Programmation OpenGL

- OpenGL 4 - Les bases -

Jonathan Fabrizio

http://jo.fabrizio.free.fr

Version : Tue May 11 10:50:32 2021

OpenGL

Heureux l'étudiant qui, comme la rivière, peut suivre son cours sans sortir de son lit...

OpenGL

Introduction

OpenGL

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

- OpenGL est une spécification
- La version N-1 de GL est libre (OpenGL)...
- Une implémentation libre : MESA

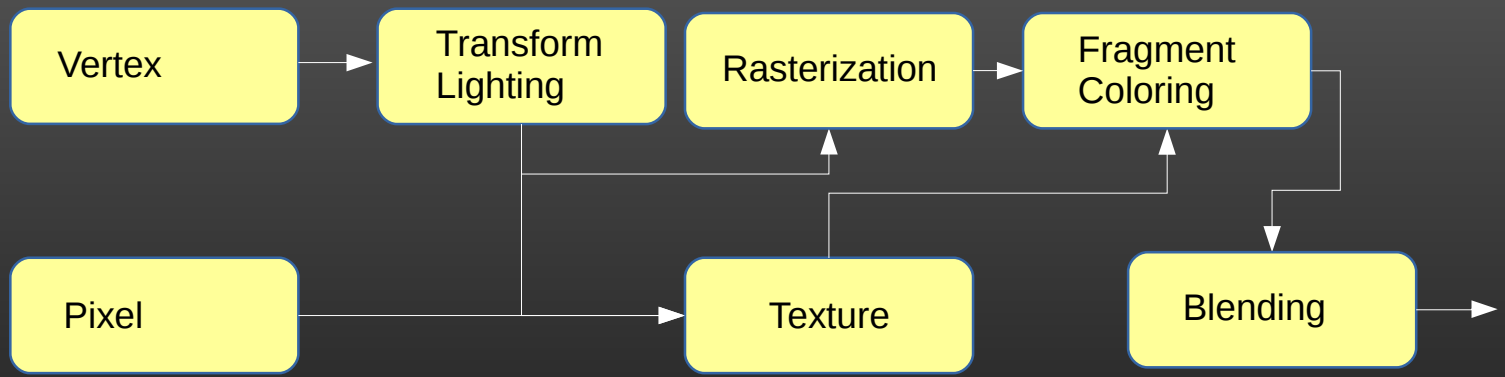
OpenGL

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)

OpenGL : Les temps anciens

Pipeline graphique fixe (OpenGL <2.0)



OpenGL : Les temps anciens

```
glBegin();
  glVertex();
  glNormal();
  glColor();
glEnd();

glMatrixMode();
glPushMatrix();
glTranslated();
glFrustum();
```

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

```
glShadeModel();
glEnable(GL_LIGHT
glEnable(GL_LIGHT
```

OpenGL : Les temps anciens

```
glBegin();
  glVertex();
  glNormal();
  glColor();
glEnd();

glMatrixMode();
glPushMatrix();
glTranslated();
glFrustum();
```

Deprecated !

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

```
glShadeModel();
glEnable(GL_LIGHT
glEnable(GL_LIGHT
```

OpenGL : Les temps anciens

```
glBegin();
  glVertex();
  glNormal();
  glColor();
glEnd();

glMatrixMode();
glPushMatrix();
glTranslated();
glFrustum();
```

Deprecated !

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

Deprecated !

```
glShadeModel();
glEnable(GL_LIGHT
glEnable(GL_LIGHT
```

OpenGL : Les temps anciens

```
glBegin();
  glVertex();
  glNormal();
  glColor();
glEnd();

glMatrixMode();
glPushMatrix();
glTranslated();
glFrustum();
```

Deprecated !

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

Deprecated !

```
glShadeModel();
glEnable(GL_LIGHT
glEnable(GL_LIGHT
```

Deprecated !

OpenGL : Les temps anciens

```
glBegin();
  glVertex();
  glNormal();
  glColor();
glEnd();
```

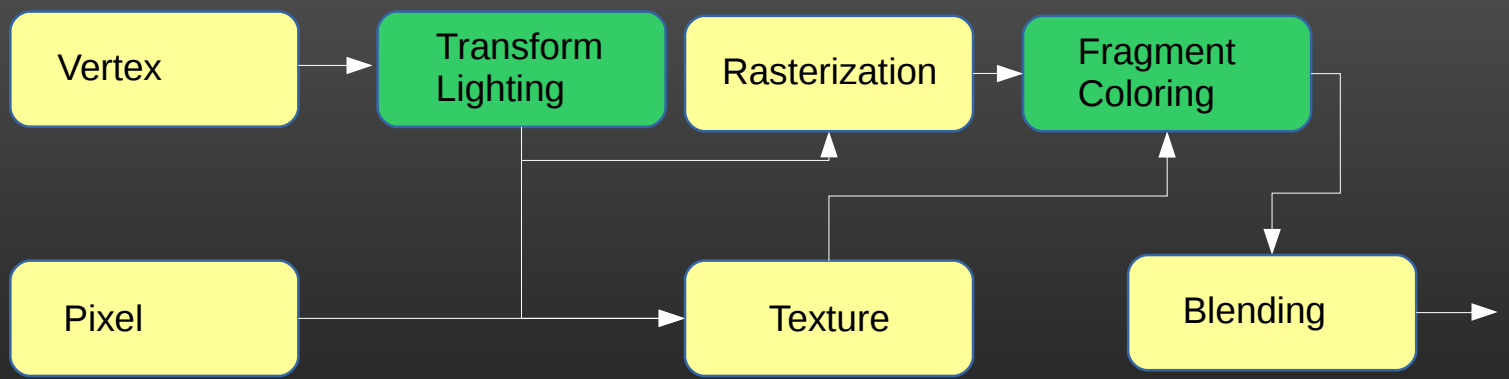
```
glMatrixMode();
glPushMatrix();
glTranslated();
glFrustum();
```

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

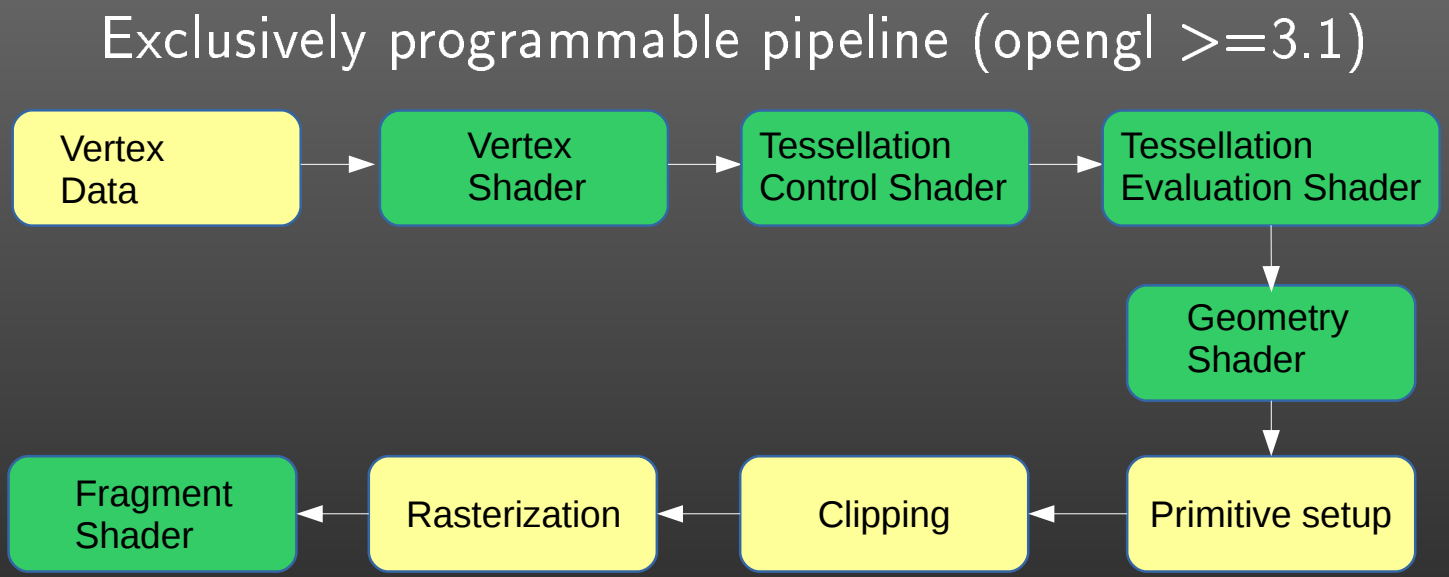
```
glShadeModel();
glEnable(GL_LIGHT
glEnable(GL_LIGHT
```

OpenGL 2.0 : La fin des temps anciens

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



OpenGL 3.0 : Le début d'une aire nouvelle

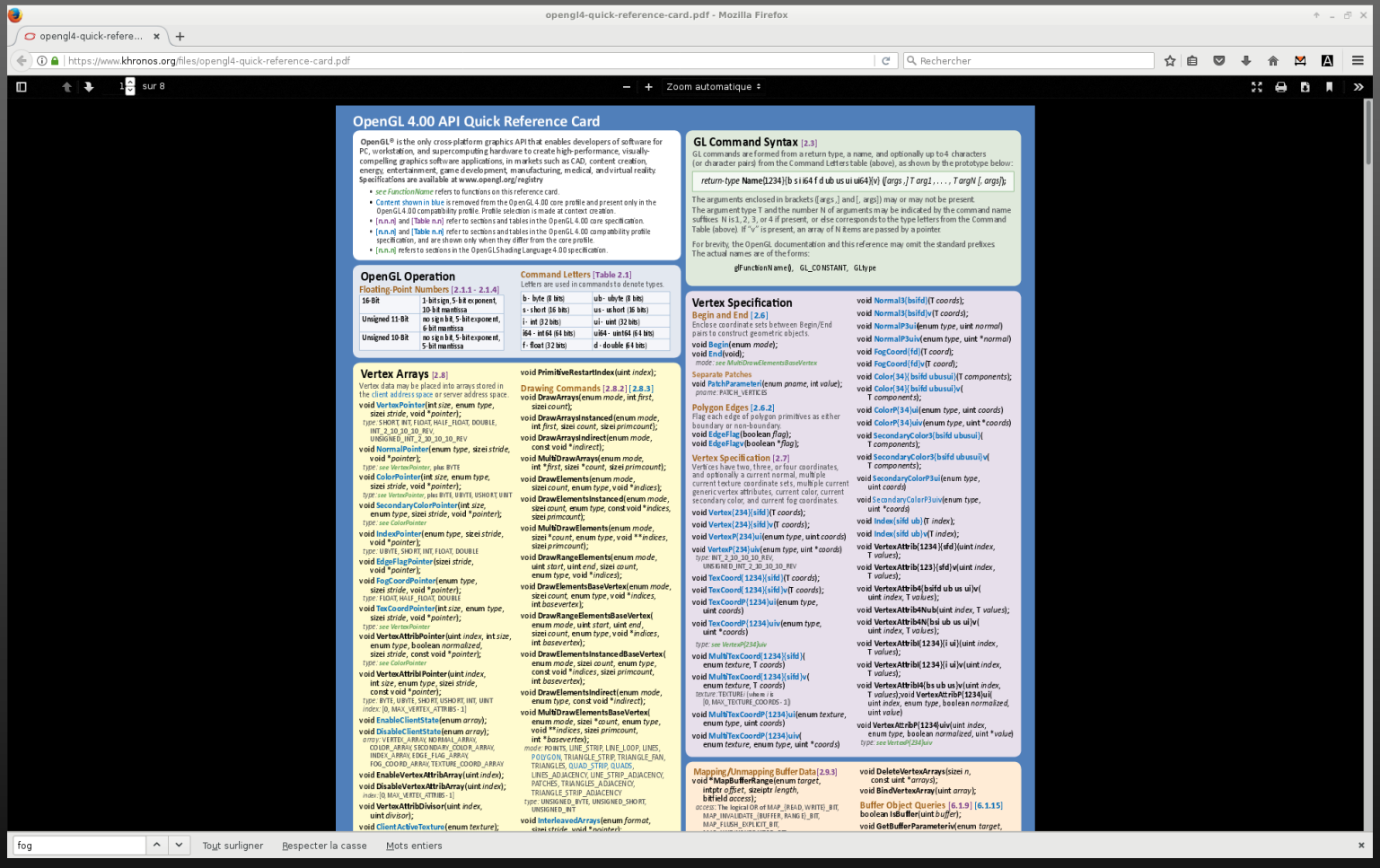


- Vertex shader
- Pixel shader
- Almost all data is GPU-resident

OpenGL 3.0 : Le début d'une aire nouvelle

- 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

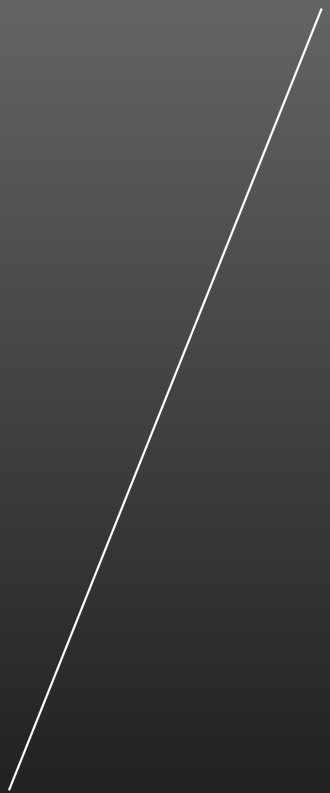
OpenGL : Respect de la norme actuelle



source : www.khronos.org

Structure

CPU
RAM



GPU
Shaders (GLSL)
VRAM

Principe général

Principe général

En simplifié :

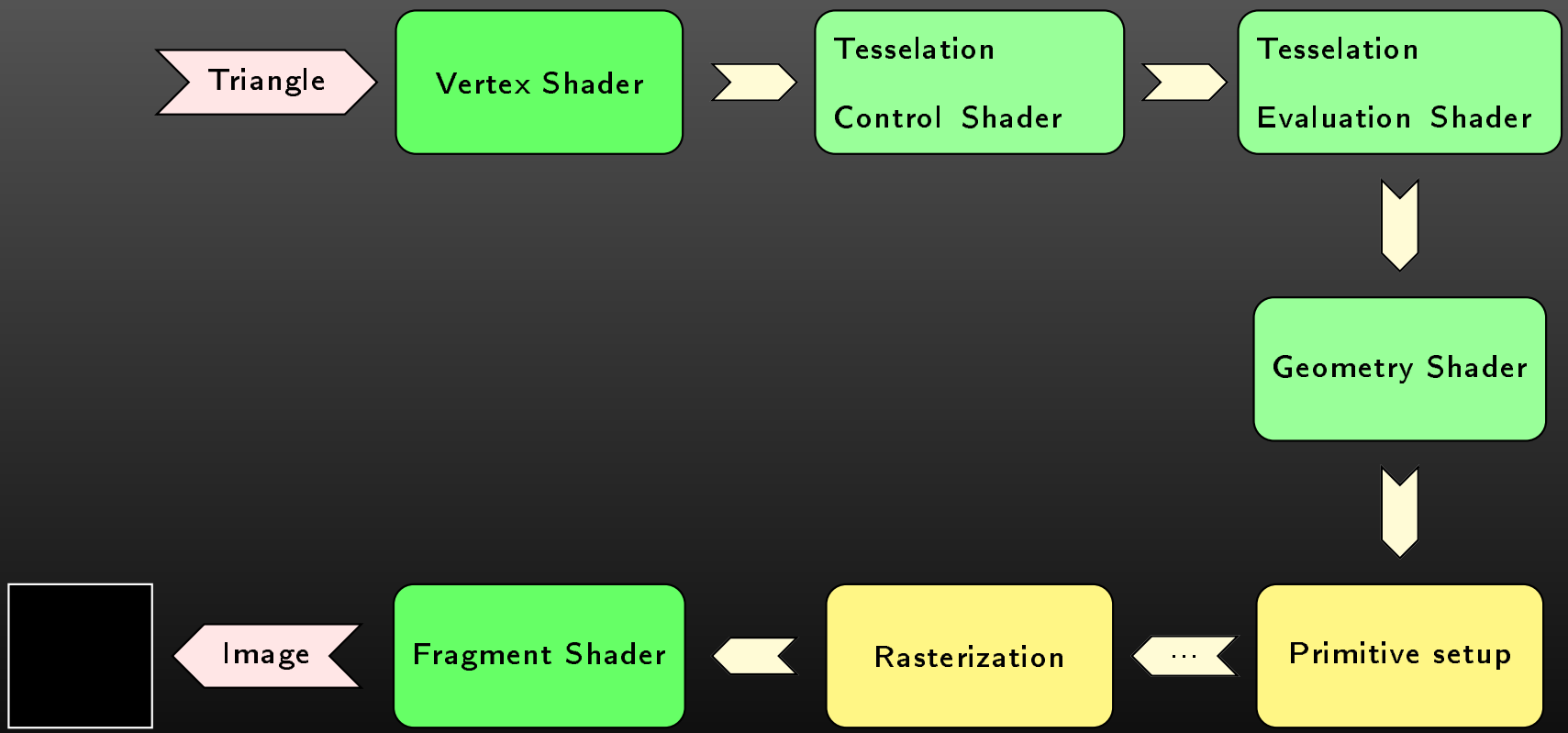
1. faire les différentes initialisations,
2. compiler les *shaders*,
3. initialiser les données,
4. activer le bon programme (*shaders*) puis envoyer les données vers le pipeline graphique,
5. récuperer l'image

Principe général

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

Principe général

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*.



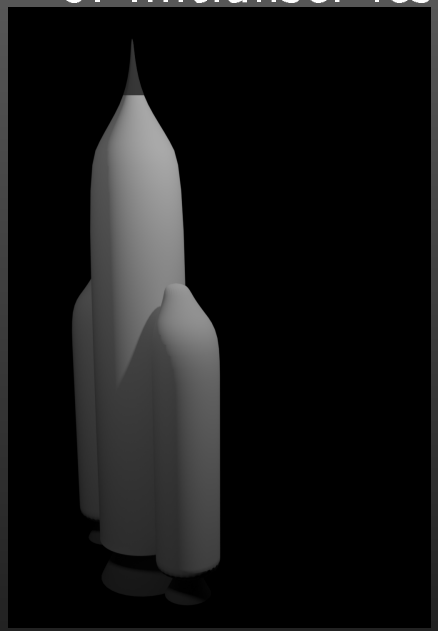
Ils sont écrits en *GLSL*.

Principe général

3. Initialiser les données

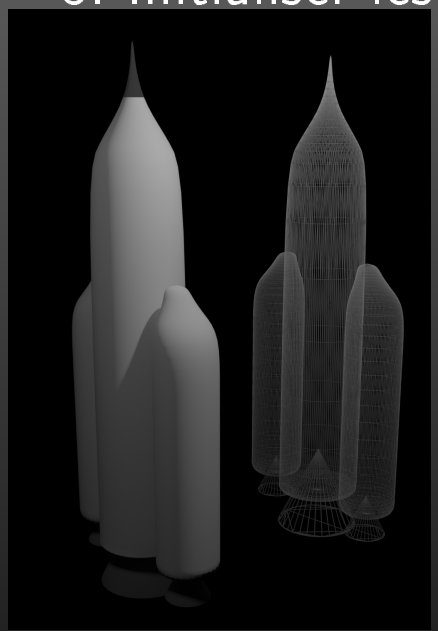
Principe général

3. Initialiser les données



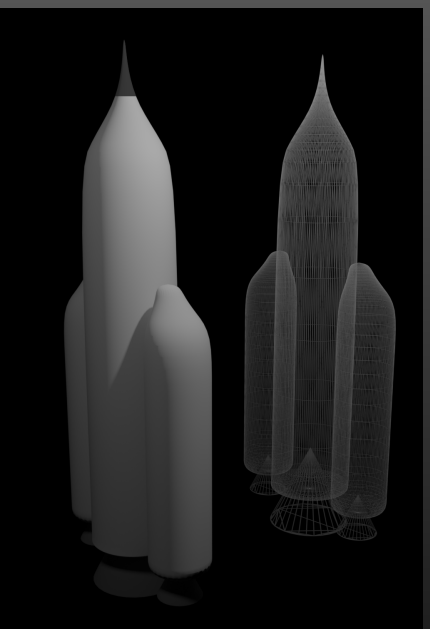
Principe général

3. Initialiser les données



Principe général

3. Initialiser les données



Préparer les données du maillage dans un buffer (le VBO[a]).

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

Expliquer comment les données sont organisées/découpées dans le buffer

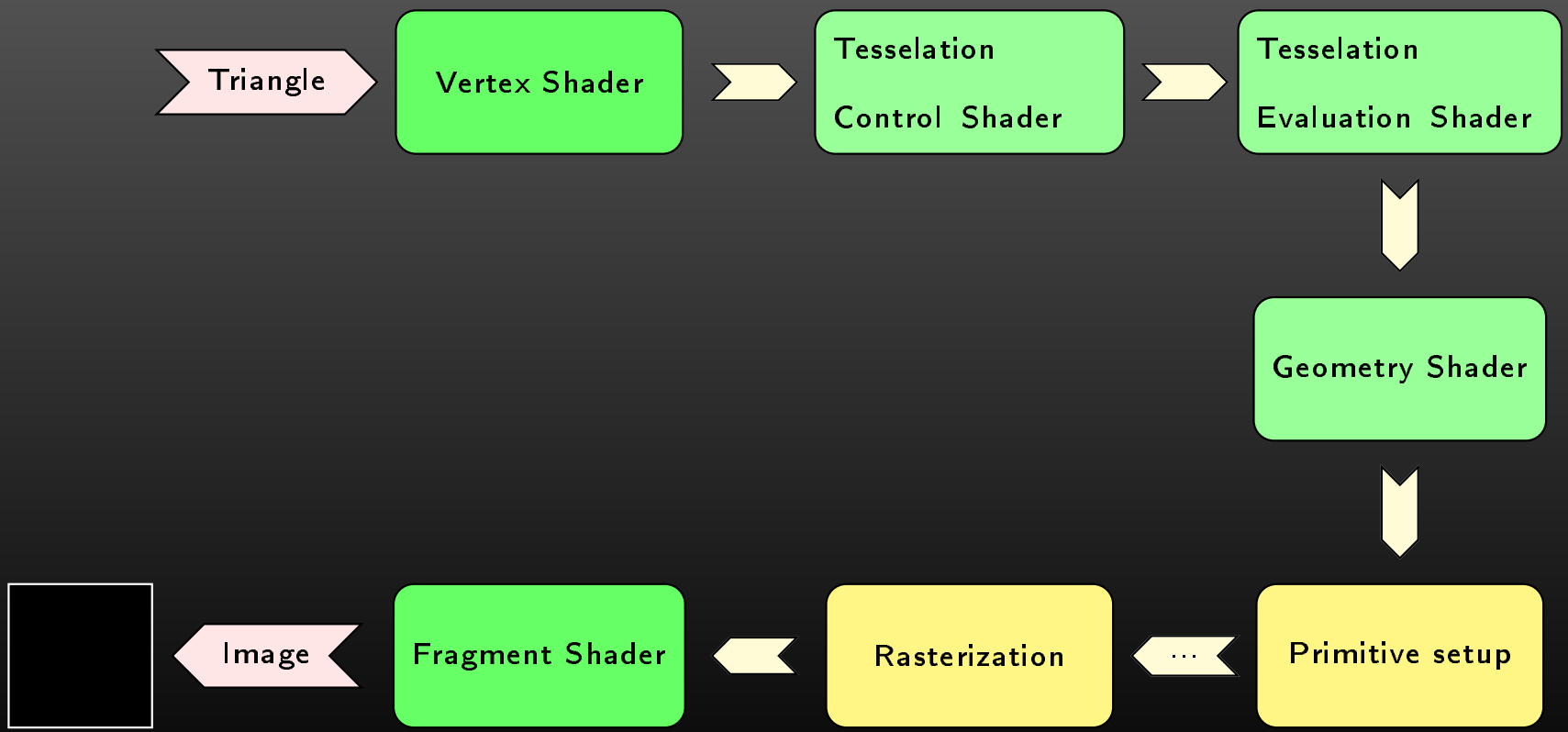
a. Vertex Buffer Object

Principe général

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

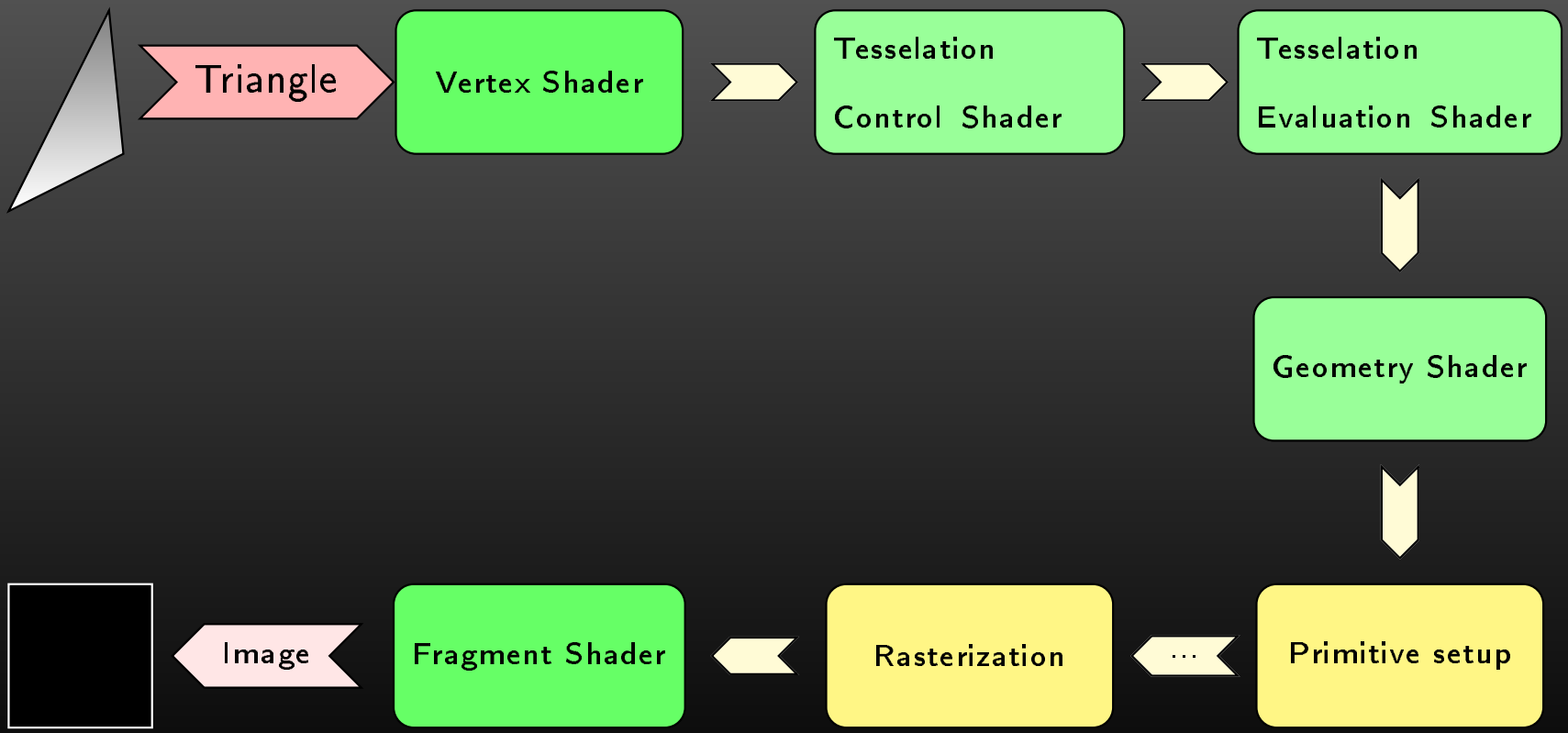
Principe général

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



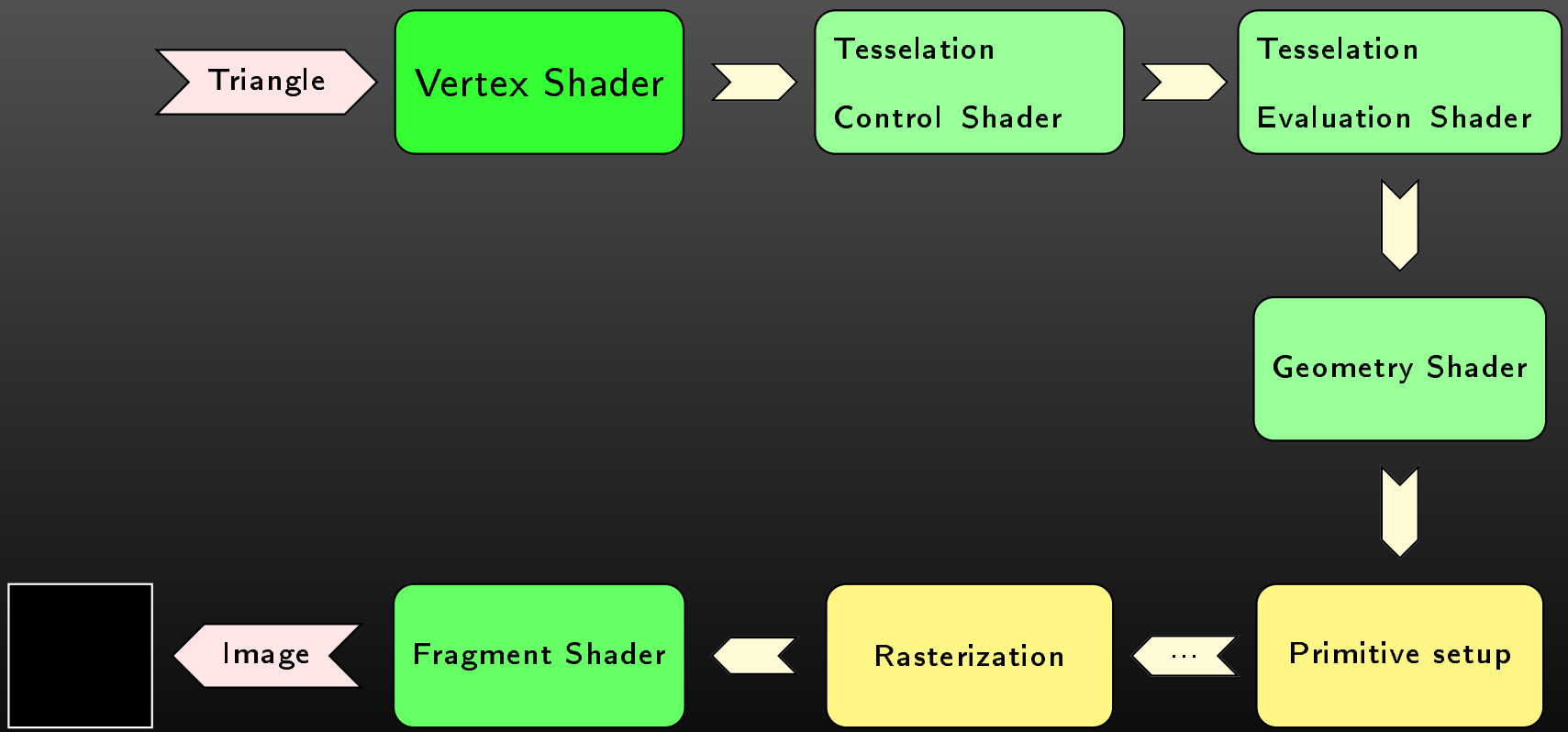
Principe général

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



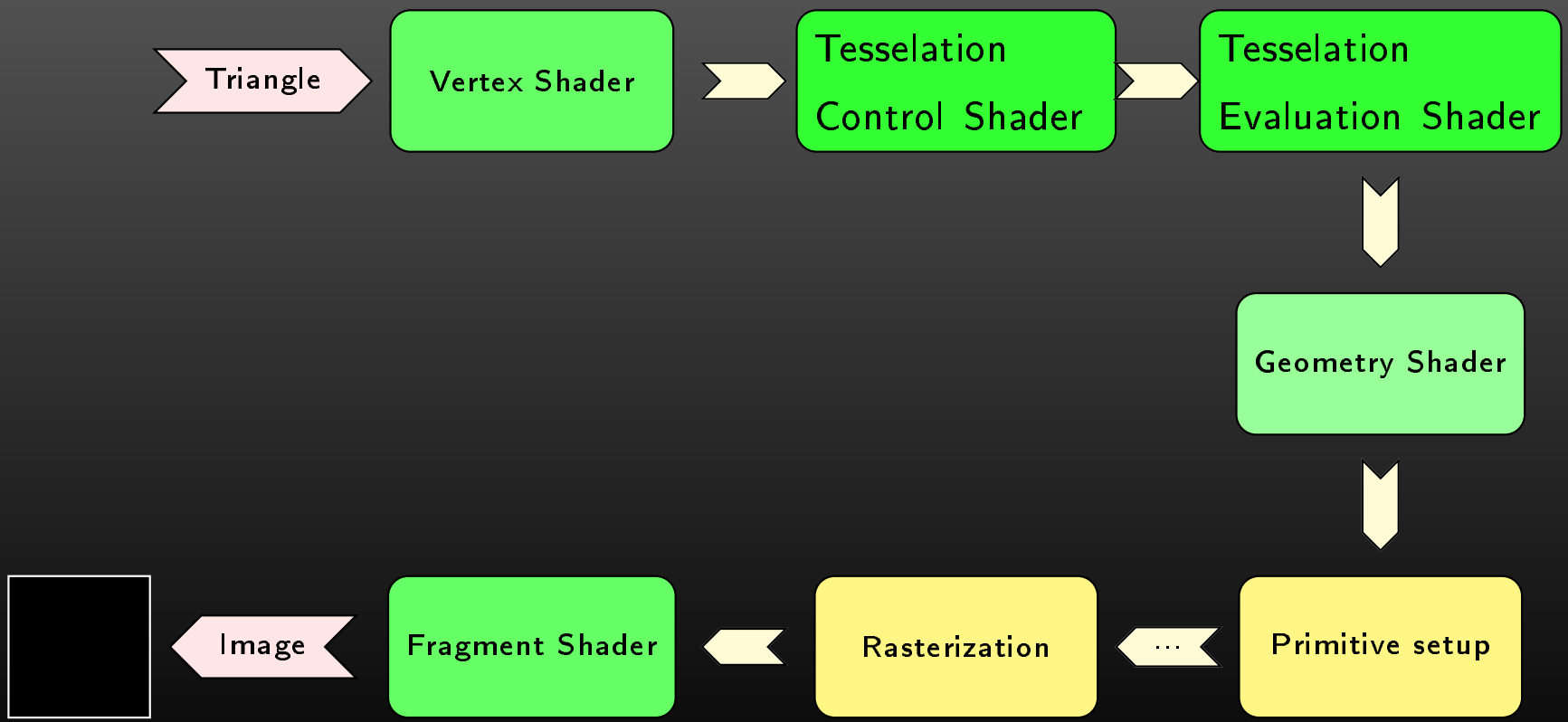
Principe général

4. Activer le bon programme (*shaders*) puis envoyer les données (buffers) vers le pipeline graphique :
Vertex Shader - Changement de repère pour préparer la projection



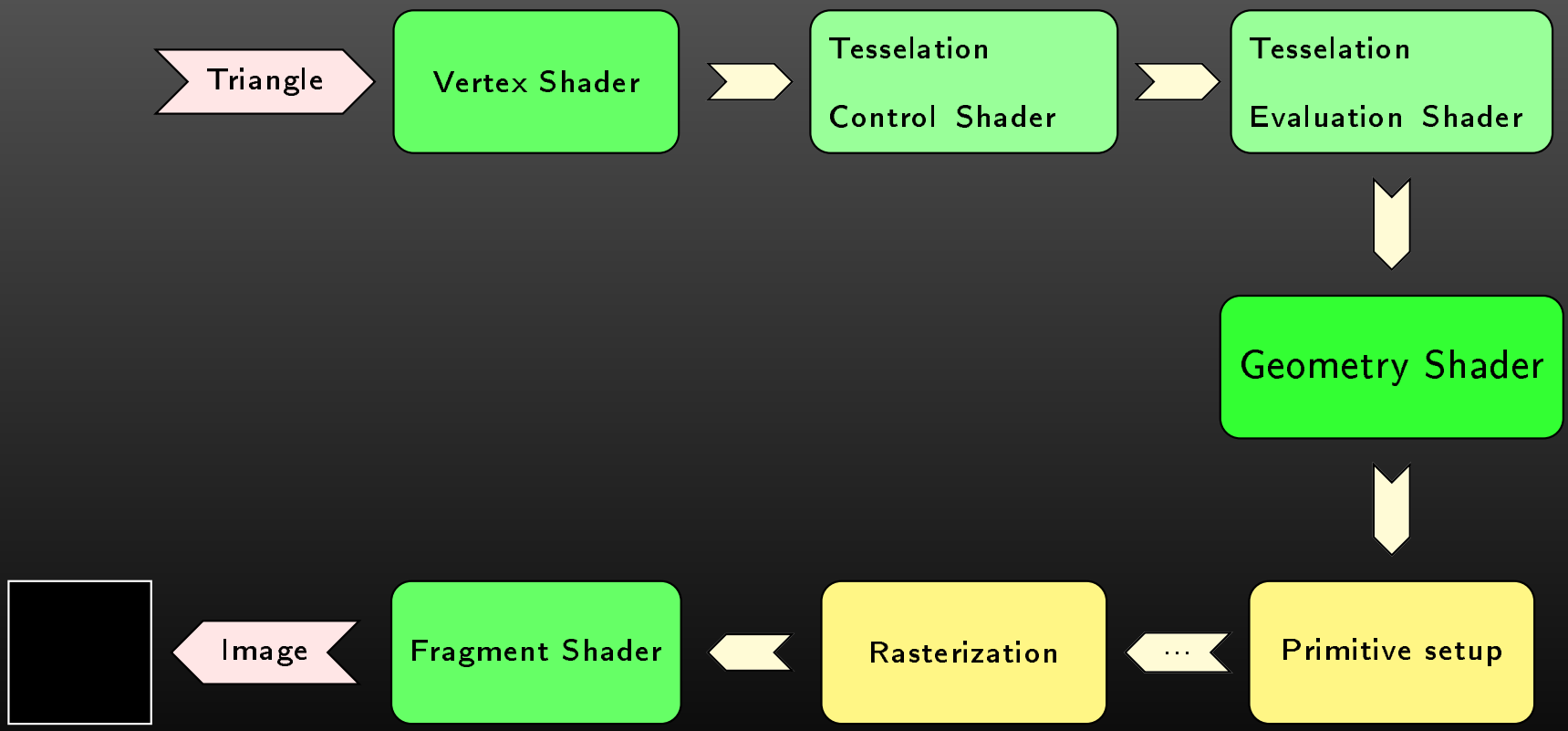
Principe général

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



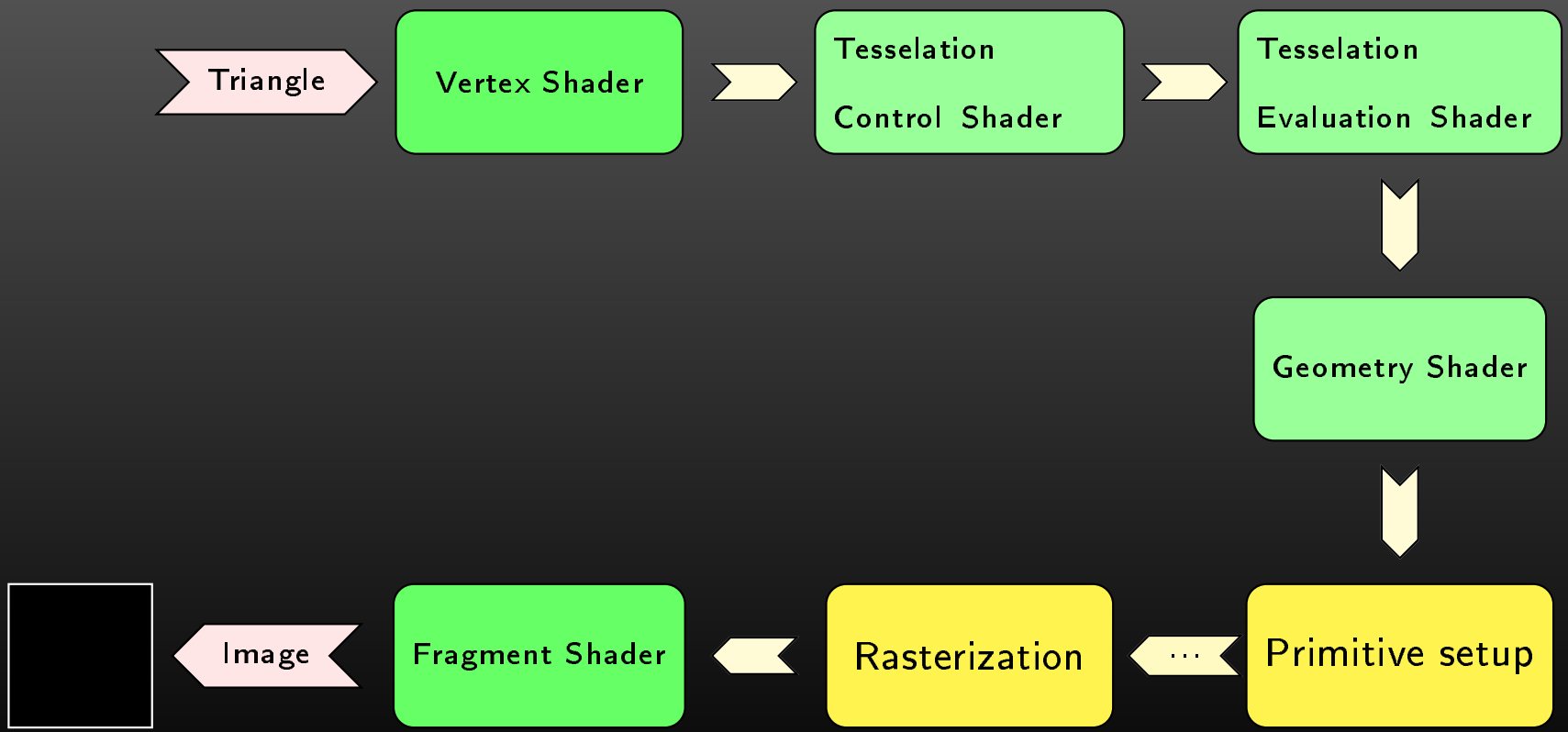
Principe général

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



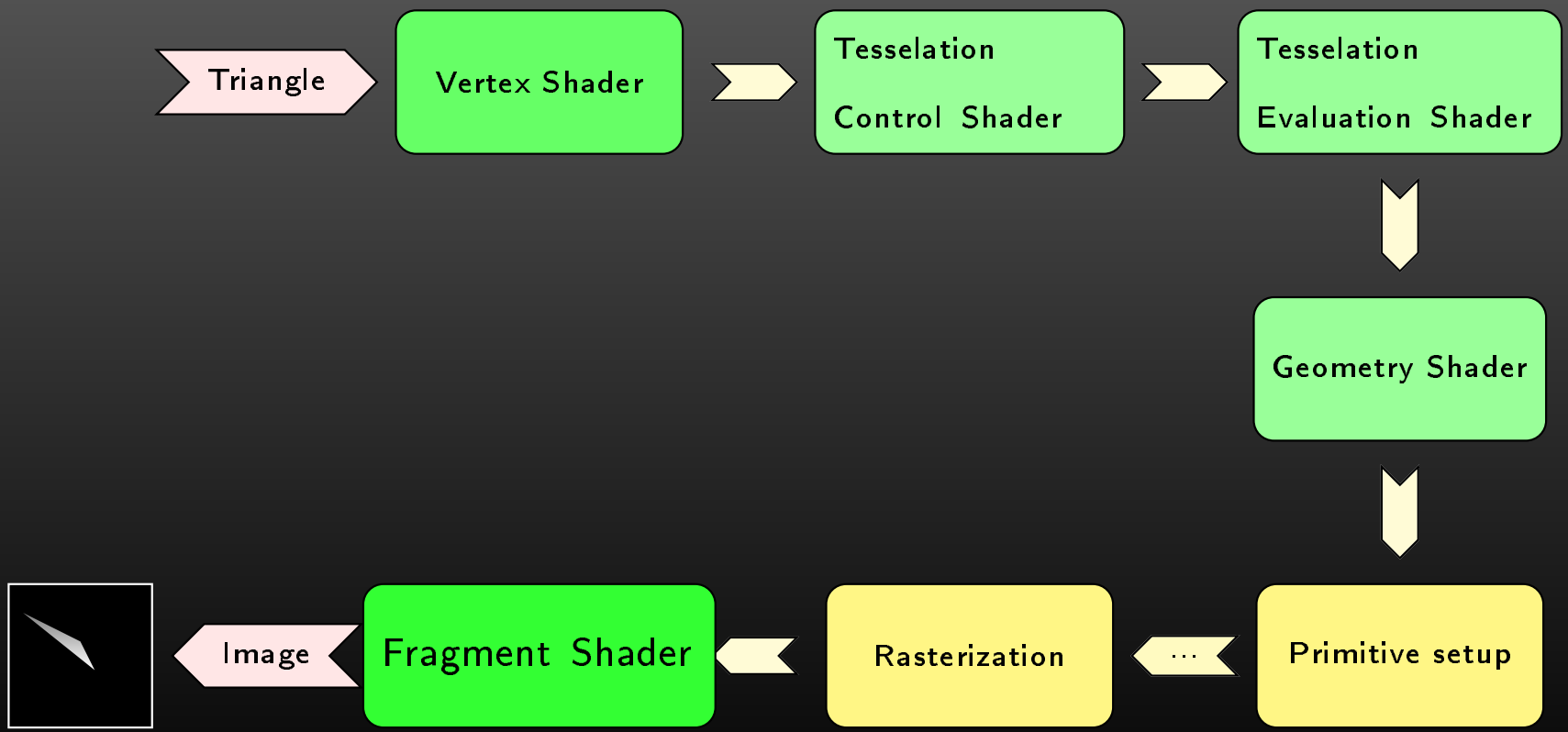
Principe général

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



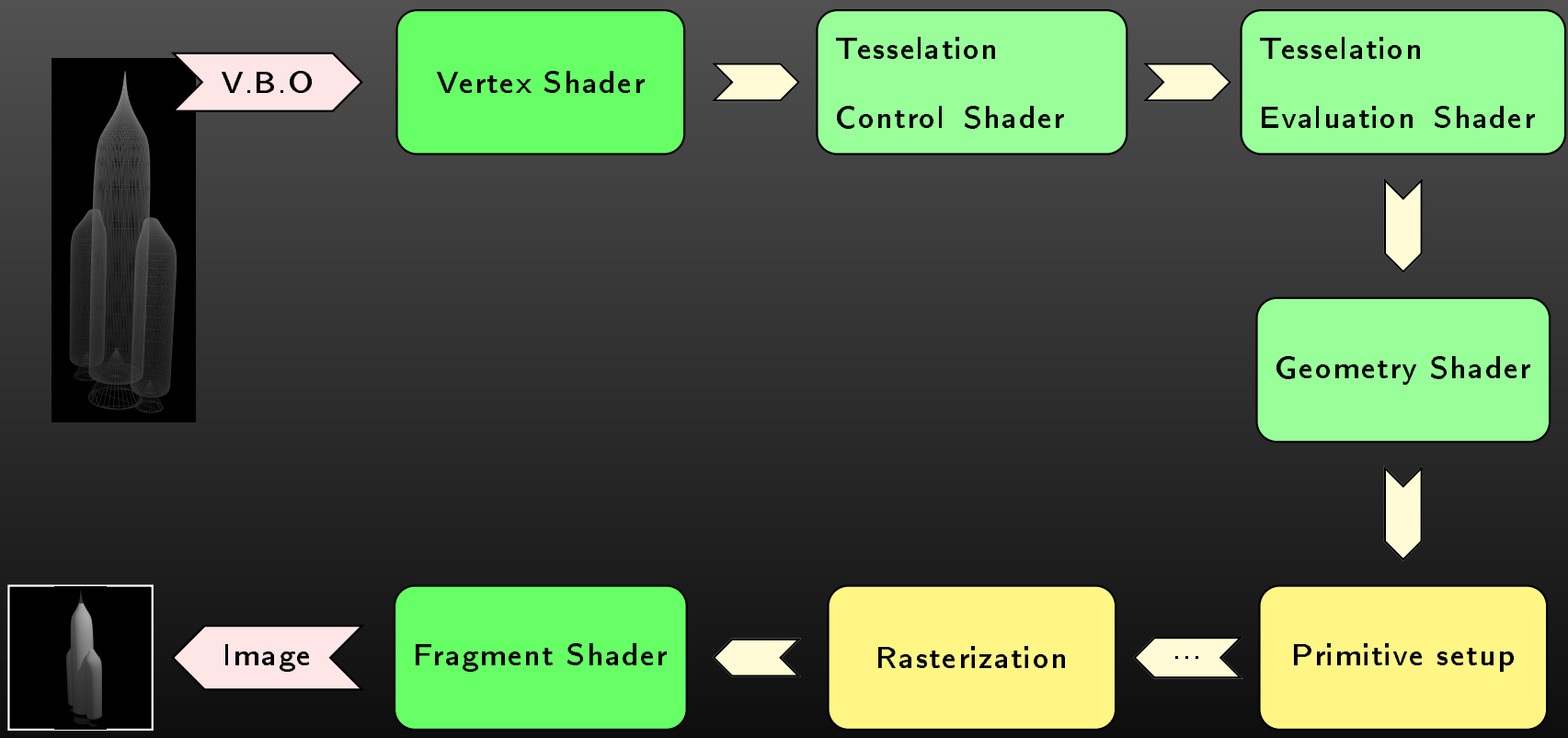
Principe général

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



Principe général

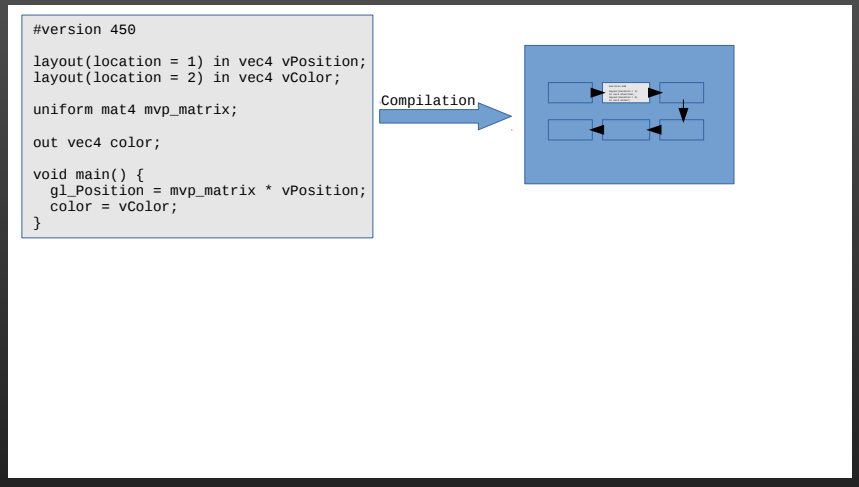
4. Activer le bon programme (*shaders*) puis envoyer les données (buffers) vers le pipeline graphique :
Récupérer l'image.



Principe général

GLSL : Langage utilisé pour écrire les shaders.

- Compilé pour la carte video.

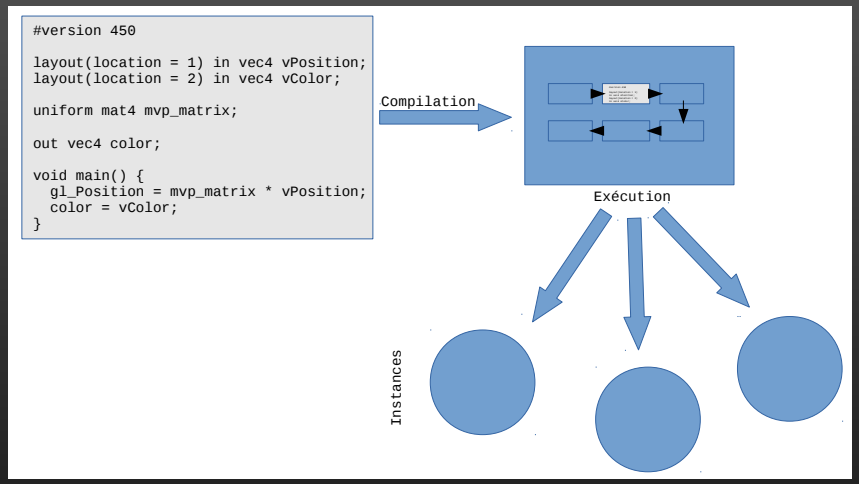


Execution/Instances

Execution des shaders

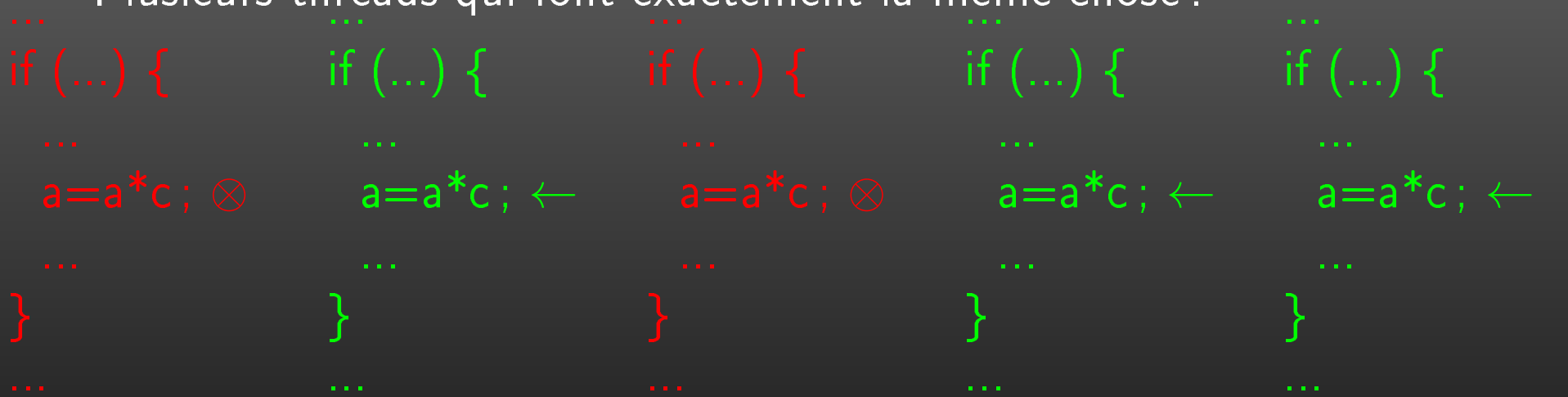
- Single Instruction Multiple Instances

```
#version 450

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

uniform mat4 mvp_matrix;

out vec4 color;

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



Execution des shaders

Plusieurs threads qui font exactement la même chose !



Présentation du langage

GLSL - Variables

Principaux types

- scalaires (limités !) : bool, int, uint, float, double
- vecteurs : bvecn, ivecn, 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.rgba/t.xy/t.xyz...
- operateurs : multiplication matricielle...

Structures

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

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

GLSL - instructions

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

Shaders - Principales données

- 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 - Principales données

- 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 connaitre ou récuperer (`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;
```

Buffers

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...

UBO : Uniform Buffer Object

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 coté Shader.
- Restreint aux types possibles dans GLSL.
- Taille fixée a priori.

UBO : Uniform Buffer Object

Exemple :

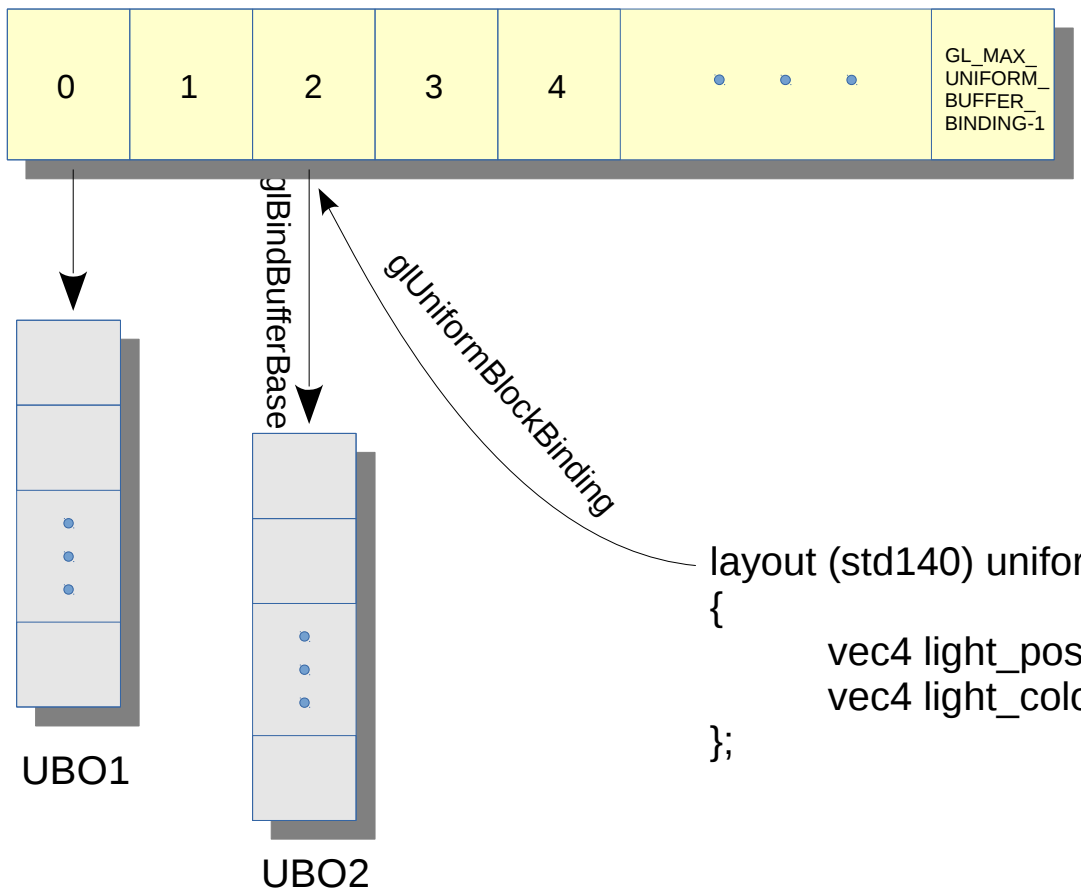
- On peut remplacer :

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

par :

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

UBO : Uniform Buffer Object




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

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
- Acces en lecture et/ou écriture
- Taille pas nécessairement fixée dans le shader.

Limitations :

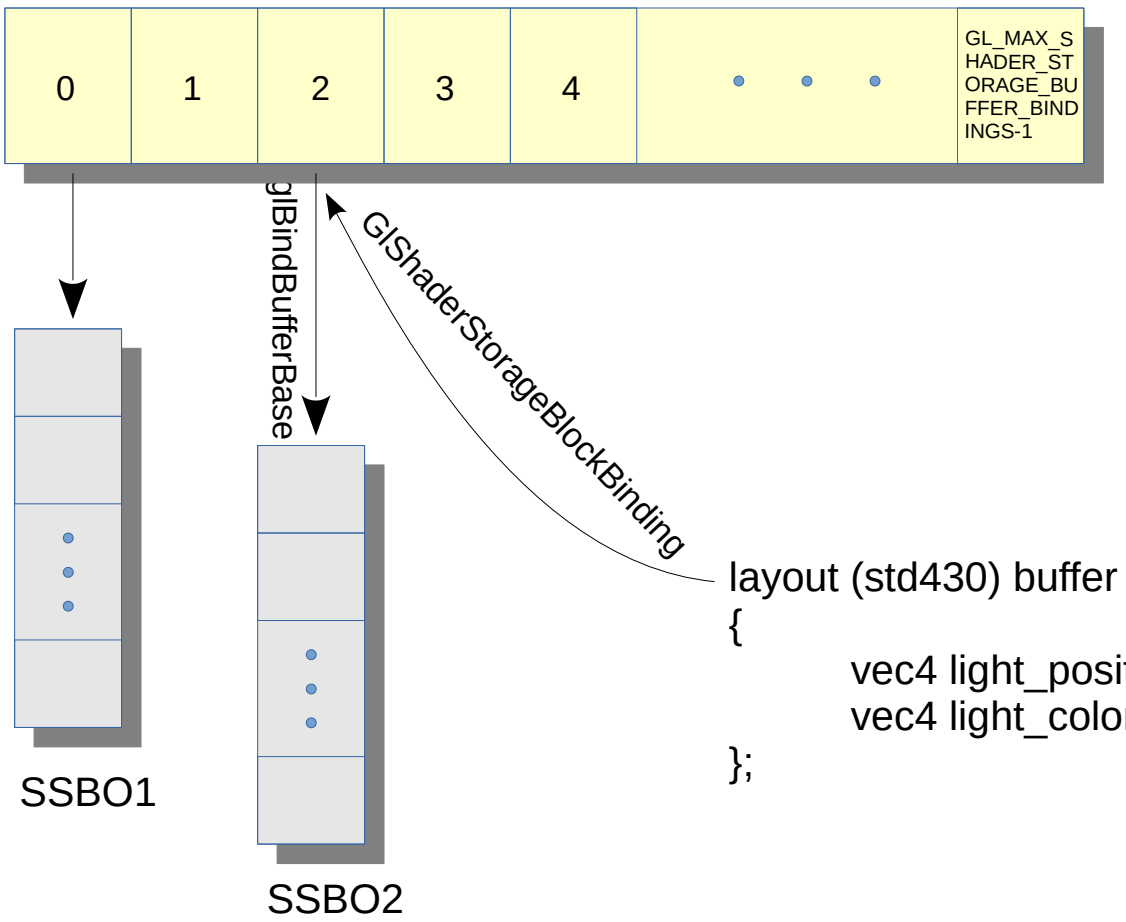
- Seulement depuis Opengl 4.3
- Nombre maximum de SSBOs (GL_MAX_SHADER_STORAGE_BUFFER_BINDINGS)
- Limite de taille (16Mo garanti, en pratique pas -> taille de la mémoire libre) (GL_MAX_SHADER_STORAGE_BLOCK_SIZE)
- Nombre limité de buffers actifs par type de shaders (L_MAX_???_SHADER_STORAGE_BLOCKS) +GL_MAX_COMBINED_SHADER_STORAGE_BLOCKS
- 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.

SSBO et UBO : Alignement des données

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

Coté CPU

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

Coté GPU

```
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];
};
```

SSBO et UBO : Alignement des données

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

Attention à l'alignement des données coté shader ! ! !

Coté CPU

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

Coté GPU

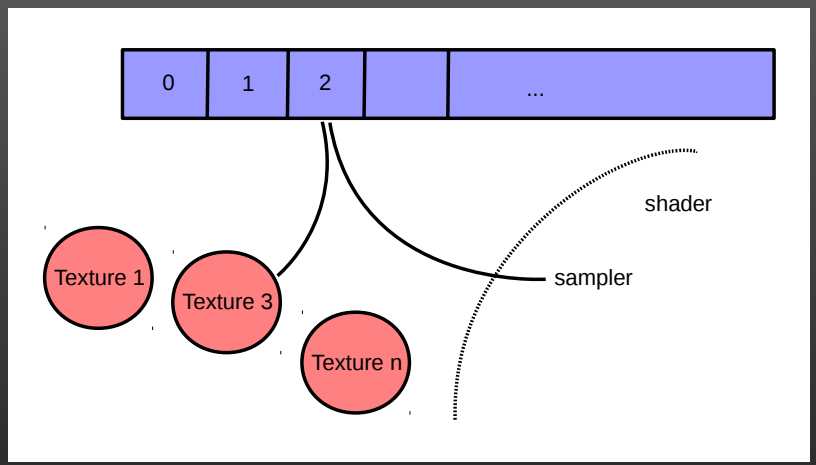
```
struct Line {
  vec3 pos;
  float prop1;
  vec3 color;
  float prop2;
};
```

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

Textures

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

Textures



- 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 = texture2D(tex1_sampler, interpolated_uv_position)
           +texture2D(tex2_sampler, interpolated_uv_position);
...
```

Textures

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_TEXURE_2D, GL_TEXTURE_CUBE_MAP...). Apres il n'est plus posssible 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

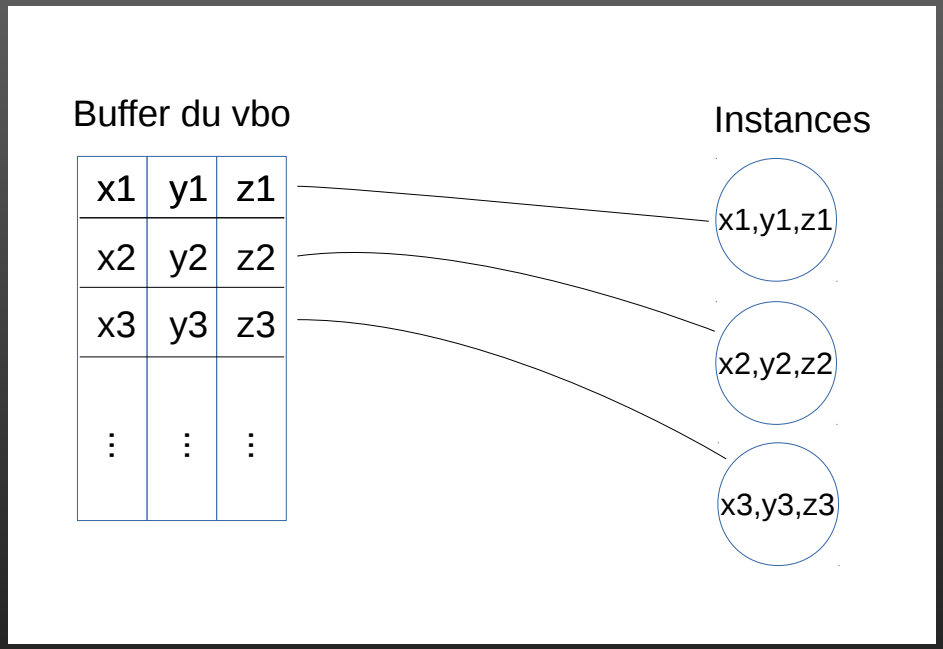
```
glBindImageTexture(image_unit_in_id0, texture_in_id0, 0, GL_FALSE, 0,
                   GL_READ_ONLY, GL_RGBA8UI);
glBindImageTexture(image_unit_out_id1, texture_out_id, 0, GL_FALSE, 0,
                   GL_WRITE_ONLY, GL_RGBA8UI);
```

```
#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);
}
```

VBO

- 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


Buffer du vbo
Instances
x1
y1
z1
x2
y2
z2
x3
y3
z3
x1,y1,z1
x2,y2,z2
x3,y3,z3

VBO

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

xyzxyzxyzxyz...xyz
rgbrgbrgbrgb...rgb
ststststststst...st

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);%//xyz
```

stride = 0 => données consécutives

VBO

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

xyzxyzxyzxyz...xyzrgbrgbrgbrgb...rgbststststststst...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));//rg
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
                      8*sizeof(GLfloat), (void*)6*sizeof(GLfloat));//st
```

Note : Attention, `glVertexAttribIPointer (...)` != `glVertexAttribPointer (..., GL_INT, ...)`

VBO/VAO

Vertex Array Object
Encapsule les VBOs (**Obligatoire en OpenGL 4 ! ! !**)

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

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

VBO

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

VBO

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

VBO

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, ...) puis glBindBuffer(GL_ARRAY_BUFFER, ...) sur le meme buffer.

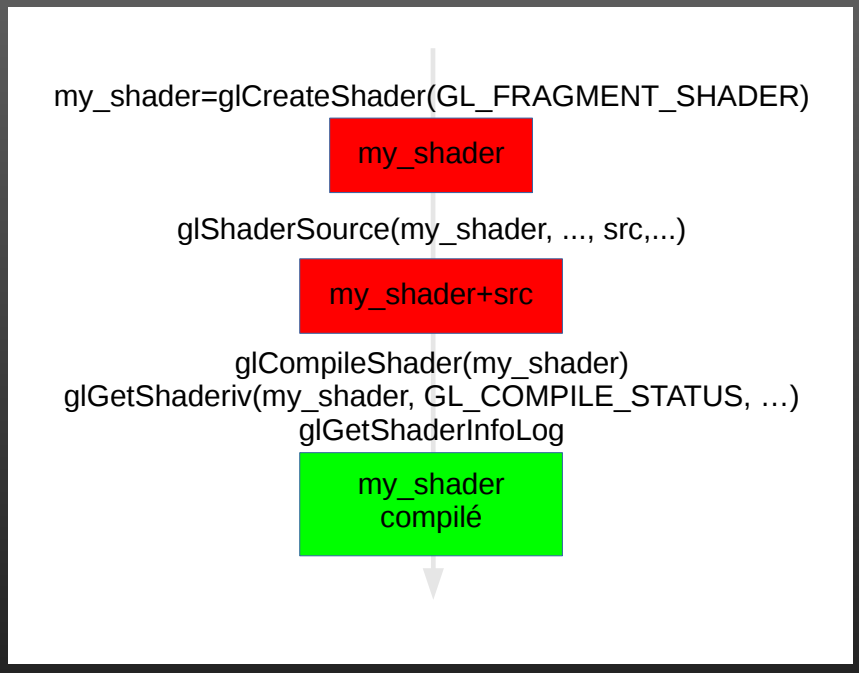
Shaders - Communications avec et entre shaders

- in/out
- shared

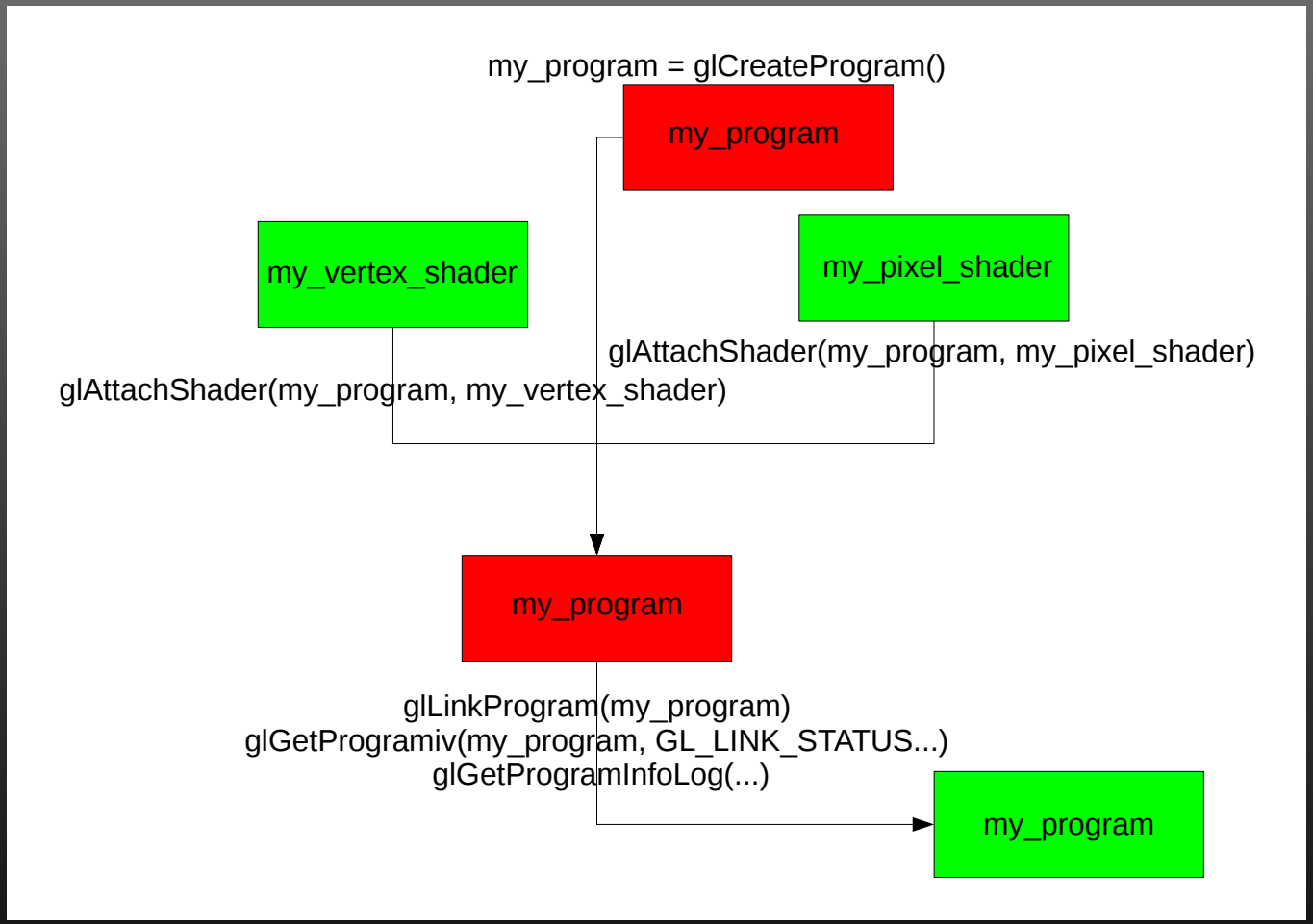
Shaders - Principales données (Rappel)

- 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

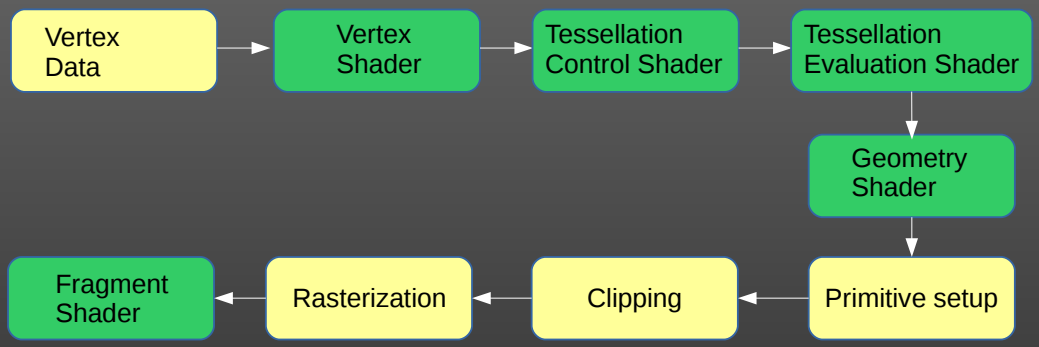


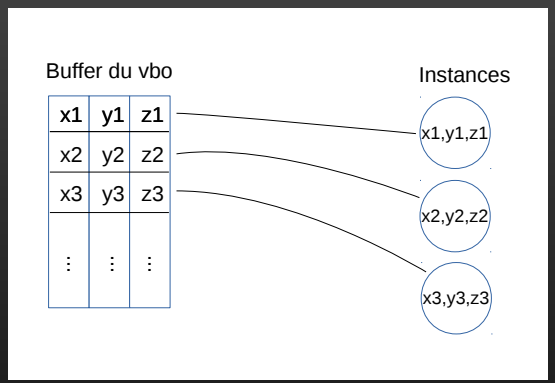
Shaders : Edition de liens



Activation : glUseProgram(...)

Vertex Shader





Vertex Shader

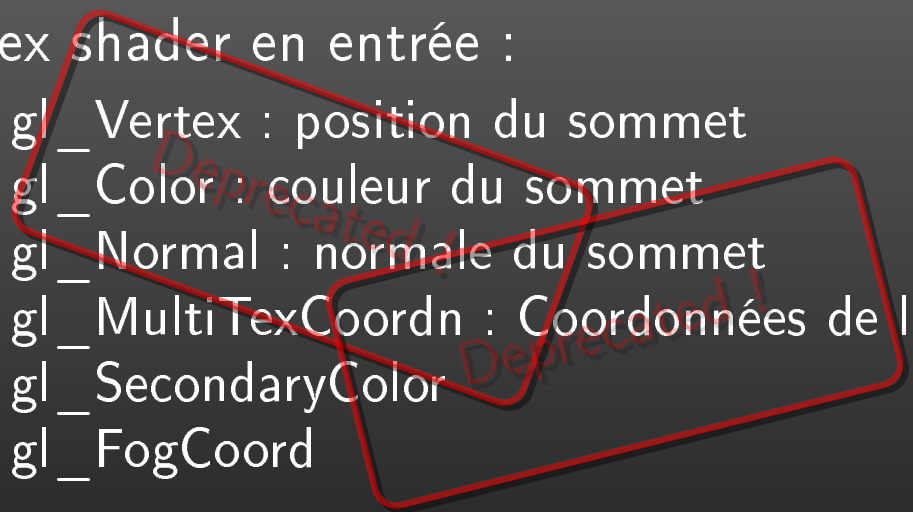
- Vertex shader en entrée :
  - gl_Vertex : position du sommet
  - gl_Color : couleur du sommet
  - gl_Normal : normale du sommet
  - gl_MultiTexCoordn : Coordonnées de la texture n
  - gl_SecondaryColor
  - gl_FogCoord

Vertex Shader

- 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

- 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

- Vertex shader en sortie :
  - gl_Position : position (*to normalized device coordinates*) du sommet
  - gl_PointSize
  - gl_FrontColor/gl_BackColor : couleur du sommet
  - SecondaryColor...
  - gl_TexCoordn : Coordonnées de la texture n
  - gl_FogFragCoord
  - gl_ClipVertex

Vertex Shader

- Vertex shader en sortie :
  - gl_Position : position (*to normalized device coordinates*) du sommet
  - gl_PointSize
  - gl_FrontColor/gl_BackColor : couleur du sommet
  - SecondaryColor...
  - gl_TexCoordn : Coordonnées de la texture n
  - gl_FogFragCoord
  - gl_ClipVertex

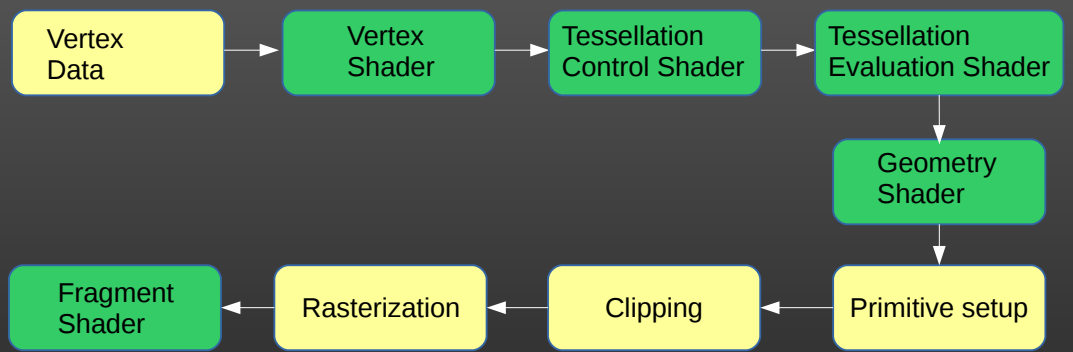
Deprecated !

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;
}
```

Fragment shader



Fragment shader

- Fragment shader en entrée :
  - gl_FragCoord : Coordonnée (écran) du pixel
  - gl_Color : couleur du sommet
  - Secondary color
  - gl_TexCoordn
  - gl_FogFragCoord

Fragment shader

- Fragment shader en entrée :
  - gl_FragCoord : Coordonnée (écran) du pixel
  - gl_Color : couleur du sommet
  - Secondary color
  - gl_TexCoordn
  - gl_FogFragCoord

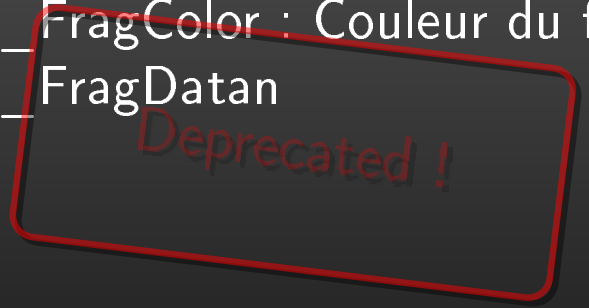
Deprecated !

Fragment shader

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

Fragment shader

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



Fragment shader

Utililsation 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);
}
```

Shaders : exemples

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);
}
```

Sharders : exemples

Quelques remarques :

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

OpenGL

OpenGL : Types et conventions

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)...

OpenGL : détection d'erreurs

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 replace la variable d'état erreur à GL_NO_ERROR après interrogation.

OpenGL : Interaction

Interaction avec OpenGL :

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

Dans la pratique

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

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

Compléments utiles :

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

Glut (freeGlut)

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*...);
```

Glut (freeGlut)

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 :


```
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);
  ...
}

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_
glVertexAttribPointer(vertex_position_location, 3, GL_FLOAT, GL_FALSE, 6*sizeof(
glEnableVertexAttribArray(vertex_position_location);

glVertexAttribPointer(vertex_normal_location, 3, GL_FLOAT, GL_FALSE, 6*sizeof(G
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_
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

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_postion;
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);
  float coef = dot(normalize(vertex_normal.xyz), normalize(light_dir.xyz));
  coef = clamp(coef, 0, 1);
  interpolated_color = vec4(vec3(1, 1, 1)*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 = texture2D(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...

Conclusion

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