```
========================================
Comparaison de façons de dessiner une ligne

   en Java, sans OpenGL:

      Graphics g = ...
      g.drawLine( x1, y1, x2, y2 );

   avec OpenGL:

      glBegin( GL_LINES );
         glVertex2f( x1, y1 );
         glVertex2f( x2, y2 );
      glEnd();

Remarque:
1. Quand on utilise glVertex2f(),
   la coordonnée z est implicitement 0.

========================================

Comparaison de façons de dessiner un rectangle

   en Java, sans OpenGL:

      Graphics g = ...
      g.drawRect( x1, y1, x2-x1, y2-y1 );

   avec OpenGL:

      glBegin( GL_QUADS );
         glVertex2f(x1,y1);
         glVertex2f(x2,y1);
         glVertex2f(x2,y2);
         glVertex2f(x1,y2);
      glEnd();

Remarques:
1. En Java, sans OpenGL, on utilise habituellement
   Graphics (ou Graphics2D) pour dessiner.
   Dans ces cas, l'axe x+ est vers la droite,
   et l'axe y+ est vers le BAS.

   En OpenGL, les coordonnées passées aux appels
   à glVertex*() sont dans un système de coordonnées
   3D, orienté main droite (pouce x, index y, majeur z).
   La correspondence entre ces axes 3D et les axes
   de l'écran varie selon la position et l'orientation
   de la vue de caméra, mais par défaut,
   c'est x+ vers la droite, y+ vers le HAUT,
   et z+ sortant de l'écran.

2. L'appel à Graphics.drawRect() passe des
   coordonnées en pixels, tandis que les appels
   à glVertex*() passent des coordonnées en unités
```

3D arbitraires. Avec le OpenGL, la correspondence
entre les unités 3D et les pixels dépend de
la caméra.

3. Lorsque le "backface culling" est activé via
```
   glEnable( GL_CULL_FACE );
```
par défaut, OpenGL suppose que les sommets
("vertexes") des polygones sont stockés dans
un sens anti-horaire, et il se sert de cet
ordre pour déterminer quels polygones sont
face à la caméra ou non.
Donc, si vous utilisez le "backface culling" pour accélerer
le rendu, vous devez normalement faire vos appels
à glVertex*() en spécifiant les sommets
dans un sens anti-horaire.

Vous pouvez toutefois désactiver le "backface culling"
avec
```
   glDisable( GL_CULL_FACE );
```
pour éviter des problèmes, ou encore l'activer en
utilisant la convention inverse
```
   glEnable( GL_CULL_FACE );
   glFrontFace( GL_CW ); // sens horaire (CW = "Clock Wise")
```

==========

Voici une routine pour dessiner un visage:

```
drawFace( Graphics g ) {
   g.drawOval(-25,-25,50,50);
   g.drawRect(-10,-10,20,5);
   g.drawLine(-10,10,10,10);
}
```

Le visage est toujours dessiné au même endroit.

==========

Pour pouvoir dessiner le visage à différents endroits,
considérons les deux approches suivantes:

```
drawFace( Graphics g, int x, int y ) {
   g.drawOval( x-25, y-25, 50, 50 );
   g.drawRect( x-10, y-10, 20, 5 );
   g.drawLine( x-10, y+10, 10, 10 );
}
```

ou bien

```
drawFace( Graphics g, int x, int y ) {
   setupTranslation(x,y);

   g.drawOval(-25,-25,50,50);
   g.drawRect(-10,-10,20,5);
   g.drawLine(-10,10,10,10);
```

```
   }
```

Le deuxième est peut-être moins naturelle,
mais c'est la deuxième approche qu'on utilise
souvent en OpenGL. (D'ailleurs, on peut aussi
utiliser la deuxième approche avec Java2D et
Graphics2D, via la classe java.awt.geom.AffineTransform)

==========

Voici une routine pour dessiner un triangle avec OpenGL,
toujours au même endroit:

```
   drawTriangle() {
      glBegin( GL_TRIANGLES );
         glColor3f( 1, 0, 0 );
         glVertex3f( 1,    1, 1 );
         glVertex3f( 1.5f, 1, 1 );
         glVertex3f( 1,    2, 1 );
      glEnd();
   }
```

==========

Encore une fois, pour dessiner le triangle à différents
endroits, on a les deux approches:

```
   drawTriangle( float x, float y, float z) {
      glBegin( GL_TRIANGLES );
         glColor3f( 1, 0, 0 );
         glVertex3f( x+1,    y+1, z+1 );
         glVertex3f( x+1.5f, y+1, z+1 );
         glVertex3f( x+1,    y+2, z+1 );
      glEnd();
   }
```

ou bien

```
   drawTriangle( float x, float y, float z) {
      glTranslatef(x,y,z);

      glBegin( GL_TRIANGLES );
         glColor3f( 1, 0, 0 );
         glVertex3f( 1,    1, 1 );
         glVertex3f( 1.5f, 1, 1 );
         glVertex3f( 1,    2, 1 );
      glEnd();
   }
```

L'appel à glTranslatef() va établir une transformation
qui sera appliquée à tous les sommets émis par la
suite.

Encore une fois, la deuxième approche peut sembler moins
naturelle, mais il y a avec cette approche au moins deux
avantages:

1. Les additions de x et y aux coordonnées
   n'ont plus besoin d'être faites par le CPU;
   elle peuvent être faites par le GPU.
   Cela peut faire une grande différence si
   on a des millions de sommets à rendre,
   ou si on a des transformations plus
   compliquées (par exemple: une translation avec
   une rotation et un changement d'échelle)

2. La deuxième approche permet de faire de la
   modélisation hiérarchique / rendu hiérarchique
   (explications données en classe, avec l'exemple
   de l'auto qui a 4 roues, chacune ayant 5 boulons,
   formant une hiérarchie d'espaces ou une
   hiérarchie de systèmes de coordonnées)

==========

```
drawTriangle() {
   glBegin( GL_TRIANGLES );
      glColor3f( 1, 0, 0 );
      glVertex3f( 1,    1, 1 );
      glVertex3f( 1.5f, 1, 1 );
      glVertex3f( 1,    2, 1 );
   glEnd();
}

drawManyTriangles() {
   glPushMatrix();
      glTranslatef(x,y,z);
      glRotatef(...);
      drawTriangle();
   glPopMatrix();

   glPushMatrix();
      glTranslatef(x,y,z);
      glRotatef(...);
      glScalef(...);
      drawTriangle();
   glPopMatrix();
   ...
}
```

Remarques:

1. Les transformations (translations, rotations, etc.)
   peuvent être encodées dans des matrices.

2. glPushMatrix() et glPopMatrix() manipulent
   une pile de matrices, utile dans la
   modélisation hiérarchique / rendu hiérarchique.

==========

À quoi ressemblent ces matrices de transformation ?

Exemple: une matrice de rotation de 90 degrés en 2D:

```
     R = [0 -1]
         [1  0]
```

On peut faire l'exercise de transformer un point avec
cette matrice.

Soit a = [a_x] , un vecteur en colonne, le point à transformer.
         [a_y]

Pour appliquer la matrice de transformation,
on multiplie la matrice par le point à transformer:

```
   a' = R a

      = [ 0 -1 ] [ a_x ]
        [ 1  0 ] [ a_y ]

      = [ -a_y ]
        [  a_x ]
```

Vous pouvez vérifier qu'il s'agit d'une rotation de 90 degrés
autour de l'origine, dans le sens anti-horaire. Par exemple,
si a = (100, 30), alors a' = (-30, 100).

Exemple: une matrice de rotation de theta en 2D:

```
[ cos(theta)   -sin(theta) ]
[ sin(theta)    cos(theta) ]
```

Vous pouvez vérifier que, lorsque theta = 90 degrés,
la matrice si haute se réduit à l'exemple précédent.

Exemple: une matrice de rotation en 3D, avec un angle theta,
autour d'un axe de rotation donné par le vecteur unitaire (x,y,z):

```
[ x^2(1-c)+c    xy(1-c)-zs    zx(1-c)+ys ]
[ xy(1-c)+zs    y^2(1-c)+c    yz(1-c)-xs ]
[ zx(1-c)-ys    yz(1-c)+xs    z^2(1-c)+c ]
```

où c = cos(theta) et s = sin(theta).

Vous pouvez vérifier que, lorsque l'axe de rotation
est (x,y,z) = (0,0,1), c.-à-d. l'axe des z,
la matrice de rotation devient

```
[ c  -s  0 ]
[ s   c  0 ]
[ 0   0  1 ]
```

ce qui resemble drôlement à la matrice de rotation 2D !

Exemple: une matrice pour effectuer un changement d'échelle ("scale" en anglais) en 3D, avec les facteurs (s_x,s_y,s_z):

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

Si s = s_x = s_y = s_z, on a alors un changement d'échelle uniforme ("uniform scale") donné par la matrice

$$\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{bmatrix}$$

Rappel: pour appliquer une matrice de rotation ou de changement d'échelle à un point, il faut multiplier la matrice par le point.

Par contre, pour appliquer une translation (t_x,t_y,t_z) à un point, il faut additionner la translation.

$$a' = a + T$$

$$= \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

$$= \begin{bmatrix} a_x+t_x \\ b_y+t_y \\ c_z+t_z \end{bmatrix}$$

Donc, une formule générale pour appliquer une rotation R, un changement d'échelle S, et une translation T pourrait être

$$a' = S\ R\ a + T$$

$$= \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} . & . & . \\ . & . & . \\ . & . & . \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Si on veut dessiner un million de sommets avec ces transformations, on pourrait faire

```
// configuration des transformations à utiliser
glTranslatef( ... );
glScalef( ... );
glRotatef( ... );

glBegin( ... );
   for ( int i = 0; i < 1000000; ++i ) {
      glVertex3f( ... );
   }
glEnd();
```

et les transformations vont être appliquées par le GPU.
Comment peut faire le GPU pour optimiser les calculs à faire ?

Le GPU peut multiplier S et R pour donner une nouvelle matrice
3x3 qu'on appelle M:

```
   a' = S R a + T
      = M a + T

        [ . . . ] [ a_x ]   [ t_x ]
      = [ . . . ] [ a_y ] + [ t_y ]
        [ . . . ] [ a_z ]   [ t_z ]
```

Cela fait moins de calculs à faire pour chaque sommet.

En fait, M pourrait être une multiplication d'un enchaînement
de plusieurs rotations et changements d'échelle,
tous encodés dans une seule matrice 3x3 qu'on appelle M.

Malheureusement, la translation reste en dehors de tout ça,
car c'est une addition qu'on fait à la fin. Donc,
si on a un enchaînement de plusieurs
translations et rotations, une à la suite de l'autre,
il semble ne pas avoir de moyen d'accumuler ces
transformations ensemble.

Heureusement, il existe ce qu'on appele les coordonnées
homogènes, qui nous permettent d'exprimer les translations
comme des multiplications de matrice !

Un point ordinaire (x,y,z) correspond à une infinité de
coordonnées homogènes de la forme (xw, yw, zw, w)
où w est un nombre réel positif.
Autrement dit, pour convertir des coordonnées non-homogènes
(x,y,z) en coordonnées homogènes, on peut utiliser la
formule de conversion

```
   (x,y,z) -> (x,y,z,1)
```

Et pour convertir des coordonnées homogènes (x,y,z,w)
en coordonnées non-homogènes, on utilise la formule

```
   (x,y,z,w) -> ( x/w, y/w, z/w )
```

Maintenant, avec des coordonnées homogènes, nos matrices
de transformations deviennent des matrices 4x4.
Notre matrice de rotation devient

```
   [ x^2(1-c)+c    xy(1-c)-zs    zx(1-c)+ys   0 ]
   [ xy(1-c)+zs    y^2(1-c)+c    yz(1-c)-xs   0 ]
   [ zx(1-c)-ys    yz(1-c)+xs    z^2(1-c)+c   0 ]
   [ 0             0             0            1 ]
```

Notre matrice de changement d'échelle devient

```
[ s_x 0   0    0 ]
[ 0   s_y 0    0 ]
[ 0   0   s_z  0 ]
[ 0   0   0    1 ]
```

Et, pour encoder une translation (t_x,t_y,t_z),
on utilise la matrice

```
[  1  0  0  t_x  ]
[  0  1  0  t_y  ]
[  0  0  1  t_z  ]
[  0  0  0   1   ]
```

On peut faire l'exercise de vérifier que notre matrice
de translation fonctionne tel que voulu.
Prenons le point a = (a_x, a_y, a_z). Comme coordonnées
homogènes pour a, on peut prendre (a_x, a_y, a_z, 1).
Maintenant on calcul

```
  a' = T a

       [  1  0  0  t_x  ] [ a_x ]
     = [  0  1  0  t_y  ] [ a_y ]
       [  0  0  1  t_z  ] [ a_z ]
       [  0  0  0   1   ] [  1  ]

       [ a_x+t_x ]
     = [ a_y+t_y ]
       [ a_z+t_z ]
       [    1    ]
```

et si on converti les coordonnées homogènes de a'
en coordonnées non-homogènes, on obtient
(a_x+t_x, a_y+t_y, a_z+t_z), le resultat voulu.

Maintenant, on peut réécrire notre formule générale pour
appliquer une rotation R, un changement d'échelle S,
et une translation T:

```
a' = T S R a

     [  1  0  0  t_x  ] [ s_x 0   0    0 ] [ . . . . ] [ a_x ]
   = [  0  1  0  t_y  ] [ 0   s_y 0    0 ] [ . . . . ] [ a_y ]
     [  0  0  1  t_z  ] [ 0   0   s_z  0 ] [ . . . . ] [ a_z ]
     [  0  0  0   1   ] [ 0   0   0    1 ] [ . . . . ] [ a_w ]
```

et on peut définir une matrice 4x4 qu'on appelle M = T S R,
nous permettant d'écrire

$$
\begin{aligned}
a' &= T\ S\ R\ a \\
&= M\ a \\
&= \begin{bmatrix} m11 & m12 & m13 & m14 \\ m21 & m22 & m23 & m24 \\ m31 & m32 & m33 & m34 \\ m41 & m42 & m43 & m44 \end{bmatrix} \begin{bmatrix} a\_x \\ a\_y \\ a\_z \\ a\_w \end{bmatrix}
\end{aligned}
$$

En fait, on n'est pas limité à encoder seulement
UNE rotation R, UN changement d'échelle S,
et UNE translation T dans M.
On peut encoder un enchaînement arbitrairement long
de transformations dans la matrice M, dans n'importe
quel ordre. C'est-ce que le GPU fait: il accumule
les transformations en une seule matrice 4x4, pour
que les sommets puissent être rapidement transformés
par la suite.

De plus, il n'y a pas seulement une matrice M,
mais une pile de matrices, sur laquelle on peut
faire des "push" et des "pop", facilitant
la modélisation hiérarchique et le rendu hiérarchique.
Si une sous-routine veut manipuler les transformations
avant de dessiner quelque chose, elle peut faire

```
glPushMatrix();
```

(pour pousser une copie de la matrice actuelle sur la pile),
ensuite elle peut faire des appels arbitraires à
glRotatef(), glTranslatef(), et glScalef()
pour multiplier la matrice actuelle par des nouvelles
transformations.
Ensuite, elle peut émettre des sommets avec glVertex*()
(ces sommets seront transformés selon la matrice actuelle),
et lorsqu'elle a fini elle devra faire

```
glPopMatrix();
```

pour remettre la pile de matrices dans l'état où elle
était lorsque la sous-routine à été appelée.

Et DE PLUS, il n'y a pas seulement une pile de matrices,
mais DEUX piles de matrices: la pile MODELVIEW
(pour transformer des sommets de façon hiérarchique)
et la pile PROJECTION (associée avec la vue de la caméra).
Pour dire à OpenGL laquelle des piles on veut manipuler,
on fait

```
   glMatrixMode( GL_PROJECTION );
```

ou

```
   glMatrixMode( GL_MODELVIEW );
```

avant d'appeler glPushMatrix(), glPopMatrix(),
glRotatef(), glTranslatef(), glScalef(), ou toute
autre routine qui manipule la pile de matrices.

Remarque: les routines glVertex4f() et glVertex4*()
sont utiles si on veut spécifier la coordonnée w.
Mais, normalement, ce n'est pas nécessaire, et on
utilise glVertex3f(), laissant w égale à 1 implicitement.