Transformations géométriques

Par *Xavier Michelon*

LinuxGraphic.org

Introduction :

Après deux didacticiels destinés à vous enseigner les principes de base d'OpenGL, nous allons aujourd'hui nous intéresser à un sujet un peu plus délicat qui demande, pour une bonne compréhension, de posséder quelques bases mathématiques : les transformations géométriques.

Dans les didacticiels précédents, nous avons créé nos objets en indiquant la position dans l'espace de chacun des points. Il se trouve que le carré et le cube que nous avons utilisés se trouvaient centrés en l'origine. La dernière fois nous avons vu qu'il était possible de stocker des objets dans des structures de données. Nous l'avons fait pour notre cube. Imaginons à présent que nous voulions afficher une scène contenant plusieurs cubes. La première solution qui vient à l'esprit consiste à décrire les points de chaque cube à la position à laquelle ils doivent être affichés : si on souhaite afficher un cube dont le centre se situe au point (5,0,0). Il est aisé de faire le calcul des coordonnées des points. Supposons maintenant que nous voulions qu'un second cube soit centré en l'origine, et qu'il subisse une rotation autour de l'axe Z de 5 degrés. Vous conviendrez que le calcul "à la main" n'est pas envisageable. OpenGL propose pour nous aider un certain nombre de transformations de base : les translations, les rotations et les homothéties. Pour bien comprendre comment utiliser les transformations, intéressons−nous dans un premier temps aux aspects mathématiques de la 3D.

Des maths, des maths !!!

Les algorithmes 3D utilisent le calcul matriciel, car il permet de rassembler tous les calculs nécessaires sous une forme compacte. Dans la suite du didacticiel, je suppose que vous savez ce que sont une matrice et un produit matriciel. Si ce n'est pas le cas, je vous conseille de jeter un coup d'oeil dans un ouvrage traitant du sujet avant de lire ce qui suit. Mon but n'est pas de faire un cours de mathématiques, mais de vous faire comprendre ce qui ce passe "à l'intérieur" d'une application 3D.

Il semblerait logique de penser que puisqu'on travaille en 3 dimensions, les matrices utilisées sont elles aussi de dimension 3. En fait ce n'est pas le cas. OpenGL travaille avec des matrices 4x4, dans un système de coordonnées dit "homogène". L'avantage de ce système est qu'il permet de représenter les transformations par des matrices et de composer toutes ces transformations par un simple produit matriciel.

Les points de l'espace sont stockés dans des vecteurs de 4 lignes (voir figure 1). Les 3 premières lignes contiennent respectivement les coordonnées x, y et z du point. La quatrième ligne contient un coefficient noté w, appelé facteur d'échelle, qui vaut souvent 1. Considérons un point P en coordonnées homogènes, pour lequel le facteur d'échelle ne vaut pas 1. Pour retrouver les coordonnées cartésiennes du point, il suffit de diviser toutes les composantes par w. Vous remarquerez qu'avec ce système, un même point de l'espace a une infinité de représentations : les coordonnées (2,10,8,2) et (1,5,4,1) correspondent au même point. Il est également intéressant de noter que les coordonnées homogènes permettent de définir des points situés à l'infini, en choisissant w=0.

$$P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1.5 \\ 10 \\ 2.1 \\ 1 \end{bmatrix}$$

Figure 1 : Représentation d'un point P en coordonnées homogènes

Vous trouverez sur les figures 2, 3 et 4 les formes des matrices de translation, de rotation, et d'homothétie. Je pense que vous savez ce que sont une rotation et une translation, mais l'homothétie est peut−être plus mystérieuse. Une homothétie est un changement d'échelle de l'objet. Une homothétie uniforme de facteur 2 va faire doubler la taille d'un objet, une homothétie de facteur 0,5 va la diminuer de moitié, et bien sûr, une homothétie de facteur 1 va laisser l'objet à sa taille d'origine. Dans la matrice d'homothétie on trouve 3 facteurs hx,hy et hz. Si ces trois coefficients sont égaux, on est dans le cas d'une homothétie uniforme. Si les facteurs ne sont pas identiques, on a une homothétie non uniforme, qui vous donne la possibilité d'étirer et de rétrécir différemment suivant les axes X, Y et Z.

$$M = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2 : Matrice de translation de vecteur (tx,ty,tz)

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3 : Matrice de rotation d'angle thêta autour de l'axe x

$$M = \begin{bmatrix} h_x & 0 & 0 & 0 \\ 0 & h_y & 0 & 0 \\ 0 & 0 & h_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4 : Matrice d'homothétie de facteurs hx,hy,hz

Transformer un objet consiste à appliquer la transformation à l'ensemble des points qui le constituent. On souhaite appliquer une transformation de matrice M à un point P. Appelons P' le résultat de la transformation. Le résultat s'obtient par une simple multiplication matricielle P'=M*P. La composition de transformations est toute aussi simple. Supposons qu'on veuille appliquer à une objet une translation de matrice Mt, puis une rotation de matrice Mr. Dans un premier temps, on applique la translation en calculant Mt*P, puis on multiplie à gauche le résultat par Mr. Le résultat de la composition des deux transformations appliquées à P vaut Mr*(Mt*P). Le produit matriciel a pour propriété d'être associatif, ce qui implique que Mr*(Mt*P)= (Mr*Mt)*P. Par conséquent, on peut considérer la composée de transformation comme une unique transformation de matrice Mr*Mt. En revanche, la multiplication matricielle n'est pas commutative, c'est−à−dire que Mr*Mt est à priori différent de Mt*Mr. Ceci implique que l'ordre dans lequel on multiplie les matrices de transformation est crucial, et il est important de noter que puisque nous lisons de gauche à droite, l'ordre d'écriture des matrices est inversé par rapport à l'ordre d'application des transformations.

$$
\begin{aligned}
P' &= M_r.M_t.P \\
P' &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1.5 \\ 10 \\ 2.1 \\ 1 \end{bmatrix} \\
P' &= \begin{bmatrix} 3.5 \\ -6.1 \\ 11 \\ 1 \end{bmatrix}
\end{aligned}
$$

Figure 5: Transformation d'un point par composition d'une translation et d'une rotation

Retour à OpenGL :

Revenons maintenant à OpenGL. Si vous vous rappelez le didacticiel précédent, je vous ai indiqué comment faire tourner le cube avec la fonction glRotatef(). Grâce au paragraphe précédent, vous allez pouvoir comprendre ce que font réellement la fonction glRotatef() et les autres fonctions de transformation que nous allons voir.

La bibliothèque OpenGL met à la disposition du programmeur trois matrices : une matrice de transformation−visualiation (celle qui nous intéresse aujourd'hui), une matrice de projection, et une matrice de texture. Ces trois matrices sont stockées dans des structures de données internes à la bibliothèque, et en général, l'utilisateur n'y accède directement. Afin de travailler sur ces matrices, OpenGL définit une matrice "active", c'est−à−dire la matrice sur laquelle vont être effectuer les transformations qui suivront. Pour choisir la matrice active, on utilise

```
glMatrixMode(GLenum mode)
```

Le paramètre "mode" désigne la matrice que l'on souhaite activer. Il peut prendre comme valeur GL_MODELVIEW, GL_PROJECTION, et GL_TEXTURE. Pour effectuer des transformations sur les objets de la scène, il faut modifier la matrice de transformation−visualisation (GL_MODELVIEW). Nous reviendrons sur les deux autres matrices dans les prochains didacticiels. Par défaut, la matrice de modélisation−visualisation est la matrice active, ce qui explique pourquoi nous n'avons pas eu besoin d'utiliser glMatrixMode() la dernière fois.

Définir une transformation avec OpenGL consiste à placer la matrice correspondant à cette transformation dans la matrice de modélisation−visualisation. Bien qu'il soit possible de spécifier directement les 16 coefficients de la matrice active grâce à la fonction glLoadMatrix(), on procède en général autrement, en utilisant les fonctions glLoadIdentity(), glRotate(), glTranslate et glScale().

```
void glLoadIdentity();
```

cette fonction a pour effet de placer dans la matrice active la matrice dite d'identité, représentée sur la figure 6. Cette matrice correspond à une transformation nulle : les points ne sont pas déplacés. Pour vous en convaincre, multipliez une matrice d'identité par un vecteur P, vous verrez que le résultat du produit n'est autre que P. Quel en est l'intérêt ? Cette fonction permet de "remettre à zéro" la matrice de transformation.

$$
I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Figure 6 : La matrice identité

```
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 glRotatef(float theta,float x,float y,float z);
```

Cette fonction multiplie la matrice active par une matrice de rotation d'angle thêta autour de l'axe passant par l'origine et porté par le vecteur (x,y,z).

```
void glScalef(float hx,float hy,float hz);
```

Cette fonction multiplie la matrice active par une matrice de d'homothétie dont les facteurs suivant les axes X, Yet Z sont respectivement hx, hy et hz.

Prenons un exemple concret : on souhaite placer dans la matrice de modélisation−visualisation une matrice correspondant à une translation de vecteur (0,1,0) suivie d'une rotation d'angle 45° autour de l'axe Z. En notant Mt la matrice de translation et Mr la matrice de rotation, on désire placer dans la matrice active le résultat du produit Mr.Mt (rappelez vous que l'ordre d'écriture des matrices est inversé par rapport à l'ordre d'application des transformations). La portion de code OpenGL pour accomplir cette tâche est la suivante :

```
glLoadIdentity();
glRotatef(45.0,0.0,0.0,1.0);
glTranslatef(0.0,1.0,0.0);
```

A priori, on ne sait pas ce qui se trouve dans la matrice, et donc avant d'effectuer la moindre opération, il convient de la réinitialiser en y plaçant la matrice identité avec un appel à glLoadIdentity(). on enchaîne ensuite les transformations à ajouter dans la matrice, puis on peut commencer la description de l'objet 3D. Lorsque vous spécifiez un point de l'espace, OpenGL lui applique la transformation contenue dans la matrice de modélisation−visualisation, et l'objet entier subit donc la transformation.

Le programme exemple :

L'exemple que je vous propose illustre le principe de composition de transformations. Intéressons−nous tout d'abord à la fonction d'affichage. L'objectif est de dessiner un carré 2D centré en l'origine, auquel on a appliqué une rotation d'angle a autour de l'axe Z, suivie d'une translation de vecteur (0.5,0.0,0.0) et d'une rotation d'angle b autour de Z. En appliquant la procédure que je viens d'énoncer dans plus haut, voici ce que nous devons mettre dans la fonction d'affichage :

```
glLoadIdentity();
glRotatef(b,0.0,0.0,1.0);
glTranslatef(0.5,0.0,0.0);
glRotatef(a,0.0,0.0,1.0);


glBegin(GL_POLYGON);
glVertex3f(−0.2,−0.2, 0.0);
glVertex3f( 0.2,−0.2, 0.0);
glVertex3f( 0.2, 0.2, 0.0);
glVertex3f(−0.2, 0.2, 0.0);
glEnd();
```

Pour animer notre cube, nous allons utiliser une nouvelle fonction de rappel : la fonction d'oisiveté (idle en anglais). Cette fonction est appelée chaque fois que le gestionnaire d'évènement n'a aucun autre évènement à traiter.

```
void idle()
{
  a+=inca;
  if (a>360)
    a−=360;
  b+=incb;
    if (b>360)
  b−=360;
  glutPostRedisplay();
}
```

Dans la fonction idle, les valeurs des angles de rotation a et b sont incrémentées, puis une demande de rafraîchissement est faite. Le cube est alors redessiné à l'écran avec prise en compte des nouvelles valeurs des angles de rotation. En tenant compte du fait qu'une rotation de 360 degrés nous ramène au point de départ, on prend soin d'éviter les dépassements de capacité en ôtant 360 degrés aux angles dès qu'ils dépassent cette valeur.

Afin de vous permettre de visualiser l'effet de chacune des rotations, le programme vous donne la possibilité de modifier les incréments appliqués aux angles a et b à chaque pas d'animation. La touche 'a' augmente l'incrément de la rotation d'angle a. 'Shift+a' le diminue. De la même manière, les combinaisons de touches 'b' et 'Shift+b' modifient l'incrément de la rotation d'angle b.

Précisions sur les rotations :

Vous avez peut−être remarqué que glRotate() ne permet de générer que des rotations dont l'axe passe par l'origine. Comment faire si on souhaite obtenir une rotation dont l'axe passe par un point P de coordonnées (Px,Py,Pz) ? C'est très simple : il suffit de ramener le point P à l'origine par une translation de vecteur (−Px,−Py,−Pz), d'appliquer la rotation, puis de ramener l'objet à sa position initiale avec une translation de vecteur (Px,Py,Pz), ce qui donne en OpenGL :

```
glTranslatef(Px,Py,Pz);
glRotatef(a,Ax,Ay,Az);
glTranslatef(−Px,−Py,−Pz);
```

Conclusion :

C'est tout pour aujourd'hui. Vous en savez maintenant un peu plus certains aspects théoriques qui se cachent dans les moteurs 3D. La prochaine fois, nous aborderons les transformations de visualisation qui permettent de définir le point de vue et l'orientation de la caméra dans la scène. Nous aborderons également le système de pile de matrices, et nous étudierons la question de la perspective.

Code source :

```
/*******************************************************/
/*                     transf.c                        */
/*******************************************************/
/* Exemple de composition de transformations           */
/*******************************************************/

#include <GL/glut.h>

int a=0,b=0;
int inca=2,incb=2;

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

int main(int argc,char **argv)
{
  /* initialisation de glut et creation
     de la fenetre */
  glutInit(
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
  glutInitWindowPosition(200,200);
  glutInitWindowSize(500,500);
  glutCreateWindow("transf");

  /* Initialisation d'OpenGL */
```

```
  glClearColor(0.0,0.0,0.0,0.0);
  glColor3f(1.0,1.0,1.0);
  glShadeModel(GL_FLAT);

  /* enregistrement des fonctions de rappel */
  glutDisplayFunc(affichage);
  glutKeyboardFunc(clavier);
  glutReshapeFunc(reshape);
  glutIdleFunc(idle);

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

void affichage()
{

  glClear(GL_COLOR_BUFFER_BIT);

  glLoadIdentity();
  glRotatef(b,0.0,0.0,1.0);
  glTranslatef(0.5,0.0,0.0);
  glRotatef(a,0.0,0.0,1.0);

  glBegin(GL_POLYGON);
  glVertex3f(−0.2,−0.2, 0.0);
  glVertex3f( 0.2,−0.2, 0.0);
  glVertex3f( 0.2, 0.2, 0.0);
  glVertex3f(−0.2, 0.2, 0.0);
  glEnd();
  glFlush();

  glutSwapBuffers();
}

void clavier(unsigned char touche,int x,int y)
{
  switch (touche)
    {
    case 'a':
      inca++;
      if (inca>10)
        inca=10;
      glutPostRedisplay();
      break;
    case 'A':
      inca−−;
      if (inca10)
        incb=10;
      glutPostRedisplay();
      break;
    case 'B':
      incb−−;
      if (incb360)
    a−=360;
  b+=incb;
  if (b>360)
    b−=360;
  glutPostRedisplay();
}
```