
TD OPENGL MODERNE

- LES BASES -

Malek.bengougam@gmail.com

A. Fonctionnalités de base

- I. Création et compilation des shaders
- II. Rendu à base de triangles
- III. Spécifications des attributs

B. Fonctionnalités avancées

- IV. Vertex Buffer Objects (VBO)
- V. Index/Element Buffer Objects (IBO ou EBO)
- VI. Vertex ArrayObjects (VAO)

Note: Ce TD suppose l'utilisation de la classe **GLShader** qui encapsule les fonctionnalités de chargement, compilation et linkage des shaders dans un shader program.

Nous utiliserons également *glew* afin de pouvoir facilement charger les extensions qui nous intéressent (ceci n'est pas utile sous MacOS X car elles sont déjà exposées).

Je vous renvoie aux cours pour le détail des fonctionnalités ainsi qu'à votre moteur de recherche, mais n'hésitez pas à me poser des questions si nécessaire.

La méthodologie choisie est une approche du bas vers le haut, en abordant en premier lieu les commandes effectives de rendu avant d'aborder la configuration du rendu.

Dans le cadre de ce document nous allons nous limiter à trois fonctions que sont l'initialisation, la terminaison et le rendu. Pour un bon fonctionnement de l'application, il faut également définir une fonction de rappel (*callback*) lorsque la fenêtre est redimensionnée.

Voir https://www.glfw.org/docs/3.3/window_guide.html#window_size

Si vous souhaitez également gérer les résolutions high-DPI (Retina etc...) et adapter la résolution du framebuffer: https://www.glfw.org/docs/3.3/window_guide.html#window_fsize

Vous pouvez également ajouter une fonction de mise à jour de la simulation (Update) en plus de la fonction Render().

A. Fonctionnalités de base

I. Création et compilation des shaders

Voici un premier exemple d'utilisation de la classe GLShader :

```
GLShader g_BasicShader;

bool Initialise()
{
    g_BasicShader.LoadVertexShader("basic.vs");
    g_BasicShader.LoadFragmentShader("basic.fs");
    g_BasicShader.Create()

    // cette fonction est spécifique à Windows et permet d'activer (1) ou non (0)
    // la synchronisation vertical. Elle nécessite l'inclure wglew.h
    #ifdef WIN32
    wglSwapIntervalEXT(1);
    #endif
    return true;
}

void Terminate() {
    g_BasicShader.Destroy();
}
```

Le code précédent fonctionne avec toutes les versions des pilotes OpenGL (hormis les versions pré-OpenGL 2.0 mais cela ne nous concerne plus).

Par défaut, dans les cours qui vont suivre nous spécifions toujours la version minimale du shader à la version 120 (OpenGL 2.1) qui est la plus répandue des versions modernes d'OpenGL. Lorsque cela s'avèrera nécessaire nous indiquerons une version spécifique..

Nous avons vu précédemment, la syntaxe des shaders en OpenGL ES 2.0 (#version 100 es) et en OpenGL 2.0 (#version 110) et OpenGL 2.1 (#version 120):

Basic.vs

```
attribute vec2 a_position;
attribute vec3 a_color;

varying vec4 v_color;

void main(void) {
    gl_Position = vec4(a_position, 0.0, 1.0);
    v_color = vec4(a_color, 1.0);
}
```

Basic.fs

```
varying vec4 v_color;

void main(void) {
    gl_FragColor = v_color;
}
```

II. Rendu à base de triangles

Nous allons maintenant nous intéresser à la fonction de rendu principale.
Une passe de rendu peut se résumer aux étapes suivantes :

- a. définition du viewport
- b. effacement des buffers du framebuffer (généralement au moins le back buffer)
- c. spécification du shader à utiliser
- d. définition d'une géométrie
- e. paramétrer le rendu (optionnelle)
- f. rendu d'une géométrie

Nous verrons par la suite comment se passer de l'étape d dans la boucle de rendu.

```
void Render()
{
    // etape a. A vous de recuperer/passer les variables width/height
    glViewport(0, 0, width, height);

    // etape b. Notez que glClearColor est un etat, donc persistant
    glClearColor(0.5f, 0.5f, 0.5f, 1.f);
    glClear(GL_COLOR_BUFFER_BIT);

    // etape c. on specifie le shader program a utiliser
    Auto basicProgram = g_BasicShader.GetProgram();
    glUseProgram(basicProgram);

    // etape d.

    // etape e.

    // etape f. dessin de triangles dont la definition provient d'un tableau
    // le rendu s'effectue ici en prenant 3 sommets a partir du debut du tableau (0)
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // on suppose que la phase d'echange des buffers front et back
    // le « swap buffers » est effectuee juste apres
}
```

III. Spécifications des attributs (étape d)

L'étape 'd' est cruciale car c'est ce qui permet à OpenGL d'indiquer au GPU comment interpréter les données à fournir aux variables de type 'attribute' dans le Vertex Shader.

Il s'agit d'utiliser les fonctions `glVertexAttrib**()` mais comme nos données sont sous formes de tableau il faudra donc le spécifier à OpenGL et qui plus est spécifier comment les données sont structurées en mémoire. Prenons un exemple simple, 3 sommets en 2D:

```
static const float triangle[] = {
    -0.5f, -0.5f,
    0.5f, -0.5f,
    0.0f, 0.5f
};
```

Chaque vertex est composé d'un seul attribut qui est la position du sommet et chaque position est composée de 2 float-s. Cela implique que les attributs d'un sommet sont séparés des attributs du sommet suivant par une distance en octet de **sizeof(float) * 2**.

Ce descriptif est exactement ce dont a besoin OpenGL (et le GPU) afin d'interpréter correctement les données et produire le rendu escompté. Ceci se traduit par le code OpenGL suivant :

```
// premier parametre = 0, correspond ici au canal/emplacement du premier attribut
// glEnableVertexAttribArray() indique que les donnees sont generiques (proviennent
// d'un tableau) et non pas communes a tous les sommets
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float) * 2, triangle);
```

glVertexAttribPointer(index, taille, type, normalisation, écart, adresse)

<https://www.khronos.org/opengl/wiki/GLAPI/glVertexAttribPointer>

index : il s'agit du canal ou registre correspondant a une variable 'attribute' dans le shader

taille : il s'agit en fait du nombre de composantes par attribut, donc 1, 2, 3 ou 4

type : une valeur énumérateur indiquant le type des données

normalisation : les données doivent-elles être normalisées durant le transfert (généralement non)

écart : 'stride' en anglais, l'écart en octet séparant les données N d'un attribut des données N+1

adresse : l'adresse de la première donnée d'un attribut

Le code est encore imparfait en OpenGL (ES) 2. En effet, il n'est pas garanti que le canal / location 0 corresponde bien à l'attribut de `a_position` que nous avons déclaré dans le vertex shader.

Pour s'en assurer il faut utiliser une commande OpenGL pour faire une requête sur le shader actuellement actif –après `glUseProgram()` donc.

```
int location = glGetAttribLocation(GLuint program, const char* name)
```

Cette fonction retourne le canal/location de l'**attribut** 'name' d'un shader program qui s'est lié correctement (sans erreurs). Le code précédent devient alors :

```
int loc_position = glGetAttribLocation(program, "a_position");
glEnableVertexAttribArray(loc_position);
glVertexAttribPointer(loc_position, 2, GL_FLOAT, false, sizeof(float) * 2, triangle);
```

Exercice A.1 –

Ajoutez un attribut couleur au tableau triangle. Les couleurs des sommets doivent être différentes de manière à ce que votre objet (triangle ici) s'affiche sous la forme d'un dégradé. Il est donc impératif de communiquer l'information de couleur à l'attribut « `a_color` ».

Prêtez bien attention à l'écart (stride) ainsi qu'à l'adresse de départ de chaque attribut.

Exercice A.2 –

Remplacez votre tableau de float par un tableau de Vertex, où Vertex est une structure composée des attributs position (2D) et color (3D). Il est très fortement recommandé de créer une structure/classe pour chacun des types fondamentaux que l'on utilisera pour les points, vecteurs et couleurs en 2D et 3D (voire 4D).

B. Fonctionnalités avancées

Le principal problème de l'exemple précédent tient au fait que l'on est obligé de retransmettre les informations topologiques au GPU à chaque trame. Ceci n'est pas bien grave pour un simple triangle mais pour des modèles 3D à plusieurs milliers de triangle cela commence à faire beaucoup en termes de bande passante.

De plus au fur et à mesure que l'on va complexifier nos attributs (ajouts de coordonnées de texture, normales, information de skinning etc...), et potentiellement des objets avec une variété d'attributs, le driver va effectuer un travail redondant de reconfiguration des attributs en entrée du Vertex Shader.

OpenGL permet le stockage de données –idéalement en mémoire vidéo- via le mécanisme des Buffer Objects.

Un BO se crée à l'aide de la fonction **glGenBuffers()** qui permet de réserver un identifiant de type entier (handle ou name) que l'on peut détruire ensuite avec la fonction **glDeleteBuffers()** – fonction qui libère également la mémoire allouée.

Cette mémoire est allouée par un appel à la fonction **glBufferData()** comme on le verra en détail.

Une particularité de l'API OpenGL est de ne pouvoir travailler que sur un seul buffer d'un même genre (on parle de « target ») à la fois. Le mécanisme effectuant le lien entre un identifiant (« name ») créé par **glGenBuffers()** et l'objet est appelé « binding ». Il faut donc spécifier quel est le Buffer Object qui doit être référencé à l'aide de la fonction **glBindBuffer()**.

La finalité est d'essayer d'avoir le moins de code de gestion dans la boucle de rendu et d'essayer de tout faire à l'initialisation.

IV. Vertex Buffer Objects (VBO)

Le premier objectif est d'éviter la redéfinition à chaque trame des données topologiques et, lorsque l'on sait que les données ne vont pas être modifiées, qu'elles restent résidentes en mémoire vidéo. Un paramètre de l'API OpenGL permet de spécifier à OpenGL comment seront utilisées ces données.

Le principal changement au niveau du code va se traduire par un couper-coller de la partie définissant le triangle dans la boucle de rendu vers la fonction d'initialisation.

Le premier Buffer Object que nous allons utiliser à comme « binding » ou « target » **GL_ARRAY_BUFFER**. Le rôle de celui-ci consiste à stocker les données des attributs de vertex.

D'où l'appellation commune de Vertex Buffer Object (VBO).

Exemple dans l'initialisation :

```
GLuint VBO;
// [...]
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// glBufferData alloue et transfère 4 * 2 * 3 octets issus du tableau triangle
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 2 * 3, triangle, GL_STATIC_DRAW);
// je recommande de réinitialiser les états à la fin pour éviter les effets de bord
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

On veillera à ne pas oublier de libérer la mémoire utilisée par le BO en quittant le programme **glDeleteBuffers(1, &VBO);**

Le code de la fonction de rendu principale devient alors :

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float)*2, 0);
glEnableVertexAttribArray(0);

glDrawArrays(GL_TRIANGLES, 0, 3);
```

Notez que le dernier paramètre de **glVertexAttribPointer()** est devenu **0** (ou **NULL**, **nullptr**). En interne le code de **glVertexAttribPointer** teste la valeur de **GL_ARRAY_BUFFER**.

Au moment de l'appel à **glVertexAttribPointer()**, tandis que la valeur de « binding » de **GL_ARRAY_BUFFER** est différente de 0 (autrement dit lorsqu'un Buffer Object est lié, « bind ») la fonction interprète le dernier paramètre non plus comme une adresse absolue en mémoire système (RAM) mais comme un offset (ou adresse relative).

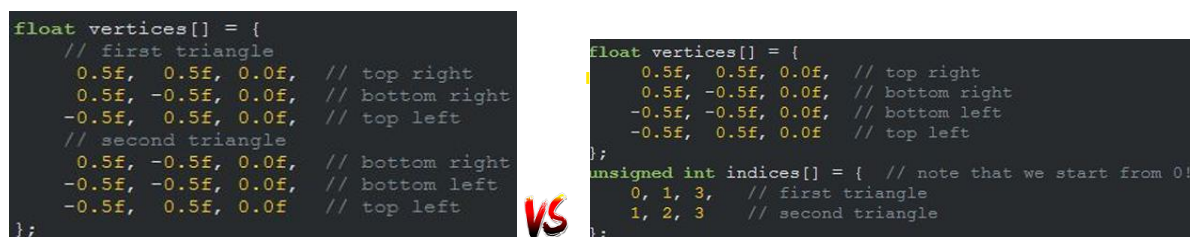
En effet, le Buffer Object peut se trouver en VRAM, donc dans une mémoire externe qui n'est pas adressable directement par le CPU mais seulement accessible par le contrôleur de bus.

Exercice B.1 –

Modifiez votre code afin d'intégrer l'utilisation des VBOs.
Notez que le dernier paramètre de **glVertexAttribPointer()** indique toujours le début du premier attribut mais cette fois-ci de manière relative.
Vous pouvez vous aider de la pseudo-fonction **offsetof()** du C++ pour vous aider à calculer l'offset exact en octets.

V. Index Buffer Objects (IBO)

L'intérêt d'un IBO est double : réduire l'empreinte mémoire des sommets, et optimiser le temps de traitement des Vertex par le GPU. En effet, ce dernier est maintenant capable d'identifier, par leur index, les sommets déjà traités et donc éviter de les transformer une n-ième fois (si déjà dans son cache).



```
float vertices[] = {
    // first triangle
    0.5f, 0.5f, 0.0f, // top right
    0.5f, -0.5f, 0.0f, // bottom right
    -0.5f, 0.5f, 0.0f, // top left
    // second triangle
    0.5f, -0.5f, 0.0f, // bottom right
    -0.5f, -0.5f, 0.0f, // bottom left
    -0.5f, 0.5f, 0.0f // top left
};

float vertices[] = {
    0.5f, 0.5f, 0.0f, // top right
    0.5f, -0.5f, 0.0f, // bottom right
    -0.5f, -0.5f, 0.0f, // bottom left
    -0.5f, 0.5f, 0.0f // top left
};
unsigned int indices[] = { // note that we start from 0!
    0, 1, 3, // first triangle
    1, 2, 3 // second triangle
};
```

Exercice B.2 –

Modifiez le code précédent afin de dessiner le triangle en utilisant un tableau d'indices. Commencez d'abord par utiliser la fonction **glDrawElements()** à la place de **glDrawArrays()** en utilisant un tableau d'indices (de type *unsigned short* ou *unsigned int*).

Puis utilisez un Buffer Object tout comme pour un **ARRAY_BUFFER** à cela près que le binding est **GL_ELEMENT_ARRAY_BUFFER** et que le dernier paramètre de **glDrawElements()** est **NULL**.

VI. Vertex Array Objects (VAO)

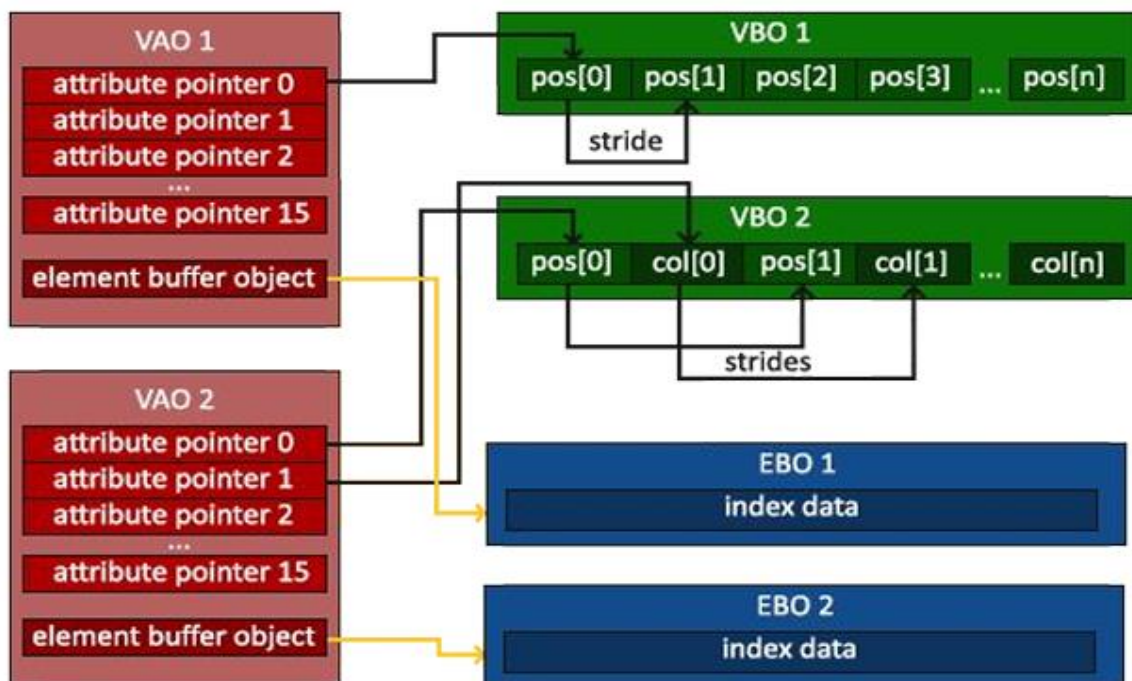
L'autre problématique réside dans le fait que l'on est obligé de re-spécifier à chaque trame le format des attributs en entrée du Vertex Shader alors que la correspondance (mapping) est constante pour le rendu d'un objet avec un shader.

Un VAO est un objet OpenGL qui permet de stocker les états des attributs (le nom Vertex Array peut prêter à confusion, mais il s'agit bien ici des paramètres et non du contenu des attributs).

Un VAO va surtout enregistrer le « mapping » entre un VBO et la spécification des attributs via `glVertexAttribPointer` (taille et type des données, tableau ou valeur simple etc...).

L'identifiant d'un VBO est également enregistré s'il est déclaré pendant l'initialisation du VAO.

Note : En OpenGL 2 et 3 l'usage d'un VAO introduit un couplage fort avec un VBO. On verra plus tard comment il est possible d'avoir un mécanisme plus souple sans ce couplage.



Exemple de 2 VAOs: VAO1 référence EBO1 et VBO1 qui ne contient que des positions, alors que deux attributs du VAO2 référencent les positions et couleurs stockées dans VBO2.

Création :

```
glGenVertexArrays(1, &VAO);
```

Destruction :

```
glDeleteVertexArrays(1, &VAO);
```

Utilisation :

```
glBindVertexArray(VAO);
```

Attention ! Une erreur récurrente consiste à oublier que le VAO enregistre tout ce qui est en rapport avec un `glDraw***()` ce qui concerne les fonctions suivantes :

```
glBindBuffer(GL_ARRAY_BUFFER ou GL_ELEMENT_ARRAY_BUFFER)
```

```
glVertexAttribPointer()
```

```
glEnable/DisableVertexAttribArray()
```

Le but d'un VAO est d'éviter d'avoir à appeler ces fonctions dans la boucle de rendu principale. Tout appel à l'une de ces fonctions pendant qu'un VAO autre que celui par défaut (zéro) est actif (bind) entraîne l'écrasement de la valeur précédente.

Par exemple, voici une erreur classique en fin d'initialisation d'un VAO :

```
glBindVertexArray(VAO);  
[...]  
glBindBuffer(GL_ARRAY_BUFFER, VBO);  
[...]  
glBindBuffer(GL_ARRAY_BUFFER, 0); // oups ecrase la reference de VBO dans le VAO  
glBindVertexArray(0);
```

C'est en effet une bonne pratique que de réinitialiser les valeurs par défaut afin d'éviter les effets de bords mais autant faut-il le faire dans le bon ordre.

La bonne pratique consiste à toujours forcer le VAO par défaut (zéro) avant de procéder à toute opération sur les VBO, IBO etc...

```
glBindVertexArray(0);  
// ok, maintenant on peut modifier le VBO et IBO courant sans ecraser le VAO  
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Exercice B.3 –

Modifiez votre code en intégrant l'utilisation des VAO.

Le but d'un VAO étant d'éviter le code de paramétrage des attributs dans la boucle de rendu, le code principal de la boucle de rendu doit se réduire à

```
glBindVertexArray(VAO);  
glDrawArrays(GL_TRIANGLES, 0, N);
```

Ou, si vous utilisez un IBO

```
glBindVertexArray(VAO);  
glDrawElements(GL_TRIANGLES, N, GL_UNSIGNED_SHORT /*ou _INT*/, 0 /*nullptr*/);
```