TP

3

TP OpenGL

Nicholas Journet - TP OpenGL - IUT - [1]

Objectifs du TP

- Manipulation de primitives de base
- Interaction avec périphériques
- Transformations géométriques simples
- Gestion des tampons d'affichage

3.1 Affichage d'une primitive 2D

3.1.1 Quelques précisions

Les fonction OpenGL sont régies par un certain nombre de règles :

- Tous les noms de fonctions OpenGL sont préfixés : les noms de fonction OpenGL commencent par 'gl'. Pour les bibliothèques annexes telles que GLU et GLX, les préfixes respectifs sont 'glu' et 'glX'.
- Certaines commandes OpenGL existent en plusieurs variantes, suivant le nombre et le type de paramètres qu'admet la variante. Ainsi, la fonction glVertex3f() prend trois paramètres ('3') de type Float ('f'), alors que glVertex2i() fonctionne avec 2 entiers. Certaines fonctions possèdent une version suffixée par un 'v'. Les paramètres sont passés à ces fonctions par l'intermédiaire d'un pointeur sur un tableau.
- OpenGL redéfinit des types de données numériques : GLint correspond aux entiers, GLfloat aux flottants, GLbyte aux caractères non signés... Il est préférable d'utiliser ces types de données plutôt que les types standards du C.

3.1.2 Initialisation de glut et création de la fenêtre

La bibliothèque OpenGL est conçue comme une machine à états. Par conséquent, une des premières choses à faire est d'initialiser cette machine. Le nombre de variables d'état accessibles est assez impressionnant. Le mode de dessin, la couleur de fond et le type d'éclairage en sont des exemples.
L'initialisation de glut se fait de la manière suivante :

```
1 #include <GL/glut.h>    // fichiers d'entête OpenGL
2 /* Initialise la bibliothèque glut (communication avec serveur X) */
3 glutInit(&argc, argv);
4 glutInitDisplayMode(GLUT_RGB);  // paramètres d'affichage
5 glutInitWindowPosition(200, 200);
6 glutInitWindowSize(250, 250);
7 glutCreateWindow("ogl1");
```

OpenGL dispose d'un état par défaut, et nous nous contenterons donc dans la phase d'initialisation de régler 2 états : la couleur de fond, et la taille d'un point. Les fonctions à utiliser sont les suivantes :

```
1 void glClearColor(GLclampf rouge, GLclampf vert, GLclampf bleu, GLclampf alpha);
2 void glPointSize(GLfloat taille);
```

1. Ce tp est issu des notes de cours de Xavier Michelon - linuxorg. Toute modification de ce support de cours doit y faire référence

`glClearColor()` permet de spécifier la couleur de remplissage utilisée lors d'un effacement de la scène. On peut donc l'assimiler à la couleur du fond. Le mode de spécification de couleur est le classique RGB avec une composante supplémentaire alpha, utilisée pour la gestion de la transparence des objets. Nous ne l'utiliserons pas ici et lui affecterons systématiquement la valeur 0. Chacune des 4 composantes doit être comprise entre 0 et 1. Si vous avez l'habitude de travailler avec des entiers compris entre 0 et 255, une simple division de chaque composante par 255 vous permettra de trouver la bonne valeur. Ainsi `glClearColor(1.0,1.0,1.0,0.0)` nous donne un fond blanc, et `glClearColor(0.0,0.0,0.0,0.0)` un fond noir.
Par défaut, lorsqu'on choisit de représenter les objets simplement par leurs sommets, la taille des sommets à l'écran est de 1 pixel. Afin de les rendre plus visible, nous réglerons la taille des sommets à 2 pixels :

```
1 glPointSize(2.0);
```

Le système de gestion des événements offert par glut est relativement similaire à celui de GTK. Des fonctions sont associées aux différents types d'événements envoyés par le serveur X, puis une boucle d'attente est lancée. A chaque fois qu'un événement est émis par le serveur X, la fonction de rappel associée à l'événement est appelée. Vous devez associer au moins une fonction de rappel, la fonction d'affichage, qui est celle dans laquelle vous devez décrire votre scène 3D. Vous pouvez associer des fonctions de rappel aux événements liés :

- au clavier
- à des périphériques d'entrée répandus en infographie (spaceballs, boîtes de potentiomètres, tablettes graphiques...)
- à la souris

Il existe par ailleurs deux fonctions de rappel spéciales qui permettent d'enregistrer des fonctions qui seront appelées à intervalles de temps réguliers ou pendant les temps d'oisiveté (lorsque le gestionnaire n'a aucun événement à traiter). Dans notre programme, en plus de la fonction d'affichage obligatoire, nous allons mettre en place une fonction de rappel pour le clavier. Les deux appels suivants permettent de spécifier quelles seront les fonctions respectivement associées à l'affichage et au clavier :

```
1 /* affichage est le nom de la fonction callback associée à l'affichage */
2 glutDisplayFunc(affichage);
3 /* clavier est le nom de la fonction callback associée à un événement sur un périphérique */
4 glutKeyboardFunc(clavier);
```

Après cela, il ne nous reste plus qu'à lancer la boucle d'attente des événements :

```
1 glutMainLoop();
```

Question 1
Recopiez le code précédement présenté dans un fichier `.c`. Les fonctions `clavier` et `affichage` ont la signature suivante :

- `void clavier(unsigned char touche, int x, int y)`
- `void affichage()`

Compilez ce code et observez que la fenêtre est bien créée.
La commande de compilation est la suivante : `gcc -std=c99 Q1.c -lGL -lglut -o Q1`

3.1.3 Gestion de l'affichage

La fonction d'affichage constitue le cœur du programme. C'est ici que nous allons décrire la scène. Pour être plus exact, la fonction d'affichage contient la procédure à appliquer pour redessiner notre scène dans la fenêtre, en commençant par un remplissage de la fenêtre avec la couleur de fond que nous avons définie dans la phase d'initialisation. On utilise pour cela

```
1 void glClear(GLbitfield masque);
```

OpenGL travaille avec plusieurs zones tampons en plus de la zone d'image (tampon de profondeur, d'accumulation...). Le paramètre masque permet de spécifier les tampons que l'on souhaite effacer. Nous n'utilisons, pour le moment, que le tampon d'image. Notre masque vaudra `GL_COLOR_BUFFER_BIT`. Il nous faut ensuite décrire la scène 3D. La méthode de représentation d'une scène d'OpenGL est la plus classique qui soit :

➤ une scène est constituée d'objets.

➤ Un objet est défini par un ensemble de polygones.

➤ Un polygone est un ensemble de points de l'espace reliés entre eux par des arêtes.

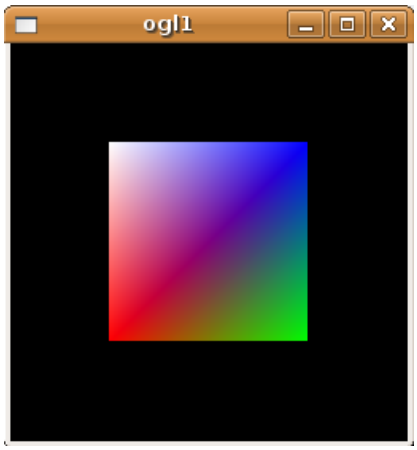
La méthode est la suivante : on indique le début de la description du polygone, on explicite chaque point du polygone, puis on indique la fin de l'énumération. A chaque déclaration de points, on peut modifier certaines variables d'état, comme la couleur active. Notre scène ne comportant qu'un polygone, la description sera vite faite.

```
glBegin(GL_POLYGON); // début description de polygones
glColor3f(1.0,0.0,0.0); // modifie la couleur active en rouge
/* Ici on dessine un polygone en 2D, il n'y a donc besoin que de deux coordonnées */
glVertex2f(-0.5,-0.5); // primitive de spécification d'un sommet du polygone
glColor3f(0.0,1.0,0.0);
glVertex2f(0.5,-0.5);
glColor3f(0.0,0.0,1.0);
glVertex2f(0.5,0.5);
glColor3f(1.0,1.0,1.0);
glVertex2f(-0.5,0.5); // par défaut le dernier point est relié au premier
glEnd(); // fin description
glFlush(); // forcer l'affichage du résultat
```

Question 2

En associant à chaque point de notre carré une couleur différente, et compte tenu du mode de remplissage de polygone proposé par défaut, notre carré sera rempli en interpolant les couleurs des sommets. En termes simples, on obtient un dégradé.

Recopiez le code qui vous est donné afin de remplir la fonction `affichage`. L'exécution de ce programme doit vous permettre d'afficher le carré suivant.



3.1.4 Gestion du clavier

La dernière étape de la création de notre programme va consister à écrire le contenu de la fonction `void clavier(unsigned char touche, int x, int y);`.

Lorsque l'utilisateur appuiera sur une touche du clavier, la boucle de gestion effectuera un appel à `clavier()`. La variable touche contiendra le caractère correspondant à la touche pressée, x et y indiqueront la position de la souris lors de la frappe sur le clavier. x et y sont données relativement à la fenêtre OpenGL. Dans la plupart des cas, la fonction de rappel pour les événements de type clavier est constituée d'une structure switch dont la clause de test porte sur le caractère passé en paramètre. Voici notre fonction `clavier`

```
void clavier(unsigned char touche, int x, int y) {
    switch (touche) {
      case 'p': /* Affichage du carré plein */
        /* modifier l'état de la machine OpenGL */
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        glutPostRedisplay(); // on redessine la fenêtre
        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 'q' : /* la touche 'q' permet de quitter le programme */
        exit(0);
  }
}
```

Chaque polygone possède une face avant et une face arrière définie par l'ordre de saisie des sommets du polygone. Il est possible d'associer un mode de dessin différent pour la face avant et arrière d'un polygone. Le paramètre face indique la face dont on veut changer le mode de dessin :

- GL_FRONT : face avant
- GL_BACK : face arrière
- GL_FRONT_AND_BACK : les deux faces

Le paramètre mode définit le mode de dessin souhaité pour la ou les faces choisies :

- GL_FILL : le polygone est entièrement rempli.
- GL_LINE : seuls les contours du polygone sont dessinés (mode fil de fer)
- GL_POINT : seuls les sommets du polygone sont affichés

Question 3
Recopiez le code précédent, compilez, exécutez.

3.2 Affichage d'une primitive 3D

Dans cette partie nous allons transformer le carré en cube et ajouter des fonctions permettant de le faire bouger à la souris.

3.2.1 Les méthodes de stockage des scènes

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 créées par les professionnels de l'infographie), et on pourrait tout à fait se contenter, comme précédemment, 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.
Nous allons donc 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}};
```

$f[i][j]$ contient l'indice du j^{ime} sommet de la face numéro i. Ainsi, la face numéro 4 est définie par les sommet $p[1]$, $p[5]$, $p[6]$ et $p[2]$.
Il ne reste plus qu'a afficher le cube. pour cela il suffit de recopier le code suivant dans la fonction d'affichage.

```
for (int i=0; i<6; i++) // 6 faces
{
  glBegin(GL_POLYGON);

  for (int j=0; j<4; j++) // 4 points
  {
    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();
}
```

3.2.2 Animation OpenGL

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 ?". Pour cela, il faut vous référer à la partie du cours parlant des transformations géométriques. Nous les avons vues en 2D. Le principe reste exactement le même en 3D.
OpenGL permet de manipuler une matrice de transformation (combiner des translations, homothéties et rotations) grâce aux coordonnées homogènes. Lors du (ré)affichage de la fenêtre, la matrice de transformation est appliquée à la forme dessinée.

```
glLoadIdentity(); // réinitialisation de la matrice de transformation
glRotatef(-angley,1.0,0.0,0.0); // rotation d'un angle -angley sur l'axe des x
glRotatef(-anglex,0.0,1.0,0.0); // rotation d'un angle -anglex sur l'axe des y
```

Il n'y a pas d'erreur sur le nom des variables. Un déplacement horizontal (suivant x) de la souris correspond intuitivement à une rotation du cube autour de l'axe y.

Question 4
Vous pouvez maintenant recopier le code précédent dans la fonction d'affichage et faire en sorte d'afficher un cube en perspective 3D.

3.2.3 Fonction de rappel souris pour rotation

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

```
glutMouseFunc(mouse);
```

Le prototype de la fonction mouse() doit être le suivant : `void mouse(int bouton,int etat,int x,int y)`

```
/* button = (GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON)
contient le nom du bouton ayant généré l'interruption
state = (GLUT DOWN) ou (GLUT_UP)
permet de savoir si le bouton est pressé ou relâché */

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 booléen presse passe a 1 (vrai)
    xold = x;  // on sauvegarde la position de la souris
    yold = y;
  }

  /* si on  le bouton gauche */
  if (button == GLUT_LEFT_BUTTON && state == GLUT_UP)
    presse = 0; // le booléen 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 pressé */
    {
      /* on modifie les angles de rotation de l'objet en fonction de la position actuelle
    de la souris et de la dernière position sauvegardée */
      anglex += (x-xold);
      angley += (y-yold);
      glutPostRedisplay(); // on demande un rafraîchissement de l'affichage
    }

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

Question 5
Ecrivez ce code, compilez, exécutez.

Tout comme pour la rotation, il est possible d'opérer des translation et des homothéties :

➤ `void glTranslatef(float x,float y,float z)` : Cette fonction (et sa variante glTranslated() dont les paramètres sont de type 'double') multiplie (à droite) la matrice active par une matrice de translation de vecteur (x,y,z).

➤ `void glScalef(float hx,float hy,float hz);` Cette fonction multiplie la matrice active par une matrice d'homothétie dont les facteurs suivant les axes X, Y, et Z sont respectivement hx, hy et hz.

Si vous voulez combiner les transformations géométriques, n'oubliez pas de tenir compte tenu du fait qu'en OpenGL, les opérations se décrivent dans l'ordre inverse de leur application.

3.2.4 Fonction de rappel pour redimensionnement

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 $w>h$. Voici la fonction `reshape` permettant de de modifier la taille de la forme tout en tenant compte du problème de dimension de la fenêtre.

```
1 void reshape(int x, int y)
2 {
3   if (x<y)
4     glViewport(0,(y-x)/2,x,x);
5   else
6     glViewport((x-y)/2,0,y,y);
7 }
```

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 :

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

et x et y sont les coordonnées du coin supérieur gauche de la sous-fenêtre.

Question 6
Ecrivez ce code, compilez, exécutez et vérifiez que lorsque l'on redimensionne la fenêtre, les proportions restent les mêmes. Le $(0, 0)$ de la fenêtre est il réellement dans le coin de la fenêtre ?

3.3 Gestion de la profondeur de l'affichage

3.3.1 Simple tampon

Le passage du carré 2D au cube 3D fait surgir un nouveau problème : celui de l'ordre d'affichage des polygones de la scène. Avec nos 6 faces, il est important d'ordonner correctement l'affichage.

Pour résoudre ce problème de faces cachées, OpenGL propose une technique extrêmement répandue en synthèse d'images : le tampon de profondeur (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. À l'origine, le tampon est rempli avec une valeur dite "de profondeur maximale". 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 :

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

L'utilisation ou non du tampon de profondeur est gérée par une variable d'état 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, donnez à l'utilisateur la possibilité d'activer et de désactiver le tampon de profondeur avec les combinaisons de touches 'd' et 'D'. Pour cela, ajoutez ces quelques lignes à notre fonction de rappel `clavier()`

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

3.3.2 Double tampon

Si vous compilez et exécutez le programme tel quel, vous vous rendrez compte qu'un problème subsiste. On peut voir les polygones se dessiner les un après les autres. Évidemment, moins votre configuration sera puissante, plus l'effet sera marqué.
Pour remédier à cet effet désagréable, vous allez utiliser 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 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.