

# PROGRAMMATION OPENGL

- OPENGL 4 - LES BASES -

Jonathan Fabrizio

<http://jo.fabrizio.free.fr>

Version : Wed Feb 12 09:24:39 2025

Introduction

Principe général

Le langage GLSL

OpenGL

Exemple

## Introduction

GL (1992) développée initialement par SGI puis maintenant Khronos Group

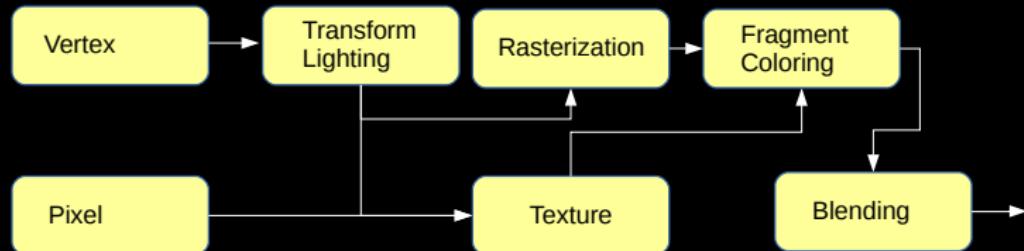
- Permet le rendu d'images par rasterisation
- OpenGL est une spécification
- La version N-1 de GL est libre (OpenGL)...
- Une implémentation libre : MESA

Ce qu'OpenGL ne fait pas :

- Ne gère pas l'interaction avec le système/l'utilisateur
- Ne traite pas les ombres
- Repose sur le modèle de Lambert ou de Gouraud mais pas le modèle de Phong
  - A partir de la version 3, il n'y a même plus aucun modèle
- N'a aucun modèle physique
- N'a aucun détecteur de collision
- Ne fait pas le café (mais peut quand même vous proposer une théière)



## Pipeline graphique fixe (OpenGL <2.0)



```
glBegin();  
    glVertex();  
    glNormal();  
    glColor();  
    glEnd();  
  
    glGenLists();  
    glNewList();  
    glEndList();  
    glCallList();  
  
glMatrixMode();  
glPushMatrix();  
glTranslated();  
glFrustum();  
  
glShadeModel();  
 glEnable(GL_LIGHT0);  
 glEnable(GL_LIGHT1);
```

```
glBegin();  
    glVertex();  
    glNormal();  
    glColor();  
    glEnd();  
  
glMatrixMode();  
glPushMatrix();  
glTranslated();  
glFrustum();
```

*Deprecated*

```
glGenLists();  
glNewList();  
glEndList();  
glCallList();
```

```
glShadeModel();  
glEnable(GL_LIGHT0);  
glEnable(GL_LIGHT1);
```

## OpenGL : Les temps anciens

```
glBegin();  
    glVertex();  
    glNormal();  
    glColor();  
    glEnd();  
  
glMatrixMode();  
glPushMatrix();  
glTranslated();  
glFrustum();
```

*Deprecated*

```
glGenLists();  
glNewList();  
glEndList();  
glCallList();
```

*Deprecated*

```
glShadeModel();  
glEnable(GL_LIGHT0);  
glEnable(GL_LIGHT1);
```

## OpenGL : Les temps anciens

```
glBegin();  
    glVertex();  
    glNormal();  
    glColor();  
    glEnd();  
  
glMatrixMode();  
glPushMatrix();  
glTranslated();  
glFrustum();
```

glGenLists();  
glNewList();  
glEndList();  
glCallList();

glShadeModel();  
glEnable(GL\_LIGHTING);  
glEnable(GL\_LIGHT0);

```
glBegin();  
    glVertex();  
    glNormal();  
    glColor();  
    glEnd();  
  
glMatrixMode();  
glPushMatrix();  
glTranslated();  
glFrustum();
```

Deprecated!

Deprecated!

Deprecated!

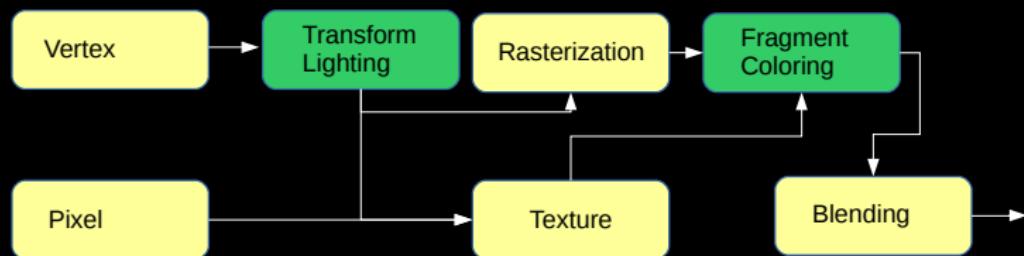
```
glGenLists();  
glNewList();  
glEndList();  
glCallList();
```

```
glShadeModel();  
glEnable(GL_LIGHT0);  
glEnable(GL_LIGHT1);
```

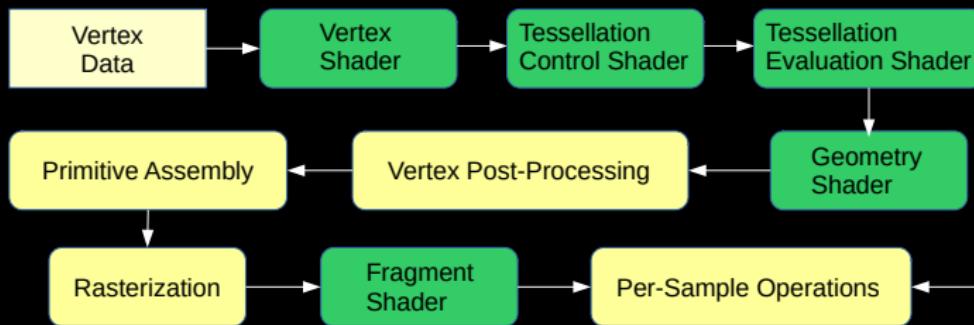
Deprecated!

Deprecated!

Pipeline fixe toujours disponible mais introduction des vertex shader et fragment shader



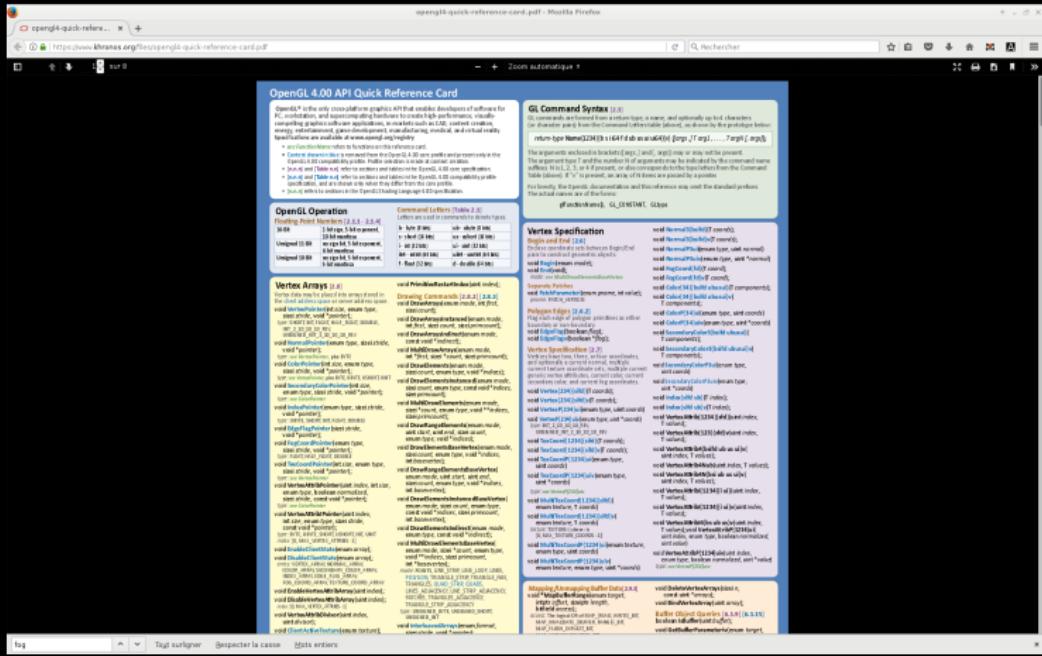
## Exclusively programmable pipeline (`OpenGL >= 3.1`)



- Vertex shader
- Pixel shader
- Almost all data is GPU-resident

- OpenGL 3.2
  - Geometry shader (Permet la modification/génération de formes)
- OpenGL 4.1
  - Tessellation-control shader
  - Tessellation-evaluation shader
- OpenGL 4.3/4.5
  - Compute shader

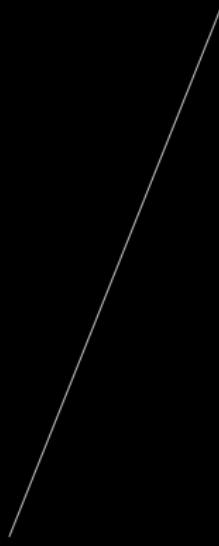
## OpenGL : Respect de la norme actuelle



source : www.khronos.org

# Structure

CPU  
RAM



GPU  
Shaders (GLSL)  
VRAM

## Principe général

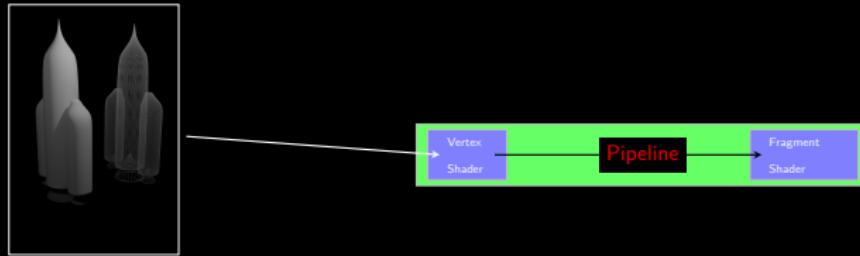
En simplifié :



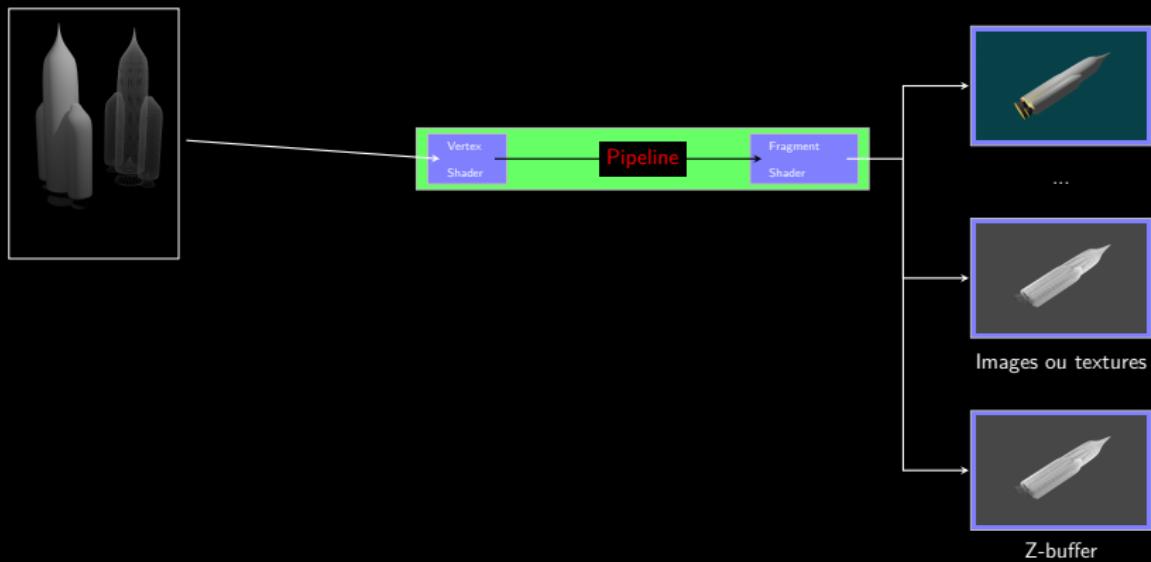
En simplifié :



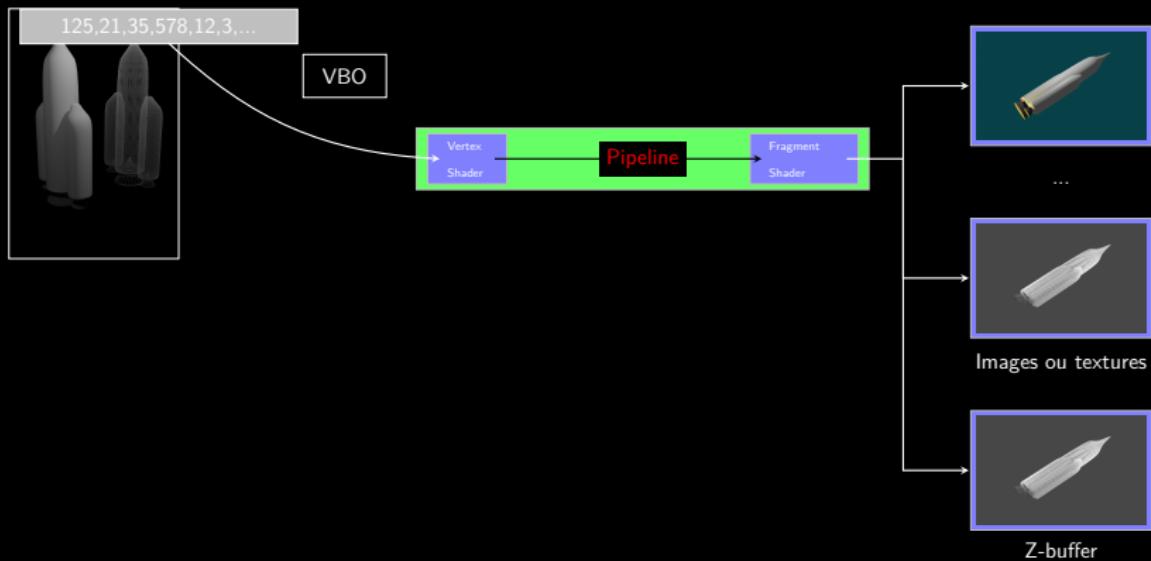
En simplifié :



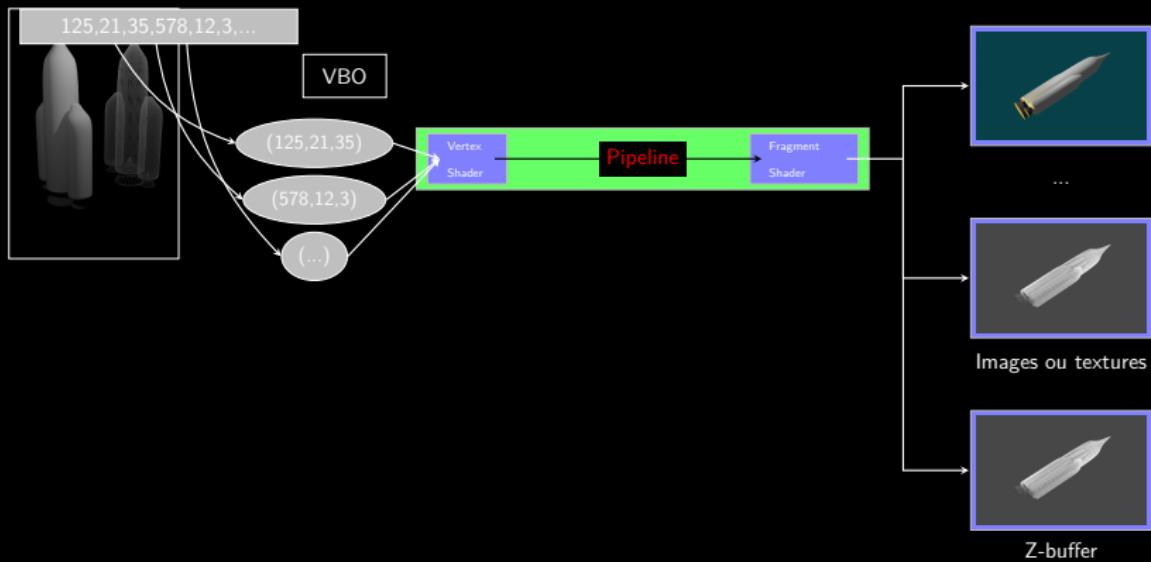
En simplifié :



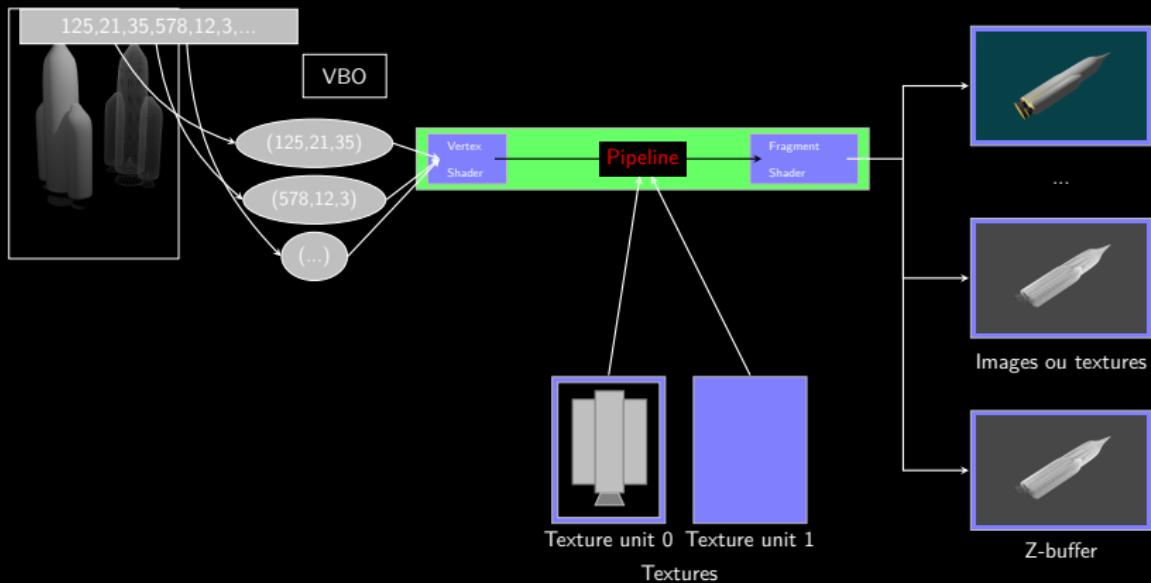
En simplifié :



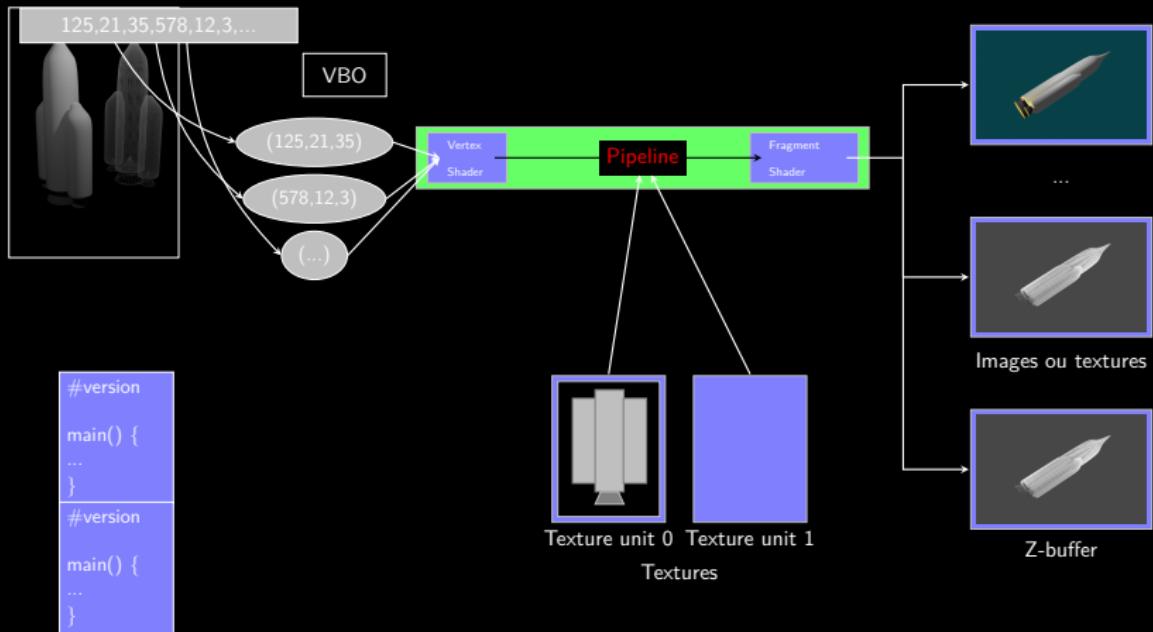
En simplifié :



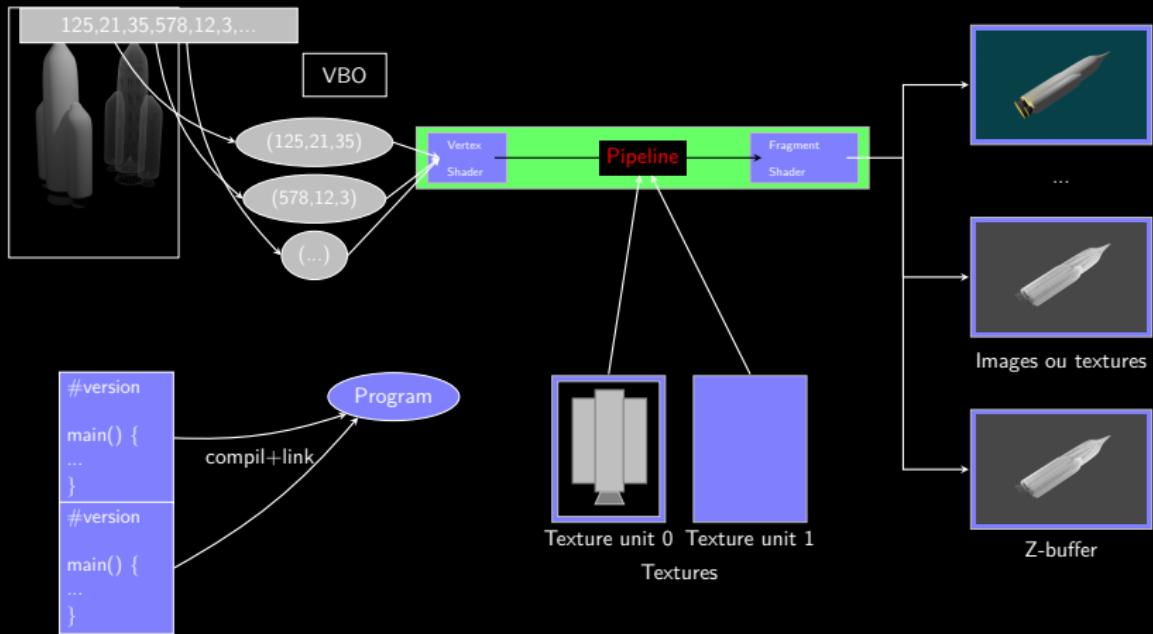
En simplifié :



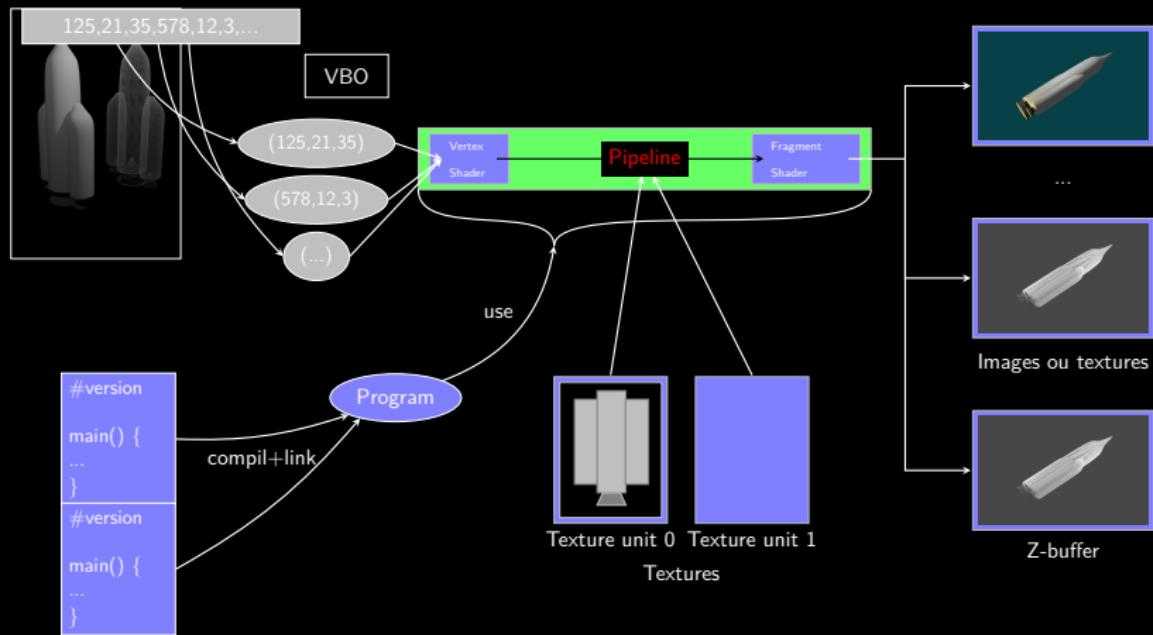
En simplifié :



En simplifié :



En simplifié :



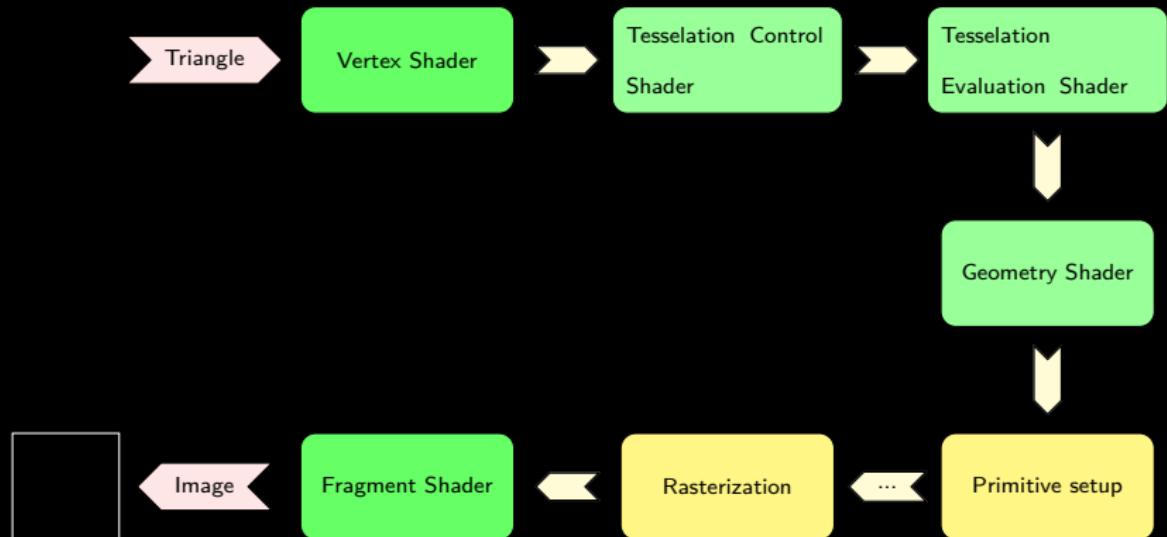
En simplifié :

1. faire les différentes initialisations,
2. compiler les *shaders*,
3. initialiser les données,
4. activer le bon programme (*shaders*) puis envoyer les données vers le pipeline graphique,
5. récupérer l'image

1. Faire les différentes initialisations OpenGL gère pas mal de choses qu'il faut initialiser/activer : *Z-buffer, Back face culling...*

## 2. Compiler les *shaders* et lier les programmes

Les *shaders* sont des programmes qui s'exécutent directement dans le pipeline graphique, ils sont liés dans un *program*.



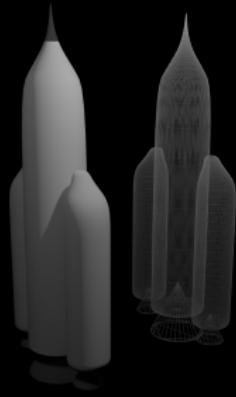
Ils sont écrits en *GLSL*.

## 3. Initialiser les données

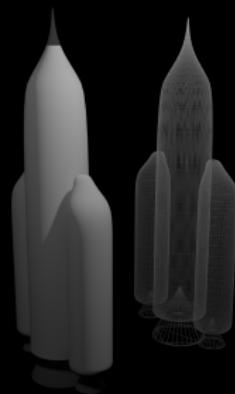
## 3. Initialiser les données



## 3. Initialiser les données



## 3. Initialiser les données



Préparer les données du maillage dans un buffer (le VBO<sup>a</sup>).

- sommets
- couleurs
- coordonnées textures
- normales
- ...

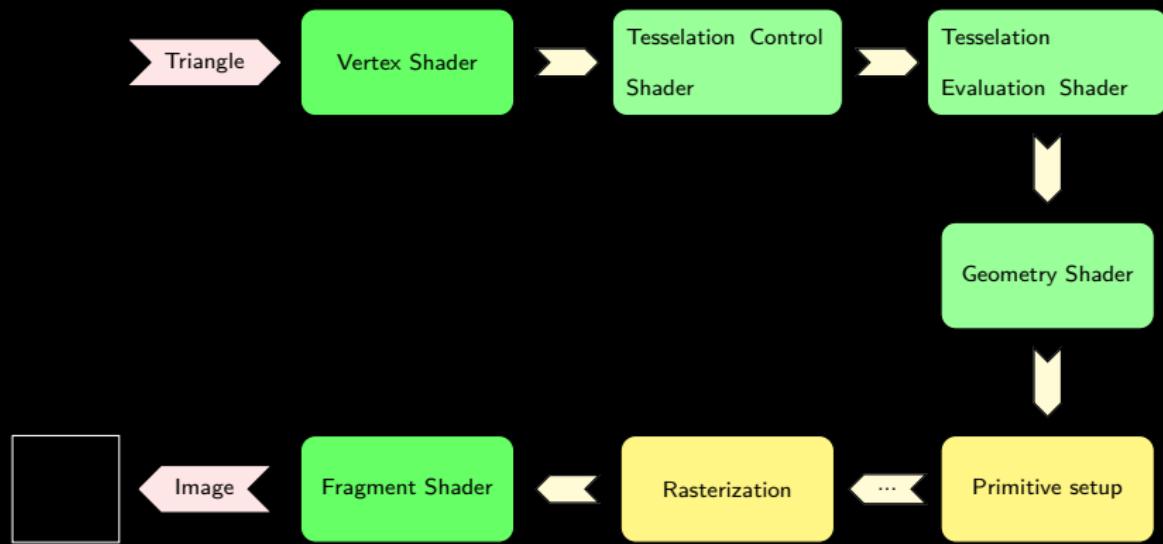
Expliquer comment les données sont organisées/découpées dans le buffer

---

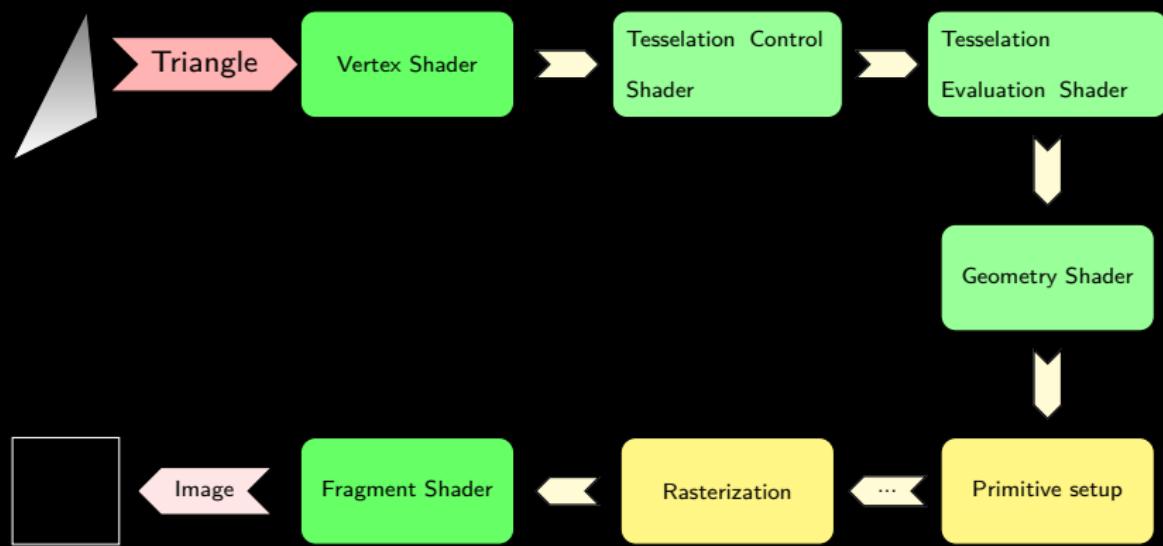
a. Vertex Buffer Object

4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique

4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :  
Activation du bon *program*.

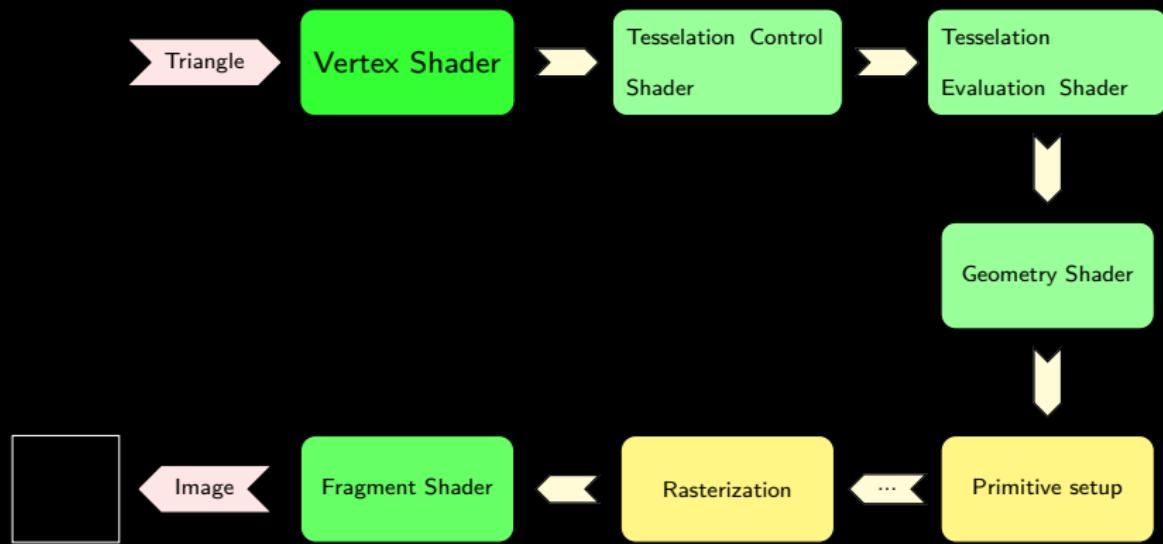


4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :  
Envoie des données du maillage

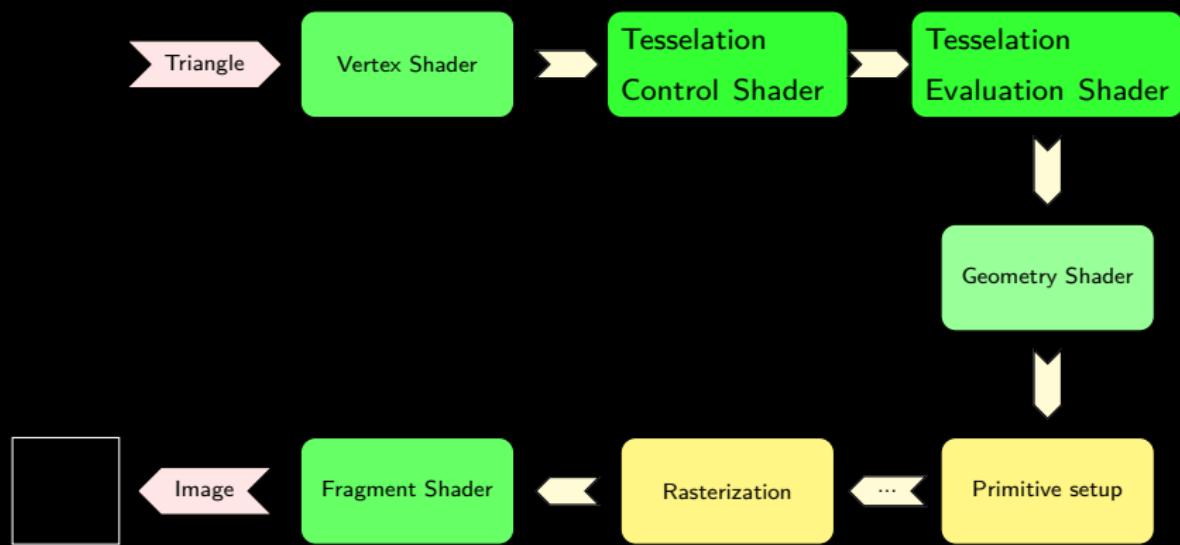


4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :

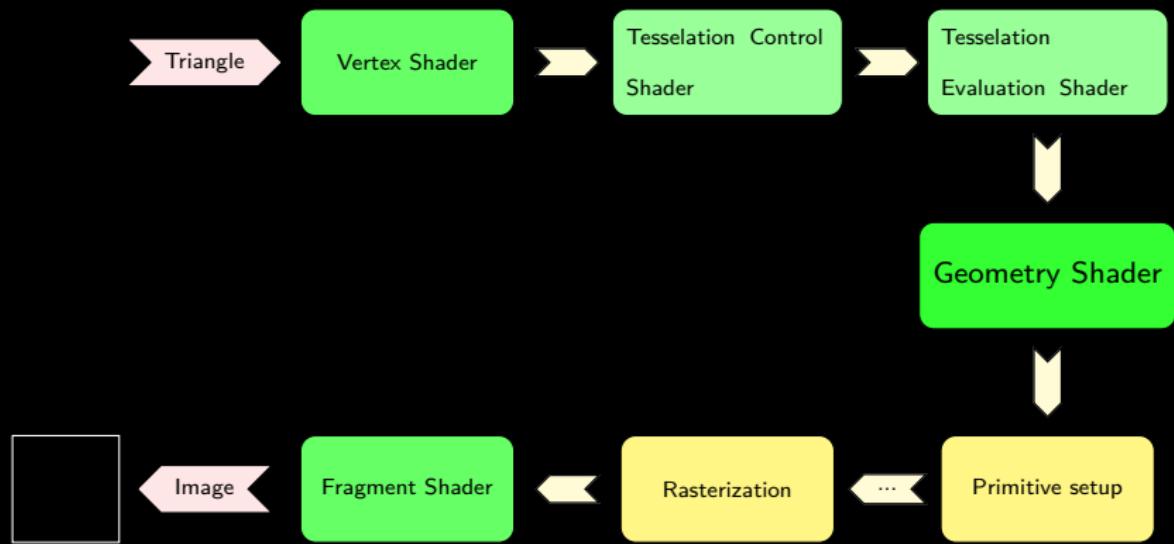
*Vertex Shader* - Changement de repère pour préparer la projection



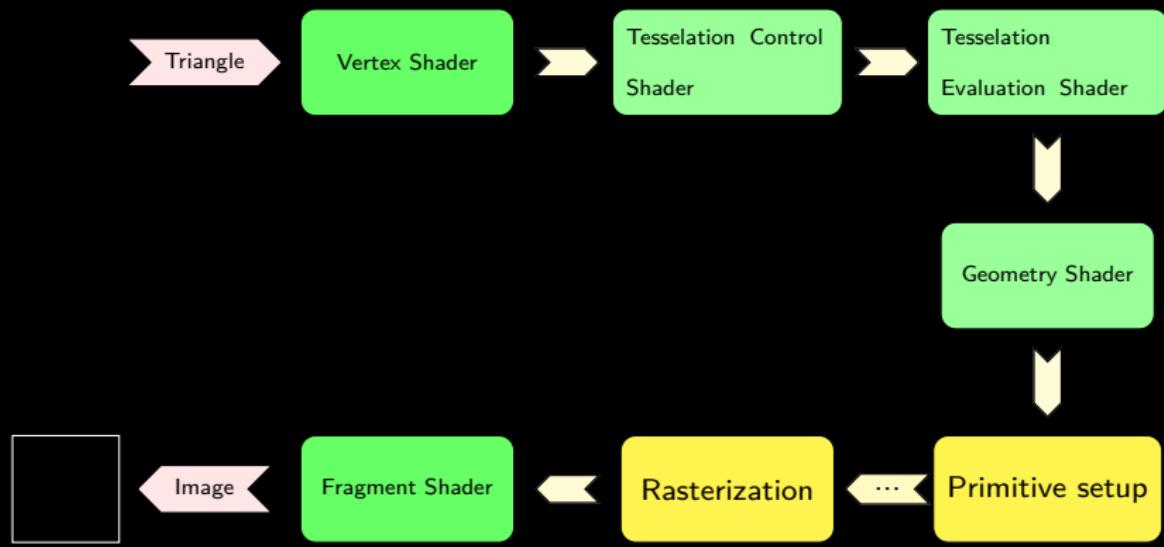
4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :
- Tesselation Shaders - Enrichir le maillage* (Optionnel)



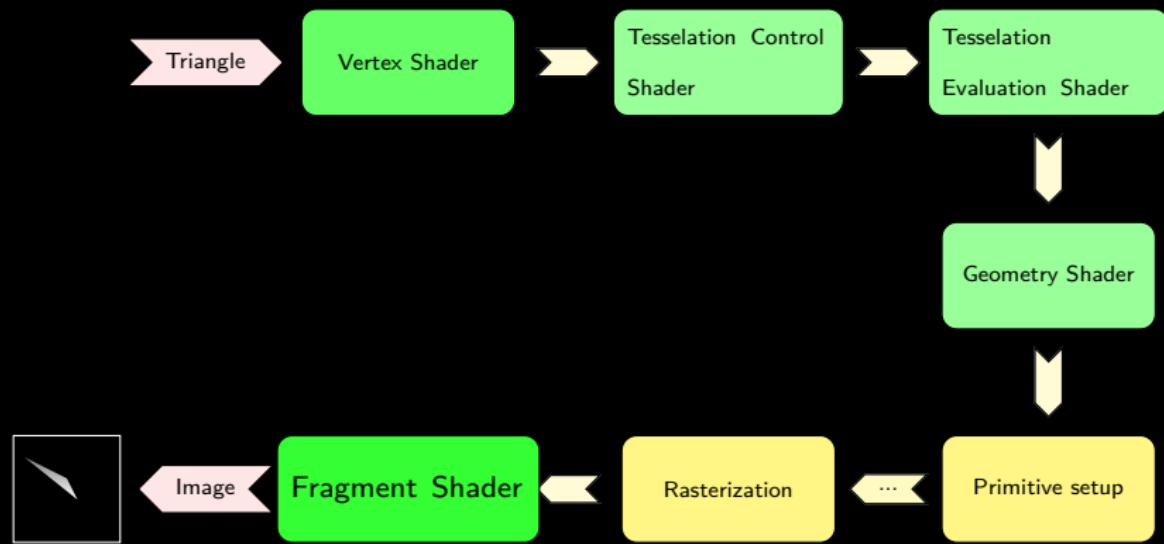
4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :  
*Geometry Shader - Changer la nature/enrichir les primitives* (Optionnel)



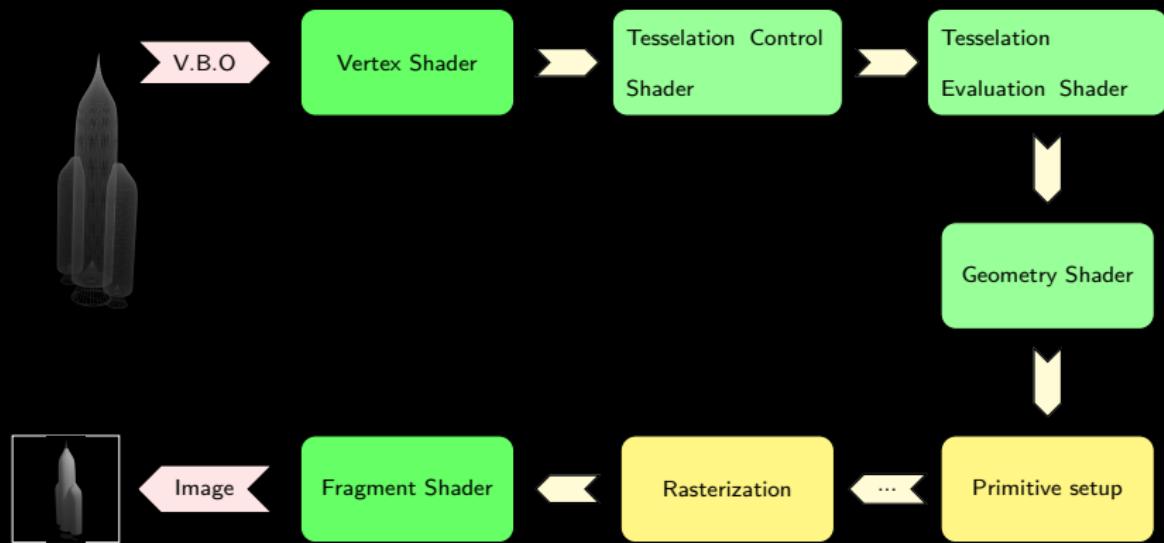
4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :  
*Primitive setup/Rasterization* - Préparer le dessin.



4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :  
*Fragment Shader* - Dessiner un fragment.

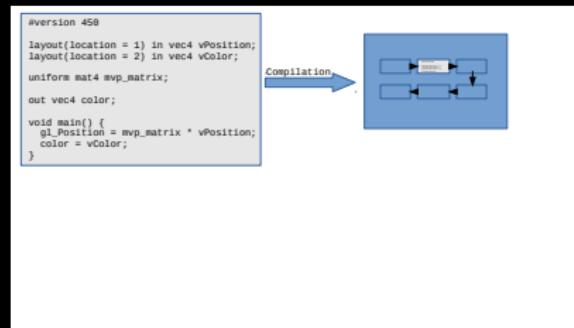


4. Activer le bon programme (*shaders*) puis envoyer les données (*buffers*) vers le pipeline graphique :  
Récupérer l'image.



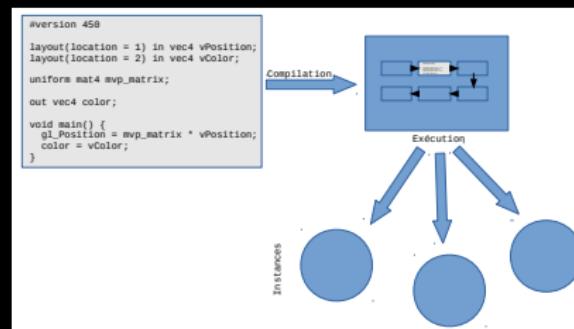
GLSL : Langage utilisé pour écrire les shaders.

- Compilé pour la carte video.



## Execution des shaders

- Single Instruction Multiple Instances



## Execution des shaders

Plusieurs threads qui font exactement la même chose !

```
if (...) {           if (...) {           if (...) {           if (...) {           if (...) {  
    ...                   ...                   ...                   ...                   ...  
    a=a*c; ⊗       a=a*c; ←     a=a*c; ⊗       a=a*c; ←     a=a*c; ←  
    ...                   ...                   ...                   ...                   ...  
}           }           }           }           }
```

## Présentation du langage

## Principaux types

- scalaires (limités !) : bool, int, uint, float, double
- vecteurs : bvecn, ivecн, uvecn, vecn, dvecn ( $n=2..4$ )
- matrices : matn, matnb, dmatn, dmatnm ( $n/m=2..4$ )
- samplers/images

## Acces

- vecteurs : ivec4 t ; t[2]/t.r/t.rgb/t.rgb/a/t.xy/t.xyz...
- operateurs : multiplication matricielle...

## Structures

- Arrays : vec3[5][2] multidim ;

- Structures :

```
struct Light  
{  
    vec3 eyePosOrDir ;  
    bool isDirectional ;  
} variableName ;
```

- flux : if/switch
- boucles : for/while/do while
- fonctions : int fun(int i)
- prepro : #define...
- et beaucoup de fonctions (clamp...)

- Partagées entre toutes les instances (Uniform)
  - variables *uniform*
  - UBO
  - SSBO
  - Textures (sampler)
  - Images
- Spécifiques à chaque instance (Vertex Shader)
  - VBO
- Sorties
  - FBO
- Communication avec et entre shaders
  - in/out
  - shared

- Partagées entre toutes les instances (Uniform)
  - variables uniform
  - UBO
  - SSBO
  - Textures (sampler)
  - Images
- Spécifiques à chaque instance (Vertex Shader)
  - VBO
- Sorties
  - FBO
- Communication avec et entre shaders
  - in/out
  - shared

# Uniform

- Partagée entre toutes les instances
- Read-only coté GLSL

Coté CPU

Il faut connaître ou récupérer  
(`glGetUniformLocation()`) l'adresse de  
la variable

Puis faire l'assignation :

```
glUniform*( location , value );
```

Coté GPU

Déclaration de la variable :

```
uniform int v;  
layout(location = 1) uniform float t;
```

## Bloc de mémoire

- Déclaration :

```
GLuint buffer_id;  
glGenBuffers(1, &buffer_id);
```

- Activation/désactivation :

```
glBindBuffer(--TYPE--, buffer_id);  
glBindBuffer(--TYPE--, 0);
```

- Allocation :

```
glBufferData(...);
```

- Ecriture/modification :

```
glBufferData(...)  
glMapBuffer(...)/glUnmapBuffer(...)
```

- Destruction :

```
glDeleteBuffers(1, &buffer_id);
```

## Buffers

Brique de base des FBOs, UBOs, TextureBuffer...

### Utilisation :

- Regrouper plusieurs *uniforms* en un seul bloc
- Utilisation dans différents programmes et mise à jour indépendamment du programme.

### Limitations :

- Quelques dizaines de ko par bloc (`GL_MAX_UNIFORM_BLOCK_SIZE`).
- Nombre limité de buffers actifs par type de shaders (`GL_MAX_??_UNIFORM_BLOCKS`).
- Nombre total d'uniform buffer (`GL_MAX_UNIFORM_BUFFER_BINDINGS`).
- En lecture seule côté Shader.
- Restreint aux types possibles dans GLSL.
- Taille fixée a priori.

# UBO : Uniform Buffer Object

Exemple :

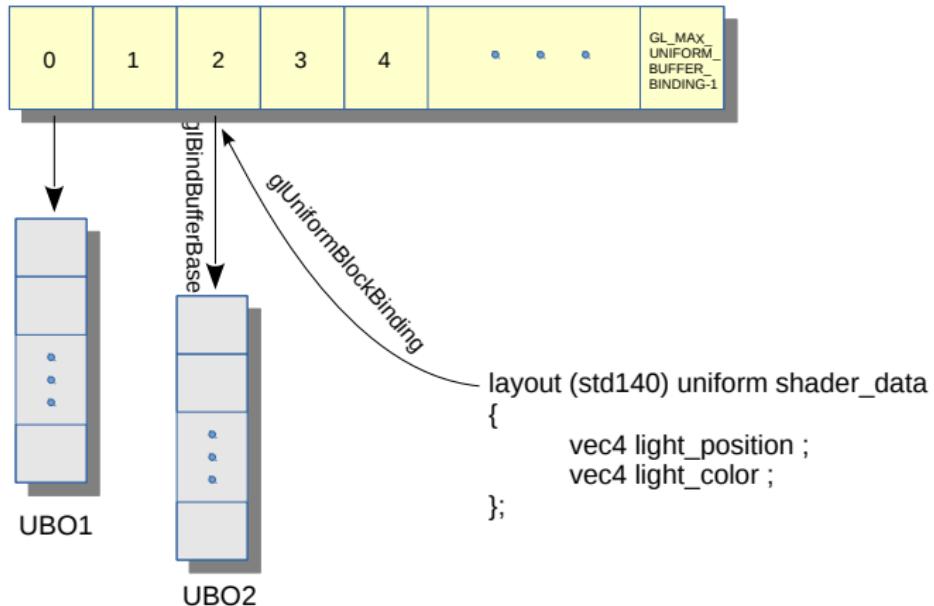
- On peut remplacer :

```
uniform vec4 light_position;  
uniform vec4 light_color;
```

par :

```
layout (std140) uniform shader_data  
{  
    vec4 light_position;  
    vec4 light_color;  
};
```

# UBO : Uniform Buffer Object



# UBO : Uniform Buffer Object

Création et activation :

- `GLuint buffer_id ;  
glGenBuffers(1, &buffer_id) ;  
 glBindBuffer(GL_UNIFORM_BUFFER, buffer_id) ;`

Bindind :

- `glBindBufferBase(GL_UNIFORM_BUFFER, binding_point_index, buffer_id) ;`

binding\_point\_index :

- soit fixé dans le code du shader
- soit fixé par `glUniformBlockBinding(program, uniform_bloc_index  
(glGetUniformBlockIndex), binding_point_index) ;`

Manipulation :

- Comme les autres buffers.

## SSBO : Shader Storage Buffer Objects

Utilisation :

- Gros blocs de données
- Accès en lecture et/ou écriture
- Taille pas nécessairement fixée dans le shader.

Limitations :

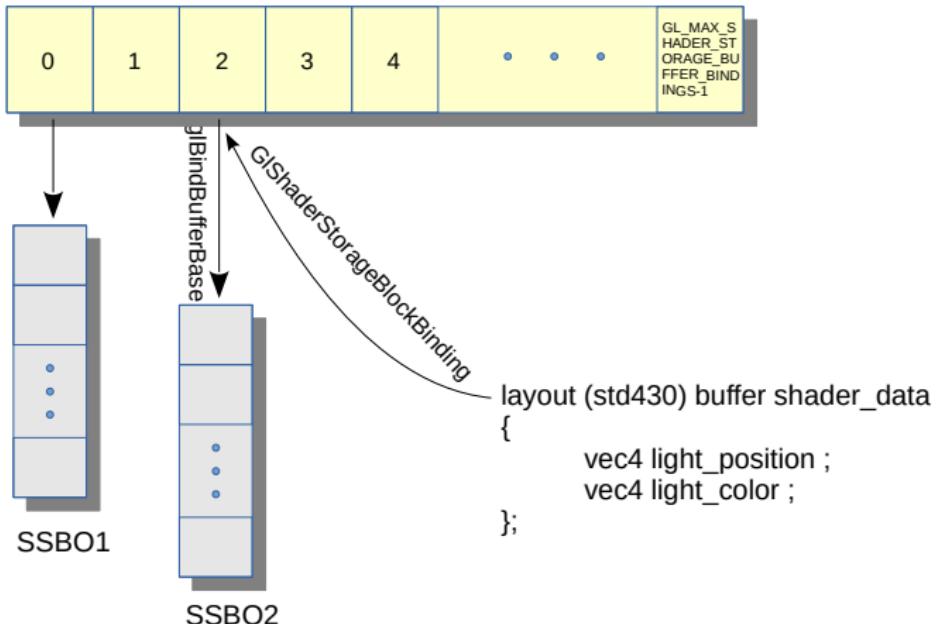
- Seulement depuis OpenGL 4.3
- Nombre maximum de SSBOs  
(`GL_MAX_SHADER_STORAGE_BUFFER_BINDINGS`)
- Limite de taille (16Mo garanti, en pratique pas -> taille de la mémoire libre) (`GL_MAX_SHADER_STORAGE_BLOCK_SIZE`)
- Nombre limité de buffers actifs par type de shaders  
(`L_MAX_???_SHADER_STORAGE_BLOCKS`)  
+ `GL_MAX_COMBINED_SHADER_STORAGE_BLOCKS`
- Restreint aux types possibles dans GLSL.
- En théorie un peu plus lent que les UBOs.

# SSBO : Shader Storage Buffer Objects

Définition dans le shader :

```
layout (std430, binding = 1) buffer shader_data
{
    vec4 light_position;
    vec4 light_color;
};
```

# SSBO : Shader Storage Buffer Objects



# SSBO : Shader Storage Buffer Objects

Création et activation :

- ```
GLuint ssbo_id;
glGenBuffers(1, &ssbo_id);
 glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo_id);
```

Binding :

- ```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, binding_point_index,
 ssbo_id);
```

binding\_point\_index :

- soit fixé dans le code du shader `binding = xx`
- soit fixé par `glShaderStorageBlockBinding(program, storage_bloc_index  
(glGetProgramResourceIndex), binding_point_index);`

Manipulation :

- Comme les autres buffers.

Attention à l'alignement des données coté shader !!! On connaît les problèmes de padding et d'alignement coté CPU, il y a aussi des pbs coté GPU !!!

### Coté GPU

#### Coté CPU

```
struct Line {  
    GLfloat old_pos[4];  
    GLfloat old_color[4];  
    GLfloat new_pos[4];  
    GLfloat new_color[4];  
};
```

```
struct Line {  
    vec4 old_pos;  
    vec4 old_color;  
    vec4 new_pos;  
    vec4 new_color;  
};  
layout(std430, binding = 2) buffer line_buffer  
{  
    Line line_list[NB_PARTICLES];  
};
```

*Padding coté GPU :*

Coté CPU

```
struct line {  
    GLfloat pos[3];  
    GLfloat color[3];  
    GLfloat prop1;  
    GLfloat prop2  
};
```

Coté GPU

```
struct Line {  
    vec3 pos;  
    /* Padding 1 byte */  
    vec3 color;  
    float new_pos;  
    float new_color;  
};
```

Les deux structures ne correspondent plus !

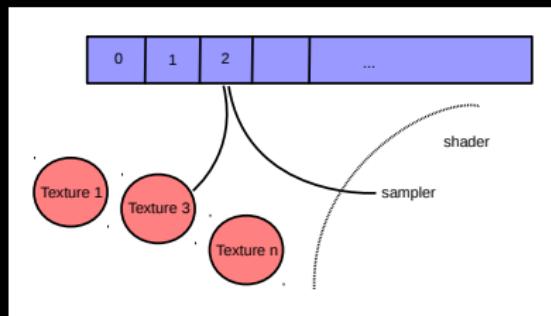
## SSBO et UBO : Alignement des données

```
struct line {
    GLfloat pos[3];
    GLfloat prop1;
    GLfloat color[3];
    GLfloat prop2
};
```

```
struct Line {  
    vec3 pos;  
    float prop1;  
    vec3 color;  
    float prop2;  
};
```

Les deux structures correspondent : Toujours bien réfléchir à l'ordre de champs.

- Déclaration
  - `glGenTextures(1, &id);`
- Activer/Descativer
  - `glBindTexture (...);`
- Allocation
  - `glTexStorage2D(); // Best`
  - `glTexImage2D(); // Declaration of the bitmap.`
- Remplissage
  - `glTexImage2D();`
  - `glTexSubImage2D();`
  - Eventuellement texture buffers : `glTexBuffer (...);`
- Destruction
  - `glDeleteTextures (...);`



- Pour utiliser une texture, il faut d'abord l'activer (`glBindTexture()`) sur un *texture unit* (`glActivateTexture()`).
- Il faut indiquer au *sampler* du *shader* sur quel *texture unit* il doit travailler

# Textures

```
GLint tex1_loc = glGetUniformLocation(prog, "tex1_sampler");
glUniform1i(tex1_loc, 0);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, id_tex1);
```

```
GLint tex2_loc = glGetUniformLocation(prog, "tex2_sampler");
glUniform1i(tex1_loc, 1);
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, id_tex2);
```

---

```
uniform sampler2D tex1_sampler;
uniform sampler2D tex2_sampler;

...
vec4 texel = texture(tex1_sampler, interpolated_uv_position)
    +texture(tex2_sampler, interpolated_uv_position);
...
```

## Attention :

- Pour une texture 2D, l'origine est en bas à gauche !
- Contrairement à un buffer, le premier bind determine le type de la texture (`GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP...`). Apres il n'est plus possible d'en changer.
- Pour activer un *texture unit* : `glActiveTexture(GL_TEXTURE0 + i)`; plutot que `glActiveTexture(GL_TEXTUREi)`; car pas assez de constantes (jusqu'à `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`)

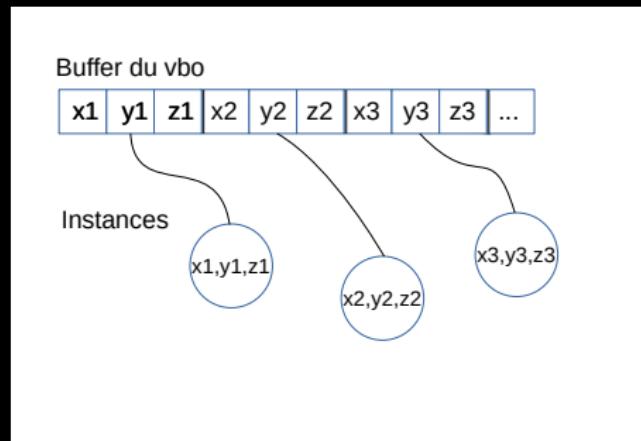
# Images

```
glBindImageTexture(image_unit_in_id0, texture_in_id0, 0, GL_FALSE, 0,  
                    GL_READ_ONLY, GL_RGBA8UI);  
glBindImageTexture(image_unit_out_id1, texture_out_id, 0, GL_FALSE, 0,  
                    GL_WRITE_ONLY, GL_RGBA8UI);
```

---

```
#version 430 core  
  
layout (local_size_x = 16, local_size_y = 16) in;  
  
layout (rgba8ui, binding = 0) readonly uniform uimage2D input_image;  
layout (rgba8ui, binding = 1) writeonly uniform uimage2D output_image;  
  
void main(void) {  
    ivec2 pos = ivec2(gl_GlobalInvocationID.xy);  
    uvec4 result = imageLoad(input_image, pos)/2;  
    imageStore(output_image, pos, result);  
}
```

- Déclarer/détruire : `glGenBuffers()`/`glDeleteBuffers()`
- Activer : `glBindBuffer(GL_ARRAY_BUFFER, ...)`
- Envoyer les données : `glBufferData()`/`glMapBuffer()`
- Lier avec le shader : `glVertexAttribPointer()`
- Activer l'association/l'envoie de données :  
`glEnableVertexAttribArray()`
- Encapsuler dans *Vertex Array Object* - VAO ! (**Obligatoire**)



# VBO

```
glVertexAttribPointer(location, nb_comp, type, normalize, stride, offset);  
  
xyzxyzxyzxyz...xyz  
rgbgrgbgrgb...rgb  
ststststst...st  
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);% //xyz
```

stride = 0 => données consécutives  
Dans le vertex shader :

```
layout(location = 0) in vec3 position;
```

```
glVertexAttribPointer(location, nb_comp, type, normalize, stride, offset);
```

xyzxyzxyzxyz...xyzrgbgrgbgrgb...rgbststststst...st

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                     0, (void*)0);%//xyz
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
                     0, (void*)3*nb_vertices*sizeof(GLfloat));//rgb
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
                     0, (void*)2*3*nb_vertices*sizeof(GLfloat));//st
```

xyzrgbstxyzrgbst...xyzrgbst

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                     8*sizeof(GLfloat), (void*)0);%//xyz
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
                     8*sizeof(GLfloat), (void*)3*sizeof(GLfloat));//rgb
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
                     8*sizeof(GLfloat), (void*)6*sizeof(GLfloat));//st
```

Dans le vertex shader :

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 color;
layout(location = 2) in vec3 uv;
```

Note : Attention, glVertexAttribPointer (...) != glVertexAttribPointer (... , GL\_INT, ...)

## Vertex Array Object

Encapsule les VBOs (**Obligatoire en OpenGL 4 !!!**)

```
glGenVertexArrays(...);  
 glBindVertexArray(...);  
 glEnableVertexAttribArray(...);  
 glBindVertexArray(...);
```

glDeleteVertexArrays(...)

VAO 0 -> static VAO ; valeur par défaut pour `glDisableVertexAttribArray ()`;

```
glBindVertexArray(vao_id);  
glDraw ...  
glBindVertexArray(0);
```

Certaines questions à se poser :

- Un unique VBO ou plusieurs pour une même forme ?
- Données entrelacées ou par bloc ?
- glBufferData()/glMapBuffer() ?

Best practices

- Uniformiser le plus possible le format des données
- Minimiser au mieux la taille des données
- Utiliser lorsque c'est possible un adressage indexé pour gagner de la place

Les VBOs peuvent être remplis :

- Coté CPU comme un buffer normal (`GL_ARRAY_BUFFER`)
- Coté GPU - à l'aide d'un compute shader par exemple (`GL_SHADER_STORAGE_BUFFER`).

Pour cela il est possible de faire

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ...)  
glBindBuffer(GL_ARRAY_BUFFER, ...)
```

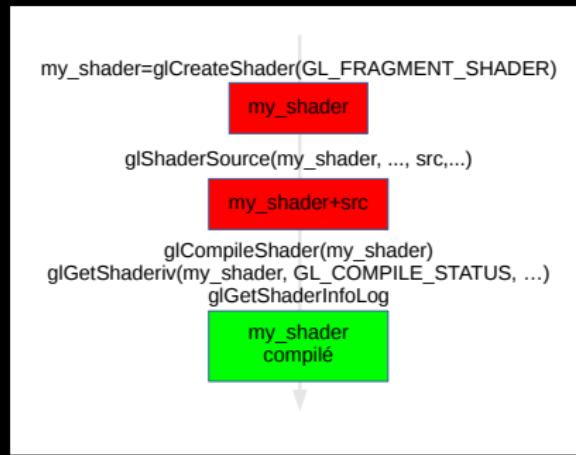
sur le même buffer.

## Shaders - Communications avec et entre shaders

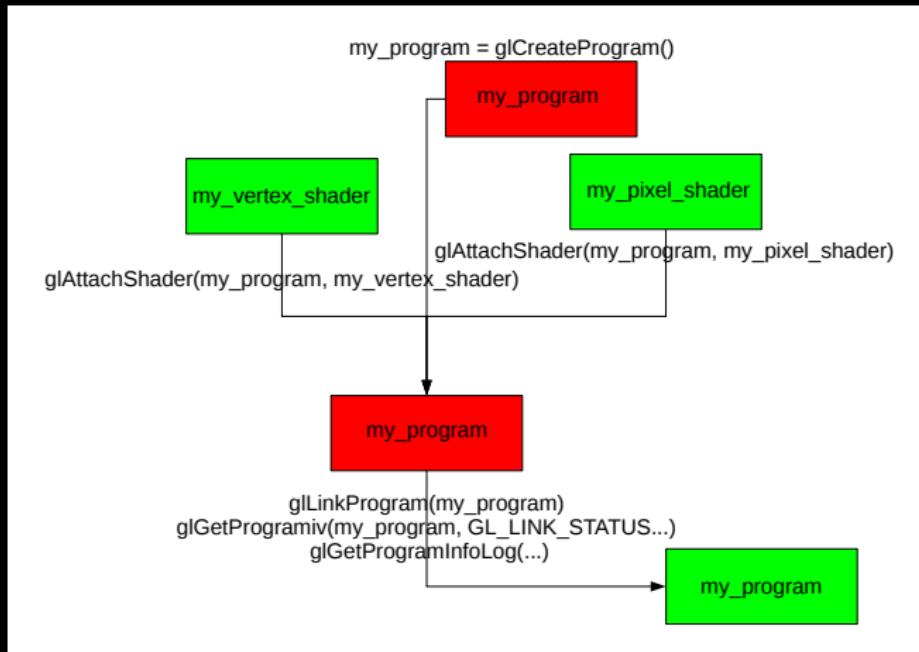
- in/out
- shared

- Partagées entre toutes les instances (Uniform)
  - variables *uniform*
  - UBO
  - SSBO
  - Textures (sampler)
  - Images
- Spécifiques à chaque instance (Vertex Shader)
  - VBO
- Sorties
  - FBO
- Communication avec et entre shaders
  - in/out
  - shared

## Shaders : compilation

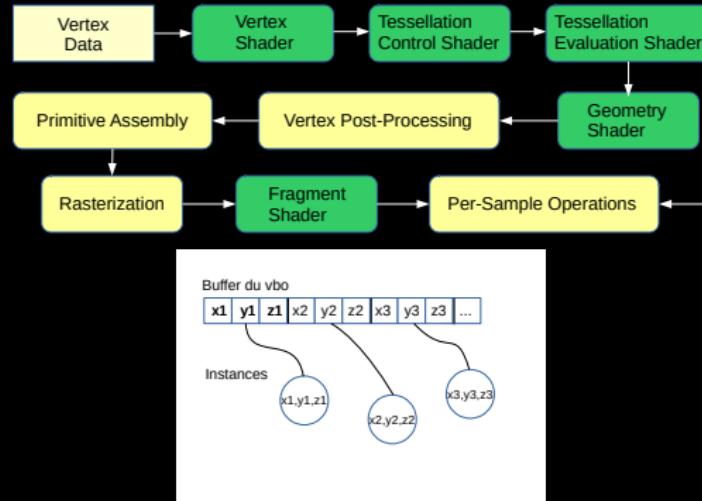


## Shaders : Edition de liens



Apres l'édition de lien on peut appeler `glDetachShader()` et  
`glDeleteShader()`  
Activation : `glUseProgram(...)`

# Vertex Shader



- Vertex shader en entrée :
  - gl\_Vertex : position du sommet
  - gl\_Color : couleur du sommet
  - gl\_Normal : normale du sommet
  - gl\_MultiTexCoordn : Coordonnées de la texture n
  - gl\_SecondaryColor
  - gl\_FogCoord

- Vertex shader en entrée :
  - gl\_Vertex : position du sommet
  - gl\_Couleur du sommet *Deprecated*
  - gl\_Normal : normale du sommet
  - gl\_MultiTexCoordn : Coordonnées de la texture n
  - gl\_SecondaryColor
  - gl\_FogCoord

- Vertex shader en entrée :
  - gl\_Vertex : position du sommet
  - gl\_Color *Deprecated* : couleur du sommet
  - gl\_Normal : normale du sommet
  - gl\_MultiTexCoordn : Coordonnées de la texture n
  - gl\_SecondaryColor *Deprecated* :
  - gl\_FogCoord

- Vertex shader en entrée :
  - Seulement ce qui est déclaré **in**

- Vertex shader en sortie :
  - gl\_Position : position (*the clip-space output position of the current vertex*) du sommet
  - gl\_PointSize
  - gl\_ClipDistance[]
- gl\_FrontColor/gl\_BackColor : couleur du sommet
- SecondaryColor...
- gl\_TexCoordn : Coordonnées de la texture n
- gl\_FogFragCoord
- gl\_ClipVertex

- Vertex shader en sortie :
  - gl\_Position : position (*the clip-space output position of the current vertex*) du sommet
  - gl\_PointSize
  - gl\_ClipDistance[]
- gl\_FrontColor/gl\_BackColor : couleur du sommet
- SecondaryColor...
- gl\_TexCoordn : coordonnées de la texture n  
*Deprecated*
- gl\_FogFragCoord
- gl\_ClipVertex

# Vertex Shader

## Utilisation de in/out

Un exemple :

```
#version 450

layout(location = 1) in vec4 vPosition;
layout(location = 2) in vec4 vColor;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;

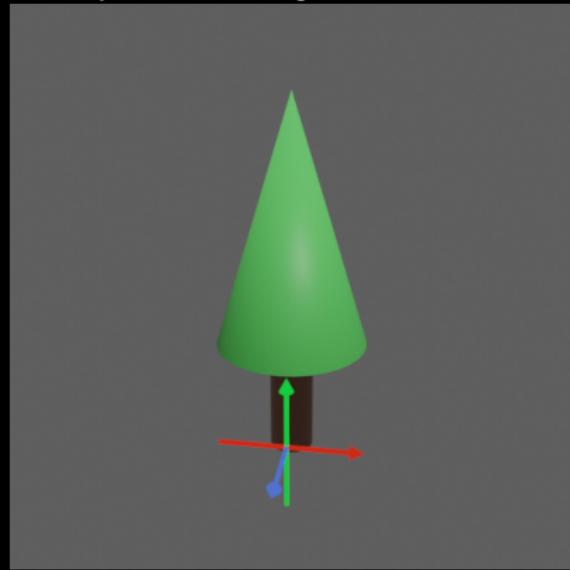
out vec4 color;

void main() {
    gl_Position = projection_matrix * model_view_matrix * vPosition;
    color = vColor;
}
```

Nous devons fixer les coordonnées des sommets dans le *clip-space*. On peut décomposer de la manière suivante (mais ce n'est pas obligé) :

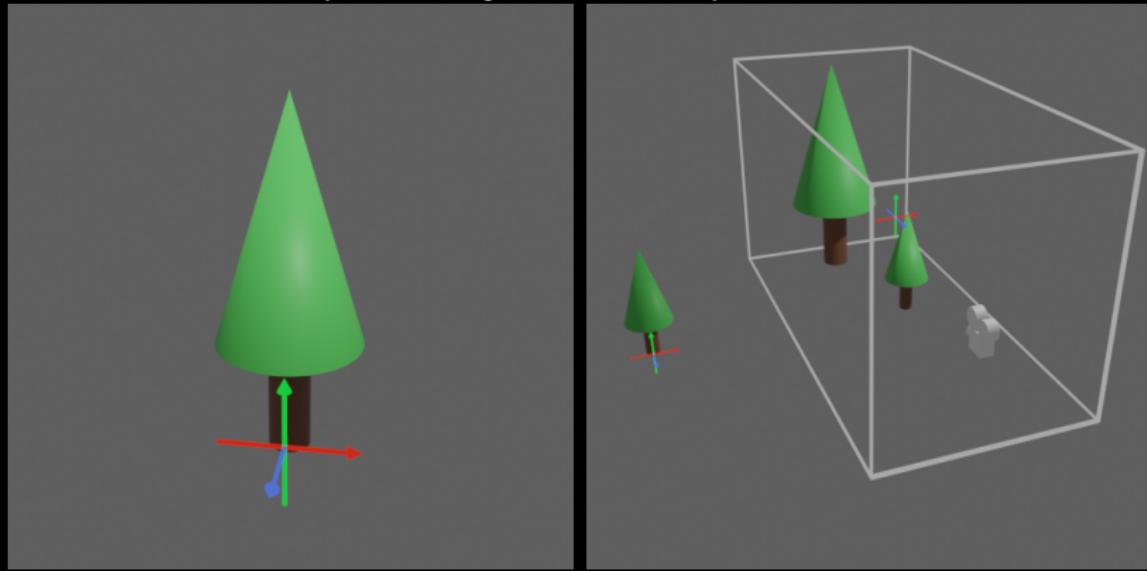
- Le repere de l'objet
- La matrice *model* place l'objet dans le repere de la scene
- La matrice *view* place l'ensemble dans le repere camera
- La matrice *projection* place l'ensemble dans le *clip space*
- (Puis les coordonnées sont divisées par *w* pour tout envoyer dans le *normalized device coordinates*) et par la suite envoyer dans le repere écran)

## Le repere de l'objet



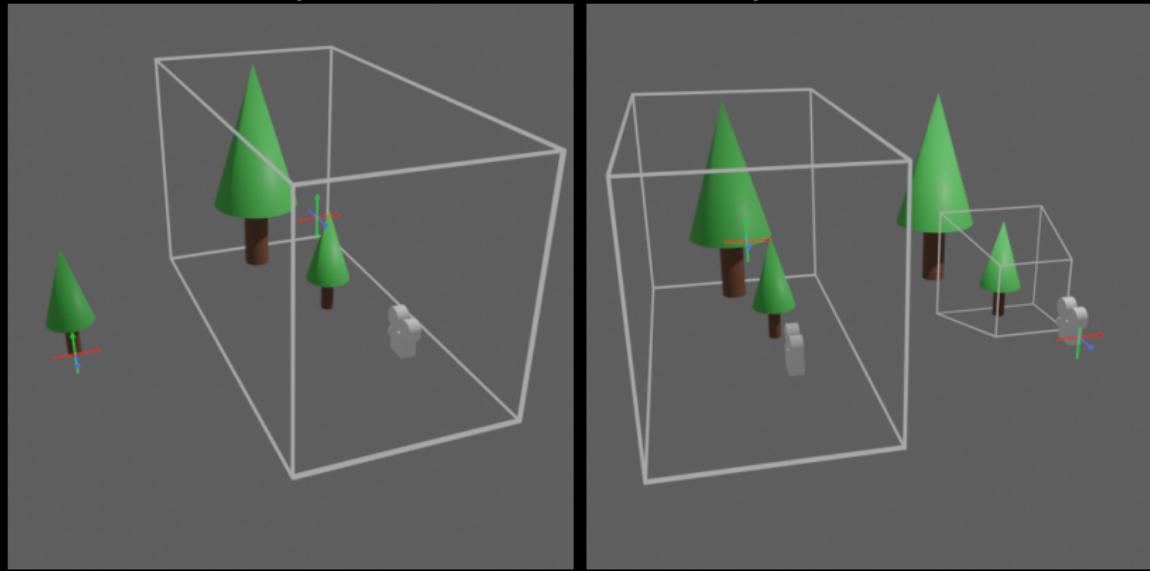
## Vertex Shader - les différents repères

La matrice *model* place l'objet dans le repère de la scène



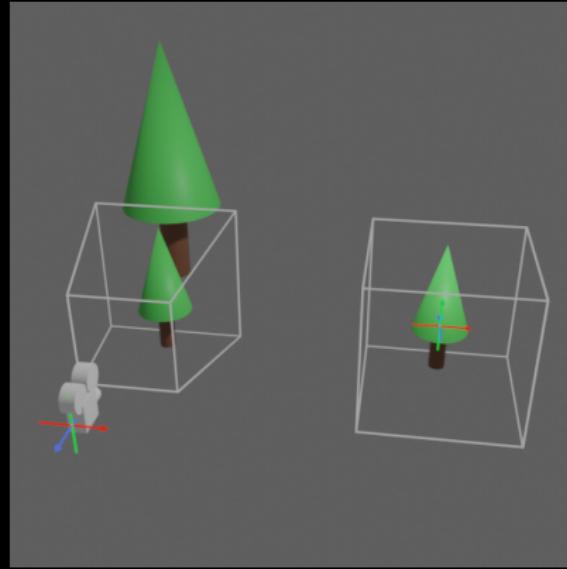
## Vertex Shader - les différents repères

La matrice *view* place l'ensemble dans le repere camera



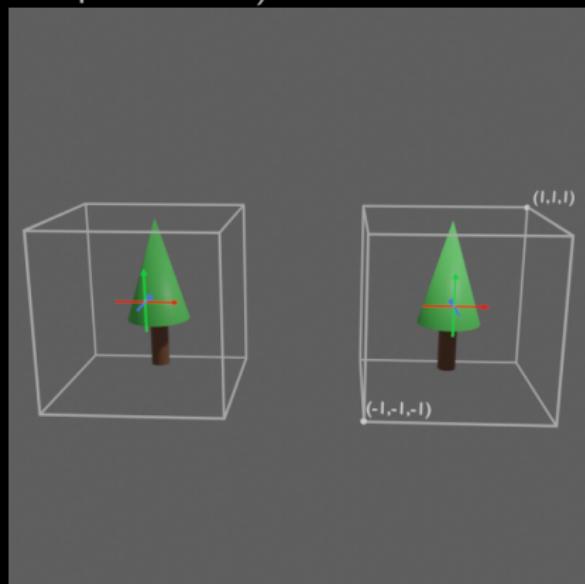
## Vertex Shader - les différents repères

La matrice *projection* place l'ensemble dans le *clip space*



## Vertex Shader - les différents repères

(Puis les coordonnées sont divisées par  $w$  pour tout envoyer dans le *normalized device coordinates*) et par la suite envoyer dans le repère écran)

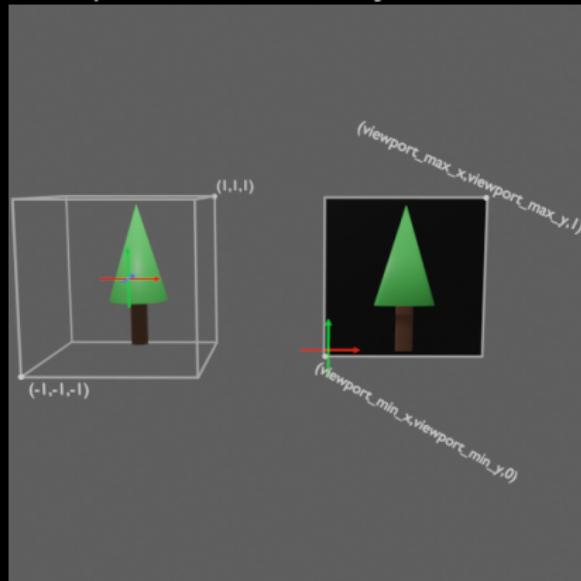


$$v_{clip} = \begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} \quad (1)$$

$$v_{NDC} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \\ w_{clip}/w_{clip} \end{pmatrix} = \begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \\ 1 \end{pmatrix} \quad (2)$$

## Vertex Shader - les différents repères

et par la suite envoyer dans le repère écran



$$v_{screen} = \begin{pmatrix} \frac{width}{2}(x_{NDC} + 1) + off_x \\ \frac{height}{2}(y_{NDC} + 1) + off_y \\ \frac{1}{2}z_{NDC} + \frac{1}{2} \end{pmatrix} \quad (3)$$

## Comment passer du model au repere camera ?

D'abord on doit changer l'origine

$$\begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

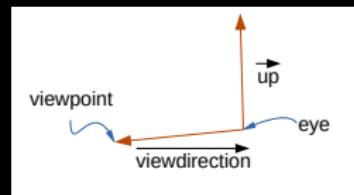
## Comment passer du model au repere camera ?

Nous pouvons definir l'orientation la caméra par 2 vecteurs (et le centre) :

$$\begin{bmatrix} viewdirection_x \\ viewdirection_y \\ viewdirection_z \end{bmatrix} = \begin{bmatrix} viewpoint_x \\ viewpoint_y \\ viewpoint_z \end{bmatrix} - \begin{bmatrix} eye_x \\ eye_y \\ eye_z \end{bmatrix} \quad (5)$$

et

$$\begin{bmatrix} up_x \\ up_y \\ up_z \end{bmatrix} \quad (6)$$



## Comment passer du model au repere camera ?

Or  $\overrightarrow{up}$  et  $\overrightarrow{viewdirection}$  ne sont peut être pas orthogonaux.

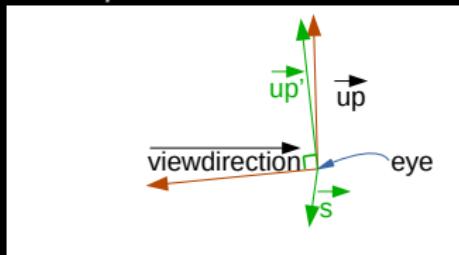
On normalise  $\overrightarrow{up}$  et  $\overrightarrow{viewdirection}$  puis

$$\vec{s} = \overrightarrow{viewdirection} \times \overrightarrow{up} \quad (7)$$

puis

$$\overrightarrow{up'} = \vec{s} \times \overrightarrow{viewdirection} \quad (8)$$

Nous avons un nouveau  $up'$  correct.



## Comment passer du model au repere camera ?

La transformation finale pour un sommet  $P = (x, y, z, 1)^t$  :

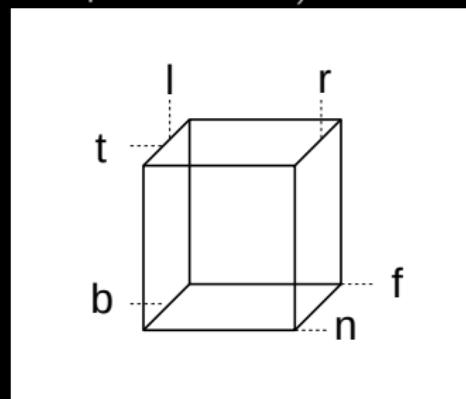
$$P' = \begin{bmatrix} s_x & s_y & s_z & 0 \\ up'_x & up'_y & up'_z & 0 \\ -viewdirection_x & viewdirection_y & viewdirection_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \end{bmatrix} P \quad (9)$$

## Comment passer du repere camera au *clip space*?

Avec une projection parallele

Si on veut grader les points entre l et r (horizontalement), t et b verticalement, n et f en profondeur, on doit mapper  $l..r$  entre  $-1..1$ ,  $b..t$  entre  $-1..1$  et  $n..f$  entre  $-1..1$ . On definit donc des droites affines :

La transformation finale pour un sommet  $P' = (x', y', z', w)^t$  (dans le repere camera) :



## Comment passer du repere camera au *clip space*?

Avec une projection parallele

Si on veut grader les points entre l et r (horizontalement), t et b verticalement, n et f en profondeur, on doit mapper l..r entre -1..1, b..t entre -1..1 et n..f entre -1..1. On definie donc des droites affines :

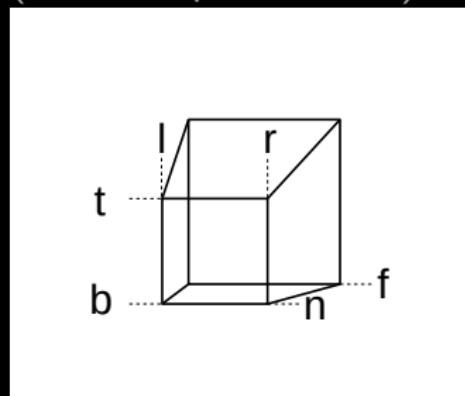
La transformation finale pour un sommet  $P' = (x', y', z', w)^t$  (dans le repere camera) :

$$P'' = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} P' \quad (10)$$

Comment passer du repere camera au *clip space*?

Avec une projection perspective

La transformation finale pour un sommet  $P' = (x', y', z', w')^t$   
(dans le repere camera) :



Note :  $P''[4] = -1 * z'$

## Comment passer du repere camera au *clip space*?

Avec une projection perspective

La transformation finale pour un sommet  $P' = (x', y', z', w')^t$   
(dans le repere camera) :

$$P'' = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} P' \quad (11)$$

Note :  $P''[4] = -1 * z'$

A la fin OpenGL divise par  $w''$

Comment passer du repere camera au *clip space*?

Avec le changement de repere que vous voulez...  
On n'est pas limité à un type de projection

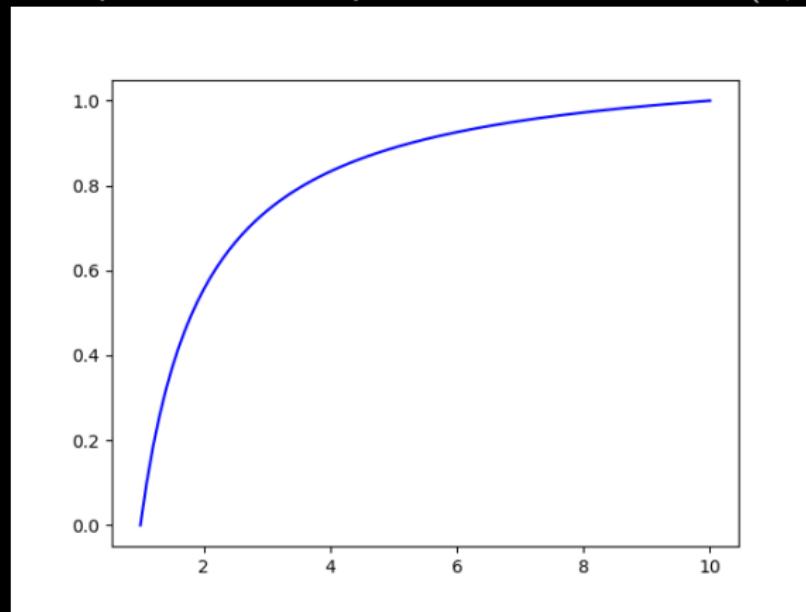
## Vertex Shader - la précision du z-buffer

```
def f(i,A):
    j=-(i)
    B= np.array ([[0.0],[0.0],[j],[1.0]])
    #z_clip=A.dot(B)[2]
    z_NDC=(A.dot(B)/A.dot(B)[3])[2]
    z_screen= z_NDC/2.0+1/2.0;
    return z_screen
```

Avec une matrice de projection orthogonale,  $z_{screen}$  est linéaire.  
Avec une matrice de projection perspective,  $z_{screen}$  n'est pas linéaire.

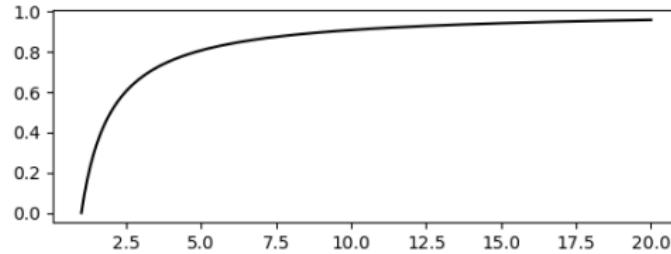
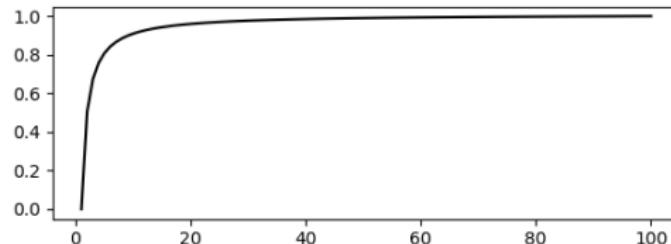
## Vertex Shader - la précision du z-buffer

near plane = 1, far plane = 10,  $z_{screen} = f(z, A_{perspective})$



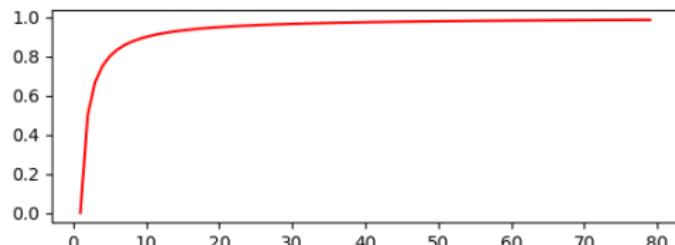
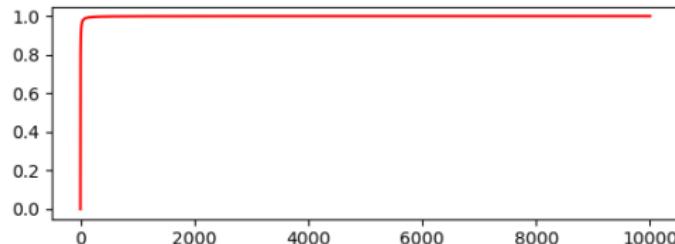
## Vertex Shader - la précision du z-buffer

near plane = 1, far plane = 100,  $z_{screen} = f(z, A_{perspective})$

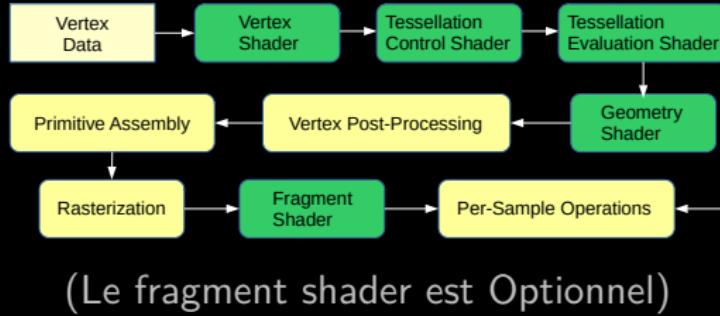


## Vertex Shader - la précision du z-buffer

near plane = 1, far plane = 10 000,  $z_{screen} = f(z, A_{perspective})$



# Fragment shader



- Fragment shader en entrée :
  - gl\_FragCoord : Coordonnée (écran) du pixel
  - gl\_FrontFacing
  - gl\_PointCoord
- gl\_Color : couleur du sommet
- Secondary color
- gl\_TexCoordn
- gl\_FogFragCoord

- Fragment shader en entrée :
  - gl\_FragCoord : Coordonnée (écran) du pixel
  - gl\_FrontFacing
  - gl\_PointCoord
  - gl\_Color : couleur du sommet
  - Secondary color  
*Deprecated !*
  - gl\_TexCoordn
  - gl\_FogFragCoord

- Pixel shader en sortie :
  - `gl_FragDepth` : profondeur
  - `gl_FragColor` : Couleur du fragment
  - `gl_FragData`

- Pixel shader en sortie :
  - gl\_FragDepth : profondeur
  - gl\_FragColor : Couleur du fragment  
*Deprecated!*
  - gl\_FragData

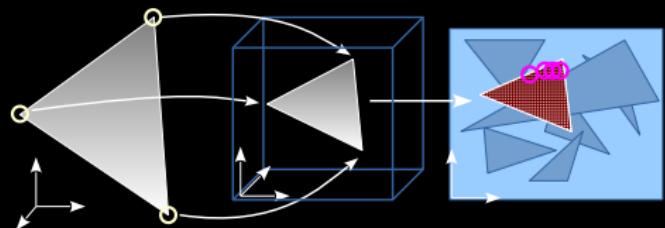
# Fragment shader

Utilisation de in/out

Exemple :

```
#version 450  
  
in vec4 color;  
out vec4 output_color;  
  
void main() {  
    output_color = color;  
}
```

Valeur de color interpolée !



## Shaders : exemples

### Calcul de l'illumination en chaque sommets

#### Vertex shader

```
#version 450

layout(location = 1) in vec3 vPosition;
layout(location = 2) in vec3 vNormal;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;
uniform vec3 object_color;
uniform vec3 light_position;

out vec3 color;
vec3 normal;
vec3 light_dir;

void main() {
    gl_Position =
        projection_matrix
        * model_view_matrix
        * vec4(vPosition, 1.0);
    normal = normalize(vNormal);
    light_dir = normalize(light_position - vPosition);
    color = clamp((dot(normal, light_dir)) * object_color, 0, 1);
}
```

#### Fragment shader

```
#version 450

in vec3 color;
out vec4 output_color;

void main()
{
    output_color = vec4(color, 1.0);
}
```

## Calcul de l'illumination en chaque fragment

### Vertex shader

```
#version 450

layout(location = 1) in vec3 vPosition;
layout(location = 2) in vec3 vNormal;

uniform mat4 model_view_matrix;
uniform mat4 projection_matrix;
uniform vec3 object_color;
uniform vec3 light_position;

out vec3 color;
out vec3 normal;
out vec4 light_dir;

void main() {
    gl_Position =
        projection_matrix
        * model_view_matrix
        * vec4(vPosition, 1.0);
    light_dir = normalize(light_position - vPosition);
    normal = normalize(vNormal);
    color = object_color;
}
```

### Fragment shader

```
#version 450
in vec3 color;
out vec4 output_color;
in vec3 normal;
in vec3 light_dir;

void main() {
    output_color = vec4(
        clamp(
            dot(normal, light_dir) * color
            , 0.0, 1.0
        )
        , 1.0);
```

Quelques remarques :

- Usage du `vec3`
- `normalize / glVertexAttribPointer (normal = normalize(vNormal);)`
- produit `projection_matrix * model_view_matrix`
- clamp / max



b entier 8 bits - signed char - GLbyte

s entier 16 bits - short - GLshort

i entier 32 bits - long, int - GLint, GLsizei

f réel 32 bits - float - GLfloat, GLclampf

d réel 64 bits - double - GLdouble, GLclampd

ub entier non signé 8 bits - unsigned char - GLubyte, GLboolean

us entier non signé 16 bits - unsigned short - GLushort

ui entier non signé 32 bits - unsigned long - GLuint, GLenum,  
GLbitfield

v vecteur

Correspondance int (glsl) GLint (OpenGL), float (glsl) et GLfloat  
(OpenGL)...

```
GLenum glGetError(void);
```

Retourne la valeur actuelle de la variable d'état erreur de l'environnement OpenGL. Valeurs possibles :

- GL\_NO\_ERROR,
- GL\_INVALID\_ENUM,
- GL\_INVALID\_VALUE,
- GL\_INVALID\_OPERATION,
- GL\_STACK\_OVERFLOW,
- GL\_STACK\_UNDERFLOW,
- GL\_OUT\_OF\_MEMORY.

glGetError remplace la variable d'état erreur à GL\_NO\_ERROR après interrogation.

Interaction avec OpenGL :

- glEnable(GLenum cap)
- glDisable(GLenum cap)

OpenGL est bas niveau et spécifique pour la génération des images.

- pas de gestion des fenêtres
- pas d'interaction utilisateur
- ...

Compléments utiles :

- glut/SDL/Qt/...
- glew
- ...

## Gestion de la fenêtre et des initialisations d'OpenGL

```
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize (800,600);
    glutFullScreen ();
glutInitContextVersion (4, 5);
glutCreateWindow (char *....);
```

## Evènements/interaction

```
glutDisplayFunc(display); (→ glutPostRedisplay());  
glutReshapeFunc(reshape);  
glutKeyboardFunc(keyboard);  
glutKeyboardUpFunc(keyboardup);  
glutSpecialFunc(special);  
glutSpecialUpFunc(specialup);  
glutMouseFunc(mouse);  
glutMotionFunc(motion);  
glutPassiveMotionFunc(motion);  
glutIdleFunc(idle_func);  
glutTimerFunc(timer);
```

## Glut (freeGlut)

Enregistre un callback pour le traitement de l'évènement paint :

```
glutDisplayFunc(display);
```

Boucle d'évènements

```
glutMainLoop();
```

Lance l'évènement paint

```
glutPostRedisplay();
```

Erreur classique :

```
void display(void){  
    ...  
    glutPostRedisplay();  
}
```

## Glut (freeGlut)

Enregistre un callback pour le traitement de l'évènement paint :

```
glutDisplayFunc(display);
```

Boucle d'évènements

```
glutMainLoop();
```

Lance l'évènement paint

```
glutPostRedisplay();
```

Erreur classique :

```
Nooooo!  
void display(void){  
    ...  
    glutPostRedisplay();  
}
```

# GLEW

```
glewExperimental = GL_TRUE;
GLenum err = glewInit();
if (GLEW_OK != err) {
    std::cerr << "GlewInit fails : " << glewGetErrorString(err) << std::endl;
    return false;
}
return true;
```

## Exemple

## Exemple - Initialisation

```
void init_gl( void ) {
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glDepthRange(0.0, 1.0);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glEnable(GL_CULL_FACE);
    glCullFace(GL_FRONT);
    glFrontFace(GL_CCW);
    glClearColor(0.0, 0.0, 0.0);
    ...
}

init_glut(argc, argv);
init_glew();
init_gl();
```

## Exemple - Déclaration

```
glGenVertexArrays(1, &vao_id);
 glBindVertexArray(vao_id);

 glGenBuffers(1, &vbo_position_normal_earth_id);
 glBindBuffer(GL_ARRAY_BUFFER, vbo_position_normal_id);
 glBufferData(GL_ARRAY_BUFFER, nb_vertex*sizeof(GLfloat), vertex_list, GL_STATIC_DRAW);
 glVertexAttribPointer(vertex_position_location, 3, GL_FLOAT, GL_FALSE, 6*sizeof(float),
 glEnableVertexAttribArray(vertex_position_location);

 glVertexAttribPointer(vertex_normal_location, 3, GL_FLOAT, GL_FALSE, 6*sizeof(GLfloat),
 glEnableVertexAttribArray(vertex_normal_location);

 glGenBuffers(1, &vbo_texture_coord_earth_id);
 glBindBuffer(GL_ARRAY_BUFFER, vbo_texture_coord_id);
 glBufferData(GL_ARRAY_BUFFER, nb_vertex/3*2*sizeof(GLfloat), uv_list, GL_STATIC_DRAW);
 glVertexAttribPointer(uv_location, 2, GL_FLOAT, GL_FALSE, 2*sizeof(GLfloat), 0);
 glEnableVertexAttribArray(uv_location);

 glBindVertexArray(0);
```

## Exemple - Texture

```
texture_bitmap = load_image("texture_monde.tga");
GLint texture_location;

glGenTextures(1, &earth_texture);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, earth_texture);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture_bitmap->iw, texture_bitmap->ih, 0, GL_RGB);

glTexParameteri(...);

earth_texture_location = glGetUniformLocation(program->program_id, "tex_sampler");
 glUniform1i(earth_texture_location, 0);
```

## Exemple - Vertex Shader (code must be improved !)

```
#version 420

in vec3 vertex_position;
in vec3 vertex_normal;
in vec2 vertex_uv;

uniform mat4 model_view;
uniform mat4 projection;
uniform vec3 light_position;

out vec4 interpolated_color;
out vec2 interpolated_uv_position;

void main ()
{
    vec4 light_dir = vec4(light_position - vertex_position.xyz, 1.0);
    float coef = dot(normalize(vertex_normal.xyz), normalize(light_dir.xyz));
    coef = clamp(coef, 0.0, 1.0);
    interpolated_color = vec4(vec3(1.0, 1.0, 1.0) * coef, 1.0);
    gl_Position = projection * model_view * vec4(vertex_position, 1.0);
    interpolated_uv_position = vertex_uv;
}
```

## Exemple - Fragment Shader (code must be improved !)

```
#version 420

uniform sampler2D tex_sampler;

in vec4 interpolated_color;
in vec2 interpolated_uv_position;

out vec4 output_color;

void main()
{
    vec4 texel = texture(tex_sampler, interpolated_uv_position);
    output_color = interpolated_color * vec4(texel.rgb, 1.0);
}
```

## Exemple - Start drawing

```
glBindVertexArray(vao_id);
glDrawArrays(GL_TRIANGLES, 0, vertex *3);
glBindVertexArray(0);
```

## Exemple - Variante

```
    ...
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, &vbo_position_shape_id);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                 my_shape.nb_faces*3*sizeof(GLuint),
                 my_shape.triangles_list,
                 GL_STATIC_DRAW);
    ...

    ...
    glDrawElements(GL_TRIANGLES, my_shape.faces*3, GL_UNSIGNED_INT, 0);
    ...
```

- Adressage indexé (différent du format obj par exemple).

## Résultat



A suivre...

- Bas niveau
  - Vulcan < OpenGL < Unity
- Portable (+WebGL, OpenGL ES...)