

Les bases de la construction d'une application 3D

par [Alain Mari](#)

Date de publication : 19/09/2005

Dernière mise à jour :

Dans ce second chapitre, nous allons aborder les bases d'une applications 3D, c'est à dire le repère 3D, le modèle graphe de scène propre à Java 3D ainsi que l'étude de classes utilitaires que nous utiliserons tout au long de cette partie consacrée à Java 3D

- 1 - Repère 3D
- 2 - Arborescence d'une scène 3D
 - 2.1 - Les différents éléments d'un arbre
 - 2.2 - La compilation
 - 2.3 - Les capacités d'un noeud
- 3 - Classes utilitaires
 - 3.1 - La classe Point3f
 - 3.2 - La classe Color3f
 - 3.3 - La classe Vector3f
- IV - En préparation
- V - Téléchargements

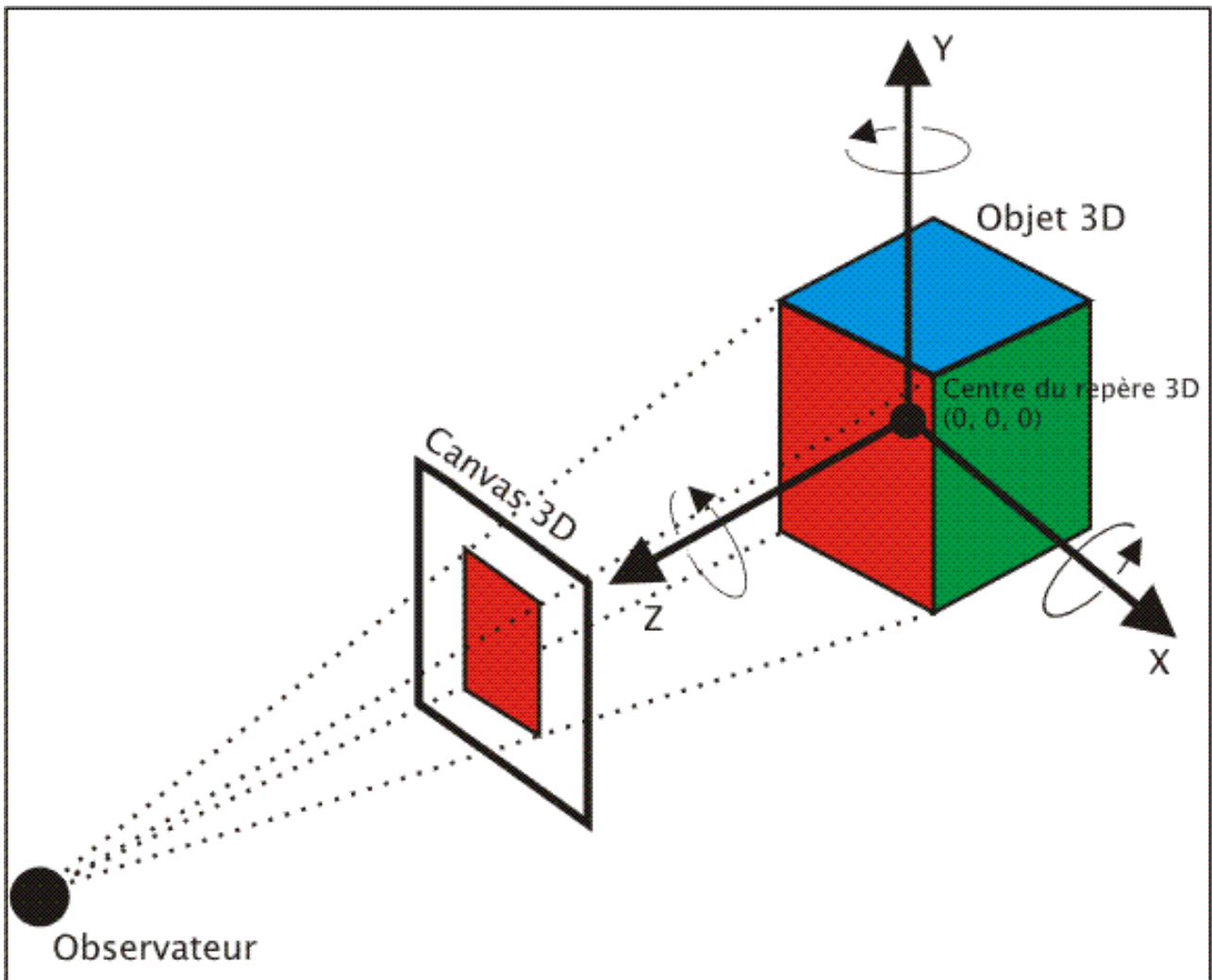
1 - Repère 3D

Dans une application 3D, tous les objets sont positionnés dans l'espace grâce aux coordonnées (x,y,z) des points qui les constituent.

Cependant, le positionnement des objets 3D n'est pas tout, il faut aussi pouvoir les visualiser. Pour cela, il est nécessaire de positionner le canvas 3D par rapport à l'objet 3D ainsi qu'un point d'observation, ou caméra. Nous rappelons que le canvas3D sert à représenter des objets 3D dans une fenêtre à deux dimensions qui est celle affichée à l'écran.

Lorsqu'on utilise la classe SimpleUniverse pour initialiser notre scène 3D, l'axe X est orienté vers la droite, l'axe Y vers le haut (et non vers le bas comme dans le cas des applications 2D) et l'axe Z est orienté vers l'observateur. L'observateur, le centre du canvas 3D et le centre de repère 3D sont alignés.

On peut représenter le positionnement relatif de ces éléments selon le schéma suivant :



Les flèches indiquent le sens positif de rotation (sens direct) autour de chaque axe.

Le code suivant positionne le point d#observation à une distance de 2.41 mètres du centre du repère 3D. Le canvas 3D est ainsi placé de façon à ce que la face avant d#un cube de 2 mètres d#arête centrée en (0, 0, 0) soit intégralement visible par l#observateur. C'est le paramétrage par défaut de la classe SimpleUniverse :

```
simpleU.getViewingPlatform().setNominalViewingTransform()
```

2 - Arborescence d'une scène 3D

2.1 - Les différents éléments d'un arbre

En Java 3D, une scène 3D se construit selon une arborescence bien précise que l'on appelle un graphe. Un graphe est constitué de noeuds (node en anglais) qui sont reliés entre eux par une relation parent-enfant (parent-child) et qui forment une structure appelée arbre. Il y a deux sortes de noeuds : les groupes (group) qui peuvent avoir un ou plusieurs enfants mais seulement un parent, et les feuilles (leaf) qui n'ont qu'un seul parent et pas d'enfants. Les classes Group et Leaf étendent logiquement la classe Node. C'est la méthode addChild(Node noeud) de la classe Group qui permet d'ajouter un enfant à un groupe.

On retiendra que les noeuds, quels qu'ils soient, ne peuvent avoir qu'un seul et unique parent.

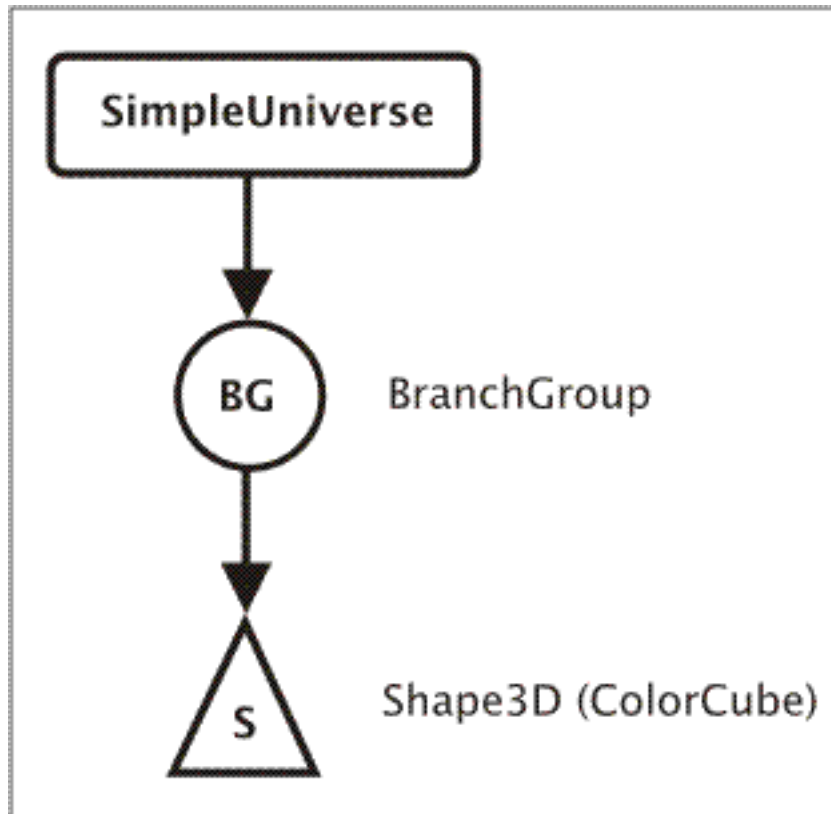
Les types de groupes les plus utilisés en Java 3D sont représentés par les classes : BranchGroup, Primitive, TransformGroup. Ces trois classes dérivent toutes de la classe Group. Un objet de type BranchGroup désigne soit la racine de l'arbre (dans ce cas il n'a pas de parent), soit un groupe d'objets intermédiaires entre la racine et les feuilles. Il faut toujours garder à l'esprit qu'il n'existe qu'un unique chemin entre la racine de l'arbre et chacune de ses feuilles.

La classe Primitive sera étudiée dans le chapitre consacré aux objets 3D tandis que la classe TransformGroup sera étudiée en détail dans le chapitre dédié aux transformations géométriques.

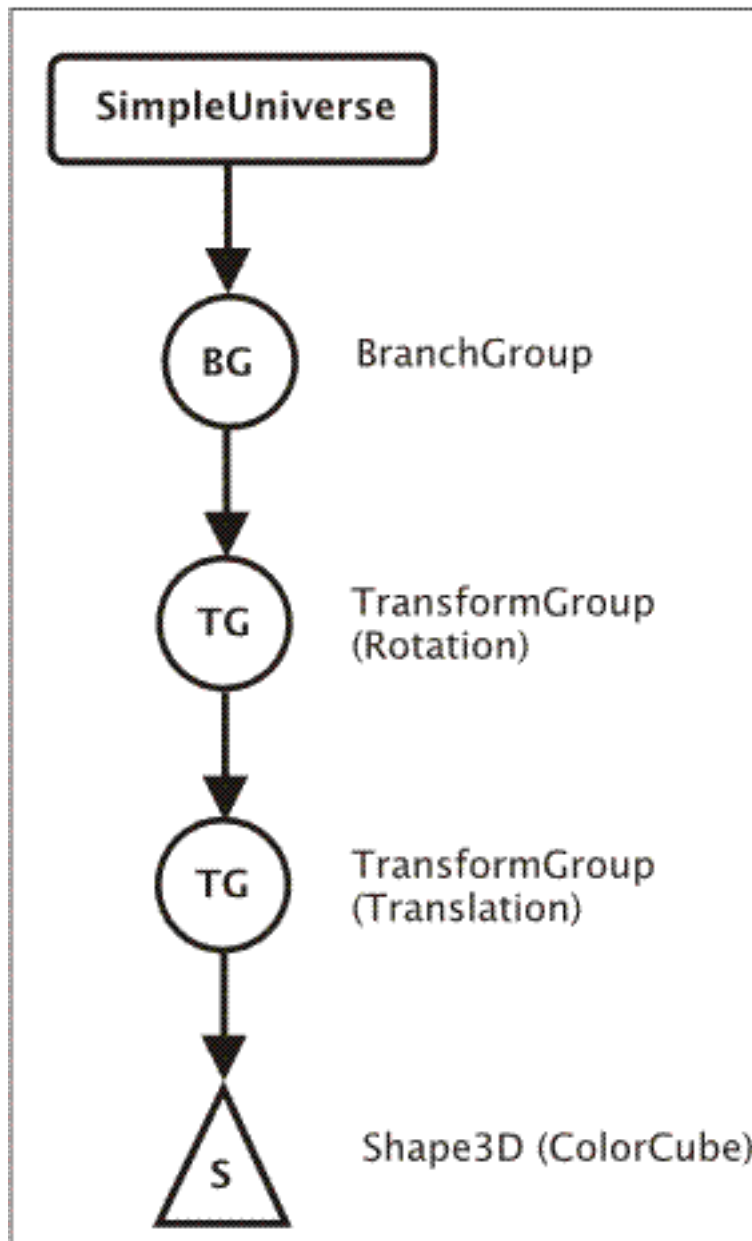
Les types de feuilles les plus utilisés en Java 3D sont représentés par les classes : BackGround, Behavior, Light, Shape3D. Ces quatre classes dérivent toutes de la classe Leaf.

Les classes BackGround et Shape3D sont étudiées dans le chapitre consacré aux objets 3D, Behavior dans le chapitre Interaction et Light dans le chapitre Eclairage.

Lorsqu'on représente schématiquement l'arborescence d'une scène 3D, les noeuds de type Group sont toujours représentés schématiquement par des cercles et les noeuds de type Leaf par des triangles. Voici, à titre d'exemple, l'arborescence correspondant à notre premier petit programme 3D traçant un cube multicolore :

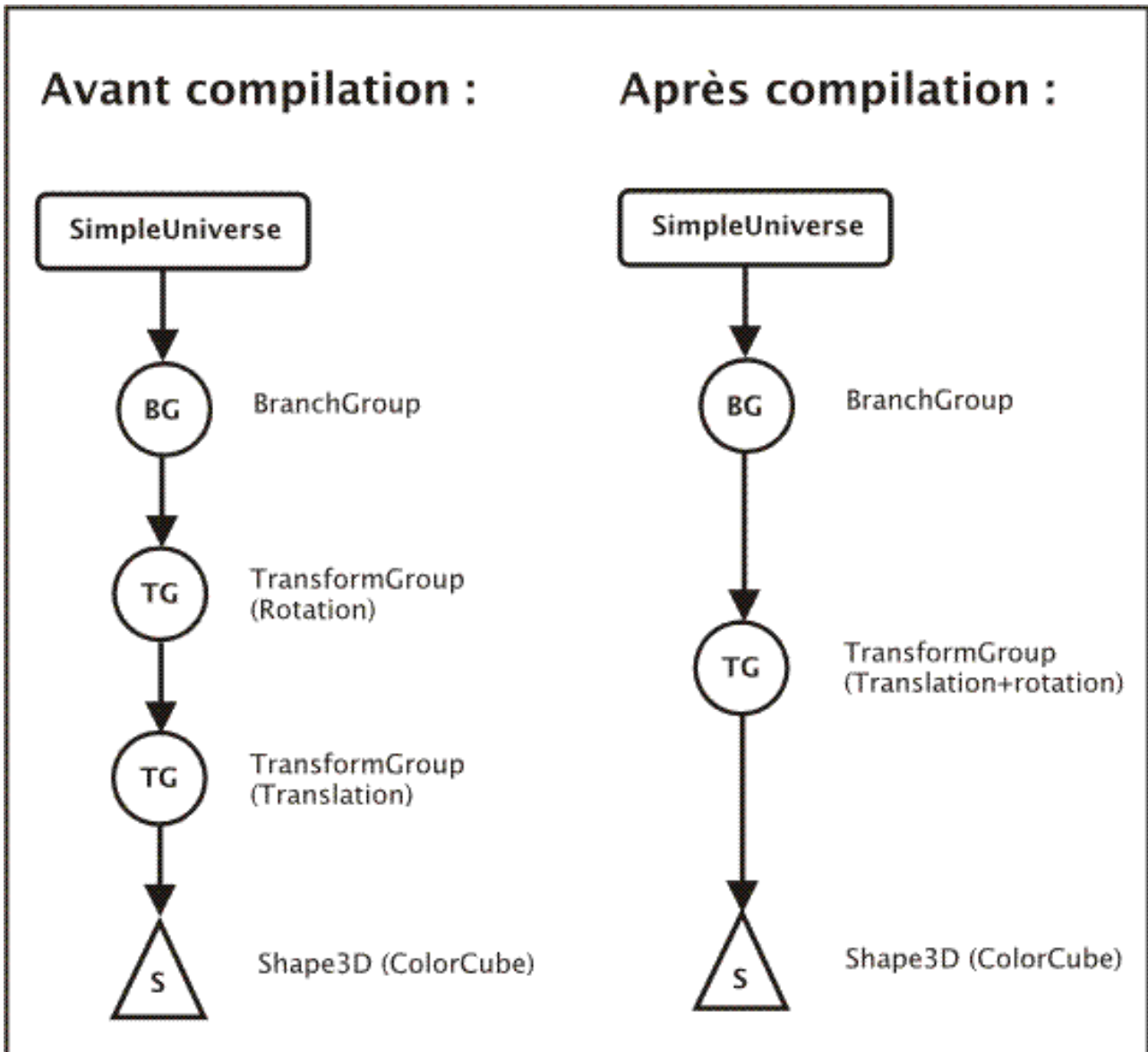



Voici un autre exemple d'arbre correspondant à une scène 3D affichant un objet ayant subi deux transformations géométriques au préalable. Nous verrons dans le chapitre consacré aux transformations que celles - ci sont toujours appliquées dans un ordre bien précis : de la feuille vers la racine. Dans notre exemple, c'est la translation qui est d'abord appliquée à l'objet ColorCube puis la rotation ensuite. L'ordre d'exécution des transformations est primordial car nous n'obtiendrions pas du tout le même résultat si la rotation était appliquée en premier :



2.2 - La compilation

La classe BranchGroup possède une méthode compile(). L'appel de cette méthode convertit toute la branche de l'arbre située sous le noeud BranchGroup en une représentation interne qui est optimisée pour le moteur de rendu afin d'avoir un affichage rapide. Les optimisations effectuées par cette méthode compile() sont multiples, l'une d'entre elles consiste à transformer plusieurs noeuds de type TransformGroup consécutifs en un seul objet TransformGroup. La transformation 3D résultante n'en sera que plus performante :



 Il est vivement conseillé d'utiliser systématiquement la méthode compile() sur un noeud de type BranchGroup afin d'améliorer la rapidité d'affichage d'une scène 3D.

2.3 - Les capacités d'un noeud

Tout noeud appartenant à un graphe qui a lui-même été attaché à un univers de type SimpleUniverse par la méthode addBranchGraph() est considéré comme vivant. Tout noeud de type BranchGroup auquel on a appliqué la méthode compile() est considéré comme compilé.

Il arrive très fréquemment dans une application 3D que nous ayons à interroger ou modifier une propriété d'un noeud vivant ou compilé. Par exemple, un objet peut être initialement construit avec une géométrie, une couleur et une apparence bien particulières puis une interaction de l'utilisateur peut par exemple demander à changer la forme, la couleur ou l'apparence de cet objet.

Or il est impossible de changer les propriétés d'un tel objet si celui-ci est vivant ou compilé sans avoir au préalable défini les capacités (capabilities en anglais) de cet objet. Le fait de définir correctement les capacités d'un noeud va permettre d'en modifier les propriétés même une fois celui-ci vivant ou compilé.

C'est la méthode `setCapability(int bits)` de la classe `SceneGraphObject` qui permet de régler les capacités d'un noeud.

La classe `SceneGraphObject` est un peu la super classe de toutes les classes de Java 3D, un peu comme la classe `Object` en Java.



La méthode `setCapability()` doit être impérativement appelée avant que le noeud ne soit rendu vivant (par l'appel à la méthode `addBranchGraph()` de la classe `SimpleUniverse`) ou compilé (par l'appel à la méthode `compile()` de `BranchGroup`)

Le paramètre `bits` de la méthode `setCapability()` peut prendre les valeurs des champs statiques des noeuds auxquels on veut appliquer cette méthode. La valeur de ces champs commence toujours par `ALLOW_`. On peut citer le cas des objets géométriques que nous étudierons un peu plus loin. La plupart des capacités liées aux objets géométriques ont pour valeur les champs statiques de la classe `GeometryArray` commençant par `ALLOW_`.

Par exemple, pour changer la forme d'un objet géométrique déjà vivant ou compilé, il faudra au préalable appeler la méthode `setCapability(ALLOW_COORDINATES_WRITE)` sur cet objet. Pour changer sa couleur, il faudra appeler la méthode `setCapability(ALLOW_COLOR_WRITE)`.

On peut s'interroger sur l'utilité qu'ont eue les concepteurs de Java 3D à introduire la notion de capacité en regard de la complexité au niveau de la programmation que cela ajoute. En fait, une fois que la racine de l'arbre d'une scène 3D est rattachée à un univers par la méthode `addBranchGraph()`, Java 3D va vérifier la capacité de chacun des noeuds de la scène grâce à la méthode `getCapability()` de la super classe `SceneGraphObject`. Cela va permettre d'optimiser encore davantage la scène 3D avant de l'envoyer au moteur de rendu pour affichage.

3 - Classes utilitaires

Java 3D fournit des classes utilitaires pour représenter des points, des couleurs, des vecteurs ou des matrices par exemple. Comme ces classes peuvent avoir des domaines d'applications autres que la 3D pure, elles sont regroupées dans un package indépendant : `javax.vecmath`.

Parmi toutes ces classes utilitaires, celles que nous utiliserons le plus sont les points, les couleurs et les vecteurs. Les deux premières sont notamment utilisées pour construire des objets géométriques en couleur et la troisième est utilisée en particulier dans le cadre des transformations géométriques. Nous allons décrire plus en détail les classes `Point3f`, `Color3f` et `Vector3f`.

Il est à noter que ces trois classes dérivent toutes d'une classe générique nommée `Tuple3f`.

3.1 - La classe `Point3f`

Il existe différentes classes pour représenter un point avec Java 3D. Parmi elles, la classe `Point3f` qui sert à construire un point dont les trois coordonnées (x,y,z) sont représentées par des nombres flottants (d'où le 3f de `Point3f`).

Champs (hérités de la classe `Tuple3f`)

```
public float x
//Coordonnée X dans le repère 3D

public float y
//Coordonnée Y dans le repère 3D

public float z
//Coordonnée Z dans le repère 3D
```

Principal constructeur

```
public Point3f(float x, float y, float z)
//Construit un point Point3f à partir de ses trois coordonnées x, y et z.
```

Principales méthodes

```
public final float distance(Point3f p1)
//Renvoie la distance entre le point courant et le point p1

public final void set(Tuple3f t1) (méthode héritée de la classe Tuple3f)
//Mets à jour les coordonnées du point Point3f avec celles de t1
```

3.2 - La classe `Color3f`

Il existe différentes classes pour représenter une couleur avec Java 3D. Parmi elles, la classe `Color3f` qui sert à définir une couleur dont les trois composantes RGB (x,y,z) sont représentées par des nombres flottants variant de 0 à 1 inclus (d'où le 3f de `Color3f`).

Champs (hérités de la classe `Tuple3f`)

```
public float x
//Composante rouge dans la base RGB des couleurs. x varie de 0 à 1 inclus.
//Dans un système de couleurs à 8 bits (256 niveaux) par composante, x varie de 0/256 à 256/256.

public float y
//Composante verte dans la base RGB des couleurs. y varie de 0 à 1 inclus.
//Dans un système de couleurs à 8 bits (256 niveaux) par composante, y varie de 0/256 à 256/256.

public float z
//Composante bleue dans la base RGB des couleurs. z varie de 0 à 1 inclus.
```

Champs (hérités de la classe Tuple3f)

```
//Dans un système de couleurs à 8 bits (256 niveaux) par composante, z varie de 0/256 à 256/256.
```

Principal constructeur

```
public Color3f(Color color)
//Construit une couleur Color3f à partir d'un objet Color de AWT.

public Color3f(float x, float y, float z)
//Construit une couleur Color3f à partir de ses trois composantes (x,y,z) dans la base RGB
```

Principales méthodes

```
public final void set(Color color)
//Modifie l'objet Color3f avec la couleur color (objet de type Color de AWT)
```

3.3 - La classe Vector3f

Il existe différentes classes pour représenter un vecteur avec Java 3D. Parmi elles, la classe Vector3f qui sert à construire un vecteur au sens mathématique du terme dont les trois composantes (x,y,z) sont représentées par des nombres flottants (d'où le 3f de Vector3f).

Champs (hérités de la classe Tuple3f)

```
public float x
Composante X du vecteur dans le repère 3D

public float y
Composante Y du vecteur dans le repère 3D

public float z
Composante Z du vecteur dans le repère 3D
```

Principal constructeur

```
public Vector3f(float x, float y, float z)
//Construit un vecteur Vector3f à partir de ses trois composantes x, y et z.
```

Principales méthodes

```
public final float length()
//Renvoie la valeur de la norme du vecteur Vector3f

public final void normalize()
//Normalise le vecteur Vector3f

public final void set(Tuple3f t1) (méthode héritée de la classe Tuple3f)
//Mets à jour les composantes du vecteur Vector3f avec celles de t1
```

IV - En préparation

Le chapitre 3 de ce tutoriel sera intitulé les Objets 3D et couvrira : Arrière-plans, Formes de base, Objets à géométrie complexe, Texte 3D

V - Téléchargements

[Article au format PDF \(mirroir\)](#)

[Article HTML/ZIP \(mirroir\)](#)