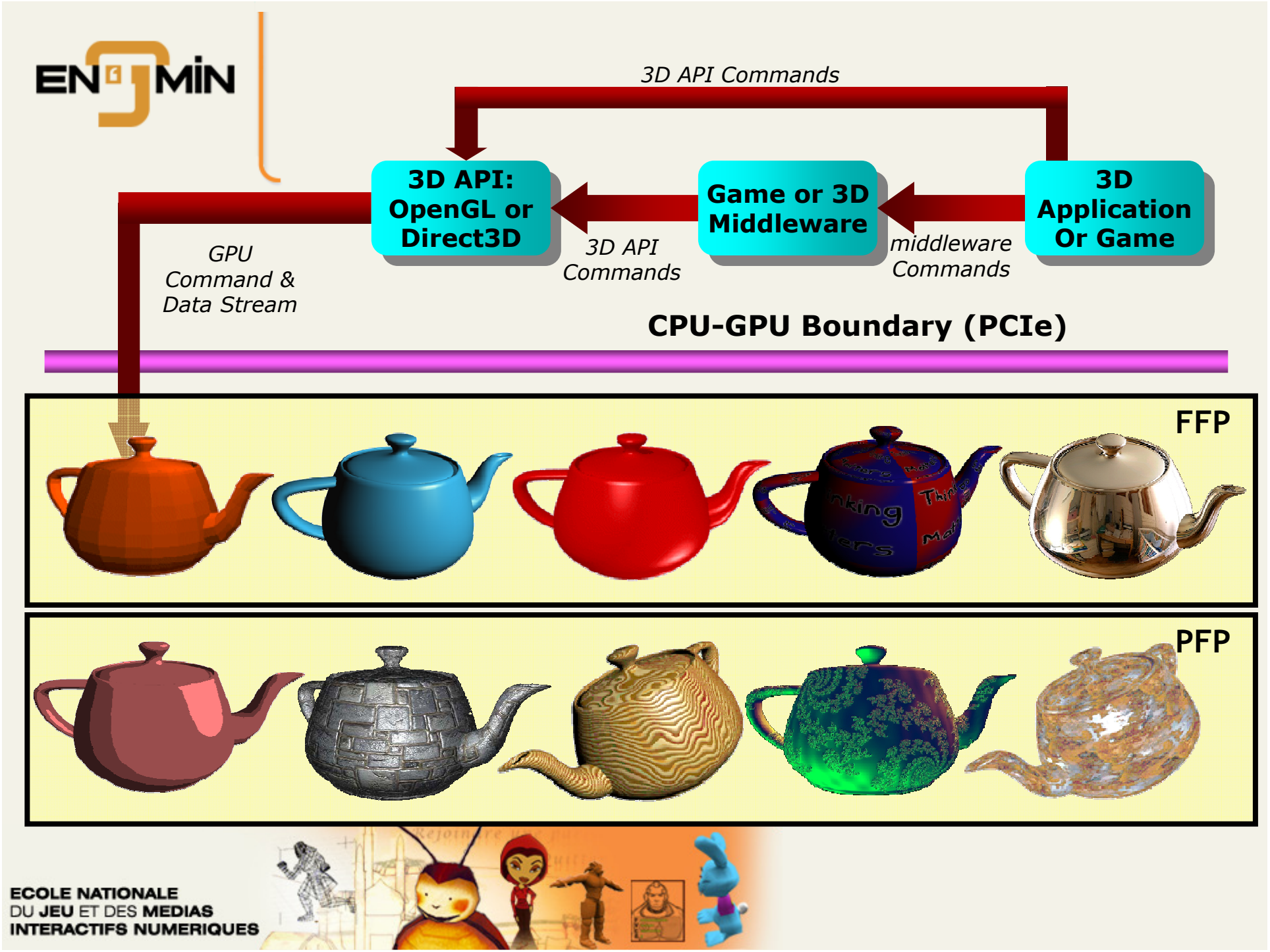
Generation V : GeForce9/Radeon HD (2008+)

- Uniformisation des possibilités
- Compatibilité OpenGL 3.0
- Compatibilité DirectX 10
- Geometry shaders





ECOLE NATIONALE
DU JEU ET DES MEDIAS
INTERACTIFS NUMERIQUES


3D API Commands
3D API: OpenGL or Direct3D
Game or 3D Middleware
3D Application Or Game
GPU Command & Data Stream
3D API Commands
middleware Commands
CPU-GPU Boundary (PCIe)
FFP
PFP
ECOLE NATIONALE
DU JEU ET DES MEDIAS
INTERACTIFS NUMERIQUES

Les données du pipeline

Vertex

(x, y, z)

(r, g, b,a)

(Nx,Ny,Nz)

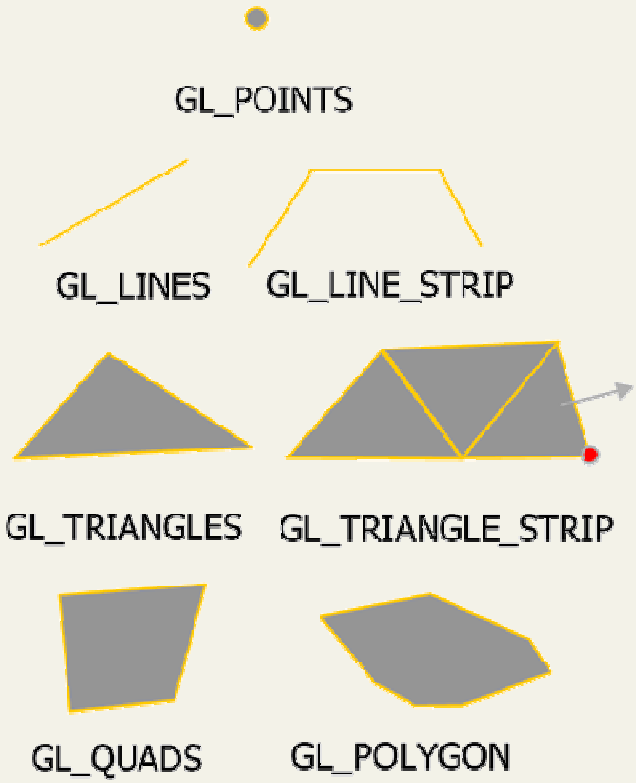
(tx, ty,[tz])

(tx, ty)

(tx, ty)

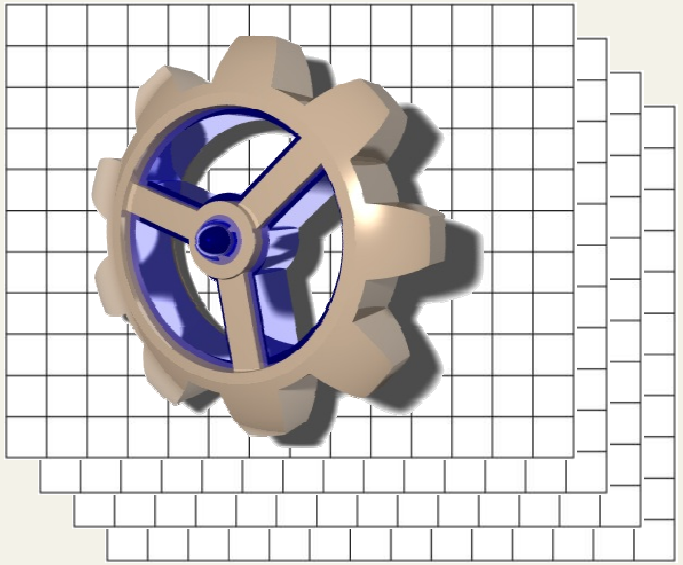
Material properties*

Geometry



Image

F(x,y) = (r,g,b,a)



Possibilités

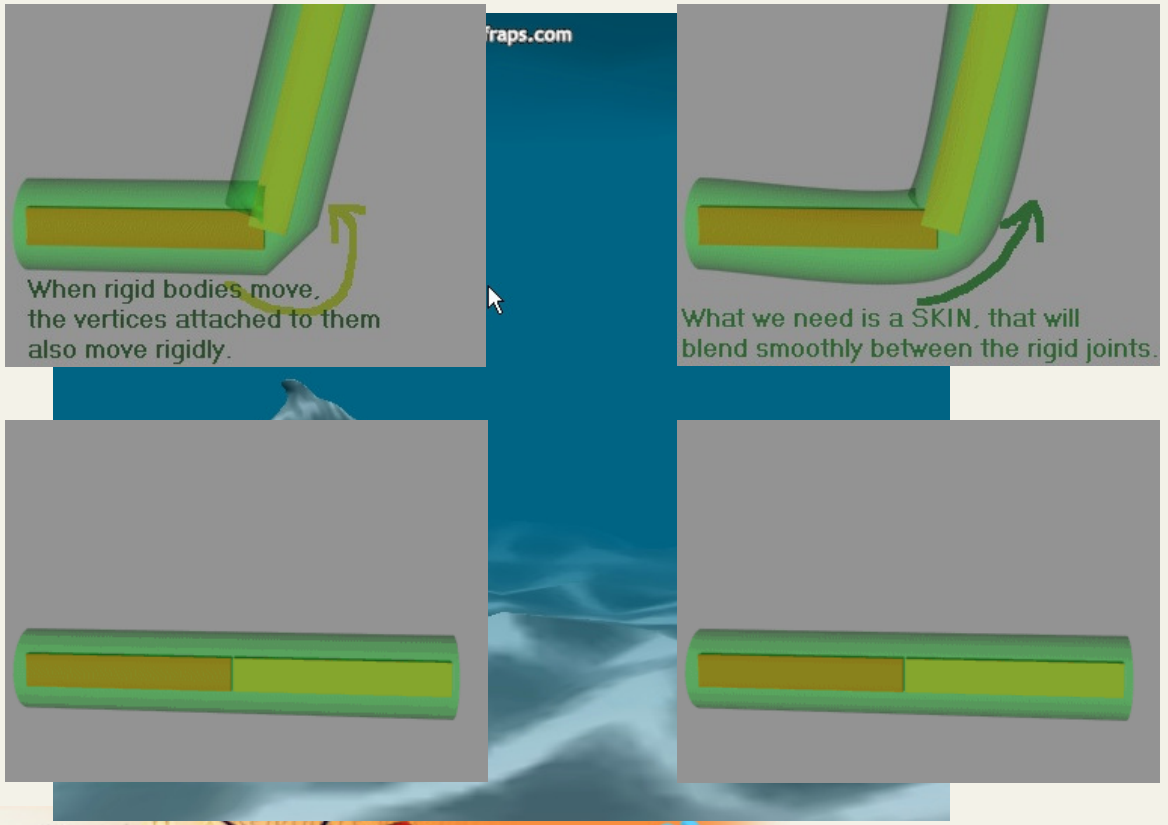
- N’ont de limite que l’imagination des programmeurs graphiques (enfin …. Ou presque)
- Restent quelques limitations dues au GPU
- Limitations dues au temps de calcul nécessaire
  - En particulier pour le pixel shader très coûteux
  - Faire le maximum de choses dans le vertex shader

Possibilités des vertex shaders

- Remplace l'étape de T&L
- Possibilités:
  - Position Procédurale
  - Mélange de transformations ('Vertex Blend')
  - Mélange de maillages ('Morphing')
  - Déformation du maillage ('Vertex Deformation')
  - Eclairage par sommet ('Per Vertex Lighting')
  - Tissu, peau, interpolation, displacement maps…
  - Coordonnées de texture
  - Brouillard particulier
  - Taille d'affichage d'un point

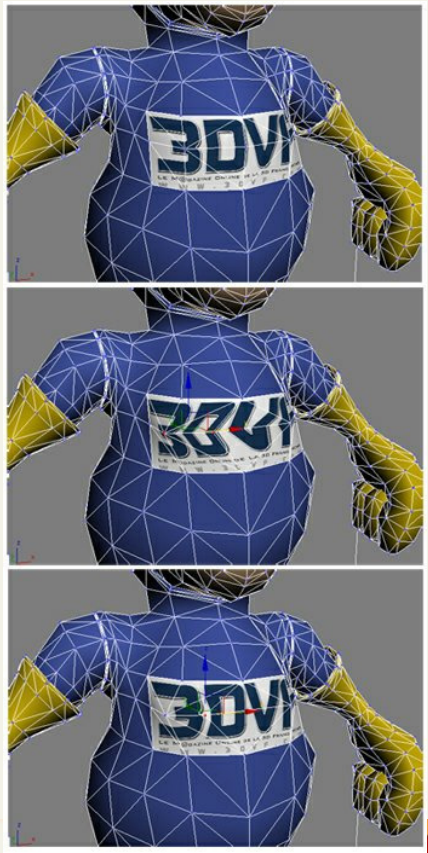
Possibilités des vertex shaders

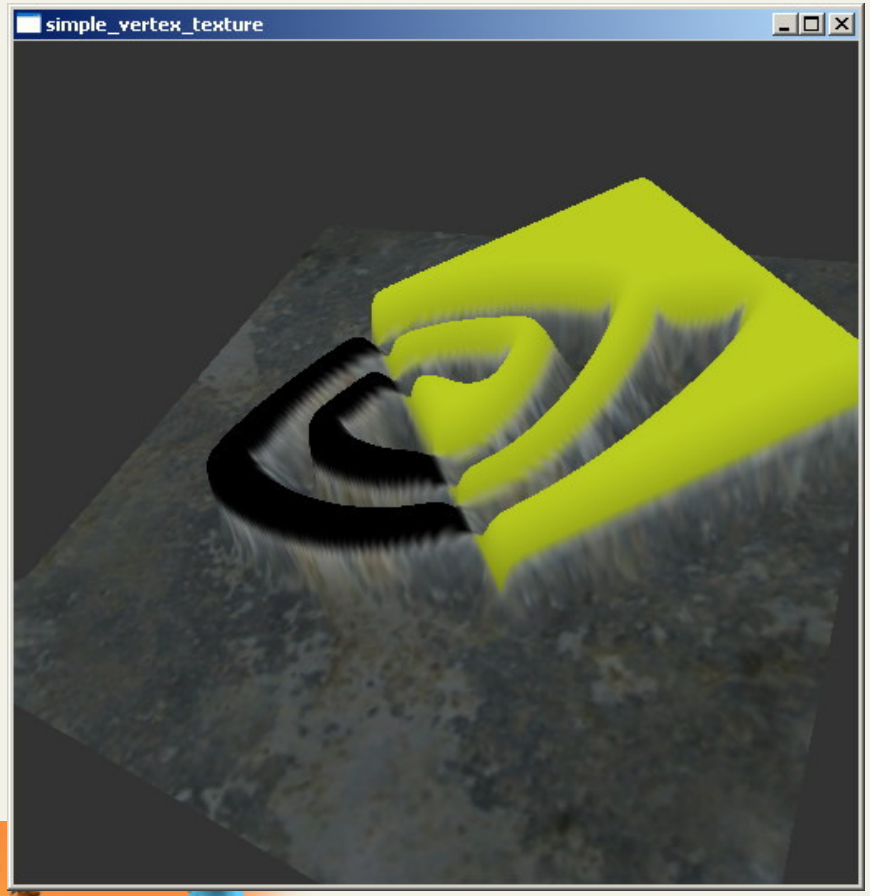
- Mélange de transformations (*Vertex blend*)



Possibilités des vertex shaders

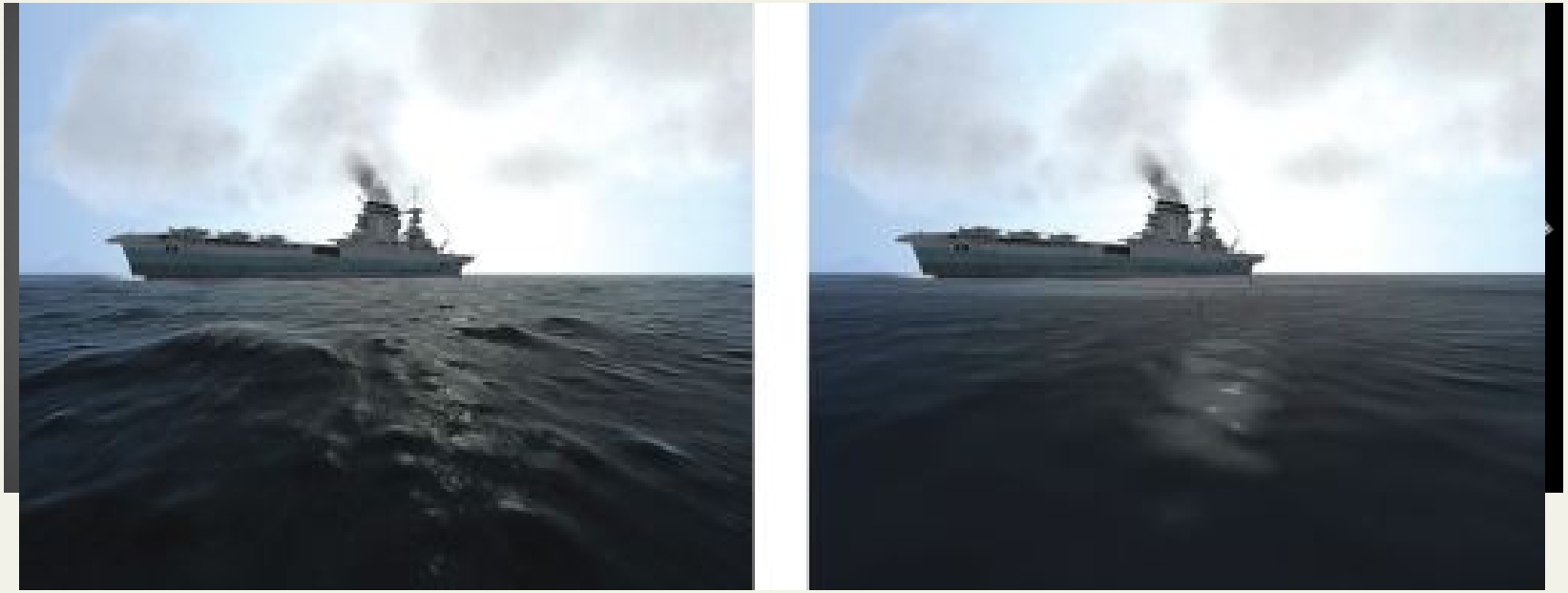
- Déformation de maillage (*Vertex deformation*)





Possibilités des vertex shaders

- *Displacement Mapping*

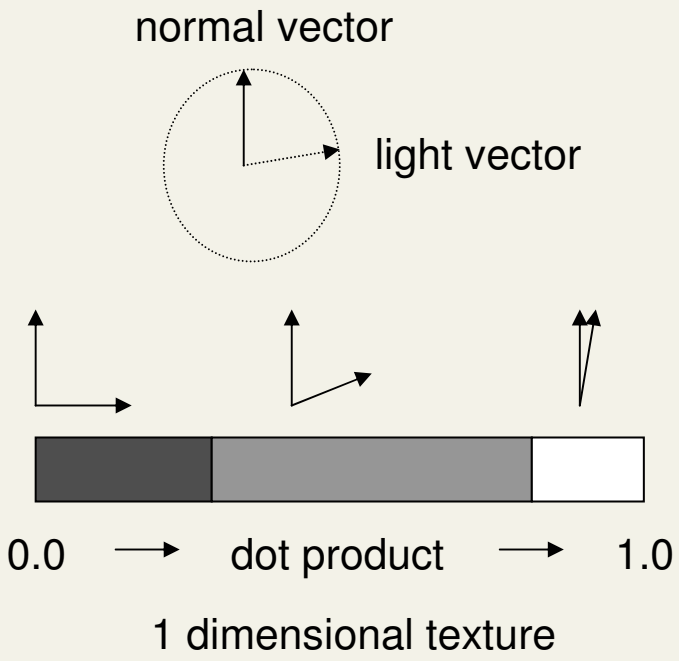


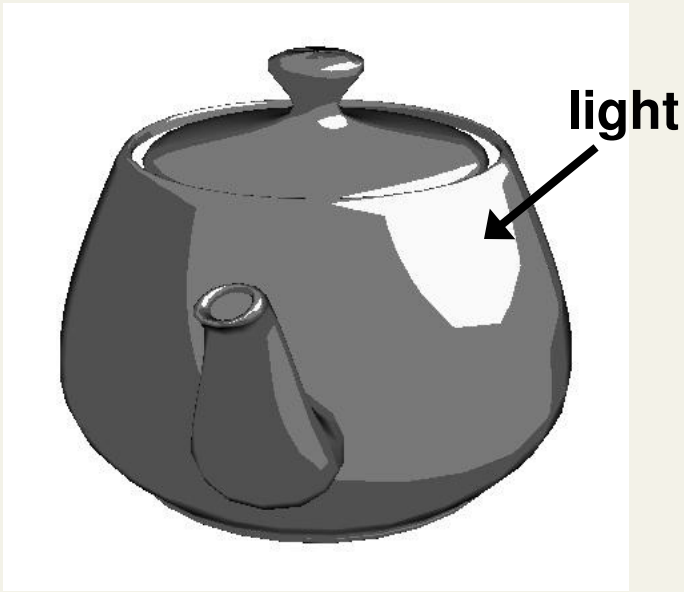
Possibilités des pixel shaders

- Remplace l'étape du calcul de la couleur
- Possibilités:
  - Réflexion par pixel
  - Illumination par pixel (phong, BRDF)
  - Textures procédurales
  - Et … pleins d'effets au niveau des pixels

Possibilités des pixel shaders

- *Cartoon shading*
  - Produit scalaire (lumière x normale) pour indexer une texture 1D



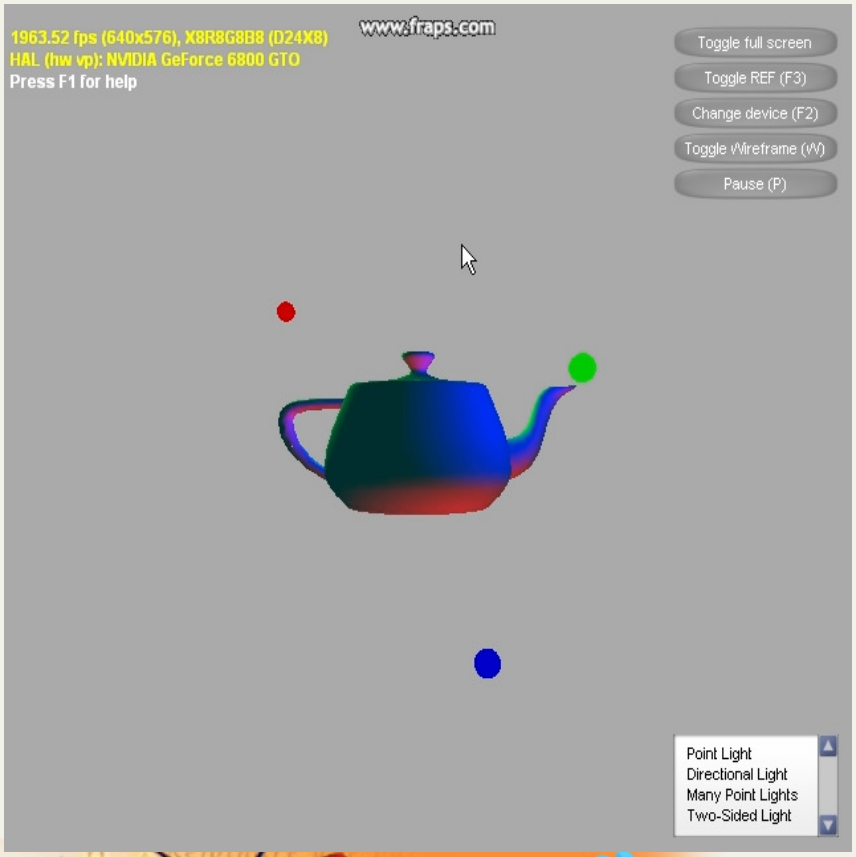


Possibilités des pixel shaders

- Exemple : « cartoon shading »
  - Zelda – The Wind Maker
  - A Scanner Darkly

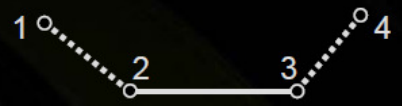
Possibilités des pixel shaders

- Éclairage par pixel

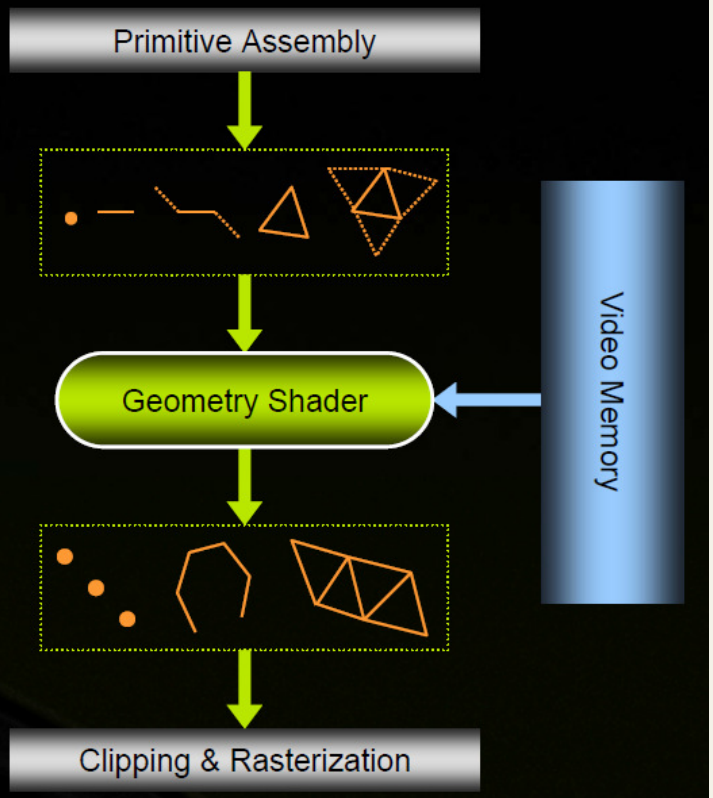


Geometry Shader Basics

- Input
  - Standard primitives
    - point, line, triangle…
  - New primitive types include neighboring vertices
    - Line with adjacency

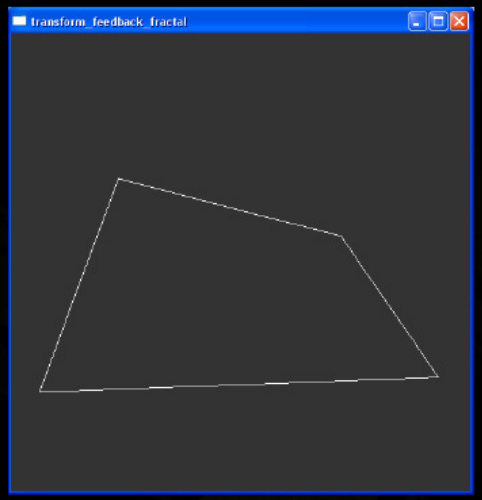


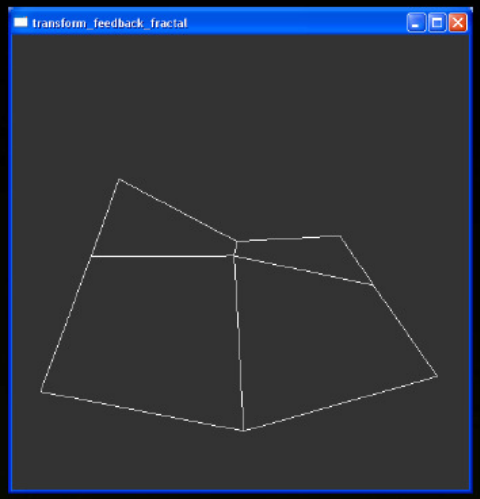
    - Triangle with adjacency

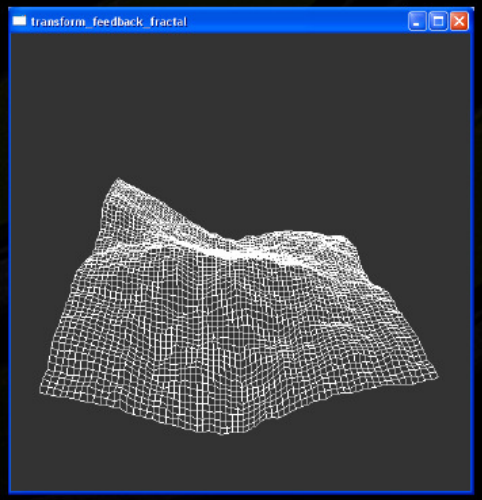


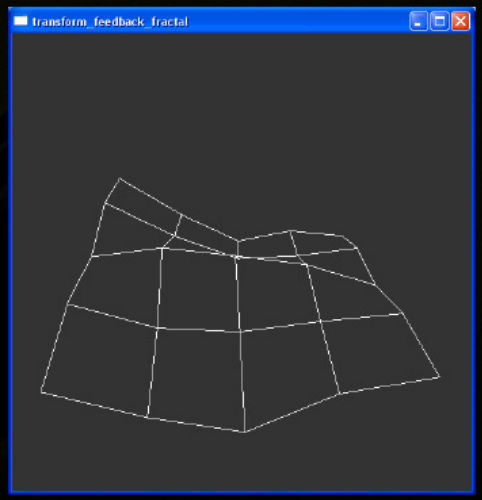
- Output
  - Unique output type (independent from input type)
  - Points, line strips or triangle strips
  - Can output zero or more primitives
  - Generated primitive stream is in the same order as inputted

Terrain Subdivision









Les précurseurs

- Renderman
  - Interface de description de scène 3D
  - Créée quand OpenGL n'existait pas encore
  - Créée par Pixar.
  - A introduit la notion de shaders.
  - Les shaders : procédures écrites dans un langage appelé RenderMan Shading Language.
  - v3.1 1998, v3.2 2000, v3.3 coming soon
- Stanford Shading Language
  - Apparu en 1999.
  - Véritable inspirateur des langage de shaders haut niveau pour la 3D par projection de facettes

PIXAR's RenderMan

Assembleur vs Langage haut-niveau

En assembleur

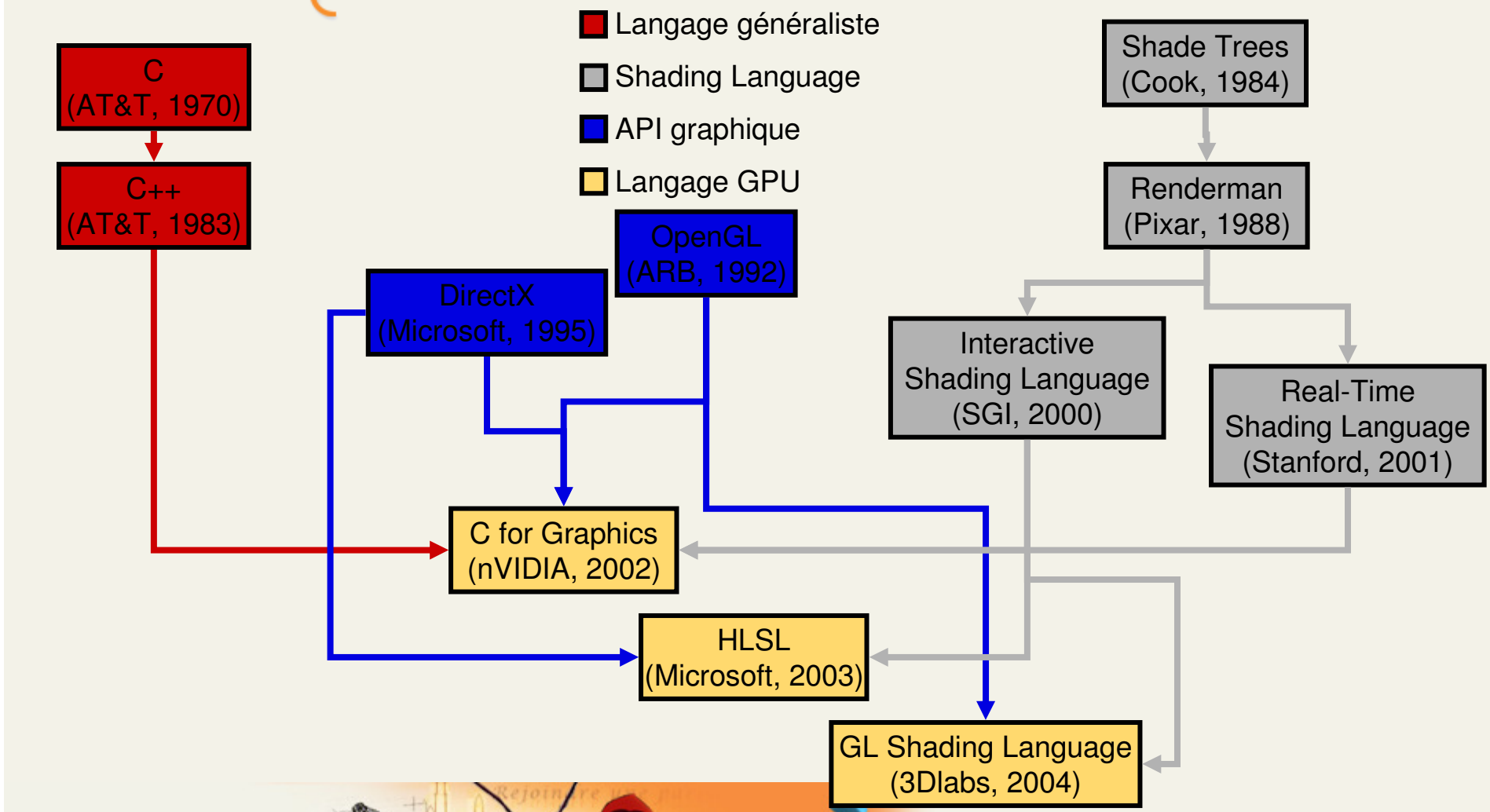
```
…
dp3 r0, r0, r1
max r1.x, c5.x, r0.x
pow r0.x, r1.x, c4.x
mul r0, c3.x, r0.x
mov r1, c2
add r1, c1, r1
mad r0, c0.x, r1, r0
…
```

Blinn-Phong shader écrit en utilisant les deux méthodes

En langage haut-niveau

```
…
float4 cSpec = pow(max(0, dot(Nf, H)), phongExp).xxx;
float4 cPlastic = Cd * (cAmbi + cDiff) + Cs * cSpec;
…
```


Arbre généalogique des shaders

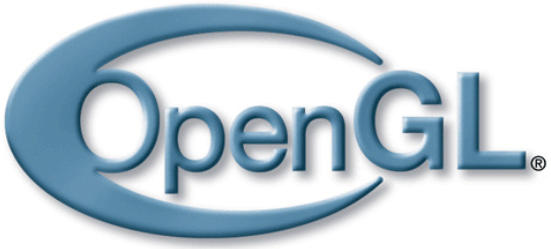


Aides au développement

- FXComposer & ShaderDebugger 2.5 de NVidia (HLSL, Cg)
  - 30 jours en eval pour shaderDebugger
- Rendermonkey d'ATI/AMD (HLSL, GLSL, Cg)
  - Debugger par log
- Pix de microsoft (HLSL)
  - Fonctionnalités pour la Xbox
  - debugger intégré pour HSLS
- Shader Designer de TyphoonLabs (GLSL)
  - Objectif : aide au développement de vertex & pixel shader
  - Windows & Linux
- GLIntercept (pour OpenGL)
  - Log tous les appels OpenGL, shaders inclus
- glslDevil Shader Debugger (pour GLSL)
  - Un nouvel outil sans éditeur mais bons mécanismes de débuggage.

ENJMIN

ECOLE NATIONALE
DU JEU ET DES MEDIAS
INTERACTIFS NUMERIQUES



Shading Language

Alexandre TOPOL

OpenGL Shading Language (GLSL)

- Apparition après Cg
- Par soucis de standardisation et de clarification dans la jungle des langages de shaders

- Le résultat fut OpenGL Shading Language, qui fait partie du standard OpenGL 2.0 (octobre 2004)

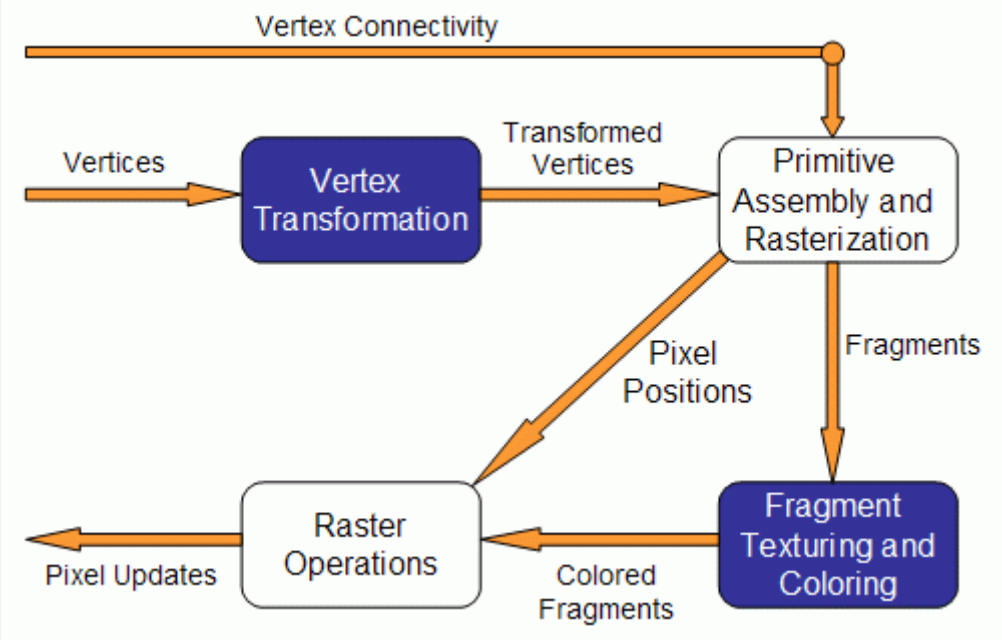
- GLSL est souvent prononcé "*GLslang*"

- GLSL et Cg sont très similaires
- GLSL est cependant plus proche d'OpenGL
- Et donc plus facile à utiliser/intégrer

- Mais ... Cg fonctionne également avec Direct3D

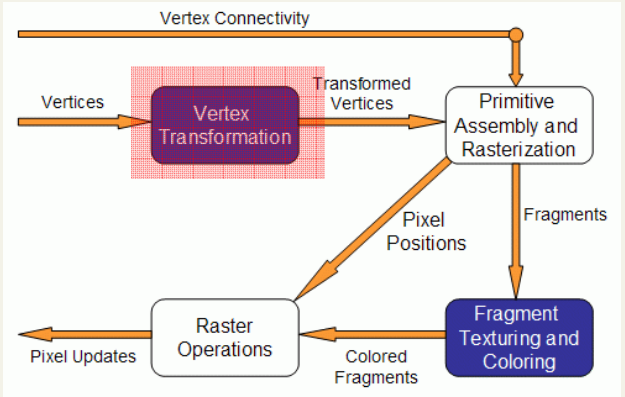
Fixed Function Pipeline (FFP)

- HLSL, GLSL et Cg ne sont “que” des interfaces pour programmer le pipeline 3D
- Une n^{ième} vue simplifiée du pipeline fixe



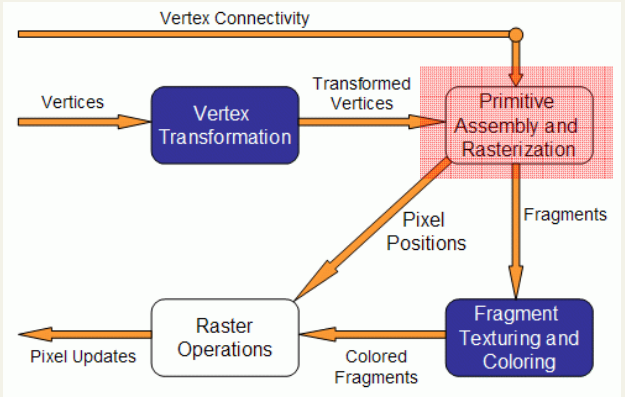
FFP – *Vertex Transformation*

- Un vertex (sommet) est un ensemble d'attributs donnant une position dans l'espace, une couleur, une normale, des coordonnées de texture, ...
- L'entrée de cette étape est un unique vertex
- Opérations effectuées :
  - Transformation de la position du sommet
  - Calcul d'illumination du sommet
  - Génération et transformation des coordonnées de texture
- Ces opérations sont effectuées pour tous les sommets de la scène pris un par un



FFP – *Primitive Assembly and Rasterization*

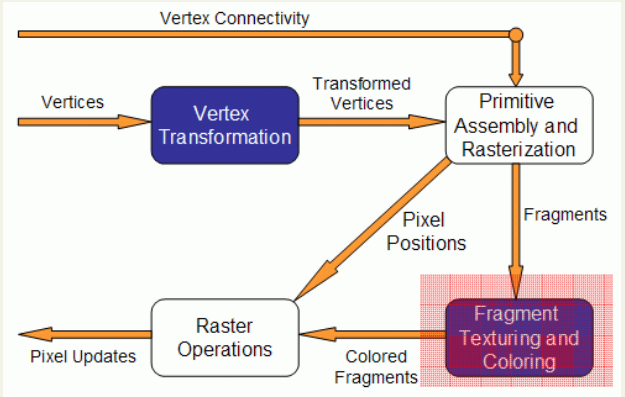
- Entrées : les sommets transformés et les informations de connectivité
- Op 1 : cloturage et test de vue
  (*clipping* et *back face culling*)
- Op 2 : Le coloriage (*rasterization*) détermine les fragments, et les positions des pixels de la primitive
- Sortie :
  - position des fragments dans le *frame buffer*
  - attributs interpolés pour chaque fragment



FFP – *Fragment Texturing and Coloring*

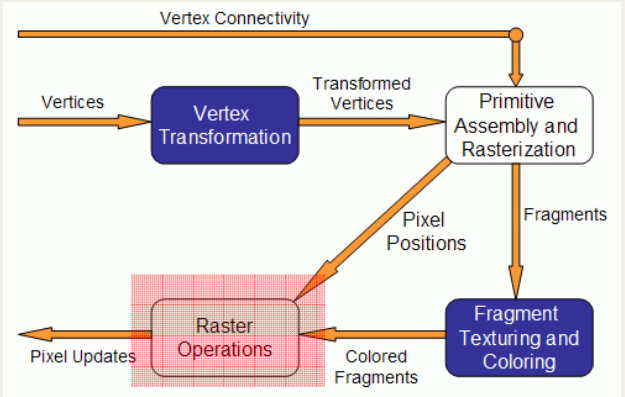
- En entrée : informations de fragment interpollées
- Une couleur a déjà été calculée à lors de l'étape précédente par interpolation
- Elle peut être combinée avec un texel (un élément de texture)
- Les coordonnées de texture ont également été interpolées lors de l'étape précédente.
- Application du texel au fragment

- En sortie : une couleur et une profondeur pour chaque fragment
- Etape effectuée pour chaque fragment pris un par un

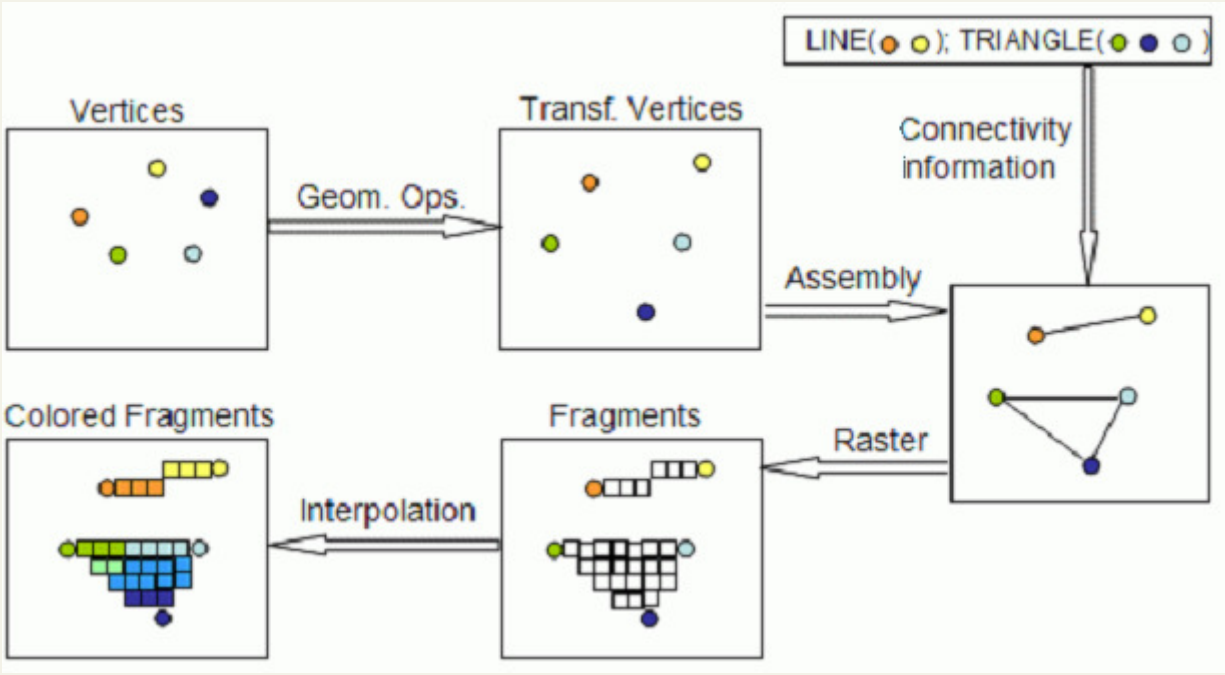


FFP – *Raster Operations*

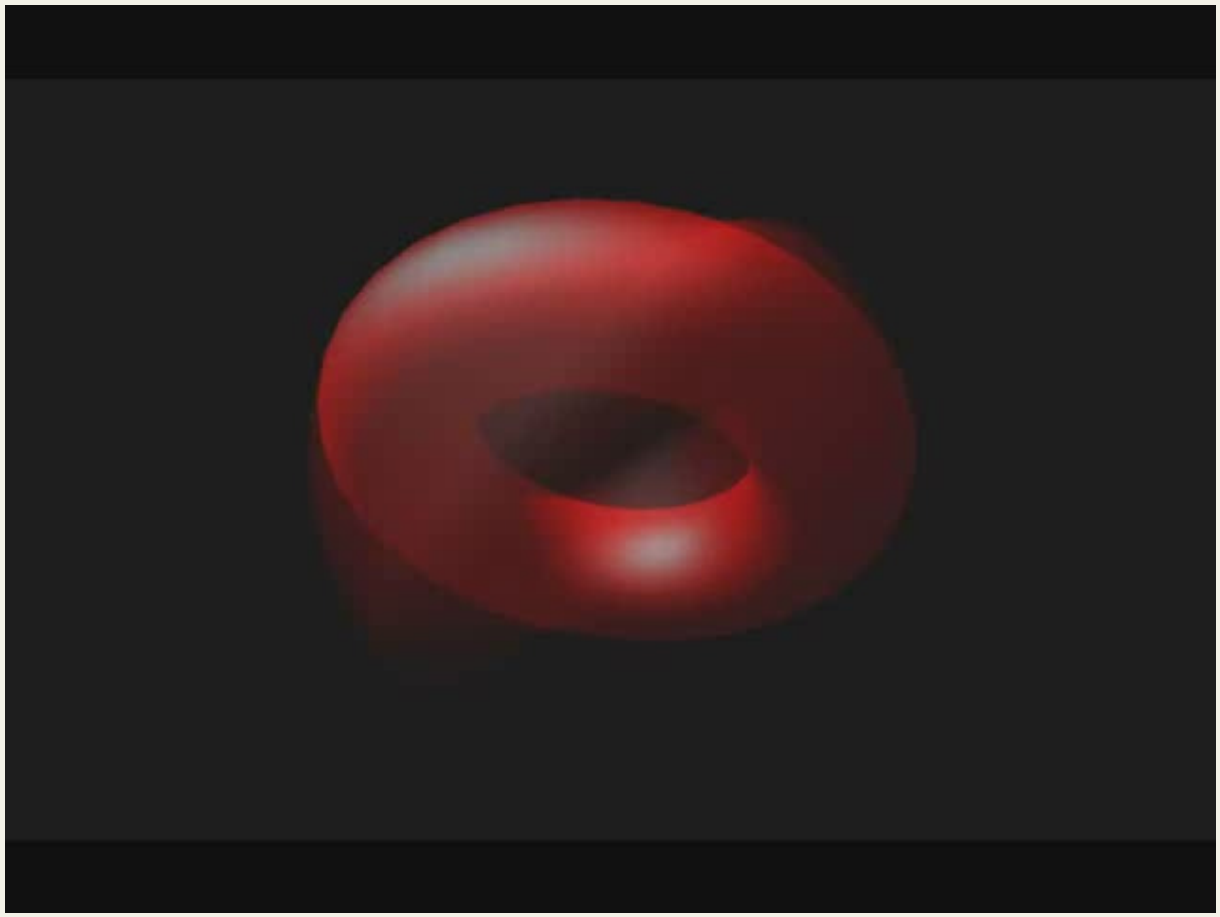
- En entrée :
  - Position des pixels (dans le viewport en 2D)
  - Couleur et profondeur des fragments
- Opérations :
  - Scissor test
  - Alpha test
  - Stencil test
  - Depth test



FFP – Un résumé en image



Exemples de rendu avec le FFP

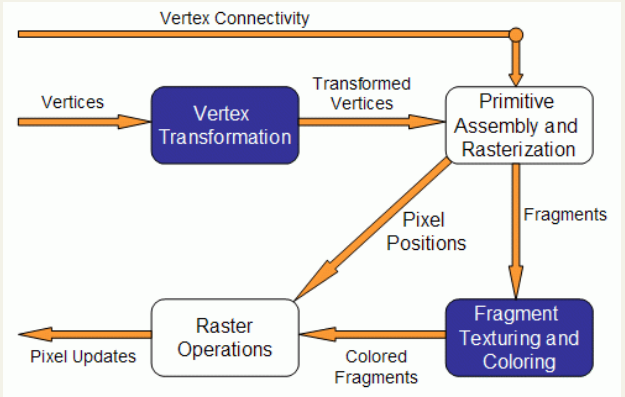


Du FFP au PFP

Du FFP au PFP

- Étape de Vertex Transformation ➔ vertex shaders
- Étape de Fragment Texturing and Coloring ➔ fragment shaders
- Étape de Primitive assembly and rasterization ➔ geometry shaders

- Puisqu'on remplace des fonctionnalités fixées dans le pipeline par des shaders programmables, on ne parle plus d'étapes
- mais plutôt de vertex processors, de geometry processors et de fragment processors

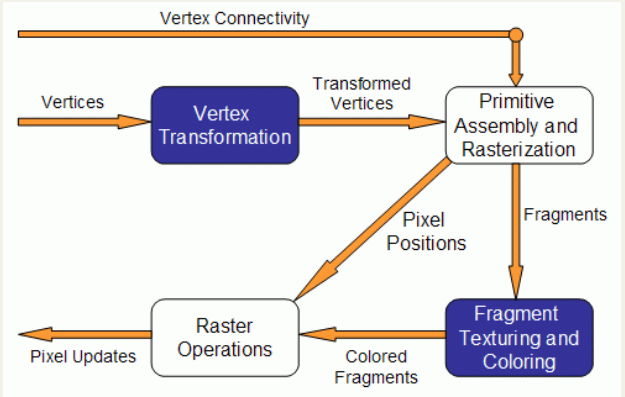


Vertex Processors

- En entrée : les données du vertex (sa position, sa couleur, sa normale, etc) en fonction de ce qui est envoyé depuis le code OpenGL
- Un exemple de code OpenGL qui envoie des données au vertex processor :

```
glBegin(...);
glColor3f(0.2,0.4,0.6);
glVertex3f(-1.0,1.0,2.0);
glColor3f(0.2,0.4,0.8);
glVertex3f(1.0,-1.0,2.0);
glEnd();
```

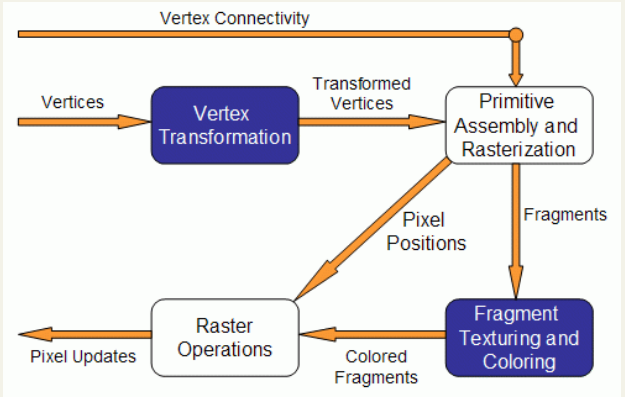
Différences par rapport à un code classique ??



Vertex Processors

```
glBegin(...);
glColor3f(0.2,0.4,0.6);
glVertex3f(-1.0,1.0,2.0);
glColor3f(0.2,0.4,0.8);
glVertex3f(1.0,-1.0,2.0);
glEnd();
```

- A première vue, et sans autres informations : aucune différence
- Mais, si préalablement un programme a été linké **et** activé, alors ces appels sont dirigés vers ce programme
- Sinon (si pas de programme activé) c'est bien le pipeline fixe qui est exécuté



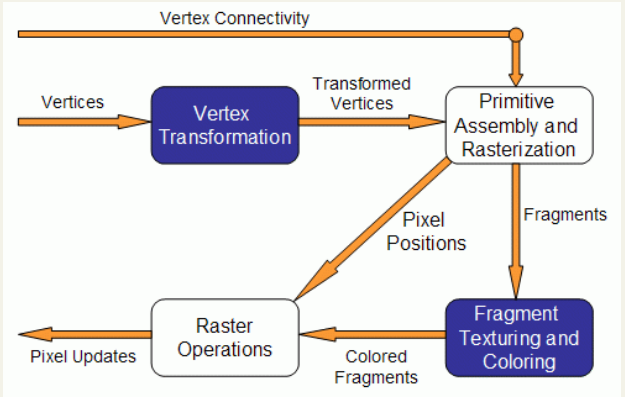
Vertex Processors

- Dans les vertex shaders, les opérations classiques à effectuer sont :
  - Transformation de la position du vertex en utilisant les matrices de vue et de projection
  - Transformation de la normale et si nécessaire sa normalisation
  - Génération et transformation des coordonnées de textures
- Illumination du sommet ou calcul de valeurs pour l'illumination pour chaque pixel
  - f(couleur, normale)

Flat shading

Gouraud shading (smooth)

- Un vertex shader peut ne rien faire
- Mais peu importe ce qu'il fait, s'il est défini, l'étape de fonctionnalité fixe sera remplacée



Vertex Processors

- Le processeur de sommets opère sur les sommets pris individuellement et n'a aucune information sur leur connectivité.
  - Aucune opération qui requiert une connaissance topologique ne peut être faite ici
  - Par exemple : pas de programmation du back face culling

⇒ **Geometry shader**

- Le vertex shader doit au moins calculer une valeur : *gl_Position*
  - La plupart du temps celà se fait en utilisant les matrices de vue et de projection
- Un processeur de sommets a accès aux états d'OpenGL
  - Du coup il peut utiliser l'illumination et les matériaux définis
- Un processeur de sommets peut accéder aux textures (pas sur toutes les cartes)
- Il ne peut pas accéder au frame buffer

Vertex Processors

- Exemple

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
  // Transforming The Vertex
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  // Transforming The Normal To ModelView-Space
  normal = gl_NormalMatrix * gl_Normal;

  // Transforming The Vertex Position To ModelView-Space
  vec4 vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex;

  // Calculating The Vector From The Vertex Position To The Light Position
  vertex_to_light_vector = vec3(gl_LightSource[0].position - vertex_in_modelview_space);
}
```


Vertex *Vs* Geometry Processors

- Ils reçoivent les mêmes types de données en entrée
- Ils peuvent générer les mêmes types en sortie
- La différence c'est que le geometry shader peut accéder à plusieurs sommets (ceux de la géométrie en cours) et générer de nouveaux sommets et primitives
- Le nombre de sommets de la géométrie courante gl_VerticesIn
- EmitVertex() permet de générer/d'ajouter un nouveau sommet à la géométrie courante (si aucun emitVertex alors aucun sommet)
- EndPrimitive() permet d'envoyer un ensemble de sommet à la rasterization (optionnel si une seule géométrie)

Geometry Processors

- Le geometry processor peut recevoir un certain type de géométrie
  - Types possibles = GL_POINTS, GL_LINES, GL_LINES_ADJACENCY_EXT, GL_TRIANGLES or GL_TRIANGLES_ADJACENCY_EXT
- Le geometry processor peut générer en sortie un autre type de geométrie
  - Types possibles = GL_POINTS, GL_LINE_STRIP or GL_TRIANGLE_STRIP
- Cela se spécifie lorsque l'on attache le geometry shader au programme (et pas par le glBegin)
- Attention ! Le nombre de sommets émis est limité en fonction du type de sortie

Geometry Processors

- Example

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main()
{
  for(int i = 0; i < gl_VerticesIn; ++i)
  {
    gl_Position = gl_PositionIn[i];
    EmitVertex();
  }
  EndPrimitive();   // fin de la primitive courante

  // génération de nouvelles géométries
  vec4 vertex;
  for(int i = 0; i < gl_VerticesIn; ++i)
  {
    vertex = gl_PositionIn[i];
    vertex.z = -vertex.z;
    gl_Position = vertex;
    EmitVertex();
  }
  EndPrimitive();
}
```

Quand on progamme un GS, on doit programmer également un VS. On utilise souvent le VS minimal suivant :

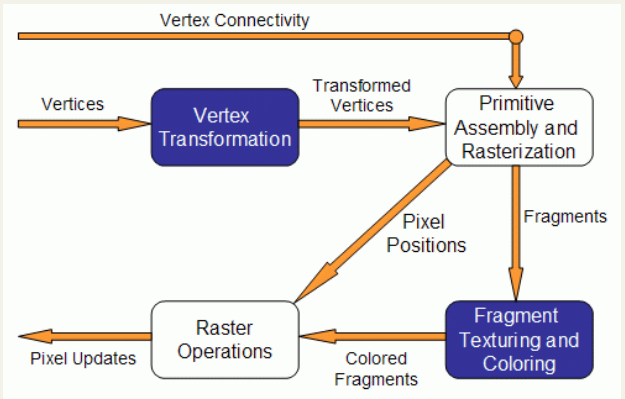
```
Void main() {
  gl_Position=ftransform();
}
```

Geometry Processors

Fragment Processors

- Entrées : les valeurs interpollés calculées lors de l'étape précédente du pipeline
  - C-a-d. positions des sommets, leur couleur, leur normale, etc...
- + les uniforms et les const accessibles dans tous les shaders
- Dans le vertex processor les valeurs sont calculées pour chaque sommet. Ici, ce sont des interpolations sur les fragments
- Quand on écrit un fragment/pixel shader, ça remplace l'étape *Fragment Texturing & Coloring* du pipeline fixe
- Le programmeur doit coder tous les effets graphiques

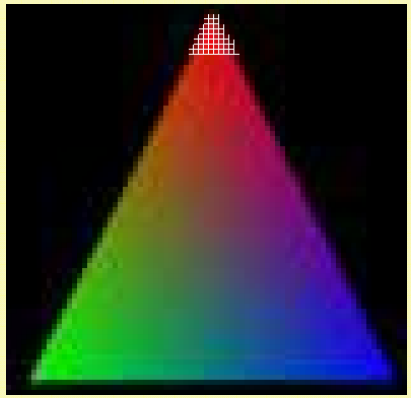
Un fragment shader a 2 options en sortie :
- Rejeter le fragment (*discard*)➔ aucune sortie
- Calculer soit *gl_FragColor* (la couleur finale du fragment), soit *gl_FragData* quand le rendu se destine à plusieurs cibles



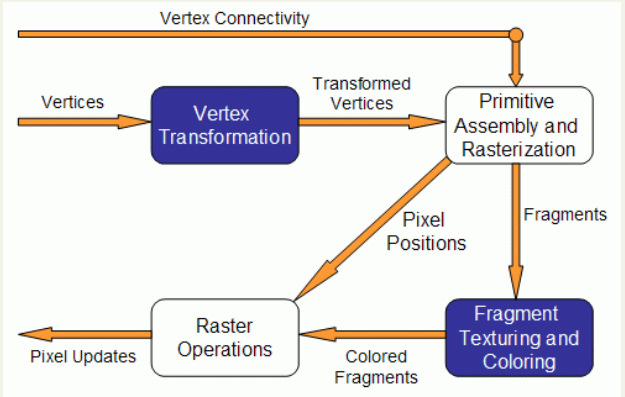
ECOLE NATIONALE
DU JEU ET DES MEDIAS
INTERACTIFS NUMERIQUES

Fragment Processors

- Le fragment processor opère sur les fragments pris un par un (*i.e.* Il n'a aucune information sur les fragments voisins)
- Le shader a accès aux états de la machine OpenGL
  - Un fragment shader a accès en lecture seule aux coordonnées du pixel.
- La profondeur est accessible en lecture/écriture
- Le fragment shader n'a pas accès au *frame buffer*
- Les operations telles que le blending ont lieu après l'exécution du fragment shader



Triangle
~3,042 pixels
Chaque pixel est traité
par le fragment shader
à chaque trame



Fragment Processors

- Exemple

```
// Couleur interpolées pour le pixel courant
varying vec4 color;

Void main ( void ) {
  gl_FragColor = vec4(color);
  return;
}
```


Utilisation de GLSL

- Si vous possédez un driver OpenGL 2.0, GLSL est inclu
- Sinon, vous aurez besoin des deux extensions suivantes
  - GL_ARB_fragment_shader
  - GL_ARB_vertex_shader

- Au niveau syntaxique, la seule différence entre les 2 solutions est la présence ou non de la constante symbolique “ARB” dans les noms de fonctions et de flags. Exemples :

OpenGL < 2.0	OpenGL >= 2.0
`glShaderSourceARB`	`glShaderSource`
`glCompileShaderARB`	`glCompileShader`
`glCreateShaderObjectARB`	`glCreateShader`
`GL_VERTEX_SHADER_ARB`	`GL_VERTEX_SHADER`

Utilisation de GLSL

- Pour OpenGL < 2.0, on passe par la librairie GLEW pour gérer les extensions :

```
#include <GL/glew.h>
#include <GL/glut.h>

void main(int argc, char **argv) {
  glutInit(&argc, argv);
  [...]

  glewInit();

  if (GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader)
    printf("Ready for GLSL\n");
  else {
    printf("Not totally ready :( \n");
    exit(1);
  }

  setShaders();

  glutMainLoop();
}
```


Utilisation de GLSL

- Pour OpenGL >= 2.0, GLEW ne nous sert qu'à tester la version d'OpenGL :

```
#include <GL/glew.h>
#include <GL/glut.h>

void main(int argc, char **argv) {
  glutInit(&argc, argv);
  [...]

  glewInit();

  if (glewIsSupported("GL_VERSION_2_0"))
    printf("Ready for OpenGL 2.0\n");
  else {
    printf("OpenGL 2.0 not supported\n");
    exit(1);
  }

  setShaders();

  glutMainLoop();
}
```


Utilisation de GLSL

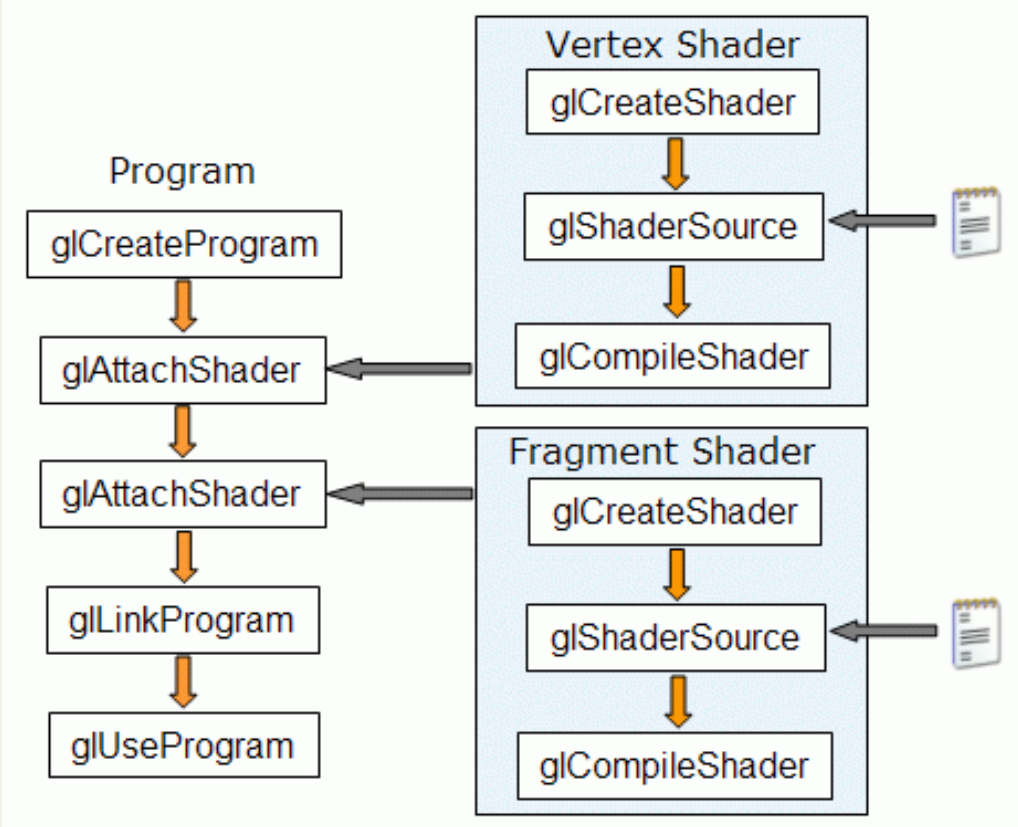
- On peut également utiliser GLee
  - Pour tester la version d'OpenGL

```
if (GLEE_VERSION_2_0) // GL2.0 dispo ?
{
  glBlendColor(...); // ok pour les fonctions OpenGL 2.0
}
```

  - Pour tester la présence d'une extension

```
#include <gl\GLee.h> // (pas besoin de spécifier gl.h)
[...]
if (GLEE_ARB_multitexture) {  // multitexture présent
  glMultiTexCoord2fARB(...);  // ok
}
else {
  // eh non ... Ici on code l'alternative
}
```


Le processus général



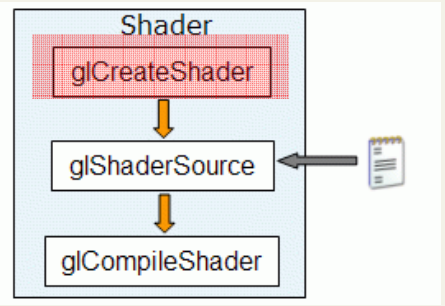
On ne donne par la suite que les exemples en OpenGL 2.0.

Création d'un shader

- La première étape est de créer un objet qui jouera le rôle d'un conteneur de shaders. La fonction de création d'un tel objet renvoie un handle

```
GLuint glCreateShader(GLenum shaderType);
Paramètres:
     shaderType – GL_VERTEX_SHADER ou GL_FRAGMENT_SHADER
```

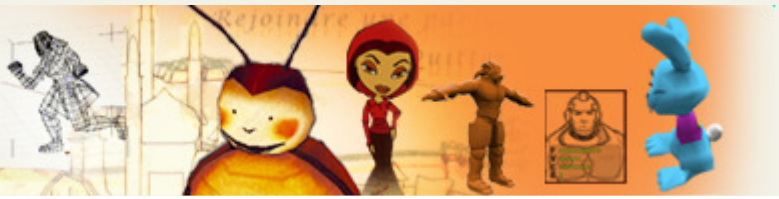
- Il n'y a pas de restriction sur le nombre de shaders mais un *main* par type de shader
- Un programme peut être composé de l'une des combinaisons suivantes :
  - Fonctions fixes (programme d'id 0)
  - VS + fonctions fixes G et F
  - VS+GS + fonctions fixes F
  - Fonctions fixes V et G + FS
  - VS + GS + FS

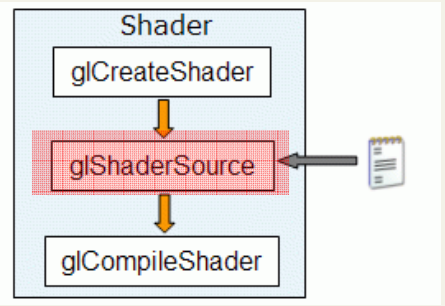


Création d'un shader

- La 2ième étape est d'ajouter un code source pour le shader
  - Le code source d'un shader est stocké sous la forme d'un tableau de chaînes de caractères (on peut utiliser également une unique chaîne de caractères)
- La syntaxe de la fonction est la suivante :

```
void glShaderSource(GLuint shader, int numOfStrings,
                                    const char **strings, int *lenOfStrings);

Paramètres :

  shader       - le handle du shader

  numOfStrings - nb de chaînes de caractères dans le tableau

  strings      - le tableau de chaînes de caractères

  lenOfStrings - un tableau avec les longueurs de chaque chaîne ou NULL signifiant que
les chaînes se terminent par NULL
```

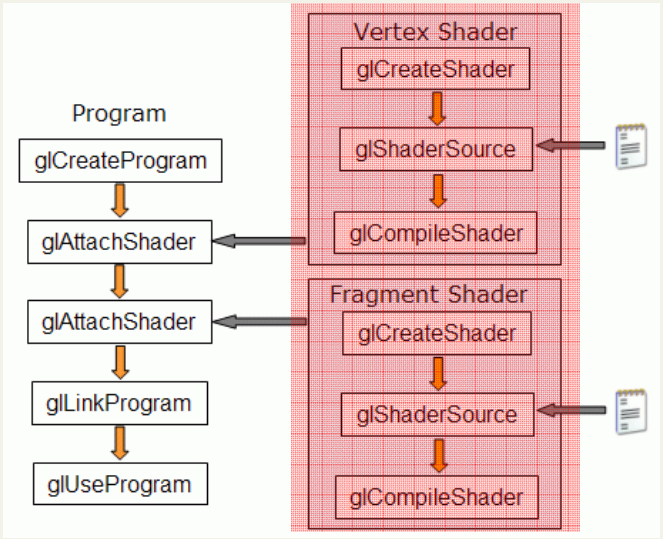


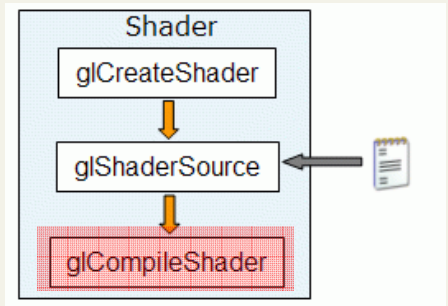


Création d'un shader

- La dernière étape : le shader doit être compilé
- La fonction à utiliser est la suivante :

```
void glCompileShader (GLuint program);
Paramètres:
     program – le handler vers le programme
```



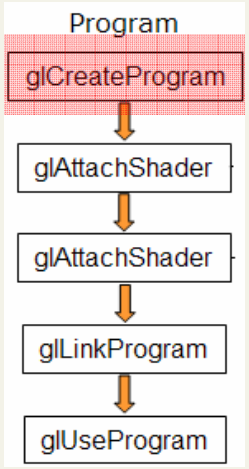


Création d'un programme

- 1ère étape : création du conteneur de programmes
- La fonction renvoie un handle vers le programme

```
GLuint glCreateProgram(void);
```

- On peut créer autant de programmes que l'on veut. En cours de rendu d'une trame, on peut changer de programme et même choisir les fonctions fixes du pipeline
  - Par exemple, on peut dessiner un objet avec un shader calculant les réflexions et réfractions et un autre avec les fonctionnalités du pipeline fixe d'OpenGL



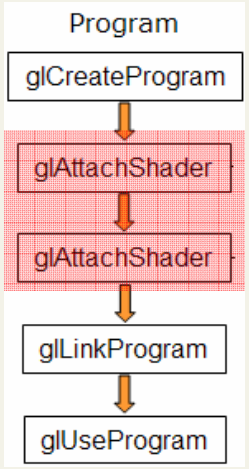
Création d'un programme

- La 2ème étape consiste à attacher le shader créé précédemment au programme
- Le shader n'a pas besoin à ce moment-ci ni d'être compilé ni d'avoir de code attaché. Seul le conteneur de shader est nécessaire à cette étape

```
void glAttachShader(GLuint program, GLuint shader);

Paramètres:
     program - handler du programme
     shader  - handler du shader à attacher
```

- Si besoin d'un triplet vertex/geometry/fragment shaders pour le traitement voulu, on doit attacher les 3 shaders au programme (3 appels à *attach*)
- On peut même attacher de nombreux shaders de même type (vertex, geométrie ou fragment) au même programme (multiples appels à *attach*)

Comme en C ou en Java, pour chaque type de shader, il ne peut y avoir qu'une unique fonction *main*. On peut attacher un shader à de multiples programs, pour utiliser ce même shader dans les différents programmes

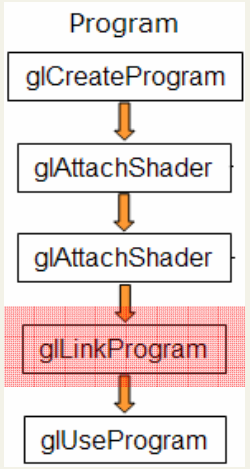


Création d'un programme

- L'étape finale est la liaison du programme. Pour cette étape, les shaders attachés doivent être compilés.

```
void glLinkProgram(GLuint program);
Paramètres:
     program – handler du programme
```

- Après ce linkage, le source d'un shader peut être modifié et recompilé

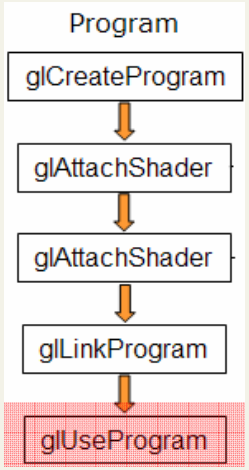


Utilisation d'un programme

- Il y a une fonction pour charger et utiliser un programme

```
void glUseProgram(GLuint prog);
Paramètres:
     prog – identifiant du programme à utiliser, ou 0 pour le FFP
```

- Chaque programme a un identifiant unique et l'on peut en avoir autant que l'on veut prêts à être utilisés (linkés et chargés)
- Si un programme est en cours d'utilisation et qu'il est linké à nouveau, il sera automatiquement remis en utilisation. Donc ce n'est pas nécessaire d'appeler cette fonction une nouvelle fois.
- Si le praramètre est 0 alors c'est le pipeline fixe qui sera utilisé



Initialisation des shaders

- Voici un exemple de fonction pour charger 3 shaders et les utiliser :

```
void setShaders() /* GLuint p,f,v, g déclarés en global */
{
  char *vs,*fs,*gs;
  const char * vv = vs, * ff = fs, * gg = gs;

  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);
  g = glCreateShader(GL_GEOMETRY_SHADER_EXT);

  vs = textFileRead("toon.vert");
  fs = textFileRead("toon.frag");
  gs = textFileRead("toon.geom");

  glShaderSource(v, 1, &vv, NULL);
  glShaderSource(f, 1, &ff, NULL);
  glShaderSource(g, 1, &gg, NULL);

  free(vs);  free(fs); free(gs);
```

textFileRead est une fonction permettant de stocker le contenu d'un fichier texte dans une chaîne de caractères

Initialisation des shaders

- Voici un exemple de fonction pour charger 3 shaders et les utiliser :

```
    glCompileShader(v);
    glCompileShader(f);
    glCompileShader(g);

    p = glCreateProgram();

    glAttachShader(p,v);
    glAttachShader(p,f);
    glAttachShader(p,g);

    glProgramParameteriEXT(p,GL_GEOMETRY_INPUT_TYPE_EXT,GL_LINES);
    glProgramParameteriEXT(p,GL_GEOMETRY_OUTPUT_TYPE_EXT,GL_LINE_STRIP);

    int temp;
    glGetIntegerv(GL_MAX_GEOMETRY_OUTPUT_VERTICES_EXT,&temp);
    glProgramParameteriEXT(p,GL_GEOMETRY_VERTICES_OUT_EXT,temp);

    glLinkProgram(p);
    glUseProgram(p);
}
```


Nettoyage

- Une fonction pour détacher un shader d'un programme :

```
void glDetachShader(GLuint program, GLuint shader);
Paramètres :
     program - le program duquel on détache.
     shader  - le shader à détacher
```

- Seuls les shaders détachés peuvent être effacés
- Pour effacer un shader on utilise la fonction suivante :

```
void glDeleteShader(GLuint shader);
Paramètre :
     shader - le shader à effacer
```


Récupération des erreurs

- C'est assez difficile de debugger un shader (pas de printf, pas de debugger pour l'instant)
- Les erreurs de compilation et/ou de liaison peuvent être récupérées dans les logs

```
void glGetShaderInfoLog(GLuint object, int maxLen, int *len, char *log);
void glGetProgramInfoLog(GLuint object, int maxLen, int *len, char *log);

Paramètres :
    object - identifiant de l'object (shader ou programme)
    maxLen - le nombre max de caractères à récupérer dans les logs.
    len    - renvoie la taille du log récupéré.
    log    - les logs
```

- Il faut donc connaître la longueur des logs avant récupération pour ne pas en perdre des bouts

```
void glGetShaderiv(GLuint object, GLenum type, int *param);
void glGetProgramiv(GLuint object, GLenum type, int *param);

Avec type = GL_INFO_LOG_LENGTH.
```

Récupération des erreurs

- Exemple de routines pour afficher les logs pour un programme (quasiment la même chose pour un shader) :

```
void printProgramInfoLog(GLuint obj)
{
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;

    glGetProgramiv(obj, GL_INFO_LOG_LENGTH,&infologLength);

    if (infologLength > 0)
    {
        infoLog = (char *)malloc(infologLength);
        glGetProgramInfoLog(obj, infologLength, &charsWritten, infoLog);
        printf("%s\n",infoLog);
        free(infoLog);
    }
}
```


Types de données

- 3 types de données simples en GLSL :
  - float, bool, int
  - Mêmes possibilités qu'en langage C
- Les vecteurs ayant 2, 3 ou 4 composantes sont déclarés comme suit :
  - vec{2,3,4}: un vecteur de 2, 3,or 4 flottants
  - bvec{2,3,4}: vecteur de booléens
  - ivec{2,3,4}: vecteur d'entiers
- Les matrices carrées 2x2, 3x3 et 4x4 :
  - mat2
  - mat3
  - mat4

Types de données

- Un ensemble de types spéciaux est disponible pour accéder aux textures, ils sont appelés *sampler* :
  - sampler1D – pour les textures 1D
  - sampler2D – pour les textures 2D
  - sampler3D – pour les textures 3D
  - samplerCube – pour les textures cube map
- Les tableaux peuvent être déclarés comme en C mais ils ne peuvent pas être initialisés lors de leur création. L'accès à un élément d'un tableau se fait également comme en C
- Les structures sont également supportées avec la même syntaxe que le C :

```
struct dirlight
{
  vec3 direction;
  vec3 color;
};
```


Variables

- Déclaration des variables quasi identique au langage C

```
float a,b;          // 2 flottants (oui, commentaires comme en C)
int c = 2;          // c entier initialisé à 2
bool d = true;      // d booléen initialisé à true
```

- Différences : GLSL s'appuie fortement sur les constructeurs pour l'initialisation et la conversion explicite

```
float b = 2;        // incorrect, pas de conversion implicite
float e = (float)2; // incorrect, constructeur nécessaire pour la conversion
int a = 2;
float c = float(a); // correct. c = 2.0
vec3 f;             // déclaration de f étant un vec3
vec3 g = vec3(1.0,2.0,3.0); // déclaration et initialisation de g
```

- GLSL est assez flexible lors d'initialisation de variables avec d'autres variables

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


Variables

- Les matrices suivent ce même schéma

```
mat4 m = mat4(1.0)       // initialisation de la diagonale de la matrice avec 1.0
vec2 a = vec2(1.0,2.0);
vec2 b = vec2(3.0,4.0);
mat2 n = mat2(a,b);      // ordre pour la création des matrices = colonne majeure
mat2 k = mat2(1.0,0.0,1.0,0.0); // tous les éléments sont spécifiés
```

- Déclaration et initialisation de structures

```
struct dirlight {        // type definition
    vec3 direction;
    vec3 color;
};
dirlight d1;
dirlight d2 = dirlight(vec3(1.0,1.0,0.0),vec3(0.8,0.8,0.4));
```


Variables

- L'accès à un vecteur peut se faire en utilisant des lettres mais également avec un sélecteur standand C

```
vec4 a = vec4(1.0,2.0,3.0,4.0);
float posX = a.x;
float posY = a[1];
vec2 posXY = a.xy;
float depth = a.w;
```

- Lettres autorisées pour les vecteurs :
  - x, y, z, w pour accéder aux composantes d'une vecteurs,
  - r, g, b ,a pour les composantes de couleurs,
  - s, t, p, q pour les coordonnées de textures.
- Pour les structures, le nom du champ peut bien sûr être utilisé comme en C :

```
d1.direction = vec3(1.0,1.0,1.0);
```


Qualifieurs de variable

- Les qualifieurs donnent un sens spécial à une variable. En GLSL, les qualifieurs suivants sont disponibles :
  - `const` : la déclaration est transformée en constante au moment de la compilation
  - `attribute` : (uniquement utilisable dans les vertex et geometry shaders et en lecture seule) variable dont la valeur peut changer d'un sommet à l'autre. Les valeurs sont passées depuis l'application OpenGL
  - `uniform` : (utilisable dans les vertex/fragment shaders en lecture seule) variables qui peuvent changer d'une primitive à l'autre (mais dont la valeur ne peut être modifiée dans glBegin ... glEnd)
  - `varying` : utilisé pour les données interpolées entre un vertex shader et un fragment shader. Ecriture dans le vertex shader, et lecture seule dans le fragment shader.

Instructions de contrôle

- Instructions de contrôle : presque comme le C

```
if (bool expression)
     ...
else
    ...

for (initialization; bool expression; loop expression)
    ...

while (bool expression)
   ...

do
  ...
while (bool expression)
```

Seulement les *if* sont utilisables sur certains GPU

Instructions de contrôle

- Quelques *jumps* sont également définis :
  - `continue` : disponible dans les boucles pour sauter directement à la prochaine itération
  - `break` : disponible dans les boucles pour sortir de la boucle
  - `discard` : ne peut être utiliser que dans les fragment shaders. Celà provoque l'arrêt immédiat du traitement pour le fragment en cours, sans écriture ni dans le frame buffer, ni dans le ZBuffer

Fonctions

- Comme dans un programme en C, un shader en GLSL est structuré en fonctions
- Chaque type de shader doit avoir au minimum une fonction principale déclarée avec la syntaxe suivante : `void main()`
- Les fonctions peuvent avoir un type de retour et utiliser le `return` pour passer leur résultat. Une fonction peut être `void`. Tout type de résultat est autorisé sauf les tableaux
- Des qualifieurs peuvent être utilisés pour les paramètres des fonctions :
  - in – pour une lecture seule
  - out – pour des sorties de la fonction (écriture seule). Si le return ne suffit pas
  - inout – pour les paramètres qui sont des entrées et sorties
  - Si aucun qualifieur n'est spécifié, par défaut le paramètre sera de type *in*

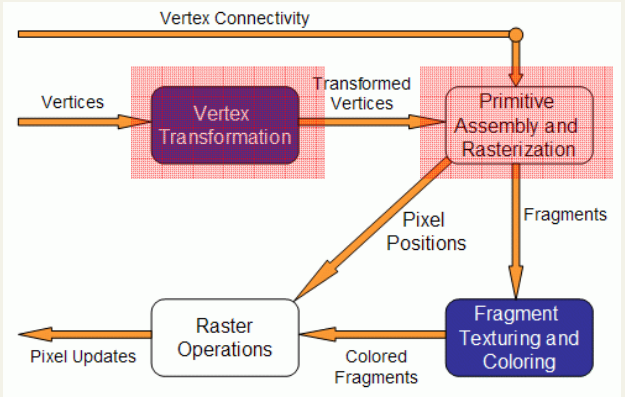
Fonctions

- Pour les détails :
  - Deux fonctions peuvent avoir un même nom si la liste de leurs paramètes est différente
  - Le comportement de la récursivité est indifini dans les spécification
- Exemple :

```
vec4 toonify(in float intensity)
{
        vec4 color;
        if (intensity > 0.98)
            color = vec4(0.8,0.8,0.8,1.0);
        else if (intensity > 0.5)
            color = vec4(0.4,0.4,0.8,1.0);
        else if (intensity > 0.25)
            color = vec4(0.2,0.2,0.4,1.0);
        else color = vec4(0.1,0.1,0.1,1.0);
        return(color);
}
```

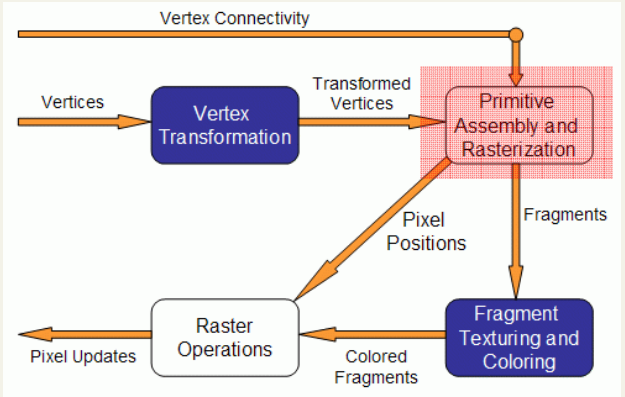

Varying Variables

- Prenons un exemple concret
  - OpenGL fait du smooth (gouraud) shading
  - Le coloriage de type Phong est d'une qualité visuelle supérieure mais demande des calculs plus complexes pour le GPU (et couteux par le FFP)
- L'Illumination prend place dans la phase de *vertex transformation* et le coloriage (l'interpolation des couleurs) dans la phase suivante
- Mais le coloriage de type Phong nécessite une illumination par fragment



Varying Variables

- Les variables qualifiées de *varying* sont interpolées à partir des sommets en utilisant les données topologiques, pendant l'étape de rasterization
- GLSL a des variables interpolées prédéfinies, comme les couleurs, les coordonnées de texture, etc
- Malheureusement, les normales (nécessaires pour l'illumination par fragment) n'est pas une variables interpolées prédéfinies ☹
- En GLSL, pour faire du coloriage de type Phong il faut donc construire une variable interpolée pour les normales

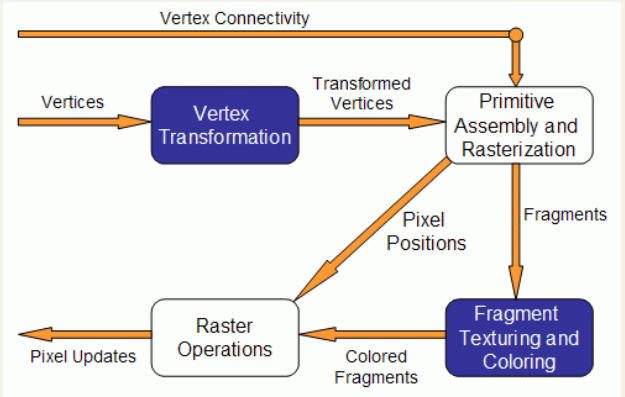


Varying Variables

- Définition d'une variable interpolée dans les vertex, geometry et fragment shaders

```
varying vec3 normal;
```

- Pour la n^{ième} fois :
  - Les variables interpolées doivent être affectées dans les vertex shaders
  - Les variables interpolées peuvent être lues uniquement dans les fragment shaders



Uniform Variables

- Elles fournissent un medium de communication entre le programme C/C++ et les shaders (ex: depuis quand l'obus a-t-il été tiré ?)
- La valeur d'une variable uniforme peut être changée uniquement par primitive, c-à-d qu'elle ne peut être changée à l'intérieur d'une paire *glBegin* / *glEnd*
- Elles sont donc intéressantes pour des valeurs qui restent constantes pour toute une primitive géométrique, toute une trame, ou toute une scène
- Elles peuvent être lues (mais non modifiées) dans les deux types de shaders

Uniform Variables

- Pour la manipulation d'une variable uniforme, la 1ère chose à faire est de récupérer la localisation mémoire de la variable
  - Cette information est uniquement disponible après linkage du programme. Avec certains drivers il se peut même qu'il faille utiliser le programme en question (*glUseProgram*)
- La fonction à utiliser est :

```
GLint glGetUniformLocation(GLuint program, const char *name);

Paramètres :

     program - identifiant du program

     name    - le nom de la variable

En retour, on récupère la localisation de la variable qui peut être utilisée
ensuite pour lui affecter une valeur
```


Uniform Variables

- Toute une famille d'instructions permet d'assigner une valeur
- Ci dessous, l'ensemble de fonctions est défini pour affecter des valeurs flottantes. Un ensemble similaire est disponible pour les entiers en remplaçant le 'f' par un 'i'

```
void glUniform1f(GLint location, GLfloat v0);
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);

GLint glUniform{1,2,3,4}fv(GLint location, GLsizei count, GLfloat *v);

Paramètres :
          location    - la localisation demandée précédemment
          v0,v1,v2,v3 - valeurs flottantes
          count       - le nombre d'éléments dans le tableau
          v           - un tableau de flottants
```


Uniform Variables

- On a accès également à un ensemble de fonctions pour modifiers les variables uniformes de type matrice

```
GLint glUniformMatrix{2,3,4}fvARB(GLint location, GLsizei count, GLboolean transpose,
GLfloat *v);

Paramètres:

   location  - la localisation préalablement demandée

   count     - le nombre de matrices. 1 si une unique matrice, n pour un tableau de n

   transpose - pour signifier l'ordre des données dans la matrice
               (1 indique l'ordre ligne majeure, 0 colonne majeure)

   v         - un tableau de flottants
```


Uniform Variables

- Les variables affectées avec ces fontions gardent leur valeur lusqu'à ce que le programme soit re-linké
- Lorsqu'un nouveau linkage est effectué, toutes les valeurs des variables uniformes sont mises à 0

Uniform Variables

- Un exemple :

Supposons que les variables suivantes sont utilisées dans le shader:

```
uniform float specIntensity;
uniform vec4 specColor;
uniform float t[2];
uniform vec4 colors[3];
```

Dans OpenGL, le code pour attribuer des valeurs aux variables :

```
GLint loc1,loc2,loc3,loc4;
float specIntensity = 0.98;
float sc[4] = {0.8,0.8,0.8,1.0};
float threshold[2] = {0.5,0.25};
float colors[12] = {0.4,0.4,0.8,1.0,
                    0.2,0.2,0.4,1.0,
                    0.1,0.1,0.1,1.0};

loc1 = glGetUniformLocation(p,"specIntensity");
glUniform1f(loc1,specIntensity);
loc2 = glGetUniformLocation(p,"specColor");
glUniform4fv(loc2,1,sc);
loc3 = glGetUniformLocation(p,"t");
glUniform1fv(loc3,2,threshold);
loc4 = glGetUniformLocation(p,"colors");
glUniform4fv(loc4,3,colors);
```


Variables attributs

- Ce type de variables permet également à un programme en C de communiquer avec les shaders
- Elles peuvent être mises à jour à tout instant mais sont en lecture seule dans les shaders
- Les variables attributs ont trait aux vertex shaders et sont inutiles dans un fragment shader
- Pour affecter une valeur à ces variables attributs il faut d'abord obtenir sa localisation mémoire (comme pour les uniforms)
  - Le programme doit être linké préalablement et certains drivers demanderont que le programme soit en utilisation

```
GLint glGetAttribLocation$(GLuint program,char *name);

Paramètres:
     program - identifiant du program.
     name    - nom de la variable
```

Variables attributs

- Comme pour les variables uniformes, un ensemble de fonctions sont données pour affecter des valeurs à ces variables attributs (remplacer ‘f’ avec ‘i’ donne les appels de fonction pour les entiers)

```
void glVertexAttrib1f(GLint location, GLfloat v0);
void glVertexAttrib2f(GLint location, GLfloat v0, GLfloat v1);
void glVertexAttrib3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glVertexAttrib4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);

ou

GLint glVertexAttrib{1,2,3,4}fv(GLint location, GLfloat *v);


Paramètres:
          location    - localisation de la variable
          v0,v1,v2,v3 - valeurs flottantes
          v           - un tableau de flottants
```


Variables attributs

- Un exemple :

Si dans le vertex shader on a :

```
attribute float height;
```

Dans le programme OpenGL on pourra faire :

```
GLint loc = glGetAttribLocation(p,"height");

glBegin(GL_TRIANGLE_STRIP);
glVertexAttrib1f(loc,2.0);
glVertex2f(-1,1);
glVertexAttrib1f(loc,2.0);
glVertex2f(1,1);
glVertexAttrib1f(loc,-2.0);
glVertex2f(-1,-1);
glVertexAttrib1f(loc,-2.0);
glVertex2f(1,-1);
glEnd();
```


Built-in variables

- Les 2 catégories attributs & uniformes
- Pour faciliter la programmation
- Les états OpenGL sont mappés sur des variables
- Des variables doivent avoir une valeur affectée, d'autres sont optionnelles

Special built-ins

- Vertex & geometry shader

```
vec4  gl_Position;      // must be written
vec4  gl_ClipPosition;  // may be written
float gl_PointSize;     // may be written
```

- Fragment shader

```
float gl_FragColor;     // may be written
float gl_FragDepth;     // may be read/written
vec4  gl_FragCoord;     // may be read
bool  gl_FrontFacing;   // may be read
```


Attributs

- Built-in

```
attribute vec4  gl_Vertex;
attribute vec3  gl_Normal;
attribute vec4  gl_Color;
attribute vec4  gl_SecondaryColor;
attribute vec4  gl_MultiTexCoordn;
attribute float gl_FogCoord;
```

- User-defined

```
attribute vec3  myTangent;
attribute vec3  myBinormal;
Etc…
```


Built-in Uniforms

```
uniform   mat4  gl_ModelViewMatrix;
uniform   mat4  gl_ProjectionMatrix;
uniform   mat4  gl_ModelViewProjectionMatrix;
uniform   mat3  gl_NormalMatrix;
uniform   mat4  gl_TextureMatrix[n];
...

struct gl_MaterialParameters {
  vec4  emission;
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```


Built-in Uniforms

```
struct gl_LightSourceParameters {
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  vec4  position;
  vec4  halfVector;
  vec3  spotDirection;
  float spotExponent;
  float spotCutoff;
  float spotCosCutoff;
  float constantAttenuation
  float linearAttenuation
  float quadraticAttenuation
};
Uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```


Built-in Varyings

```
varying   vec4  gl_FrontColor       // en écriture vs et gs
varying   vec4  gl_BackColor;       // en écriture vs et gs
varying   vec4  gl_FrontSecColor;   // en écriture vs et gs
varying   vec4  gl_BackSecColor;    // en écriture vs et gs

varying   vec4  gl_Color;           // en lecture dans le fs
varying   vec4  gl_SecondaryColor;  // en lecture dans le fs

varying   vec4  gl_TexCoord[];      // écriture vs, gs et lecture fs
varying   float gl_FogFragCoord;    // écriture vs, gs et lecture fs
```


Built-in functions

- Angles & Trigonometry
  - **radians, degrees, sin, cos, tan, asin, acos, atan**
- Exponentials
  - **pow, exp2, log2, sqrt, inversesqrt**
- Common
  - **abs, sign, floor, ceil, fract, mod, min, max, clamp**

Built-in functions

- Interpolations
  - **mix**(x,y,a) **x*(1.0-a) + y*a**)
  - **step**(edge,x) **x <= edge ? 0.0 : 1.0**
  - **smoothstep**(edge0,edge1,x)

```
t = (x-edge0)/(edge1-edge0);
t = clamp( t, 0.0, 1.0);
return t*t*(3.0-2.0*t);
```


Built-in functions

- Geometric
  - **length, distance, cross, dot, normalize, faceForward, reflect**
- Matrix
  - **matrixCompMult**
- Vector relational
  - **lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, any, all**

Built-in functions

- Texture
  - **texture1D, texture2D, texture3D, textureCube**
  - **texture1DProj, texture2DProj, texture3DProj, textureCubeProj**
  - **shadow1D, shadow2D, shadow1DProj, shadow2Dproj**
- Vertex
  - **ftransform**

Hello world

Vertex shader

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
attribute vec4 gl_Vertex;

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

ou

```
uniform mat4 gl_ModelViewProjectionMatrix
void main() {
  gl_Position =
gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

ou

```
void main() {
  gl_Position = ftransform();
}
```

Geometry shader

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main()
{
  for(int i = 0; i < gl_VerticesIn; ++i)
  {
    gl_Position = gl_PositionIn[i];
    EmitVertex();
  }
}
```

Fragment shader

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


Toon – vertex shader

```
varying vec3 normal, lightDir;

void main()
{
    lightDir = normalize(vec3(gl_LightSource[0].position));
    normal = normalize(gl_NormalMatrix * gl_Normal);

    gl_Position = ftransform();
}
```


Toon – fragment shader

```
varying vec3 normal, lightDir;

void main()
{
    float intensity;
    vec3 n;
    vec4 color;

    n = normalize(normal);
    intensity = max(dot(lightDir,n),0.0);

    if (intensity > 0.98)
          color = vec4(0.8,0.8,0.8,1.0);
    else if (intensity > 0.5)
          color = vec4(0.4,0.4,0.8,1.0);
    else if (intensity > 0.25)
          color = vec4(0.2,0.2,0.4,1.0);
    else
          color = vec4(0.1,0.1,0.1,1.0);

    gl_FragColor = color;
}
```


Tools

- OpenGL Extensions Viewer
  - http://www.realtech-vr.com/glview/download.html
- Simple Shaders
  - ogl2brick (http://developer.3dlabs.com/downloads/glslexamples/)
  - Hello GPGPU (http://www.gpgpu.org/developer/)
- ShaderGen
  - http://developer.3dlabs.com/downloads/shadergen/
- Shader data structures – Brook, glift
- Littérature recommandée – OpenGL RedBook, OpenGL OrangeBook, GPU Gems

Travaux Pratiques

- TODO :
  - Un loader de shaders (DONE)
  - Un vertex shader permettant de bruiter la position d'un sommet
  - Un fragment shader pour faire du dessin fil de fer
  - Un geometry shader permettant de créer un cube à partir d'un unique sommet
- A rendre … demain 12h30