

Vertex Buffer Object

Christophe [Groove] Riccio
www.g-truc.net
Création : 01 Mai 2006
Révision 1 : 30 Juin 2006
Révision 2 : 28 Septembre 2006

Sommaire

Introduction

1. Description
 - 1.1. L'héritage d'OpenGL
 - 1.2. Intérêts des VBOs
 - 1.3. Support non complet
2. Pratique
 - 2.1. Utilisation de base façon Vertex Array (classe CTest1)
 - 2.2. Utilisation avec index buffer (classe CTest2)
 - 2.3. Utilisation des tableaux entrelacés (classe CTest3)
 - 2.4. Utilisation des tableaux sérialisés (classe CTest4)
 - 2.5. Vertex mapping (classe CTest5)
3. Références
 - 3.1. Usage des buffers
 - 3.2. Les fonctions de rendu
 - 3.3. Les types de tableaux
 - 3.4. Les données des tableaux
 - 3.5. Types de primitive

Introduction

Le vertex buffer object (VBO) est une nouvelle méthode utilisée pour décrire les primitives géométries introduite avec les extensions `NV_vertex_array_range` et `ATI_vertex_array_object` promu en `ARB_vertex_buffer_object` puis intégré dans les spécifications d'OpenGL 1.5. Seules les VBOs devrait subsister à OpenGL LM.

OpenGL LM est le projet de l'ARB pour simplifier l'API d'OpenGL pour ainsi faciliter l'optimisation des drivers. Son objectif est de venir concurrencer Direct Graphics dans le secteur du jeu vidéo et devrait se baser sur les spécifications d'OpenGL 3.0. L'ARB entend simplifier l'API d'OpenGL, ce que de nombreux développeurs souhaitaient pour OpenGL 2.0. Pour se faire, le mode immédiat, les vertex arrays et `glRect` devraient disparaître.

Les VBOs sont supportés par à partir des cartes nVidia TNT et des premières ATI Radeon. Cependant toutes les fonctionnalités ne sont pas forcément disponibles et certaines ne sont supportées par aucune carte actuelle.

Fonctionnalités, performance et longévité, trois raisons pour utiliser les VBOs au plus vite.

Ce document est accompagné d'un programme d'exemples disponible aux adresses suivantes :

www.g-truc.net/article/vbo.zip (9 Mo)
www.g-truc.net/article/vbo.7z (2.5 Mo)

1. Description

1.1. L'héritage d'OpenGL

Les débutants dans l'utilisation d'OpenGL apprécient sa simplicité par le biais du mode immédiat apparu avec OpenGL 1.0. Ce mode se présente ainsi :

```
glBegin(GL_QUADS);  
    glColor3f(1.0f, 0.5f, 0.0f);  
    glVertex2f(0.0f, 0.0f);
```

```
    glVertex2f(1.0f, 0.0f);
    glVertex2f(1.0f, 1.0f);
    glVertex2f(0.0f, 1.0f);
glEnd();
```

Ce code est très compréhensible, mais cette méthode peut devenir très lente lorsqu'il s'agit de décrire une primitive géométrique complexe composée de milliers de sommets, simplement du fait des appels de fonctions trop nombreux.

OpenGL 1.1 a vu l'apparition des vertex arrays permettant l'envoi d'un très grand nombre de données en peu d'appels de fonctions. Ce mode se présente ainsi :

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

float ColorArray[] = {...};
float VertexArray[] = {...};

glColorPointer(3, GL_FLOAT, 0, ColorArray);
glVertexPointer(3, GL_FLOAT, 0, VertexArray);

glDrawArrays(GL_TRIANGLES, 0, sizeof(VertexArray) / sizeof(float));

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

Les vertex buffer objects sont similaires aux vertex arrays dans la pensée mais proposent de nombreuses améliorations.

1.2. Intérêts des VBOs

Les VBOs fournissent 3 modes de transferts des données au lieu d'un seul pour les vertex arrays.

Le premier nommé *GL_STREAM_DRAW* est équivalent au fonctionnement des vertex arrays, c'est-à-dire que les données sont transférées pour chaque appel de *glDrawArrays*, *glDrawElements*, *glDrawRangeElements* (EXT_draw_range_elements, OpenGL 1.2) *glMultiDrawArrays* ou *glMultiDrawElements* (EXT_multi_draw_array, OpenGL 1.4). Ce mode est particulièrement efficace pour les objets animés tel un personnage.

Un second mode nommée *GL_STATIC_DRAW* hérite des fonctionnalités de l'extension EXT_compiled_vertex_array. Il permet de n'envoyer les informations qu'une seule fois à la carte graphique qui sont alors stockées dans la mémoire de la carte graphique. Ce mode est parfaitement adapté pour les objets non déformables, c'est-à-dire pour toute géométrie qui reste valable pour plusieurs frames. C'est également le mode pouvant offrir de plus de puissance brute car l'espace mémoire est réservé et la bande passante mémoire des cartes graphiques est sans commune mesure plus élevée que celle du CPU.

Le dernier mode est *GL_DYNAMIC_DRAW*. Dans ce mode, ce sont les drivers de la carte graphique qui vont choisir l'emplacement des données. Ce mode est recommandé pour le rendu d'objets animés affichés plusieurs fois par frame, par exemple pour le rendu multi passe.

Pour résumer, utilisez le mode :

- *GL_STREAM_DRAW* lorsque les informations sur les sommets peuvent être mise à jour entre chaque affichage.
- *GL_DYNAMIC_DRAW* lorsque les informations sur les sommets peuvent être mise à jour entre chaque frame.
- *GL_STATIC_DRAW* lorsque les informations sur les sommets ne sont jamais ou presque jamais mise à jour.

Notez cependant que ces trois modes sont de plus en plus vu par les drivers comme des recommandations, ils n'en feront peut-être qu'à leur tête, il est donc tout à fait possible que vous n'observiez pas de gains spécifiques entre les différents modes.

La seconde amélioration qui caractérise les VBOs est nommée Vertex mapping. Elle représente la possibilité de ne pas utiliser un tableau temporaire pour stocker les données mais directement un tableau alloué par OpenGL. Notons que les VBOs sont aussi disponibles avec OpenGL ES 2.0 mais que cette fonctionnalité est optionnelle. Cette fonctionnalité a vu le jour au travers de l'extension `ATI_map_buffer_object` disponible depuis les premières ATI Radeon. nVidia propose également cette fonctionnalité depuis l'extension `NV_vertex_array_range` cependant l'allocation mémoire était faite manuellement via le contexte OpenGL. Avec les VBOs cette allocation est transparente.

Enfin, les VBOs améliorent véritablement les tableaux entrelacés. Pour rappel, les tableaux entrelacés permettent d'envoyer les informations sur les sommets à la carte graphique en utilisant un unique tableau de données décrit par la fonction *glInterleavedArrays*. Avec les vertex arrays, il fallait utiliser des structures de données prédéfinies dans un ordre précis. Avec l'arrivée de la programmation GPU, les attributs personnalisés du programmeur ne trouvent plus leurs places dans ce système. Les VBOs résolvent ce problème et permettent de ne mettre à jour qu'une partie des données.

1.3. Support incomplet

Les VBOs proposent une API vraiment très aboutie parfaitement en accord avec son temps et avec l'avenir. Ainsi, ils proposent 6 autres modes de transfert nommés `GL_STREAM_READ`, `GL_STREAM_COPY`, `GL_DYNAMIC_READ`, `GL_DYNAMIC_COPY`, `GL_STATIC_READ` et `GL_STATIC_COPY`.

Malheureusement, aucune carte graphique actuelle ne supporte ces fonctionnalités à ma connaissance. La technologie Render to Vertex Buffer (R2VB) d'ATI sur les Radeon X1*00 semble vaguement correspondre aux fonctionnalités proposées par ces 6 modes. Cependant, ATI n'a pas communiqué sur la partie OpenGL du sujet mais elle semble correspondre aux pixel buffer objects (`ARB_pixel_buffer_object`) et n'apporte rien de neuf au point de vue d'OpenGL.

Les modes utilisant le terme `GL_*_READ` permettent de lire des informations gérées par OpenGL. Les règles sur les termes `GL_STREAM_*`, `GL_DYNAMIC_*` et `GL_STATIC_*` sont identiques mais réfèrent à présent au nombre de lectures de données par le programme. Les modes utilisant le terme `GL_*_COPY` permettent d'afficher une géométrie à partir d'une source gérée par OpenGL.

Concrètement, à quoi serviront ces modes ? Pour la prochaine génération de cartes graphiques connues sous les noms de code G80 chez nVidia et R600 chez ATI de nouvelles possibilités seront disponibles. Elles sont galvaudées sous des noms DirectX tel que Shader Model 4.

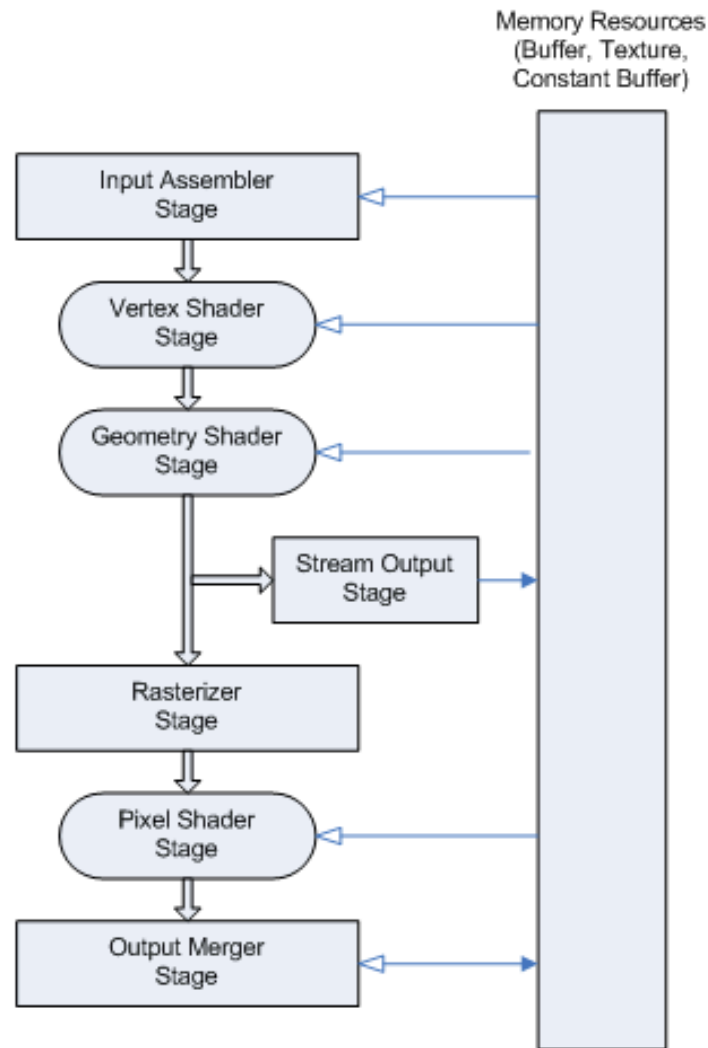


Image tirée de la documentation de DirectX 10 représentant le pipeline graphique

Les geometry shaders, nommés primitive shaders sous OpenGL, permettront d'instancier des sommets depuis le GPU. Pour cela et comme le montre la figure précédente, le pipeline graphique va évoluer en permettant une sortie d'information indiquée ici sous le terme « stream output stage ». Le flux de données nécessite une API pour être traité, les modes *GL_*_COPY* et *GL_*_READ* des VBOs sont tout indiqués pour répondre à ces nouveaux besoins. Les modes *GL_*_COPY* semblent à même de créer des traitements récursifs sur les sommets ce qui pourrait considérablement augmenté les ressources nécessaire au niveau des vertex pipelines. La démarche marketing d'ATI avec le R2VB s'appuie sur une analogie avec ce nouveau « stream output stage » en utilisant les pixels comme stream output. Cependant, il n'est pas possible de créer des pixels avec une Radeon X1*00.

2. Pratique

Chaque sous partie de la partie pratique est associée à une classe dans le programme accompagnant cette article. Deux autres exemples non décrit dans ce document montre l'utilisation de GLSL et Cg avec les VBOs. L'implémentation avec des classes a pour but unique de rendre interchangeable les différentes méthodes dans le programme d'exemple.

2.1. Utilisation de base façon Vertex Array (classe CTest1)

Pour faciliter la compréhension, commençons par un exemple correspondant parfaitement au fonctionnement de base des vertex arrays. Les VBOs utilisent une API similaire aux objets de textures pour leurs gestions.

```
GLvoid glGenBuffers(GLsizei n, GLuint* buffers);
```

```
GLvoid glDeleteBuffers(GLsizei n, const GLuint* buffers);
```

buffers est un tableau créé par l'utilisateur dans lequel sont stockés les identifiants des VBOs. *n* objets sont créés ou détruits, attention donc à la taille de *buffers*.

Admettons que nous souhaitons afficher un carré à l'écran au moyen de deux triangles. Nos sources sont par exemple :

```
static const GLsizeiptr PositionSize = 6 * 2 * sizeof(GLfloat);
static const GLfloat PositionData[] =
{
    -1.0f, -1.0f,
     1.0f, -1.0f,
     1.0f,  1.0f,
     1.0f,  1.0f,
    -1.0f,  1.0f,
    -1.0f, -1.0f,
};

static const GLsizeiptr ColorSize = 6 * 3 * sizeof(GLubyte);
static const GLubyte ColorData[] =
{
    255,  0,  0,
    255, 255,  0,
     0, 255,  0,
     0, 255,  0,
     0,  0, 255,
    255,  0,  0
};
```

Nous allons utiliser 2 VBOs pour afficher les 6 sommets décrits par les tableaux précédant. Les tableaux sont identifiés par *GL_POSITION_OBJECT* et *GL_COLOR_OBJECT* simplement par commodité. La création et la destruction des VBOs s'effectuent avec les fonctions *glGenBuffers* et *glDeleteBuffers*. La fonction *glBindBuffer* permet de sélectionner le VBO actif.

```
static const int BufferSize = 2;
static GLuint BufferName[BufferSize];

static const GLsizei VertexCount = 6;

enum
{
    POSITION_OBJECT = 0,
    COLOR_OBJECT = 1
};
```

Le code pour afficher ce carré est alors :

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[COLOR_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, ColorSize, ColorData, GL_STREAM_DRAW);
glColorPointer(3, GL_UNSIGNED_BYTE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, PositionData, GL_STREAM_DRAW);
glVertexPointer(2, GL_FLOAT, 0, 0);

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, VertexCount);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

glBufferData initialise le stockage de données du VBO. Le dernier paramètre spécifie l'usage du VBO tel qu'il est décrit en section 1.2 et 1.3. La liste de tous les usages est disponible en section 3.1. Les fonctions

glColorPointer et *glVertexPointer* permettent de spécifier l'emplacement pour trouver respectivement les couleurs et les coordonnées spatiales des sommets.

L'ordre de ces trois appels de fonctions est particulièrement important. Il est intuitif qu'il faut sélectionner le VBO actif en premier pour le paramétrer cependant l'ordre de *glBufferData* et *gl*Pointer* est également important. En effet, *gl*Pointer* réfère à la source des données du VBO actif, cette source étant initialisée par *glBufferData*.

Remarques :

- Parfois, on préférera séparer la tâche de chargement des données et la tâche de description des données pour des problèmes de flexibilité d'utilisation. Ainsi, la solution suivante est tout à fait viable :

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, PositionData, GL_STREAM_DRAW);
...
glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glVertexPointer(3, GL_FLOAT, 0, 0);
```

- Dès lors que la fonction *glBindBuffer* est appelée avec un nom de VBOs valide, OpenGL bascule en mode VBO. Pour revenir en mode Vertex Array, il faut utiliser *glBindBuffer* avec la valeur 0 comme nom d'objet.

Le rendu s'effectue avec l'une des fonctions dédiées aux rendus de tableaux : *glDrawArrays* ou *glMultiDrawArrays* dans cette exemple.

Dans le cas plutôt rare où la taille totale de la mémoire de la carte graphique est inférieure à la taille que l'on demande de réserver pour un unique VBO, une erreur de type *GL_OUT_OF_MEMORY* est émise et récupérable comme habituellement avec *glGetError*.

2.2. Utilisation avec index buffer (classe CTest2)

Lors du premier exemple nous avons utilisé la cible *GL_ARRAY_BUFFER*. Elle est utilisée pour tous types de données excepté les tableaux d'indices qui utilisent la cible *GL_ELEMENT_ARRAY_BUFFER*.

L'initialisation d'un tableau d'indices s'effectue ainsi :

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, BufferName[INDEX_OBJECT]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndexSize, IndexData, GL_STREAM_DRAW);
```

C'est la partie qui diffère le plus des vertex arrays. Si la fonction de rendu par élément est utilisée avec la valeur nulle à la place d'un tableau d'indices alors le VBO actif ayant pour cible *GL_ELEMENT_ARRAY_BUFFER* est utilisé.

Dans cette exemple, le rendu s'effectue avec l'une des fonctions dédiées aux tableaux indexés : *glDrawElements*, *glDrawRangeElements* ou *glMultiDrawElements*.

2.3. Utilisation des tableaux entrelacés (classe CTest3)

Il n'y a pas eu de mise à jours de la fonction *glInterleavedArrays* [3.4.2] depuis son arrivée avec OpenGL 1.1. Cette fonction fut largement utilisée pour le rendu avec tableau entrelacé. Elle peut encore être utilisée avec les VBOs cependant, il y a une alternative bien plus intéressante basée sur les fonctions *gl*Pointer*. L'idée est de spécifier pour chaque attribut du tableau entrelacé la source des données en utilisant le paramètre de *stride*.

```
#pragma pack(push, 1)
struct SVertex
{
    GLubyte r;
    GLubyte g;
    GLubyte b;
    GLfloat x;
    GLfloat y;
};
#pragma pack(pop)
```

```

glBindBuffer(GL_ARRAY_BUFFER, BufferName);
glBufferData(GL_ARRAY_BUFFER, VertexSize, VertexData, GL_STREAM_DRAW);

glColorPointer(3, GL_UNSIGNED_BYTE, sizeof(SVertex),
BUFFER_OFFSET(ColorOffset));
glVertexPointer(2, GL_FLOAT, sizeof(SVertex), BUFFER_OFFSET(VertexOffset));

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, VertexCount);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);

```

Pour cet exemple nous utilisons une structure qui nous permet d'entrelacer les données. Notons que la définition de la structure est entourée des instructions du pré-processeur standard **#pragma pack**. En effet, si l'on recherche la taille de la structure avec l'instruction `sizeof`, il y a toutes les chances que la valeur retournée soit 12 voir 16 octets au lieu de 11 dans le cas présent. **GLfloat** fait usuellement 4 octets, et **GLubyte** fait usuellement 1 octet donc une taille requise de 11 octets. Cependant, les compilateurs alignent les données en mémoire car les processeurs sont optimisés pour récupérer des données de la taille de leurs registres. 4 octets pour les CPU 32 bits et 8 octets pour les CPU 64 bits. Bonne initiative mais quand l'espace mémoire coûte chère nous allons oublier cette optimisation. En effet, dans notre cas, non seulement les structures coûtent 1/12 de mémoire supplémentaire mais c'est aussi 1/12 de données à transférer jusqu'à la carte graphique. Enfin, il se peut que des difficultés apparaissent quant à la gestion des octets supplémentaires, particulièrement dans le cas présent. Où se trouve l'octet supplémentaire ?

Les fonctions **glColorPointer** et **glVertexPointer** doivent toujours indiquer les sources et les types des données stockés dans le VBO. Pour cela, les spécifications proposent une macro nommée **BUFFER_OFFSET** :

```
#define BUFFER_OFFSET(i) ((char*)NULL + (i))
```

Avec les VBOs, il ne s'agit plus de donner l'adresse de la source de donnée car la source est stockée quelques par, par le VBO mais un offset qui indique où l'on doit commencer à lire les données dans la mémoire du VBOs. **sizeof(SVertex)** est dans cet exemple la valeur de **stride** c'est-à-dire qu'elle indique le nombre d'octets entre deux sommets pour un même attribut. Habituellement, cette valeur est nulle pour simplifier l'API OpenGL. Ceci signifie que les valeurs sont contiguës, c'est-à-dire que le VBO ne contient qu'un seul attribut par vertex et aucun espace vide. En conséquence, si nous créons un tableau ne contenant que les positions spatiales 3D des sommets, alors les deux appels suivant sont équivalents :

```

glVertexPointer(3, GL_FLOAT, 0, 0);
glVertexPointer(3, GL_FLOAT, sizeof(float) * 3, 0);

```

La macro **BUFFER_OFFSET** permet également d'éviter un warning de conversion d'un entier en pointer.

2.4. Utilisation des tableaux sérialisés (classe CTest4)

Pour de nombreux cas, les données utilisées pour décrire les primitives géométriques n'ont aucune raison d'être entrelacé, voir ce choix peut-être nuisible. Il est tout à fait possible que nous ne voulions mettre à jour qu'une partie des données. Par exemple dans le cas de l'animation d'un maillage, tel qu'un personnage, les coordonnées de textures n'ont pas besoin d'être mise à jour au contraire des positions et des normales des sommets.

Pour cela, nous n'utilisons qu'un seul VBO dans lequel nous insérons plusieurs types de données au moyen de la fonction **glBufferSubData**.

En premier lieu, nous réservons l'espace mémoire pour l'intégralité des données avec la fonction **glBufferData**. Au lieu de passer la source des données dans le troisième paramètre nous utilisons la valeur 0.

En suite, nous utilisons la fonction *glBufferSubData* pour remplir le tableau. Le second paramètre correspond à l'offset dans les données du VBO. Le troisième indique la taille des données sources à ajouter et le dernier est la source des données.

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName);
glBufferData(GL_ARRAY_BUFFER, ColorSize + PositionSize, 0, GL_STREAM_DRAW);

glBufferSubData(GL_ARRAY_BUFFER, 0, ColorSize, ColorData);
glBufferSubData(GL_ARRAY_BUFFER, ColorSize, PositionSize, PositionData);

glColorPointer(3, GL_UNSIGNED_BYTE, 0, 0);
glVertexPointer(2, GL_FLOAT, 0, BUFFER_OFFSET(ColorSize));

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, VertexCount);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

La fonction *glBufferSubData* peut être utilisée pour mettre à jour seulement une partie des données par exemple dans le cas de modèles partiellement animés ou lorsque plusieurs modèles sont stockés dans un seul VBO, ce qui peut-être très efficaces.

2.5. Vertex mapping (classe CTest5)

Dans certains cas nous voudrions nous passer d'un tableau intermédiaire pour stocker les données de la géométrie. Ceci peut accélérer le rendu en évitant une copie de données inutile. Le vertex mapping utilise la fonction *glMapBuffer* pour accéder via un pointeur à la mémoire réservée par le VBO.

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, NULL, GL_STREAM_DRAW);
GLvoid* PositionBuffer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
memcpy(PositionBuffer, PositionData, PositionSize);
glUnmapBuffer(GL_ARRAY_BUFFER);
glVertexPointer(2, GL_FLOAT, 0, 0);
```

Une fois de plus, la fonction *glBufferData* n'est utilisée que pour réserver l'espace mémoire, l'initialisation est effectuée à la main par l'utilisateur au moyen de la fonction *glMapBuffer*. Il existe trois types d'accès aux données du VBO : *GL_WRITE_ONLY*, *GL_READ_ONLY* et *GL_READ_WRITE*. Les noms sont particulièrement explicites. Les modes autorisant la lecture sont également très utiles car ils évitent la duplication de données pour des utilisations non graphique. La fonction *glUnmapBuffer* invalide le pointeur. Il est préférable d'appeler *glUnmapBuffer* le plus tôt possible car le vertex mapping implique des synchronisations du CPU et du GPU.

Lorsque plusieurs VBOs sont utilisés une bonne optimisation consiste à les initialiser en parallèle car ce procédé diminue le nombre de synchronisation CPU/GPU. Voici une solution :

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[COLOR_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, ColorSize, NULL, GL_STREAM_DRAW);
GLvoid* ColorBuffer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, NULL, GL_STREAM_DRAW);
GLvoid* PositionBuffer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

memcpy(ColorBuffer, ColorData, ColorSize);
memcpy(PositionBuffer, PositionData, PositionSize);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[COLOR_OBJECT]);
glUnmapBuffer(GL_ARRAY_BUFFER);
glColorPointer(3, GL_UNSIGNED_BYTE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
```

```
glUnmapBuffer(GL_ARRAY_BUFFER);
glVertexPointer(2, GL_FLOAT, 0, 0);
```

Ceci n'est valable cas titre d'exemple, car l'idée même du Vertex Mapping est d'éviter une copie de données ... ce que *memcpy* fait ici.

3. Références

3.1. Usage des buffers

| | STREAM ¹ | DYNAMIC ¹ | STATIC ¹ |
|-------------------|---------------------|----------------------|---------------------|
| DRAW ¹ | GL_STREAM_DRAW | GL_DYNAMIC_DRAW | GL_STATIC_DRAW |
| READ ² | GL_STREAM_READ | GL_DYNAMIC_READ | GL_STATIC_READ |
| COPY ² | GL_STREAM_COPY | GL_DYNAMIC_COPY | GL_STATIC_COPY |

¹ Disponible avec toutes les cartes OpenGL 1.5 et celles supportant l'extension GL_ARB_vertex_buffer_object

² Non disponible à la date d'écriture de cet article, devrait l'être avec les cartes G80 de nVidia et R600 de ATI.

3.2. Les fonctions de rendu

3.2.1. glVertexElement (Obsolète)

Cette fonction est obsolète car elle s'utilise uniquement en mode immédiat.

```
GLvoid glVertexElement(GLint i);
```

Affiche le $i^{\text{ème}}$ sommet d'un tableau et s'utilise en mode immédiat de la manière suivante:

```
glColorPointer(3, GL_FLOAT, 0, Colors);
glVertexPointer(3, GL_FLOAT, 0, Positions);

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glBegin(GL_TRIANGLES);
    glVertexElement(0);
    glVertexElement(1);
    glVertexElement(2);
glEnd();

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

Ce qui correspond à la séquence suivante :

```
glBegin(GL_TRIANGLES);
    glColor3fv(Colors + 0 * 3);
    glVertex3fv(Positions + 0 * 3);
    glColor3fv(Colors + 1 * 3);
    glVertex3fv(Positions + 1 * 3);
    glColor3fv(Colors + 2 * 3);
    glVertex3fv(Positions + 2 * 3);
glEnd();
```

3.2.2. glDrawElements

Affiche une primitive géométrique en fonction d'un tableau d'index composé de *count* indexes.

```
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, GLvoid* indices)
{
    glBegin(mode);
```

```
for(GLint i = 0; i < count; ++i)
    glArrayElement(indices[i]);
glEnd();
}
```

3.2.3. glDrawRangeElements

Affiche une primitive géométrique en fonction d'un tableau d'indexes en retenant seulement les indexes compris entre *start* et *end* inclus.

```
GLvoid glDrawRangeElements(GLenum mode, GLuint start, GLuint end,
    GLsizei count, GLenum type, GLvoid* indices)
{
    glBegin(mode);
    for(GLint i = 0; i < count; ++i)
        if(indices[i] >= start && indices[i] <= end)
            glArrayElement(indices[i]);
    glEnd();
}
```

3.2.4. glDrawArrays

Affiche une primitive géométrique composée de *count* sommets commençant par l'élément *first* inclus.

```
GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count)
{
    glBegin(mode);
    for(GLint i = 0; i < count; ++i)
        glArrayElement(first + i);
    glEnd();
}
```

3.2.5. glMultiDrawArrays

Affiche plusieurs primitives géométriques composées de *count* sommets commençant par les éléments *first* inclus.

```
GLvoid glMultiDrawArrays(GLenum mode, GLint* first, GLsizei* count,
    GLsizei primcount)
{
    for(GLint i = 0; i < primcount; ++i)
    {
        if(count[i] > 0)
            glDrawArrays(mode, first[i], count[i]);
    }
}
```

3.2.6. glMultiDrawElements

Affiche plusieurs primitives géométriques en fonction de tableaux d'indexes.

```
GLvoid glMultiDrawElements(GLenum mode, GLsizei* count, GLenum type,
    GLvoid** indices, GLsizei primcount)
{
    for(GLint i = 0; i < primcount; ++i)
    {
        if(count[i] > 0)
            glDrawElements(mode, count[i], type, indices[i]);
    }
}
```

3.3. Les types de tableaux

3.3.1. Types de tableaux du pipeline fixe

Spécifier le tableau du pipeline fixe à activer ou désactiver :

```
GLvoid glEnableClientState(GLenum array);
GLvoid glDisableClientState(GLenum array);
```

Liste des tableaux prédéfinis disponibles:

- *GL_VERTEX_ARRAY*
- *GL_NORMAL_ARRAY*
- *GL_COLOR_ARRAY*
- *GL_SECONDARY_COLOR_ARRAY*
- *GL_INDEX_ARRAY*
- *GL_EDGE_FLAG_ARRAY*
- *GL_FOG_COORD_ARRAY*
- *GL_TEXTURE_COORD_ARRAY*

3.3.2. Types de tableaux du pipeline programmable

Spécifier le tableau du pipeline programmable à activer ou désactiver :

```
GLvoid glEnableVertexAttribArray(GLuint index);
GLvoid glDisableVertexAttribArray(GLuint index);
```

index est un identifiant d'une variable d'attribut.

3.4. Les données des tableaux

3.4.1. Description de données du pipeline fixe

Depuis OpenGL 1.1 et *GL_EXT_vertex_array* et *GL_ARB_vertex_buffer_object* :

```
GLvoid glVertexPointer(GLint size, GLenum type, GLsizei stride, GLvoid* pointer);
GLvoid glNormalPointer(GLenum type, GLsizei stride, GLvoid* pointer);
GLvoid glColorPointer(GLint size, GLenum type, GLsizei stride, GLvoid* pointer);
GLvoid glEdgeFlagPointer(GLsizei stride, GLvoid* pointer);
GLvoid glTexCoordPointer(GLint size, GLenum type, GLsizei stride, GLvoid* pointer);
```

La fonction *glIndexPointer* n'est utile que pour les vertex arrays.

```
GLvoid glIndexPointer(GLenum type, GLsizei stride, GLvoid* pointer);
```

Depuis OpenGL 1.4 et *GL_EXT_secondary_color* :

```
GLvoid glSecondaryColorPointer(GLint size, GLenum type, GLsizei stride,
GLvoid* pointer);
```

Depuis OpenGL 1.4 et *GL_EXT_fog_coord* :

```
GLvoid glFogCoordPointer(GLenum type, GLsizei stride, GLvoid* pointer);
```

3.4.2. Description de toutes les données (Obsolète)

La fonction *glInterleavedArrays* implique l'utilisation de tableaux à structure prédéfinis ce qui est à la fois lourd et inadapté aux rendus utilisant le multitexturing ou des shaders. Elle est obsolète depuis OpenGL 1.3 et *GL_ARB_multitexture*. De plus, la solution des VBOs pour gérer les tableaux entrelacés est bien meilleure car elle corrige ces deux problèmes.

```
GLvoid glInterleavedArrays(GLenum format, GLsizei stride, GLvoid* pointer);
```

3.4.3. Description de donnée du pipeline programmable

Tous les attributs définis par le pipeline fixe sont accessibles par les vertex shaders avec OpenGL. Si l'utilisateur souhaite envoyer des données personnalisées, il peut créer une nouvelle variable d'attributs et les données seront envoyées aux vertex shaders via la fonction *glVertexAttribPointer*.

```
GLvoid glVertexAttribPointer(GLuint index, GLint size, GLenum type,  
                             GLboolean normalized, GLsizei stride, const GLvoid* pointer);
```

OpenGL ES ne supporte pas les attributs du pipeline fixe. En conséquence, il faudra utiliser des attributs personnalisés et *glVertexAttribPointer* pour tous les attributs. Cette approche peut également être utilisée avec OpenGL et semble se généraliser dans l'avenir particulièrement avec OpenGL LM. Cette méthode améliore la flexibilité et la robustesse du code pour une approche tout shader. Les drivers de nVidia sont pres mais pas ceux de ATI pour le moment. La classe *CTest5* parmi les exemples présentent cet aspect.

3.5. Types de primitive

GL_POINTS : Points.

GL_LINES : Segments dissociés les uns des autres.

GL_LINE_STRIP : Ligne formée de segments.

GL_LINE_LOOP : Ligne formée de segments qui boucle.

GL_TRIANGLES : Triangles dissociés les uns des autres.

GL_TRIANGLE_STRIP : Succession de triangles juxtaposés formant une bande.

GL_TRIANGLE_FAN : Eventail créé en utilisant le premier sommet comme référence.

GL_QUADS : Quadrilatères dissociés les uns des autres.

GL_QUAD_STRIP : Succession de quadrilatères juxtaposés formant une bande.

GL_POLYGON : Polygone convexe.