

Pipeline graphique + GLSL

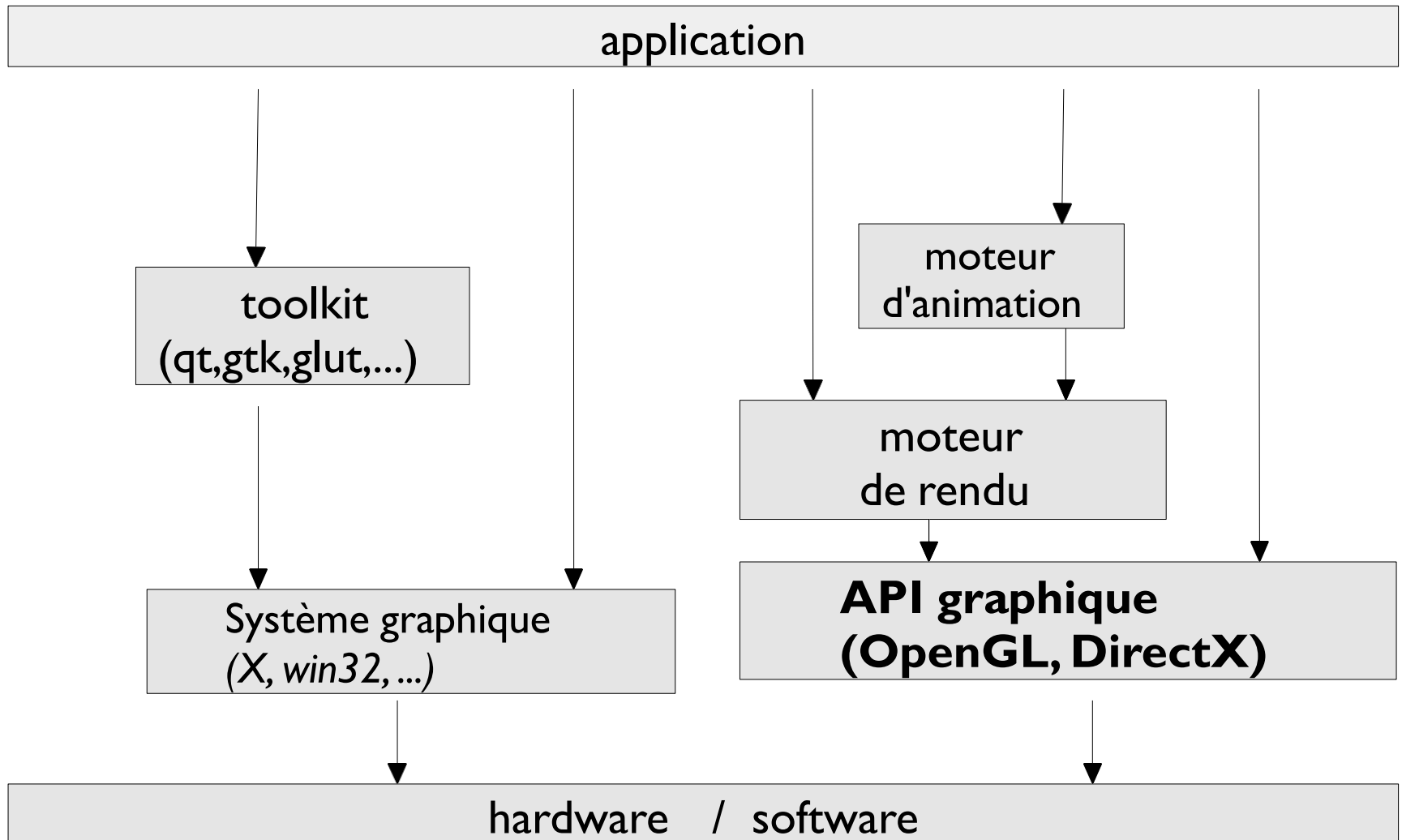
HAI7191 – Cours 2

Introduction au pipeline OpenGL

App 3D OpenGL

- Manipule un ensemble de polygones, structuré par l'application
 - De la simple liste
 - Au graphe de scène complet, avec description sémantique
- Textures : images couleurs plaquées sur les polygones
- Rendu temps réel de la scène sous forme d'images couleurs affichées à l'écran
 - Boucle de rendu
- Interaction : événements utilisateurs (e.g., clavier, souris, *touch screen*)
 - Callbacks

Architecture logicielle



Boucle de rendu

- Rendu temps-réel : en général effectué par le GPU
 - Effectue des appels à une API graphique
 - API dédiés OpenGL, DirectX, Metal, Optix, etc
- Données
 - Maillage polygonal échantillonnant la scène
 - Propriétés de surface : normal, coordonnées de textures,
 - Textures
 - Matériaux
 - Sources de lumières
 - Paramètres caméra

GPU : Données en entrée

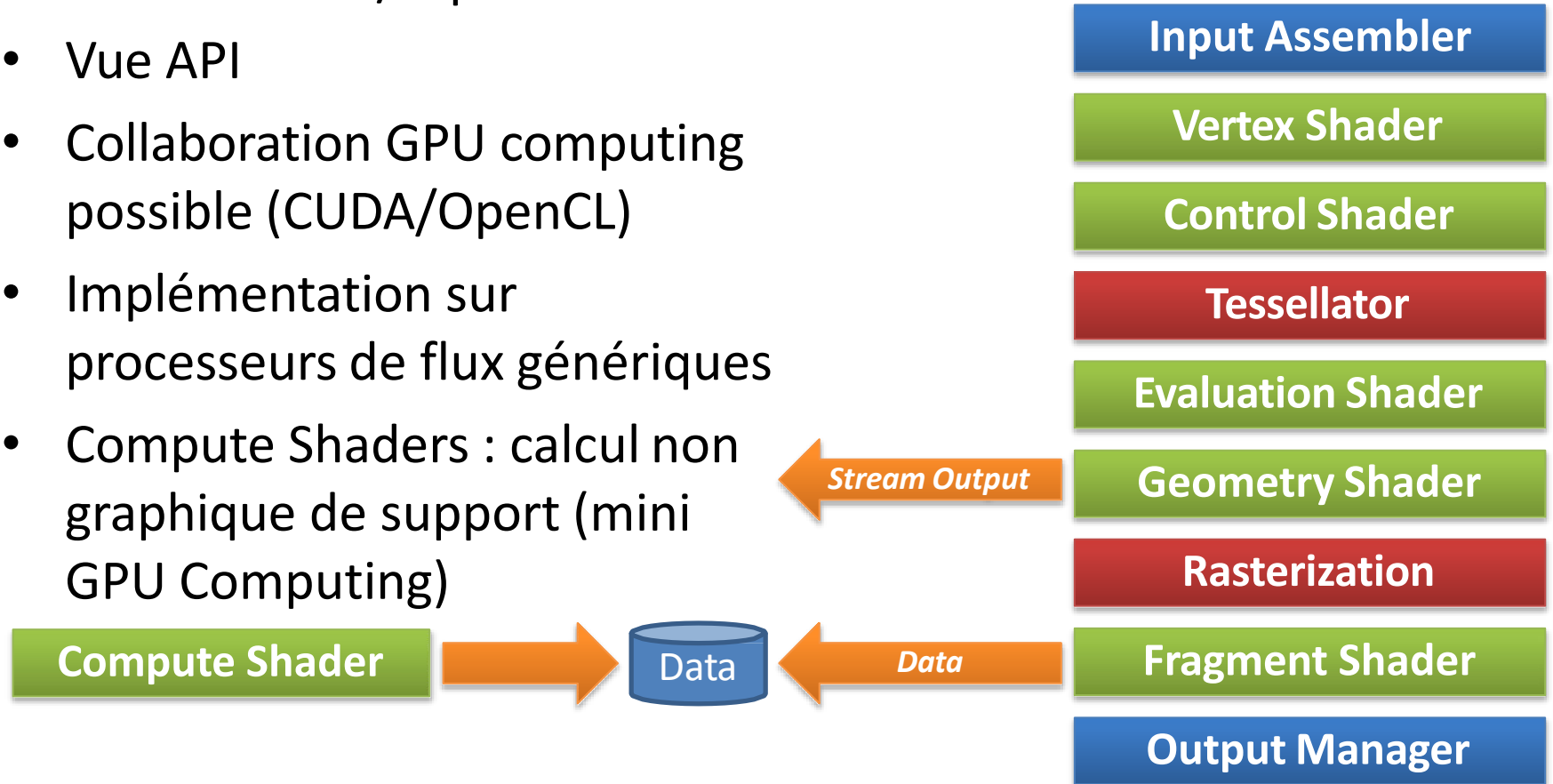
Maillage polygonal : approximation de la surface d'un objet à l'aide d'un ensemble de polygones

- **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)

Pipeline Graphique Moderne

- Direct3D 11+ / OpenGL 4+
- Vue API
- Collaboration GPU computing possible (CUDA/OpenCL)
- Implémentation sur processeurs de flux génériques
- Compute Shaders : calcul non graphique de support (mini GPU Computing)



Programmable



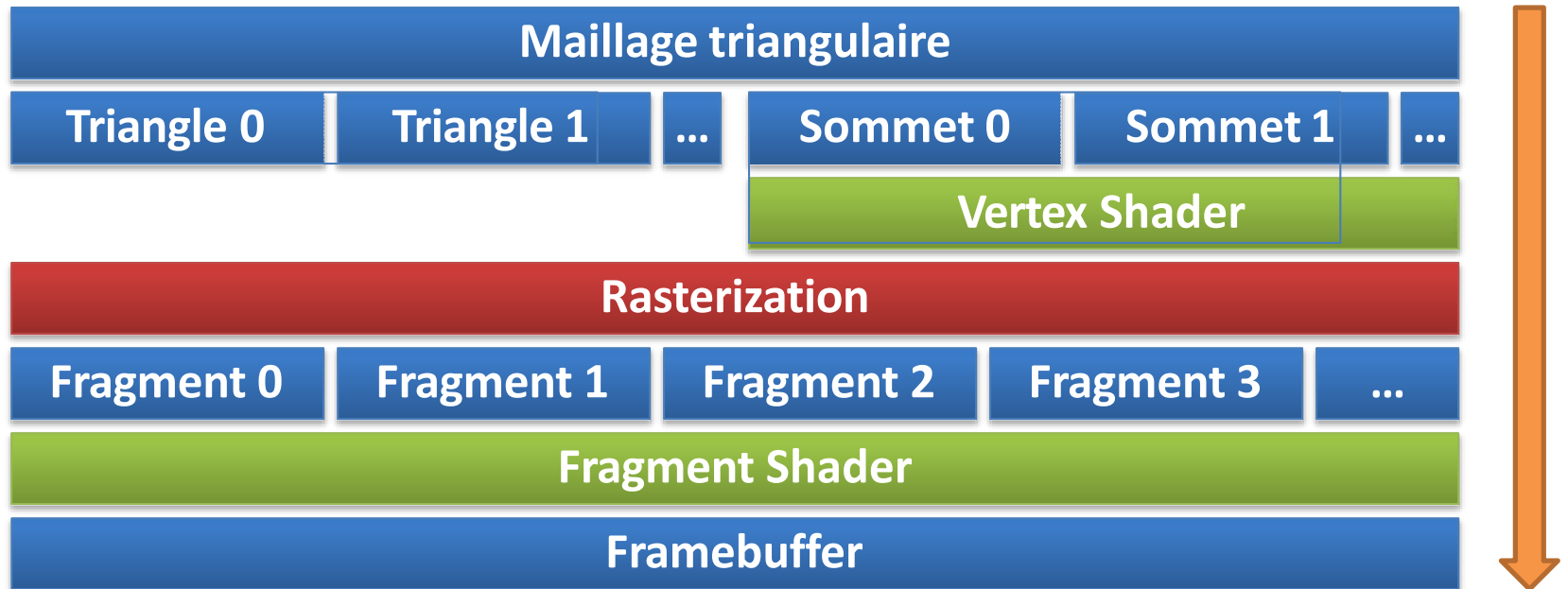
Configurable



Fixe

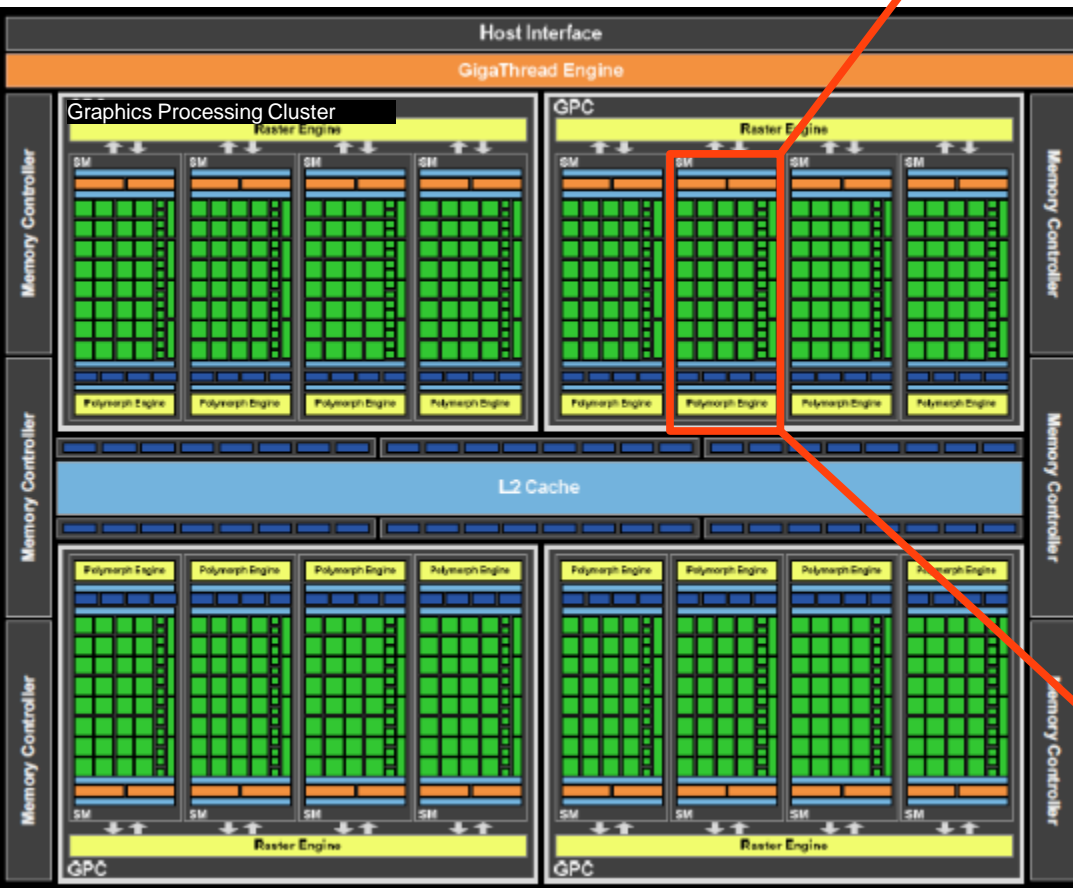
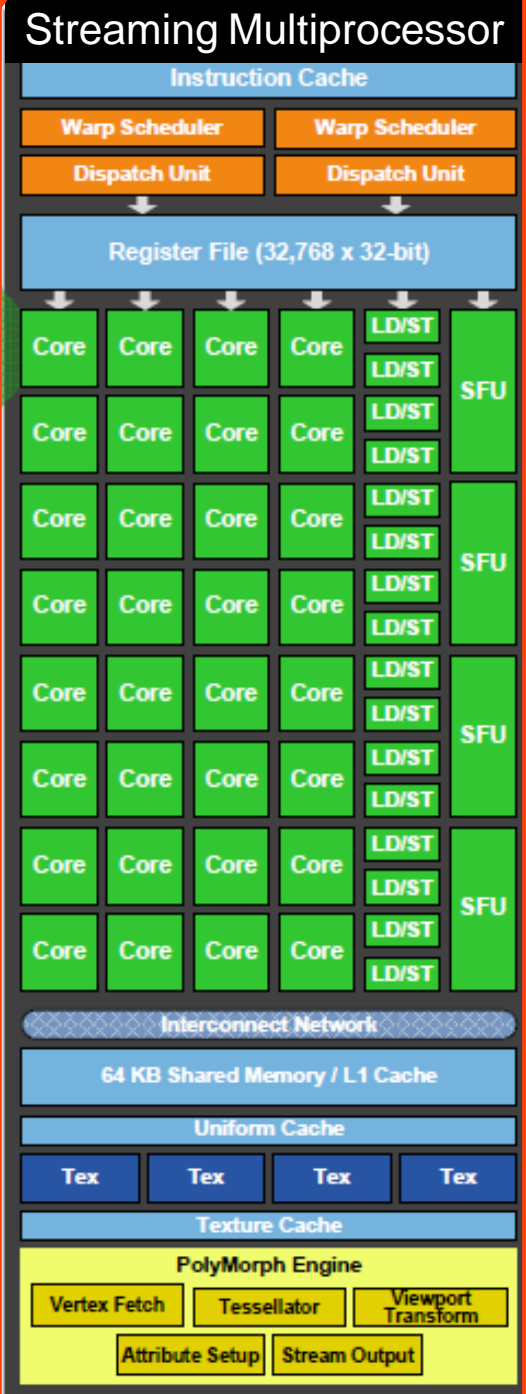
Etages majeurs

- Calcul en flux



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

Architecture d'un GPU



GPU

- **GPU = Processeur Graphique**

- permet des calculs complexes réalisés par la carte graphique
 - le CPU est libre pour réaliser d'autre tâches

- **Ce n'est pas un CPU !**

- Hautement parallèle : jusqu'à 4000 opérations en parallèle !
 - architecture hybride (SIMD/MIMD)
- Accès mémoire via des buffers spécialisés (*de – en - vrai*):
 - Textures (images, tableaux en lecture, 1D, 2D, 3D)
 - Vertex Buffers (tableaux de sommets, 1D)
 - Frame buffers (images en écriture, 2D)
- Circuits spécialisés non programmable
 - rasterisation, blending, etc.

- **Accès via une API graphique**

- **OpenGL**, DirectX, Vulkan, etc.

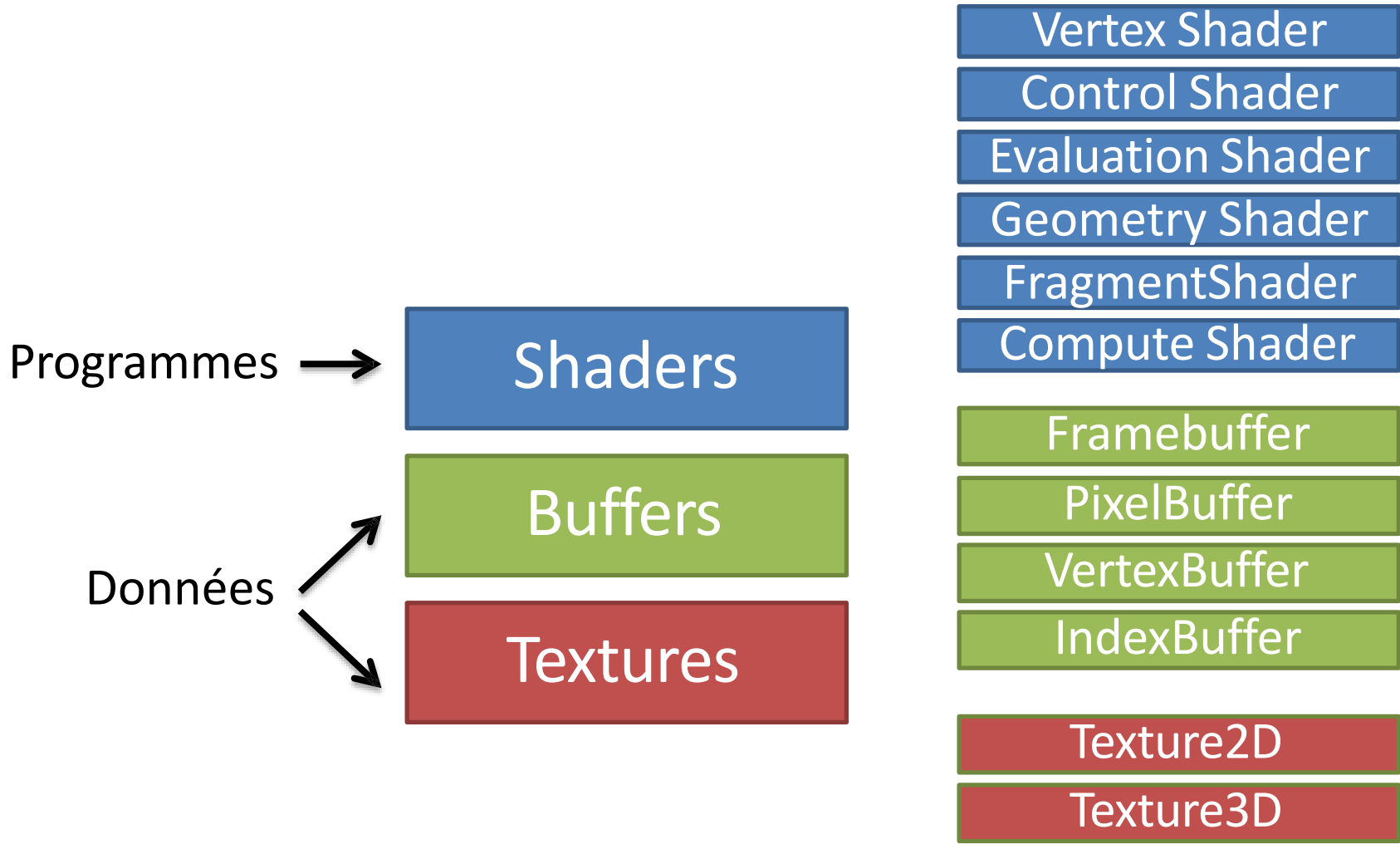
API OpenGL

- API graphique générique
 - Mac/PC/Linux/iOS/Android/html5/etc
- Plusieurs versions
 - OpenGL Classique (v1.2)
 - Pas de programmation GPU
 - Pipeline fixe
 - OpenGL Programmable (v2.0)
 - Programmation GPU (shaders)
 - Entrées-sorties formatées
 - **OpenGL Moderne (v3/v4)**
 - Shaders graphiques et shaders calcul,
 - Entrées sorties redéfinissables.
 - Nouveaux étages : geometry, tessellation
- Les bibliothèques GLAD et GLEW permettent de travailler avec les versions modernes d'OpenGL

Interfaçage au système d'exploitation

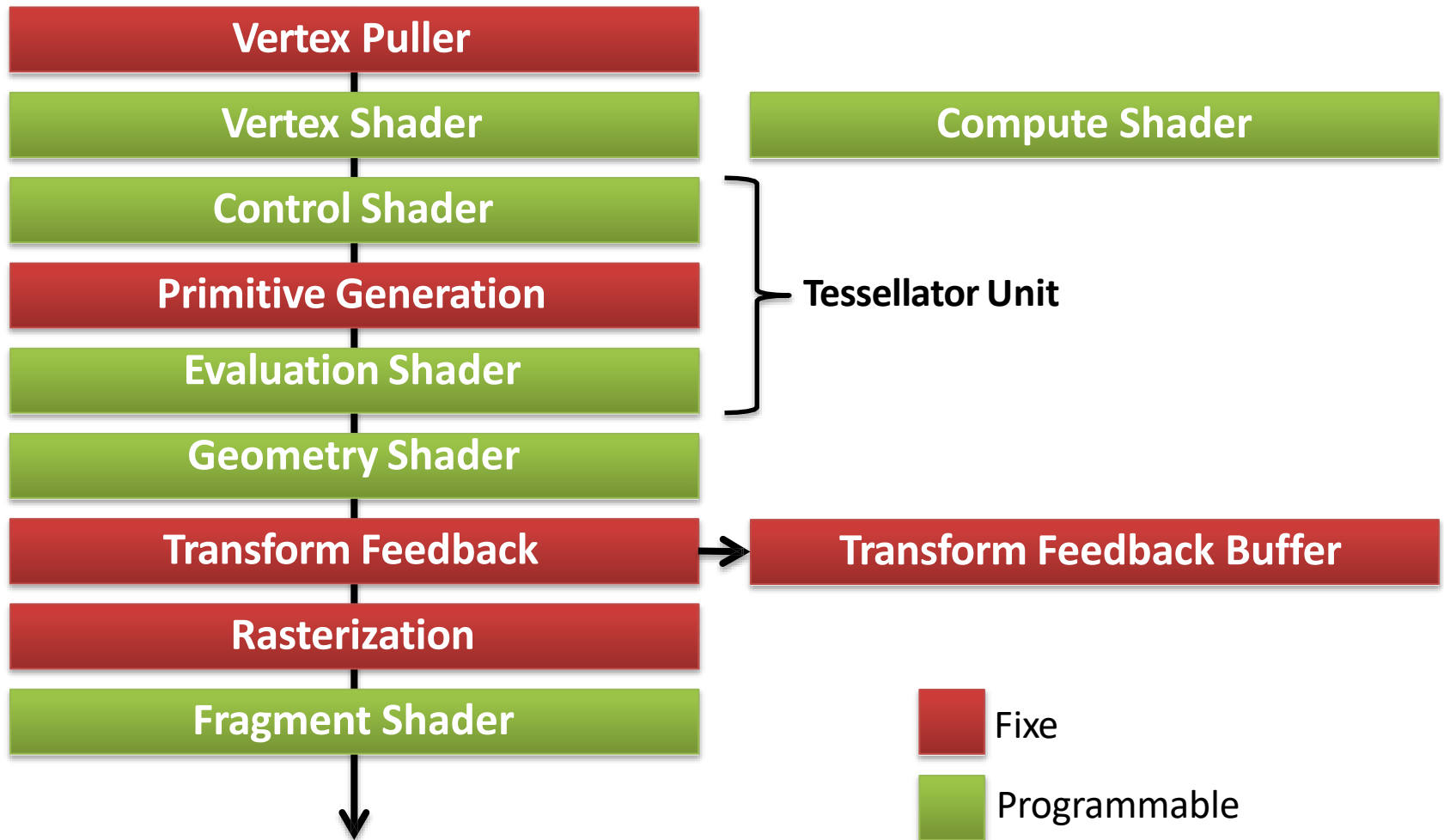
- Plusieurs bibliothèques existent:
 - **GLFW** (fortement recommandé)
 - FreeGlut
 - Qt
- Fournissent en général un mécanisme de **callback** pour
 - la mise à jour de l'image affichée
 - les évènements claviers
 - les évènements souris

OpenGL Moderne (v4.x)



Pipeline Moderne

OpenGL 4.3



Shaders

- **Vertex**
- **Control**
- **Evaluation**
- **Geometry**
- **Fragment**
- **Compute**
(calcul non graphique)
- **Mesh (2018)**

Pipeline Graphique = ensemble de shaders

- Synchronisation I/O

Amplification geometrique

- Geometry Shader :
 - Très Flexible
 - Faible amplification
- Tessellation :
 - Peu flexible
 - Grande amplification

OpenGL

- **Bibliothèque graphique 2D/3D**

- API entre le GPU et le programme utilisateur
- Rendu d'images composées de primitives :
 - Géométriques : points, lignes, polygones ...
 - Images : bitmap, textures
- Bas niveau
 - **Machine à états**, contrôlé par des **commandes**
 - **Sait uniquement convertir un triangle 2D en un ensemble de pixels !**
 - — Accéléré par le matériel graphique
- Portable (Linux, Windows, MacOS, SmartPhone, WebBrowser, ...)
 - langage C + interface pour tous les autres langages
(Java, Python, C#, OCaml, JavaScript, etc.)

OpenGL ?

ATTENTION aux versions !

- OpenGL 1.x, OpenGL 2.x => obsolètes !
- OpenGL 3.x avec rétro-compatibilité => à éviter !
- **OpenGL 3.x « core »** → **recommandé**
- OpenGL 4.x → = 3.x + nouvelles fonctionnalités
- WebGL, OpenGL-ES → proches de OpenGL 3.x « core »

Pipeline Graphique sur GPU

Page 8

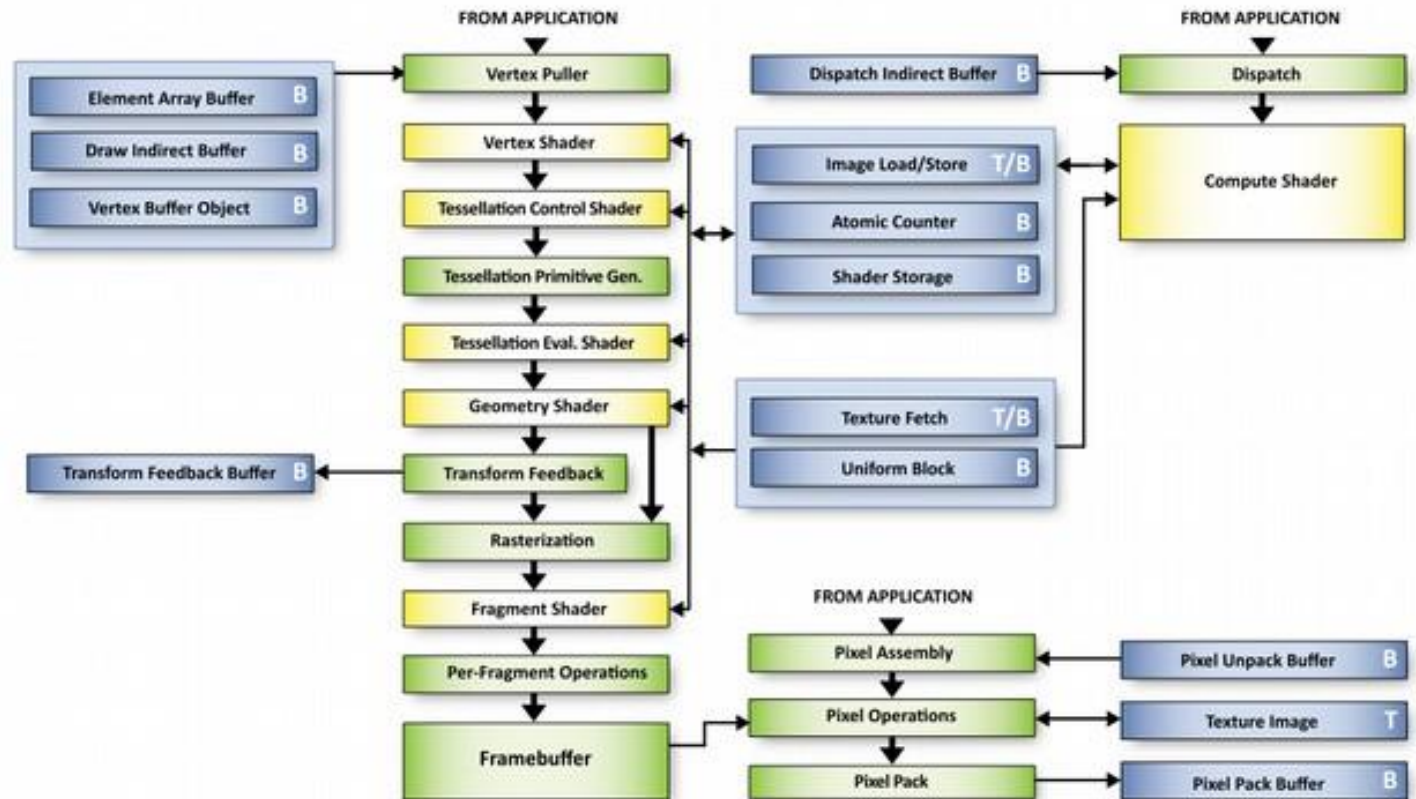
OpenGL 4.5 API Reference Card

OpenGL Pipeline

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Calls are made to allocate a GL context which is then associated with the window, then OpenGL commands can be issued.

The heavy black arrows in this illustration show the OpenGL pipeline and indicate data flow.

- Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- Green blocks indicate fixed function stages.
- Yellow blocks indicate programmable stages.
- T Texture binding
- B Buffer binding



Pipeline Graphique sur GPU

Vertex & Tessellation Details

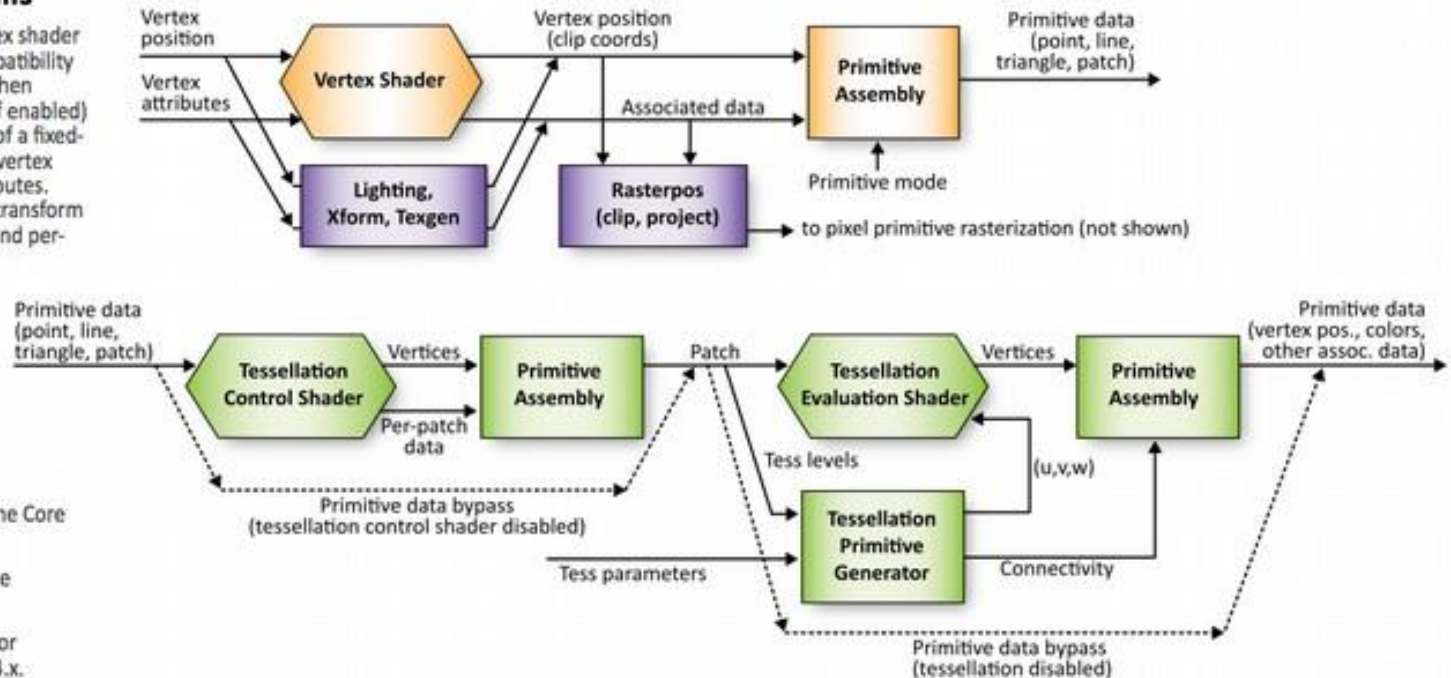
Each vertex is processed either by a vertex shader or fixed-function vertex processing (compatibility only) to generate a transformed vertex, then assembled into primitives. Tessellation (if enabled) operates on patch primitives, consisting of a fixed-size collection of vertices, each with per-vertex attributes and associated per-patch attributes. Tessellation control shaders (if enabled) transform an input patch and compute per-vertex and per-patch attributes for a new output patch.

A fixed-function primitive generator subdivides the patch according to tessellation levels computed in the tessellation control shaders or specified as fixed values in the API (TCS disabled). The tessellation evaluation shader computes the position and attributes of each vertex produced by the tessellator.

Orange blocks indicate features of the Core specification.

Purple blocks indicate features of the Compatibility specification.

Green blocks indicate features new or significantly changed with OpenGL 4.x.



Pipeline Graphique sur GPU

Geometry & Follow-on Details

Geometry shaders (if enabled) consume individual primitives built in previous primitive assembly stages. For each input primitive, the geometry shader can output zero or more vertices, with each vertex directed at a specific vertex stream. The vertices emitted to each stream are assembled into primitives according to the geometry shader's output primitive type.

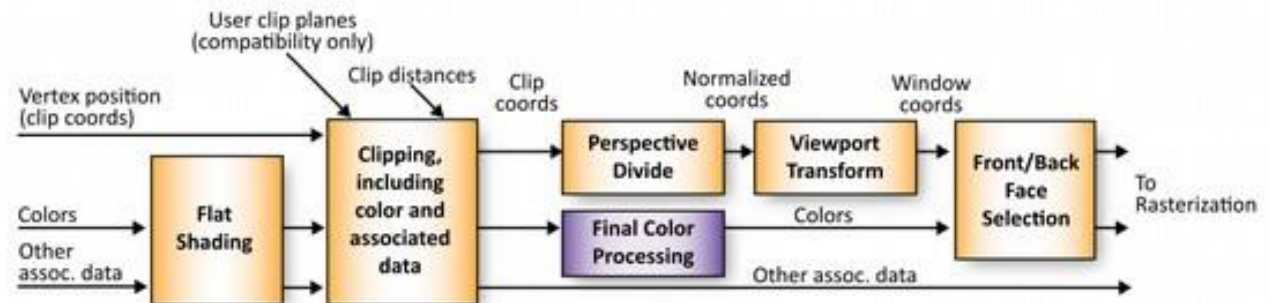
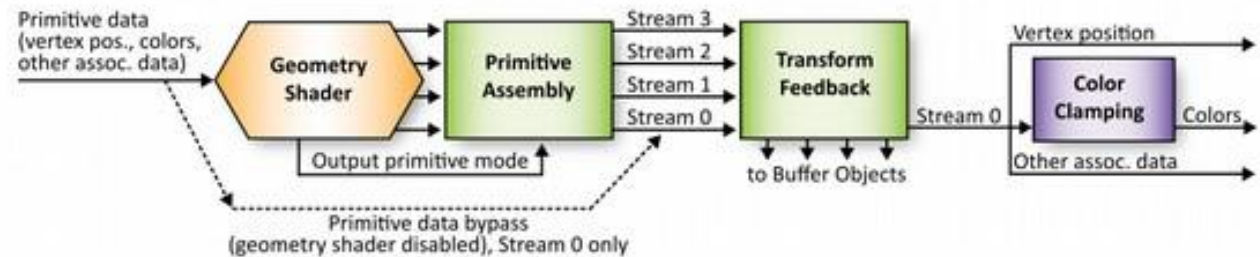
Transform feedback (if active) writes selected vertex attributes of the primitives of all vertex streams into buffer objects attached to one or more binding points.

Primitives on vertex stream zero are then processed by fixed-function stages, where they are clipped and prepared for rasterization.

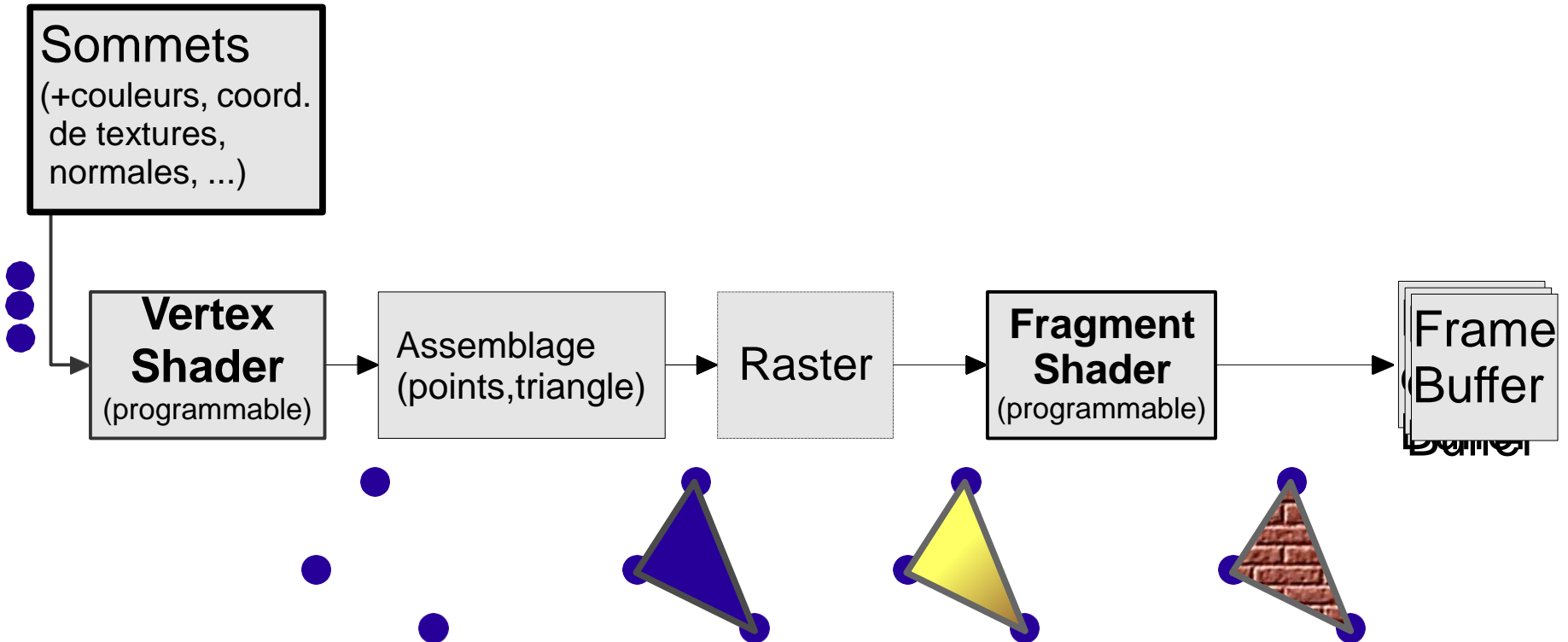
Orange blocks indicate features of the Core specification.

Purple blocks indicate features of the Compatibility specification.

Green blocks indicate features new or significantly changed with OpenGL 4.x.



Pipeline Graphique sur GPU



Pipeline : in/out

En entrée

- Une description numérique de la géométrie de la scène
 - = {primitives rastérisables}
 - ex. : ensemble de polygones
- Ensemble de paramètres :
 - un point de vue (caméra)
 - des attributs de matériaux associés à chaque objet
 - un ensemble de lumières
 - etc.

En sortie

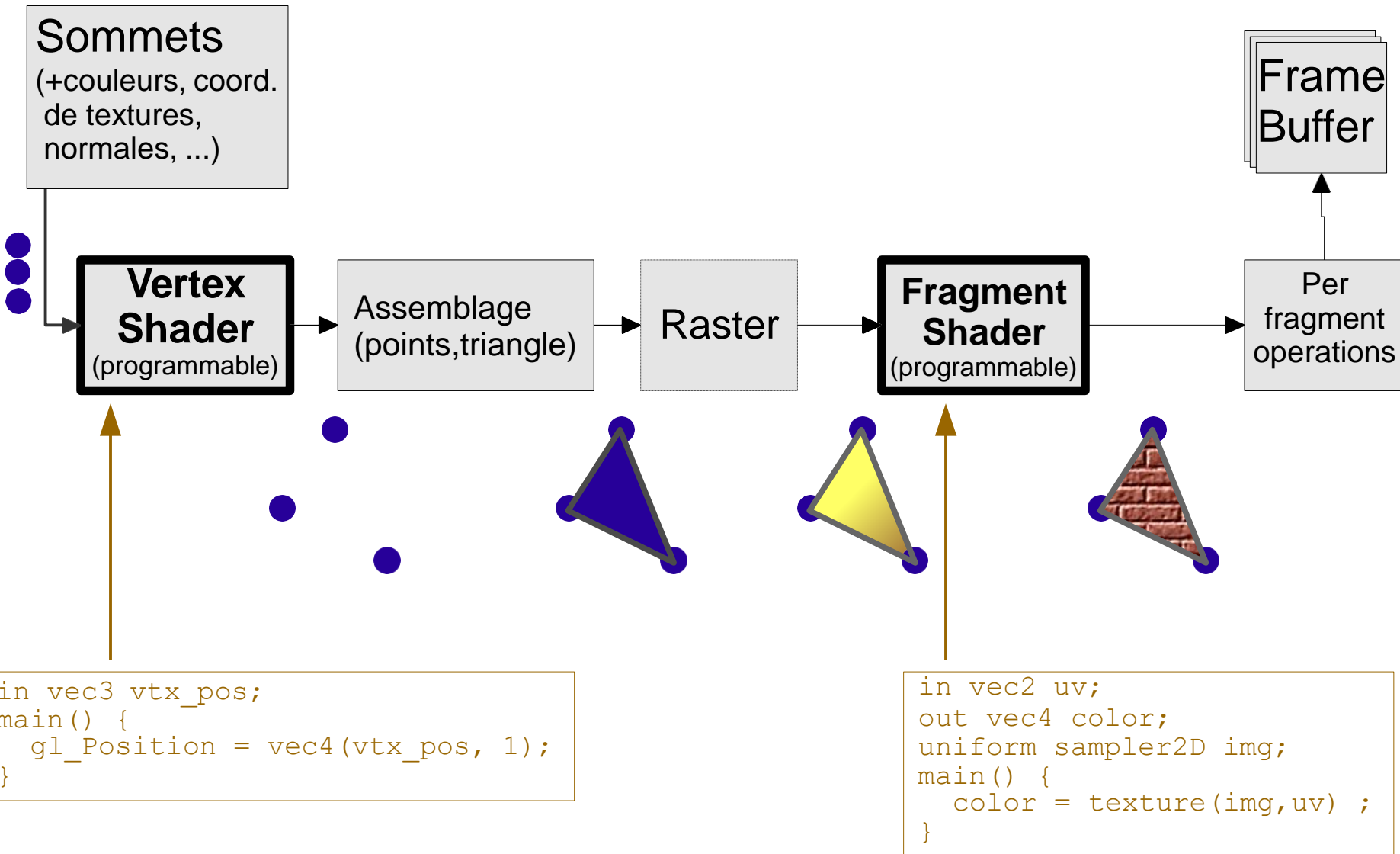
- une image = un tableau de pixels (couleurs RGB)

Algorithme de rendu par rasterisation

Pour chaque image :

- Effacer les tampons de destination (l'écran)
- Configurer la scène pour l'image courante:
 - positionner la caméra
 - positionner les sources lumineuses
 - etc.
- Pour chaque objet:
 - Charger la géométrie
 - Charger les textures
 - Configurer les shaders : matériau, transformations, etc.
 - Activer les shaders
 - **Tracer l'objet**
 - Restaurer les différents états
- Afficher l'image calculée à l'écran (double buffering)
- Calculer l'image suivante ...

Les étages programmables



Langages de programmation

Shader

- (petit) programme exécuté sur le GPU

Programmable via

- des langages de haut niveau (proche du C/C++)
 - **GLSL** (*OpenGL Shading Language*)
 - compilateur intégré dans le driver OpenGL (≥ 2.0)
 - génération et compilation de code à la volée
 - standard ouvert
 - **HLSL** (*Microsoft*)
 - DirectX only

OpenGL Shaders

Langage : GLSL (OpenGL Shading Language)

- Proche du C

Pas d'accès mémoire direct

- En lecture :
 - Vertex Buffer Objects – VBO (tableau de sommets avec attributs)
 - accès indirect
 - Textures (tableaux 1D, 2D, 3D, etc.)
- En écriture :
 - Frame Buffer Objects – FBO (tableaux 2D)

Compilation à la volée par le driver OpenGL

- Shader == char*
- Source code spécifié via des fonctions OpenGL

OpenGL Shaders

Deux types shaders (threads) :

- sommets et pixels

Pas de synchronisation, pas de mémoire partagée

- « *on ne veut pas que les pixels parlent entre eux...* » !!

Fonction principale

- fonction **main** (sans arguments)

```
void main(void) {  
    /* ... */  
}
```

Paramètres constants

- variable globale avec le qualificatif « uniform »

```
uniform float intensity;
```

- valeur définie par l'hôte (**glUniform***(...))

OpenGL Shaders

- données -

En entrée

- qualificatif « in »
`in vec3 vertex_position;`
- VBO, variable spéciale, ou valeur calculée par l'étage précédent

En sortie

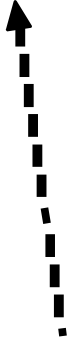
- qualificatif « out »
`out vec4 color;`
- variable spéciale, valeur envoyée à l'étage suivant, ou FBO

GLSL : 1^{er} exemple

Vertex shader

```
in vec3 vtx_position ;
```

```
void main(void) {  
    gl_Position.xy = vtx_position.xy;  
    gl_Position.zw = vec2(0,1) ;  
}
```

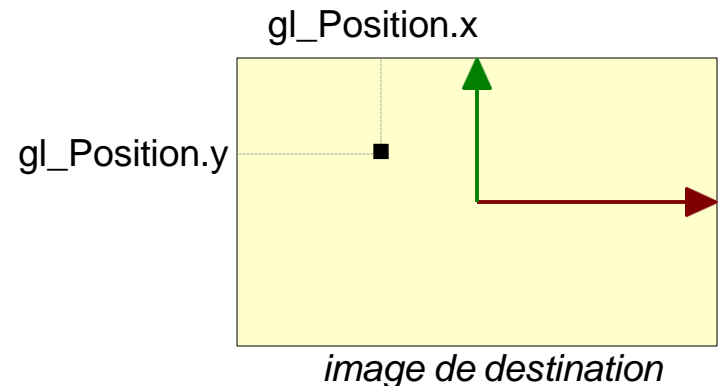


variable spéciale = position du sommet
dans l'image 2D normalisée : $[-1,1] \times [-1,1]$

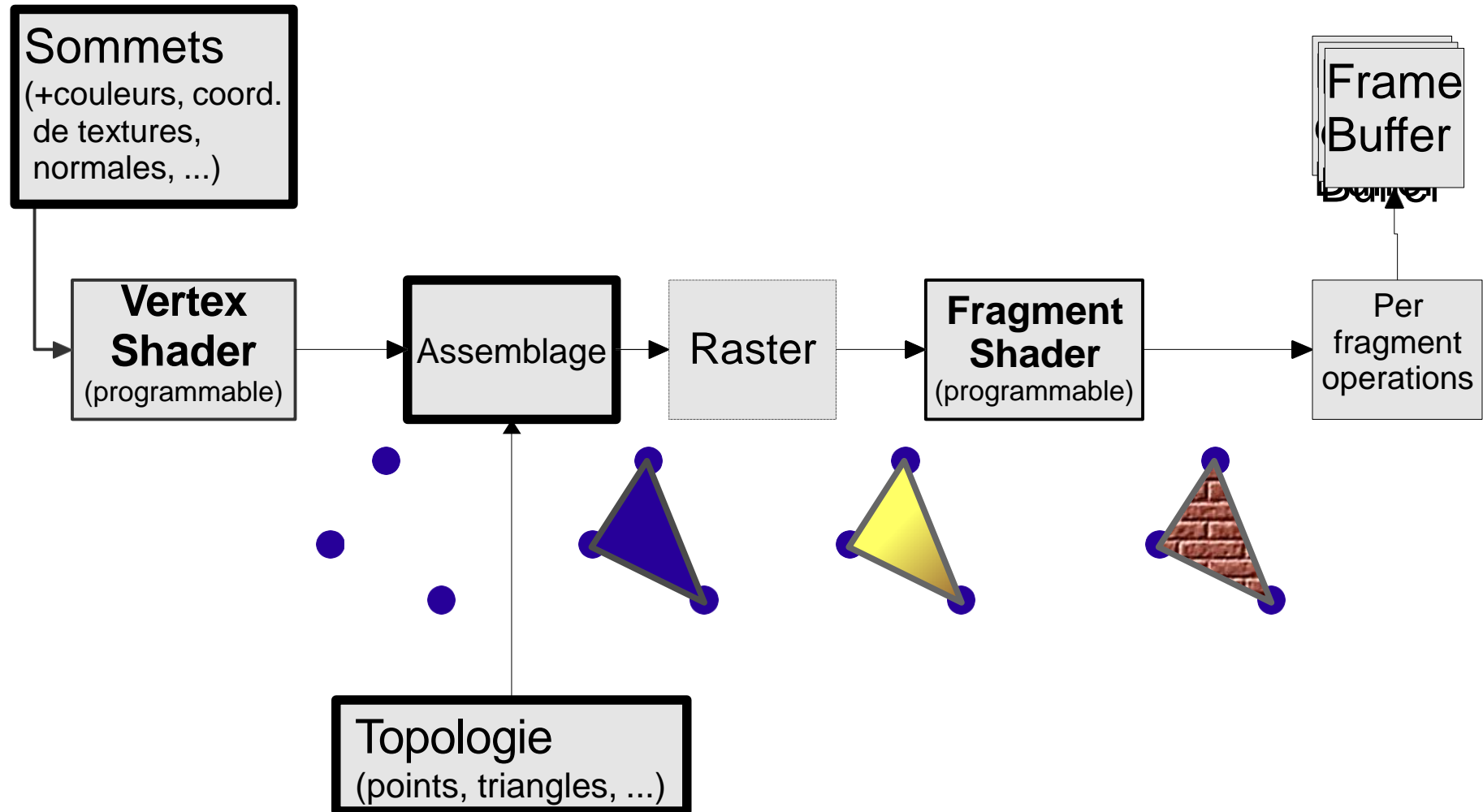
Fragment shader

```
out vec4 color ;
```

```
void main(void) {  
    color.rgb = vec3(1,0,0);  
    color.a = 1 ;  
}
```



Pipeline Graphique sur GPU

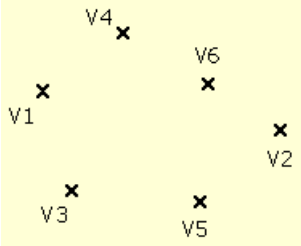
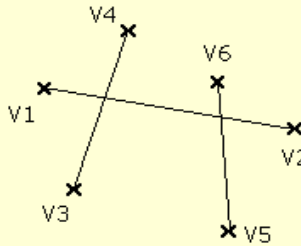
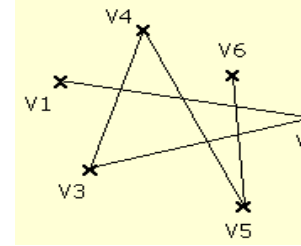
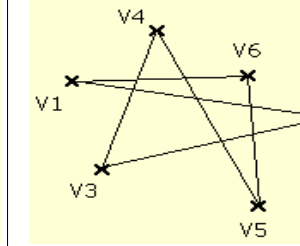
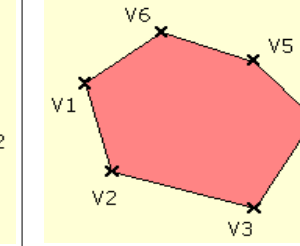
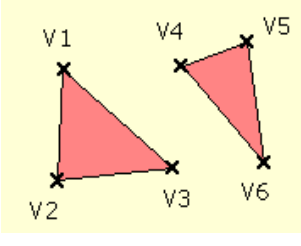
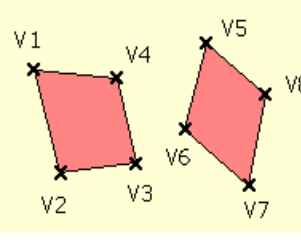
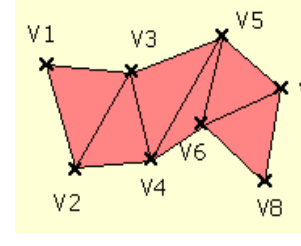
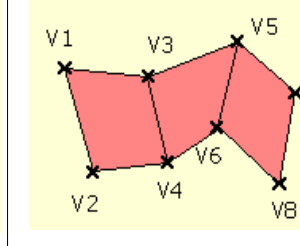
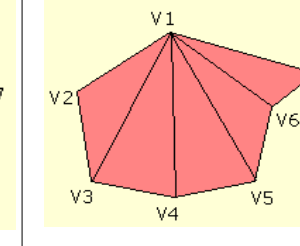


Primitives Géométriques

Primitives de bases:

— points, lignes, **triangles**

Liste des primitives:

				
GL_POINTS	GL_LINES	GL_LINE_STRIP	GL_LINE_LOOP	GL_POLYGON
				
GL_TRIANGLES	GL_QUADS	GL_TRIANGLE_STRIP	GL_QUAD_STRIP	GL_TRIANGLE_FAN

Spécifier la géométrie

API référence

Géométrie = tableaux de sommets avec attributs

- positions + couleurs, normales, coordonnées de texture, etc.
- stockée dans des **BufferObjects** (données accessibles par le GPU)
 - création :
`uint buffer_id ;`
`glGenBuffers(1, &buffer_id) ;`
 - activation :
`glBindBuffer(GL_ARRAY_BUFFER, buffer_id) ;`
 - copie (de la mémoire CPU vers la mémoire du GPU) :
`glBufferData(GL_ARRAY_BUFFER, size, pointer, GL_STATIC_DRAW) ;`
 - **GL_ARRAY_BUFFER** : type de *buffers* pour les attributs de sommets

Spécifier la géométrie

API référence

Memory layout specification:

- **glVertexAttribPointer**(uint index, int size, enum type, bool normalized, uint stride, const void *offset)
 - index : numéro d'attribut (lien avec le vertex shader)
 - type = GL_FLOAT, GL_INT, etc...
 - size = nombre de coordonnées
 - stride = nbre d'octets entre le début de deux données (0 si les données sont compactées), permet d'entrelacer les attributs
 - offset = offset en octets du 1^{er} attribut
- **gl{Enable,Disable}VertexAttribArray**(uint index)
 - activation des tableaux

Tracer la géométrie

API référence

Direct:

- `glDrawArrays(GLenum mode, int first, sizei count)`
 - utilisé lorsque les sommets peuvent être envoyés linéairement (ex. : « triangle strip »)

Via une liste d'indices (stockée dans un `ELEMENT_ARRAY_BUFFER`) :

- `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_buffer_id) ;`
`glDrawElements(mode, sizei count, enum type, (void*)0) ;`
 - Exemple : pour les maillages

Exemple : tracer un maillage

Un maillage c'est :

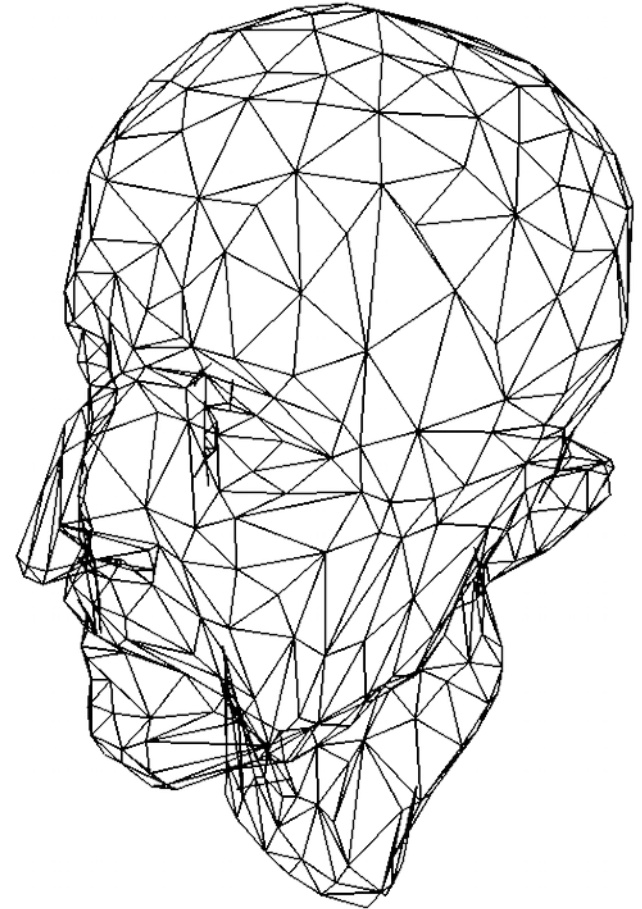
- une liste de sommets attribués

```
struct MyVertex {  
    float position[3];  
    float normal[3];  
    /* ... */  
};
```

```
MyVertex vertices[nb_vertices];
```

- une liste de faces (triangulaires)
 - un triangle = indices des trois sommets

```
uint faces[3][nb_faces];
```



Exemple : tracer un maillage

Initialisation:

```
GLuint vertex_buffer, index_buffer ;
glGenBuffers(2, &vertex_buffer) ;
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer) ;
glBufferData(GL_ARRAY_BUFFER, sizeof(MyVertex)*nb_vertices, vertices) ;
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_buffer) ;
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(int)*3*nb_faces, faces) ;
```

Rendu:

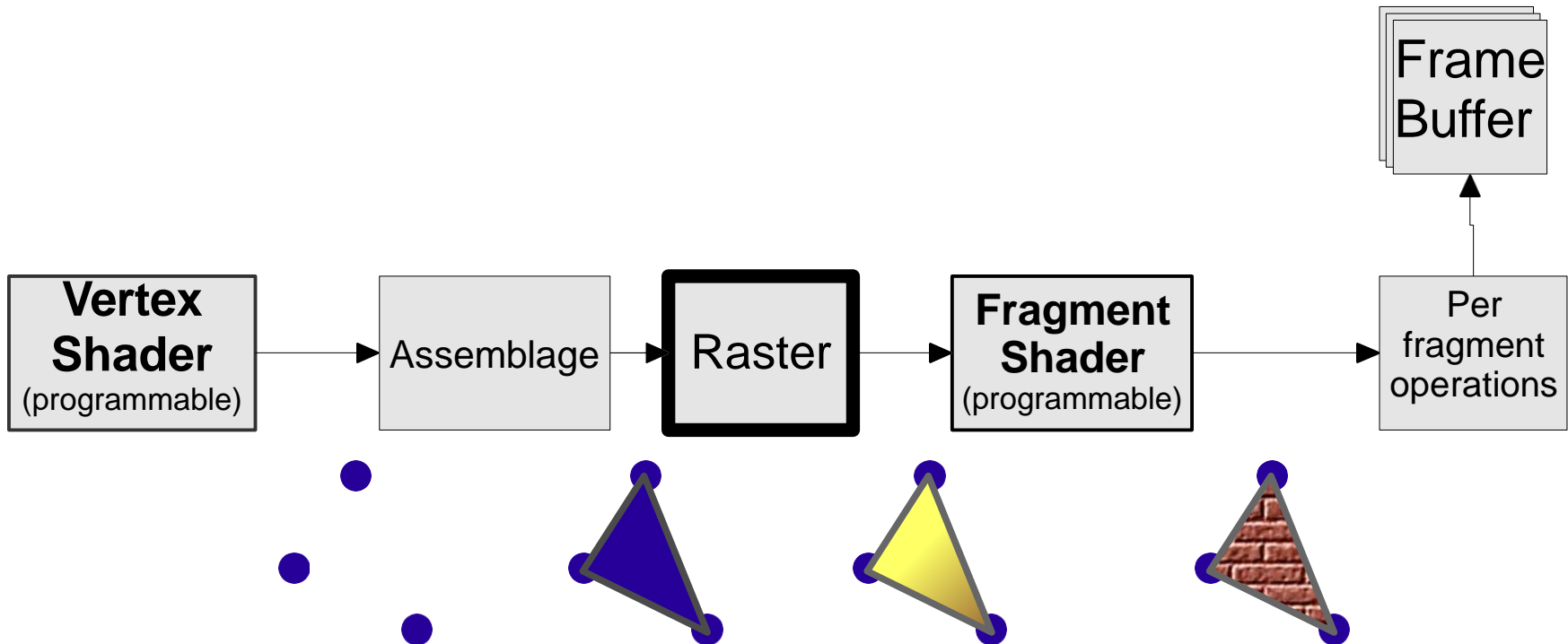
```
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer) ;

glVertexAttribPointer(0, 3, GL_FLOAT,          GL_FALSE, sizeof(MyVertex), (void*)0);
glVertexAttribPointer(1, 3, GL_FLOAT,          GL_FALSE, sizeof(MyVertex), (void*)12);

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

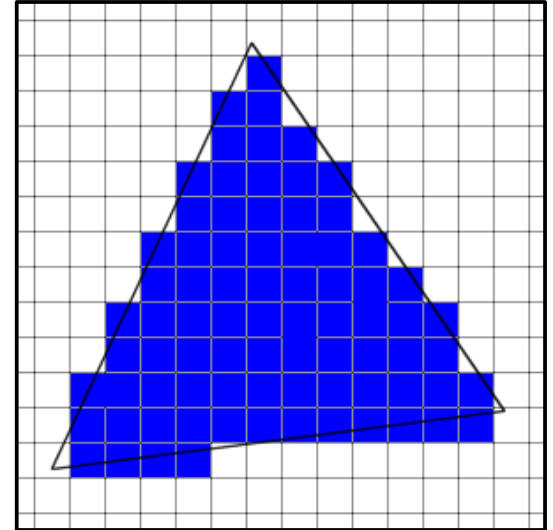
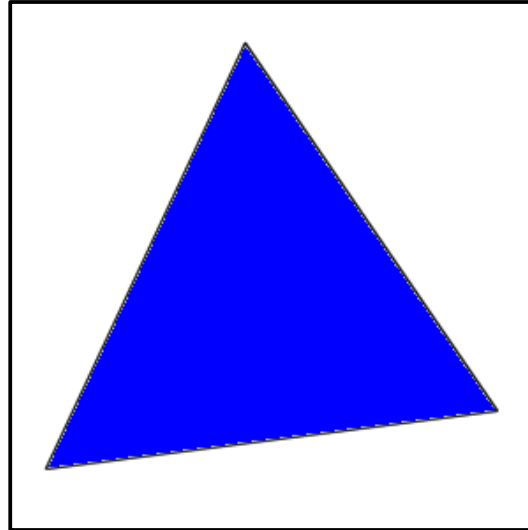
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vertex_buffer) ;
glDrawElements(GL_TRIANGLES, 3*nb_faces, GL_UNSIGNED_INT, (void*)0);
```

Pipeline Graphique sur GPU



Rastérisation : Génération des fragments

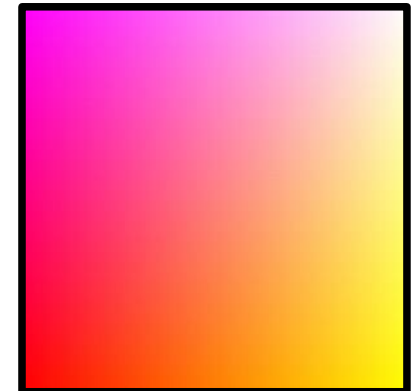
- Un fragment correspond à un pixel dans l'image final



- Les informations de couleur, coordonnées de textures, profondeurs, etc. sont assignés à chaque fragment
- Ces informations sont interpolées à partir des valeurs aux sommets

- Exemple:

```
float colors[3][] =  
    {{1,0,0},{1,0,1},{1,1,1},{1,1,0}};  
float positions[2][] =  
    {{0,0},{1,0},{1,1},{0,1}};
```



GLSL : 2^{ème} exemple

- *lien vertex-fragment* -

Vertex shader

```
in vec3 vtx_position ;  
out vec3 var_color ;  
  
void main(void) {  
    gl_Position.xy = vtx_position.xy ;  
    gl_Position.zw = vec2(0,1) ;  
  
    var_color = vtx_position.xyz ;  
}
```

- variable en sortie du vertex shader
- valeur différente pour chaque sommet
- interpolée par le raster et transmit au fragment shader

Fragment shader

```
out vec4 color ;  
  
in vec3 var_color ;  
  
void main(void) {  
    color.rgb = var_color;  
    color.a = 1 ;  
}
```

GLSL : 3^{ème} exemple

- et la 3D ? -

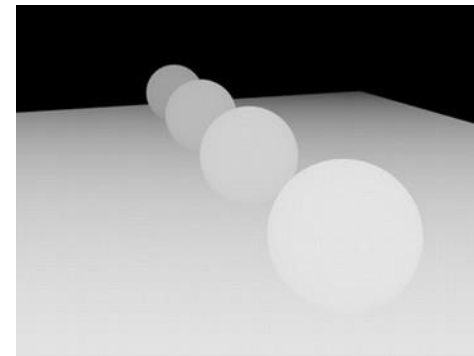
Vertex shader

```
in vec3 vtx_position ;  
out vec3 var_color ;  
  
void main(void) {  
    gl_Position.xyz = vtx_position;  
    gl_Position.w = 1 ;  
  
    var_color = vec3(vtx_position.z) ;  
}
```

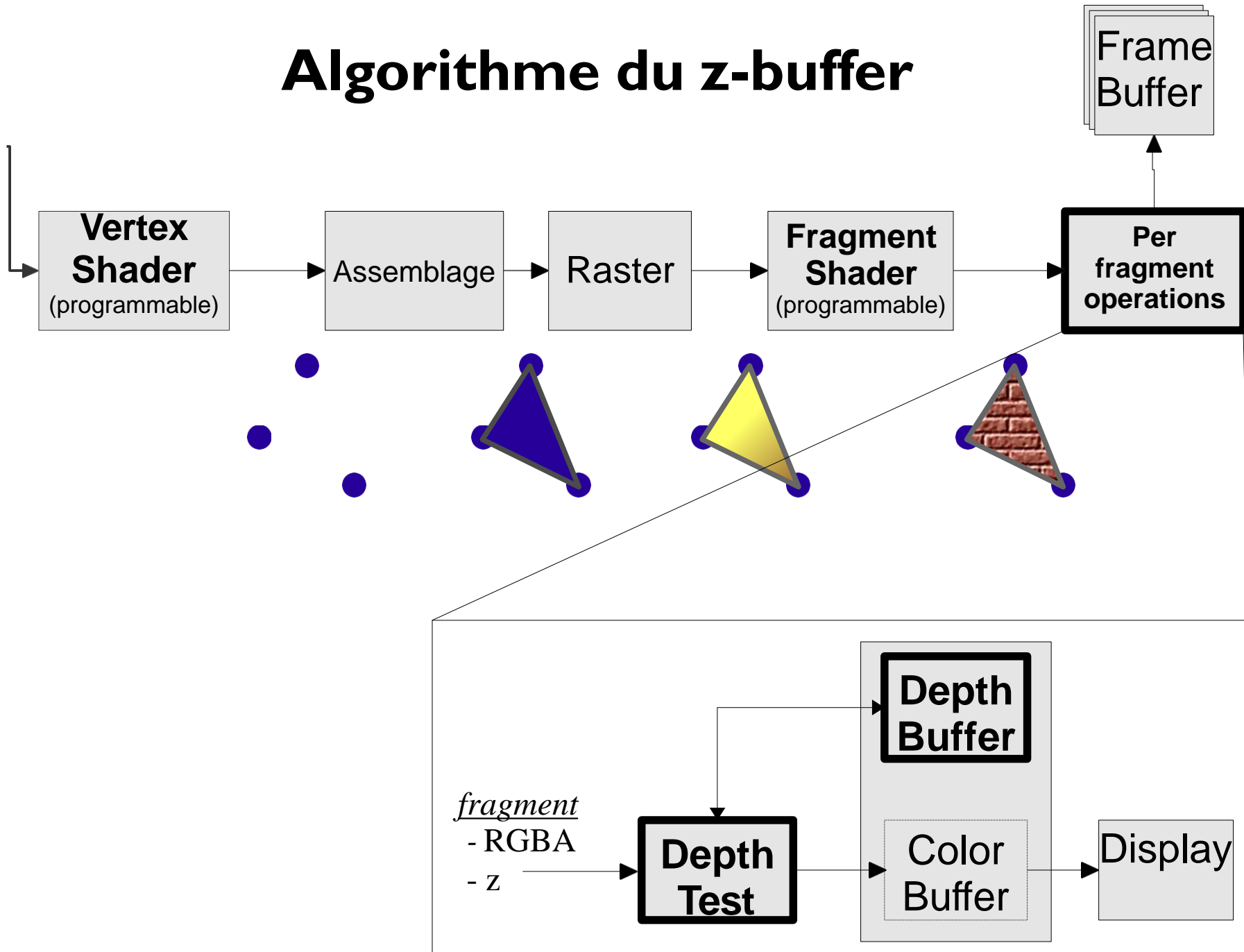
Fragment shader

```
out vec4 color ;  
in vec3 var_color ;  
  
void main(void) {  
    color.rgb = var_color;  
    color.a = 1 ;  
}
```

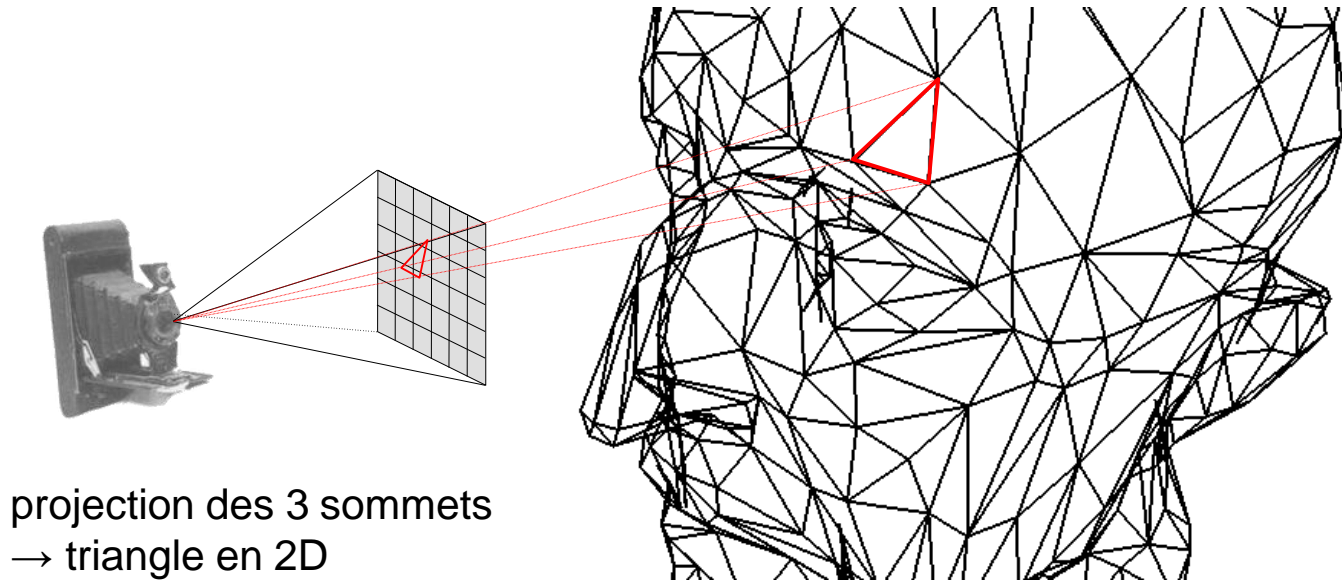
- spécifier la profondeur « z » du sommet (entre[-1,1])
- et s'en servir pour « éliminer les parties cachées »
→ algorithme du « z-buffer »



Algorithme du z-buffer



Rappel : Rastérisation



- **Rastérisation**

- Pour chaque primitive P_i , trouver les rayons intersectant P_i
- Rendu en deux étapes
 - projection des primitives sur l'écran (*forward projection*)
 - discrétisation (conversion des primitives 2D en pixels)
- scène = ensemble de primitives « rastérisables »
 - **problème : élimination des parties cachées**
(*lancer de rayon : intersection la plus proche*)

Algorithme du Z-Buffer

Permet l'élimination des parties cachées

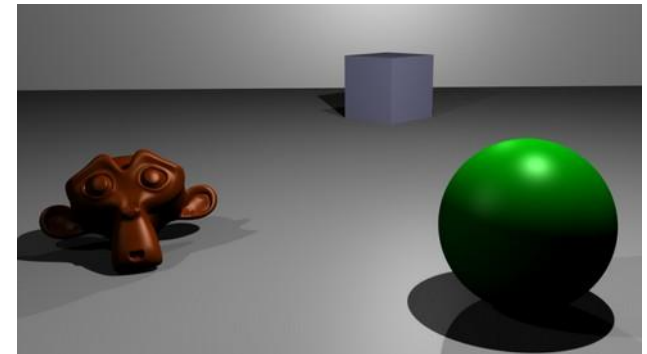
— Profondeur = distance par rapport à l'oeil (normalisé entre 0 et 1)

Comparer la profondeur du fragment avec la valeur stockée dans le tampon de profondeur

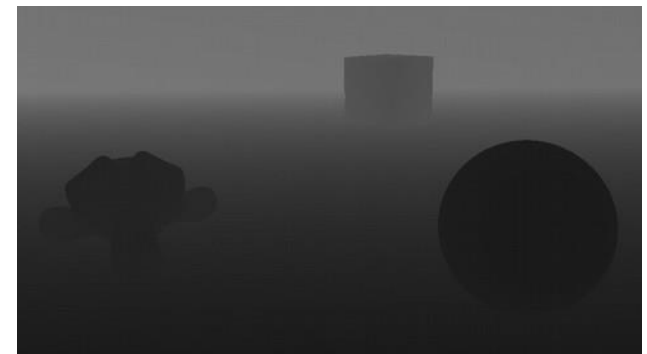
```
if (fragment.z < depthBuffer(x,y))  
{  
    depthBuffer(x,y) <- fragment.z  
    colorBuffer(x,y) <- fragment.color  
}
```

OpenGL :

```
glEnable/Disable(GL_DEPTH_TEST)  
glDepthClear(1.0)  
glClear(GL_DEPTH_BUFFER_BIT)  
glDepthFunc(GLenum func)  
    func = GL_LESS, GL_GREATER,  
          GL_EQUAL, GL_LEQUAL, ...
```



color buffer



depth buffer

Z-buffer : pour et contre

Pour :

- Facile à implémenter
- Travaille dans l'espace image
 - Pas de prétraitement

Contre :

- Coût en mémoire
- Travaille dans l'espace image
 - Aliasing / artefacts
- N'enlève pas le problème d'ordre pour la transparence
- Coût en $O(n \cdot p)$
 - n = nombre de pixel moyen par polygone
 - p = nombre de polygones
 - Un même pixel peut être calculé plusieurs fois
- Comparaison
 - Tracé de rayon : $O(N \cdot \ln(p))$ (N = nombre de pixel de l'image)

GLSL : 4^{ème} exemple

- *lien host-shaders* -

Vertex shader

```
in vec3 vtx_position ;  
out vec3 var_color ;  
uniform vec2 offset ;  
  
void main(void) {  
    gl_Position.xy = vtx_position.xy + offset ;  
    gl_Position.zw = vec2(0,1) ;  
  
    var_color = vtx_position.xyz ;  
}
```

Fragment shader

```
out vec4 color ;  
in vec3 var_color ;  
uniform float intensity ;  
  
void main(void) {  
    color.rgb = var_color * intensity ;  
    color.a = 1 ;  
}
```

- variables « uniformes »
- valeurs définies par l'application
- valeurs constantes durant le tracé d'un ensemble de primitives
ex. : valeurs définis par image, par objet, mais **pas** par primitives

GLSL : 5^{ème} exemple

- *résumé - système de particules 2D -*

Vertex shader

```
in vec2 vtx_position ;
in float vtx_velocity ;
out float velocity ;
uniform float scale ;
uniform vec2 translation ;

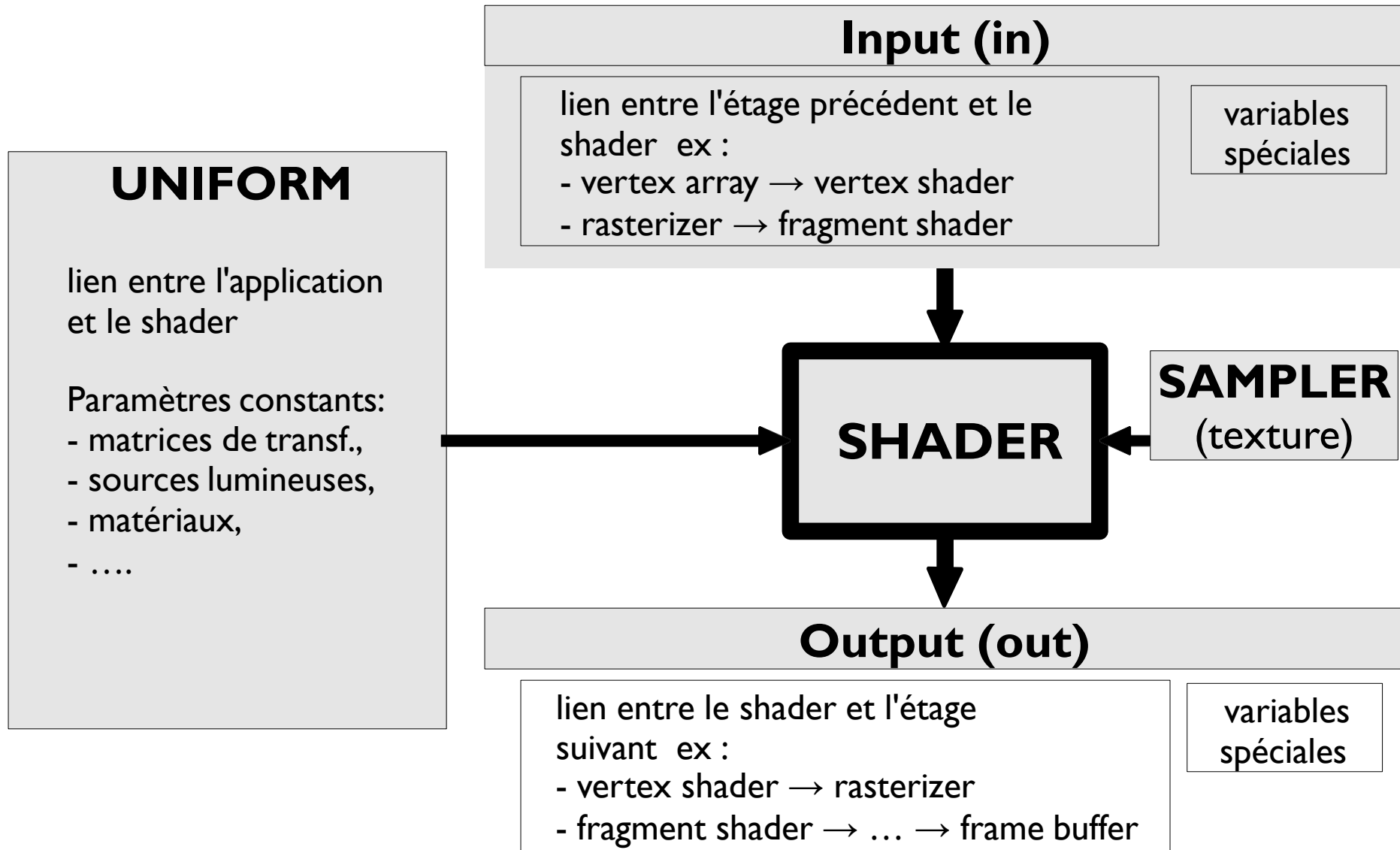
void main(void) {
    gl_Position.xy = vtx_position * scale
                    + translation ;
    gl_Position.zw = vec2(0,1) ;
    velocity = vtx_velocity ;
}
```

Fragment shader

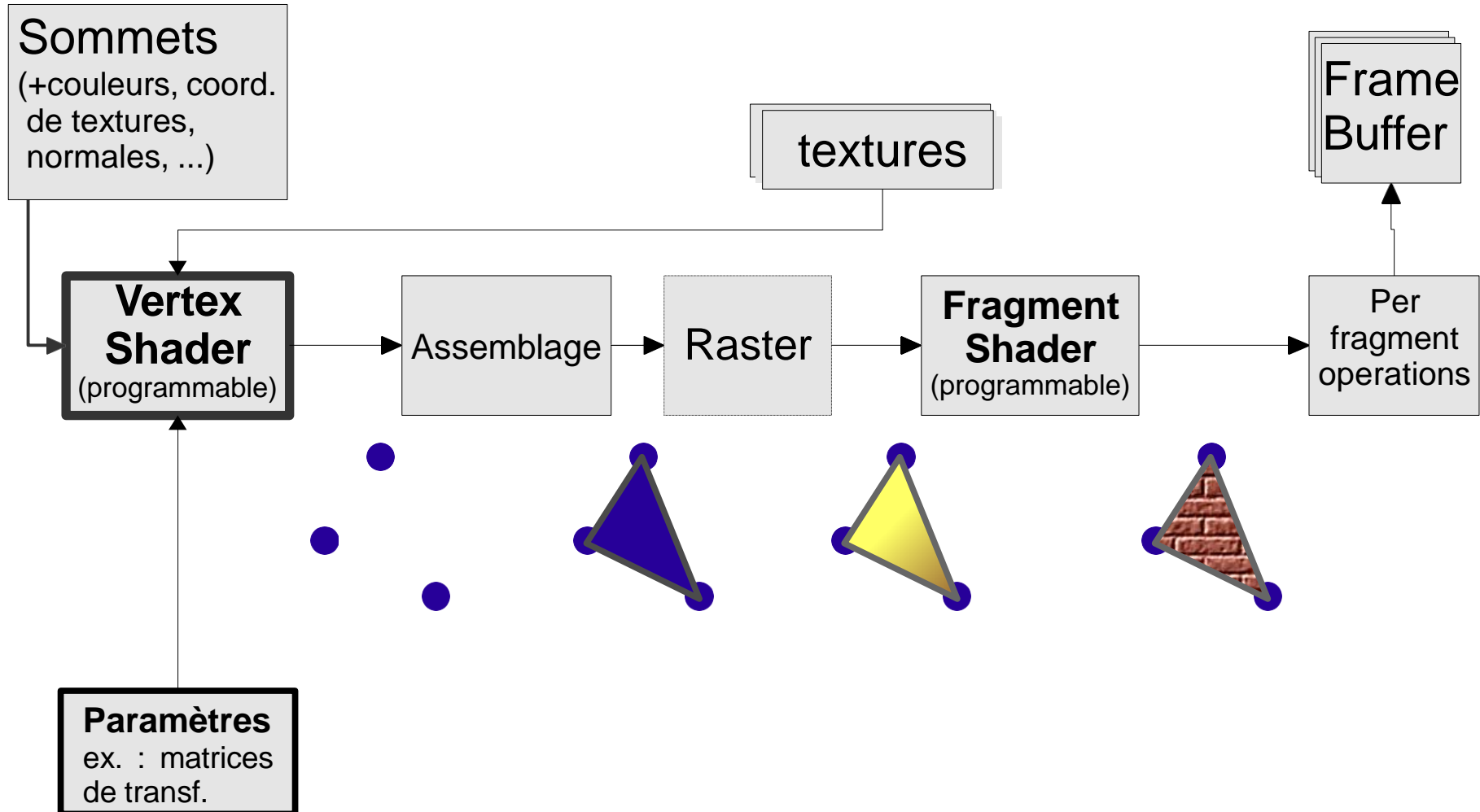
```
in float velocity ;
uniform float velocity_max ;
out vec4 color ;

void main(void) {
    color.rgb = vec3(velocity / velocity_max);
    color.a = 1 ;
}
```

Shading Processor - résumé



Pipeline Graphique sur GPU



Vertex Shader Input

Exécuté pour chaque sommet

En entrée : un sommet

- les attributs du sommet
 - sémantique définie par l'utilisateur
 - ex. : position, normale, couleurs, coordonnées de textures, ...
`in vec3 position;`
`in vec4 color;`
 - spécifiés via les vertex array
 - « **in** » : lien entre l'étage précédant et le shader courant
- un ensemble de paramètres constants pour un groupe de sommets (**uniform**)
 - lien entre l'application et les shaders
`uniform vec3 light_position;`
- on ne dispose d'aucune information topologique
 - pas d'arête, pas de face, pas d'info. sur le voisinage

Vertex Shader Output

On ne peut pas générer ou détruire des sommets

- 1 sommet en entrée → 1 sommet en sortie

Sortie : un sommet avec ses attributs transformés

- attributs spéciaux
 - coordonnées homogènes du sommet dans l'espace écran normalisé : **gl_Position** (obligatoire !)
 - ...
- attributs interpolés par le raster
 - couleurs, coordonnées de texture, normales, etc.
 - correspondent aux données d'entrées (*varying*) du fragment shader

```
out vec3 color;
```

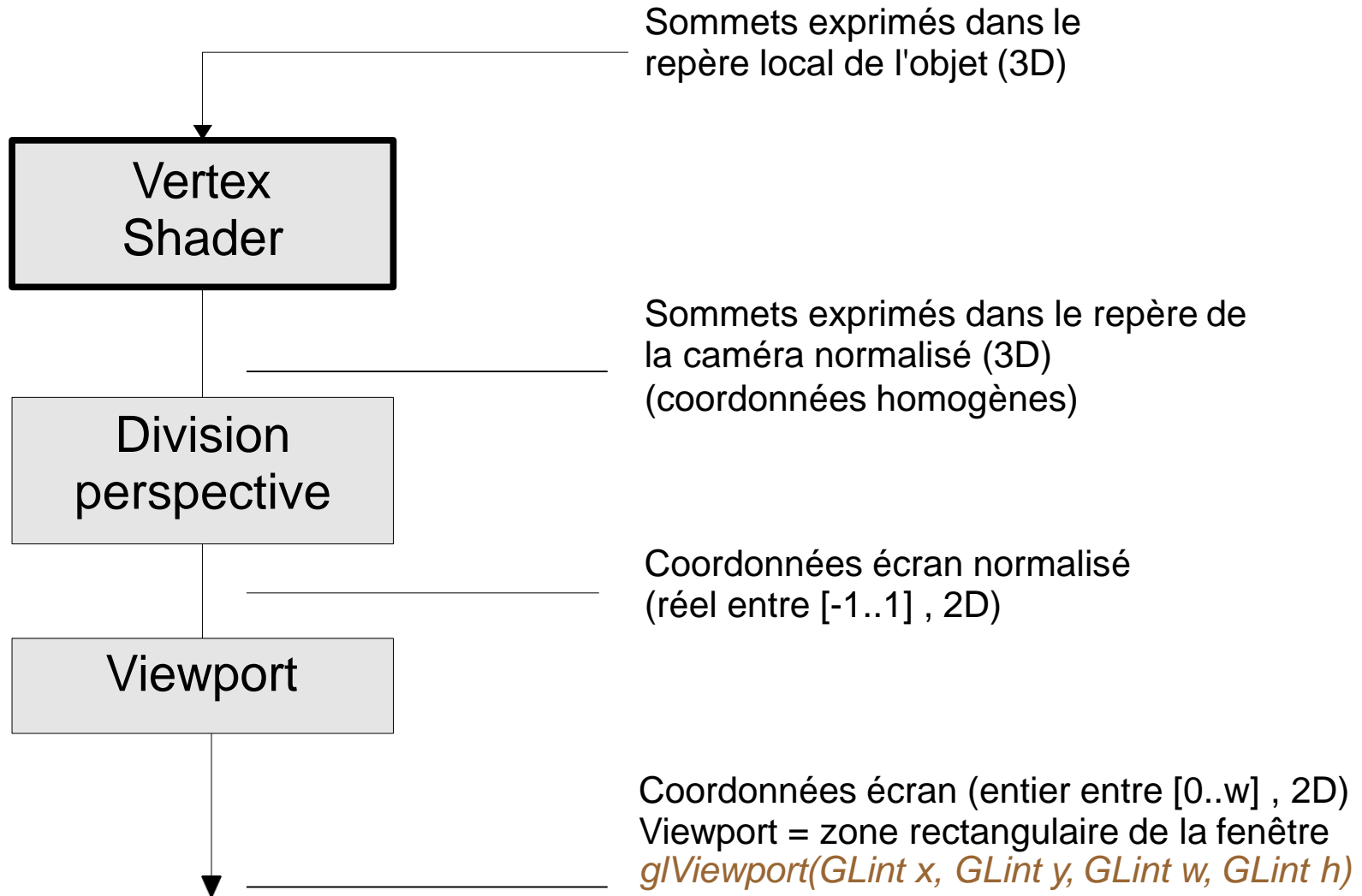
```
out vec2 tex_coord;
```

Vertex Shader

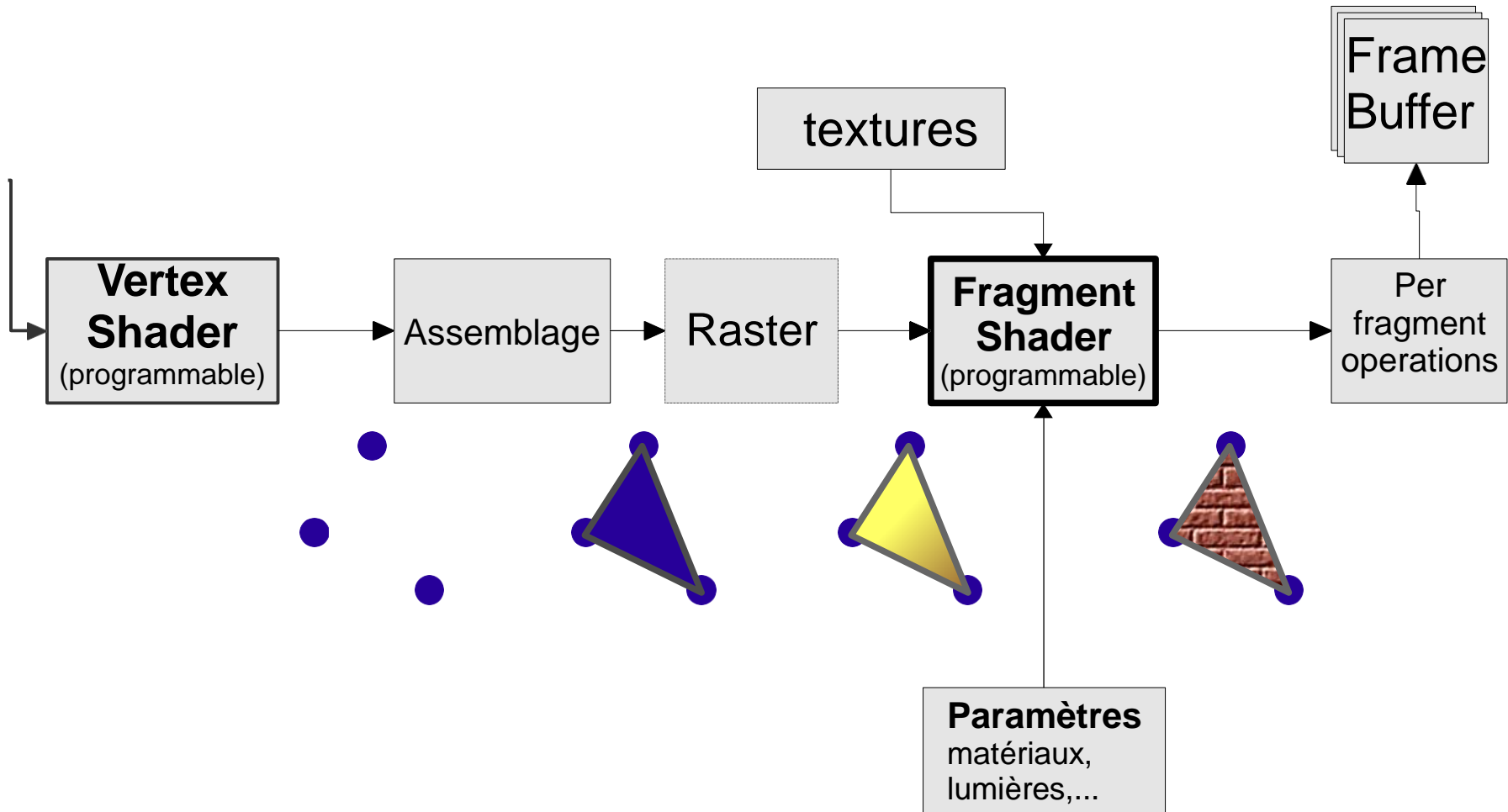
Ce qui peut/doit être fait

- transformations de modélisation et de projection des sommets (obligatoire), et des normales
- éclairage par sommet (+ color material)
- génération et transformation des coordonnées de texture
- ...

Vertex pipeline : résumé



Pipeline Graphique sur GPU



Fragment Shader

Programmation au niveau des fragments

Exemple:

- éclairage par pixels,
- application des textures,
- effet de brouillard,
- traitement d'images,
- etc.

Ne contrôle/permets pas:

- pas d'accès en lecture ou écriture au frame-buffer
 - => le blending, alpha test, stencil op, depth test, etc. sont réalisés après
- le mode d'accès aux textures (filtrage, répétition, ...)

Fragment Shader Input

Exécuté pour chaque fragment généré par la rasterisation

En entrée : un fragment

- attributs résultant de l'interpolation
doivent correspondre à la sortie du vertex shader
`in vec3 color;`
`in vec2 tex_coord;`
- variables spéciales
`gl_FrontFacing`, `gl_FragCoord` (x,y,z,1/w), ...
- paramètres constants (**uniform**)
comme pour les vertex shaders:
 - variable définie par l'utilisateur
 - textures (**sampler**)

Fragment Shader Output

Possibilité de supprimer un fragment

`discard`

En sortie

- un fragment avec une profondeur (optionnel) et des valeurs (ex. : couleur)

`gl_FragDepth`

`out` `vec4 color_out;`

- valeurs destinées à être écrites dans le *frame buffer* et le *depth buffer* ...
... après les diverses opérations sur les fragments (alpha test, depth test, blending ...)
- on ne peut pas modifier sa position (juste sa profondeur)

Introduction au GLSL

Modèle général de programmation : SIMD

Rappels:

- Exécution parallèle
- Plusieurs vertex (ou fragments) sont traités simultanément par un même programme : **S**ingle **I**nstruction on **M**ultiple **D**ata
 - Flux de données dans un seul sens
 - Pas de variable globale en écriture
 - Pas de sortie en lecture/écriture
 - Traitement conditionnel (if) souvent coûteux.

GLSL : issu du C

Basé sur la syntaxe du C ANSI

fonctionnalités graphiques

un peu de C++

- surcharge des fonctions
- déclaration des variables lorsqu'on en a besoin
- déclaration des *struct*
- type *bool*

qques fonctionnalités

- switch, goto, label
- union, enum, sizeof
- pointeurs, chaîne de caractères

GLSL : tableaux et structures

Tableaux

- comme en C
- Limité aux tableaux 1D

Structures

- comme en C++

Exemple

```
struct MonMateriau
{
    vec3 baseColor;
    float ambient, diffuse, specular;
};
MonMateriau mesMateriaux[12];
```

Types : vecteurs et matrices

Vecteurs:

- float, vec2, vec3, vec4
- int, ivec2, ivec3, ivec4
- bool, bvec2, bvec3, bvec4

Matrices:

- mat2, mat3, mat4
 - matrice carrée de réels (en colonne d'abord)
 - utilisées pour les transformations
- mat2x2, mat2x3, mat2x4
- mat3x2, mat3x3, mat3x4
- mat4x2, mat4x3, mat4x4

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;
}
```


GLSL : intro

- Commentaires comme en C++
- Support des directives de pré-compilation
 - #define, #ifdef, #if, #elif, #else, #endif, #pragma
- Un shader doit avoir une fonction "main"

```
in vec4 vert_position;  
uniform mat4 MVP;  
void main(void)  
{  
    gl_Position = MVP * vert_position;  
}
```

Fonctions prédéfinies

Math

- radians(deg), degrees(rad), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x,y), atan(x_sur_y), pow(x,y), exp(x), log(x), exp2(x), log2(x), sqrt(x), invsersesqrt(x)
- abs(x), sign(x), floor(x), ceil(x), fract(x), mod(x,y), min(x,y), max(x,y), clamp(x, min, max), mix(x, y, t) = interpolation linéaire, step(t, x) = $x < t ? 0 : 1$
- smoothstep(t0, t1, x) = interpolation d'Hermite

Géométrie

- **length**(x), **distance**(x,y), **dot**(x,y), **cross**(x,y), **normalize**(x), reflect(l,N), refract(l,N,eta)

Relation entre vecteurs

- bvec lessThan(x,y) (composante par composante)
lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual
- bool any(bvec x), bool all(bvec x), bvec not(bvec x)

GLSL : "type qualifieurs"

const

- variable ne pouvant être modifiée
- valeur écrites lors de la déclaration

in

- déclare une variable provenant de l'étage précédent

out

- déclare une variable envoyée à l'étage suivant

uniform

- variable constante pour un groupes de primitives
- valeurs définies par le programme hôte

layout()

- contrôle sur le stockage des blocks
`layout(row_major) uniform mat3x4 A;`

in/out/inout

- pour les fonctions

GLSL : 1er exemple

// vertex shader

in vec4 attrib_position;

in vec3 attrib_normal;

out vec3 normal;

uniform mat4 mvp ;

uniform mat3 mat_normal ;

*// modelview * projection*

// pour transformer les normales

void main(void) {

gl_Position = mvp * attrib_position;

normal = normalize(mat_normal * attrib_normal);

}

// fragment shader

in vec3 normal;

out vec3 out_color;

uniform vec3 color;

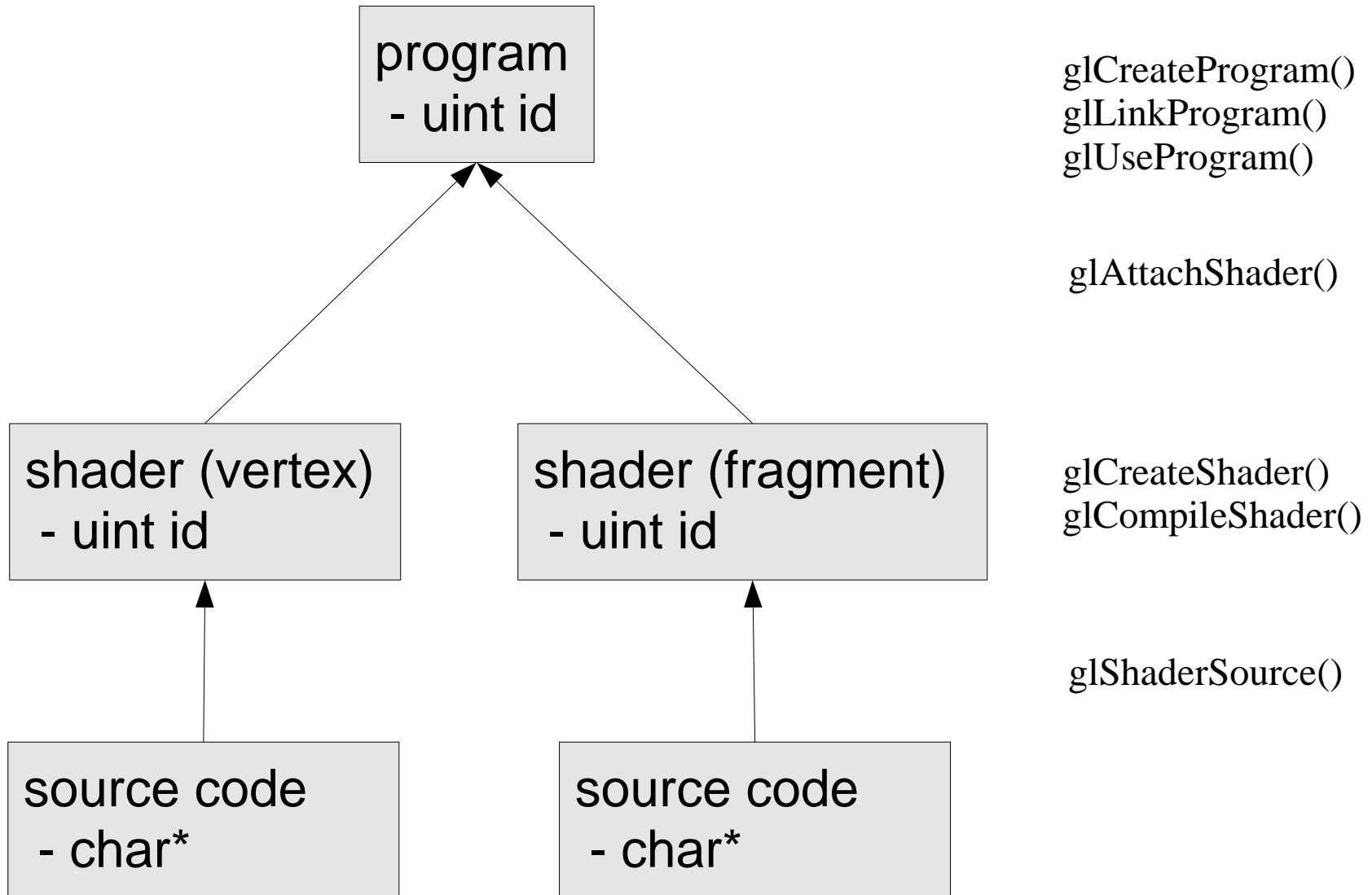
uniform vec3 light_dir;

void main(void) {

out_color = color * max(0,dot(normalize(normal),light_dir));

}

OpenGL: program & shader



API : intro

Pour créer un programme exécuté par le GPU

- créer les shaders et le programme (`glCreateShader()`, `glCreateProgram()`)
- spécifier les sources des shaders à OpenGL (`glShaderSource()`)
- compiler les shaders (`glCompileShader()`)
 - vérifier les erreurs de compilation
- faire l'édition des liens (`glAttachShader()`, `glLinkProgram()`)
 - vérifier les erreurs de "linkage"

Utilisation de l'exécutable

- activer l'exécutable (`glUseProgram()`)
- mäj des *uniforms* et *samplers* (`glUniform*()`)
- spécifier les attributs des sommets
- envoyer la géométrie

Attributs

Attributs génériques

- Lien entre le nom de l'attribut et son numéro,

```
in vec3 normal;  
void main() {...}
```

- 2 méthodes :

- avant `glLinkProgram()`

- *`glBindAttribLocation(programID, 3, "normal")`*
 - force l'édition des liens à utiliser le numéro 3 pour l'attribut *"tangent"*

- après `glLinkProgram()`

- *`int normal_id = glGetAttribLocation(programID, "normal");`*
 - retourne le numéro de l'attribut name, utilisation:
`float normals[3][] ;`
`glVertexAttribPointer(normal_id, 3, GL_FLOAT, false, 0, normals);`
`glEnableVertexAttribArray(normal_id);`

Uniforms

A chaque variable *uniform* correspond un numéro unique

- Récupérer le numéro d'une variable *uniform*
 - *int loc = glGetUniformLocation(programID, GLchar *name)*
 - doit être appelée après l'édition des liens
- Spécifier la valeur d'une variable *uniform*
 - *glUniform{1234}{fi}(uint location, TYPE value)*
 - *glUniform{1234}{fi}v(uint location, int count, TYPE value)*
 - *glUniformMatrix{234}fv(uint location, int count, GLboolean transpose, TYPE value)*
 - *count* = nombre de variables mises à jour (utilisé pour les tableaux)
 - *transpose* = la/les matrices doivent elles être transposées ?

— Exemple:

// shader code:

uniform mat4 mvp;

// host code:

float mat[16] = {...};

glUniformMatrix4fv(glGetUniformLocation(program_id, "mvp"), 1, false, mat);

Références

Cours de Pierre Benard et Gael Guennebaud

- <http://www.labri.fr/perso/pbenard/teaching/mondes3d>

Ressources

OpenGL:

- <https://www.opengl.org/>
- <https://www.khronos.org/registry/OpenGL-Refpages/gl4/>
- <http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Table-of-Contents.html>

Articles scientifiques

- <http://kesen.realtimerendering.com/>

Tutoriels OpenGL complets :

- <https://learnopengl.com/>
- <http://www.opengl-tutorial.org/fr/>