Programmation sur Carte Graphique

Introduction

sources:

Jean-Sébastien Franco, Romain Vergne & Xavier Granier (LaBRI)

Gabriel Fournier (LIRIS)
Christian Trefftz / Greg Wolffe

OpenGL Spec 2.1

GLSL Spec 1.3

But de ce cours

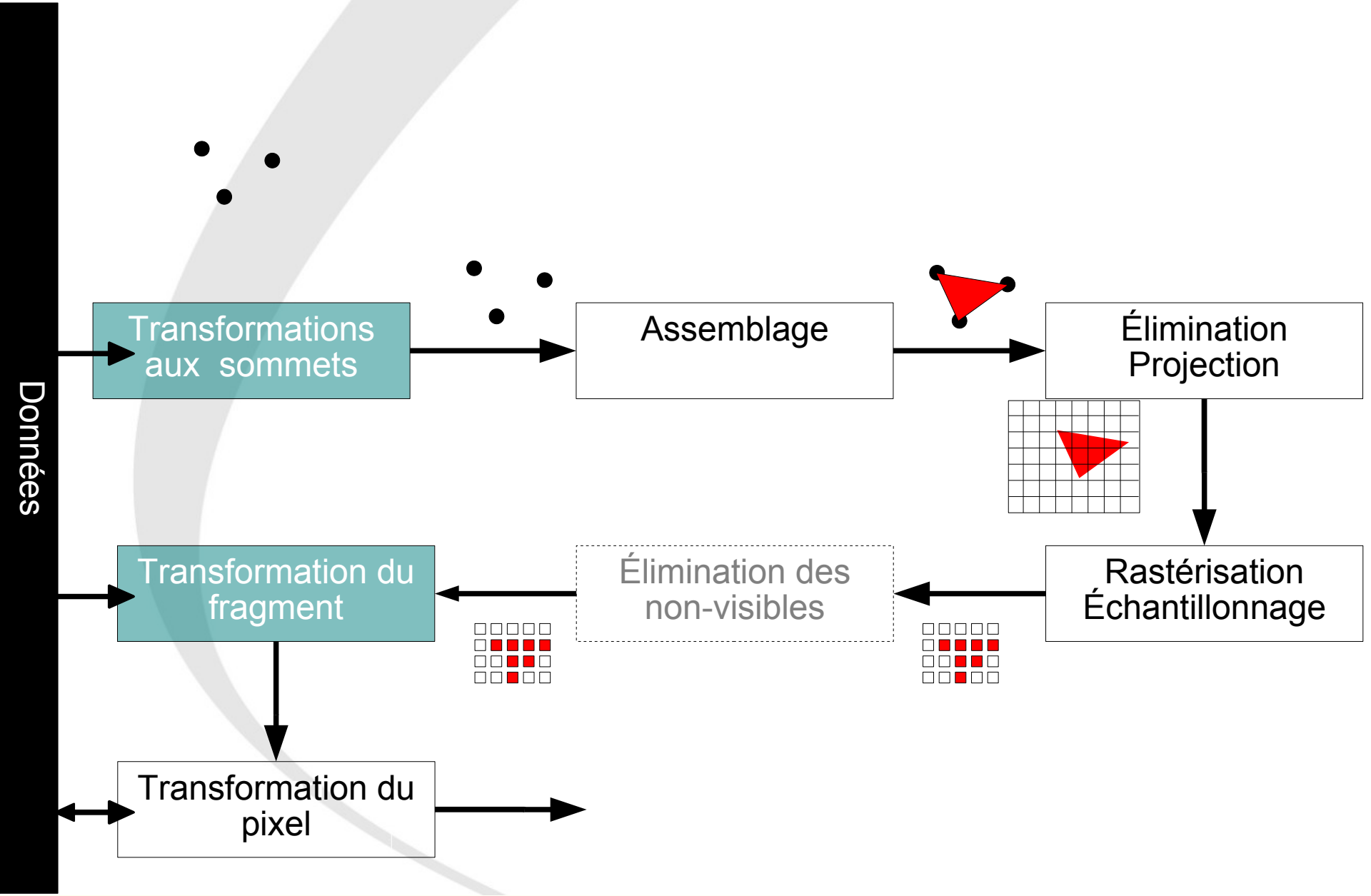
- **Non-buts**
  - Spécialistes GLSL
  - Experts en GLSL
- **Exploration du pipeline graphique**
  - Séparation sommet (vertex)/fragment/géométrie
  - Différentes possibilités
- **Lecture de documents techniques**
  - Spécifications
  - Description de langage

Pourquoi des GPU programmables?

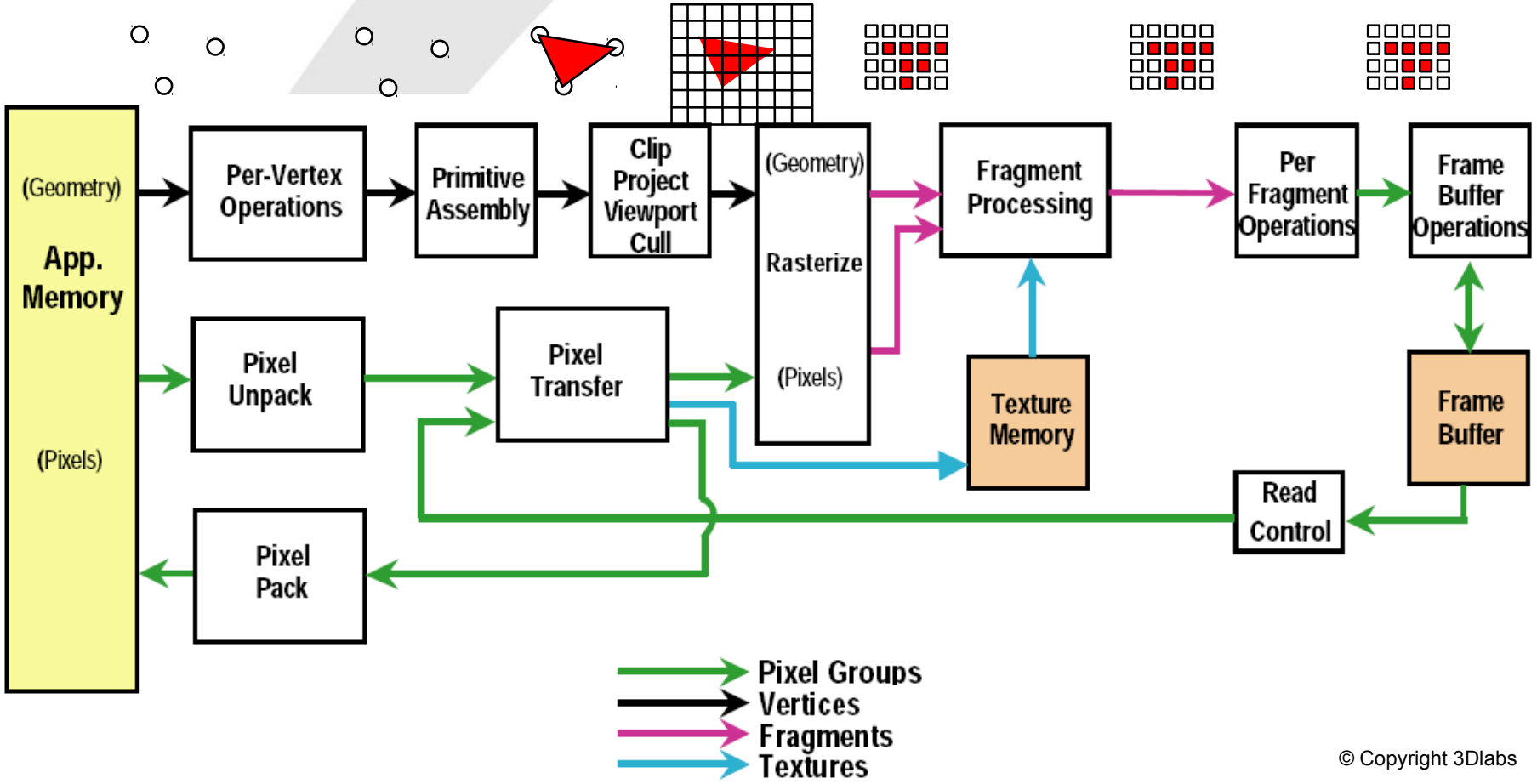
- **Pipeline classique figé et limité :**
  - Rendu : Éclairement calculé aux sommets
  - Textures : bitmaps
  - Projection : perspective ou parallèle.
- **Ne gère pas**
  - Calculs d'ombres
  - Réflexions, éclairement global
  - La physique des scènes (collisions, mouvements…)
- **Traitement des vertex, des fragments, et de la géométrie**

- **Pipeline programmable :**
  - Apparence plus sophistiquées (plus de détails)
  - Textures procédurales / Matériaux plus réalistes
  - Rendu non photo-réaliste
  - Animation procédurale
  - Traitement d'image
  - Anti Alissage
  - …

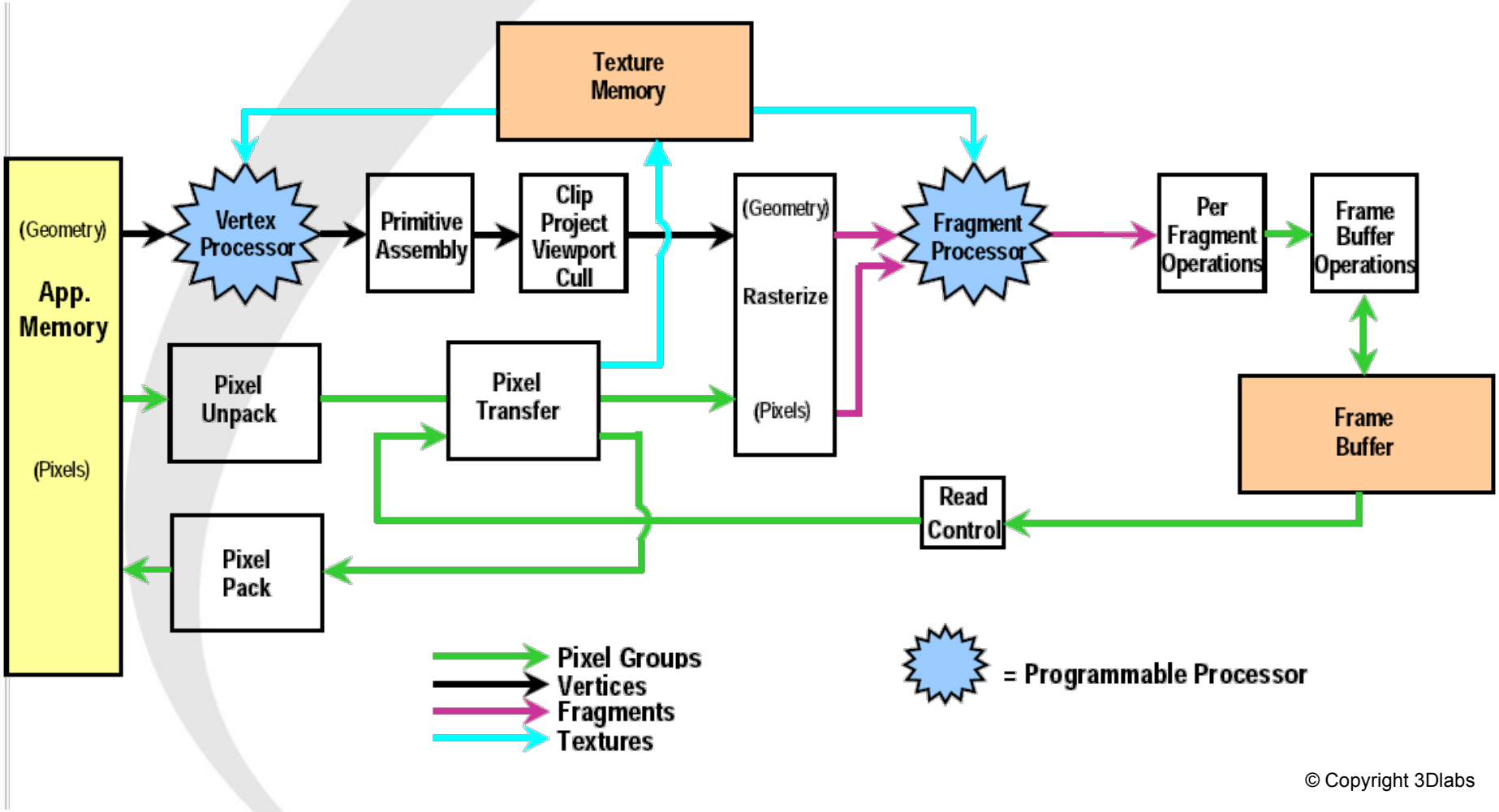
La chaîne de rendu



Pipeline Graphique



OpenGL 2.0



Langages et évolutions

- **2000 : Register Combiners**
  - NVIDIA GeForce 2
  - Opération sur les textures: premières flexibilités dans le pipeline
- **2001 : Assembleur**
  - Vertex Program : NVIDIA GeForce 3
  - Fragment Shader : ATI Radeon 8500
  - *bas niveau, peu lisible*
- **2002 : Cg**
  - C pour le Graphique
  - Langage de haut niveau
  - Indépendant de la plateforme: DirectX, OpenGL
  - *Compilateur externe*
- **2003 : HLSL (Microsoft)**
  - Langage de haut niveau
  - Uniquement pour DirectX/Direct3D
- **2004 : GLSL (3DLabs, OpenGL)**
  - Langage de haut niveau
  - Uniquement OpenGL
  - Compilateur intégré

Terminologies

- **DirectX/Direct3D**
  - *Vertex Shader*
    - Opération sur les sommets
  - *Pixel Shader*
    - Opération sur les fragments (après la rastérisation)
  - HLSL
    - *High Level Shading Language*
    - Langage de style C

Terminologies

- **OpenGL**
  - *Vertex Shader*
    - Opération sur les sommets
  - *Fragment Shader*
    - Opération sur les fragments (après la rasterisation)
  - GLSL
    - *OpenGL Shading Language*
    - Langage de style C
    - *Fragment Shader* / *Vertex Shader*

Vertex Shader

- **Peut faire**
  - Transformation des propriétés des sommets
    - Position/normale/couleurs/coordonnées texture/…
  - Calcul de l'éclairage
  - Génération de coordonnées texture
  - …
- **Ne peut pas faire**
  - Élimination des faces non-visibles
  - Backface culling
  - Assemblage des primitives
  - Réduire/augmenter le nombre de primitives
  - ...

Fragment Shader

- **Peut faire**
  - Opérations sur les valeurs interpolées
  - Texturing
  - Éliminer des fragments non-visibles
  - ....
- **Ne peut pas faire**
  - Histogramme
  - Alpha test
  - Test de profondeur
  - Stencil test
  - Alpha blending
  - Opération logique
  - Dithering
  - …

Pourquoi GLSL ?

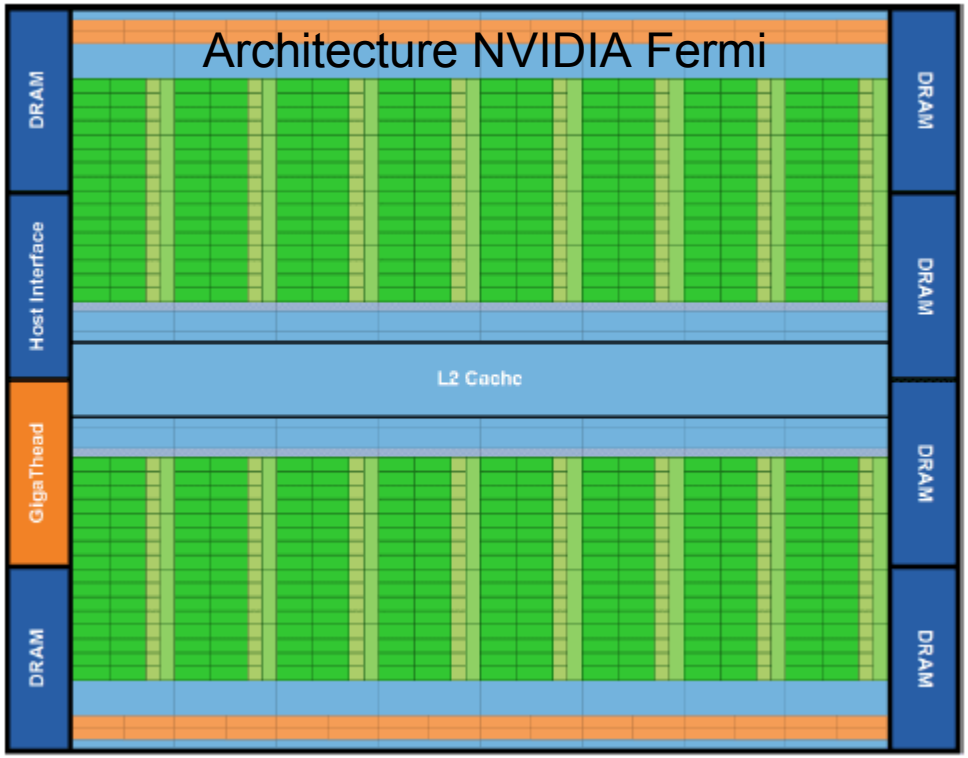
- **OpenGL**
  - Standard ouvert
  - Multi-plateformes
    - Unix (IRIX, Linux, MacOS X), Windows
- **Intégré aux drivers**
  - Mise à jour des pilotes
- **Extensible à d'autres langages**
  - Cg (*GL_EXT_Cg_shader*)
- **Similaire aux autres solutions**
  - HLSL / Cg

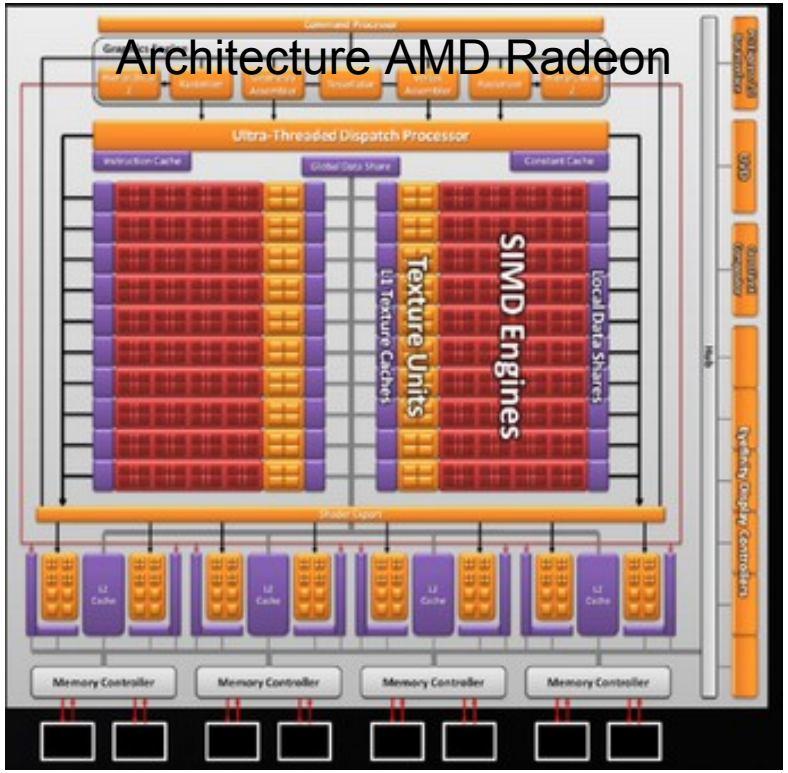
Modèle général de programmation: SIMD

- **Exécution parallèle**
- **Plusieurs vertex (ou fragment) sont traités simultanément par un même programme : SIMD**
  - **Flux de données dans un seul sens**
    - Pas de variable globale en écriture
    - Pas de sortie en lecture/écriture
  - **Traitement conditionnel (if) souvent coûteux.**
- **Pas d'accès aux sommets voisins**
- **Pas d'accès aux pixels voisins**
- **Pour certains effets : multi-passe**

Processeur massivement parallèle

- **Plus d'une centaine de processeurs**
- **Calcul vectoriel (SIMD)**
  - Données : Vecteur 4D
  - Opérations : Unitaire par composantes (+ - * / …) , produits scalaires, ...
- **Des unités spécialisées**
  - Compression/Décompression d'images
  - Conversion de primitives en pixels (rastérisation).

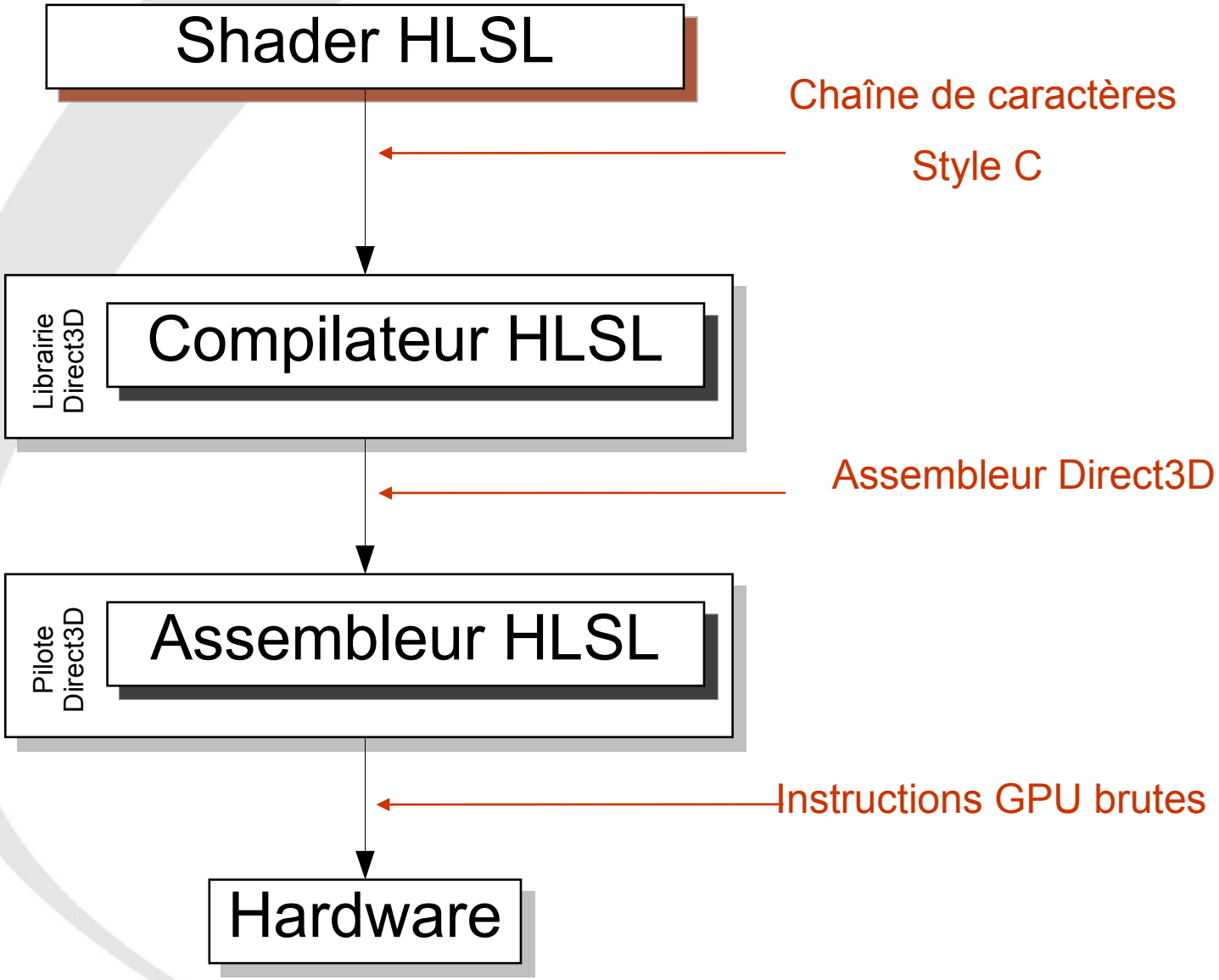




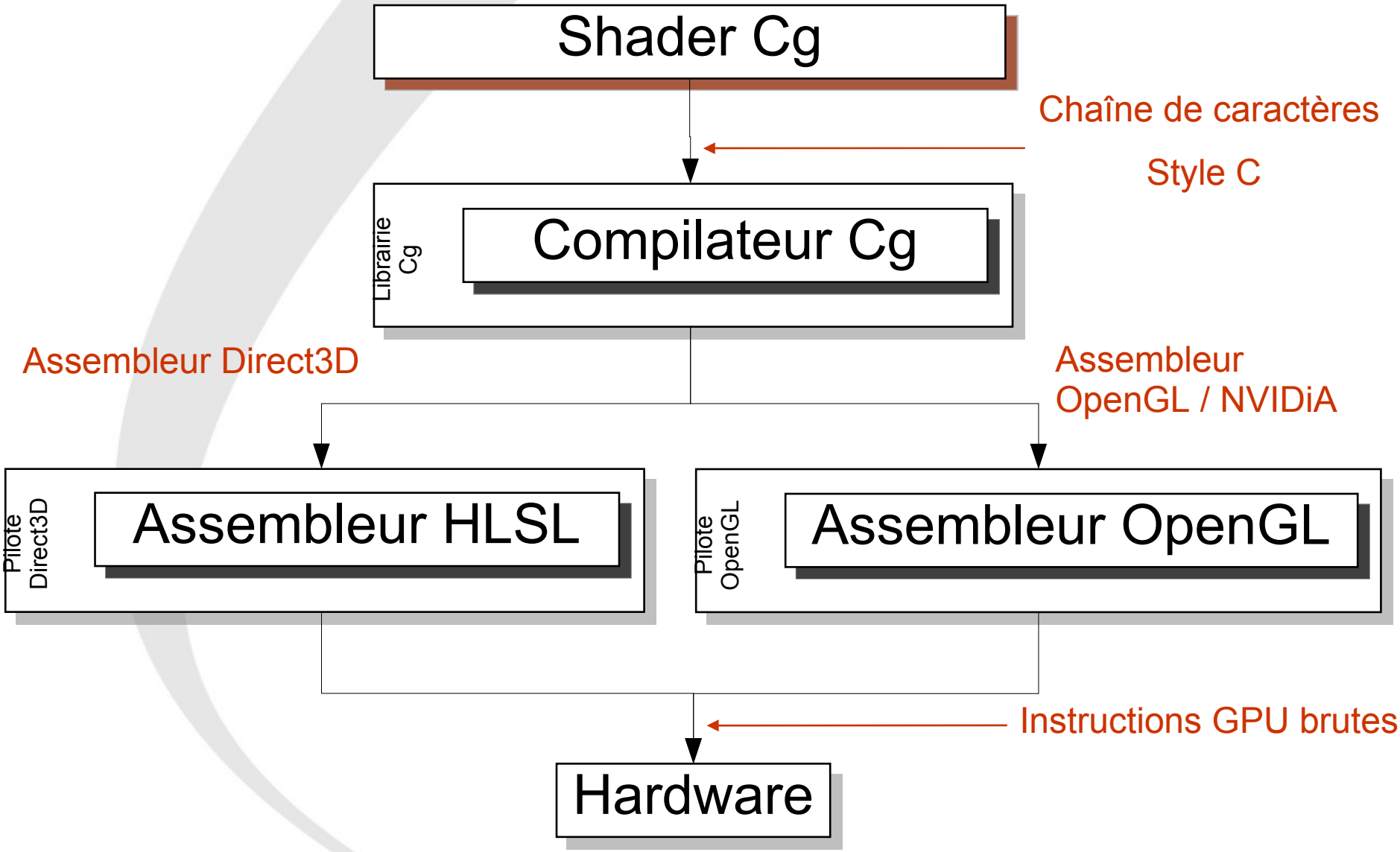
Modèles de production des shaders

- Modèle HLSL
- Modèle Cg
- Modèle GLSL

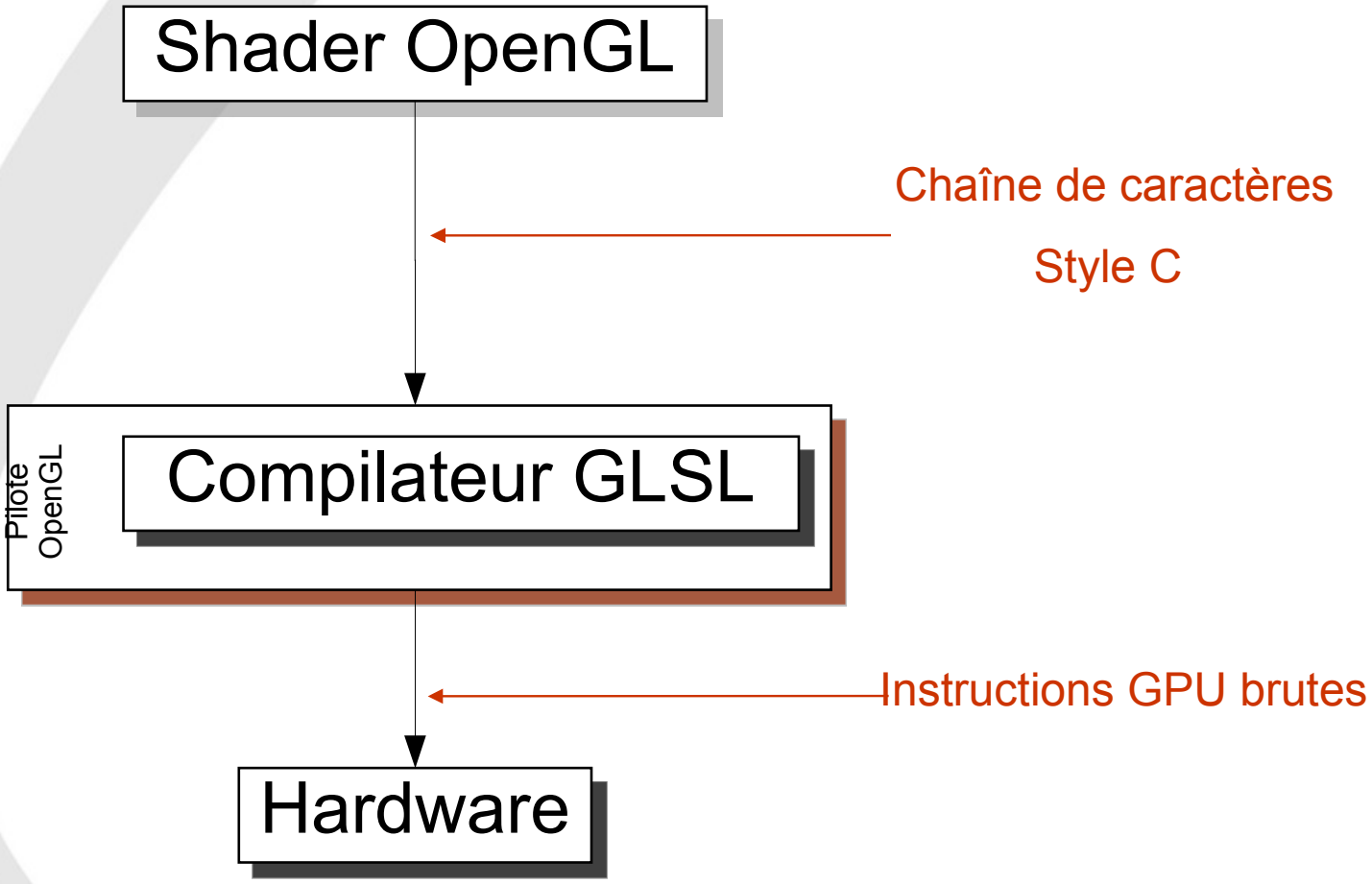
Modèle HLSL


Shader HLSL
Chaîne de caractères
Style C
Librairie Direct3D
Compilateur HLSL
Assembleur Direct3D
Pilote Direct3D
Assembleur HLSL
Instructions GPU brutes
Hardware

Modèle Cg


Shader Cg
Chaîne de caractères
Style C
Librairie Cg
Compilateur Cg
Assembleur Direct3D
Assembleur OpenGL / NVIDiA
Pilote Direct3D
Assembleur HLSL
Pilote OpenGL
Assembleur OpenGL
Instructions GPU brutes
Hardware

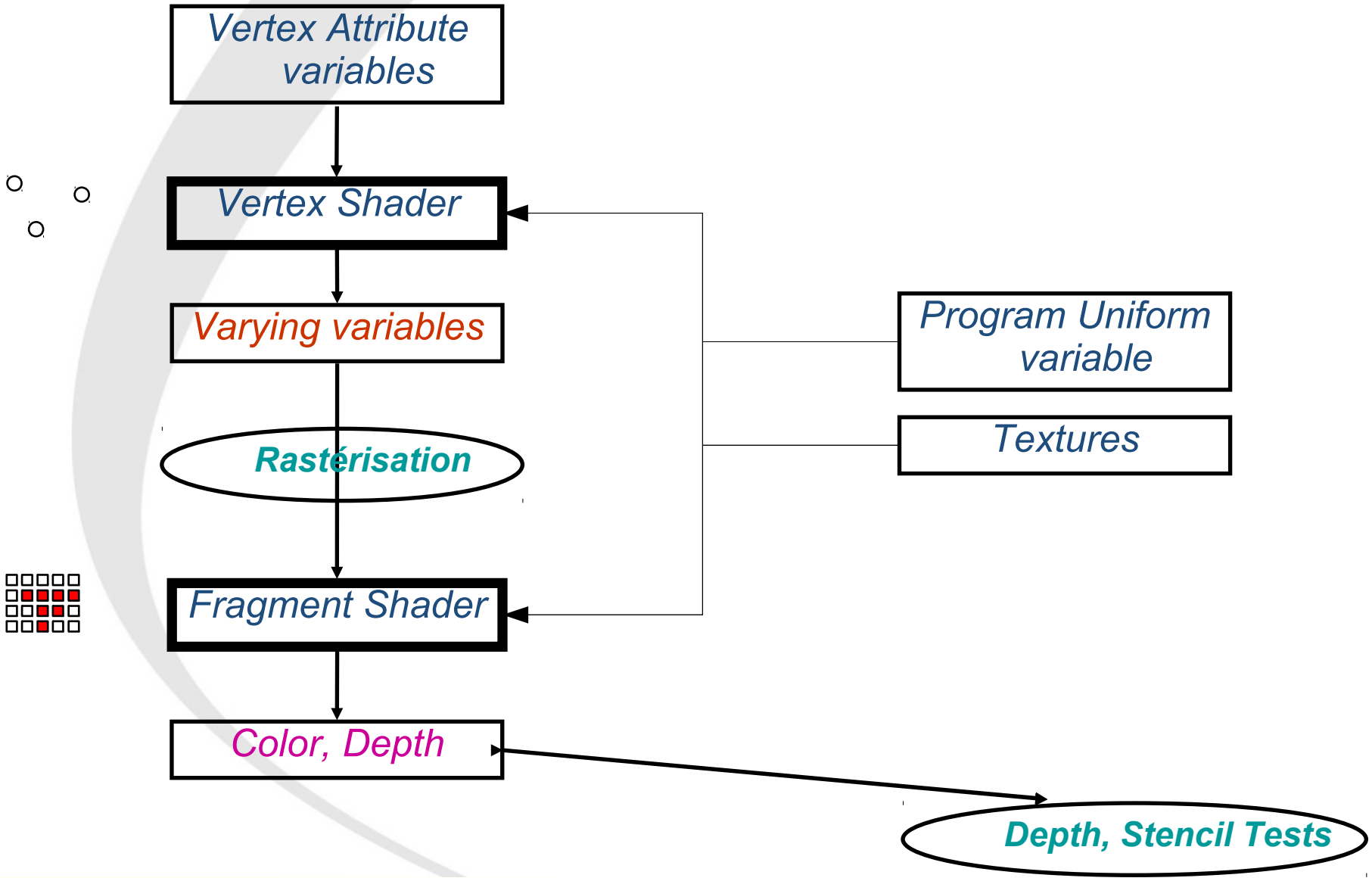
Modèle GLSL

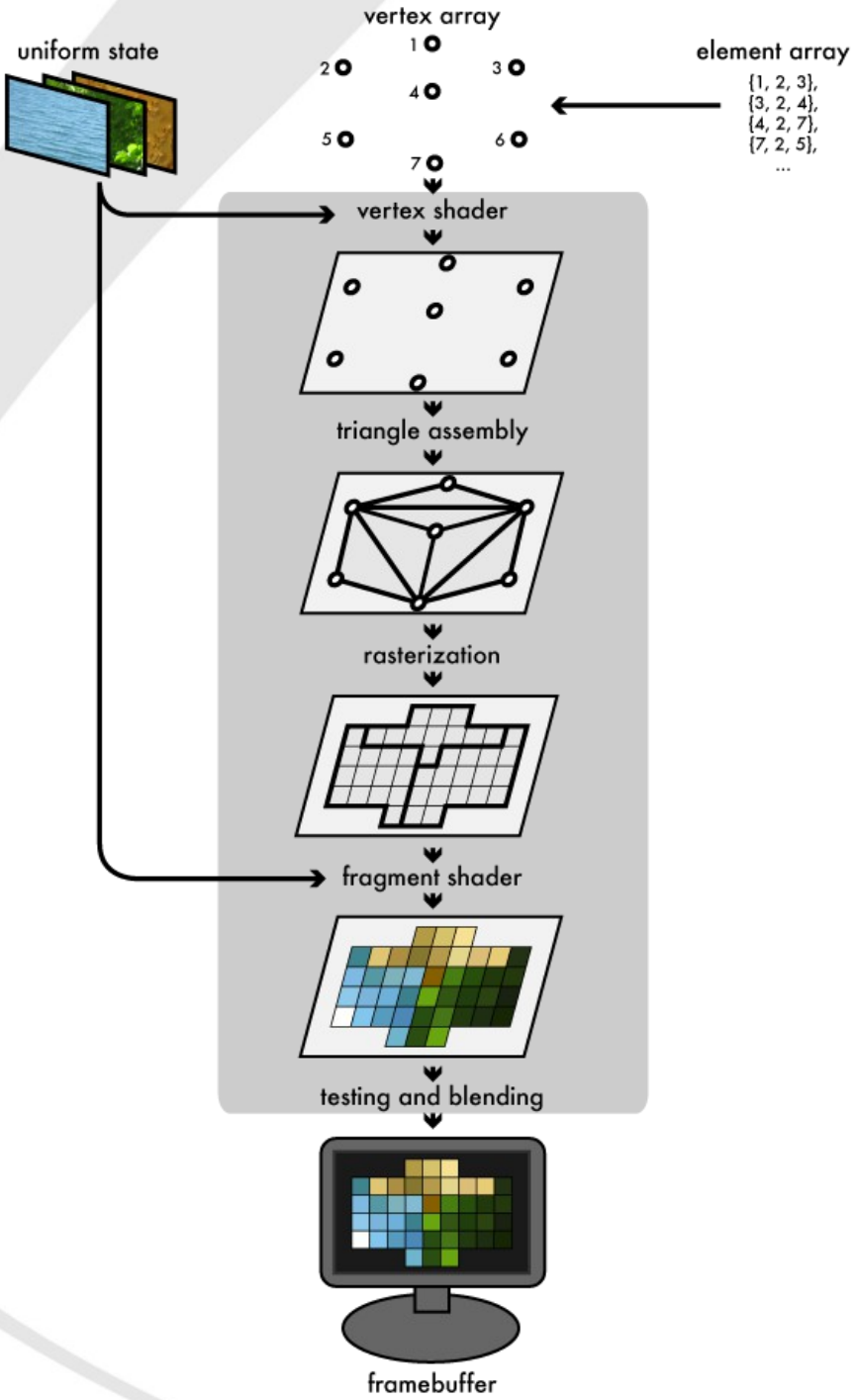

Shader OpenGL
Chaîne de caractères
Style C
Pilote OpenGL
Compilateur GLSL
Instructions GPU brutes
Hardware

Qualificateur de variables

- **Attributs :**
  - Changent typiquement pour chaque sommet
  - Uniquement dans Vertex Shader
  - Lecture seule
- **Uniforme :**
  - Changent au plus à chaque primitive
  - Communes à tous les shaders et à tout le pipeline
  - Lecture Seule
- **Varying :**
  - Communication entre Vertex et Fragment Shader
  - Écriture seule dans Vertex Shader
  - Lecture seule dans Fragment Shader
- **Constantes :**
  - Constantes locale à un Shader
  - Lecture seule
- **Sans qualifieur:**
  - Variables locales à un Shader (une seule exécution)
  - Lecture et écriture locale
- **PAS de variables static !**

Variables et communication

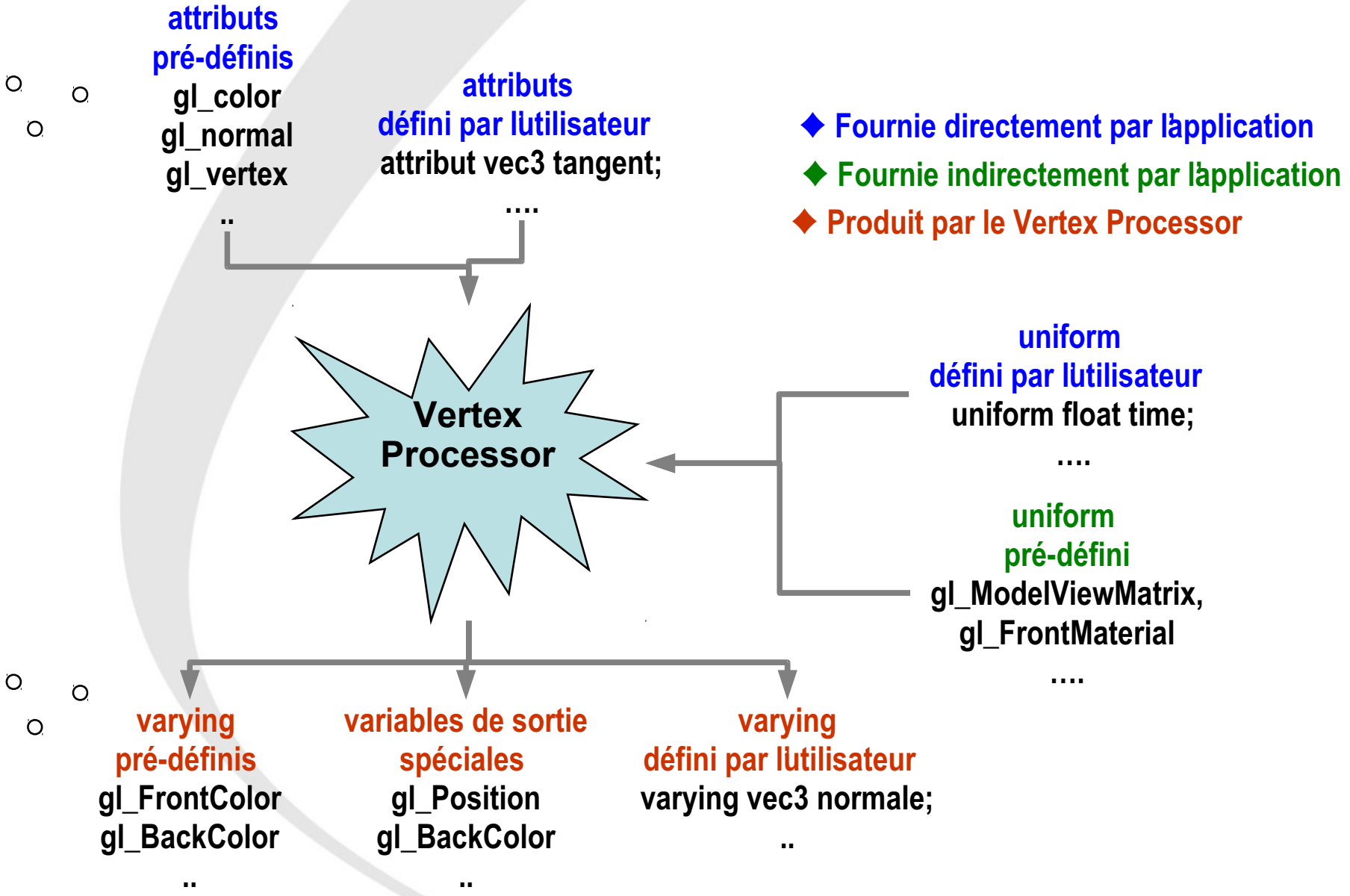



vertex array
uniform state
element array
1
2
3
4
5
6
7
{1, 2, 3},
{3, 2, 4},
{4, 2, 7},
{7, 2, 5},
...
vertex shader
triangle assembly
rasterization
fragment shader
testing and blending
framebuffer

Vertex Shader

- **Opérations traditionnelles**
  - Transformation spatiale des sommets et des normales
  - Calcul des coordonnées dans la texture
  - Éclairement
- **Entrées**
  - **Attribute variables** : associées à chaque sommet
    - Traditionnelles: gl_color, gl_normal, gl_vertex, gl_multiTexCoord0,…
    - Définies par l'utilisateur
  - **Uniform variables** : associées au programme
    - Traditionnelles : gl_modelViewMatrix, gl_frontMaterial, gl_lightSource0, …
    - Définies par l'utilisateur
  - **Textures**
- **Sorties**
  - **Spéciales** : gl_position, gl_pointSize, gl_clipVertex
  - **Varying variables** : entrée du fragment shader, interpolées entre les sommets
    - Traditionnelles : gl_frontColor, gl_backColor, gl_fogFragCoord,…
    - Définies par l'utilisateur

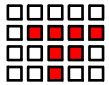
Vertex Shader



Vertex Shader

- **Ce qui peut/doit être fait**
  - transformations de modélisation et de projection des sommets (obligatoire), et des normales
  - éclairage par sommet (+ color material)
  - génération et transformation des coordonnées de texture
  - calcul par sommet de la taille de la primitive point (optionnel)
  - plans de clipping supplémentaires

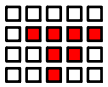
- **Ce qui ne peut pas être fait/remplacé**
  - clipping par le volume de visualisation, division perspective, transformation dans le repère du viewport
  - backface culling, sélection de la face avant ou arrière
  - assemblage des primitives, ...



Fragment shader

- **Opérations traditionnelles**
  - Opérations sur les valeurs interpolées
  - Application des textures
  - Calcul de la couleur du pixel
- **Entrées :**
  - **Varying variables** : interpolées entre les sommets
    - **Traditionnelles** : gl_color, gl_SecondaryColor, gl_TexCoord0,…
    - **Définies par l'utilisateur**
  - **Spéciales** : gl_FragCoord, gl_FrontFacing
  - **Uniform variables** : associées au programme
    - Les mêmes que pour le vertex shader
  - **Textures**
- **Sorties :**
  - gl_FragColor, gl_FragDepth

Fragment Shader



varying
pré-définis
gl_FrontColor
gl_BackColor
..

variables dèntrée
spéciales
gl_FragCoord
gl_Color
..

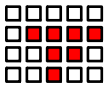
varying
défini par lùtilisateur
varying vec3 normale;
..

- Produit par la rasterisation
- Fournie directement par làpplication
- Fournie indirectement par làpplication
- Produit par le fragment processor

Fragment Processor

uniform
défini par lùtilisateur
uniform float time;
....

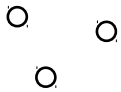
uniform
pré-défini
gl_ModelViewMatrix,
gl_FrontMaterial
....



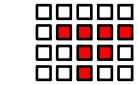
variables de sortie
spéciales
gl_FragDepth
gl_FragColor
..

Exemple

```
//Vertex Shader
uniform float temp_min;
uniform float temp_max;
attribute float vertex_temp;

varying float temperature;

void main()

{
    temperature = (vertex_temp-temp_min)/(temp_max-temp_min);
    gl_position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```



```
//Fragment Shader
uniform vec3 col_froid;
uniform vec3 col_chaud;
varying float temperature;

void main()

{
    vec3 color = mix(col_froid,col_chaud,temperature);
    gl_FragColor = vec4(color,1.0);
}
```



Différences avec ANSI C

- **Vecteur** : 2-, 3-, ou 4- dimensions
- **Matrices de flottants** : 2x2, 3x3, or 4x4
- Type “**sampler**” pour l’accès aux textures
- Qualificateur de types
  - “attribute”, “uniform”, et “varying”
- Accès aux états OpenGL
- Fonctions pré-définies pour le graphique
- Mot-clef “**discard**” pour tuer un fragment / une primitive
- Accès aux membres des vecteurs
  - .**rgba**, .**xyzw**, .**stpq**
  - Permutations (.**xxx**, .**stst**, …)

Types

- **float, vec2, vec3, vec4**
  - 1, 2, 3, ou 4 flottants
- **int, ivec2, ivec3, ivec4**
  - 1, 2, 3, ou 4 entiers
  - Pas de nécessité de support matériel
  - Limitée à 16 bits de précision, avec le signe
  - Pas de garantie de bouclage
  - pas d'operateurs C << & ...
- **bool, bvec2, bvec3, bvec4**
  - 1, 2, 3, ou 4 booléens
  - Pas de nécessité de support matériel
- **Pas de promotion de type :**
  - float a=3;//FAUX ➡ float a=3.0;
- **Conversions :**
  - float a=float(3); vec4 b=vec4(3);//tous à 3

Types (suite)

- **mat2, mat3, mat4**
  - Matrices de flottants
  - accès: Mat4 m; vec4 v=m[0]; float f=m[0][0];
- **void**
  - Pour les fonctions sans valeur retournée
- **sampler[123]D, samplerCube**
  - accès aux textures 1D, 2D, et 3D
  - Texture « cube map »
- **sampler1DShadow, sampler2DShadow**
  - Texture de profondeurs 1D et 2D
- **Construction de types**
  - Structures (**struct** comme en C, mais pas union, pas besoin typedef)
  - Tableau: *taille pas forcement précisée à la déclaration mais connue à la compilation*

GLSL: opérateurs et tableaux

- **Opérateurs:**
  - **+, -, *, /, ++, --**
    - Terme à terme sauf : mat * vec, vec*mat (=vec) et mat * mat
    - matrixcompmult(mat,mat) : terme à terme
  - **<=, >=, <, >** sur int et float scalaire
  - **==, !=** terme à terme sur tous les types sauf les tableaux
  - **&&, ||, ^^** sur bool (scalaire), *(pas d'opérateur bits à bits)*
    - Introduits dans GLSL 1.4
  - **=, +=, -=, *=, /=, ?:**
  - **lessThan(vec, vec), lessThanEqual, equal, notEqual, greater, greaterThan** : terme à terme sur les vecteurs
  - **any(bvec), all(bvec), not(bvec)**

- **Indexation et swizzling**
  - **[n] sur tableau, vec, mat: accède au n-ième élément**
  - **. permute les éléments :**
    - ivec4 a=ivec4(1,2,3,4); ➡ a.wzyx est égal à {4,3,2,1}

Fonctions et structures de contrôle

- **Fonctions**
  - cos, sin, tan, acos, asin, atan, radians, degrees
  - pow, exp2, log2, sqrt, inversesqrt
  - abs, sign, floor, ceil, fract, mod, min, max, clamp, step, smoothstep
  - length, distance, dot, cross, normalize, faceforward, reflect
  - noise (non disponible sur la plupart des cartes)
  - texture1D, texture2D, texture3D, textureCube ...

- **Structures de contrôle:**
  - if, if-else
  - for, while, do-while
  - return, break, discard

- **Fonctions**
  - **Paramètres qualifiés : in, out, inout, par défaut : in.**
    - void calcu_coord(in vec3 pos, out coord)

- **Programme principal:**
  - void main()

GLSL : les constructeurs

- **Utilisés pour**
  - convertir un type en un autre
  - initialiser les valeurs d'un type

- **Exemples:**

```
vec3 v0 = vec3(0.1, -2.5, 3.7);
float a = float(v0);        // a==0.1
vec4 v1 = vec4(a);          // v1==(0.1, 0.1, 0.1, 0.1)

struct MyLight {
 vec3  position;
 float intensité;
};
MyLight light1 = MyLight(vec3(1,1,1), 0.8);
```

GLSL : manipulation des vecteurs

- **Nommage des composantes**
  - via *.xyzw* ou *.rgba* ou *.stpq*
  - ou [0], [1], [2], [3]
- **Peuvent être ré-organisées, ex:**

```
vec4 v0 = vec4(1, 2, 3, 4);
v0 = v0.zxyw + v0.wwyz;        // v0 = (7,5,4,7)
vec3 v1 = v0.yxx;           // v1 = (5,7,7)
v1.x = v0.z;                // v1 = (4,7,7)
v0.wx = vec2(7,8);          // v0 = (8,5,4,7)
```

- **Manipulation des matrices**

```
mat4 m;
m[1] = vec4(2);    // la colonne #1 = (2,2,2,2)
m[0][0] = 1;
m[2][3] = 2;
```

GLSL : les fonctions

- **Arguments : types de bases, tableaux ou structures**
- **Retourne un type de base ou void**
- **Les récursions ne sont pas supportées**
- **les arguments peuvent être *in, out, inout***
  - Par défaut les arguments sont "in"

- **Exemple**

```
vec3 myfunc(in float a, inout vec4 v0, out float b)
{
 b = v0.y + a;  // écrit la valeur de b
 v0 /=  v0.w;   // màj de v0
 return v0*a;
}
```

Extensions OpenGL

- **Plusieurs types**
  - GL_ : extensions OpenGL
  - WGL_/AGL_/GLX_ : communication avec le fenêtrage
- **Plusieurs sources**
  - ARB : organisme de standardisation OpenGL (officiel)
  - EXT : couramment supportée
  - ATI/NV/SGI/… : vendeur-spécifique
  - ATIX/NVX/SGIX/ … : expérimentale
  - OES/OML : pour OpenGL|ES et OpenML (consortium Khronos)
  - …

Extension : Spécification

Name

ARB_texture_rectangle

Name Strings

GL_ARB_texture_rectangle

….

Version

Date: March 5, 2004 Revision: 0.9

Number

ARB Extension #38

Dependencies

OpenGL 1.1 is required

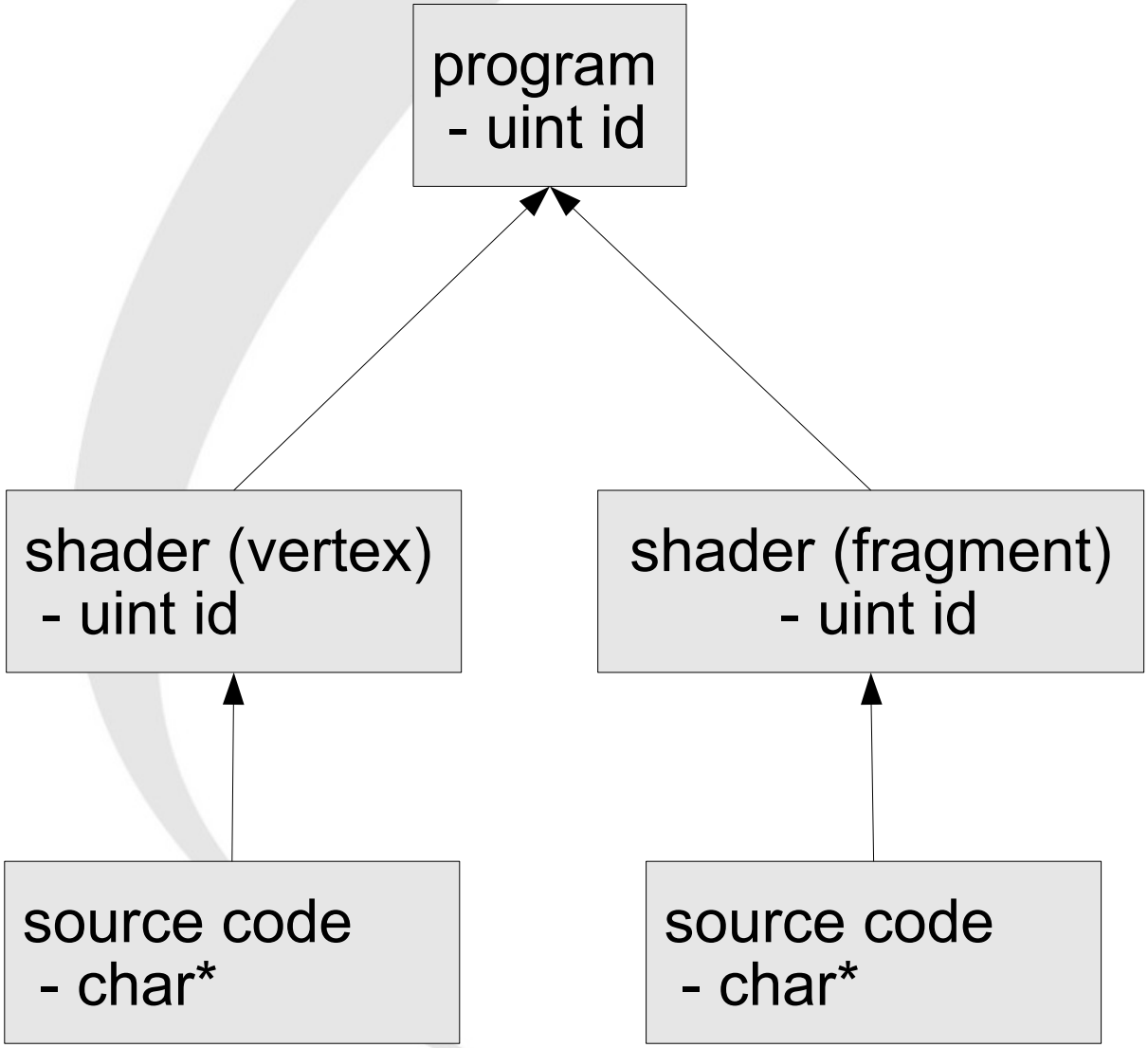
OpenGL 1.4 (or ARB_texture_mirrored_repeat) affects the definition of this extension.

…..

Overview

- .....

OpenGL: program & shader



glCreateProgram()
glLinkProgram()
glUseProgram()

glAttachShader()

glCreateShader()
glCompileShader()

glShaderSource()

OpenGL API : Shader et Program

- **ShaderObject**
  - Équivalent à un .o en C
  - Un même Shader peut être découpé en modules dans plusieurs ShaderObject
  - Création
    - GLhandleARB glCreateShaderObjectARB(GLenum shaderType)
  - Chargement
    - void glShaderSourceARB(GLhandleARB shader, GLuint nstrings, const GLcharARB **strings, GLint *lengths)
  - Compilation
    - void glCompileShaderARB(GLhandleARB shader)
- **Program**
  - Équivalent à l'exécutable
  - Création
    - GLhandleARB glCreateProgramObjectARB(void)
  - Rattachement
    - void glAttachObjectARB(GLhandleARB program, GLhandleARB shader)
  - Création des liens
    - void glLinkProgramARB(GLhandleARB program)
  - Insertion dans le pipeline
    - void glUseProgramObjectARB(GLhandleARB program)

OpenGL API : interaction avec le shader

- **Spécification des attributs des sommets**
  - glColorXX(…) ➔ gl_color , ….
  - glVertexAttribXXARB(GLuint index, …) ➔ attribut générique
  - Pour envoyer le sommet, on finit par :
    - glVertexXX(…) ou
    - glVertexAttribXXARB(0, …)
- **Lien entre attributs génériques et variables attributs du Shader**
  - glBindAttribLocationARB(GLhandleARB program, GLuint index, **const GLcharARB *name)**
    - glBindAttribLocation(shader_id,1, ''couleur '');
  - GLint glGetAttribLocationARB(GLhandleARB prog, const GLcharARB *name)
    - Disponible après la création des liens , -1 si erreur
- **On peut utiliser les vertex arrays**
  - void glVertexAttribPointerARB(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid *pointer)

OpenGL API : Interaction avec le Shader

- **Spécifications des variables uniformes**
  - GLint glGetUniformLocationARB(GLhandleARB program,
    const GLcharARB *name)
    - Après le linkage, -1 si erreur
  - glUniformXXARB(GLuint location, …)
  - glUniformMatrix (GLuint location, GLint location,
    GLuint count, GLboolean transpose,
    const GLfloat *v);

Au travail !

Créer un ShaderProgram contenant les programmes suivants.
Dessiner un triangle en passant une température différente à chacun de ses sommets

```
//Vertex Shader : temp_vs.txt
uniform float temp_min;
uniform float temp_max;
attribute float vertex_temp;

varying float temperature;

void main()

{
    temperature = (vertex_temp-temp_min)/(temp_max-temp_min);
    gl_position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
//Fragment Shader : temp_fs.txt
uniform vec3 col_froid;
uniform vec3 col_chaud;
varying float temperature;

void main()

{
    vec3 color = mix(col_froid,col_chaud,temperature);
    gl_FragColor = vec4(color,1.0);
}
```

Exemple : Création des shaders

```
GLhandleARB prog=glCreateProgramObjectARB();
GLhandleARB vertex_s=glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
GLhandleARB fragment_s=glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

char* src=readFile(“vertex.txt’’);
glShaderSourceARB(vertex_s,1,(const GLcharARB**)&src,NULL);
glCompileShaderARB(vertex_s);

src=readFile(‘’fragment.txt’’);
glShaderSourceARB(fragment_s,1,(const GLcharARB**)&src,NULL);
glCompileShaderARB(fragment_s);

glAttachObjectARB(prog,vertex_s);
glAttachObjectARB(prog,fragment_s);
glLinkProgramARB(prog);

int r=0;
glGetObjectParameterivARB(prog,GL_OBJECT_LINK_STATUS,&r);

glUseProgramObjectARB(prog);
```

Exemple : Attachement des paramètres

```
int temp_min_uni, temp_max_uni, col_min_uni, col_max_uni;
int temp_att;

temp_min_uni=glGetUniformLocationARB(prog,''temp_min'');
temp_max_uni=glGetUniformLocationARB (prog,''temp_max'');
col_min_uni=glGetUniformLocationARB (prog,''col_froid'');
col_max_uni=glGetUniformLocationARB (prog,''col_chaud'');

temp_att=glGetAttribLocationARB(prog, ''temperature'');

glUniform1fARB(temp_min_uni,-10);
glUniform1fARB(temp_max_uni,30);
glUniform3fARB(col_min_uni,0,0,1);
glUniform3fARB(col_max_uni,1,0,0);

glBegin(GL_TRIANGLE);
glVertexAttrib1fARB(temp_att,-5);
glVertex3f(0,1,0);
glVertexAttrib1fARB(temp_att,5);
glVertex3f(-1,0,0);
glVertexAttrib1fARB(temp_att,15);
glVertex3f(1,0,0);
glEnd();
```

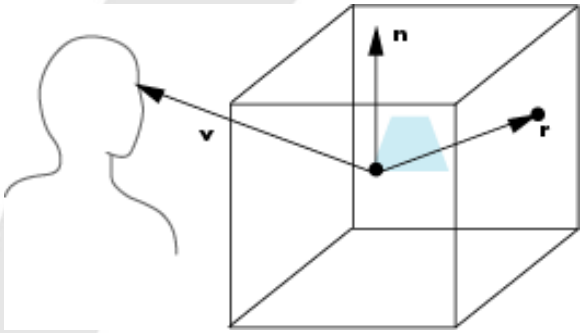
Les textures

- **Dans l'application**
  - Choisir une unité de texture : ***glActiveTexture( )***
  - Créer une texture et l'associer à l'unité de texture ***glBindTexture( )***
  - Fixer les paramètres (wrapping, filtering) : ***glTexParameter ( )***
  - Associer une image ***glTexImage ( )***
  - Initialiser le sampler du shader : ***glUniform1i( )***
- **Dans le Shader :**
  - Déclarer un uniform de type Sampler ***(sampler1D, sampler2D, sampler3D, samplerCube, samplerShadow1D, samplerShadow2D)***
  - Pour accéder à un élement de la texture utiliser la fonction textureXXX ***(texture1D, …)***
    - texture2D(sampler, coord);

Programmation

- **Développez petit à petit et testez souvent!**
  - Débuggage très difficile (pas de printf !)
- **Optimisation**
  - Réfléchissez au meilleur endroit pour placer un calcul:
    - Vertex shader : 1x par sommet
    - Fragment shader : 1x par fragment : beaucoup plus souvent !
  - Si un paramètre change
    - Vite : fragment shader
    - Lentement : vertex shader
    - Rarement : CPU ➔ uniform
  - Textures pour encoder les fonctions trop complexes
  - Utilisez les fonctions fournies plutôt que de les re-développer.
  - Choisissez la bonne fonction : inversesqrt a des chances d'être plus rapide que 1/sqrt.

Environment mapping



- **Utiliser la direction réfléchie pour accéder à la texture**

- **L'accès à un « cube map » se fait par le sampler cubemap.**
  vec4 texColor = textureCube(mycube, texcoord);
  - La coordonée doit être 3D

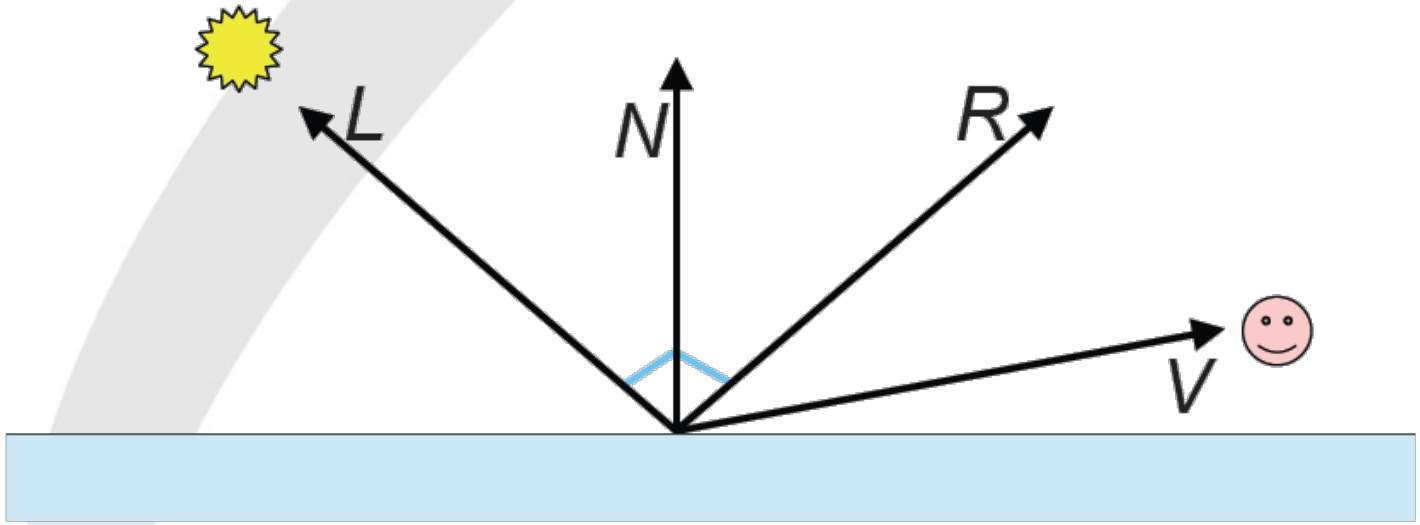
Cube Map Vertex Shader

```
uniform mat4 modelMat;
uniform mat3 invTrModelMat;
uniform vec4 eyew;
varying vec3 reflectw;
void main(void)
{
    vec4 positionw = modelMat*gl_Vertex;
    vec3 normw = normalize(invTrModelMat*gl_Normal);
    vec3 viewv = normalize(eyew.xyz-positionw.xyz);
   /* reflection vector in world frame */
   reflectw = reflect(normw, viewv);

   gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Cube Map Fragment Shader

```
/* fragment shader for reflection map */
varying vec3 reflectw;
uniform samplerCube MyMap;
void main(void)
{
    gl_FragColor = textureCube(myMap,      reflectw);
}
```

Au travail !

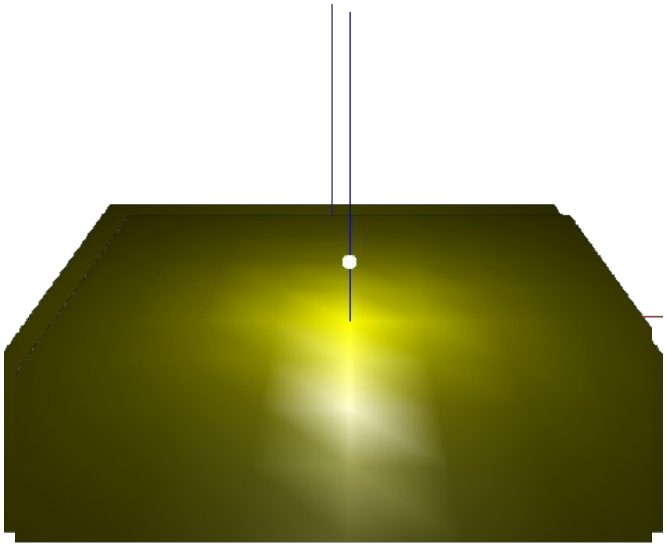


- Color=AmbientColor+
  DiffuseColor * (*N*. *L*)+
  SpecularColor * (*V*. *R*)^*n*

Calcul par sommet

```
void main() {

    gl_Position= gl_ModelViewMatrix* gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix* gl_Normal) ;
    vec3 L = normalize(gl_LightSource[0].position);

    gl_FrontColor=gl_FrontMaterial.diffuse*gl_LightSource[0].ambient ;
    float diffuse = dot(N, L);
    If (diffuse>0)
    {
       gl_FrontColor+= gl_FrontMaterial.diffuse*diffuse *gl_LightSource[0].diffuse ;
       vec3 R = reflect(-L, N);
       vec3 V = normalize(-gl_Position.xyz);
       float specular= dot(V, R) ;
       if (specular>0)
           gl_FrontColor+= gl_FrontMaterial.specular *pow(specular,
    gl_FrontMaterial.shininess)*gl_LightSource[0].specular ;
    }
    gl_Position= gl_ProjectionMatrix* gl_Position;
}
```

Phong par pixels



- **Par sommet : on interpole la couleur linéairement entre chaque sommet**
- **Par pixel :**
  - Le vertex programme calcule une normale dans le repère de l'oeil
  - Le fragment programme reçoit une normale interpolée et réalise le calcul complet avec cette normale

Le rendu multi passes

- **Problèmes types:**
  - Filtres : besoin d'accéder aux pixels voisins
  - Rendu HDR et tone mapping :
    - le rendu s'effectue 32 bits par composante (on ne peut pas l'afficher)
    - Séparation du rendu et du tone mapping
  - Deffered shading :
    - shader très complexe,
    - 1ere passe : rendu rapide dans plusieurs buffers (position, normale, paramètre…)
    - 2nd passe : calcul de l'éclairement uniquement pour les pixels réellement affichés
  - Textures dynamiques (réflexions, …)

- **Frame Buffer Object (FBO)**
  - Contexte de rendu hors écran
  - Associable à une texture
  - Changement de FBO rapide

Frame Buffer Object

- **FBO**
  - Contient des buffers de couleurs, stencil, profondeur
  - Multiple render target (MRT)
  - Associable à une texture
- **Render Buffer**
  - Comme FBO, non associé à une texture
  - Read/Write Pixels
- **Usage (performences décroissantes):**
  - Unique FBO, plusieurs textures attachées (même taille) : glDrawBuffer() pour changer de destination
  - Unique FBO, plusieurs textures attachées (même taille): FramebufferTexture() pour changer
  - Multiples FBO : BindFramebuffer() pour changer
    - obligatoire si rendu dans des textures de différentes taille...

API des FBO

Création

void glGenFramebuffersEXT (sizei n, uint *framebuffers) ;

void glDeleteFramebuffersEXT(sizei n, uint *framebuffers) ;

Association comme contexte de rendu

void glBindFramebufferEXT(enum target, uint framebuffer);

target =FRAMEBUFFER_EXT, framebuffer =0 : window,= id du fbo

Vérification de l'état

enum glCheckFramebufferStatusEXT(enum target);

➔ FRAME_BUFFER_COMPLETE

Association d'une texture au FBO

void glFramebufferTexture2DEXT(enum target, enum attachment, enum textarget, uint texture, int level);

target= FRAMEBUFFER_EXT,

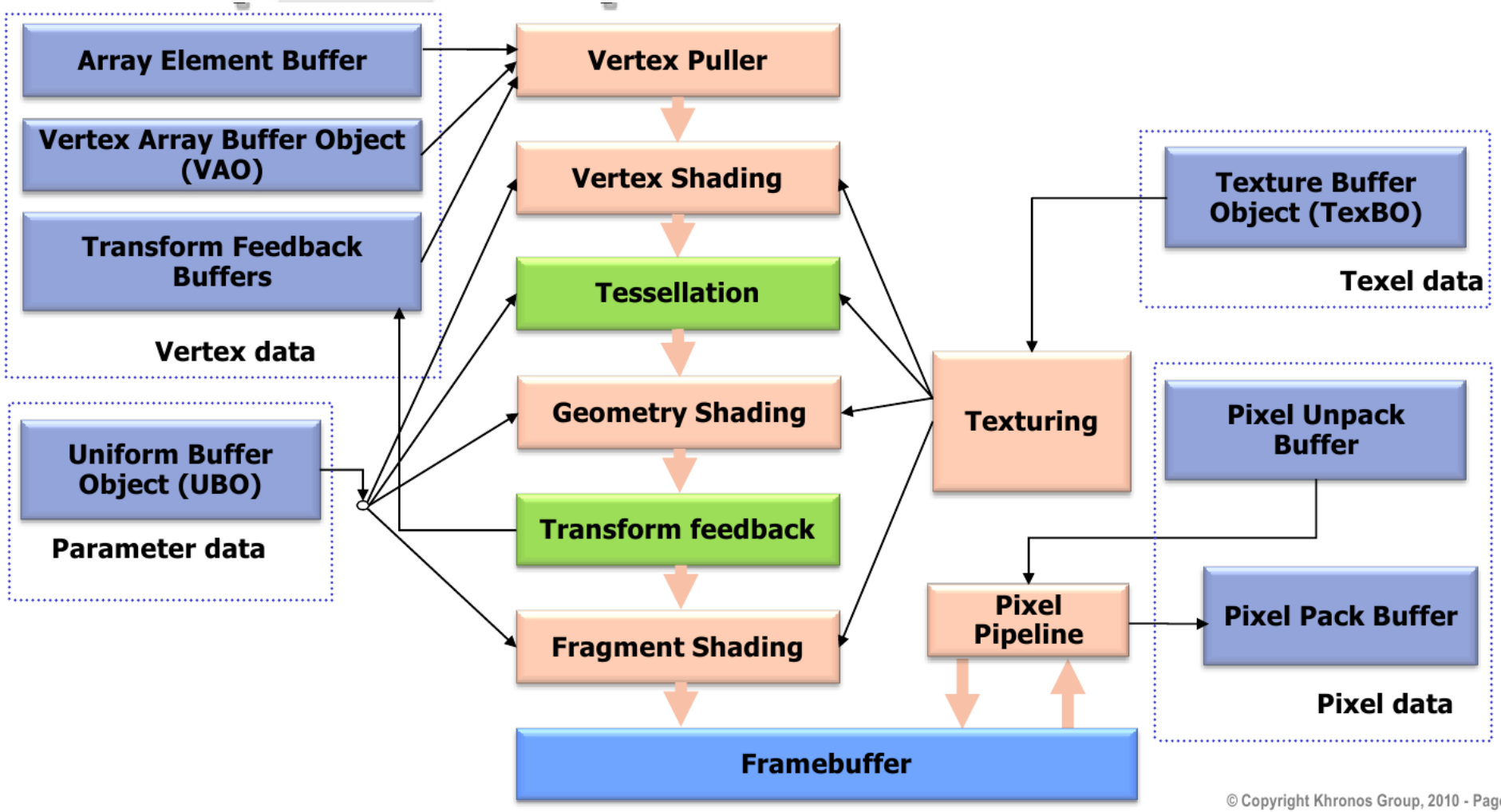
attachment = COLOR_ATTACHMENT0_EXT, DEPTH_ATTACHMENT_EXT …

textarget=TEXTURE2D,TEXTURE_RECTANGLE,…

Exemple

```
GLuintfb, depth_rb, tex;
glGenFramebuffersEXT(1, &fb);
glGenTextures(1, &tex);

glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fb);
glBindTexture( GL_TEXTURE_2D, tex);
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, NULL);

glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, tex, 0);

glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fb);
RENDU DANS LA TEXTURE

glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, 0);
glBindTexture( GL_TEXTURE_2D, tex);
RENDU DANS LA FENETRE AVEC LA TEXTURE
```

OpenGL 4 : All buffers



GPGPU

- **GPU évoluent vers une forme de CPU quasi-généraliste massivement parallèle**
- **Pipeline programmable de plus en plus flexible**
  - types de moins en moins restreints: float32, int, en hard
  - shaders de moins en moins limités en taille
  - architecture évolue vers l'exécution de threads plus ou moins généralistes sur plein de cores.
- **Explosion du GPGPU:**
  - utilisation du GPU pour tâches généralistes (vision, Image processing, simulations, calcul scientifique...)
  - Pourquoi?
    - GTX 280: 1 Teraflops. $400.
    - Intel Core Duo 3GHz: <50GFlops. >$500
  - Intel entre dans la danse des GPU/GPGPU: GPU Larrabee...
  - frontière de moins en moins évidente entre CPUs multicore et GPU

GPGPU

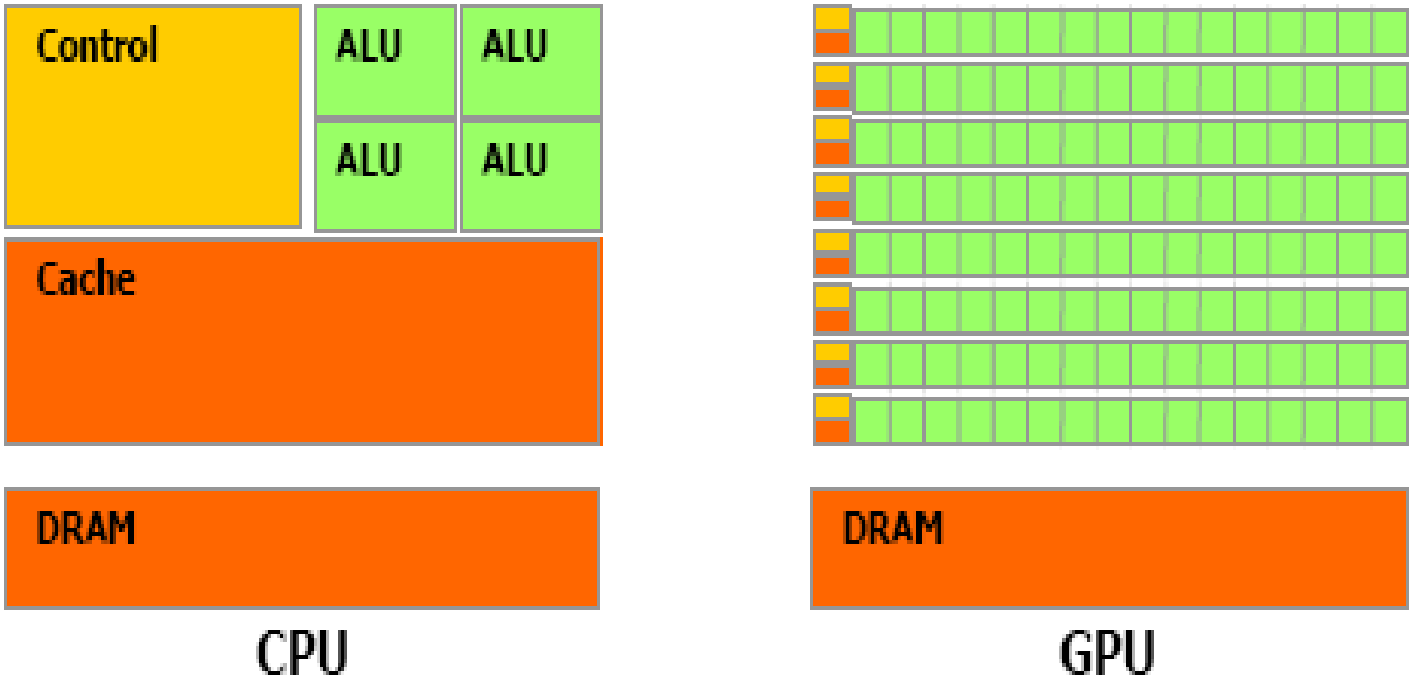
- **Modèle de programmation:**
  - Stream : données devant subir un même traitement
  - Kernel : traitement à appliquer
- **Implémentation possible en pur Langage graphique/GLSL**
  - Stream : texture
  - Kernel : fragment program sur un Quad couvrant l'ecran
  - Feed back : Render To Texture (FBO)
  - Enchaînement de passes pour faire progresser les calculs
- **Exemple: reconstruction 3D à partir d'images**
- **langages dédiés: CUDA (NVIDIA 2006), Brooke, Sh**
- **Uniformisation: OpenCL / DirectCompute**

CPU vs. GPU

- **CPU**
  - Mémoire cache rapide
  - Branchement efficace
  - Très haute performance
- **GPU**
  - Multiple ALUs
  - Mémoire rapide directement sur la carte
  - Très grand débit pour des taches parallèles
    - Exécute un programme pour chaque fragment/sommet

- **Les CPUs sont très bon pour paralléliser des taches**
- **Les GPUs sont très bon pour paralléliser le traitement de données**

CPU vs. GPU - Matériel



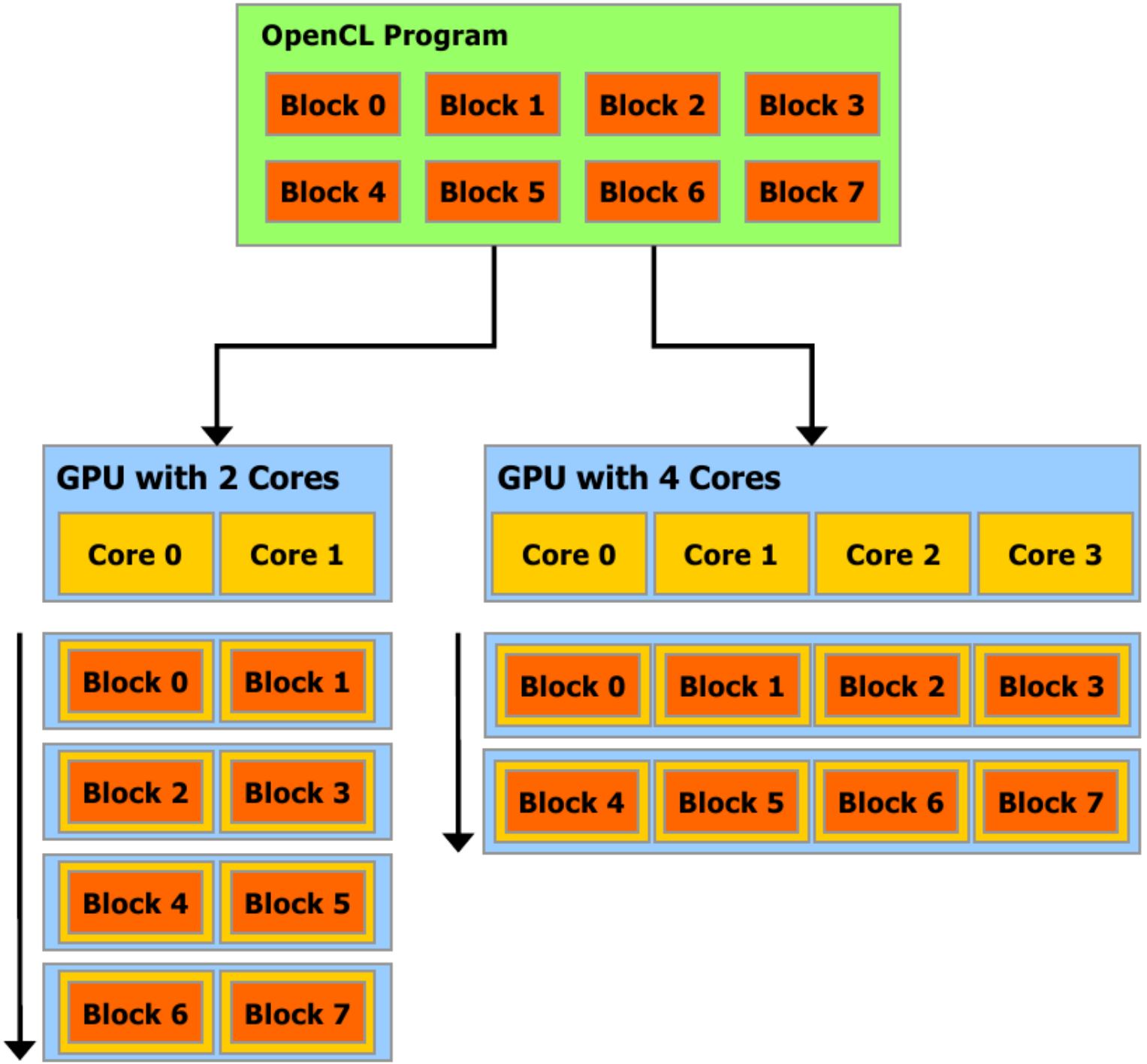
- More transistors devoted to data processing

Principe

- **Décomposition en petits calculs élémentaires**
  - Chacun est identifié :
    - La coordonnée texture pour GLSL
    - Un indice
  - *thread* (CUDA) / *work item* (OpenCL) / *fragment* – *vertex* (OpenGL)

- **Organisation hiérarchique**
  - Groupes : *thread block* (CUDA) / *work group* (OpenCL)
  - Tous les calculs élémentaire d'un groupe partage la même mémoire

- **Donnée régulière**
  - Tableaux 1D/2D/3D
  - Divisible en bloc


OpenCL Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 Cores
Core 0
Core 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 Cores
Core 0
Core 1
Core 2
Core 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

Addition de vecteurs

C classique

```
void
vectorAdd(const float * a,
          const float * b,
          float * c)
{
    for ( int nIndex = 0 ; nIndex < Size ; nIndex++)
    {
        c[nIndex] = a[nIndex] + b[nIndex] ;
    }
}
```

Addition de vecteurs

C for CUDA Kernel Code:

```
__global__ void
vectorAdd(const float * a,
          const float * b,
          float * c)
{
    // Vector element index
    int nIndex = blockIdx.x * blockDim.x + threadIdx.x;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL Kernel Code

```
__kernel void
vectorAdd(__global const float * a,
          __global const float * b,
          __global float * c)
{
    // Vector element index
    int nIndex = get_global_id(0);
    c[nIndex] = a[nIndex] + b[nIndex];
}
```