Synthèse d'Images / Animation
Cartes Graphiques

Sébastien Barbier

Laboratoire Jean Kuntzmann/EVASION

Grenoble

sebastien.barbier@inrialpes.fr

Plan

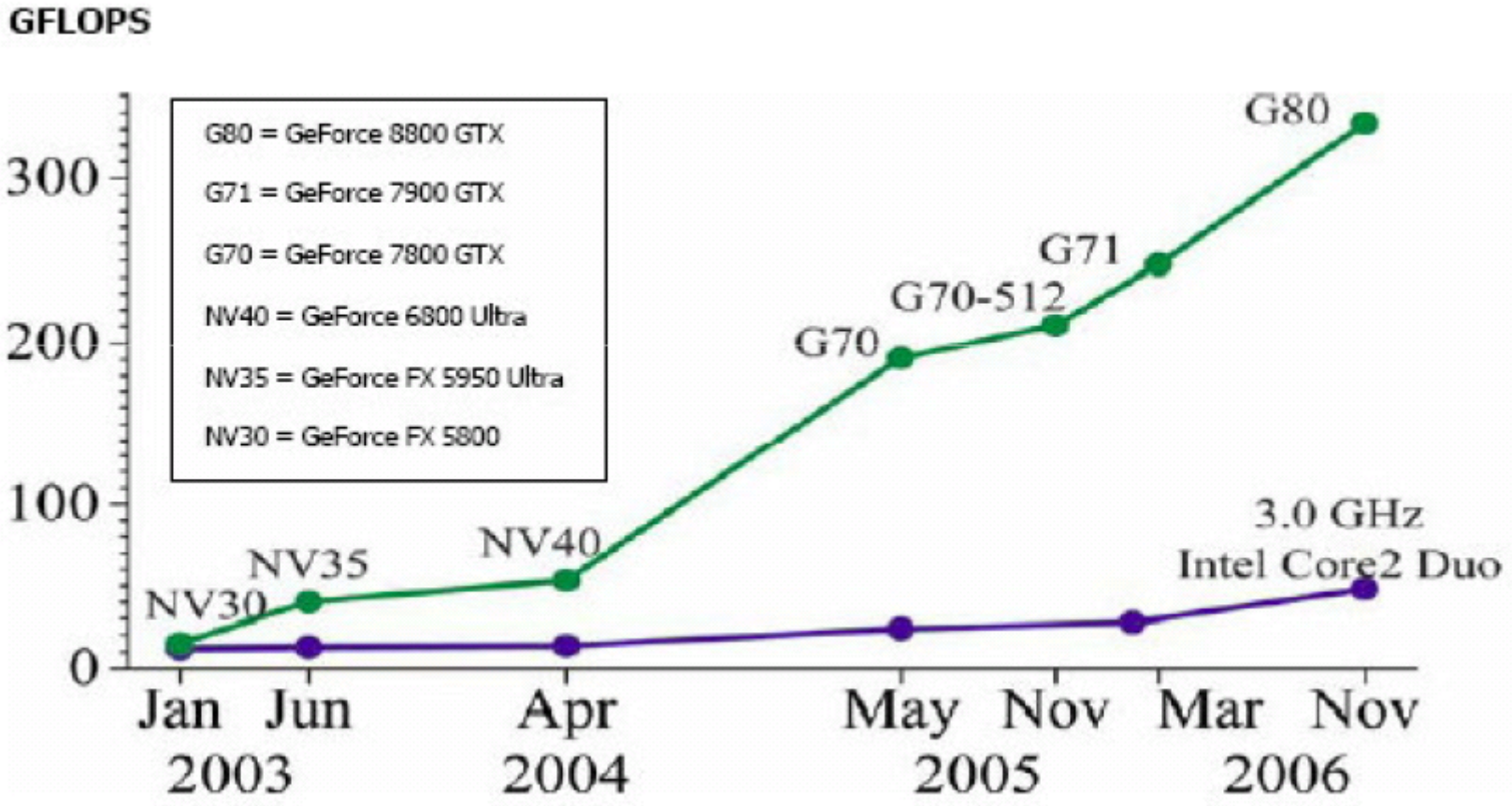
1. Historique
2. Fonctionnalités des cartes graphiques
3. GLSL
4. Communications CPU $\leftrightarrow$ GPU
5. GPGPU

Motivations

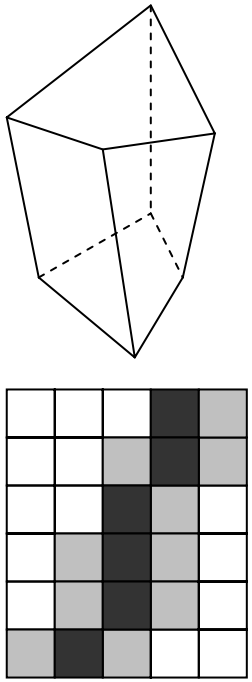
- Le CPU s'occupe :
  - Simulation physique, Intelligence Artificielle, Son, Réseau...
- Le GPU doit vérifier :
  - Accès mémoire rapide
    - Nombreux accès [vertices, normal, textures, ...]
  - Une bonne bande passante
    - Go/s au meilleur cas
  - Une grande force de calcul
    - Flops = Floating Point Operations [ADD, MUL, SUB, ...]
    - Illustration: matrix-vector products
      - (16 MUL + 12 ADD) x (#vertices + #normals) x fps =
      - (28 Flops) x (6.000.000) x 30 ≈ 5GFlops

Motivations

- Interactivité : 15-60 fps
- Haute Résolution

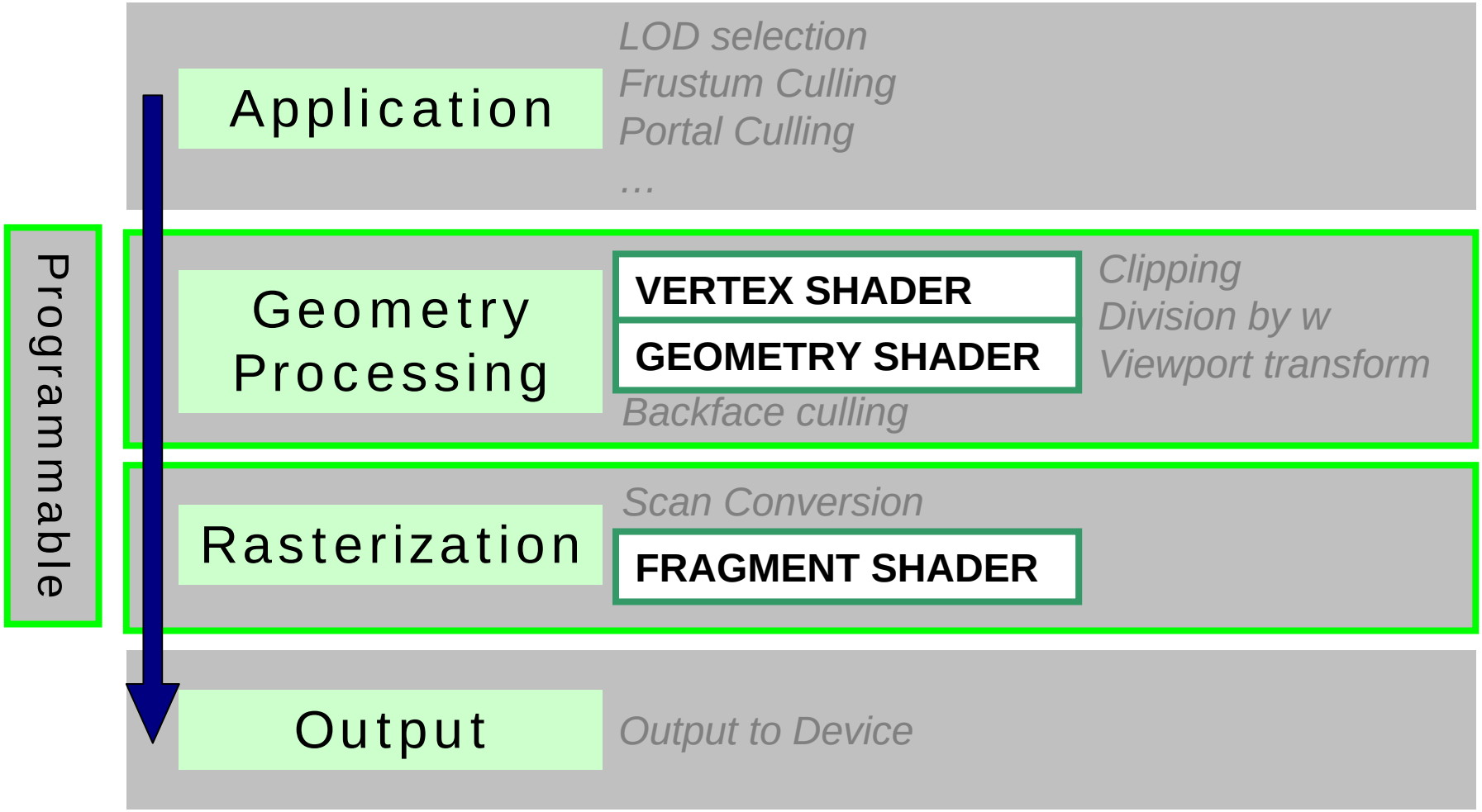


Pipeline Graphique

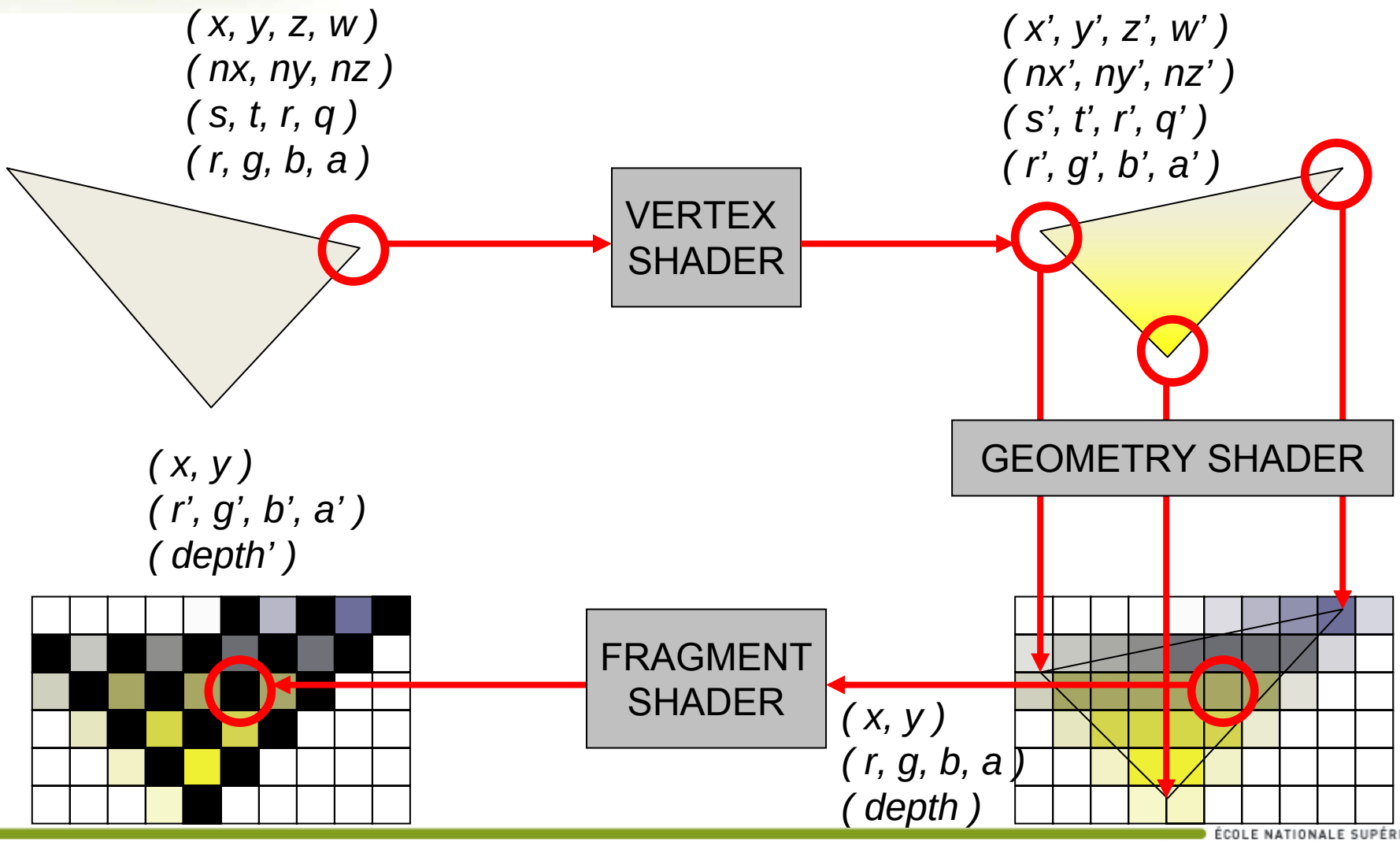


Application	LOD selection Frustum Culling Portal Culling …	
Geometry Processing	Modelview/Projection tr. Lighting Primitive Assembly Backface culling	Clipping Division by w Viewport transform
Rasterization	Scan Conversion Fragment Shading [Color and Texture interpol.] Frame Buffer Ops [Z-buffer, Alpha Blending,…]	
Output	Output to Device	

Pipeline Graphique



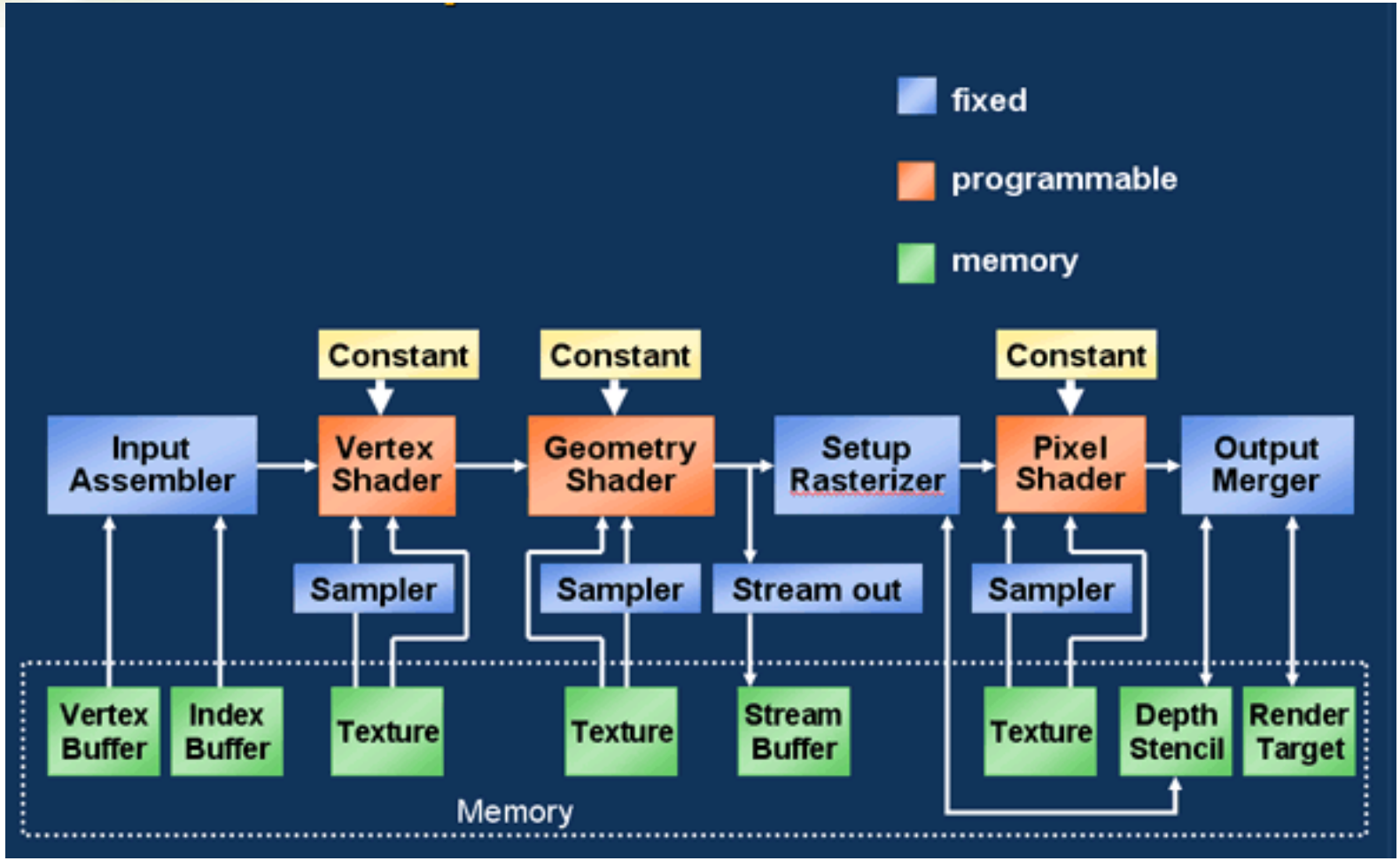
Shaders


(x, y, z, w)
(nx, ny, nz)
(s, t, r, q)
(r, g, b, a)
VERTEX SHADER
(x', y', z', w')
(nx', ny', nz')
(s', t', r', q')
(r', g', b', a')
GEOMETRY SHADER
(x, y)
(r', g', b', a')
(depth')
FRAGMENT SHADER
(x, y)
(r, g, b, a)
(depth)

Plan

1. Historique
2. Fonctionnalités des cartes graphiques
3. GLSL
4. Communications CPU $\leftrightarrow$ GPU
5. GPGPU

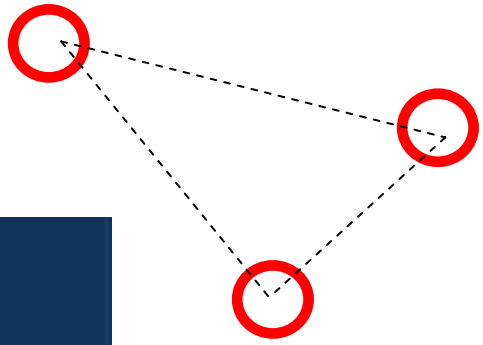
Pipeline Graphique

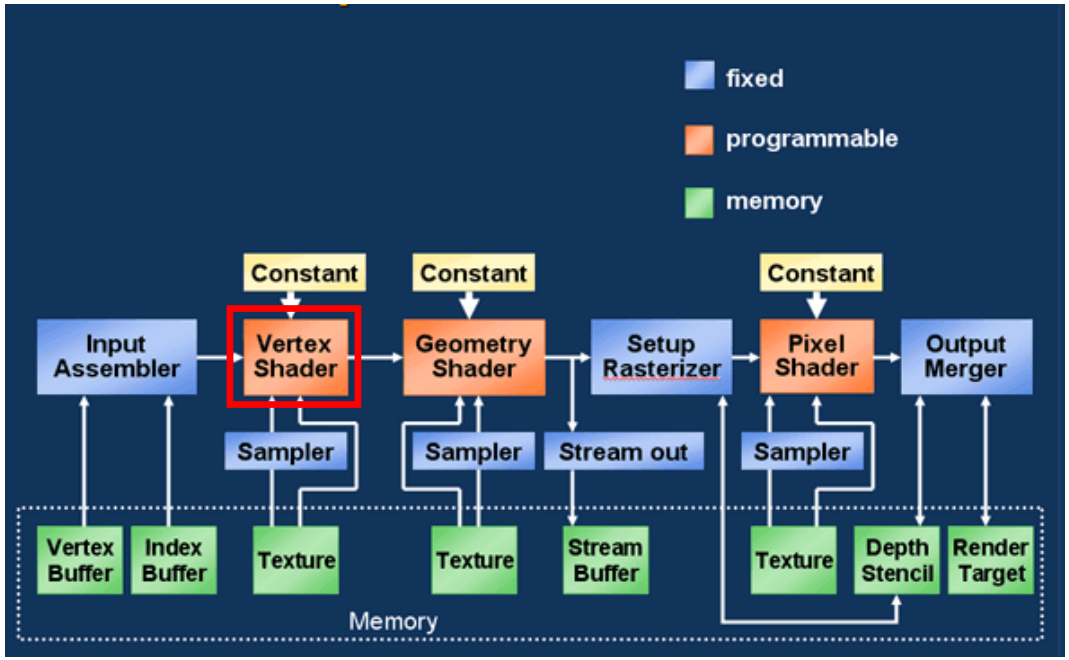


Fonctionnalités des cartes graphiques
Shaders

On agit au niveau **local** :

- Vertex shader : un sommet à la fois
  - On ne connaît pas les sommets voisins

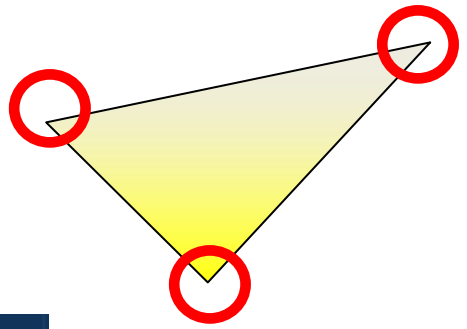


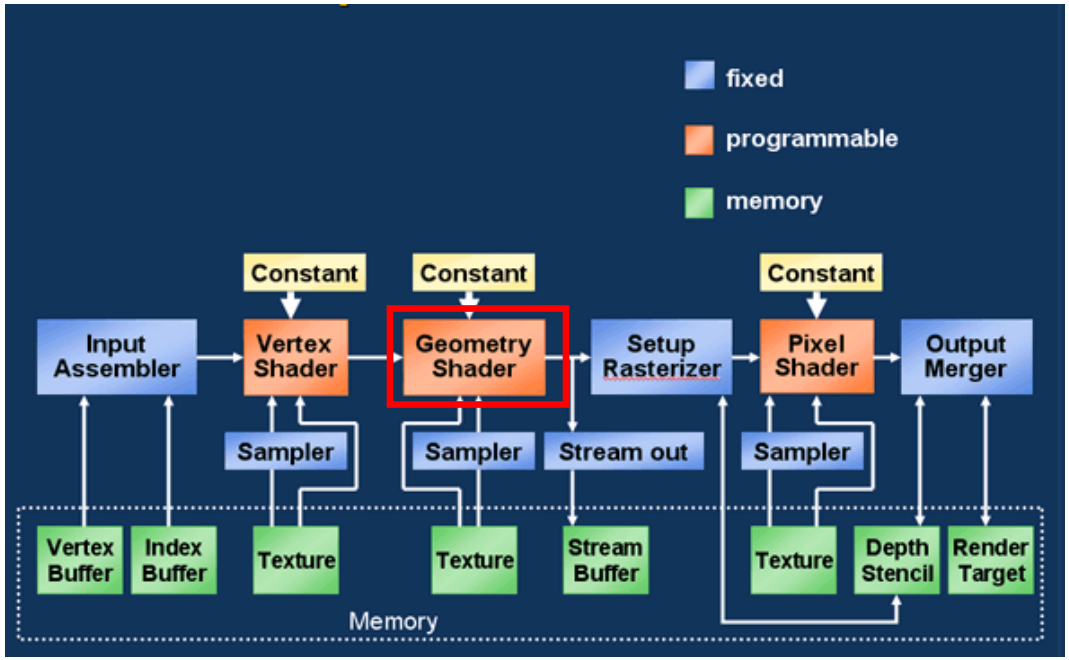


Fonctionnalités des cartes graphiques
Shaders

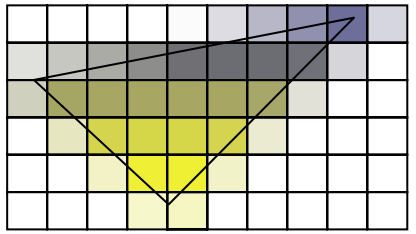
On agit au niveau **local** :

- Geometry shader : une primitive à la fois
  - On ne connait que les sommets de la primitive courante
  - On peut aussi connaître ses voisins



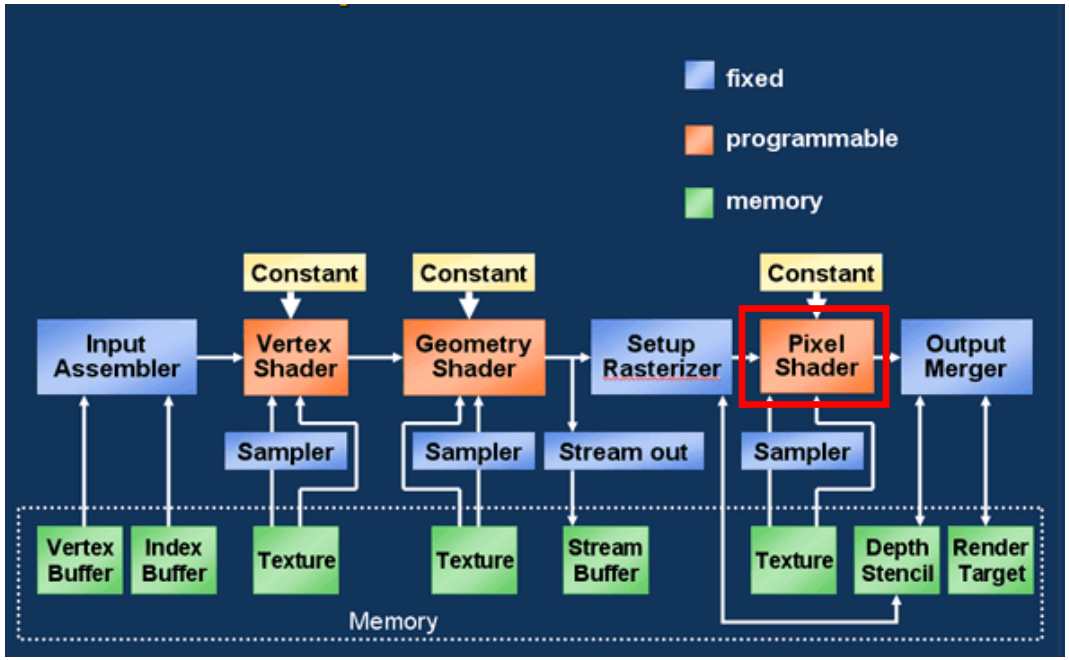


Fonctionnalités des cartes graphiques
Shaders



On agit au niveau **local** :

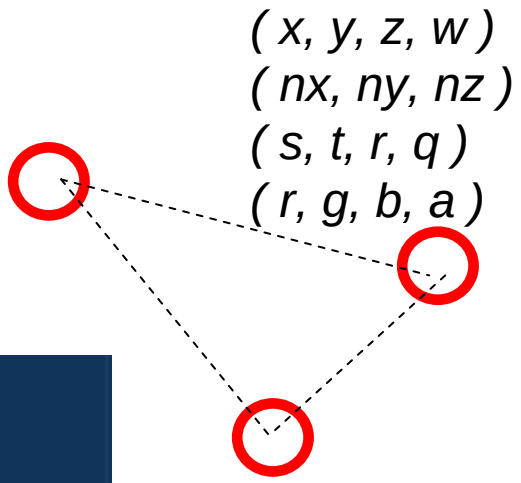
- Pixel shader : un pixel à la fois
  - On ne connaît pas les pixels voisins
  - Au mieux, on peut avoir la variation d'une valeur par rapport au pixel d'à côté

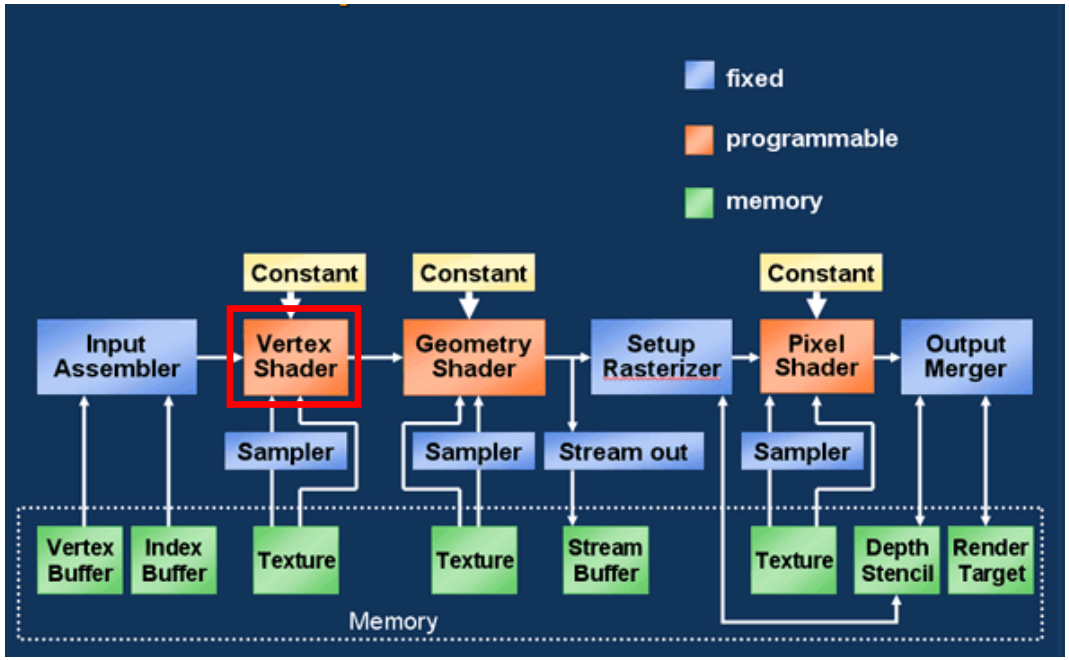


Fonctionnalités des cartes graphiques
Shaders

Ce qu'on peut faire

- Au niveau des sommets :
  - Des transformations/projections différentes
  - Des calculs de coordonnées de textures différents
  - Des calculs d'illumination différents

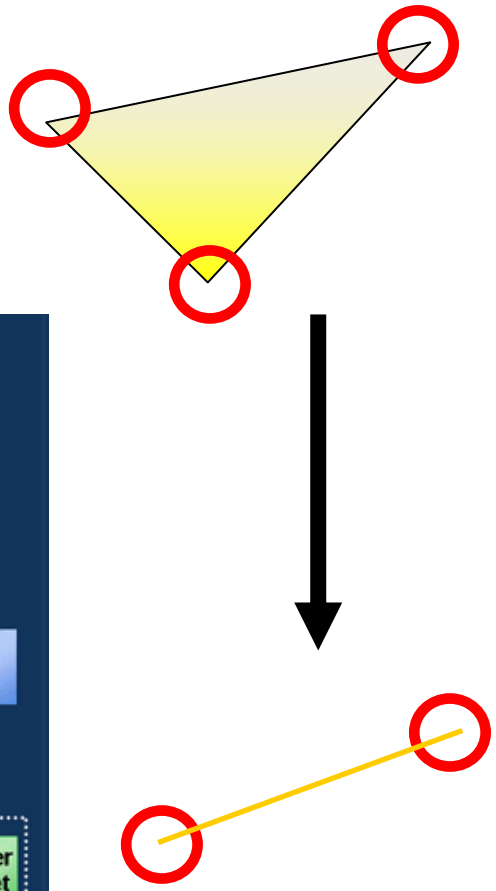


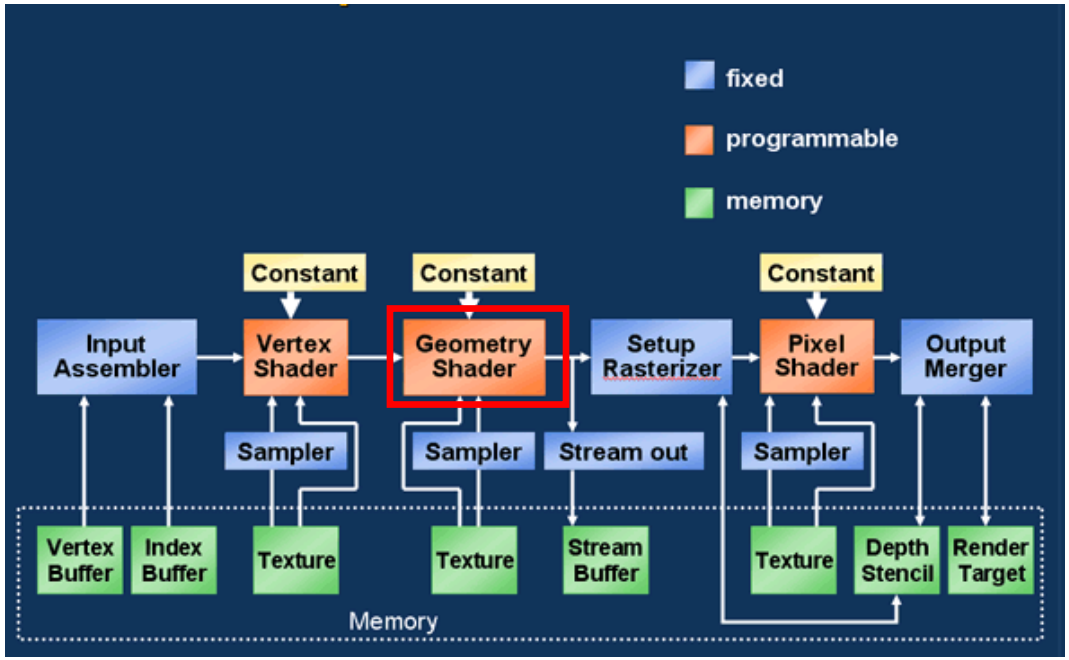


Fonctionnalités des cartes graphiques
Shaders

Ce qu'on peut faire

- Au niveau des primitives :
  - Ajouter/Supprimer des sommets
  - Modifier les primitives
  - Récupérer directement la géométrie sans "tramage".

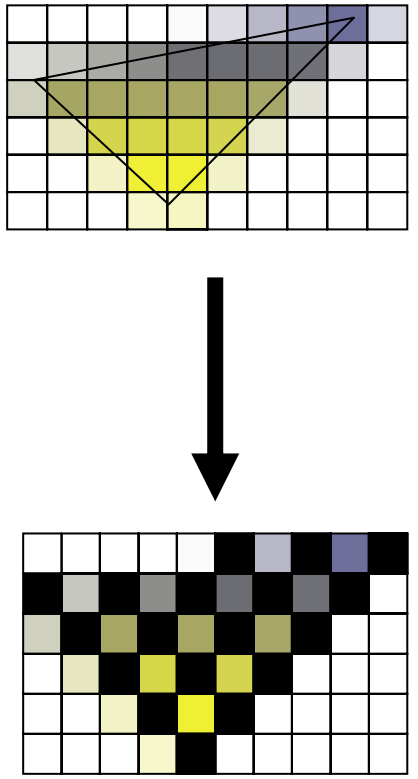


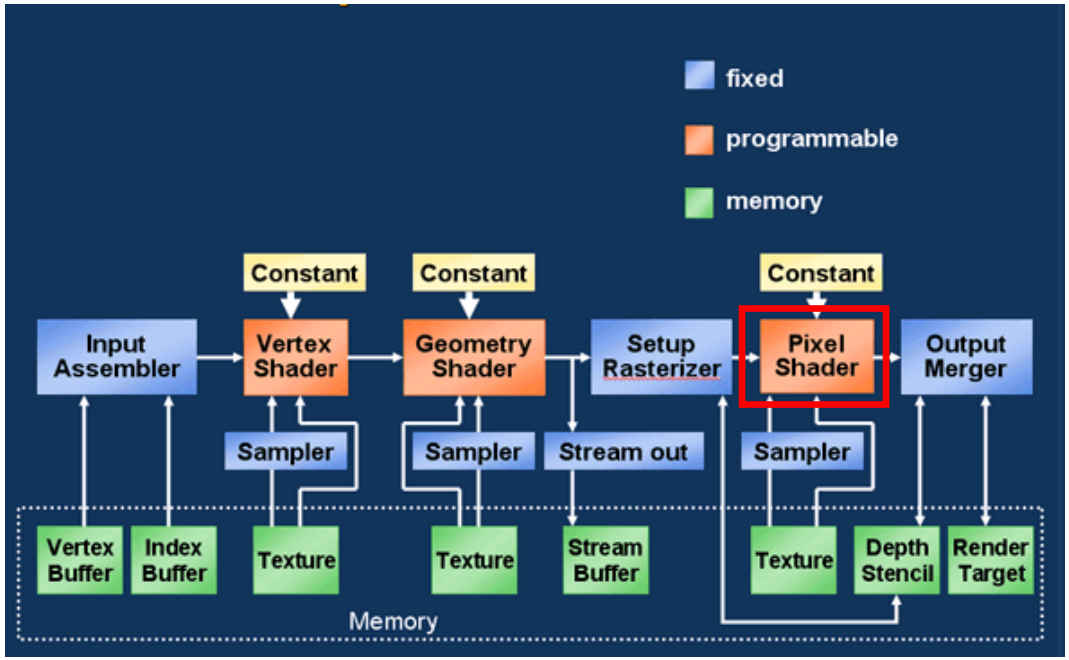


Fonctionnalités des cartes graphiques
Shaders

Ce qu'on peut faire

- Au niveau des pixels :
  - La même chose qu'aux sommets, mais par pixel
  - Utiliser le contenu de textures dans des calculs
  - Changer la profondeur des pixels

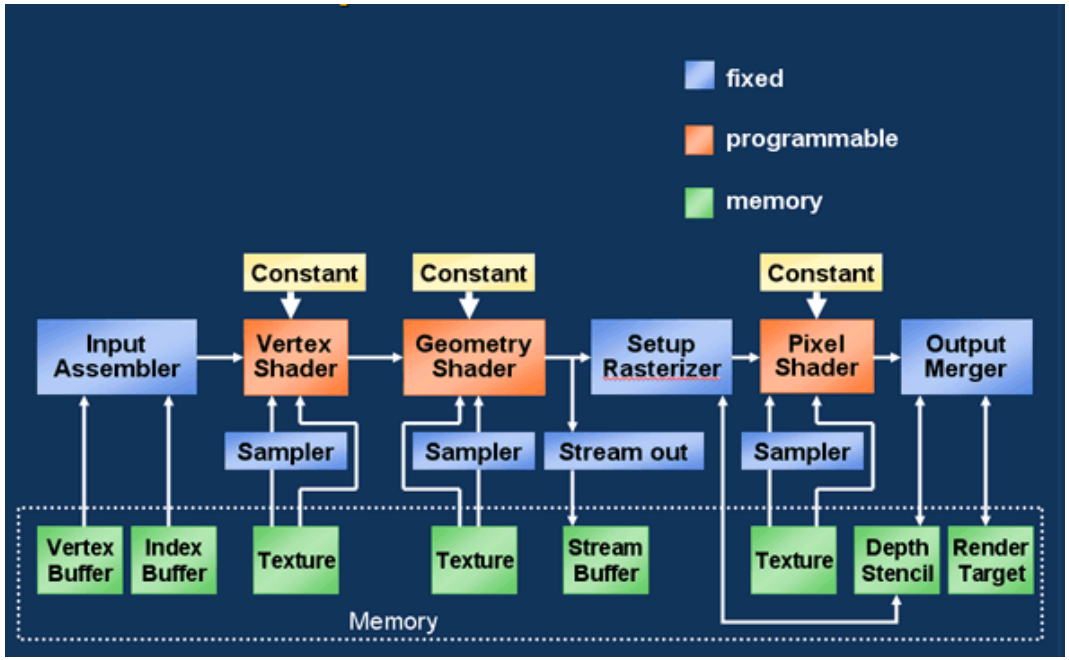




Fonctionnalités des cartes graphiques
Shaders

Ce qu'on ne peut pas (encore ?) faire

- Modifier le tramage (rasterizer)
- Modifier la composition (output merger)
- Lire le buffer de dessin sur la fenêtre



Plan

1. Historique
2. Fonctionnalités des cartes graphiques
3. GLSL
4. Communications CPU $\leftrightarrow$ GPU
5. GPGPU

GLSL
Types de base

- Flottants, entiers, booléens
  - `float`, `bool`, `int`, `unsigned int`
- Vecteurs 2,3,4
  - `[b,u,i]vec{2,3,4}`
- Matrices 2x2, 3x3, 4x4
  - `mat{2,3,4}`
- Accesseurs de textures
  - `sampler{1,2,3}D`, `samplerCube`, `samplerRect`, …
- Structures
  - `struct my_struct { int index; float value};`
- Tableaux
  - `int array[5];`

GLSL
Entrées

- *Built-in* : tous les états d'OpenGL, passés par OpenGL
  - Position, couleurs, directions des lumières
  - Textures flottantes ou entières
  - Matrices
- *Attribute* : passés par le programme OpenGL
  - Peuvent varier pour chaque sommet (couleurs, textures, normales)
- Uniform : passés par le programme OpenGL
  - Ne varie pas entre glBegin/glEnd (matrices, textures, lumières)
- *Varying* : échanges entre les différents shaders
- *Constant*

GLSL
Entrées built-in

Vertex/Geometry shader :

- Position : **`gl_Vertex`**
- Couleurs : **`gl_Color`**, **`gl_SecondaryColor`**
- Normale : **`gl_Normal`**
- Coordonnées de textures : **`gl_MultiTexCoord`**

Fragment shader :

- **`gl_FragCoord`** : coordonnées du pixel dans la fenêtre
- **`gl_Color`** : couleur du pixel interpolée
- **`gl_TexCoord[]`** : coordonnées de textures interpolées
- **`gl_FrontFacing`** : face ou dos du triangle

GLSL
Sorties

Vertex/Geometry shader :

- **`gl_Position`** : position du sommet en coordonnées homogènes (obligatoire)
- **`gl_PointSize`** : taille d'un point en rendu par point
- **`gl_FrontColor`**, **`gl_BackColor`** : couleurs
- **`gl_TexCoord[]`** : coordonnées de textures

Fragment Shader :

- **`gl_FragColor`** : couleur du pixel (obligatoire)
- **`gl_FragDepth`** : profondeur du pixel

GLSL
Génération

Geometry shader :

- EmitVertex()
- EndPrimitive()

Types des primitives

- Points, Lignes, Triangles
- Lignes, Triangles avec Adjacences

GLSL
Fonctions built-in

Trigonométrie
- sin, cos, tan, asin, acos, atan, ...

Exponentiation
- exp, pow, log, exp2, log2, sqrt, ...

Arithmétiques
- abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, ...

Géométriques
- length, distance, dot, cross, normalize, reflect, refract

Accès aux textures
- texture1D, texture2D, texture3D, textureCube, shadow, ...

Manipulation de bits
- << , >> , | , & ...

Et d'autres...

GLSL
Premier Programme

```
uniform vec4 Bidule;
```

Entrée

Fonction

```
vec4 UneFonction( vec4 Entree )
{
        return Entree.zxyw;
}
```

Swizzle

Point d'entrée

Variable locale

Entrées OpenGL

```
void main()
{
        vec4 pos = gl_ModelViewProjectionMatrix * gl_Vertex;
        gl_Position = pos + UneFonction( Bidule );
}
```

Sortie OpenGL

Multiplication matrice-vecteur

GLSL
Compilation

- Création Kernel
  - shader_id = glCreateShaderObjectARB(type);
    - Type = {GL_VERTEX_SHADER_ARB, GL_FRAGMENT_SHADER_ARB, GL_GEOMETRY_SHADER_EXT
  - glShaderSourceARB(shader_id,1,&const_shader_src,NULL);
    - const_shader_src = programme
- Compilation
  - glCompileShaderARB(shader_id);
- Debug
  - glGetProgramivARB(shader_id,GL_OBJECT_INFO_LOG_LENGTH_ARB,&info_log_length);
  - c_infolog = new char[info_log_length];
  - glGetInfoLogARB(shader_id,info_log_length,&nread,c_infolog);

GLSL
Compilation

- Création Programme
  - _program_shader = glCreateProgramObjectARB();
- Propriétés Geometry Kernel
  - glProgramParameteriEXT(_program_shader, GL_GEOMETRY_INPUT_TYPE_EXT, _input_device);
  - glProgramParameteriEXT(_program_shader, GL_GEOMETRY_OUTPUT_TYPE_EXT, _output_device);
  - glProgramParameteriEXT(_program_shader, GL_GEOMETRY_VERTICES_OUT_EXT, _nb_max_vertices);
- Attacher
  - glAttachObjectARB(_program_shader,_vertex_shader);
  - glAttachObjectARB(_program_shader, _geometry_shader);
  - glAttachObjectARB(_program_shader,_fragment_shader);
- Lier
  - glLinkProgramARB(_program_shader);

GLSL
Uniforms / Attributes

```
glUseProgramObjectARB( Program );
```

Utilisation d'un programme

```
glGetUniformLocationARB();
glUniform{1,2,3,4}f[v]ARB();
glUniformMatrix{2,3,4}fvARB();
```

Réglage d'un uniform

```
glGetAttribLocationARB();
glVertexAttrib{1,2,3,4}f[v]ARB();
```

Réglage d'un attribute

```
glUseProgramObjectARB(0);
```

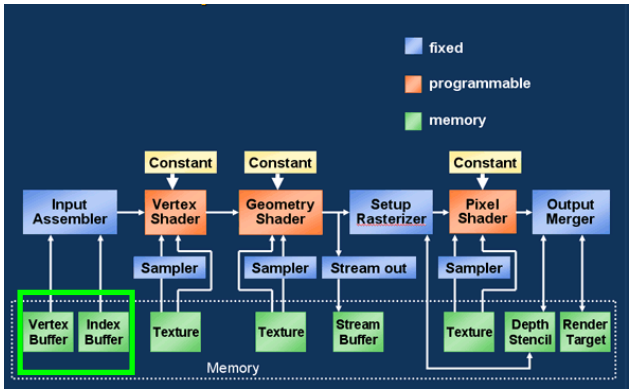
Fin de programme

Plan

1. Historique
2. Fonctionnalités des cartes graphiques
3. GLSL
4. Communications CPU $\leftrightarrow$ GPU
5. GPGPU

Communications CPU ↔ GPU

du CPU vers le GPU



- Vertex et Index Arrays
  - Initialisation

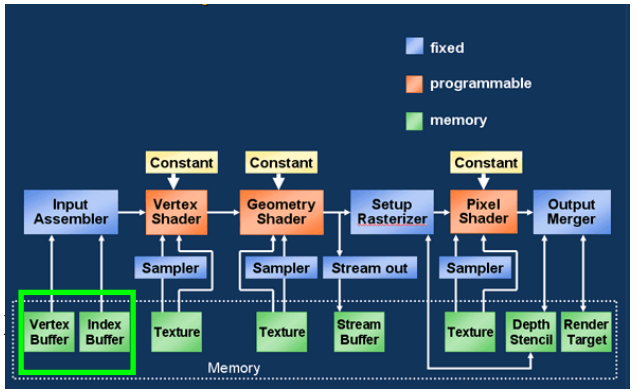
```
glGenBuffersARB(1,&_vbo_vertex);
glBindBufferARB(GL_ARRAY_BUFFER_ARB,_vbo_vertex);
glBufferDataARB(GL_ARRAY_BUFFER_ARB,nv*3*sizeof(float),pv,GL_STATIC_DRAW_ARB);
glGenBuffersARB(1,&_vbo_attrib);
glBindBufferARB(GL_ARRAY_BUFFER_ARB,_vbo_attrib);
glBufferDataARB(GL_ARRAY_BUFFER_ARB,3*nv*sizeof(float),pa,GL_STATIC_DRAW_ARB);
glGenBuffersARB(1,&_vbo_index);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB,_vbo_index);
glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER_ARB,ne*3*sizeof(int),pe,GL_STATIC_DRAW_A
 RB);
```

  - Mise à Jour

```
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, _vbo_index);
void* mem = glMapBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, GL_WRITE_ONLY_ARB);
memcpy(new_index, mem, 3*ne*sizeof(int));
glUnmapBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB);
```

Communications CPU ↔ GPU

du CPU vers le GPU

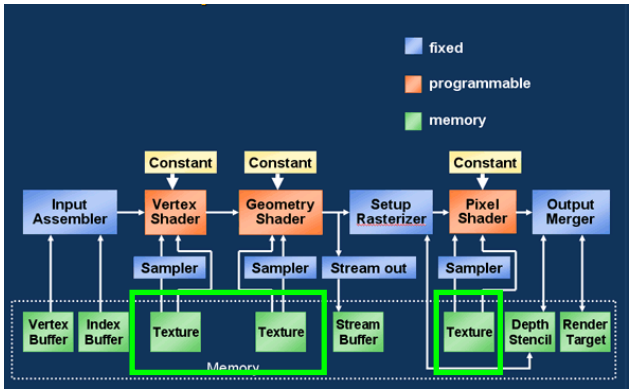


- Vertex et Index Arrays : Affichage

```
glEnableClientState(GL_VERTEX_ARRAY);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, _vbo_verte
glVertexPointer(3,GL_FLOAT,0,(char*)NULL);

glEnableVertexAttribArrayARB(_attrib);
glBindBuffer(GL_ARRAY_BUFFER_ARB, _vbo_attrib);
glVertexAttribPointerARB(_attrib, 3, GL_FLOAT, GL_FALSE, 0, (char*)NULL);

glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, _vbo_index);
glDrawElements(GL_TRIANGLES, 3*_n_elements, GL_UNSIGNED_INT, NULL);

glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB,0);
glBindBufferARB(GL_ARRAY_BUFFER_ARB,0);
glDisableVertexAttribArrayARB(_attrib);
glDisableClientState(GL_VERTEX_ARRAY);
```

Communications CPU ↔ GPU

du CPU vers le GPU



- Pixel Buffer Object UNPACK
  - Initialisation

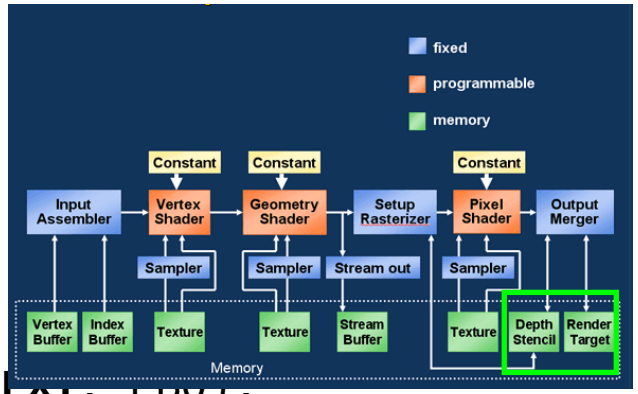
```
glGenBuffersARB(1, &PBO);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, PBO);
glBufferDataARB(GL_PIXEL_UNPACK_BUFFER_ARB, size, NULL,
  GL_STREAM_DRAW_ARB);
void *mem = glMapBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, GL_WRITE_ONLY);
memcpy(mem, offset, 4*screen_height*screen_width*sizeof(float));
glUnmapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB);
```

  - Utilisation

```
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, PBO);
glBindTexture(GL_TEXTURE_2D, TEX);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB, screen_width,
  screen_height, 0, GL_RGBA, GL_FLOAT, NULL);
glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
```

Communications CPU ↔ GPU

du GPU vers le CPU



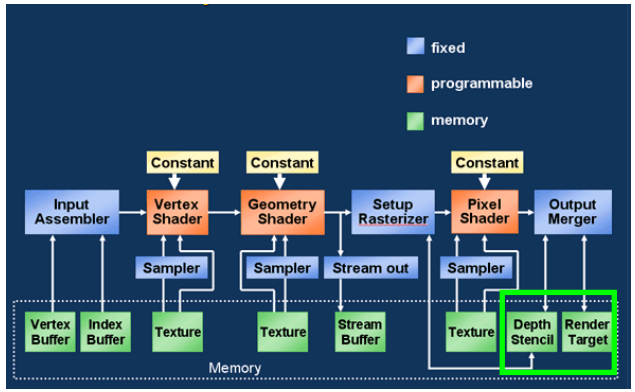
- Pixel Buffer Object PACK
  - Initialisation

```
glGenBuffersARB(1, &PBO);
glBindBufferARB(GL_PIXEL_PACK_BUFFER_EXT, PBO);
glBufferDataARB(GL_PIXEL_PACK_BUFFER_EXT, _size, NULL,
  GL_STREAM_DRAW_ARB);
```

  - Utilisation

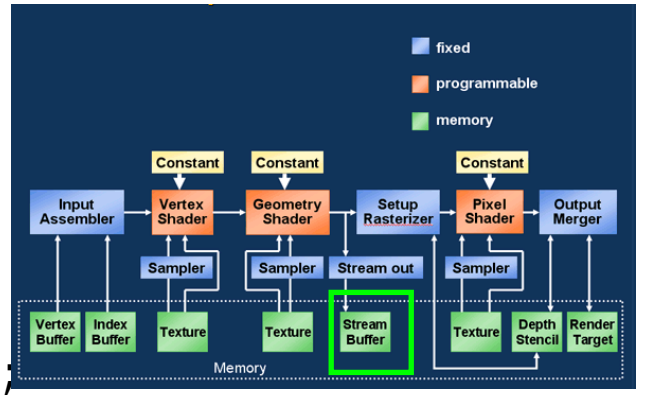
```
glBindBufferARB(GL_PIXEL_PACK_BUFFER_EXT, PBO);
glReadPixels(0, 0, w, h, GL_RED, GL_FLOAT, 0);
void  *mem  = glMapBufferARB(GL_PIXEL_PACK_BUFFER_ARB,
  GL_READ_ONLY_ARB);
float *data = (float*) malloc(w*h*sizeof(float));
memcpy(data,mem,w*h*sizeof(float));
glReadBuffer(GL_NONE);
glBindBufferARB(GL_PIXEL_PACK_BUFFER_EXT, 0 );
```

Communications CPU ↔ GPU
du GPU vers le GPU



- FrameBuffer Objects
  - Initialisation
    - `glGenFramebuffersEXT( 1, &id);`
  - Ajout de textures, de depth buffer
    - `glFramebufferTexture2DEXT(`**`GL_FRAMEBUFFER_EXT`**`, FBO_color[index],target ,tex[index],0);`
    - `FBO_color[] = {GL_COLOR_ATTACHMENT0_EXT, …}`
    - `glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,target,depth,0);`
  - Affichage
    - `glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,id);`
    - `GLenum buffers[] = {GL_COLOR_ATTACHMENT0_EXT,..};`
    - `glDrawBuffersARB(1,buffers);`
    - `// display`
    - `glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,0);`

Communications CPU ↔ GPU
du GPU vers le GPU



- Transform Feedback Object
  - Initialisation

```
glGenBuffersARB(1, &_TF);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, _TF);
glBufferDataARB(GL_ARRAY_BUFFER_ARB, _size*sizeof(float),0, type);
type = GL_{DYNAMIC, STREAM}_{COPY,READ}_ARB
loc = glGetVaryingLocationNV(shader_id, name);
glTransformFeedbackVaryingsNV(shader_id,number,loc,GL_SEPARATE_ATT
  RIBS_NV);
_index = 0;
```

  - Affichage

```
glBindBufferBaseNV(GL_TRANSFORM_FEEDBACK_BUFFER_NV, _index, _TF);
glEnable(GL_RASTERIZER_DISCARD_NV);
glBeginTransformFeedbackNV(GL_TRIANGLES);
// DISPLAY
glDisable(GL_RASTERIZER_DISCARD_NV);
glBindBufferBaseNV(GL_TRANSFORM_FEEDBACK_BUFFER_NV, _index, 0);
```

Plan

1. Historique
2. Fonctionnalités des cartes graphiques
3. GLSL
4. Communications CPU $\leftrightarrow$ GPU
5. GPGPU

GPGPU

- General-Purpose Computation Using Graphics Hardware
- Un GPU = un processeur SIMD
- Une texture = un tableau d'entrée
- Une image = un tableau de sortie

GPGPU : Applications

- Rendu avancé
  - Illumination globale
  - Image-based rendering
  - ...
- Traitement du signal
- Géométrie algorithmique
- Algorithmes génétiques
- A priori, tout ce qui peut se paralléliser

GPGPU : Limitations

- Récupérer l'image rendue = lent
  - PCI Express
- Opérateurs, fonctions, types assez limités
- Un algorithme parallélisé n'est pas forcément plus rapide que l'algorithme séquentiel

GPGPU : Langage

- CUDA (Compute Unified Device Architecture)
  - Basé sur le langage C
  - Propose une mémoire partagée de 16 Ko pour les threads
  - Pensé pour simplifier les accès, les retours et la compilation des kernels pour les non-spécialistes d'OpenGL.
  - Librairies fournies
    - FFT
    - BLAS : Algèbre Linéaire

GPGPU : Exemple Cuda

```
cudaArray* cu_array; float* gpu_array;
float* cpu_array  = new float[width*height];
texture<float, 2, cudaReadModeElementType> tex;

//Allocate array
cudaMalloc((void**)&gpu_array, width*height*sizeof(float));
// Bind the array to the texture
cudaBindTextureToArray(tex, cuArray);
// Run kernel
dim3 blockDim(16, 16);
dim3 gridDim(width / blockDim.x, height / blockDm.y);
kernel <<<gridDim, blockDim>>>(gpu_array, cu_array, width);
cudaUnbindTexture(tex);
//Copy GPU data to array
cudaMemcpy(cpu_array,gpu_array,width*height*sizeof(float),cudaMemcpyDeviceToHost);
//Free memory
cudaFree(gpu_array); delete [] cpu_array;

__global__ void kernel(float* odata, float* idata, int width)
{
   unsigned int x   = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int y   = blockIdx.y*blockDim.y + threadIdx.y;
   odata[y*width+x] = idata[x+y*width];
}
```