

La géométrie en OpenGL : vers les VertexBufferObject

par Cyril Doillon ([Page Perso](#))

Date de publication : 05/08/2008

Dernière mise à jour : 15/08/2008

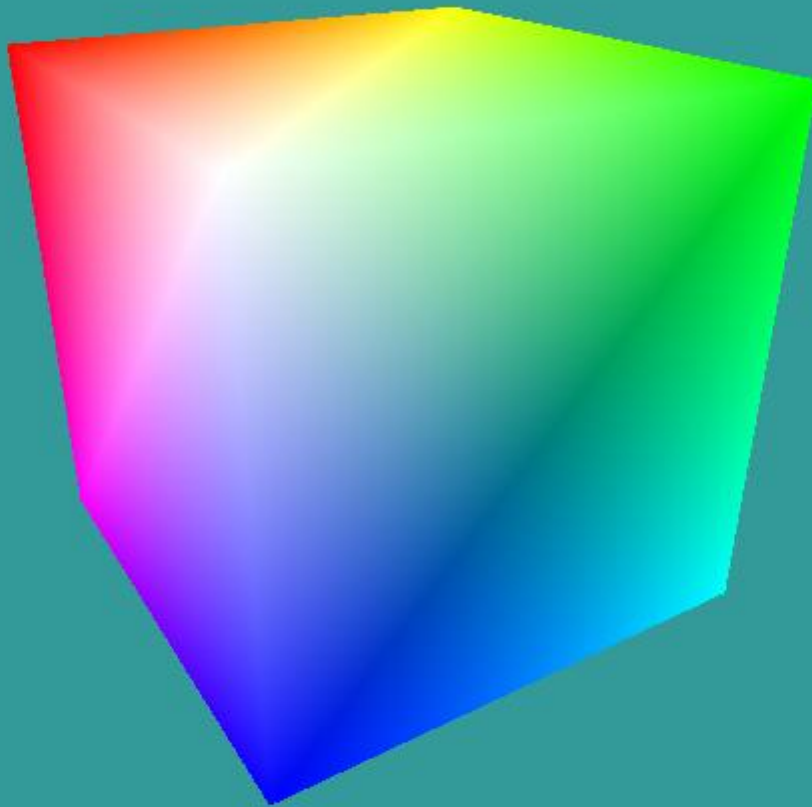
Pourquoi faire un tutoriel sur la géométrie en OpenGL ? Pourquoi faut-il s'y intéresser ? La réponse est simple, les technologies 3D sont sans cesse en cours d'évolution et d'améliorations. OpenGL ne fait pas exception, il y a régulièrement de nouvelles extensions pour améliorer ses performances et son efficacité. L'une d'elle se concentre sur l'amélioration de la géométrie : les Vertex Buffer Object. Comment en est on arrivé là ?

Introduction.....	3
II - Les prémices d'OpenGL : la géométrie immédiate.....	4
II-A - Rappel sur le dessin immédiat.....	4
II-B - Utilisation dans notre exemple.....	5
II-C - Avantages / Inconvénients de cette technique.....	6
III - Quelques avancées : les "vertex array" (VA).....	6
III-A - Utilisation des Vertex Array.....	6
III-B - Complément par les "Display List".....	8
III-C - Pour notre exemple.....	9
III-D - Avantages / Inconvénients de cette technique.....	10
IV - Nouvelle technologie : les Vertex Buffer Object (VBO).....	11
IV-A - Par où commencer ?.....	11
IV-B - Différents cas d'utilisations.....	12
IV-C - Utiliser un buffer.....	12
IV-D - Pour notre exemple.....	13
Conclusion.....	14
Remerciements.....	14

Introduction

Vous trouvez que votre application est lente ? Peut-être avez vous choisi un mauvais système pour rendre tous vos modèles 3D dans votre application ou votre jeu. Ce tutoriel a pour objectif de vous présenter toutes les techniques pour rendre des objets en OpenGL avec leurs avantages et leurs inconvénients. Nous commencerons par une présentation de la première méthode utilisée par OpenGL pour rendre des objets, le mode immédiat. Nous verrons en quoi ce système n'est pas forcément adapté au rendu et pourquoi les "VertexArray" ont fait leur apparition. Enfin, nous verrons en détail l'utilisation de la dernière, la plus récente et nouvelle technologie que sont les "Vertex Buffer Object".

Au travers de ce tutoriel, nous verrons le fonctionnement général de chacune des technologies au travers de l'exemple simple qu'est le dessin d'un cube coloré. Nous ferons une petite étude du coût de chaque méthode pour pouvoir les comparer et savoir dans quels cas utiliser l'une plutôt que l'autre.



Rendu de notre cube

II - Les prémices d'OpenGL : la géométrie immédiate

II-A - Rappel sur le dessin immédiat

La première méthode utilise, pour dessiner des formes 3D dans notre application, les fonctions "**immédiates**". Ce type de dessin est le premier que l'on apprend lorsque que l'on découvre l'API OpenGL. Cette méthode est dite "*immédiate*" car elle consiste en une succession d'appels de fonctions qui dessineront directement à l'écran. Pour décrire la géométrie, on utilise un ensemble de fonctions qui décrira toutes les composantes de notre objet 3D dessiné.

Chaque description de forme est encadrée par les fonctions **glBegin()** et **glEnd()** qui permettent de délimiter notre dessin. La première prend comme paramètre le type de forme à dessiner. Voici une description de ces fonctions :

Prototype de la fonction	Description
void glBegin (GLenum mode);	Marque le début du dessin d'un groupe de primitives. <i>mode</i> précise le type des primitives parmi GL_POINTS , GL_LINES , GL_LINE_STRIP , GL_LINE_LOOP , GL_TRIANGLES , GL_TRIANGLE_STRIP , GL_TRIANGLE_FAN , GL_QUADS , GL_QUAD_STRIP et GL_POLYGON .
void glEnd (void);	Marque la fin du dessin d'un groupe de primitives.

Chaque primitive est découpée en *vertex* qui correspondent à un sommet de notre forme 3D. Nous pouvons définir plusieurs types d'informations par vertex comme sa position, sa couleur, ... Chaque information est définie par un appel de fonction correspondant. Par exemple, nous utiliserons **glVertex*()** pour définir la position du vertex et **glColor*()** pour en définir la couleur. Ces fonctions sont décrites ici :

Prototype de la fonction	Description
void glVertex* (TYPE paramètres);	Spécifie la position du vertex à dessiner
void glNormal* (TYPE paramètres);	Spécifie la normale du vertex
void glColor* (TYPE paramètres);	Spécifie la couleur du vertex
void glSecondaryColor* (TYPE paramètres);	Spécifie la seconde couleur du vertex
void glIndex* (TYPE paramètres);	Spécifie l'index de la couleur du vertex
void glTexCoord* (TYPE paramètres);	Spécifie les coordonnées de la texture principale du vertex
void glMultiTexCoord* (TYPE paramètres);	Spécifie les coordonnées de texture en cas de "multi-texturing" du vertex
void glFogCoord* (TYPE paramètres);	Spécifie les coordonnées de fog du vertex



* doit être remplacé par la description du **TYPE** de données passé en paramètre.

II-B - Utilisation dans notre exemple

Pour l'exemple du cube, que nous allons suivre tout au long du tutoriel, nous allons créer un cube dont chacun des vertex a une couleur différente. Nous allons constituer notre cube de 6 faces. Chacune des faces est constituée de 2 triangles dont chaque vertex à une couleur différente. Nous obtenons donc un cube de 12 triangles.

Dans le cas que nous étudions actuellement, nous allons choisir le mode de primitives `GL_TRIANGLES` pour rendre notre objet. Ainsi, pour un cube, nous devons appeler 36 fois la fonction `glVertex*()` (12 triangles * 3 vertex). Il faut multiplier ce nombre par la quantité d'information par vertex pour obtenir le nombre total d'appels de fonction. Dans un cas extrême qui paramètrerait une couleur par vertex, la normale au vertex, des coordonnées de fog et les coordonnées pour deux textures, nous obtiendrions 180 appels de fonctions pour un simple cube avec 12 triangles. Pour notre exemple, nous nous contenterons d'une seule propriété de couleur par vertex donc nous obtiendrons 72 appels de fonctions plus les 2 appels pour `glBegin` et `glEnd`. Enfin, trois valeurs par appel de fonction nous donnent un total de 216 valeurs envoyées au processeur graphique.

Rendu d'un cube en mode immédiat

```
glBegin( GL_TRIANGLES );

// gauche (x=-1)
glColor3f( 1.0f, 0.0f, 0.0f ); glVertex3f( -1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( -1.0f, 1.0f, 1.0f );
glColor3f( 1.0f, 1.0f, 0.0f ); glVertex3f( -1.0f, 1.0f, 0.0f );
glColor3f( 1.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, 1.0f );
glColor3f( 0.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, 0.0f );

// droite (x=1)
glColor3f( 0.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, 1.0f );
glColor3f( 0.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, 1.0f );
glColor3f( 1.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, -1.0f );
glColor3f( 0.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, 1.0f );
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, -1.0f );

// bas (y=-1)
glColor3f( 0.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, 1.0f );
glColor3f( 1.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, -1.0f );
glColor3f( 0.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, 1.0f );
glColor3f( 0.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, 1.0f );
glColor3f( 1.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, -1.0f );

// haut (y=1)
glColor3f( 1.0f, 0.0f, 0.0f ); glVertex3f( -1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( -1.0f, 1.0f, 1.0f );
glColor3f( 1.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( -1.0f, 1.0f, 1.0f );
glColor3f( 0.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, 1.0f );

// arriere (z=-1)
glColor3f( 1.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, -1.0f );
glColor3f( 1.0f, 0.0f, 0.0f ); glVertex3f( -1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 0.0f, 0.0f ); glVertex3f( -1.0f, 1.0f, -1.0f );
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, -1.0f );
glColor3f( 1.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, -1.0f );

// avant (z=1)
glColor3f( 1.0f, 1.0f, 1.0f ); glVertex3f( -1.0f, 1.0f, 1.0f );
glColor3f( 0.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, 1.0f );
glColor3f( 0.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, 1.0f );
glColor3f( 0.0f, 1.0f, 0.0f ); glVertex3f( 1.0f, 1.0f, 1.0f );
glColor3f( 0.0f, 0.0f, 1.0f ); glVertex3f( -1.0f, -1.0f, 1.0f );
glColor3f( 0.0f, 1.0f, 1.0f ); glVertex3f( 1.0f, -1.0f, 1.0f );
```

Rendu d'un cube en mode immédiat

```
glEnd();
```

II-C - Avantages / Inconvénients de cette technique

Notre premier exemple permet de mettre en évidence certains avantages et certains inconvénients de la technique de dessin "*Immédiat*". Nous allons donc commencer par évoquer les avantages d'une telle technique :

- Permet de définir indépendamment chacune des informations des vertex
- Permet d'augmenter la lisibilité du code
- Regroupe clairement chaque rendu par type de primitives

Malgré ceux-ci, on remarque rapidement un grand nombre d'inconvénients très gênants pour des applications "temps-réel". En voici une liste non-exhaustive :

- Un grand nombre d'appels de fonction pour des géométries simples
- Grande répétitivité de commandes identiques
- Complexifie les structures de données utilisables
- Obligation de séparer chacune des composantes de nos informations (X, Y et Z pour les positions de nos vertex, par exemple)

La technique de dessin "*Immédiat*" est très contraignante du point de vue performance pour une application "temps-réel". Les deux problèmes principaux sont que le transfert des données est extrêmement lent au vu de la quantité d'appel de fonctions et du peu d'informations envoyées au GPU par chaque appel. Le second problème vient du nombre d'appels identiques à chaque frame. OpenGL nous propose une nouvelle technique permettant de résoudre notamment le nombre d'appels de fonctions, de limiter l'envoi d'information en double. Cette technique utilise le principe des "*vertex array*". De plus, cette technique pourra être complétée par la méthode des "Display List" qui évitera un grand nombre d'appels de fonction répétitifs à chaque frame dans le cas de géométrie "statique".

III - Quelques avancées : les "vertex array" (VA)

III-A - Utilisation des Vertex Array

La seconde technique utilisable en OpenGL est donc celle des "Vertex Array". Le principe de cette technique est de 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 "entrelacé" contenant toutes les informations. L'utilisation des Vertex Array se fait en trois étapes principales. La première consiste en l'activation / désactivation des types de tableaux à utiliser. Ensuite, il faut remplir les tableaux activés avec les informations de notre géométrie. Enfin, on peut lancer l'appel du rendu de notre géométrie.

La première chose à faire pour utiliser les VertexArray est d'en activer l'utilisation. Il faut également les désactiver quand on n'en a plus besoin. Pour cela, on utilise deux fonctions :

Prototype de la fonction	Description
void glEnableClientState (GLenum type);	Spécifie le <i>type</i> de tableau à activer
void glDisableClientState (GLenum type);	Spécifie le <i>type</i> de tableau à désactiver

Le type de tableau est à choisir parmi `GL_COLOR_ARRAY`, `GL_EDGE_FLAG_ARRAY`, `GL_FOG_COORD_ARRAY`, `GL_INDEX_ARRAY`, `GL_NORMAL_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_TEXTURE_COORD_ARRAY` et `GL_VERTEX_ARRAY`.


La seconde étape consiste à indiquer à OpenGL où se trouvent les données à utiliser pour dessiner notre géométrie. On va donc créer un tableau local qui sera directement utilisé par OpenGL. Il existe ainsi une fonction par type d'information pour exécuter cette étape.


Prototype de la fonction	Description
void glVertexPointer (GLint taille, GLenum type, GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau de coordonnées de vertex. Les données sont au nombre de <i>taille</i> par vertex, de type <i>type</i> et espacées de <i>stride</i> octets entre deux vertex.
void glColorPointer (GLint taille, GLenum type, GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau de couleurs principales par vertex.
void glSecondaryColorPointer (GLint taille, GLenum type, GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau de couleurs secondaires par vertex.
void glIndexPointer (GLenum type, GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau de couleurs indexées par vertex.
void glNormalPointer (GLenum type, GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau des normales par vertex.
void glFogCoordPointer (GLenum type, GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau des coordonnées de brouillard par vertex.
void glTexCoordPointer (GLint taille, GLenum type, GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau des coordonnées de texture par vertex. Dans le cas de <i>Multitexturage</i> , les coordonnées correspondent à la texture active
void glEdgeFlagPointer (GLsizei stride, const GLvoid * pointeur);	Spécifie l'emplacement du tableau d'indicateur de contours par vertex.

On peut remarquer que certaines fonctions ne prennent pas de paramètre *taille* car certaines informations ont un nombre de composantes fixes. Le paramètre *stride* peut être différent de zéro, par exemple, dans le cas d'un tableau unique pour les couleurs primaires et secondaires.

Une fois que nous avons indiqué où se situaient toutes les informations de notre géométrie, il existe donc plusieurs façons de décrire notre géométrie grâce à différentes fonctions.

Prototype de la fonction	Description
void glArrayElement (GLint index);	Doit être placé entre glBegin et glEnd . Récupère les informations du vertex d'index <i>index</i> dans tous les tableaux activés
void glDrawElements (GLenum mode, GLsizei taille, GLenum type, void * indices);	Crée automatiquement une géométrie avec le type de primitive <i>mode</i> , composée des vertex des <i>taille</i> index contenu dans le tableau <i>indices</i> de type <i>type</i>
void glMultiDrawElements (GLenum mode, GLsizei taille, GLenum type, void ** indices, GLsizei taille_primitive);	Correspond à l'appel de <i>taille_primitive</i> fois la fonction glDrawElements où <i>indice</i> est un tableau à 2 dimensions. Chaque sous-tableau est passé en paramètre à chaque appel.
void glDrawArrays (GLenum mode, GLint premier, GLsizei taille);	Utilise l'ensemble des vertex des différents tableaux pour dessiner la géométrie utilisant le type de primitive <i>mode</i> et les <i>taille</i> indices en commençant par le <i>premier</i>
void glMultiDrawArrays (GLenum mode, GLint * premier, GLsizei * taille, GLsizei taille_primitive);	Correspond à l'appel de <i>taille_primitive</i> fois la fonction glDrawArrays où <i>premier</i> et <i>taille</i> contiennent les paramètres de chaque appel.

 *glArrayElement* correspond au mode "Immédiat" donc donne des performances identiques.

 On peut remarquer deux types de fonctions : **gl*Elements** qui récupèrent chaque vertex indépendamment dans les tableaux d'informations, **gl*Arrays** qui utilisent l'ensemble des vertex des tableaux. Pour utiliser un tableau unique **entrelacé**, on utilise la fonction *glInterleavedArrays*.

III-B - Complément par les "Display List"

Nous avons vu que les commandes OpenGL pouvaient être nombreuses et surtout répétitives entre deux frames. OpenGL offre la possibilité de pallier à ce problème par l'enregistrement de commandes successives précompilées. Cette technologie, appelée "Display List", est tout à fait utilisable pour le dessin immédiat ainsi que pour les Vertex Array. Le principe d'utilisation des listes est d'abord de créer un identifiant par liste, de créer la liste et de l'appeler à chaque fois que l'on en a besoin.

Prototype de la fonction	Description
GLuint glGenLists (GLsizei nombre);	Crée <i>nombre</i> identifiants séquentiels de listes d'affichages disponibles.
void glNewList (GLuint index, GLenum mode);	Spécifie le début de la création de la liste d'identifiant <i>index</i> où <i>mode</i> est défini parmi <i>GL_COMPILE</i> , qui compile seulement la liste et <i>GL_COMPILE_AND_EXECUTE</i> , qui compile et exécute la liste.
void glEndList (void);	Spécifie la fin de la liste courante.
void glCallList (GLuint index);	Appelle les commandes de la liste <i>index</i> précompilée.
void glCallLists (GLsizei nombre, GLenum type, const GLvoid * liste);	Appelle les commandes successives des lites dont les <i>nombre</i> identifiants sont de type <i>type</i> dans le tableau <i>liste</i> .
void glDeleteLists (GLuint premiere, GLsizei nombre);	Supprime les <i>nombre</i> commandes successives en commençant par l'identifiant <i>premiere</i> .

III-C - Pour notre exemple

L'exemple du cube est très simple pour appliquer tout ce que nous avons vu précédemment. Nous allons donc définir nos 8 vertex dans des tableaux de vertex et de couleurs. Nous allons ensuite définir un tableau des index successifs des vertex à utiliser pour des primitives de type *GL_TRIANGLES*. Nous choisissons ici le cas simple sans "Display List" et avec la fonction `glDrawElements` pour optimiser la quantité d'information envoyée à OpenGL.

Nous remarquons tout de suite que le nombre d'appels de fonction a bien été réduit puisqu'il passe de 74 pour le mode immédiat à 7 pour les vertex array. De plus, la quantité d'information a diminué car elle est passée de 216 pour le mode immédiat à 84 (8 vertex * 3 composantes * 2 informations + 36 indices) valeurs pour les vertex array. Par contre, reste le problème que toutes les informations sont envoyées à chaque frame de notre application alors que ce ne serait pas nécessaire pour une géométrie statique (sauf par l'utilisation des Display List).

Rendu d'un cube avec VertexArray

```

GLfloat VertexArray[24] = {
    -1.0f, 1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
    -1.0f, 1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    1.0f, -1.0f, 1.0f,
    1.0f, 1.0f, -1.0f,
    1.0f, -1.0f, -1.0f
};

GLfloat ColorArray[24] = {
    1.0f, 0.0f, 0.0f,
    1.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
    0.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 0.0f,
    1.0f, 1.0f, 1.0f
};

GLuint IndiceArray[36] = {
    0,1,2,2,1,3,
    4,5,6,6,5,7,
    3,1,5,5,1,7,
    0,2,6,6,2,4,
    6,7,0,0,7,1,
    2,3,4,4,3,5
    
```

Rendu d'un cube avec VertexArray

```
};

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

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

glDrawElements( GL_TRIANGLES, 36, GL_UNSIGNED_INT, IndiceArray );

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

III-D - Avantages / Inconvénients de cette technique

Nous avons pu voir que les techniques des "Vertex Array" et des "Display List" pouvaient grandement améliorer le rendu de géométrie avec OpenGL. Pour les "Vertex Array", nous avons vu qu'il en existait deux types : sans ou avec **partage de vertex** (*glDrawArrays* vs *glDrawElements*). La version sans partage de vertex (*glDrawArrays*) a été **déconseillée par Nvidia** ainsi que par **ATI**. Il est préférable d'utiliser la version *glDrawElements* qui permet de créer un cache de vertex uniques et, grâce à un tableau d'indices, de partager le vertex pour une même géométrie. Nous pouvons en déduire un certain nombre d'avantages :

- Réduction du nombre d'appels de fonction
- Réduction de la quantité d'information envoyée à la carte graphique
- Précompilation d'une série de commandes OpenGL

A cela, nous pouvons ajouter quelques optimisations possibles qu'offrent les "Vertex Array" :

- Utilisation du type `GL_UNSIGNED_SHORT` (16bit) à la place du type `GL_UNSIGNED_INT` (32bit) (optimisation mémoire et temps de transfert)
- Utilisation de `GL_TRIANGLE_STRIP` (1/3 d'indices en moins en moyenne)
- Dans le cas d'information couleur, il est conseillé d'utiliser `GL_UNSIGNED_BYTE` ou `GL_FLOAT` (optimisation de conversion des données)
- Dans le cas d'information de normale, il est conseillé d'utiliser `GL_FLOAT`, mais déconseillé d'utiliser `GL_BYTE`.

Malgré la résolution de certains problèmes par rapport au dessin "immédiat", on conserve certains inconvénients avec cette technique. Voici une liste des problèmes les plus remarquables :

- Espace mémoire occupé par les "Display List" compilées
- Temps de compilation des "Display List"
- Applicable uniquement pour une géométrie statique

Malgré tous les gains apportés par ces nouvelles technologies, il reste quelques problèmes assez importants. Tout d'abord, les "Display List" permettent de précompiler des commandes OpenGL mais cette précompilation a un surcoût mémoire ainsi qu'un surcoût en temps. Le surcoût mémoire provient de l'enregistrement des commandes elles mêmes, l'idéal serait de stocker uniquement les informations de notre géométrie. De plus, ces deux techniques associées obligent à dessiner uniquement une géométrie statique non-modifiable. Si on utilise uniquement les "VertexArray" sans les "DisplayList", le problème de l'envoi répétitif des informations (comme dans notre exemple) réapparaît. OpenGL propose donc une solution à tout cela et cette solution s'appelle les "Vertex Buffer Object".

IV - Nouvelle technologie : les Vertex Buffer Object (VBO)


IV-A - Par où commencer ?

Nous avons donc vu que les "VertexArray", accompagnés des "DisplayList", étaient une grande avancée dans le rendu de géométrie. Mais il persiste quelques problèmes assez importants notamment dans les surcoûts mémoire et temps de ces technologies. Pour cela, après l'apparition des extensions *NV_vertex_array_range* et *ATI_vertex_array_object* par, respectivement, Nvidia et ATI et l'approbation par l'ARB (*Architecture Review Board*) de l'extension *ARB_vertex_buffer_object*, a été créée la technologie des **VertexBufferObject** (VBO). C'est ainsi que 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. Grâce à un procédé proche des "VertexArray", les VBO permettent d'envoyer et de stocker toutes les informations de notre géométrie dans notre carte graphique. Ce principe, utilisé pour les textures, est maintenant utilisé pour la géométrie.

Spécification de l'extension `GL_ARB_vertex_buffer_object`

Pour utiliser ces buffers, il faut d'abord commencer par demander à OpenGL un ou plusieurs identifiants pour ces buffers. Cet identifiant sera utilisé pour signaler à OpenGL que nous voulons utiliser le buffer de la géométrie correspondante. Nous pouvons, bien évidemment, utiliser et enregistrer plusieurs buffers pour notre application, tant qu'il y a suffisamment de mémoire dans la carte graphique.

Prototype de la fonction	Description
void glGenBuffers (GLsizei nombre, GLuint * buffers);	Crée <i>nombre</i> identifiants de buffer et les place dans le bloc mémoire commençant par <i>buffers</i> (élément simple ou tableau)
void glBindBuffer (GLenum type, GLuint buffer);	Spécifie que l'on utilise le buffer d'identifiant <i>buffer</i> comme buffer courant et qu'il est de type <i>type</i> . <i>type</i> est à choisir parmi GL_ARRAY_BUFFER et GL_ELEMENT_ARRAY_BUFFER .
void glDeleteBuffers (GLsizei nombre, const GLuint * buffers);	Supprime les <i>nombre</i> buffers de la liste <i>buffers</i> .

 **GL_ARRAY_BUFFER** correspond aux tableaux des informations sur les vertex et **GL_ELEMENT_ARRAY_BUFFER** correspond aux tableaux des indices de vertex.

L'étape suivante est de remplir notre buffer avec les informations sur nos vertex. Nous allons donc envoyer des tableaux de données à notre carte graphique qui va remplir sa mémoire comme elle le ferait pour une texture. C'est dans cette partie que nous allons définir quel type d'utilisation nous allons faire de notre buffer par l'intermédiaire du paramètre *utilisation*.

Prototype de la fonction	Description
void glBufferData (GLenum type, GLsizeiptr taille, const GLvoid * donnees, GLenum utilisation);	Crée et initialise le buffer actif de type <i>type</i> avec les <i>taille</i> informations contenues dans le tableau <i>donnees</i> . L'utilisation de ce buffer sera définie par <i>utilisation</i>
void glBufferSubData (GLenum type, GLintptr decalage, GLsizeiptr taille, const GLvoid * donnees);	Met à jour le buffer de type <i>type</i> à partir de la zone commençant par <i>decalage</i> avec les <i>taille</i> informations contenues dans <i>donnees</i> .

IV-B - Différents cas d'utilisations

Les VBO supporte différents types d'utilisations qui peuvent être différents pour chaque buffer. Les cas d'utilisations se définissent par deux critères qui sont la fréquence avec laquelle nous allons **mettre à jour les informations** et la façon dont nous allons **accéder aux données**. 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

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

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

Ce paramètre est très important car OpenGL va se charger seul d'organiser sa mémoire en fonction du type d'utilisation. Les utilisations du type `GL_*_DRAW` sont les plus utilisées pour dessiner une géométrie car les informations proviennent de l'application et non d'OpenGL. Ensuite, les utilisations de type `GL_STATIC_*` vont plutôt s'appliquer à une géométrie qui ne va jamais être modifiée au cours de l'application comme des bâtiments, des statues, ... Les utilisations `GL_STREAM_*` vont s'appliquer généralement à une géométrie qui est modifiée une seule fois juste avant un rendu. Ceci est très proche du type `GL_DYNAMIC_*` qui est prévu pour de nombreuses modifications de données avant une seule utilisation. Ces deux dernières utilisations peuvent s'appliquer, par exemple, à des applications d'animations de personnages (skinning), à des objets déformables par des simulations physiques, ...

IV-C - Utiliser un buffer

Maintenant que nous avons créé notre buffer, il faudrait pouvoir le modifier et l'utiliser tout au long de notre application. OpenGL offre la possibilité d'accéder directement aux données du buffer actif. Cet accès se fait par les deux fonctions :

Prototype de la fonction	Description
void * glMapBuffer (GLenum type, GLenum acces);	Retourne le pointeur vers les données du buffer courant de type <i>type</i> autorisant un accès de type <i>acces</i> parmi GL_READ_ONLY (lecture seule), GL_WRITE_ONLY (écriture seule), et GL_READ_WRITE (lecture et écriture).
void glUnmapBuffer (GLenum type);	Signifie à OpenGL que l'accès au buffer courant de type <i>type</i> est terminé.

Pour utiliser les données contenues dans le buffer, nous allons utiliser les mêmes fonctions que pour les "Vertex Array". Il faut donc commencer par activer le buffer correspondant à notre géométrie grâce à la fonction `glBindBuffer`. Ainsi les fonctions `glVertexPointer`, `glColorPointer`, `glSecondaryColorPointer`, `glNormalPointer`, `glFogCoordPointer`, `glTexCoordPointer` et `glEdgeFlagPointer` vont-elles être utilisées pour les buffers de type `GL_ARRAY_BUFFER`. La seule différence sera qu'aucun tableau de données ne sera utilisé mais sera remplacé par un décalage mémoire dans notre buffer par rapport à son début. Nous pouvons donc créer un VBO par type d'information et par géométrie mais également un seul VBO pour toutes les informations de notre géométrie. Voici quelques exemples d'utilisation :

Quelques exemples

```
glVertexPointer(3, GL_FLOAT, 3 * sizeof(float), 0);
glColorPointer(3, GL_FLOAT, 3 * sizeof(float), ((void*)NULL + (3)));
```

IV-D - Pour notre exemple

Nous allons donc étudier notre exemple. Nous devons dessiner un cube coloré qui ne changera jamais. Aucune information ne sera modifiée ainsi nous pourrions utiliser un buffer de type `GL_STATIC_DRAW`. Nous allons donc stocker toutes nos informations dans un tableau unique qui sera stocké une seule fois à l'initialisation :

Initialisation de notre VBO

```
GLfloat CubeArray[48] = {
    1.0f, 0.0f, 0.0f, -1.0f, 1.0f, -1.0f,
    1.0f, 0.0f, 1.0f, -1.0f, -1.0f, -1.0f,
    1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
    0.0f, 0.0f, 1.0f, -1.0f, -1.0f, 1.0f,
    0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 1.0f,
    0.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f, 1.0f, -1.0f,
    1.0f, 1.0f, 1.0f, 1.0f, -1.0f, -1.0f
};

GLuint IndiceArray[36] = {
    0,1,2,2,1,3,
    4,5,6,6,5,7,
    3,1,5,5,1,7,
    0,2,6,6,2,4,
    6,7,0,0,7,1,
    2,3,4,4,3,5
};

// Génération des buffers
glGenBuffers( 2, CubeBuffers );

// Buffer d'informations de vertex
glBindBuffer(GL_ARRAY_BUFFER, CubeBuffers[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(CubeArray), CubeArray, GL_STATIC_DRAW);

// Buffer d'indices
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, CubeBuffers[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(IndiceArray), IndiceArray, GL_STATIC_DRAW);
```

Une fois créé, nous pouvons l'utiliser à chaque frame. Nous devons spécifier les buffers à utiliser ainsi que l'emplacement de chaque information.

Rendu du cube avec notre VBO

```
// Utilisation des données des buffers
glBindBuffer(GL_ARRAY_BUFFER, CubeBuffers[0]);
glVertexPointer( 3, GL_FLOAT, 6 * sizeof(float), ((float*)NULL + (3)) );
glColorPointer( 3, GL_FLOAT, 6 * sizeof(float), 0 );

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, CubeBuffers[1]);

// Activation d'utilisation des tableaux
```

Rendu du cube avec notre VBO

```

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

// Rendu de notre géométrie
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);

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

Nous pouvons remarquer que d'un point de vue mémoire, seules les informations de notre géométrie seront stockées contrairement aux "Display List" qui enregistrent également toute la succession des commandes OpenGL. De plus, le temps demandé par la compilation des "Display List" n'est plus nécessaire car on se contente d'envoyer directement notre géométrie. Enfin, grâce au mapping de buffer, on peut uniquement modifier les données des géométries dynamiques et nous ne sommes plus obligés de nous contenter des formes statiques.


Conclusion

Nous avons donc vu les différentes optimisations qu'a apportées OpenGL au rendu de géométrie. Chacune des techniques a ses avantages propres mais les "**Vertex Array**" et les "**Vertex Buffer Object**" ont pour objectif de combler les lacunes des précédentes. A titre d'exemple, sur une carte Nvidia GeForce 7600 GS, les dessins de notre cube nous donnent les performances suivantes :

Immédiat	Vertex Array	Vertex Buffer Object
1445 FPS	1485 FPS	1515 FPS

Cet exemple n'est pas vraiment significatif car il se contente de dessiner 12 triangles mais l'on remarque déjà une légère différence en faveur de VBO. Pour augmenter les performances de votre application OpenGL, vous pouvez trouver quelques informations dans notre FAQ :

[FAQ](#) FAQ Optimisation OpenGL

Une des dernières extensions d'OpenGL ( **GL_EXT_draw_instanced**) permet le dessin de type "instancing". Cette extension permet de dessiner plusieurs fois le même objet avec un seul appel de fonction.

Pour toutes questions, n'hésitez pas à demander directement sur le forum OpenGL

Remerciements

Un grand merci à **Bafman**, **IrmatDen**, **Loka** et surtout **Diogene** pour leurs relectures et leurs conseils.