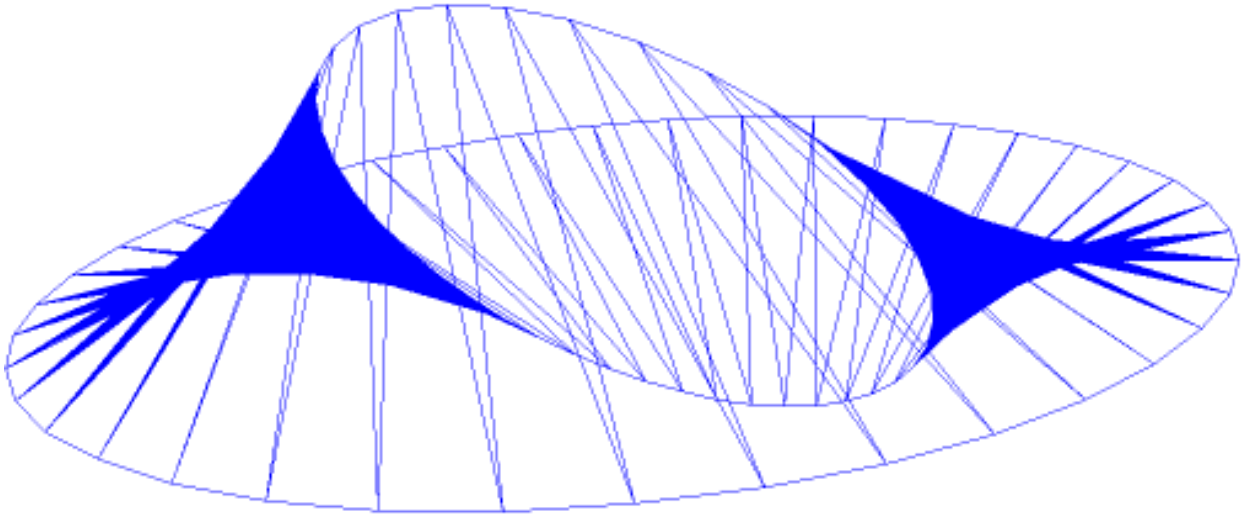
Initiation à l'API Java 3D™

Un tutorial pour les débutants

Chapitre 2
Création de géométries



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 2

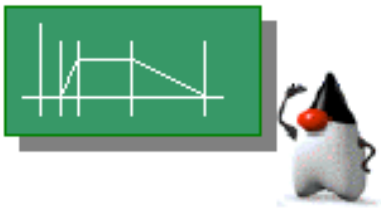
Ce document est la deuxième partie d'un tutorial sur l'utilisation de l'API Java 3D.

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 2
Création de la géométrie



Objectifs de ce chapitre

Après la lecture de ce chapitre, vous serez capable :

• D'utiliser les classes utilitaires de géométrie simple.

• D'écrire des classes pour définir des objets visuels.

• De spécifier la géométrie par l'usage des classes racines.

• De déterminer l'apparence des objets visuels.

Le Chapitre 1 explorait les concepts de base de la construction d'un univers virtuel Java 3D, se concentrant sur la détermination de transformations et de comportements simples. L'exemple HelloJava3D du chapitre un utilise la classe ColorCube comme seul objet visuel. Avec le ColorCube, le programmeur ne spécifie pas de forme ou de couleur. La classe ColorCube est commode d'utilisation mais ne peut pas être utilisée pour créer d'autres objets visuels.

Il y a trois manières différentes de créer un nouveau volume géométrique. La première utilise les classes utilitaires Box, Cone, Cylinder, et Sphere. Dans la seconde c'est le programmeur qui détermine les cordonnées des sommets pour les points, segments de ligne et/ou surfaces polygonales. La dernière étant d'utiliser un chargeur de géométrie. Ce chapitre aborde la création d'un volume géométrique par l'usage de ces deux premières techniques.

Ce chapitre se concentre sur la création de volumes géométriques, qui sont à la base des objets visuels. Quelques thèmes en rapport avec la géométrie sont également abordés comme les classes mathématiques et d'apparence. Avant de décrire la création d'un volume géométrique, une information plus complète sur le système de coordonnées de l'univers virtuel est présenté dans la Partie 2.1.

2-1 Le système de coordonnées de l'univers virtuel

Comme décrit dans le Chapitre 1, une instance de la classe VirtualUniverse sert de racine au graphe scénique dans tous les programmes Java 3D. Le terme *d'univers virtuel* se réfère communément à l'univers virtuel en trois dimensions peuplé par les objets. Chaque objet Locale dans l'univers virtuel met en place un monde virtuel ayant un système de coordonnées Cartésien.

Un objet Locale sert de point de référence pour les objets visuels de l'univers virtuel. Avec un seul Locale pour le SimpleUniverse, il n'y a qu'un seul système de coordonnées dans l'univers virtuel.

Le système de coordonnées de l'univers virtuel Java 3D est tourné à droite. L'axe des x est positif vers la droite, l'axe des y est positif vers le haut, et l'axe des z est positif vers le spectateur, avec toutes les unités en mètres. La Figure 2-1 montre l'orientation du SimpleUniverse par rapport au spectateur.

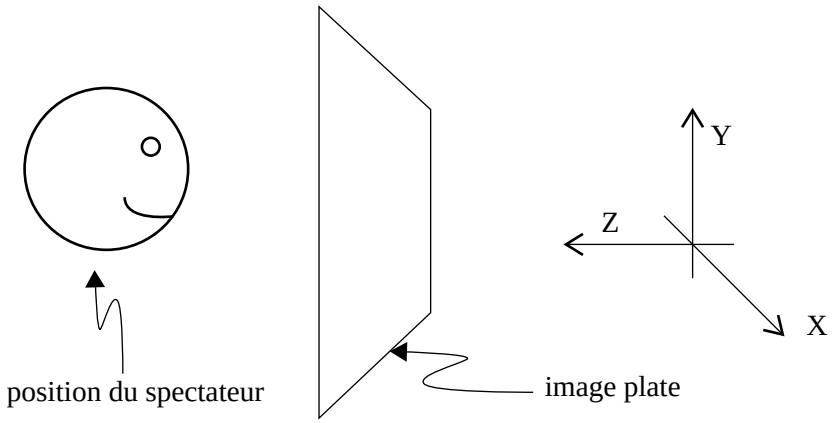


Figure 2-1 Orientation des axes dans le monde virtuel.

2.2 Les bases de la définition d'un objet visuel

La Partie 2.2.1 présente la classe Shape3D. Une approche générale de la classe NodeComponent suit dans la Partie 2.2.2. Après avoir abordé les géométries de base définies dans le package utilitaire, le reste du chapitre couvre les composants de nœud Geometry et Appearance.

2.2.1 Une instance de Shape3D définit un objet visuel

Un nœud Shape3D du graphe scénique définit un objet visuel [1]. Shape3D est une des sous-classes de la classe Leaf ; cependant, les objets Shape3D peuvent seulement exister dans le graphe scénique. L'objet Shape3D ne contient pas d'informations sur la forme et la couleur de l'objet visuel. Ces informations sont stockés dans les objets NodeComponent se référant à cet objet Shape3D. Un objet Shape3D peut se référer à un composant de nœud Geometry et un composant de nœud Appearance.

Dans le graphe scénique HelloJava3D du Chapitre 1, le symbole de base de l'objet (rectangle) est utilisé pour représenter l'objet ColorCube. Le graphe scénique simple dans la Figure 2-2 montre un objet visuel représente par un Shape3D de type Leaf (triangle) et deux NodeComponents (ovales) à place du rectangle de base [2].

[1] Les objets Shape3D définissent les objets visuels les plus communs d'un univers virtuel, mais il y a d'autres manières

[2] Ce graphe scénique n'est pas correct pour un objet ColorCube. ColorCube ne peut pas utiliser un Appearance NodeComponent. C'est l'exemple d'un objet visuel typique.

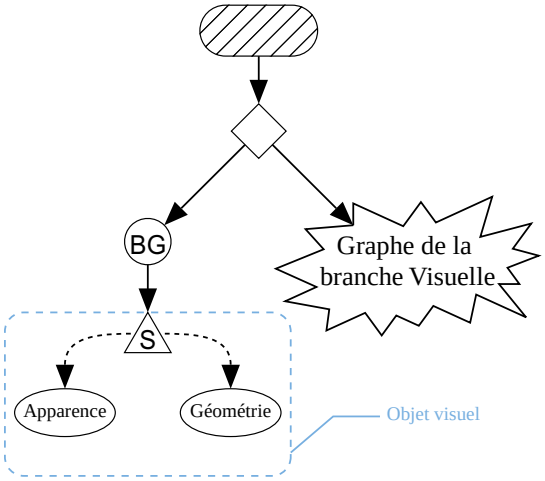


Figure 2-2 Un objet Shape3D définit un objet visuel dans le graphe scénique.

Un objet visuel peut être définit en utilisant seulement un seul Shape3D et un composant de nœud Geometry. En option, l'objet Shape3D peut très bien se référer à un composant de nœud Appearance. Le constructeur pour un Shape3D (présenté dans le Bloc de référence qui suit) montre que l'objet Shape3D peut être crée sans aucune référence à un composant de nœud, avec un seul composant de nœud Geometry, ou avec une référence à chaque type de composants de nœud.

Constructeurs du Shape3D

`Shape3D()`
Construit et initialise un objet Shape3D sans composants de nœud de géométrie et d'apparence.

`Shape3D(Geometry geometry)`
Construit et initialise un objet Shape3D avec le composant de nœud de géométrie spécifié et aucun composant d'apparence.

`Shape3D(Geometry geometry, Appearance appearance)`
Construit et initialise un objet Shape3D avec le composant de nœud de géométrie spécifié et des composants d'apparence.

Tant que l'objet Shape3D ne sera pas vivant ou compilé, le composant de nœud référencé peut etre changé grâce aux méthodes présentés dans le Bloc de référence suivant. Cette méthode peut être utilisée sur des objets Shape3D vivants ou compilés si les aptitudes nécessaires leurs ont d'abord été accordées. Le Bloc de référence qui suit liste les aptitudes du Shape3D. Lisez bien la Partie «Lecture des Blocs de référence ». Cela concerne tous les Blocs de référence suivants.

Méthodes du Shape3D (liste partielle)

Un objet Shape3D se réfère aux objets de type NodeComponent Geometry et/ou Appearance. En plus des définitions de méthode montrés ici, il y a des renvois de méthodes complémentaires.

`void setGeometry(Geometry geometry)`

`void setAppearance(Appearance appearance).`

Lecture de Blocs de référence

Les blocs de références de ce tutorial n'établissent pas une liste de tous les constructeurs, méthodes, et aptitudes pour chaque classe de l'API Java3D. Par exemple, le bloc de réference de la méthode du Shape3D (page précédente) ne liste pas toutes les méthodes de la classe Shape3D. Deux des méthodes non listés sont les méthodes de renvois qui concordent avec les méthodes de définition montrés. C'est à dire que le Shape3D possède des méthodes `getGeometry()` et `getAppearance()`.Chacune de ces méthodes renvoie une référence au NodeComponent approprié. Comme les classes de Java3D ont de nombreuses méthodes, toutes ne sont pas listées. Celles qui sont présentés dans les Blocs de références de ce tutorial sont celles qui sont pertinentes pour les sujets abordés dans celui-ci. Aussi, de nombreuses classes ont des méthodes de renvoi qui concordent avec les méthodes de définition. Les méthodes de renvoi ne sont pas listées dans les Blocs de références de ce tutorial afin de réduire la longueur de la taille des Blocs de référence.

Le Bloc de référence suivant montre les aptitudes des objets Shape3D. Ce Bloc de référence introduit un raccourci de notation pour le listing des aptitudes. Chaque ligne du Bloc de reference définit deux aptitudes à la place d'une seule. Il y a une aptitude ALLOW_GEOMETRY_READ et une aptitude ALLOW_GEOMETRY_WRITE dans chaque objet Shape3D. Bien souvent l'aptitude d'écrire et de lire vont de paire. Pour réduire la taille des Blocs de référence, les Blocs de références sur les aptitudes listent par paire les méthodes appropriées de lire et d'écrire dans une notation en raccourci.

Consultez les Spécifications de l'API pour une liste complète des constructeurs, méthodes, et aptitudes.

Aptitudes Shape3D

Les objets Shape3D héritent des aptitudes des classes SceneGraphObject, Node, et Leaf. Elles ne sont pas listées ici. Se référer à la Partie 1.8.2 pour plus d'informations sur les aptitudes.

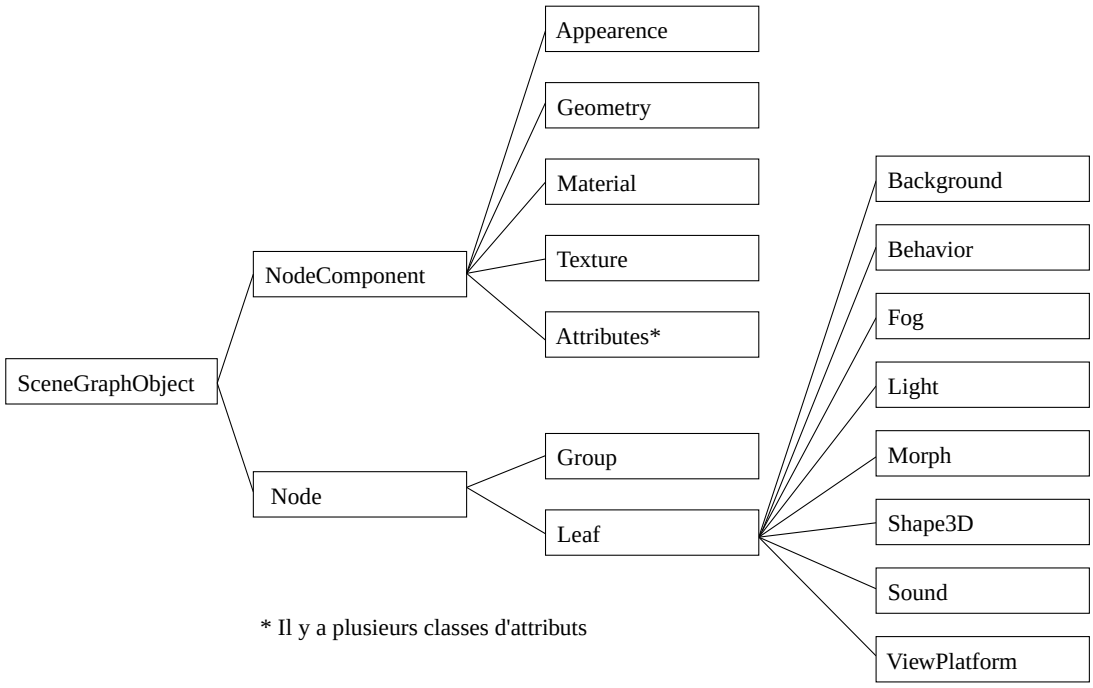
ALLOW_GEOMETRY_READ | WRITE

ALLOW_APPEARANCE_READ | WRITE

ALLOW_COLLISION_BOUNDS_READ | WRITE

2.2.2 Node Components (composants de nœud)

Les objets NodeComponent contiennent la spécification exacte des attributs d'un objet visuel. Chacune des nombreuses sous-classes de NodeComponent définissent certains attributs visuels. La Figure 2-3 montre une partie de la hiérarchie de l'API Java 3D contenant la classe NodeComponent et ces descendants. La Partie 2.5 présente le composant de nœud Geometry. La Partie 2.6 présente le composant de nœud



Appearance.

Figure 2-3 Hiérarchie partielle des classes de l'API Java 3D présentant les sous-classes de NodeComponent..

2.2.3 Définition des classes d'objets visuels

Le même objet visuel pourra souvent apparaître plusieurs fois dans un univers simple. Il semble logique de définir une classe pour y créer les objets visuels à l'intérieur que de reconstruire les objets visuel à chaque fois. Il y a plusieurs manières de concevoir une classe définissant un objet visuel.

Le Fragment de code 2-1 montre le squelette du code d'une classe VisuelObject, comme un exemple d'organisation générique possible pour une classe d'objet visuel. Les méthodes sont vides dans ce code. Le code du VisualObject ne peut pas apparaître dans les exemples fournis parce qu'il n'est pas particulièrement utile sous cette forme.

```
1.    public class VisualObject extends Shape3D{
2.
3.          private Geometry voGeometry;
4.          private Appearance voAppearance;
5.
6.          // crée un Shape3D avec une géométrie et une apparence
7.          // la géométrie createGeometry
8.          // l'apparence est crée dans la mé@thode createAppearance
9.          public VisualObject() {
10.
11.               voGeometry = createGeometry();
12.               voAppearance = createAppearance();
```

```
13.               this.setGeometry(voGeometry);
14.               this.setAppearance(voAppearance);
15.               }
16.
17.               private Geometry createGeometry() {
18.               // code pour créer une géométrie par défaut de l'objet visuel
19.               }
20.
21.               private Appearance createAppearance () {
22.               // code pour créer une apparence par défaut de l'objet visuel
23.               }
24.
25.         } // fin de la classe VisualObject
```

Fragment de code 2-1 Squelette de code pour une classe VisualObject.

L'organisation de la classe VisualObject dans le Fragment de code 2-1 est similaire à la classe utilitaire ColorCube par le fait qu'elle étend l'objet Shape3D. La classe VisualObject est une proposition de point départ pour définir des classes de volume sur mesure pour une utilisation dans la construction du graphe scénique. Chaque programmeur Java 3D, individuellement, pourra personnaliser la classe VisualObject pour son propre usage. Pour un exemple plus complet de l'organisation de cette classe, lisez le code source de la classe ColorCube dans le package `com.sun.j3d.utils.geometry`, lequel est fournit avec l'API Java 3D.

Prendre le Shape3D comme une base pour créer des classes d'objets visuels les rend facile d'utilisation dans un programme Java 3D. La classe VisualObject peut etre utilisée aussi facilement que la classe ColorCube dans les exemples HelloJava3D du Chapitre 1. Le constructeur peut etre appelé et le nouvel objet crée inséré comme un enfant d'un quelconque Group par une seule ligne de code. Dans les exemples suivants de ligne de code, objRoot est une instance de Group. Ce code crée un VisualObject et l'ajoute comme un enfant de l'objRoot au graphe scénique :

```
objRoot.addChild(new VisualObject());
```

Le constructeur du VisualObject fabrique le VisualObject par la création d'un objet Shape3D qui référence les composants de nœud crées par les méthodes `createGeometry()`et `createAppearance()`. La méthode `createGeometry()` crée un composant de nœud de Geometry qui sera utilisé par l'objet visuel. La méthode `createAppearance()` est responsable de la création du composant de nœud qui définira l'Appearance de l'objet visuel.

Une autre organisation possible pour un objet visuel est de définir une classe container non dérivée des classes de l'API Java 3D. Dans cette conception, la classe objet visuel devra prendre comme racine pour la définition de son sous-graphe un Node (nœud), un Group ou un Shape3D. La classe devra définir une ou des méthodes pour renvoyer une référence à cette racine. Cette technique représente un peu plus de travail, mais peut etre plus simple à comprendre. Une partie des programmes présentés plus tard dans ce chapitre donnent des exemples de définitions de classe d'objet visuel indépendant.

La troisieme organisation possible d'une classe d'objet visuel est celle similaire aux classes Box, Cone, Cylinder, et Sphere définies dans le package `com.sun.j3d.utils.geometry`. Chacune étendant la Primitive, laquelle étend le Group. Les détails de conception de la Primitive et de ses descendants ne sont pas abordés par ce tutorial, mais le code source pour toutes ces classes est fournit avec l'API Java 3D. À partir de la source de la classe Primitive, ainsi que celles des autres classes utilitaires, le lecteur pourra en apprendre un peu plus sur le design de cette classe.

2.3 Classes utilitaires de géométrie

Cette partie couvre les classes utilitaires pour la création de primitives géométriques boite, cône, cylindre, et sphère. Les primitives géométriques fournisssent une seconde solution pratique pour créer du volume dans un univers virtuel. La plus simple étant d'utiliser la classe ColorCube.

Les classes de primitives fournissent au programmeur plus de flexibilité que la classe ColorCube. Un objet ColorCube définit la géométrie et la couleur dans un seul composant de nœud Geometry. Par conséquent, tout le ColorCube est figé, excepté sa taille [3]. La taille d'un ColorCube est seulement spécifiée quand cet objet est crée.

Un objet de type primitive procure plus de flexibilité en déterminant la forme sans définir de couleur. Dans une classe utilitaire de primitive géométrique, le programmeur ne peut pas modifier cette géométrie, mais peut en modifier l'apparence [4].Les classes de primitive donnent au programmeur la flexibilité d'avoir de nombreuses instances de la même primitive de géométrique, où chacune peu avoir une apparence différente par un référencement à différents composants de nœuds Appearance.

Les classes utilitaires Box, Cone, Cylinder et Sphere sont définies dans le package `com.sun.j3d.utils.geometry`. Les détails des classes Box, Cone, Cylinder, et Sphere sont présentés dans les Parties 2.3.1 à 2.3.4, respectivement. La super-classe de ces primitives, Primitive, est abordée à la section 2.3.5. La partie de la hiérarchie du package `com.sun.j3d.utils.geometry` qui contient les classes de primitive est montrée dans la Figure 2-4.

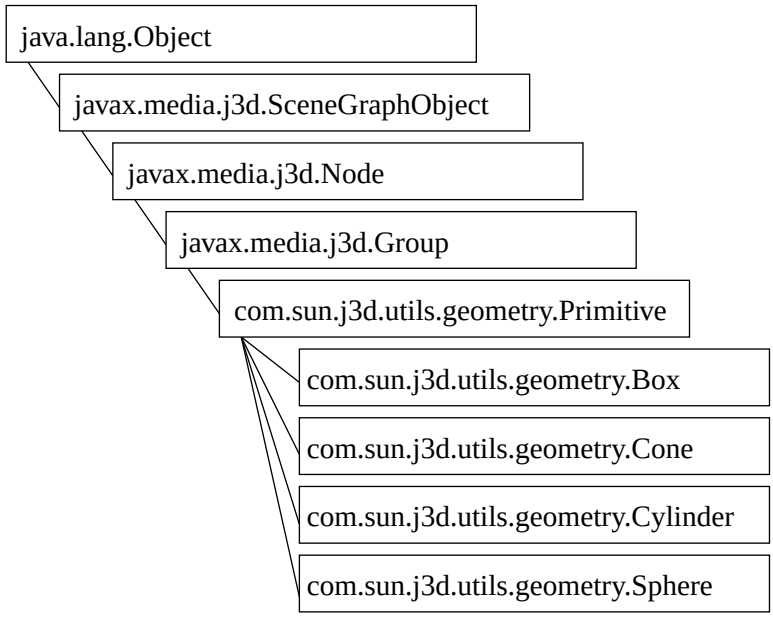


Figure 2-4 Hiérarchie de la classe d'utilitaires des primitives géométriques : Box, Cone, Cylinder, et Sphere.

[3] Le composant de nœud Géométrique référencé par un ColorCube peut être changé, mais alors il n'apparaîtra pas comme un ColorCube.

[4] Comme avec le ColorCube, le composant de nœud Geometry référencé par un objet primitive peut être modifié, mais alors il n'apparaîtra pas comme une primitive.

2.3.1 Box

La géométrie Box crée des objets visuel de la forme d'une boite 3D [5]. Les longueurs, largeurs et hauteurs par défaut sont de 2 mètres, avec le centre à l'origine, ce qui donne un cube avec les coins à (-1, -1, -1) et (1, 1, 1). La longueur, largeur, et hauteur peut etre spécifiée au moment de la création de l'objet. Bien sûr, le TransformGroup le long du chemin du graphe scénique peut etre utilisé pour changer l'emplacement et/ou l'orientation des instances de Box et des autres objets visuels.

Constructeurs de la Box (liste partielle)

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

La Box étend une Primitive, une autre classe du package com.sun.j3d.utils.geometry.

`Box()`
Construit une boite par défaut de 2.0 mètres de hauteur, largeur et hauteur, centré à l'origine.

`Box(float xdim, float ydim, float zdim, Appearance appearance)`
Construit une boite de dimensions et d'apparence donnés, centrée à l'origine.

Tandis que le constructeur diffère suivant les classes Box, Cone et Cylinder, elles utilisent les mêmes méthodes. Le Bloc de référence suivant liste les méthodes pour ces classes.

Méthodes de Box, Cone, and Cylinder

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

Ces méthodes sont définies pour chaque classe de type Primitive : Box, Cone, et Cylinder. Ces primitives sont composées de multiples objets Shape3D dans un groupe.

`Shape3D getShape(int id)`
Prend une des faces (Shape3D) de la primitive qui contient la géométrie et l'apparence. Les objets Box, Cone, et Cylinder sont composés de plus d'un objet Shape3D, chacun avec leur propres composants de nœuds Geometry. La valeur utilisée pour la fonction id spécifie quel composant de nœud Geometry prendre.

`void setAppearance(Appearance appearance)`
Modifie l'apparence de la primitive (pour tous les objets Shape3D).

2.3.2 Cone

La classe Cone définit un objet de la forme d'un cône centré à l'origine avec l'axe central aligné le long de l'axe y. Le rayon par défaut est de 1.0 et la hauteur de 2.0. Le centre du cône est défini comme le centre de son cube d'application [bounding box] plutôt que son point central.

[5] Techniquement, une boite est un polyèdre à six coté avec des faces rectangulaires..

Constructeurs du Cone (liste partielle)

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

Cone étends Primitive, une autre classe du package com.sun.j3d.utils.geometry.

`Cone()`
Construit un Cone par défaut de rayon 1.0 et de hauteur 2.0.

`Cone(float radius, float height)`
Construit un Cone par défaut de rayon et de hauteur donné.

2.3.3 Cylinder

La classe Cylinder crée un objet cylindre, dont les extrémités sont closes, centré à l'origine avec l'axe central aligné sur l'axe des y. La valeur par défaut pour le rayon étant de 1.0 et la hauteur de 2.0.

Constructeurs du Cylinder (liste partielle)

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

Cone étends Primitive, une autre classe du package com.sun.j3d.utils.geometry.

`Cylinder()`
Construit un Cylinder par défaut de rayon 1.0 et de hauteur 2.0.

`Cylinder(float radius, float height)`
Construit un cylindre par défaut de rayon et de hauteur donné.

`Cylinder(float radius, float height, Appearance appearance)`
Construit un cylindre par défaut de rayon de hauteur et d'apparence donné.

2.3.4 Sphere

La classe Sphere crée un objet sphérique centré à l'origine. Le rayon par défaut est de 1.0.

Constructeurs de Sphere (liste partielle)

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

Sphere étends Primitive, une autre classe du package com.sun.j3d.utils.geometry.

`Sphere()`
Construit une Sphere par défaut de rayon 1.0.

`Sphere(float radius)`
Construit une Sphere par défaut de rayon donné.

`Sphere(float radius, Appearance appearance)`
Construit une Sphere de rayon et d'apparence donné.

Méthodes de Sphere

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

Étant une extension d'une Primitive, une Sphere est un objet Group qui possède un seul objet Shape3D comme enfant.

`Shape3D getShape()`
Accède au Shape3D qui contient la géométrie et l'apparence.

`Shape3D getShape(int id)`
Cette méthode est incluse pour une compatibilité avec les autres classes Primitive : Box, Cone, et Cylinder. Toutefois, comme une Sphere possède un seul objet Shape3D, il ne peut etre appelé qu'avec id = 1.

`void setAppearance(Appearance appearance)`
Modifie l'apparence la sphère.

2.3.5 Plus de détails sur les primitives géométriques.

La géométrie d'une primitive de la classe utilitaires ne définit pas de couleur. Geometry ne définissant pas de couleur, la couleur provient de son composant de nœud Appearance. Sans aucune référence à un composant de nœud Appearance, l'objet visuel sera blanc, couleur d'apparence par défaut. La couleur d'abord abordée dans la Partie 2.4.2 et ajouté à la géométrie à la Partie 2.5.1. La Partie 2.6 présentant les détails du composant de nœud Appearance.

La classe Primitive définit les valeurs habituelles par défaut pour les Box, Cylinder, Cone et Sphere. Par exemple, la Primitive définit la valeur par défaut pour le nombre de polygones utilisés pour représenter les surfaces.

La Partie 2.3.8 présente quelques détails de la classes Primitive. Étant donné que les valeurs par défaut définies par la Primitive son adéquates pour la plupart des applications, les programmes Java 3D peuvent etre écrits sans utiliser la classe Primitive. Pour cette raison, la partie décrivant la classe Primitive est considérée comme un sujet avancé (que l'on peut passer). Vous reconnaîtrez les parties avancés, lorsque vous y arriverez, par la figure du Duke suspendu à la double ligne.

2.3.6 ColorCube

La classe ColorCube est présentée ici par contraste avec les classes primitives de géometries Box, Cone, Cylinder, et Sphere. La classe ColorCube étends une hiérarchie différente de celle des classes primitives graphiques. C'est une sous-classe de Shape3D. Cette hiérarchie du ColorCube est présentée dans la Figure 2-5. Le chapitre un contient le Bloc de référence pour le ColorCube.

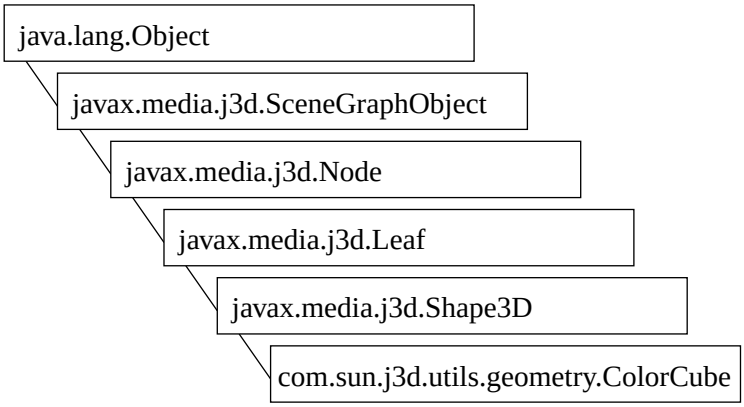


Figure 2-5 Hiérarchie de la classe ColorCube, classe utilitaire de géométrie.

ColorCube est la seule classe distribué avec l'API Java 3D qui permet au programmeur d'ignorer le problème des couleurs et des lumières. Pour cette raison, la classe ColorCube est utile pour assembler rapidement des graphes scéniques pour des tests et du prototypage.

2.3.7 Exemple : création d'un simple Yo-Yo composé de deux Cones

Cette partie décrit un exemple simple qui utilise la classe Cone : `ConeYoyoApp.java`. Le but de ce programme est de rendre un Yo-Yo. Deux cônes sont utilisés pour former le yo-yo. Les comportements [behaviors] de l'API Java 3D peuvent être utilisés pour faire bouger le yo-yo vers le haut et le bas, mais c'est hors du sujet de ce Chapitre. Le programme fait tourner le yo-yo afin d'en apprécier la géométrie. Le schéma du graphe scénique dans la Figure 2-5 montre les dessins des classes ConeYoyo et ConeYoyoAp du programme d'exemple ConeYoyoApp. La position par défaut d'un objet Cone est avec son cube d'application [bounding box] centré à l'origine. L'orientation par défaut l'objet Cone part de la pointe dans la direction positive des axes y. Le yo-yo est formé par de deux cônes qui sont tournés autour des axes z et translatés le long de l'axe x pour placer les sommets des cônes ensemble à l'origine. D'autres combinaisons de rotations et de translations pourraient placer les pointes des objets Cone au même emplacement.

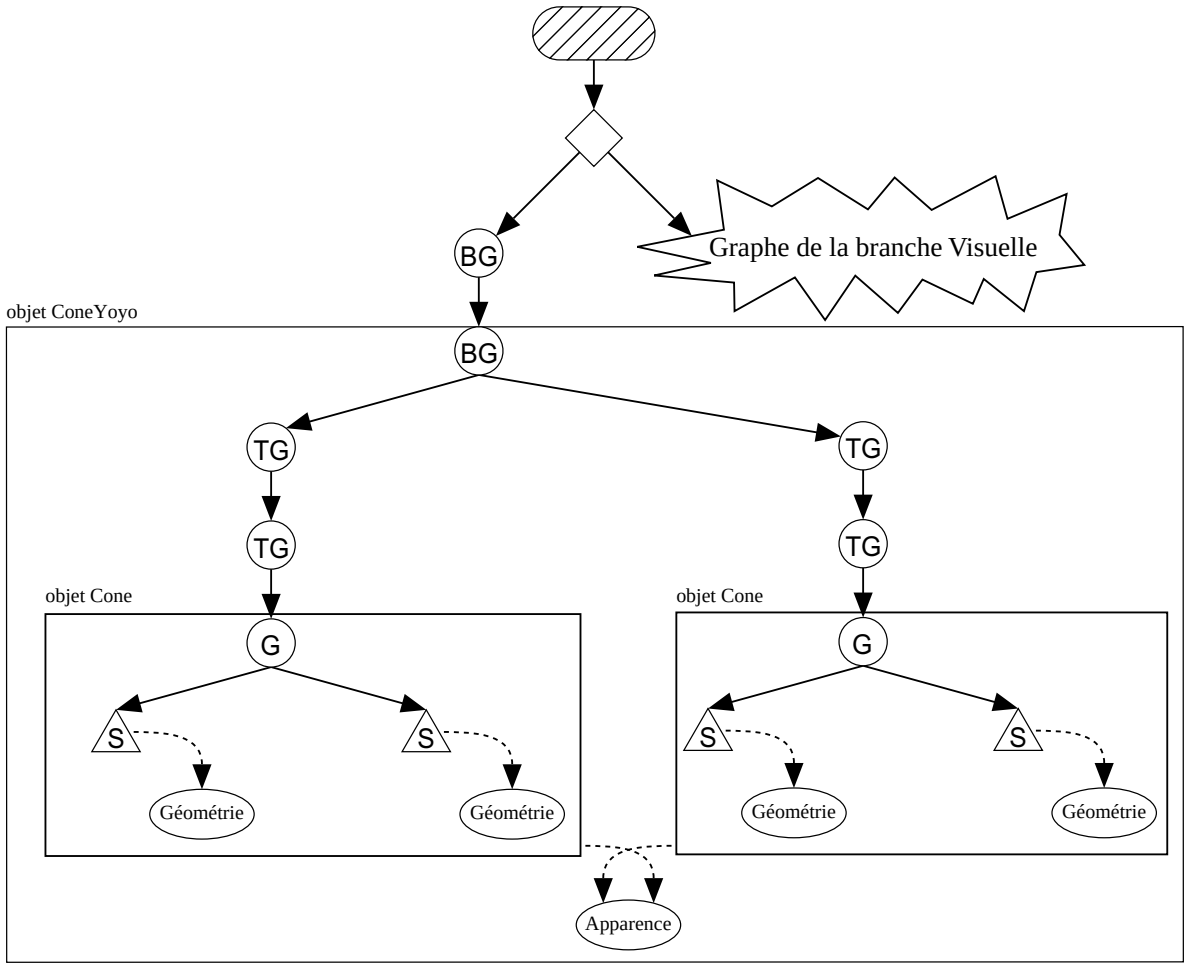


Figure 2-6 Graphe scénique pour ConeYoyoApp [6].

Dans la branche du graphe scénique qui commence avec l'objet BranchGroup crée par l'objet ConeYoyo, le chemin du graphe vers chaque objet Cone débute par un objet TransformGroup spécifiant la translation, suivit du TransformGroup spécifiant la rotation, aboutissant à l'objet Cone.

Plusieurs graphes scènique pourraient représenter le même monde virtuel. En prennant la Figure 2-6 comme exemple, quelques modifications simples peuvent etre éffectués. Une des modifications éliminant l'objet BranchGroup dont l'enfant est l'objet ConeYoyo et inserant l'objet ConeYoyo directement à la Locale. Le BranchGroup étant là pour ajouter de futurs objets visuels au monde virtuel. Une autre modification associe les deux objets TransformGroup à l'intérieur de l'objet ConeYoyo. Ces transformations sont montées simplement comme des exemples.

[6] Actuellement, la primitive Cone est partagée automatiquement comme une caractéristique de la classe Primitive. Cette caractéristique est abordée dans la Partie 2.3.8.

Les nœuds Shape3D des objets Cone font référence aux composants de nœuds Geometry. Ils sont à l'intérieur des objets Cones. Les objets Shape3D du Cone sont les enfants d'un Group dans le Cone. Étant donné que les objets Cones descendent d'un Group, le même Cone (ou d'autres objets Primitive) ne peut pas etre utilisé plus d'une fois dans un graphe scénique. La Figure 2-7 montre un exemple de message d'erreur produit lors de la tentative d'utilisation du même objet Cone dans un seul graphe scénique. Cette erreur n'existe pas dans le programme d'exemple distribué avec le tutorial.

```
Exception in thread «main» javax.media.j3d.MultipleParentException:
Group.addChild: child already has a parent
      at javax.media.j3d.GroupRetained.addChild(GroupRetained.java:246)
      at javax.media.j3d.Group.addChild(Group.java:241)
      at ConeYoyoApp$ConeYoyo.<init>(ConeYoyoApp.java:89)
      at ConeYoyoApp.createSceneGraph(ConeYoyoApp.java:119)
      at ConeYoyoApp.<init>(ConeYoyoApp.java:159)
      at ConeYoyoApp.main(ConeYoyoApp.java:172)
```

Figure 2-7 Exception de parents multiples lors de la tentative de réutilisation d'un objet Cone.

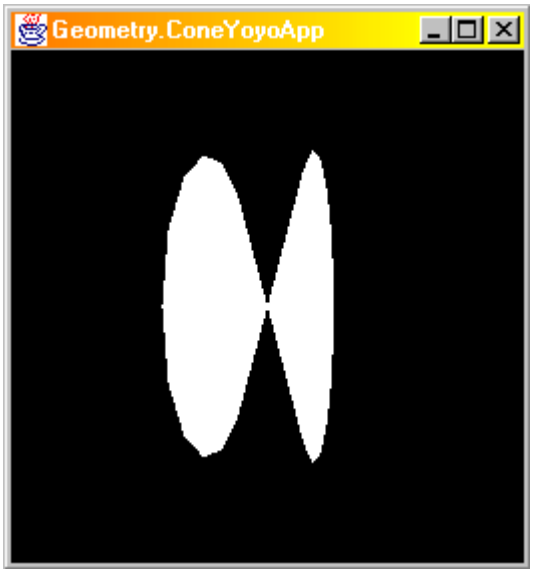


Figure 2-8 Une image rendue par ConeYoyoApp.java.

La Figure 2-8 montre une des images possibles rendue par `ConeYoyoApp.java` pendant que l'objet ConeYoyo tourne. `ConeYoyoApp.java` ce trouve dans le répertoire `example/Geometry`. La classe ConeYoyo du programme est reproduite à la page suivante dans le Fragment de code 2-2.

Les lignes 14 à 21 créent les objets de la moitié du graphe scénique du yo-yo. Les lignes 23 à 25 créent les relations entre ces objets. Ce procédé est répété pour l'autre moitié du yo-yo aux lignes 27 à 38.

La ligne 12 crée **yoyoAppear**, un composant de nœud Appearance avec des valeurs par défauts, afin d'être utilisé avec les objets Cone. Les lignes 21 et 31 modifient l'apparence pour les deux cônes.

```
1.     public class ConeYoyo{
2.
3.          private BranchGroup yoyoBG;
4.
5.          // crée un Shape3D avec une géométrie et une apparence
6.          //
7.          public ConeYoyo() {
8.
9.          yoyoBG = new BranchGroup();
10.         Transform3D rotate = new Transform3D();
11.         Transform3D translate = new Transform3D();
12.         Appearance yoyoAppear = new Appearance();
13.
14.         rotate.rotZ(Math.PI/2.0d);
15.         TransformGroup yoyoTGR1 = new TransformGroup(rotate);
16.
17.         translate.set(new Vector3f(0.1f, 0.0f, 0.0f));
18.         TransformGroup yoyoTGT1 = new TransformGroup(translate);
19.
20.         Cone cone1 = new Cone(0.6f, 0.2f);
21.         cone1.setAppearance(yoyoAppear);
22.
23.         yoyoBG.addChild(yoyoTGT1);
24.         yoyoTGT1.addChild(yoyoTGR1);
25.         yoyoTGR1.addChild(cone1);
26.
27.         translate.set(new Vector3f(-0.1f, 0.0f, 0.0f));
28.         TransformGroup yoyoTGT2 = new TransformGroup(translate);
29.
30.         rotate.rotZ(-Math.PI/2.0d);
31.         TransformGroup yoyoTGR2 = new TransformGroup(rotate);
32.
33.         Cone cone2 = new Cone(0.6f, 0.2f);
34.         cone2.setAppearance(yoyoAppear);
35.
36.         yoyoBG.addChild(yoyoTGT2);
37.         yoyoTGT2.addChild(yoyoTGR2);
38.         yoyoTGR2.addChild(cone2);
39.
40.         yoyoBG.compile();
41.
42.   } // fin du constructeur de ConeYoyo
43.
44.   public BranchGroup getBG(){
45.   return yoyoBG;
46.   }
47.
48. } // fin de la classe ConeYoyo
```

Fragment de code 2-2 Classe ConeYoyo du programme d'exemple ConeYoyoApp.java.

2.3.8 Partie avancée: Primitive géometriques

La hiérarchie de la classe de la Figure 2-4 montre la Primitive comme une super-classe des classes of Box, Cone, Cylinder, Sphere. Elle définit un nombre de champs [fields] et de méthodes communes pour ces classes, en plus des valeurs par défaut pour les champs.

La classe Primitive fournit une voie pour distribuer les composants de nœud Geometry entre les instances de primitive de même taille. Par défaut, toutes les primitives de même taille partagent un composant de nœud de Géométrie

Un exemple de champ définit dans la classe Primitive est le nombre entier [integer] GEOMETRY_NOT_SHARED. Ce champ détermine la géometrie devant etre crée qui ne sera pas partagée avec les autres. Établir ce flag (drapeau) permet d'éviter le partage de la géometrie entre les primitives de même paramètre (e.g., des sphères avec un rayon de 1).

```
myCone.setPrimitiveFlags(Primitive.GEOMETRY_NOT_SHARED)
```

Méthodes de Primitive (liste partielle)

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

La Primitive etends le Group et est également une super-classe pour Box, Cone, Cylinder, et Sphere.

`public void setNumVertices(int num)`
Modifie le nombre total de sommets [vertices] dans cette primitive.

`void setPrimitiveFlags(int fl)`
Les flags de la primitve sont :

GEOMETRY_NOT_SHARED	Les normales sont produites avec ces positions.
GENERATE_NORMALS_INWARD	Les normales de la surface sont retournées.
GENERATE_TEXTURE_COORDS	Produit des coordonnées de Texture.
GEOMETRY_NOT_SHARED	La géométrie crée ne sera pas partagée avec un autre nœud.

`void setAppearance(int partid, Appearance appearance)`
Modifie l'apparence d'une sous-partie donnée partid. Les objets Box, Cone, et Cylinder sont composé de plusieurs objets Shape3D, chacun ayant potentiellement son propre composant de nœud Appearance. La valeur utilisée pour partid détermine quel composant de nœud Appearance modifier.

`void setAppearance()`
Modifie l'apparence principale de la primitive (et toutes ses sous-parties) en une apparence par défaut de couleur blanche.

Les constructeurs additionnels pour les Box, Cone, Cylinder, et Sphere autorise la spécification de flags de Primitive à la création de l'objet. Consuter les Descriptions [spécification] de l'API Java 3D pour plus d'information.

2.4 Classes Mathématiques

Pour créer des objets visuels, la classe Geometry et ses sous-classes sont nécessaires. De nombreuses sous-classes de Geometry décrivent des primitives de base-sommet [vertex-based], tel que des points, lignes et polygones pleins. Les sous-classes de Geometry sont abordés dans la Partie 2.5, mais avant d'en débattre , plusieurs classes mathématique (Point*, Color*, Vector*, TexCoord*) employés pour déterminer les données associées-au-sommet doivent etre abordées [7].

Notez que l'astérisque utilisé ci-dessus est un raccourci pour représenter les variations des noms de classes. Par exemple, Tuple* se réfère à toutes les classes de Tuple : Tuple2f, Tuple2d, Tuple3b, Tuple3f, Tuple3d, Tuple4b, Tuple4f, et Tuple4d.

Dans tous les cas le nombre indique le nombre d'éléments dans le tuple, et la lettre indique le type de donnée des éléments. 'f' indique un nombre simple à virgule flottante, 'd' indique un nombre double a virgule flottante, et 'b' pour des bytes. Donc, Tuple3f est une classe qui manipule trois valeurs de nombre simple à virgule flottante.

Toutes ces classe mathématiques se trouvent dans la package `javax.vecmath.*`. Celui-ci définit de nombreuses classes Tuple* comme des super-classes abstraites génériques. D'autres classes plus utiles sont derivés de diverses classes Tuple. La hiérarchie de certains de ces packages est montré dans la Figure 2-9.

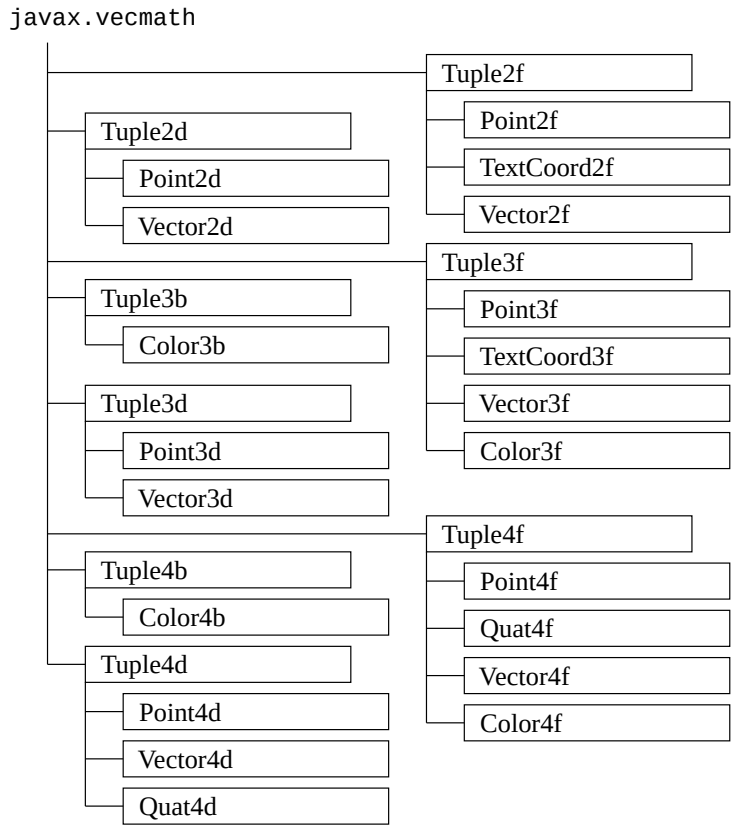


Figure 2-9 Package des classes mathématiques et sa hiérarchie.

[7] Les classes TexCoord* ne sont pas utilisées dans la version 1.1 de l'API Java 3D. Ceci sera modifié dans des versions ultérieures.

Chaque sommet d'un objet visuel doit preciser jusqu'à quatre objets `javax.vecmath`, représentant des coordonnées, des couleurs, des normales de surface, et des coordonnées de texture. Les classes suivantes sont les plus utilisées :

- Point* (pour des coordonnées)
- Color* (pour des couleurs)
- Vector* (pour des normales de surface)
- TexCoord* (pour des coordonnées de texture)

Notez que les coordonnées (objets Point*) sont nécessaires pour positionner chaque vertex. Les autres données sont optionnelles, dépendant de la façon dont la primitive est rendue. Par exemple, une couleur (un objet Color*) doit être définie pour chaque sommet et les couleurs de la primitive sont interpollées entre les couleurs des sommets. Si l'eclairage est employé, les normales de la surface (et donc les objets Vector*) sont nécessaires. Si le mapping de texture est employé, alors les coordonnées de texture sont nécessaires.

(Les objets Quat* représentent des quaternions, lequels sont seulement utilisés pour des transformations de matrice 3D avancées.)

Étant donné que toutes les classes utilitaires héritent des classes résumés par Tuple*, il est important d'être familiarisé avec les constructeurs et les méthodes de Tuple, lequels sont listés ci-dessous.

Constructeurs de Tuple2f

Package: `javax.vecmath`

Les classes de Tuple* ne sont pas directement utilisées dans les programmes Java 3D mais fournissent la base des classes Point*, Color*, Vector*, et TexCoord*. Les constructeurs décrits ici sont valables pour ces sous-classes. Tuple3f et Tuple4f ont des jeux similaires de constructeurs.

`Tuple2f()`
Construit et initialise un objet Tuple avec les coordonnes (0,0).

`Tuple2f(float x, float y)`
Construit et initialise un objet Tuple depuis les coordonnées indiquées x et y.

`Tuple2f(float[] t)`
Construit et initialise un objet Tuple depuis le tableau specifié.

`Tuple2f(Tuple2f t)`
Construit et initialise un objet Tuple depuis la donnée d'un autre objet Tuple.

`Tuple2f(Tuple2d t)`
Construit et initialise un objet Tuple depuis la donnée d'un autre objet Tuple.

Méthodes de Tuple2f (liste partiellle)

Package: `javax.vecmath`

Les classes de Tuple* ne sont pas directement utilisées dans les programmes Java 3D mais fournissent la base des classes Point*, Color*, Vector*, et TexCoord*. En particulier, Tuple2f fournit une base pour les classes Point2f, Color2f, et TexCoord2f. Les méthodes décrites ici sont valables pour ces sous-classes. Tuple3f et Tuple4f ont des jeux similaires de méthodes.

void set(float x, float y)

void set(float[] t)
Modifie la valeur de ce tuple en la valeur spécifié.

boolean equals(Tuple2f t1)
Retourne true si la donnée dans le Tuple t1 est egale à la donnée correspondante dans ce tuple.

final void add(Tuple2f t1)
Modifie la valeur de ce tuple en un vecteur somme de lui-même et de tuple t1.

void add(Tuple2f t1, Tuple2f t2)
Modifie la valeur de ce tuple en un vecteur somme de tuple t1 et t2.

void sub(Tuple2f t1, Tuple2f t2)
Modifie la valeur de ce tuple en un vecteur difference de tuple t1 et t2 (this = t1 - t2).

void sub(Tuple2f t1)
Modifie la valeur de ce tuple en un vecteur difference de lui-même et de tuple t1 (this = this - t1).

void negate()
Rend négatif la valeur de ce vecteur à cet endroit.

void negate(Tuple2f t1)
Modifie la valeur de ce tuple en le négatif de tuple1.

void absolute()
Modifie chaque composant de ce tuple en sa valeur absolue.

void absolute(Tuple2f t)
Modifie chaque composant de ce tuple en sa propre valeur absolue, et place cette valeur modifiée dans ce tuple.

Il y a de subtiles, mais prévisibles, différences entre les constructeurs et les méthodes de Tuple*, causées par le nombre et le type de données. Par exemple, Tuple3d diffère de Tuple2f, parce qu'elle a une méthode constructeur :

```
Tuple3d(double x, double y, double z);
```

Laquelle s'attend à trois, et non pas deux, parametres de double-précision à virgule flottante, et non pas simple-précision.

Chacune des classes Tuple* possede des membres public. Pour Tuple2*, ce sont x et y. Pour Tuple3* les membres sont x, y et z. Pour Tuple4* les membres sont x, y, z, et w.

2.4.1 Les classes Point

Les objets Point* représentent d'ordinaire les coordonnées d'un sommet, quoiqu'ils peuvent aussi représenter la position d'une image tramée représentée par des pixels, un point de source lumineuse, la position spatiale d'un son, ou tout autre donnée de positionnement. Les constructeurs pour les classes Point* sont similaires aux constructeurs de Tuple*, à la différence qu'ils retournent des objets Point*. (Quelques constructeurs sont des paramètres transmits comme les objets Point*, à la place d'objets Tuple*).

Methodes de Point3f (liste partielle)

Package: `javax.vecmath`

Les classes Point* sont dérivés des classes Tuple*. Chaque instance des classes Points* représente un simple point dans deux-, trois-, ou quatre-espace. En plus des méthodes Tuple*, les classes Points* ont des méthodes additionnelles, quelques-unes sont listées ci-dessous.

`float distance(Point3f p1)`
Renvoie la distance Euclidienne entre ce point et le point p1.

`float distanceSquared(Point3f p1)`
Renvoie le carré de la distance Euclidienne entre ce point et le point p1.

`float distanceL1(Point3f p1)`
Renvoie la distance L1 (Manhattan) entre ce point et le point p1. La distance L1 est égale à :

$$abs(x_1 - x_2) + abs(y_1 - y_2) + abs(z_1 - z_2)$$

Une fois encore, il y a ici une subtilité, des différences sont prévisibles entre les constructeurs et les méthodes Point*, dues au nombre et au type de données. Par exemple, pour Point3d, la méthode distance renvoie une valeur à double-précision du point.

2.4.2 Les classes Color

Les objets Color* représentent une couleur, laquelle peut être celle d'un sommet, une proprietée de matériau, du brouillard, ou tout autre objet visuel. Les couleurs sont déterminées soit par Color3* ou Color4*, et seulement pour des types de données de points sous forme de byte ou de flottante à double précision. Les objets Color3* déterminent une couleur par la combinaison de valeurs de rouge, vert et bleu (RGB). Les objets Color4* déterminent une valeur de transparence en addition au RGB. (Par défaut, les objets Color3* sont opaques.) Pour des données de types byte, les valeurs de la couleurs se rangent entre 0 et 255, inclus. Pour des données de flottante à simple précision, les valeurs de la couleur se rangent entre 0.0 et 1.0, inclus.

Une fois encore les constructeurs, pour les classes Color* sont similaires aux constructeurs de Tuple*, à l'exception qu'ils retournent des objets Color*. (Certains constructeurs sont des paramètres passés comme le sont les objets Color*.) Les classes Color* n'ont pas de méthodes additionnelles, alors elles comptent sur les méthodes dont elles héritent de leur super-classe Tuple*.

Il est parfois plus facile de créer des constantes pour les couleurs qui seront utilisées de manières répétitives dans la création d'objets visuel. Par exemple,

```
Color3f red = new Color3f(1.0f, 0.0f, 0.0f);
```

instancie l'objet Color3f **red** qui pourra etre utilisée plusieurs fois. Il peut être utile de créer une classe qui contient un nombre de couleurs constantes. Un exemple de cette classe est présenté dans le Fragment de code 2-1.

```
1. import javax.vecmath.*;
2.
3. class ColorConstants{
4.    public static final Color3f red = new Color3f(1.0f,0.0f,0.0f);
5.    public static final Color3f green = new Color3f(0.0f,1.0f,0.0f);
6.    public static final Color3f blue = new Color3f(0.0f,0.0f,1.0f);
7.    public static final Color3f yellow = new Color3f(1.0f,1.0f,0.0f);
8.    public static final Color3f cyan = new Color3f(0.0f,1.0f,1.0f);
9.    public static final Color3f magenta = new Color3f(1.0f,0.0f,1.0f);
10.   public static final Color3f white = new Color3f(1.0f,1.0f,1.0f);
11.   public static final Color3f black = new Color3f(0.0f,0.0f,0.0f);
12.}
```

Fragment de code 2-3 Exemple de classe ColorConstants.

Classes Color*

Package: `javax.vecmath`

Les classes Color* sont dérivées des classes Tuple*. Chacune des instances des classes Color* représente une couleur simple de trois composants (RGB), ou de quatre composants (RGBA). Les classes Color* n'ajoutent pas de méthodes a celles fournies par les classes Tuple*.

2.4.3 Les classes Vector

Les objets Vector* représentent généralement les normales de surface des sommets, bien qu'ils peuvent aussi représenter la direction d'une source de lumière ou d'une source sonore. De nouveau, les constructeurs des classes Vector* sont semblables aux constructeurs de Tuple*. Pourtant, les objets Vector* ajoutent de nombreuses méthodes que l'on ne trouve pas dans les classes Tuple*.

Méthodes Vector3f (liste partielle)

Package: `javax.vecmath`

Les classes Vector* sont dérivées des classes Tuple*. Chacune des instances des classes Vector* représentent un vecteur simple en deux-, trois-, ou quatre-espaces. En plus des méthodes de Tuple, les classes Vector* ont des méthodes additionnelles, qui sont listées ici.

`float length()`
Renvoie la longueur de ce vecteur.

`float lengthSquared()`
Renvoie la longueur au carré de ce vecteur.

`void cross(Vector3f v1, Vector3f v2)`
Modifie ce vecteur pour être le produit croisé des vecteurs v1 et v2.

`float dot(Vector3f v1)`
Calcule et renvoie le point produit par ce vecteur et le vecteur v1.

`void normalize()`
Normalise ce vecteur.

`void normalize(Vector3f v1)`
Modifie la valeur de ce vecteur à la normale du vecteur v1.

`float angle(Vector3f v1)`
Renvoie l'angle en radians entre ce vecteur et le paramètre du vecteur, la valeur renvoyée est contrainte à l'intervalle [0,PI].

Bien sûr, il y a une subtilité, des différences sont prévisibles entre les constructeurs et les méthodes de Vector*, dues au nombre ou au type de donnée.

2.4.4Les classes TextCoord

Il y seulement deux classes TextCoord qui peuvent être utilisées pour représenter un jeu de coordonnées de texture à un sommet : TexCoord2f et TexCoord3f. Les classes TexCoord2f conservent les coordonnées de texture comme une paire (s,t) de coordonnées ; TexCoord3f par trois (s, t, r) coordonnées.

Les constructeurs pour les classes TexCoord* sont une fois encore similaires aux constructeurs de Tuple*. Comme les classes Color*, les classes TextCoord* n'ont pas de méthodes additionnelles, donc elles s'appuient sur les méthodes dont elles héritent depuis leur super-classe Tuple*.

2.5 Les classes Geometry

En graphisme 3D sur ordinateur tout, depuis le simple triangle vers un jumbo jet (avion géant) le plus compliqué, est modelé et rendu en prenant comme données de base des sommets. Avec Java 3D, chaque objet Shape3D se doit d'appeler sa méthode setGeometry() pour faire référence à un et un seul objet Geometry. Pour être plus précis, Geometry est une super-classe abstract, de cette manière l'objet référencé est une instance d'une sous-classe de Geometry.

Les sous-classes de Geometry tombent dans trois vastes catégories :

- Geometry de base-sommet non-indexé (chaque fois qu'un objet visuel est rendu, ces sommets seront utilisés une seule fois)
- Geometry de base-sommet indexé (chaque fois qu'un objet visuel est rendu, ces sommets seront réutilisés).
- Autres objets visuels (les classes Raster, Text3D, et CompressedGeometry).

Cette section couvre les deux premières catégories mentionnées ci-dessus. La hiérarchie de classe pour les classes et les sous-classes de Geometry sont décrites par la Figure 2-10 Hiérarchie de classe Geometry.

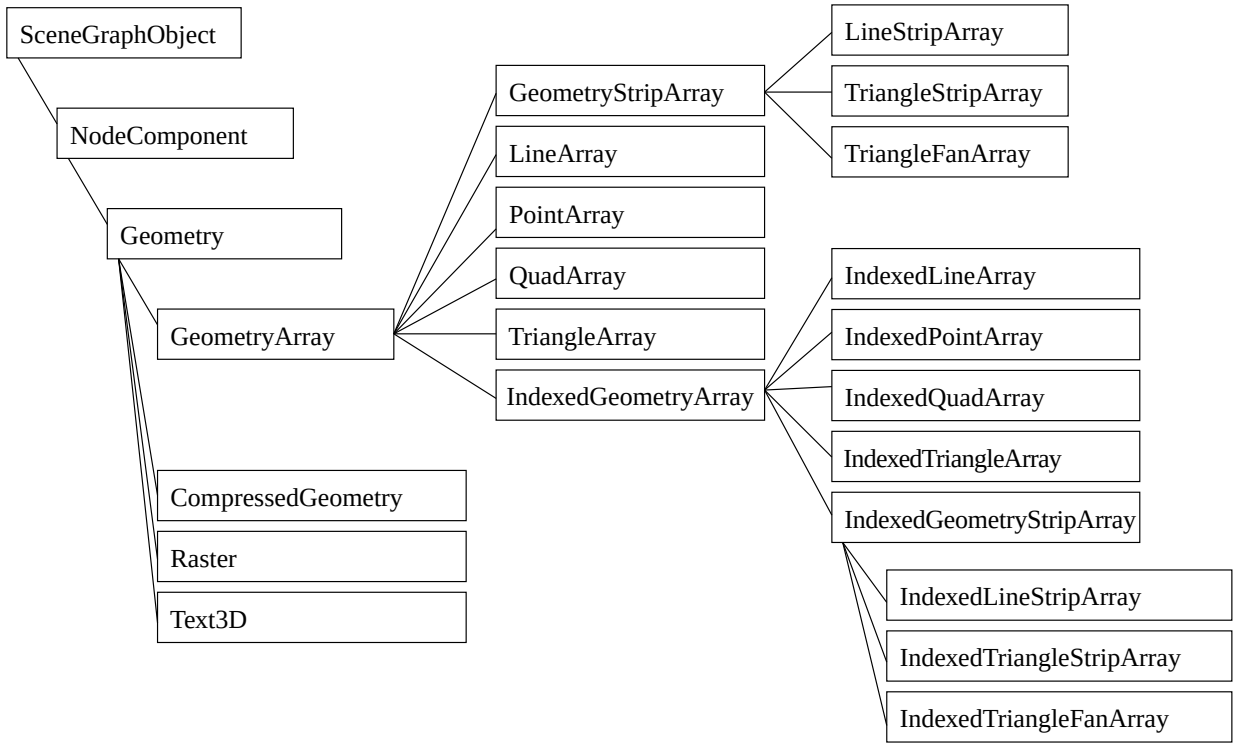


Figure 2-10 Hiérarchie de classe Geometry

2.5.1 La classe GeometryArray

Comme vous devez en déduire par le nom de la classe, les sous-classes de Geometry peuvent être utilisées pour déterminer des points, des lignes, et des polygones pleins (triangles et quadrilatères). Les primitives de base-sommet sont des sous-classes de la classe abstract GeometryArray, laquelle indique que chacune d'entre-elles possède des tableaux qui conservent les données par sommet.

Par exemple, si un objet GeometryArray est utilisé pour spécifier un triangle, un tableau à trois éléments est défini : un élément pour chaque sommet. Chaque élément de ce tableau conserve la coordonnée de position pour ce sommet (lequel peu être défini par un objet Point* ou une donnée similaire). En plus de la coordonnée de localisation, trois tableaux supplémentaires peuvent être définis optionnellement pour maintenir de données de couleur, normale de surface, et de coordonnée de texture. Ces tableaux, contenants les coordonnées, les couleurs, les normales de la surface, et les coordonnées de texture, sont les « tableaux de données » [data arrays].

Il y a trois étapes dans la vie d'un objet GeometryArray :

1. Construction d'un objet vide.
2. Remplissage de cet objet avec les données.
3. Association (faire référence) à l'objet depuis les objets Shape 3D (un ou plus).

Étape 1 : Construction d'un objet GeometryArray vide

À la construction initiale d'un objet GeometryArray, deux choses doivent être définies :

- le nombre de sommets (éléments du tableau) qui seront nécéssaires.
- le type de donnée (coordonnée de position, couleur, normale de surface, et/ou coordonnée de texture) devant être stocké par chaque sommet. Ceci est nommé le *vertex format (format du sommet).*

Il y a une seule et unique méthode constructeur GeometryArray :

Le constructeur de GeometryArray

`GeometryArray(int vertexCount, int vertexFormat)`
Construit un objet GeometryArray vide avec le nombre de sommets, et de structure de sommet spécifié. Un ou plusieurs drapeau [flag] sont à opération logique « OR » pour décrire les données par sommet. Les constantes de flag utilisées pour spécifier le format sont :

COORDINATES :
Spécifie que ce tableau de sommet contient les coordonnées. Cette donnée doit être fixé.
NORMALS : Spécifie que ce tableau de sommet contient les normales de la surface.
COLOR_3 : Spécifie que ce tableau de sommet contient des couleurs hormis la transparence.
COLOR_4 : Spécifie que ce tableau de sommet contient des couleurs avec la transparence.
TEXTURE_COORDINATE_2 :
Spécifie que ce tableau de sommet contient les coordonnées 2D d'une texture.
TEXTURE_COORDINATE_3 :
Spécifie que ce tableau de sommet contient les coordonnées 3D d'une texture.

Pour chaque drapeau de format de sommet fixé, il y a un tableau correspondant crée à l'intérieur de l'objet GeometryArray. Chacun de ces tableau est de taille vertexCount.

Allons voir comment ce constructeur travaille, mais avant rappellons que GeometryArray est une classe abstraite. Donc, en fait vous appellez le constructeur pour une des sous-classes de GeometryArray, par exemple, LineArray. (Un objet LineArray décrit un jeu de sommets, et tout les deux sommets définie les points finaux d'une ligne. Le constructeur et les méthodes de LineArray sont très similaires à celles de sa super-classe GeometryArray. LineArray est décrite avec plus de détails dans la Partie 2.5.2)

Le Fragment de code 2-4 montre la classe Axis pour le programme `examples/Geometry/AxisApp.java` lequel utilise plusieurs objets LineArray pour desssiner des lignes afin de représenter les axes x, y, et z. L'objet axe X crée un objet avec deux sommets (pour dessiner une ligne entre eux), avec seulement des données de coordonnées de position. L'objet axe Y possède aussi deux sommets, mais prend une couleur RGB, ainsi que des coordonnées de position, pour chaque sommet. Donc, la ligne de l'axe Y sera dessinée par l'interpolation des couleurs depuis un sommet vers l'autre. Pour finir, l'axe Z prend dix sommets avec des données de coordonnées et de couleur pour chaque sommet. Cinq lignes de couleur interpolée sont dessinées, une entre chaque paire de sommets. Notez que l'utilisation de l'opérateur logique « OR » pour le format des deux axes Y et Y.

```
1. // construit l'objet représentant l'axe X
2. LineArray axisXLines= new LineArray (2, LineArray.COORDINATES);
3.
4. // construit l'objet représentant l'axe Y
5. LineArray axisYLines = new LineArray(2, LineArray.COORDINATES
6.                                       | LineArray.COLOR_3);
7.
8. // construit l'objet représentant l'axe Z
9. LineArray axisZLines = new LineArray(10, LineArray.COORDINATES
10.                                      | LineArray.COLOR_3);
```

Fragment de code 2-4 Les constructeurs de GeometryArray

Attention! La classe Axis dans `AxisApp.java` est differente de la classe Axis définie dans `examples/geometry/Axis.java`, laquelle utilise seulement un seul objet LineArray. Vérifiez que vous avez la bonne. La classe Axis définie dans `Axis.java` est destinée à être utilisée dans vos programmes, alors que `AxisApp.java` est le programme de démonstration pour ce tutorial. Aussi, la classe Axis définit dans Axis.java demontre la création d'une classe d'objet visuel qui étend la classe Shape3D.

Étape 2: Remplir l'objet GeometryArray avec des données

Après la création de l'objet GeometryArray, il faut assigner des valeurs aux tableaux, en correspondance avec les formats de sommets assignés. Ceci peut être fait par sommet, ou en utilisant un tableau pour assigner les données à plusieurs sommets en appellant une seule méthode. Les méthodes disponibles sont:

Les méthodes de GeometryArray (liste partielle)

GeometryArray est la super-classe pour PointArray, LineArray, TriangleArray, QuadArray,

GeometryStripArray, et IndexedGeometryArray.

void setCoordinate(int index, float[] coordinate)

void setCoordinate(int index, double[] coordinate)

void setCoordinate(int index, Point* coordinate)
Modifie les coordonnées associées aux sommets à l'index spécifié pour cet objet.

void setCoordinates(int index, float[] coordinates)

void setCoordinates(int index, double[] coordinates)

void setCoordinates(int index, Point*[] coordinates)
Modifie les coordonnées associées aux sommets à l'index spécifié pour cet objet.

void setColor(int index, float[] color)

void setColor(int index, byte[] color)

void setColor(int index, Color* color)
Modifie la couleur associée aux sommets à l'index spécifié pour cet objet.

void setColors(int index, float[] colors)

void setColors(int index, byte[] colors)

void setColors(int index, Color*[] colors)
Modifie les couleurs associées aux sommets à l'index spécifié pour cet objet.

Le méthodes de GeometryArray (liste partielle, continuée)

`void setNormal(int index, float[] normal)`

`void setNormal(int index, Vector* normal)`
Modifie la normale associée aux sommets à l'index spécifié pour cet objet.

`void setNormals(int index, float[] normals)`

`void setNormals(int index, Vector*[] normals)`
Modifie les normales associées aux sommets en commençant l'index spécifié pour cet objet.

`void setTextureCoordinate(int index, float[] texCoord)`

`void setTextureCoordinate(int index, Point* coordinate)`
Modifie les coordonnées de texture associées aux sommets à l'index spécifié pour cet objet.

`void setTextureCoordinates(int index, float[] texCoords)`

`void setTextureCoordinates(int index, Point*[] texCoords)`
Modifie les coordonnées de texture associées aux sommets en commençant à l'index spécifié pour cet

Le Fragment de code 2-5 décrit l'usage des méthodes de GeometryArray pour stocker les valeurs de coordonnées et de couleur dans les objets LineArray. L'objet axe X appelle seulement la méthode setCoordinate() pour stocker les données de coordonnées de position. L'objet axe Y appelle les deux setColor() et setCoordinate() pour charger les valeurs de couleur RGB et de coordonnées de position. Enfin l'objet axe Z appelle dix fois setCoordinate() individuellement pour chaque sommet et une fois setColors() pour charger l'ensemble de dix sommets par l'appel d'une seule méthode.

```
1. axisXLines.setCoordinate(0, new Point3f(-1.0f, 0.0f, 0.0f));
2. axisXLines.setCoordinate(1, new Point3f( 1.0f, 0.0f, 0.0f));
3.
4. Color3f red = new Color3f(1.0f, 0.0f, 0.0f);
5. Color3f green = new Color3f(0.0f, 1.0f, 0.0f);
6. Color3f blue = new Color3f(0.0f, 0.0f, 1.0f);
7. axisYLines.setCoordinate(0, new Point3f( 0.0f,-1.0f, 0.0f));
8. axisYLines.setCoordinate(1, new Point3f( 0.0f, 1.0f, 0.0f));
9. axisYLines.setColor(0, green);
10. axisYLines.setColor(1, blue);
11.
12. axisZLines.setCoordinate(0, z1);
13. axisZLines.setCoordinate(1, z2);
14. axisZLines.setCoordinate(2, z2);
15. axisZLines.setCoordinate(3, new Point3f( 0.1f, 0.1f, 0.9f));
16. axisZLines.setCoordinate(4, z2);
17. axisZLines.setCoordinate(5, new Point3f(-0.1f, 0.1f, 0.9f));
18. axisZLines.setCoordinate(6, z2);
19. axisZLines.setCoordinate(7, new Point3f( 0.1f,-0.1f, 0.9f));
20. axisZLines.setCoordinate(8, z2);
21. axisZLines.setCoordinate(9, new Point3f(-0.1f,-0.1f, 0.9f));
22.
```

```
23. Color3f colors[] = new Color3f[9];
24. colors[0] = new Color3f(0.0f, 1.0f, 1.0f);
25. for(int v = 0; v < 9; v++)
26.   colors[v] = red;
27. axisZLines.setColors(1, colors);
```

Fragment de code 2-5 Stockage des données dans un objet GeometryArray

La couleur par défaut pour les sommets d'un objet GeometryArray est le blanc, à moins que COLOR_3 ou COLOR_4 soit spécifié dans le format des sommets. Quand COLOR_3 ou COLOR_4 est spécifié, la couleur par défaut est alors le noir. Quand des lignes ou des polygones pleins sont rendus avec des couleurs différentes aux sommets, la couleur est légèrement illuminé (interpolé) entre les sommets en utilisant une illumination Gouraud.

Étape 3 : Référencer les objets Shape3D aux objets de GeometryArray

Finalement, le Fragment de code 2-6 montre comment les objets GeometryArray sont référencés par la création de nouveaux objets Shape3D. Ensuite, les objets Shape3D sont ajoutés à un BranchGroup, lequel est ajouté autre part à l'ensemble du graphe scénique. (à la différence des objets GeometryArray, lesquels sont des NodeComponent, Shape3D est une sous-classe de Node, ainsi les objets Shape3D peuvent être ajoutés en tant qu'enfants à un graphe scénique.)

```
1. axisBG = new BranchGroup();
2.
3. axisBG.addChild(new Shape3D(axisYLines));
4. axisBG.addChild(new Shape3D(axisZLines));
```

Fragment de code 2-6 Objets GeometryArray référencés par des objets Shape3D

La Figure 2-11 montre une partie du graphe scénique crée par la classe Axis dans AxisApp.java.

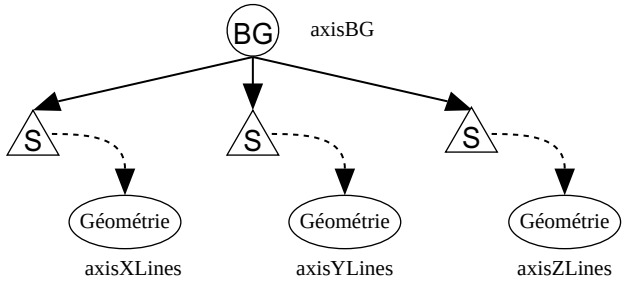


Figure 2-11 La classe Axis dans AxisApp.java crée ce graphe scénique.

2.5.2 Les sous-classes de GeometryArray

Comme il est discuté dans la Partie précédente, la classe GeometryArray est une super-classe abstraite pour bien d'autres sous-classe utilitaires, comme LineArray. La Figure 2-12 montre la hiérarchie de la classe GeometryArray et d'une partie de ces sous-classes. La principale distinction entre ces sous-classes est la façon dont le rendu Java 3D décide de rendre leurs sommets.

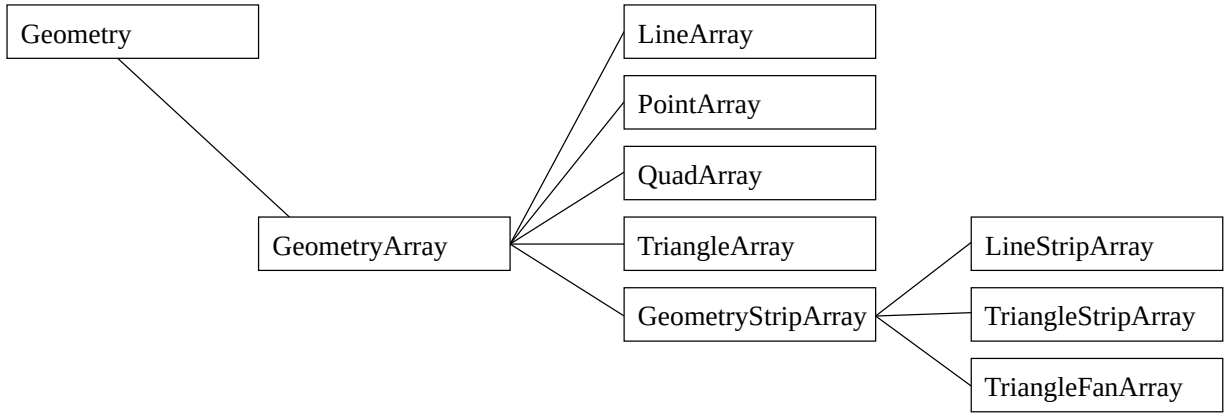


Figure 2-12 Sous-classes non-indexées GeometryArray

La Figure 2-13 montre des exemples des quatre sous-classes GeometryArray : PointArray, LineArray, TriangleArray, et QuadArray (les seules qui ne sont pas aussi des sous-classes de GeometryStripArray). Dans cette figure, les trois jeux de sommets les plus à gauche montrent les même six points sommets rendant six points, trois lignes, ou deux triangles. La quatrième figure montre quatre sommets définissant un quadrilatère. Notez qu'aucun des sommets est partagé : chaque ligne ou polygone plein est rendu indépendamment des autres.

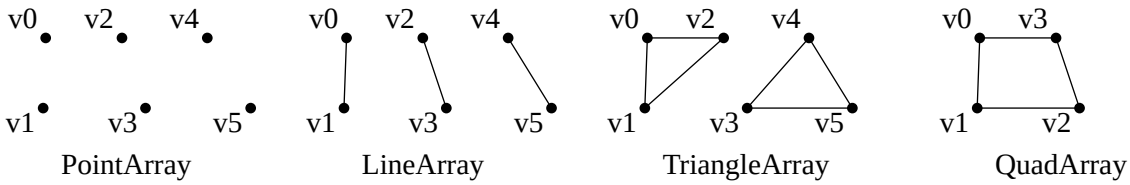


Figure 2-13 Sous-classes de GeometryArray

Par défaut, l'intérieur des triangles et des quadrilatères est plein. Dans les prochaines parties, vous apprendrez que les attributs peuvent influencer la manière de remplissage des primitives dans de différentes directions.

Ces quatre sous-classes héritent leurs constructeurs et leurs méthodes de GeometryArray. Leurs constructeurs sont listés ci-dessous. Pour leurs méthodes, retournez en arrière au Bloc de référence nommé Méthodes de GeometryArray.

Constructeurs des sous-classes GeometryArray

Construit un objet vide avec le nombre spécifié de sommets et le format du sommet. Le format est un ou plusieurs flags individuels d'opérateur logique « OR » afin de décrire les données par-sommet. Les flags de format sont les mêmes que ceux définis dans la super-classe GeometryArray.

```
PointArray(int vertexCount, int vertexFormat)
LineArray(int vertexCount, int vertexFormat)
TriangleArray(int vertexCount, int vertexFormat)
QuadArray(int vertexCount, int vertexFormat)
```

Pour voir l'utilisation de ces constructeurs et méthodes, retournez aux Fragment de code 2-4, Fragment de code 2-5, Fragment de code 2-6, lesquels utilisent un objet LineArray.

Si vous procédez au rendu de quadrilatères, faites attention que les sommets ne forment pas une géométrie concave, ou non-plane. Si c'est le cas, ils ne présenteront pas un rendu correct.

2.5.3Sous-classes de GeometryStripArray

La partie précédente décrivait quatre sous-classes de GeometryArray ne permettant pas de réutiliser les sommets. Certaines configurations géométriques invitent à la réutilisation des sommets, c'est pourquoi des classes spécialisées peuvent avoir comme résultat de meilleures performances au rendu.

GeometryStripArray est une classe abstract à partir de laquelle des primitives de bande [strip] (pour la création de lignes et de surfaces composés) sont dérivées. GeometryStripArray est la super-classe de LineStripArray, TriangleStripArray, et TriangleFanArray. La Figure 2-14 montre une instance de chaque type de ruban et comment les sommets sont réutilisés. Le LineStripArray présente des lignes connectés. Le TriangleStripArray réutilise les triangles partagant un coté, réutilisant le dernier sommet rendu. Le TriangleFanArray réutilise le tout premier sommet des son ruban pour chaque triangle.

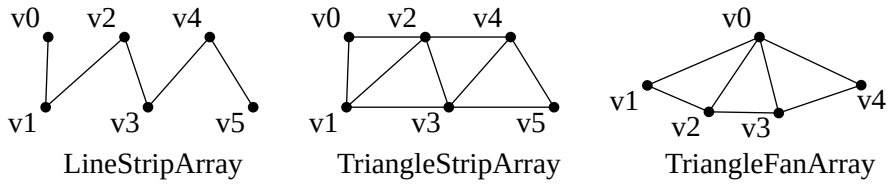


Figure 2-14 Sous-classes de GeometryStripArray.

Le GeometryStripArray possède un constructeur différent de celui du GeometryArray. Le constructeur de GeometryStripArray prend un troisième paramètre, un tableau comptant les sommets par bande, autorisant un seul objet à maintenir de multiples bandes. (GeometryStripArray introduit aussi un couple de méthodes de demande d'informations, getNumStrips() et getStripVertexCounts(), lesquelles sont rarement utilisées.)

Constructeurs des sous-classes de IndexedGeometryStripArray

Construit un objet vide avec le nombre spécifié de sommets, de format de sommet, le nombre d'indices dans ce tableau, et un tableau comptant les sommets par ruban.

```
IndexedGeometryStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedLineStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedTriangleStripArray(int vc, int vf, int ic, int stripVertexCounts[]))
IndexedTriangleFanArray(int vc, int vf, int ic, int stripVertexCounts[]))
```

Notez que Java 3D ne supporte pas les primitives pleines ayant plus de quatre cotés. Le programmeur est responsable de l'utilisation de quadrillages pour alléger les polygones complexes dans les objets Java 3D, tels que des bande ou des éventails de triangles. La classe utilitaire Triangulator convertit les polygones complexes en triangles[8].

La classe Triangulator

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

Utilisée pour convertir une géométrie non-triangulée en triangles pour le rendu en Java 3D. Les polygones peuvent être concaves, non-plan, et contenir des trous (voir GeometryInfo.setContourCounts()). Les polygones non-plans sont projetés sur le plan le plus proche. NOTE : Voir la documentation courante pour les limitations. Voir la Partie 3.3 de ce tutorial pour plus d'information.

Résumé du Constructeur

`Triangulator()`
Crée un objet Triangulator.

Résumé de la méthode

`void triangulate(GeometryInfo ginfo)`
Cette routine convertit l'objet GeometryInfo depuis la primitive de type POLYGON_ARRAY en une primitive de type TRIANGLE_ARRAY en utilisant la technique de décomposition des polygones.

Paramètres :
ginfo - `com.sun.j3d.utils.geometry.GeometryInfo` à trianguler.

Exemple d'utilisation :

```
Triangulator tr = new Triangulator();
tr.triangulate(ginfo); // ginfo contient la géometrie
shape.setGeometry(ginfo.getGeometryArray()); // shape est un Shape3D
```

Code de Yo-yo démonstration du TriangleFanArray

L'objet Yo-Yo du programme `Yoyoapp.java` montre comment utiliser l'objet TriangleFanArray pour modéliser la géométrie d'un Yo-Yo. Le TriangleFanArray contient quatre éventails indépendants : Deux faces extérieures (disques circulaires) et deux faces intérieures (cônes). Un seul objet TriangleFanArray est nécessaire pour la représentation des quatre éventails.

La Figure 2-15 montre trois rendus des TriangleFanArray. La première vue montre le rendu par défaut, comme des polygones blancs et pleins. Pourtant, il est difficile de voir les détails, surtout la position des sommets. Pour un processus de rendu des polygones pleins comme des lignes, voir la classe PolygonAttribute dans la Partie 2.6.

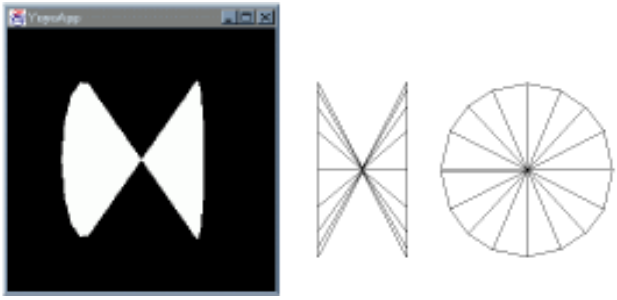


Figure 2-15 Trois vues du Yo-yo.

Dans le Fragment de code 2-7, la méthode `yoyoGeometry()` crée et renvoie le TriangleFanArray désiré. Aux lignes 15-18 est calculé le point central pour les quatre éventails. Chaque éventail possède 18 sommets, lesquels sont calculés aux lignes 20-28. Aux lignes 30-32 sont construits les objets pleins TriangleFanArray, et les calculs précédents de données de coordonnées (des lignes 15-28) se trouvent stockés à la ligne 34.

```
1. private Geometry yoyoGeometry() {
2.
3.    TriangleFanArray tfa;
4.    int N = 17;
5.    int totalN = 4*(N+1);
6.    Point3f coords[] = new Point3f[totalN];
7.    int stripCounts[] = {N+1, N+1, N+1, N+1};
8.    float r = 0.6f;
9.    float w = 0.4f;
10.   int n;
11.   double a;
12.   float x, y;
13.
14.   // Fixe les points centraux des quatre bandeaux de triangles.
15.   coords[0*(N+1)] = new Point3f(0.0f, 0.0f, w);
16.   coords[1*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
17.   coords[2*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
18.   coords[3*(N+1)] = new Point3f(0.0f, 0.0f, -w);
19.
20.   for (a = 0,n = 0; n < N; a = 2.0*Math.PI/(N-1) * ++n){
21.         x = (float) (r * Math.cos(a));
22.         y = (float) (r * Math.sin(a));
23.
24.         coords[0*(N+1)+N-n] = new Point3f(x, y, w);
25.         coords[1*(N+1)+n+1] = new Point3f(x, y, w);
26.         coords[2*(N+1)+N-n] = new Point3f(x, y, -w);
27.         coords[3*(N+1)+n+1] = new Point3f(x, y, -w);
28.   }
29.
30.   tfa = new TriangleFanArray (totalN,
31.                               TriangleFanArray.COORDINATES,
32.                               stripCounts);
33.
34.   tfa.setCoordinates(0, coords);
35.
36.   return tfa;
37.} // fin de la méthode yoyoGeometry de la classe Yoyo
```

Fragment de code 2-7 La méthode yoyoGeometry() créatrice de l'objet TriangleFanArray.

Le yo-yo tout blanc est seulement un point de départ. La Figure 2-16 montre un objet similaire, modifié pour introduire des couleurs à chaque sommet. La méthode modifiée `yoyoGeometry()`, laquelle renferme les couleurs dans l'objet TriangleFanArray, est décrite dans le Fragment de code 2-8. Les lignes 23 à 26, 36 à 39, et la ligne 49 déterminent la valeur couleur pour chaque sommet.

D'autres possibilités existent pour spécifier l'apparence d'un objet visuel par l'utilisation des lumières, des textures, et des propriétés de matériaux pour un objet visuel. Ces sujets ne sont pas abordés dans ce

module du Tutorial. Les lumières et les textures sont les sujets du module 2 du tutorial.

```
1. private Geometry yoyoGeometry() {
2.
3.    TriangleFanArray tfa;
4.    int N = 17;
5.    int totalN = 4*(N+1);
6.    Point3f coords[] = new Point3f[totalN];
7.    Color3f colors[] = new Color3f[totalN];
8.    Color3f red = new Color3f(1.0f, 0.0f, 0.0f);
9.    Color3f yellow = new Color3f(0.7f, 0.5f, 0.0f);
10.   int stripCounts[] = {N+1, N+1, N+1, N+1};
11.   float r = 0.6f;
12.   float w = 0.4f;
13.   int n;
14.   double a;
15.   float x, y;
16.
17.   // Fixe le point central pour les quatre bandeaux de triangles
18.   coords[0*(N+1)] = new Point3f(0.0f, 0.0f, w);
19.   coords[1*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
20.   coords[2*(N+1)] = new Point3f(0.0f, 0.0f, 0.0f);
21.   coords[3*(N+1)] = new Point3f(0.0f, 0.0f, -w);
22.
23.   colors[0*(N+1)] = red;
24.   colors[1*(N+1)] = yellow;
25.   colors[2*(N+1)] = yellow;
26.   colors[3*(N+1)] = red;
27.
28.   for(a = 0,n = 0; n < N; a = 2.0*Math.PI/(N-1) * ++n){
29.         x = (float) (r * Math.cos(a));
30.         y = (float) (r * Math.sin(a));
31.         coords[0*(N+1)+n+1] = new Point3f(x, y, w);
32.         coords[1*(N+1)+N-n] = new Point3f(x, y, w);
33.         coords[2*(N+1)+n+1] = new Point3f(x, y, -w);
34.         coords[3*(N+1)+N-n] = new Point3f(x, y, -w);
35.
36.         colors[0*(N+1)+N-n] = red;
37.         colors[1*(N+1)+n+1] = yellow;
38.         colors[2*(N+1)+N-n] = yellow;
39.         colors[3*(N+1)+n+1] = red;
40.   }
41.   tfa = new TriangleFanArray (totalN,
42.               TriangleFanArray.COORDINATES|TriangleFanArray.COLOR_3,
43.               stripCounts);
44.
45.   tfa.setCoordinates(0, coords);
46.   tfa.setColors(0,colors);
47.
48. return tfa;
49. } // fin de la méthode yoyoGeometry de la classe Yoyo
```

Fragment de code 2-8 méthode yoyoGeometry() modifiée ajout des couleurs.

Le lecteur observateur aura remarqué les différences aux lignes 36 à 39. Le code est écrit pour faire des faces avant de chaque triangle dans la géométrie l'exterieur du yo-yo. Le débat sur les faces avant et arrières d'un triangle, et pourquoi cela fait une différence est abordé dans la Partie 2.6.4.

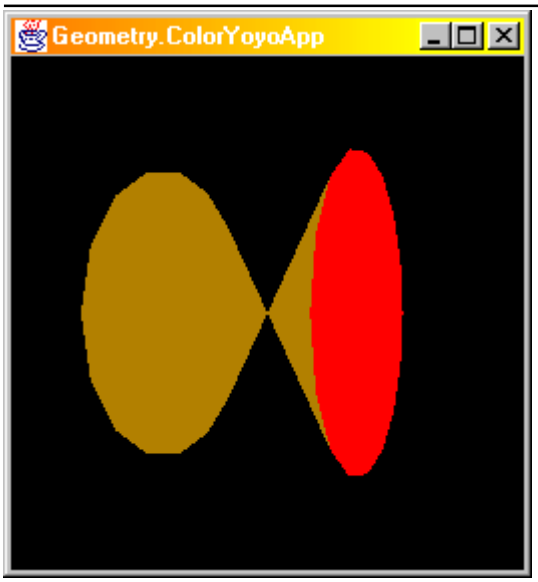


Figure 2-16 Yo-yo avec des polygones pleins et colorés

2.5.4 Sous-classes de IndexedGeometryArray

Les sous-classes précédemment décrites de GeometryArray déclarent les sommets par gaspillage. Seule la sous-classe GeometryStripArray possède la même limite de réutilisation des sommets. De nombreux objets géométriques invitent à la réutilisation des sommets. Par exemple, pour définir un cube, chacun de ces huit sommets est utilisé par trois carrés différents. De manière plus précise, un cube nécessite la spécification de 24 sommets, malgré que seuls huit uniques sommets sont nécessaires (16 des 24 sont redondant).

Les objets IndexedGeometryArray fournissent un niveau supplémentaire d'adressage, ainsi les sommets redondants seront évités. Les tableaux contenant des information de sommet doit pourtant être fourni, mais les sommets peuvent être stockés dans n'importe quel ordre, et n'importe quel sommet peu être réutilisé pendant le processus de rendu. On appelle ces tableaux, contenant les coordonnées, les couleurs, les normales de la surface, et les coordonnées de texture, les « data arrays » (tableaux de donnés).

Pourtant, les objets IndexedGeometryArray on besoin également de tableaux ("index arrays") qui contiennent des indices dans les « data arrays ». Il y a jusqu'à quatre « index arrays » : indices de coordonnées, indices de couleur, indices de normale de surface, et les indices de coordonnées de texture, lesquels correspondent aux « data arrays ». Le nombre d'éléments dans chaque index array est le même et typiquement plus grand que le nombre d'éléments dans chaque tableau de donnée.

Les "index arrays" peuvent avoir de multiples références au même sommet dans les « data arrays ». Les valeurs dans ces « index arrays » déterminent l'ordre dans lequel les données de sommet sont accèdés pendant le rendu. La Figure 2-17 montre les relations entre les tableaux d'index et de donnés de coordonnées prenant un un cube comme exemple.

Il est important de mentionner qu'il y a un prix à payer pour la réutilisation des sommets produite par une géométrie dotée d'un index – vous le payerez en performance. L'indexation d'une géométrie au moment du

rendu ajoute plus de travail au processus de rendu. Si la performance est un problème, utilisez les bandes [strips] chaque fois que c'est possible et évitez les géomètries indexées. La géométrie indexée est utile lorsque la vitesse n'est pas cruciale et il y a un peu de mémoire à gagner par l'indexation, ou quand l'indexation fournit des facilités de programmation.

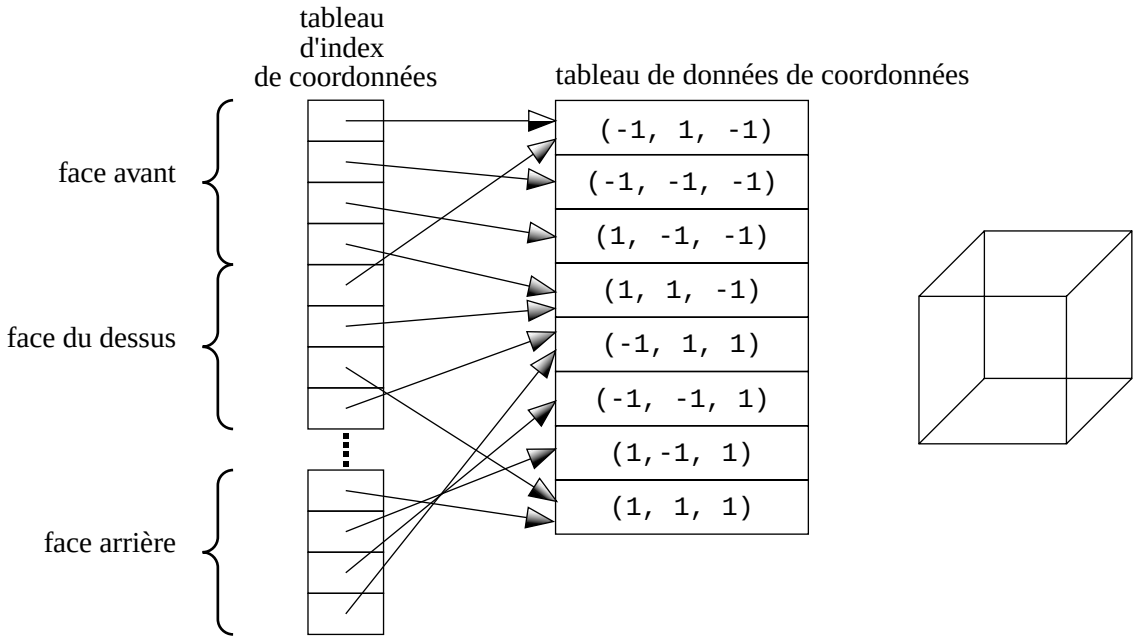


Figure 2-17 Tableaux d'index et de données pour un Cube

Les sous-classes de IndexGeometryArray mettent en parallèle les sous-classes de GeometryArray. La hiérarchie de la classe IndexGeometryArray est décrite dans la Figure 2-18.

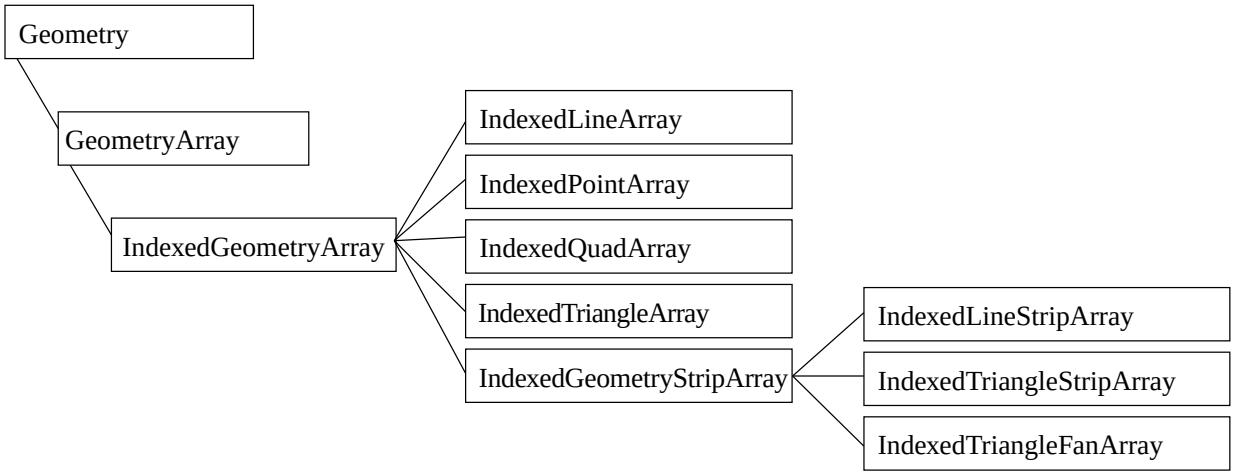


Figure 2-18 Sous-classes d'IndexedGeometryArray

Les constructeurs pour IndexedGeometryArray, IndexedGeometryStripArray, et leurs sous-classes sont similaires aux constructeurs de GeometryArray et GeometryStripArray. Les classes de données indexées possèdent un paramètre supplémentaire pour déterminer le nombre d'index qui seront utilisés pour décrire la géométrie (le nombre d'éléments dans les tableaux indexés).

Constructeurs de IndexedGeometryArray et des sous-classes

Construit un objet vide avec le nombre spécifié de sommets, de format de sommet, et du nombre d'indices dans ce tableau.

`IndexedGeometryArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedPointArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedLineArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedTriangleArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedQuadArray(int vertexCount, int vertexFormat, int indexCount)`

Constructeurs de IndexedGeometryArray et des sous-classes

Construit un objet vide avec le nombre spécifié de sommets, de format de sommet, et du nombre d'indices dans ce tableau.

`IndexedGeometryArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedPointArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedLineArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedTriangleArray(int vertexCount, int vertexFormat, int indexCount)`

`IndexedQuadArray(int vertexCount, int vertexFormat, int indexCount)`

IndexedGeometryArray, IndexedGeometryStripArray, et leurs sous-classes héritent des méthodes de GeometryArray et GeometryStripArray pour charger les « tableaux de donnés ». Les classes de données indexées ont des méthodes supplémentaires pour charger les indices dans les « tableaux d'index ».

Méthodes d'IndexedGeometryArray (liste partielle)

`void setCoordinateIndex(int index, int coordinateIndex)`
Établit l'index de coordonnée associé avec ce sommet à l'index spécifié pour cet objet.

`void setCoordinateIndices(int index, int[] coordinateIndices)`
Établit les indexes de coordonnée associés avec ces sommets commencant à l'index spécifié pour cet objet..

`void setColorIndex(int index, int colorIndex)`
Fixe la couleur d'index associée avec ce sommet à l'index spécifié pour cet objet.

`void setColorIndices(int index, int[] colorIndices)`
Fixe les indexes de couleur associés avec ces sommets commencant à l'index spécifié pour cet objet.

`void setNormalIndex (int index, int normalIndex)`
Fixe l'index de normale associée avec ce sommet à l'index spécifié pour cet objet.

`void setNormalIndices (int index, int[] normalIndices)`
Fixe les indexes de normale associés avec ces sommets commencant à l'index spécifié pour cet objet.

`void setTextureCoordinateIndex (int index, int texCoordIndex)`
Fixe l'index de texture associés avec ce sommet commencant à l'index spécifié pour cet objet.

`void setTextureCoordinateIndices (int index, int[] texCoordIndices)`
Fixe les indexes de texture associés avec ces sommets commencant à l'index spécifié pour cet objet.

2.5.5 Axis.java est un exemple d'IndexedGeometryArray

Le sous-répertoire `examples/geometry` contient le code source d' `Axis.java`. Ce fichier définit l'objet visuel Axis qui est utile pour visualiser les axes et l'origine dans un univers virtuel. Il sert également d'exemple de géométrie indexée.

L'objet Axis définit 18 sommets et 30 indices pour déterminer 15 lignes. Il y a cinq lignes par axes afin de créer une simple flèche en 3D.

2.6 Apparence et attributs

Les objets Shape3D peuvent faire référence à la fois à des objets Geometry et ou Appearence. Comme il est abordé dans la Partie 2.5, un objet Geometry détermine les informations par-sommet d'un objet visuel. Les informations par-sommet dans un objet Geometry peuvent spécifier la couleur des objets visuels. Les données dans un objet Geometry sont souvent insuffisantes pour décrire entièrement l'aspect d'un objet. Dans tous les cas, un objet Appearance est nécessaire. Un objet Appearance ne contient pas les informations sur la manière dont l'objet Shape3D va apparaître, mais un objet Appearance sait où trouver les données d'apparence. Un objet Appearence (qui est déjà une sous-classe de composant de nœud) peut référencer plusieurs objets des autres sous-classes de la classe abstraite NodeComponent. Donc, l'information qui décrit l'apparence d'une primitive géométrique est dite stockée à l'intérieur d'un « ensemble d'apparence », comme décrit à la Figure 2-19.

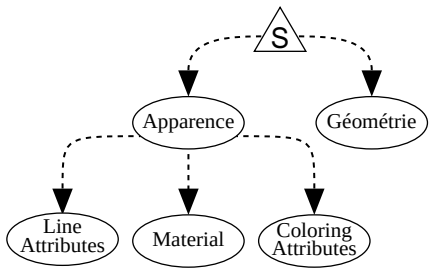


Figure 2-19 Un ensemble d'Appearence (appearence bundle).

Un objet Appearance peut se référer à plusieurs sous-classes de NodeComponent appelées les objets d'attributs d'apparence, incluants :

- PointAttributes
- LineAttributes
- PolygonAttributes
- ColoringAttributes
- TransparencyAttributes
- RenderingAttributes
- Material
- TextureAttributes
- Texture
- TexCoordGeneration

Les six premières sous-classes de NodeComponent listées ici sont expliquées dans cette partie. Pour les quatre sous-classes restantes de la liste, Material est utilisé pour l'éclairage, et les trois dernières sont utilisées pour le mapping de texture. L'éclairage et le mapping de texture sont des sujets avancés, et ne sont pas abordés dans cette section.

Un objet Appearance avec les objets attributs auxquels il se réfère est nommé appearance bundle (paquet d'apparence). Pour référencer n'importe lequel de ces composants de nœud, un objet Appearance possède une méthode avec un nom évident. Par exemple, pour un objet Appearance se référant à un objet ColoringAttributes, il utilise la méthode `Appearance.setColoringAttributes()`. Un exemple simple de code ressemble au Fragment de code 2-9 :

```
1. ColoringAttributes ca = new ColoringAttributes();
2. ca.setColor (1.0, 1.0, 0.0);
3. Appearance app = new Appearance();
4. app.setColoringAttributes(ca);
5. Shape3D s3d = new Shape3D();
6. s3d.setAppearance (app);
7. s3d.setGeometry (someGeomObject);
```

Fragment de code 2-9 Usage des objets Node Component Appearance et ColoringAttributes.

Le graphe scénique qui résulte de ce code se trouve dans la Figure 2-20.

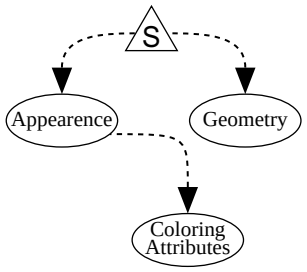


Figure 2-20 Le paquet d'apparence crée par le Fragment de code 2-9.

2.6.1 Le composant de nœud Appearance

Les deux prochains Blocs de référence listent le constructeur et autres méthodes de la classe Appearence.

Le constructeur d'Appearance

Le constructeur par défaut d'Appearance crée un objet Appearance avec toutes les références de composant d'objet à null. Les valeurs par défaut, pour les composants avec des références null, sont généralement prévisibles :

Les points et les lignes sont dessinées avec les dimensions et largeurs de 1 pixel et sans antialiasing, la couleur intrinsèque est blanche, la transparence est désactivée, et la profondeur de mémoire tampon est activée et est accessible par écriture et lecture.

`Appearance()`

Un composant Appearance référence usuellement un ou plusieurs composants d'attribut, par l'appel à la méthode suivante. Ces classes d'attributs sont décrites dans la Partie 2.6.3.

Les méthodes d'Appearance (à l'exception de l'éclairage et du texturage)

Chaque méthode établit son objet NodeComponent correspondant afin qu'il fasse partie de l'ensemble d'Apparence courant.

```
void setPointAttributes(PointAttributes pointAttributes)
void setLineAttributes(LineAttributes lineAttributes)
void setPolygonAttributes(PolygonAttributes polygonAttributes)
void setColoringAttributes(ColoringAttributes coloringAttributes)
void setTransparencyAttributes(TransparencyAttributes transparencyAttributes)
void setRenderingAttributes(RenderingAttributes renderingAttributes)
```

2.6.2 Partager les objets NodeComponent

Il est légal et souvent désirable pour différents objets de référencer, et donc partager, le même objet NodeComponent. Par exemple dans la Figure 2-21, deux objets Shape3D font référence au même composant d'Appearance. Aussi, deux objets Appearance différents partagent le même composant LineAttributes.

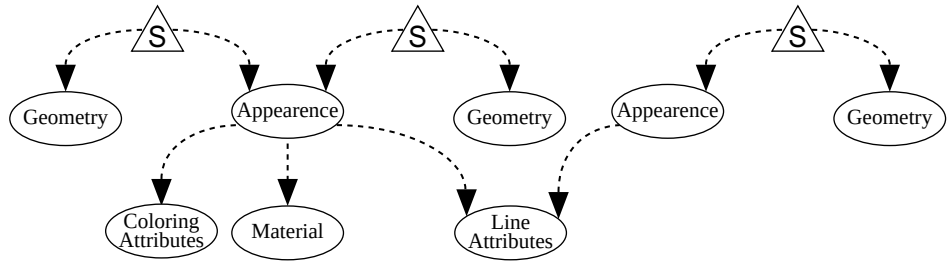


Figure 2-21 Plusieurs objets Appearance partageant un Node Component.

Le partage du même NodeComponent peut améliorer les performances. Par exemple, si plusieurs composants Appearance partagent le même composant LineAttributes, lequel autorisant l'antialiasing, l'engin de rendu Java 3D peut décider de grouper ensemble les formes en fil de fer anti-aliasées. Ce qui réduira au minimum le passage alternatif de l'antialiasing, ce qui sera plus rapide.

Notez qu'il est illégal pour un Node de posséder plus d'un parent. Pourtant, étant donné que les NodeComponents sont référencés, ils ne sont pas des objets du Node, ainsi ils n'ont pas réellement de parents. Donc, les objets NodeComponent peuvent être partagés (référencés) par n'importe quel nombre d'autres objets.

2.6.3 Les classes d'Attribute

Dans cette partie, sont décrites six des sous-classes de NodeComponent qui peuvent être référencées par Appearance (à l'exception de ceux utilisés pour l'éclairage et le texturage).

PointAttributes

Les objets PointAttributes dirigent la façon dont les primitives sont rendues. Par défaut, si un sommet est rendu comme un point, il remplit un seul pixel. Vous pouvez utiliser setPointSize() pour rendre un point plus large. Pourtant, par défaut, un point plus large restera de forme carrée, à moins que vous utilisiez aussi setPointAntialiasingEnable(). Antialiaser les points change les couleurs des pixels afin de faire apparaître le point « arrondi » (ou du moins, des carrés moins visibles).

Les constructeurs de PointAttributes

`PointAttributes()`
Crée un objet composant qui décrit un point de la taille d'un pixel sans antialiasing.

`PointAttributes(float pointSize, boolean state)`
Crée un objet composant qui décrit la taille en pixels pour le point ainsi que la permission d'antialasing.

Les méthodes de PointAttributes

`void setPointSize(float pointSize)`
Décrit la taille des pixels pour les points.

`void setPointAntialiasingEnable(boolean state)`
Active ou désactive l'antialiasing du point. Intéressant visuellement si le point est plus large qu'un pixel.

LineAttributes

Les objets LineAttributes modifient la façon d'apparaitre au rendu des primitives lignes de trois manières. Par défaut, une ligne est dessinée solidement remplie, un pixel de large, et sans aucun antialiasing (l'effet de lissage). Vous pouvez changer ces attributs par l'appel aux méthodes setLinePattern(), setLineWidth(), et setAntialiasingEnable().

Les constructeurs de LineAttributes

`LineAttributes()`
Crée un objet composant, une ligne d'un pixel de large, solidement remplie sans antialiasing.

`LineAttributes(float pointSize, int linePattern, boolean state)`
Crée un objet composant qui décrit la taille en pixel pour les lignes, le décor à utiliser pour le dessin, et si il faut autoriser l'antialiasing.

Les méthodes de LineAttributes

`void setLineWidth(float lineWidth)`
Décrit la largeur en pixel pour les lignes.

`void setLinePattern(int linePattern)`
Où linePattern est une des constantes suivante : PATTERN_SOLID (par défaut),PATTERN_DASH, PATTERN_DOT, ou PATTERN_DASH_DOT. Décrit par quoi les pixels de la ligne doivent être remplis.

`void setLineAntialiasingEnable(boolean state)`
Active ou désactive l'antialiasing de la ligne.

PolygonAttributes

Les PolygonAttributes maîtrisent le type de rendu des primitives de polygone de trois manières : comment le polygone est tramé, si il est sélectionné [culled], et si une compensation spéciale de profondeur est appliquée. Par défaut, un polygone est plein, mais setPolygonMode() peut changer le mode de tramage de telle manière que le polygone soit dessiné en fil de fer (lignes) ou seulement par les points des sommets. (Dans le dernier des deux cas, les LineAttributes, ou les PointAttributes affectent aussi le type de visualisation de la primitive.) La méthode setCullFace() peut être utilisée pour réduire le nombre de polygones qui seront rendus. Si setCullFace() est établit soit à CULL_FRONT ou CULL_BACK, en moyenne, la moitié des polygones ne sera pas rendue.

Par défaut, les sommets rendus à la fois en fil de fer et en polygone pleins ne sont pas toujours tramés avec les mêmes valeurs de profondeur, ce qui peut causer un *agrafage* quand le fil de fer est entièrement visible. Avec le `setPolygonOffset()`, la valeur de profondeur du polygone plein peut jouer des coudes vers l'image plate, de cette manière l'image fil de fer pourra proprement faire le contour de l'objet. `SetBackFaceNormalFlip()` est pratique pour rendre un polygone éclairé et plein, quand les deux cotés du polygone doivent être illuminés [shaded]. Voir la Partie 2.6.4 pour un programme d'exemple qui illumine les deux cotés des polygones.

Les constructeurs de PolygonAttributes

`PolygonAttributes()`
Crée un objet composant avec par défaut des polygones pleins, pas de sélection de face, et pas de compensation du polygone.

`PolygonAttributes(int polygonMode, int cullFace, float polygonOffset)`
Crée un objet composant pour rendre des polygones soit comme des points, des lignes, ou des polygones pleins, avec la sélection des faces déterminée, et la compensation du polygone fixée.

`PolygonAttributes(int polygonMode, int cullFace, float polygonOffset, boolean backFaceNormalFlip)`
Crée un objet composant similaire aux constructeurs précédents, mais renverse aussi la façon dont les faces avant et arrière du polygone sont fixés.

Les méthodes de PolygonAttributes

`void setCullFace(int cullFace)`
Où cullFace est un des suivants : CULL_FRONT, CULL_BACK, ou CULL_NONE. Cull (ne pas rendre) les faces frontales des polygones ou les faces arrières des polygones, ou ne pas « cull » aucun des polygones.

`void setPolygonMode(int polygonMode)`
Où polygonMode est un des suivants : POLYGON_POINT, POLYGON_LINE, ou POLYGON_FILL. Affiche les polygones soit comme des points, des lignes, ou des polygones pleins (valeur par défaut).

`void setPolygonOffset(float polygonOffset)`
Où polygonOffset est l'espace-écran de compensation ajouté pour ajuster la valeur de profondeur des primitives polygones.

`void setBackFaceNormalFlip(boolean backFaceNormalFlip)`
Où backFaceNormalFlipdétermine si les normales des sommets des polygones de faces arrières doivent être tournés (négatif) pour l'éclairage. Quand ce drapeau [flag] est établit à « true » et la sélection des faces arrières désactivée, un polygone est rendu comme si le polygone avait deux cotés avec des normales opposées.

ColoringAttributes

ColoringAttributes contrôle la façon de colorer n'importe quelle primitive. setColor() fixe une couleur intrinsèque, qui dans certaines situations détermine la couleur de la primitive. Aussi, setShadeModel() détermine comment s'interpole la couleur entre les primitives (usuellement des polygones et des lignes).

Les constructeurs de ColoringAttributes

`ColoringAttributes()`
Crée un objet composant utilisant le blanc comme couleur intrinsèque et SHADE_GOURAUD comme illumination par défaut.

`ColoringAttributes(Color3f color, int shadeModel)`

`ColoringAttributes(float red, float green, float blue, int shadeModel)`
Où shadeModel est un parmi SHADE_GOURAUD, SHADE_FLAT, FASTEST, et NICEST. Les deux constructeurs créent un objet composant utilisant des paramètres pour déterminer la couleur intrinsèque et le mode d'illumination.

(Dans la plupart des cas, FASTEST est aussi SHADE_FLAT, et NICEST est aussi SHADE_GOURAUD.)

Les méthodes de ColoringAttributes

`void setColor(Color3f color)`

`void setColor(float red, float green, float blue)`
Les deux méthodes déterminent la couleur intrinsèque.

`void setShadeModel(int shadeModel)`
Où shadeModel est une des constantes suivantes : SHADE_GOURAUD, SHADE_FLAT, FASTEST, ou NICEST. Détermine le type d'illumination pour les primitives rendues.

Comme les couleurs peuvent aussi être définies pour chaque sommet d'un objet Geometry, il peut y avoir un conflit avec la couleur intrinsèque définie par les ColoringAttributes. Dans le cas d'un tel conflit, les couleurs définies dans l'objet Geometry prennent le pas sur la couleur intrinsèque de ColoringAttributes. Aussi, si l'éclairage est autorisé, la couleur intrinsèque ColoringAttributes est entièrement ignorée.

TransparencyAttributes

Le TransparencyAttributes dirige la transparence de n'importe quelle primitive. setTransparency() définit la valeur d'opacitée (plus connue sous le nom d'« alpha blending ») pour la primitive. setTransparencyMode() autorise la transparence et sélectionne quel type de tramage est utilisé pour produire la transparence.

Les Constructeurs de TransparencyAttributes

`TransparencyAttributes()`
Créé un objet composant avec le mode de transparence FASTEST.

`TransparencyAttributes(int tMode, float tVal)`
Où tMode est un parmi BLENDED, SCREEN_DOOR, FASTEST, NICEST, ou NONE, et tVal spécifiant ainsi l'opacitée (0.0 indique l'opacité complète et 1.0 la totale transparence). Crée un objet composant avec une méthode spécifiée pour rendre la transparence et la valeur d'opacitée de l'apparence de cet objet.

Les méthodes de TransparencyAttributes

`void setTransparency(float tVal)`
Ici tVal détermine l'opacité d'un objet (où 0.0 indique l'opacité complète et 1.0 la totale transparence).

`void setTransparencyMode(int tMode)`
Où tMode (parmi BLENDED, SCREEN_DOOR, FASTEST, NICEST, et NONE) détermine si et de quelle manière appliquer la transparence.

RenderingAttributes

Le RenderingAttributes contrôle deux opérations différentes de rendu par-pixel : le test tampon de profondeur [depth buffer] et le test d'alpha. setDepthBufferEnable() et setDepthBufferWriteEnable() détermine quoi et comment le tampon de profondeur est employé pour la suppression des surfaces cachées. setAlphaTestValue() et setAlphaTestFunction() détermine quoi et comment le test d'alpha est utilisé.

Les constructeurs des RenderingAttributes

`RenderingAttributes()`
Crée un objet composant qui définit les statuts de rendu par-pixel avec une autorisation de tampon de profondeur activée et le test d'alpha désactivé.

`RenderingAttributes(boolean depthBufferEnable,`
`boolean depthBufferWriteEnable,float alphaTestValue, int alphaTestFunction)`
Où le depthBufferEnable alterne l'autorisation par on ou off des comparaisons de tampon de profondeur (test de profondeur), depthBufferWriteEnable alternant l'autorisation on ou off d'écriture tampon de profondeur, alphaTestValue est utilisé comme test contre arrivée de valeurs alpha , et alphaTestFunction est une entre ALWAYS, NEVER, EQUAL, NOT_EQUAL, LESS, LESS_OR_EQUAL, GREATER, ou GREATER_OR_EQUAL, qui indique quel type de test alpha est actif. Crée un objet composant qui définit le statut de rendu par-pixel pour les comparaisons de tampon de profondeur et les tests d'alpha.

Les méthodes du RenderingAttributes

`void setDepthBufferEnable(boolean state)`
Tourne on et off le test de tampon de profondeur.

`void setDepthBufferWriteEnable(boolean state)`
Tourne on et off l'écriture au tampon de profondeur.

`void setAlphaTestValue(float value)`
détermine la valeur utilisée pour le test contre l'arrivée de valeurs de source alpha.

`void setAlphaTestFunction(int function)`
Où function est un parmi ALWAYS, NEVER, EQUAL, NOT_EQUAL, LESS, LESS_OR_EQUAL, GREATER, ou GREATER_OR_EQUAL, lequel indique quel type de test alpha est actif. Si function est sur ALWAYS (par défaut), alors le test alpha est alors désactivé.

Les attributs d'Appearence par défaut

Le constructeur par défaut d'Appearance initialise un objet Appearance ayant tous ces références d'attribut fixés sur « null ». Le Tableau 2-1 établit une liste des valeurs par défaut pour ces attributs avec des références « null ».

couleur	`white (1, 1, 1)`
mode de texture d'environment	`TEXENV_REPLACE`
mode de couleur d'environment	`white (1, 1, 1)`
autorisation de test la profondeur	`true`
type d'illumination	`SHADE_SMOOTH`
forme de polygone	`POLYGON_FILL`
autorisation de la transparence	`false`
type de transparence	`FASTEST`
élimination des faces	`CULL_BACK`
taille du point	`1.0`
largeur de ligne	`1.0`
autorisation de l'antialiasing du point	`false`
autorisation de l'antialiasing de la ligne	`false`

Tableau 2-1 Valeurs par défaut d'Attribute.

2.6.4 Exemple: Élimination des faces arrières [back face culling]

Les polygones ont deux faces. Pour de nombreux objets visuels, une seule face du polygone doit être rendue. Pour réduire la puissance de calcul requise pour rendre les surfaces polygonales, le système de rendu peut éliminer les faces inutiles. Le comportement [behavior] d'élimination est défini pour chaque objet dans le composant PolygonAttribute d'Appearance. La face avant d'un objet est la face pour laquelle les sommets sont définis, organisé dans le sens des aiguilles d'une montre.

`TwistStripApp.java` crée un objet visuel « bandeau tordu » et le fait tourner autour de l'axe y. Comme le bandeau tordu tourne, une partie semble disparaître. Les pièces manquantes sont facilement visibles dans la Figure 2-22.

Actuellement, TwistStripApp définit deux objets visuels, chacun ayant la même géometrie - qui est un Twisted strip. Un des objets visuels est rendu en fil de fer [wireframe], l'autre comme une surface solide. Comme les deux objets visuels ont la même position et orientation, l'objet visuel en fil de fer est seulement visible quand la surface n'est pas visible.

Figure 2-22 Le bandeau tordu avec les faces arrières éliminées.

Le mode élimination [culling], cause de la disparition des polygones, n'a pas été déterminé, donc sa valeurs par défaut est CULL_BACK. Les triangles de la surface disparaissent quand leur coté arrière (face arrière) fait face à l'image plate. C'est une caractéristique qui permet au système de rendu d'ignorer les surfaces triangulaires qui ne sont pas utiles, superflues, ou les deux. Pourtant, quelquefois l'élimination des faces arrières est un problème, comme dans TwistStripApp. Le problème a une solution simple : mettre sur off l'élimination [culling]. Pour cela, il faut crée un composant Appearance qui référence un composant PolygonAttribute qui n'autorise pas l'élimination, comme il est décrit dans la Fragment de code 2-10.

```
1. PolygonAttributes polyAppear = new PolygonAttributes();
2. polyAppear.setCullFace(PolygonAttributes.CULL_NONE);
3. Appearance twistAppear = new Appearance();
4. twistAppear.setPolygonAttributes(polyAppear);
5. // plusieurs lignes plus tard, après que le twistStrip TriangleStripArray
6. // soit défini, crée un objet Shape3D avec la sélection des faces sur off
7. // dans le paquet d'apparence, et ajoute le Shape3D au graphe scénique.
8. twistBG.addChild(new Shape3D(twistStrip, twistAppear));
```

Fragment de code 2-10 Désactivation de l'élimination des faces arrières pour le ruban tordu.

Dans la Figure 2-23, la désactivation de l'élimination des faces arrières rempli clairement son rôle. Maintenant tous les polygones sont rendus, sans accorder d'importance à la direction vers laquelle ils se dirigent.

Figure 2-23 Le ruban tordu sans élimination des faces arrières.

La face avant d'un polygone est le coté pour lequel les sommets se prsentent dans un ordre du sens des aiguilles d'une montre. Elle est souvent soumise à la règle de la main droite [right-hand-rule] (voir le lexique). La règle utilisée pour déterminer la face avant d'une face d'un bandeau géometrique (i.e., bandeau triangle, bandeau carré) alterne pour chaque élément dans le bandeau. La Figure 2-24 montre des exemples d'usage de la règle-main-droite pour la détermination des faces avant.

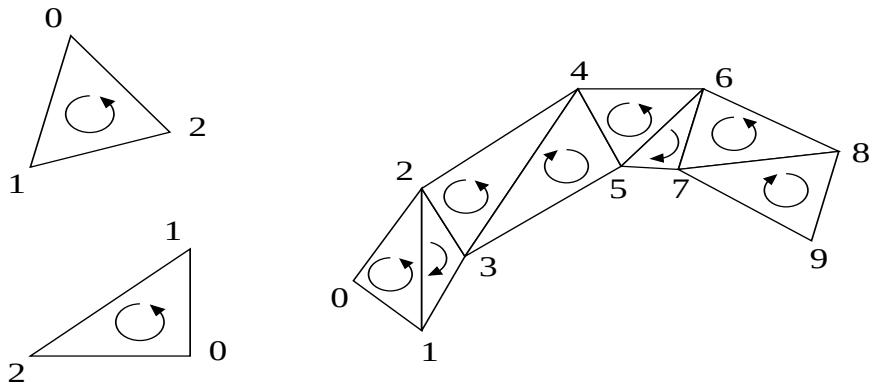


Figure 2-24 Détermination des faces avant sur les polygones et les bandeaux.

2.7 Tests personnels

Ici ce trouvent quelques exercices écrit pour tester et accroitre votre compréhension de la matière présentée dans ce chapitre. Les solutions à certains de ces exercices sont données dans l'Annexe C du Chapitre 0.

1. Essayez par vous même de créer un nouveau Yo-yo en utilisant deux cylindres à la place de deux cones. Prenez comme base `ConeYoyoApp.java`, quels sont les modifications nécessaires ?
2. Un Yo-Yo de deux cylindres peut être crée avec deux objets ruban-carré et quatre objets éventail-triangulaire. Une autre voie est de réutiliser un seul ruban-carré et un seul éventail-triangulaire. La même approche peut être utilisée pour créer le Yo-Yo cône. Quels objets devront former cet objet visuel Yo-Yo ?
3. Le mode d'élimination par défaut des arêtes (mode de cueillage [culling]) est utilisé dans YoyoLineApp.java et YoyoPointApp.java. Changez chacun ou un seul, de ces programmes pour n'afficher aucune arête, puis compilez et lancez le programme modifié. Quelle différence voyez-vous ?