Université Mohamed Khider Biskra
Faculté des Sciences Exactes, des Sciences de la Nature et de la Vie
Département d'Informatique , laboratoire LESIA, Biskra, Algérie

OpenGl 3.x

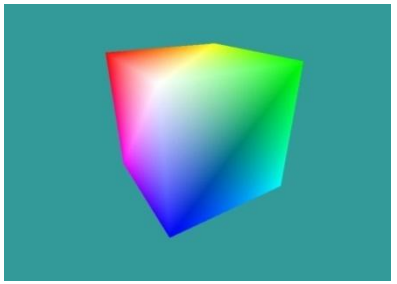
Présenté par Zerari Abd-El-Mouméne
2019/2020

Vertex Array

Problèmes avec Opengl traditionnel

- Rendus 3D avec OpenGL sont lents!
- Codes OpenGL trop gourmands!
- Nouvelle façon d'afficher des polygones avec OpenGL ?

La géométrie immédiate?



Exemple d'un cube: **(vertex, couleur),**de **6** faces. Chacune des faces est constituée de 2 triangles dont chaque vertex à une couleur différente, donc un cube de 12 triangles.

Ainsi, nous devons appeler **36** fois la fonction *glVertex*()* (12 triangles * 3 vertex).

Nb total d'appels de fonction = 36(vertex)+ 36(couleur) + 2 (appels pour *glBegin* et *glEnd)= 74 appels.*

La géométrie immédiate

Avantages de la géométrie immédiate:

- Définir indépendamment chacune des informations des vertex
- Augmenter la lisibilité du code
- Regroupe clairement chaque rendu par type de primitives

La géométrie immédiate

Inconvénients de la géométrie immédiate :

- Un grand nombre d'appels de fonction pour des géométries simples
- Grande répétitivité de commandes identiques
- Complexifie les structures de données utilisables.
- Le dessin "*Immédiat*" est très contraignant du point de vue performance pour une application "temps-réel".
- Problèmes transfert des données (envoyées au GPU) est extrêmement lent.
- Le nombre d'appels identiques à chaque frame.
- Déconseillé depuis OpenGL 3.0

La géométrie immédiate

Solution:

OpenGL propose une nouvelle technique "***vertex array***".

Vertex Array

Le principe:

Rassembler l'ensemble des vertex d'une géométrie dans un tableau unique. Il peut y avoir un tableau par type d'information ou un seul tableau contenant toutes les informations. L'utilisation des Vertex Array se fait en trois étapes principales:

1. La première consiste en l'activation / désactivation des types de tableaux à utiliser.
2. Ensuite, il faut remplir les tableaux activés avec les informations de notre géométrie.
3. Enfin, on peut lancer l'appel du rendu de notre géométrie.

Vertex Array

Avantages :

- Réduction du nombre d'appels de fonction
- Réduction de la quantité d'information envoyée à la carte graphique

Inconvénients:

- Applicable uniquement pour une géométrie statique
- Les données sont stockés dans le côté du client(CPU et RAM)
- Aller-retour entre le CPU et le GPU.

Vertex Buffer Objects

Solution : Les Vertex Buffer Objects FBO

Héberger les données directement sur le serveur (GPU).

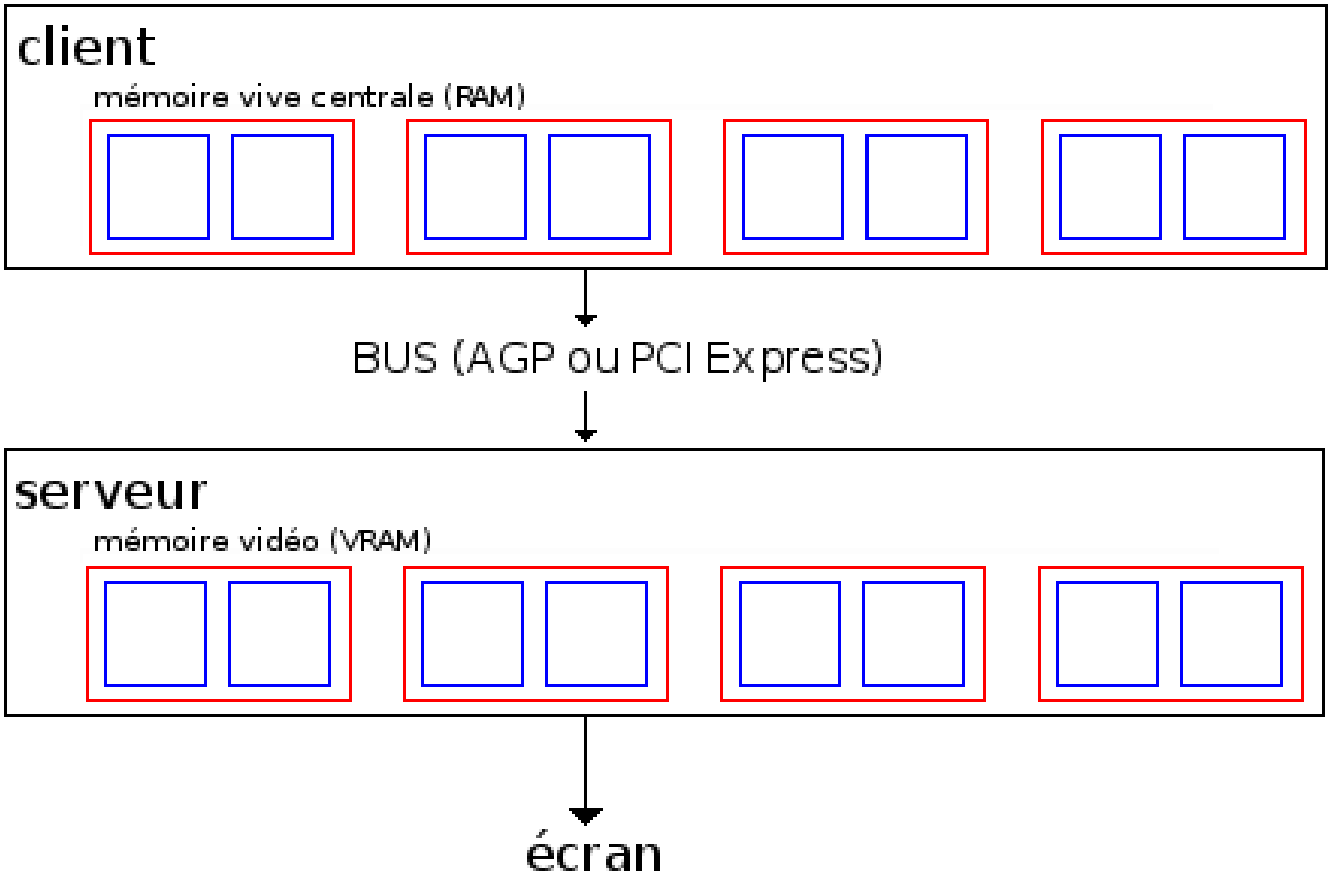
Depuis les spécifications de la version 1.5 d'OpenGL est intégrée cette technologie qui permet de stocker directement en mémoire graphique toute notre géométrie à dessiner.

Les VBO permettent d'envoyer et de stocker toutes les informations de notre géométrie dans notre carte graphique.

Vertex Buffer Objects

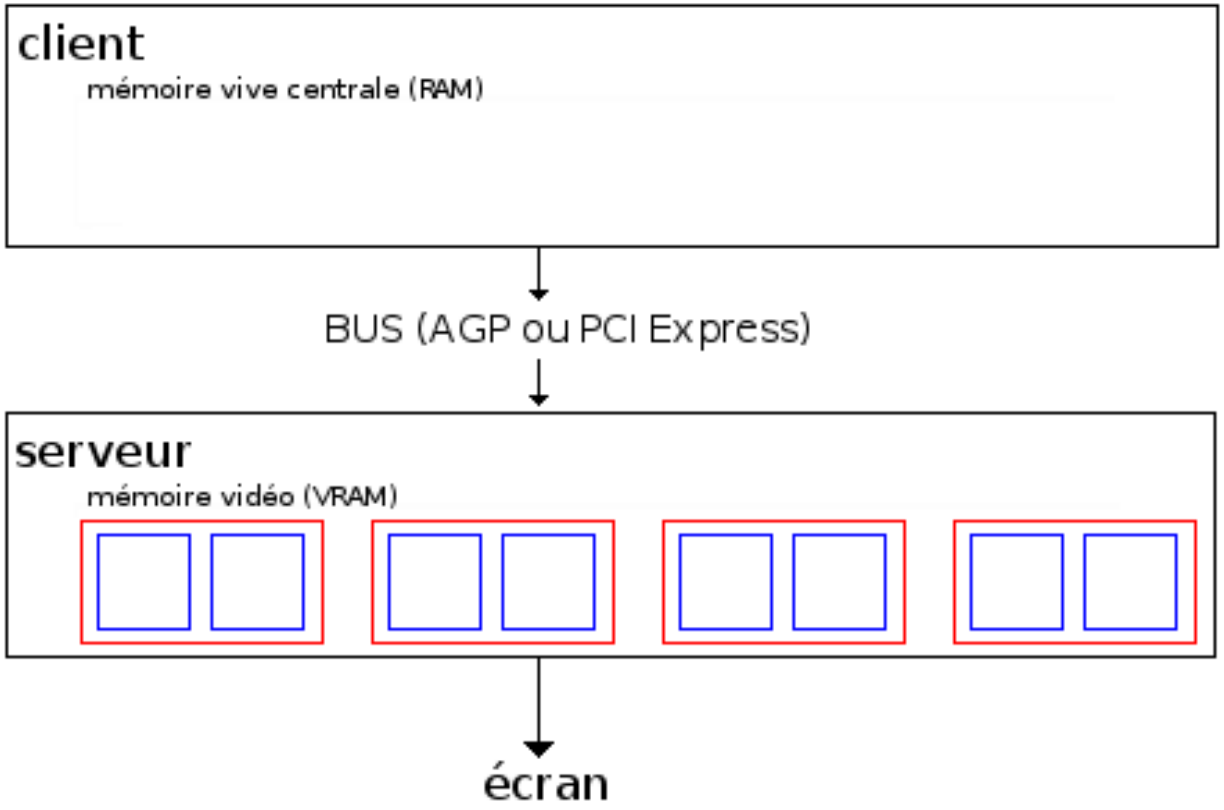
Nous avons vu que nos données (**VA**) étaient stockées côté client. Ici (**VBO**) ce sera l'inverse les données stockées côté serveur , c'est ce qui fait la puissance des VBO.

Voici ce qui se passe dans le cas des VA :



Vertex Buffer Objects

Voici ce qui se passe dans le cas des VBO :



Les données ne sont ni dupliquées, ni transférées à chaque rendu

Vertex Buffer Objects

1) Création d'un VBO
2) Bind du VBO (pour dire qu'il devient le VBO courant)
3) Allocation de mémoire dans la carte graphique
4) Copie des données des sommets dans la carte graphique

Vertex Buffer Objects

1. Création

Pour utiliser ces buffers, il faut demander à OpenGL un ou plusieurs identifiants pour ces buffers.

Création d'un objet tampon et récupération de son identifiant:

GLuint glGenBuffers(GLsizei number, GLuint *buffers);

- **number** : Le nombre d'ID à initialiser.
- **buffers** : Un tableau de type ***GLuint***. On peut également mettre l'adresse d'une variable ***GLuint*** pour n'initialiser qu'un seul ID.

```
// exemple de : Génération de l'ID
   glGenBuffers(1, &m_vboID);
```

Vertex Buffer Objects

1. Création

Le verrouillage

- A chaque fois que l'on veut configurer ou utiliser un *objet OpenGL* il faut le verrouiller car OpenGL a justement besoin de savoir sur quelle chose il doit travailler.
- La fonction permettant de faire cette opération s'appelle **glBindXXX**(), avec les **VBO** ce sera **glBindBuffer**().

void glBindBuffer(GLenum target, GLuint buffer);

target : type de l'objet que l'on veut verrouiller, dans notre cas on lui donne en paramètre : **GL_ARRAY_BUFFER**
buffer : ID qui représente le VBO. On lui donnera la valeur de l'ID et pas son adresse.

Vertex Buffer Objects

1. Création

Exemple:

```
{
   // Génération de l'ID
   glGenBuffers(1, &m_vboID);
   // Verrouillage du VBO
   glBindBuffer(GL_ARRAY_BUFFER, m_vboID);
      // Configuration
      ....
   // Déverrouillage de l'objet
   glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Vertex Buffer Objects

1. **Création**

GL_ARRAY_BUFFER correspond aux tableaux des informations sur les vertex et

GL_ELEMENT_ARRAY_BUFFER correspond aux tableaux des indices de vertex.

Vertex Buffer Objects

2. Allocation de la mémoire vidéo

✓ L'étape suivante est: Transfert des données du **CPU -> Le GPU**

✓ Remplir la mémoire de la carte graphique.

✓ Les transferts seront un peu déroutants car nous ne transférons pas directement des **float** ou des **unsigned int** mais des **bytes**.

Vertex Buffer Objects

2. Allocation de la mémoire vidéo

Allocation (RAM), nous devons utiliser le mot-clef **new[].**
Dans notre cas, nous utilisant la fonction: **glBufferData**() :
glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage)

target : on lui donnera la valeur **GL_ARRAY_BUFFER**
size : C'est la taille mémoire à allouer (en bytes), il faut prendre en compte **TOUTES** les données à envoyer .
data : Ce sont les données à transférer.
usage : Permet de définir la fréquence de mise à jour des données

Vertex Buffer Objects

2. Allocation de la mémoire vidéo

Différents cas d'utilisations GLenum usage :

Il permet à OpenGL de savoir si les données que nous stockerons dans le **VBO** seront mises à jour rarement, fréquemment ou tout le temps..

Vertex Buffer Objects

2. Allocation de la mémoire vidéo.

Différents cas d'utilisations GLenum usage :

Pour chacun des paramètres, il existe trois catégories différentes. Dans le cas de la fréquence d'utilisation, elle peut être :

- ***STATIC*** : les données seront modifiées une fois pour de nombreuses utilisations
- ***STREAM*** : les données seront modifiées une fois pour quelques utilisations
- ***DYNAMIC*** : les données seront modifiées et utilisées régulièrement de nombreuses fois

Vertex Buffer Objects

2. Allocation de la mémoire vidéo.

Une fois que nous avons déterminé cette fréquence, il faut choisir un type d'utilisation de notre buffer. Il existe trois types différents à choisir parmi :

- ***DRAW*** : les données sont modifiées par l'application et utilisées par OpenGL pour le rendu d'images *(le plus utilisé)*
- ***READ*** : les données sont modifiées par une lecture de données depuis OpenGL et utilisées en lecture par l'application.
- ***COPY*** : les données sont modifiées par une lecture de données depuis OpenGL et utilisées par OpenGL pour le rendu d'images.

Vertex Buffer Objects

2. Allocation de la mémoire vidéo.

A partir de ces deux paramètres, nous pouvons déterminer quel type de buffer nous allons utiliser et quel paramètre passer à la fonction *glBufferData*. Nous avons donc neuf possibilités, donc neuf valeurs différentes :

	STATIC	STREAM	DYNAMIC
DRAW	GL_STATIC_DRAW	GL_STREAM_DRAW	GL_DYNAMIC_DRAW
READ	GL_STATIC_READ	GL_STREAM_READ	GL_DYNAMIC_READ
COPY	GL_STATIC_COPY	GL_STREAM_COPY	GL_DYNAMIC_COPY

Vertex Buffer Objects

Exemple:

```
{
   // Génération de l'ID
   glGenBuffers(1, &m_vboID);
   // Verrouillage du VBO
   glBindBuffer(GL_ARRAY_BUFFER, m_vboID);
      // Allocation de la mémoire
      glBufferData(GL_ARRAY_BUFFER, m_tailleVerticesBytes + m_tail
leCouleursBytes, 0, GL_STATIC_DRAW);
   // Déverrouillage de l'objet
   glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Vertex Buffer Objects

3. Transfert des données.

- La fonction **glBufferData**() nous a permis d'allouer un espace mémoire pour y stocker nos données.

- Pour transférer les données, nous allons utiliser une autre fonction OpenGL dont le nom ressemble étonnamment à celui de la fonction précédente : **glBufferSubData**().

Vertex Buffer Objects

3. Transfert des données.

glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid *data);

- **target** : on lui donnera la valeur **GL_BUFFER_DATA**
- **offset** : Case en mémoire où on va commencer le transfert dans la **VRAM**
- **size** : La taille des données à copier (en bytes)
- **data** : Les données à copier, par exemple le tableau de vertices

Vertex Buffer Objects

3. Transfert des données.

Exemple: afficher un triangle colorer

```
float pos[3*3] =
   {
      -1.0, -1.0, 0,
      1.0, -1.0, 0,
      0.0, 1.0, 0
   };

   float colors[3*3] =
   {
      1.0, 0.0, 0.0,
      0.0, 1.0, 0.0,
      0.0, 0.0, 1.0
   };
```

Vertex Buffer Objects

3. Transfert des données.

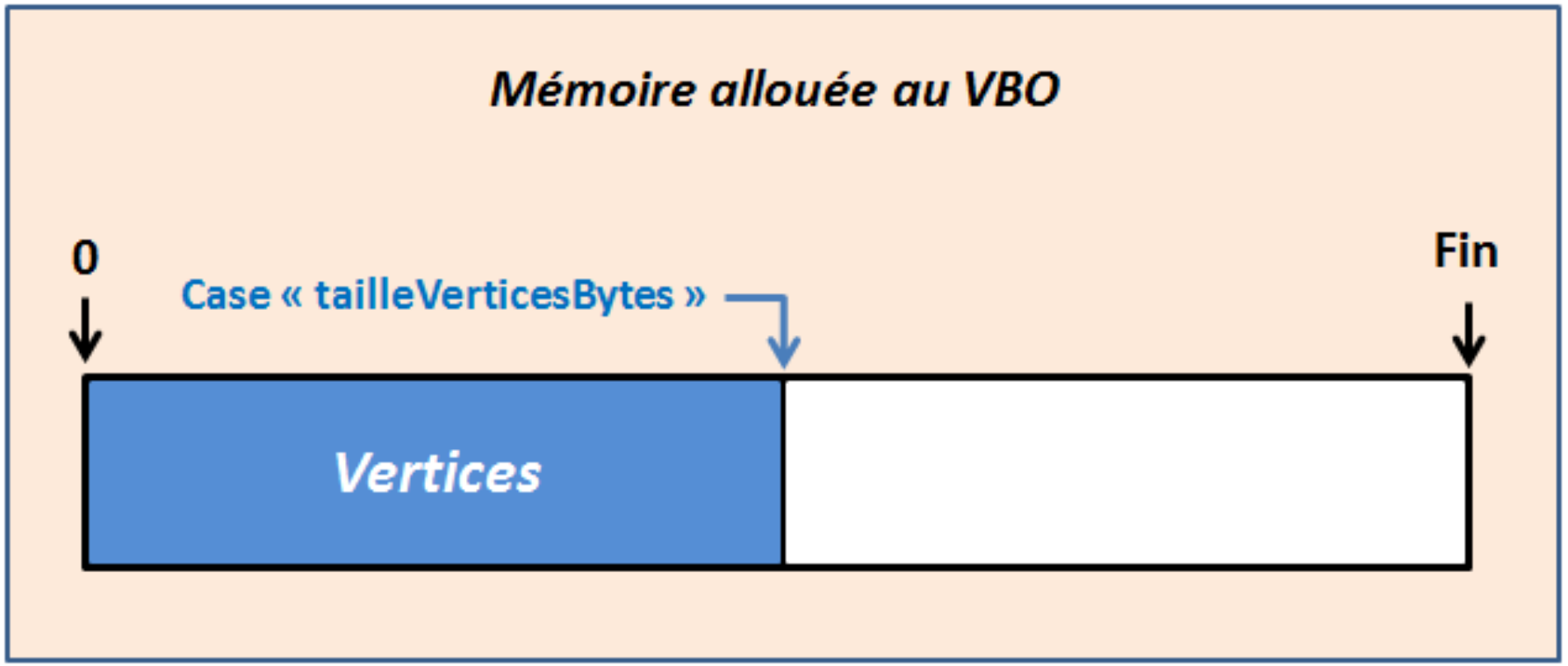
Exemple: afficher un triangle colorer

```
/* on alloue de l'espace */
  glBufferData(GL_ARRAY_BUFFER,              /* target */
               (3*3*sizeof *pos) +    /* taille des positions */
               (3*3*sizeof *colors),  /* taille des couleurs */
               NULL,                  /* ... */
               GL_STREAM_DRAW);               /* mode */
              /* on specifie les donnees */
  glBufferSubData(GL_ARRAY_BUFFER,
                  0,                  /* emplacement des donnees dans le VBO */
                  (3*3*sizeof *pos), /* taille des donnees */
                  pos);                /* adresse des donnees */
  glBufferSubData(GL_ARRAY_BUFFER,
                  (3*3*sizeof *pos),   /* emplacement */
                  (3*3*sizeof *colors) , /* taille */
                      colors);                    /* donnees */
```

Vertex Buffer Objects

3. Transfert des données.

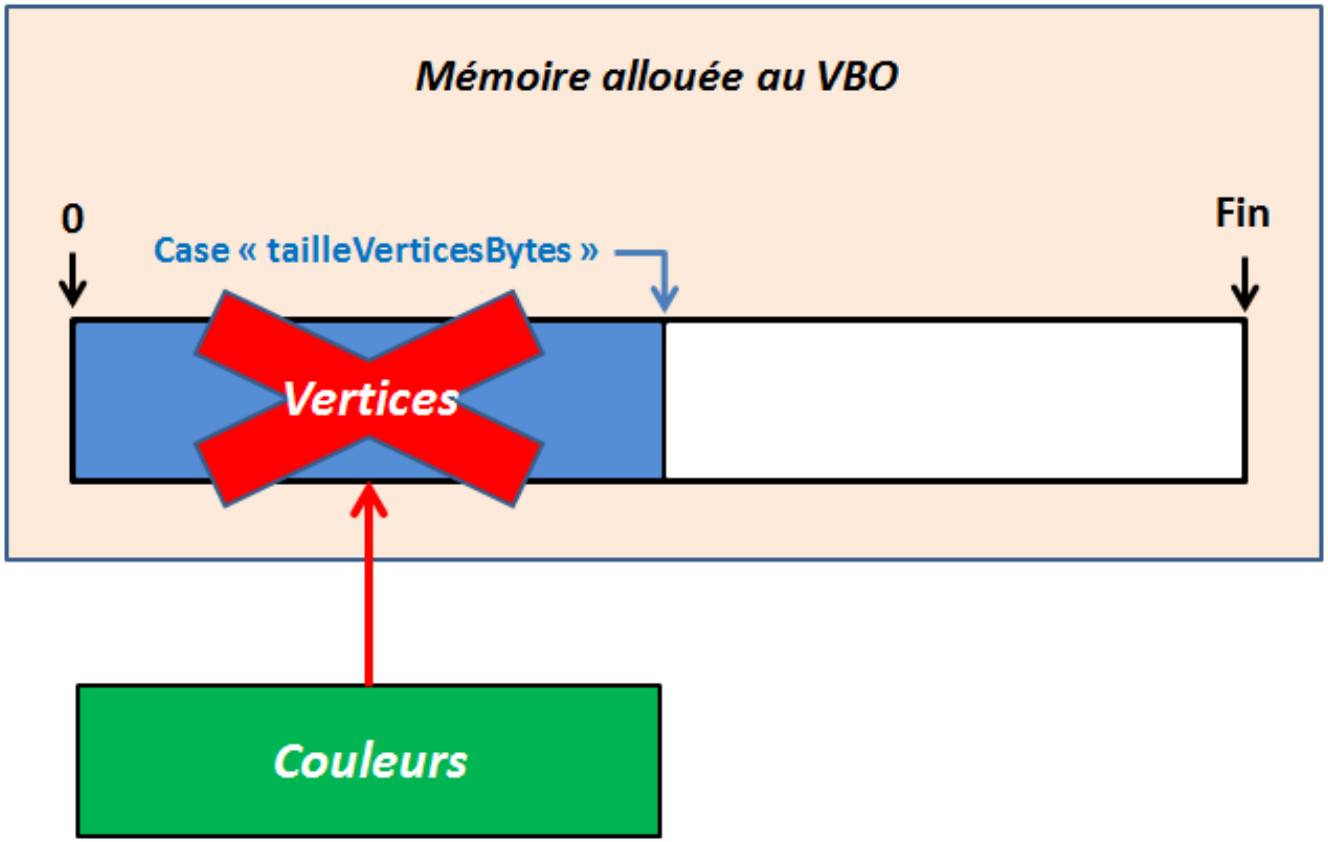
Le paramètre **offset** correspond quant à la case mémoire dans laquelle va commencer la copie. Pour le transfert des vertices, ce paramètre sera de **0** vu que l'on commence à copier au début de la zone mémoire :



Vertex Buffer Objects

3. Transfert des données.

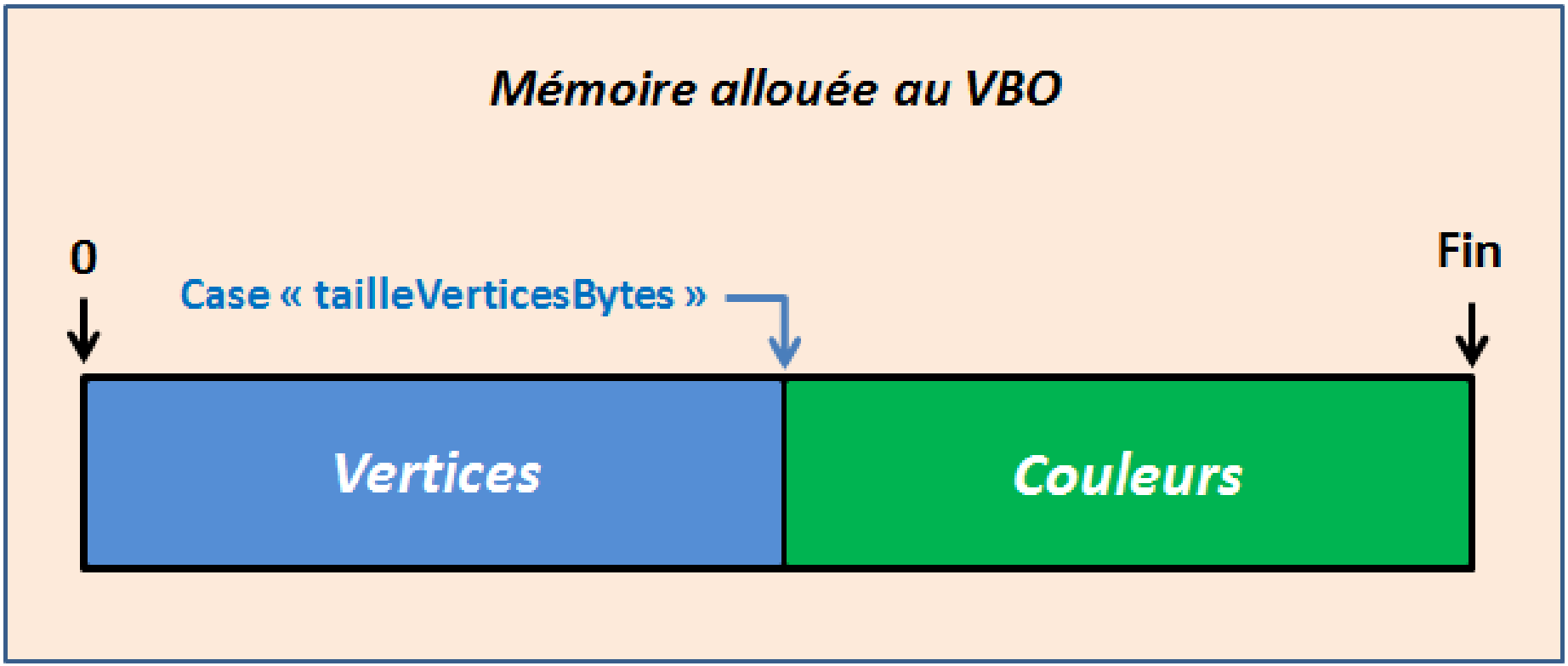
En revanche pour le transfert des couleurs, on ne va pas commencer la copie à la case **0** sinon on va écraser les valeurs transférées juste avant :



Vertex Buffer Objects

3. Transfert des données.

Il faudra commencer la copie à la fin du tableau de vertices, soit à la case **tailleVerticesBytes** :



Vertex Buffer Objects

4. **Utilisation d'un VBO.**

- Après avoir créé un **VBO** et initialisé, on doit l'utiliser dans nos programmes.

- Pour afficher un modèle 3D nous devions envoyer les données à l'aide de la fonction **glVertexAttribPointer**() puis nous affichions le tout avec la fonction **glDrawArrays**().

Vertex Buffer Objects

4. Utilisation d'un VBO

- Les données se trouvent dans la mémoire vidéo, OpenGL ne sait pas où elles se situent exactement, c'est à nous de lui dire. Pour ça, on va toujours utiliser la fonction **glVertexAttribPointer**() sauf qu'on va modifier son dernier paramètre.
- Les fonctions pour définir les attributs dans un VBO suivent le modèle ce celles pour les positions, vecteur normal, couleur, etc.

Variables attribute

4. Utilisation d'un VBO

Pour définir un tampon d'attributs dans le programme client : extern void **glVertexAttribPointer**(GLuint **index**, Glint **size**, GLenum **type**, GLboolean **normalized**, GLsizei **stride**, const GLvoid ***data**);

- **index** : Identificateur de l'attribut.
- **size** : Nombre de composante par élément (1,2,3 ou 4).
- **type** : Type des composante (GL_FLOAT, etc.).
- **normalized** : Détermine si valeurs sont normalisées ou non.
- **stride** : Espace entre les éléments du tampon (nombre d'octets).
- **data** : Le tampon lui même.

Variables attribute

4. Utilisation d'un VBO

Pour retrouver une référence (index) à un attribut défini dans le VS :

GLint glGetAttribLocation(GLuint p, const GLchar* name);

- **p** : Le programme.
- **name** : Nom de l'attribut dans le VS.

Il faut activer le tampon d'attributs avec la fonction :

void **glEnableVertexAttribArray(GLuint index);**

- **index** : Référence au tampon.

Lorsqu'on a terminé, il faut désactiver :

- **void glDisableVertexAttribArray(GLuint index);**
- **index** : Référence au tampon.

Vertex Buffer Objects

4. Utilisation d'un VBO

void glDrawArrays(GLenum primType, GLint first, GLsizei nombre);

primType : (GL_LINES , GL_TRIANGLES , etc.). Types de primitives

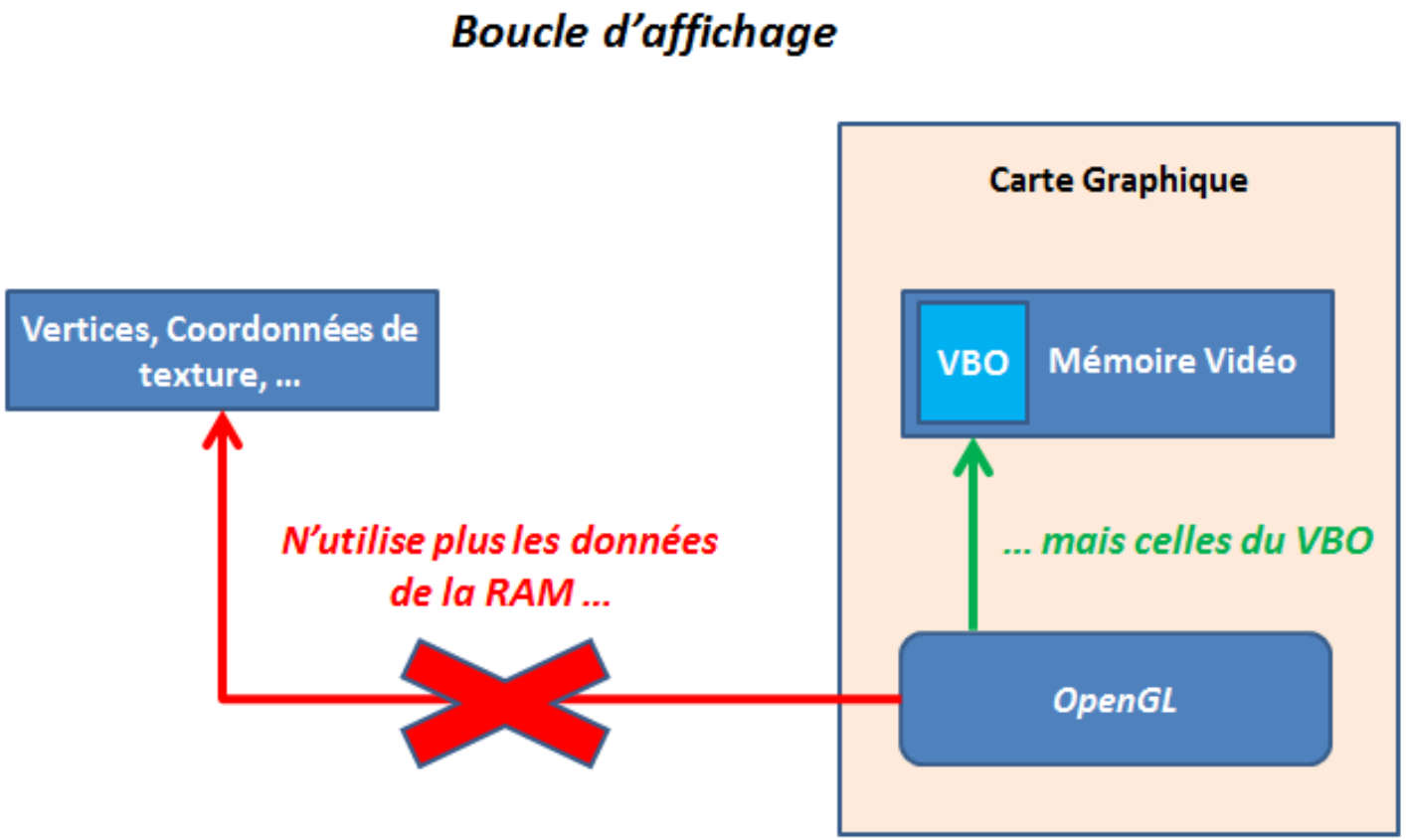
first : c'est le numéro du premier sommet que l'on voudra afficher.

nombre : c'est le nombre de sommets que l'on voudra afficher en partant de first . Le nombre total de sommets qui seront affichés sera égal à nombre .

Vertex Buffer Objects

4. Utilisation d'un VBO

- Le paramètre **pointer** permet à OpenGL de savoir où se trouvent nos données. Nous devons lui donner l'adresse des tableaux à l'intérieur de la mémoire vidéo.



Vertex Buffer Objects

4. Utilisation d'un VBO

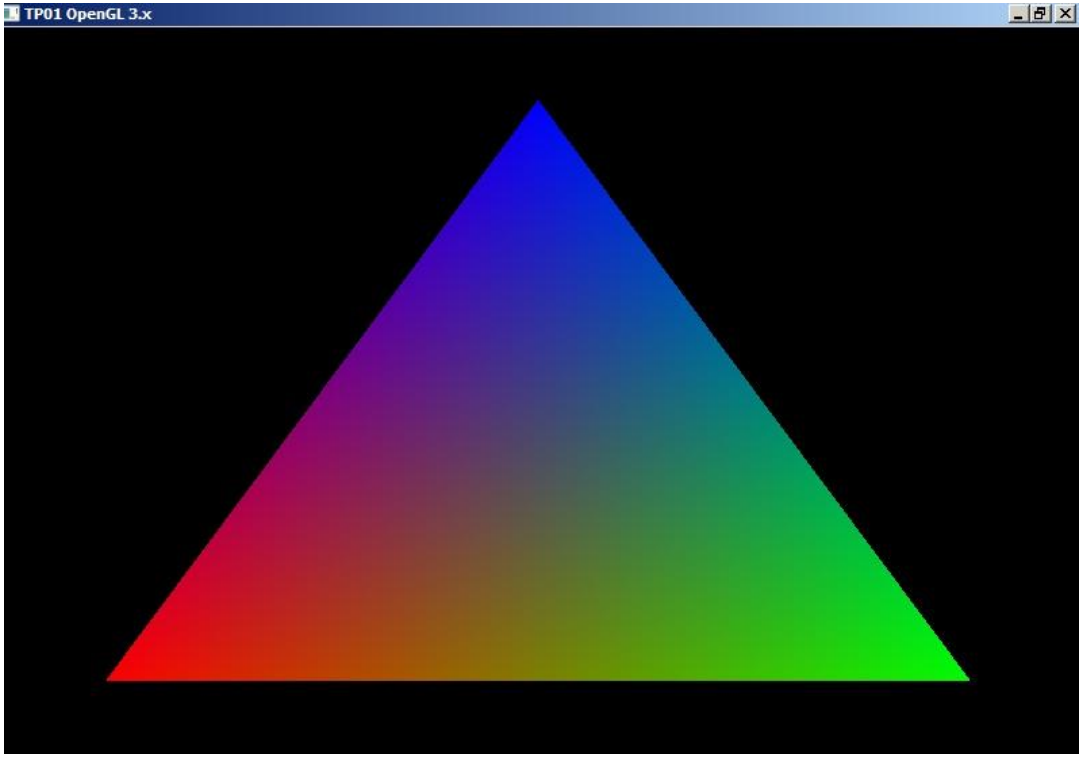
```
// boucle d'affichage principale
        glBindBuffer(GL_ARRAY_BUFFER, m_vboID);
// Accès aux vertices dans la mémoire vidéo
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));
        glEnableVertexAttribArray(0);
// Accès aux couleurs dans la mémoire vidéo
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
        BUFFER_OFFSET(N_VERTS*P_SIZE*sizeof *pos));
        glEnableVertexAttribArray(1);
        glDrawArrays(GL_TRIANGLES, 0, 3);
        glDisableVertexAttribArray (0);
        glDisableVertexAttribArray (1);
// Déverrouillage du VBO
        glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Vertex Buffer Objects

Résultat :

Exemple: afficher un triangle colorer

Code Sources **OpenGl 3 TP0**



Fin