Le pipeline graphique

Le pipeline graphique

Modèle géométrique : objets, surfaces, sources de lumière…

Modèle d'illumination : calcul des interactions lumineuses

Caméra : point de vue et ouverture (frustum)

Fenêtre (viewport) : grille de pixel sur laquelle on plaque l'image

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

Projection (to Screen Space)

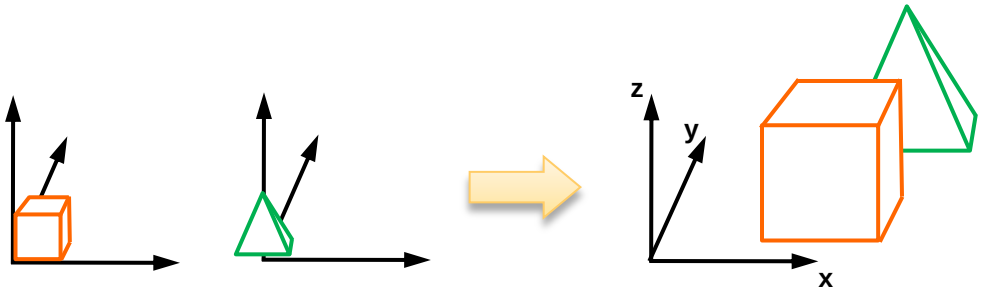
Scan Conversion (Rasterization)

Visibility / Display

Couleurs, intensités convenant à l'afficheur (ex : 24 bits, RVB)

Transformations objet

- Passage du système de **coordonnées local** de chaque objet 3D (object space) vers un **repère global** (world space)



Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

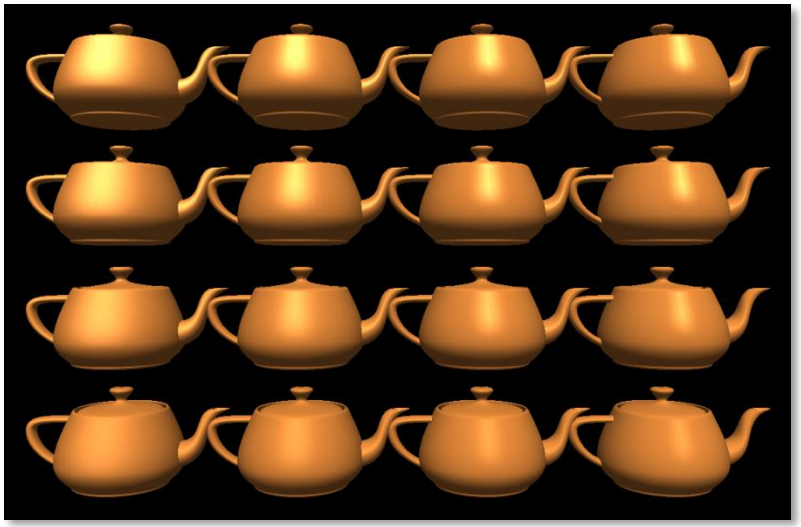
Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

Ilumination

- Les **primitives** sont éclairées selon leur matériau, le type de surface et les sources de lumière.
- Les **modèles d'illumination** sont **locaux** (pas d'ombres) car le calcul est effectué par primitive.



Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

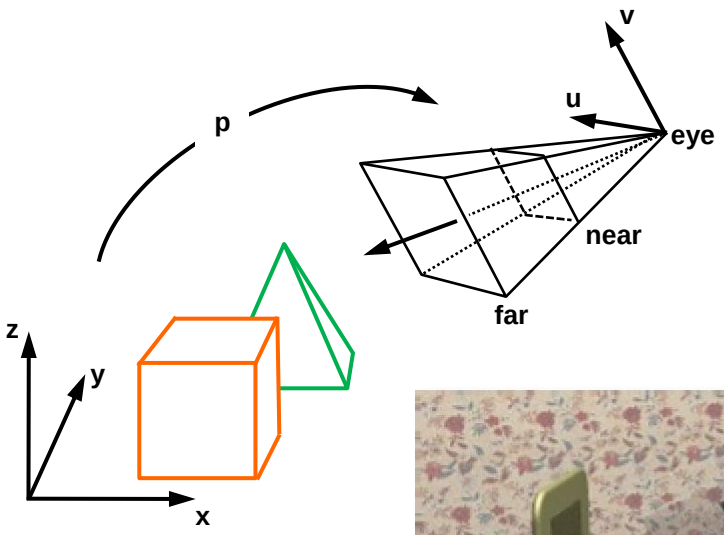
Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

Transformation caméra

- Passe des coordonnées du monde à celles du **point de vue** (repère caméra ou eye space).



Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

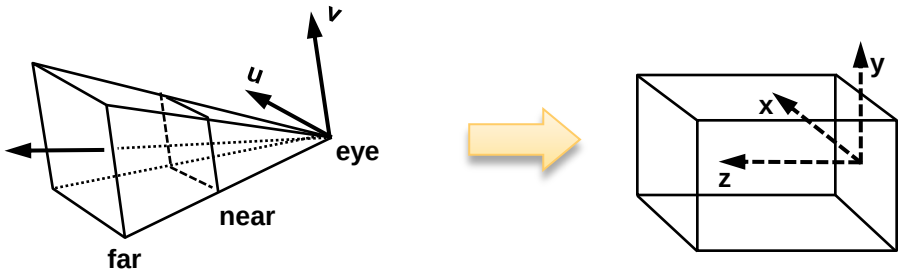
Projection (to Screen Space)

Scan Conversion (Rasterization)

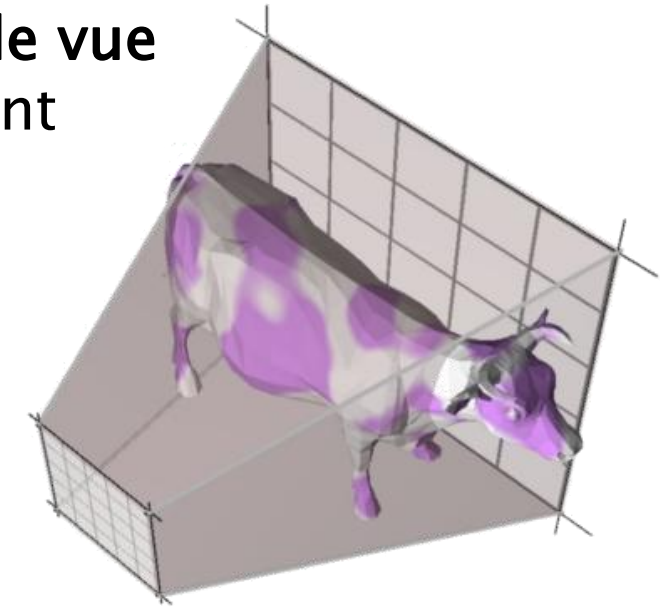
Visibility / Display

Clipping

- **Coordonnées normalisées :**



- Les portions en dehors du **volume de vue** (frustum) sont coupées.



Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

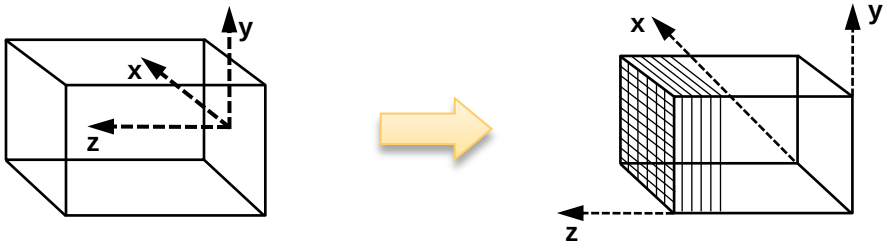
Projection (to Screen Space)

Scan Conversion (Rasterization)

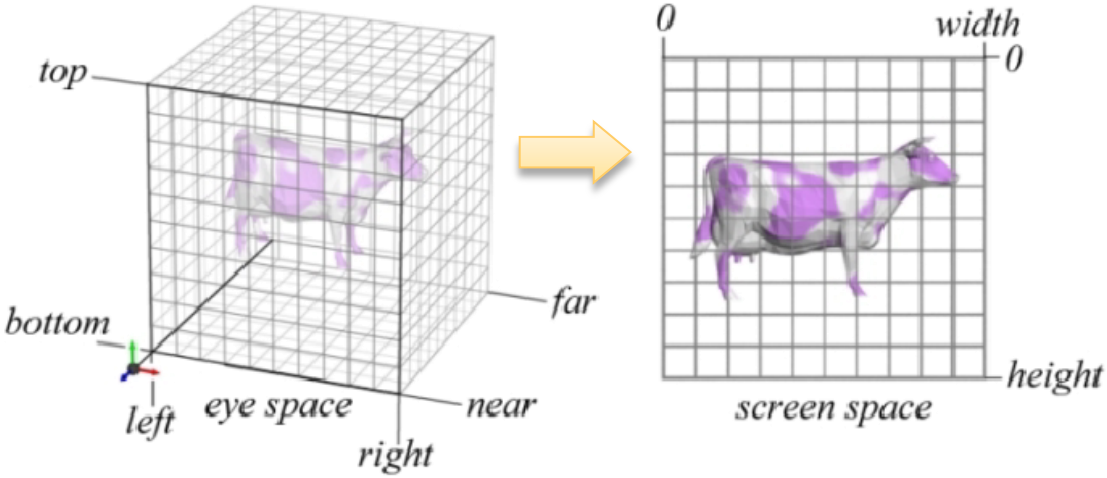
Visibility / Display

Projection

- Les primitives 3D sont **projetées** sur l'image 2D (screen space)



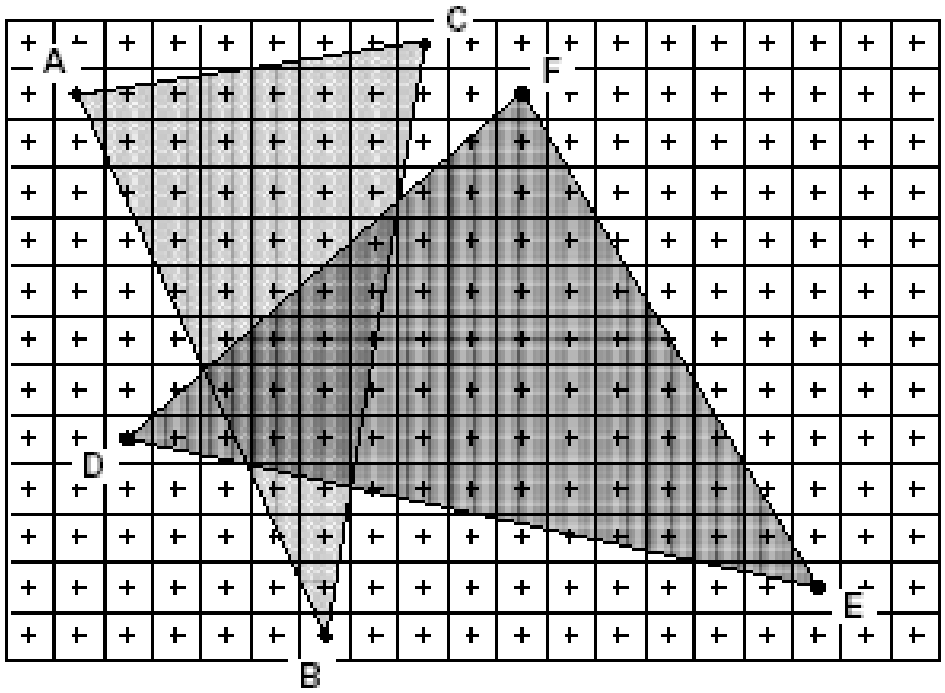
NDC **Screen Space**



- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- **Projection (to Screen Space)**
- Scan Conversion (Rasterization)
- Visibility / Display

Rastérisation

- Découpe la primitive 2D en **pixels**
- **Interpole** les valeurs connues aux sommets : couleur, profondeur,…



Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

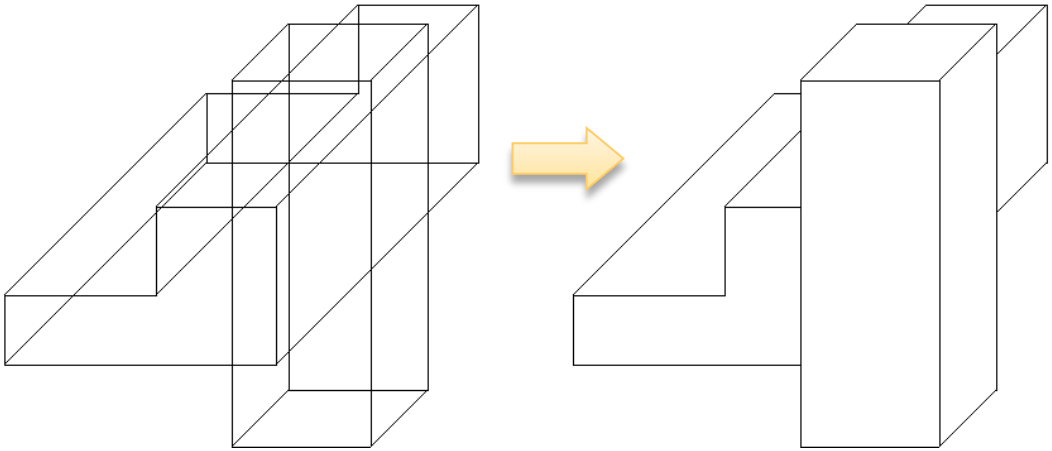
Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

Visibilité, affichage

- Calcul des **primitives visibles**
- Remplissage du **frame buffer** avec le bon format de couleur



Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

Clipping

Projection (to Screen Space)

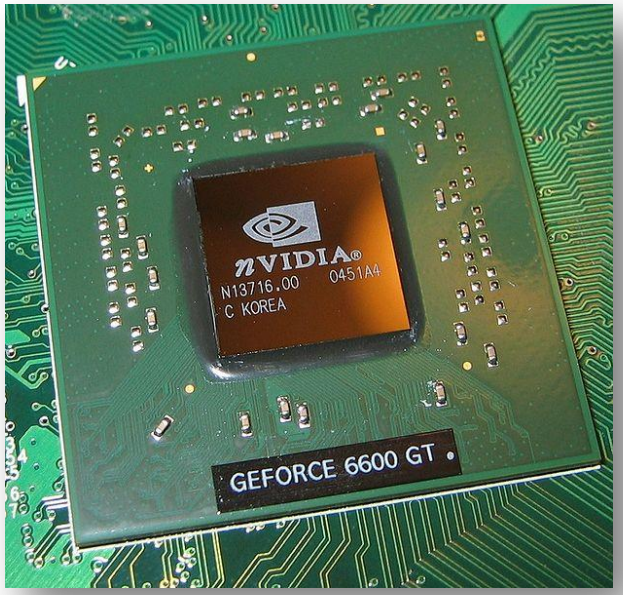
Scan Conversion (Rasterization)

Visibility / Display

Le GPU dans le pipeline

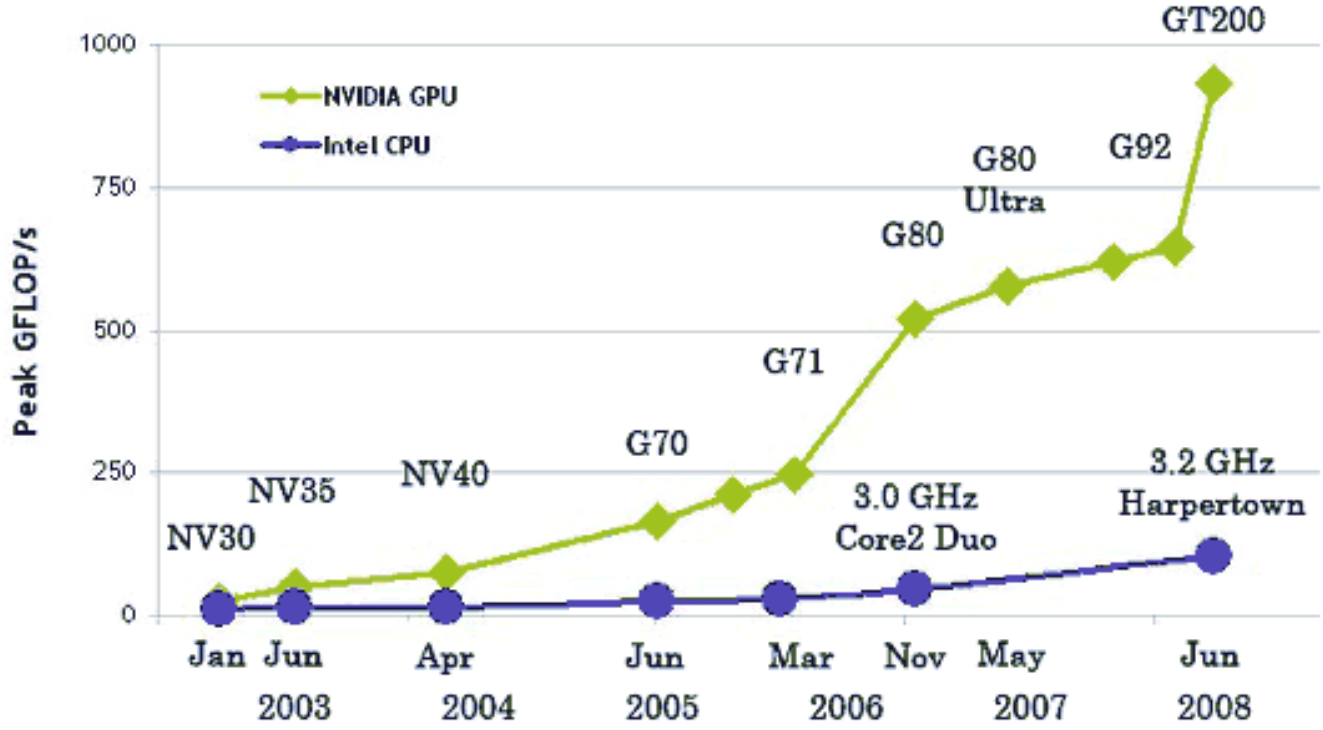
C’est quoi un GPU ?

- « Graphics Processing Unit »
- Processeur spécialisé pour le rendu 3D
- Spécificités :
  - Architecture hautement parallèle
  - Accès mémoire rapide
  - Large bande passante



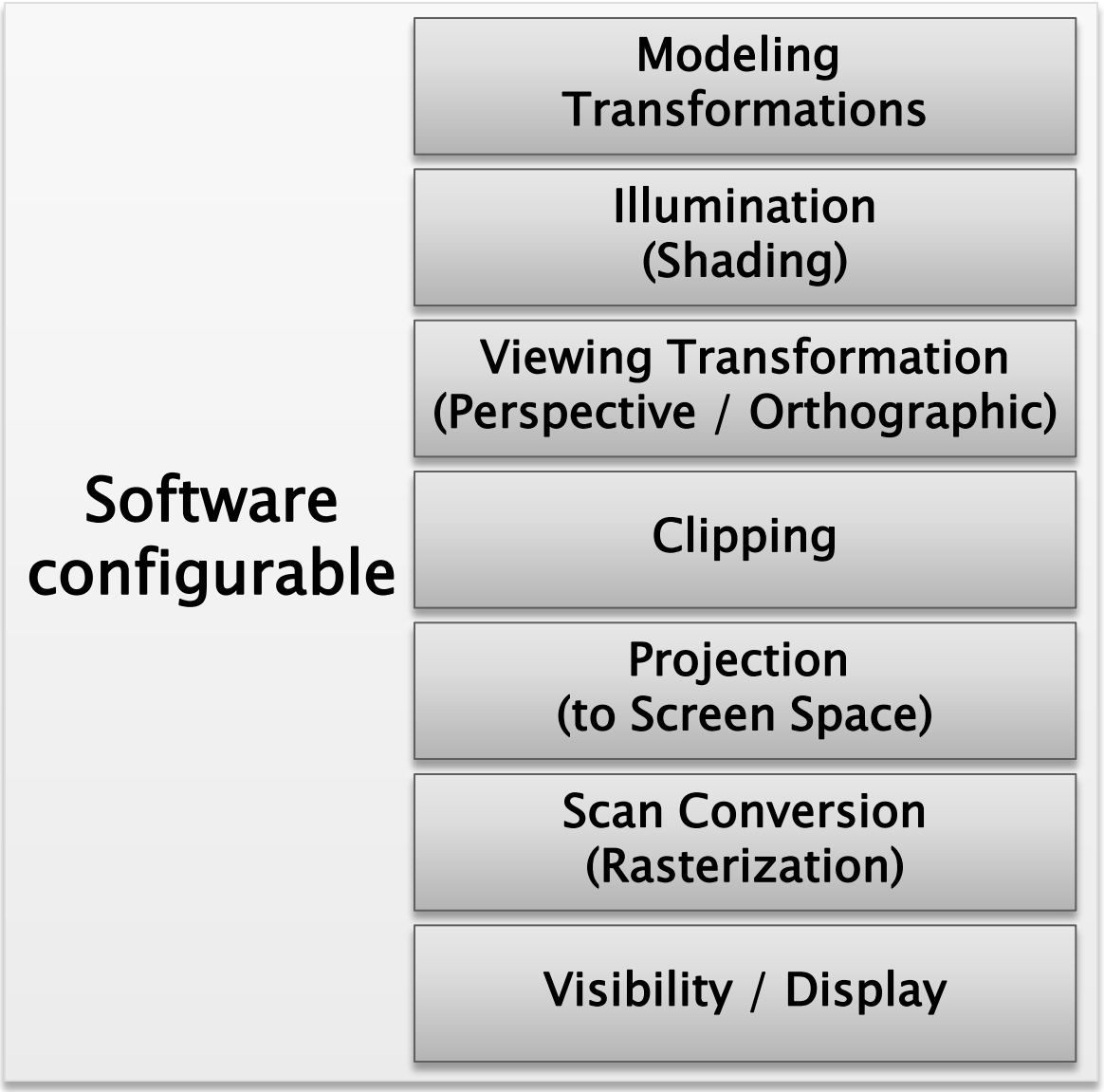
C'est quoi un GPU ?

- Un monstre de calcul parallèle :
  - GPGPU : « General-Purpose computation on GPU »



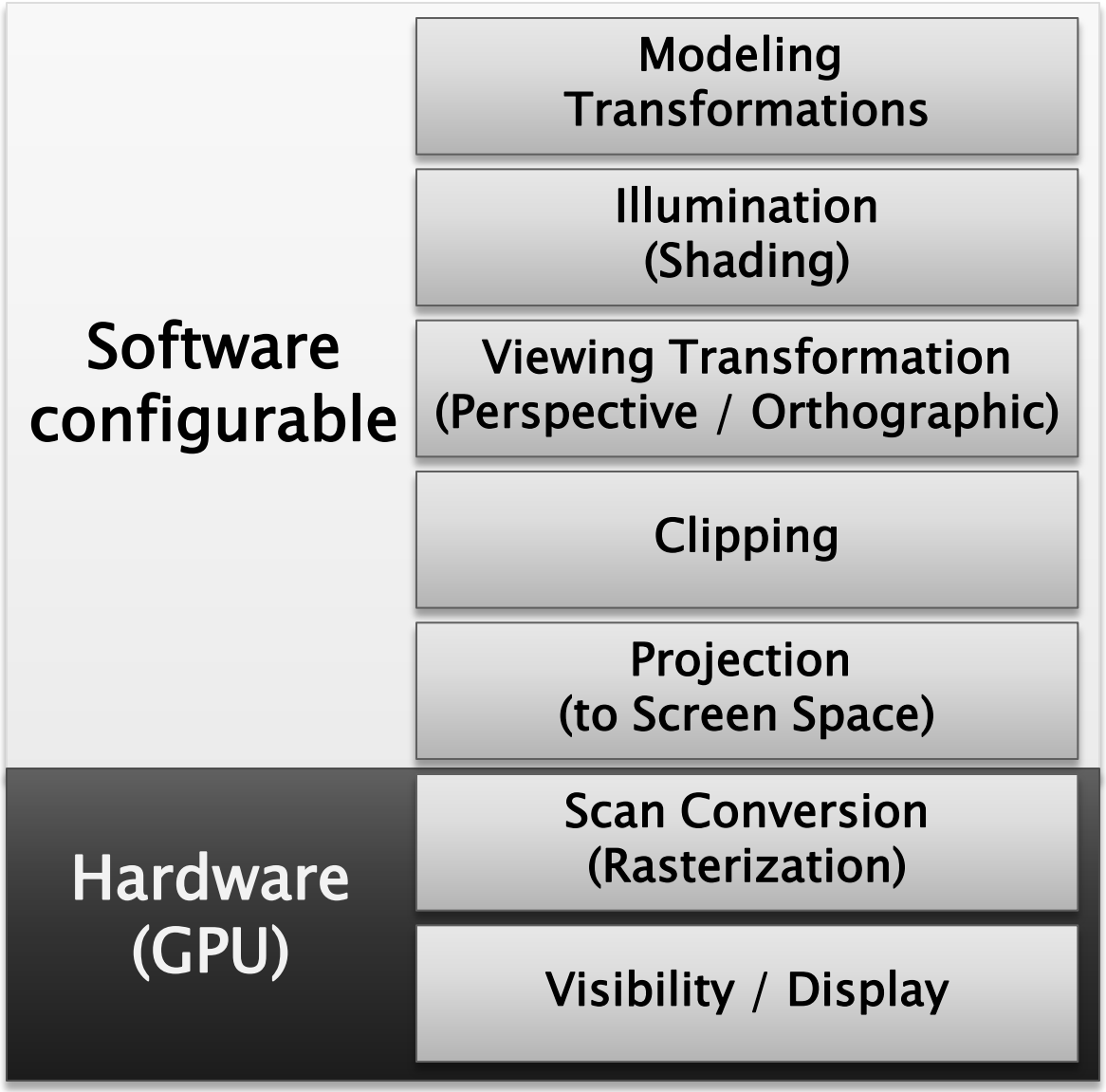
Le pipeline graphique

Sans carte graphique 3D
(1970s)



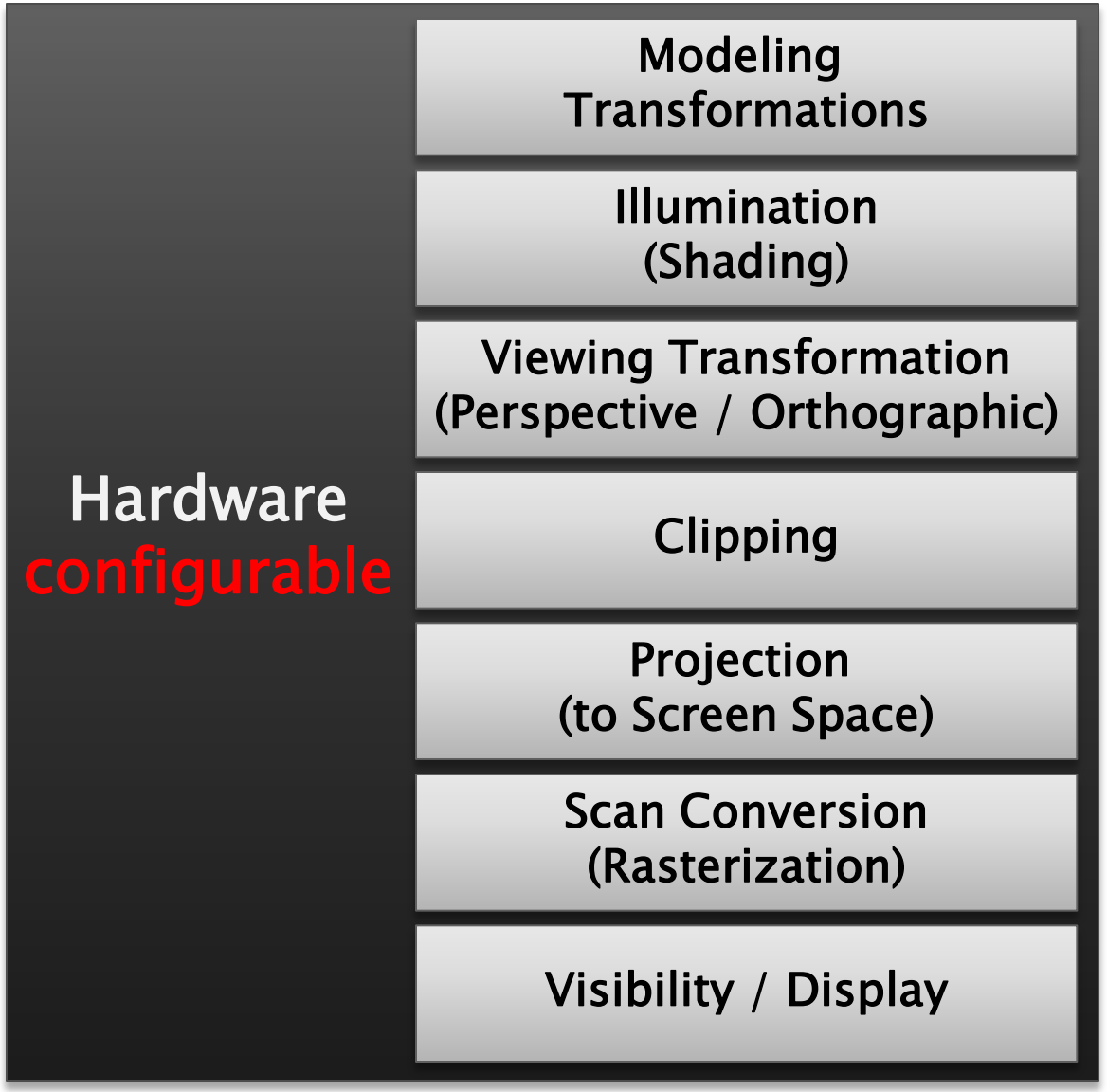
Le pipeline graphique

Cartes graphiques première génération (1980s)



Le pipeline graphique

Cartes graphiques deuxième génération
(1990s)



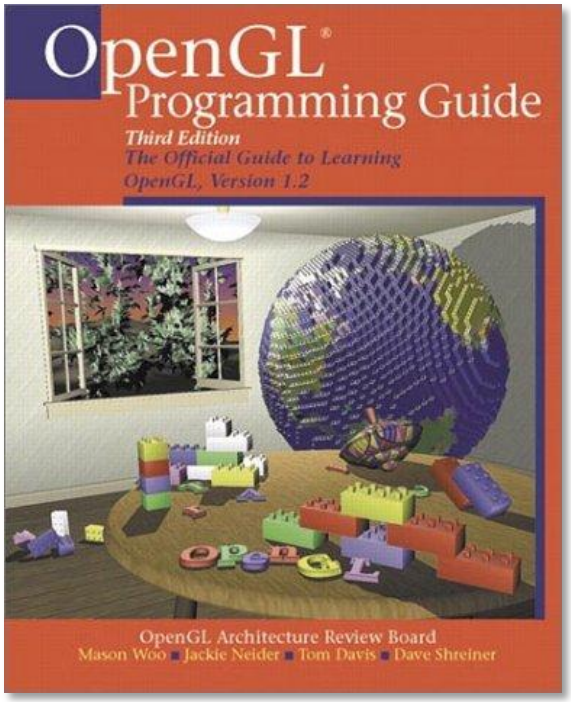
Configurable ?

- **API** (Application Programming Interface) pour l'architecture graphique
- 2 API prépondérante en graphique :
  - Direct3D (Microsoft)
  - **OpenGL** (Khronos Group)

OpenGL

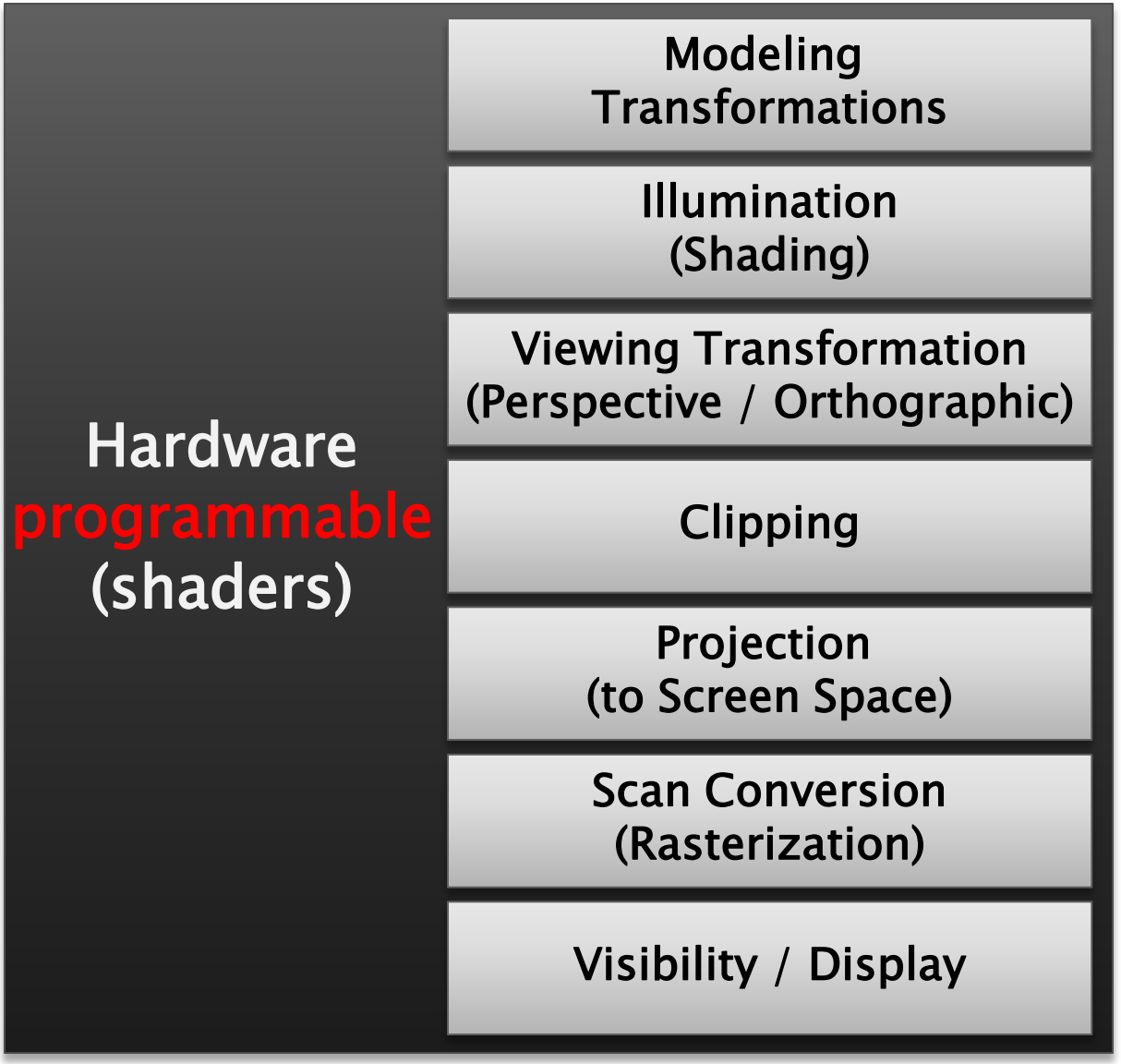
- API indépendant de l'architecture
- Développée en 1989 (GL) par Silicon Graphics et porté sur d'autre plateformes en 1993
- La référence : le « red book »

D. SHREINER, M. WOO, J. NEIDER, T. DAVIS
OpenGL Programming Guide



Le pipeline graphique

Cartes graphiques troisième génération (2000s)

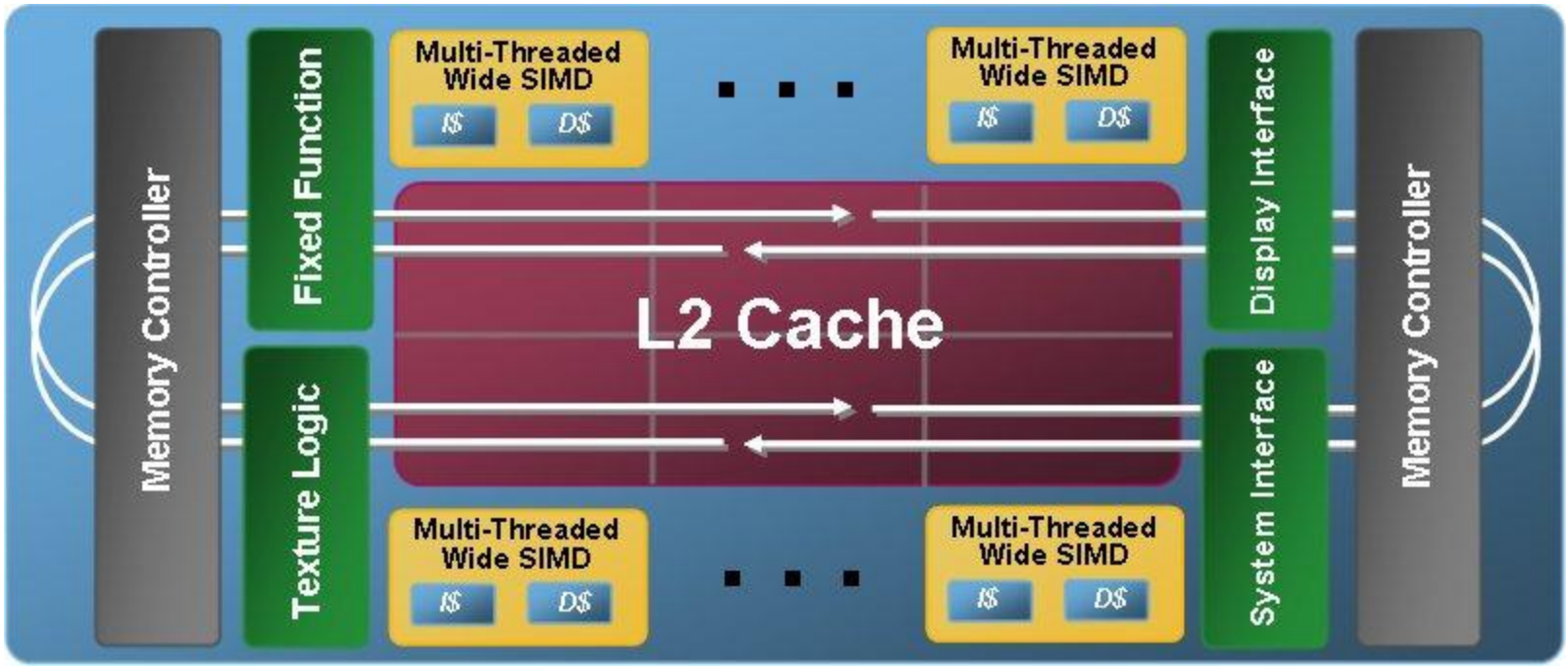


Programmable ?

- **Shaders**
  - suite d'instructions exécutable par le GPU à différentes étapes du pipeline
  - Langage différent (pseudo-C) en fonction des API :
    - NVIDIA ⇨ Cg (2002)
    - Direct3D ⇨ HLSL (2003)
    - OpenGL ⇨ **GLSL** (2004)
- Pour le GPGPU :
  - CUDA (NVIDIA)
  - ATI Stream
  - OpenCL (Khronos Group) ⇨ fin 2009

Le pipeline « graphique » ?

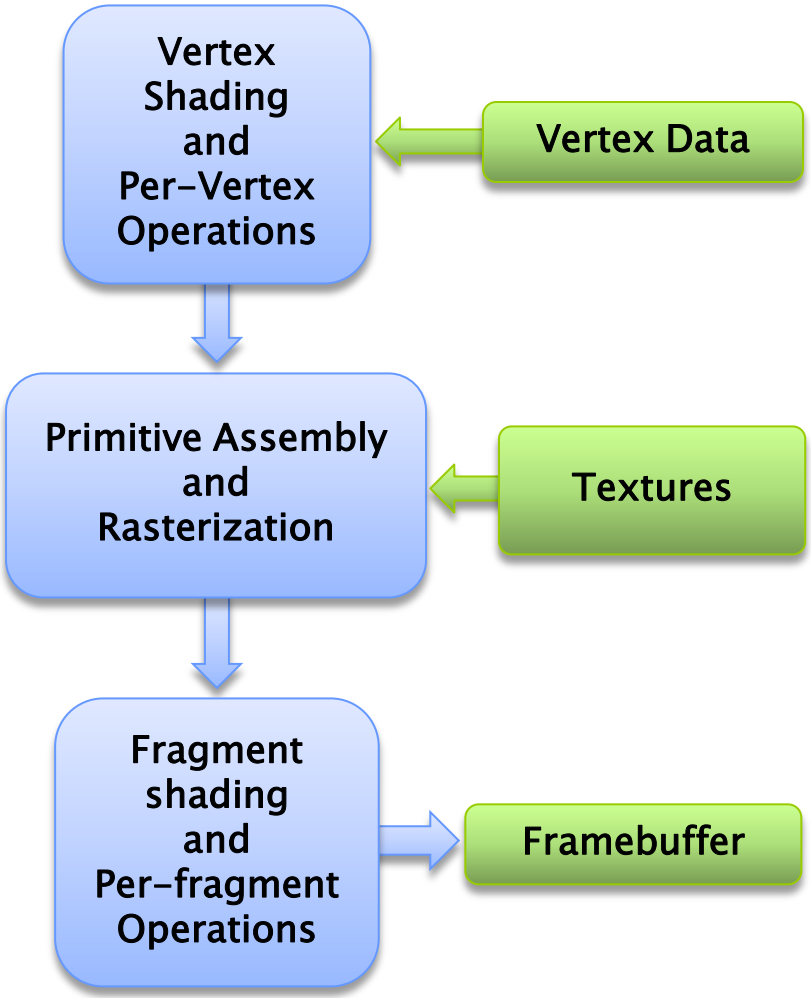
- Prochaine génération ?
  - Larrabee (Intel)



et les shaders GLSL

OpenGL

- Gère
  - Caméra virtuelle (pyramide de vision, projection)
  - Rastérisation
  - Élimination des parties cachées
  - Application de textures
  - Illumination
- Ne gère pas
  - Calculs d'ombres
  - Réflexions
  - Illumination global
  - La physique des scènes (collisions, mouvements…)

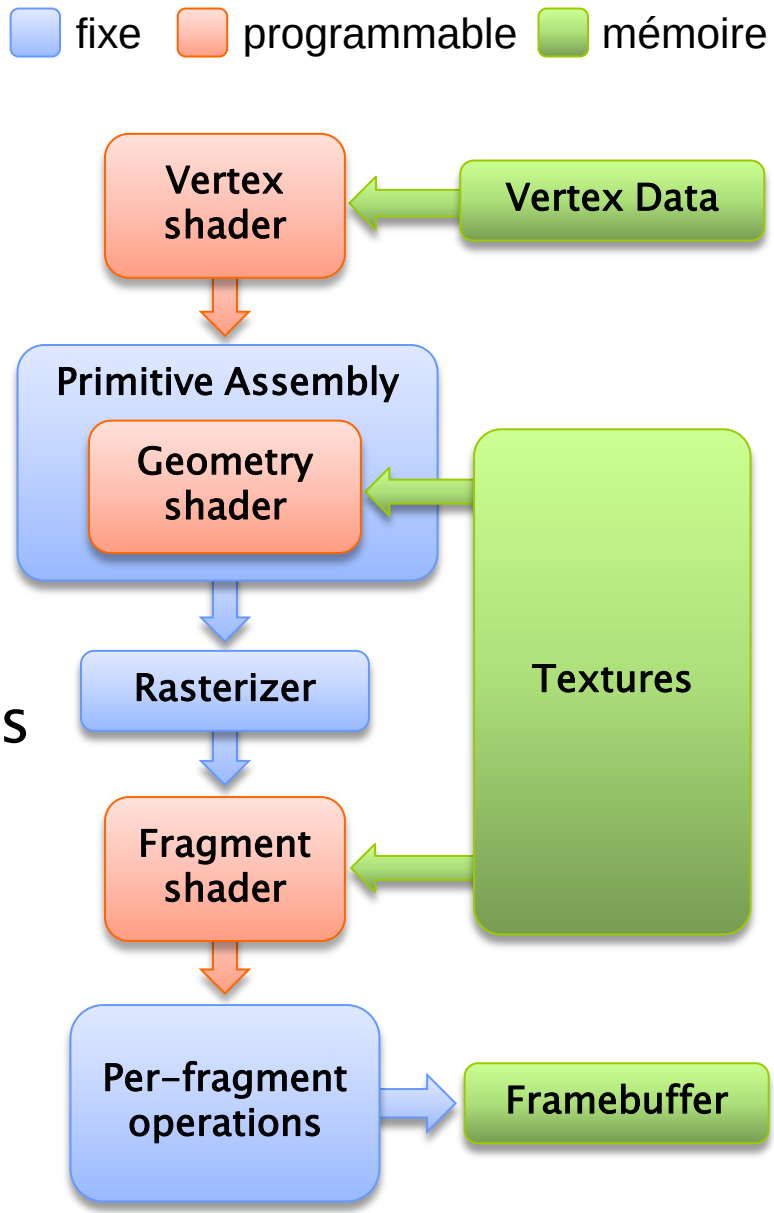


Primitives OpenGL

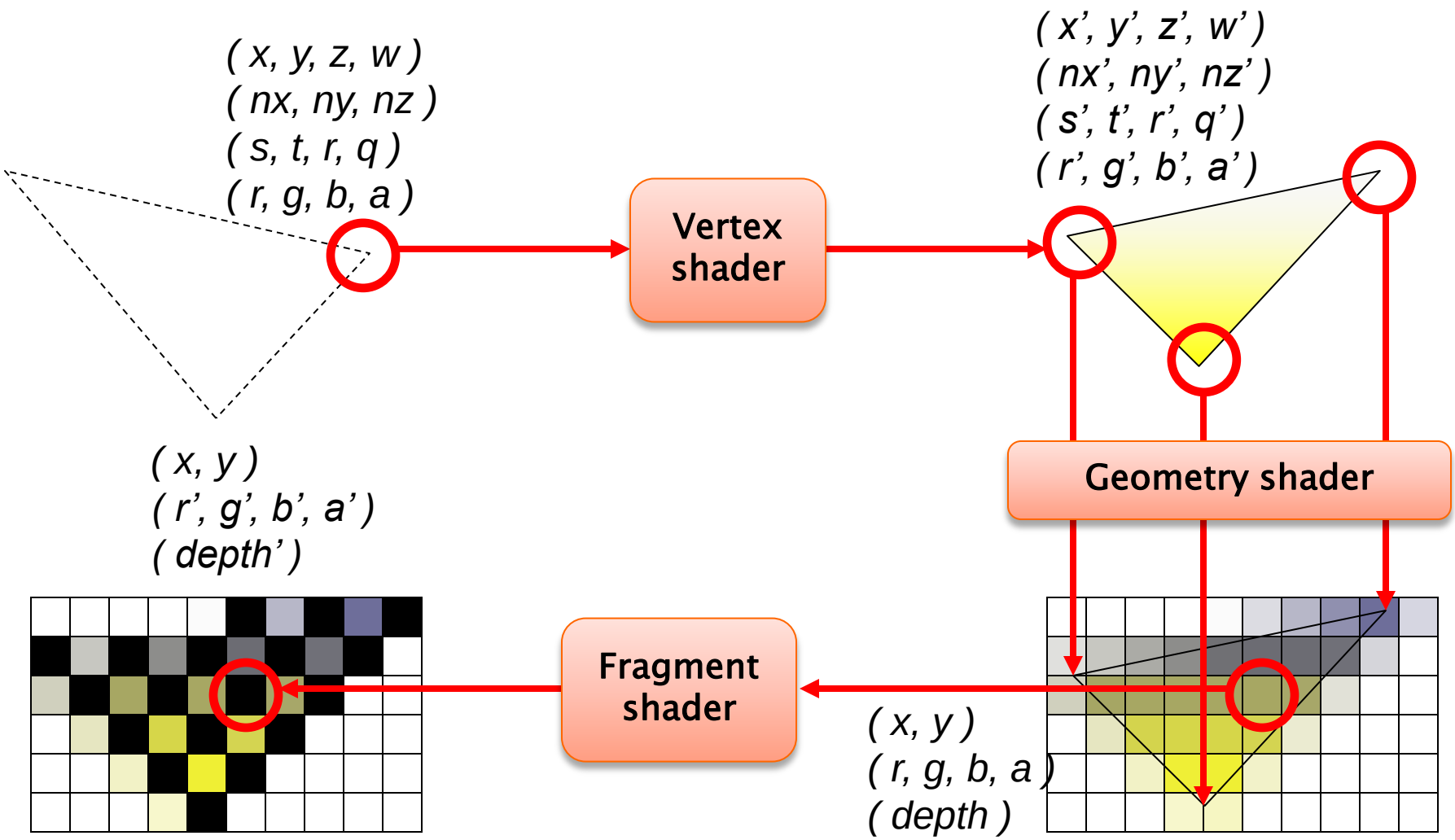
- 3 familles de primitives
  - **Sommets** (vertices) : vecteur de flottants
  - **Lignes** : segments
  - **Polygones** : polygones convexes simples
- Des attributs
  - **Position** (x,y,z,w) en coordonnées homogènes
  - **Normale** (nx, ny, nz)
  - **Couleur** (r,g,b,a)
  - **Coordonnée de texture** (s,t,r,q)
  - entre autres…

Shaders

- 3 types de shaders
  1. **Vertex** shader
  2. **Geometry** shader
  3. **Pixel** shader
- Action **locale**
  1. un sommet
  2. une primitive et ses voisines
  3. un pixel

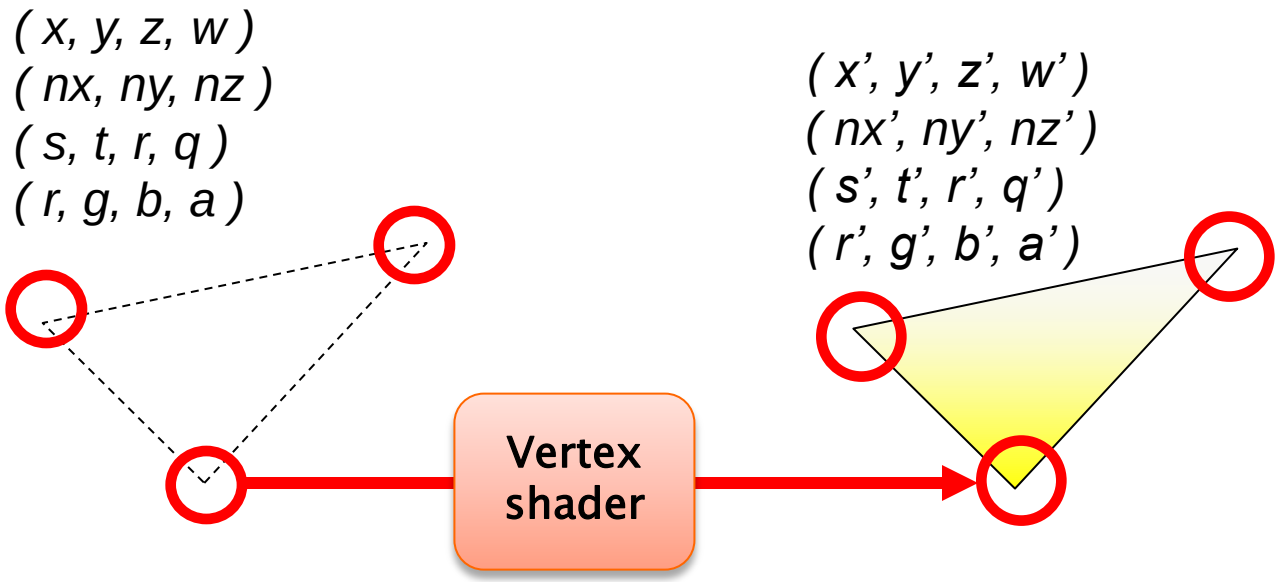


Shaders


(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)

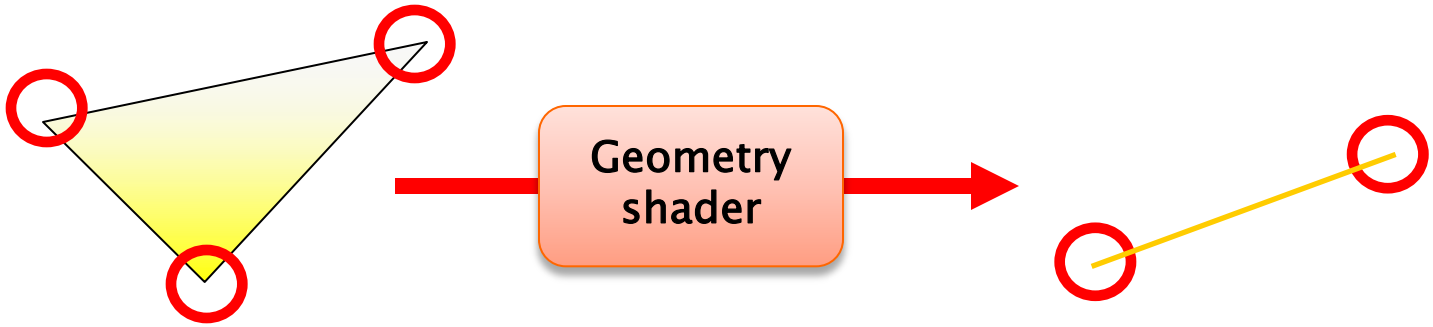
Vertex shader

- Ce qu'on peut faire
  - des transformations de position
  - des calculs d'illumination, de couleurs par sommet
  - des calculs de coordonnées de textures



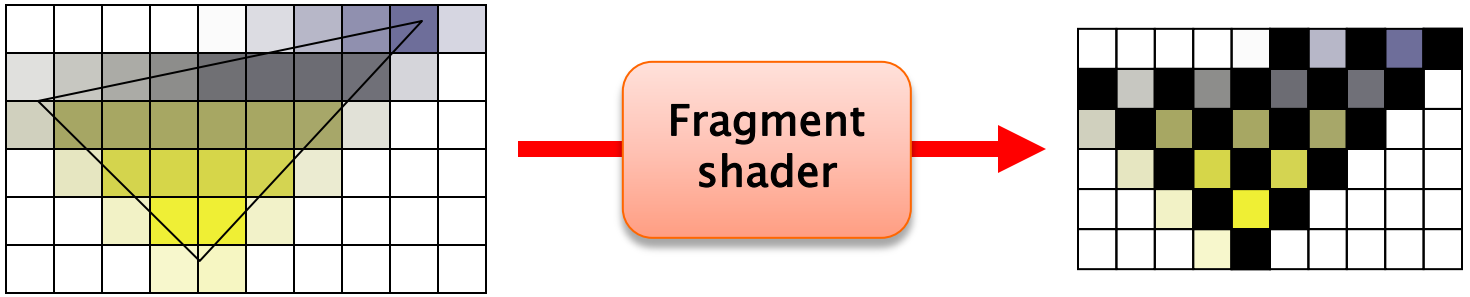
Geometry shader

- Ce qu'on peut faire
  - ajouter/Supprimer des sommets
  - modifier les primitives
  - récupérer directement la géométrie sans « tramage » (avant la rasterisation)

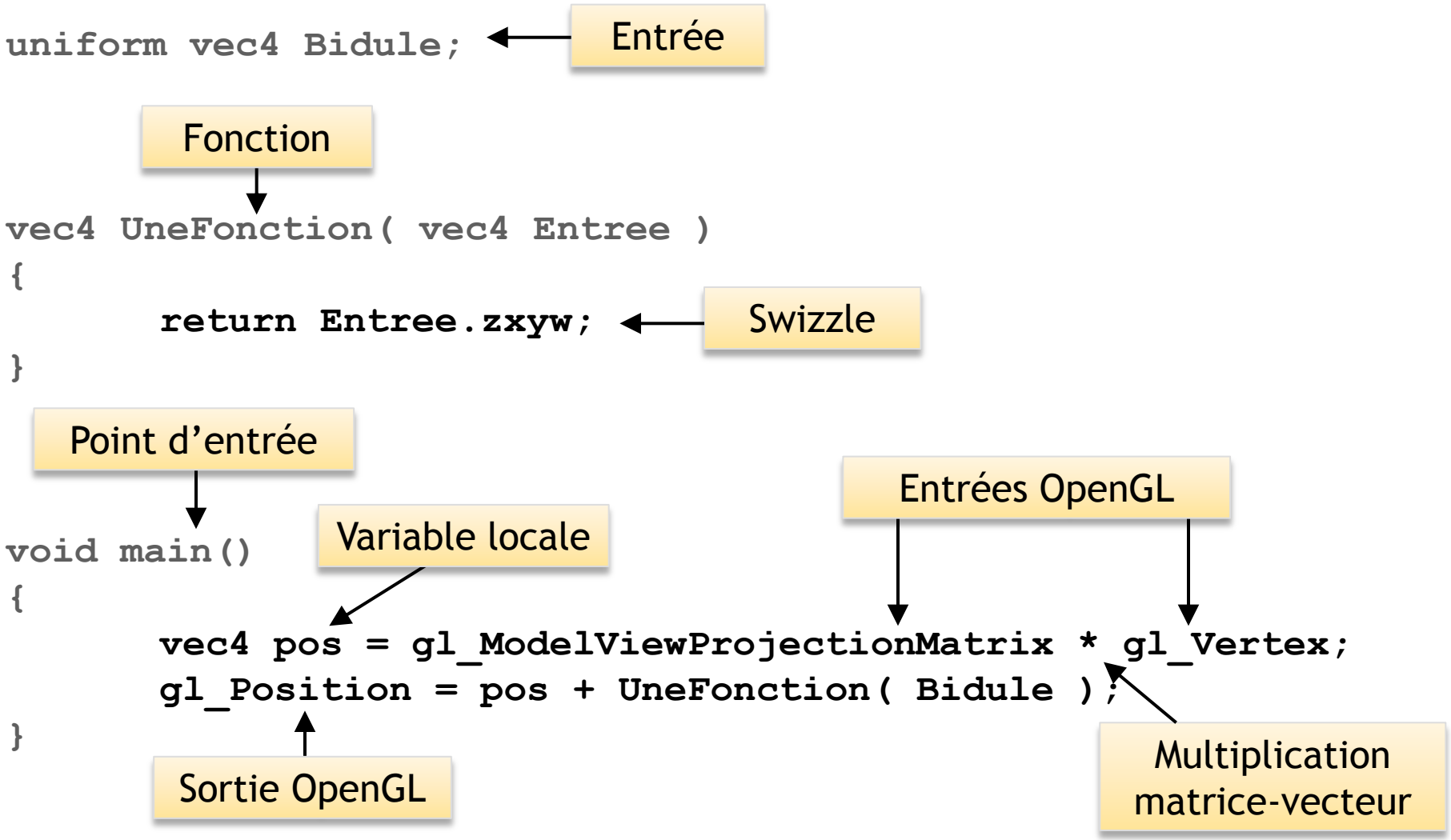


Fragment shader

- Ce qu'on peut faire
  - la même chose qu'aux sommets, mais par pixel
  - utiliser le contenu de textures dans des calculs
  - changer la profondeur des pixels



Premier Vertex Shader



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}`
- Structures
  - `struct my_struct { int index; float value};`
- Tableaux
  - `int array[5];`

Entrées

- *Built-in* : tous les états passés par OpenGL
  - Position `(glVertex3fv,…)`
  - Couleurs `(glColor4f,…)`
  - Directions des lumières `(glLighiv, …)`
  - Textures flottantes ou entières `(glTexCoord[1|2|3][i|f],…)`
  - Matrices `(glMatrixMode,…)`
  - Matériaux `(glMateriali,…)`
- **`uniform`** : passés par le programme OpenGL
  - Ne varie pas entre glBegin/glEnd (matrices, textures, lumières,…)
- **`const`** : constant au sein des shaders

Entrées / Sorties

- Communication entre les étapes du pipeline :
  - `in` : variable définie par sommet depuis OpenGL ou en entrée d'un Fragment Shader
  - `out` : variable définie comme sortie d'un Vertex ou d'un Fragment Shader
- Trois qualificateurs supplémentaires :
  - `smooth` : interpolation tenant compte de la perspective
  - `flat` : pas d'interpolation
  - `noperspective` : interpolation linéaire

Sorties *Built-In*

- 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
- Fragment Shader :
  - `gl_FragDepth` : profondeur du pixel

Swizzle

- Convention de notations (`vec4`)
  - position : **`x, y, z, w`**
  - couleur : **`r, g, b, a`**
  - texture : **`s, t, p, q`**
- Permutation des variables

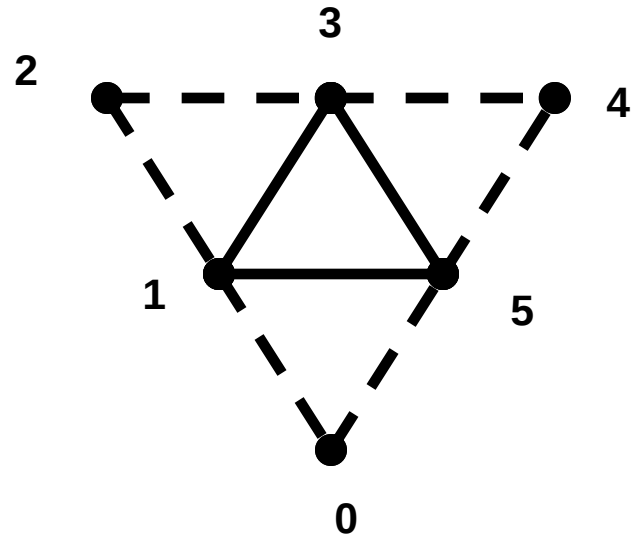
```
vec4 res;
res.xyzw (par défaut)
res.xxxx, res.xyxy, res.xyzx, res.wzyx, res.xyww, etc…
```

- Changement de dimensions

```
vec4 start; vec3 triple; vec2 couple; float alone;
triple = start.xyz;
couple = start.xy;
alone = start.x;
```

Geometry shader

- Création de primitives :
  - `EmitVertex()`
  - `EndPrimitive()`
- Types des primitives
  - Points, lignes, triangles
  - Lignes, triangles avec adjacences



Communication CPU ↔ GPU
Envoi d'une valeur

```
glUseProgram ( shaderProgram );
```

Utilisation d'un programme

```
glGetUniformLocation( shaderProgram, _name );
glUniform{1,2,3,4}f[v](…);
glUniformMatrix{2,3,4}fv(…);
```

Réglage d'un uniform

```
glGetAttribLocation( shaderProgram, _name );
glVertexAttrib{1,2,3,4}f[v](…);
```

Réglage d'un attribute

```
glUseProgramObject(0);
```

Fin de programme

Communication CPU ↔ GPU
Buffer Objects

- Mise à jour
  - **Static** : une et une seule fois pour l'application
  - **Dynamic** : modification multiples pour plusieurs rendus
  - **Stream** : une modification, un rendu
- Accès
  - **Read**: du GPU vers le CPU
  - **Draw**: du CPU vers le GPU
  - **Copy**: depuis et vers le GPU
- Création
  - `glGenBuffers(1, &_name);`
  - `glBindBuffer(NATURE, _name);`
  - `glBufferData(NATURE, size, ptr, TYPE);`

Communication CPU ↔ GPU
Buffer Objects

- **Vertex et Index Buffer Objects (VBO)**
  - Envoyer la géométrie via des tableaux
    - Position, Normal, Couleurs, Coordonnées de textures (`GL_ARRAY_BUFFER`)
    - Topologie (`GL_ELEMENT_ARRAY_BUFFER`)
  - Type : `[STATIC|DYNAMIC|STREAM]_DRAW`

Communication CPU ↔ GPU
Buffer Objects

- **FrameBuffer Objects (FBO)**
  - Object : `GL_FRAMEBUFFER`
  - Créer: `GenFramebuffers(…)` + `BindFramebuffer(…)`
  - Attacher
    - un RenderBuffer : `FramebufferRenderbuffer()`
      - n textures : `GL_COLOR_ATTACHMENTn`
      - z-buffer: `GL_DEPTH_ATTACHMENT`
      - stencil-buffer : `GL_STENCIL_ATTACHMENT`
    - une image : `FramebufferTexture[1|2|3]D(…)`

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);

Communications CPU ↔ GPU
du CPU vers le GPU

- Pixel Buffer Object UNPACK (PBO)
  - Envoyer une texture au GPU
    `GL_PIXEL_UNPACK_BUFFER_ARB`
    - Activer le buffer
    - Remplir le buffer
    - Instancier la texture
  - Type : `[STATIC|DYNAMIC|STREAM]_DRAW`

Communications CPU ↔ GPU
du GPU vers le CPU

- Pixel Buffer Object PACK (PBO)
  - Récupérer une texture du GPU
    `GL_PIXEL_PACK_BUFFER_EXT`
    - Activer le buffer
    - Lire la sortie `glReadPixels`
    - Mapping
  - Type : `[STATIC|DYNAMIC|STREAM]_READ`

Communications CPU ↔ GPU
du GPU vers le CPU/GPU

- TRANSFORM FEEDBACK BUFFER OBJECT
  - Buffer d'éléments spécial `GL_ARRAY_BUFFER_ARB`
  - Spécifier
    - Les éléments à récupérer à la fin du VS ou du GS
    - Comment ? Un/Plusieurs buffers
  - Lors du rendu : `GL_TRANSFORM_FEEDBACK_BUFFER_NV`
    - Activer ou Désactiver la Rasterisation : `glEnable(GL_RASTERIZER_DISCARD)`
  - TYPE : `[DYNAMIC|STREAM]_[COPY|READ]`

Communications CPU ↔ GPU
du CPU/GPU vers le GPU

- Texture Buffer Objects (TBO)
  - Envoyer une texture accessible comme un tableau
    - Utilisation : Réutiliser la sortie du pixel shader comme un buffer de géometrie
  - GL_TEXTURE_BUFFER_EXT
  - Type : [STATIC|DYNAMIC|STREAM]_[DRAW|COPY]
  - `glTexBufferEXT:` association avec la texture

Communications CPU ↔ GPU
du CPU vers le GPU

- Bindable Uniform Buffer Objects (BUBO)
  - Envoyer des uniforms accessibles par **TOUS** les shaders
    - Minimiser l'envoi de constantes (et aussi la place mémoire)
  - GL_UNIFORM_BUFFER_EXT
  - Type : STATIC_DRAW
  - `glUniformBufferExt:` association mémoire buffer

Rendu off-screen

- FrameBuffer Objects (FBO) :Type de Textures
  - Flottantes rectangles : `gl_FragData[n]`
  - Entières rectangles : `varying out [i|u]vec4 data`
  - Tableau de textures 1D, 2D (layers) :
    - `glFramebufferTextureArrayExt(),TEXTURE_[1D|2D]_ARRAY_EXT`
    - `gl_Layer`

Rendu off-screen

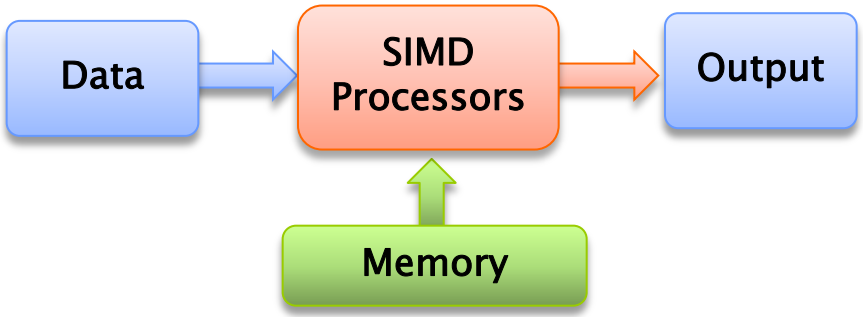
- FrameBuffer Objects (FBO)
  - Rendu
    - Activer le FrameBuffer
    - Activer le rendu dans les différentes textures
    - Dessiner
    - Terminer

Conseils

- Développez petit à petit et testez souvent !
  - Débuggage très difficile
- 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-développez.

GPGPU

- *General-Purpose Computation Using Graphics Hardware*
- Un GPU = un processeur SIMD (*Single Instruction Multiple Data*)
- 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

- 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

Références/Liens utiles

- Le red book : http://www.opengl-redbook.com/
- La spec GLSL : http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.30.08.pdf
- Cg : http://developer.nvidia.com/page/cg_main.html
- Cuda : http://www.nvidia.com/cuda
- OpenCL : http://www.kronos.org/opencl/

- Librairie pour les extensions
  - GLEW : http://glew.sourceforge.net/

- Un éditeur spécial shader (malheureusement pas à jour, mais bien pour débuter)
  - http://www.typhoonlabs.com/

- Erreurs OpenGL/GLSL : un débogueur simple, efficace, **super utile,** vite pris en main.
  - glslDevil : http://www.vis.uni-stuttgart.de/glsldevil/

- Des tas d'exemples (à tester, éplucher, torturer) :
  - http://developer.nvidia.com/object/sdk_home.html

- La référence GPGPU avec code, forums, tutoriaux : http://www.gpgpu.org/