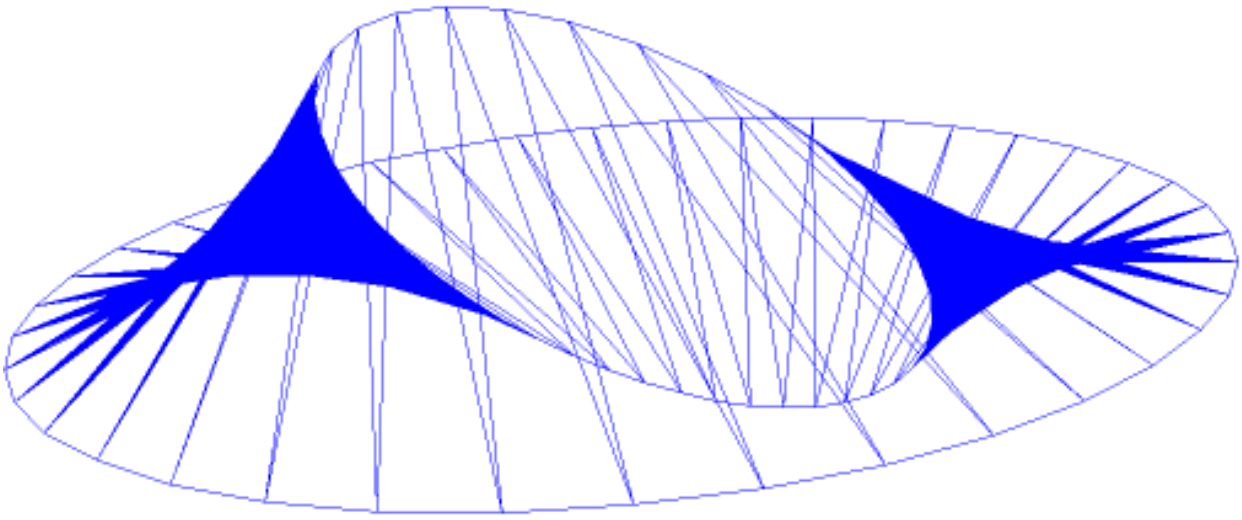
# Initiation à l'API Java 3D™

## Un tutorial pour les débutants

## Chapitre 1
## Prendre un bon départ



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 <

2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A



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

# Fragments de code



**Préface au chapitre 1**

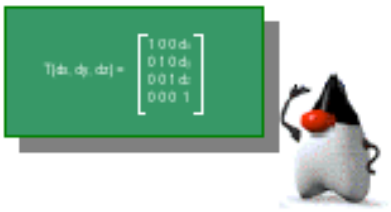
Ce document est la première partie d'un tutorial sur l'utilisation de l'API Java 3D.

Les chapitres supplémentaires, la préface, les annexes et le lexique 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 1
# Prendre un bon départ



**Objectifs de ce chapitre**

Après la lecture de ce chapitre vous serez :

• Capable d'expliquer en quoi consiste l'API Java3D.

• Capable de décrire la structure de base des programmes en Java3D.

• Capable de reconnaître plusieurs classes de l'API Java3D.

• Capable d'écrire un programme simple en Java 3D.

L'API Java 3D est une interface pour écrire des programmes afin d'afficher et de réagir à un environnement graphique en trois dimensions. Java 3D est une extension standard du kit de développement Java2 JDK. Cette API fournit une collection de constructeurs de haut-niveau pour créer, manipuler et présenter un rendu graphique d'une géométrie en 3D. Java 3D produit les fonctions nécessaires à des programmes sous la forme d'applications, produisant des images, des visualisations, des animations, et des interaction avec des graphismes 3D.

## 1.1 Qu'est ce que l'API Java 3D ?

L'API Java 3D est un classement de structures de données (classes) Java qui sert d'interface à la production d'images précises en trois dimensions, et de sons. Le programmeur travaille avec les constructeurs lourds pour créer et manipuler des objets 3D géométriques. Ces géométries existent dans un univers virtuel, qui est ensuite rendu. L'API permet de créer des univers précis d'une large variété de dimensions, de l'astronomique au sub-atomique.

Malgré ces multiples fonctionnalités, l'API est néanmoins facile d'utilisation. Le processus de rendu est manipulé automatiquement. Tirant l'avantage de la faculté des threads Java, qui permettent à plusieurs processus de se dérouler en même temps, Java 3D est capable de rendre en parallèle. Il peut également optimiser le rendu de façon automatique afin d'en améliorer les performances.

Un programme Java3D crée des instances d'objets Java 3D et les place dans un graphe scénique [scene graph en anglais]. Le graphe scénique est un agencement d'objets 3D, sous la forme d'une arborescence en arbre inversé, spécifiant le contenu de l'univers virtuel et comment il doit être rendu.

Les programmes en Java 3D peuvent être écrit pour être exécutés soit sous forme d'application, d'applet dans un navigateur configuré pour Java 3D ou bien sous les deux formes [1].

# 1.2 L'API Java 3D

Chaque programme Java 3D est, du moins partiellement, composé d'objets de la hiérarchie des classes de l'API Java3D. Cette collection d'objets décrit un *univers virtuel* [Virtual Universe], lequel est ensuite visualisé. L'API définit plus de 100 classes présentées dans le package `javax.media.j3d.` Ces classes sont généralement définies comme les *classes racines* [core class]de Java3D.

Il y a des centaines de champs [fields] et de méthodes dans les classes de l'API Java 3D. Pourtant, un univers simple incluant de l'animation peut être construit avec un petit nombre de classes. Ce chapitre en présente un ensemble minimal.

Ce chapitre inclus le développement d'un programme simple, mais complet, nommé HelloJava3D qui affiche un cube en rotation. Le programme d'exemple est présenté par paliers, sous différentes versions, afin de présenter chaque partie du processus de la programmation en Java 3D. Tous les programmes utilisés dans ce tutorial sont disponibles électroniquement. Voir la section « Où trouver ce tutorial » dans la préface du Chapitre 0 pour plus d'informations.

En plus du package racine de Java 3D, d'autres packages sont utilisés dans l'écriture de programmes Java 3D. L'un d'entre eux, le package `com.sun.j3d.utils` qui est défini comme le package des *classes utilitaires*. Le package des classes racines comprend seulement les classes de base nécessaires à la programmation Java3D. Les classes utilitaires sont des alliées commodes et puissantes en addition aux classes racines.

Les classes utilitaires se scindent en quatre catégories importantes : chargeurs de volume (content loaders), assistants à la construction du graphe scénique, classes de géométrie, et classes d'utilitaires supplémentaires. Des utilitaires de supplément, comme les nurbs, seront certainement ajoutés dans le futur, comme des classes utilitaires supplémentaires et non pas dans le package des classes racines. Plusieurs des classes utilitaires pourront être déplacés vers le package des classes racine dans les futures versions de l'API Java3D.

L'usage des classes utilitaires réduit significativement le nombre de lignes de code dans les programmes Java 3D. À l'emploi des packages racines et utilitaires, tous les programmes Java 3D ajoutent les packages `java.awt` et `javax.vecmath`. Le package `java.awt` définit une routine d'affichage de fenêtre : Abstract Windowing Toolkit (AWT ou Boite à Outils sommaire pour le Fenêtrage). Les classes AWT produisent à l'écran une fenêtre pour afficher le rendu. Le package `javax.vecmath` définit les classes de type mathématique pour les points, vecteurs, matrices, et tous les autres objets mathématiques.

Dans le restant du texte *objet visuel* sera utilisé pour faire référence à un « objet déposé dans le graphe scénique » (par exemple un cube ou une sphère). Le terme d'*objet* étant lui utilisé seulement pour l'instance d'une classe. Le terme de volume sera utilisé pour les objets visuels du graphe scénique de manière générale.

[1] Les navigateurs peuvent exécuter du Java3D grâce au Plug-in Java, celui-ci peut être téléchargé sur java.sun.com. Tous les exemples de ce tutorial sont écrits pour s'exécuter sous forme d'applications.

# 1.3 Construire un graphe scénique

Un univers virtuel Java 3D est crée depuis le *graphe scénique* travaillant avec des instances de classes Java 3D. Le graphe scénique rassemble des objets pour définir des géométrie, des sons, des lumières. Mais aussi l'emplacement, le sens d'orientation, et l'apparence de ces objets visuels et sonores.

Le graphe scénique est un ensemble de données structurelles composées de nœuds et de liens. Un nœud est une donnée, un lien permet la relation entre les données. Les nœuds du graphe scénique sont des instances des classes Java 3D. Les liens représentent deux sortes de relations d'héritage entre les instances de classes Java 3D.

La relation principale est une *relation parent-enfant*. Un groupe de nœuds peut avoir n'importe quel nombre d'enfants mais un seul parent. Le dernier nœud d'une arborescence peut avoir un seul parent mais aucun enfant. L'autre rapport d'héritage est la *référence*. La référence relie un objet *NodeComponent* (composant de nœud) à un nœud du graphe scénique. Un objet NodeComponent détermine les attributs de géométrie et d'apparence utilisée pour produire les objets visuels.

Un graphe scénique Java 3D contient des objets nœuds relies par un héritage parent-enfant formant ainsi une hiérarchie en arbre inversée. Le point de départ de cette arborescence en arbre inversée est un nœud formant la racine. Les autres nœuds sont liés aux suivants par des liens depuis la racine. Les liens ne peuvent pas fermer cette arborescence en retournant à leur racine. Le graphe scénique est crée par les objets formant les branches de l'arbre inversé prenant comme racine un objet nommé Locale (scène). Les NodeComponent (composant de nœud) et les liens de références ne font pas partie de l'arborescence du graphe scénique.

Un seul lien relie entre eux chaque composant de l'arborescence depuis la racine. Donc, il n'y a qu'un seul chemin depuis la racine jusqu'au dernier nœud de l'arborescence. Le chemin depuis la racine du graphe scénique jusqu'à un nœud de terminaison précis est *le chemin de graphe scénique* de ce nœud de terminaison. Comme le chemin du graphe scénique ne mène qu'a une seule extrémité, il n'y a qu'un seul chemin de graphe scénique pour chaque extrémité du graphe scénique.

Chaque chemin du graphe scénique dans un graphe scénique Java 3D détermine entièrement le statut de l'extrémité lui appartenant. Ce statut contient l'emplacement, l'orientation, et la taille de l'objet visuel. Par conséquent, les attributs visuels de chaque objet visuel dépendent entièrement de leur chemin de graphe scénique. Le rendu Java 3D tire l'avantage de cette particularité et rend les objets présents dans l'ordre qu'il choisit comme le plus efficace. Le programmeur en Java 3D n'a pas, normalement, le contrôle sur l'ordre de rendu des objets [2].

La représentation sous forme de schéma d'un graphe scénique peut servir d'outil et/ou de documentation pour les programmes Java 3D. Le graphe scénique est représentés par des symboles, décrits dans le schéma 1-1. Les programmes Java 3D peuvent contenir beaucoup plus d'objets que cet exemple de graphe scénique.

Pour concevoir un univers virtuel en Java3D, un graphe scénique est dessinée en utilisant cette palette de symboles standards. Une fois le schéma réalisé, celui-ci représente les instructions du programme. Puis lorsque le programme est réalisé, ce même schéma devient une représentation sommaire du programme (supposant qu'il complète une description détaillée). Un schéma du graphe scénique d'un programme existant donne des informations sur ce que le graphe scénique produit.

---

[2] Le seul contrôle que peut avoir le programmeur en Java 3D sur l'ordre de rendu est sur la classe de nœud OrderedGroup. Cette classe n'est pas étudiée dans ce tutorial. Voir « Les Spécifications de l'API Java 3D » pour plus d'information.

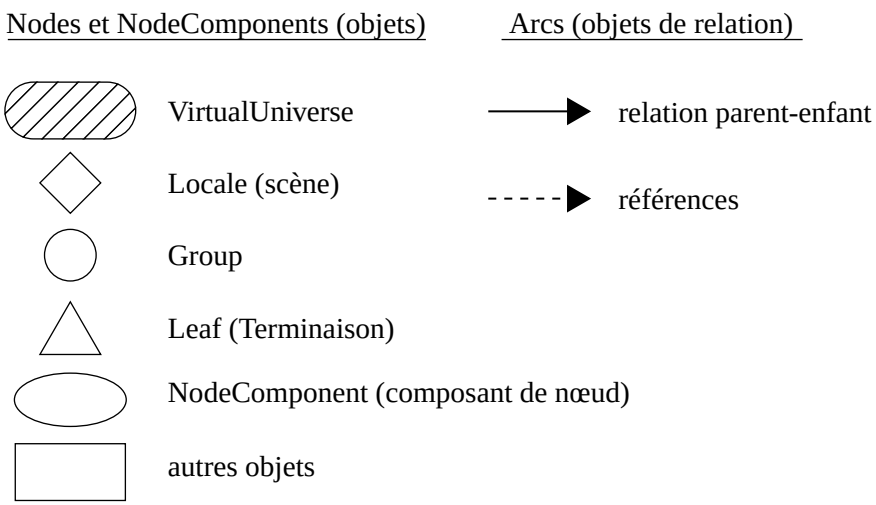



**Figure 1-1 Représentation symbolique des objets d'un graphe scénique.**

Chaque symbole du coté gauche du schéma 1-1 représente un seul objet utilisé dans le graphe scénique. Les deux premiers symboles représentent des objets de classes spécifiques : VirtualUniverse (univers virtuel) et Locale (scène). Les trois symboles suivants à gauche représentent les objets des classes Group (groupe), Leaf (terminaison), NodeComponent (composant du nœud). Le dernier symbole est utilisé pour représenter toutes les autres classes d'objet.

La flèche pleine représente une relation d'héritage parent-enfant. La flèche pointillée est une référence à tout autre objet. Les objets référencés peuvent être distribués entre différentes branches du graphe scénique. Un exemple d'un graphe scénique simple est décrit dans le schéma 1-2.

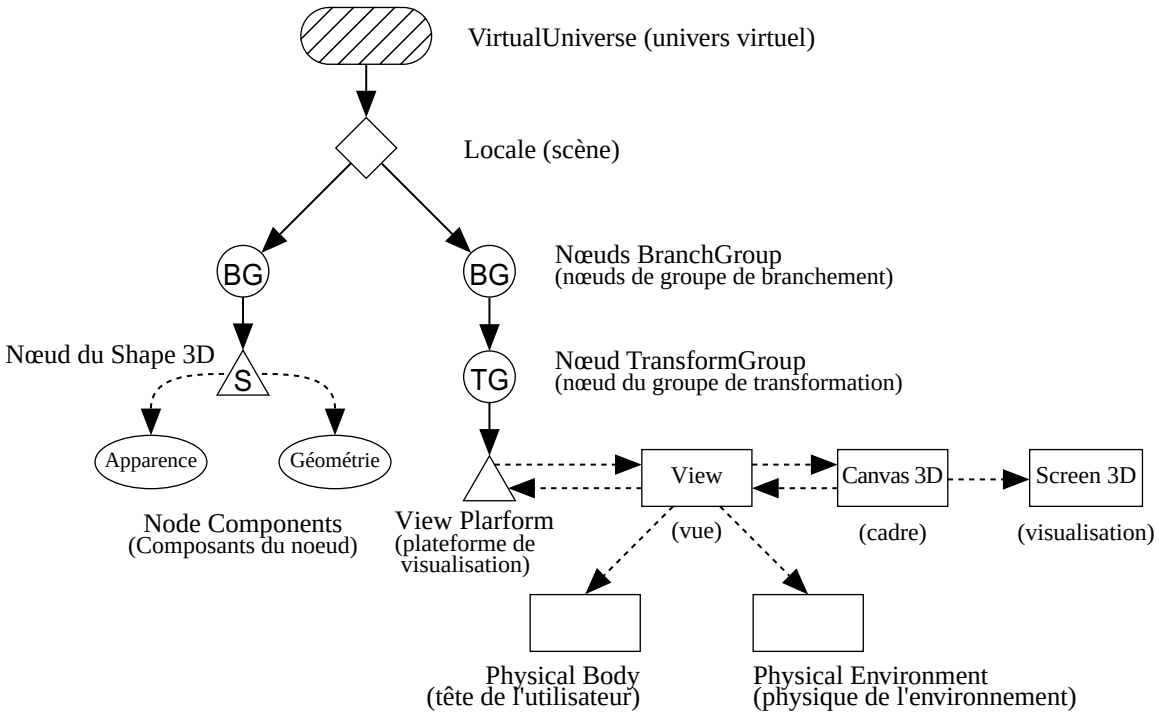



**Figure 1-2 Premier exemple de graphe scénique.**

Il est possible de créer un graphe scénique non conforme. Un exemple de graphe scénique illégal est présenté dans le schéma 1-3. Le graphe scénique décrit dans le schéma 1-3 est non conforme parce qu'il viole les propriétés d'un DAG. Le problème se produit car les deux objets TransformGroup (groupe de transformation) ont le même objet Shape3D (forme 3D) comme enfant. Rappelez-vous qu'un objet de terminaison ne peut avoir qu'un seul parent. En d'autres termes, il ne peut y avoir qu'un seul chemin depuis l'objet Locale vers un objet Leaf (ou un seul chemin depuis le Leaf vers le Locale).

Vous devez comprendre que le schéma 1-3 définit trois objets visuels dans un univers virtuel. Il apparaît que le graphe scénique définit deux objets visuels par le double usage de l'objet visuel (Shape3D) dans le coté droit du schéma. De manière conceptuelle, chaque objet du TransformGroup est parent de l'instance partagée du Shape3D pouvant placer une image de l'objet visuel à différents endroits. C'est donc une scène illégale tant que les liens parent-enfant ne forment pas une arborescence en arbre inversé. Dans cet exemple, le résultat est que le Shape3D possède plus d'un seul parent.

Cette remarque sur l'organisation de l'arborescence et du DAG (Direct Acyclic Graph) est importante. Pourtant, le système d'exécution Java 3D rapporte l'erreur en terme d'héritage parent-enfant. Le résultat de la limitation de l'arborescence montre que l'objet Shape3D est réduit à un seul lien parent-enfant. Prenons l'exemple du graphe du schéma 1-3, une exception de « multiple parents » est reporté à l'exécution. Le schéma 1-4, où l'on trouve un seul parent pour chaque objet Shape3D, présentant ainsi une réparation possible pour ce graphe scénique.

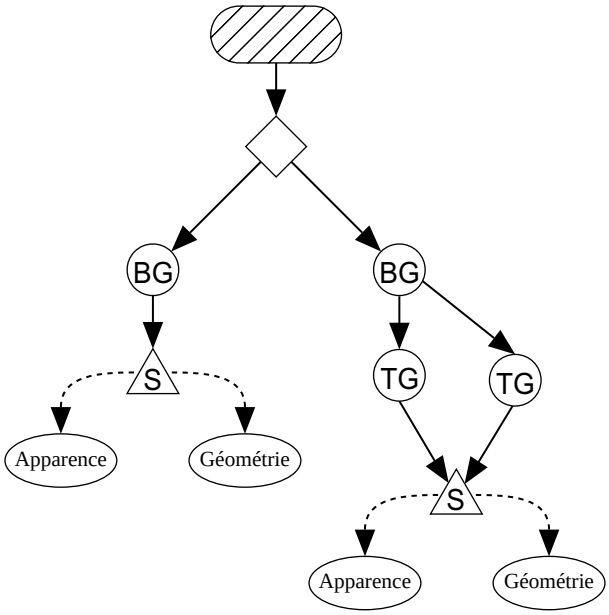



**Figure 1-3 Exemple de graphe scénique illégal.**

Un programme Java 3D définit par un graphe scénique de forme illégale peut être compilé, mais pas rendu. Quand un programme Java 3D définit par un graphe scénique de forme illégale est lancé, le système Java 3D détecte le problème. Une fois le problème détecté le système Java 3D rapporte une exception. Mais le programme peut continuer de fonctionner, et doit être stoppé. Par conséquent, aucune image ne pourra être rendue.

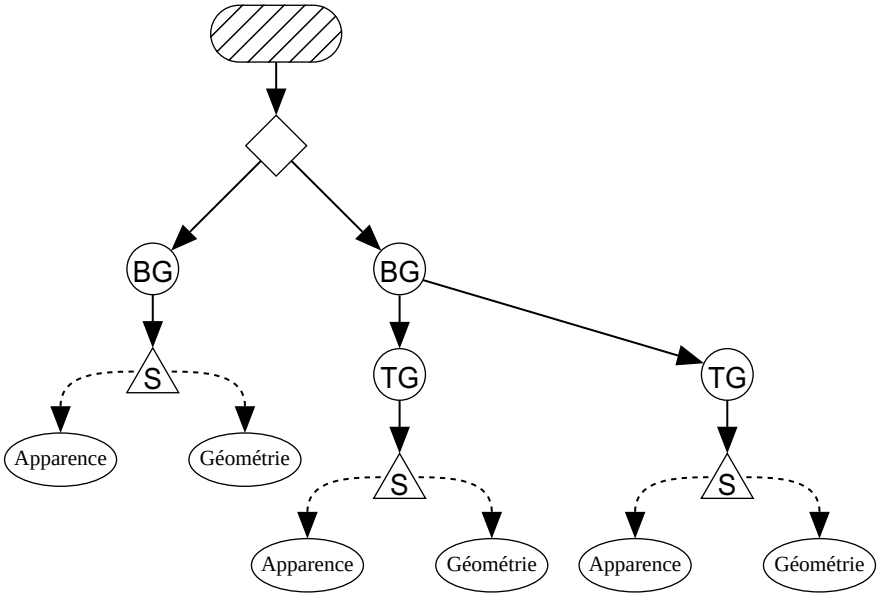



**Figure 1-4 Une résolution possible à l'exemple de graphe scénique illégal de la Figure 1-3.**

Chaque graphe scénique possède un seul VirtualUniverse (univers virtuel). L'objet VirtualUniverse possède une certaine quantité d'objets Locale (scène). Un objet Locale est le point de départ de l'univers virtuel. Pensez que l'objet Locale est un point de repère pour déterminer la position des objets visuels dans l'univers virtuel.

Il est techniquement possible à un programme Java 3D d'avoir plus d'un objet VirtualUniverse, définissant ainsi plus d'un univers virtuel. Toutefois, il n'y a pas de voie naturelle pour communiquer entre les univers virtuels. D'ailleurs un objet graphe scénique ne peut pas exister simultanément dans plusieurs univers virtuels. Il est hautement recommandé d'utiliser une seule et unique instance du VirtualUniverse dans chaque programme Java3D.

Alors que l'objet VirtualUniverse peut faire référence à plusieurs objets Locale, la plupart des programmes Java 3D possèdent un seul objet Locale. Chaque objet Locale peut servir de racine à de multiples sous-arborescences du graphe. Voir la Figure 1-2 comme exemple de graphe scénique et prendre note des deux branches d'arborescence depuis l'objet Locale.

L'objet BranchGroup est la racine d'un point d'articulation ou point de branchement. Il y a deux catégories différentes de point de branchement : le point de branchement de la *visualisation* et le point de branchement du *volume*. Le point de branchement de volume définit le *contenu* de l'univers virtuel - géométrie, apparence, comportement, emplacement, sons et lumières. Le point de branchement de la visualisation définit les paramètres visuels comme la position et la direction de la vue. Ces deux points de branchement forment la plus grande partie de ce que doit rendre le programme.

## 1.3.1 Hiérarchie des classes de haut niveau de l'API Java 3D

Un survol des trois premiers niveaux d'arborescence de l'API Java 3D sont présentés dans le schéma 1-5. Les classes VirtualUniverse, Locale, Group et Leaf (univers virtuel, scène, groupe, et terminaison) sont présentent dans cette partie de la hiérarchie. En plus des objets VirtualUniverse et Locale, le reste du graphe scénique est composé d'objets sceneGraphObject (objet du graphe scénique). sceneGraphObject est une super classe de presque toutes les classes de javax.media.j3d et com.sun.j3d.utils (classes racines et utilitaires).

Le sceneGraphObject possède deux sous-classes : Node (nœud) et NodeComponent (composant du nœud). La majeure partie des objets qui composent un graphe scénique sont les sous-classes Node (nœud). Un objet Node est soit un objet Group (groupe de nœuds) ou un objet Leaf (nœud de terminaison). Group et Leaf sont deux super classes d'un grand nombre de sous-classes. La classe Node et ses deux sous-classes sont détaillées ci-après. Cet base acquise, la construction des programmes Java 3D sera expliquée.

**La classe Node (nœud)**

La classe nœud est une super-classe regroupant les classes Group et Leaf. La classe Node définit des méthodes importantes pour ses sous-classes. Des explications plus détaillés sur ces méthodes seront fournies plus loin dans le tutorial. Les sous-classes de Node sont les constituantes du graphe scénique.

**La classe Group (groupe)**

La classe Group est une super-classe utilisée pour déterminer la localisation et l'orientation des objets visuels dans l'univers virtuel. Les classes BranchGroup (groupe de branchement) et TransformGroup (groupe de transformation) sont deux sous-classes. Dans la représentation schématique du graphe scénique, les symboles de Group (cercles) contiennent souvent une annotation BG pour BranchGroup, TG pour TransformGroup, etc. La Figure 1-2 en est un exemple.

**La classe Leaf (terminaison)**

La classe Leaf est une super classe utilisée pour déterminer les formes, les sons et les comportements des objets visuels dans l'univers virtuel. Shape3D, Light, Behavior et Sound sont des sous-classes de la classe Leaf. Ces objets ne peuvent pas avoir d'enfant mais doivent se référer à un NodeComponent.

**La classe NodeComponent (composant de nœud)**

La classe NodeComponent est une super-classe utilisée pour déterminer la géométrie, l'apparence, la texture, et les attributs de matiére d'un nœud Shape3D (terminaison). Les NodeComponent ne font pas partie du graphe scénique, mais lui sont référencés. Un NodeComponent peut être référencé par plusieurs Shape3D.

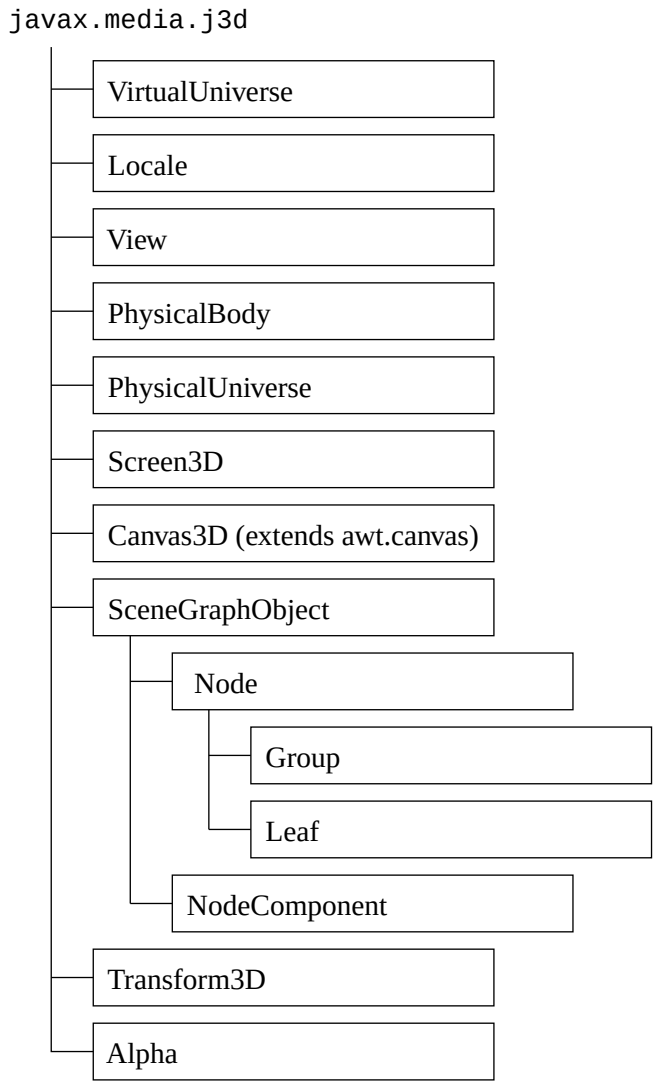



**Figure 1-5 Un survol de la hiérarchie des classes de l'API Java 3D.**

# 1.4 Une recette pour écrire des programmes Java 3D

Les sous-classes de sceneGraphObject (objet du graphe scénique) sont les blocs de construction qui sont assemblés dans le graphe scénique. Le plan schématique fondamental dans le développement d'un programme Java3D se divise en sept étapes (communément appelée recette [recipe] dans ce tutorial et dans la Documentation de l'API Java 3D) présentées dans le schéma 1-6. Cette recette peut être utilisée au montage de la plupart des programmes Java 3D.

1. Création d'un objet Canvas3D.
2. Création d'un objet VirtualUniverse.
3. Création d'un objet Locale, et l'attacher à l'objet VirtualUniverse.
4. Construction d'un point de branchement de visualisation.
    a. Création d'un objet View.
    b. Création d'un objet ViewPlatform.
    c. Création d'un objet PhysicalBody.
    d. Création d'un objet PhysicalEnvironment.
    e. Attacher les objets ViewPlatform, PhysicalBody, PhysicalEnvironment, et Canvas3D à l'objet View.
5. Construction du(des) point de branchement de volume (formes, lumières et sons).
6. Compilation de l'arborescence de la branche(s) volume.
7. Insertion des points de branchement à l'objet Locale.

**Figure 1-6 Recette pour écrire des programmes Java 3D.**

Cette recette ignore quelques détails mais illustre bien les concepts fondamentaux de la programmation Java 3D : la création de chaque point de branchement du graphe scénique représente la plus grande partie de la programmation. Plutôt que de développer cette recette, le prochain point décrit une méthode plus simple pour construire un graphe scénique très similaire avec moins de programmation.

## 1.4.1 Une recette simplifiée pour écrire des programmes Java 3D

Les programmes Java 3D écrits avec la recette de base ont une construction identique de l'arborescence de visualisation. Cette construction identique de la branche visualisation se retrouve dans la classe utilitaire nommée SimpleUniverse. La classe SimpleUniverse accomplit les étapes 2, 3 et 4 de la recette précédente. L'utilisation de la classe SimpleUniverse diminue le temps et l'effort nécessaire à la création de la branche de visualisation. Par conséquent, le programmeur peut consacrer plus de temps aux volumes. La création des volumes devenant ainsi l'essentiel de l'écriture de programmes Java 3D.

Le SimpleUniverse est un bon point de départ a la programmation Java 3D, parce qu'il permet au programmeur d'ignorer toute la branche de visualisation. Toutefois, l'utilisation du SimpleUniverse ne permet pas pour autant d'avoir plusieurs points de visualisation de l'univers virtuel.

La classe SimpleUniverse est utilisée dans tous les programmes d'exemples de ce tutorial. Les programmeurs qui désirent plus d'informations sur les classes View, ViewPlatform, PhysicalBody, et PhysicalEnvironment peuvent ce reporter sur d'autres références. Voir l'appendice B du Chapitre 0 pour la liste des références.

**La classe SimpleUniverse**

Le constructeur d'objet SimpleUniverse crée un graphe scénique incluant les objets VirtualUniverse et Locale (scène) ainsi qu'une arborescence visuelle complète. Cette arborescence visuelle crée par le SimpleUniverse utilise des instances des classes ViewingPlatform (plate-forme de visualisation) et Viewer

(visionneuse), comme classes racine pour le point de branchement de visualisation. Donc le SimpleUniverse utilise, mais indirectement, les objets ViewingPlatform et View de la racine de Java 3D. L'objet SimpleUniverse fournit toutes les fonctions des objets compris dans la grande boite du schéma 1-7.

Le package `com.sun.j3d.utils.universe` contient les classes utilitaires SimpleUniverse, et les raccourcis vers ViewingPlatform et Viewer.

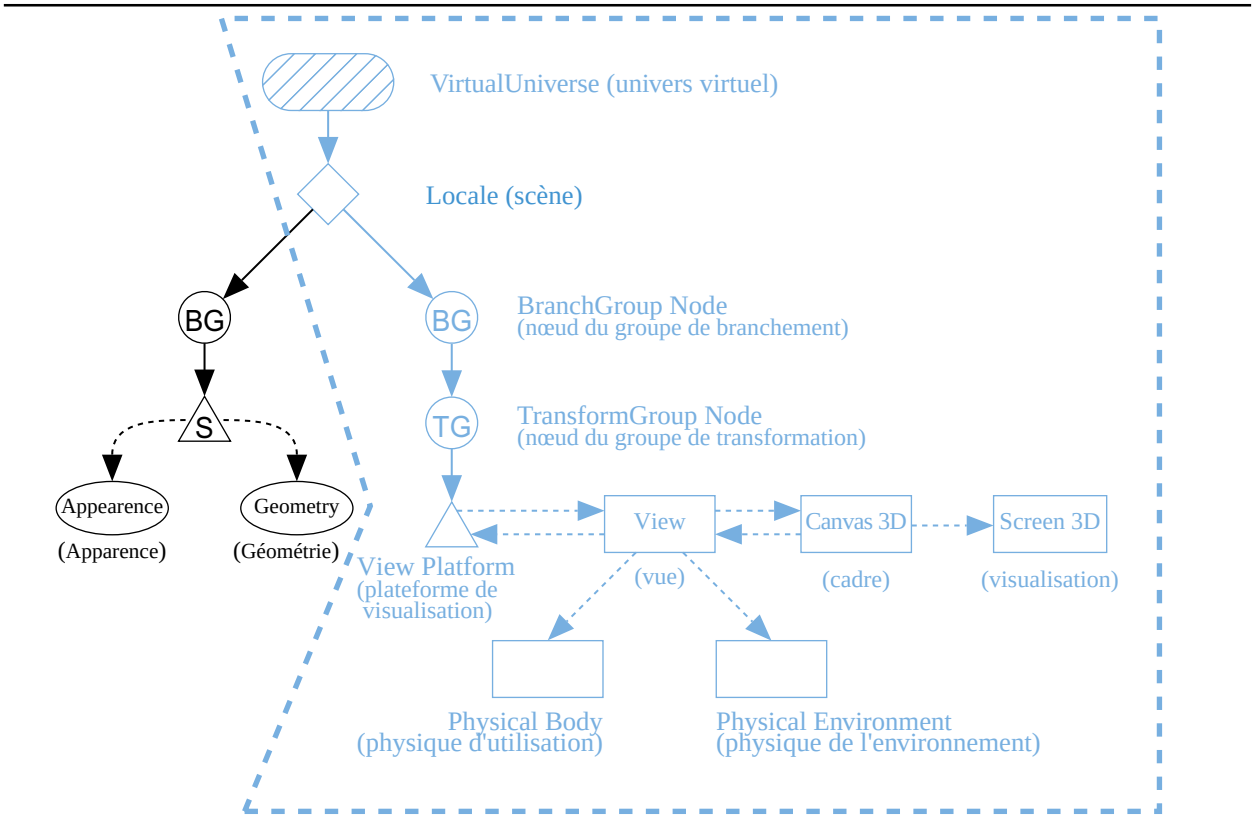



**Figure 1-7 Un objet SimpleUniverse fournit un univers virtuel minimal, indiquée par la ligne pointillée.**

L'utilisation de l'objet SimpleUniverse simplifie la première recette. Le schéma 1-8 montre la recette simplifiée, modifiant ainsi la recette de base pour l'utilisation de l'objet SimpleUniverse. Les étapes 2, 3 et 4 de la première recette sont remplacées par l'étape 2 dans la nouvelle recette.

1. Création d'un objet Canvas3D
2. Création de l'objet SimpleUniverse faisant référence à l'objet Canvas3D
    a. Adaptation de l'objet SimpleUniverse
3. Construction du point de branchement de volume
4. Compilation de la branche volume
5. Insertion du point de branchement volume à la Locale (scène) du SimpleUniverse

**Figure 1-8 Recette simple pour écrire des programmes Java 3D en utilisant le SimpleUniverse.**

La surface grisée sur la page suivante est le premier exemple de Bloc de référence. Un Bloc de référence liste les constructeurs, méthodes, et champs d'une classe. Les Blocs de référence sont conçut pour faciliter au lecteur l'apprentissage de bases de l'API Java 3D. Les Blocs de référence de ce tutorial ne présentent

pas tous les constructeurs et méthodes d'une classe. Il y a donc certaines classes sans Blocs de référence dans ce tutorial. Ce tutorial ne remplace pas la Documentation de l'API Java 3D. Cependant, les constructeurs, méthodes, et champs décrits dans les Blocs de référence du tutorial fournissent des informations plus détaillées que dans la Documentation de l'API Java 3D.

**Les Constructeurs du SimpleUniverse**

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

Cette classe dresse un environnement d'utilisation minimal pour mettre facilement et rapidement un programme Java 3D en place et en exécution. Cette classe crée tous les objets nécessaires à la branche de visualisation. Précisément, cette classe crée les objets suivants : Locale, VirtualUniverse, ViewingPlatform, et Viewer (tous avec leurs paramètres par défaut). Les objets sont mis en héritage de façon correcte pour former le point de branchement visuel.

Le SimpleUniverse fournit toutes les fonctionnalités nécessaires pour la plupart des applications simples en Java 3D. Les classes Viewer et ViewingPlatform sont des raccourcis publics. Ces classes utilisent les classes View et ViewPlatform de la racine.

**`SimpleUniverse()`**
Construit un univers virtuel simple.

**`SimpleUniverse(Canvas3D canvas3D)`**
Construit un univers simple avec une référence à l'objet nommé Canvas3D (espace 3D).

L'objet SimpleUniverse crée la branche de visualisation complète pour un univers virtuel. La branche de visualisation contient une image plate. L'image plate est une surface rectangulaire sur laquelle est projeté le volume pour créer l'image de rendu. L'objet Canvas3D, qui produit une image dans une fenêtre sur votre écran d'ordinateur, affiche l'image plate.

La Figure 1-9, montre les relations d'héritage entre l'image plate, la position de l'œil, et l'univers virtuel. L'œil est situé derrière l'image plate. Les objets visuels en face de l'image plate sont envoyés à l'image plate. Le rendu peut être vu comme une projection des objets visuels sur l'image plate. Ce concept est illustré par les quatre projecteurs sur le schéma (lignes pointillées)

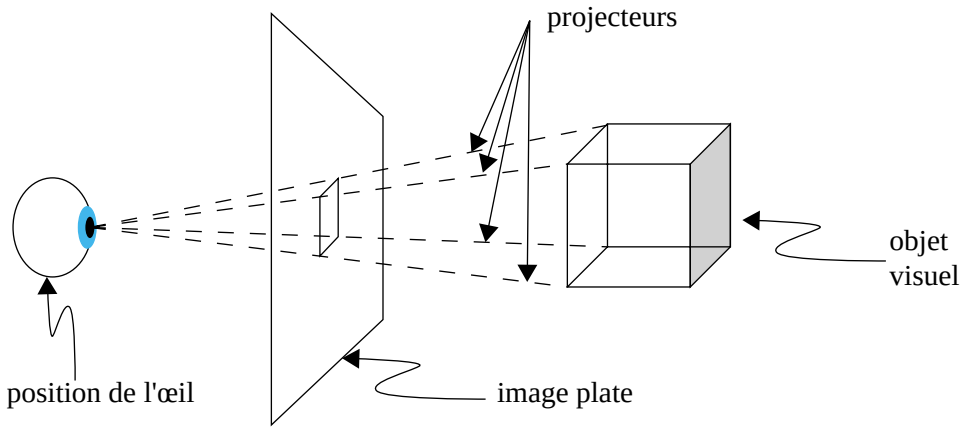



**Figure 1-9 Schéma conceptuel d'une Image Plate et de la position de l'œil dans un univers virtuel.**

Par défaut, l'image plate est centrée à l'origine du SimpleUniverse. L'orientation par défaut est dirigée vers le bas suivant l'axe z. Depuis cette position, l'axe x est une ligne qui traverse horizontalement l'image plate, avec les valeurs positives à gauche. L'axe y est une ligne qui traverse verticalement et au centre l'image plate, avec les valeurs positives en haut. Par conséquent, le point (0,0,0) est le centre de l'image plate.

Un programme Java 3D typique déplace la vue en arrière (valeur z positive) afin de placer les objets sur ou près de l'origine de la vue. La classe SimpleUniverse possède un objet membre de la classe ViewingPlatform. La classe ViewingPlatform a une méthode setNominalViewingTransform qui positionne l'œil pour être centre à (0, 0, 2.41) regardant dans la direction négative des z vers l'origine.

**La Méthode de la ViewingPlatform `setNominalViewingTransform()`**

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

La classe ViewingPlatform est utilisée pour configurer la branche de visualisation d'un graphe scénique Java 3D dans un objet SimpleUniverse. Cette méthode est normalement utilisée avec la méthode getViewingPlatform de la classe SimpleUniverse.

**`void setNominalViewingTransform()`**

Configure la distance de vision de départ à approximativement 2.41 dans le modificateur de vue (viewing transform) du SimpleUniverse. A cette distance de visualisation et dans cet environnement par défaut, des objets de 2 mètres de hauteur et largeur rentrent dans l'image plate.

Après la création des objets Canvas3D et SimpleUniverse, l'étape suivante est la création du point de branchement de volume. La régularité de construction que l'on trouve dans le point de branchement de vue (qui aboutit à l'utilisation de la classe SimpleUniverse) n'existe pas pour le point de branchement de volume. Le point de branchement de volume diffère d'un programme à l'autre rendant impossible d'émettre une recette de construction détaillée. Cela veut aussi dire qu'il n'y a pas de classe de «simple point de branchement de volume » pour n'importe lesquels des univers que vous voudrez créer.

La création d'une branche volume est le sujet des Parties 1.6, 1.7, et 1.9. La compilation de la branche volume est traitée dans la Partie 1.8. Si vous ne pouvez pas attendre une seconde pour voir un peu de code, voyez le Fragment de code 1-1 qui est un exemple de l'usage de la classe SimpleUniverse.

Après la création de la branche volume, elle est insérée dans l'univers grâce à la méthode addBranchGraph (ajout de point de branchement) du SimpleUniverse. La méthode addBranchGraph prend une instance du BranchGroup (groupe de branchement) comme seul paramètre. Ce BranchGroup est ajouté comme un enfant de l'objet Locale par le SimpleUniverse.

**Les Méthodes du SimpleUniverse (liste partielle)**

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

**`void addBranchGraph(BranchGroup bg)`**

Utilisée pour ajouter des nœuds (Nodes) à l'objet Locale du graphe scénique crée par le SimpleUniverse. Utilisé ici pour ajouter une branche de volume à l'univers virtuel.

**`ViewingPlatform getViewingPlatform()`**

Utilisé pour instancier l'objet ViewingPlatform au SimpleUniverse. Cette méthode est utilisée avec la méthode setNominalViewingTransform() de la ViewingPlatform pour régler la situation du point de vue.

# 1.5 Quelques termes Java 3D

Avant d'aborder le thème de la création du point de branchement de volume, nous allons définir deux termes de Java 3D. Cette partie traite des termes de *vivant* et de *compilé.*

L'insertion d'un point de branchement à une Locale (scène) le rend *vivant*, et par conséquent tous les objets qui partent de ce point de branchement deviennent vivants. Rendre des objets vivants engendre quelques conséquences. Les objets vivants sont susceptibles d'être rendus. Aussi, les paramètres des objets vivants ne peuvent plus être modifiés, à moins que cette *aptitude* leur ai été attribuée de façon spécifique avant que l'objet devienne vivant. Les aptitudes sont détaillés dans la Partie 1.8.2.

Les objets BranchGroup peuvent être *compilés*. La compilation d'un BranchGroup convertit les objets du BranchGroup et tous ces composants dans une forme plus efficiente pour le rendu. Il est recommandé d'effectuer la compilation du BranchGroup comme dernière étape avant de le rendre vivant. Il est bon de compiler seulement les objets BranchGroup qui seront ajoutés à la scène (Locale). La compilation est abordée plus loin dans la Partie 1.8.1.

**La méthode `compile()` du BranchGroup**

**`void compile()`**
Compile le BranchGroup source associé à cet objet en créant et dissimulant un graphe scénique compilée.

Les concepts de compilé et de vivant sont exécutés par la classe sceneGraphObject. Les deux méthodes de la classe sceneGraphObject qui sont liés à ces concepts sont détaillées dans le Bloc de référence qui suit.

**Les méthodes du `sceneGraphObject` (liste partielle)**

Le sceneGraphObject est une super-classe de toutes les classes semblables utilisées pour créer un graphe scénique composé de Group, Leaf et NodeComponent. Le sceneGraphObject fournit un grand nombre de méthodes et de champs communs à toutes ses sous-classes ; deux d'entre eux sont présentés ici. La méthode d'association du sceneGraphObject avec une «compilation » est présentée à la Partie 1.8 Aptitudes et performance.

**`boolean isCompiled()`**
Retourne un élément de la mémoire comprenant des informations booléennes indiquant si le nœud fait partie du graphe scénique qui a été compilée.

**`boolean isLive()`**
Retourne un élément de la mémoire comprenant des informations booléennes indiquant si le nœud fait partie d'un graphe d'une scène vivante.

Notez qu'il n'y a pas ici d'étape « début du rendu » dans la recette de base ou simplifiée. Le rendu Java 3D commence à s'exécuter dans une boucle infinie quand un point de branchement contenant une instance de l'objet View devient vivante dans l'univers virtuel. Une fois démarré, le rendu Java 3D exécute de façon cachée les opérations décrite dans le schéma 1-10.

```
      while(true) {
            Process input
            If (request to exit) break
                  Représente les comportements
                  Traversant le graphe scénique et
                  Rend les objets visuels.
            }
      Cleanup and exit
```

**Figure 1-10 Processus du rendu caché**

La partie précédente expliquait la construction d'un univers virtuel simple sans point de branchement de volume. Le sujet des parties suivantes aborde la création de ce point de branchement de volume. La création des volumes est décrite au moyen de programmes d'exemples.

# 1.6 HelloJava3Da : Un exemple qui suit la recette simple

Un programme typique Java 3D commence par définir une nouvelle classe étendant la classe `Applet`. Dans l'exemple `HelloJava3Da.java` qui se trouve dans le répertoire `examples/HelloJava3D`, `HelloJava3Da` est une classe définie pour étendre la classe Applet. Les programmes Java 3D peuvent être écrit comme des applications, mais l'usage de la classe Applet rend la production plus rapide d'une application fenêtrée.

La classe principale (classe main) d'un programme Java 3D défini une méthode pour construire le point de branchement de volume. Une telle méthode est définie dans l'exemple HelloJava3Da et est nommée createSceneGraph().

Toutes les étapes de la recette simple sont appliquées dans le constructeur de la classe HelloJava3Da. La première étape, créant un objet Canvas3D, est achevée à la ligne 4. L'étape 2, qui crée un objet SimpleUniverse, est terminée ligne 11. L'étape 2a, adaptant l'objet SimpleUniverse, est accomplie à la ligne 16. L'étape 3, construisant la branche de volume, est accomplie par l'appel de la méthode createSceneGraph(). L'étape 4, compilant la branche volume, est exécutée ligne 9. Pour finir, l'étape 5, insérant le point de branchement de volume à la Locale du SimpleUniverse, est achevée ligne 19.

```
1. public class HelloJava3Da extends Applet {
2.    public HelloJava3Da() {
3.          setLayout(new BorderLayout());
4.          Canvas3D canvas3D = new Canvas3D(null);
5.          add(“Center”, canvas3D);
6.
7.          // Met en place une branche volume
8.          BranchGroup scène = createSceneGraph();
9.          scene.compile();
10.
11.         // SimpleUniverse est une Classe Utilitaire de complaisance
12.         SimpleUniverse simpleU = new SimpleUniverse(canvas3D);
13.
14.         // Ce qui suit recule un peu la ViewPlatform
15.         //pour que les objets de la scène puissent être vus.
16.         simpleU.getViewingPlatform().setNominalViewingTransform();
17.
```

```
18.         // Connecte la branche volume au SimpleUniverse
19.         simpleU.addBranchGraph(scene);
20.   } // Fin d'HelloJava3Da (constructeur)
```

**Fragment de code 1-1 de la Classe HelloJava3Da**

La troisième étape de la recette simple doit créer l'arborescence de la branche volume. Un point branchement de volume est créé dans Fragment de code 1-2. C'est certainement le point branchement de volume le plus sommaire possible. Le point branchement de volume crée dans Fragment de code 1-2 contient un objet graphique statique, un ColorCube. Ce ColorCube est placé à l'origine du système de cordonnées du *monde virtuel*. Avec l'emplacement et l'orientation définie par le sens de visualisation et du cube, le cube apparaît alors comme un rectangle au rendu. L'image de rendu sera dévoilée après la présentation de tout le code nécessaire au programme.

```
1. public BranchGroup createSceneGraph() {
2.    // Crée le point d'articulation de la branche volume
3.    BranchGroup objRoot = new BranchGroup();
4.
5.    // Crée une forme simple, terminaison d'un nœud,
6.    // et l'ajoute au graphe scénique.
7.    // ColorCube est une classe utilitaire de base
8.    objRoot.addChild(new ColorCube(0.4));
9.
10.   return objRoot;
11.   } // fin de la méthode createSceneGraph d'HelloJava3Da
12. } // fin de la classe HelloJava3Da
```

**Fragment de code 1-2 Méthode createSceneGraph pour la classe HelloJava3Da.**

La classe HelloJava3Da est une dérivé d'Applet mais le programme est exécutable comme un application par l'usage de la classe MainFrame. La classe Applet sert de classe de base pour rendre plus facile a écrire un programme Java 3D qui s'exécute dans un fenêtre. La MainFrame donne un cadre AWT (fenêtre) permettant à l'applet de s'exécuter comme une application. La taille de la fenêtre, résultat de l'application, est définie dans le constructeur de la classe MainFrame. Le Fragment de code 1-3 présente l'utilisation de la MainFrame dans `HelloJava3Da.java`.

**Le Constructeur MainFrame (liste partielle)**

package: `com.sun.j3d.utils.applet`

La MainFrame produit une applet dans une application. Une classe dérivée d'applet doit avoir une méthode main() qui appelle le constructeur de la MainFrame. La MainFrame est une extension de `java.awt.Frame` et met en œuvre `java.lang.Runnable`, `java.applet.AppletStub`, et `java.applet.AppletContext`. La classe MainFrame est un Copyright (c) 1996-1998 de Jef Poskanzer email: jef@acme.com http://www.acme.com/java/

**`MainFrame(java.applet.Applet applet, int width, int height)`**

Cree un objet MainFrame qui exécute une application comme une applet.

Paramètres:

applet - le constructeur d'une classe dérivée d'une applet. La MainFrame produit un cadre AWT pour cette applet.

largeur - la largeur du cadre de la fenêtre en pixels.

hauteur- la hauteur du cadre de la fenêtre en pixels.

```
1.    // La suite permet d'exécuter cette application
2.    // comme une applet.
3.
4.    public static void main(String[] args) {
5.    Frame frame = new MainFrame(new HelloJava3Da(), 256, 256);
6. } // fin de la méthode main (méthode principale d'HelloJava3Da)
```

**Fragment de code 1-3 Méthode Main() d'HelloJava3Da qui fait appel à la MainFrame**

Les trois Fragments précédents (1-1, 1-2, et 1-3) forment un programme Java 3D complet si les déclarations d'importation adéquates sont utilisées. Les groupes de déclaration d'importation sont nécessaires à la compilation de la classe `HelloJava3Da`. Les classes les plus communes se trouvent dans les packages `javax.media.j3d`, ou `javax.vecmath`. Dans cet exemple, seul la classe utilitaire ColorCube est situé dans le package `com.sun.j3d.utils.geometry`. Par conséquent, la plupart des programmes Java 3D utilisent la déclaration d'importation présentée dans le Fragment de code 1-4, à l'exception de l'import du ColorCube.

```
1. import java.applet.Applet;
2. import java.awt.BorderLayout;
3. import java.awt.Frame;
4. import java.awt.event.* ;
5. import com.sun.j3d.utils.applet.MainFrame;
6. import com.sun.j3d.utils.universe.*;
7. import com.sun.j3d.utils.geometry.ColorCube;
8. import javax.media.j3d.*;
9. import javax.vecmath.* ;
```

**Fragment de code 1-4 Déclarations d'importation pour HelloJava3Da.java**

Dans le programme d'exemple `HelloJava3Da.java`, un objet visuel unique est placé dans une seule scène (Locale). Le graphe scénique qui en résulte est montrée dans la Figure 1-11.

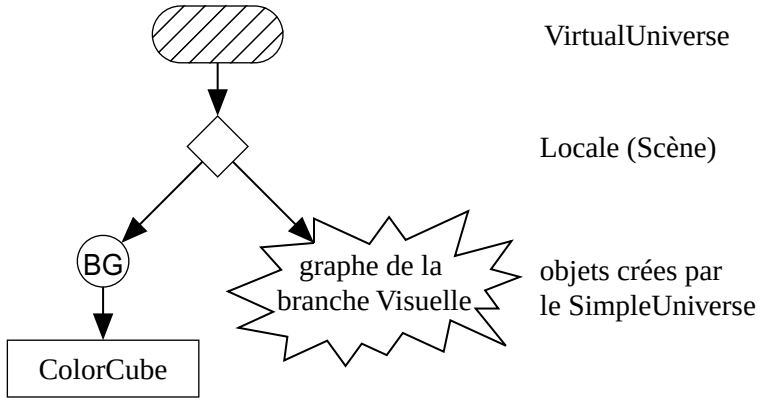



**Figure 1-11 Graphe scénique de l'exemple HelloJava3Da**

Les quatre Fragments précédents (1-1, 1-2, 1-3, and 1-4) produisent le programme d'exemple complet `HelloJava3Da.java`. L'exemple complet se trouve dans le fichier `examples/HelloJava3D`. Compilez le code en lançant la commande : `javac HelloJava3Da.java`. Exécutez le programme avec la commande : `java HelloJava3Da`. L'image produite est montrée dans la Figure1-12.

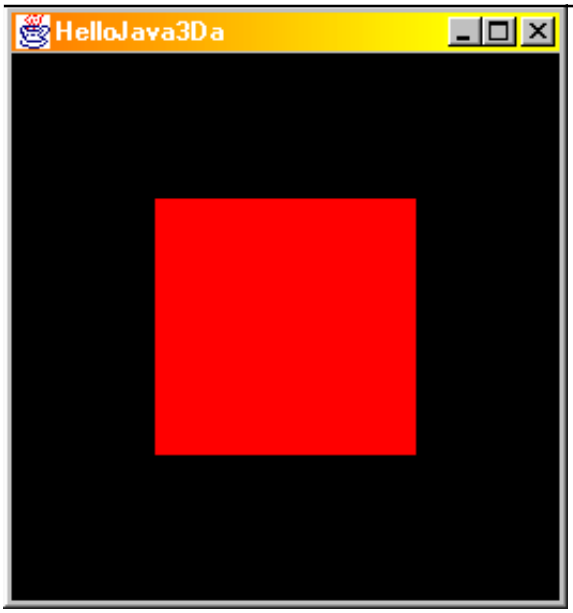



**Figure 1-12 Image produite par HelloJava3Da**

Bien que chaque ligne de code de l'exemple HelloJava3Da ne soit pas entièrement expliqué, les notions de base de l'assemblage d'un programme Java 3D doivent êtres claire à la lecture de l'exemple. La partie suivante détaille plus précisément les classes utilisées par le programme.

## 1.6.1 Les classes Java 3D utilisées dans HelloJava3Da

Pour mieux comprendre l'API Java 3D et l'exemple `HelloJava3Da`, un résumé de chaque classe utilisée dans le programme d'exemple `HelloJava3Da` est présenté ici.

### La classe BranchGroup

Les objets de cette classe sont utilisés pour produire les graphes scéniques. Les instances de BranchGroup sont les racines des points de branchements. Les objets BranchGroup sont les seuls autorisés à être des enfants d'objets Locale. Les objets BranchGroup peuvent avoir de multiples enfants. L'enfant d'un objet BranchGroup peut être d'autres objets Group ou Leaf (terminaison).

**Constructeur par défaut du BranchGroup**

**`Branchgroup ()`**
Les instances de BranchGroup servent de racines pour les branches des graphes; les objets BranchGroup sont les seuls à pouvoir être ajoutés à un ensemble d'objets de la Locale.

### La classe Canvas3D

La classe Canvas3D est dérivée de la classe Canvas de l'Abstract Windowing Toolkit (AWT). Au moins un objet Canvas 3D doit être référencé dans la branche visuelle du graphe scénique [3]. Pour plus d'information sur la classe Canvas, consultez une documentation sur l'AWT. Une liste de référence se trouve dans l'Annexe B.

[3] Il est possible d'en avoir plus d'un. Pour rester simple, en utilisant le SimpleUniverse, il n'y aura qu'une seule instance de Canvas3D dans les programmes présentés ici.

**Le Constructeur du Canvas3D**

**`Canvas3D(GraphicsConfiguration graphicsconfiguration)`**
Construit et initialise un nouvel objet Canvas3D que Java 3D peut rendre en donnant un objet GraphicsConfiguration valide. C'est un extension de la class Canvas de l'AWT. Pour des informations supplémentaires sur l'objet GraphicsConfiguration voir la Documentation de Java 2D, qui est une partie de l'AWT dans le JDK 1.2.

**La classe Transform3D**
Les objets Transform3D représentent les transformations de la géométrie 3D comme la translation et la rotation. Ces objets sont typiquement utilisés dans la création d'objets TransformGroup. Premièrement, on construit l'objet Transform3D, combinaison possible d'objets Transform3D. On construit ensuite l'objet TransformGroup en utilisant l'objet Transform3D.

**Le Constructeur par défaut de Transform3D**

Un objet de transformation est généralement représenté intérieurement par une matrice 4x4 de double précision à virgule flottante. C'est une représentation mathématique de haut-niveau. L'objet Transform3D n'est pas utilisé dans le graphe. Il est utilisé pour déterminer les transformations de l'objet TransformGroup.

**`Transform3D()`**
Construit un objet Transform3D qui représente la matrice identifiée (pas de transformation).

Un objet Transform3D peut représenter une translation, une rotation, un changement d'échelle, ou une combinaison de ces transformations. Pour déterminer une rotation, l'angle est exprimé en radians. 2 PI radians étant une rotation complète. Une manière de déterminer les angles est d'utiliser la constante `Math.PI`. L'autre est de déterminer les valeurs directement. Voici quelques approximations : 45 degrés donne 0.785, 90 degrés donne 1.57, et 180 degrés donne 3.14.

**Les Méthodes de Transform3D (liste partielle)**

Les objets Transform3D représentent des transformations géométriques comme la rotation, la translation, et le changement d'échelle. Transform3D est une des classes qui n'est pas directement utilisée par le graphe. Les transformations représentées par un objet Transform3D sont employées pour créer les objets TransformGroup qui sont eux utilisés dans les graphes scéniques.

**`void rotX(double angle)`**
Règle la valeur cette transformation de rotation autour de l'axe x dans le sens des aiguilles d'une montre. L'angle est déterminé en radians.

**`void rotY(double angle)`**
Règle la valeur cette transformation de rotation autour de l'axe y dans le sens horaire. L'angle est déterminé en radians.

**`void rotZ(double angle)`**
Règle la valeur cette transformation de rotation autour de l'axe z dans le sens horaire. L'angle est déterminé en radians.

**`void set(Vector3f translate)`**
Règle la valeur de translation de la matrice par le paramètre Vector3f, et règle les autres composants de la matrice comme si cette transformation etait un identité matrice.

### La classe TransformGroup

Comme une sous-classe de la classe Group, les instances de TransformGroup sont utilisées dans la création des graphes scéniques et possèdent une collection d'objets nœuds enfants. Les objets TransformGroup conservent les transformations géométriques comme la translation et la rotation. La transformation est crée de façon typique dans un objet Transform3D, celui-ci n'étant pas un objet du graphe scénique.

**Les Constructeurs de TransformGroup**

Les objets TransformGroup sont les détenteurs des transformations dans le graphe scénique.

**`TransformGroup()`**
Construit et initialise un TransformGroup en utilisant une transformation désignée.

**`TransformGroup(Transform3D t1)`**
Construit et initialise un TransformGroup depuis l'objet Transform3D qui lui est transmis.
Paramètres:
t1 - l'objet transform3D

La transformation conservée par l'objet Transform3D est copié vers un objet TransformGroup soit quand l'objet TransformGroup est crée, ou par l'usage de la méthode setTransform().

**La Méthode de TransformGroup `setTransform()`**

**`void setTransform(Transform3D t1)`**
Règle le composant de la transformation de ce TransformGroup à la valeur de la transformation transmise.

Paramètres:
t1 - la transformation à copier.

### La classe Vector3f

Vector3f est une classe mathématique située dans le package `javax.vecmath` pour déterminer un vecteur utilisant trois valeurs à virgule flottante. Les objets Vector sont souvent utilisés pour designer des translations de la géométrie. Les objets Vector3f ne sont pas utilisée directement dans la construction du graphe scénique. Ils sont utilisés pour déterminer des translations, des surfaces, des lignes, ou d'autres choses.

**Les constructeurs de Vector3f**

Un vecteur de trois éléments est représenté par un simple point précis à virgule flottante de coordonnées x, y et z.

**`Vector3f()`**
Construit et initialise un Vector3f à (0,0,0).

**`Vector3f(float x, float y, float z)`**
Construit et initialise un Vector3f depuis les coordonées spécifiées x,y,z..

**La classe ColorCube**

ColorCube est une classe utilitaire du package `com.sun.j3d.utils.geometry` qui définit la géométrie et les couleurs d'un cube placé à l'origine avec des couleurs différentes sur chaque face. Les dimensions par défaut de l'objet ColorCube sont de 2 mètres de hauteur, largeur et profondeur. Si on place un cube sans lui attribuer de rotation (comme dans HelloJava3Da), la face rouge est visible dans le cadre de visualisation. Les autres couleurs sont bleu, magenta, jaune, vert, et cyan.

**Les Constructeurs du ColorCube**

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

Un ColorCube est un objet visuel composé de vertex colorés simples avec des couleurs différentes pour chaque face. ColorCube étend la classe Shape3D ; c'est donc un nœud de terminaison (Leaf). Le ColorCube est simple d'utilisation quand on le pose dans un univers virtuel.

**`ColorCube()`**
Construit un cube de couleur de taille par défaut. Par défaut, un coin est placé à 1 mètre de chaque axes depuis l'origine, donnant un cube centré à l'origine et dont tous les cotés font 2 mètres.

**`ColorCube(double scale)`**
Construit un cube de couleur mis à l'échelle par la valeur spécifiée. La taille par défaut étant de 2 mètres pour chaque coin. Le ColorCube qui en résulte place les coins aux position (scale, scale, scale) et (-scale, -scale, -scale).

# 1.7 Affecter une rotation au Cube

On peut faire effectuer une rotation simple au cube afin d'en visualiser les autres faces. La première étape est la création de la transformation désiré par l'utilisation de l'objet Transform3D.

Le Fragment de code 1-5 introduit un objet TransformGroup dans le graphe scénique pour tourner le cube sur l'axe des x. La transformation de rotation est d'abord crée par l'objet **rotate** de Transform3D. L'objet Transform3D est crée à la ligne 6. La rotation est spécifiée à la ligne 8 par la méthode rotX(). L'objet TransformGroup est ensuite crée à la ligne 10, portant en argument la transformation de rotation.

Deux paramètres déterminent la rotation : l'axe de révolution, et l'angle de rotation. L'axe de rotation est choisi par la méthode appropriée. L'argument de la méthode définit la valeur de l'angle de rotation. Puisque l'angle de rotation est exprimé en radians, la valeur P /4 est 1/8 d'une rotation complète, ou bien 45 degrés.

Après la création de l'objet Transform3D, **rotate**, il est employé pour la création de l'objet de TransformGroup objRotate (ligne 11). C'est l'objet Transform3D qui est utilisé dans le graphe scénique. L'objet **objRotate** prend ensuite comme enfant l'objet ColorCube (ligne 12). Ensuite, l'objet **objRoot** prend **ObjRotate** comme enfant (ligne 13). Les méthodes Transform3D rotX(), rotY, et rotZ() sont présentées dans un Bloc de référence précèdent.

```
1. public BranchGroup createSceneGraph() {
2. // Création de la racine du point de branchement
3. BranchGroup objRoot = new BranchGroup();
4.
5. // l'objet de rotation possède une seule une matrice de
6. //transformation
7.
8. Transform3D rotate = new Transform3D();
9.
10. rotate.rotX(Math.PI/4.0d);
11.
12. TransformGroup objRotate = new TransformGroup(rotate);
13. objRotate.addChild(new ColorCube(0.4));
14. objRoot.addChild(objRotate);
15. return objRoot;
16. } // Fin de la méthode createSceneGraph
```

**Fragment de code 1-5 Rotation Unique dans la branche volume.**

La branche volume contient maintenant un objet TransformGroup sur son chemin d'arborescence allant vers le ColorCube. Chaque objet du chemin d'arborescence de la scène est nécessaire L'objet BranchGroup est le seul qui puisse être l'enfant d'une Locale (scène). L'objet TransformGroup est le seul qui puisse changer la position, l'orientation, et la taille d'un objet visuel. Dans l'exemple, l'objet TransformGroup change l'orientation. Bien sûr, l'objet ColorCube est nécessaire pour produire un objet visible.

La branche de volume produite par le Fragment de code est schématisée dans la Figure1-13.

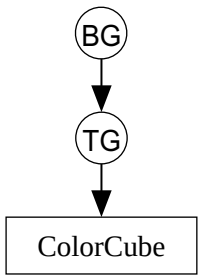



**Figure 1-13 Arborescence de la branche volume du graphe scénique crée par le Fragment de code 1-5.**

Le Fragment de code n'est pas utilisé dans le répertoire d'exemples. Il s'agit seulement d'un petit pas vers des programmes plus complets et intéressants. Le problème important, abordé toute suite après, est de combiner deux objets de transformation dans un seul objet TransformGroup.

## 1.7.1 Exemple de combinaison de transformations : HelloJava3Db

Très souvent les objets visuels sont translatés ou subissent une rotation sur un ou deux axes. Dans ces cas, deux transformations différentes sont définies pour un seul objet visuel. Ces deux transformations peuvent être combinées dans une seule matrice de transformation et portées par un seul objet TransformGroup. Un exemple de ce cas est présenté dans le Fragment de code1-6.

Le programme d'exemple HelloJava3Db combine deux rotations. Réaliser deux rotation simultanées requiert deux objets Transform3D. L'exemple tourne le cube sur les axes x et y. Deux objets Transform3D, un par rotation, sont crées (ligne 7 et 8). Les attributs de chaque rotation sont spécifiés pour les deux objets Transform3D (ligne 10 et 11) . Puis les rotations sont combinées par la multiplications des objets Transform3D (ligne 12). La combinaison des deux transformations est ensuite chargée dans l'objet TransformGroup (ligne13).

```
1. public BranchGroup createSceneGraph() {
2.          // Crée le point d'articulation du branchement
3.          BranchGroup objRoot = new BranchGroup();
4.
5.          // L'objet en rotation a deux matrices de transformation
6.          // de transformation assemblées
7.          Transform3D rotate = new Transform3D();
8.          Transform3D tempRotate = new Transform3D();
9.
10.         rotate.rotX(Math.PI/4.0d);
11.         tempRotate.rotY(Math.PI/5.0d);
12.         rotate.mul(tempRotate);
13.         TransformGroup objRotate = new TransformGroup(rotate);
14.
15.         objRotate.addChild(new ColorCube(0.4));
16.         objRoot.addChild(objRotate);
17.         return objRoot;
```

**Fragment de code 1-6 Deux Transformations de Rotation pour HelloJava3Db.**

L'un ou l'autre des Fragments de code 1-5 ou 1-6 peut remplacer le Fragment de code 1-2 d'HelloJava3Da pour créer un nouveau programme. Le Fragment de code 1-6 est utilisé dans `HelloJava3Db.java`. Le programme d'exemple complet, qui combine deux rotations, apparaît dans le fichier `HelloJava3Db.java` situé le dossier `examples/HelloJava3D/`. Ce programme tourne sous la forme d'application comme HelloJava3Da.

Le graphe scénique crée par `HelloJava3Db` est décrite par la Figure 1-14. La branche visualisation est la même que celle engendrée par `HelloJava3Da`, qui est construite par le SimpleUniverse et représenté par une étoile. La branche volume possède maintenant un TransformGroup dans son arborescence vers l'objet ColorCube.

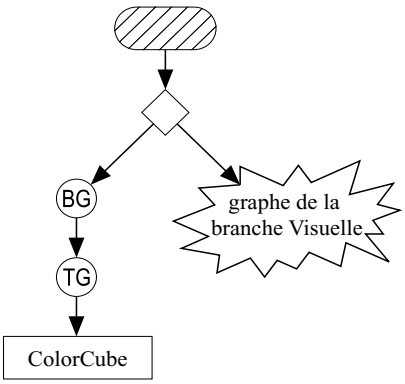



**Figure 1-14 Schéma du graphe scénique de l'exemple HelloJava3Db.**

L'image de la Figure 1-15 montre le ColorCube en rotation d'HelloJava3Db.

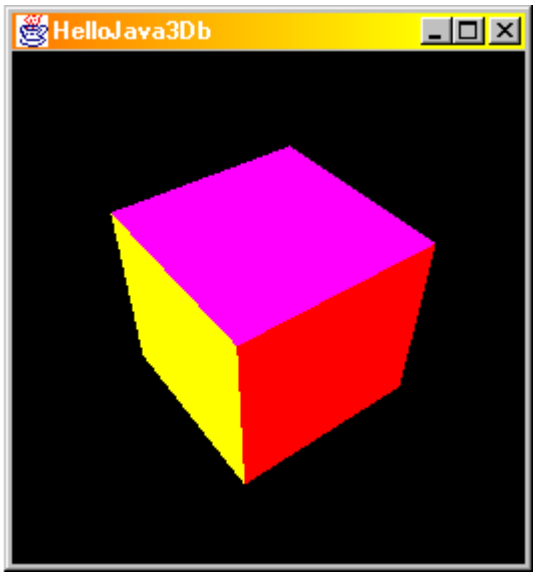



**Figure 1-15 Image de la rotation du ColorCube rendu par HelloJava3Db.**

# 1.8 Aptitudes et performances

Le graphe scénique construit par un programme Java 3D peut être utilisée directement pour le rendu. Pourtant, l'interprétation n'est pas très efficace. La flexibilité interne de chaque objets du graphe scénique (qui n'est pas abordée dans ce tutorial) produit une représentation pas du tout optimisée de ceux-ci dans l'univers virtuel. Une représentation plus efficace de l'univers virtuel doit être utilisée pour améliorer l'exécution du rendu.

Java 3D possède une caractéristique d'interprétation interne et des méthodes pour effectuer la conversion vers une représentation plus efficace. Le systeme Java 3D prend deux voies différentes pour faire la conversion vers une interprétation interne. La première compile chaque point de branchement. L'autre voie insère le point de branchement dans l'univers virtuel pour le rendre vivant.

La compilation du point de branchement est le sujet de la prochaine section. Les effets de cette conversion vers une interprétation interne sont débattus dans la Partie 1.8.2.

## 1.8.1 Compiler les volumes

L'objet BranchGroup possède une méthode de compilation. Appeler cette méthode convertit l'arborescence entière, partant du point de branchement, vers une interprétation Java 3D interne de la branche. En plus de cette conversion elle-même, l'interprétation interne peut être optimisée de plusieurs façons.

Les optimisations possibles ne sont pas spécifiées par l'API Java 3D. Pourtant, le rendement peut être rehaussé de multiples manières. Une voie possible pour l'optimisation est de combiner tout les TransformGroups de l'arborescence partant du point de branchement. Par exemple, si on trouve sur l'arborescence d'une branche du graphe scénique deux objets TransformGroup dans une relation parent enfant, ils peuvent être représentés par un seul objet TransformGroup. L'autre possibilité est de combiner les objets Shape3D qui ont une la même relation statique. Bien d'autres optimisations sont encore possibles.

La Figure 1-16 schématise de façon conceptuelle la conversion en une représentation plus rapide. La branche du graphe scénique du coté gauche du schéma est compilée et transformée dans une représentation interne figurée dans le coté droit du schéma. La Figure représente seulement le concept de l'interprétation interne, mais pas la façon dont Java 3D l'exécute réellement.

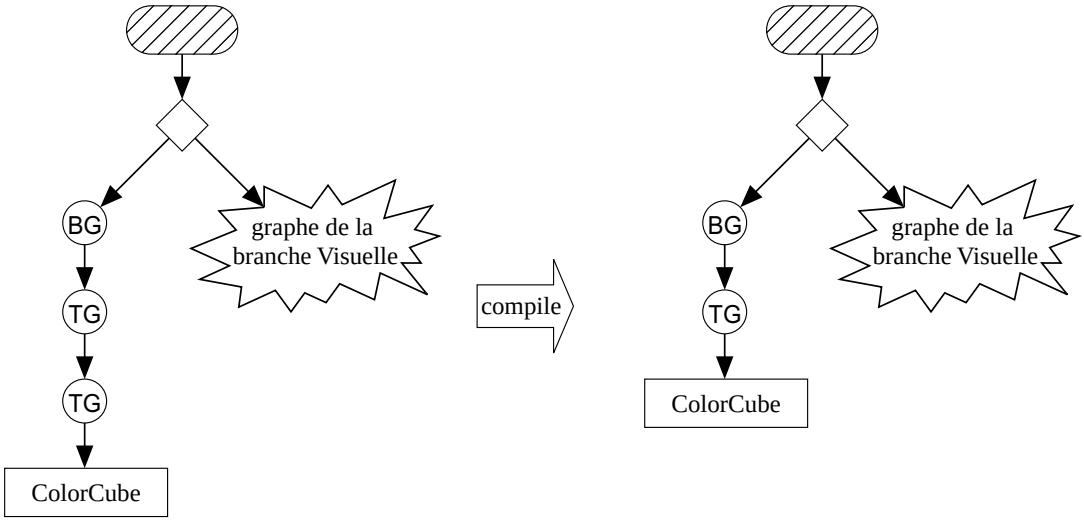



**Figure 1-16 Exemple conceptuel du résultat de la compilation d'une branche du graphe scénique.**

## 1.8.2 Aptitudes

Dès qu'une branche du graphe est rendue vivante ou compilée le système de rendu Java 3D convertit cette branche vers une représentation interne bien plus efficace. L'effet le plus important de cette représentation interne du graphe scénique est l'amélioration des performances du rendu.

Mettre les transformations en représentation interne engendre aussi d'autres effets. L'un d'eux est la stabilisation des valeurs des transformations et celle des autres objets du graphe scénique. A moins de le spécifier dans le programme, celui-ci n'a pas la possibilité de changer la valeur des objets du graphe scénique une fois qu'ils sont vivants.

Il y a des cas où le programme doit pouvoir changer les valeurs d'un objet du graphe scénique après qu'il soit devenu vivant. Par exemple, les modifications de la valeur d'un objet TransformGroup peut créer une animation. Pour que cela se produise, les transformations doivent pouvoir se modifier après leur naissance. La liste des valeurs qui peuvent être accédés de cette manière, sont appelés *aptitudes* [capabilties] de l'objet.

Chaque sceneGraphObject possède un jeu de bits d'aptitudes. Les valeurs de ces bits déterminent quelle aptitude existe pour cet objet après qu'il soit devenu vivant ou compilé. Le nombre d'aptitudes varie selon les classes.

**Méthodes du sceneGraphObject (liste partielle)**

sceneGraphObject est la superclasse de quasiment toutes les classes employés à la création d'une branche du graphe scénique, incluant des Group, Leaf, et NodeComponent. La Partie 1.5 présente d'autres méthodes du sceneGraphObject.

**`void clearCapability(int bit)`**
Efface l'aptitude du bit spécifié.

**`boolean getCapability(int bit)`**
Préserve l'aptitude du bit spécifié.

**`void setCapability(int bit)`**
Etablit l'aptitude du bit spécifié.

Par exemple, pour rendre possible la lecture de la valeur de la transformation représentée par un objet TransformGroup, cette aptitude doit être établie avant qu'elle soit compilée ou rendu vivante. De la même manière, afin d'être capable de changer la valeur de la transformation dans un objet TransformGroup, son aptitude de transformation doit être fixée avant qu'elle soit compilée ou rendue vivante. Le Bloc de référence suivant qui montre les aptitudes des classes TransformGroup non-héritées. Tenter de changer une aptitude non établie d'un objet vivant ou compilé engendre une exception.

Dans les prochains parties, des animations seront crées par l'emploi d'une transformation de rotation évoluant dans le temps. Afin que cela soit possible, l'objet TransformGroup doit posséder l'aptitude établie ALLOW_TRANSFORM_WRITE avant d'être compilé ou rendu vivant.

**Aptitudes du TransformGroup (liste partielle)**

Les deux aptitudes listées ici sont les seules définies pour un TransformGroup. Le TransformGroup reçoit en héritage un nombre de bits d'aptitudes de ses classes ancestrales: Group et Node. Les paramètres d'aptitude sont établies, relancées, ou préservés par l'usage de la méthode définie dans le sceneGraphObject.

**`ALLOW_TRANSFORM_READ`**
Désigne les nœuds du TransformGroup qui sont autorisées à accéder aux informations de transformation de cet objet.

**`ALLOW_TRANSFORM_WRITE`**
Désigne les nœuds du TransformGroup qui sont autorisées à écrire des informations de transformation de cet objet.

Les aptitudes contrôlent également l'accès aux autres aspects d'un objet TransformGroup. L'objet TransformGroup reçoit en héritage les paramètres d'aptitude de ses classes ancêtres: Group et Node. Quelques aptitudes de Group sont indiquées dans le Bloc de référence suivant.

**Aptitudes du Group (liste partielle)**

TransformGroup reçoit en héritage un nombre de bits d'aptitudes de ses classes ancestrales.

**`ALLOW_CHILDREN_EXTEND`**
Etablit cette aptitude, accordée aux enfants d'être ajoutés au Group de nœuds après la compilation ou la naissance.

**`ALLOW_CHILDREN_READ`**
Etablit cette aptitude, accordée aux références des enfants du Group de nœuds la possibilité d'être lus après la compilation ou la naissance.

**`ALLOW_CHILDREN_WRITE`**
Etablit cette aptitude, accordée aux références des enfants du Group de nœuds la possibilité d'être écrit (modifié) après la compilation ou la naissance.

# 1.9 Ajout de comportements d'Animation

En Java 3D, un Behavior (comportement) est une classe pour désigner des animations ou des interactions avec les objets visuels. Le comportement peut changer de fait n'importe quels attributs d'un objet visuel. Le programmeur peut utiliser un nombre de comportements prédéfinis ou spécifier un comportement. Dès qu'un comportement est indiqué à un objet visuel, le système Java 3D met automatiquement à jour la position, l'orientation, la couleur, ou les autres attributs, de l'objet visuel.

La différence entre animation et interaction est que le comportement est activé, respectivement, par l'écoulement du temps ou par les activités de l'utilisateur.

Chaque objet visuel de l'univers virtuel peut avoir son propre comportement. En fait, un objet visuel peu avoir de multiples comportements. Pour spécifier le comportement d'un objet visuel, le programmeur crée des objets spécifiants les comportements, ajoute ces objets visuels au graphe scénique, et crée les relations appropriées entre les objets du graphe scénique et les comportements.

Dans un univers virtuel avec plusieurs comportements, une importante quantité de calcul peut être requise seulement pour le calcul des comportements. Comme le rendu et les comportements utilisent le même processeur(s), il est probable que la puissance de calcul exigée puisse dégrader les performances du rendu [4].

Java 3D permet au programmeur de contrôler ce problème en spécifiant au comportement une limite spatiale dans lequel le comportement prend place. Cet limite est nommée *région planifiée [scheduling region]*. Un comportement n'est actif que lorsque le *volume d'activation* de la ViewPlatform intersecte avec la région planifiée du comportement. En d'autre termes, s'il n'y a pas quelqu'un dans la foret pour voir l'arbre tomber, il ne tombe pas. Le dispositif de la région planifiée rend Java 3D plus efficace dans le traitement d'un univers virtuel avec plusieurs comportements.

Un Interpolator (déterminant) est une des nombreuses classes de comportements prédéfinis présents dans le package racine de Java 3D, qui sont des sous-classes de Behavior. Fondé sur un argument temporel, l'objet Interpolator manipule les paramètres des objets du graphescénique. Prenons l'exemple du RotationInterpolator (déterminant de rotation), il actionne la rotation spécifiée par un TransformGroup pour affecter cette rotation aux objets visuels qui sont les ancêtres du TransformGroup.

La Figure 1-17 dénombre les étapes invoquées dans la création d'une animation avec un objet déterminant. Les cinq étapes de la Figure 1-17 forment la recette de création du déterminant d'un comportement d'animation.

1. Créer un TransformGroup cible.
   Lui attribuer l'aptitude ALLOW_TRANSFORM_WRITE.
2. Créer un objet Alpha.
   Spécifier les paramètres temporels de l'alpha.
3. Créer l'objet déterminant.
   Possédant comme référence les objets Alpha et TransformGroup.
   Personnaliser les paramètres de comportement.
4. Déterminer une région de planification.
   Fixer la région de planification pour le comportement.
5. Amener le comportement à devenir un enfant du TransformGroup.

**Figure 1-17 Recette pour ajouter des Behaviors à des objets visuels 3D.**

[4] Des processeurs graphiques spéciaux peuvent être impliqués dans le processus de rendu en plus de l'environnement matériel et de l'exécution de Java 3D. Il est quand même possible d'avoir trop de comportements dans l'univers virtuel pour ne pas rendre assez rapidement.

## 1.9.1 Déterminer un comportement d'animation

Un comportement peut avoir comme action sur un objet visuel la modification de sa position (PositionInterpolator), de son orientation (RotationInterpolator), de sa taille (ColorInterpolator), ou de sa transparence (TransparencyInterpolator). Comme il est dit précédemment, les Interpolators (déterminants) sont des classes prédefinies de comportement. Tous les comportements mentionnés sont possibles sans utiliser de déterminant, pourtant, les déterminants facilitent la création de comportements. Les classes d'Interpolators (déterminants)fournissent d'autres actions, y compris la combinaisons de ces actions. Ces classes sont présentés en détail dans la Partie 5.2 mais aussi dans la Documentation de l'API. La classe RotationInterpolator sera utilisée l'exemple qui suit.

**La classe RotationInterpolator**

Cette classe sert à déterminer un comportement de rotation pour un objet visuel ou groupe d'objets visuels. Un objet RotationInterpolator modifie un objet TransformGroup par une rotation spécifique qui répond à la valeur de l'objet Alpha. Si la valeur de l'objet Alpha change dans le temps, la rotation change aussi. Un objet RotationInterpolator est souple dans sa définition des axes de rotations, d'angle de départ, d'angle final. Pour des rotations constantes, la construction de l'objet RotationInterpolator peut utiliser les règles suivantes :

**Constructeur du RotationInterpolator (liste partielle)**

Cette classe défini un comportement qui modifie le composant de rotation de sa cible TransformGroup par une interpolation linéaire entre une paire d'angles connus (utilisant la valeur générée par l'objet Alpha). L'angle interpolé est utilisé pour former une transformation de rotation.

`RotationInterpolator(Alpha alpha, TransformGroup target)`
Ce constructeur utilise les valeurs par défaut de quelques paramètres du déterminant afin de construire une rotation complète autour de l'axe y par l'usage du déterminant désigné au TransformGroup.
paramètres:
alpha - la fonction de variation temporelle à référencer.

La cible de l'objet TransformGroup d'un déterminant doit avoir une aptitude d'écriture. Les informations sur les aptitudes sont présentés dans la Partie 1.8.

## 1.9.2 Fonctions de variation temporelle : Appliquer un comportement de temps

On applique une action au temps par l'usage de l'objet Alpha. La description d'un objet Alpha peut être compliquée. Quelques notions de base de la classe Alpha sont présentés ici.

**Classe Alpha**

Les objets de la classe Alpha sont utilisés pour créer une fonction de variation temporelle. La classe Alpha produit une valeur comprise entre zéro et un. De la valeur produite dépend le temps et les paramètres de l'objet Alpha. Les objets Alpha sont généralement employés avec un objet Interpolator de comportement pour animer des objets visuels.

Il y a dix paramètres d'Alpha, donnant ainsi au programmeur une énorme souplesse. Sans entrer dans le détail de chaque paramètre, sachez qu'une instance d'un Alpha peut facilement être combinée avec un comportement produisant ainsi des rotations simples, des oscillations pendulaires, et des événements uniques comme l'ouverture d'une porte, ou le lancement d'une fusée.

**Constructeur d'Alpha**

La classe Alpha fournit des objets pour convertir la temps en une valeur alpha (une valeur comprise entre 0 et 1). L'objet Alpha est effectivement une fonction de temps produisant une valeur alpha comprise entre zéro et un. Un usage des objets Alpha est de produire une valeur pour les Interpolators (Déterminants) de comportement. La fonction f(t) et les caractéristiques de l'objet Alpha sont définies par des paramètres modifiables par l'utilisateur :

**`Alpha()`**
Boucle continue avec une période de une seconde.

**`Alpha(int loopCount, long increasingAlphaDuration)`**
Ce constructeur prend comme seuls paramètres le loopCount et l'increasingAlphaDuration et assigne des valeurs par défaut a tous les autres paramètres. L'objet Alpha qui en résulte produit une valeur commençant à zéro et augmentant vers un. Elle se répète le nombre de fois définit par la loopCount. Si le loopCount prend la valeur -1, l'objet Alpha se répète indéfiniment. Le temps qu'il prend pour aller de zéro a un est définit dans le second paramètre utilisant une échelle de temps en milli-secondes.
Paramètres:
loopCount - nombre de fois que dois ce répéter cet objet alpha objet; une valeur de -1 détermine une boucle infinie à alpha.
increasingAlphaDuration - temps en milli-secondes que prend alpha pour aller de zéro à un.

## 1.9.3 Région planifiée [scheduling region]

Comme il est mentionné dans la Partie 1.9, chaque comportement possède une limite de planification. Les limites de la région pour un comportement sont fixées par l'usage de la méthode setSchedulingBounds de la classe Behavior.

Il y a de nombreuses manières de définir un région planifiée, la plus simple d'entre elles est de créer un objet BoundingSphere. D'autres options forment une région de forme carrée ou polygonale. La classe BoundingSphere est décrite ci dessous. Pour les classes BoundingBox et BoundingPolytope, le lecteur peut se référer à la Documentation de l'API.

**Méthode du Behavior (comportement) setSchedulingBounds**

**`void setSchedulingBounds(Bounds region)`**
Fixe la région planifiée du Behavior à la limite spécifiée.
Paramètres:
Region - La délimitation qui contient la région planifiée du Behavior.

**BoundingSphere Class**

Détermine une limite sphérique avec comme paramètres un point central et un rayon. L'utilisation normale de la délimitation sphérique prend comme centre le point (0, 0, 0). Le rayon est ensuite défini pour que la sphère contienne l'objet visuel, incluant toutes les position possibles de cet objet.

**Constructeurs de la Bounding Sphère (liste partielle)**

Cette classe définit une région planifié sphérique déterminée par un point central et un rayon.

**`BoundingSphere()`**
Ce constructeur crée une limite sphérique centrée a l'origine avec un rayon de 1 unité.

**`BoundingSphere(Point3d center, double radius)`**
Construit et initialise une BoundingSphere utilisant le point central et le rayon spécifiés.

## 1.9.4 Exemple de Behavior (comportement) : HelloJava3Dc

Le Fragment de code 1-7 montre un exemple complet de l'usage d'une classe déterminant pour créer une animation. L'animation produite par ce code est une rotation ininterrompue avec une rotation complète de quatre secondes. Le Fragment de code 1-7 suit la recette donnée dans la Figure 1-17, la liant avec le déterminant de l'animation.

La première étape de la recette est la création d'un objet TransformGroup qui sera modifié à l'exécution. La cible de l'objet TransformGroup d'un déterminant doit posséder l'aptitude d'écriture. L'objet TransformGroup nommé objSpin est crée à la ligne 7. L'aptitude de l'objSpin est réglée à la ligne 8 du Fragment de code 1-7.

La deuxième étape de la recette doit créer un objet alpha pour diriger le déterminant. L'objet Alpha de l'exemple simple du Fragment de code 1-7, rotationAlpha, est utilisé pour déterminer une rotation continue. Les deux paramètres définis par la ligne 16 du Fragment de code 1-7 sont le nombre de répétition de la boucle et le temps pour une révolution. La valeur « -1 » spécifie un nombre de tour ininterrompu. Le temps est indiqué en milli-secondes. La valeur 4000 utilisée dans le programme indique donc 4000 milli-secondes soit 4 secondes. Donc le comportement effectue un révolution complète toutes les quatre secondes.

La troisième étape est la création l'objet déterminant. L'objet RotationInterpolator est crée lignes 21 et 22. Le déterminant doit avoir des référence vers les objets cibles de transformation et alpha. Ce qui est accompli par le constructeur. Dans l'exemple, le comportement par défaut du RotationInterpolator est utilisée. Ce comportement par défaut du RotationInterpolator réalise une rotation complète autour de l'axe y.

La quatrième étape détermine une région planifiée. Dans le Fragment de code 1-7, un objet BoundingSphere est employé avec ces valeurs par défaut. L'objet BoundingSphere est crée ligne 26. La sphère est définie comme limite d'application à la ligne 27. L'étape finale, la cinquième étape, amène le comportement à être un enfant du TransformGroup. Ce qui est réalisé à la ligne 27.

```
1. public BranchGroup createSceneGraph() {
2.    // Crée le point d'articulation de la branche volume
3.    BranchGroup objRoot = new BranchGroup();
4.
5.    // Crée le groupe de nœud effectuant une transformation et initialise
6.    // son identité. L'ajoute à la racine de la branche.
7.    TransformGroup objSpin = new TransformGroup();
8.    objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
9.    objRoot.addChild(objSpin);
10.
11.   // Crée une forme simple de terminaison de nœud, et l'ajoute à la
12.   // branche. ColorCube est une classe utilitaire de base.
13.   objSpin.addChild(new ColorCube(0.4));
14.
15.   // Crée une fonction de variation temporelle pour conduire
16.   // l'animation
17.   Alpha rotationAlpha = new Alpha(-1, 4000);
18.   // Crée un nouvel objet Behavior qui accomplit
19.   // l'opération désirée
20.   // par l'objet de transformation spécifiée
21.   // et l'ajoute à la branche.
21.   RotationInterpolator rotator =
22.         new RotationInterpolator(rotationAlpha, objSpin);
23.
24.   // une sphère de limitation définit
25.   //la région où le Comportement est actif
26.   BoundingSphere bounds = new BoundingSphere();
27.   rotator.setSchedulingBounds(bounds);
28.   objSpin.addChild(rotator);
29.
30.   return objRoot;
31. } // fin de la méthode createSceneGraph
```

**Fragment de code 1-7 méthode createSceneGraph avec un Behavior RotationInterpolator.**

Le Fragment de code 1-7 est utilisé avec des Fragments de code précédents pour former le programme d'exemple `HelloJava3Dc`. `HelloJava3Dc.java` est situé dans le répertoire `examples/HelloJava3D/` et peut être exécuté comme une application. L'exécution de l'application rend un ColorCube effectuant une révolution toutes les quatre secondes.

Le graphe scénique crée par HelloJava3Dc est décrit par la Figure 1-18. L'objet de rotation est à la fois un enfant du TransformGroup **objSpin** et se référence à lui. Bien que cette relation semble violer les restrictions de non-retour du chemin du graphe scénique, il n'est pas nécessaire de rappeler que les liens de références (flèches pointillées) ne font pas partie du graphe scénique. Cette référence est la ligne pointillée partant du Behavior vers le TransformGroup.

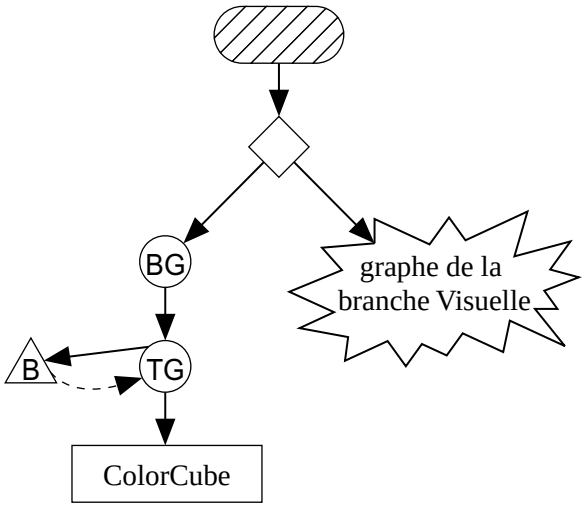



**Figure 1-18 Graphe scénique de l'exemple HelloJava3Dc.**

L'image de la Figure 1-19 montre une capture d'écran du ColorCube en rotation d'HelloJava3Dc.

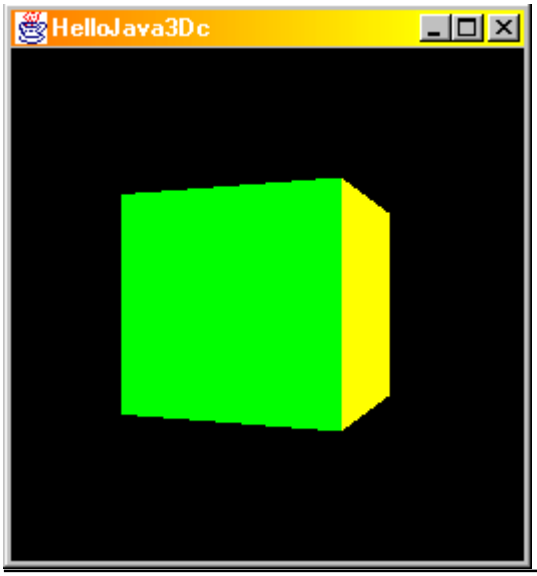



**Figure 1-19 Une image du ColorCube en Rotation comme le rend HelloJava3Dc.**

## 1.9.5 Exemple de combinaison d'une transformation et d'un comportement [Behavior] : HelloJava3Dd

Bien entendu, vous pouvez associer des comportements à la transformation de rotation de l'exemple précédent. Ceci est réalisé dans `HelloJava3Dd.java`. On trouve dans la branche volume du graphe des objets nommés **objRotate** et **objSpin**, qui distinguent respectivement la rotation statique et la révolution continue (comportement de rotation) de l'objet. Le code se trouve dans le Fragment de code 1-8. Le graphe scénique se trouve à la Figure 1-20.

```
1. public BranchGroup createSceneGraph() {
2.    // Crée le point d'articulation de la branche volume
3.    BranchGroup objRoot = new BranchGroup();
4.
5.    // l'objet rotate est composé de matrices de transformation
6.    Transform3D rotate = new Transform3D();
7.    Transform3D tempRotate = new Transform3D();
8.
9.    rotate.rotX(Math.PI/4.0d);
10.   tempRotate.rotY(Math.PI/5.0d);
11.   rotate.mul(tempRotate);
12.
13.   TransformGroup objRotate = new TransformGroup(rotate);
14.
15.   // Crée le groupe de nœud de transformation et initialise son identité
16.   // Autorise une aptitude TRANSFORM_WRITE à notre code de comportement
17.   // pour pouvoir le modifier à l'exécution. L'ajoute à son point de
18.   // branchement racine.
19.   TransformGroup objSpin = new TransformGroup();
20.   objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
21.
22.   objRoot.addChild(objRotate);3.      objRotate.addChild(objSpin);
24.
25.   // Crée une forme simple de terminaison de nœud, et l'ajoute à la
26.   // branche. ColorCube est une classe utilitaire de base.
27.   objSpin.addChild(new ColorCube(0.4));
28.
29.   // Crée un nouvel objet Behavior qui accomplit l'opération désirée
30.   // sur l'objet de transformation spécifié et l'ajoute
31.   // au graphe scénique.
32.   Transform3D yAxis = new Transform3D();
33.   Alpha rotationAlpha = new Alpha(-1, 4000);
34.
35.   RotationInterpolator rotator =
36.         new RotationInterpolator(rotationAlpha, objSpin, yAxis,
37.               0.0f, (float) Math.PI*2.0f);
38.
39.   // Une limitation sphérique définit la région où le comportement est
40.   // actif. Crée une sphère centrée à l'origine et d'un rayon de 1.
41.   BoundingSphere bounds = new BoundingSphere();
42.   rotator.setSchedulingBounds(bounds);
43.   objSpin.addChild(rotator);
44.
45.   return objRoot;
46. } // Fin de la methode createSceneGraph d'HelloJava3Dd
```

**Fragment de code 1-8 Branche volume pour un ColorCube basculé et rotatif d'HelloJava3Dd.**

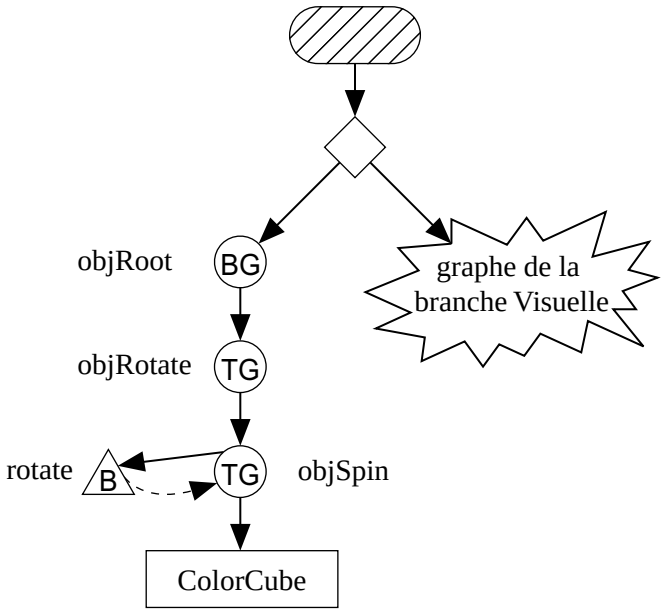



**Figure 1-20 Graphe scénique de l'exemple HelloJava3Dd.**

L'image  dans la Figure 1-21 présente une frame du ColorCube en rotation et basculé d'HelloJava3Dd.

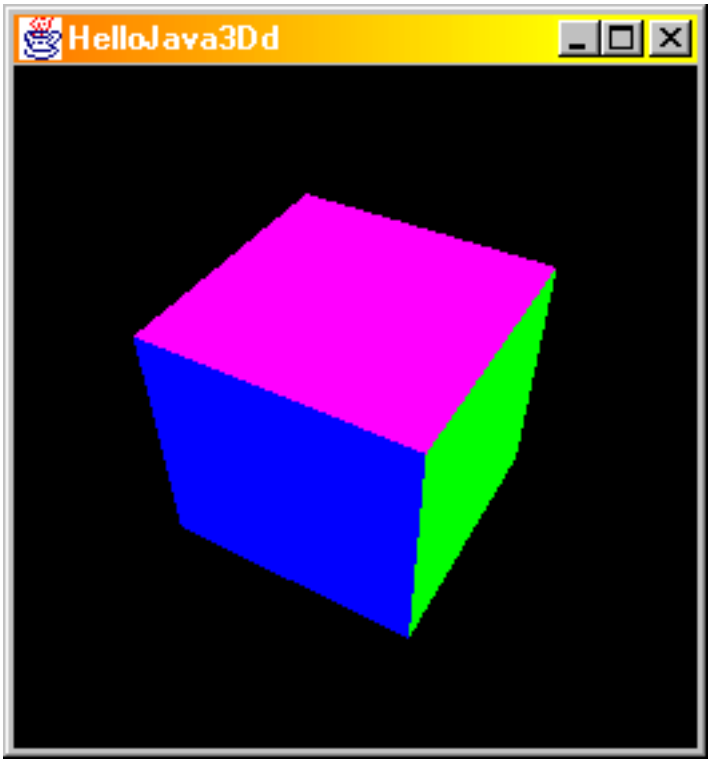



**Figure 1-21 Une image du ColorCube en révolution comme le rend HelloJava3Dd.**

# 1.10 Résumé du chapitre

Ce chapitre assume le fait que le lecteur ne connaisse rien de Java 3D. Au cours de ce chapitre, les classes les plus importantes de l'API Java 3D sont présentées au lecteur. Une explication de la manière dont ces classes, et celles des autres packages, sont utilisées pour créer un graphe scénique. Le graphe scénique, qui décrit un univers virtuel, et comment la visualisation est rendue, est abordée par plusieurs points de détails. La classe SimpleUniverse est utilisée pour monter une série de programmes d'exemples très simples en Java 3D, une transformation unique, une combinaison de transformations, de comportements, et la combinaison d'une transformation et d'un comportement. Plus loin dans le chapitre viennent les explications des aptitudes des objets et la compilation des branches du graphe scénique.

# 1.11 Tests personnels

Cette page contient quelques exercices permettant de tester d'augmenter votre compréhension de la matière présentée dans ce chapitre. Les solutions de ces exercices sont données dans l'Annexe C du Chapitre 0.

1. Dans le programme HelloJava3Db, qui combine deux rotations dans un seul TransformGroup, quelle serait la différence si vous inversiez l'ordre de multiplication dans la définition de la rotation ? Modifiez le programme pour voir si votre réponse est correcte. Il y a deux lignes de code à modifier pour cela.

2. Dans le programme HelloJava3Dd, quelle serait la différence si vous inversiez l'ordre des nœuds de transformation sous le ColorCube dans la branche volume ? Modifiez le programme pour voir si votre réponse est correcte.

3. Dans une recherche d'optimisation des performances, le programmeur veut rendre le graphe scénique moins lourd [6] . Peut-on combiner les cibles des transformations de rotation et de révolution d'HelloJava3Dd dans un seul objet TransformGroup ?

4. Déplacez le ColorCube d'une unité dans l'axe Y et tournez le cube. Vous pouvez utiliser HelloJava3Db comme point de départ. Le code qui suit la question montre l'instruction d'une transformation de translation. Essayez la transformation dans le sens inverse. Voyez-vous une différence au résultat ? Si oui, pourquoi ? Si non, pourquoi ? Essayez et comparez vos espérances au résultat actuel.

```
Transform3D translate = new Transform3D();
Vector3f vector = new Vector3f(0.0f, 1.0f, 0.0f);
translate.setTransform(vector);
```

Dans HelloJava3Dc, la sphère de limitation d'effet a un rayon de 1 mètre. Cette valeur est-elle trop grande ou trop petite que ce qu'elle doit être ? Quelle est la plus petite valeur qui permettras une rotation du cube dans la visualisation ? Expérimentez sur le programme pour vérifier vos réponses. Les lignes de code qui suivent peuvent être utilisées pour définir une sphère de limitation d'effet. Dans ces lignes, l'objet Point3D spécifiant son centre, suivit du rayon.

```
BoundingSphere bounds =
new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
```

6. Les programmes d'exemples donnent suffisamment d'informations pour assembler un univers virtuel avec plusieurs cubes colorés. Comment construiriez vous un tel graphe scénique ? Dans quelle partie du code cela doit-il être réalisé ?

[6] La performance est directement rattachée à la taille du graphescénique. La modification la plus efficace est de réduire le nombre d'objets Shape3D dans le graphe scénique. Se rapporter au « livre blanc » sur les performances de Java 3D disponible à java.sun.com/docs.