```
========================================
Dans une classe servant à stocker un point, au lieu de définir

   class Point3D {
      ...
      float x, y, z;
      ...
   }

on peut plutôt faire

   class Point3D {
      ...
      float [] p = new float[3];
      ...
      public float x() { return p[0]; }
      public float y() { return p[1]; }
      public float z() { return p[2]; }
      ...
      public float [] get() { return p; }
      ...
   }

La méthode get() est utile pour faire des appels OpenGL:

   Point3D a = ...
   ...
   glVertex3fv( a.get() ); // plus rapide que
glVertex3f(a.x(),a.y(),a.z());

Notez qu'avec JOGL, ça serait plutôt

   GL gl = ...
   gl.glVertex3fv( a.get(), 0 );

De plus, il est parfois utile de pouvoir accéder au (k)ième coordonnée,
ou k est 0, 1, ou 2.  Cela est facile avec la méthode get().  Par
exemple:

   Vector3D v1 = ...
   for ( int dimension = 0; dimension < 3; ++ dimension ) {
      Vector3D v2 = new Vector3D(0,0,0);
      v2.get()[dimension] = v1.get()[dimension];
      // Maintenant, v2 est la projection de v1 sur le (dimension)ieme
axe
      ...
   }

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

Il est utile de définir non seulement une classe pour les points,
mais des classes séparées pour les
points (par exemple, pour stocker des sommets)
et les vecteurs (par exemple, pour stocker des déplacements
ou des différences entre deux points).
```

```
class Point3D {
   float [] p = new float[3];
   ...
}

class Vector3D {
   float [] v = new float[3];
   ...
}
```

Cela sert à créer de la documentation implicite dans votre code.

Certains vont même recommender de définir une troisième classe pour les vecteurs unitaires (par exemple, pour stocker des normales ou des directions), mais nous n'irons pas à ce point là.

Les points et les vecteurs devraient permettre des opérations arithmétiques de base:

```
vecteur + vecteur = vecteur
vecteur - vecteur = vecteur
point + vecteur = point
point - vecteur = point
point - point = vecteur
(point + point n'est pas permis)
```

En C++, on pourrait faire du "operator overloading":

```
// pour faire la somme d'un point avec un vecteur
inline Point3D operator+( const Point3D& p, const Vector3D& v ) {
   return Point3D( p.x()+v.x(), p.y()+v.y(), p.z()+v.z() );
}
...

Point3D p1 = ...
Vector3D v1 = ...
Vector3D v2 = ...

// appliquer la translation v1 à p1
Point3D p2 = p1 + v1;

// Remarque: grâce à la méthode get(), on n'a pas besoin de stocker
// la somme dans une variable de façon explicite.
glVertex3fv( (p1+v2).get() );
```

En Java, on doit utiliser des méthodes habituelles pour ces opérations:

```
Point3D.diff(...)
Point3D.sum(...)
Vector3D.sum(...)
Vector3D.diff(...)
Vector3D.dot(...) // produit scalaire de deux vecteurs
Vector3D.cross(...) // produit vectoriel de deux vecteurs
```

```
   Vector3D.mult(...) // produit d'un vecteur avec un scalaire
```

==========

Dans la classe Vector3D qui vous est fournie, on a défini
les méthodes negated() et normalized() qui retournent des
vecteurs sans modifier l'objet sur lequel ils sont appelés:

```
   Vector3D {
      ...
      public Vector3D negated() {
         return new Vector3D(-x(),-y(),-z()
      }
      public Vector3D normalized() {
         float l = length();
         // supposons que l > 0 ...
         return new Vector3D( x()/l, y()/l, z()/l );
      }
      ...
   }
```

au lieu de définir des méthodes negate() et normalize()
qui modifieraient le vecteur sur lequel ils sont appelés:

```
   Vector3D {
      ...
      public void negate() {
         p[0] = -p[0];   p[1] = -p[1];   p[2] = -p[2];
      }
      public void normalize() {
         float l = length();
         // supposons que l > 0 ...
         p[0] = p[0]/l;   p[1] = p[1]/l;   p[2] = p[2]/l;
      }
      ...
   }
```

Pourquoi ? Un client va vouloir stocker le resultat des
ces opérations soit dans le même vecteur, ou bien dans
un nouveau vecteur. Comparons les posibilités,
en utilisant negated()/negate() comme exemple.

   Pour stocker le résultat dans le même vecteur, on fait soit

```
      a.copy( a.negated() );      // 1 ligne de code
```

   ou bien

```
      a.negate();                 // 1 ligne de code
```

   Pour stocker le resultat dans un nouveau vecteur, on fait soit

```
      b = a.negated();            // 1 ligne de code
```

   ou bien

```
      b.copy( a );                // 2 lignes de code
      b.negate();
```

Alors, les méthodes du genre negate() qui retourne une version
modifiée supportent les deux utilisations avec une seule ligne
de code, contrairement aux méthodes du genre negated().

Il est aussi possible qu'un client veuille utiliser les
résultats de ces opérations dans des expressions plus
compliquées, sans stocker les résultats intermédiaires
dans des variables de façon explicite:

```
   Vector3D d = Vector3D.sum(
      Vector3D.cross( a, b ).normalized().negated(),
      c.normalized()
   ).normalized();
```

==========

Lorsque possible, éviter des calculs de racine carré.
Par exemple, pour savoir si un vecteur est plus long
qu'un autre, utiliser la longueur au carré:

```
   if ( v1.lengthSquared() > v2.lengthSquared() ) ...
```

au lieu d'utiliser la longueur:

```
   if ( v1.length() > v2.length() ) ...
```

==========

Dans la classe Vector3D qui vous est fournie,
on calcul la longueur du vecteur sur demande,
de façon semblable à l'exemple suivant:

```
   Vector3D {
      ...
      public float length() {
         return (float)Math.sqrt( x()*x() + y()+y() + z()*z() );
      }
      ...
   }
```

Quel est un désavantage potentiel de cette approche ?

Une approche alternative: on pré-calcul la longueur dans le constructeur
(et dans toute autre méthode qui modifie le vecteur):

```
   Vector3D {
      ...
      float length = 0;
      ...
      public Vector3D( float x, float y, float z ) {
         ...
         length = (float)Math.sqrt( x()*x() + y()+y() + z()*z() );
      }
      ...
      public float length() { return length; }
      ...
   }
```

Quel est un désavantage potentiel de cette deuxième approche ?

Voici une troisième approche, qui utilise de
l'évaluation paresseuse / évaluation retardée ("lazy evaluation")
avec une antémémoire ("cache") et un "dirty bit"
(bit d'impureté ?):

```
   Vector3D {
      ...
      float length = 0;
      boolean isLengthDirty = true;
      ...
      public float length() {
         if ( isLengthDirty ) {
            length = (float)Math.sqrt( x()*x() + y()+y() + z()*z() );
            isLengthDirty = false;
         }
         return length;
      }
      ...
   }
```

Avec cette l'approche, toute methode qui modifie les coordonnées
x, y, ou z du vecteur doit simplement faire

```
   isLengthDirty = true
```

par exemple:

```
      public void copy( Vector3D V ) {
         v[0] = V.v[0];
         v[1] = V.v[1];
         v[2] = V.v[2];
         isLengthDirty = true;
      }
```

Quel est un désavantage potentiel de cette deuxième approche ?

Nous n'avons pas utilisé l'approche d'une évaluation paresseuse
dans la classe Vector3D qui vous est fournie.

Toutefois, nous l'avons utilisé dans le calcul de la
boîte englobante ("bounding box") dans la classe Scene.
Voir
   Scene.boundingBoxOfScene
   Scene.isBoundingBoxOfSceneDirty

==========

La classe AlignedBox3D, qui vous est fournie, est pour
stocker une boîte dont les arêtes sont parallèles
aux axes x, y, et z (en anglais: "axis aligned box").
Une telle classe est utile pour stocker la
boîte englobante d'un ou plusieurs objets,
peut-être même de façon hiérarchique,
pour accélérer les tests d'intersection
ou de collision.

Dans une telle classe, il est utile d'avoir une méthode
getDiagonal():

```
   class AlignedBox3D {
      ...
      public Vector3D getDiagonal() { return Point3D.diff(p1,p0); }
      ...
   }
```

qui retourne la différence des coins avec coordonnées
minimales et maximales. Le vecteur retourné contient
toutes les dimensions de la boîte. On pourrait faire
la même chose pour des rectangles 2D

```
   class AlignedRectangle2D {
      ...
      public Vector2D getDiagonal() { return Point2D.diff(p1,p0); }
      ...
   }
```

et accéder aux dimensions facilement:

```
   AlignedRectangle2D rect = ...
   ...
   float width = rect.getDiagonal().x();
   float height = rect.getDiagonal().y();
```

Il est aussi utile de numéroter les 8 coins d'une boîte 3D
avec les indices suivants:

```
   0: (min_x, min_y, min_z)
   1: (max_x, min_y, min_z)
   2: (min_x, max_y, min_z)
   3: (max_x, max_y, min_z)
   4: (min_x, min_y, max_z)
   5: (max_x, min_y, max_z)
   6: (min_x, max_y, max_z)
   7: (max_x, max_y, max_z)
```

Si on écrit les indices 0 à 7 en binaire, on obtient 000, 001, ..., 111.
En binaire, le bit le moins significatif est 0 pour les coins
du côté négatif en x, et 1 pour les coins du côté positif en x.
De façon semblable, le bit qui est le deuxième moins significatif
est pour y, et le troisième est pour z.

Avec cette numérotation, il devient facile
de réaliser la méthode getCorner():

```
   public Point3D getCorner(int i) {
      return new Point3D(
         ((i & 1)!=0) ? p1.x() : p0.x(),
         ((i & 2)!=0) ? p1.y() : p0.y(),
         ((i & 4)!=0) ? p1.z() : p0.z()
      );
   }
```

Il devient également facile de réaliser
une méthode qui nous retourne le coin qui est
le plus loin dans la direction d'un vecteur donné:

```
   public Point3D getExtremeCorner( Vector3D v ) {
      return new Point3D(
         v.x() > 0 ? p1.x() : p0.x(),
         v.y() > 0 ? p1.y() : p0.y(),
         v.z() > 0 ? p1.z() : p0.z()
      );
   }

   public int getIndexOfExtremeCorner( Vector3D v ) {
      int returnValue = 0;
      if (v.x() > 0) returnValue |= 1;
      if (v.y() > 0) returnValue |= 2;
      if (v.z() > 0) returnValue |= 4;
      return returnValue;
   }
```

Finalement, si i est l'indice d'un coin,
l'indice du coin opposé en diagonale est simplement (i^7)

==========

Remarquez dans AlignedBox3D.intersects(),
on se sert d'une sphère englobante pour
couper court au test si le rayon n'intersecte
pas la sphère.

==========

Remarquez dans SceneViewer.updateHiliting(), on appelle
repaint() seulement lorsque l'objet à dessiner en surbrilliance
a changé. Encore une fois, c'est pour minimiser le travail
à faire, car un rendu peut prendre longtemps s'il y a beaucoup

de choses à dessiner.

==========