Introduction

Dans le précédent didacticiel, vous avez eu l'occasion d'écrire votre tout premier programme OpenGL. Ce programme était sans doute un peu décevant, car il se contentait d'afficher un carré à l'écran. Aujourd'hui, je vous propose de passer à la vitesse et à la dimension supérieures avec 'LE' classique de la 3D, celui avec lequel ont débuté bon nombre de demomakers : le cube tournant.

Pour être plus précis, nous allons partir du programme que nous avons écrit la dernière fois, et lui ajouter de nouvelles fonctionnalités : nous allons remplacer le carré par un cube et offrir à l'utilisateur la possibilité de faire tourner le cube sur lui−même par l'intermédiaire de la souris. Ce programme à pour but d'aborder :

- les méthodes de stockage des scènes tridimensionnelles,
- les grands principes de l'animation avec OpenGL,
- de nouvelles fonctions de rappel (pour la gestion de la souris et des redimensionnements).

Le cube : stockage :

La dernière fois, nous n'avions pas rencontré de problème concernant le stockage de la scène, puisqu'elle ne contenait que 4 sommets. La figure 1 représente la scène que nous souhaitons afficher aujourd'hui.

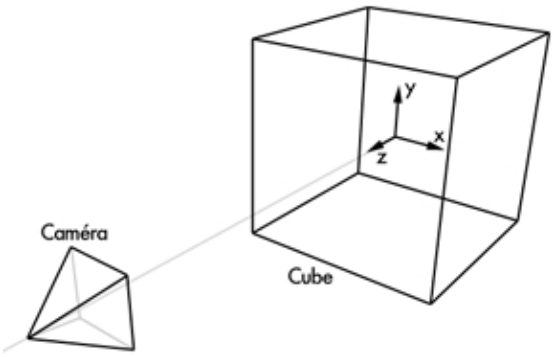

Figure 1 : La scène 3D

Si on fait le compte, notre cube comporte 6 faces et 8 sommets. La scène est encore assez simple (si on la compare aux dizaines de milliers de sommets que comportent généralement les scènes crées par les professionnels de l'infographie), et on pourrait tout à fait se contenter, comme nous l'avons fait dans le précedent didacticiel, de décrire linéairement chaque polygone. Nous allons cependant mettre en place une structure de données pour stocker notre scène. OpenGL fournit des fonctions permettant la mise en place de structures de stockage, mais nous allons voir qu'il est tout à fait possible de le faire soi−même.

Première chose, nous allons définir un type de données pour stocker chaque point : il s'agit d'une structure composée de 6 nombres de type flottant qui contient les coordonnées cartésiennes du point (x, y et z), ainsi que la couleur du sommet (r, g et b).

```
typedef struct
```

```
{
  float x;
  float y;
  float z;
  float r;
  float g;
  float b;
} point;
```

Pour stocker tous les sommets de la scène, on définit un tableau de points, et on en profite pour le remplir avec les coordonnées des points de notre cube et leurs couleurs :

```
point p[8]={
    {-0.5,-0.5, 0.5,1.0,0.0,0.0},
    {-0.5, 0.5, 0.5,0.0,1.0,0.0},
    { 0.5, 0.5, 0.5,0.0,0.0,1.0},
    { 0.5,-0.5, 0.5,1.0,1.0,1.0},
    {-0.5,-0.5,-0.5,1.0,0.0,0.0},
    {-0.5, 0.5,-0.5,0.0,1.0,0.0},
    { 0.5, 0.5,-0.5,0.0,0.0,1.0},
    { 0.5,-0.5,-0.5,1.0,1.0,1.0}};
```

Reste maintenant à trouver une structure permettant de stocker les 6 faces de notre cube. Sachant qu'une face comporte exactement 4 sommets, nous pouvons utiliser un tableau à 2 dimensions contenant les indices (dans le tableau p que nous venons de créer) de chacun des points de la face :

```
int f[6][4]={
  {0,1,2,3},
  {3,2,6,7},
  {4,5,6,7},
  {0,1,5,4},
  {1,5,6,2},
  {0,4,7,3}};
```

Ce tableau peut à première vue paraître obscur, aussi voici de quoi éclairer votre lanterne : f[i][j] contient l'indice du j−ième sommet de la face numéro i. Ainsi, la face numéro 4 est définie par les sommet p[1], p[5], p[6] et p[2] (je vous rappelle qu'en C, les indices de tableaux commencent à 0, et non à 1).

Le cube : affichage

Voyons maintenant comment utiliser cette structure de données lors de l'affichage de la scène, c'est−à−dire dans notre fonction affichage(). La dernière fois, nous avons vu que la description d'un polygone se fait par énumération des sommets (avec la fonction glVertex()), le tout encadré par un glBegin() et un glEnd(). Avec notre structure de données, nous allons utiliser 2 boucles for imbriquées. La première permet de parcourir chacune des faces, la seconde permet d'énumérer les 4 points de chaque face :

```
  for (i=0;i<6;i++)
    {
      glBegin(GL_POLYGON);
      for (j=0;j<4;j++)
        {
          glColor3f(p[f[i][j]].r,p[f[i][j]].g,p[f[i][j]].b);
          glVertex3f(p[f[i][j]].x,p[f[i][j]].y,p[f[i][j]].z);
        }
      glEnd();
    }
```

Si vous avez du mal à comprendre la portion de code ci−dessus, je vous conseille d'essayer de simuler 'à la main' sur une feuille de papier le fonctionnement des boucles. Vous verrez que finalement, c'est moins compliqué qu'il n'y parait :).

Animation

Nous souhaitons donner du mouvement à notre cube grâce à la souris. Aussi, il faut essayer de répondre à la question "Comment faire tourner notre cube ?". Sachant que le sujet des transformations d'objets (translation, rotation, mise à l'échelle) est vaste, je vais pour aujourdhui vous 'balancer' une portion de code sans donner d'explications poussées. Dès la prochaine fois, nous étudierons en détail la question.

```
  glLoadIdentity();
  glRotatef(-angley,1.0,0.0,0.0);
  glRotatef(-anglex,0.0,1.0,0.0);
```

Ces lignes sont à placer dans la fonction affichage(), juste avant la description des faces. Les transformations d'objets sont stockées dans une matrice. La fonction GlLoadIdentity(angle,x,y,z) permet de réinitialiser la matrice de transformation, glRotatef() ajoute à la matrice de transformation une rotation dont l'axe passe par l'origine et est porté par le vecteur (x,y,z). Ici, nous avons défini une rotation d'angle '-angley' autour de l'axe x, et une rotation d'angle '-anglex' autour de l'axe y. La notation utilisée peut vous paraître bizarre, mais elle est dictée par la souris : un déplacement horizontal (suivant x) de la souris correspond intuitivement à une rotation du cube autour de l'axe y. Il est important de noter que les rotations sont appliquées dans le sens inverse de leur description. Ainsi, nous avons défini ci-dessus une rotation autour de l'axe y suivie d'une rotation autour de x (dont le résultat est en général différent d'une rotation autour de x suivie d'une rotation autour de y).

Il nous faut maintenant déterminer les valeurs des variables anglex et angley en fonction de la position de la souris. Le comportement souhaité est le suivant : pour faire tourner le cube, l'utilisateur doit cliquer sur la fenêtre OpenGL, maintenir le bouton enfoncé et déplacer la souris. Nous allons utiliser deux nouvelles fonctions de rappel offertes par glut.

Les fonctions de rappel liées à la souris :

Glut permet de définir une fonction de rappel pour les boutons de la souris. On met en place le rappel avec glutMouseFunc(). La fonction enregistrée sera appelée à chaque fois que l'utilisateur appuiera sur un des boutons de la souris ou le relâchera. Enregistrons une fonction appelée mouse() :

```
  glutMouseFunc(mouse);
```

Le prototype de la fonction mouse() doit être le suivant :

```
void mouse(int bouton,int etat,int x,int y)
```

Lorsque la fonction est lancée par le gestionnaire d'événements de glut, 'bouton' contient le nom du bouton qui a été pressé ou relâché (GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON), 'etat' indique si le bouton a été pressé (GLUT DOWN) ou relaché (GLUT_UP) et 'x' et 'y' contient les coordonnées de la souris au moment de l'événement. Nous souhaitons que le mouvement de la souris ne soit appliqué au cube que lorsque le bouton gauche est enfoncé. Nous allons mettre en place une variable globale 'pressé' (de type char) qui vaudra 1 lorsque le bouton est pressé, et 0 lorsque le bouton est relâché. Nous allons également introduire deux entiers x_old et y_old (défini comme variable globale afin qu'ils soient accessibles depuis l'autre fonction de rappel pour la souris). Ils vont servir à stocker la position de la souris lors de l'appui sur le bouton de gauche.

```
void mouse(int button, int state,int x,int y)
{
  /* si on appuie sur le bouton gauche */
  if (button == GLUT_LEFT_BUTTON &state == GLUT_DOWN)
  {
    presse = 1; /* le booleen presse passe a 1 (vrai) */
    xold = x; /* on sauvegarde la position de la souris */
    yold=y;
  }
  /* si on relache le bouton gauche */
  if (button == GLUT_LEFT_BUTTON &state == GLUT_UP)
    presse=0; /* le booleen presse passe a 0 (faux) */
}
```

Le second rappel lié à la souris génère un événement lorsque la souris est déplacée alors qu'au moins un des boutons est pressé. L'enregistrement de la fonction de rappel se fait avec glutMotionFunc(), et le prototype de la fonction

enregistrée doit être le suivant :

```
void mousemotion(int x,int y)
```

'x' et 'y' sont les coordonnées de la souris par rapport au coin supérieur gauche de la fenêtre au moment de la génération de l'événement. Notre fonction mousemotion a pour but de mettre à jour les valeurs de 'anglex' et 'angley' en fonction du déplacement relatif de la souris si le bouton gauche est enfoncé (c'est−à−dire si 'presse' vaut 1) :

```
void mousemotion(int x,int y)
  {
    if (presse) /* si le bouton gauche est presse */
    {
      /* on modifie les angles de rotation de l'objet
         en fonction de la position actuelle de la souris et de la derniere
         position sauvegardee */
      anglex=anglex+(x−xold);
      angley=angley+(y−yold);
      glutPostRedisplay(); /* on demande un rafraichissement de l'affichage */
    }

    xold=x; /* sauvegarde des valeurs courante de le position de la souris */
    yold=y;
  }
```

Le tampon de profondeur :

Le passage du carré 2D au cube 3D fait surgir un nouveau problème : celui de l'ordre d'affichage des polygones de la scène. La dernière fois, la scène ne comportait qu'un polygone et donc la question ne se posait pas. Aujourd'hui, avec nos 6 faces, il est important d'ordonner correctement l'affichage. Pour vous en convaincre, prenez une feuille de papier, et 6 crayons de couleurs. L'objectif est de dessiner un cube vu de coté dont chaque face sera remplie avec une couleur. Afin de simuler le comportement (relativement stupide) d'un ordinateur, je vais vous demander de colorier chacune des faces du polygones, même celles qui seront entièrement masquées par d'autres faces. Vous comprenez bien qu'il faut choisir judicieusement l'ordre dans lequel vous devez colorier les faces pour obtenir un résultat correct : il faut commencer par colorier les faces qui sont les plus éloignées du point de vue.

Pour résoudre ce problème de faces cachées, OpenGL propose une technique extrêmement répandue en synthèse d'images : le tampon de profondeur, plus connu sous son nom anglais : Z−buffer. Le tampon de profondeur est une technique simple et très puissante. L'idée principale est de créer en plus de notre image un tampon de même taille. Ce tampon va servir à stocker pour chaque pixel une profondeur, c'est−a−dire la distance entre le point de vue et l'objet auquel appartient le pixel considéré. A l'origine, le tampon est rempli avec une valeur dite "de profondeur maximale" : l'image est vide. A chaque fois qu'on dessine un polygone, pour chaque pixel qui le constitue, on calcule sa profondeur et on la compare avec celle qui est déjà stockée dans le tampon. Si la profondeur stockée dans le tampon est supérieure à celle du polygone qu'on est en train de traiter, alors le polygone est plus proche (au point considéré) de la caméra que les objets qui ont déjà été affichés. Le point du polygone est affiché sur l'image, et la profondeur du pixel est stockée dans le tampon de profondeur. Dans le cas inverse, le point que l'on cherche à afficher est masqué par un autre point déjà placé dans l'image, donc on ne l'affiche pas. Vous noterez qu'avec cette technique, on peut afficher les polygones dans un ordre quelconque.

Nous allons mettre en place un tampon de profondeur pour notre programme. Première chose à faire, il faut modifier l'appel à la fonction glut d'initialisation de l'affichage pour lui indiquer qu'il faut allouer de l'espace pour le tampon de profondeur :

```
glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
```

L'utilisation ou non du tampon de profondeur est gérée par une variable d'etat OpenGL : GL_DEPTH_TEST. L'activation du tampon de profondeur se fait en plaçant la ligne suivante dans la phase d'initialisation du programme :

```
glEnable(GL_DEPTH_TEST);
```

Tout comme le tampon image, il faut effacer le tampon de profondeur à chaque fois que la scène est redessinée. Il suffit de modifier l'appel glClear() de la fonction d'affichage :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Afin de vous permettre de visualiser l'intérêt du tampon de profondeur, je vous propose de donner à l'utilisateur la possibilité d'activer et de désactiver le tampon de profondeur avec les combinaisons de touches 'd' et 'Shift'+'d'. Pour cela, ajoutons ces quelques lignes a notre fonction de rappel clavier() :

```
case 'd':
  glEnable(GL_DEPTH_TEST);
  glutPostRedisplay();
  break;
case 'D':
  glDisable(GL_DEPTH_TEST);
  glutPostRedisplay();
  break;
```

Le double tampon :

Si vous compilez et exécutez le programme tel que nous l'avons modifié jusqu'à présent, vous vous rendrez compte qu'un problème subsiste. On peut voir les polygones se dessiner les un après les autres. Evidemment, moins votre configuration sera puissante, plus l'effet sera marqué. Pour remédier à cet effet désagréable, nous allons mettre en place un système de double tampon d'image (double−buffering). Pour comprendre ce système, raisonnons par analogie : imaginons une personne en train de faire un exposé en utilisant comme support un rétroprojecteur. Si la personne n'a qu'un transparent, elle va dessiner sur le celui−ci alors qu'il est posé sur le rétroprojecteur éclairé. Le public va voir sur l'écran le crayon effectuer le tracé. Si en revanche l'orateur dispose de deux transparents, il peut projeter un transparent, dessiner le transparent suivant sur une table, échanger les deux transparents, effacer celui qu'il vient d'ôter du rétroprojecteur puis dessiner le prochain transparent, et ainsi de suite. De cette manière, le public a en permanence sous les yeux un transparent complet. La technique du double tampon consiste à dessiner une image dans un tampon mémoire pendant que la précédente est affichée à l'écran, puis à 'échanger' les deux images.

Glut permet de gérer un double tampon de façon extrêmement simple. Tout d'abord, comme pour le tampon de profondeur, il faut indiquer au système que l'on souhaite utiliser un double tampon d'image. Nous modifions donc à nouveau l'appel à glutInitDisplayMode() :

```
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
```

Il ne reste plus qu'à indiquer à glut le moment opportun pour l'échange des buffers : la fin du dessin de la scène. On place à cet effet un appel à glutSwapBuffers() à la fin de notre fonction d'affichage. Il est difficile de faire plus simple, n'est−ce pas ?.

La fonction de rappel 'redimensionnement' :

J'ai évoqué la dernière fois un problème qui se pose lorsque qu'on redimensionne la fenêtre OpenGL. Par défaut, la scène OpenGL occupe toute la fenêtre, et la scène représente tous les points dont les coordonnées en x et y sont comprises entre −1 et 1. Si on redimensionne la fenêtre et que le nouveau rapport hauteur/largeur de la fenêtre ne vaut pas 1, l'image subit une déformation. Une technique pour éviter ceci consiste à limiter la zone utilisée pour le dessin à la plus grande sous partie carrée de la fenêtre OpenGL, qu'on prendra soin de centrer. Si on redimensionne la fenêtre à une taille 'w' par 'h', la plus grande sous partie carré de la fenêtre mesure w pixels si w<h et h pixels si w>h.

glut propose une fonction de rappel pour les redimensionnements de la fenêtre. Cette fonction doit être enregistrée grâce à glutReshapeFunc(). Voici la fonction de rappel que je vous propose :

```
void reshape(int x,int y)
{
  if (x<y)
    glViewport(0,(y−x)/2,x,x);
  else
    glViewport((x−y)/2,0,y,y);
}
```

Lors de l'appel à reshape(), 'x' contient la nouvelle largeur de la fenêtre, et 'y' la nouvelle hauteur. La fonction glViewport() permet de limiter la zone de dessin à une portion de la fenêtre. Son prototype est :

```
void glViewport(GLint x,GLint y,GLSizei largeur,GLSizei hauteur)
```

'x' et 'y' sont les coordonnées du coin supérieur gauche de la sous−fenêtre. Je vous laisse le soin de deviner ce que sont les paramètres 'largeur' et 'hauteur'.

Conclusion :

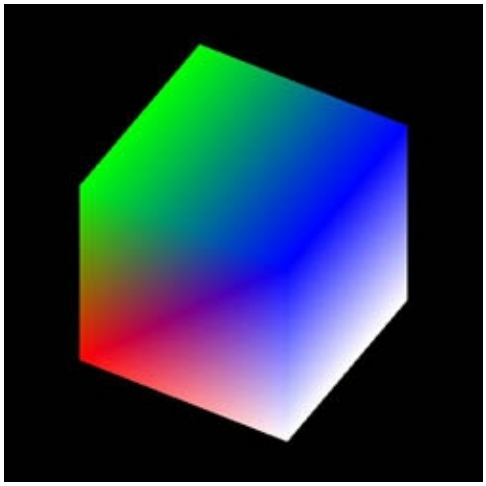

Figure 2 : notre cube en 3D

Le programme que nous venons d'obtenir donne un résultat qui reste perfectible. Vous remarquerez peut être que le cube peut sembler quelque peu déformé lorsqu'on le fait tourner. Cela vient du fait que pour l'instant, nous utilisons une projection dite orthogonale, qui ne prend pas en compte les effets de perspective. Nous aborderons ces questions de perspective dans un prochain didacticiel. Si vous souhaitez expérimenter par vous−même, je vous propose l'exercice suivant : modifier le programme de façon à ce que chaque face soit de couleur unie.

Références :

OpenGL 1.2	Woo, Neider, Davis et Shreiner − Campus Press Référence La traduction française de la dernière édition du livre de référence en matière de programmation OpenGL
www.opengl.org	Le site officiel d'OpenGL. Tout y est : présentation, documents de spécification, liens vers des didacticiels, bibliographie
www.mesa3d.org	le site de Mesa, l'implémentation libre d'OpenGL la plus utilisée sous Linux
reality.sgi.com/mjk/glut3	la page de glut. Vous y trouverez le manuel de référence glut

Code source:

```
/*********************************************************/
/*                      cube.c                           */
/*********************************************************/
/* Affiche a l'ecran un cube en 3D                       */
/*********************************************************/

/* inclusion des fichiers d'en−tete Glut */

#include <GL/glut.h>

/* Notre structure point */
```

```
typedef struct
{
  float x;
  float y;
  float z;
  float r;
  float g;
  float b;
} point;

point p[8]={
    {-0.5,-0.5, 0.5,1.0,0.0,0.0},
    {-0.5, 0.5, 0.5,0.0,1.0,0.0},
    { 0.5, 0.5, 0.5,0.0,0.0,1.0},
    { 0.5,-0.5, 0.5,1.0,1.0,1.0},
    {-0.5,-0.5,-0.5,1.0,0.0,0.0},
    {-0.5, 0.5,-0.5,0.0,1.0,0.0},
    { 0.5, 0.5,-0.5,0.0,0.0,1.0},
    { 0.5,-0.5,-0.5,1.0,1.0,1.0}};

int f[6][4]={
  {0,1,2,3},
  {3,2,6,7},
  {4,5,6,7},
  {0,1,5,4},
  {1,5,6,2},
  {0,4,7,3}};

char presse;
int anglex,angley,x,y,xold,yold;

/* Prototype des fonctions */
void affichage();
void clavier(unsigned char touche,int x,int y);
void reshape(int x,int y);
void idle();
void mouse(int bouton,int etat,int x,int y);
void mousemotion(int x,int y);

int main(int argc,char **argv)
{
  /* initialisation de glut et creation
     de la fenetre */
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  glutInitWindowPosition(200,200);
  glutInitWindowSize(250,250);
  glutCreateWindow("cube");

  /* Initialisation d'OpenGL */
  glClearColor(0.0,0.0,0.0,0.0);
  glColor3f(1.0,1.0,1.0);
  glPointSize(2.0);
  glEnable(GL_DEPTH_TEST);

  /* enregistrement des fonctions de rappel */
  glutDisplayFunc(affichage);
  glutKeyboardFunc(clavier);
  glutReshapeFunc(reshape);
  glutMouseFunc(mouse);
  glutMotionFunc(mousemotion);

  /* Entree dans la boucle principale glut */
  glutMainLoop();
  return 0;
}
```

```
void affichage()
{
  int i,j;
  /* effacement de l'image avec la couleur de fond */
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glLoadIdentity();
  glRotatef(-angley,1.0,0.0,0.0);
  glRotatef(-anglex,0.0,1.0,0.0);
  /* Dessin du cube */
  for (i=0;i<6;i++)
    {
      glBegin(GL_POLYGON);
      for (j=0;j<4;j++)
        {
          glColor3f(p[f[i][j]].r,p[f[i][j]].g,p[f[i][j]].b);
          glVertex3f(p[f[i][j]].x,p[f[i][j]].y,p[f[i][j]].z);
        }
      glEnd();
    }
  glFlush();

  /* On echange les buffers */
  glutSwapBuffers();
}

void clavier(unsigned char touche,int x,int y)
{
  switch (touche)
    {
    case 'p': /* affichage du carre plein */
      glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
      glutPostRedisplay();
      break;
    case 'f': /* affichage en mode fil de fer */
      glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
      glutPostRedisplay();
      break;
    case 's' : /* Affichage en mode sommets seuls */
      glPolygonMode(GL_FRONT_AND_BACK,GL_POINT);
      glutPostRedisplay();
      break;
    case 'd':
      glEnable(GL_DEPTH_TEST);
      glutPostRedisplay();
      break;
    case 'D':
      glDisable(GL_DEPTH_TEST);
      glutPostRedisplay();
      break;
    case 'q' : /*la touche 'q' permet de quitter le programme */
      exit(0);
    }
}

void reshape(int x,int y)
{
  if (x<y)
    glViewport(0,(y-x)/2,x,x);
  else
    glViewport((x-y)/2,0,y,y);
}

void mouse(int button, int state,int x,int y)
{
  /* si on appuie sur le bouton gauche */
  if (button == GLUT_LEFT_BUTTON &state == GLUT_DOWN)
```

```
  {
    presse = 1; /* le booleen presse passe a 1 (vrai) */
    xold = x; /* on sauvegarde la position de la souris */
    yold=y;
  }
  /* si on relache le bouton gauche */
  if (button == GLUT_LEFT_BUTTON &state == GLUT_UP)
    presse=0; /* le booleen presse passe a 0 (faux) */
}

void mousemotion(int x,int y)
  {
    if (presse) /* si le bouton gauche est presse */
    {
      /* on modifie les angles de rotation de l'objet
         en fonction de la position actuelle de la souris et de la derniere
         position sauvegardee */
      anglex=anglex+(x-xold);
      angley=angley+(y-yold);
      glutPostRedisplay(); /* on demande un rafraichissement de l'affichage */
    }

    xold=x; /* sauvegarde des valeurs courante de le position de la souris */
    yold=y;
  }
```