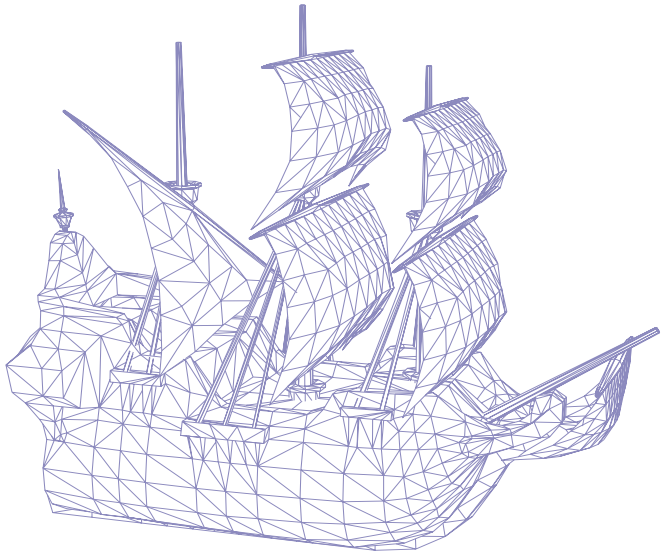
# Initiation à l'API Java 3D™

## Un tutorial pour les débutants

## Chapitre 3
## Création facile de volumes



Dennis J Bouvier / K Computing
Traduction Fortun Armel

Le Tutorial de Java 3D v1.5 (API Java 3D v1.1.2)
> traduction fr. version 0.1 <

This documentation was prepared for Sun Microsystems by K Computing (530 Showers Drive, Suite 7-225, Mountain View, CA 94040, 770-982-7881, www.kcomputing.com). For further information about course development or course delivery, please contact either Sun Microsystems or K Computing.

# Table des matières

# Blocs de références

# Figures et tableaux

# Fragments de code



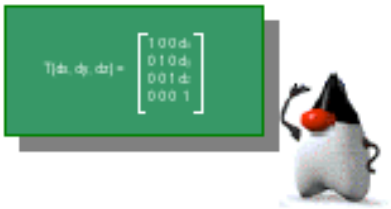
**Préface au chapitre 3**

Ce document est la troisième partie d'un tutorial sur l'utilisation de l'API Java 3D. Les chapitres supplémentaires, la préface et les annexes de cet ensemble sont présentés dans le Chapitre 0 disponible à :

Version originale de Sun Microsystems — http://java.sun.com/products/javamedia/3d/collateral

Version française — http://perso.wanadoo.fr/armel.fortun/

# Chapitre 3
# Création facile de volumes



### Objectifs du chapitre

Après la lecture de ce chapitre, vous serez capable :

- D'utiliser les classes de chargement pour importer des géométries depuis des fichiers dans des mondes Java 3D.
- D'utiliser le GeometryInfo pour déterminer la géométrie par des polygones arbitraires.
- D'utiliser le Text2D pour ajouter du texte aux mondes Java 3D.
- D'utiliser le Text3D pour ajouter du texte géométrique aux mondes Java 3D.
- De déterminer les couleurs, images, ou géométries pour l'arrière-plan.
- D'utiliser le BoundingLeaf pour spécifier les limites pour les arrières-plans, les comportements, et les lumières.
- D'utiliser le champ UserData de la classe SceneGraphObject pour des applications diverses.

Comme troisième chapitre du module « Initiation à Java 3D », le Chapitre 3 présente des techniques plus simples pour la création de volumes visuels. Les chapitres un et deux présentaient les méthodes de base pour la création de mondes virtuels, incluant la création d'objets visuels depuis les classes de géométrie. Il suffit de programmer un peu pour apprendre que la création de volumes visuels complexes, triangle par triangle, est quelque peu rébarbative. Heureusement, il y a une variété de méthodes pratiques pour la production de volumes visuels. Ce chapitre étudie une partie des méthodes de création de volume et aborde thème du volume en allant plus loin que la création de géométries simples.

## 3.1 Contenu de ce chapitre

Si vous désirez créer un objet visuel de grande taille ou complexe, une grande quantité de programmation sera nécessaire pour la seule spécification des coordonnées et des normales. Si vous êtes intéressé par la performance, vous dépenserez plus de temps, et de programmation, à déterminer la géométrie par le plus petit nombre possible de bandeaux de triangles. La programmation de géométries, exigeante en détails, peut être un grand précipice pour votre temps de développement. Heureusement, il y a des méthodes pour la création d'objets visuels qui requièrent moins de programmation, aboutissant à moins d'erreurs, et prenant bien souvent moins de temps.

La Partie 3.2 présente les classes de chargement de volume, ou « Loaders » (chargeurs) comme ils sont communément nommés. Les chargeurs, alternative à la géométrie codé à la main, créent des objet visuels Java 3D depuis des fichiers crées par un logiciel de modelage 3D. Les chargeurs existent aujourd'hui pour les fichier Alias `obj`, les fichiers `VRML`, les fichiers Lightwave, les fichiers `dfx` d'Autocad, et une variété d'autres formats de fichier 3D. De nouveaux chargeurs sont bien sûr en développement. La caractéristique la plus importante étant de pouvoir écrire des chargeurs sur mesure pour Java 3D.

La Partie 3.3 présente la classe utilitaire GeometryInfo, une autre alternative au codage manuel de la géométrie. GeometryInfo, avec les classes Triangulator, Stripifier, et NormalGenerator vous permettra de déterminer la géométrie d'un objet visuel par une polygonalisation arbitraire. Ces classes convertissent les polygones en triangles, forment des bandeaux de triangles, et calculent les normales des triangles à l'exécution, vous évitant potentiellement de passer trop de temps à coder.

Les trois parties suivantes présentent les techniques spécifiques de la création de volume. Les Parties 3.4 et 3.5 présentent respectivement les classe utilitaires Text2D et Text3D. Ces deux classes représentent deux voies pratiques pour l'ajout de texte aux contenus de vos mondes virtuels. La Partie 3.6 présente la classe Background. La classe Background vous permet de spécifier une couleur, une image ou une géométrie comme arrière-plan d'un monde virtuel.

Les deux parties suivantes n'ont plus grand chose à voir avec le volume. Cependant, ce sont des sujets importants. La Partie 3.7 présente la classe BoundingLeaf. Les objets BoundingLeaf sont utiles en combinaison avec les arrières-plans, les comportements, les lumières, le brouillard, et toutes les autres classes Java 3D qui requièrent des spécifications de limites. La Partie 3.8 aborde l'usage du champ UserData de la classe SceneGraphObject.

Bien sûr, le Chapitre se termine par un résumé et des exercices de Tests-personnels pour les audacieux.

# 3.2 Les chargeurs

Une classe de chargement lit des fichiers de scènes 3D (pas des fichiers Java 3D) et crée des représentations Java 3D du contenu de ceux-ci, qui pourront être ajoutés de manière sélective à un monde Java 3D et complétés par d'autre code Java 3D. Le package utilitaire `com.sun.j3d.loaders` fournit les moyens au chargement de volumes depuis des fichiers crées dans d'autres applications 3D à l'intérieur d'une application Java 3D. Les classes de chargement appliquent l'interface de chargement définie dans le package `com.sun.j3d.loaders`.

Étant donné qu'il y a une grande variétés de formats de fichiers ayant pour but de représenter des scènes 3D (e.g. ., `*.obj, *.vrml`, etc.) et qu'il y aura toujours plus de formats, le véritable code pour charger un fichier ne fait pas partie de Java 3D ou du package Loader ; seule l'interface pour le mécanisme de chargement est incluse. Avec la définition de cette interface, l'utilisateur de Java 3D peut développer des classes de chargement ayant la même interface que les autre classes de chargement.

## 3.2.1 Exemple simple d'utilisation d'un chargeur.

Sans une classe effectuant la lecture d'un fichier, il n'est pas possible de charger le volume depuis celui-ci. Avec une classe de chargement, c'est facile. La Figure 3-1 présente la recette d'utilisation d'un chargeur.

---

0. Trouver un chargeur (si il n'y en a pas de valable, en écrire un : voir la Partie 3.2.4).
1. Importer la classe de chargement pour votre format de fichier.(voir la Partie 3.2.2 pour trouver un chargeur.)
2. Importer les autres classes nécessaires.

3. Déclarer une variable de Scene (ne pas utiliser le constructeur).
4. Créer un objet chargeur.
5. Charger le fichier dans un bloc de vérification, assigner le résultat à la variable de Scene.
6. Insérer la Scene dans le graphe scénique.

**Figure 3-1 Recette pour l'usage d'un chargeur.**

Un exemple de chargeur basé sur la classe ObjectFile est distribué avec le JDK1.2. Il se trouve dans le répertoire `jdk1.2/demo/java3d/ObjLoad`. Le Fragment de code 3-1 présente un extrait du code de cette exemple.

La classe ObjectFile est distribué dans le package com.sun.j3d.loaders comme un exemple de chargeur de fichier. D'autres chargeurs sont disponibles (quelques uns sont listés dans le Tableau 3-1).

**La classe ObjectFile**

Package : `com.sun.j3d.loaders`

Implémente : `Loader`

La classe ObjectFile exécute l'interface de Loader pour le format de fichier Wavefront *.obj, un format de fichier 3D standard crée pour utiliser avec le Wavefront's Advanced Visualizer™. Les Objets Fichiers sont des fichiers à base de texte maintenant à la fois des géométries polygonales et de formes-libres (courbes et surfaces). Le chargeur Java 3D de fichier *.obj supporte un sous-ensemble du format de fichier, mais il est assez complet pour charger presque tous les Fichiers Objet les plus communs. Les géométries en forme-libre ne sont pas supportées.

Le Fragment de code 3-1 est annoté avec les nombres correspondant à l'usage de la recette de chargement donnée à la Figure 3-1.

```
1. import com.sun.j3d.loaders.objectfile.ObjectFile;          ❶
2. import com.sun.j3d.loaders.ParsingErrorException;          ❷
3. import com.sun.j3d.loaders.IncorrectFormatException;       ❷
4. import com.sun.j3d.loaders.Scene;                          ❷
5. import java.applet.Applet;
6. import javax.media.j3d.*;
7. import javax.vecmath.*;
8. import java.io.*;                                          ❷
9.
10. public class ObjLoad extends Applet {
11.
12.    private String filename = null;
13.
14.    public BranchGroup createSceneGraph() {
15.         // Crée la racine de la branche du graphe
16.         BranchGroup objRoot = new BranchGroup();
17.
18.    ❸    ObjectFile f = new ObjectFile();
19.    ❹    Scene s = null;
20.    ❺    try {
21.              s = f.load(filename);
22.         }
23.         catch (FileNotFoundException e) {
```

```
24.     |   System.err.println(e);
25.     |   System.exit(1);
26.     |   }
27.     |   catch (ParsingErrorException e) {
28.     |   System.err.println(e);
29.     |   System.exit(1);
30.     |   }
31.     |   catch (IncorrectFormatException e) {
32.     |   System.err.println(e);
33.     \   System.exit(1);
34.         }
35.
36.     ❻   objRoot.addChild(s.getSceneGroup());
37.     }
```

**Fragment de code 3-1 Un extrait de jdk1.2/demo/java3d/ObjLoad/ObjLoad.java.**

Ce programme est complèté par l'ajout des comportements (la rotation par défaut, ou l'interactivité de la souris - abordée au Chapitre 4) et des lumières (Chapitre 6) pour fournir un rendu illuminé du modèle de l'objet. Bien sûr, vous pouvez beaucoup d'autres choses avec le modèle dans un programme Java 3D comme des animations, l'ajout de géométries, la modification de la couleur du modèle, et ainsi de suite.

Un exemple de chargeur pour Lightwave est fournit avec la distribution du JDK 1.2 et ce trouve à `jdk1.2/demos/java3d/lightwave/Viewer.java`. Ce chargeur effectuera le chargement des lumières et des animations définies dans un fichier `*.lws` de Lightwave

## 3.2.2 Chargeurs du domaine publique

Il existe un variété de classes de chargement pour Java 3D. Le Tableau 3-1 liste les formats de fichier pour lesquels les chargeurs sont disponibles au public. Au moment d'écrire ceci, au moins un chargeur est disponible pour chaque formats de fichier listés au Tableau 3-1.

| Format de fichier | Description |
|---|---|
| 3DS | 3D-Studio |
| COB | Caligari trueSpace |
| DEM | Digital Elevation Map |
| DXF | Fichier d'Échange de dessin AutoCAD |
| IOB | Imagine |
| LWS | Lightwave Scene Format |
| NFF | WorldToolKit NFF format |
| OBJ | Wavefront |
| PDB | Protein Data Bank |
| PLAY | PLAY |
| SLD | Solid Works (fichiers prt et asm) |
| VRT | Superscape VRT |
| VTK | Visual Toolkit |
| WRL | Virtual Reality Modeling Language |

**Tableau 3-1 Chargeurs Java 3D publiques disponibles.**

Pour une liste à jour des classes de chargement, jetez un coup d'œil sur le web. Les classes de chargement peuvent être téléchargés depuis le web. Les chargeurs peuvent être trouvés en suivant les liens depuis la page d'accueil de Java 3D, ou en allant à la partie de ce tutorial consacrée aux références pour les adresses web (Chapitre 0).

### 3.2.3 Interfaces du package Loader et classes de base

Les designers de Java 3D ont rendu facile l'écriture d'un chargeur [1], il existe donc un certain nombre et une grande variété de chargeurs. Les classes Loaders mettent en œuvre l'Interface (groupe de méthodes) de la classe Loader, laquelle diminue le niveau de difficulté dans l'écriture d'un chargeur. Plus important, l'usage d'interfaces rend les différentes classes de chargement cohérentes avec leur interface. Comme dans l'exemple, un programme chargeant un fichier 3D utilise actuellement à la fois un objet Loader et un objet Scene. Le Loader lit, analyse, et crée la représentation Java 3D du contenu du fichier. L'objet Scene stocke le graphe scénique crée par le Loader. Il est possible de charger des scènes depuis plusieurs fichiers (du même format) utilisant le même objet Loader créant de multiples objets Scene. Des fichiers de formats différents peuvent être combinés dans un programme Java 3D en utilisant les classes de chargement adéquate.

Le Bloc de référence suivant liste les interfaces du package `com.sun.j3d.loaders`. Un Loader applique l'interface de chargement et utilise une classe qui applique une interface de Scene.

| **Résumé de l'Interface com.sun.j3d.loaders** | |
|---|---|
| **`Loader`** | L'interface de Loader est utilisée pour déterminer l'emplacement et les éléments à charger depuis un format de fichier. |
| **`Scene`** | L'interface de Scene est un ensemble de méthodes utilisées pour extraire les informations du graphe scénique Java 3D depuis un utilitaire de chargement de fichier. |

En plus des interfaces, le package com.sun.j3d.loaders fournit les constructions de base des interfaces.

| **Résumé des classes de com.sun.j3d.loaders** | |
|---|---|
| **`LoaderBase`** | Cette classe exécute l'interface de Loader et y ajoute des constructeurs. Cette classe est éten due par les auteurs de classes de chargement spécifiques. |
| **`SceneBase`** | Cette classe exécute l'interface de Scene et l'étend afin d'incorporer les méthodes utilisées par les chargeurs. |

Les méthodes définies dans l'interface du Loader sont employées par les programmeurs utilisant les classes de chargement.

**Résumé des méthodes de l'interface de Loader**

Package : `com.sun.j3d.loaders`

L'interface de Loader est utilisée pour déterminer l'emplacement et les éléments à charger depuis un format de fichier. L'interface est utilisée pour donner aux chargeurs de formats de fichiers différents une interface publique commune. Idéalement l'interface de Scene serait appliquée pour donner à l'utilisateur une interface homogène pour extraire les données.

[1] Posséder l'interface de chargement et les classes de base permet actuellement d'écrire facilement un chargeur pour un format de fichier simple. Cela rend aussi possible l'écriture d'un chargeur pour un format de fichier complexe.

**`Scene load(java.io.Reader reader)`**
Cette méthode charge le Reader et renvoie l'objet Scene contenant la scène.

**`Scene load(java.lang.String fileName)`**
Cette méthode charge le fichier appelé et renvoie l'objet Scene contenant la scène.

**`Scene load(java.net.URL url)`**
Cette méthode charge le fichier appelé et renvoie l'objet Scene contenant la scène.

**`void setBasePath(java.lang.String pathName)`**
Cette méthode fixe le nom du chemin de base pour les fichiers de données associées avec le fichier transmit dans la méthode load(String).

**`void setBaseUrl(java.net.URL url)`**
Cette méthode fixe le nom de l'URL de base pour les fichiers de données associées avec le fichier transmit dans la méthode load(URL).

**`void setFlags(int flags)`**
Cette méthode détermine les flags de chargement pour le fichier :

**`LOAD_ALL`**
Ce flag autorise le chargement de tous les objets dans la scène.
**`LOAD_BACKGROUND_NODES`**
Ce flag autorise le chargement des objets background dans la scène.
**`LOAD_BEHAVIOR_NODES`**
Ce flag autorise le chargement de comportements dans la scène.
**`LOAD_FOG_NODES`**
Ce flag autorise le chargement d'objets brouillard dans la scène.
**`LOAD_LIGHT_NODES`**
Ce flag autorise le chargement d'objets lumière dans la scène.
**`LOAD_SOUND_NODES`**
Ce flag autorise le chargement d'objets son dans la scène.
**`LOAD_VIEW_GROUPS`**
Ce flag autorise le chargement d'objets de visualisation (caméra) dans la scène.

La classe LoaderBase fournit des outils pour chacune des trois méthodes `load()` de l'interface du Loader. LoaderBase exécute aussi deux constructeurs. Notez que les trois méthodes du chargeur renvoient un objet Scene.

**Résumé des constructeurs de LoaderBase**

Package: `com.sun.j3d.loaders`

Implémente: `Loader`

Cette classe implémente l'interface de Loader. L'auteur d'un fichier de chargement voudra étendre cette classe. L'utilisateur d'un fichier de chargement voudra utiliser ces méthodes.

**`LoaderBase()`**
Construit un Loader avec les valeurs par défaut pour toutes les variables.

**`LoaderBase(int flags)`**
Construit un Loader avec le mot spécifié comme flag.

En plus des constructeurs listés dans le Bloc de référence ci-dessus, les méthodes listées dans le Bloc de référence suivant sont utilisées par les programmeurs utilisant n'importe quelle classe de chargement.

**Résumé des méthodes de SceneBase (liste partielle : méthodes pour l'utilisateur de chargeur)**

Par une déviation de la mise en forme normale d'un Bloc de référence, ce Bloc de référence liste des méthodes.

```
Background[] getBackgroundNodes()
Behavior[] getBehaviorNodes()
java.lang.String getDescription()
Fog[] getFogNodes()
float[] getHorizontalFOVs()
Light[] getLightNodes()
java.util.Hashtable getNamedObjects()
BranchGroup getSceneGroup()
Sound[] getSoundNodes()
TransformGroup[] getViewGroups()
```

## 3.2.4 Écriture d'un Loader

Comme il mentionné ci-dessus, une des caractéristiques la plus importante des chargeur est la possibilité d'en écrire un vous-même — ce qui veut dire que tous les utilisateurs de Java 3D le peuvent aussi ! L'espace et le temps ne me permet pas, ici, d'aller plus loin dans les détails de l'écriture d'un chargeur. Toutefois, le but de cette partie est d'en esquisser le processus. Si vous n'avez pas l'intention d'écrire un chargeur, du moins pas maintenant, vous pouvez passer la prochaine partie.

En écrivant un chargeur, l'auteur de la nouvelle classe de chargement étends la classe LoaderClass définie dans le package `com.sun.j3d.loaders`. La nouvelle classe de chargement utilise la classe SceneBase du même package.

Il y aura probablement pour les futurs chargeurs un petit désir de sous-classer SceneBase, ou d'implémenter directement Scene, la fonctionnalité d'un SceneBase étant assez directe. La classe SceneBase est à la fois responsable du stockage et du renvoi des données crées par un chargeur au moment de la lecture d'un fichier. Les méthodes de stockage (utilisées seulement par les auteurs de Loader) sont toute les routines `add*`. Le méthodes de renvoi (utilisées essentiellement par les utilisateurs de Loader) sont toutes les routines `get*`.

Écrire un chargeur de fichier peut être un peu complexe en fonction de la complexité du format de fichier. La partie la plus ardue étant d'effectuer l'analyse du fichier. Bien sûr, vous devrez commencer par étudier la documentation du format de fichier pour lequel vous voudrez écrire une classe de chargement. Une fois que le format sera compris, commencez par lire les classes de base de chargeur et de scène.

En étendant la classe de chargement de base, la tache la plus importante sera d'écrire les méthodes qui reconnaîtront les différents types de contenus qui pourront être représentés dans le format de fichier. Chacune de ces méthodes créant ensuite le composant graphe scénique Java 3D correspondant et le plaçant dans l'objet scène. Le constructeur de SceneBase est décrit dans le Bloc de référence suivant. D'autres Blocs de référence pertinents sont présentés dans la partie précédente.

**Résumé du constructeur de SceneBase**

Package: `com.sun.j3d.loaders`

Implémente: `Scene`

Cette classe implémente l'interface de Scene et l'étends pour incorporer les utilitaires pouvant être utilisés par les chargeurs.

Cette classe est responsable à la fois du stockage et du renvoi des données depuis la Scene.

**`SceneBase()`**
Crée un objet SceneBase — il n'y a aucune raison d'utiliser ce constructeur excepté dans la réalisation d'une nouvelle classe de chargement.

# 3.3 GeometryInfo

Si vous n'avez pas accès à des fichiers de modèles géométriques, ou à des logiciels de modélisation de volumes, vous devrez créer vos géométries à la main. Comme il est mentionné dans l'introduction du chapitre, la géométrie codée à la main requiers souvent plus de temps et est une activité entraînant des erreurs. Comme vous le savez, quand vous déterminez une géométrie à travers les classes racines [core classes], vous êtes limités au triangles et aux carrés. Utiliser la classe utilitaire GeometryInfo peut alléger le temps et l'ennui à la création de géométries. À la place de spécifier chaque triangle, vous pouvez déterminer des polygones arbitraires, lesquels peuvent être des polygones concaves et non-plan — même avec des trous [2]. L'objet GeometryInfo, et les autres classes utilitaires, convertissent la géométrie en une géométrie triangulaire que Java 3D pourra rendre.

Par exemple, si vous voulez créer une voiture dans Java 3D, à la place de décrire les triangles, vous pouvez définir le profil de la voiture par un polygone dans un objet GeometryInfo. Puis, en utilisant un objet Triangulator, le polygone pourra être subdivisé en triangles. L'image de gauche de la Figure 3-2 montre un profile de voiture comme un polygone. L'image de droite montre le polygone subdivisé en triangles [3].

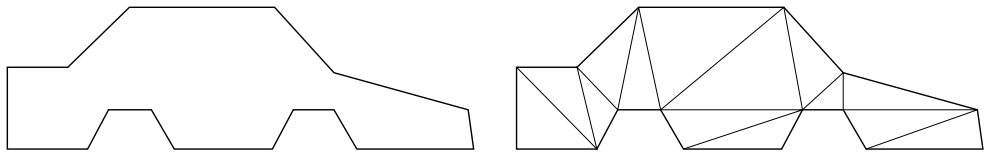


**Figure 3-2 Un polygone GeometryInfo et une triangulation possible.**

Si vous êtes intéressé par la performance, qui ne l'est pas, utilisez un objet Stripifier pour convertir les triangles en bandeaux de triangles. Si vous voulez *illuminer* l'objet visuel, utilisez le NormalGenerator pour calculer les normales pour la géometrie [4].

[2] Alors que vous pouvez spécifier de polygones non-plans dans GeometryInfo, et un objet Triangulator pourra créer depuis celui-ci une surface, les formes non-planes ne spécifient pas de surface unique. En d'autre mots, si vous déterminez une forme non-plane, vous n'obtiendrez pas du Triangulator la surface désirée.

[3] Notez que la figure ne représente pas nécessairement la qualité de la triangulation produite par la classe Triangulator.

[4] Si vous n'êtes pas familier avec le terme d'*illumination* employé dans le contexte du graphisme informatique, jetez un coup d'œil au lexique et lisez la partie d'introduction du Chapitre 6.

Un programme d'exemple, `GeomInfoApp.java`, utilisant les classes GeometryInfo, Triangulator, Stripifier, et NormalGeneration pour créer une voiture, se trouve dans le répertoire `examples/easyContent`. La Figure 3-3 montre deux rendus produits par `GeomInfoApp.java`. Dans les deux rendus la ligne de contour bleue montre le profil déterminé dans l'objet GeometryInfo. Les triangles rouges (pleins et illuminés à gauche, sous la forme de lignes à droite) sont calculés par l'objet GeometryInfo avec les Triangulation, NormalGeneration, et Stripification réalisées de façon automatique.

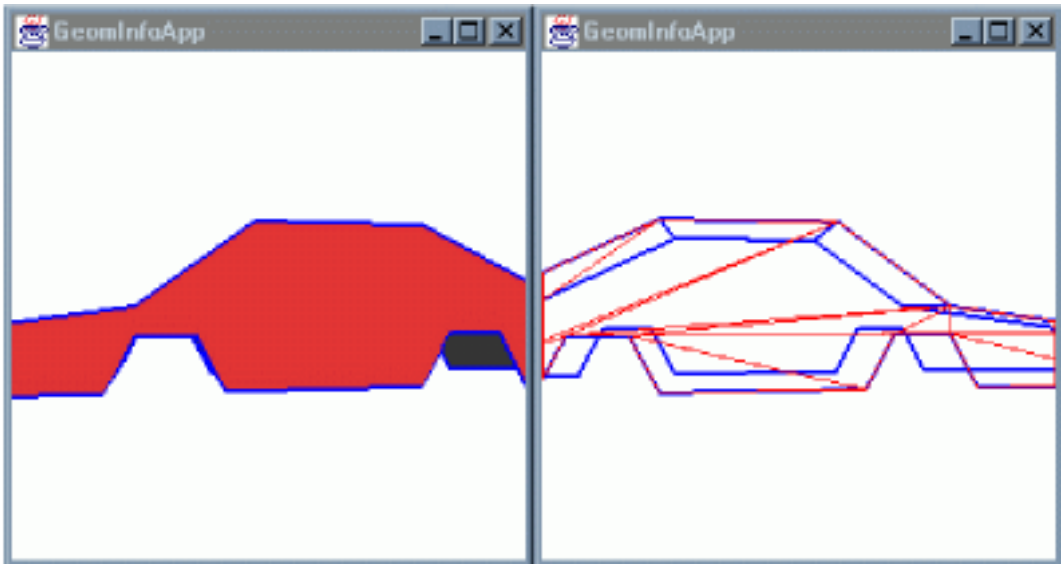



**Figure 3-3 Deux rendus de la voiture (présentés dans des directions opposées) crées en utilisant GeometryInfo.**

Un simple polygone plan, similaire à celui présenté dans la Figure 3-2, détermine le profil de la voiture (chaque coté) dans l'exemple GeomInfoApp. Des quadrilatères déterminent le capot, le toit, et les autres surfaces de la voiture.

## 3.3.1 Exemple de GeometryInfo simple

L'utilisation d'un objet GeometryInfo est aussi simple que celle des classes racines GeometryArray, voire plus. En créant un objet GeometryInfo, spécifiez simplement le type de géométrie dont vous aurez besoin. Les choix sont `POLYGON_ARRAY`, `QUAD_ARRAY`, `TRIANGLE_ARRAY`, `TRIANGLE_FAN_ARRAY`, et `TRIANGLE_STRIP_ARRAY`. Puis déterminez les coordonnées et les compteurs de bande pour la géométrie. Vous n'avez pas à appeler l'objet GeometryInfo qu'importe le nombre de coordonnées présentes dans la donnée; il sera calculée automatiquement.

Le Fragment de code 3-2 montre un exemple d'application du GeometryInfo. Les lignes de 1 à 3 du Fragment de code 3-2 montrent la création d'un objet GeometryInfo, et la spécification initiale de la géométrie.

Après avoir crée l'objet GeometryInfo, les autres classes pourront être employés. Si vous voulez utiliser le NormalGenerator, par exemple, créez en premier un objet NormalGenerator, puis passez lui l'objet GeometryInfo. Les lignes 8 à 9 du Fragment de code 3-2 s'occupent justement de cela.

```
1. GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON_ARRAY);
2. gi.setCoordinates(coordinateData);
3. gi.setStripCounts(stripCounts);
4.
5. Triangulator tr = new Triangulator();
6. tr.triangulate(gi);
7.
8. NormalGenerator ng = new NormalGenerator();
9. ng.generateNormals(gi);
10.
11. Stripifier st = new Stripifier();
12. st.stripify(gi);
13.
14. Shape3D part = new Shape3D();
15. part.setAppearance(appearance);
16. part.setGeometry(gi.getGeometryArray());
```

**Fragment de code 3-2 Utilisation des utilitaires GeometryInfo, Triangulator, NormalGenerator, et Stripifier.**

## 3.3.2 Classes pour GeometryInfo

Les classes de GeometryInfo et celles apparentées sont des éléments du package `com.sun.j3d.util.geometry` et sont des sous-classes d'Object. La Figure 3-4 montre la hiérarchie de ces classes.

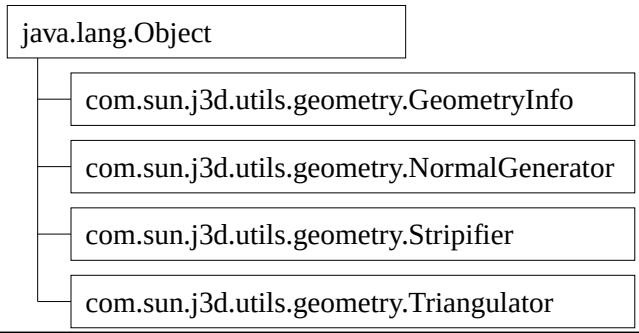



**Figure 3-4 Hiérarchie de classe de la classe utilitaire GeometryInfo, et des classes apparentées.**

La classe GeometryInfo possède un seul constructeur et c'est dans ce constructeur que vous spécifiez le type de géométrie devant être déterminée par la donnée de location [coordinate data]. Le Bloc de référence suivant en fournit les détails.

**Résumé des constructeurs de GeometryInfo**

Package: `com.sun.j3d.utils.geometry`

Étends : `java.lang.Object`

L'objet GeometryInfo est l'endroit ou vous posez votre géométrie si vous voulez utiliser les librairies d'utilitaires Java 3D. Une fois que vos données sont dans l'objet GeometryInfo, vous pouvez l'envoyer à n'importe lequel (ou lesquels) des nombreux utilitaires pour mener des opérations sur celui-ci, comme la génération de normales, ou la transformation en de long bandeaux pour un rendu plus efficace ("stripifying"ou «bandification»). La géométrie est chargée de la mêm manière que pour l'objet Java 3D GeometryArray, mais il y a moins d'options pour amener les données dans l'objet. GeometryInfo contient lui-même quelques utilitaires simples, comme le calcul des index pour des données non-indexées ("indexifying") et la mise à l'écart des données inutilisées dans vos informations de géométrie indexée ("compacting").

**`GeometryInfo(int primitive)`**
Construit un objet GeometryInfo, ou la primitive est une parmi :

`POLYGON_ARRAY`
Peut être un contour multiple, peut-être des polygone non-plans.

`QUAD_ARRAY`
Chaque jeu de quatre sommets forme un quadrilatère indépendant.

`TRIANGLE_ARRAY`
Chaque jeu de trois sommets forme un triangle indépendant

`TRIANGLE_FAN_ARRAY`
Le tableau stripCounts indique le nombre de sommets à utiliser pour chaque éventail.

`TRIANGLE_STRIP_ARRAY`
Le tableau stripCounts indique le nombre de sommet à employer pour chaque bandes de triangles.

La classe GeometryInfo possède de nombreuses méthodes. La majorité servent à fixer (ou obtenir) des données de coordonnées, de couleur, de normale, ou de coordonnée de texture. La plupart des applications n'utiliseront que certaines des méthodes disponibles. Pourtant, il est commode d'être capable de désigner la géométrie à n'importe quel niveau de détail et de laisser le reste ce calculer.

**Résumé des méthodes de GeometryInfo (liste partielle)**

**`void recomputeIndices()`**
Refaire les indexes pour garantir la connexion de l'information.

**`void reverse()`**
Renverse l'ordre de toutes les listes.

**`void setColorIndic es(int[] colorIndices)`**
Positionne le tableau d'indices dans le tableau Color.

**`void setColors(Color3f[] colors)`**
Établit le tableau des couleurs.

**`void setColors(Color4f[] colors)`**
Établit le tableau des couleurs. Il y a d'autres méthodes setColor.

**`void setContourCounts(int[] contourCounts)`**
Établit la lise des compteurs de contour.

**`void setCoordinateIndices(int[] coordinateIndices)`**
Positionne le tableau des indices dans le tableau Coordinate.

**`void setCoordinates(Point3f[] coordinates)`**
Établit le tableau des coordonnées.

**`void setCoordinates(Point3d[] coordinates)`**
Établit le tableau des coordonnées. Il y a d'autres méthodes setCoordinates.

**`void setNormalIndices(int[] normalIndices)`**
Positionne le tableau des indices dans le tableau Normal.

**`void setNormals(Vector3f[] normals)`**
Établit le tableau des normales.

**`void setNormals(float[] normals)`**
Établit le tableau des normales.

**`void setStripCounts(int[] stripCounts)`**
Établit le tableau du compteur de bande.

**`void setTextureCoordinateIndices(int[] texCoordIndices)`**
Positionne le tableau des indices dans le tableau TextureCoordinate.

**`void setTextureCoordinates(Point2f[] texCoords)`**
Établit le tableau TextureCoordinates. Il y a d'autres méthodes setTextureCoordinates.

Chacune des classes « d'aide » GeometryInfo est utilisée de façon similaire à la classe GeometryInfo. Les Blocs de référence suivants présentent les constructeurs et les méthodes pour Triangulator, Stripifier, et NormalGenerator, dans cet ordre. C'est l'ordre dans lequel ils seront employés pour un `POLYGON_ARRAY`.

L'utilitaire Triangulator est utilisé seulement avec une géométrie `POLYGON_ARRAY`. Les objets GeometryInfo avec les autres primitives géométriques utiliseront de manière adéquate seulement Stripifier et NormalGenerator

Le constructeur par défaut de la classe Triangulator crée simplement un objet Triangulation. Voir le Bloc de référence pour plus d'informations.

**Résumé du constructeur Triangulator**

Package : `com.sun.j3d.utils.geometry`

Étends: `java.lang.Object`

Triangulator est un utilitaire pour transformer arbitrairement les polygones en triangles afin qu'ils puissent être rendus par Java 3D. Les polygones peuvent être concaves, non-plans, et contenir des trous (voir GeometryInfo).

**`Triangulator()`**
Crée une nouvelle instance de Triangulator.

L'unique méthode de la classe Triangulator permet de trianguler un objet tableau de polygone GeometryInfo.

**Résumé de la méthode de Triangulator**

**`void triangulate(GeometryInfo gi)`**
Cette routine convertit l'objet GeometryInfo depuis une primitive de type `POLYGON_ARRAY` en une primitive de type `TRIANGLE_ARRAY` en utilisant les techniques de décomposition polygonales.

Le seul constructeur de la classe Stripifier crée un objet de bandification [5] [stripification].

**Résumé du constructeur de Stripifier**

Package: `com.sun.j3d.utils.geometry`

Étends: `java.lang.Object`

L'utilitaire Stripifier transformera la primitive de l'objet GeometryInfo en bandes de triangle. Les bandes sont fabriquées par l'analyse des triangles dans la donnée d'origine et les connecte entre eux.
Pour de meilleurs résultats NormalGeneration devra être exécuté sur l'objet GeometryInfo avant la bandification.

**`Stripifier()`**
Crée l'objet Stripifier.

L'unique méthode de la classe Stripifier est de bandifier la géométrie d'une classe GeometryInfo.

**Résumé de la méthode Stripifier**

**`void stripify(GeometryInfo gi)`**
Tourne la géométrie contenue dans l'objet GeometryInfo en un tableau de bandes de triangles [Triangle Strips].

La classe NormalGenerator possède deux constructeurs. Le premier construit un NormalGenerator une valeur par défaut pour le plissage de l'angle. Le second constructeurs autorise la détermination d'un angle de plis à la construction de l'objet NormalGenerator. Voir le Bloc de référence ci-après.

[5] Chaque paragraphe apporte un mot nouveau à la langue Anglaise (et aussi française).

**Résumé des constructeurs de NormalGenerator**

Package: `com.sun.j3d.utils.geometry`

Étends : `java.lang.Object`

L'utilitaire NormalGenerator effectuera le calcul et le placement des normales dans un objet GeometryInfo. Les normales calculées sont basées sur une estimation, analyse des informations des coordonnées indexés. Si vos données ne sont pas indexées, une liste d'index sera crée.

Si deux triangles (ou plus) dans la forme partagent le même index de cordonnées, alors le générateur de normales tentera de générer une normale pour le sommet, résultant en une surface paraissant « lissée ». Si deux coordonnées n'ont pas le même index alors elles auront deux normales séparées, même si elles ont la même position. Ce qui engendrera un « faux-pli » dans votre objet. Si vous soupçonnez que votre géométrie n'est pas correctement indexée, faites appel à `GeometryInfo.recomputeIndexes()`.

Bien sûr, de temps à autre un faux-pli apparaîtra sûr votre forme. Si les normales de deux triangles diffèrent par plus que le creaseAngle, alors le sommet obtiendra deux normales séparées. Ce qui est parfait pour le bord d'une table ou le coin d'un cube, par exemple.

**`NormalGenerator()`**
Construit un NormalGenerator avec un angle de faux-pli par défaut (0.76794 radians, ou 44°).

**`NormalGenerator(double radians)`**
Construit un NormalGenerator avec l'angle de faux-pli spécifié en radians.

Les méthodes pour la classe NormalGenerator en incluent une permettant de régler et d'acquérir l'angle de faux-pli, et de calculer les normales pour la géométrie d'un objet GeometryInfo. Voir le Bloc de référence résumant les constructeurs pour le NormalGenerator pour un débat sur l'angle de faux-pli.

**Résumé des méthodes de NormalGenerator**

**`void generateNormals(GeometryInfo geom)`**
Génère les normales pour l'objet GeometryInfo.

**`double getCreaseAngle()`**
Renvoie la valeur actuelle pour l'angle de faux-plis, en radians.

**`void setCreaseAngle(double radians)`**
Fixe l'angle de faux-plis en radians.

# 3.4 Text2D

Il y a deux manières d'ajouter du texte à une scène Java 3D. La première façon est d'utiliser la classe Text2D, la seconde la classe Text3D. Évidemment, la différence significative est que les objets Text2D sont en deux dimensions et le objets Text3D sont en trois dimensions. Un autre différence significative est le mode de création de ces objets.

Les objets Text2D sont des polygones rectangulaires avec le texte appliqué sous la forme d'une texture (le texturage est le sujet du Chapitre 7). Les objets Text3D sont des objets en 3D géométrique crées par une extrusion du texte. Voir la Partie 3.5 pour plus d'informations sur la classe Text3D et ses classes apparentées.

Comme une sous-classe de Shape3D, les instances de Text2D peuvent êtres des enfants d'objets groupe. Pour placer un objet Text2D dans une scène Java 3D, il suffit simplement de créer un objet Text2D et de l'ajouter au graphe scénique. La Figure 3-5 présente cette recette simple.

1. Créer d'un objet Text2D.
2. L'ajouter au graphe scénique.

**Figure 3-5 La recette de Text2D.**

Les objets Text2D sont réalisé par l'emploi d'un polygone et d'une texture. Le polygone est transparent ainsi seule la texture est visible. La texture est déterminée par le texte d'une chaîne de caractères dans le style de police nommé avec les paramètres spécifiés de caractères de police [6].

Les styles de polices disponibles sur votre système peut varier. Généralement, les styles Courier, Helvetica, and TimesRoman sont disponibles, entre autres. N'importe quelle police de caractère disponible dans l'AWT est à votre disposition pour une application en Text2D (et Text3D). L'usage de l'objet Text2D est assez directe comme il est démontré dans la partie suivante.

## 3.4.1 Exemple simple de Text2D

Le Fragment de code 3-3 montre un exemple d'ajout d'un objet Text2D dans une scène. L'objet Text2D est crée de la ligne 21 à 23. Dans ce constructeur sont déterminés, le texte de la chaîne de caractère [string], la couleur, le style de police, la taille et le type de caractère. L'objet Text2D est ajouté au graphe scénique à la ligne 24. Notez la déclaration d'importation de Font (ligne 5) utilisée pour les constantes de style de police de caractère.

```
1. import java.applet.Applet;
2. import java.awt.BorderLayout;
3. import java.awt.Frame;
4. import java.awt.event.*;
5. import java.awt.Font;
6. import com.sun.j3d.utils.applet.MainFrame;
7. import com.sun.j3d.utils.geometry.Text2D;
8. import com.sun.j3d.utils.universe.*;
9. import javax.media.j3d.*;
10. import javax.vecmath.*;
11.
12. // Text2DApp rend un simple objet Text2D.
13.
14. public class Text2DApp extends Applet {
15.
16.    public BranchGroup createSceneGraph() {
17.         // Crée la racine de la branche du graphe.
18.         BranchGroup objRoot = new BranchGroup();
19.
20.         // Crée un nœud de terminaison Text2D, l'ajoute au graphe scénique.
21.         Text2D text2D = new Text2D("2D text is a textured polygon",
22.                                    new Color3f(0.9f, 1.0f, 1.0f),
23.                                    "Helvetica", 18, Font.ITALIC));
24.         objRoot.addChild(text2D);
```

**Fragment de code 3-3 Création d'un objet Text2D (extrait de Text2DApp.java)**

[6] Une police de caractère est un style de police déterminé par une taille avec un jeu d'attributs de style de caractères. Se référer au Lexique pour les définitions de police de caractère et style de police.

`Text2DApp.java` est programme complet incluant le Fragment de code ci-dessus. Dans cet exemple, l'objet Text2D effectue une rotation autour de l'origine de la scène. À l'exécution de l'application vous pouvez voir, par défaut, que le polygone texturé est invisible lorsqu'il est visualisé par l'arrière.


**Figure 3-6 Image de Text2DApp.java.**

Quelques attributs d'un objet Text2D peuvent être changés par une modification de l'ensemble d'apparence référencé et /ou du composant de nœud Geometry. Le Fragment de code 3-4 expose le code pour réaliser text2d, l'objet Text2D crée dans le Fragment de code 3-3, à deux cotés grâce à la modification de son ensemble d'apparence.

```
25.     Appearance textAppear = text2d.getAppearance();
26.
27.     // Les quatre lignes de code suivantes rendent l'objet Text2D  double-face.
28.     PolygonAttributes polyAttrib = new PolygonAttributes();
29.     polyAttrib.setCullFace(PolygonAttributes.CULL_NONE);
30.     polyAttrib.setBackFaceNormalFlip(true);
31.     textAppear.setPolygonAttributes(polyAttrib);
```

**Fragment de code 3-4 Fabrication d'un objet Text2D à deux-cotés.**

La texture crée par un objet Text2D peut aussi bien être appliquée aux autres objets visuels. Étant donné que l'application de textures sur des objets visuels est le sujet du Chapitre 7, les détails d'application sont laissée de coté jusqu'à celui-ci.

## 3.4.2 Les classes utilisés dans la création d'objets Text2D

La seule classe nécessaire est la classe Text2D. Comme vous pouvez le voir à la Figure 3-7, Text2d est une classe utilitaire qui étends Shape3D.

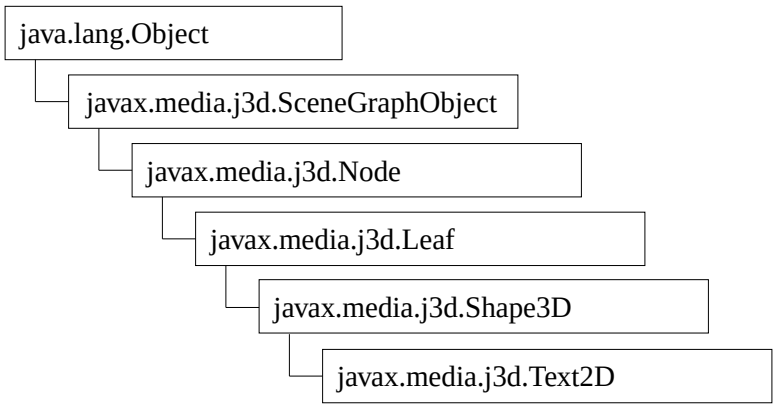



**Figure 3-7 Hiérarchie de la classe Text2D.**

Dans le constructeur de Text2D sont spécifiés : la chaîne de caractères, le style de police, la taille des caractères, et le style des caractères.

**Résumé du constructeur de Text2D**

Package : `com.sun.j3d.utils.geometry`

Cette classe crée un rectangle texturé-mappé qui affiche la chaîne de caractères envoyés par l'utilisateur, avec les paramètres d'apparence donnés également par l'utilisateur. La taille du rectangle (et sa texture mappée) est déterminée par les paramètres de la police de caractère envoyés au constructeur. L'objet Shape3D résultant est un rectangle transparent (à l'excéption du texte) placé à (0,0,0) et s'étallant vers le haut sur l'axe positif des y et latéralement sur l'axe positif des x.

**`Text2D(java.lang.String text, Color3f color, java.lang.String fontName, int fontSize, int fontStyle)`**
Constructeur.

Avec le constructeur de Text2D, on trouve une seule méthode. Cette méthode détermine un facteur de mise à l'échelle pour créer des objets Text2D plus grand ou plus petit que la taille de police spécifiée. Cette méthode n'est pas utile dans la version 1.1x de l'API, étant donné quelle n'est utilisé qu'une fois le texte spécifié. Dans la version 1.2 une méthode setText()sera introduite rendant setRectangleScaleFactor() utile.

**Résumé de la méthode de Text2D**

**`void setRectangleScaleFactor(float newScaleFactor)`**
Établit le facteur de mise à l'echelle utilisé dans la conversion 3D des valeurs largeur/hauteur en largeur/hauteur.

# 3.5 Text3D

Une autre voie pour ajouter du texte dans un monde virtuel Java 3D est de créer un objet Text3D pour le texte. Où Text2D créait un texte avec une texture, le Text3D crée du texte utilisant la géométrie. La géométrie tirée du texte d'un objet Text3D est un extrusion de la police de caractère.

La création d'un objet Text3D est un peu plus compliqué que la création d'un objet Text2D. La première étape étant la création d'un objet Font3D du style de police, de taille et de style de caractère désirés. Ensuite un objet Text3D est fabriqué pour une chaîne de caractère précise en utilisant l'objet Font3D. Étant donné que la classe Text3D est une sous-classe de Geometry, l'objet Text3D est un composant de nœud qui est référencé par un ou plusieurs objet(s) Shape3D. La Figure 3-8 résume le processus d'addition d'objets Text3D à un graphe scénique.

1. Créer un objet Font3D depuis une police de caractère d'AWT.
2. Créer un Text3D pour une chaîne de caractères utilisant l'objet Font3D, avec en option la spécification du point de référence.
3. Référence l'objet depuis un objet Shape3D ajouté au graphe scénique.

**Figure 3-8 Recette pour la création d'un objet Text3D.**

## 3.5.1 Exemple de simple Text3D

Le Fragment de code 3-5 montre les bases de la construction d'un objet Text3D. L'objet Text3D est crée de la ligne 19 à 20. Le style de police utilisé ici est une « Helvetica ». Exactement comme le Text2D, n'importe quel style de police disponible dans AWT peut être utilisé pour les objets Font3D mais également pour les Text3D. Ce constructeur Font3D (lignes 19 à 20 du Fragment de code 3-5) fixe également la taille des caractères à 10 points et utilise l'extrusion par défaut.

Les déclarations des lignes 21 à 22 créent un objet Text3D utilisant l'objet Font3D précédemment crée pour la chaîne de caractère « 3DText » en spécifiant un point de référence pour l'objet. Les deux dernières déclarations créent un objet Shape3D pour l'objet Text3D et l'ajoute au graphe scénique. Notez que la déclaration d'importation pour les Font (ligne 5) est nécessaire depuis qu'un objet Font est employé dans la création du Font3D.

```
1. import java.applet.Applet;
2. import java.awt.BorderLayout;
3. import java.awt.Frame;
4. import java.awt.event.*;
5. import java.awt.Font;
6. import com.sun.j3d.utils.applet.MainFrame;
7. import com.sun.j3d.utils.universe.*;
8. import javax.media.j3d.*;
9. import javax.vecmath.*;
10.
11. // Text3DApp rend un objet Text3D unique.
12.
13. public class Text3DApp extends Applet {
14.
15.    public BranchGroup createSceneGraph() {
16.         // Crée la racine de la branche du graphe.
17.         BranchGroup objRoot = new BranchGroup();
18.
19.         Font3D font3d = new Font3D(new Font("Helvetica", Font.PLAIN, 10),
20.         new FontExtrusion());
21.         Text3D textGeom = new Text3D(font3d, new String("3DText"),
22.         new Point3f(-2.0f, 0.0f, 0.0f));
23.         Shape3D textShape = new Shape3D(textGeom);
24.         objRoot.addChild(textShape);
```

**Fragment de code 3-5 Création d'un objet visuel Text3D.**

La Figure 3-9 montre un objet Text3D illuminé afin d'illustrer l'extrusion de la police. Dans la figure, l'extrusion est affichée en gris tandis que la police est affichée en noir. Pour recréer cette figure en Java 3D, un objet Material et une DirectionalLight sont nécessaires. Comme le Chapitre 6 couvre ce sujet, il n'est pas abordé ici. Vous pouvez déterminer la couleur des sommets individuellement dans l'objet Text3D puisque vous n'avez pas accès à la géométrie de l'objet Text3D.


**Figure 3-9 Le points de référence et l'extrusion par défaut pour un objet 3DText.**

Le texte d'un objet Text3D peut être orienté dans diverses directions. L'orientation est déterminée par un chemin directionnel. Les choix sont right, left, up, et down. Voir le Tableau 3-2 et le Bloc de référence de Text3D (commençant à la page 3-20) pour plus d'informations.

Chaque objet Text3D prend un point de référence. Le point de référence pour un objet Text3D est l'origine de l'objet. Le point de référence de chaque objet est défini par la combinaison du chemin et de l'alignement du texte. Le Tableau 3-2 montre les effets de la spécification du chemin sur l'orientation du texte et l'emplacement du point de référence.

L'emplacement du point de référence peut être défini explicitement en imposant un positionnement du chemin et de l'alignement. Voir les Blocs de références (commençant à la page suivante) pour plus d'information.

| | ALIGN_FIRST (default) | **ALIGN_FIRST** | **ALIGN_FIRST** |
|---|---|---|---|
| **PATH_RIGHT (default)** | Text3D | Text3D | Text3D |
| **PATH_LEFT** | D3txeT | D3txeT | D3txeT |
| **PATH_DOWN** | T<br>e<br>x<br>t | T<br>e<br>x<br>t | T<br>e<br>x<br>t |
| **PATH_UP** | t<br>x<br>e<br>T | t<br>x<br>e<br>T | t<br>x<br>e<br>T |

**Tableau 3-2 L'orientation du texte et le positionnement du point de référence par les combinaisons de chemins et d'alignement de Text3D.**

Les objets Text3D ont des normales. L'ajout d'un ensemble d'apparence incluant un objet Material à un objet Shape3D référençant une geometrie Text3D autorisera l'éclairage de l'objet Shape3D.

## 3.5.2 Les classes utilisées dans la création d'objets Text3D

Cette partie présente une référence importante pour les trois classes utilisées dans la création d'objets Text3D : Text3D, Font3D, et FontExtrusion, dans cet ordre. La Figure 3-10 montre la hiérarchie des classes pour la classe Text3D.

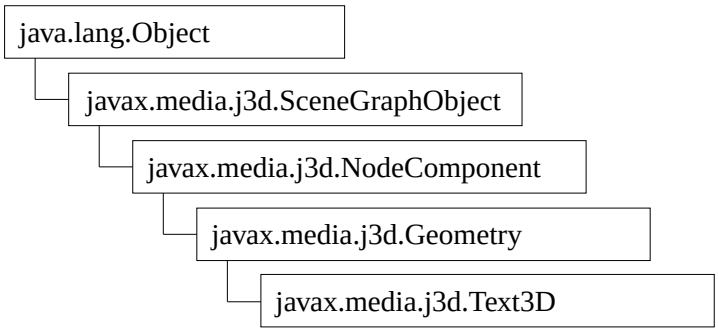



**Figure 3-10 Hiérarchie des classes pour Text3D.**

La classe Text3D définit de un certain nombre de constructeurs. Chaque constructeur vous permet de spécifier aucun, quelques-uns, ou tous les attributs d'un objet Text3D. Le Bloc de référence suivant liste les constructeurs, avec leurs valeurs par défaut, pour Text3D.

**Résumé des constructeurs de Text3D**

Un objet Text3D est le texte d'une chaîne de caractères qui a été converti en une géométrie 3D. L'objet Font3D détermine l'apparence de l'objet composant de nœud de l'objet Text3D. Chaque objet Text3D possède un emplacement — un point de référence passant l'objet Text3D. Le texte 3D peut-être placé autour de cet emplacement en utilisant différents alignements et chemins.

**`Text3D()`**
Crée un objet Text3D vide. Les valeurs par défaut utilisés par celui-ci, et les autres constructeurs appropriés, sont :

```
font 3D              null
string               null
position             (0,0,0)
alignment            ALIGN_FIRST
path                 PATH_RIGHT
character spacing    0.0
```

**`Text3D(Font3D font3D)`**
Crée un objet Text3D avec l'objet Font3D donné.

**`Text3D(Font3D font3D, String string)`**
Crée un objet Text3D avec l'objet Font3D donné et une chaîne de caractères.

**`Text3D(Font3D font3D, String string, Point3f position)`**
Crée un objet Text3D avec l'objet Font3D donné et une chaîne de caractères. Le point de positionnement détermine un point de référence pour l'objet Text3D. Sa position est définie relativement au coin de face en bas à gauche de la géométrie.

**`Text3D(Font3D font3D, String string, Point3f position,int alignment, int path)`**
Crée un objet Text3D avec l'objet Font3D donné et une chaîne de caractères.

**`ALIGN_CENTER`**
alignement : le centre de la chaîne est placé sur le point de positionnement.

**`ALIGN_FIRST`**
alignement : le premier caractère de la chaîne est placé sur le point de positionnement.

**`ALIGN_LAST`**
alignement : le dernier caractère de la chaîne est placé sur le point de positionnement

**`PATH_DOWN`**
chemin : les glyphes successifs sont placés en-dessous du glyphe précédent.

**`PATH_LEFT`**
chemin : les glyphes successifs sont placés à la gauche du glyphe précédent.

**`PATH_RIGHT`**
chemin : les glyphes successifs sont placés à la droite du glyphe précédent.

**`PATH_UP`**
chemin : les glyphes successifs sont placés au-dessus du glyphe précédent.

Voir les exemples au Tableau 3-2.

La classe Text3D défini aussi un nombre de méthodes. Chacune vous permet de modifier (set) les attributs de l'objet Text3D. Cette classe détermine aussi les méthodes `get*` correspondantes. Le Bloc de référence suivant liste les méthodes `set*` pour la classe Text3D.

**Résumé des méthodes de Text3D**

**`void setAlignment(int alignment)`**
Établit la méthode d'alignement du texte pour cet objet Text3D composant de nœud.

**`void setCharacterSpacing(float characterSpacing)`**
Fixe l'espacement des caractères utilisé pour la construction de la chaîne de caractère Text3D.

**`void setFont3D(Font3D font3d)`**
Détermine l'objet Font3D utilisé par cet objet Text3D composant de nœud.

**`void setPath(int path)`**
Détermine la direction du chemin du nœud.

**`void setPosition(Point3f position)`**
Détermine le point de référence du nœud pour le paramètre donné.

**`void setString(java.lang.String string)`**
Copie la chaîne de caractères depuis le paramètre donné dans le nœud Text3D.

Le Bloc de référence suivant liste les aptitudes de la classe Text3D.

**Résume des aptitudes de Text3D**

**`ALLOW_ALIGNMENT_READ | WRITE`**
autorise la lecture (écriture) de l'alignement du texte.

**`ALLOW_BOUNDING_BOX_READ`**
autorise la lecture de la valeur de la boite de limitation [bounding box] pour la chaîne de caractères.

**`ALLOW_CHARACTER_SPACING_READ | WRITE`**
autorise la lecture (écriture) de la valeur d'espacement des caractères du texte.

**`ALLOW_FONT3D_READ | WRITE`**
autorise la lecture (écriture) des informations du composant Font3D.

**`ALLOW_PATH_READ | WRITE`**
autorise la lecture (écriture) de la valeur du chemin du texte.

**`ALLOW_POSITION_READ | WRITE`**
autorise la lecture (écriture) la valeur de positionnement du texte.

**`ALLOW_STRING_READ | WRITE`**
autorise la lecture (écriture) de l'objet String.

Chaque objet Text3D est crée depuis un objet Font3D. Un simple objet Font3D peut être utilisé pour créer un nombre illimité d'objets Text3D [7]. Un objet Font3D détient la géométrie d'extrusion pour chaque glyphe dans le style de police. Un objet Text3D copie la géométrie pour produire la chaîne de caractères spécifiée. Les objets Font3D peuvent être collectés par le ramasse ordures sans affecter les objets Text3D crées d'après eux.

Le Bloc de référence suivant montre le constructeur pour la classe Font3D.

**Résumé des constructeurs de Font3D**

Étend: `java.lang.Object`

Une police de caractère en 3D se compose d'une police de caractère 2D et d'un chemin d'extrusion. Le chemin d'extrusion décrit comment le bord d'un glyphe varie sur l'angle des Z. L'objet Font3D est utilisé pour stocker les glyphes 2D extrudés. Ces glyphes 3D peuvent ensuite être utilisés pour construire des objets composant de nœud Text3D. Les polices de caractères 3D sur mesure en plus des méthodes pour les stocker sur le disque seront adressées dans une prochaine version.
Voir aussi : java.awt.Font, FontExtrusion, Text3D.

**`Font3D(java.awt.Font font, FontExtrusion extrudePath)`**
Crée un objet Font 3D depuis l'objet Font spécifié.

Le Bloc de référence suivant liste les méthodes de la classe Font3D. Normalement, les méthodes `get*` ne sont pas listées dans les Blocs de référence de ce tutorial. Cependant, comme la classe Font3D n'a pas de méthode `set*`, les méthodes `get*` de Font3D sont listées ici. Le résultat d'une méthode `set*` sera essentiellement le même que l'invoquation d'un constructeur, alors pour garder la classe petite, aucune méthodes `set*` n'est définie.

[7] Bien sûr, les contraintes de mémoire pourrons limiter le nombre réel d'objets Text3D à un nombre évidemment plus petit que l'infini.

**Résumé des méthodes de Font3D**

**`void getBoundingBox(int glyphCode, BoundingBox bounds)`**
Renvoie la boite de limitation 3D du code du glyphe spécifié.

**`java.awt.Font getFont()`**
Renvoie la police de caractère (Font) Java 2D utilisée pour créer cet objet Font3D.

**`void getFontExtrusion(FontExtrusion extrudePath)`**
Copie l'objet FontExtrusion utilisé pour crée cet objet Font3D dans le paramètre spécifié.

La classe Font est employée pour la création d'un objet Font3D. Le Bloc de référence suivant liste un constructeur pour Font. Les autres constructeurs et les autres méthodes ne sont pas listées dans ce tutorial. Voir les Spécifications de l'API Java 3D pour les détails.

**Résume du constructeur de Font (liste partielle)**

Package: `java.awt`

Une classe AWT qui crée une représentation interne des polices de caractères. Font étend java.lang.Object.

**`public Font(String name, int style, int size)`**
Crée une nouvelle Font depuis le nom, le style et la taille en point spécifiés.
Paramètres :
**`name -`** le nom du style de la police. Cela peut être un nom logique ou le nom d'un style de police. Un nom logique doit être un parmi : Dialog, DialogInput, Monospaced, Serif, SansSerif, et Symbol.
**`style -`** la constante de style pour Font. L'argument de style est un nombre entier [integer] avec un bit-masqué qui peut être `PLAIN`, ou avec bit-comparateur union de `BOLD` et/ou `ITALIC` (par exemple, `Font.ITALIC` ou `Font.BOLD|Font.ITALIC`. Tous les autres bits spécifiés dans les paramètres du style sont ignorés. Si les arguments du style ne sont pas conforme à un des nombres entiers bit-masqué alors le style est fixé à `PLAIN`.
**`size -`** la taille en point de Font.

Le Bloc de référence suivant liste les constructeurs pour la classe FontExtrusion.

**Résumé des constructeurs de FontExtrusion**

Étend : `java.lang.Object`

L'objet FontExtrusion est employé pour définir le chemin d'extrusion d'un objet Font3D. Le chemin d'extrusion est utilisé en combinaison avec un objet Font2D. Le chemin d'extrusion défini le bord du contour du texte 3D. Ce contour est perpendiculaire à la surface du texte. L'extrusion prend sa racine au bord du glyphe avec 1.0 étant la hauteur du plus petit glyphe. Le contour doit être monotonique en X. L'utilisateur est responsable de l'integrité des données et doit être sur que l'extrusionShape n'intersecte pas avec les glyphes adjacents ou avec un autre glyphe solitaire.
Le rendu est indéterminé pour les extrusions causant des intersections.

**`FontExtrusion()`**
Construit un objet FontExtrusion avec les paramètres par défaut.

**`FontExtrusion(java.awt.Shape extrusionShape)`**
Construit un objet FontExtrusion de la forme désignée.

Le Bloc de référence suivant liste les méthodes pour la classe FontExtrusion.

**Résume des Méthodes de FontExtrusion**

**`java.awt.Shape getExtrusionShape()`**
Obtient les paramètres de la forme du FontExtrusion.

**`void setExtrusionShape(java.awt.Shape extrusionShape)`**
Fixe le paramètre de la forme de FontExtrusion.

# 3.6 Background

Par défaut, l'arrière-plan d'un univers virtuel Java 3D est solide et noir. Cependant, si vous pouvez spécifier d'autres arrières-plans pour vos mondes virtuels. L'API Java 3D permet de manière simple de spécifier une couleur solide, un image, une géométrie, ou une combinaison de ceux-ci, comme arrière-plan.

Quand vous désignez une image pour l'arrière-plan, elle annule la couleur d'arrière-plan déterminée, si il y en a une. Quand une géometrie est spécifie, elle est dessinée par dessus la couleur d'arrière-plan ou image.

La seule partie de ruse se situe dans la détermination d'un arrière-plan géométrique. Tous les arrières-plans géométriques sont spécifiés comme des points sur un élément sphérique. Si votre géometrie est un PointArray, lesquels peuvent représenter des étoiles à des années lumière, ou un TriangleArray qui pourra représenter des montagnes dans le lointain, toutes les coordonnées sont spécifiées à une distance d'une unité. La géometrie d'arrière-plan est projeté à l'infini au rendu.

Les objets Background ont des limites d'Application lesquelles permettent de spécifier différents arrière-plans pour différente régions du monde virtuel. Un nœud Background est actif quand sa région d'application intersecte le volume d'activation de la ViewPlatform.

Si plusieurs nœuds d'arrière-plans sont actifs, le nœud de Background qui est le plus « proche » de l'œil sera utilisé. Si il n'y a aucun nœud de Background actif, alors la fenêtre est remplie en noir. Toutefois, la notion de plus « proche » n'est pas spécifié. Pour proche, est choisi l'arrière-plan avec la limite d'application la plus profonde englobant la ViewPlatform.

Il est peu probable que vos applications nécessitent un arrière-plan géométrique éclairé — dans la réalité le système visuel humain ne peux pas percevoir les détails visuels à grande distance. Pourtant, une géométrie d'arrière-plan peut être illuminé. Le sous-graphe de la géométrie d'arrière-plan ne peut pas contenir de Lights, mais les Lights définies dans le graphe scénique peuvent influencer la géometrie d'arrière-plan.

Pour créer un arrière-plan, suivez la recette simple donnée à la Figure 3-11. Les exemples d'arrière-plans sont présentés dans la partie suivante.

1. Créer un objet Background spécifiant une couleur ou une image.
2. Ajouter une géométrie (option).
3. Fournir une limite d'Application ou un BoundingLeaf.
4. Ajouter l'objet Background au graphe scénique.

**Figure 3-11 Recette pour les Backgrounds.**

## 3.6.1 Exemples de Background

Comme il est expliqué dans la partie précédente, un arrière-plan peut être soit une couleur ou une image. Une géometrie peut apparaître en arrière-plan avec soit une couleur ou une image. Cette partie fournit un exemple d'arrière-plan blanc solide. Un second exemple montre l'addition d'une géometrie à un arrière-plan.

### Exemple d'arrière-plan coloré

Comme dans la Figure 3-11, la recette pour créer un arrière-plan coloré et solide est assez directe. Les lignes de code dans le Fragment de code 3-6 correspondent aux étapes de la recette. En plus de la personnalisation de la couleur, le seul autre ajustement possible à ce code serait la définition d'une limite d'application plus appropriée pour l'arrière-plan (ou l'emploi d'un BoundingLeaf).

```
1.      Background backg = new Background(1.0f, 1.0f, 1.0f);
2.      //
3.      backg.setApplicationBounds(BoundingSphere());
4.      contentRoot.addChild(backg);
```

**Fragment de code 3-6 Ajouter un arrière-plan coloré.**

### Exemple d'arrière-plan géométrique

Une fois encore, les lignes du Fragment de code 3-7 correspondent aux étapes de le recette pour l'arrière-plan de la Figure 3-11. Dans ce Fragment de code, la méthode `createBackGraph()` est appelée pour la création de la géométrie d'arrière-plan. Cette méthode renvoie un objet BranchGroup. Pour un exemple plus complet, voir la `BackgroundApp.java` dans le répertoire `examples/easyContent`.

```
1.   Background backg = new Background(); // arrière-plan de couleur noire.
2.   backg.setGeometry(createBackGraph());// ajoute le BranchGroup d'arrière-plan.
3.   backg.setApplicationBounds(new BoundingSphere(new Point3d(), 100.0));
5.   objRoot.addChild(backg);
```

**Fragment de code 3-7 Ajout d'un arrière-plan géométrique.**

### BackgroundApp.java

Pour apprécier les arrières-plan, vous devrez les experimenter. `BackgroundApp.java`, un programme inclus dans le répertoire `examples/easyContent`, est une application fonctionnant entièrement avec un arrière-plan géométrique. Cette application permet le déplacement dans le monde virtuel Java 3D. Pendant le déplacement, vous pouvez voir le mouvement relatif entre la géometrie de la scène et la géometrie de l'arrière-plan.

BackgroundApp utilise la classe KeyNavigatorBehavior fournie par la librairie d'utilitaires pour le mouvement de visualisation. L'Interaction (qui est appliquée au moyens de comportements) est le sujet du Chapitre 4, donc les détails sur cette programmation sont différée jusqu'à celui-ci.

KeyNavigatorBehavior répond aux touches fléchés, au touches PgUp, et PgDn pour le mouvement. La touche Alt joue également un rôle (plus de détails au Chapitre 4). Quand vous exécuterez `BackgroundApp`, effectuez une rotation pour trouver la « constellation », qui en plus de se déplacer dans le lointain gardera ces distances.

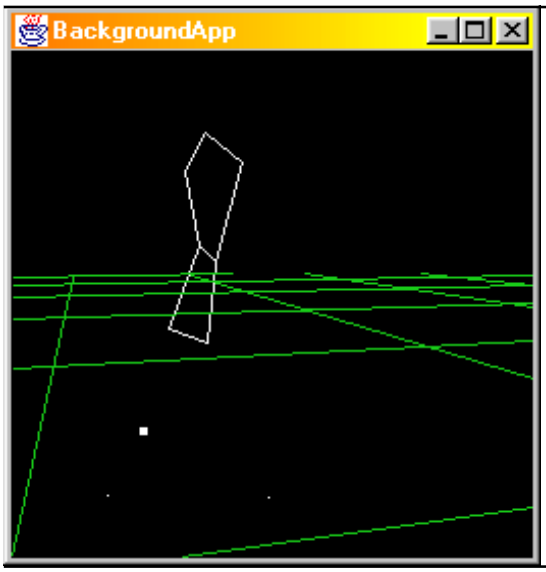



**Figure 3-12 Vue de la « Constellation » dans l'arrière-plan de BackgroundApp.java.**

## 3.6.2 La classe Background

La Figure 3-13 montre la hiérarchie de la classe Background. Comme une extension de la classe Leaf, une instance de classe Background peut être un enfant d'un objet Group.

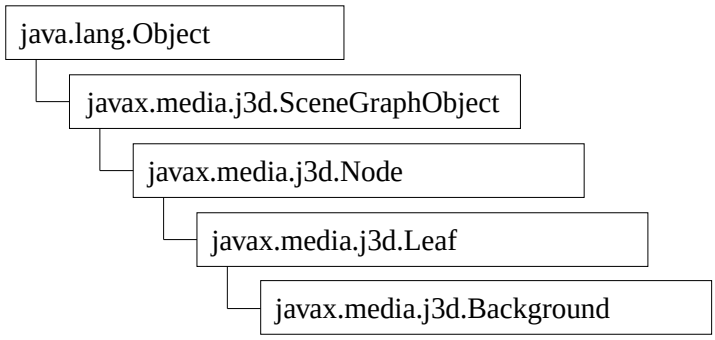



**Figure 3-13 Hiérarchie de classe pour Background.**

Background possède une variété de constructeurs avec des paramètres autorisant la détermination d'une couleur ou d'une image pour un arrière-plan. Le Bloc de référence suivant donne plus de détails. La géométrie d'arrière-plan peut seulement être spécifiée par la méthode appropriée.

**Résume des constructeurs de Background**

Le nœud de terminaison Background détermine soit une couleur solide d'arrière-plan ou une image d'arrière-plan utilisée pour remplir la fenêtre au début de chaque nouvelle frame. Il permet en option de référencer une géometrie d'arrière-plan. La géometrie d'arrière-plan doit être pré-quadrillée sur un élément sphérique et dessinée à l'infini. Il détermine aussi une région d'application dans laquelle l'arrière-plan est actif.

**`Background()`**
Construit un nœud Background avec la couleur par défaut (noir).

**`Background(Color3f color)`**
Construit un nœud Background avec la couleur donnée.

**`Background(float r, float g, float b)`**
Construit un nœud Background avec la couleur donnée.

**`Background(ImageComponent2D image)`**
Construit un nœud Background avec limage donnée.

N'importe quel attribut d'un Background peut être réglé par une méthode. Le Bloc de référence suivant liste les méthodes de la classe Background.

**Résume des Méthodes de Background**

**`void setApplicationBoundingLeaf(BoundingLeaf region)`**
Établit la région d'application du Background pour la terminaison de limitation spécifiée.

**`void setApplicationBounds(Bounds region)`**
Établit la région d'application du Background pour la limitation spécifiée.

**`void setColor(Color3f color)`**
Établit la couleur du Background à la couleur spécifiée.

**`void setColor(float r, float g, float b)`**
Établit la couleur du Background à la couleur spécifiée.

**`void setGeometry(BranchGroup branch)`**
Établit la géométrie d'arrière-plan pour le nœud de BranchGroup spécifiée.

**`void setImage(ImageComponent2D image)`**
Établit comme image d'arrière-plan l'image spécifiée.

Le Bloc de référence suivant liste les bits d'apptitudes pour la classe Background.

**Résume des aptitudes de Background**

**`ALLOW_APPLICATION_BOUNDS_READ | WRITE`**
autorise un accès d'écriture (lecture) à sa limite d'application.

**`ALLOW_COLOR_READ | WRITE`**
autorise un accès d'écriture (lecture) à sa couleur.

**`ALLOW_GEOMETRY_READ | WRITE`**
autorise un accès d'écriture (lecture) à sa géométrie d'arrière-plan.

**`ALLOW_IMAGE_READ | WRITE`**
autorise un accès d'écriture (lecture) à son image.

# 3.7 BoundingLeaf (terminaison de limitation)

Les limites sont utilisées avec des lumières, des arrière-plans, et une variété d'autres applications en Java 3D. Les limites permettent au programmeur de varier les actions, les apparences, et/ou les sons sur le paysage virtuel. La détermination de limites permet aussi au système de rendu Java 3D d'appliquer une élimination des exécutions inutiles, et de ce fait améliorer les performances du rendu [8].

La spécification de limites typique utilise un objet Bounds pour la spécification de la région de limitation. Son application dans un graphe scénique donne un objet Bounds se déplaçant avec l'objet auquel il fait référence. Ce qui est formidable pour de nombreuses applications ; pourtant, il y aura des situations dans lesquelles il sera préférable d'avoir une région de limitation qui se déplacera indépendamment de l'objet utilisant cette limite. Par exemple, dans un univers incluant une source de lumière statique qui illumine un objet se déplaçant, la limite pour la lumière devra inclure l'objet se déplaçant. Ce qui n'est pas la meilleure réponse dans la majorité des cas. Une meilleure solution est l'utilisation d'un BoundingLeaf. Placé dans le graphe scénique avec l'objet visuel, l'objet BoundingLeaf se déplace avec l'objet visuel et indépendamment de la source de lumière. La Figure 3-14 montre un graphe scénique dans lequel une lumière utilise un nœud BoundingLeaf.

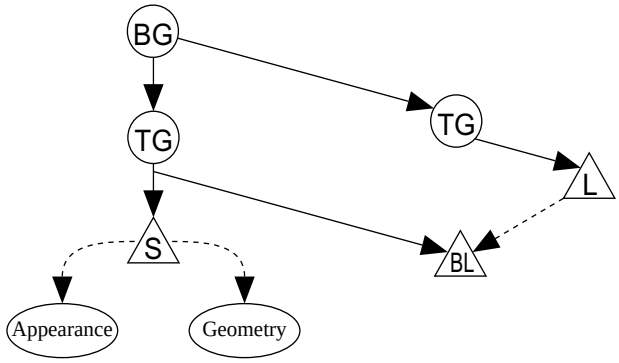



**Figure 3-14 Un BoundingLeaf de déplaçant avec un objet visuel et indépendamment d'une source de lumière.**

Comme les applications pour les BoundingLeaf incluent les limites d'application [ApplicationBounds] des objets Background (Partie 3.5), les SchedulingBounds des Behaviors (Partie 4.2) et les InfluencingBounds des Lights (Partie 6.6), il ne serait pas logique de répéter cette information dans ces trois endroits [9].

[8] Les Bounds devront être choisis aussi petit que possible, jusqu'à l'exécution de l'effet désiré, afin de réduire le calcul requis pour le rendu Java 3D de la scène.

[9] Les nœuds BoundingLeaf sont utiles pour les objets brouillard, surfaces de sélection [clip], sons et les fonds sonores, ceux-ci n'étant pas abordés dans ce tutorial.

Une application intéressante d'un objet BoundingLeaf met en place un BoundingLeaf dans la viewPlatform. Ce BoundingLeaf peut être utilisé pour un un réglage de la limitation sur « toujours en fonction » pour un Behavior, ou « toujours en application » la limite d'application pour un Background ou un Fog. Le Fragment de code 3-8 présente un exemple de mise en œuvre de « toujours en application » pour un BoundingLeaf en utilisation avec un Background. Un autre exemple d'application d'un BoundingLeaf est présenté dans la Partie 6.6.

Le Fragment de code 3-8 présente un exemple d'addition d'un BoundingLeaf comme enfant de PlatformGeometry pour produire une limite « toujours en application » pour un Background [10]. Dans ce code, la méthode standard `createSceneGraph()` est modifiée pour ne prendre qu'un seul paramètre, l'objet SimpleUniverse [11]. Ceci est nécessaire pour créer un objet PlatformGeometry.

Les lignes 2, 3, et 4, créant l'objet BoundingLeaf, créant l'objet PlatformGeometry, et rendant l'objet BoundingLeaf enfant de PlatformGeometry, dans cet ordre. Si il se trouve qu'il y a d'autres BoundingLeaf pour la PlatformGeometry, ils seront ajoutés à cet endroit. L'objet PlatformGeometry est ensuite ajouté à la branche visuelle du graphe à la ligne 6.

L'objet BoundingLeaf est est établit comme limite d'application pour l'objet d'arrière-plan à la ligne 11. Ce même BoundingLeaf pourra être utilisé pour d'autres choses dans ce programme. Par exemple, il pourra être aussi employé par les comportements. Notez que l'utilisation dans ce programme du BoundingLeaf comme InfluencingBoundingLeaf d'une lumière ne rend pas la lumière influente sur tous les objets de l'univers virtuel.

```
1. void createSceneGraph (SimpleUniverse su) {
2.      BoundingLeaf boundingLeaf = new BoundingLeaf();
3.      PlatformGeometry platformGeom = new PlatformGeometry();
4.      platformGeom.addChild(boundingLeaf);
5.      platformGeom.compile();
6.      simpleUniv.getViewingPlatform().setPlatformGeometry(platformGeom);
7.
8.      BranchGroup contentRoot = new BranchGroup();
9.
10.         Background backg = new Background(1.0f, 1.0f, 1.0f);
11.         backg.setApplicationBoundingLeaf(boundingLeaf);
12.         contentRoot.addChild(backg);
```

**Fragment de code 3-8 Ajout d'un BoundingLeaf à la plate-forme de visualisation pour une limite « toujours en application ».**

[10] PlatformGeometry se déplace avec le spectateur, c'est alors le parent approprié pour la géométrie associée à une visualisation. Par exemple, si le spectateur se trouve à conduire un véhicule, il sera approprié de faire de la géometrie représentant l'instrumentation du véhicule un enfant de PlatformGeometry.

[11] La méthode createSceneGraph() est seulement un standard par le fait qu'elle garde sa forme dans les exemples de ce tutorial.

### 3.7.1 Classe BoundingLeaf

La classe BoundingLeaf étend la classe Leaf. La Figure 3-15 présente la hiérarchie complète de la classe BoundingLeaf.

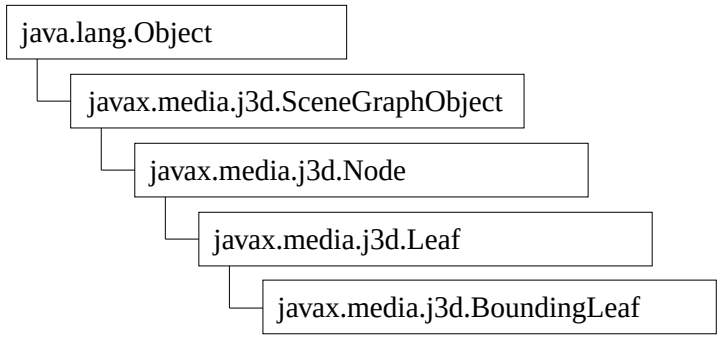



**Figure 3-15 Hiérarchie de la classe BoundingLeaf de l'API Java 3D.**

Le constructeur de base pour un BoundingLeaf crée une limitation d'unité sphérique. Les autres constructeurs autorisent la détermination de la limitation pour l'objet BoundingLeaf. Le Bloc de référence suivant liste les deux constructeurs de BoundingLeaf.

**Résume des constructeurs de BoundingLeaf**

Le nœud BoundingLeaf définit un objet region de limitation qui peut être référencé par les autres nœuds pour définir une région d'influence, un région d'activation, ou une région planifiée.

**`BoundingLeaf()`**
Construit un nœud BoundingLeaf avec un objet d'unité sphérique.

**`BoundingLeaf(Bounds region)`**
Construit un nœud BoundingLeaf avec la region de limitation spécifiée.

Le Bloc de référence suivant liste les deux méthodes de BoundingLeaf. La méthode `get*` est listée si ces paramètres sont différents de sa méthode `set*` correspondante.

**Résume des Méthodes de BoundingLeaf**

**`Bounds getRegion()`**
Renvoie la region de limitation de ce BoundingLeaf.

**`void setRegion(Bounds region)`**
Établit la region de limitation de ce nœud BoundingLeaf.

# 3.8 User Data

N'importe quel SceneGraphObject peut faire référence à n'importe lequel objet UserData [12]. Premièrement, vous devez réaliser que presque toutes les classes racines de l'API Java 3D sont des descendant du SceneGraphObject. La liste des descendants du SceneGraphObject incluant Appearance, Background, Behavior, BranchGroup, Geometry, Lights, Shape3D, et TransformGroup.

[12] Ce n'est pas une limitation des classes de l'API Java 3D, mais de toutes les classes dérivés de java.lang.Object.

Les applications des champs UserData ne sont limitées que par votre imagination. Par exemple, une application pourra avoir un nombre d'objets selectionnables. Chacun de ces objets pouvant avoir certains textes informatifs stockés dans l'objet User Data. Quand un utilisateur sélectionnera un objet, les données d'utilisateur [User Data] pourront être affichés.

Une autre application pourra stocker certaines valeurs calculées pour un objet du graphe scénique par exemple sa position dans le monde virtuel. Ou bien une application pourra stocker certaines informations spécifiques de comportement qui pourront contrôler un comportement appliqué à une variété d'objets.

**Méthodes de SceneGraphObject (Liste partielle - Méthodes d'User Data)**

Un SceneGraphObject est une super-classe commune à tous les objets composant le graphe scénique. Ceci comprenant Node, Geometry, Appearance, etc.

**`java.lang.Object getUserData()`**
Renvoie le champ userData depuis cet objet graphe scénique .

**`void setUserData(java.lang.Object userData)`**
Détermine le champ userData associé à cet objet graphe scénique.

# 3.9 Résumé du chapitre.

Ce chapitre présente les caractéristiques de Java 3D permettant une création facile de volumes. Les utilitaires Loader et les classes GeometryInfo étant les techniques primaires de création de volume. Ces thèmes sont abordés dans les Parties 3.2 et 3.3, respectivement. Le texte est ajouté au monde Java 3D en utilisant les classes Text2D et Text3D dans les Parties 3.4 et 3.5. L'arrière-plan est couvert en détail à la Partie 3.6 et le BoundingLeaf est abordé dans la Partie 3.7. La Partie 3.8 présentant le champ UserData de la classe SceneGraphObject. Vous lisez actuellement la Partie 3.9.

# 3.10 Tests personnel

1. En utilisant la vue en fil de fer du programme GeomInfoApp.java, vous pouvez voir les effets de la triangulation. Pour augmenter votre expérience, changez l'ordre des polygones pour utiliser trois polygones (un pour chaque côté, un pour le toit, un pour le capot et les autres surfaces). Comment le Triangulator se comporte-t-il avec cette surface ?
2. Le code pour rendre l'objet Text2D visible des deux cotés est inclus dans Text2DApp.java. Vous pouvez enlever les caractères // de commentaire de ce code, puis le compiler et l'exécuter. D'autres expérimentations se trouvent dans ce programme, comme l'application de la texture de l'objet Text2D sur d'autres objets visuels. Par exemple, essayez d'ajouter une géométrie primitive et appliquez la texture sur celui-ci. Bien sur, vous devrez attendre la lecture du Chapitre 7 pour cet exercice.
3. En prenant Text3DApp.java comme point de départ, expérimentez les différents alignements et placements du point de référence. D'autres expérimentations pouvant être le changement d'apparence de l'objet Text3D.
4. Lancez le programme d'exemple BackgroundApp.java, si vous vous déplacez trop loin de l'origine du monde virtuel, l'arrière-plan disparaît. Pourquoi cela se produit-il ? Si vous ajoutez un autre Background à la BackgroundApp, quelle conséquence cela aurait-il ?