

# Les Tableaux de Sommets

# En quête de performance...

```
glBegin(GL_POLYGON);
```

```
    glVertex3fv(v0, 0);
```

```
    glVertex3fv(v1, 0);
```

```
    glVertex3fv(v2, 0);
```

```
    . . . . .
```

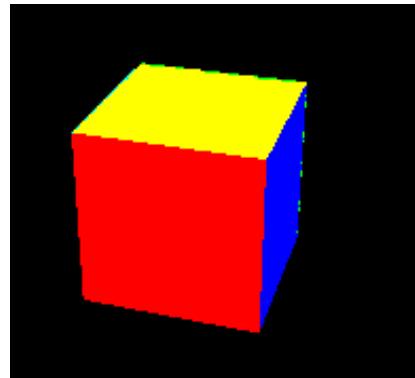
```
    glEnd();
```

# En quête de performance...

- Il s'agit d'autre part d'éviter la description redondante de sommets partagés par des polygones adjacents.
- Par exemple un cube est formé de 6 faces et 8 sommets. Si les faces sont décrites indépendamment, chaque sommet est traité 3 fois, une fois pour chaque face à laquelle il appartient.
- Les tableaux de sommets sont standards depuis la version 1.1 d'OpenGL

# Exemple:

```
static GLubyte devant[] = {0,1,2,3};  
static GLubyte deriere[] = {4,5,6,7};  
static GLubyte droite[] = {1,5,6,2};  
static GLubyte gauche[] = {0,3,7,4};  
static GLubyte bas[] = {0,4,5,1};  
static GLubyte haut[] = {3,2,6,7};  
  
GLfloat points[] = {  
    0.0f, 0.0f, 0.0f,  
    1.0f, 0.0f, 0.0f,  
    1.0f, 1.0f, 0.0f,  
    0.0f, 1.0f, 0.0f,  
    0.0f, 0.0f, -1.0f,  
    1.0f, 0.0f, -1.0f,  
    1.0f, 1.0f, -1.0f,  
    0.0f, 1.0f, -1.0f,  
}; ;
```



```
glVertexPointer(3, GL_FLOAT, 0, points);  
glTranslatef(-0.5, -0.5, 0.5);  
glColor3f(1.1f, 0.0f, 0.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, devant);  
glColor3f(0.0f, 1.0f, 0.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, deriere);  
glColor3f(0.0f, 0.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, droite);  
glColor3f(0.0f, 1.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, gauche);  
glColor3f(1.0f, 1.0f, 0.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, haut);  
glColor3f(1.0f, 0.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bas);
```

# Comment procéder de manière générale

- Il s'agit d'abord d'activer jusqu'à six tableaux, stockant chacun un des six types de données suivantes :
  - Coordonnées des sommets
  - Couleurs RVBA
  - Index de positions dans les tableaux
  - Normales de surfaces
  - Coordonnées de texture
  - Indicateurs de contour des polygones
- Ensuite spécifier les données du ou des tableaux.
- Puis déréférencer le contenu des tableaux ( accéder aux éléments ) pour tracer une forme géométrique avec les données.

# Comment procéder de manière générale

- Il y a trois méthodes différentes pour déréférencer le contenu des tableaux :
  - Accéder aux éléments du tableau un par un : accès aléatoire
  - Créer une liste d'éléments du tableau : accès méthodique
  - Traiter les éléments du tableau de manière séquentielle : accès séquentiel ou systématique

# Activer les tableaux

- Cela se fait en appelant:

`glEnableClientState(GLenum array)`

- avec comme paramètre :  
`GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`,  
`GL_INDEX_ARRAY`, `GL_NORMAL_ARRAY`,  
`GL_TEXTURE_COORD_ARRAY`, ou  
`GL_EDGE_FLAG_ARRAY`.

- La désactivation d'un état se fait en appelant:

`glDisableClientState(GLenum array)`

- avec les mêmes paramètres.

# Spécifier les données des tableaux

- Il faut indiquer l'emplacement où se trouvent les données et la manière dont elles sont organisées.
- Les différents types de données (coordonnées des sommets, couleurs, normales, ...) peuvent avoir été placées dans différentes tables, ou être entrelacées dans une même table.

# Routines de spécification des tableaux

- Des routines permettent de spécifier des tableaux en fonction de leur type :
  - `void glVertexPointer(GLint taille, GLenum type, GLsizei stride, const GLvoid *pointeur);`
- Le paramètre `pointeur` indique l'adresse mémoire de la première valeur pour le premier sommet du tableau.
- Le paramètre `type` indique le type de données
- Le paramètre `taille` indique le nombre de valeurs par sommets, et peut prendre suivant les fonctions les valeurs 1, 2, 3 ou 4.
- Le paramètre `stride` indique le décalage en octets entre deux sommets successifs. Si les sommets sont consécutifs, il vaut zéro.

# Routines de spécification des tableaux

- Six routines permettent de spécifier des tableaux en fonction de leur type :
  - `void glVertexPointer(GLint taille, GLenum type, GLsizei stride, const GLvoid *pointeur);`
  - `void glColorPointer(GLint taille, GLenum type, GLsizei stride, const GLvoid *pointeur);`
  - `void glIndexPointer(GLenum type, GLsizei stride, const GLvoid *pointeur);`
  - `void glNormalPointer(GLenum type, GLsizei stride, const GLvoid *pointeur);`
  - `void glTexCoordPointer(GLint taille, GLenum type, GLsizei stride, const GLvoid *pointeur);`
  - `void glEdgeFlagPointer(GLsizei stride, const GLvoid *pointeur);`

# Accès à une liste d'éléments

- La méthode:

```
void glDrawElements(GLenum mode, GLsizei nombre,
GLenum type, void *indices)
```

permet d'utiliser le tableau **indices** pour stocker les indices des éléments à afficher.
- Le nombre d'éléments dans le tableau d'indices est **nombre**
- Le type de données du tableau d'indices est **type**, qui doit être `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, ou `GL_UNSIGNED_INT`
- Le type de primitive géométrique est indiqué par **mode** de la même manière que dans `glBegin()`.
- `glDrawElements()` ne doit pas être encapsulé dans une paire `glBegin()/ glEnd()`.

# Accès à une liste d'éléments

- glDrawElements() vérifie que les paramètres mode, nombre, et type sont valides, et effectue ensuite un traitement semblable à la séquence de commandes :

```
int i;  
glBegin(mode);  
for (i = 0, i < nombre ; i++)  
    glArrayElement(indice[i]);  
glEnd();
```

- Exemple d'accès à un ensemble d'éléments des tableaux construits dans l'exemple 1 :

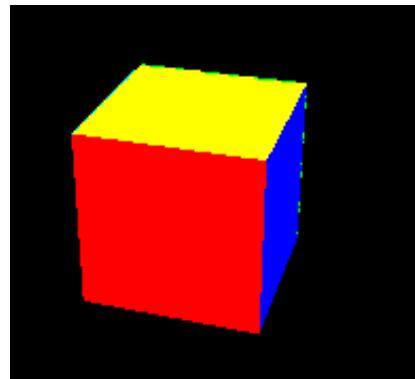
```
static GLubyte mesIndices[] = {1, 4};  
glDrawElements(GL_LINES, 2,  
               GL_UNSIGNED_BYTE, mesIndices);
```

# Exemple d'accès à une liste d'éléments de tableaux :

```
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, devant);  
glColor3f(0.0f, 1.0f, 0.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, derriere);  
glColor3f(0.0f, 0.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, droite);  
glColor3f(0.0f, 1.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, gauche);
```

# Exemple:

```
static GLubyte devant[] = {0,1,2,3};  
static GLubyte deriere[] = {4,5,6,7};  
static GLubyte droite[] = {1,5,6,2};  
static GLubyte gauche[] = {0,3,7,4};  
static GLubyte bas[] = {0,4,5,1};  
static GLubyte haut[] = {3,2,6,7};  
  
GLfloat points[] = {  
    0.0f, 0.0f, 0.0f,  
    1.0f, 0.0f, 0.0f,  
    1.0f, 1.0f, 0.0f,  
    0.0f, 1.0f, 0.0f,  
    0.0f, 0.0f, -1.0f,  
    1.0f, 0.0f, -1.0f,  
    1.0f, 1.0f, -1.0f,  
    0.0f, 1.0f, -1.0f,  
}; ;
```

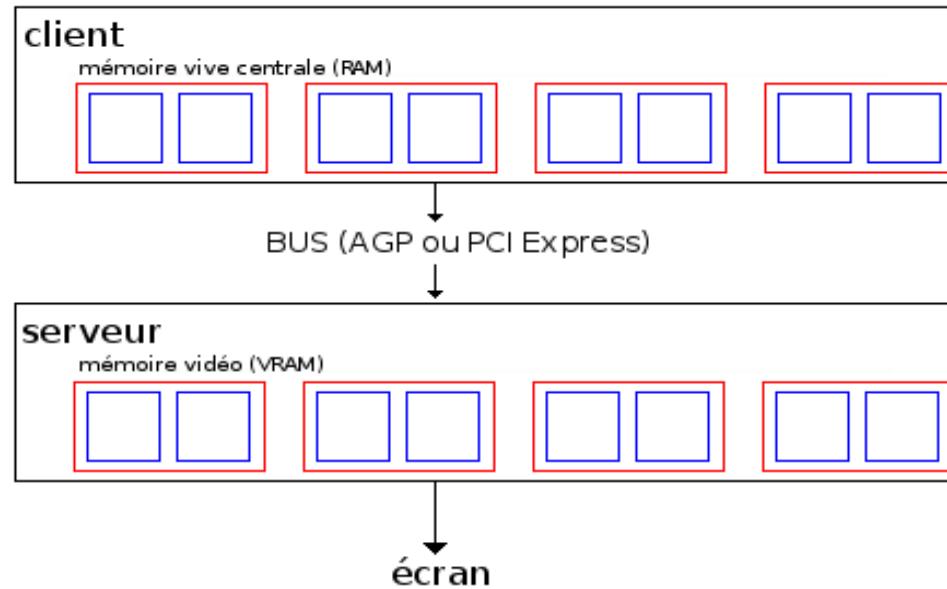


```
glVertexPointer(3, GL_FLOAT, 0, points);  
glTranslatef(-0.5, -0.5, 0.5);  
glColor3f(1.1f, 0.0f, 0.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, devant);  
glColor3f(0.0f, 1.0f, 0.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, deriere);  
glColor3f(0.0f, 0.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, droite);  
glColor3f(0.0f, 1.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, gauche);  
glColor3f(1.0f, 1.0f, 0.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, haut);  
glColor3f(1.0f, 0.0f, 1.0f);  
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bas);
```

# Tableaux de sommets en mémoire graphique

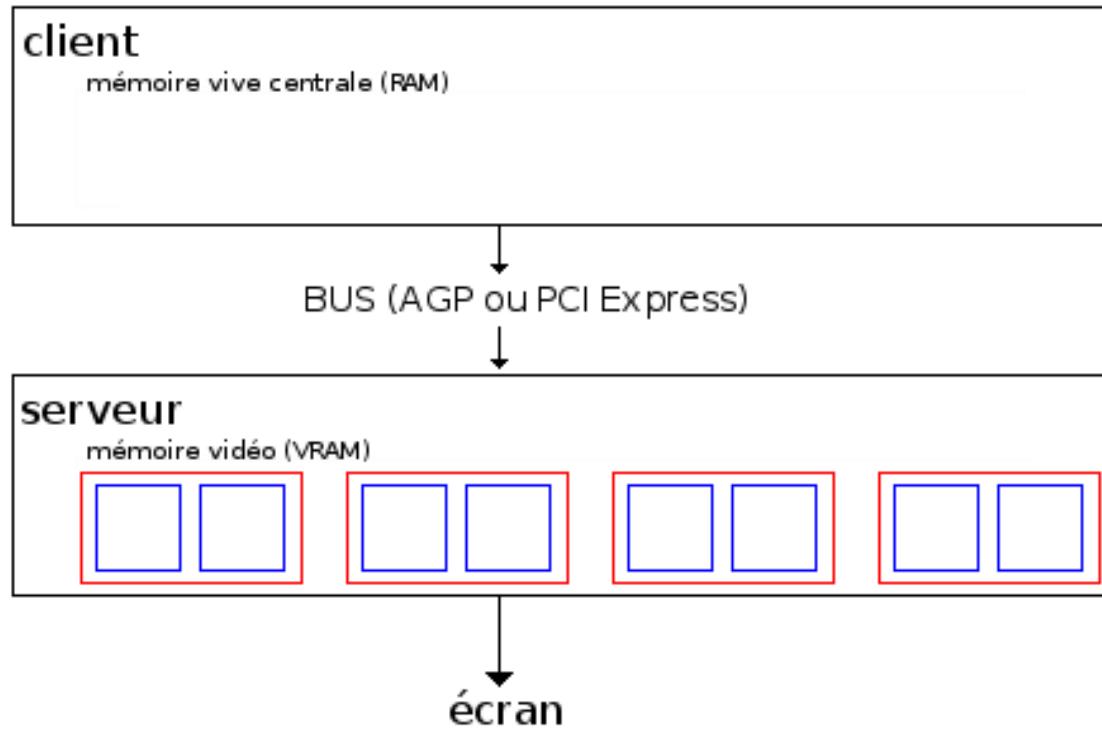
- Résumons le principe des vertex arrays :
  - 1.activation;
  - 2.spécification des données;
  - 3.rendu;
  - 4.désactivation.
- Lors de la seconde étape, OpenGL ne fait que retenir un pointeur et des informations sur la structuration des données (stride, type et size).
- À la 3eme étape, OpenGL envoie les informations au serveur pour le traitement.

# Tableaux de sommets en mémoire graphique



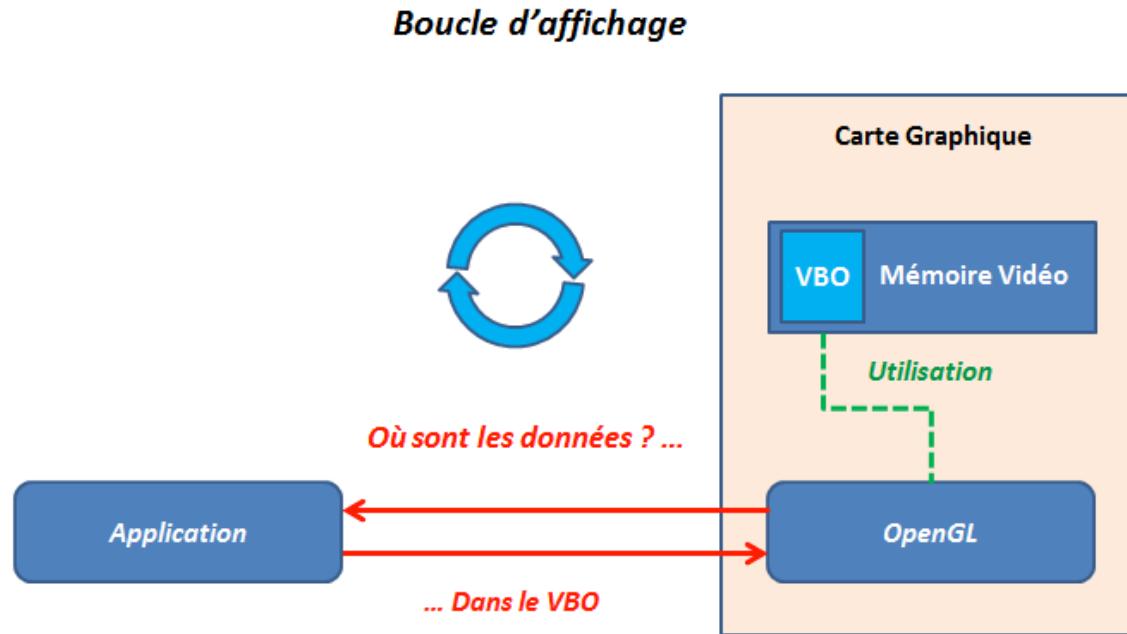
- Les données sont dupliquées en mémoire vidéo, puis sont traitées pour enfin être affichées à l'écran.
- Ce processus a lieu à chaque rendu ( `glDraw*()` )
- Le transfert de millions de vertices à chaque rendu mènerait à une saturation de la bande passante, et donc à une limitation de frame rate.

# Tableaux de sommets dans les objets tampon



- Les données ne sont ni dupliquées, ni transférées à chaque rendu
- Lors d'un appel à `glDraw*()`, OpenGL ira directement chercher les données en mémoire vidéo.

# Tableaux de sommets dans les objets tampon

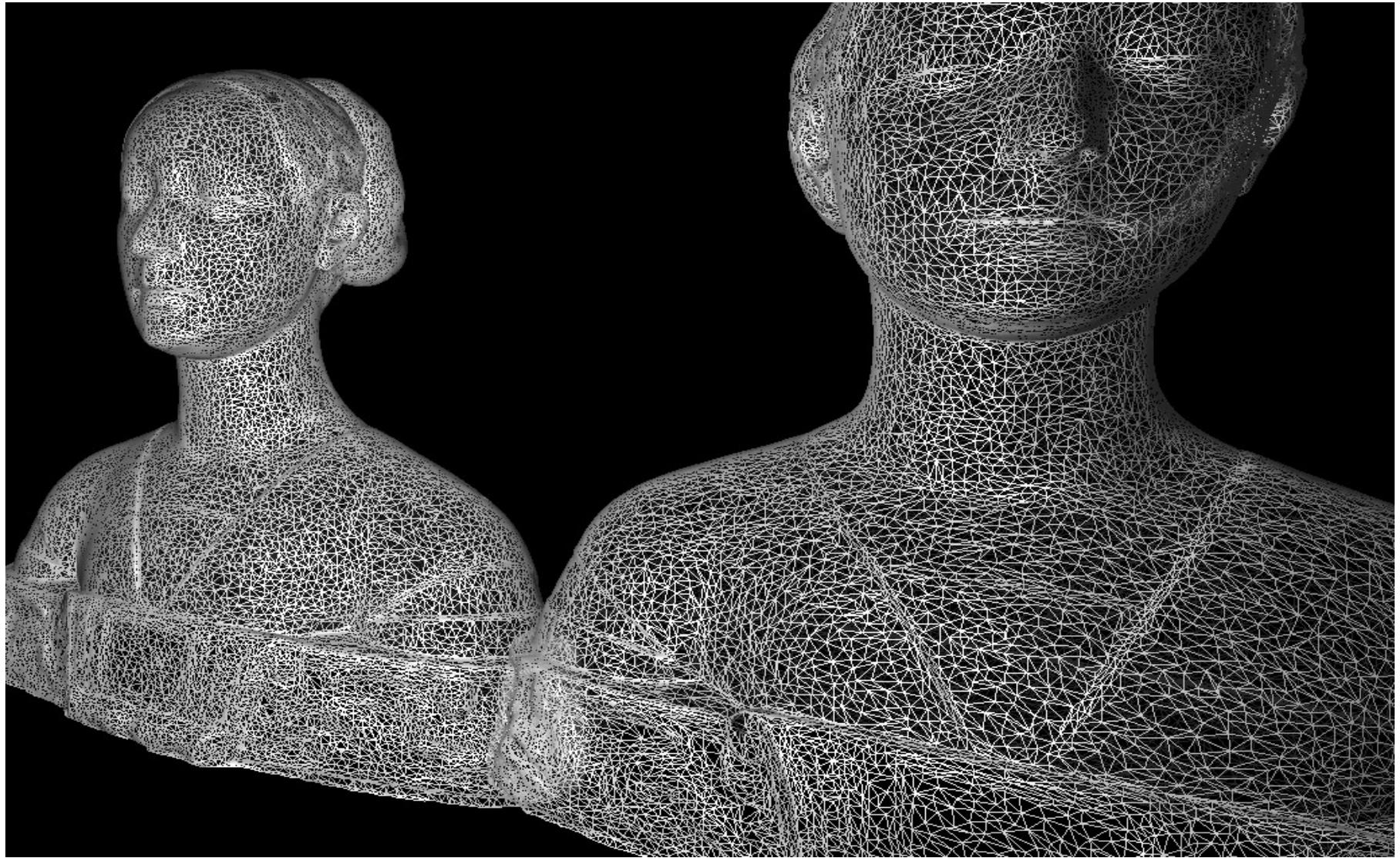


- Les données ne sont ni dupliquées, ni transférées à chaque rendu
- Lors d'un appel à `glDraw*()`, OpenGL ira directement chercher les données en mémoire vidéo.

# Sans VBO



# Avec VBO



# Exploiter les objets tampon

- Un objet tampon est un objet OpenGL, donc un GLuint.
- Voici comment créer des identifiants pour des objets tampon :
- Déclaration:

```
#define VERTICES 0  
  
#define INDICES 1  
  
#define NUM_BUFFERS 2  
  
GLuint buffers[NUM_BUFFERS];
```

- Génération des noms d'objets tampons (ils sont garantis non précédemment utilisé):

```
glGenBuffers(NUM_BUFFERS, buffers);
```

# Exploiter les objets tampon

- Pour utiliser l'objet tampon il faut le récupérer et le **lier** (bind) un nom à la machine à état:

```
glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTICES]);
```

- Liaison spécifique pour les indices

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[INDICES]);
```

- Prototype:

```
void glBindBuffer(GLenum target, GLuint id);
```

- 0 comme id désactive la gestion des tampons

# Exploiter les objets tampon

- Allouer et initialiser des objets tampon avec des données de sommet:

```
void glBufferData(GLenum target, GLsizei size, const GLvoid *data,  
GLenum mode);
```

target : Type objet tampon cible (GL\_ARRAY\_BUFFER ou  
GL\_ELEMENT\_ARRAY\_BUFFER)

size : c'est la taille des données, en Bytes.

data : l'emplacement des données(pointeur en mémoire CPU).

mode : paramètre qui sert à définir le mode de traitement des données lors de leur mise à jour. Il définit la fréquence d'accès aux données.

GL\_STREAM\_DRAW: mise à jour entre chaque affichage (équiv. mode classique des vertex arrays).

GL\_STATIC\_DRAW: mise à jour rare.

GL\_DYNAMIC\_DRAW:mise à jour plusieurs fois par frame (rendu multipass)

# Exploiter les objets tampon

GL\_STREAM\_DRAW/GL\_STATIC\_DRAW/GL\_DYNAMIC\_DRAW

Il existe d'autres modes utiles aux geometry shaders (ou primitive shaders) qui permettent d'instancier des sommets depuis le GPU.:

GL\_STREAM\_READ/GL\_STATIC\_READ/GL\_DYNAMIC\_READ

GL\_STREAM\_COPY/GL\_STATIC\_COPY/GL\_DYNAMIC\_COPY

# Exploiter les objets tampon

- Allouer et initialiser des objets tampon avec des données de sommet:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
```

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

- avec:

```
static GLfloat points[][3] = {{0.0f, 0.0f, 0.0f}, {1.0f, 0.0f, 0.0f}};
```

```
static GLubyte indice[] = {0,1};
```

# Exploiter les objets tampon

- Afficher les données:

```
glVertexPointer(3, GL_FLOAT, 0, 0);
```

- En fait, à la place de donner un tableau à gl\*Pointer(), on donne sa **position** de stockage dans le tampon

# Exploiter les objets tampon

- Afficher les données:

```
glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTICES]);  
glVertexPointer(3, GL_FLOAT, 0, 0);  
glEnableClientState(GL_VERTEX_ARRAY);  
glDrawElements(GL_TRIANGLE_STRIP, 12, GL_UNSIGNED_BYTE, 0);
```

# Exploiter les objets tampon en C: init()

```
#define VERTICES 0
#define DEVANT 1
#define DERRIERE 2
#define DROITE 3
#define GAUCHE 4
#define BAS 5
#define HAUT 6
#define COLOR 7
//#define INDICES 8

#define NUM_BUFFERS 9

#define BUFFER_OFFSET(bytes) ((GLubyte*)NULL + (bytes))
```

```
glGenBuffers(NUM_BUFFERS, buffers); //Création des identifiants des objets tampons
glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTICES]); //activations du tampon de vertices

glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW); // allocation et initialisation des données
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
//spécification de tableau lié cette fois au tampon
 glEnableClientState(GL_VERTEX_ARRAY); //activation de tableau

//idem pour les couleurs
glBindBuffer(GL_ARRAY_BUFFER, buffers[COLOR]);
glBufferData(GL_ARRAY_BUFFER, sizeof(couleurs), couleurs, GL_STATIC_DRAW);
glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));
 glEnableClientState(GL_COLOR_ARRAY);
```

# Exploiter les objets tampon en C: init()

```
#define VERTICES 0
#define DEVANT 1
#define DERRIERE 2
#define DROITE 3
#define GAUCHE 4
#define BAS 5
#define HAUT 6
#define COLOR 7
//#define INDICES
8

#define NUM_BUFFERS 9

#define BUFFER_OFFSET(bytes) ((GLubyte*)NULL + (bytes))
```

```
//mise en place des tableaux d'indices en tampon
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[DEVANT]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(devant), devant, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[DERRIERE]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(derriere), derriere, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[DROITE]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(droite), droite, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[GAUCHE]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(gauche), gauche, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[BAS]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(bas), bas, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[HAUT]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(haut), haut, GL_STATIC_DRAW);
```

# Exploiter les objets tampon en C: draw()

```
#define VERTICES 0
#define DEVANT 1
#define DERRIERE 2
#define DROITE 3
#define GAUCHE 4
#define BAS 5
#define HAUT 6
#define COLOR 7
//#define INDICES 8

#define NUM_BUFFERS 9

#define BUFFER_OFFSET(bytes)
((GLubyte*) NULL + (bytes))
```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[DEVANT]);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[DERRIERE]);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[DROITE]);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[GAUCHE]);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[BAS]);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));

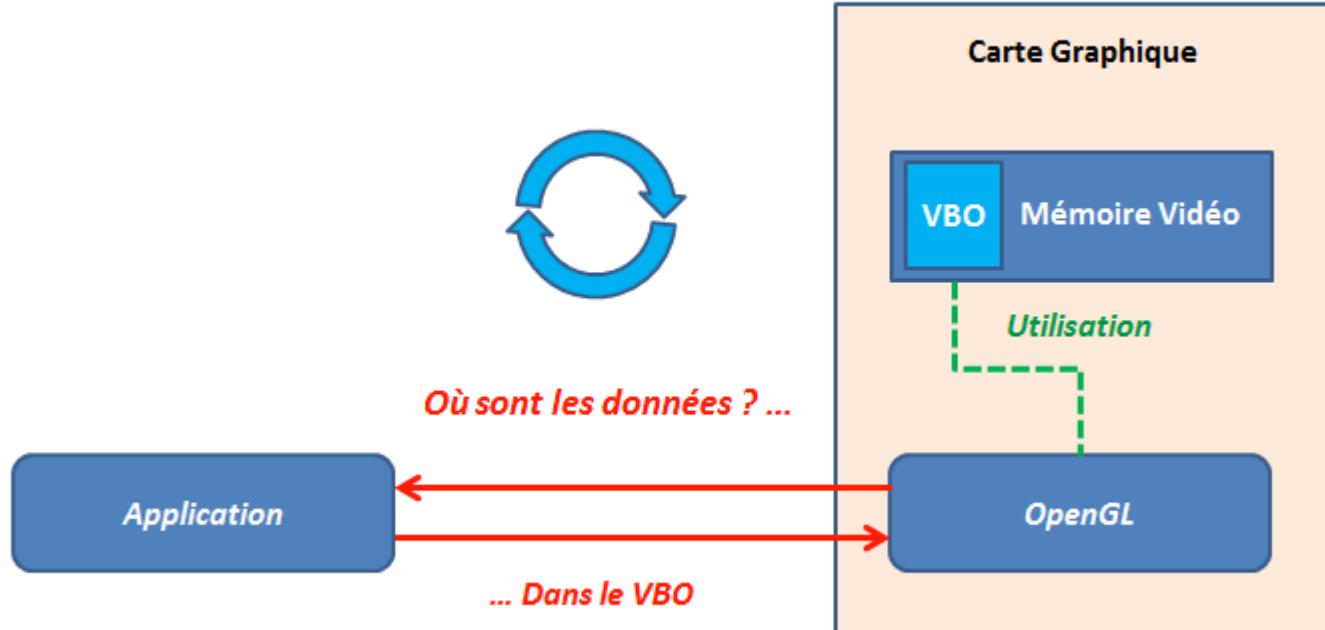
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[HAUT]);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, BUFFER_OFFSET(0));
```

# Et les VAO ?

- Idem que VBO mais cette fois pour les appels de fonctions
- Idée : supprimer les derniers appels du CPU

## Boucle d'affichage

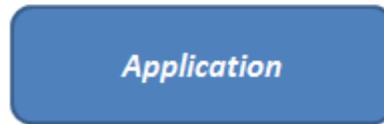
AVEC  
les  
VBO  
Sans  
les  
VAO



# Et les VAO ?

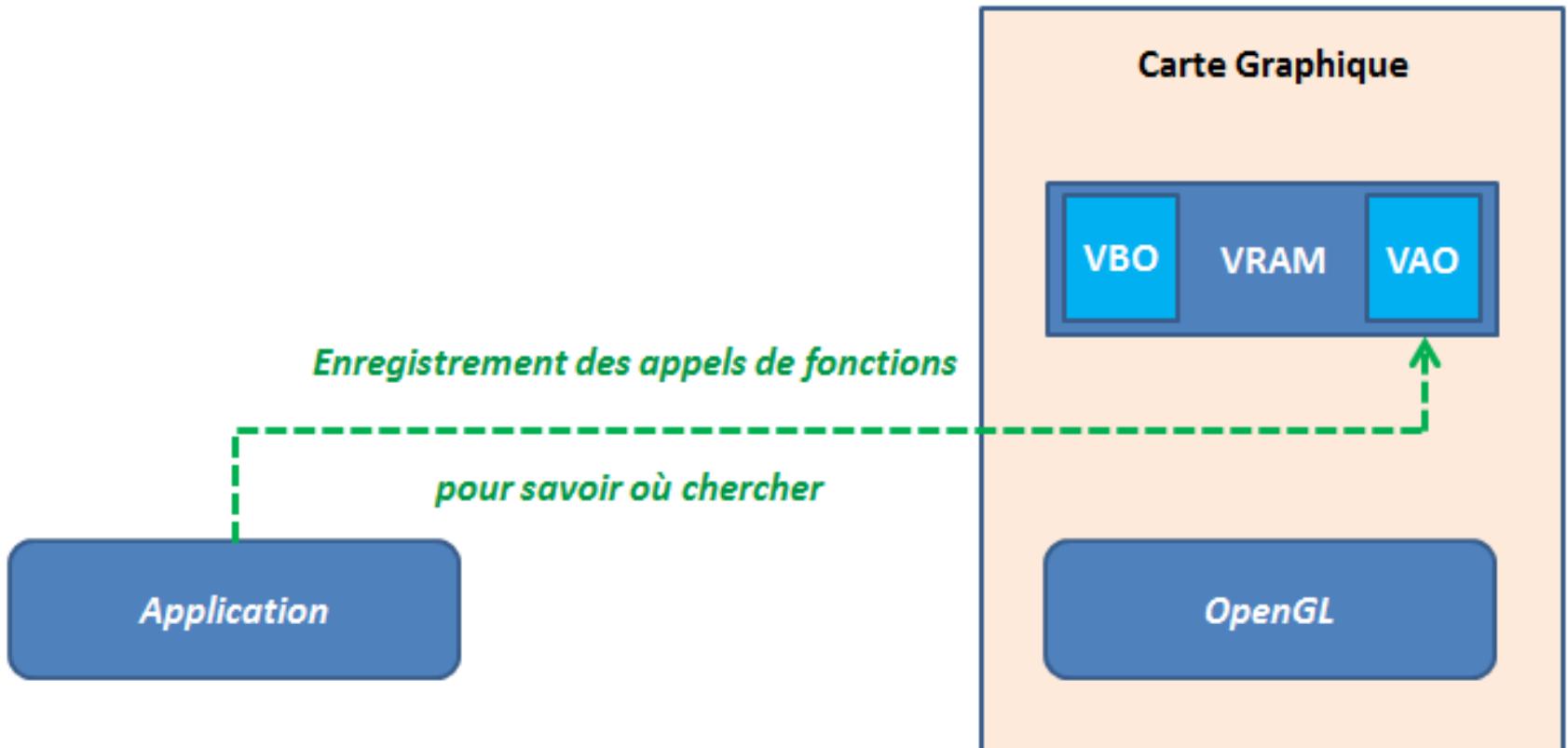
- Idem que VBO mais cette fois pour les appels de fonctions
- Idée : supprimer les derniers appels du CPU

*Boucle d'affichage*



# Mise en place des VAO ?

## *Chargement du modèle 3D*



# Mise en place des VAO ?

- Uniquement dans le cadre des shaders

```
// Génération d'un Vertex Array Object contenant 2 Vertex Buffer Objects.  
  
    glGenVertexArrays( 1, &vaoIds[ 0 ] );  
    glBindVertexArray( vaoIds[ 0 ] ); //Activation du VAO  
  
    // Génération de 2 VBO.  
    glGenBuffers( 2, vboids );  
  
    // VBO contenant les sommets.  
    glBindBuffer( GL_ARRAY_BUFFER, vboids[ 0 ] );  
    glBufferData( GL_ARRAY_BUFFER, vertices.size() * sizeof( float ),  
vertices.data(), GL_STATIC_DRAW );  
  
    // VBO contenant les couleurs.  
    glBindBuffer( GL_ARRAY_BUFFER, vboids[ 1 ] );  
    glBufferData( GL_ARRAY_BUFFER, colors.size() * sizeof( float ),  
colors.data(), GL_STATIC_DRAW );  
  
    // VBO contenant les indices.  
    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, vboids[ 2 ] );  
    glBufferData( GL_ELEMENT_ARRAY_BUFFER, indices.size() *  
sizeof( unsigned short ), indices.data(), GL_STATIC_DRAW );  
  
    glBindVertexArray( 0 );// Désactivation du VAO.
```

# Utilisation des VAO ?

- Uniquement dans le cadre des shaders

```
// Calcul de la matrice mvp.
```

```
mvp = proj * view;
```

```
// Passage de la matrice mvp au shader (envoi vers la carte graphique).
```

```
glUniformMatrix4fv( mvpid , 1, GL_FALSE, &mvp[0][0]);
```

```
// Dessin de 1 triangle à partir de 3 indices.
```

```
glBindVertexArray( vaoids[ 0 ] );
```

```
glDrawElements( GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, 0 );
```

```
glutSwapBuffers();
```