Ce cours concerne le démarrage de votre projet LO13. Après quelques compléments sur le langage C, on définit la structure informatique d'un modèle discret (par abus de langage, un maillage), en particulier le cube unité, puis on s'intéresse à la visualisation de ce modèle en utilisant la bibliothèque OpenGL.

Complément du C

Pour déclarer un ensemble on utilise les [et] en précisant la dimension entre [et] devant le nom de la variable.

Exemple :

```
float    point[3] ;
```

la variable point est un ensemble de 3 réels : point[0], point[1], point[2]. Les indices en C commencent toujours à 0 pour les ensembles. Comme point est de dimension 3, on aura comme indice 0, 1 et 2 (attention point[3] n'est pas alloué en mémoire donc ne peut être utilisé). La règle générale pour les indices est donc de 0 à dimension-1.

Une affection possible après cette déclaration peut être :

```
point[1] = 1.0 ;
```

Vous pouvez utiliser les types prédéfinis (int, float, char, …) pour définir des ensembles. Dans un premier temps, on s'intéresse aux ensembles dont la dimension (la taille) est définie à l'avance (pour le point c'est 3). Plus tard, on introduira les ensembles dont la taille a priori n'est pas définie. Dans ce cas, on utilisera une allocation dynamique dès que la taille est connue.

Pour définir un objet (ici un modèle discret), on utilise une structure permettant de combiner plusieurs types en un seul type composé. La syntaxe est la suivante :

```
typedef struct nom_structure
{
  type1 champ1 ;
  type2 champ2 ;
  …
} Nom_structure ;
```

Telle que déclarée, la structure peut être désignée par deux façons différentes, soit « struct nom_structure » soit « Nom_structure » (attention à la majuscule de la première lettre). La deuxième façon est une convention que j'utilise moi-même pour la déclaration de structures (ne pas écrire « struct nom_structure » mais écrire « Nom_structure » en reprenant le nom de la structure avec la première lettre en majuscule), mais vous pouvez définir un autre nom pour cela. Les champs de la structure (les vrais type prédéfinis) sont les informations que l'on souhaite regrouper pour définir la structure ou l'objet. Pour une variable de type une structure donnée, chaque champ (comportant l'information souhaitée) est accessible via un « . » devant la variable.

Exemple :

```
typedef struct voiture
{
  char  immatriculation[7] ;
  char  modele[80] ;
  int    annee;
  …
} Voiture ;
```

Ici « struct voiture » ou « Voiture » est un type permettant de définir un objet voiture. Cet objet regroupe l'ensemble des informations que l'on souhaite avoir pour cet objet. En particulier le champ « immatriculation » est une chaîne de caractères (un ensemble de caractères) de dimension 7, « modele » une chaîne de caractères de dimension 80 et « annee » un entier. Pour déclarer une variable de ce type, on a la syntaxe suivante :

```
struct voiture  mon_voiture ;
```

ou

```
Voiture mon_voiture ;
```

Et pour des affectations :

```
mon_voiture.immatriculation = "LO00013" ;
mon_voiture.modele = "RENAULT" ;
mon_voiture.annee = 2006 ;
```

Les structures sont généralement définies dans l'entête du programme après la déclaration des macros (#include, #define, …).

Modèle discret et structure informatique associée

Le modèle discret est défini par une triangulation (par abus de langage un maillage). La triangulation est composée d'une liste de sommets incluant les coordonnées de chaque sommet et une liste de triangles incluant les numéros des sommets (les indices dans la liste des sommets) de chaque triangle. Pour notre modèle, il s'agit du cube unité comprenant 8 sommets et 12 triangles (chaque face du cube est composée de 2 triangles), voir la figure jointe. Pour définir le modèle discret, dans un premier temps on considère la structure simple suivante :

```
typedef struct mesh
{
 float  sommets[8][3] ;
 int     triangles[12] [3];
} Mesh ;
```

Le champ sommets est une matrice d'ordre 8*3. La première dimension représente le numéro du sommet variant de 0 à 7 (on a 8 sommets), et la deuxième, les coordonnées (abscisse,

indice 0, ordonnée, indice 1 et côte, indice 2) du sommet correspondant. Ainsi, si on considère la déclaration suivante pour la variable cube de type Mesh :

Mesh cube ;

On a :

cube.sommets[6][0] = 1.0 ;
cube.sommets[6][1] = 1.0 ;
cube.sommets[6][2] = 1.0 ;

Le sommet numéro 6 a pour abscisse (indice 0) 1.0, pour ordonnée (indice 1) 1.0 et pour côte (indice 2) 1.0 ;

Le champ triangles est aussi une matrice mais d'ordre 12*3. La première dimension représente le numéro du triangle variant de 0 à 11 (on a 12 triangles), et la deuxième les numéros des sommets (premier sommet, indice 0, deuxième sommet, indice 1 et troisième sommet, indice 2) du triangle correspondant. On a par exemple :

cube.triangles[0][0] = 1 ;
cube.triangles[0][1] = 2 ;
cube.triangles[0][2] = 5 ;

Le triangle numéro 0 a pour premier sommet (indice 0) 1, pour deuxième sommet (indice 1) 2 et pour troisième sommet (indice 2) 5 ; L'ordre des sommets n'est pas figé, par exemple on peut avoir :

cube.triangles[0][0] = 2 ;
cube.triangles[0][1] = 5 ;
cube.triangles[0][2] = 1 ;

Mais de manière générale, pour un triangle ABC l'ordre A, B, C est défini de manière à ce que la normale = vecteur AB ∧ vecteur BC (définie par le produit vectoriel) soit une normale sortante de l'objet.

A titre indicatif l'ordonnée du sommet j du triangle i est :
cube.sommets[cube.triangles[i][j]][1].

Ceci dit, on utilisera une procédure pour définir la géométrie du cube :

```
void    Cube (void)
{
 /* definition des sommets */
 cube.sommets[0][0] = ? ;
 …
 cube.sommets[7][2] = ? ;

 /* definition des triangles */
 cube.triangles[0][0] = ? ;
 …
```

```
  cube.triangles[11][2] = ? ;
}
```

A vous de remplir tous les champs pour le cube. En particulier reportez-vous à la figure pour les sommets et vous êtes libre pour le choix de la triangulation (en faisant attention à l'orientation des triangles pour une normale sortante).

Transformation de vue avec OpenGL

La transformation de vue est la composition de deux applications linéaires :

- Le changement de repère (repère de vue) suivi d'éventuelles transformations géométriques (rotations, translations).
- La projection parallèles ou perspective (centrale).

La première application est en fait une composition de transformations et dans un premier temps se résume en un changement de repère. Les autres transformations comprennent la manipulation géométrique de l'objet (essentiellement des rotations et des translations) via l'utilisation de la souris que l'on verra plus tard.

Chacune de ces deux applications linéaires (matrices 4*4) est calculée de manière indépendante dans OpenGL.

Je rappelle les trois règles fondamentales d'OpenGL :

1- Toute transformation est calculée via l'utilisation d'une seule matrice présente dans OpenGL, la matrice courante Mc
2- La matrice courante Mc peut être initialisée avec l'identité ou peut être stockée dans une variable utilisateur
3- L'appel à une transformation via une procédure dans OpenGL multiplie la transformation (établie par OpenGL) à droite de la matrice courante : Si T est la transformation alors après l'appel, Mc devient McT (T n'est donc pas directement accessible).

Les deux applications linéaires changement de base (éventuellement enrichi des transformation géométriques) et projection peuvent être définies en pointant la matrice courante sur l'une ou l'autre des instances. Ainsi pour ce faire, on a les instructions suivantes :

- glMatrixMode(GL_MODELVIEW) ;
- glMatrixMode(GL_PROJECTION) ;

La première instruction pointe la matrice courante sur l'instance GL_MODELVIEW. Après l'emploi de cette instruction, toute transformation invoquée via OpenGL et la matrice courante affecte cette instance. L'instance GL_MODELVIEW concerne le changement de repère (repère de vue) suivi d'éventuelles transformations géométriques (rotations, translations). La deuxième instruction pointe la matrice courante sur l'instance GL_PROJECTION. Après l'emploi de cette instruction, toute transformation invoquée via OpenGL et la matrice courante affecte cette instance. L'instance GL_PROJECTION concerne la projection et comme on verra plus tard le « zoom » !

Changement de repère dans OpenGL

L'instruction OpenGL suivante calcule une matrice de changement de repère et la multiplie à droite de la matrice courante :

gluLookAt(ox, oy, oz, fx, fy, fz, vx, vy, vz) ;

Cette instruction comprend 9 paramètres réels (les paramètres de vue) : L'observateur (ox, oy, oz), le point focal (fx, fy, fz) et la verticale (vx, vy, vz) ;

Pour visualiser le cube, vous devriez définir ces 9 paramètres dont 6 sont évidents : fx = fy = fz = 0.5 (le centre du cube) et (vx, vy, vz) = (0.0, 0.0, 1.0) le z du repère global. Les trois premiers paramètres sont à définir pour avoir une vue raisonnable du cube (où toutes les arêtes sont distinctes après la projection). Pour commencer, prenez par exemple (ox, oy, oz) = (3, 4, 2).

Projection perspective dans OpenGL

L'instruction OpenGL suivante calcule une matrice de projection centrale par rapport à l'observateur sur le plan de vue :

glFrustum(umin, umax, vmin, vmax, dmin, dmax) ;

Cette instruction comprend 6 paramètres réels (au lieu de 1, distance du plan de vue à l'observateur, en théorie). Dans OpenGL, on définit un parallélépipède de vue et toute la partie de l'objet inclue dans ce parallélépipède est seulement visible. Ce dernier est défini dans le repère de vue (Ov, U, V, N) par deux bornes en U (umin, umax), deux bornes en V (vmin, vmax) et deux distances (attention des distances et non pas des bornes) des deux plans avant (le plus proche) et arrière (le plus éloigné) à l'observateur. Ces deux plans sont orthogonaux (parallèles à N) à la direction de vue et le plan avant (le plus proche de l'observateur) est le plan de vue (plan de projection de l'image).

De même vous devriez définir ces paramètres pour une vue raisonnable du cube (occupant 3/4 de la fenêtre graphique). Je rappelle que le cadrage réel de la fenêtre graphique sera (umin, umax, vmin, vmax). Une indication pour commencer : umin = vmin = -1.0, umax = vmax = 1.0, dmin = 1.0 et dmax = 100.0 (dmin et dmax sont des distances).

Visualisation du Cube, programme général

Voici enfin la structure générale du code à développer et à élaborer :

```
/* LO13-votre-nom.cpp */
/* Commentaires généraux sur le code */

/* Liste des macros notamment les #include et #define */
….

/* Liste des structures */

typedef struct mesh
```

….

```
/* Liste des variables globales */
/* pour l'instant une variable globale msh pour le cube */

Mesh   msh ;
/* Liste des procédures dans l'ordre :
   Procédure de définition de la géométrie : Cube
   Procédures d'initialisation pour l'instant un seul
   Procédures utiles pour les événements
   Les événements : display pour l'instant
*/

void Cube(void)
{
   …
}

void EffacerEcran(void)
{
  …
}

void ViderMemoireGraphique(void)
{
   glFlush() ;
}

void MatriceProjection(void)
{
  glMatrixMode(GL_PROJECTION) ;
  gloLoadIdentity() ;
  glFrustum( ?, ?, ?, ?, ?, ?) ;
}

void InitialisationEnvironnementOpenGL(void)
{
  /* définition de la couleur du fond */
  …

  /* définition de la couleur courante */
  …

  /* définition de l'instance GL_PROJECTION via une procédure utilisateur */
  MatriceProjection() ;
}

void MatriceModelVue(void)
{
  glMatrixMode(GL_MODELVIEW) ;
```

```
  gloLoadIdentity() ;
  gluLookAt( ?, ?, ?, ?, ?, ?, ?, ?, ?) ;
}

void TraceObjet(void)
{
  int     i, j, s;

  /* Mode de tracé en filaire */
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE) ;

  /* Tracé des triangles */
  /* mode GL_TRIANGLES : trois sommets consécutives sont connectés pour former un
triangle */
  glBegin(GL_TRIANGLES) ;
    for (i=0 ; i<12 ; i++)
    {
      /* groupe des trois sommets du triangles i pour former un triangle */
      for (j=0 ; j<3 ; j++)
      {
        /* s est le numéro du sommet j du triangle i */
        s = msh.triangles[i][j] ;
        glVertex3f(msh.sommets[s][0], msh.sommets[s][1], msh.sommets[s][2]) ;
       }
    }
  glEnd() ;
}

void display(void)
{
   /* effacer l'écran */
    EffacerEcran() ;

   /* Définir l'instance GL_MODELVIEW via une procédure utilisateur */
    MatriceModelVue() ;

   /* Tracé du modèle en mémoire via une procédure utilisateur */
    TraceObjet() ;

    /* Envoie de la mémoire sur l'écran */
     ViderMemoireGraphique() ;
}

int main(int argc, char **argv )
{
   /* Initialiser la librairie graphique */
  …

   /* Définir la géométrie */
   Cube() ;
```

```
  /* Définir la fenêtre graphique *.
  …

  /* initialiser l'environnement OpenGL */
  …

  /* Liste des événements */
  glutDisplayFunc(display) ;

  /* Boucle infinie */
  glutMainLoop() ;

  /* et on n'y passera jamais mais pour la beauté du programme !! */
  return(0) ;
}
```

Ce programme est à compléter et je pense que vous avez tous les éléments pour le faire. Pensez toujours à favoriser les procédures (groupement d'instructions pour réaliser une action précise) car le cœur du programme (le concept algorithmique) peut être écrit en français avec des noms de procédures en français et chaque procédure utilisera des spécificités propres à une bibliothèque graphique et dans notre cas OpenGL. En particulier, inspirez-vous de la procédure display écrit dans cet esprit !

Pour le TD, je vous propose de calculer l'image du cube (à la main) en considérant les paramètres suivants :
Ov(2, 2, 1), F(0, 0, 0), V(0, 0, 1) et d=1 (un paramètre en théorie).

Bon courage