OPENGL 2

TP OpenGL : Transformation de visualisation

Nicholas Journet - TP OpenGL - IUT - [1]

Objectifs du TP

➤ Transformation de visualisation

2.1 initialisation de la caméra

Au cours du TP1, lorsque nous avons abordé les transformations géométriques, nous avons déplacé et fait tourner des objets. Mais nous n'avons jamais parlé de l'observateur (la position et l'orientation de la caméra virtuelle). Il existe différentes manières d'aborder la question des transformations de visualisation. Nous allons étudier la plus simple : un déplacement de l'observateur de x unités dans une direction donnée équivaut à un déplacement de tous les objets de la scène de $-x$ unités dans cette direction. De même, faire tourner la caméra de d degrés produit le même résultat qu'une rotation d'angle $-d$ appliquée à tous les objets de la scène. Finalement, on se rend compte que les transformations de visualisation ne sont rien d'autre que des transformations de modélisation.
La bibliothèque GLU permet de faire cela très facilement.

```
1 void gluLookAt(GLdouble camx,GLdouble camy,GLdouble camz, GLdouble
     centrex,GLdouble  centrey,
2  GLdouble centrez,GLdouble hautx,GLdouble hauty,GLdouble hautz);
```

`gluLookAt()` simplifie la spécification de transformations de visualisation. La caméra est positionnée en $(camx,camy,camz)$, elle est dirigée vers le point $(centrex,centrey,centrez)$, et l'axe $(hautx,hauty,hautz)$ correspond à la verticale de la caméra. `gluLookAt()` calcule les rotations et translations nécessaires pour positionner la caméra correctement, et multiplie le résultat à droite de la matrice active. Si on réécrit la fonction `display()` comme précisé dans le code, on considère que les deux translations sont des transformations de visualisation, on crée un polygone et on positionne la caméra en $(-1, 0, 5)$.

```
1 void display()
2  {
3  glLoadIdentity();
4   gluLookAt(-1,0,5,0,0,0,0,1,0);
5   glBegin(GL_POLYGON);
6   ...
7   glEnd();
8  }
```

2.2 Pile de matrices

Jusqu'à présent, toutes nos scènes 3D ne comportaient qu'un seul objet. Imaginons maintenant une scène contenant deux cubes placés respectivement en $(1, 0, 2)$ et $(5, 0, 0)$, et supposons que nous disposions d'une fonction `dessineCube()` qui génère un cube centré en l'origine. On pourrait penser que ce qui suit permet de créer la scène :

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

```
glLoadIdentity();
glTranslatef(1.0,0.0,2.0);
dessineCube();
glTranslatef(5.0,0.0,0.0);
dessineCube();
```

Les plus malins auront vu qu'il y a un problème : le second cube sera mal positionné. Lors du dessin de ce dernier, la matrice de modélisation-visualisation contient une composition des deux translations, et le second cube sera affiché en $(6, 0, 2)$. Pour y remédier, il suffit de réinitialiser la matrice après le dessin du premier cube, en appelant `glLoadIdentity();`

A présent, supposons qu'en plus de ces deux transformations, on souhaite appliquer une transformation de visualisation pour déplacer l'observateur :

```
glLoadIdentity();
gluLookAt(10.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0);
/* Affichage et positionnement  du premier cube */
glTranslatef(1.0,0.0,2.0);
dessineCube();
/* Affichage et positionnement du second cube */
glLoadIdentity();
glTranslatef(5.0,0.0,0.0);
dessineCube();
```

Là encore, il y a un problème : la fonction `gluLookAt()` a pour effet de modifier la matrice de modélisation-visualisation. Etant donné qu'on remet la matrice à zéro après le dessin du premier cube, les transformations de visualisation sont effacées et ne sont pas appliquées au second cube. La première solution qu'on peut envisager consiste à refaire un `gluLookAt()` après la remise à zéro de la matrice. C'est une solution peu satisfaisante : on spécifie la position de l'observateur à chaque création d'objet. Une seconde solution consiste à utiliser un mécanisme proposé par OpenGL : les piles de matrices.

OpenGL propose un mécanisme de pile pour la matrice de modélisation-visualisation. La fonction `glPushMatrix()` place une copie de la matrice active au dessus de la pile. La fonction `glPopMatrix()` retire la matrice située au sommet de la pile et la place dans la matrice active (en écrasant l'ancienne matrice active).

Le code qui suit permet de régler le problème que nous avions plus haut :

```
glLoadIdentity();
gluLookAt(10.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0);
glPushMatrix();
glTranslatef(1.0,0.0,2.0);
dessineCube();
glPopMatrix();
glTranslatef(5.0,0.0,0.0);
dessineCube();
```

Les deux premières lignes mettent en place la matrice de visualisation dans la matrice active. On place ensuite une copie de cette matrice dans la pile, puis on génère le premier cube. Une fois ceci fait, on récupère la matrice de visualisation grâce à un `glPopMatrix()`, et on affiche le second cube.

2.3 Les transformations de projection

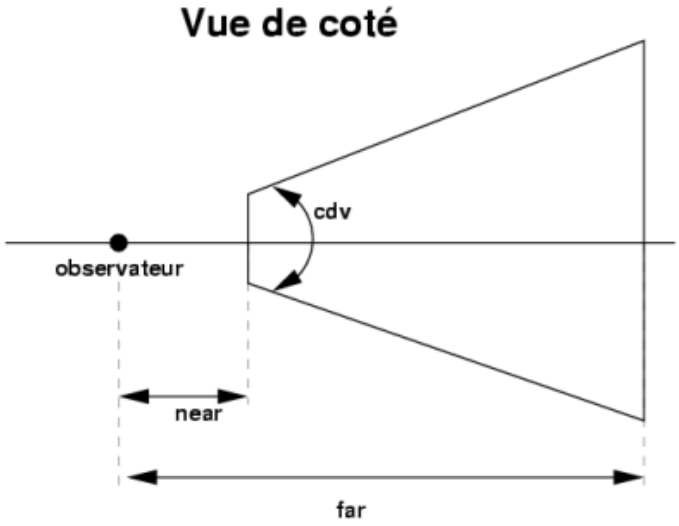
`OpenGL` vous fournit des fonctions permettant de spécifier les matrices des deux types de projection les plus fréquents (la projection orthogonale et la projection perspective), tout en gérant le problème du fenêtrage.

OpenGL fournit deux fonctions pour spécifier une projection perspective. La plus simple (et la seule que nous verrons) se nomme `gluPerspective()`, et a le prototype suivant :

```
void gluPerspective(GLdouble cdv,GLdouble rapport,GLdouble devant,GLdouble derriere);
```

Les paramètres de `gluPerspective()` permettent de définir le volume de vue associé à la projection perpective. La signification de chacun des paramètres est expliquée sur la figure suivante, excepté `rapport`, qui correspond au

rapport $largeur/hauteur$ de l'image. Jusqu'à présent, nous avons toujours travaillé avec des images carrées, donc pour éviter les distorsions, il nous faut choisir un rapport de 1.



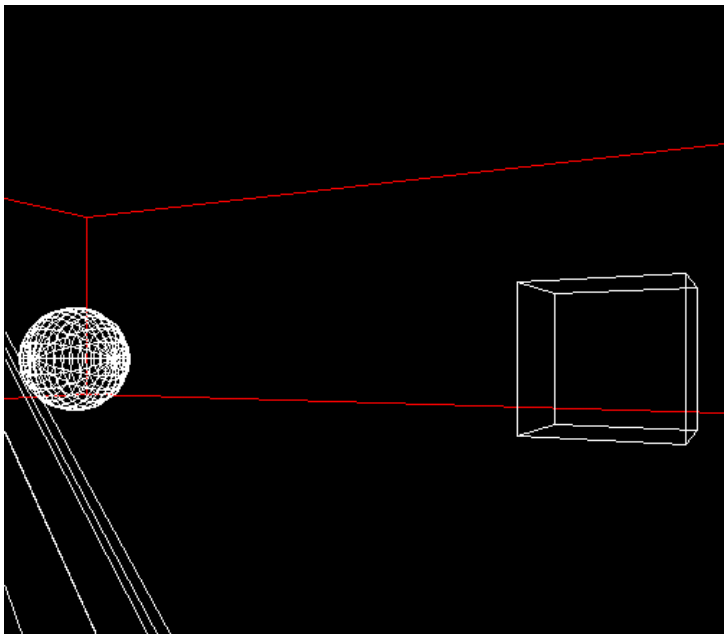
De manière standard, la définition d'une méthode de projection pour un scène se fait de la manière suivante :

```
glMatrixMode(GL_PROJECTION); /* on rend active la matrice de projection */
 glLoadIdentity(); /* on reinitialise la matrice de projection */
gluPerspective(....)
 glMatrixMode(GL_MODELVIEW); /* retour a la matrice de
    modelisation-visualisation  */
```

A l'instar des transformations géométriques, il faut penser à réinitialiser la matrice avec `glLoadIdentity()` avant de spécifier la projection.

2.4 Mise en application

Pour illustrer le tout, vous allez développer le code permettant une ballade virtuelle dans un pièce contenant 4 objets, le tout en vue fil de fer.



La scène est composée de 5 primitives : deux cubes (dont un qui représente les murs de la pièce), un cône, une sphère et un tore. Décrire chacun de ces objets point par point serait fastidieux, et nous allons donc faire appel à glut pour nous aider : en effet, l'API de Mark Kilgard contient des fonctions qui se chargent pour nous de décrire les primitives suivantes : sphère, cube, cone, tore, dodécaèdre, octaèdre, tétraèdre, icosaèdre et .... théière [2].

2. allez faire un tour sur http ://web2.iadfw.net/sjbaker1/software/teapot.html pour plus d'infos

Par exemple, `glutWireSphere(x)` permet de générer un sphère de rayon x en vue fil de fer. Si vous souhaitez affichez une sphère pleine, utilisez `glutSolidSphere()`..
La pièce est un cube de 20 unités de longueur auquel on a appliqué une homothétie de rapport 0.25 suivant l'axe des Y. Elle est centrée en l'origine. Les quatre autres objets sont placés dans le cube et sont "posés" sur le sol.
On souhaite pouvoir déplacer l'observateur à l'intérieur de la scène. Pour cela, on s'impose les règles suivantes :

- L'observateur peut se déplacer n'importe où dans la pièce. En notant px et pz la position de l'observateur (par défaut, l'axe y est l'axe vertical), cela revient à dire que px et pz doivent être compris entre -10 et $+10$.
- L'observateur peut avancer en appuyant sur la touche 'z' et reculer avec la touche 's'.
- L'observateur peut tourner sur lui même (en appuyant sur le bouton gauche de la souris puis en déplaçant celle-ci horizontalement).
- L'observateur regarde toujours l'horizon. Il ne peut ni lever ni baisser la tête.

Pour repérer notre utilisateur dans la pièce, nous utiliserons trois variables : px, pz (position de l'observateur) et r (angle indiquant la direction dans laquelle l'utilisateur regarde).
Voici la fonction `main()` de ce programme ainsi que la fonction `calcCosSinTable()`.Nous utilisons dans nos calculs de position les opérateurs mathématiques `sinus` et `cosinus` dont les temps d'exécution sont relativement longs. Pour ne pas ralentir l'application, on précalcule les cosinus et sinus pour les valeurs allant de 0 à 359, et on les stocke dans deux tableaux. Bien sûr, ceci nous force à n'utiliser que des valeurs d'angle entières.

```
1  /*********************************************************/
2  /*                      ballade.c                        */
3  /*********************************************************/
4  /* Petite ballade dans un monde (reduit) en fil de fer  */
5  /*********************************************************/
6
7  /* inclusion des fichiers d'entete Glut */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <GL/glut.h>
12 #include <math.h>
13
14 void display();
15 void keyboard(unsigned char key,int x, int y);
16 void mousePress(int bouton,int state,int x,int y);
17 void MouseMotion(int x,int y);
18 void calcCosSinTable();
19 void testPosition();
20 void changePerspective();
21
22 float pz=0.0,px=0.0,Sin[360],Cos[360];
23 int xold,r=0;
24
25
26 int main(int argc,char **argv)
27 {
28
29   /* initialisation de glut et creation
30      de la fenetre */
31   glutInit(&argc,argv);
32   glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE);
33   glutInitWindowPosition(200,200);
34   glutInitWindowSize(500,500);
35   glutCreateWindow("29");
36
37   /* Initialisation d'OpenGL */
38   glClearColor(0.0,0.0,0.0,0.0);
39
40
```

```
41    //Reglage de la perspective
42    glMatrixMode(GL_PROJECTION);
43    glLoadIdentity();
44    //Vous pouvez jouer avec les parametres pour observer leur impact sur la
          visualisation de la scene
45    gluPerspective(50,1.0,0.1,40.0);
46    glMatrixMode(GL_MODELVIEW);
47
48    /* Precalcul des sinus et cosinus */
49    calcCosSinTable();
50
51    /* enregistrement des fonctions de rappel */
52    glutDisplayFunc(display);
53    glutKeyboardFunc(keyboard);
54    glutMouseFunc(mousePress);
55    glutMotionFunc(MouseMotion);
56
57    /* Entre dans la boucle principale glut */
58    glutMainLoop();
59    return 0;
60  }
61  void calcCosSinTable()
62  {
63    int i;
64    for (i=0;i<360;i++){
65      Sin[i]=sin(i/360.0*6.283185);
66      Cos[i]=cos(i/360.0*6.283185);
67    }
68  }
```

Complétez la fonction `mousePress` afin que l'on mémorise dans la variable globale `xold` l'abscisse correspondant à l'impact de la souris.

```
1   void mousePress(int bouton,int state,int x,int y)
2  {
3    if (bouton==                    && state==           ){
4       xold=    ;
5    }
6  }
```

Complétez la fonction `mousePress` afin de mettre à jour l'angle de vue (variable `r`) en fonction du déplacement de la souris.

```
1
2
3  void MouseMotion(int x,int y)
4  {
5    r+=x-xold;
6  //Gerer le fait que 360+1=0
7  //et que 0-1=360
8    if (r>=360)
9          ;
10   if (r<0)
11      ;
12   //N'oubliez pas de mettre a jour la nouvelle position de xold (apres
         avoir deplace la souris)
13   xold=    ;
14 //Penser a demander le reaffichage
15        ;
16 }
```

Nous allons maintenant écrire la fonction de rappel `keyboard` qui permet de déplacer la caméra dans la scène.

```
1
2  void keyboard(unsigned char key,int x, int y)
3  {
4    switch (key)
5      {
6      case 'q':
7        exit(0);
8  //Quand l'utilisateur avance de n unites dans la direction dans laquelle il
       regarde, il suffit pour calculer sa nouvelle position de decrementer pz
       de n*cos(r) et d'incrementer px de n*sin(r).
9  //Faites en sorte que la touche z permette d'avancer de 0.5
10     case 'z':
11       pz  ____________ ;
12       px  ____________ ;
13       //testPosition();
14       //on redemande l'affichage
15         ___________________ ;
16       break;
17 //Quand l'utilisateur recule de n unites par rapport a la direction dans
       laquelle il regarde, il suffit pour calculer sa nouvelle position
       d'incrementer pz de n*cos(r) et de decrementer px de n*sin(r).
18 //Faites en sorte que la touche s permette de reculer de 0.5
19     case 's':
20       pz  ____________ ;
21       px  ____________ ;
22       //testPosition();
23       //on redemande l'affichage
24         ___________________ ;
25       break;
26
27     }
28 }
```

Nous allons maintenant écrire la fonction `display` Au lancement du programme, l'utilisateur est positionné à l'origine, et il regarde en direction de l'axe des z négatif, ce qui revient à dire que $px = pz = r = 0$.

```
1  void display()
2  {
3    /* effacement de l'image avec la couleur de fond */
4    glClear(GL_COLOR_BUFFER_BIT);
5    glLoadIdentity();
6
7
8    /*Application des transfos de visualisation */
9   //la rotation est faite sur l'axe des y d'un angle r
10   glRotated(r,0.0,1.0,0.0);
11 //la translation est faite sur x et z selon les vecteurs de deplacements
       calcules dans la fonction keyboard
12   glTranslatef( ____ ,0.0, ____ );
13
14   //Il faut mettre la matrice de transition dans la pile afin de pouvoir
       dessiner plusieurs objets
15     _____________ ;
16
17  /* Dessin des objets */
18 //Forme 1:placer un cube (pour faire les murs de la salle)
19 //Ce mur sera place en 1.5 sur l'axe des y
20
21   glTranslatef( _______ );
22 //effectuez une homotethie de 1/4 sur l'axe des y pour donner l'impression
```

```
    que ce cube sera rectangulaire
  glScalef( ____________ );
//Donnez lui la couleur que vous souhaitez
  glColor3d( ______ );
//Donnez lui la dimension 20
  glutWireCube( ___ );

//Depilez la matrice pour se retrouver dans l'etat initial
  _____________ ;
//empilez la de nouveaux pour pouvoir ajouter une deuxieme forme
  _____________ ;

//Forme 2: Placer un cone
//Position en (5,-1,5)
  glTranslated( _______ );
//Faites lui faire une rotation de -90 degres si vous voulez le voir droit
  glRotated( _________ );
//Donnez lui la couleur que vous voulez
  _____________________ ;

//creation du cone
  glutWireCone( _________ );
//Gestion de la pile
  _____________
  _____________

//Forme 3 : creation d'un Tor
//Positionnez le ou vous voulez (il faut qu'il reste dans le cube initial)
  _____________________ ;
  glutWireTorus(0.2,0.8,20,30);


//Gestion de la pile
  _____________ ;
  _____________ ;
//Creation de la forme 4 : un cube
//Placez le ou vous voulez
  _____________________ ;
  glutWireCube(2);

//Gestion de la pile
  glPopMatrix();
//Pas besoin de faire de push car il n'y aura pas d'autre forme

//Derniere forme : une sphere
//Positionnez le ou vous voulez
  ____________________ ;
  glutWireSphere( ________ );


  /* on force l'affichage du resultat */
  glFlush();
  glutSwapBuffers();
}
```

Arrivée à ce niveau vous êtes sensés pouvoir naviguer entre les objets à l'aide des deux touches et de la souris.
Il reste un problème à régler. En effet, nous n'avons pas encore géré le fait qu'il nous est possible de sortir de la pièce principale.
Ecrivez la fonction `void testPosition()` permettant de gérer la collision avec les murs. Cette fonction est appelée dans `display()` avant l'affichage de chaque déplacement. Pour détecter la collision, il vous suffit de tester la valeur

de px et pz pour savoir si vous avez dépassé. Si oui, il vous suffit alors de mettre à jour la valeur de px ou pz. N'oubliez pas que la caméra est centrée ($px = pz = 0$) au milieu de la scène. Vous entez donc en collision avec un mur si vous vous déplacez de 10 (ou -10) selon x.

```
void testPosition()
{
  if (px>  ____ )
    px=  ____ ;
  if (px<  _____ )
    px=  _____ ;
  if (pz>  ____ )
    pz=  ____ ;
  if (pz<  _____ )
    pz=  _____ ;
}
```