

OpenGL > 3.3

- Rappel sur les versions et la gestion des extensions
- Les shaders = intervenir dans le pipeline graphique
- Changement de mode de programmation : + de travail, + de souplesse, + rapide
 - Envoi des données au GPU
 - Gestion des transformations
 - Communication entre pg CPU / GPU
- **Mais les bases restent les mêmes**
 - Modèle géométrique : sommets, faces, normales,
 - Textures : coordonnées de textures
 - Rendu : modèle d'illumination

Les extensions OpenGL

- OpenGL = ensemble normalisé de fonctions pour la 2D et 3D
 - se traduit par des APIs
- La norme OpenGL permet aux différents fabricants d'ajouter de nouvelles fonctionnalités sous forme d'extensions.
- Les fabricants (NVIDIA, AMD, Intel,...)
 - améliorent les performances
 - et ajoutent de nouvelles fonctionnalités :
- « Chaque » carte graphique possède ses spécificités
- Donne lieu à des fonctionnalités spécifiques : **les extensions**
- => Différences selon les fabricants, les plateformes
 - Fonction pas normalisée (appel)
 - La bazar quoi : développement, maintenance du code
 - **Pas répertoriée dans l'API OpenGL gl.h** (même si présent dans la librairie)

Version API OpenGL/GLSL

- OpenGL 4.5
 - OpenGL Shading Language 4.50 Specification
- OpenGL 4.4
 - OpenGL Shading Language 4.40 Specification
- OpenGL 4.3
 - OpenGL Shading Language 4.30 Specification
- OpenGL 4.2
 - OpenGL Shading Language 4.20 Specification
- OpenGL 4.1
 - OpenGL Shading Language 4.10 Specification
- OpenGL 4.0
 - OpenGL Shading Language 4.00 Specification
- OpenGL 3.3
 - OpenGL Shading Language 3.30 Specification
- OpenGL 3.2
 - OpenGL Shading Language 1.50 Specification
- OpenGL 3.1
 - (with GL_ARB_compatibility extension)
 - OpenGL Shading Language 1.40 Specification
- OpenGL 3.0
 - OpenGL Shading Language 1.30 Specification
- OpenGL 2.1
 - OpenGL Shading Language 1.20 Specification
- OpenGL 2.0
 - OpenGL Shading Language 1.10 Specification
- OpenGL 1.x
 - **PAS de SHADER**
- Older GLX Specifications
 - GLX 1.3 Specification
 - GLX 1.3 Protocol Encoding Specification
 - GLX 1.2 Specification (PostScript format)
 - GLX Protocol Slides (PostScript format; only of historical interest)
 - OpenGL Utility Library (GLU) Specification
 - GLU 1.3 Specification (November 4, 1998)

Le dernier né

- VULKAN
 - `sudo apt install vulkan-utils`
 - `vulkaninfo`
- Réécriture des APIs
- Plus performant, mieux adapté aux nouvelles générations de GPU
- <https://vulkan-tutorial.com/>

Les extensions OpenGL

- Le nom de chaque extension contient des informations sur sa disponibilité :
- GL_ : toutes les plates-formes
- GLX_ : Linux et Mac seulement (X11)
- WGL_ : Windows seulement
- EXT : une extension générique (définie par plusieurs fabricants)
- NV/AMD/INTEL (propre à chaque fabricant)
- ARB : l'extension a été acceptée par tous les membres du OpenGL Architecture Review Board (les extensions EXT sont souvent promues ARB au bout d'un moment)

Accès au extension openGL

- On utilise une « OpenGL Loading Library »
 - Permet de charger les pointeurs vers les fonctions openGL au cours du runtime : pour le noyau comme les extensions
 - Extension loading libraries
 - Permet d'abstraire les différences entre
 - Les GPUs (spécificités des fabricants et des modèles de GPU)
 - les mécanismes de chargement des différentes plateformes.

Accès au extension OpenGL

- Exemples de bibliothèques de chargement d'OpenGL
 - **GLEW** = OpenGL Extension Wrangler library
 - Comme pour la plupart des « loaders » d'extension
 - **Ne pas** inclure gl.h, glext.h, ou autre fichier d'entête « gl » avant glew.h,
 - Sinon message d'erreur « gl.h included before glew.h.
 - Il n'est plus nécessaire d'inclure gl.h : **glew.h remplace gl.h.**
 - The **GL3W** library
 - Est spécifique au noyau OpenGL 3 and 4.
 - **GLAD**
 - ...

Les bibliothèques requises

- Glut (freeGlut) ou SDL ou
 - Gestion des fenêtres et événements
- GLEW : OpenGL Extension Wrangler library
 - Accès aux extensions d'OpenGL
- GLM : OpenGL math library
 - Structure de données, syntaxe très proche de GLSL
 - Surtout pour la gestion des transformations

OpenGL > 3.3 : les shaders

- On met tout à la poubelle :(
 - Plus de glVertex, glNormal, glColor, glTexCoord
 - Plus de gltranslate, glScale,..., glPushMatrix,..
 - Plus glLight, glMaterial,
 - ...
 - Mais reste compatible
- Pour aller plus vite et avoir plus de souplesse
 - Grace aux shaders = programme envoyé au GPU

Tester la version d'OpenGL

- Directement via info du GPU
 - Sous windows
 - <https://support.esri.com/en/technical-article/000011375>
 - Sous linux :
 - `glxinfo | grep "OpenGL version"`
 - Si glxinfo n'est pas disponible installer le package « mesa-utils »

Tester la version d'OpenGL

```
#include <GL/glut.h>
#include <iostream>
int main(int argc, char **argv)
{
    /* initialisation de glut */
    glutInit(&argc,argv);
    // créer la fenetre en un context opengl attaché
    // nécessaire car la fonction glGetString donne des infos sur le context
    opengl Courant
    glutCreateWindow("TORE VBO SHADER ");

    //info version oenGL / GLSL :
    std::cout << std::endl<< "***** Info GPU *****" << std::endl;
    std::cout << "Fabricant : " << glGetString (GL_VENDOR) << std::endl;
    std::cout << "Carte graphique: " << glGetString (GL_RENDERER) << std::endl;
    std::cout << "Version : " << glGetString (GL_VERSION) << std::endl;
    std::cout << "Version GLSL : " << glGetString (GL_SHADING_LANGUAGE_VERSION)
    << std::endl << std::endl;
    return 0;
}
```

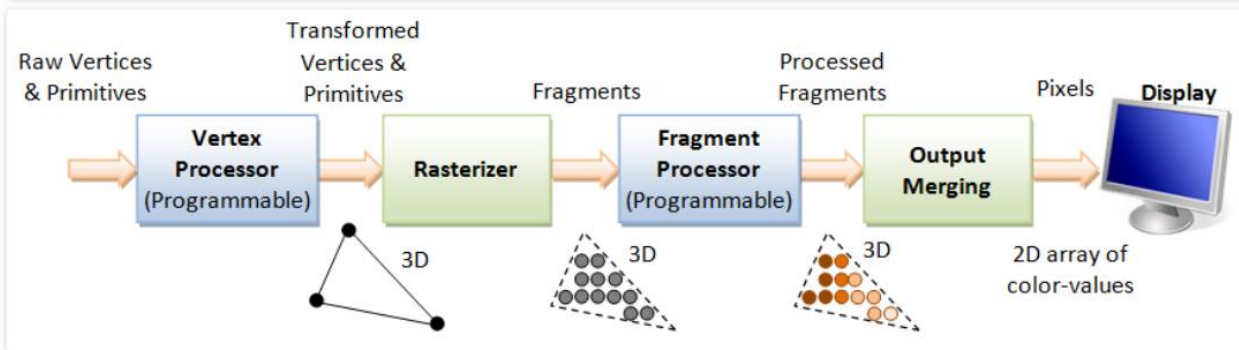
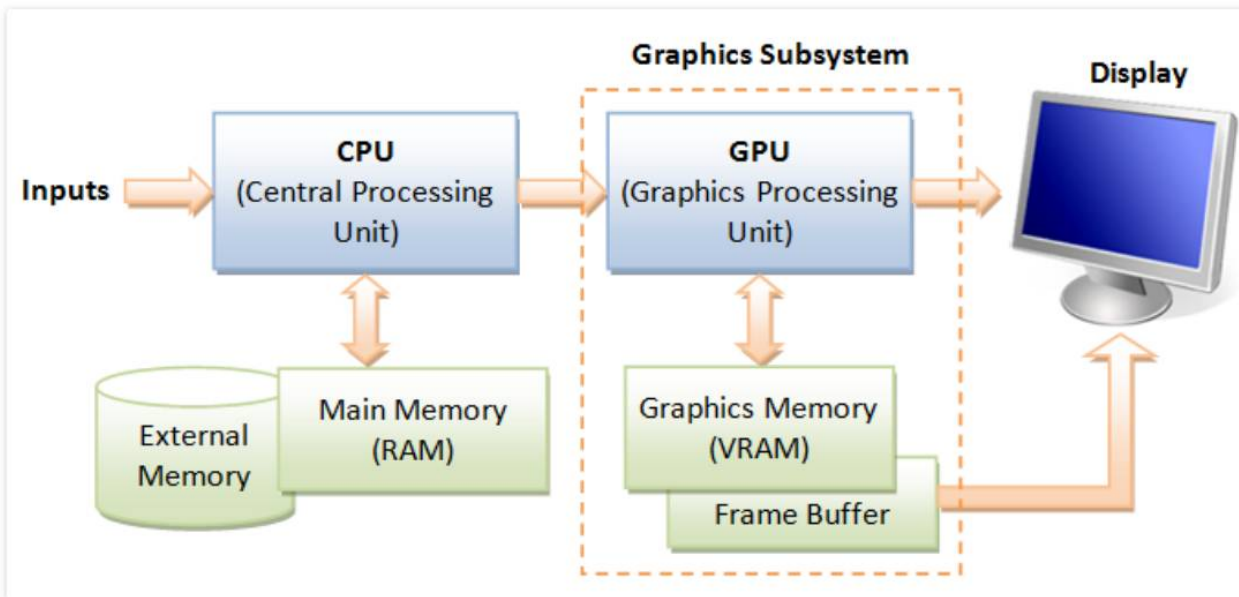
Exercice / préparation TP

- Exercice 1.1 :
 - Récupérer sur plubel le fichier starterKitShader (GLUT ou SDL)
 - Ajouter le code pour déterminer votre version d'openGL

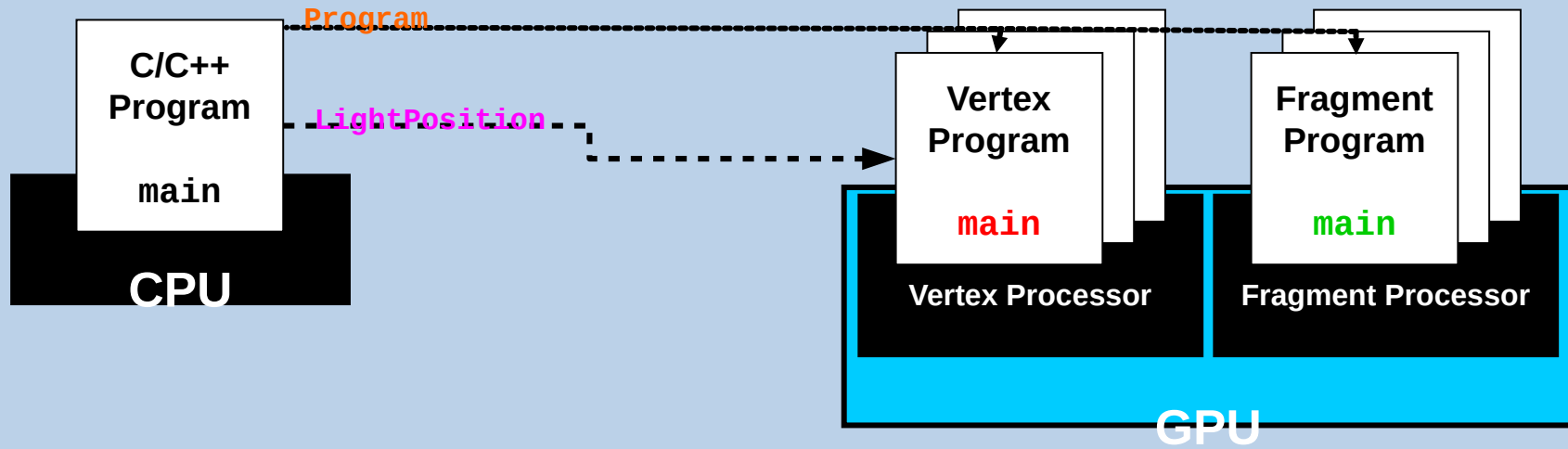
Si pb voir installation d'OpenGL

- https://www.khronos.org/opengl/wiki/Getting_Started#Downloading_OpenGL
- Ubuntu (par défaut rien à faire : déjà installé)
 - Installation de glut
 - `sudo apt-get install freeglut3`
 - `sudo apt-get install freeglut3-dev`
 - Installation de GLEW
 - `sudo apt-get install glew-utils`
 - `sudo apt-get install libglew-dev`

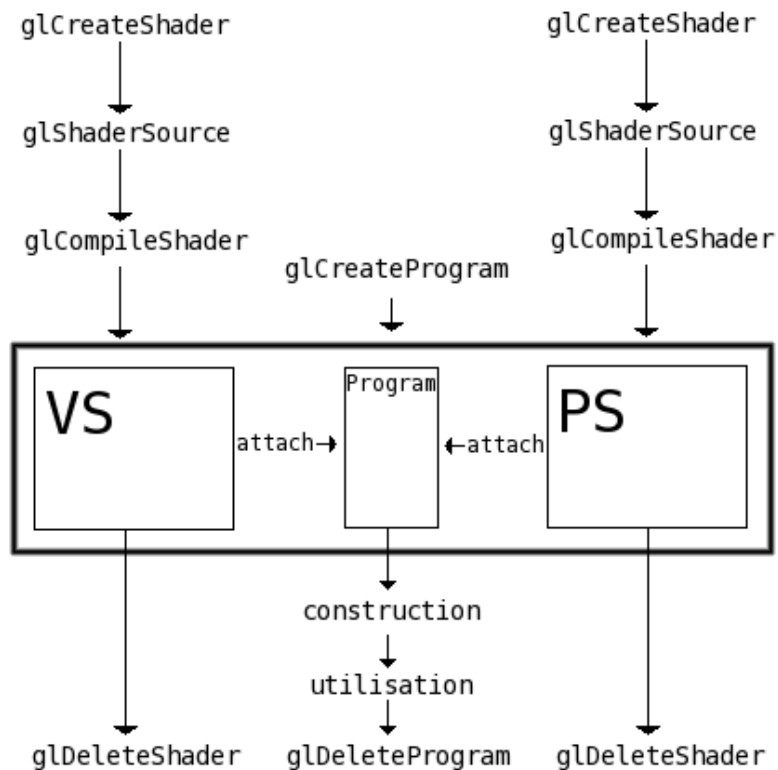
Rôle des shaders



Ca marche comment ?



Shader : création, compilation



```
VShader = glCreateShader(GL_VERTEX_SHADER);  
glShaderSource(VShader, 1, (const GLchar**) &VSource,  
              NULL);  
glCompileShader(VShader);
```

```
FShader = glCreateShader(GL_FRAGMENT_SHADER);  
glShaderSource(FShader, 1, (const GLchar**) &FSource,  
              NULL);  
glCompileShader(FShader);
```

```
IdProgram = glCreateProgram();  
glAttachObject(IdProgram, VShader);  
glAttachObject(IdProgram, FShader);  
glLinkProgram(IdProgram);
```


Shader : qu'est ce qu'on y fait

- Ce qu'on veut (ou ce qu'on peut)
- Vertexshader (programme)
 - Opère sur les sommets
 - Transformation : Modèle, vue, perspective
 - Déformation, animation, ...
 - Calcul d'illumination, déplacement map
- FragmentShader (programme)
 - Opère sur les fragments
 - Calcul d'illumination
 - Calcul de couleur (transparence, application des textures, color, normal,...)
- OK mais comment on récupère les données sur les sommets ?
 - Dans le pg OpenGL :
 - On mets les données dans des tableaux que l'on copie dans des « VBO » (tableau sur GPU)
 - Ou on récupère des références sur des variables que l'on réserve sur le GPU et on envoie les données via ces références.

Comment transmettre les données au GPU

- Il y a deux « types » de données
 - Les données propres
 - à chaque sommets
 - Positions, couleurs, normales, coordonnées de textures,...
 - ou à chaque fragment
 - Tout ce qui est en sortie du vertexShader
 - Les données communes
 - Transformations : modèle, vue, perspective
 - Propriétés des lumières : position, couleur,
 - ...

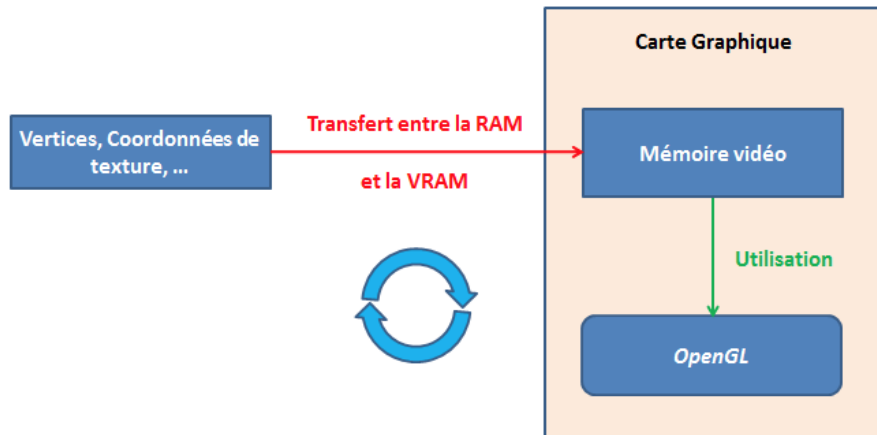
Envoi des données CPU vers GPU

- Les données spécifiques aux sommets sont envoyées au GPU dans des tableaux
 - VBO = vertex Buffer Objects
 - Pour chaque sommet
 - le GPU passe au vertexShader les informations du sommet
 - Provenant du VBO (ou des VBO, si plusieurs).
- Par la suite les informations des sommets sont assemblées pour définir des faces
 - suivant le IBO : index buffer objects
- Les caractéristiques des VBO, IBO, sont stockées dans un VAO sur GPU

Intérêts

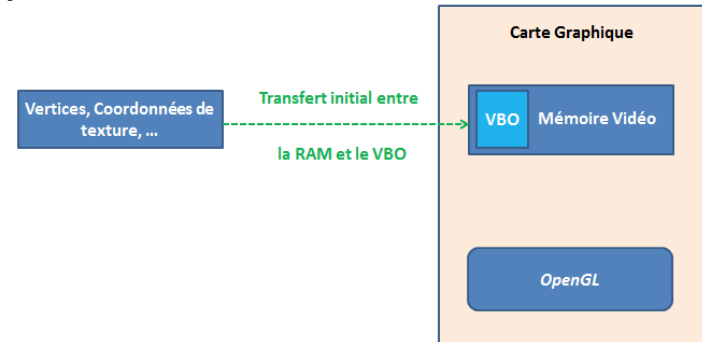
Avant

Boucle d'affichage

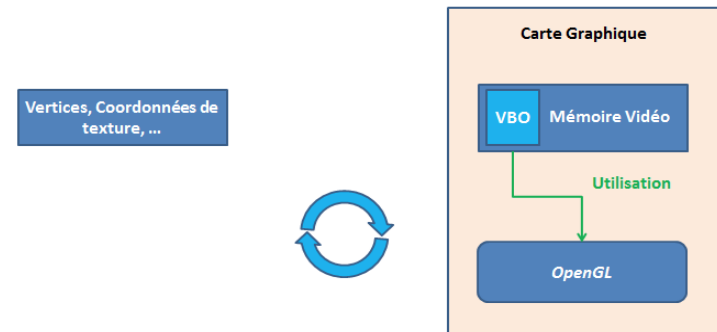


Après

Chargement du modèle 3D

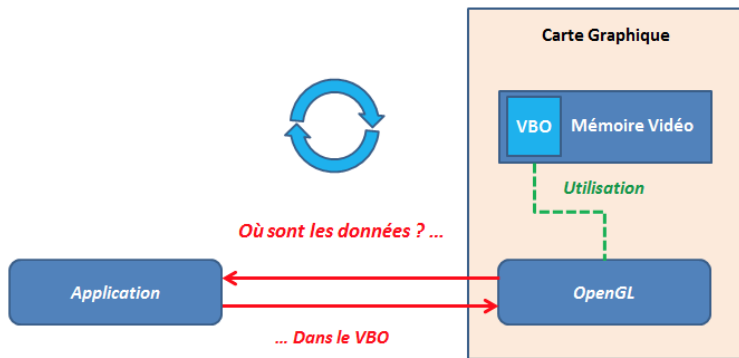


Boucle d'affichage

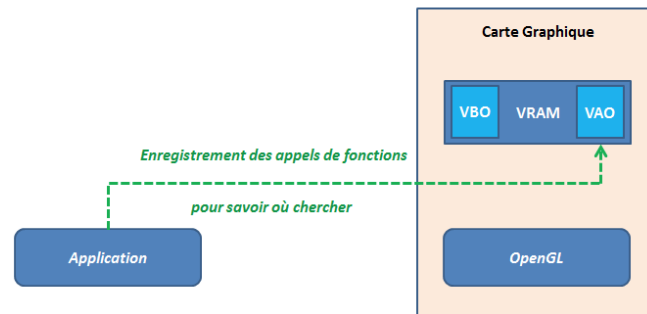


Encore mieux

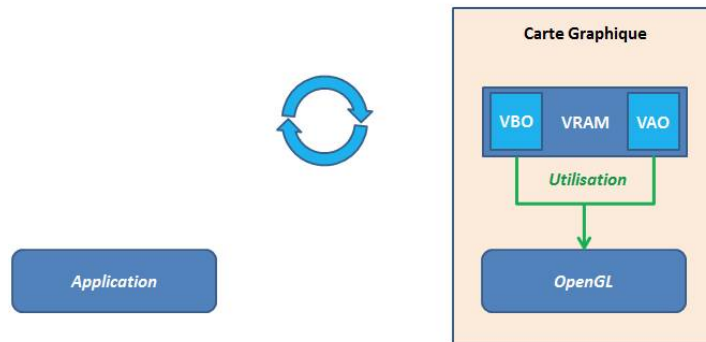
Boucle d'affichage



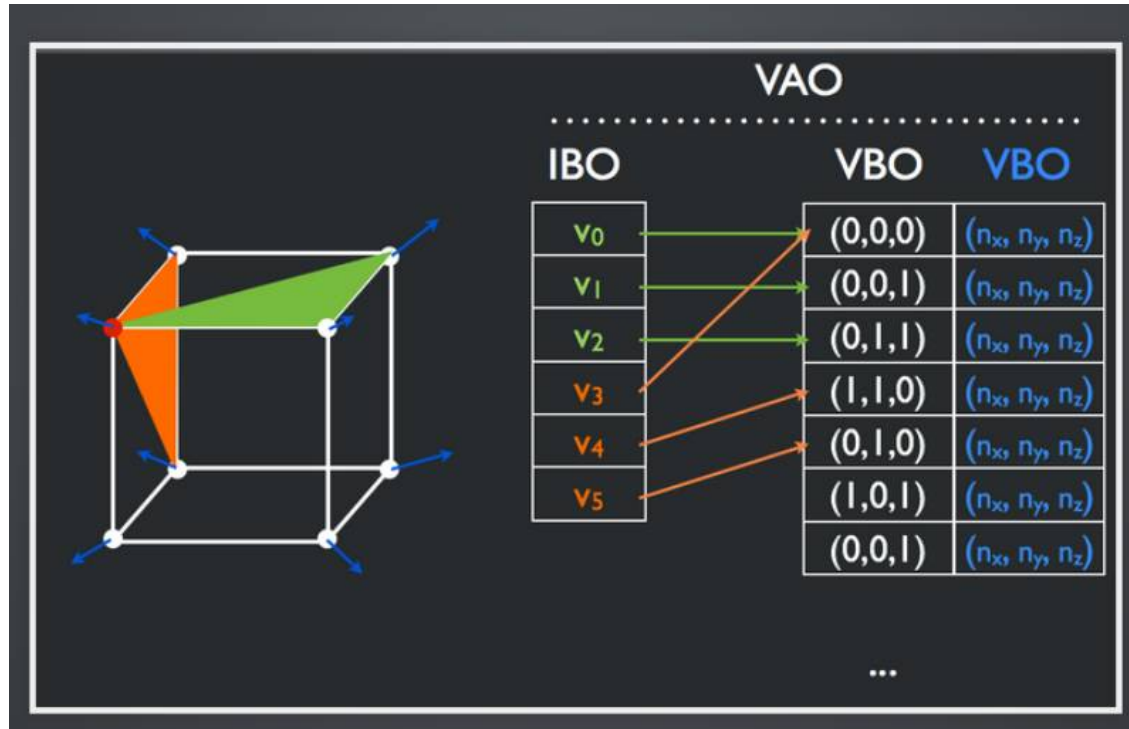
Chargement du modèle 3D



Boucle d'affichage



Organisation des VBO / IBO



Organisation VBO, IBO => VAO

- Les VBO sont fait pour pouvoir y mettre ce qu'on veut (gestion générique)
 - => différents types d'attributs et différentes organisations mémoires.
 - = > Il faut expliciter cette organisation pour le GPU puisse la traiter.
Cela est à l'aide d'un VAO - Vertex Array Object.
- Le VAO stocke
 - le format des données des sommets (l'organisation mémoire au sein de chaque buffer)
 - les **références** vers les VBOs
 - Permet au GPU de tranférer les données aux shaders
 - En GLSL (dans le shader) on utilise l'index de la commande « layout »

https://www.khronos.org/opengl/wiki/Vertex_Specification

Précision sur les buffers du GPU

Ces buffers = VBO - Vertex Buffer Object (ou IBO).

- Les VBO sont similaires à des pointeurs C: blocs de données contigus
- VBO est référencé par un identifiant (GLuint) sur le GPU.
- la valeur du VBO est un identifiant d'un buffer sur le GPU.
- Les données d'un VBO peuvent contenir n'importe quoi (faisant référence à un sommet):
 - les coordonnées des sommets,
 - leurs attributs (couleurs, coordonnées de textures, etc)
- **Ces données seront des variables d'entrée dans le vertex shader.**
 - **Chaque appel du vertex shader est lié à un sommet (un appel par sommet)**
 - **Dans le shader, il faudra récupérer les données **du** sommet concerné.**

http://imagecomputing.net/damien.rohmer/teaching/2018_2019/semester_2/inf443_graphique_3d/td/02_opengl/content/004_envoie_de_donnees_sur_la_carte_graphique/index.htm

[https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_\(C_/_SDL\)](https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL))

https://www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Array_Object

Remarque sur les formats de données

- GLSL = Langage de programmation des shader
 - GLSL réalise ces calculs en flottants simples précisions.
 - Pour éviter des conversions supplémentaires :
 - Dans le code C++ il est préférable d'utiliser également la simple précision
 - i.e. float et non pas double.
 - OpenGL définit le type GLfloat pour assurer la cohérence d'encodage pour toutes les architectures.
- => et C++ le plus simple est de stocker des données en tant que **std::vector<float>**
où les éléments sont stockés de façon contiguë.

Enfin un exemple : data

```
// *****  
//          2 - Sending data to GPU          //  
// ***** //  
std::cout<<"*** Setup Data ***"<<std::endl;  
// 2.1 Setup contiguous array of floating point value  
// ***** //  
//      Here the coordinates of the vertices position  
const std::vector<GLfloat> position = {  
    -0.5f, -0.5f, 0.0f,  
     0.5f, -0.5f, 0.0f,  
     0.0f,  0.5f, 0.0f  
};
```

Enfin un exemple : VBO

```
// 2.2 Create VBO - Send data to GPU
// *****
GLuint vbo = 0; // initialisé à 0 = pas d'indice
// Create an empty VBO identifiant (génère le VBO)
glGenBuffers(1, &vbo); // et met dans la variable vbo le handle du vbo

// Setup the current VBO
glBindBuffer(GL_ARRAY_BUFFER, vbo); // on dit avec quel buffer on va travailler (openGL = machine
à états)

// Send data to GPU: Fill the currently designated VBO with the buffer of data
glBufferData(GL_ARRAY_BUFFER, position.size()*sizeof(GLfloat), &position[0], GL_STATIC_DRAW );

// Good practice to set the current VBO to 0 (=disable VBO) after its use
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Enfin un exemple : IBO

```
// 2.2 Create IBO - Send data to GPU
```

```
/ ***** //
```

```
GLuint ibo = 0;
```

```
glGenBuffers(1, &ibo);
```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
```

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices , GL_STATIC_DRAW);
```

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Enfin un exemple : VAO

```
// 2.3 Create VAO - Relation between VBO organization and input variables of shaders
// *****
GLuint vao = 0; // déclaration pour le handle du vao
// Create an empty VAO identifiant
glGenVertexArrays(1,&vao);
// Setup the current VAO
glBindVertexArray(vao); // on dit avec quel vao on va travailler
// Indicate the VBO we will refer to in the next lines
glBindBuffer(GL_ARRAY_BUFFER, vbo);
// Activate the use of the variable at index layout=0 in the shader
glEnableVertexAttribArray(0);
// Define the memory model of this VBO: here contiguous triplet of floating
values (x y z) at index layout=0 in the shader
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
// attache le tableau d'indices.
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); // 1 seul par VAO
// As a good practice, disable VBO and VAO after their use
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Affichage

```
// 2.2 on demande au GPU d'afficher
/ ***** //

// on spécifie avec quel shader on veut afficher
glUseProgram(IdProgram);

// on active le VAO
glBindVertexArray(vao);

// on appelle la fonction dessin
glDrawElements(GL_TRIANGLES, sizeof(indices), GL_UNSIGNED_INT, 0);

// on désactive le VAO
glBindVertexArray(0); // on désactive les VAO
```

Récup coté shader (vertex shader)

```
#version 450
```

```
// location permet spécifier par quel « flux » on récupère les données
```

```
// la valeur de location doit être la même
```

```
// que glEnableVertexAttribArray(0); du CPU
```

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

```
layout(location = 2) in vec3 normal; // idem pour la normale
```

Transfert entre vertexShader / fragmentShader

- Fragment Shader

- 1 seule sortie : la couleur du fragment (contribution au pixel)

```
out vec4 finalColor; // déclaration
```

```
...
```

```
finalColor = vec4(ambient + attenuation*(diffuse + specular),1.);
```

- vertex Shader

- Autant de sortie que l'on veut : exemple

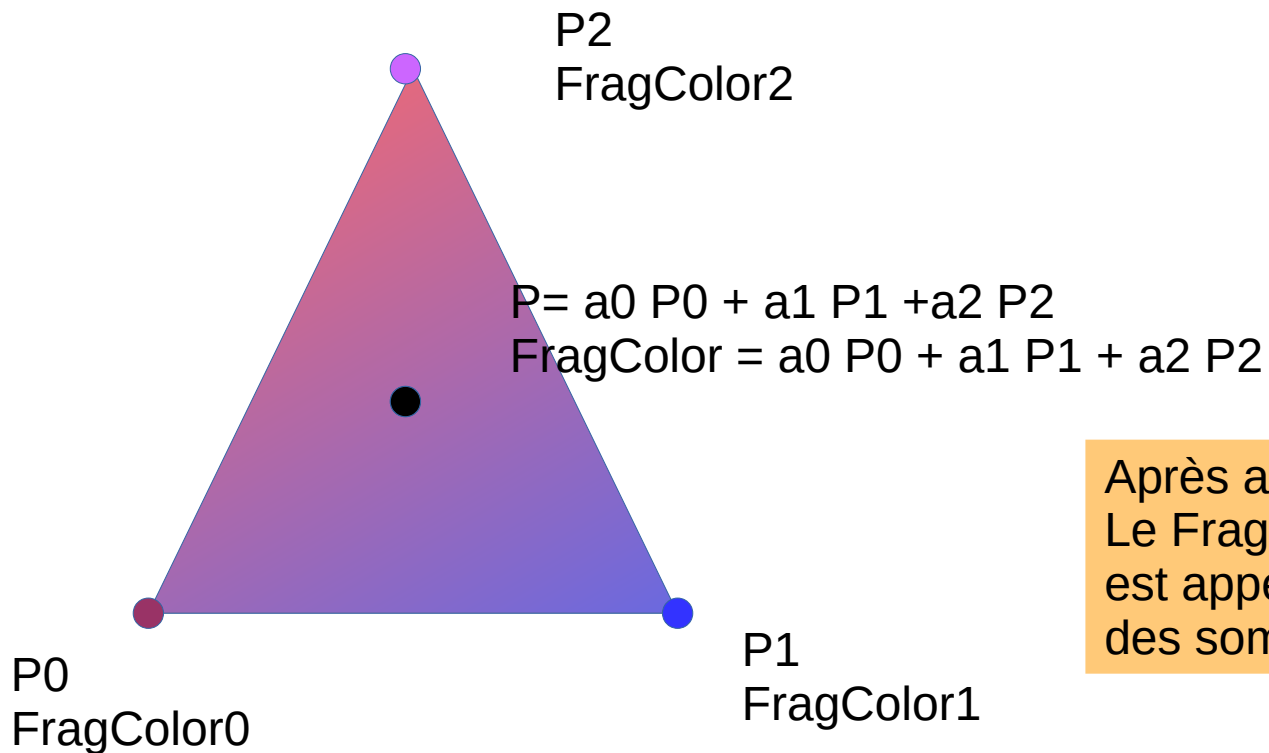
```
out vec3 fragPosition;
```

```
out vec3 fragColor;
```

```
out vec3 fragNormal;
```

- Toutes les variables en sortie de vertex shader se retrouve interpolées dans le fragment shader

Input fragment shader



Après assemblage et rasterisation
Le Fragment shader
est appelé avec les paramètres interpolés
des sommets de la face

Fragment Shader

- Récupération des paramètres (out du vertex Shader)

```
in vec3 fragPosition;
```

```
in vec3 fragColor ;
```

```
in vec3 fragNormal ;
```

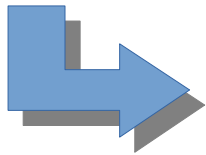
Exercice / préparation TP

- Exercice 1.2 :
 - Faites un tableau contenant les sommets d'un cube
 - Faites un tableau d'index représentant les faces du cube.
 - Créez 1 VBO, un IBO et un VAO contenant les description des données
 - Affichez le VAO.
 - Créez un vertex shader et un fragment shader (diapo 14)
 - Modifiez le fragment shader pour afficher la couleur rouge pour chaque fragment
- Exercice 1.3
 - Modifiez le vertex shader pour faire suivre les positions (x,y,z) des sommets au fragment shader
 - Modifiez le fragment shader pour afficher chaque fragment de la couleur correspondant à ces coordonnées (x,y,z) (i.e. rouge=x, vert = y , bleu = z).

Précision sur les attributs des sommets d'un VAO

- Peut avoir entre 0 to `GL_MAX_VERTEX_ATTRIBS - 1` attributs
- Par défaut ils ne sont pas activés

```
glBindVertexArray(GLuint VAOIndex) ;  
glEnableVertexAttribArray(GLuint attributIndex);
```



- Définit l'état actif de l'attribut numéro «attributIndex»
- Cet état fait partie du VAO (qui lui-même doit être activé préalablement)

RQ : attributIndex à utiliser en GLSL pour récupérer les données

VAO / VBO

```
1)glBindBuffer(GL_ARRAY_BUFFER, buf1);  
2)glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);  
3)glBindBuffer(GL_ARRAY_BUFFER, 0);
```

- Ligne 1 = activation de buf1 dans le GL_ARRAY_BUFFER activé.
- Ligne2 = spécifie que l'attribut (index 0) a son tableau de données dans **buf1**, et définit comment y accéder.
- ATTENTION : l'association de l'attribut au buffer est faite grâce à
 - 1) l'activation du buffer (glBindBuffer)
 - 2) l'appel glVertexAttribPointer
 - 3) Et tout ça, doit être fait dans le VAO actif (glBindVertexArray)

VBO et glVertexAttribPointer

- Les attributs des sommets peuvent être gérés de différentes manières via les VBO
 - 1 buffer par attribut
 - 1 buffer contenant tous les attributs
 - Les positions, les couleurs, les normales
 - Les positions, les normales, les couleurs
 - Un mixte des deux
- Il faut spécifier au GPU comment sont structurées les données pour qu'il puisse y accéder
 - Fonction **glVertexAttribPointer**

Spécification de l'accès aux données du buffer

- void **glVertexAttribPointer**(index, **size**, type, normalized, **stride**, **pointer**);
 - Index (location)
 - **size** = nombre de composants par sommet
valeur = 1, 2, 3 ou 4
 - type : type des données du buffer
 - GL_BYTE, GL_UNSIGNED_BYTE, GL_FLOAT...
 - normalized : les données doivent être normalisée ou pas
 - GL_TRUE / GL_FALSE.
 - **stride** = pas (en byte) entre deux de données de sommets
 - **pointer** = Décalage dans le buffer pour la première composante

Spécification de l'accès aux données du buffer

- Précision sur le paramètre pointer
 - If pointer is not NULL, a non-zero named buffer object must be bound to the GL_ARRAY_BUFFER target (see glBindBuffer),
 - otherwise an error is generated. pointer is treated as a byte offset into the buffer object's data store.

VBO : 1 buffer par attribut

Buffer 1

x	y	z	x	y	z	x	y	z	x	y	z
---	---	---	---	---	---	---	---	---	---	---	---

Stride = 3 * size(GLfloat)

Pointer = (void *)0

Buffer 2

r	g	b	r	g	b	r	g	b	r	g	b
---	---	---	---	---	---	---	---	---	---	---	---

Stride = 3 * size(GLfloat)

Pointer = (void *)0

Buffer 3

u	v	u	v	u	v	u	v	
---	---	---	---	---	---	---	---	--

Stride = 2 * size(GLfloat)

Pointer = (void *)0

VBO : 1 buffer pour tous les attributs

1 seul buffer



attribut position
Stride = 8 * size(GLfloat)
Pointer = (void *)0

attribut couleur
Pointer = (void *) (0 + 3 * sizeof(GLfloat))

attribut coord. Texture
Pointer = (void *) (0 + 6 * sizeof(GLfloat))

MAIS ATTENTION :

1 appel de glVertexAttribPointer
par attribut
Sur cet exemple 3 appels

Aspect technique (programmation)

- Valeur de pointer :

```
struct structureAttribut
{
    GLfloat position[3];
    GLfloat normal[3];
    Glubyte color[4];
};
```

```
reinterpret_cast<void*>(baseOffset + offsetof(structureAttribut, position))
```

- La macro *offsetof* calcule le décalage en byte du champs spécifié pour la structure données
- *reinterpret_cast* fait le cast plus proprement

Exercice / préparation TP

- Exercice 2.1 : (1 buffer pour tous les attributs)
 - Modifier le VAO
 - Pour chaque sommet faire une structure « vertexAttribut »
 - {position, couleur}
 - Affectez des couleurs différentes aux sommets
 - Le tableau (`std::vector`) doit contenir les structures `vertexAttribut`
 - Modifier le VAO pour envoyer à la carte graphique le tableau d'attribut des sommets
 - Modifier le vertex shader pour faire suivre au fragment shader la couleur de chaque sommet,
 - Modifier le fragment shader pour afficher la couleur du fragment (interpolation des couleurs des sommets).

Exercice / préparation TP

- Exercice 2.2 : (1 buffer par attribut)
 - Idem exercice 2.1 mais avec un buffer par attribut

Envoi des informations communes à tous les vertex et fragments : **données Uniform**

- 2 étapes sur le CPU
 - Définir le nom de la variable pour les programmes shader
 - Envoyer les données

// Code openGL

```
vec3 cameraPosition(0.,0.,3.);  
GLuint locCameraPosition ;
```

// 1) définir le nom de la variable pour le shader

```
locCameraPosition = glGetUniformLocation(IdProgram, "cameraPosition");
```

// 2) envoi des données

```
glUseProgram(programID); // ATTENTION indispensable
```

```
glUniform3f(locCameraPosition,cameraPosition.x, cameraPosition.y, cameraPosition.z);
```

//code GLSL

```
uniform vec3 cameraPosition;
```

Envoi des données uniform

- Avec des structures

// Code OpenGL

..

// 1) définir le nom de la variable pour le shader

locLightIntensities = **glGetUniformLocation**(programID, "**light.intensities**");

locLightPosition = **glGetUniformLocation**(programID, "**light.position**");

// 2) envoi des données

glUseProgram(**programID**); // **ATTENTION** indispensable

glUniform3f(**locLightPosition**, LightPosition.x, LightPosition.y, LightPosition.z);

glUniform3f(**locLightIntensities**, LightIntensities.x, LightIntensities.y, LightIntensities.z);

//code GLSL

uniform struct Light {

 vec3 position ;

 vec3 intensities;

 float ambientCoefficient; } light;

```
// VERTEX SHADER
// recup des position des sommets
layout(location = 0) in vec3 position;
// recup de la matrice ModelViewPerspective
uniform mat4 MVP;

void main(){
// Output position of the vertex, in clip space : MVP * position
    gl_Position = MVP * vec4(position,1);
    ...
}
```


Exercice / préparation TP

- Exercice 3.1 : lumière diffuse
 - Lumière :
 - Définir dans le programme OpenGL la position d'une lumière.
 - Envoyez cette position sur GPU pour les shaders
 - Normale en chaque sommet :
 - Dans la structure vertexAttribut ajoutez la normale.
 - Ajoutez un canal d'accès pour la normale les shaders
 - Calcul de la lumière diffuse :
 - Dans le vertex shader, calculer pour

Gestions des transformations

- Dans le programme OpenGL
 - Plus de glTranslate, glRotate...
 - On crée des matrices
 - On les envoie aux shaders (via variable uniforme)
 - glUniformMatrix4fv(...)
 - On les récupère dans les shader et on les applique aux sommets

```
// recup de la matrice ModelViewPerspective
```

```
uniform mat4 MVP;
```

```
...
```

```
    gl_Position = MVP * vec4(vertexPosition_modelspace,1);
```

GLM pour se faciliter la vie

- Définition des matrices de transformation
 - Perspective,
 - Vue
 - Modèle
- Type prédéfinis pour le 3D
 - Vec3, vec4,...

// Include GLM

```
#include <glm/glm/glm.hpp>
```

```
#include <glm/glm/gtc/matrix_transform.hpp>
```

```
using namespace glm;
```

Coté CPU

// TRANSFORMATIONS

```
GLuint MatriceID = glGetUniformLocation(programID, "MVP");
```

// Matrice de Projection : 45° Field of View, ratio 1, intervalle affichage : 0.1 unité <-> 100 unités

```
Projection = glm::perspective(45.0f, 1.0f, 0.1f, 100.0f);
```

// Camera matrix

```
View      = glm::lookAt(glm::vec3(0,0,3), // Place de la Camera ( World Space)
                        glm::vec3(0,0,0), // pointe a l'origine
                        glm::vec3(0,1,0)  // on regarde en haut
                        );
```

// Matrice de modelisation : identité + transfos

```
Model = glm::mat4(1.0f);
```

```
Model = glm::translate(Model, glm::vec3(-0.5f,0.5f,0.0f));
```

```
Model = glm::scale(Model, glm::vec3(0.5f,0.5f,1.0f));
```

// Our ModelViewProjection : multiplication des 3 matrices

```
MVP      = Projection * View * Model; // ordre inverse pour multiplication matrices
```

```
glUniformMatrix4fv(MatriceID, 1, GL_FALSE, &MVP[0][0]);
```

GLM en très court

- Gère les coordonnées homogènes (pour perspective)
 - `glm::mat4` myMatrix;
 - `glm::vec4` myVector;
 - `//... initialisations...`
 - `glm::vec4` transformedVector = myMatrix * myVector;
- Avec une syntaxe identique à GLSL (shader)
 - `mat4` myMatrix;
 - `vec4` myVector;
 - `//... initialisations...`
 - `vec4` transformedVector = myMatrix * myVector;

GLM : les transformations

- // matrice identité
 - glm::mat4 myIdentityMatrix = glm::mat4(1.0f); // paramètre = valeur de la diagonale
- //Translation
 - glm::mat4 myMatrix = glm::translate(uneMatricemat4, glm::vec3(10.0f, 0.0f, 0.0f));
 - = uneMatricemat4 * translation // multiplication à gauche : la translation est appliquée en 1er
- //Changement d'échelle
 - glm::scale(uneMatricemat4, vec3(2.0f, 2.0f ,2.0f));
- // Rotation
 - glm::vec3 myRotationAxis(aX, aY, aZ);
 - glm::rotate(uneMatricemat4, angle_in_degrees, myRotationAxis);

GLM : les transformations

- // composition

- `glm::mat4 myModelMatrix = myTranslationMatrix * myRotationMatrix * myScaleMatrix`

- Débugage

- `#include "glm/gtx/string_cast.hpp"`

- `Cout << glm ::To_string(myMatrice)<< endl ;`

- Mais attention au stockage des matrices voir
<https://stackoverflow.com/questions/59222806/how-does-glm-handle-translation>

GLSL

Open Graphics Library Shading Language
=
OpenGL Shading Language

GLSL : Les variables


- Type de variable
 - float, bool, int
 - vec{2,3,4}, bvec{2,3,4}, ivec{2,3,4} : 2,3,ou 4 float, bool, integer
 - mat2, mat3 , mat4 : matrices 2x2, 3x3 et 4x4
 - sampler1D, sampler2D, sampler3D : pour textures 1D, 2D, 3D
- Possibilité de définir des structures :
- Possibilité de définir des tableaux 1D, 2D
- Variables / fonctions : \approx comme en C

```
struct dirlight {  
    vec3 direction;  
    vec3 color;  
};
```

```
float tab[3][3] =  
{  
    {0.0, 0.0, 0.0},  
    {0.0, 0.0, 2.0},  
    {0.0, 0.0, 0.0}  
};
```

GLSL : Entrées et sorties

- Vertex shader entrées :
 - Données des sommets (attributs des sommets)
 - Location :
`layout (location = 0) in vec3 position ;`
`// La variable position a l'attribut de position 0`
 - Variables communes (à toutes les instances de shaders)
 - Variable globale
 - Syntax de la déclaration pour récupération
 - `Uniform mat4 MVP ;`
 - `// mais doit être définie avant en opengl`
 -



```
glUseProgram(..) ;  
glGetUniformLocation(...)  
glUniformXX(...)
```

GLSL : Entrées et sorties

- Vertex shader sorties :
 - Variable de sortie prédéfinie (pas de déclaration à faire) :
 - `vec4 gl_Position // position du sommets`
 - `out ... (avant le main{}`)
 - `out vec4 FragPosition ;`
 - `out vec4 FragNormale ;`
 - Toutes ces variables seront accessibles en entrée dans le fragment shader

GLSL : Entrées et sorties

- Fragment shader Entrée :

- in ... (avant le main{ })

in vec4 FragPosition ; // avec le meme nom que out du vertex shader

in vec4 FragNormale ; // avec le meme nom que out du vertex shader

- Fragment shader sortie :

- Une seule sortie autorisée (celle de la couleur du fragment en rgba)

out **vec4** FragColor; // on peut choisir le nom que l'on veut

GLSL : fonctions

- `normalize()`
- `dot()` : // produit scalaire
- `cross()` ; // produit vectoriel
- `reflect(vecIncident, vecNormal)` ; // calcule le vecteur réfléchi
- `transpose()`
- `inverse()`
- `pow()`
- `cos()`, `sin()`, ...

GLSL : exemple initialisation

- vecteur
 - `Vec3 monVecteur(1.,0.,1.)`
- équivalent à
 - `MonVecteur.x = 1. ;`
 - `MonVecteur.y = 0. ;`
 - `MonVecteur.z = 0. ;`
- équivalent à
 - `MonVecteur.r = 1. ;`
 - `MonVecteur.g = 0. ;`
 - `MonVecteur.b = 0. ;`
- Matrice
 - `mat3 m = mat3(1.0); // les coefficients de la diagonale valent 1. (matrice identité)`

GLSL : cast utile

- Exemple vecteur
 - `vec4 fragColor ;`
 - `vec3 couleurRouge(1.,0.,0.) ;`
 - `FragColor = vec4(couleurRouge, 1.) ;`
- Exemple matrice
 - `vec4 a, b, c, d;`
 - `a = vec4(1.0, 0.0, 0.0, 0.0);`
 - `b = vec4(0.0, 1.0, 0.0, 0.0);`
 - `c = vec4(0.0, 0.0, 1.0, 0.0);`
 - `d = vec4(0.0, 0.0, 0.0, 1.0);`
 - `mat4 m = mat4(a, b, c, d);` // ATTENTION a = vecteur 1ere ligne de la matrice, b = 2eme ligne ,...

Shaders mini

version 430

Vertex SHADER

```
layout(location = 0) in vec3 position; // recup de la position du sommet
void main(){
    gl_Position = MVP * vec4(position,1.); // Output position
}
```

version 430

Fragment SHADER

```
Out vec4 fragColor ; // déclaration de la variable de sortie
void main(){
    fragColor = vec4(1.,0.,1.,1.); // Output couleur du fragment
}
```


Les textures

- Ce qu'il faut faire
 - Coté opengl
 - Coté shader

Les textures : coté openGL

```
//définir le nom de la variable pour le shader
glUseProgram(programID); // activer le shader en question
locationTexture = glGetUniformLocation(programID, "colorMap");
// charger la texture
GLubyte * image = NULL;
image = glmReadPPM("../texture/StonesCOL.ppm", &iwidth, &iheight);
glUniform1i(locationTexture, 0);
// activer l'unité de texture 0 pour y attacher notre texture
glActiveTexture(GL_TEXTURE0); // ATTENTION à la cohérence avec glUniform1i(locationTexture, 0);
// création du buffer de texture
glGenTextures(1, &bufTexture);
glBindTexture(GL_TEXTURE_2D, bufTexture); // activation
// def des propriétés
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
...
// on envoi la texture au GPU
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, iwidth, iheight, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
```

Coté opengl

- Charger la texture
- Déterminer ces attributs
- L'envoyer au GPU

Les textures

- Ne pas oublier de définir
 - les coordonnées de texture pour chaque sommets
 - Structure vertexAttribut
 - Et de spécifier comment les récupérer dans le shader
 - + glVertexAttribPointer

```
struct vertexAttribut {  
    vec3 position;  
    vec3 normale;  
    vec2 textcoord ;  
};
```

Les textures : coté shaders

// dans vertex shader

```
layout(location = 2) in vec2 VertTexCoord; // on récupère les coord de texture  
out vec2 fragTexCoord;
```

```
main(){  
FragTexCoord = VertTexCoord ; // on fait suivre les coord. De texture au fragment shader
```

// dans fragment shader

```
uniform sampler2D colorMap; // on récupère la texture  
in vec2 fragTexCoord; // on récupère les coord. de texture (interpolée)  
main(){  
    vec4 surfaceColor ;  
    surfaceColor = texture(myTextureSampler, fragTexCoord); // on récupère la couleur
```