TPs Architecture des ordinateurs
DUT Informatique - M4104c

–SUJETS–

R. Raffin
Aix-Marseille Université
romain.raffin-At-univ-amu.fr

2016

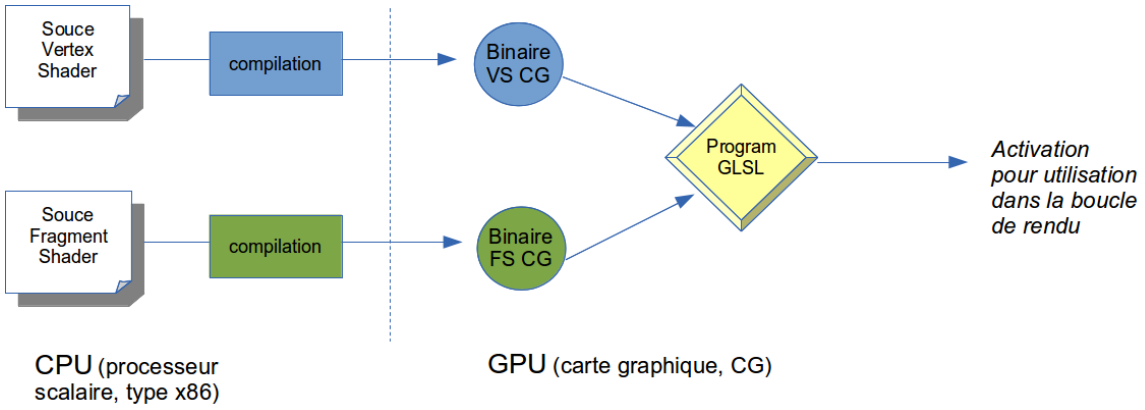
Table des matières

1 TP 1 : prise en main

Il faut télécharger sur Ametice l'archive des sources des TP. Nous aurons également besoin de la librairie GLEW.

1.1 Introduction

Pour plus de facilité, vous allez utiliser dans ces TP la bibliothèque open-source « GLEW ». Elle permet d'éviter les appels de fonctions OpenGL pour les shaders, qui peuvent être différents d'une version d'OpenGL à une autre. Un shader OpenGL porte notamment [1] sur la géométrie (« Vertex Shader » ou VS) ou sur les fragments (« Fragment Shader » ou FS) qui sont des pixels avec une pseudo-profondeur (comme dans le Z-Buffer). Pour les utiliser, il faut construire un« Shader Program » qui est l'union d'un VS et d'un FS. Chaque *shader* est compilé par le pilote de la carte graphique, on peut donc faire cette étape avant le début de la boucle de rendu. La création d'un « Program » suivra donc les étapes de la figure suivante (cf. « SetShaders() » dans le source « ArchiINtp1.cpp »).



1.2 Sources à disposition

1.2.1 Fichiers

Pour commencer à travailler sur des fragment ou des vertex shader, vous trouverez l'archive « `TP1_premierPas.zip` » sur le site. Cette archive contient des fichiers pour décrire les classes dont on a besoin :

- `GLSL_VS .cpp/.h`, pour décrire un objet « Vertex Shader » , avec les fonctions :
  - « `ReadSource(const char* fichier)` », pour lire les sources du shader dans un fichier (par exemple « tp11.vert ») ;
  - « `Compile()` » pour créer le code binaire correspondant.
- `GLSL_FS .cpp/.h`, identique au précédent (mêmes fonctions) pour un « Fragment Shader » ;
- `GLSL_Program .cpp/.h` qui gérera votre « Shader Program » avec :
  - « `Use_VertexShader(identifiant)` » et « `Use_FragmentShader(identifiant)` » qui permettent d'identifier les shaders (déjà compilé) que le programme utilisera ;
  - « `LinkShaders()` » pour lier dans la carte graphique les 2 binaires ;
  - « `Activate()` » qui rend actif le programme pour la boucle de rendu OpenGL.

Les fonctions de GLEW sont encapsulées dans ces objets, donc n'hésitez pas à parcourir les sources. Vous trouverez aussi dans les sources « ArchiINtp1.cpp » , qui est le programme principal (OpenGL classique), des fonctions pour l'affichage des erreurs, quelques *shaders* (dont le nom de fichier contient `_trav` pour vous aider dans le TP) et le *Makefile* pour compiler (taper « make »).

1.2.2 Compilation

Pour compiler ces fichiers, le « Makefile » effectue les opérations suivantes (si un fichier source a été modifié ou que le binaire exécutable n'existe pas) :

1. il existe aussi les *tesselation* et *geometry shaders*

1. `g++ -c -O2 -Wall -o XXX.o XXX.cpp` , pré-processing du code de « XXX.cpp », sortie « XXX.o » pour tous les fichiers sources ;
2. et finalement, `g++ -Lglew/lib -lGLEW -lglut XXX.o YYY.o ZZZ.o -o ArchiINtp1` , éditions des liens pour créer l'exécutable « ArchiINtp1 ». On utilise les librairies de GLEW (« -lGLEW »).

Si tout se passe bien, vous avez un fichier « ArchiINtp1 » qui est exécutable et qui utilise des *vertex* et *fragment shaders*.

1.3 Comment cela fonctionne ?

Pour aller un peu plus loin dans l'analyse, voici la liste des fonctions utilisées dans ce programme :

1. charger et lire le fichier source (attention à l'encodage, UTF8 ou non peut entraîner des erreurs) ;
2. créer des identifiants de *shader* (ces variables sont de type `GLuint`) :

   ```
   VertexShader = glCreateShader(GL_VERTEX_SHADER);
   FragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
   ```

3. lier ces deux sources (chaîne de caractères) à un structure OpenGL :

   ```
   glShaderSource(IDVertexShader, 1, &constVS, NULL);
   glShaderSource(IDFragmentShader, 1, &constFS, NULL);
   ```

4. compiler ces 2 sources :

   ```
   glCompileShader(IDVertexShader);
   glCompileShader(IDFragmentShader);
   ```

5. éventuellement, s'il y a erreur, afficher le rapport de compilation)

   ```
   printShaderInfoLog(IDVertexShader);
   printShaderInfoLog(IDFragmentShader);
   ```

6. créer un ProgramObject, qui contient les binaires du *vertex* et du *fragment shader* et qui remplacera le programme natif de la carte graphique :

   ```
   IDProgramObject = glCreateProgram();
   ```

7. attacher les 2 *shaders* à ce programme (on peut les détacher et en attacher d'autre dynamiquement) :

   ```
   glAttachShader(IDProgramObject, IDVertexShader);
   glAttachShader(IDProgramObject, IDFragmentShader);
   ```

8. lier le programme au moteur graphique :

   ```
   glLinkProgram(IDProgramObject);
   ```

9. éventuellement, s'il y a erreur, afficher le rapport de compilation :

   ```
   printProgramInfoLog(ProgramObject);
   ```

10. et pour finir, utiliser ce programme pour toutes les actions OpenGL qui suivent :

    ```
    glUseProgram(IDProgramObject);
    ```

1.4 Qu'est-ce que l'on peut faire ?

1. compiler et exécuter le programme d'exemple. Si tout va bien vous avez une magnifique théière bleue affichée alors que la boucle de rendu OpenGL prévoyait du rouge (vérifiez dans les sources) ;
2. modifier le *fragment shader* : plutôt que de fixer la couleur de manière arbitraire, on récupère la couleur des sommets dans le *vertex shader* et on s'en sert dans le *fragment shader*. Les codes des *shaders* sont :

```
//VERTEX SHADER
void main(void)
{
  //renvoie la couleur de la face avant sur les sommets dans le pipeline
  gl_FrontColor = gl_Color;
  //effectue les transfos de modélisation et de vue
  gl_Position = ftransform();
}
```

```
//FRAGMENT SHADER
void main (void)
{
  vec4 vectcolor = gl_Color;  //on récupère la couleur dans le pipeline
  gl_FragColor = gl_Color;  //et on la renvoie pour l'affichage
}
```

3. modifier le *vertex shader* : comme on récupère une position dans « `gl_Vertex` », on peut la faire modifier par la carte graphique. Par exemple, une translation de $(1, 1, 0)$:

```
//VERTEX SHADER
void main(void) {
  //Translation
  vec4 point = gl_Vertex;
    point.x = point.x + 1.0;
    point.y = point.y + 1.0;
  gl_Position = gl_ModelViewProjectionMatrix * point;
  }
```

4. modifier le *vertex shader*, en fonction de données qui sont transmises par le programme en C++. On utilise des variables de type « Uniform » . Comme elles sont stockées sur la carte graphique, on doit depuis le prog. C++, pouvoir les modifier. Il faut connaître l'endroit où la carte les stocke (la carte renvoie une référence d'un emplacement mémoire). Par exemple, on passe une variable « mytime » incrémentée dans le prog en C++ « RenderScene » et on s'en sert pour déformer les sommets dans le *vertex shader* :

```
//VERTEX SHADER
uniform float cputime;

void main(void)
{
vec4 v = vec4(gl_Vertex);

  v.z = sin(5.0*v.x + cputime)*0.5; //sin() est une fonction implémentée dans le  GPU, cf GLSL référence
  gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

```
//PROGRAMME C++
<variables en global>

GLint loc;
float mytime=0.0f;

  <fonction RenderScene()>
  glRotatef(angle,1,1,0);
  glColor3f(1.0, 0.0, 0.0);
  glUniform1fARB(loc, mytime);  //on fait la liaison entre la valeur en GPU à l'adresse loc et la valeur dans mytime glutSolidTeapot(1);
  mytime+=0.01;

  <fonction setShaders()>
  loc = glGetUniformLocation(mes_shaders -> idprogram ,"cputime"); //on récupère l'adresse de la variable cputime du GPU, par son nom, dans loc
```

5. et ensuite ? Remplacer la théière par une sphère (on peut changer plus facilement le nombre de sommets et de faces) ; Effectuer une déformation radiale (qui part du centre vers l'extérieur de la sphère) dans le *vertex shader*, fonction d'un paramètre *Uniform* envoyé par le programme C++ et qui sera aléatoire (*random*). Si on note ce paramètre « chaos », on effectuera dans le *vertex shader* l'opération : point = *point* × chaos (ou point+ = chaos).
   « chaos » est un `float`, la sphère est centrée en $(0, 0, 0)$, ce qui veut dire que l'on fait une mise à l'échelle de chacun des points de la sphère, de manière globale (une valeur de « chaos » par géométrie affichée). La géométrie est encore conforme car on sait comment les sommets d'une sphère sont reliés entre eux par les arêtes et les faces, donc il n'y aura pas de trou dans l'objet. Ne prenez pas de trop grande valeur de « chaos » (entre $-0,2$ et $+0,2$ pour commencer). « chaos » peut aussi varier en fonction du temps (par exemple avec une fonction `sin()`)...

1.5 Liens sur la programmation de shaders

— `http://nehe.gamedev.net/` (article 21)
— `http://glew.sourceforge.net/`
— `http://www.opengl.org/sdk/`
— `http://www.opengl.org/sdk/docs/tutorials/`
— `http://www.lighthouse3d.com/tutorials/`

2 TP2 : traitement de géométrie et d'image

Il faut télécharger l'archive des sources du TP et éventuellement la librairie FreeImage [1] qui permet l'ouverture « facile » de divers formats d'images bitmap.

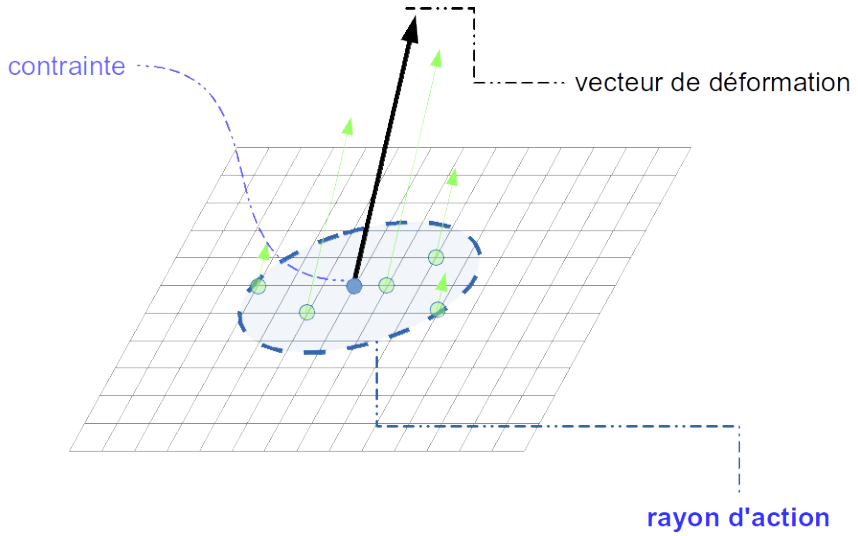
2.1 Déformation d'une géométrie

mots-clés de cet exercice : `varying`, `uniform`, interaction VS*rightarrow*FS, envoi de valeurs au VS

Pour commencer à travailler vous trouverez l'archive du TP « `TP2deformation.zip` » sur le site intranet. Cette archive contient les sources des 2 exercices et leurs *shaders* (.vert et .frag), ainsi qu'un objet `.off` (format d'objet 3D qui peut être exporté par Blender par ex.) qui représente un plan sur X et Y.

La déformation utilisée est effectuée par le *vertex shader*, nous l'avons déjà vu en CM et TD. Son principe est le suivant :

1. on définit un centre (coordonnées x, y, z) pour la contrainte de déformation ;
2. on donne un rayon d'action à cette contrainte ;
3. on définit un vecteur qui permet de déformer le plan ;



On parcourt ensuite tous les points de l'objet, si un point est dans le voisinage de la contrainte (voisinage donné par le rayon) alors on déforme ce point : on lui ajoute les coordonnées du vecteur de déformation. Il y a une règle très simple dans le « tpdeformation.vert » qui dit que si le point est au centre de la contrainte, sa déformation est totale (= vecteur) et vaut 0 si le point est en dehors du voisinage. Cela ne devrait donc déformer qu'un point de la géométrie (si la contrainte ne bouge pas). Et encore, il se peut que la contrainte ne soit pas exactement sur un sommet de l'objet. Pour avoir plus de point déformés, on fait une interpolation linéaire de la déformation entre le maximum (si le point testé est au centre de la contrainte) et le minimum (si le point testé est au-delà du rayon d'action).

À faire :

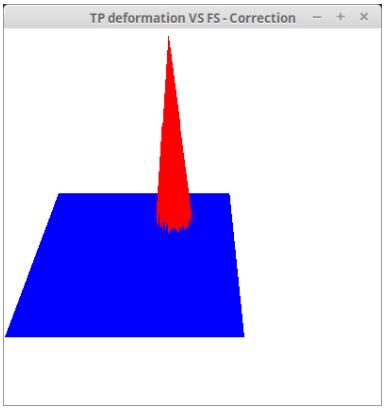
1. faire tourner la contrainte autour d'un cercle sur le plan. Pour cela, il faut passer du CPU vers le GPU les coordonnées du centre de la contrainte et son rayon. C'est déjà fait dans le programme d'exemple, il reste à implémenter le mouvement de la contrainte (calcul fait par le CPU). Souvenez-vous de l'équation paramétrique du cercle :

$$\begin{cases} x = & rcos(\alpha) \\ y = & rsin(\alpha) \end{cases}$$

2. afficher en rouge les points touchés par la déformation. Remarque : pour passer des valeurs d'un *vertex shader* à un *fragment shader*, on utilise des variables de type `varying`. Elles doivent être présentes dans ces deux *shaders*. Par exemple :

1. `http://freeimage.sourceforge.net/`

```
varying float Vfactif;
```



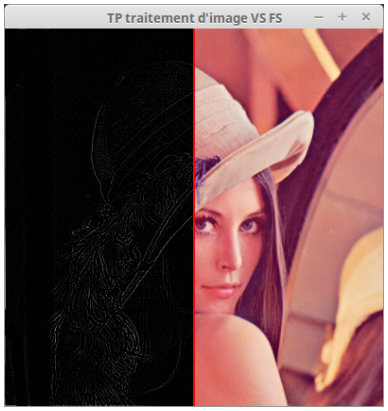
2.2 Traitement d'une image

mots-clés de cet exercice : *fragment shader*, convolution et masque, *offset* de texture, `uniform sampler2D`

Pour traiter une image avec un *fragment shader*, on a besoin de sommets qui passent dans un VS puis qui sont utilisés dans un FS. Le plus simple est donc d'utiliser un `GL_QUAD` sur lequel on va plaquer une texture, et avec lequel on effectuera le *fragment shader*. Vous trouverez les sources de ce TP dans l'archive « `TP2_traitementImage.zip` ». Le *vertex shader* contient juste la transformation des points 3D en points 2D et le passage de la texture au *fragment shader* :

```
void main( void )
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

La texture est récupérée dans le *fragment shader* dans un « uniform sampler2D ». On peut alors utiliser le point (`gl_TexCoord[0].st`) de la 1ère texture (« [0] ») du *sampler2D* pour avoir sa couleur. On peut aussi utiliser des décalages pour tenir compte des points voisins (« offset »). Cela nous permet de programmer des masques de Traitement d'images (convolution) très facilement (cf. « tptraitementimage.frag »).



À faire :

1. écrire un *shader* qui met l'image en niveaux de gris ;
2. écrire un *shader* qui passe l'image en noir et blanc, en passant le seuil en `Uniform` depuis le programme C++, faire varier le seuil à l'aide des touches '+' et '-' ;
3. reprendre l'exemple du TP, utiliser d'autres masques de convolution (cf le *fragment shader* qui contient d'autres matrices), attention aux masques moyenneurs (coefficient et division) ;
4. mettre le noyau de convolution, sa taille et les *offsets* en `uniform` et les calculer dans le programme principal en `.cpp` plutôt que dans le *fragment shader*.

Quelques liens sur les convolutions possibles en traitement d'images (en plus du cours de deuxième année de DUT Informatique) :

— `http://docs.gimp.org/fr/plug-in-convmatrix.html`
— `http://xmcvs.free.fr/astroart/Chapitre4.pdf`

3

TP 3 : optimisation des transferts de géométrie (Display List, VA, VBO)

Il faut récupérer sur l'intranet l'archive des sources du TP.

3.1 Les géométries et primitives, à l'ancienne

mots-clés de cet exercice : `Display List`, `Vertex Arrays`, `Vertex Buffer Object`.
Dans d'anciennes versions d'OpenGL vous manipuliez des :

```
glBegin(GL_TRIANGLES);
  glVertex3f(...);
  glVertex3f(...);
  glVertex3f(...);
glEnd();
```

dans la boucle de rendu d'OpenGL. Cela implique qu'à chaque image (à chaque fois qu'OpenGL a besoin de retracer), on re-décrit toute la géométrie de la scène et qu'on la renvoie de la RAM du CPU vers la RAM du GPU. Si l'objet n'est pas modifié, c'est beaucoup de temps et d'énergie perdus ! Vous avez peut-être donc vu les `DisplayLists`, qui permettent la pré-compilation des objets en RAM. La carte graphique arrange les données, les prépare pour leur utilisation et repère leurs emplacements grâce à un identifiant. Par exemple :

```
id = glGenLists(1);

glNewList(id, GL_COMPILE);

for (int i=0; i < nbfaces; i++)
  glBegin(GL_TRIANGLES);
    glVertex3f(...);
    glVertex3f(...);
    glVertex3f(...);
  glEnd();
}

glEndList();
```

On effectue donc un pré-travail pour la carte graphique, il n'y a plus qu'à appeler la liste (`glCall(id)`) pour l'envoyer au GPU pour traitement. Il reste 2 inconvénients : les transmissions CPU->GPU et l'impossibilité de modifier ces géométries sans recréer la liste. Bon, le 2^{e} inconvénient on l'a résolu dans le TP2, en implémentant un *vertex shader* qui savait modifier une géométrie donc potentiellement aussi dans une `DisplayList`. C'est évidemment plus rapide, pas la peine de recréer la géométrie à chaque fois mais on peut peut-être faire mieux...

Note : cette partie est faite par la fonction `CreateDisplayListFromObject()` du source « ArchiINtp31.cpp » (lignes 236-259). Cette fonction utilise un objet chargé au début du programme et le compile en une `DisplayList`. Pour tester cette fonction, lancer le programme avec l'option « DL » :

```
./ArchiIN_tp3  DL
```

3.2 Les géométries, en mieux

mots-clés de cet exercice : `VertexArray`
Depuis la version 1.2 d'OpenGL, on a la possibilité de préparer un peu mieux les données pour la carte graphique. Les `VertexArray` permettent de stocker la géométrie d'un objet, mais aussi sa (ses) couleur(s), ses normales, ses coordonnées de texture... dans des tableaux. Lors du rendu, on envoie ces tableaux à la carte graphique qui n'a plus qu'à lire.

3.2.1 Comment fait-on ?

On crée des tableaux qui vont contenir les données de l'objet (des tableaux de `float`), on met dedans toutes les données de l'objet chargé, et on les utilise plus tard. Attention, ces tableaux doivent être accessibles par la boucle de rendu (en global pour faire simple mais *moche*). Dans l'exemple du TP, la géométrie tirée du fichier était indexée, on en crée une géométrie de triangles non indexée (à la `GL_TRIANGLES`) [1].

Dans le rendu, on active le rendu depuis des *buffers*, pour chaque type de données à transmettre :

```
glEnableClientState(GL_VERTEX_ARRAY); //attention, on va se servir de sommets
glEnableClientState(GL_NORMAL_ARRAY); //attention, on va se servir de normales
glEnableClientState(GL_COLOR_ARRAY);  //attention, on va se servir de couleurs

  //et voilà le buffer des couleurs, sur 3 float pour chaque couleur
  glColorPointer(3, GL_FLOAT, 0, ColorArray);

  //le buffer des normales, sur des floats (3 implicite)
  glNormalPointer(GL_FLOAT, 0, NormalArray);

  //le buffer des sommets, 3 floats pour chaque coordoonées
  glVertexPointer(3, GL_FLOAT, 0, VertexArray);

    //et hop, on envoie au rendu de la CG
    glDrawArrays(GL_TRIANGLES, 0, monobjet.nbfaces * 3);

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

Note : cette partie est faite par les fonctions `CreateVertexArraysFromObject')` (lignes 175-234) et `DrawVertexArray()` (lignes 366-385) du source « `ArchiIN_tp31.cpp` ». Cette fonction utilise le même objet que précédemment. Pour tester cette fonctionnalité, lancer le programme avec l'option « VA » :

```
./ArchiIN_tp3  VA
```

Mais on peut faire encore mieux...

3.3 Les géométries, en encore mieux

mots-clés de cet exercice : `VertexBufferObject`

Depuis OpenGL 1.5 on peut se servir des `VertexBufferObject` - VBO. L'intérêt de ces petites bêtes est que l'on peut stocker directement la géométrie sur la carte graphique. Comme pour les `VertexArrays`, on utilise des tableaux pour stocker la géométrie d'un objet, on lie ensuite ces tableaux avec des zones mémoires de la carte graphique, on copie les données, et hop, c'est prêt ! (du coup on peut désallouer la mémoire en RAM du CPU puisque la géométrie est sur la carte graphique). Il devient donc inutile de conserver la géométrie à la fois dans le CPU et le GPU, d'où un gain de mémoire en plus du gain de transfert.

On peut gérer les VBO sous 3 modes :

- `GL_STREAM_DRAW` lorsque les informations sur les sommets peuvent être mises à jour entre chaque rendu (similaire aux `VertexArrays`). On envoie les tableaux à chaque image.
- `GL_DYNAMIC_DRAW` lorsque les informations sur les sommets peuvent être mises à jour entre chaque frame. On utilise ce mode pour laisser la carte graphique gérer les emplacements mémoires des données, pour le rendu multi-passe notamment.
- `GL_STATIC_DRAW` lorsque les informations sur les sommets ne sont pas mises à jour - ce qui redonne le fonctionnement d'une `DisplayList`.

Se rajoute à ces modes, des modes d'accès aux données : `READ`, `WRITE`, `COPY`, `READ_WRITE`. Ce qui permet également de copier des parties d'un objet dans un autre, et même de faire des interactions avec le CPU.

Note : attention aux différents modes, cela ne fonctionne pas sur toutes les cartes graphiques.

3.3.1 Comment fait-on cela ?

Préparation des données :

1. on crée les tableaux de géométrie comme pour les `VertexArray`
2. on crée les *buffers* GPU qui contiendront ces données (`glBindBuffer()`, `glBufferData()`)

```
//VBO pour les sommets
glGenBuffers((GLsizei) 1, &VBOSommets);
  glBindBuffer(GL_ARRAY_BUFFER, VBOSommets);
  glBufferData(GL_ARRAY_BUFFER, _monobjet -> nbfaces * 9 * sizeof(float), lsommets, GL_STATIC_DRAW);

//VBO pour les couleurs
glGenBuffers((GLsizei) 1, &VBOCouleurs);
  glBindBuffer(GL_ARRAY_BUFFER, VBOCouleurs);
  glBufferData(GL_ARRAY_BUFFER, _monobjet -> nbfaces * 9 * sizeof(float), lcolors, GL_STATIC_DRAW);
```

1. Ceux qui ne sont pas d'accord n'ont pas lu le sujet jusqu'à la section 5...

```
//VBO pour les normales
glGenBuffers((GLsizei) 1, &VBONormales);
  glBindBuffer(GL_ARRAY_BUFFER, VBONormales);
  glBufferData(GL_ARRAY_BUFFER, _monobjet -> nbfaces * 9 * sizeof(float) , lnormales, GL_STATIC_DRAW
    );
```

Et pour le rendu :

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

  glBindBuffer( GL_ARRAY_BUFFER, VBOSommets);
  glVertexPointer( 3, GL_FLOAT, 0, (char *) NULL );

  glBindBuffer( GL_ARRAY_BUFFER, VBOCouleurs);
  glColorPointer( 3, GL_FLOAT, 0, (char *) NULL );

  glBindBuffer( GL_ARRAY_BUFFER, VBONormales);
  glNormalPointer(GL_FLOAT, 0, (char *) NULL );

    glDrawArrays(GL_TRIANGLES, 0, monobjet.nbfaces * 3);

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

Note : cette partie est faite par les fonctions `CreateVBOFromObject()` (lignes 87-173) et `DrawVBO()` (lignes 341-364) du source « `ArchiIN_tp31.cpp` ». Cette fonction utilise le même objet que précédemment.

Pour tester cette fonctionnalité, lancer le programme avec l'option « VBO » :

```
./ArchiIN_tp3  VBO
```

3.4 À faire

1. faire des `VertexArrays` en géométrie indexée
2. faire des VBO en géométrie indexée

Note : si vous avez les mêmes valeurs d'images/seconde pour les différents mode (`Display List`, `Vertex Array`, `Vertex Buffer Object`), vérifiez les paramètres de votre carte graphique (performance/qualité, synchronisation verticale).

Note2 : au passage allez voir le fonctionnement de la fonction `atexit()`.