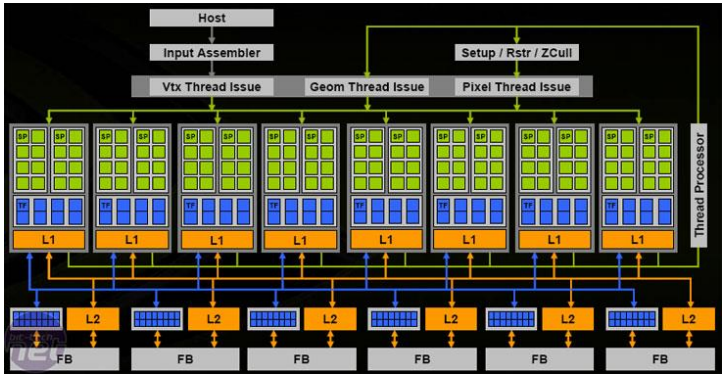
Informatique Graphique 3D & Réalité Virtuelle

Synthèse d'Images

Processeurs Graphiques

Tamy Boubekeur

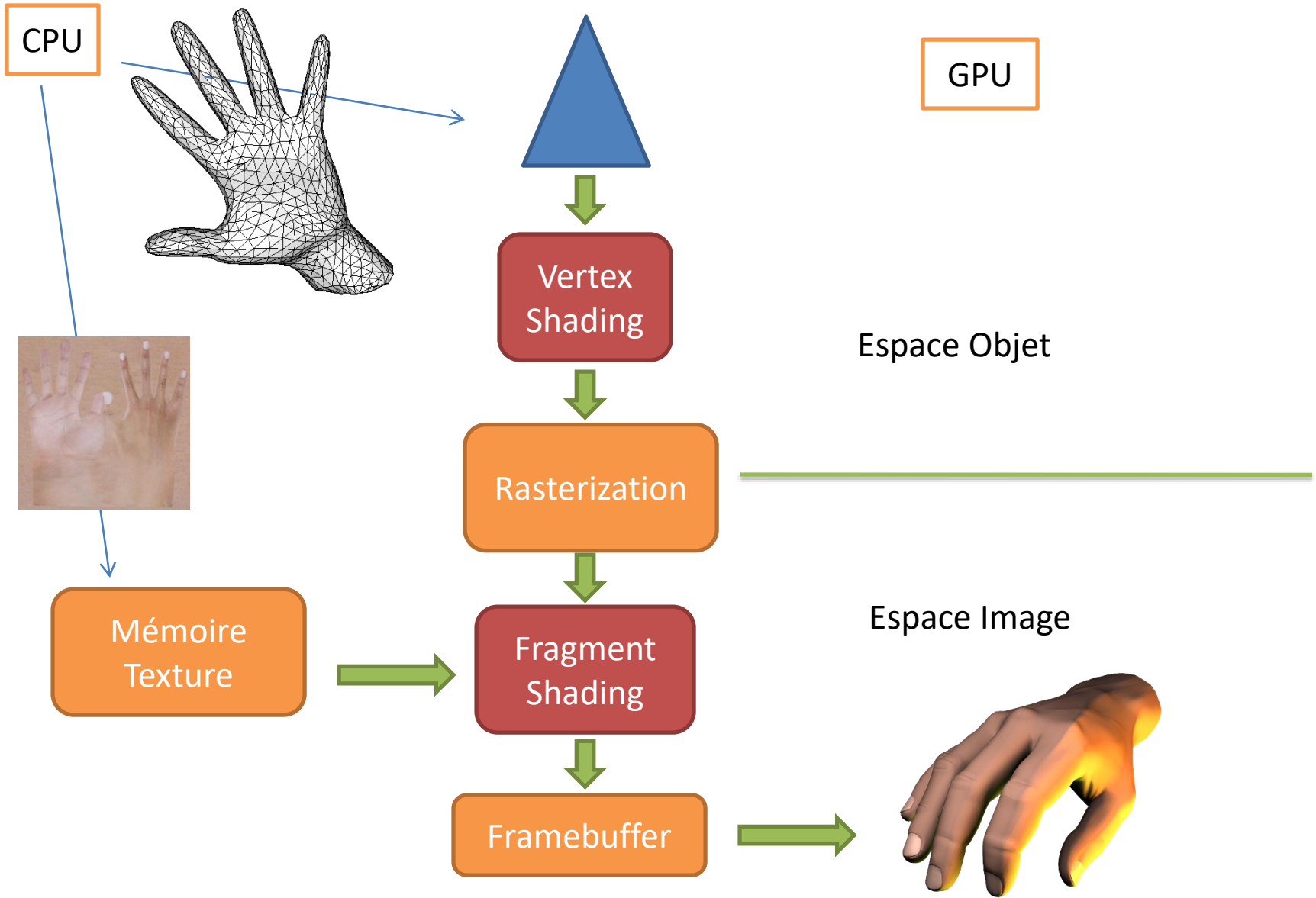


PROGRAMMATION GPU

GPU

- Graphics Processor Unit
- Conçu pour le calcul 3D en synthèse d'image
- Optimisé pour le rendu par *rasterization*
- Poussé par le marché des jeux vidéo

Rendu GPU par *Rasterization*



Architecture Many-Core

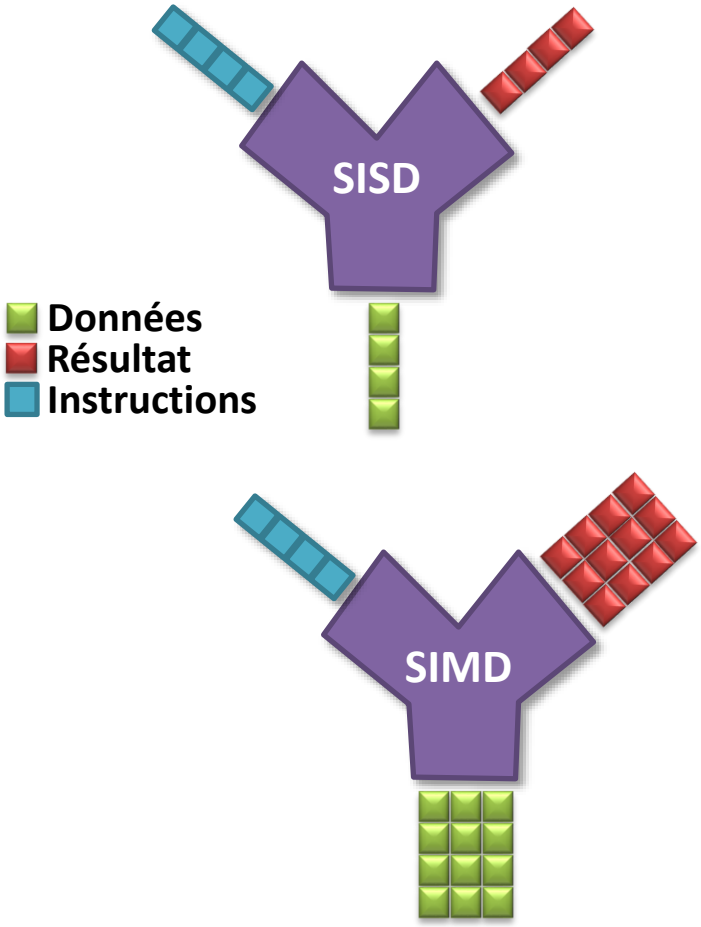
- Point de convergence CPU/GPU

Défis :

- Techniques
  - Coût de production
  - Coût énergétique
- Scientifiques
  - Algorithmique parallèle et calcul distribué
  - Machine hybrides : effets NUMA, flexibilité vs performances
- Pédagogiques
  - Difficile à enseigner en général
  - Cas particulier de la programmation GPU « de base » : *shaders*, GPU Computing (CUDA/OpenCL/Compute Shaders)

Machine Massivement Multi-Threads

- GPU = centaines de cœurs
- Architecture **SIMD**
  - **Single Instruction / Multiple Data**
- Cœurs limités :
  - Pas d'allocation dynamique de mémoire
  - Pas de pile > pas de récursion
- Hiérarchie de mémoire
  - Effets NUMA
- Bien adapté au calcul intensif « naturellement parallèle »
- Algèbre linéaire efficace



GPU vs Multi-cœur CPU

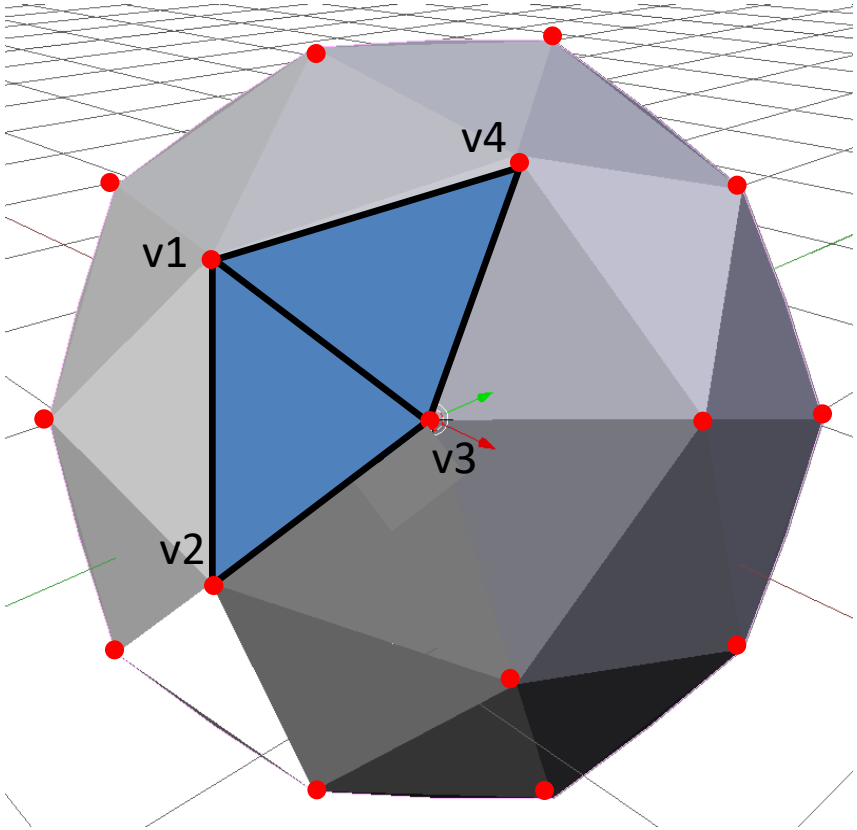
	CPU (Core i7-965)	GPU (Tesla S1070)
Nombre de coeurs	4	4x240
Puissance brute	70 GFlops	3.73-4.14 TFlops
Bande passante	18	408

GPU : Données en entrée

- **Maillage polygonal** : approximation de la surface d'un objet à l'aide d'un ensemble de polygones
  - **Soupe de Polygones** : suites de n-uplets de coordonnées 3D correspondants aux polygones
  - **Maillages indexés** : graphe avec géométrie et topologie séparés
    - Une liste de sommets (V)
    - Une liste de relation topologique:
      - Arêtes (Edge, E)
      - Faces (F)
- En pratique, {V,F} (exemple : OpenGL)

Exemple

- Ensemble de sommets (géométrie)
  - v1 (x, y, z)
  - v2 (x, y, z)
  - v3 (x, y, z)
  - v4 (x, y, z)
- Ensemble de faces (topologie)
  - (v1, v2, v3)
  - (v1, v3, v4)

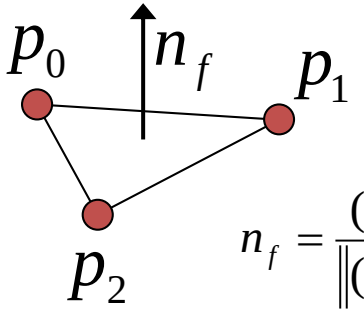


Attributs

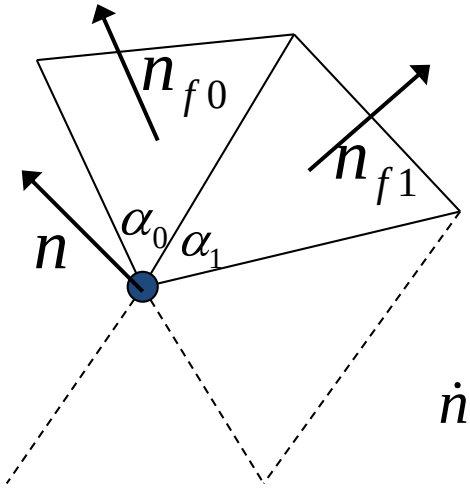
- **Définition :** propriétés associées aux sommets (le plus souvent), arêtes ou faces
- Attributs de sommets :
  - Position (« p »)
  - Vecteur normal (« n »)
  - Coordonnées paramétrique (« (u,v) »)
  - Apparence : couleur, indice de matériel, etc
  - Paramètres physiques pour la simulation
  - Etc
- Arêtes :
  - Plis vif (discontinuité du gradient)
- Faces :
  - Couleur

Normales

- **Essentielles pour le rendu**
  - BRDF
- Stockées par sommets
- Utiles pour certains traitement géométrique
  - Simplification
- Calcul :
  - Moyennes des normales des faces incidentes
  - Moyennes pondérée par les angles des arêtes incidentes
    - Plus robustes pour les distributions de triangles non uniformes



$$n_f = \frac{(p_1 - p_0) \wedge (p_2 - p_0)}{\left\| (p_1 - p_0) \wedge (p_2 - p_0) \right\|}$$



$$\dot{n} = \frac{\sum_i \alpha_i n_i}{\sum_i \alpha_i}$$

$$n = \frac{\dot{n}}{\left\| \dot{n} \right\|}$$

Coordonnées paramétriques

$(u,v) \in \Re^2$ par convention: $(u,v) \in [0,1]^2$

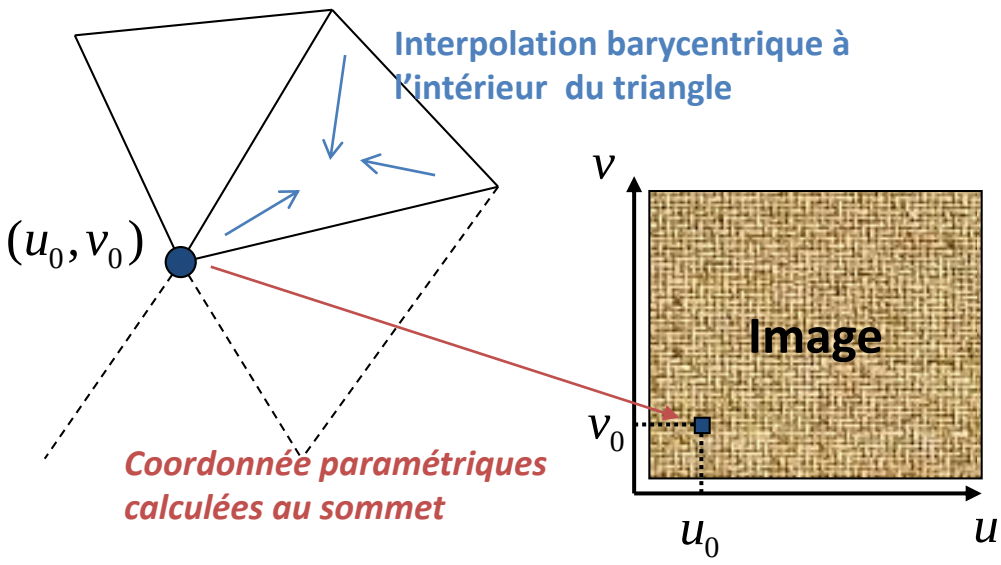
Définition d'une propriété de surface à partir d'un fonction bi-variée:

$f :$

$\Re^2 \rightarrow \Re^n$

$u, v \rightarrow c$

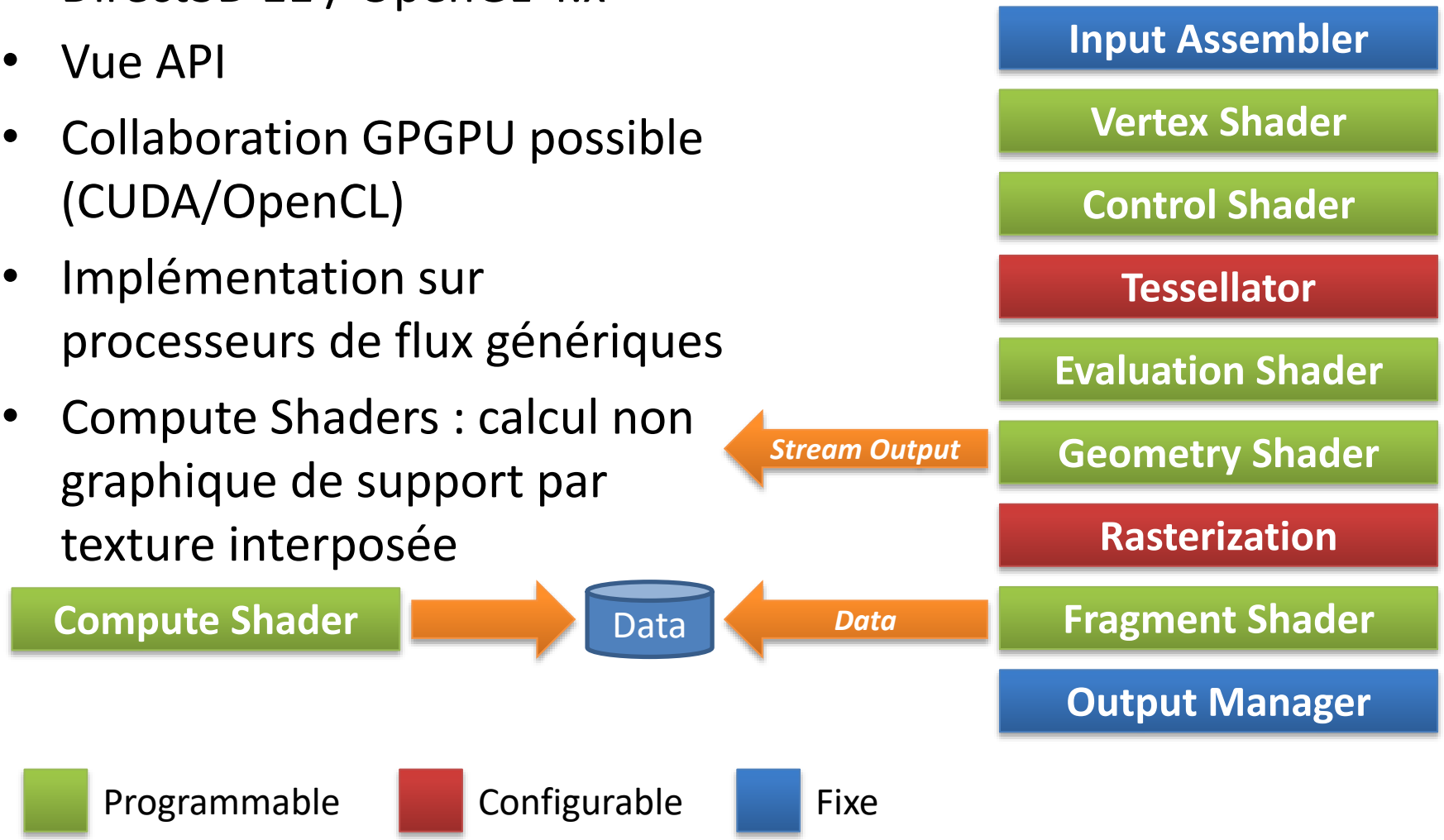
Exemple : valeur d'un pixel dans une images (« texture mapping »)



Définition de coordonnées paramétriques continues sur l'ensemble des sommets d'un maillage : ***Paramétrisation***

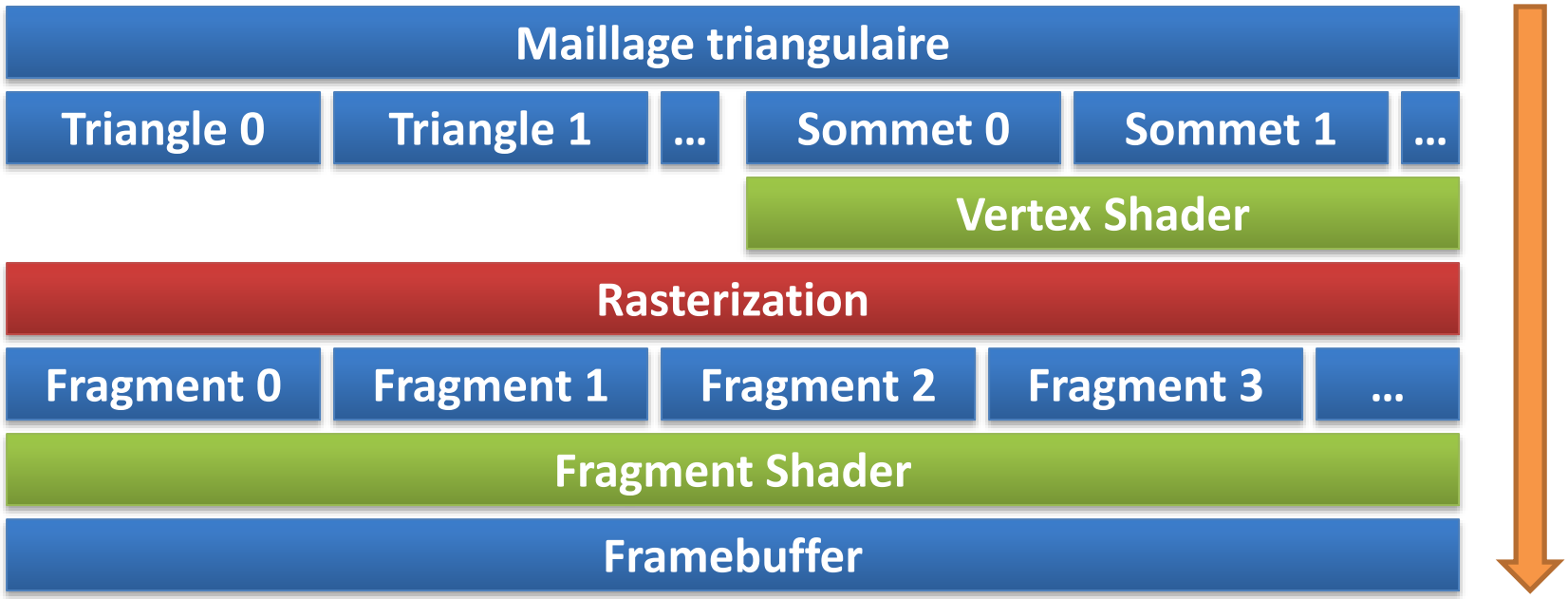
Pipeline Graphique Moderne

- Direct3D 11 / OpenGL 4.x
- Vue API
- Collaboration GPGPU possible (CUDA/OpenCL)
- Implémentation sur processeurs de flux génériques
- Compute Shaders : calcul non graphique de support par texture interposée



Etages majeurs

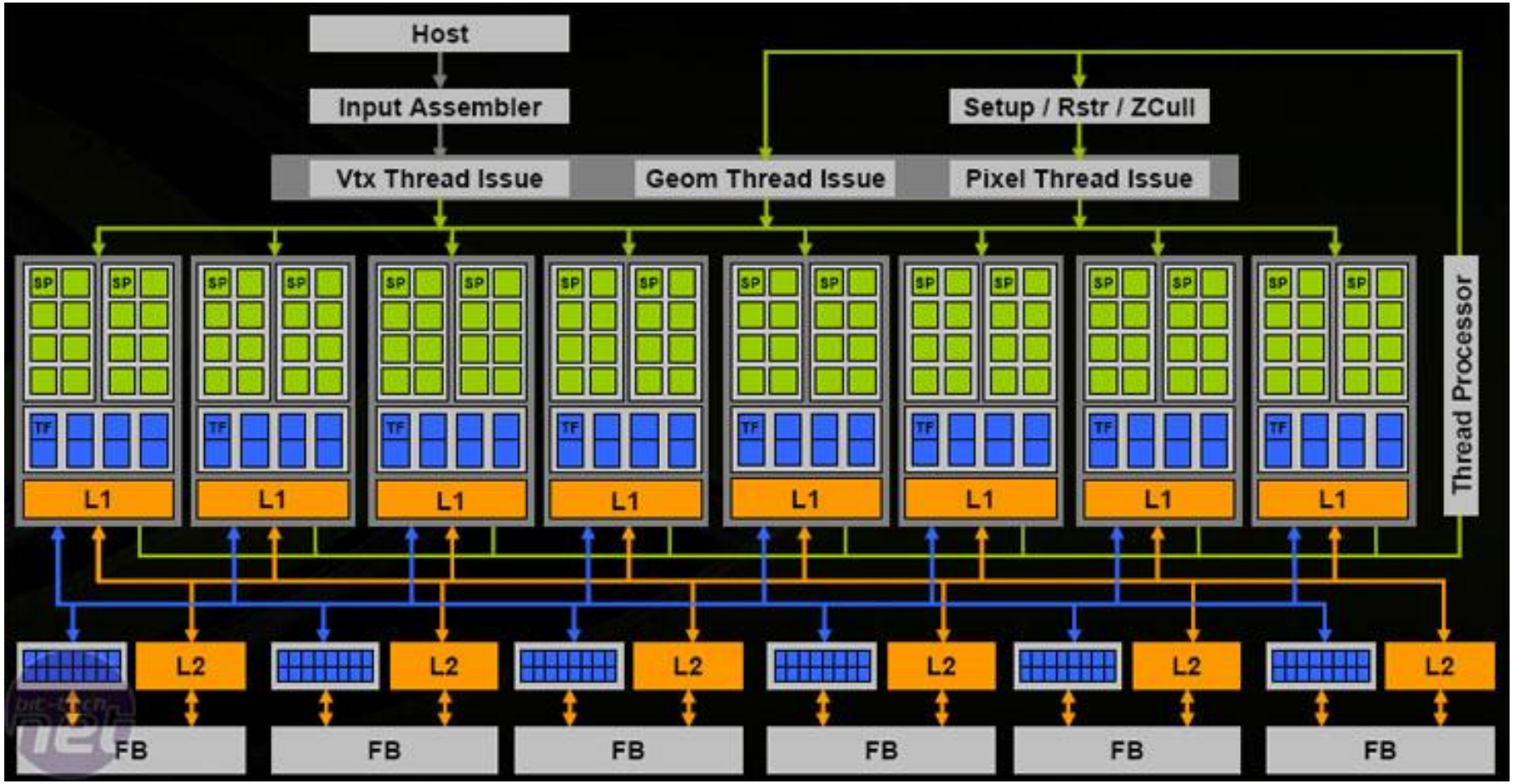
- Calcul en flux



- Intégralement parallèle
  - Chaque sommet traité indépendamment
  - Chaque fragment traité indépendamment

Architecture Vue Matérielle

(NVIDIA G80)

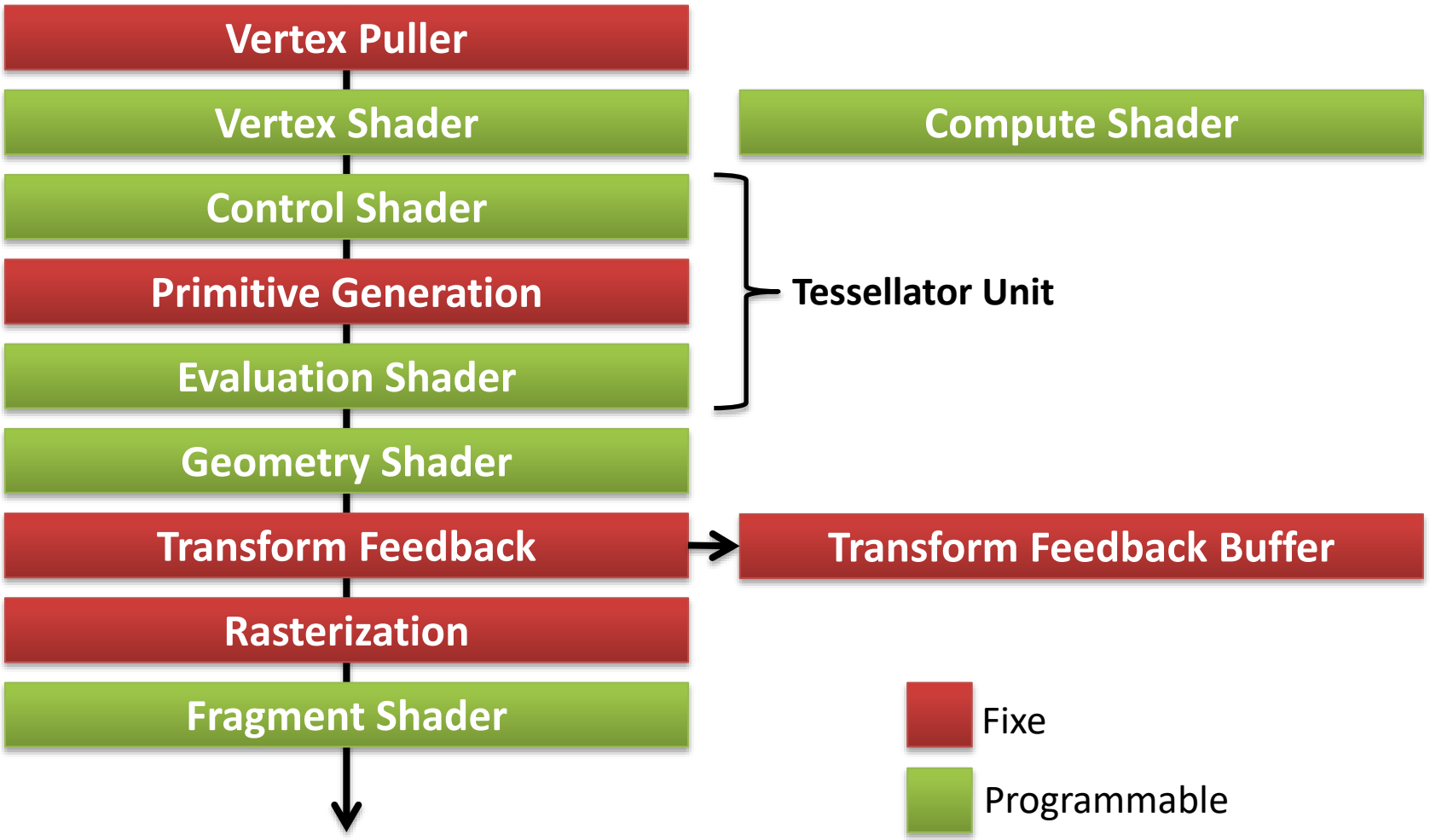


API 3D

- **OpenGL**
  - Maintenue par Khronos Group (consortium)
  - Multiplateforme
  - Robuste (CAD)
  - Evolution lente, *suiveur*
- **Direct 3D**
  - Développé par Microsoft
  - Standard sous MS Windows (PC, XBOX, PocketPC)
  - Evolue très vite
  - Dicte la marche à suivre aux constructeurs de GPUs
    - Drame du *Géométrie Shader*
    - Futurs drames possibles : Tessellator Unit, Raytracing

Pipeline Moderne

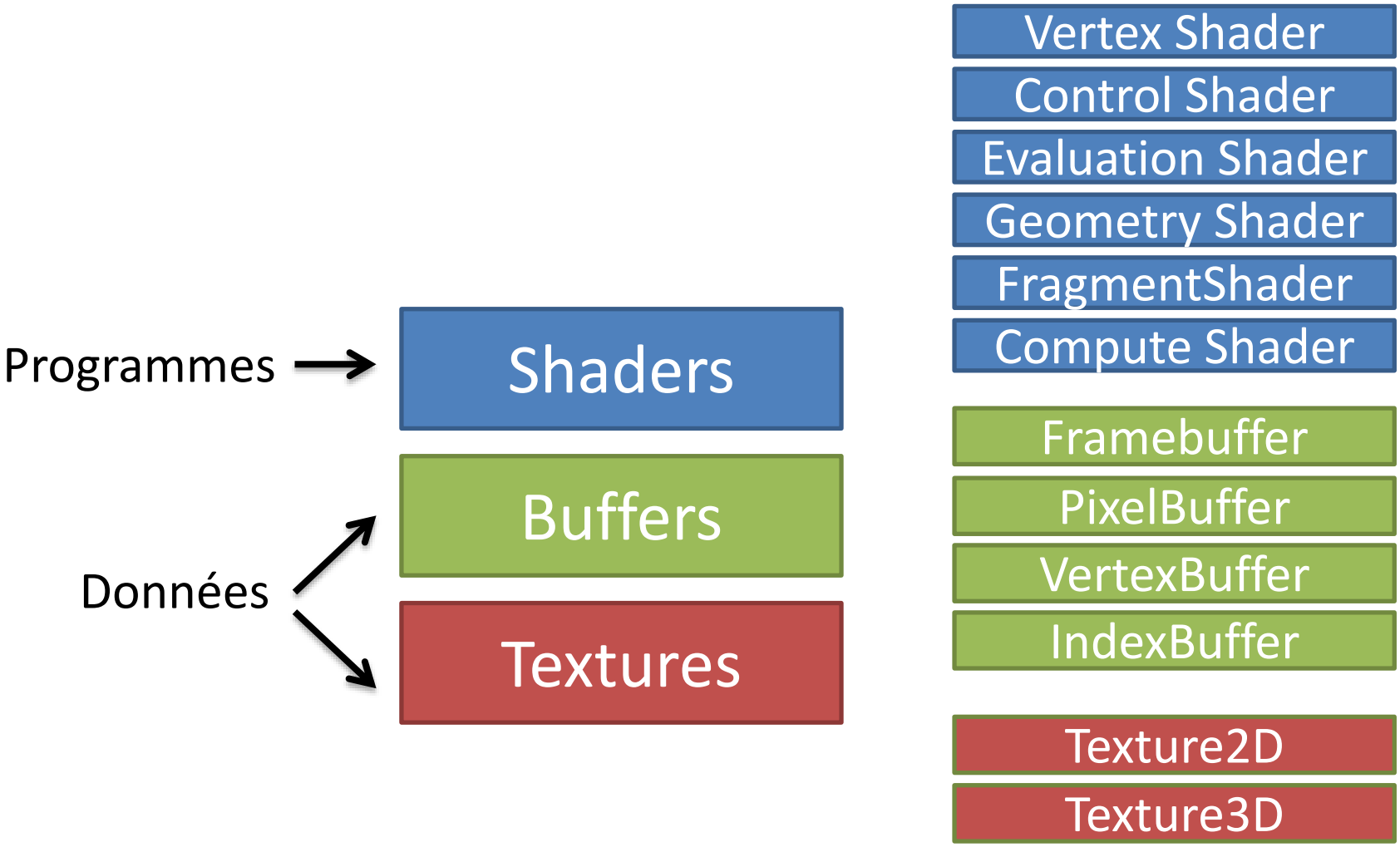
OpenGL 4.3



OpenGL

- OpenGL classique (v1)
  - Pas de programmation GPU, pipeline fixe
- OpenGL programmable (v2,v3)
  - Programmation shaders, entrées-sorties formatées
- **OpenGL moderne (v4)**
  - Shaders graphiques et shaders calcul, entrées sorties redéfinissables.

OpenGL Moderne



Shaders

- **Vertex**
- **Control**
- **Evaluation**
- **Geometry**
- **Fragment**
- ***Compute***

- **<u>Pipeline</u>**
  - Synchronisation I/O
- Faiblement dynamique
  - Geometry Shader :
    - Très Flexible
    - Faible amplification
  - Tessellation :
    - Peu flexible
    - Grande amplification

Langages de Shaders

- Premiers langages développés pour la programmation GPU
- « Types de bases »
  - Sommets, vecteurs, matrices, textures, pixel (fragment), couleur, etc
- De plus en plus flexibles
  - Boucles, branchements conditionnels, macros, structures
- Exemples :
  - Cg (nVidia)
  - HLSL (Direct3D/Microsoft)
  - **GLSL** (OpenGL)
  - tous relativement équivalents
- *Premier succès populaire de la programmation parallèle*

GLSL I

- OpenGL Shading Language
- Complète OpenGL
  - Programmation des différents étages de rendu
  - 2009 : Vertex, Geometry, Fragment
  - 2011 : Control, Evaluation (Tessellator)
  - 2012 : Compute
- Structure similaire à C/C++

GLSL II

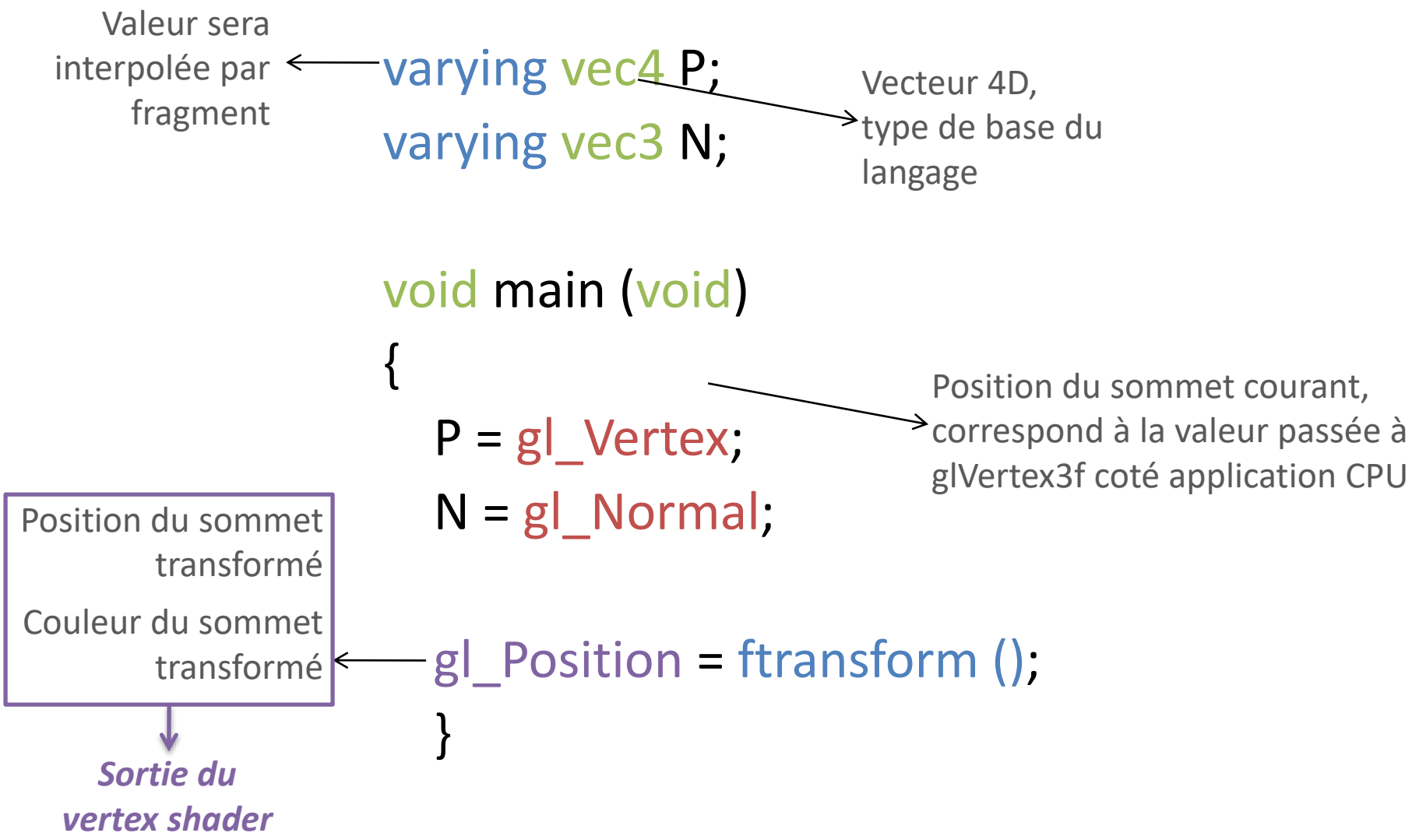
- Types spécifiques au calcul graphique
  - vec3, vec4, Texture Sampler
- Construction standard :
  - *Boucles*
  - *Branchements*
  - *Fonctions à retour de valeur*
- Variable globales spécifiques
  - Définit au niveau du code application (C/C++/Java/Python + OpenGL)
  - Position des sources de lumières, propriétés de matériaux, matrice de transformation « model-vue »
  - Accès mémoire : texture (lecture seulement)
- Primitives de synchronisation (barrier ()) pour geometry shader et compute shader.
- Limitations :
  - Pas de récursion
  - Pas d'allocation

GLSL III

OpenGL v2/3: Entrées-sortie formatées, spécialisées à l'étage

Etage	Entrée	Sortie
Vertex	•Un sommet •Matrice de transformation •Propriétés par sommets : position (glVertex), normale (glNormal), etc	Un sommet transformé et colorié (e.g. éclairage)
Geometry	Une primitive (line, triangle, etc)	Plusieurs primitives (nombre borné et petit)
Fragment	•Coordonnée image x,y •Couleur et coordonnées de texture par sommet interpolée au fragment •Option : toute variable de type « varying » spécifiée aux étages supérieurs	•Un fragment RGBA •MRT : plusieurs fragment pour les même coordonnées

Vertex Shader (GL2)



Fragment Shader (GL2)

Evaluation de la BRDF de Phong par pixel

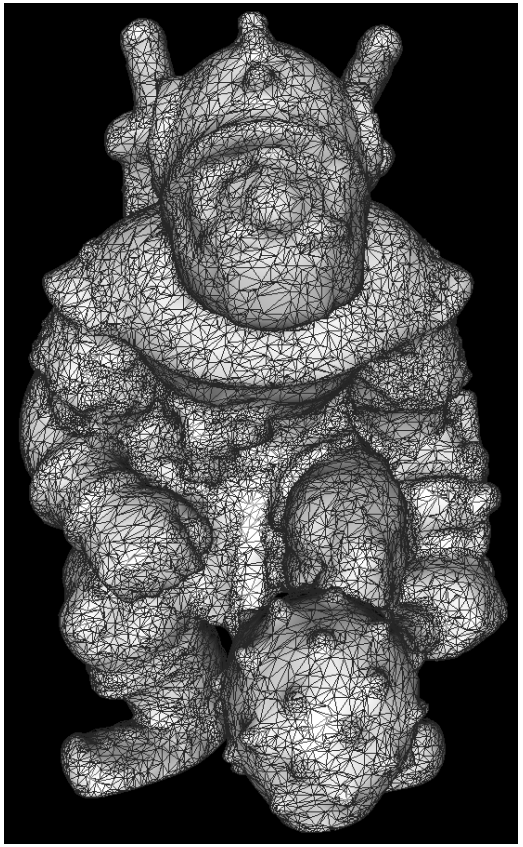
```
uniform float diffuseRef;
uniform float specRef;
uniform float shininess;

varying vec4 P;
varying vec3 N;

void main (void) {
   gl_FragColor = vec4 (0.0, 0.0, 0.0, 1);
   for (int i = 0; i < 8; i++) {
      vec3 p = vec3 (gl_ModelViewMatrix * P);
      vec3 n = normalize (gl_NormalMatrix * N);
      vec3 l = normalize
      (gl_LightSource[i].position.xyz - p);
      vec3 v = normalize (-p);
      float fd= max (dot (l, n), 0.0);
      vec3 r = reflect (-l, n);
      float fs= pow (max(dot(r, v), 0.0), shininess);
      vec3 colorResponse =
      diffuseRef * fd* gl_LightSource[i].diffuse.rgb
      + specRef * fs* gl_LightSource[i].specular.rgb;
      gl_FragColor.rgb += colorResponse;
   }
}
```

Fragment Shader

Rendu binaire style « Sin City »



```
//------------------------------
// GLSL Fragment Shader
// for "Frank Miller's Shading"
// Author: Tamy Boubekeur
// Version: 0.01
//------------------------------

varying vec4 P;
varying vec3 N;

void main (void) {
    gl_FragColor = vec4 (0.0, 0.0, 0.0, 1);

    int light = 0;

    vec3 p = vec3 (gl_ModelViewMatrix * P);
    vec3 n = normalize (gl_NormalMatrix * N);
    vec3 c = gl_LightSource[light].position;
    vec3 l = normalize (c - p);
    vec3 r = reflect (-l, n);
    vec3 v = normalize (-p);

    float spec = max(dot(r, v), 0.0);
    spec = pow (spec, 16.0);
    spec = max (0.0, spec);

    float sil = dot (v, n);

    if (spec > 0.0 || sil < 0.3)
        gl_FragColor = vec4 (1,1,1,1);
}
```

Maillage

Rendu

Vertex Shader (GL4)

```
Uniform mat4 u_matMVP;
in vec4 i_vPosition;
out vec4 o_vPosition;

void main () {
    // sans tessellation:
    gl_Position = u_matMVP * i_vPosition;
    // avec tessellation:
    o_vPosition = i_vPosition;
}
```

Tessellation Control Shader (GL4)

```
layout( vertices = 3 ) out;       // Sommets de contrôle

in vec4 v_vPosition[];
out vec4  tc_vPosition[];
uniform float u_fTessLevelInner;
uniform float  u_fTessLevelOuter;

void main () {
  tc_vPosition[gl_InvocationID]=v_vPosition[gl_InvocationID];
  if (gl_InvocationID == 0 ) {
            gl_TessLevelInner[ 0 ] = u_fTessLevelInner;
            gl_TessLevelOuter[ 0 ] = u_fTessLevelOuter;
            gl_TessLevelOuter[ 1 ] = u_fTessLevelOuter;
            gl_TessLevelOuter[ 2 ] = u_fTessLevelOuter;
     }
}
```

Tessellation Evaluation Shader (GL4)

```
layout (triangles, fractional_odd_spacing, ccw ) in;

in vec4 tc_vPosition[];
uniform mat4 u_matMVP;

void main () {
    vec3 vtePosition = vec3( 0.0 );
    // simple tessellation linéaire :
    vtePosition += gl_TessCoord.x * tc_vPosition[0].xyz;
    vtePosition += gl_TessCoord.y * tc_vPosition[1].xyz;
    vtePosition += gl_TessCoord.z * tc_vPosition[2].xyz;

    gl_Position = u_matMVP * vec4( vtePosition, 1 );
}
```

Geometry Shader (GL4)

```
#version 400
#extension GL_EXT_geometry_shader4 : enable

layout( triangles )                          in;
layout( triangle_strip, max_vertices = 3 )   out;

void main()
{
    for  (int i = 0; i < 3; i++) {
        gl_Position = gl_in[ i ].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

Fragment Shader (GL4)

```
#version 400

out vec4 FragColor;

void main()
{
    FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
```

Compute Shader (GL4.3)

CUDA	Compute Shader
__syncthreads()	barrier() groupMemoryBarrier()
__shared__	shared
threadIdx	gl_LocalInvocationID
blockIdx	gl_WorkGroupID
__threadfence()	memoryBarrierShared()

Compute Shader : simple copie

```
#version 430
#define TILE_WIDTH 16
#define TILE_HEIGHT 16
const ivec2 tileSize = vec2( TILE_WIDTH, TILE_HEIGHT );
layout( binding=0, rgba8 ) uniform image2D input_image;
layout( binding=1, rgba8 ) uniform image2D output_image;
layout( local_size_x = TILE_WIDTH,
        local_size_y = TILE_HEIGHT ) in;
void main()
{
    const ivec2 tile_xy = ivec2(gl_WorkGroupID);
    const ivec2 thread_xy = ivec2(gl_LocalInvocationID);
    const ivec2 pixel_xy = tile_xy*tileSize + thread_xy;
    vec4 pixel = imageLoad(input_image, pixel_xy);
    imageStore(output_image, pixel_xy, pixel);
}
```

Tampons GPU

- *Buffer Objects*
  - *Frame buffer*
  - *Pixel buffer (transactions CPU pour le calcul en flux)*
  - *Texture buffer (accès de haut niveau)*
  - *Vertex buffer*
    - *Géométrie*
    - *Topologie*

Vertex Buffer (VBO)

- Stocke la géométrie des objets
- Peut-être
  - Entrelacé (interleaved) : un unique tableau d'attributs par primitive
    - e.g. (x0, y1, z0, nx0, ny0, nz0, x1, y1, …)
  - Séparé : un tableau par attribut
    - E.g., VBO position: (x0, y0, z0, x1, y1, …), VBO normal: (nx0, ny0, nz0, nx1,…)
- Type de primitive : points, triangle, lignes, etc
- Sous le drivers : « triangle strips »
- Cache critique
  - Réordonnancement de maillages [Sanders 2006]
  - Compression

Index Buffer (IBO)

- Stocke une géométrie indexée sur un VBO
- Tableau d'entier non signé, interprété selon le type courant de primitive utilisé (e.g., triangle)
- Cache critique
  - Réordonnancement de maillages [Sanders 2006]
  - Compression

Vertex Array (VAO)

- Un unique objet OpenGL liant
- un IBO
- un VBO (interleaved) ou un ensemble de VBOs (attributs séparés)

Framebuffer (FBO)

- Tableau/image 2D
- Sortie du pipeline GPU

- Contexte de la programmation GPU
  - FBOs non affichés
  - Rendu multi-passes et calcul multipass
    - Tampon de stockage intermédiaire
    - Calcul de chaque passe encapsulé dans les shaders
    - *1 pixel = 1 thread*

Framebuffer (FBO)

```
Concept OpenGL:
Texture        g_ActiveTextures[8];

struct FBO {
   Texture             m_Stencil;
   Texture             m_Depth;
   Texture             m_RenderTargets[8];
};
```

```
 C++:
 Texture tex;
 /*… charger les données */

 int iTextureSlot = 0;
 glActiveTexture (GL_TEXTURE0 + iTextureSlot);
 tex.Use ();
 m_Program.SetUniform1i (iTexLocation, iTextureSlot);
 iTextureSlot++;
```

Références

- OpenGL Red Book
- GPU Gems Séries
- GPU Pro Series
- developer.nvidia.com
- developer.amd.com
- Modern OpenGL