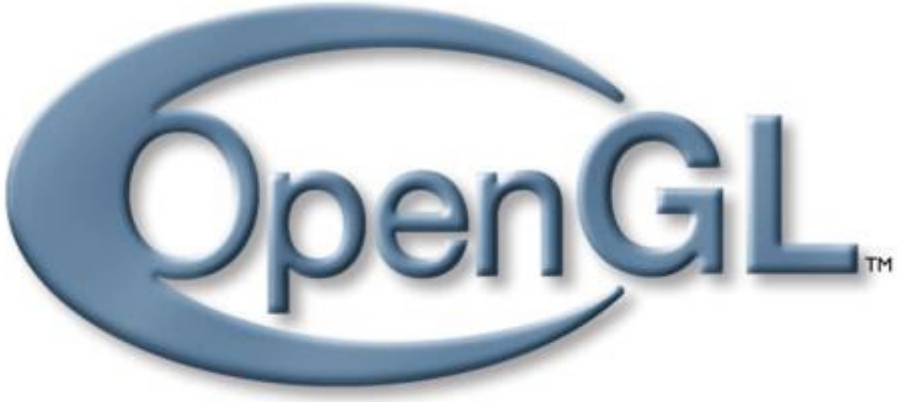
Exposé d'OpenGL



Sujet : Augmenter la rapidité d'affichage en utilisant les extensions d'OpenGL

Professeur : M. Christophe LOHOU
Élève : Boris Brugevin

Sommaire

I - Introduction

Intéressé depuis toujours par les jeux vidéos, j'ai voulu connaître leur fonctionnement. Je me suis donc engagé dans des études de programmation orientées vers l'imagerie numérique : IUT Informatique - option Imagerie Numérique d'Arles. Ces études m'ont passionné, j'ai étudié les mathématiques et différents langages de programmation qui m'ont fortement aidé à manier la 3D. J'ai eut des cours sur différentes API graphiques telles que Direct3D et OpenGL. Mais nous n'avons vu que les bases en cours, j'ai donc beaucoup travaillé tout seul pour en savoir plus sur ces librairies. Pour améliorer mes compétences j'ai voulu continuer mes études dans la même voie. J'ai donc opté pour une licence professionnelle en Imagerie Numérique au Puy-en-Velay. Avec cette licence, je devrais avoir plus de connaissances, surtout sur les Shaders (vertex et fragment programmes) que je ne connais que très peu. Je compte continuer mes études si cela est possible. Je voudrais faire un Master et un DESS toujours dans le même domaine à Reims : STIC – Informatique multimédia.

II - OpenGL

1 Qu'est ce qu'OpenGL

OpenGL est une librairie graphique 3D. Cela signifie qu'on lui donne des ordres de tracé de primitives graphiques directement en 3D, une position de caméra, des lumières, des textures à plaquer sur les surfaces, etc, et qu'à partir de là OpenGL se charge de faire les changements de repère, la projection en perspective à l'écran, le clipping, l'élimination des parties cachées, d'interpoler les couleurs, et de *rasteriser* les faces pour en faire des pixels.

OpenGL s'appuie sur le hardware disponible selon la carte graphique. Toutes les opérations de base sont a priori accessibles sur toute machine, simplement elles iront plus ou moins vite selon qu'elles sont implémentées en hard ou pas. Pour les fonctions évoluées qui ne font pas partie de la norme OpenGL mais figurent parmi les extensions, il se peut par contre qu'elles ne soient disponibles que sur certaines machines.

2 Énonciation du sujet

OpenGL est un générateur d'image 3D qui tourne en temps réel. Plusieurs modes pour tracer des primitives existent. Y'en a t'il un plus rapide ? Et si oui lequel ?

Nous allons tester plusieurs méthodes d'affichage qu'OpenGL propose afin de déterminer laquelle est là plus rapide et permet donc un affichage optimisé.

III - Différentes méthodes d'affichage

Tout d'abord, comment peut-on mesurer la performance d'un programme 3D ? Pour cela on réalise une application qui va tourner à l'infini afin de réaliser une mesure : le fps (frames par seconde). Le fps est le nombre de rafraîchissement de la fenêtre OpenGL en une seconde. On peut coupler à cette méthode un calcule du nombre de primitives afficher en une frame. Grâce à ces deux méthodes, on peut étudier et comparer les différentes techniques d'affichage sous OpenGL.

1 L'affichage basic : « Immediate mode »

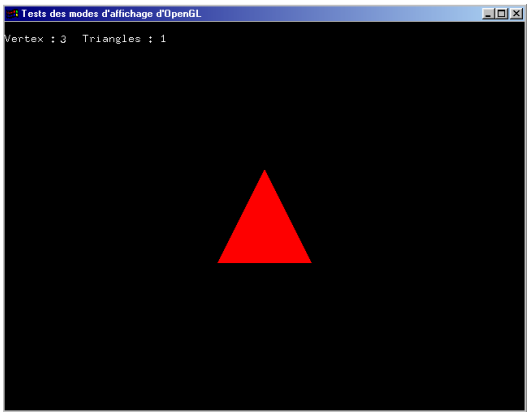
C'est le mode d'affichage le plus simple à utiliser. Il suffit de faire appel aux fonctions glVertex*() pour placer un point, glNormal*() pour lui associer une normale, glColor*() pour donner une couleur au vertex et glTexCoord*() pour lui donner des coordonnées de texture. Ce mode est donc très flexible et permet de rapidement dessiner une primitive en OpenGL.

Les points négatifs sont nombreux. Tout d'abord, ce mode d'affichage produit beaucoup d'appels aux fonctions citées au-dessus, de plus la vitesse d'affichage va dépendre du processeur de la machine, car les données vont transiter par le bus pour passer dans la carte graphique.

Exemple de programme :

```
…
// Affichage
glColor4f(1.0f,0.0f,0.0f,1.0f);  // Attribut une couleur au polygone ( r, g, b, a)
glNormal3f(0.0f,0.0f,1.0f);      // Attribut une normale au polygone
glBegin(GL_TRIANGLES);   // mode de polygon, ici : triangle
  glVertex3f(0.0f,1.0f,0.0f);     glTexCoord2f(0.5f,1.0f) ;   // Coord. du vertex dans l'espace
  glVertex3f(-1.0f,-1.0f,0.0f);  glTexCoord2f(-1.0f,0.0f) ;  // et coord. de texture
  glVertex3f(1.0f,-1.0f,0.0f);   glTexCoord2f(-1.0f,0.0f) ;
glEnd();
…
```

Le code est simple et clair. Et voici le résultat :



2 Compilation de primitives : « Display Lists »

C'est une autre méthode simple et qui permet un affichage rapide. Elles permettent de stocker les vertex et leurs attributs (coordonnées de texture, couleurs, etc…) dans la mémoire vidéo si les drivers le permettent. Par contre une fois compilée, la display liste ne peut être changée. On est donc limité si l'on souhaite interagir sur les vertex de la forme. Une fois crée, la fonction qui génère la display liste envoie un identifiant pour la localiser en mémoire vidéo. Pour l'afficher on n'a plus qu'a appeler cet Id. En fin de programme il ne faut pas oublier de relâcher la mémoire vidéo prise par la liste d'OpenGL.

Exemple de programme :

```
…
// Début du programme
GLuint Liste = glGenLists(1);          // création de la liste GL
glNewList(Liste,GL_COMPILE);
  …                                    // (Affichage de la primitive Ex : en Immediate Mode)
glEndList();
…
// Affichage de la liste :
glCallList(Liste);
…
// Fin du programme
glDeleteLists(Liste, 1);               // suppression de la listeGL
…
```

Le paramètre de glGenList() est en fait le nombre de liste que nous voulons créer. S'il y en a plusieurs, c'est toujour un unsigned int qui devra réceptionner les Ids. Le DeleteList() prend aussi le l'UINT et le nombre de liste à supprimer. Si l'on souhaite créer plusieurs listes avec un seul Id il faudra appeler la fonction glListBase(Liste) après glGenLists().

3 Tableau de vertex : « Vertex Arrays »

Différents tableaux stockent les vertex, les corrdonnées de texture, les normales, les couleurs et les indexes. Les données peuvent être modifié rapidement car tout cela est traité en ram. C'est tableau sont donnés à OpenGL afin qu'il gère la création des vertex buffer et donc éviter l'appel des fonctions basic vu en immediate mode. Ce mode permet aussi d'indexé l'affichage grâce à un tableau d'indexe, ceci évite d'envoyer plusieurs fois les même points à la carte graphique.

Exemple de programme :

```
…
// Début du programme
GLuint *index = NULL;          // Tableau des faces
GLfloat *Normals = NULL;       // Tableau des normales
GLfloat *Color = NULL;         // Tableau des couleurs
GLfloat *Vertices = NULL;      // Tableau des vertex
…
// init et remplissage des tableaux
```

```
…
// affichage
glEnableClientState(GL_COLOR_ARRAY);          // activation du tableau de couleurs
glColorPointer(4, GL_FLOAT, 0, Color);        // on donne le pointeur sur le tableau

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, Vertices);

glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, Normals);

// affichage de la primitive indexé
glDrawElements( GL_TRIANGLES, nb_faces*3, GL_UNSIGNED_INT, index);

glDisableClientState(GL_COLOR_ARRAY);         // désactivation du tableau de couleurs
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
…
// Fin du programme
delete [] index ; index  = NULL ;  // suppression des tableaux
delete [] Normals; Normals = NULL;
delete [] Color; Color  = NULL;
delete [] Vertices; Vertices = NULL;
…
```

Tout d'abord il faut déclarer plusieurs tableaux qui vont contenir les différentes données nécessaires pour tracer la primitive. Ensuite il faut les remplir. Il faut dire à OpenGL quels tableaux doivent être appelé pour dessiner l'objet ainsi que le pointeur sur chacun des tableaux. Enfin, à la fin du programme, on supprime l'espace mémoire utilisé par les tableaux.

4 Tableau de structure : « Interleaved Vertex Arrays »

C'est le même principe que les vertex arrays, sauf qu'on crée un tableau de structure comme en direct3d. Le code est donc plus facile à la lecture tout en gardant les avantages des vertex arrays.

Exemple de programme :

```
…
// Début du programme
struct VERTEX {                 // un vertex
        GLfloat r,g,b,a;        // Couleur
        GLfloat nx,ny,nz;       // Normale
        GLfloat x,y,z;          // Coordonnées
};

VERTEX *vertexes = NULL;        // Tableau des vertex
GLuint *index = NULL ;          // Tableau des index
…
// init et remplissage des tableaux
…
```

```
// affichage
glInterleavedArrays(GL_C4F_N3F_V3F, sizeof(VERTEX), vertexes);
glDrawElements(GL_TRIANGLES, nb_faces*3, GL_UNSIGNED_INT, index);
…
// Fin du programme
delete [] index ; index  = NULL ;  // suppression des tableaux
delete []vertexes; vertexes = NULL;
```

Au début il faut faire une structure d'un vertex. Les données peuvent être diverses mais dans un ordre bien précis car pour l'affichage il faudra dire à OpenGL comment lire la structure.

Voici les types de lecture qu'OpenGl connaît :

```
// gl.h
…
#define GL_V2F                  0x2A20
#define GL_V3F                  0x2A21
#define GL_C4UB_V2F             0x2A22
#define GL_C4UB_V3F             0x2A23
#define GL_C3F_V3F              0x2A24
#define GL_N3F_V3F              0x2A25
#define GL_C4F_N3F_V3F          0x2A26
#define GL_T2F_V3F              0x2A27
#define GL_T4F_V4F              0x2A28
#define GL_T2F_C4UB_V3F         0x2A29
#define GL_T2F_C3F_V3F          0x2A2A
#define GL_T2F_N3F_V3F          0x2A2B
#define GL_T2F_C4F_N3F_V3F      0x2A2C
#define GL_T4F_C4F_N3F_V4F      0x2A2D
…
```

Les 'V' correspondent au mot vertex, les 'C' à la couleur, les 'T' aux coordonnées de texture et les 'N' aux normales. Les 'UB' et 'F' correspondent aux types unsigned byte et float. Par exemple dans le programme : GL_C4F_N3F_V3F correspond à 4 couleurs : r, g, b et a, 3 float pour la normale et 3 float pour le vertex. L'ordre d'écriture de ces constantes doit être respecté dans la structure.

5 Stockage en mémoire vidéo : « GL_ARB_vertex_buffer_object »

C'est le mode d'affichage le plus rapide d'OpenGL. On crée des vertex arrays et on appel une extension d'OpenGL qui va copier les différents tableaux dans la mémoire vidéo (AGP). Les données ne transitent donc qu'une seule fois par le bus à la création. La carte graphique disposant des points dans ça mémoire affiche beaucoup plus vite les primitives. Ces vertex buffer ARB peuvent être dynamiques on peut aller modifier les données directement dans la mémoire graphique ou statiques juste en lecture seule.

Exemple de programme :

```
…
// Début du programme
struct VERTEX {                 // un vertex
```

```
        GLfloat r,g,b,a;        // Couleur
        GLfloat nx,ny,nz;       // Normale
        GLfloat x,y,z;          // Coordonnées
};

VERTEX *vertexes = NULL;        // Tableau des vertex
GLuint *index = NULL ;          // Tableau des index

// ID Index & Vertex Buffer ARB
GLuint IndexBuffer = -1;
GLuint VertexBuffer = -1;

// pointeur sur les fonctions ARB_vertex_buffer_object
PFNGLBINDBUFFERARBPROC  glBindBufferARB = NULL;
PFNGLGENBUFFERSARBPROC glGenBuffersARB = NULL;
PFNGLBUFFERDATAARBPROC glBufferDataARB = NULL;
PFNGLGETBUFFERPARAMETERIVARBPROC glGetBufferParameterivARB = NULL;
PFNGLDELETEBUFFERSARBPROC  glDeleteBuffersARB = NULL;
…
```

Tout d'abord comme vu précédemment, on déclare une structure de vertex. Ensuite, nous devons créer les tableaux nécessaires au stockage de nos points et index. Nous créons aussi des identifiants pour la génération de ces vertex et index buffers en mémoire vidéo. Nous déclarons les pointeurs sur les fonctions nécessaires à la récupération des fonctions étendues d'OpenGL.

Pour tester si les drivers de la carte graphique détient les extensions OpenGL nécessaire à la création de ces buffers, on utilise la fonction suivante :

```
…
// recuperation de la chaine de char des extension
char *ext = (char*)glGetString( GL_EXTENSIONS );

// on test si l'extension voulue est présente
if( strstr( ext, "ARB_vertex_buffer_object" ) == NULL )
   {
      MessageBox(NULL,"ARB_index_buffer_object n'existe pas",
         "ERROR",MB_OK|MB_ICONEXCLAMATION);
      return E_FAIL; // extension non présente
   }
…
```

Si l'extension est présente, il suffit de récupérer les pointeurs sur les fonctions.

```
…
// on récupère les fonctions de l'extension
glBindBufferARB = (PFNGLBINDBUFFERARBPROC)wglGetProcAddress("glBindBufferARB");
glGenBuffersARB = (PFNGLGENBUFFERSARBPROC)wglGetProcAddress("glGenBuffersARB");
glBufferDataARB = (PFNGLBUFFERDATAARBPROC)wglGetProcAddress("glBufferDataARB");
glGetBufferParameterivARB =
```

```
(PFNGLGETBUFFERPARAMETERIVARBPROC)wglGetProcAddress("glGetBufferParameterivARB");
glDeleteBuffersARB = (PFNGLDELETEBUFFERSARBPROC)wglGetProcAddress("glDeleteBuffersARB");
…
```

Si les extensions sont présentes, on peut générer les buffers ARB voulu. Il suffit de faire appel à la function glGenBuffersARB(), de sélectionner le buffer à remplir avec glBindBufferARB() puis de remplir sa mémoire avec glBufferDataARB(). Ceci copie automatiquement les informations des buffers de la ram vers la mémoire video.

```
…
// Génération de l'index buffer
glGenBuffersARB( 1, &IndexBuffer );
glBindBufferARB( GL_ELEMENT_ARRAY_BUFFER_ARB, IndexBuffer );  // sélection du buffer
glBufferDataARB( GL_ELEMENT_ARRAY_BUFFER_ARB, sizeof(UINT) * nb_faces*3, index,
GL_STATIC_DRAW_ARB );

glGetBufferParameterivARB( GL_ELEMENT_ARRAY_BUFFER_ARB, GL_BUFFER_SIZE_ARB,
&nParam_ArrayObjectSize );

glBindBufferARB( GL_ELEMENT_ARRAY_BUFFER_ARB, NULL );  // désélection du buffer

// Génération de du vertex buffer
glGenBuffersARB( 1, &VertexBuffer );
glBindBufferARB( GL_ARRAY_BUFFER_ARB, VertexBuffer );
glBufferDataARB( GL_ARRAY_BUFFER_ARB, sizeof(VERTEX) * nb_vertexes, vertexes,
GL_STATIC_DRAW_ARB );
glGetBufferParameterivARB( GL_ARRAY_BUFFER_ARB, GL_BUFFER_SIZE_ARB,
&nParam_ArrayObjectSize );

glBindBufferARB( GL_ARRAY_BUFFER_ARB, NULL );

delete [] index ; index  = NULL ;  // suppression des tableaux
delete []vertexes; vertexes = NULL;
```

Pour l'affichage, il faut sélectionner le buffer et indiquer à OpenGL qu'il faudra l'appeler avec la fonction glEnableClientState().

```
…
// affichage
// Sélection de l'indexbuffer
glBindBufferARB( GL_ELEMENT_ARRAY_BUFFER_ARB, IndexBuffer );
glIndexPointer( GL_UNSIGNED_INT, sizeof(GLuint), BUFFER_OFFSET(0) );
glEnableClientState( GL_INDEX_ARRAY );

// Sélection du vertexbuffer
glBindBufferARB( GL_ARRAY_BUFFER_ARB, VertexBuffer );

glColorPointer( 4, GL_FLOAT, sizeof(VERTEX), BUFFER_OFFSET(0) );
glNormalPointer( GL_FLOAT, sizeof(VERTEX), BUFFER_OFFSET(sizeof(GLfloat) * 4 ));
glVertexPointer( 3, GL_FLOAT, sizeof(VERTEX), BUFFER_OFFSET(sizeof(GLfloat) * 7) );
```

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

// affichage indexé ARB
glDrawElements( GL_TRIANGLES, nb_faces*3, GL_UNSIGNED_INT, BUFFER_OFFSET(0));

glDisableClientState( GL_COLOR_ARRAY );
glDisableClientState( GL_NORMAL_ARRAY );
glDisableClientState( GL_VERTEX_ARRAY );

// Désélection des index & vertex buffer
glBindBufferARB( GL_ARRAY_BUFFER_ARB, NULL );
glBindBufferARB( GL_ELEMENT_ARRAY_BUFFER_ARB, NULL );
…
```

La macro BUFFER_OFFSET sert à créer un offset pour la lecture des informations dans la structure. Elle a la forme suivante :

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

A la fin du programme on supprime les buffers situés en mémoire vidéo :

```
// Fin du programme
glDeleteBuffersARB( 1, &IndexBuffer );  // suppression des buffers de la mémoire vidéo
glDeleteBuffersARB( 1, &VertexBuffer );
```

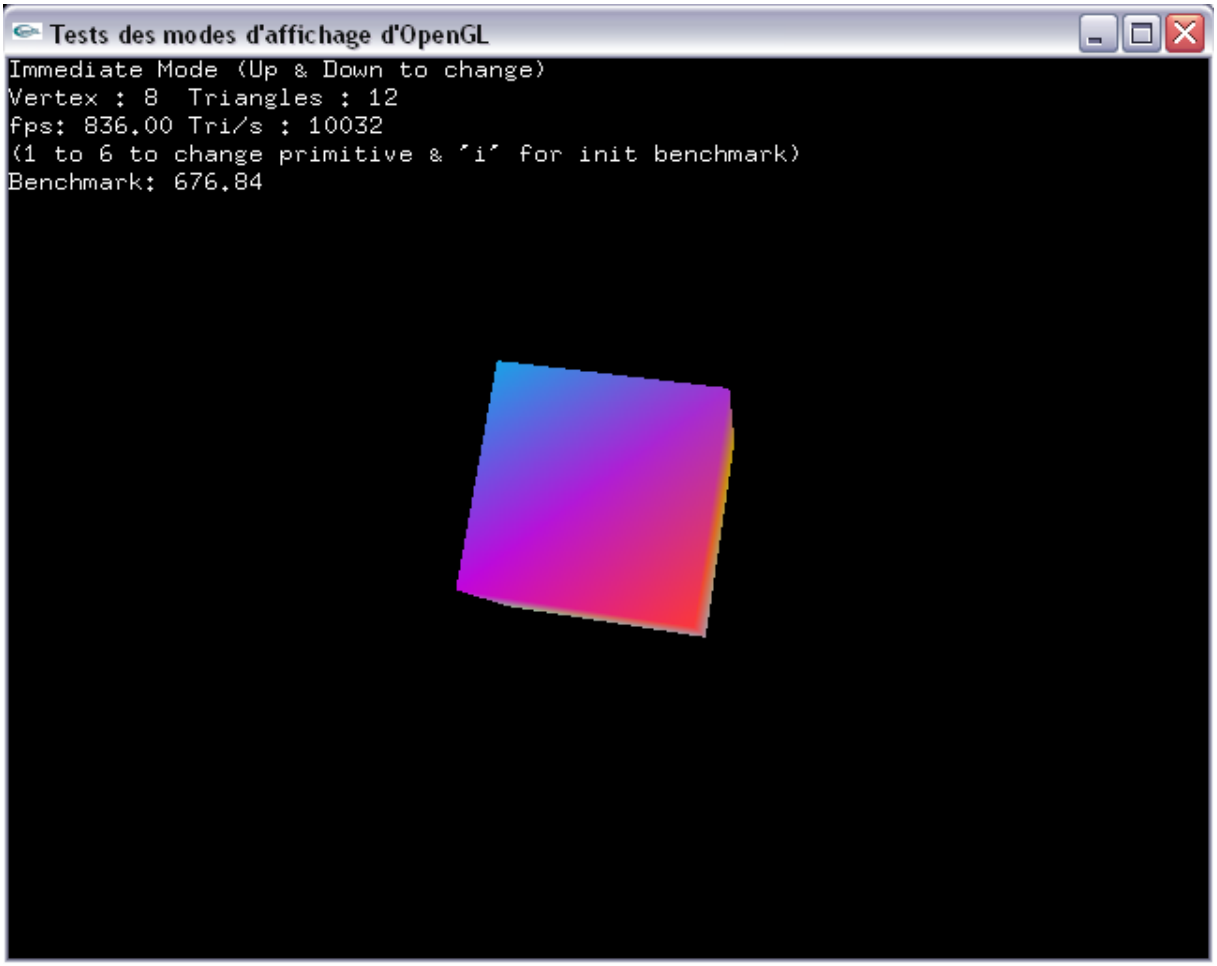
6 Récapitulatif des méthodes

Méthode	Niveau	Lieu de stockage	Dynamique/statique
Immediate mode	Facile	Aucun	Dynamique
Display list	Facile	Mémoire graphique (en cache selon les drivers)	Statique
Vertex Arrays	Moyen	Mémoire Ram du pc	Dynamique
Interleaved Vertex Arrays	Moyen	Mémoire Ram du pc	Dynamique
Buffers ARB	Difficile	Mémoire graphique	Dynamique

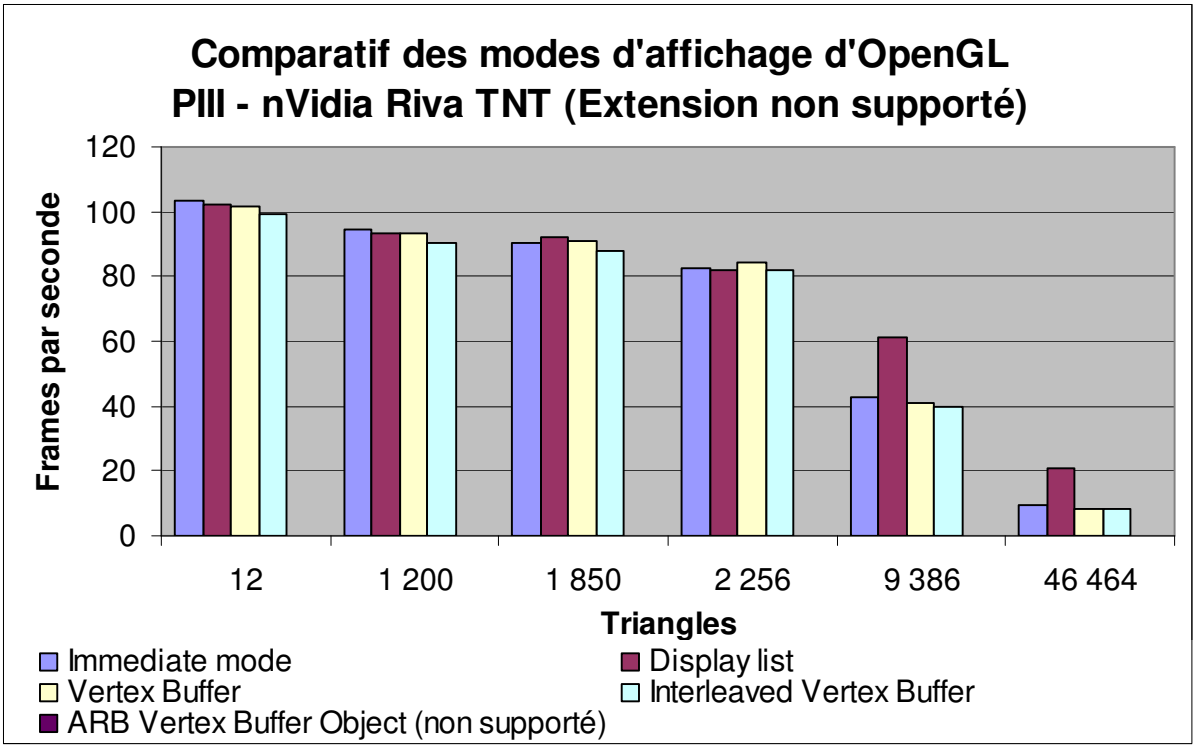
IV - Benchmark des différents systèmes d'affichage

Afin de réaliser un test de ces différentes méthodes d'affichage, nous essaierons ces techniques avec différentes scènes n'ayant pas le même nombre de polygones. De plus, nous testerons aussi avec différent type de carte graphique pour avoir plus d'informations à recouper pour une meilleure évaluation de ces modes d'affichage. J'ai donc conçu un programme qui va avoir les différentes méthodes d'affichage possible et réaliser des testes de fps.

Voici un screenshot du programme :

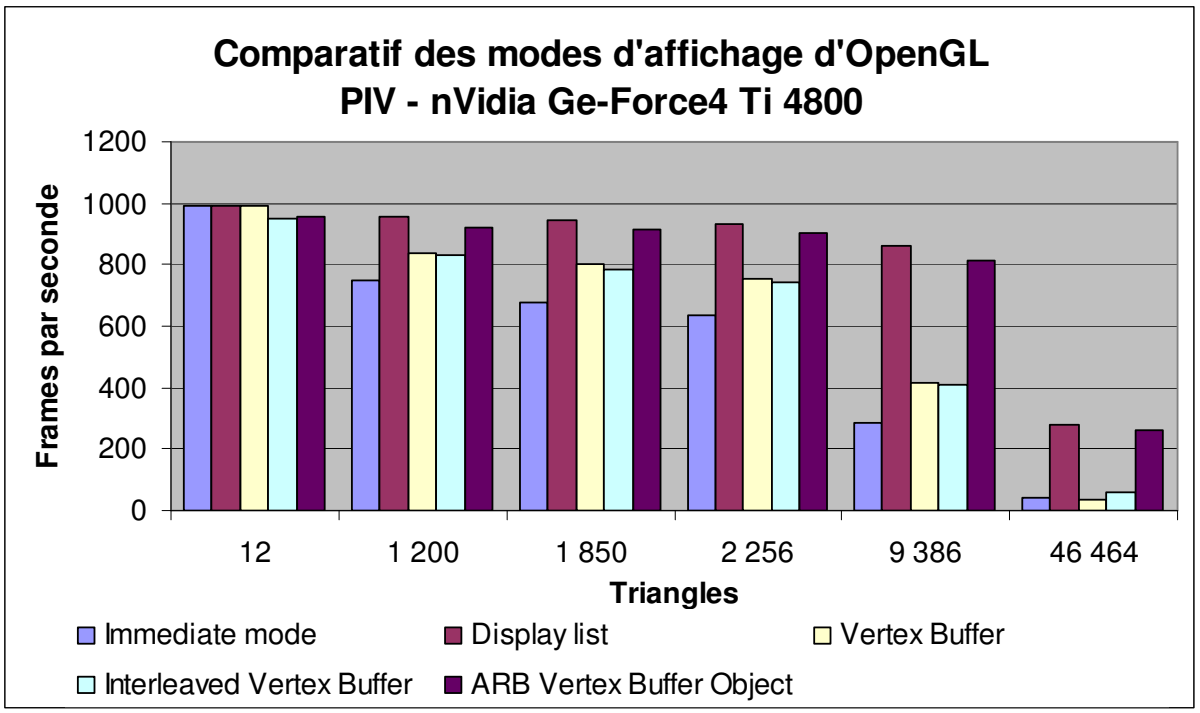


1 Test 1 : nVidia Riva TNT



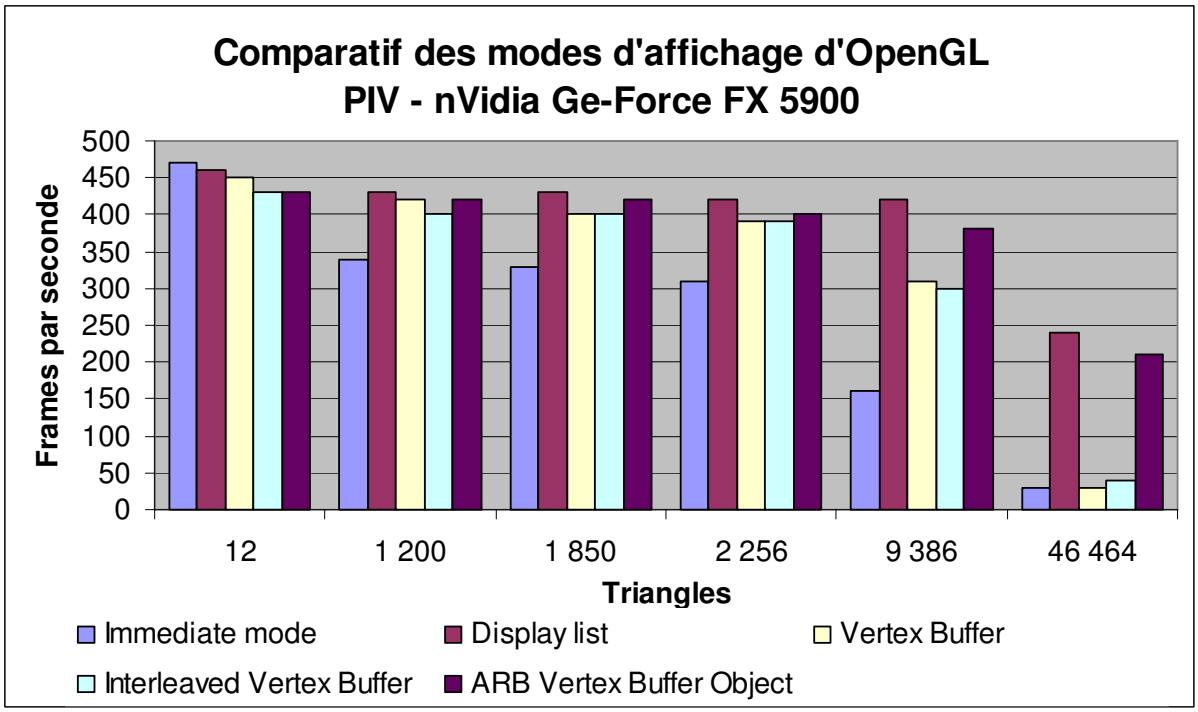
Tout d'abord, nous pouvons constater que le nombre de triangles dans la scène joue sur le fps. Plus il y a de triangles, plus celui-ci baisse. Nous pouvons aussi remarquer que les méthodes d'affichage sont à peu près équivalentes en dessous d'un certain seuil de triangles. Passé ce seuil, on peut comparer la vitesse d'affichage. Ici le seuil se trouve au alentour des 9 000 triangles. On voit nettement la supériorité des GL Liste. Les buffers ARB n'étant pas supporté, nous ne pouvons les comparés aux autres méthodes.

2 Test 2 : nVidia Ge-Force4 Ti 4800



Ce test est très concluant, on voit nettement la supériorité des méthodes utilisant la mémoire vidéo pour stocker les informations des sommets. La méthode des buffers ARB est cependant un peu moins rapide qu'une liste d'OpenGL.

3 Test 3 : nVidia Ge-Force FX 5900



Ce test affirme celui du dessus, et prouve une fois de plus la rapidité d'exécution de la display liste et des buffers ARB.

4 Conclusion

Tout ces testes réalisé on a peu près les même résultats. La Liste Gl et les buffers ARB sont les plus rapides. Mais la performance dépend beaucoup du nombre de polygones que contient la scène. Cela parait assez logique vu que les données transite par le bus pour les autres méthodes. Plus il y a de triangles à affiché, plus le bus s'encombre pour passer les informations à la carte graphique. Le seule moyen de contourner ce système et donc de mettre toutes les informations dans la mémoire interne à la carte graphique. La display liste est là méthode la plus rapide, mais elle est limité à cause de sa compilation, on ne peut changer la position des vertexes. Donc si on souhaite utiliser des formes qui vont bouger en fonction du temps, il faudrait utiliser les buffers dynamique. Ceux-ci permettent l'accès rapide aux données stockées en mémoire vidéo afin de pouvoir effectuer les modifications nécessaires.

V - Pourquoi Timer les actions ?

Pourquoi est-il important de timer les actions ? Comme nous l'avons vu précédemment, le fps peut changer en fonction des cartes graphique et des moyens utilisés pour afficher les primitives. Si l'on souhaite faire une rotation à chaque frame d'un cube par exemple, selon les ordinateurs, le cube va allé plus ou moins vite. Ce que l'on veut c'est que le cube aille à la même vitesse sur toutes les machines. Il ne faut donc pas calculé le déplacement en fonction de la vitesse d'exécution du programme, mais en fonction du temps. Pour cela nous avons accès à des fonctions de temps : TimeGetTime() ou GetTickCount() sont donc des fonctions utiles pour la gestion des animations pour un rendu 3D. D'autres fonctions encore plus précises peuvent être utilisé comme par exemple le compteur de performance de windows.

VI - Conclusion

OpenGL propose divers moyen d'afficher les primitives, ces méthodes sont plus ou moins rapides d'exécution. La méthode de dessin statique la plus rapide est l'utilisation des listes GL, mais si l'utilisateur souhaite modifier ses sommets rapidement, les buffers ARB sont la méthode la plus appropriée.