CONSERVATOIRE NATIONAL DES ARTS ET METIERS
PARIS

MÉMOIRE

présenté en vue d'obtenir le

DIPLÔME D'INGÉNIEUR C.N.A.M.

en

INFORMATIQUE

par

Jean-Marie ABISROR

VRML97 sous Java-OpenGL

soutenu le 10 Juin 2002

Jury

Président : Stéphane Natkin (CNAM)

Membres : Pierre Cubaud (CNAM)

Jean-Marc Farinone (CNAM)

Jean-Marc Le Gallic (IGN)

Alain Nowak (ATOS MULTIMEDIA)

Alexandre Topol (CNAM)

Résumé : C'est en mai 1995 que la première version officielle du VRML fut définie, celle-ci ne permettait que la description de scènes immuables, sans possibilité d'animation. VRML 1.0 dérivait du langage graphique *Open Inventor* de *Silicon Graphics* permettant de réaliser des descriptions complètes de scènes 3D incluant la géométrie, les éclairages, les textures. Durant le SIGGRAPH 96 qui se déroulait à la *Nouvelle Orleans*, le *VRML Architecture Group* (V.A.G.) présenta les spécifications officielles de VRML 2.0 et créa le *VRML Consortium* (VRMLC). L'élaboration formelle de VRML 2.0 pour devenir un standard international avait débuté en juin 1996 avec les spécifications préliminaires du *Consortium* pour le VRML 2.0.
La nouvelle version de VRML devait permettre des interactions avec des documents HTML, des scripts Java et inclure des comportements permettant des animations d'objets. En avril 1997, ces nouvelles spécifications furent l'objet d'une soumission à l'ISO sous le nom de VRML97. C'est finalement en décembre 97 que VRML97 fut approuvé par l'ISO.

Alors que HTML est un langage permettant de représenter des documents hypertextes en 2D, VRML permet de modéliser ces informations en 3D. En VRML, une scène 3D est constituée par un ensemble d'objets appelés nœuds ayant comme particularité de respecter une hiérarchie dont l'abstraction est un graphe sans cycle, encore appelé graphe de scène. VRML n'est pas un langage de programmation mais un langage descriptif : l'implémentation des différents nœuds n'est pas à la charge de l'utilisateur mais d'un navigateur spécifique.

Le rendu de la scène est assuré ici par OpenGL, une bibliothèque graphique très évoluée et totalement portable offrant de nombreuses ressources aux programmeurs qui souhaitent élaborer un moteur 3D. Elle reçoit des ordres de tracé de primitives graphiques (facettes,etc.) directement en 3D, une position de caméra, des lumières, etc. Ensuite, OpenGL prend en charge les changements de repère, la projection en perspective à l'écran, l'élimination des parties cachées, l'interpolation des couleurs, le tracé ligne à ligne des faces pour en faire des pixels. OpenGL est conçue comme une interface rationnelle, indépendante du matériel, qu'il est possible d'implémenter sur de nombreuses plate-formes matérielles différentes. OpenGL ne propose pas de commandes de haut niveau permettant de décrire des modèles d'objets tridimensionnels. Il faut construire le modèle à partir d'un jeu restreint de formes géométriques primitives : des points, des lignes et des polygones.

L'objet de ce mémoire a été, à partir d'un logiciel capable de gérer la hiérarchie des nœuds VRML 2.0 en java, appelé *CyberVRML97*, et d'un logiciel permettant l'utilisation des fonctions OpenGL en Java au travers de *bindings* Java-OpenGL, appelé *GL4Java*, d'écrire une interface Java (non exhaustive) entre VRML et OpenGL, permettant d'obtenir un navigateur VRML écrit entièrement en Java. Pour réaliser cela, il a fallu traduire la sémantique des nœuds VRML en utilisant des fonctions OpenGL adéquates. Par exemple, une boite 3D décrite par un nœud VRML *Box* très simple traduit dans sa sémantique la présence des six faces d'un cube. Dans la syntaxe d'OpenGL, il conviendra dans ce cas précis de décrire ces six faces à l'aide de polygones et de préciser l'orientation de la lumière pour que toutes les faces de la figure soient potentiellement visibles. Un module d'animation capable de prendre en charge les scènes animées a été développé. Une animation pouvant être déclenchée par un clic de souris, il a été nécessaire d'implanter les objets de type *Sensor* de VRML97, sortes de capteurs, en utilisant le mécanisme de sélection d'OpenGL, qui permet de contrôler la sélection d'un objet 3D à l'aide du pointeur de la souris.

Mots-clés : VRML, OpenGL, Graphe de scène, Navigation 3D, Animation 3D

KeyWords : VRML, OpenGL, scene graph, 3D browsing, 3D animation

REMERCIEMENTS

J'ai effectué mon stage au sein de l'équipe *Multimédia et Interaction Homme Machine* (M.I.H.M.) dans le laboratoire du C.E.D.R.I.C. (Centre d'Etudes et Recherche en Informatique du CNAM) [CED02] sous la responsabilité de Pierre Cubaud, Maître de Conférences et Alexandre Topol, doctorant. La durée du stage a été de huit mois, celui-ci ayant débuté courant mars 2000. Je les remercie tous les deux ainsi que Alain Nowak (de la Société Atos Origin Multimédia) et Jean-Marc Farinone (CNAM) pour leur aide précieuse qui m'a permis de mener à terme ce mémoire.
Je tiens également à remercier Satoshi Konno et Sven Goethel pour la mise à disposition gracieuse des codes sources de leur logiciel respectif. Enfin, je remercie les membres du jury pour avoir accepté de participer à ma soutenance.

SOMMAIRE

INTRODUCTION

L'arrivée du *World Wide Web* en 1989 offrait la possibilité de présenter des documents hypertextes au format HTML[1] à l'aide de navigateurs dédiés tels que MOSAIC[2], permettant ainsi à l'internaute de naviguer à travers la toile mondiale. Ces documents HTML comportaient alors du texte, des images et du son. A cette époque, la 3D n'existait pas sur le *Web*. C'est en 1994 que deux chercheurs américains de la Société *Intervista*, Mark Piesce et Tony Parisi, eurent l'idée de créer le premier navigateur 3D appelé *Labyrinth*. C'est Tim Berners-Lee[3] qui les invita la même année à la première Conférence sur le *World Wide Web* à Genève. Il fut alors décidé de définir un langage commun pour décrire des scènes en trois dimensions. L'appellation VRML, pour *Virtual Reality Markup Language*, fut adoptée lors de cette conférence (par la suite, le terme *Markup* fut remplacé par *Modeling* pour marquer la différence de syntaxe entre VRML et les langages de type SGML[4]). Brian Behlendorf du magazine *Wired* et Mark Piesce établirent une liste de diffusion afin de faciliter les discussions relatives à ce travail. En moins d'un mois, il y eut plus d'un millier d'adhérents. L'ensemble des besoins pour VRML était l'indépendance vis-à-vis des plate-formes, la capacité d'évolution, à savoir l'utilisation possible dans tous les logiciels 3D et visualiseurs, la possibilité de charger des scènes localisées sur le *Web*, le fonctionnement sur un réseau à faible débit. C'est en mai 1995 que la première version officielle du VRML fut définie, celle-ci ne permettait que la description de scènes immuables, sans possibilité d'animation. VRML 1.0 dérivait du langage graphique *Open Inventor* de *Silicon Graphics* [SGI01] permettant de réaliser des descriptions complètes de scènes 3D incluant la géométrie, les éclairages, les textures. Des extensions furent apportées pour gérer l'accès au *Web*. Par la suite, quelques experts de la liste de diffusion constituèrent le *VRML Architecture Group* (V.A.G.). C'est au cours du Congrès d'août 1995 que l'on y définit les grandes lignes du VRML 2.0. et que le VAG changea son optique de conception technique pour une approche orientée objet. Les objectifs de la nouvelle version de VRML étaient :

- être accessible à des non-programmeurs,
- permettre des interactions avec des documents VRML 1.0, HTML, des scripts Java,
- constituer un standard libre d'utilisation,
- inclure des comportements permettant des animations d'objets.

Un appel à contributions pour la nouvelle version de VRML fut lancé en février 1996 par le V.A.G. Six propositions furent retenues par celui-ci :

- *Moving Worlds*, Silicon Graphics et autres,
- *Active VRML*, Microsoft,
- *Out of this World*, Apple,
- *The Power of Dynamic World*, German National Research Center for Information Technology et autres,

---

[1] *Hypertext Markup Language*

[2] Développé en 1993 par les Ingénieurs de la Société NCSA (*National Center for Supercomputer Applications*)

[3] Inventeur du langage HTML

[4] *Standard Generalized Markup Language*

- *Holoweb*, Sun Microsystems
- *Reactive Virtual Environment*, IBM Japon.

Ce fut encore la proposition de Silicon Graphics qui fut retenue par le V.A.G. , qui proposa ensuite d'élaborer une méthode formelle pour le choix des spécifications de VRML 2.0, et c'est de nouveau *Silicon Graphics* qui fut retenue. Le projet *Moving Worlds*, avec la contribution de *Sony Research*, *Mitra* et beaucoup d'autres permit de créer les spécifications du VRML 2.0. En août 1996, durant le SIGGRAPH 96 qui se déroulait à la *Nouvelle Orleans*, le VAG présenta les spécifications officielles de VRML 2.0 et créa le *VRML Consortium* (VRMLC). L'élaboration formelle de VRML 2.0 pour devenir un standard international avait débuté en juin 1996 avec les spécifications préliminaires du *Consortium* pour le VRML 2.0. Les représentants du VAG et du JTC/SC24 (*Joint Technical Commitee* , ISO/IEC, Infographie et traitement d'images) se rencontrèrent à Kyoto, Japon, et signèrent un accord de coopération. En avril 1997, ces nouvelles spécifications furent l'objet d'une soumission à l'ISO[5] sous le nom de VRML97. En août 1997, au cours du SIGGRAPH 97, VRMLC rendit publiques les spécifications. C'est finalement en décembre 97 que VRML97 fut approuvé par l'ISO.

L'objet de cette étude est la construction d'un navigateur VRML écrit en Java pour le Centre d'Etudes et Recherches en Informatique (CEDRIC) [CED01]. Bien qu'il existe déjà des navigateurs VRML sur le marché, ceux-ci ne constituent pas des outils de recherche, les codes source de ces logiciels n'étant pas disponibles gratuitement (pour la plupart, ils ne sont pas accessibles du tout), interdisant toute possibilité d'évolution ou d'apprentissage du savoir faire en terme d'implémentation d'objets VRML. Les plus connus d'entre eux sont deux navigateurs disponibles sous forme de plugins de navigateurs HTML, à savoir *Cosmo Player* et *WorldView*, de la Société Platinium Technology [PLAT02]. D'autre part, ces navigateurs ne sont généralement pas multi plate-forme. Il existe par ailleurs un navigateur VRML développé en C, ayant fait l'objet d'un mémoire d'Ingénieur CNAM par Alexandre TOPOL [TOP01]. La structure interne ne permettait pas d'ajouter aisément des fonctionnalités telles que les animations, prévues dans la norme VRML97. Par ailleurs, le choix s'est porté ici sur le langage Java. A la suite de diverses recherches, j'ai choisi d'utiliser comme point de départ deux logiciels écrits en Java, dont les codes source sont libres d'utilisation et disponibles gratuitement sur le réseau Internet. Nous détaillerons plus loin la structure de ces logiciels.
Le premier d'entre eux, *CyberVRML97*, permet de gérer au sein d'un *Bloc des Données*, ce que l'on appelle un graphe de scène VRML, c'est à dire la hiérarchie des objets ou nœuds de VRML. Le travail de développement le plus important, objet du présent mémoire, a consisté en l'extension de ce logiciel par l'ajout de classes Java appropriées. Celles-ci permettront alors de gérer ce que nous nommerons un *Bloc Affichage* et un *Bloc Animation*. Le *Bloc Affichage* permet, à partir de la syntaxe des nœuds VRML les plus importants, de traduire la sémantique de ces nœuds sous une forme compréhensible dans le contexte de la bibliothèque OpenGL. Par exemple, une boite 3D décrite par un nœud VRML *Box* très simple traduit dans sa sémantique la présence des six faces d'un cube. Dans la syntaxe d'OpenGL, il conviendra dans ce cas précis de décrire ces six faces à l'aide de polygones et de préciser l'orientation de la lumière pour que toutes les faces de la figure soient potentiellement visibles. Face à cette traduction, il n'est pas toujours évident de trouver une correspondance exacte entre la spécification VRML d'un objet et sa représentation par OpenGL. Le *Bloc Animation*, quant à

[5] *International Standards Organisation*

lui, est capable de prendre en charge les scènes animées. Une animation pouvant être déclenchée par un clic de souris, il a été nécessaire d'implanter les objets de type *Sensor* de VRML97, sortes de capteurs, en utilisant le mécanisme de sélection d'OpenGL, qui permet de contrôler la sélection d'un objet 3D à l'aide du pointeur de la souris. Les animations utilisant également le paramètre temps, il a été nécessaire de mettre en œuvre un capteur particulier de VRML97 appelé *TimeSensor*, permettant de générer des tops d'horloge à intervalles réguliers.
Le deuxième logiciel, *GL4Java*, conçu à l'aide d'un mécanisme de lien entre Java et OpenGL, appelé *binding* Java-OpenGL, sera principalement chargé du rendu des scènes VRML en communication étroite avec la bibliothèque graphique OpenGL. Il n'a pas été nécessaire de rajouter des fonctionnalités à ce logiciel mais uniquement d'en réaliser l'intégration de la structure de ses classes dans celles du nouveau développement. Tout a été mis en œuvre pour que l'architecture résultante possède les caractéristiques d'un logiciel orienté objet.
L'avantage de cette architecture modulaire réside dans une séparation clairement identifiée au sein du logiciel des trois fonctionnalités suivantes :

- gestion du graphe de scène VRML,
- gestion de l'interface entre VRML et OpenGL,
- gestion du rendu des scènes 3D.

Nous débuterons cette étude par une présentation succincte du langage descriptif VRML à l'aide de quelques exemples de base de navigation en trois dimensions à travers des scènes VRML simples. Nous verrons alors des concepts fondamentaux de la norme VRML97. Ensuite, je présenterai les aspects les plus importants de la bibliothèque OpenGL et de l'utilisation de Java à travers les *bindings* Java-OpenGL. Ces éléments techniques parcourus, nous pourrons voir plus en détails la conception du navigateur, la façon dont ont été implémentées les animations, et quelques expérimentations à travers des scripts Java et le multi-fenêtrage.

En conclusion, nous verrons quelle distance reste à franchir par rapport à la norme VRML97, puis les avantages et limites de l'utilisation du langage Java dans cette architecture.

Chapitre 1 – PRESENTATION DE VRML97

1.1 – DEFINITION DE VRML

Alors que HTML est un langage permettant de représenter des documents hypertextes en 2D, VRML permet de modéliser ces informations en 3D. En VRML, une scène 3D est constituée par un ensemble d'objets appelés nœuds. Ces nœuds ont comme particularité de respecter une hiérarchie dont l'abstraction est un graphe sans cycle, encore appelé graphe de scène. VRML n'est pas un langage de programmation mais un langage descriptif. En effet, l'implémentation des différents nœuds de la scène 3D (formes géométriques, lumière, etc.) n'est pas à la charge de l'utilisateur mais d'un navigateur spécifique.
Dans la pratique, on réalise la construction d'une scène VRML en créant simplement un fichier au format texte ayant une extension wrl. Le contenu de celui-ci devra être conforme à la syntaxe VRML [VRM97] et sera soumis au navigateur qui chargera la scène dans son moteur graphique pour en effectuer un rendu tridimensionnel.

1.2 EXEMPLES DE NAVIGATION ET SCENES 3D

1.2.1 – FIGURES 3D ELEMENTAIRES

PRODUCTION D'UNE SIMPLE BOITE

Voici le fichier VRML correspondant à l'affichage d'une simple boîte (Box) :

```
#VRML V2.0 utf8
# Shape : Box
# Une boite verte
Shape {
     appearance Appearance {
          material  Material {
               ambientIntensity 0.4
               diffuseColor  0.3 1.0 0.3
          }
     }
     geometry Box {
          size   2 3 4
     }
}
```

- Le bloc *Shape* permet d'introduire une forme dans une scène
- Le bloc *Appearance* permet de définir l'aspect visuel de la forme
- Le bloc *Material* permet de modifier l'aspect visuel de la forme associée
- Le bloc *geometry* permet de définir le type de forme (ici une boite, référencée *Box*)
- Le bloc *Box* permet de définir un parallélépipède. Il ne contient qu'un seul champ (*size*) permettant de définir une largeur, une hauteur et une profondeur.

Résultat[6] :

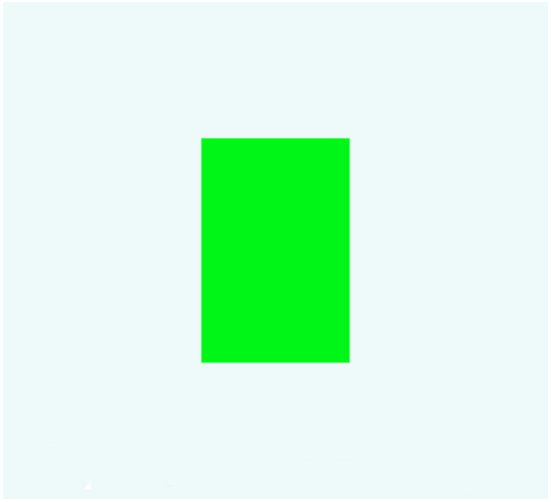


Fig. 1 – Simple Boite

Pour positionner l'objet différemment au démarrage, il faut utiliser un attribut de transformation :

```
#VRML V2.0 utf8
   # Shape : Box
   # Une boite jaune
Transform {

          translation 1 1 -1.5
          rotation 0.7 0.7 -0.12 0.49
          children [
                    Shape {
                              appearance Appearance {
                                        material  Material {
                                                  diffuseColor
1 1 0
                                        }
                              }
                              geometry Box {
                                   size   2.0 5.0 5.0
                              }
                    }
          ]

}
```

[6] Les figures présentées dans ce chapitre ont été générées à l'aide du navigateur VRML développé dans le cadre de ce mémoire

- Le bloc *Transform* permet de positionner, de mettre à l'échelle et de faire effectuer des rotations à un bloc ou à un ensemble de blocs.
- Le champ *Translation* permet de positionner un objet par rapport au point 0 0 0.
- Le champ *Rotation* permet d'effectuer une rotation en donnant un vecteur x,y,z et un angle en radians.
- Le champ *Children* contient le ou les blocs subissant la transformation.

Résultat :

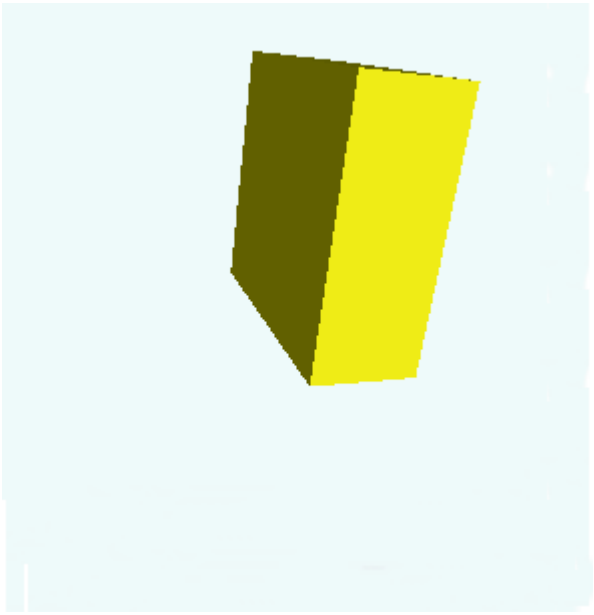


Fig. 2 – Simple Boite avec *Transform*

PRODUCTION D'UN CONE

Voici le fichier VRML correspondant à l'affichage d'un cône :

```
#VRML V2.0 utf8
#Shape : Cone
# Un cone rouge
     Shape {
          appearance Appearance {
               material  Material {
                    ambientIntensity 0.4
                    diffuseColor  1.0 0.3 0.3
                    }
               }
     geometry Cone {
          bottomRadius  2.2
          height 2.5
     }
}
```

Résultat :

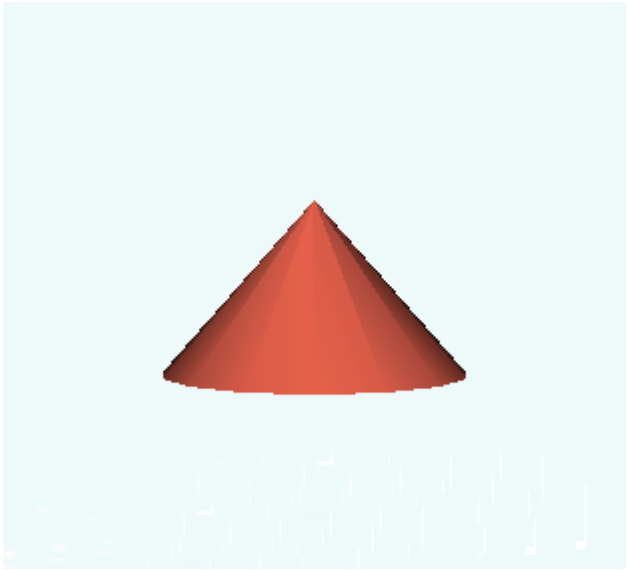


Fig. 3 - Cone

1.2.2 – COMBINAISONS D'OBJETS ELEMENTAIRES

FIGURES GEOMETRIQUES

Cette scène[7] est composée de trois figures géométriques de base, ayant chacune un nœud *Transform* permettant de les positionner.

Résultat :

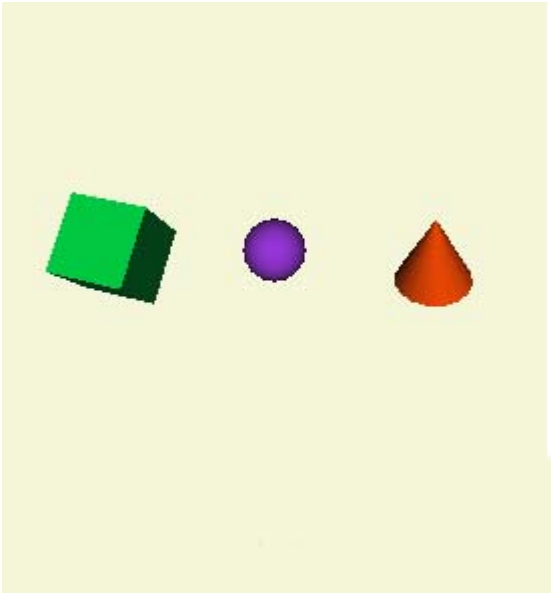


Fig. 4 – Combinaison d'objets

REPRESENTATION D'UN SOFA

La scène suivante[8] a été construite à partir d'une série de nœuds *Transform* permettant de positionner tous les composants de l'objet final.
Résultat :

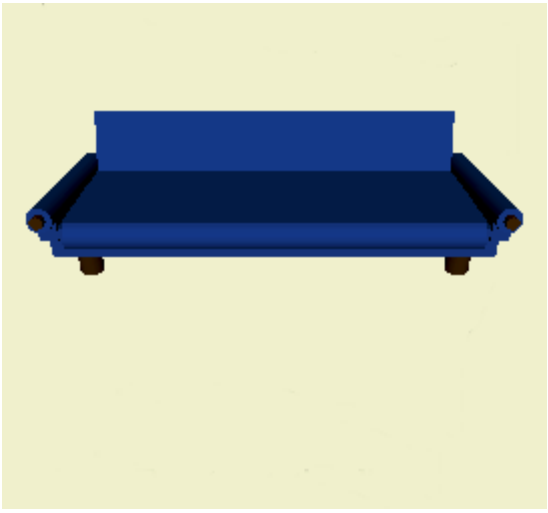


Fig. 5 – Représentation d'un sofa

[7] Voir scène en Annexe 1
[8] Voir scène en Annexe 2

1.2.3 – FIGURES A BASE DE POLYEDRES

OBJETS CONTENANT UNE COULEUR PAR FACE

Fichier VRML correspondant à l'affichage d'une simple boîte décrite à l'aide de polyèdres (ou facettes) avec une couleur par face[9] :

```
#VRML V2.0 utf8
#- KDO - Cours VRML : IndexedFaceSet
NavigationInfo {

type "EXAMINE"
headlight FALSE
}
     Shape {
          appearance Appearance {
               material Material {}

          }
     geometry IndexedFaceSet {
          solid FALSE
               coord Coordinate {
                    point [
                         -2 -2  2, -2 2  2, 2  2  2, 2 -2  2,
                         -2 -2 -2, -2 2 -2, 2  2 -2, 2 -2 -2
                    ]
               }
               coordIndex [
                    0 1 2 3 -1, 7 6 5 4 -1,
                    4 5 1 0 -1, 3 2 6 7 -1,
                    3 7 4 0 -1, 1 5 6 2
               ]
               colorPerVertex FALSE
               color Color {
                    color [
                         0 0 1, 0 1 0, 0 1 1, 1 0 0,
                         1 0 1, 1 1 0, 1 1 1, 0 1 0
                    ]
               }
     }
}
```

[9] Pour les explications détaillées de cette scène, voir page 96

Résultat :

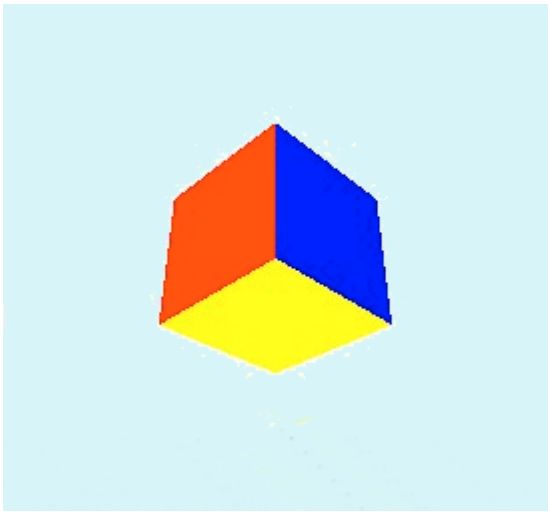


Fig . 6 – Cube avec une couleur par face

OBJETS CONTENANT UNE COULEUR PAR SOMMET

Le fichier VRML est identique au précédent, la seule différence est :

```
colorPerVertex TRUE
```

Résultat :

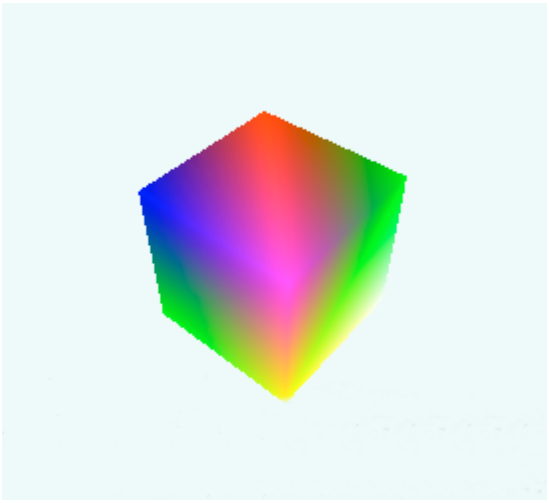


Fig. 7 – Cube avec une couleur
Interpolée aux sommets

Nous verrons en détail le fonctionnement de tels objets au chapitre 3 (Classe *IndexedFaceSetNodeOpenGL*).

1.2.4 – SCENES DYNAMIQUES

PENDULE EN MOUVEMENT

Cette scène VRML comporte un script java. Cet exemple est présenté en détails dans la partie animations (Page 38).

Voici une image du résultat :

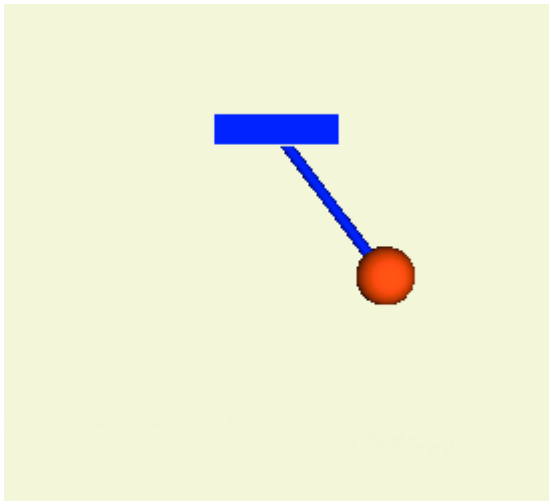


Fig. 8 – Pendule en mouvement

OBJETS 3D SENSIBLES A LA SOURIS

Cette scène VRML[10] comporte également un script Java (cf. animations Par. 2.1.2).
Le cube, sensible aux mouvements du pointeur de la souris, change de couleur au passage de celui-ci, tandis que la sphère réagit aux clics de la souris.

Voici une image du résultat :

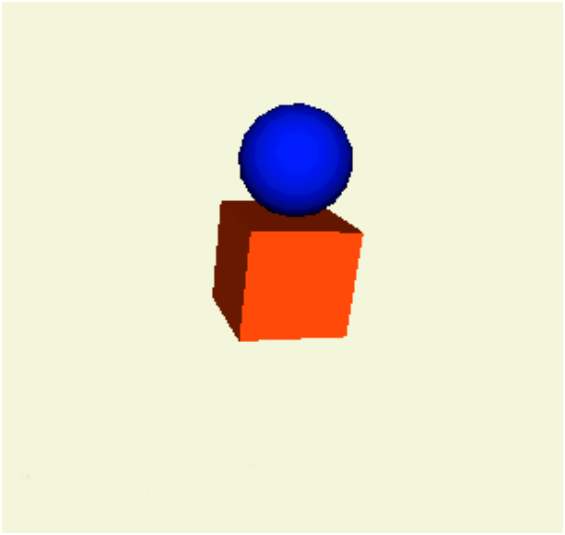


Fig. 9 – Implémentation du capteur TouchSensor

[10] Voir scène VRML et scripts Java en Annexe 3

1.3 - PRESENTATION DE LA NORME VRML97

Comme nous venons de le voir avec les exemples précédents, un fichier VRML consiste en plusieurs composants :

- l'en-tête (Header),
- les prototypes,
- le Graphe de Scène,
- le routage des événements.

Le contenu de ce fichier est utilisé pour la présentation et l'interaction par un programme navigateur.

EN-TETE

Pour une identification aisée des fichiers VRML, ils devront commencer par :

#VRML V2.0 <type d'encodage> [commentaire] <terminaison de ligne>

L'entête est une ligne unique qui identifie le fichier comme étant un fichier VRML, généralement au format utf-8. Il peut aussi contenir des informations supplémentaires, considérées comme des commentaires.

GRAPHE DE SCENE

Le graphe de scène contient des nœuds qui décrivent des objets et leurs propriétés. Il contient des descriptions géométriques groupées hiérarchiquement pour fournir une représentation audio/visuelle des objets, ainsi que les nœuds qui participent à la génération d'événements et au mécanisme de routage.

PROTOTYPES

Les prototypes permettent l'extension par l'utilisateur des types de nœuds VRML. Des définitions de prototypes peuvent être incluses dans les fichiers dans lesquels ils se trouvent ou bien définis de façon externe.

Les prototypes peuvent être définis en terme de nœuds VRML différents ou en utilisant un mécanisme d'extension spécifique au navigateur. Alors que la norme ISO/IEC 14772 a un format standard pour identifier de telles extensions, leur implémentation dépend du navigateur.

LE MECANISME DE ROUTAGE DES EVENEMENTS

Certains nœuds VRML génèrent des événements en réponse aux changements de l'environnement ou aux interactions de l'utilisateur. Le routage d'événements fourni un mécanisme séparé de la hiérarchie du graphe de scène à travers lequel ces événements peuvent être propagés pour rendre effectifs les changements dans d'autres nœuds. Une fois générés, les événements sont envoyés à leur destination routés dans l'ordre d'appel et traités par le nœud récepteur. Ce procédé peut changer l'état du nœud, générer des événements additionnels ou changer la structure du graphe de scène.

Les nœuds scripts permettent de traiter les événements par l'exécution de routines écrites par l'utilisateur. Un événement reçu par un nœud script entraîne l'exécution d'une fonction dans un script, qui est habilitée à envoyer des événements à travers le mécanisme normal de routage d'événements ou bien en envoyant des événements directement à n'importe quel nœud que le nœud script peut référencer.

Les scripts peuvent aussi ajouter ou supprimer dynamiquement des routes et de cette façon changer la topologie de routage par événement.

Le modèle d'événement idéal traite tous les événements instantanément dans l'ordre où ils sont générés. Ce modèle utilise une estampille : c'est un dispositif conceptuel utilisé pour décrire le flot chronologique du mécanisme d'événement. Il assure que des résultats déterministes peuvent être obtenus par des implémentations qui font correspondre des délais de traitement et des interactions asynchrones avec des dispositifs externes. En second lieu, les estampilles sont aussi utilisées dans les nœuds *Script* pour permettre aux événements d'être traités en fonction de l'ordre des actions de l'utilisateur ou le temps écoulé entre les événements.

PRESENTATION ET INTERACTION

L'interprétation, l'exécution et la présentation des fichiers VRML sont typiquement prises en charge par un navigateur qui affiche les formes et émet des sons présents dans le graphe de scène. Cette présentation est connue sous le nom de monde virtuel et elle est parcourue par une entité humaine ou mécanique, l'utilisateur. Le monde virtuel est affiché à partir d'un point de vue particulier ; cette position et orientation sont appelés visualisateur. Le navigateur fournit des paradigmes de navigation (tels que marcher ou voler) qui permettent à l'utilisateur de pouvoir se déplacer à travers le monde virtuel.

En plus de la navigation, le navigateur fournit un mécanisme permettant à l'utilisateur d'interagir avec le monde par l'intermédiaire des nœuds *Sensor* dans la hiérarchie du graphe de scène. Les *sensors* répondent à l'interaction de l'utilisateur avec des objets géométriques dans le monde virtuel, le mouvement de l'utilisateur à travers le monde ou l'écoulement du temps. La présentation visuelle d'objets géométriques dans un monde VRML suit un modèle conceptuel conçu pour ressembler aux caractéristiques physiques de la lumière. Le modèle VRML de l'éclairement décrit comment les propriétés d'apparence et de lumière dans le monde virtuel sont combinées pour produire des couleurs.

La figure 10 illustre un modèle conceptuel d'un navigateur VRML. Celui-ci est représenté comme une application de présentation qui accepte les entrées de l'utilisateur sous la forme d'une sélection de fichier (explicite et implicite), des mouvements provenant de l'interface utilisateur (par exemple manipulation et navigation utilisant un dispositif d'entrée comme la souris).

Les trois principaux composants du navigateur sont :

- *Parser*,
- Graphe de Scène,
- Présentation Audio/Visuelle.

Le *parser* lit le fichier VRML et crée le graphe de scène qui consiste en une hiérarchie de transformations (les nœuds) et le graphe des Routes. Le graphe de scène inclus aussi le moteur d'exécution qui traite les événements, lit le graphe des Routes et fait des modifications sur la hiérarchie de Transformation (les nœuds).

L'entrée utilisateur affecte généralement des capteurs et la navigation est donc liée au composant du graphe des Routes (sensors) et au composant de présentation Audio/Visuelle (navigation). Ce composant exécute le rendu visuel et audio de la hiérarchie de transformation qui retourne vers l'utilisateur.

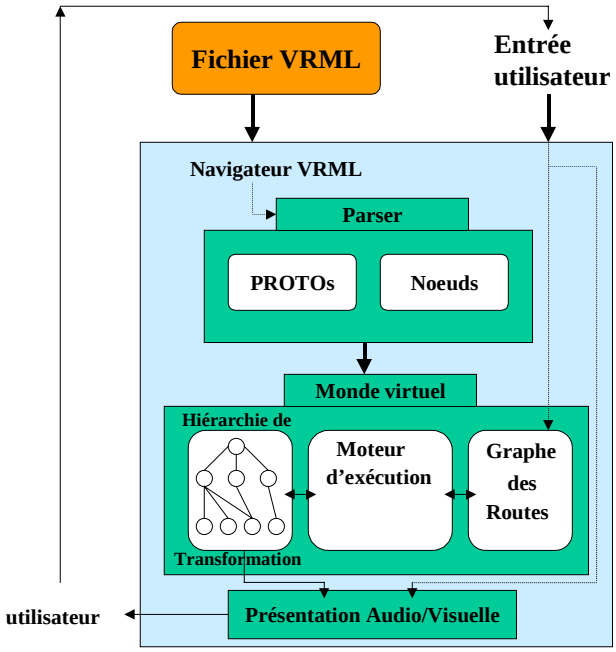


Fig. 10 - Modèle conceptuel d'un navigateur VRML

1.3.1 – LES INSTRUCTIONS VRML

Après l'entête obligatoire, un fichier VRML peut contenir l'une des combinaisons suivantes :

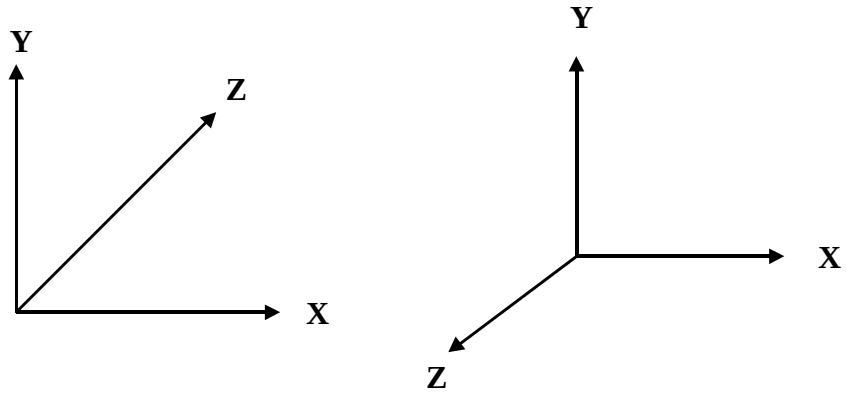
a) un certain nombre d'instructions *PROTO* ou *EXTERNPROTO*,
b) un certain nombre de nœuds racine,
c) un certain nombre d'instructions *USE*,
d) un certain nombre d'instructions *ROUTE*.

Nous ne décrirons pas en détails la syntaxe de chacun des nœuds qui se trouve dans la norme VRML97 [VRM97].

1.3.2 – UNITES STANDARDS ET SYSTEME DE COORDONNEES

La norme ISO/IEC 14772 définit l'unité de mesure du système de coordonnées spatial en mètre. Les angles sont définis en radian, le temps en seconde et les couleurs sont représentées dans le système R.V.B. (RGB).

Le système de coordonnées du repère spatial est un repère cartésien orthonormé dans le sens direct (main droite) à trois dimensions. Par défaut, l'observateur a l'axe des X vers la droite, l'axe des Y en haut et l'axe des Z vers lui :



Repère "main gauche" Repère "main droite"

Fig. 11 - Les deux types de repères en 3D

1.3.3 – STRUCTURE HIERARCHIQUE DU GRAPHE DE SCENE

Un fichier VRML contient zéro ou plusieurs nœuds racine [VRM97]. Les nœuds racine d'un fichier VRML sont les nœuds définis par l'instruction de déclaration du nœud ou les instructions *USE* qui ne sont pas contenus dans d'autres nœuds ou dans des instructions *PROTO*.
Un fichier VRML contient un graphe orienté sans cycle[11]. Les instructions de nœud peuvent contenir des instructions de champ *SFNode* ou *MFNode* qui, à leur tour, contiennent des instructions de nœud (ou *USE*). Cette hiérarchie de nœuds est appelée graphe de scène. Chaque arc dans le graphe de A vers B signifie que le nœud A a un champ *SFNode* ou *MFNode* dont la valeur contient directement le nœud B.
Les descendants d'un nœud sont tous les nœuds dans ses champs *SFNode* ou *MFNode* aussi bien que les descendants de ces nœuds. Les ancêtres d'un nœud sont tous les nœuds qui ont ce nœud comme descendant.

[11] Configuration de sommets et d'arcs les reliant, ayant un sens prédéfini et tels que l'on ne puisse pas revenir au sommet initial

1.3.4 – HIERARCHIE DE TRANSFORMATION

La hiérarchie de transformation est représentée par tous les nœuds racine et les descendants des nœuds racine qui sont considérés comme ayant une ou plusieurs localisations particulières dans le monde virtuel.
Le langage VRML contient la notion de systèmes de coordonnées locaux, définis en terme de transformations à partir des systèmes de coordonnées ancêtres (utilisant les nœuds *Transform* ou *Billboard*). Le système de coordonnées dans lequel les nœuds racines sont affichés est appelé le *système de coordonnées du monde*.
La tâche d'un navigateur VRML est de présenter un fichier VRML à l'utilisateur en lui présentant la hiérarchie de transformation. Celle-ci décrit les parties directement perceptibles du monde virtuel. Certains types de nœuds comme *TimeSensor* ou *PositionInterpolator* ne sont pas affectés par la hiérarchie de transformation.
La hiérarchie de transformation est un graphe orienté sans cycle ; les résultats sont indéterminés si un nœud dans cette hiérarchie est son propre ancêtre.

1.3.5 - SEMANTIQUE DES NŒUDS

INTRODUCTION

Chaque nœud a la caractéristique suivante :

a) Un nom de type. Des exemples sont : *Box, Color, Group,* etc.
b) Zéro ou plusieurs champs qui définissent comment chaque nœud diffère d'autres nœuds du même type. Les valeurs des champs sont enregistrées dans le fichier VRML avec les nœuds.
c) Un ensemble d'événements qu'il peut recevoir et envoyer. Chaque nœud peut recevoir zéro ou plusieurs types d'événement qui auront pour effet de changer l'état du nœud.
d) Une implémentation. L'implémentation de chaque nœud définit comment il réagit aux événements qu'il peut recevoir quand il génère des événements et son apparence visuelle ou audio dans le monde virtuel (s'il y en a une). Le standard VRML définit la sémantique des nœuds internes (c'est à dire les nœuds dont l'implémentation est fournie par le navigateur VRML). L'instruction PROTO peut être utilisée pour définir de nouveaux types de nœuds avec des comportements, par ceux d'autres nœuds.
e) Un nom. Les nœuds peuvent être nommés. Ceci est utilisé par d'autres instructions pour référencer une instanciation spécifique d'un nœud.

SEMANTIQUE DE DEF ET USE

Un nœud nommé grâce au mot clé *DEF* peut être référencé par son nom plus loin dans le même fichier avec les instructions *USE* ou *ROUTE*. L'instruction *USE* ne crée pas de copie d'un nœud. A la place, le même nœud est inséré dans le graphe de scène une seconde fois. L'utilisation de l'instance d'un nœud plusieurs fois est appelée instanciation. Si le même nom est attribué à plusieurs nœuds, chaque instruction *USE* référence le nœud le plus proche ayant ce nom qui le précède soit dans le fichier VRML ou dans la définition d'un prototype.

FORMES ET GEOMETRIES

Un objet au sens VRML est composé de deux éléments principaux :
Une géométrie et une apparence. Ce couple définit ce que l'on appelle une forme en VRML. Pour pouvoir utiliser un objet, on est donc obligé de passer par le nœud correspondant, le nœud *Shape*, qui associe un nœud *Geometry* avec des nœuds qui définissent l'apparence de cette géométrie. Les nœuds *Shape* feront partie de la hiérarchie de transformation pour avoir n'importe quel résultat visible. Un nœud *Shape* contient exactement un nœud *Geometry* dans son champ *geometry*.
Les types de nœud suivants sont des nœuds *Geometry* :
Box, Cone, Cylinder, ElevationGrid, Extrusion, IndexedFaceSet, IndexedLineSet, PointSet, Sphere et *Text.*

Les nœuds *Shape* peuvent préciser un nœud *Appearance* qui décrit les propriétés d'apparence (matériel et texture) qui doivent être appliquées à la géométrie de la forme.
Un nœud *Material* peut être spécifié dans le champ *material* du nœud *Appearance*.

Les nœuds *ImageTexture, PixelTexture* et *MovieTexture* peuvent être spécifiés par le champ *texture* du nœud *Appearance*.

Les champs *material* et *texture* contiennent des nœuds que nous allons voir immédiatement. Si le champs *material* est nul, aucune lumière n'est prise en compte. Si le champs *texture* est nul, l'objet n'est pas texturé. Maintenant que nous avons les formes géométriques, nous allons donner une apparence aux objets. Il existe deux manières essentielles de modifier l'apparence d'un objet : définir un nœud *Material* (couleur de l'objet, manière dont il a réfléchi la lumière, etc.), et définir une texture (image calquée directement sur l'objet).

Définition d'un nœud *Material* :

```
material Material {
     ambientIntensity 0.2
     diffuseColor   0.8 0.8 0.8
     specularColor 0 0 0
     emissiveColor 0 0 0
     shininess  0.2
     transarency 0
}
```

Les paramètres sont soit des composantes de couleur R.V.B. comprises entre 0 et 1, soit des coefficients également compris entre 0 et 1.

diffuseColor : définit la couleur de l'objet au sens le plus général du terme. Les lumières de la scène influent sur l'apparence de l'objet grâce à cette couleur, en fonction de l'angle de réflexion de la lumière sur l'objet.
ambientIntensity : correspond à l'influence de la lumière ambiante de la scène sur l'objet. La lumière ambiante est aussi produite par les lumières de la scène mais ne dépend pas des diverses réflexions (lumière isotropique).
specularColor : définit la couleur des rayons lumineux réfléchis sur la surface de l'objet et renvoyés vers le point de vue de l'utilisateur. Cela peut servir à faire un effet de réflexion lumineux sur un objet.

emissiveColor : définit une couleur propre à l'objet, tel qu'il apparaît même sans lumière.
shininess : définit la brillance de l'objet.
transparency : définit la transparence de l'objet.

Les formes dont la géométrie est définie comme un nœud *IndexedFaceSet* (ou un nœud *IndexedLineSet* ou *PointSet*) peut contenir un nœud *Color* qui définit une couleur pour chaque face ou sommet (ou ligne ou point). Ces couleurs remplacent alors les couleurs du matériel. Si une forme n'a aucune couleur spécifiée, elle a une couleur par défaut blanc uniforme.

1.3.6 – TRANSFORMATION ET POSITIONNEMENT DES OBJETS

Les objets que nous avons défini peuvent être manipulés par les opérations géométriques usuelles de translation, rotation et homothétie. Il existe un nœud du langage permettant de procéder simultanément au positionnement (translation, rotation) et à la transformation (homothétie) des formes géométriques :

```
Transform {

     center 0 0 0
     translation 0 0 0
     rotation 0 0 1 0
     scale 1 1 1
     scaleOrientation 0 0 1 0
     children [ ]

}
```

Ce nœud fait partie des nœuds de groupe : ils peuvent contenir d'autres nœuds du langage dans leurs paramètres. Le champ *children* peut contenir d'autres nœuds, par exemple des nœuds *Shape* sur lesquels les transformations géométriques seront appliquées.

translation : déplace l'objet selon l'axe X, Y et Z
rotation : (x, y, z, t) tourne de l'angle t exprimé en radians autour du vecteur (x, y, z).
scale : homothétie modifiant à l'échelle selon les trois axes du repère défini par *ScaleOrientation*.

1.3.7 – MODELE DE BASE DE LA LUMIERE

Le modèle de couleur et de lumière de VRML est défini en deux parties : les géométries éclairées et non éclairées. Dans le cas non éclairé, le concepteur « peint les murs » avec des couleurs et des textures R.V.B. Dans le cas d'éclairement, ces couleurs et textures sont modulées par l'éclairage. Le modèle d'éclairement en VRML est calculé sur une base par pixel. Cependant, tous les navigateurs courants font des approximations en calculant les couleurs en chaque sommet en interpolant alors entre les couleurs des sommets. Cela introduit alors des artefacts visuels dans le rendu de la scène.

La géométrie sans lumière (sans nœud *Material*) permet au concepteur de contrôler précisément la couleur des objets dans la scène simplement en lui spécifiant comme valeurs R.V.B. des tableaux ou valeurs de texture. Ceci est un cas très important pour la fidélité des couleurs en VRML car la spécification et l'affichage des couleurs est direct : ce que l'utilisateur spécifie est la couleur affichée. La géométrie sans lumière est un espace de conception intéressant car l'auteur a un contrôle complet sur l'apparence.
Dans la géométrie avec lumière, les composants qui étaient simplement peints dans le cas sans lumière sont utilisés pour définir la couleur diffuse du matériel dans le cas avec éclairement. Les couleurs peintes remplacent les couleurs du matériel dans le cas du R.V.B., modulent la couleur du matériel dans le cas avec intensité. S'il n'y a pas de couleurs peintes, alors la couleur dans le matériel définit la couleur diffuse.

Il y a trois types de lumières en VRML :

DirectionalLight est la plus simple des lumières VRML. Elle définit une lumière d'intensité constante dans une direction précise. Un polygone sera illuminé par DirectionalLight si la normale par face forme un angle aigu avec la direction de la lumière (jusqu'à 45 degrés). Si cet angle dépasse 45 degrés, le polygone ne sera plus illuminé.
DirectionalLight représente une source directionnelle. Ce nœud définit une source de lumière illuminant la scène selon des rayons parallèles. La syntaxe est :

```
DirectionalLight {
          direction x y z
          color r v b
          intensity i
          ambientIntensity ai
}
```

PointLight est définie comme une lumière positionnée dans l'espace et rayonnant dans toutes les directions. Le nœud *PointLight* permet d'illuminer tous les objets à l'intérieur du champ *radius*.

```
PointLight {
     intensity 1
     color 1 1 1
     location 0 0 0
     radius 1 0 0
     attenuation 1 0 0
     ambientIntensity 0
}
```

intensity : spécifie l'intensité de la lumière.
color : spécifie sa couleur.
location : spécifie la position de la source.
radius : spécifie la distance en mètres à l'intérieur de laquelle cette source lumineuse a un effet sur les objets.
atténuation : donne les trois coefficients d'une équation du second degré donnant l'atténuation de la lumière en fonction de la distance à l'objet éclairé.
ambientIntensity : spécifie la quantité de lumière ambiante générée par cette source.

SpotLight produit une source lumineuse localisée et directionnelle de type spot. Celle-ci est la plus complexe des lumières VRML. La lumière est formée de telle façon que l'intensité change en fonction de la distance qui sépare l'objet de la lumière et en fonction de la distance par rapport au centre du spot. Le paramètre d'atténuation est le même que pour une source de lumière *PointLight*.

```
SpotLight {
     intensity 1
     direction 0 0 –1
     color 1 1 1
     location 0 0 0
     radius 100
     attenuation 1 0 0
     ambientIntensity 0
     beamWidth 1.570
     cutoffAngle 0.78
}
```

direction : donne la direction ou l'axe du cône.
cutOffAngle : détermine l'angle d'ouverture du cône à l'intérieur duquel la lumière est émise. Plus on se rapproche des bords du cône, plus la lumière est atténuée grâce au champ *attenuation*. Inférieur ou égal à 180° soit environ 1.57 radians.
beamWidth : détermine un cône intérieur dans lequel la lumière n'est pas atténuée (elle ne commencera à l'être qu'après franchissement de ce sous cône)

1.3.8 – VRML97 ET LES ANIMATIONS

Le langage VRML97 permet d'animer les objets de deux façons différentes [VER98]:

- directement en VRML, en utilisant le mécanisme d'événements,
- en décrivant des comportements plus fins dans un véritable langage de programmation, par l'intermédiaire des nœuds *Scripts*. Ces scripts (en Java, JavaScript ou VRMLscript) sont interprétés directement par le navigateur VRML car faisant partie de la norme (contrairement à l'E.A.I.[12]).

On peut déclencher une animation par un événement utilisateur récupéré par des capteurs spécifiques (détection d'un clic souris) ou par une horloge. Pour lisser une animation décrite par un ensemble de valeurs, on utilise des interpolateurs.

[12] External Authoring Interface. Permet la réalisation d'applications externes écrites en Java et pouvant communiquer avec la scène VRML au sein d'une même page Web.

LES EVENEMENTS

Un événement est un message qui contient une valeur d'un certain type (le type du champ considéré). Chaque nœud VRML est composé de champs qui peuvent être :

- des événements en entrée (*eventIn*)
- des événements en sortie (*eventOut*)
- des événements en entrée/sortie (*exposedField*)
- des champs en lecture seule (*field*)

On connecte un événement en sortie d'un nœud à un événement en entrée du même type d'un autre nœud par une instruction *ROUTE*. Pour pouvoir créer une *ROUTE*, il faut avoir nommé les nœuds concernés (cf. *DEF* et *USE*).

LE SYSTEME D'EVENEMENTS

Certains objets ont la possibilité d'émettre et/ou de recevoir des événements (attributs de type *eventIn* et *eventOut*) vers d'autres objets de la scène. Le seul moyen de communiquer avec des composants extérieurs au monde VRML est par l'utilisation du nœud *TouchSensor*, permettant de récupérer les événements émis par la souris [LEG99].

En plus des attributs *eventIn* et *eventOut*, les attributs *exposedField* que possèdent certains objets VRML représentent la contraction deux événements implicites :

- Un événement entrant (*set_attr*), qui permet de modifier la valeur de l'attribut *attr* avec la valeur reçue,
- Un événement sortant (*attr_changed*), qui permet d'émettre la valeur de l'attribut *attr* à chaque modification de celui-ci.

LE ROUTAGE ET LES SCRIPTS

Le routage

L'ensemble des événements est piloté par un système de routage (mots clé *ROUTE …TO …*) qui permet de propager un événement (sortant) d'un objet vers un autre objet (événement entrant). Grâce à ce mécanisme et à la richesse des types de nœuds, il est possible de créer des animations complexes.
La seule manière de modifier la valeur d'un attribut en VRML est de lui envoyer un événement par ce système de routage. L'exemple suivant montre comment on peut allumer un projecteur (nœud *PointLight*) par un clic souris. Dans cet exemple, il nous faut définir une zone sensible au curseur de la souris (nœud *TouchSensor*). Nous avons choisi un cylindre. Cet exemple illustre l'utilisation des attributs *eventIn* et *eventOut*. Le mécanisme est identique avec les attributs *exposedField*.

```
#VRML V2.0 utf8
#- KDO - Cours VRML : ROUTE ... TO ...

NavigationInfo {
   type "NONE"
   headlight TRUE
}
DEF LUM PointLight {
   location 0 1 1
   ambientIntensity 1.0
   intensity 0.5
   color 1 1 0
   on FALSE
}

DEF objet Transform[13] {
          children [
                        Shape {
                   geometry Cylinder {}
                        appearance Appearance {
                             material Material {

     diffuseColor 0.5 0.5 0.5
                                  }
                             }
                   }
     ]
}

DEF capteur TouchSensor {}
ROUTE capteur.isActive TO LUM.set_on[14]
```

Les événements entrant et sortant d'une clause *ROUTE … TO …* doivent impérativement être du même type (sur l'exemple précédent, les événements *isActive* et *on* sont de type booléen.

Il est évidemment possible d'enchaîner les clauses *ROUTE … TO …* et de réaliser des animations beaucoup plus complexes. L'exemple présenté fig. 12 explique la propagation et le déclenchement de plusieurs événements : un clic souris (via l'objet *Touch* de type *TouchSensor*) déclenche la mise en route d'une horloge (objet *Timer1* de type *TimeSensor*[15]). A chaque top d'horloge, l'attribut *fraction* du *TimeSensor* augmente, entraînant la modification de l'attribut *fraction* de l'objet *Rot1* (de type *OrientationInterpolator*). A chaque changement de fraction, l'attribut *value* prend une nouvelle valeur, entraînant la rotation d'une fenêtre (objet *Frame1* de type *Transform*).

---

[13] Le cylindre « objet » est la zone sensible au curseur de la souris (définie dans le même *children* que le noeud *TouchSensor*)

[14] *isActive* est un événement sortant (*eventOut*) de l'objet *TouchSensor* qui émet la valeur *TRUE* lors du clic souris. Ici, l'événement se propage et déclenche un autre événement, entrant, de l'objet *PointLight.* Donc, en cliquant sur le cylindre nommé « objet », l'attribut « on » du nœud LUM devient *TRUE* et la lumière apparaît !

[15] Nœud dont certains attributs changent d'état, en fonction d'un "top d'horloge" dont on peut définir la fréquence

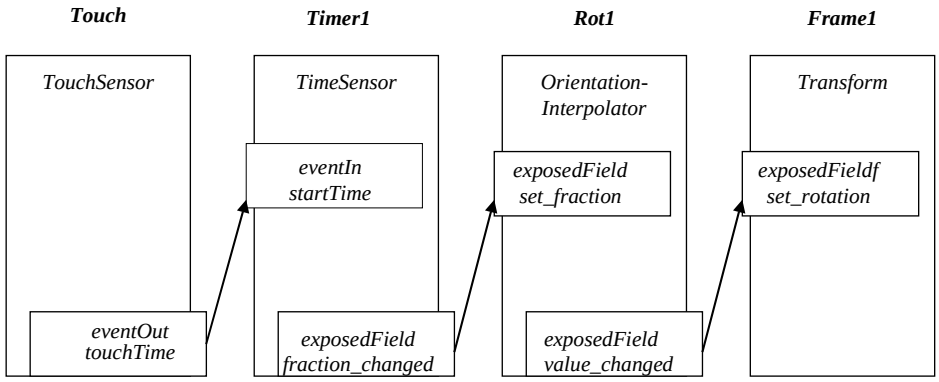


Fig. 12 - Déclenchement d'événements en série par routage

Voici les instruction correspondant à ce routage:

```
ROUTE Touch.touchTime                TO   Timer1.set_startTime
ROUTE Timer1.fraction_changed        TO   Rot1.set_fraction
ROUTE Rot1.value_changed             TO   Frame1.set_translation
```

Les scripts [LEG99]

Pour que l'utilisateur crée ses propres événements, VRML met à sa disposition le nœud *Script*. Comme la plupart des nœuds, le nœud *Script* peut recevoir et émettre des événements. Un événement reçu par un nœud *Script* déclenche l'exécution d'une fonction écrite par l'utilisateur (en Java ou JavaScript). Le système de routage est identique.

Le nœud *Script* fournit donc la possibilité d'inclure des animations plus complètes au sein d'une scène. Il peut également être utilisé pour communiquer avec un autre serveur ou un autre "monde VRML" à travers le réseau.

Un script peut être prototypé, donc exporté dans une scène et instancié.

Le principe est simple : lorsqu'un nœud de type *Script* reçoit un événement, il passe la valeur de l'événement au script Javascript (ou Java) spécifié par l'attribut *url* de ce nœud. Le script peut alors mettre à jour la valeur de l'événement sortant du nœud *Script*.

La fonction définie dans le script reçoit en paramètre la valeur de l'événement lorsque celui-ci est émis. Notons que le nom de la fonction doit être le même que celui de l'événement entrant. Par exemple, si le nœud *Script* contient un événement entrant *touchTime* et un événement sortant *startTime*, le script doit contenir une fonction qui peut faire quelque chose comme :

```
function touchTime (timeValue) {
     startTime = timeValue + 5 ;
}
```

Voici un exemple plus complet de script, qui accentue la couleur d'une forme (par exemple une sphère) au passage de la souris sur cette forme. Le nœud script contient un prototype définissant les attributs nécessaires.

```
PROTO Highlighter[16] [
     eventIn SFBool isActive
     eventOut SFColor color
     field SFColor activeColor 0.8 0.8 0.8
      field SFColor inactiveColor 0.5 0.5 0.5
]
{
  Script {
     eventIn SFBool isActive IS isActive
     eventOut SFColor color IS color
     field SFColor activeColor IS activeColor
     field SFColor inactiveColor IS inactiveColor
url [
      "javascript[17]:
        function isActive (eventValue) {
          if (eventValue == true)
               color = activeColor;
                    else
               color = inactiveColor;
        }   "
    ]
  }
}
```

Dans ce cas, le nœud *Script* fait partie d'un processus plus complet comprenant un nœud *TouchSensor* et une forme (nœud *Transform*). La figure 13 présente le déclenchement des événements.

[16] Valeurs par défaut des couleurs

[17] Si le curseur de la souris est sur l'objet, eventValue vaut TRUE et l'événement émis par le script est la couleur 0.8 0.8 0.8 sinon la couleur vaut 0.5 0.5 0.5

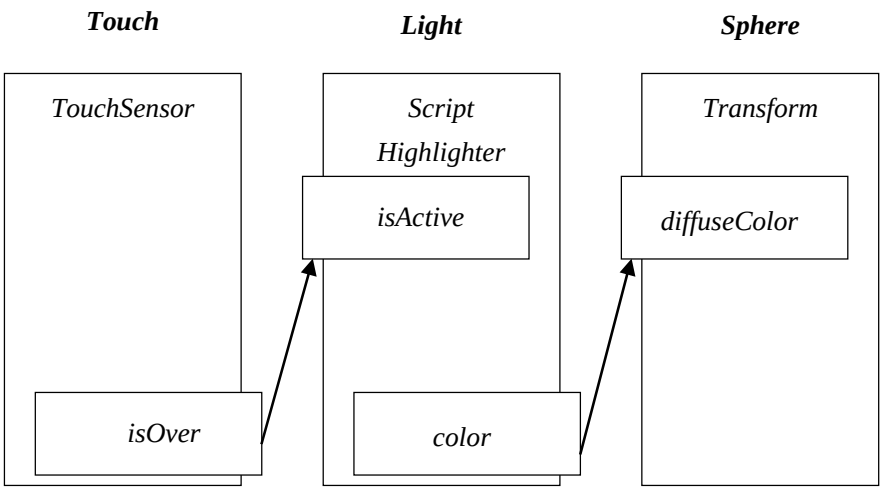


Fig. 13 - Exemple de déclenchement des événements avec un script [LEG99]

LES HORLOGES

Le détecteur *TimeSensor* permet de générer des événements correspondants à des tops d'horloge à intervalles réguliers.

```
TimeSensor {
exposedField SFTime cycleInterval 1.0
exposedField SFBool enabled TRUE
exposedField SFBool loop FALSE
exposedField SFTime startTime 0.0
exposedField SFTime stopTime 0.0
eventOut SFTime cycleTime
eventOut SFFloat fraction_changed
eventOut SFBool isActive
eventOut SFTime time
}
```

Le champ *cycleInterval* détermine la durée d'un intervalle de temps en secondes (par défaut, 1 seconde) pendant lequel l'horloge renvoie des événements. Le booléen *enabled* permet d'activer l'horloge tandis que *loop* permet de générer continuellement des événements au lieu de s'arrêter au premier intervalle. Les champs *startTime* et *stopTime* permettent de définir les dates de début et de fin de génération d'événements à partir du début de la scène. Les événements en sortie sont utilisés pour créer des animations; en particulier, l'événement *fraction_changed* pourra être récupéré par un interpolateur.

L'événement *cycleTime* renvoie l'heure à chaque début de cycle (utile pour synchroniser différents événements). Les événements *isActive* et *time* renvoient respectivement l'état de l'horloge et « l'heure » de chaque fraction.

LES INTERPOLATEURS

Un interpolateur permet de lisser les animations en générant les valeurs intermédiaires à partir d'une liste de valeurs-clés. Les animations de ce type sont appelées *keyframed animations* car on utilise des instants-clés que l'on associe à des valeurs-clés pour définir l'animation, puis c'est au moteur 3D de faire l'interpolation.

LE KEYFRAMING

Supposons que vous vouliez déplacer un objet d'un point A vers un point B, puis de B vers A [KDO98]. Vous pourriez envoyer à intervalles réguliers des coordonnées à un objet pour le placer successivement aux différents points de la trajectoire. Ce serait une charge très lourde et difficile à mettre en œuvre. Le *keyframing* repose sur une idée simple : on fournit le point de départ A et le point d'arrivée B, puis de nouveau le point de départ A et c'est le programme de visualisation qui calcule tous les points intermédiaires de la trajectoire.

Pour mettre en œuvre le *keyframing*, il faut :
- une horloge pour cadencer le calcul de l'animation,
- spécifier la durée de l'animation,
- spécifier les moments clés de l'animation, c'est à dire les moments caractéristiques (dans notre exemple le moment de départ du point A, le moment d'arrivée au point B et le moment du retour au point A),
- un système d'interpolation, c'est à dire un système qui calcule à chaque moment intermédiaire la valeur intermédiaire entre deux valeurs clés.

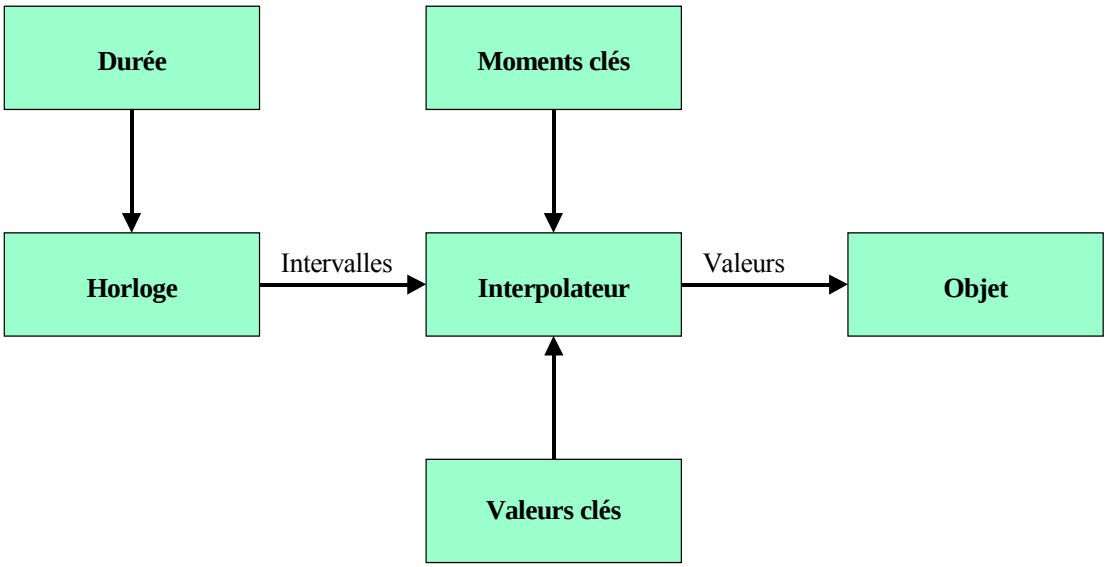


Fig 14 - Mécanisme du keyframing - [KDO98]

LE MECANISME D'INTERPOLATION

L'interpolation se fait de manière linéaire. En prenant notre exemple, si la durée de l'animation est de 10 secondes et que l'arrivée au point B se passe au bout de 5 secondes. Si le point A se trouve à la coordonnée X=0 et le point B à la coordonnée X=10, nous aurons :

TEMPS ECOULE T	POSITION EN X DE L'OBJET
T = 0	X = 0
T = 1	X = 2
T = 2	X = 4
T = 3	X = 6
T = 4	X = 8
T = 5	X = 10
T = 6	X = 8
T = 7	X = 6
T = 8	X = 4
T = 9	X = 2
T = 10	X = 0

LE KEYFRAMING EN VRML

Nous allons mettre en correspondance les fonctions nécessaires au *keyframing* et les instructions du langage VRML.

FONCTION	VRML
Durée de l'animation	Champ *CycleInterval* de *TimeSensor*
Moments clés	champ *key* d'un interpolateur
Valeurs clés	champ *keyValue* d'un interpolateur
Calcul d'interpolation	Interpolateurs de VRML

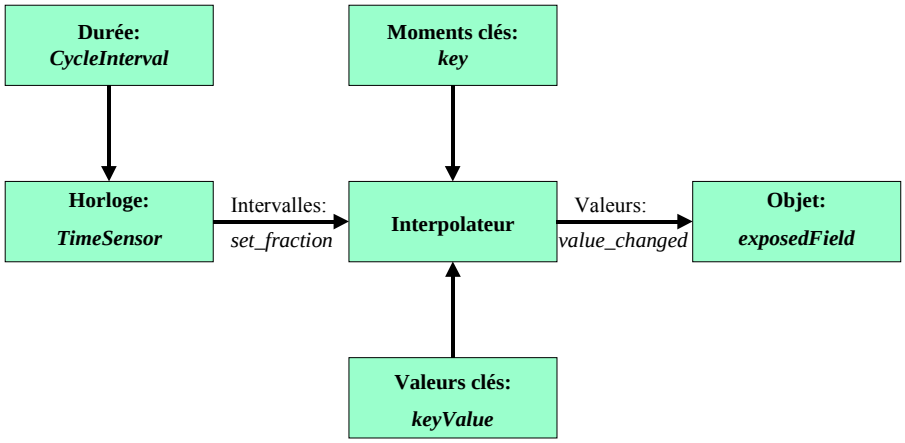


Fig. 15 - Le keyframing en VRML - [KDO98]

INTERPOLATEURS DE VRML

ColorInterpolator	Permet de modifier la couleur d'un objet. L'interpolateur calcule les couleurs intermédiaires entre les couleurs fournies pour chaque moment clé.
CoordinateInterpolator	Permet de modifier les coordonnées des points constituant les blocs *IndexedFaceSet*, *IndexedLineSet* et *PointSet*.
NormalInterpolator	Permet de modifier les vecteurs d'un sous-bloc *Normal* des blocs *ElevationGrid* et *IndexedFaceSet*, ce qui a pour effet de modifier la façon dont la lumière semble se réfléchir sur la surface.
OrientationInterpolator	Permet de faire effectuer des rotations à un objet.
PositionInterpolator	Permet de déplacer un objet.
ScalarInterpolator	Permet de modifier la valeur de tous les champs du type *exposedField SFFloat.*

Tous les interpolateurs de VRML ont la même structure :

```
Interpolator {
     eventIn SFFloat set_fraction
     exposedField MFFloat key [ ]
     exposedField Type keyValue [ ]
     eventOut        Type value_changed
}
```

set_fraction :
Evénement. Permet de fixer la valeur de temps entre 0.0 et 1.0. En général, ce champ reçoit la valeur *fraction_changed* d'un *TimeSensor*.

key :
Moments clés compris entre 0.0 et 1.0 (1.0 = 100% du temps écoulé).

keyValue :
Valeurs clés de chaque moment clé. Le nombre de valeurs doit être un multiple du nombre de moments clés. Le type des clés dépend de l'interpolateur.

value_changed :
Evénement. Renvoie la nouvelle valeur calculée par l'interpolateur. Le type renvoyé dépend de l'interpolateur.

Exemple : *PositionInterpolator*
L'interpolateur PositionInterpolator permet d'animer un objet en le déplaçant sur une trajectoire pré définie.

```
PositionInterpolator {
eventIn SFFloat set_fraction
exposedField MFFloat key []
exposedField MFVec3f keyValue []
eventOut SFVec3f value_changed
}
```

Le champ *key* est la liste des instants-clés de l'animation (ici un déplacement) et le champ *keyValue* est la liste correspondante (ici des positions). L'événement en entrée permet de récupérer les fractions de temps générées par un *TimeSensor* et l'événement en sortie *value_changed* renvoie la *valeur-clé* ou la valeur interpolée correspondant à la fraction de temps reçue.

Chapitre 2 – LE NAVIGATEUR ELEMENTAIRE

2.1 – FONCTIONALITES DU NAVIGATEUR

2.1.1 - MULTI PLATE-FORME

Dans sa conception, le logiciel obtenu est multi plate-forme, à savoir que tout le code écrit en Java est réutilisable sans aucune migration sur les plate-formes les plus courantes comme Win32 et Linux. Par contre, la partie native des *bindings* de *GL4Java* (correspondant au lien entre Java et OpenGL) requiert des fichiers bibliothèques spécifiques à chaque plate-forme, mais cela présente l'avantage de ne pas avoir eu à développer toutes les fonctions de la bibliothèque OpenGL en Java, ce qui n'aurait été raisonnable ni en terme de faisabilité ni en terme de performances.

D'après les tests que j'ai effectué, le logiciel fonctionne avec des fichiers bibliothèques identiques (c'est-à-dire la même installation) sous Windows 95, 98, *Millenium Edition, 2000* ou NT 4 *Workstation*. Sous Linux, il faut bien sûr prendre les fichiers bibliothèques correspondant aux *bindings* Java-OpenGL pour Linux. Une fois l'installation complète, le logiciel fonctionne correctement. Il existe des fichiers bibliothèques correspondant à d'autres versions d'UNIX, mais je n'ai pas eu l'occasion de les installer. Les tests ont été effectués sur un PC bi-processeur Pentium III, cadencés à 500 Mhz avec une carte graphique Matrox G400.

Je n'ai pas expérimenté non plus la plate-forme MacOs, cependant le logiciel a toutes les chances de fonctionner sur ce système d'exploitation car il existe des *bindings* de *GL4Java* pour Macintosh (MacOs 9).

2.1.2 - ANIMATIONS

Le point de départ d'une animation pouvant être l'activation d'un objet 3D suite à un clic sur le bouton de la souris, ce sont les capteurs (*sensors*) de VRML qui sont à même de modéliser cette action, notamment le détecteur *TouchSensor*, qui est déclenché par un clic de souris sur le ou les objets associés dans le même groupe. C'est pourquoi nous avons besoin d'un mécanisme susceptible d'implémenter les actions de *TouchSensor*, à savoir :

- connaître les coordonnées 3D de l'objet survolé,
- savoir si le détecteur *TouchSensor* a été activé par un clic de souris,
- savoir si le pointeur de la souris survole le détecteur *TouchSensor*,
- disposer d'un horodatage de l'activation.

La manipulation d'objets 3D demande l'activation des mode de sélection et de *feed-back* d'OpenGL [WOO99], qui servent respectivement à sélectionner une région de l'écran ou un objet, et récupérer le résultat des calculs de restitution. Il s'agit donc de gérer l'interactivité entre l'utilisateur et les objets 3D. Il peut être difficile, sans de tels mécanismes, de déterminer

quel objet l'utilisateur a sélectionné dans une scène 3D, car ceux-ci peuvent subir des opérations telles que des rotations, des translations, etc.

Le mécanisme de sélection permet à l'utilisateur de l'application de sélectionner un objet dessiné à l'écran. On commence par stocker dans la mémoire tampon les informations de la scène, puis on passe en mode sélection et on redessine la scène. Une fois en mode sélection, le contenu de la mémoire tampon ne change plus tant que l'on n'a pas quitté ce mode. Lorsque que l'on quitte le mode sélection, OpenGL retourne une liste de primitives qui entrent en intersection avec le volume visionné, celui-ci étant défini par les matrices de modélisation-visualisation et de projection actives.

Chaque primitive entrant en intersection avec le volume visionné provoque un *hit*, et une liste de primitives est retournée sous forme de tableau de *noms* (valeurs numériques entières) auxquels sont associés des données, les *enregistrements de hit*, qui correspondent au contenu actuel de la *pile de noms*. Pour construire la pile de noms, on charge les noms au fur et à mesure que l'on émet des commandes de dessin de primitives en mode sélection : lorsque la liste des noms est retournée, on peut l'exploiter pour déterminer les primitives qui ont été sélectionnées à l'écran.
Voici les différentes étapes du mécanisme de sélection :

- Spécifier le tableau à employer pour les enregistrements de *hits* retournés via *glSelectBuffer(),*
- Entrer en mode sélection en déclarant le paramètre *GL_SELECT* avec *glRenderMode(),*
- Initialiser la pile de noms via *glInitNames()* et *glPushName(),*
- Définir le volume visionné concerné par la sélection, qui peut être différent du volume visionné originalement pour dessiner la scène, ce qui oblige à sauvegarder puis à restaurer l'état actif des transformations via *glPushMatrix()* et *glPopMatrix()*,
- Alterner les commandes de dessin des primitives et les commandes de manipulation de la pile de noms pour que chaque primitive concernée soit nommée,
- En quittant le mode sélection, traiter les données de sélection retournées, c'est-à-dire les enregistrements de hits.

Avec le mécanisme de sélection, on utilise le *picking* pour déterminer si un objet a été cliqué. Pour ce faire, OpenGL emploie une matrice spécifique en conjonction avec une matrice de projection afin de restreindre le dessin à une région réduite du cadrage, proche du pointeur de la souris. Ensuite, on met en œuvre du code pour récupérer une information, telle que le clic sur le bouton de la souris, qui va déclencher le mode sélection.
Une fois le mode de sélection activé, on utilise la matrice spéciale de *picking* et les objets qui sont dessinés aux abords du curseur de la souris provoquent des *hits* de sélection. Ainsi, pour autoriser le *picking*, il faut déterminer les objets proches du curseur.

L'autre mécanisme pouvant participer à la mise en œuvre des animations est celui de la gestion du temps, c'est le rôle d'un autre détecteur de VRML97, à savoir *TimeSensor*, qui a été décrit en détail au chapitre 1.
Dans le navigateur, si le nœud *TimeSensor* avait été géré dans le graphe de scène comme les autres nœuds de VRML, celui-ci aurait été tributaire du rafraîchissement de la scène, ce qui aurait été illogique, le rôle de *TimeSensor* étant principalement de générer des tops d'horloge, il aurait donc pénalisé les performances globales des animations utilisant ce détecteur. C'est pourquoi le nœud *TimeSensor* est géré au sein d'une thread d'animation java, en dehors du graphe de scène, c'est-à-dire en dehors de la classe *SceneGraphOpenGL*. Cette thread

d'animation est crée la première fois qu'un nœud *TimeSensor* est rencontré dans la scène VRML.

Pour mettre en œuvre les animations dans VRML97, nous avons vu les mécanismes de sélection d'un objet 3D et de la gestion des tops d'horloge. Il reste maintenant à utiliser celui de la gestion des événements, c'est-à-dire les instructions *ROUTE … TO…* , mécanisme qui a été décrit au chapitre 1. Rappelons que le nœud VRML *TouchSensor* comporte notamment les événements sortants suivants :

- *isActive* : renvoie *TRUE* si le détecteur est activé par un clic souris,
- *isOver* : renvoie *TRUE* lorsque le pointeur de la souris survole le détecteur,
- *TouchTime* : renvoie l'heure lors d'une activation.

Pour modéliser ces mécanismes dans le navigateur, j'ai procédé de la façon suivante. Une boucle teste les événements à savoir l'appui sur le bouton de la souris (*MousePressed*), le mouvement de la souris avec bouton relâché (*MouseMoved*), le relâchement du bouton de la souris (*MouseReleased*). Dans chacune de ces routines, nous allons entre autre engager le mécanisme de sélection en passant comme paramètre l'événement que nous désirons tester :

- Dans la routine *MousePressed*, nous allons engager le mécanisme de sélection en testant l'événement *isActive*, pour savoir si le détecteur *TouchSensor* a été déclenché par un clic de souris, ce qui indiquerait que l'objet 3D a été cliqué par l'utilisateur,

- Dans la routine *MouseMoved*, nous allons engager le mécanisme de sélection en testant l'événement *isOver*, pour savoir si le détecteur *TouchSensor* a été déclenché par le passage du pointeur de la souris, ce qui indiquerait que l'objet 3D a été survolé par l'utilisateur,

- Dans la routine *MouseReleased*, on engage le mécanisme de sélection en testant l'événement *touchTime*, qui renvoie l'heure de l'activation.

Il faut ensuite activer le système d'événements de VRML pour que les instructions *ROUTE* puissent les propager correctement : par exemple, si l'événement *isActive* du détecteur *TouchSensor* est déclenché, celui-ci étant un événement sortant (*eventOut*) du nœud précité, il pourra être transmis à un événement en entrée (*eventOut*) d'un autre nœud, de manière à être capable d'agir sur ce dernier, par exemple en activant un spot de lumière. En voici une illustration :

```
#VRML V2.0 utf8

NavigationInfo {
   headlight TRUE
}

DEF LUM  PointLight {
   location 2 1 1
   color 1 0 0
   ambientIntensity 0.2
   intensity 0.5
   radius 45
   on FALSE
}
     Shape {
          geometry Cylinder {}
               appearance Appearance {
                    material Material {
                              diffuseColor 1 1 0
                    }
               }
          }
DEF CAPTEUR TouchSensor {}
ROUTE CAPTEUR.isActive TO LUM.set_on
```

Dans cet exemple, on définit un cylindre de couleur jaune. Le nœud *PointLight* désigné par *LUM* définit un spot de lumière rouge qui, par défaut n'est pas activé (*on FALSE*): au départ, le cylindre apparaît donc avec sa couleur initiale, jaune. Le capteur *TouchSensor* désigné par *CAPTEUR* contrôle la figure géométrique de la manière suivante : lorsque l'on clique sur le cylindre, celui-ci étant contrôlé par le capteur *TouchSensor*, le clic de souris va réveiller le détecteur qui va donc (par construction) déclencher l'événement *isActive.* Cet événement en sortie (*eventOut*) du capteur prendra alors la valeur *TRUE*. Cette valeur sera propagée par l'instruction *ROUTE* vers l'événement *on* en entrée (*eventIn*) du nœud *PointLight*. Cet événement *on* prendra donc à son tour la valeur *TRUE*, ce qui permettra alors d'activer le nœud *LUM*, ce qui aura pour effet d'allumer un spot rouge sur le cylindre. Au moment où le bouton de la souris sera relâché, l'événement *isActive* prendra la valeur *FALSE* et le spot de lumière rouge s'éteindra, le cylindre reprendra alors son aspect jaune.
Des animations plus conséquentes sont obtenues à l'aide du détecteur *TimeSensor* autorisant la gestion d'une horloge. Rappelons que le nœud VRML *TimeSensor* comporte notamment les événements sortants suivants :

- *cycleTime* : renvoie l'heure à chaque début de cycle,
- *fraction_changed* : renvoie la fraction de temps comprise entre 0 et 1 du cycle écoulé,
- *isActive* : renvoie l'état de l'horloge,
- *time* : renvoie l'heure de chaque fraction

Dans l'exemple suivant, nous allons voir de quelle manière le capteur *TimeSensor* peut servir à créer une animation :

```
#VRML V2.0 utf8
NavigationInfo {
headlight FALSE
   }
   Transform {
      rotation .5233 -.8449 .1112 .4929
      children [
           Shape {
                appearance Appearance {
                     material Material {
                          diffuseColor 1 1 1
                        }
                   }
                   geometry Box {
                      size 3 3 3
                   }
              }
        ]
   }
   DEF PL PointLight {
      ambientIntensity 1.0
      location 1 3 4
   }
   DEF TS TimeSensor {
      cycleInterval 3
      loop TRUE
      startTime 1.0
      stopTime 0.0
   }
   ROUTE TS.fraction_changed TO PL.set_intensity
```

Cette scène affiche un cube éclairé par un spot (nœud *PointLight* nommé PL). Le détecteur *TimeSensor* effectue un cycle de trois secondes. Durant ce cycle, on redirige par l'instruction *ROUTE* l'événement sortant *fraction_changed* qui représente la fraction de temps en cours comprise entre 0 et 1 vers la valeur d'intensité de la lumière dont l'événement entrant est *intensity*.
Ce cycle est perpétuel puisque *LOOP* est à *TRUE*. Le résultat donne une variation d'intensité de la lumière sur le cube. Comme nous l'avons vu précédemment, le nœud *TimeSensor* est géré dans une thread. C'est notamment à l'intérieur de celle-ci que sont testés les états des événements sortants du nœud : ceux qui ont fait l'objet d'une activation sont propagés en sortie du nœud.

2.1.3 - SCRIPTS JAVA

Dans un script Java, le source du script se trouve dans une classe externe au fichier VRML. Il suffit alors de préciser dans le champ *url* le nom du fichier *class* à appeler. Bien entendu, l'utilisation de scripts Java implique la maîtrise de ce langage de programmation.
L'appel d'un script Java est désigné par le terme de J.S.A.I. (*Java Scripting Autoring Interface*). Pour que le script fonctionne avec le fichier VRML appelant, le fichier source Java doit déclarer les classes du paquetage cv97 de *CyberVRML97*.
Pour faire fonctionner les scripts Java dans le navigateur, j'ai adapté le code de *CyberVRML97* pour prendre en compte le fait que dans une instruction *ROUTE*, le nœud précisé en sortie pouvait faire partie d'un script, auquel cas, il fallait passer l'événement en entrée de ce nœud (*eventIn*) comme paramètre d'entrée de ce script. La structure du programme Java est la suivante :

- La classe du programme script doit dériver de la classe *Script()*,
- Cette classe peut comporter des déclarations (par exemple, une rotation),
- Une procédure *initialize()* qui est exécutée en premier et qui peut servir par exemple à lire le contenu d'un événement en sortie (*eventOut)*,
- Une procédure *shutdown()* qui peut être utilisée pour exécuter des instructions de fin de programme,
- Une procédure *processEvent(cv97.Event e)* qui permet de traiter les événements en entrée (*eventIn*) et de réaliser le traitement prévu par le programme script Java.

Nous allons voir l'exemple d'un pendule animé :

```
#VRML V2.0 utf8
# Script Java : Pendule
# S. Geunier - Membre du GVF
NavigationInfo {
   type "EXAMINE"
}
DirectionalLight {
   ambientIntensity 1
}
Transform {
   translation 0 2 0
   children [
        Shape {
             geometry Box {          #Haut du pendule
                  size 2 0.5 0.5
               }
               appearance Appearance {
                  material Material {
                       diffuseColor 0 0 1
                     }
               }
          }
```

```
          DEF T_R Transform {
             translation 0 -1.5 0
             center 0 1.5 0
             children [
                  Shape {
                       geometry Cylinder { #La barre du milieu
                            height 3
                            radius 0.1
                          }
                     }
                     Transform {
                        translation 0 -1.5 0
                        children [
                             Shape {
                                  geometry Sphere { # Le bas
                                       radius 0.5
                                    }
                                    appearance Appearance {
                                       material Material {
                                            diffuseColor 1 0 0
                                         }
                                    }
                               }
                          ]
                     }
                ]
          }
     ]
}
DEF TS TimeSensor {
   stopTime -1
   loop TRUE
   cycleInterval 3
}
DEF Programme Script {
   url "pprog.class"
   eventIn SFFloat temps
   eventOut SFRotation rotate
}
ROUTE TS.fraction_changed TO Programme.set_temps
ROUTE Programme.rotate_changed TO T_R.set_rotation
```

Ce fichier VRML décrit les trois parties d'un pendule, à savoir une parallélépipède pour représenter le haut, un cylindre qui représente le milieu et une sphère pour le bas. Un détecteur *TimeSensor* définit un cycle de deux secondes. Ce cycle est perpétuel puisque *LOOP* est à *TRUE*. D'autre part, le fichier VRML comporte un nœud *Script* nommé *Programme* indiquant la présence d'un programme script Java nommé *pprog.class*, externe au fichier VRML.

Le nœud *Script* nommé *Programme* déclare deux paramètres :
- un événement de type *eventIn* nommé *temps*,
- un événement de type *eventOut* nommé *rotation*.

D'autre part, il déclare dans son champ *url* le nom du programme script Java, pprog.class.

La première instruction *ROUTE* indique que l'événement de type *eventOut, fraction_changed* (fraction du temps comprise entre 0 et 1) de *TimeSensor* est routé vers l'événement *eventIn temps* du nœud *Script Programme* : ce paramètre (temps) va donc être passé au programme script Java.
La deuxième instruction *ROUTE* indique que l'événement de type *eventOut*, *rotation_changed*, est routé vers l'événement de type *eventIn set_rotation* du nœud *transform* nommé T_R. Ceci a donc pour effet de passer un paramètre du programme script Java (après traitement) vers le nœud T_R, en modifiant une rotation.

Voici le programme Java[18] :

```
import cv97.*;
import cv97.node.*;
import cv97.field.*;
import java.lang.*;
import java.lang.String.*;

//-----------------------------------------
// Objet  : Animation d'un pendule en Java
// Auteur : Sébastien Geunier membre du GVF
//-----------------------------------------
// Classe pprog  (nom identique au fichier)
//-----------------------------------------
public class pprog extends Script {
SFRotation MaRotation ;

public void initialize() {
MaRotation= (SFRotation)getEventOut("rotate");
}
public void shutdown() {
}
    public void processEvent(cv97.Event e) {
     if(e.getName().equals("temps")) {
          ConstSFFloat LeTemps = (ConstSFFloat)e.getValue();
          float  teta  =  (float)  Math.sin(2*Math.PI*((double)
     LeTemps.getValue()));
          MaRotation.setValue(0,0,1,teta);
     }
    }
}
```

[18] Dans le cas de l'utilisation de ce script Java à travers un navigateur VRML classique, il y a lieu de remplacer cv97 par vrml et de s'assurer que ces classes existent et sont correctement installées sur la machine.

2.1.4 – EXPERIMENTATIONS

Quelques expérimentations ont été effectuées sur le navigateur, comme la création du double fenêtrage et l'utilisation de X3D :

- Le double fenêtrage

Des aménagements ont été apportés au code Java du navigateur pour prendre en compte une fonction de double fenêtrage, à savoir être capable de charger et donc de représenter deux scènes VRML évoluant en parallèle, c'est-à-dire de façon indépendante l'une de l'autre, étant chacune dans sa propre fenêtre OpenGL. Dans le cas de scènes statiques pour lesquelles on n'a pas besoin du mécanisme de sélection (picking), cet essai a été concluant. Cependant, cela est relativement gênant puisque ce mécanisme doit être activé en permanence dans le logiciel pour faire fonctionner les capteurs.

- L'utilisation de X3D [X3D]

« Extensible 3D » est un standard définissant le Web interactif et le contenu 3D intégré au multimédia. Il s'adresse à un usage pour une variété de matériels et un large éventail d'applications telles que l'ingénierie et la visualisation scientifique, les présentations multimédia, l'enseignement, les divertissements, etc. X3D se veut également un format universel d'échange de données graphiques 3D et multimedia intégrées. Il sera probablement le successeur de VRML.
Des expériences ont été réalisées en X3D à l'aide du navigateur développé. En effet, le logiciel *CyberVRML97* disposant dans son architecture d'un parseur X3D de base, il a été possible de prendre des scènes VRML très simples, de les charger dans le navigateur et après en avoir obtenu la transcription en une scène X3D (que l'on peut alors sauvegarder sur un fichier disque), la scène X3D a été représentée à l'écran.

2.2 – LA BIBLIOTHEQUE GRAPHIQUE OPENGL

OpenGL est une bibliothèque graphique très évoluée et totalement portable offrant de nombreuses ressources aux programmeurs qui souhaitent élaborer un moteur 3D.

2.2.1 – PRESENTATION D'OPENGL

OpenGL est une bibliothèque graphique 3D : elle reçoit des ordres de tracé de primitives graphiques (facettes,etc.) directement en 3D, une position de caméra, des lumières, des textures à plaquer sur les surfaces, etc. Ensuite, OpenGL prend en charge les changements de repère, la projection en perspective à l'écran, le *clipping*, l'élimination des parties cachées, l'interpolation des couleurs, la rasterisation (tracé ligne à ligne) des faces pour en faire des pixels [NEY98].
OpenGL est conçu comme une interface rationnelle, indépendante du matériel, qu'il est possible d'implémenter sur de nombreuses plate-formes matérielles différentes. OpenGL ne propose aucune commande permettant d'effectuer des taches de fenêtrage ni de récupérer des entrées utilisateur ; pour cela, il faudra déterminer le gestionnaire de fenêtres qui permet de contrôler le matériel que l'on utilise.

De même, OpenGL ne propose pas de commandes de haut niveau permettant de décrire des modèles d'objets tridimensionnels. Il faut construire le modèle à partir d'un jeu restreint de formes géométriques primitives : des points, des lignes et des polygones. La restitution (*rendering*) est le processus par lequel un ordinateur crée des images à partir de modèles. Ces modèles ou objets sont construits à partir des formes géométriques primitives définies par leurs sommets.

L'image rendue au final est composée de pixels affichés à l'écran : un pixel est le plus petit élément visible que peut afficher un écran. Les informations concernant les pixels (par exemple leur couleur) s'organisent dans la mémoire en *plans de mémoire-image*, ou *bitplans*. Ces zones de mémoire détiennent un bit d'information pour chaque pixel affiché ; le bit peut indiquer l'intensité du rouge à appliquer au pixel concerné. Les plans de mémoire-image s'organisent eux-mêmes dans un *framebuffer* ou *mémoire tampon graphique* qui contient toutes les informations graphiques dont l'écran a besoin pour contrôler la couleur et l'intensité de tous les pixels qu'il affiche. [WOO99]

OpenGL s'appuie sur le matériel disponible selon la carte graphique. Toutes les opérations de base sont à priori accessibles sur toute machine, mais elles s'exécuteront plus ou moins rapidement selon qu'elles soient accélérées ou non par le matériel de la carte graphique ou par les instructions spécialisées des nouveaux processeurs.

La machine OpenGL se décompose en trois moteurs :

- Le *« geometric engine »* s'occupe de la partie purement 3D : triangulation de polygones, changements de repère, projection en perspective à l'écran, clipping (élimination de ce qui sort de l'écran).

- Le *« raster engine »* prend en charge la partie 2D :
  - il rasterise les triangles de manière à produire des pixels (appelés fragments tant qu'ils ne sont pas affichés),
  - interpole au passage les couleurs et utilise les coordonnées de texture pour plaquer une image,
  - élimine les parties cachées par *Z-buffer* : en fonction de la profondeur z du fragment en cours de tracé et du z actuellement stocké dans le pixel visé, le pixel est ou non modifié (i.e. sa couleur et son z).

Ceci constitue le schéma de base.

- Le *« raster manager »* à qui le pixel est confié pour être tracé à l'écran après d'éventuels traitements supplémentaires (comme *l'anti-aliasing*).

2.2.2 – LES PRINCIPES FONDAMENTAUX

- Lorsque l'on a effectué une transformation, tous les objets dessinés après cette transformation sont affectés tant que la fonction *glLoadIdentity()* n'est pas appelée (auquel cas les objets ne sont plus transformés)

- Les modifications affectent directement le repère de coordonnées et se basent sur le repère de coordonnées. Les axes utilisés sont donc toujours les axes locaux : si on a effectué une rotation autour de l'axe des X et que l'on souhaite faire une rotation autour de l'axe des Y, cette rotation s'effectuera autour de l'axe modifié par la première rotation

- Un objet dessiné ne peut plus être modifié.

L'homothétie (*glScalef / glScaled*)

Il s'agit d'un agrandissement ou d'une réduction par rapport au centre du repère de coordonnées. Le repère est agrandi ou réduit selon certains axes. Toutes les transformations qui suivent seront alors basées sur le nouveau repère, tant que celui-ci n'aura pas été réinitialisé par *glLoadIdentity()*. Donc, on translate un objet après avoir effectué une réduction par *glScale()*, le déplacement sera donc plus court puisque le repère aura diminué.

La translation (*glTranslatef / glTranslated*)

C'est un déplacement selon le vecteur passé en paramètre. Les transformations affectant aussi le système de coordonnées, si l'on a effectué une translation et qu'ensuite on veuille faire une homothétie, le centre de cette homothétie aura lui aussi été déplacé.

La rotation (*glRotatef / glRotated*)

Elle permet d'effectuer un mouvement de rotation autour de n'importe quel axe. On lui passe comme paramètre l'angle en degré et les coordonnées x, y, z du vecteur de rotation. La rotation fait tout tourner, y compris le repère lui-même (comme les autres transformations). Donc si l'on effectue une première rotation selon un vecteur v1 et que l'on veuille faire une deuxième rotation selon un vecteur v2, il ne faut pas oublier que v2 a lui aussi subi la première rotation.

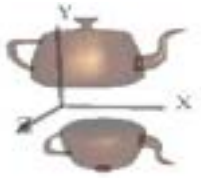
La figure suivante illustre ces trois types de transformations.

• Translation selon un vecteur (t_x, t_y, t_z) :

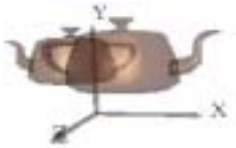
$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• Rotations par rapport aux axes X, Y et Z données ici pour un repère " main droite " :

$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$R_Y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$R_Z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• Homothétie d'un facteur h_x, h_y et h_z sur chacune des coordonnées

$$H = \begin{bmatrix} h_x & 0 & 0 & 0 \\ 0 & h_y & 0 & 0 \\ 0 & 0 & h_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 17 - Les transformations 3D [TOP01]

LE MECANISME DE DOUBLE BUFFER

En mode *true colors*, la couleur se code avec trois octets représentant les composantes rouge, verte et bleue. Mais les pixels peuvent contenir bien plus d'information au-delà de la simple couleur : on utilise aussi une profondeur z (utilisée pour le z-buffer) et une opacité alpha (d'où la notation RGBA). De plus, ces valeurs sont doublées en mode *double buffer*, lequel permet de faire des animations fluides en laissant visible l'image précédente pendant que l'on trace la nouvelle dans le *back buffer*. Comme ces diverses informations référencent les mêmes pixels, on parle de « plans » superposés. Le nombre de bits alloués aux divers plans est paramétrable, la quantité disponible dépendant de la mémoire de la carte graphique. Par défaut, sur les machines disposant de peu de mémoire, passer en *double buffer* impose de prendre sur les autres plans, notamment *alpha* et z. Il est cependant obligatoire pour la 3D temps réel d'utiliser un "double buffer" pour obtenir un plus grand confort visuel.

2.2.3 – LA PORTABILITE

Si OpenGL est 100% portable avec tous les systèmes d'exploitation, c'est parce que ce n'est pas lui qui se charge de l'affichage. C'est la raison pour laquelle OpenGL est obligé de s'appuyer sur un système d'exploitation ayant une interface graphique : OpenGL sert d'intermédiaire entre le programme et le système d'exploitation, en transformant et en traduisant les informations données par le programme et en les envoyant au système d'exploitation. Plus précisément, précisons le travail à faire par le programmeur pour initialiser OpenGL : tout d'abord, il doit initialiser un environnement graphique dans le système d'exploitation (c'est-à-dire une fenêtre dans le cas de Windows). Cette fenêtre sera définie de manière unique par un *Device Context,* une sorte de *handler* qui permet au programme d'accéder à cette fenêtre et de lui faire subir toutes les informations d'affichage possible (affichage de pixels, etc.). Ensuite, le travail sera d'associer à ce *Device Context* un *Rendering Context* qui permettra à OpenGL de s'adresser à la fenêtre pour lui envoyer ses ordres de dessin :

Programme → OpenGL → Rendering Context → Device Context → Affichage
Géré par le programme Géré par le système d'exploitation

Voilà pourquoi OpenGL est si portable: il se contente de mixer les informations et de les transmettre au système d'exploitation. A lui ensuite de les afficher. La seule contrainte pour le système d'exploitation est donc de pouvoir dialoguer avec OpenGL. Ceci est par exemple possible sous Windows grâce au driver Opengl32.dll disponible dès l'installation et qui s'occupe de la traduction des données OpenGL en données compréhensibles pour la carte vidéo: c'est là qu'intervient l'accélération matérielle, car OpenGL est désormais accéléré matériellement par quasiment toutes les cartes 3D actuelles.
Pour obtenir que vos programmes 3D soient accélérés par votre carte 3D, il suffit de remplacer (dans le cas de Windows) le fichier opengl32.dll de base qui fait du rendu logiciel par un fichier opengl32.dll traduisant les données OpenGL en données directement compréhensibles par la carte 3D, c'est-à-dire installer les drivers de la carte 3D.
Ce mode de fonctionnement impose intrinsèquement une contrainte pour le programmeur: il doit d'abord initialiser l'affichage à travers le système d'exploitation (c'est-à-dire créer une fenêtre pour récupérer un *Device Context*) pour initialiser OpenGL. Cela n'est pas encore trop lourd sous Windows, mais cela est plus compliqué sous X-Window. Heureusement, il existe une bibliothèque nommée glut, qui permet de faire cela sans se préoccuper du système d'exploitation, ce qui rend le code 100% portable.

2.2.4 – LE PROCESSUS DE RESTITUTION

Ce processus est décrit par ce que l'on appelle « *OpenGL rendering pipeline* ». Il représente l'ordonnancement des commandes effectuées par OpenGL pour effectuer ses actions de rendu. La figure suivante représente ces différentes actions :

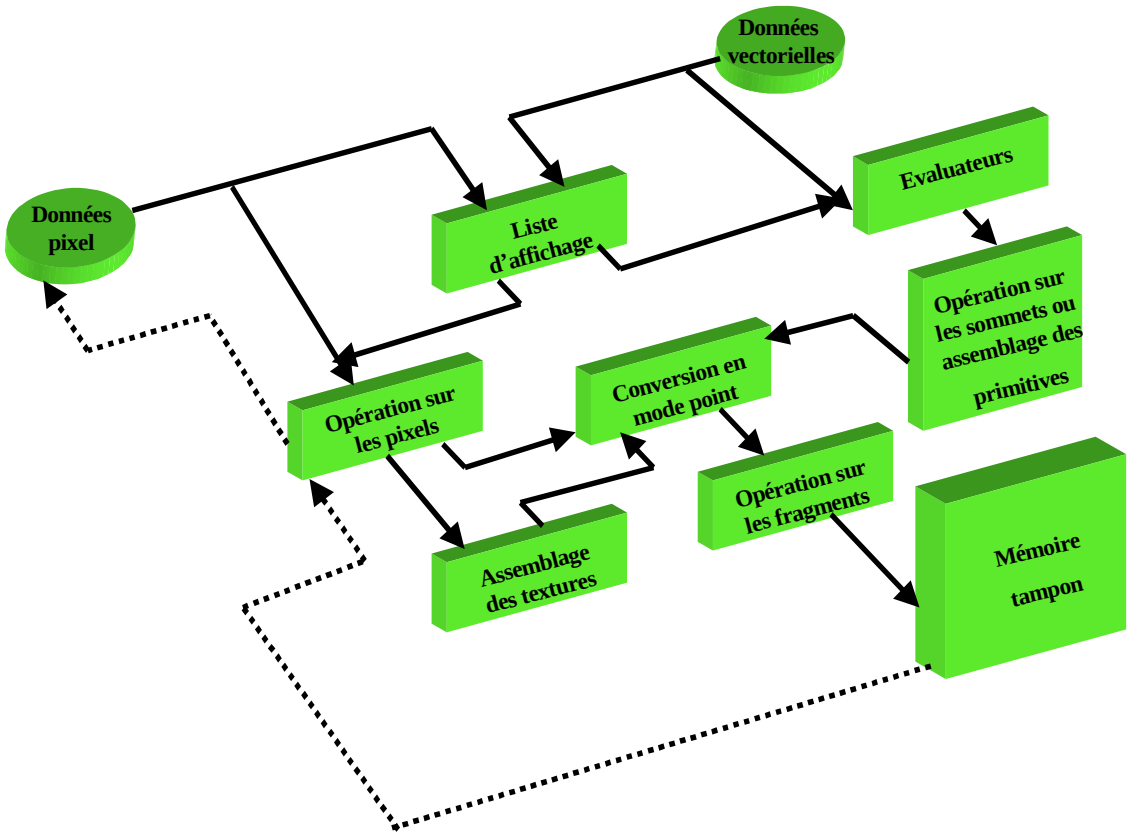


Fig. 18 - Ordre des opérations - [WOO99]

Certaines de ces commandes se réfèrent au dessin d'un objet et vont traverser le pipeline jusqu'à la mémoire d'image ou *frame buffer*, sorte de grille de dimension 2, d'autres ont trait à l'aspect et vont changer telle ou telle variable d'état (couleur, matrice de transformation, texture, etc.).

En entrée, les commandes sont soit envoyées directement dans le pipeline soit stockées dans une liste d'affichage ou *display list.* Les listes d'affichage permettent d'enregistrer toutes les données (décrivant une géométrie ou des pixels), de façon à les utiliser ultérieurement. Si l'application est distribuée sur le réseau, la liste d'affichage permet de réduire le trafic client-serveur car elle réside sur le serveur. Bien entendu, toute commande renvoyant une valeur ne pourra être incluse dans une liste. Lorsque celle-ci est exécutée, l'information contenue dans la liste est renvoyée comme si elle provenait de l'application en mode exécution.

Les données géométriques peuvent se présenter sous la forme d'un ensemble de sommets ou *vertex* de type point, ligne, polygone ou de courbes et surfaces se présentant sous une forme paramétrique :

c(u) = [X(u)Y(u)Z(u)]
s(u, v) = [X(u, v)Y(u, v) Z(u, v)]

de types NURBS (Non-Uniform Rational B-Spline), ou Bezier.
Un *vertex* n'est pas seulement une simple référence à une coordonnée homogène de dimension 4 (x, y, z, w), c'est un objet plus complexe qui comprend des attributs d'orientation angulaire (vecteur normal), de couleur, de texture et d'arêtes.
Les opérations géométriques qui vont suivre devront donc traiter ces *vertex* en tenant compte de tous ces aspects. Ces opérations consistent essentiellement à appliquer un certain nombre de transformations (découpage, clôture) pour respecter le point de vue demandé.
Les étapes finales alimentent le *frame buffer* en analysant la contribution de chacun de ces *vertex* sur la base des critères de couleur, de visibilité et autres effets spéciaux (brouillard, transparence, …).
Si les données ne se présentent pas sous forme de *vertex* mais sous forme de *pixels*, c'est pour alimenter soit une matrice de texture, soit le processus de rasterisation.

2.2.5 – LE PROCESSUS DE VISUALISATION

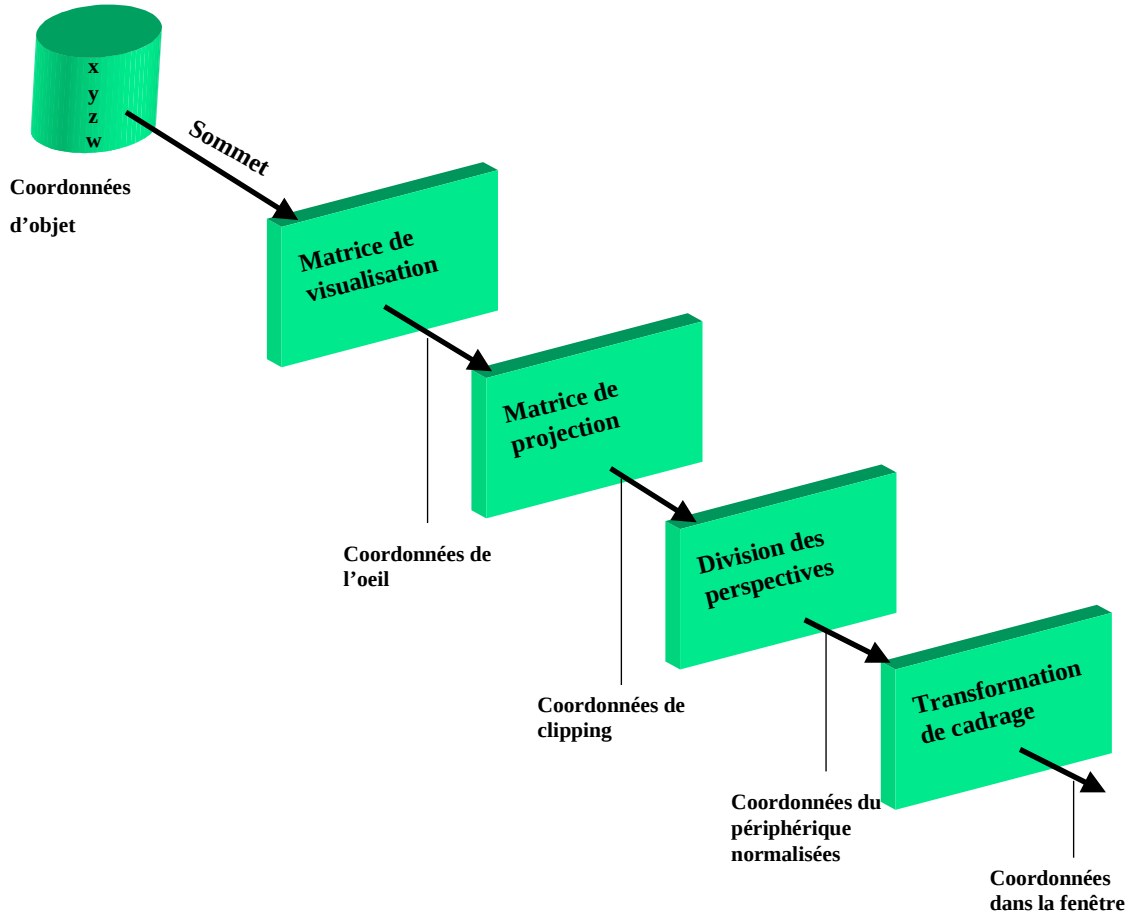


Fig. 19 - Etapes de la transformation des sommets [WOO99]

On distingue quatre transformations successives pour obtenir une image:

- Transformation de visualisation (*View*)

Permet de fixer la position et l'orientation de la caméra de visualisation. La visualisation est analogue à l'action de positionner une caméra et la faire pointer dans une direction donnée. Avant de spécifier une transformation de visualisation, on positionne la matrice active dans la matrice identité avec *glLoadIdentity()*, ce qui permet d'éviter de combiner les matrices de transformation précédentes avec celles que l'on va entrer. Ensuite, on spécifie la transformation de visualisation à l'aide de la commande *gluLookAt()* (si l'on utilise la librairie *glu*), permettant d'indiquer où est placé la caméra et préciser dans quelle direction elle est orientée.

- Transformation de modélisation (*Model*)

Permet de créer la scène à afficher par création, placement et orientation des objets qui la composent. On peut utiliser la commande *glScalef()* comme transformation de modélisation : elle permet le positionnement et l'orientation du modèle.

- Transformation de projection (*Projection*)

Permet de fixer les caractéristiques optiques de la caméra de visualisation (focale, mode de projection, etc.). Il s'agit donc de déterminer le champ de vision. Cette transformation définit de plus la façon dont les objets sont projetés à l'écran. Il existe deux types de projections : la *projection en perspective conique* correspond à ce que l'on voit dans la réalité, les objets éloignés apparaissant plus petits. Celle-ci peut être spécifiée soit par *glFrustum()*, soit par la routine de la bibliothèque d'utilitaires *gluPerspective()*.
La *projection en perspective cavalière*, quant à elle, colle directement les objets à l'écran, sans affecter leurs tailles relatives : la distance par rapport à la caméra n'affecte pas la taille des objets. Celle-ci est utilisée lorsque l'image finale doit refléter les dimensions des objets plutôt que leur aspect. Celle-ci peut être spécifiée par la commande *glOrtho()*.

- Transformation d'affichage ou de cadrage (*ViewPort*)

Elle permet d'établir une correspondance entre les coordonnées transformées et les pixels affichés à l'écran. Elle fixe la taille et la position de l'image sur l'écran.

Les transformations de visualisation et de modélisation n'en forment qu'une pour OpenGL (transformation *ModelView*). Cette transformation fait partie de l'environnement OpenGL. La transformation de projection existe en tant que telle dans OpenGL, et fait elle aussi partie de l'environnement OpenGL. Chacune de ces transformations peut être modifiée indépendamment de l'autre ce qui permet d'obtenir une indépendance des scènes modélisées vis à vis des caractéristiques de la "caméra" de visualisation. [JAN01]

La matrice de modélisation-visualisation s'applique aux coordonnées d'objet entrantes pour produire les *coordonnées de l'œil*. Ensuite, OpenGL applique la matrice de projection pour produire les *coordonnées de clôture (clipping)*, ce qui correspond à la définition d'un volume dont les objets se situant à l'extérieur de celui-ci disparaissent de la scène finale.

La *division des perspectives* correspond à la division des valeurs des coordonnées par w (quatrième coordonnée, généralement égale à 1), on obtient ainsi des *coordonnées de périphérique normalisées*.
Les coordonnées transformées sont alors converties en *coordonnées dans la fenêtre* à l'aide de la transformation de cadrage. Les dimensions de cadrage servent à agrandir ou réduire l'image finale (fig. 15).

2.2.6 – LES PRIMITIVES GEOMETRIQUES

Les sommets peuvent être représentés de la manière suivante :

glVertex2s(3,4) : représente un sommet avec les coordonnées 3D (3,4,0),
glVertex3d(3.0, 1.0, 2.0) : représente des nombres en virgule flottante à double précision.
glVertex4f(2.0, 1.0, -3.0, 2.0) : représente un sommet en tant que coordonnées homogènes (x, y, z, w). Les coordonnées x, y et sont alors divisées par w (w différent de 0).

Gldouble vecteur[3] = {7.0, 8.0 , 200.0} ;
glVertex3dv (vecteur) ;

vecteur représente un pointeur vers un tableau de nombres en virgule flottante à double précision.

Les primitives géométriques sont définies entre les deux instructions OpenGL *glBegin (<nom de la primitive>) et glEnd()*.

Exemple : Polygone plein

```
glBegin (GL_POLYGON) ;
     glVertex2f(0.0, 0.0) ;
     glVertex2f(0.0, 3.0) ;
     glVertex2f(4.0, 3.0) ;
     glVertex2f(6.0, 1.5) ;
     glVertex2f(4.0, 0.0) ;
glEnd() ;
```

La figure obtenue est alors la suivante :

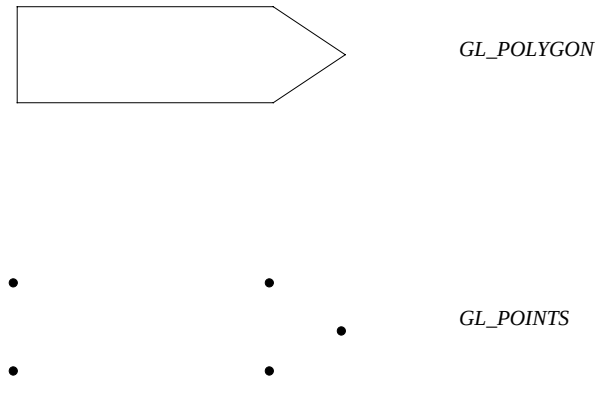


Fig. 20 - Tracé d ’un polygone ou d ’un ensemble de points [WOO99]

En précisant GL_POINTS à la place de GL_POLYGON, on obtient la figure du bas.
Voici maintenant la liste des dix arguments possibles et les types de primitives que l'on peut obtenir :

GL_POINTS	points isolés
GL_LINES	paires de sommets interprétées en tant que segments de ligne isolés
GL_LINE_STRIP	séries de segments de ligne joints
GL_LINE_LOOP	idem, avec ajout d'un segment entre le dernier et le premier sommet
GL_TRIANGLES	regroupe par trois les sommets interprétés comme des triangles
GL_TRIANGLE_STRIP	bande liée de triangles
GL_TRIANGLE_FAN	éventail lié de triangles
GL_QUADS	regroupe par quatre des sommets interprétés comme des polygones à quatre cotés
GL_QUAD_STRIP	bande liée de quadrilatères
GL_POLYGON	contour externe d'un polygone convexe simple

On suppose, dans la description suivante, que les sommets n (v0, v1, v2, …, vn-1) sont décrits entre *glBegin()* et *glEnd()*.

GL_POINTS

Dessine un point à chacun des n sommets.

GL_LINES

Trace une série de segments de lignes non joints. Les segments sont dessinés entre v0 et v1, entre v2 et v3, etc. Si n est impair, le dernier segment est tracé entre vn-3, vn-2 et vn-1 est ignoré.

GL_LINE_STRIP

Dessine un segment de ligne de v0 à v1,puis de v1 à v2, etc. jusqu'à dessiner le segment de vn-2 à vn-1. Ainsi, un total de n-1 segments de ligne est tracé. Rien n'est tracé dans que n est inférieur à 1. Il n'y a aucune restriction quant au sommets décrivant une bande (ou une boucle). Les lignes peuvent s'entrecouper de manière arbitraire.

GL_LINE_LOOP

Semblable à *GL_LINE_STRIP*, excepté que le dernier segment de ligne est tracé de vn-1 à v0 pour terminer la boucle.

GL_TRIANGLES

Dessine une série de triangles (polygones à trois côtés) en utilisant les sommets v0, v1, v2 puis v3, v4, v5, etc. Si n n'est pas un multiple de trois, le ou les deux derniers sommets sont ignorés.

GL_TRIANGLE_STRIP

Dessine une série de triangles (polygones à trois côtés) en utilisant les sommets v0, v1, v2, puis v1, v2, v3, puis v2, v3, v4, etc. L'ordre sert à garantir que les triangles sont tous dessinés avec la même orientation de manière que la bande forme correctement une partie de la surface. n doit être au moins de 3 pour que quelque chose soit dessiné.

GL_TRIANGLE_FAN

Semblable à GL_TRIANGLE_STRIP, excepté que les sommets sont v0, v1, v2, puis v0, v2, v3, puis v0, v3, v4, etc.

GL_QUADS

Dessine une série de quadrilatères (polygones à quatre côtés) en commençant par v0, v1, v2, v3, puis v4, v5, v6, v7, etc. Si n n'est pas un multiple de 4, le ou les deux ou trois derniers sommets sont ignorés.

GL_QUAD_STRIP

Dessine une série de quadrilatères (polygones à quatre côtés) en commençant par v0, v1, v3, v2 puis v2, v3, v5, v4 puis v4, v5, v7, v6, etc. n doit être au moins de quatre pour que quelque chose soit dessiné. Si n est impair, le dernier sommet est ignoré.

GL_POLYGON

Dessine un polygone en utilisant les points v0, … , vn-1 comme sommets. n doit être au moins de 3, sinon rien n'est dessiné. En outre, le polygone spécifié ne doit pas se couper lui-même et être convexe. Si les sommets ne satisfont pas à ces conditions, le résultat est imprévisible.

La figure suivante représente un exemple de chacun d'eux :

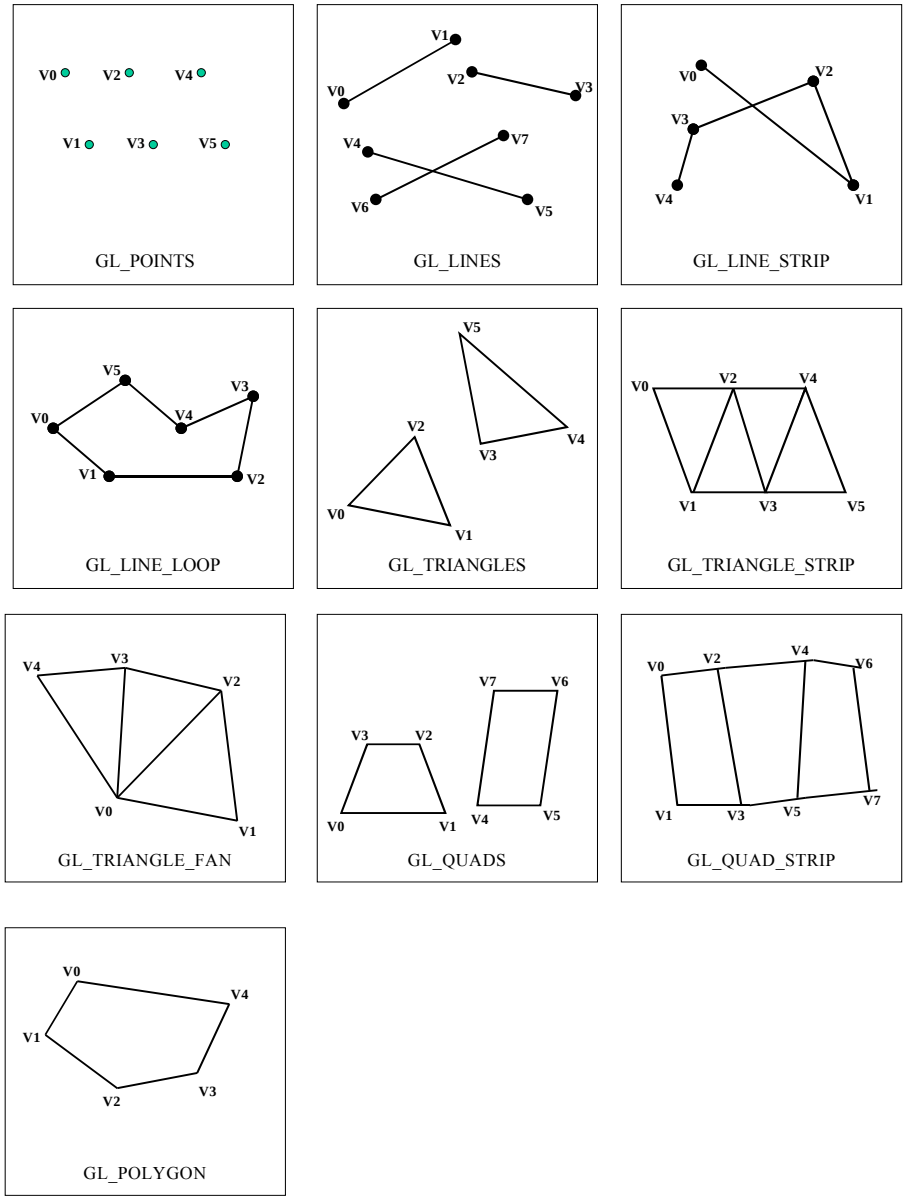


Fig. 21 - Types de primitives géométriques [WOO99]

NB : Il existe plusieurs manières de tracer la même primitive. La méthode choisie dépend des données dont on dispose sur les sommets.

Toute primitive surfacique 3D est décomposée en triangles par OpenGL. Le triangle, la ligne et le point sont donc les seules primitives géométriques traitées par le matériel, ce qui ramène toutes les interpolations au cas linéaire (facile à traiter de façon incrémentale dans le matériel,

d'où cette limitation aux triangles, mais qui peut produire un aspect légèrement anguleux). On spécifie un triangle de la façon suivante :

```
glBegin(GL_TRIANGLES) ;
glVertex3f(x1, y1, z1) ;
glVertex3f(x2, y2, z2) ;
glVertex3f(x3, y3, z3) ;
glEnd ;
```

OpenGL permet de spécifier des primitives plus complexes, qui seront décomposées :

- en quadrilatères avec *GL_QUADS*,
- en polygones avec *GL_POLYGON*.

On peut également ne tracer que les sommets avec *GL_POINTS* (cf. plus haut) ou que les contours avec *GL_LINE_LOOP*.

La syntaxe de *glVertex()*, comme celle de tous les attributs (couleur, normales, etc.) peut avoir plusieurs formats :

- on peut fournir deux à quatre composantes (la quatrième correspond à la coordonnée homogène pour des positions, à l'opacité *alpha* pour des couleurs,
- On peut utiliser des *float*, des *double*, des *short,* des *int* pour les coordonnées (suffixe f,d,s ou i),
- On peut spécifier explicitement les coordonnées ou passer par un vecteur (suffixe v).

2.2.7 – POSITIONNEMENT ET REPERAGE DES OBJETS

- Pour positionner un objet, on passe par la matrice *GL_MODELVIEW* :

```
glMatrixMode(GL_MODELVIEW) ;
glLoadIdentity() ;
glTranslatef(tx, ty, tz) ;
glRotatef(a, axeX, axeY, axeZ) ;
```

On peut alors effectuer le tracé dans un repère local, et les coordonnées subiront ensuite la pile de transformations indiquée (en remontant l'ordre des déclarations).

Il existe bien une pile car à tout moment on peut faire *glPushMatrix()* ou *glPopMatrix()* pour effectuer une transformation temporaire. Ainsi, si l'on est placé dans un repère local à une personne, et que l'on sait dessiner un bras dans un repère local, on pourra procéder comme suit :

```
tracé_du_tronc() ;
glPushMatrix() ;
glRotatef(40.0, 0, 0, 1) ; // le gauche
tracé_du_bras() ;
glPopMatrix() ;
```

```
glPushMatrix() ;
glRotatef(-40.0, 0, 0, 1) ; // le droit
tracé_du_bras ;
glPopMatrix() ;
```

- Pour décrire la caméra, on passe par la matrice *GL_PROJECTION*. On y décrit tout d'abord le type de projection :

glMatrixMode(GL_PROJECTION) ;
glLoadIdentity() ;
glOrtho(-1.0, 1.0, -1.0, 1.0, *near*, *far*) ; // si projection parallèle
gluPerspective(45.0, 1.0, *near*, *far*) ; //si projection en perspective

Pour la projection parallèle, on indique le domaine [xmin,xmax] x [ymin,ymax] des coordonnées visibles, pour la projection en perspective, on donne l'angle d'ouverture et le ratio largeur/hauteur. Dans les deux cas on précise le domaine de validité de la profondeur z.
Puis on peut orienter la caméra soit en visant un point avec *gluLookAt* (position, visée, haut), soit en procédant par rotations et translations (cette dernière solution étant utilisée en OpenGL pur).

- Pour déplacer les lumières, on utilise la matrice *GL_MODELVIEW* comme pour un objet :

Le point indiqué à *glLightf()* est transformé au moment où l'on appelle cette fonction. On peut alors réinitialiser GL_*MODELVIEW* et s'en servir pour positionner les objets.

- La fonction *gluLookAt() :*

Elle permet de définir un point de vue. Les trois premiers paramètres de cette fonction définissent les coordonnées du point de vue, les trois suivants l'endroit où l'on regarde, et les trois derniers un vecteur indiquant où se trouve le haut de la caméra. *GluLookAt()* effectue les opérations nécessaires sur la matrice active afin que le repère soit bien orienté. Nous devons donc activer nous-mêmes la matrice *ModelView* grâce à *glMatrixMode()*. Si gluLookAt() n'est pas appelée, la caméra prend une position et une orientation par défaut.

2.2.8 – LES ATTRIBUTS D'UNE FACE

Il existe plusieurs façons de spécifier l'apparence d'une facette :

- Couleur de la face :

On utilise *glColor3f(r,g,b).* On peut également préciser l'opacité en ajoutant un paramètre *alpha*.

- Couleur aux sommets, à interpoler sur la face :

on redéfinit simplement cette couleur avec *glColor()* pour chaque sommet, juste avant le *glVertex()*.

- Modèle d'illumination (Gouraud) en fonction de l'orientation des lumières.

Un tel modèle nécessite la définition d'une *matière* composée d'une couleur ambiante (celle qui apparaît dans l'ombre), d'une couleur diffuse (celle qui apparaît du côté éclairé, souvent la même), d'une couleur spéculaire (celle des reflets), et d'un coefficient de rugosité (qui contrôle l'épaisseur de la tache spéculaire).

Exemple :

```
static Glfloat ambient[]  = {0.1, 0.1, 0.1, 0.1} ;
static Glfloat diffuse[]   = {0.8, 0.4, 0.4, 1.0} ;
static Glfloat specular[] = {0.8, 0.8, 0.8, 1.0} ;
static Glfloat shininess  = 50.0 ;
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular) ;
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, shininess) ;
glEnable{GL_LIGHTING) ;
```

On peut également définir une couleur d'émissivité avec le *flag GL_EMISSION*. Comme le suggère le *flag GL_FRONT_AND_BACK*, on peut définir une matière différente sur les deux faces.

On peut définir jusqu'à huit sources de lumières dont on peut spécifier de nombreux attributs par *glLightfv* (*GL_LIGHT0*, attribut, vecteur) : tout d'abord la position (*GL_POSITION*) et la couleur dont on donne séparément les composantes ambiante, diffuse et spéculaires, permettant d'obtenir des rendus peu physiques tels que des sources qui ne génèrent pas de reflets ou qui ne contrôlent que la lumière ambiante), mais également l'atténuation en fonction de la distance et de l'angle (pour un spot). Une lumière est activée par *glEnable*(*GL_LIGHT0*) , *glEnable*(*GL_LIGHT1*), etc.

L'illumination tient compte de la direction de la lumière et de l'observateur par rapport à l'orientation de la facette. C'est pourquoi l'on doit préciser cette orientation dans OpenGL à l'aide de *glNormal**() pour chaque sommet avant l'instruction *glVertex**(). En effet, un vecteur normal est un vecteur pointant dans la direction perpendiculaire à une surface. A partir d'une surface plane, la direction perpendiculaire est la même pour tous les points de la surface, mais à partir d'une surface courbe, la direction normale peut être différente pour chaque point de la surface.
Avec OpenGL, on peut spécifier une normale pour chaque polygone ou chaque sommet. Les sommets d'un même polygone peuvent partager la même normale (pour une surface plane) ou avoir des normales différentes (pour une surface courbe). [WOO99]

On peut également demander à OpenGL de normaliser les normales par *glEnable(GL_NORMALIZE)* mais cela entraîne un calcul supplémentaire par sommet, lequel inclut l'évaluation d'une racine carrée, sachant que chaque sommet est redéfini pour toutes les faces auxquelles il appartient.

2.2.9 – STRUCTURE D'UN PROGRAMME OPENGL

Un programme OpenGL est structuré de la façon suivante :

- Création d'une fenêtre
- Récupération d'un *Device Context* pour cette fenêtre
- Récupération d'un *Rendering Context* pour ce *Device Context*
- Initialisation d'OpenGL à l'aide de ce *Rendereing Context*

…

- Création d'objets à partir de primitives simples
- Affichage des objets

…

- Libération du *Rendering Context*
- Libération du *Device Context*
- Fermeture du programme

2.2.10 - LE PRINCIPE DES BINDINGS JAVA-OPENGL

Les applications écrites avec un langage compilé sous un format binaire natif peuvent accéder directement aux implémentations natives des bibliothèques OpenGL. En d'autres termes, ils chargent les bibliothèques OpenGL natives et appellent directement les interfaces disponibles dans ces bibliothèques [GOR98].

Les applications écrites en Java devraient aussi bien tirer profit des implémentations natives des bibliothèques OpenGL mais certains écueils doivent être évités pour faire cela. Les développeurs Java ont rapidement pris conscience qu'en même temps que l'idée de « écrire une seule fois, exécuter partout », il est parfois nécessaire de dépasser Java et tirer parti des implémentations natives.

Ainsi, l'Interface Java Native (*JNI*, *Java Native Interface*) a été développée. Une application Java qui utilise des méthodes natives dépend de l'exécution du code qui tourne directement sur le matériel de la machine hôte. Le code Java est compilé en bytecode Java pour être exécuté sur la Machine Virtuelle Java (*Java Virtual Machine*, *JVM*). Celle-ci agit comme un intermédiaire entre le bytecode et le matériel hôte.
Le code natif, pour sa part, peut être vu comme fonctionnant en dehors de la *JVM*. C'est un code objet spécifique à la plate-forme. Ce code natif existe typiquement dans une bibliothèque que l'on peut charger dynamiquement, ayant été préalablement construite en utilisant le compilateur et le linker natifs de la machine. Le contenu de cette bibliothèque est nécessairement dépendant de la plate-forme car le code objet généré par le compilateur natif est nécessairement spécifique au processeur et au système.

De plus, il peut il y avoir toute sorte de code dans cette bibliothèque qui référence aussi des dispositifs spécifiques à la plate-forme. Si les objets binaires dans une bibliothèque construite pour la plate-forme x adhère à l'API *JNI*, alors cette bibliothèque native peut être utilisée, sans reconstruction, avec n'importe quelle *JVM* qui supporte le *JNI* sur la plate-forme x.

Si les distributeurs A et B fournissent tous les deux une *JVM* avec un support *JNI* sur la machine x, alors n'importe quelle bibliothèque utilisant seulement les appels *JNI* à l'interface avec la *JVM* fonctionnera à la fois avec la *JVM* de A ou de B.

Du fait que les bibliothèques OpenGL natives ne s'adaptent pas à la spécification *JNI*, nous avons besoin de développer une bibliothèque native enveloppante qui s'adapte à cette spécification et donne accès aux bibliothèques OpenGL natives.

Une bibliothèque native enveloppante différente est nécessaire pour chaque implémentation OpenGL native et pour chaque plate-forme sur laquelle cette application doit être déployée. Alors que cela outrepasse le but « Ecrire une fois, exécuter partout », ce principe nous permet de pouvoir déplacer nos applications Java-OpenGL vers d'autres plate-formes plus facilement car seule la bibliothèque native enveloppante a besoin d'être migrée vers chaque nouvelle plate-forme.
L'application Java-OpenGL devrait être portable vers chaque nouvelle plate-forme sans modification ni compilation vers du code natif. Sur le schéma ci-dessous, seulement le bloc du centre qui représente cette bibliothèque native enveloppante a besoin d'être migrée vers chaque nouvelle plate-forme.

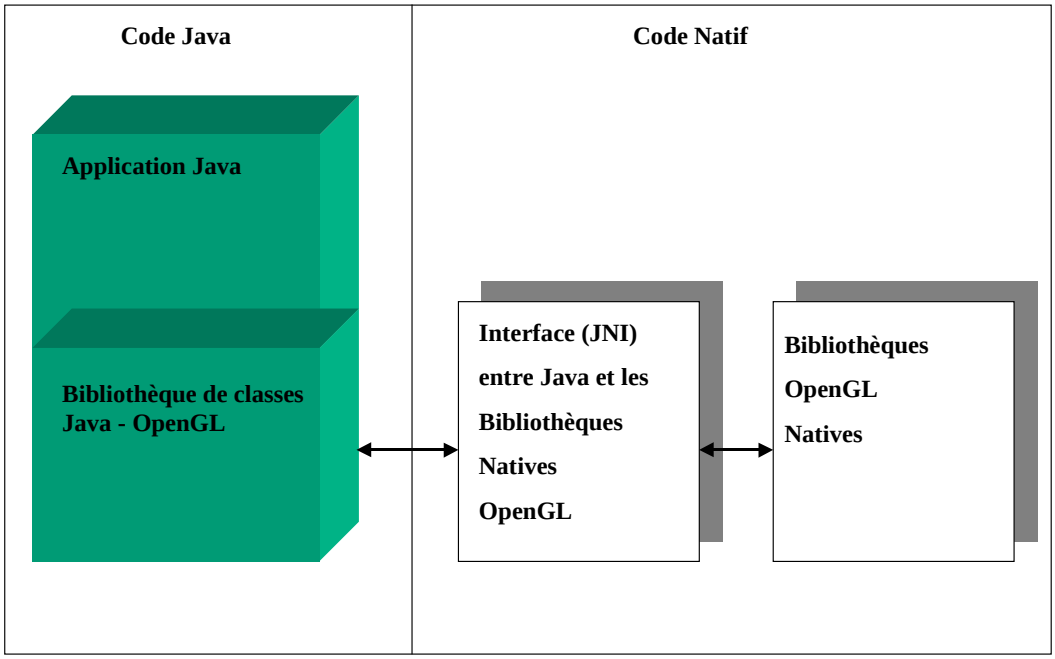


Fig. 22 - Interface entre Java et OpenGL [GOR98]

2.2.11 – LES DIFFERENTES IMPLEMENTATIONS JAVA-OPENGL

Il existe deux approches permettant aux programmeurs d'utiliser les fonctions OpenGL depuis Java :

- La première consiste à réécrire OpenGL en Java (cf. JAVAGL).
- La deuxième, infiniment plus réaliste, consiste à relier une implémentation OpenGL native avec des classes Java utilisant des méthodes Java natives. On utilise alors des librairies natives qui peuvent être accélérées par le matériel.

Ce deuxième groupe d'implémentations est plus fourni et vaut la peine d'en voir l'historique.

Au début de l'année 1997, Léo Chan de l'Université de Waterloo mis rapidement au point un ensemble de reliures (bindings) Java-OpenGL non officielles pour le JDK 1.0.2 qui fournissaient des classes Java qui s'interfaçaient avec une implémentation OpenGL native. Ce code était relativement instable, supportait 50-60% des fonctions OpenGL et tournait seulement sous X11 et UNIX. Il n'intégrait pas non plus Awt ou Swing mais créait à la place sa propre gestion de fenêtre. Ce code original était appelé *OpenGL4Java*.

En septembre 1997, *OpenGL4Java* avait été complètement abandonné. Cependant, en décembre 1997, plusieurs camps de développeurs avaient pris le code de *OpenGL4Java*, l'avaient amélioré et avaient repris ce travail en rajoutant des fonctionnalités pour utiliser la nouvelle interface native Java.

JAVAGL

Ce paquetage est une implémentation OpenGL en 100% pur Java. Il supporte environ 20% des fonctions OpenGL et a très peu d'exemples. Il fonctionne de façon très lente. Ce paquetage semble avoir été abandonné.

JOGL [JOG]

Il s'agit probablement du binding *OpenGL4Java* le plus avancé. Il supporte environ 60-70% des fonctions OpenGL et le rendu à travers les composants AWT. Ce paquetage supporte UNIX à la base mais est connu pour fonctionner également sous Win32. JOGL a quelques limitations fondamentales en ce qu'il ne supporte pas le rendu sur des fenêtres multiples. Il ne gère pas non plus le plaqué de textures et il n'y a pas de support de la bibliothèque *glu*. Enfin, JOGL ne fonctionne pas de façon fiable avec le multi-threading.
JOGL n'a pas été mis à jour depuis décembre 1997, date à laquelle il a été arrêté.

GL4JAVA [SGO99]

Ce paquetage a été développé à partir de JOGL-0.6 sur le même modèle. Une des différences est que dans JOGL on ne conservait pas le préfixe "gl" alors que *GL4Java* le conserve. *GL4Java* est toujours maintenu et progresse constamment et s'appelle aujourd'hui *OpenGL4Java*. Un développeur extérieur à *Jausoft,* Gerard Ziemski [ZIE01], a récemment mis au point une version Apple Mac de *GL4Java*.
Tous les sources du paquetage de *GL4Java* sont disponibles sur le site web de Jausoft. Ce paquetage est celui qui a été choisi pour réaliser le rendu du Navigateur et repris en détail au paragraphe 2.3.7.

JSPARROW [JSP]

JSParrow est né d'une deuxième vague de développement au début de 1998 quand PFU LIMITED (filiale de Fujitsu) décidèrent qu'ils avaient besoin d'un binding Java-OpenGL pour leurs projets internes.
JSParrow utilise une structure de classe similaire aux clones *OpenGL4Java* en ce que tout le système de fenêtrage, le code GL, GLU et GLUT est joint en une seule et unique classe. JSParrow offre l'avantage que les programmes C/C++ basés sur GLUT peuvent être convertis en Java avec peu d'efforts. Il semble également que ce paquetage supporte des environnements multi-thread. JSParrow supporte Windows 95/98/NT4/2000, Solaris (OpenGL sun ou Mesa), Linux(Mesa), IRIX et MacOS.
En revanche, les programmes sources ne sont pas disponibles sur le site web de JSParrow. Seuls les fichiers ".class" sont proposés. JSParrow est toujours maintenu à ce jour.

MAGICIAN [MAG]

La dernière vague de développement avait commencé en mai 1997 quand *Arcane Technologies L.T.D.* commençait le nouveau développement de l'interface Java-OpenGL *Magician* sans référence au code *OpenGL4Java*, prétendant ainsi corriger les défauts architecturaux de l'approche *OpenGL4Java* et également à cause de l'absence de bindings officiels Java-OpenGL.
Au cœur de *Magician* apparaît la notion de « pipelines composables » qui permettrait de gérer plusieurs contextes OpenGL indépendants. Arcane Technologies annonce également la possibilité de rendu avec multi-fenêtrage avec une gestion interne des différents contextes OpenGL.
Magician intègre les composants AWT et Swing, utilise le multi-threading et supporte 100% des fonctions OpenGL et GLU y compris les évaluateurs, l'interface NURBS et les quadriques. D'autre part, un groupe de classes utilitaires inclut la gestion de trackballs virtuels, plaqué de texture et timers. *Magician* permet de laisser ou retirer le préfixe "gl" selon un paramètre prédéfini. *Magician* supporte Linux, Irix, Solaris, Windows 95/98/NT, AIX, OS/2 et MacOS. Il supporte également les *JVM* les plus courantes y compris le JDK de SUN, le SDK de Microsoft Java, Internet Explorer, Netscape Communicator et Symantec Café.
Magician est une version commerciale, ce qui a pour conséquence que l'accès aux code sources et exécutables est payant (environ 7000$). Seule reste disponible une version exécutable bridée en vitesse d'exécution, ce qui limite énormément l'expérimentation de ce logiciel.

2.2.12 – AVANTAGES ET INCONVENIENTS DES BINDINGS JAVA-OPENGL

AVANTAGES D'UTILISER LES BINDINGS JAVA-OPENGL *[BIL99]*

- OpenGL fournit un modèle procédural pour le graphique, qui correspond à beaucoup d'algorithmes et méthodes que les programmeurs de graphique ont toujours utilisé. Le modèle procédural est très intuitif pour les habitués de la 3D.

- OpenGL fournit un accès direct au pipeline de rendu.
  Cela est vrai de n'importe quel binding de langage, y compris de la plupart des bindings Java. OpenGL autorise les programmeurs à spécifier directement comment doit être réalisé le rendu. Il ne s'agit plus seulement de suggérer et faire une requête comme en *Java3D*, mais de le stipuler.

- OpenGL est optimisé dans tous les sens possibles.
  OpenGL est optimisé d'un point de vue matériel et logiciel et existe sur de nombreuses plate-formes, depuis les PC au meilleur marché et consoles de jeux jusqu'aux calculateurs graphiques les plus puissants.

- Les distributeurs de chaque type de matériel 3D supportent OpenGL.
  OpenGL est le standard sur lequel les distributeurs de matériel mesurent leur technologie graphique.

INCONVENIENTS D'UTILISER LES BINDINGS JAVA-OPENGL

Par ailleurs, rien n'est parfait. OpenGL, et certainement les bindings Java-OpenGL ont quelques insuffisances.

- Les points forts de l'approche procédurale de la programmation graphique deviennent simultanément une faiblesse pour beaucoup de programmeurs Java.
  Pour des programmeurs débutants dont beaucoup d'entre eux ont appris la programmation orientée objet, la méthode procédurale d'OpenGL ne s'accorde pas très bien avec une approche orientée objet. (Cependant, il existe Open Inventor, sur-couche objet d'OpenGL).

- Beaucoup d'optimisation OpenGL des distributeurs signifie une diminution dans le choix du matériel.
  Il est de l'intérêt de chaque distributeur de construire des extensions propriétaires et faire des optimisations propriétaires pour vendre plus de leur propre matériel. Comme avec toutes les optimisations matérielles, vous devez utiliser des accélérations OpenGL spécifiques étant entendu que chaque optimisation pour une plate-forme diminue la portabilité et la performance de beaucoup d'autres. Les optimisations de portée plus générale de *Java3D* visent principalement à maximiser la portabilité des applications *Java3D*.

- L'exposition des détails internes du pipeline de rendu OpenGL peut compliquer de façon significative des programmes graphiques 3D par ailleurs simples.
  La puissance et la souplesse viennent au prix de la complexité. Dans les cycles de développement rapides des technologies actuelles, la complexité est quelque chose qui doit être évitée quand cela est possible. Le vieil adage à propos des défauts est vrai : plus le nombre de lignes de code augmente, plus de défauts il y a (en général).

2.3 - CONCEPTION DU NAVIGATEUR

Le développement qui a été réalisé dans le cadre de ce mémoire s'appuie sur deux logiciels ("*Open Source"*) existants:
CyberVRML97 pour Java de Satoshi Konno [SKO99] et *GL4Java* de Sven Goethel [SGO99].

- Le premier logiciel, écrit entièrement en Java, permet de gérer la hiérarchie d'un graphe de scène à travers des nœuds VRML97, fait du rendu à travers *Java3D* mais ne permet pas de faire du rendu à l'aide d'OpenGL.
- Le second permet de gérer des primitives OpenGL en Java (à travers le mécanisme de *binding Java-OpenGL* que nous détaillerons au chapitre suivant) permettant de réaliser du rendu de figures 3D, mais ne gère pas la hiérarchie des nœuds VRML.
- L'objet du présent développement est de réaliser l'interface OpenGL permettant, à partir du graphe de scène VRML, de représenter les objets 3D correspondants.

La figure suivante donne une vue générale de l'architecture logicielle du navigateur[19].

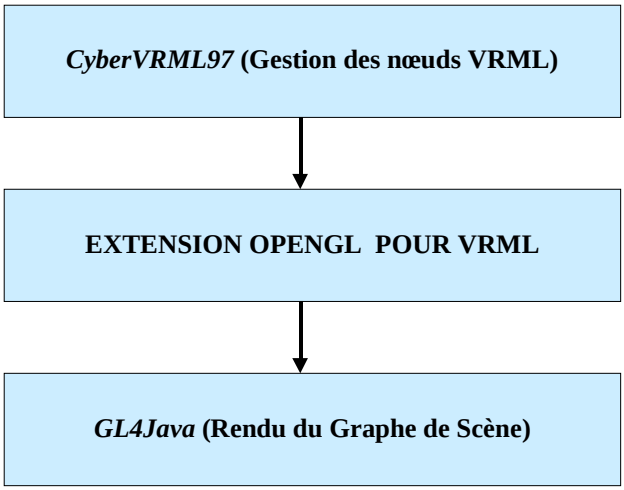


Fig. 16 - Architecture logicielle du Navigateur

2.3.1 – COMMENT GENERER UN ANALYSEUR AVEC *JAVACC [JCC]*

JavaCC est un compilateur qui génère de façon automatique un programme d'analyse syntaxique (*parser*). Un analyseur syntaxique est l'un des composants les plus connus dans une application informatique. Il convertit du texte lisible par des humains en structures de données connues sous le nom d'arbres d'analyse syntaxique compréhensibles par l'ordinateur.

JavaCC est écrit entièrement en Java. Ainsi, le fait de disposer entièrement des classes *JavaCC* signifie que cet outil est instantanément disponible sur toute plate-forme supportant la Machine Virtuelle Java. A l'exécution de *JavaCC*, il crée plusieurs fichiers sources Java

[19] Un schéma d'architecture avec des explications détaillées sont donnés en Annexe 4

dans le répertoire courant. Ces fichiers seront à leur tour compilés en Java pour obtenir un programme d'analyse syntaxique interprétable en Java. Dans la suite de ce document, nous nommerons un analyseur syntaxique simplement analyseur.

2.3.2 – COMMENT DECRIRE UNE GRAMMAIRE AVEC *JAVACC*

QUELQUES EXEMPLES DE BASE

Les fichiers de description de l'analyseur *JavaCC* ont l'extension .jj et sont divisés en trois parties :

Les exemples suivants sont issus de la documentation de *JavaCC*.
Exemple 1:

```
 options {
  LOOKAHEAD = 1;
  CHOICE_AMBIGUITY_CHECK = 2;
  OTHER_AMBIGUITY_CHECK = 1;
  STATIC = true;
  DEBUG_PARSER = false;
  DEBUG_LOOKAHEAD = false;
  DEBUG_TOKEN_MANAGER = false;
  ERROR_REPORTING = true;
  JAVA_UNICODE_ESCAPE = false;
  UNICODE_INPUT = false;
  IGNORE_CASE = false;
  USER_TOKEN_MANAGER = false;
  USER_CHAR_STREAM = false;
  BUILD_PARSER = true;
  BUILD_TOKEN_MANAGER = true;
  SANITY_CHECK = true;
  FORCE_LA_CHECK = false;
}

PARSER_BEGIN(Simple1)

public class Simple1 {

  public static void main(String args[]) throws ParseException
{
    Simple1 parser = new Simple1(System.in);
    parser.Input();
  }

}

PARSER_END(Simple1)
```

```
void Input() :
{}
{
  MatchedBraces() ("\n"|"\r")* <EOF>
}

void MatchedBraces() :
{}
{
  "{" [ MatchedBraces() ] "}"
}
```

Description détaillée de l'exemple Simple1.jj :

Il s'agit d'une grammaire *JavaCC* simple qui reconnaît un ensemble de d'accolades ouvrantes suivies par le même nombre d'accolades fermantes et finalement suivies par zéro ou plusieurs caractères de fin de ligne puis une fin de fichier.

Des exemples de chaînes de caractères correctes dans cette grammaire sont :

"{}", "{{{{{}}}}}", etc.

Exemples de chaînes illégales :

"{{{{", "{}{}", "{}}", "{{}{}}", etc.

Dans l'exemple Simple1, la classe dans laquelle l'analyseur est généré contient un programme principal (classe main). Celui-ci crée une instance de l'objet analyseur (un objet de type Simple1) en utilisant un constructeur qui prend un argument du type java.io.InputStream (ici, "System.in").

Le programme principal fait alors un appel au non-terminal dans la grammaire qu'il souhaite analyser - "Input" dans notre cas. Tous les non-terminaux ont un statut équivalent dans un analyseur généré par *JavaCC*, par conséquent, nous pouvons analyser par rapport à n'importe quel non-terminal de la grammaire.

Ensuite, vient la liste des productions. Dans cet exemple, il y a deux productions qui définissent les non-terminaux "Input" et "*MatchedBraces*" respectivement.

Nous détaillerons plus loin la façon dont les productions sont décrites en *JavaCC*. (*)

Les symboles (*tokens*) lexicaux ou expressions régulières sont soit des chaînes de caractères simples ("{", "}", "\n", "\r" dans notre exemple), ou une expression régulière plus complexe.

Dans notre exemple, une telle expression est <EOF>, qui correspond à une fin de fichier. Toutes les expressions régulières complexes sont entourées par des caractères "< >".

La première production de l'exemple exprime que le non-terminal "Input" se développe par le non-terminal "*MatchedBraces*" suivi par zéro ou plusieurs terminateurs de ligne ("\n" ou "\r") et la fin de fichier.

La seconde production exprime que le non-terminal "*MatchedBraces*" se développe par le symbole "{" suivi par un développement optionnel imbriqué de "*MatchedBraces*" suivi par le symbole "}". Des crochets […] dans un fichier de grammaire *JavaCC* indiquent que son contenu est optionnel.

Les autres structures pouvant apparaître dans les développements de productions sont :

e1 | e2 | e3 | …	:	un choix de e1, e2, e3, etc.
(e) +	:	une ou plusieurs occurrences de e
(e) *	:	zéro ou plusieurs occurrences de e

Pour construire cet analyseur, on exécute simplement *JavaCC* sur ce fichier et on compile les fichiers java résultants :

```
javacc Simple1.jj
javac *.java
```

Nous sommes alors capables d'exécuter notre nouvel analyseur en invoquant simplement :

```
java Simple1
```

Il suffit alors de saisir une séquence d'accolades correspondantes suivies par un retour chariot et une fin de fichier (CTRL Z sous Windows ou CTRL D sur les machines UNIX).

Voici quelques exemples d'exécution :

```
% java Simple1
{{}}<return>
 <control-d>
%
% java Simple1
 {x<return>
 Lexical error at line 1, column 2. Encountered: "x"
ParseError
          at
     Simple1TokenManager.getNextToken(Simple1TokenManager.java:
     207)
          at Simple1.getToken(Simple1.java:96)
          at Simple1.MatchedBraces(Simple1.java:238)
          at Simple1.Input(Simple1.java:231)
          at Simple1.main(Simple1.java:5)
%

% java Simple1
{}}<return>
Parse error at line 1, column 3. Encountered:
     "}"
```

```
     Was expecting one of:
          <EOF>
          "\n" ...
          "\r" ...

ParseError
          at Simple1.jj_consume_token(Simple1.java:63)
          at Simple1.Input(Simple1.java:232)
          at Simple1.main(Simple1.java:5)

%
```

LES PRINCIPALES OPTIONS DE COMPILATION DE JAVACC

Les options peuvent être spécifiées soit dans le fichier grammaire, soit directement sur la ligne de commande, auquel cas, elles sont prioritaires.

LOOKAHEAD: nombre de symboles à examiner par avance avant de prendre une décision durant l'analyse. Sa valeur par défaut est égale à 1. Plus ce nombre sera petit, plus rapide sera l'analyseur. Ce nombre peut être modifié pour des productions spécifiques.

STATIC: option de type booléen dont la valeur par défaut est vraie (*true*). Dans ce cas, toutes les méthodes et variables de classes sont déclarées statiques dans l'analyseur généré ainsi que dans le gestionnaire de symboles (*token manager*). Cela n'autorise qu'un seul objet analyseur à être présent, mais cela améliore l'exécution de l'analyseur. Pour exécuter plusieurs analyses pour une exécution du programme Java, il faudra appeler la méthode *ReInit()* pour réintialiser l'analyseur s'il est statique. Si celui-ci est non-statique, on peut utiliser l'opérateur "*new*" pour construire autant d'analyseurs que l'on désire. Ils peuvent tous être utilisés simultanément à partir de différentes threads.

DEBUG_PARSER: option de type booléen dont la valeur par défaut est fausse (*false*). Cette option est utilisée pour obtenir une information de déboggage sur l'analyseur généré. En positionnant cette option à vrai (*true*) elle entraîne l'analyseur à générer une trace de ses actions. Cette trace peut être désactivée par appel de la méthode *disable_tracing()* ou réactivée par la méthode *enable_tracing()* dans la classe de l'analyseur généré.

DEBUG_TOKEN_MANAGER: option de type booléen dont la valeur par défaut est fausse. Cette option est utilisée pour obtenir une information de déboggage sur le gestionnaire de symboles (token manager). En positionnant cette option à vrai, elle entraîne celui-ci à générer une trace de ses actions. Cette trace pouvant être importante, elle ne doit être positionnée que pour des erreurs lexicales signalées, que l'on ne peut pas résoudre simplement.

FORCE_LA_CHECK: option de type booléen dont la valeur par défaut est fausse. Cette option contrôle la vérification de l'ambiguïté de l'instruction "*lookahead*" exécutée par *JavaCC*. Par défaut, (quand cette option est fausse), cette vérification est exécutée pour tous les points de décision où un *lookahead* par défaut de 1 est utilisé. La vérification de l'ambiguïté n'est pas exécutée aux points de décision où il y a une spécification explicite du *lookahead* ou si l'option *LOOKAHEAD* est positionnée à une valeur supérieure à 1.

En positionnant cette option à vrai, cela provoque la vérification de l'ambiguïté du *lookahead* pour tous les points de décision, indépendamment des spécifications du *lookahead* dans le fichier grammaire.

IGNORE_CASE: option de type booléen dont la valeur par défaut est fausse. En positionnant cette option à vrai, elle entraîne le gestionnaire de symboles à ignorer les minuscules/majuscules dans les spécifications des symboles et dans les fichiers d'entrée. Ceci est utile pour écrire des grammaires telles que HTML. Il est aussi possible de localiser l'effet de *IGNORE_CASE* en utilisant un autre mécanisme que nous verrons plus loin.

LA CLASSE DE BASE DE L'ANALYSEUR

Les deux balises *PARSER_BEGIN* et *PARSER_END* entourent la classe qui deviendra le code de base Java pour l'analyseur résultant.
Le nom de la classe utilisée dans la spécification de l'analyseur doit être le même au début, au milieu et à la fin de cette section. Comme on peut le voir sur l'exemple, cette classe définit une méthode main statique de telle façon que la classe puisse être appelée par l'interpréteur Java sur la ligne de commande. La méthode main instancie simplement un nouvel analyseur avec un flot d'entrée (dans l'exemple, *System.in*) et appelle alors la méthode Input. La méthode Input est un non-terminal dans notre grammaire.

Cette méthode est définie sous la forme d'un élément E.B.N.F., qui signifie « *Extended Backus-Naur Form* ».

Le fichier grammaire commence par une liste d'options. Ceci est suivi par une unité de compilation Java entourée par "*PARSER_BEGIN*(nom)" et "*PARSER_END*(nom)". Ensuite vient une liste de règles de grammaire.

Le nom qui suit *"PARSER_BEGIN"* et *"PARSER_END*" doit être identique. Il identifie le nom de l'analyseur généré. Par exemple, si le nom est "*MyParser*", les fichiers suivants seront générés :

MyParser.java:	L'analyseur généré.
MyParserTokenManager.java:	Le gestionnaire des symboles (ou analyseur lexical).
MyParserConstants.java:	Un groupe de constantes utiles.

JavaCC n'exécutant pas de contrôles détaillés sur l'unité de compilation, il est ainsi possible pour un fichier grammaire de passer à travers *JavaCC* et de générer des fichiers sources Java produisant des erreurs de compilation.
Si l'unité de compilation contient une déclaration de package, celle-ci sera incluse dans l'analyseur généré. Le fichier de l'analyseur contient alors l'intégralité de l'unité de compilation ainsi que le code se trouvant à la fin de la classe de l'analyseur.

L'analyseur généré contient une déclaration de méthode "public" correspondant à chaque non-terminal (voir javacode_production et bnf_production) dans le fichier grammaire. L'analyse par rapport à un non-terminal est réalisée par l'appel à la méthode correspondant à ce non-terminal. A la différence de yacc, il n'y a pas un symbole de départ unique dans *JavaCC* : on peut analyser par rapport à n'importe quel non-terminal dans la grammaire.

Voici la BNF de notre exemple Simple1 :

```
      Input     ::=  MatchedBraces "\n" <EOF>
MatchedBraces ::=  "{" [ MatchedBraces ] "}"
```

Les mots en italiques représentent les non-terminaux sur la partie gauche des productions et les mots en gras montrent les littéraux. Comme nous l'avons déjà décrite, la grammaire analyse simplement des ensembles correspondant d'accolades ouvrantes et fermantes : '{' et '}'. Il y a deux productions dans le fichier *JavaCC* pour décrire cette grammaire. Le premier terminal, Input, est défini comme étant composé de trois éléments en séquence : un terminal *MatchedBraces,* un caractere de nouvelle ligne "\n" et un symbole de fin de fichier <EOF>, de telle façon que nous n'avons pas à le spécifier pour une plate-forme particulière.

Quand cette grammaire est générée, les côtés gauches des productions sont transformés en méthodes dans la classe Simple1. Après avoir été compilée, la classe Simple1 lit des caractères à partir de *System.in* et vérifie alors qu'ils contiennent un ensemble correspondant d'accolades ouvrantes et fermantes. Ceci est accompli par invocation de la méthode Input qui est transformée par le processus de génération en une méthode qui analyse un non-terminal Input. Si l'analyse échoue, la méthode lance l'exception *ParseError* qui peut être récupérée par la routine main pour personnaliser un message d'erreur.

De plus, le bloc délimité par {} après le terminal (qui est vide dans cet exemple) peut contenir n'importe quel code Java qui sera inséré en-tête de la méthode générée. Ensuite, après chaque expansion, il y a un autre bloc optionnel qui pourra également contenir du code Java pouvant être exécuté si l'analyseur analyse avec succès cette expansion.

LES REGLES DE PRODUCTION EN B.N.F.
(BACKUS-NAUR FORM)

La B.N.F. est une méthode qui spécifie les grammaires à contexte libre. La spécification consiste en un terminal sur le côté gauche, un symbole de production, typiquement : := , et une ou plusieurs productions sur le côté droit :

```
Motclef ::= "if" | "then" | "else"
```

Ceci doît être lu comme étant : « Le terminal Motclef est l'une des chaînes de caractères littérales if, then ou else ».

En *JavaCC*, cette forme a été étendue pour permettre à la partie gauche d'être représentée par une méthode et les expressions conditionnelles par des expressions régulières ou autres non-terminaux.

Voici la spécification B.N.F. d'un programme *JavaCC* :

```
javacc_input  ::=  Options JavaCC
                  "PARSER_BEGIN" "(" <IDENTIFICATEUR> ")"
                   java_compilation_unit
```

```
                    "PARSER_END" "(" <IDENTIFICATEUR> ")"
                    ( production )*
                    <EOF>


production  ::= javacode_production
              |   regular_expr_production
              |   bnf_production
              |   token_manager_decls
```

Il existe quatre types de productions dans *JavaCC* :
<javacode_production> et <bnf_production> sont utilisées pour définir la grammaire dont est issu l'analyseur généré.

<regular expr production> est utilisé pour définir les symboles (tokens) de la grammaire. Le gestionnaire de symboles (token manager) est généré à l'aide de cette information.

Nous ne détaillerons pas ici les parties <javacode_production> et <token_manager_decls> qui n'ont pas été utilisées pour générer la grammaire VRML97.

```
bnf_production::=   Java_return_type java_identifier "("
                    java_parameter_list ")" ":"
                    Java_block
                    "{" expansion_choices "}"
```

La BNF est la production standard utilisée pour spécifier les grammaires *JavaCC*. Chaque production en BNF a un côté gauche qui correspond à une spécification de non-terminal. La production en BNF définit alors ce non-terminal en terme d'expansions BNF sur le côté droit.

Le non-terminal est écrit exactement de la même manière qu'une méthode est déclarée en Java. Du fait que chaque non-terminal soit converti en une méthode dans l'analyseur généré, ce style d'écriture du non-terminal rend cette association évidente. Le nom du non-terminal est le nom de la méthode et les paramètres et valeurs de retour déclarées sont le moyen de passer les valeurs de haut en bas de l'arbre d'analyse.

Comme il sera vu plus loin, les non-terminaux sur les côtés droits des productions sont écrits comme des appels de méthodes, ainsi le passage de valeurs de haut en bas de l'arbre est réalisé en utilisant exactement le même paradigme que l'appel et retour de méthode.

Il y a deux parties sur le côté droit d'une production en BNF :

- la première partie est un ensemble de diverses déclarations et code Java (le bloc Java). Ce code est généré au début de la méthode elle-même générée pour le non-terminal Java. Par conséquent, a chaque fois que ce non-terminal est utilisé dans le procesus d'analyse, ces déclarations et ce code sont exécutés. Les déclarations à cet endroit sont visibles par tout

le code Java participant aux expansions BNF. *JavaCC* ne réalise aucun traitement de ces déclarations ou du code, sauf de se positionner à la parenthèse fermante correspondante. De ce fait, un compilateur Java peut détecter des erreurs dans ce code de *JavaCC*.

- la seconde partie de la partie droite sont les expansions BNF. Ceci sera décrit plus tard.

```
regular_expr_production      ::=    [ lexical_state_list ]
                                    regexpr_kind [ "[" "IGNORE_CASE" "]" ] ":"
                                    "{" regexpr_spec ( "|" regexpr_spec )* "}"
```

<*regular_expr_production*> est utilisé pour définir des entités lexicales qui seront traitées par le gestionnaire de symboles. Elle commence avec une spécification des états lexicaux pour lesquels ils s'appliquent (lexical_state_list). Il existe un état lexical standard appelé "*DEFAULT*". Si <lexical_state_list> est omis, <*regular_expr_production*> s'applique à l'état lexical "*DEFAULT*".

Ensuite, vient une description du genre d'expression régulière dont il s'agit (voir plus loin). Puis il y a un "[*IGNORE_CASE*]" optionnel. S'il est présent, l'expression de production régulière ne différencie pas les minuscules/majuscules : il a le même effet que l'option *IGNORE_CASE* sauf que dans ce cas il s'applique localement à cette expression de production régulière. Ceci est alors suivi par une liste de spécifications d'expression régulières qui décrit plus en détail les entités lexicales de cette expression de production régulière.

```
lexical_state_list ::=  "<" "*" ">"
                    |   "<" java_identifier ( "," java_identifier
                        )* ">"
```

<*lexical_state_list*> décrit l'ensemble des états lexicaux pour lesquels l'expression de production régulière correspondante s'applique. Si elle est écrite comme étant "<*>", la production d'expression régulière s'applique à tous les états lexicaux. Autrement, elle s'applique à tous les états lexicaux dans la liste d'identificateurs entre crochets.

```
regexpr_kind ::= "TOKEN"
          | "SPECIAL_TOKEN"
          | "SKIP"
          | "MORE"
```

< *regexpr_kind*> spécifie le genre de production d'expression régulière. Ils sont au nombre de quatre :

- *TOKEN*: les expressions régulières dans cette production décrivent des *symboles* dans la grammaire. Le gestionnaire de symboles crée un objet Token pour chaque correspondance d'une telle expression régulière et la retourne à l'analyseur.

- *SPECIAL_TOKEN:* les expressions régulières dans cette production décrivent des *symboles spéciaux*. Ils sont comme des symboles sauf qu'ils n'ont pas de signification durant l'analyse, c'est à dire que les productions BNF les ignorent. Ils peuvent être utiles dans le traitement d'entités lexicales comme les commentaires.

- *SKIP*: les correspondances avec des expressions régulières dans cette production sont simplement ignorées par le gestionnaire de symboles (appelé anciennement IGNORE_IN_BNF).

- *MORE*: n'est pas utilisé dans le contexte de la génération d'une grammaire VRML97.

SPECIFICATION DE TOKENS LEXICAUX

Exemple Simple3.jj :

```
PARSER_BEGIN(Simple3)
public class Simple3 {
  public static void main(String args[]) throws ParseException
  {
    Simple3 parser = new Simple3(System.in);
    parser.Input();
  }
}
PARSER_END(Simple3)
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}
TOKEN :
{
  <LBRACE: "{">
| <RBRACE: "}">
}

void Input() :
{ int count; }
{
  count=MatchedBraces() <EOF>
  { System.out.println("The levels of nesting is " + count); }
}
```

```
int MatchedBraces() :
{ int nested_count=0; }
{
  <LBRACE> [ nested_count=MatchedBraces() ] <RBRACE>
  { return ++nested_count; }
}
```

Cet exemple illustre l'utilisation du mot clé "*TOKEN*" pour spécifier des *tokens* lexicaux. Dans notre exemple, "{" et "}" sont définis comme des *tokens* auxquels on attribue les noms respectifs de "*LBRACE*" et "*RBRACE*". Ces noms peuvent alors être utilisés à l'intérieur de crochets pour les référencer. De telles spécifications sont utilisées pour des *tokens* complexes tels que des identificateurs ou des littéraux.

2.3.3 – PRESENTATION DU LOGICIEL *CyberVRML97*

CyberVRML97 pour Java est un paquetage développé par Satoshi Konno [SKO99]. Celui-ci est capable de gérer complètement un graphe de scène VRML97(ou autre format tel que X3D) et de réaliser un rendu de la scène à l'aide de Java3D. Ce développement s'adresse plutôt à des programmeurs. Il se présente sous la forme d'une hiérarchie de classes respectant la structure des composants VRML. Le logiciel *CyberVRML97* est disponible sur le site de Satoshi Konno à l'adresse suivante : http://www.cybergarage.org. On y trouve notamment les fichiers sources et classes Java du paquetage cv97. C'est la version 1.3 qui a été utilisée pour développer le navigateur.

2.3.4 – STRUCTURE DE *CyberVRML97*

HIERARCHIE DES CLASSES

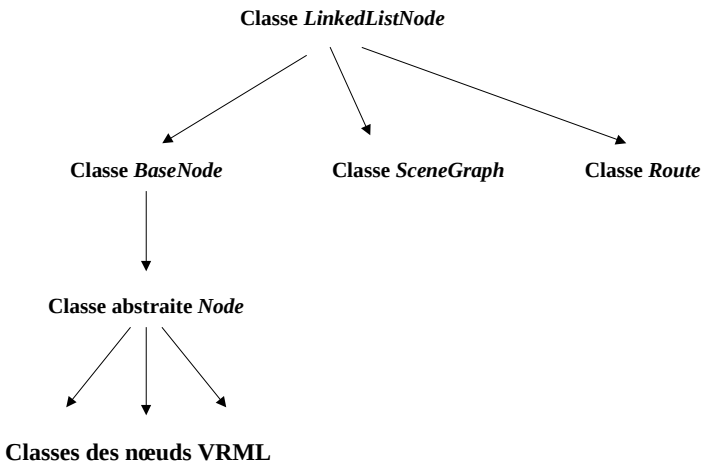


Fig. 23 - Schema general des classes de *CyberVRML97*

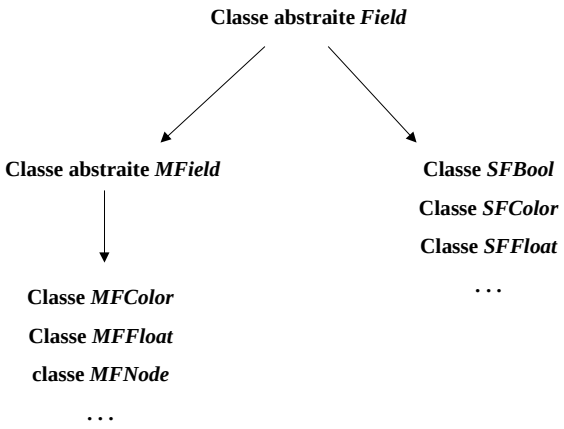


Fig. 24 - Schema de la classe *Field*

LA CLASSE SceneGraph

Le paquetage de la version 1.3 (05 janv. 2000) est nommé cv97. A la racine de celui-ci se trouve le fichier *SceneGraph.java* capable de charger un fichier d'entrée de données en fonction de sa nature (VRML, X3D, DXF, 3DS). Le parseur correspondant est alors appelé pour opérer une analyse syntaxique et lexicale sur ces données. Si cette étape ne signale pas d'erreur, les nœuds sont mémorisés dans des structures adéquates. Après le chargement, l'instance de *SceneGraph* dispose des informations des nœuds en utilisant des instances de la classe *Node*, qui est une super-classe de toutes les classes de nœuds vrml.
Il n'est pas fourni de programme standard pour instancier un graphe de scène. Nous verrons plus loin comment procéder.

La classe SceneGraph permet de gérer des Entrées/Sorties de fichiers de données :
Les méthodes :

```
load (char filename)
add (char filename)
save (char filename)
```

permettent respectivement de charger, d'ajouter ou de sauvegarder un fichier de données au format spécifié, ou bien encore de faire des conversions de format (charger un fichier au format VRML et le sauvegarder au format X3D).

Il existe en outre des méthodes permettant :

- l'ajout d'un nœud : *void addNode (Node node)*
- l'obtention d'une liste des nœuds : *Node getNodes()*
- l'obtention d'un nœud spécifique : *BoxNode getBoxNodes()*
- l'obtention d'une liste de nœuds : *CylinderNode findCylinderNode()*
- la mise à jour du graphe de scène : *void update()*

Autres méthodes utiles :
Méthode permettant de référencer le nœud racine :

```
RootNode getRootNode()
```

Méthode permettant d'associer un format d'entrée à une extension de fichier :

```
public static int getFileFormat(String fileName) {
     if (fileName == null)
          return FILE_FORMAT_NONE;
     int idx = fileName.lastIndexOf('.');
     if (idx < 0)
          return FILE_FORMAT_NONE;
     String ext = new String(fileName.getBytes(), (idx+1),
          fileName.length() - (idx+1));
     int formatType = FILE_FORMAT_NONE;
     if (ext.equalsIgnoreCase("WRL") == true)
          formatType = FILE_FORMAT_WRL;
     if (ext.equalsIgnoreCase("3DS") == true)
          formatType = FILE_FORMAT_3DS;
     …

     if (ext.equalsIgnoreCase("X3D") == true)
          formatType = FILE_FORMAT_X3D;
     if (ext.equalsIgnoreCase("XML") == true)
          formatType = FILE_FORMAT_X3D;
     Debug.message("File format = " + formatType);
     return formatType;
}
```

LA CLASSE Node

La classe *Node* est une classe abstraite, on ne peut donc pas créer directement des instances de *Node*. Cette classe dérive de la classe *LinkedListNode*. Ainsi, tous les nœuds sont gérés par une structure de liste, et une instance de nœud peut avoir d'autres instances de nœuds fils.
Tous les nœuds de VRML sont représentés dans cette classe, soit directement, soit indirectement par des classes abstraites, selon une structure hiérarchique que l'on retrouve dans la nature des nœuds VRML :

Classes abstraites dérivant de *Node* :

BindableNode
GeometryNode
GroupingNode
InterpolatorNode
LightNode
TextureNode

Il n'est donc possible d'instancier que les nœuds dérivés de ces classes abstraites. A titre d'exemple, détaillons les nœuds rattachés à *GeometryNode* :

GeometryNode
BoxNode
ConeNode
CylinderNode
ElevationGridNode
ExtrusionNode
IndexedFaceSetNode
IndexedLineSetNode
PointSetNode
SphereNode
TextNode

Tous ces nœuds, s'ils existent, apparaissent après un bloc *Geometry*, c'est pourquoi ils dérivent tous de la classe *GeometryNode*.

Détaillons à présent le contenu d'un nœud, par exemple *BoxNode* :

Ce nœud est représenté par le fichier *BoxNode.java* situé dans le répertoire cv97/node.
Ce fichier source est structuré de la façon suivante :

- Un constructeur *BoxNode()* appelé dans le parseur quand un nœud Box est reconnu et doit être crée, positionne le type *Box* pour ce nœud, crée un champ *size* de type SFVec3f avec une valeur par défaut égale à (2.0, 2.0, 2.0). Ce champ *size* est alors enregistré comme étant un *ExposedField*.
- Un ensemble de méthodes correspondant à la gestion des différents champs du nœud, soit dans notre exemple :

  - Modification du champ *size* par différentes méthodes *void setSize(…)*
  - Accès au champ *size* par la méthode *void getSize(float value[])*
  - Accès individuel à l'un des éléments x, y ou z du champ *size* par l'une des méthodes *getX(), getY() ou getZ().*

Voici les principales méthodes de la classe *Node* :

Obtenir le nœud suivant dans la hiérarchie courante :
Node next()

Obtenir le nœud suivant du même type dans la hiérarchie courante :
Node nextSameType()

Obtenir le nœud suivant en le recherchant à partir du nœud parent (permet l'accès à tous les nœuds du graphe de scène) :

Node nextTraversal()

Obtenir le nœud suivant du même type en le recherchant à partir du nœud parent (permet l'accès à tous les nœuds de même type du graphe de scène) :

Node nextTraversalSameType()

Autres méthodes :

Obtenir le nombre de nœuds fils d'un nœud :

int getNChildNodes()

Obtenir le nœud fils n d'un nœud :

Node getChildNode(int n)

Chaque nœud de VRML 97 est représenté de façon normalisée dans la classe Node, sous forme d'un fichier java :

<Nom du nœud>Node.java

Exemple :

Nœud Box représenté par le fichier: BoxNode.java

LA CLASSE Field

La classe *Field* permet de représenter tous les types de champs de VRML 97.

La méthode *setType* permet de positionner le type du champ :

```
public void setType(String type) {
     if (type.equals("SFBool"))
          setType(fieldTypeSFBool);
     else if (type.equals("SFColor"))
          setType(fieldTypeSFColor);
     else if (type.equals("SFFloat"))
          setType(fieldTypeSFFloat);
     else if (type.equals("SFInt32"))
          setType(fieldTypeSFInt32);
     else if (type.equals("SFRotation"))
          setType(fieldTypeSFRotation);
     else
          …
     else
          setType(fieldTypeNone);
}
```

La méthode getTypeName permet d'associer le libellé correspondant au type de champ :

```
public String getTypeName() {
          switch (getType()) {
          // Field
          case fieldTypeSFBool          : return "SFBool";
          case fieldTypeSFColor         : return "SFColor";
          case fieldTypeSFFloat         : return "SFFloat";
          case fieldTypeSFInt32         : return "SFInt32";
          case fieldTypeSFRotation : return "SFRotation";
          case fieldTypeSFString        : return "SFString"
     …
}
```

LA CLASSE Route

Le constructeur de la classe Route est appelé quand une instruction *ROUTE* est rencontrée dans un fichier VRML. Celui-ci mémorise les nœuds *EventOut*, *EventIn*, les champs *EventOut*, *EventIn*. Voici les méthodes qu'il utilise :

```
void setEventOutNode(Node node)
void setEventInNode(Node node)
void setEventOutField(Field field)
void setEventInField(Field field)
```

La méthode *update()* permet, entre autre, de mémoriser le type des champs *EventIn* et *EventOut* correspondant. En voici un exemple :

```
          case fieldTypeSFFloat :
               {
                    SFFloat fieldOut = (SFFloat)eventOutField;
                    float value = fieldOut.getValue();
                    switch (eventInFieldType) {
                    case fieldTypeSFFloat :
                         {
                              SFFloat fieldIn =
                         (SFFloat)eventInField;
                              fieldIn.setValue(value);
                         }
                         break;
                    case fieldTypeConstSFFloat :
                         {
                              ConstSFFloat fieldIn =
                         (ConstSFFloat)eventInField;
                              fieldIn.setValue(value);
                         }
                         break;
                    }
               }
               break;
```

2.3.5 – GESTION DYNAMIQUE DU GRAPHE DE SCENE

Il est possible d'ajouter ou supprimer tous les nœuds VRML dynamiquement. Dans l'exemple suivant, un fichier VRML est chargé dans le graphe de scène, puis le premier nœud Box est supprimé, une nouvelle géometrie cone de couleur bleue est ajoutée et le graphe de scène est sauvegardé dans un autre fichier VRML :

```
SceneGraph sg ;
sg.load("test.wrl") ;

//Supprimer le premier nœud Box
BoxNode box = sg.findBoxNode() ;
If (box != null)
     box.remove() ;

//Ajouter un nœud cone
ShapeNode shape = new ShapeNode() ;
sg.addNode(shape) ;

AppearanceNode appearance = new AppearanceNode() ;
shape.addChildNode(appearance) ;

MaterialNode material = new MaterialNode() ;
material.setDiffuseColor(0.0f, 0.0f, 1.0f) ; //Bleu
appearance.addChildNode(material) ;

ConeNode cone = new ConeNode() ;
shape.addChildNode(cone) ;
cone.setBottomRadius(10.0) ;
sg.save("new_test.wrl") ;
```

2.3.6 – UTILISATION AVEC *JAVA3D*

Java3D [J3D] est une API de programmation graphique 3D de bas niveau pour le langage Java, basée sur le graphe de scène. Une API de bas niveau fournit des routines pour la création de géométries 3D dans une structure de graphe de scène qui soit indépendante de l'implémentation matérielle sous-jacente. L'API fournit la possibilité de compilation du graphe de scène et d'autres techniques d'optimisation. Java3D fonctionne sous Linux, Win32 et la plupart des systèmes d'exploitation UNIX. Il ne fonctionne pas encore sous MacOs. Ce qui est présenté dans ce paragraphe n'a pas fait l'objet d'expérimentation. Ces informations proviennent de la documentation de *CyberVRML97*.

Le paquetage est prévu pour dessiner les instances de formes avec leur comportement dans un graphe de scène avec une structure appelée Canvas3D de *Java3D*. Pour utiliser cette possibilité, il faut créer une instance de la classe vrml.j3d.SceneGraphJ3dObject avec une cible Canvas3D et déclarer l'instance comme objet de l'instance de la classe SceneGraph.

Exemple :

```
Public class ViewerJ3D extends Frame implements Constants {
Private SceneGraph mSceneGraph ;
Private SceneGraphJ3dObject mSceneGraphObject ;

Public ViewerJ3D() {
Super ("VRML Simple Viewer") ;
…
mSceneGraph = new SceneGraph(SceneGraph.NORMAL_GENERATION) ;
setLayout(new BorderLayout()) ;
Canvas3D c = new Canvas3D(null) ;
add("Center", c) ;
mSceneGraphObject = new SceneGraphJ3dObject(c,mSceneGraph) ;
mSceneGraph.setObject(mSceneGraphObject) ;
…
setSize(400,400) ;
show() ;
}
…
…
}
```

Dans cette version, seulement une liste restreinte de nœuds est supportée. Ce paquetage possède d'autres interfaces pour *Java3D*, VRML97Loader et VRML97Saver.
VRML97Loader charge un fichier au format VRML97 et retourne un graphe de scène *Java3D*.L'exemple suivant charge un fichier VRML97 et fournit le BranchGroup *Java3D* :

```
VRML97Loader loader = new VRML97Loader() ;
Loader.load("Sample.wrl") ;
BranchGroup branchGroup = loader.getBranchGroup() ;
```

Dans la version actuelle,*VRML97Loader* converti quelques nœuds VRML97 en nœuds *Java3D*.

VRML97Saver produit un fichier VRML97 à partir d'un graphe de scène *Java3D* :

```
BranchGroup j3dBranchGroup =  …
VRML97Saver saver = new VRML97Saver() ;
saver.setBranchGroup(j3dBranchGroup) ;
saver.save(outputFileName) ;
```

Dans la version actuelle,*VRML97Saver* converti quelques nœuds *Java3D* en nœuds VRML97.

2.3.7 – PRESENTATION DU LOGICIEL *GL4Java*

LA PROGRAMMATION JAVA-OPENGL

GL4Java a été développé en 1997 par Sven Goethel et Atilla Kolac [SGO99]. Ce développement avait pour objectif de clore leur diplôme de thèse sous la responsabilité du Professeur Dr Wolfgang Bunse de l'*Ecole Technique Supérieure de Bielefeld*. *GL4Java* est maintenu à ce jour par Sven Goethel. La version actuelle est la 2.8.2.0. L'intention de *GL4Java* est d'utiliser le langage Java comme plate-forme de programmation indépendante pour les applications utilisant OpenGL.
L'interface OpenGL pour Java a été commencée par Leo Chan qui implémenta la bibliothèque avec les J.N.I. (*Java Native Calls*) version 1.0.2. Le travail de Leo Chan fut repris par Adam King . Son développement *OpenGL4Java* pouvait être compilé en Java 1.1 et le rendu OpenGL se faisait à travers une fenêtre Java.
Tom Reilly participa au travail d'Adam King et le projet s'appela alors *Jogl* dont les points forts reposaient sur une auto-configuration puissante et des fonctions systèmes *X-Window* améliorées, supportant ainsi la plupart des systèmes *UNIX*. Un autre point fort de *Jogl* est le support de *Win32*. Les sources et les fichiers dll pré-compilés sont toujours distribués. *Jogl* est différent sur le plan de la convention de nommage.
Le projet *GL4Java* a finalement rejoint celui de *Jogl*. Les résultats actuels et les modifications futures de *Jogl* sont portés vers *GL4Java*. Comme *Jogl*, *GL4Java* utilise "*Java Native Interface*" (J.N.I.) 1.1.
Un des points importants de *GL4Java* est l'utilisation des conventions de nommage OpenGL, le support des fonctions *glu** et *glut**. *GL4Java* étend l'A.P.I. OpenGL avec une convention de nommage personnalisée. Des fonctions spécialisées de gestion de fenêtre comme *glXSwapBuffers* ont le préfixe *glj*, comme *gljSwap*. La classe *GLFrame* s'enregistre elle-même comme un *ComponentListener*, ainsi nous avons un gestionnaire d'événements en Java, comme *reshape* pour *glut*.

INSTALLATION DU LOGICIEL GL4JAVA [SGO99]

Il est possible d'obtenir le logiciel complet *GL4Java* sur le site http://www.jausoft.com. On y trouve notamment :
- les fichiers binaires pré-compilés pour chaque type de machine,
- des fichiers de démonstrations (sources et classes),
- les fichiers sources de *GL4Java*,
- la documentation,
- une liste de diffusion : http://gl4java.sourceforge.net

Il est également possible de régénérer les fichiers sources Java pour la partie J.N.I.

CONCEPTION DE *GL4JAVA*

APPEL NATIF A OPENGL

Les programmes en code natif sont écrits dans un langage de programmation avec lequel le compilateur est capable de générer du code binaire pour le processeur correspondant à la plate-forme. Ces programmes OpenGL utiliseront la bibliothèque correspondant à cette plate-forme : ils chargeront la bibliothèque et appelleront les procédures qui existent dans la bibliothèque chargée.

APPEL JAVA A OPENGL

Le programme Java fait appel à des méthodes natives. Les programmes Java utilisent la bibliothèque native OpenGL sur la plate-forme spécifique. Le programme Java doit charger la bibliothèque enveloppante qui, à son tour, chargera la bibliothèque OpenGL.
Tout comme la bibliothèque OpenGL, la bibliothèque enveloppante dépend de la plate-forme cible et sera chargée à l'exécution, ce que l'on appelle bibliothèque dynamique. L'application Java OpenGL indépendante de la plate-forme a besoin de la bibliothèque enveloppante spécifique à la plate-forme, et toutes les bibliothèques que celle-ci a besoin, par exemple la bibliothèque OpenGL. Pour pouvoir exécuter une application Java OpenGL, l'utilisateur doit avoir installé la bibliothèque OpenGL et la bibliothèque enveloppante.

APPEL J.N.I. A OPENGL

Correspondance des types de données :

Par le fait que Java doit passer ses données sans aucune perte de temps dans la conversion de ces données, nous avons besoin d'un modèle type de correspondance, où les types de données OpenGL [2] entrent dans les types Java et vice-versa. Le premier et important critère pour la correspondance d'un type est la taille en *bytes*. Le second critère est de trouver des types équivalents en Java, où il y a le moins de conversions à réaliser par la bibliothèque enveloppante et le programmeur Java-OpenGL lui-même.
Sauf pour les types OpenGL marqués, une correspondance fine du type a été trouvée. Tous les types non signés doivent correspondre en taille de byte à l'équivalent Java. Ainsi, un moulage spécial pour Java (*cast*) doit être fait par le programmeur Java-OpenGL.
Du fait que le programmeur Java-OpenGL utilise des tableaux, on génère la fonction *GL4Java* pour tous les types de tableaux Java pour réaliser la compatibilité avec le type *GLvoid* * (cf. tableau ci-dessous).

(a) : Pas de correspondance Java pour *unsigned*, utilisation de conversions supplémentaires
(b) : Pas de correspondance Java pour les pointeurs, utilisation de tableaux.

Type de donnée	Type C	Type JNI	Type Java	Type OpenGL
Entier 8 bits	caractère non signé	jboolean	boolean	GLboolean
Entier 8 bits	caractère signé	jbyte	byte	GLbyte
Entier 8 bits non signé	caractère non signé	jbyte (a)	byte (a)	GLubyte
Entier 16bits	short	jshort	short	GLshort
Entier 16 bits non signé	short non signé	jshort (a)	short (a)	GLushort
Entier 32 bits	int ou long	jint	int	GLint GLsizei
Entier 32 bits non signé	Entier non signé	jint (a)	int (a)	GLuint GLenum GLbitfield
Flottant 32 bits	float	jfloat	float	GLfloat GLclampf
Flottant 64 bits	double	jdouble	double	GLdouble GLclampd
Entier 32 bits	void *	jbyteArray (b) jshortArray (b) jintArray (b) jfloatArray (b) jdoubleArray (b) jbooleanArray (b) jlongArray (b)	byte [ ] (b) short [ ] (b) int [ ] (b) float [ ] (b) double [ ] (b) boolean [ ] (b) long [ ] (b)	GLvoid * . . .

Tableau 1 - Types de données OpenGL correspondant aux types Java [SGO99]

Génération des *J.N.I* :

La bibliothèque enveloppante qui est appelée par l'application Java est spécifiée par l'Interface Native Java *(Java Native Interface, J.N.I.).* Toutes les fonctions natives qui doivent être appelées par l'application Java doivent être déclarées comme fonctions natives dans une classe Java, par exemple :

public native void glClear(int mask) ;

Le compilateur *JNI* crée un fichier en-tête pour le langage de programmation C, par exemple *test.h*. Ce fichier d'en-tête peut être copié comme un patron (*template*) vers le fichier de définition C, soit par exemple, *test.c*. Ce fichier de définition peut être modifié pour ajouter les corps des fonctions dont on a besoin. Ces fonctions n'ont besoin d'appeler leur fonction C qu'en dehors de la bibliothèque OpenGL avec une conversion d'un type de donnée standard. Pour garantir que toutes les fonctions sont enveloppées par le J.N.I. et pour simplifier l'écriture de l'enveloppe appropriée de chaque fonction, Sven Goethel et Attila Kollac ont développé leur propre générateur de fonctions enveloppantes OpenGL, nommé *C2J*.
C2J est le générateur des fichiers J.N.I. et de *MSJDirect*. Il crée les déclarations natives Java et les fonctions C.
C2J prend en entrée un fichier en-tête Mesa OpenGL et génère les déclarations natives Java dans un fichier et les fonctions C intégrales dans un autre fichier.

C2J est écrit en E.B.N.F. (*Extended Backus Naur Form*) du générateur de parseurs *JavaCC* (cf. Par. 2.3.1). Le parseur C fourni avec *JavaCC* a été modifié pour écrire *C2J*. La syntaxe de C2J peut être examinée dans : *GL4Java/C2J/C2J.jj*.

2.3.8 – VRML INTERFACE AVEC OPENGL

INTRODUCTION

La description qui suit correspond à la majeure partie du développement ayant été réalisé dans le cadre de ce mémoire.
Le but recherché par l'extension du logiciel *CyberVRML97* est la possibilité de représenter des scènes VRML97 à l'aide de la bibliothèque graphique OpenGL. Pour cela, les opérations suivantes ont été réalisées :

- Extension de la hiérarchie des classes de *CyberVRML97* de manière à traduire la sémantique des nœuds VRML en instructions OpenGL.
- création d'un nouveau graphe de scène ayant comme référence l'instance de la classe *SceneGraph*,
- modification du logiciel *GL4Java* et intégration de celui-ci au paquetage cv97 de manière à envoyer des instructions OpenGL et réaliser ainsi le rendu des scènes 3D.

AJOUT DE CLASSES DANS CyberVRML97

J'ai fait le choix de conserver intacte l'architecture des classes de *CyberVRML97*, cela permettant d'offrir des fonctionalités supplémentaires tout en restant indépendant de la structure existante.
D'un point de vue pratique, ce développement se présente comme une extension du paquetage cv97, appelée *opengl*. De ce fait, tous les fichiers sources seront crées dans un répertoire appelé *opengl*, et auront comme déclaration :

```
package cv97.opengl ;
```

Nous désignerons l'architecture des classes existante par *Bloc des Données*, celle du paquetage *opengl* par *Bloc Affichage* et nous verrons plus loin la création d'un *Bloc Animation*.

LA CLASSE CanvasOpenGL

La classe *CanvasOpenGL* dérive de la classe *GLAnimCanvas* issue du logiciel *GL4Java* que nous décrirons plus loin. Cette classe est instanciée par la classe *SceneGraphOpenGL* décrite ci-dessous. Elle contient les méthodes suivantes :

Le constructeur *CanvasOpenGL()* :
Permet de créer une fenêtre de navigation.

La méthode *PreInit()* :
Permet de positionner des paramètres comme le double tampon (*Double Buffering*).

La méthode *Init()* :
Permet de démarrer la thread associée à *GL4Java* chargée du rendu des *frames*, de créer un contexte GLUT, de positionner un modèle de dégradé de Gouraud (*gl.glShadeModel(GL_SMOOTH*), de demander le test du tampon de profondeur *(gl.glEnable(GL_DEPTH_TEST),* de préciser un modèle de lumière (gl.glLightfv *(GL_LIGHT0, GL_POSITION, light))* et l'activer par défaut.

La méthode *Reshape(int w, int h)* :
Permet de redimensionner une fenêtre de navigation en récupérant ses paramètres de taille w et h.

La méthode *keyPressed (KeyEvent e)* :
Permet la gestion des touches du clavier, notamment :

A – Z	Rotation
W – Q	Rotation
Pg Up – Pg Down	Avancer / Eloigner
Flèches	Positionner

La méthode *mousePressed(MouseEvent e)* :
Permet la gestion de l'enfoncement du bouton de la souris.

La méthode *mouseReleased(MouseEvent e)* :
Permet la gestion du relachement du bouton de la souris.

La méthode *mouseMoved(MouseEvent e)* :
Permet la gestion du déplacement de la souris.

La méthode *mouseDragged(MouseEvent e)* :
Permet la gestion du déplacement de la souris avec le bouton enfoncé.

La méthode *test_select(int x, int y, String event)* :
Permet la gestion de la sélection d'un objet en trois dimensions (picking).

LA CLASSE SceneGraphOpenGL

La classe *SceneGraphOpenGL* est instanciée par la classe principale j4 : une première instanciation de la classe *SceneGraph* est d'abord réalisée par appel au constructeur de *SceneGraph* :

```
SceneGraph sceneGraph = new SceneGraph() ;
```

Le fichier des données d'entrée est ensuite chargé dans le graphe de scène :

```
sceneGraph.load(file0) ;
```

Ensuite, nous instancions notre nouveau graphe de scène avec la référence du Bloc de Données :

```
SceneGraphOpenGL sceneGraphOpenGL =
          new SceneGraphOpenGL(sceneGraph) ;
```

Il s'en suit un appel de constructeurs en cascade :

le constructeur de *SceneGraphOpenGL* fait appel à celui de sa superclasse *CanvasOpenGL* par l'instruction *super().* A son tour, celui-ci appelle le deuxième constructeur de la classe *CanvasOpenGL* avec des valeurs par défaut pour spécifier la taille de la fenêtre qui sera créée :

```
public CanvasOpenGL() {
     this(480,480);
}
```

Le constructeur appelé prend donc en paramètre w et h, largeur et hauteur de la fenêtre à créer. A son tour, il appelle le constructeur de sa superclasse *GLAnimCanvas*, qui fait partie intégrante du logiciel *GL4Java* , à l'aide de l'instruction super(w,h) :

```
public CanvasOpenGL(int w, int h) {
super(w,h);

...

}
```

Ce dernier appel a donc pour effet d'appeler le constructeur de la classe *GLAnimCanvas* :

```
public GLAnimCanvas( int width, int height )
        {
                super( width, height);
          …
        }
```

A son tour, celui-ci appelle le constructeur de *GLCanvas* :

```
public GLCanvas( int width, int height )
    {
     this(width, height, null, null);
    }
```

qui référence :

```
public GLCanvas( int width, int height,
                         String gl_Name,
                         String glu_Name
               )
    {
     super( );
     …
}
```

Enfin, c'est le constructeur de la classe *Canvas* de Java qui est appelé pour créer une fenêtre *Canvas*. Ce processus en cascade peut être résumé par le schéma suivant :

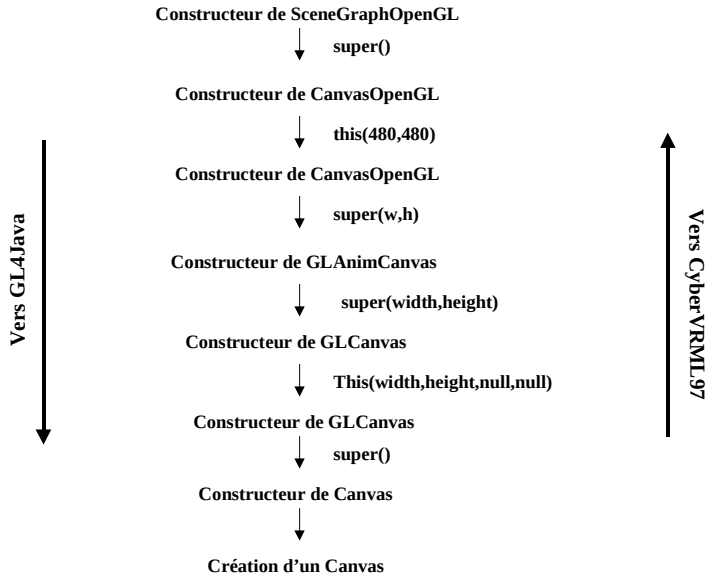


Fig. 25 – Cascade des constructeurs

A présent, une fenêtre *Canvas* est crée. Revenons maintenant au constructeur de *SceneGraphOpenGL*. Celui-ci référence le nœud racine du graphe de scène, par appel à la méthode getRootNode :

```
mRootNode = sg.getRootNode() ;
```

Celui-ci est alors sauvegardé dans la variable *node* :

```
node = mRootNode ;
```

Le constructeur de *SceneGraphOpenGL* se termine par le passage des instances des graphes de scènes (données et affichage) à la classe *CanvasOpenGL* par la méthode :

```
setSGGL(this, sg);
```

La méthode *display()* déclarée dans la classe *SceneGraphOpenGL* est une redéfinition de la méthode *display()* déclarée dans sa superclasse *GLAnimCanvas*. Ainsi, c'est celle de *SceneGraphOpenGL* qui sera appelée de façon périodique par *Repaint()* dans la classe *GLAnimCanvas* (logiciel *GL4Java*). En procédant ainsi, le graphe de scène est parcouru sans fin.

Principales fonctions de la méthode *display()* de la classe *SceneGraphOpenGL* :

- vérifier que le contexte graphique de *GL4Java* est bien crée,
- sauvegarder la matrice de transformation (*gl.glPushMatrix()*).
- assurer les mouvements de translation et rotation,
- appeler la méthode *display(Node node)* ayant comme paramètre le nœud courant qui appellera récursivement tous les nœuds du graphe de scène,
- restaurer la matrice de transformation (*gl.glPopMatrix()*),

- appeler la méthode de gestion du *double buffer* (*glj.gljSwap()*),
- effectuer le rendu de la scène.

Description de la méthode *display (Node node) :*

- parcours de tous les nœuds fils du nœud fourni en paramètre,
- gestion de l'interaction des *sensors* (cf. Par. 2.1.2),
- pour chaque nœud fils du nœud fourni en paramètre, appel de la méthode *display (Node node)* correspondant au type de nœud à traiter (*getNChildNodes()* donne le nombre de fils d'un nœud) :

```
public void display(Node node) {
     . . .
     for (i=0; i<node.getNChildNodes(); i++) {
                    Node child = node.getChildNode(i);

          if (child.isShapeNode()) {    . . .
               ShapeNodeOpenGL.display((ShapeNode)child,this);
               traite = true;
          }

          if (child.isTransformNode()) {

     TransformNodeOpenGL.display((TransformNode)child,this);
               traite = true;
                    }
          . . .
          if (!traite) display(child);
      }
}
```

LA CLASSE TransformNodeOpenGL

Gestion particulière du nœud *Transform* :
Le nœud *Transform* devant être appliqué à tous les nœuds englobés par *Children*, il est donc nécessaire que les transformations affectent ces nœuds. A cet effet, dans la procédure *TransformNodeOpenGL*, on rappelle la procédure *display (node)* de *SceneGraphOpenGL* pour chaque nœud fils de *Transform* :

```
public class TransformNodeOpenGL {

     public static void display (TransformNode node,
                    SceneGraphOpenGL sgGL) {

          sgGL.gl.glPushMatrix();
          …
          sgGL.display(node);
          …
          sgGL.gl.glPopMatrix();
     }
}
```

LA CLASSE ShapeNodeOpenGL

Cette classe permet la déclaration d'un nœud *Shape* ayant un bloc *Appearance* et *Geometry* :

```
public class ShapeNodeOpenGL {

public  void display(ShapeNode node, SceneGraphOpenGL sgGL) {
   Node appearance, geometry;
    appearance = (Node)(node.getAppearanceField().getValue());
    geometry   = (Node)(node.getGeometryField().getValue())     ;
    // Box , Cylinder , Cone , PointSet
    // Traitement de l'apparence
    if (appearance!=null) {
     if (appearance.isAppearanceNode()) {
     Node material =
(Node)(((AppearanceNode)appearance).getMaterialField().getValue());
     Node texture =
(Node)(((AppearanceNode)appearance).getTextureField().getValue());
      if (material != null)
        MaterialNodeOpenGL.display((MaterialNode)material,sgGL);
      if (texture != null) {
        /*      Non implémenté
        if (texture.isImageTextureNode()) {
ImageTextureNodeOpenGL.display((ImageTextureNode)texture,sgGL);
        }
        if (texture.isMovieTextureNode()) {
MovieTextureNodeOpenGL.display((MovieTextureNode)texture,sgGL);
        }
        if (texture.isPixelTextureNode()) {
PixelTextureNodeOpenGL.display((PixelTextureNode)texture,sgGL);
       }
        */
      }
    }
   }
    // Traitement de la géométrie
    if (geometry.isSphereNode()) {
       SphereNodeOpenGL.display((SphereNode)geometry, sgGL);
    }
    if (geometry.isBoxNode()) {
      BoxNodeOpenGL.display((BoxNode)geometry, sgGL);
    }
    if (geometry.isCylinderNode()) {
      CylinderNodeOpenGL.display((CylinderNode)geometry, sgGL);
    }
    if (geometry.isConeNode()) {
      ConeNodeOpenGL.display((ConeNode)geometry, sgGL);
    }
    if (geometry.isPointSetNode()) {
    }
    if (geometry.isIndexedFaceSetNode()) {
     IndexedFaceSetNodeOpenGL.display((IndexedFaceSetNode)geometry,
sgGL);
    }
  }
}
```

LA CLASSE BoxNodeOpenGL

D'après la norme VRML97 (cf. Par. 1.3), le nœud *Box* représente un parallélépipède rectangle centré en (0, 0, 0) dans le système de coordonnées local, dans l'alignement des axes de coordonnées locaux.
Par défaut, le cube mesure 2 unités dans chaque dimension, de –1 à +1. Le champ *size* spécifie les limites du cube le long des axes X, Y et Z.

Le nœud VRML de *Box* est par défaut le suivant :

```
Box {
     field SFVec3f size 2 2 2
}
```

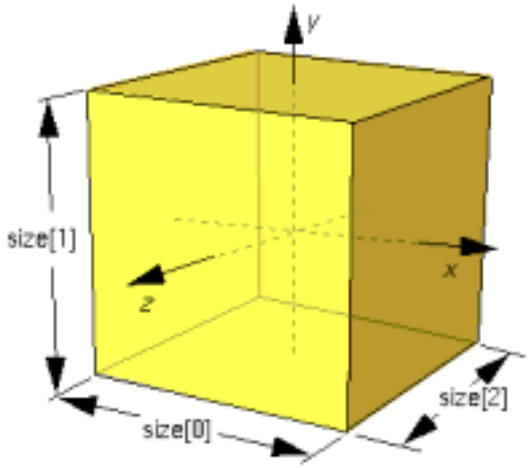


Fig. 26 – Parallélépipède [VRM97]

Pour implémenter ce nœud à l'aide d'OpenGL, il est nécessaire de décrire les six faces du cube par des polygones, de la façon suivante :

```
sgGL.gl.glBegin(sgGL.GL_POLYGON);

   // Face avant
   sgGL.gl.glNormal3f(0.0f, 0.0f, -1.0f);
sgGL.gl.glVertex3f(xmin,ymin,zmin);
sgGL.gl.glVertex3f(xmax,ymin,zmin);
sgGL.gl.glVertex3f(xmax,ymax,zmin);
sgGL.gl.glVertex3f(xmin,ymax,zmin);

   sgGL.gl.glEnd();

[etc.]
```

```
      sgGL.gl.glBegin(sgGL.GL_POLYGON);
         // Face arrière
         sgGL.gl.glNormal3f(0.0f, 0.0f, 1.0f);
         sgGL.gl.glVertex3f(xmin,ymin,zmax);
      sgGL.gl.glVertex3f(xmax,ymin,zmax);
      sgGL.gl.glVertex3f(xmax,ymax,zmax);
      sgGL.gl.glVertex3f(xmin,ymax,zmax);

      sgGL.gl.glEnd();
```

LA CLASSE ConeNodeOpenGL

Le nœud *Cone* spécifie une cône centré dans le système de coordonnées local et dont l'axe central est aligné avec l'axe des Y. Le champ *bottomRadius* spécifie le rayon de la base du cône et le champ *height* précise la hauteur du cône du centre de la base jusqu'au sommet. Par défaut, le cône a un rayon de 1.0 à la base et une hauteur égale à 2.0 avec son sommet en y = hauteur/2 et sa base en y= - hauteur/2. Le champ *side* précise si les côtés du cône sont crées et le champ *bottom* précise si le couvercle de la base est crée.

Le nœud VRML de *Cone* est par défaut le suivant :

```
Cone {
     field     SFFloat   bottomRadius 1
     field     SFFloat   height    2
     field     SFBool    side      TRUE
     field     SFBool    bottom    TRUE
}
```

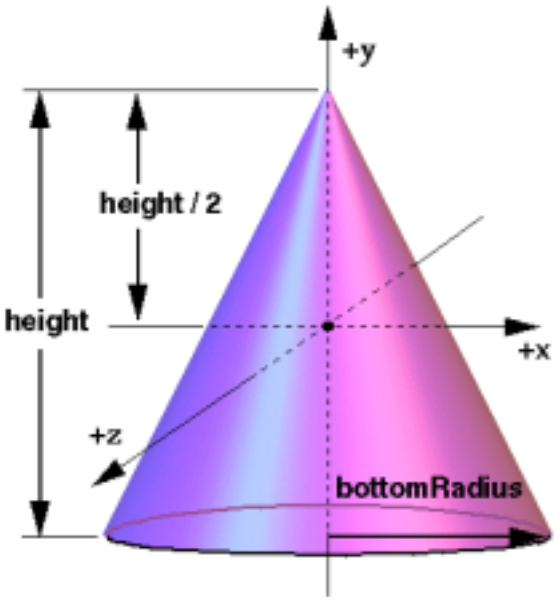


Fig. 27 – Cône [VRM97]

Implémentation du nœud sous OpenGL :

On récupère les paramètres rayon, base et hauteur :

```
radius=node.getBottomRadius();
height=node.getHeight();
```

On teste l'état des booléens *bottom* et *side* :

```
bottom = node.getBottom();
 side  = node.getSide();
```

On sauvegarde la matrice de transformation :

```
sgGL.gl.glPushMatrix();
```

Translation du cône d'une demi hauteur :

```
sgGL.gl.glTranslatef(0.0f, height/2, 0.0f);
```

Rotation de 90 degrés :

```
sgGL.gl.glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
```

Génération d'un quadrique :

```
qobj = sgGL.glu.gluNewQuadric();
sgGL.glu.gluQuadricDrawStyle  (qobj, CanvasOpenGL.GLU_FILL);
sgGL.glu.gluQuadricNormals    (qobj, CanvasOpenGL.GLU_SMOOTH);
```

Si le booléen *side* est vrai, dessiner le cône à l'aide d'une figure cylindrique dont une des bases a un rayon nul :

```
     if (side) {
         sgGL.glu.gluCylinder(qobj,0,radius,height,16,1);
     }
```

Si le booléen *bottom* est vrai, dessiner la base et la translater de la hauteur (height):

```
if (bottom) {
          sgGL.glu.gluQuadricOrientation(qobj,
                          CanvasOpenGL.GLU_INSIDE);
            sgGL.gl.glTranslatef(0.0f, 0.0f, height);
            sgGL.glu.gluDisk(qobj,0,radius,16,1);
}
```

Restaurer la matrice de transformation :

```
    sgGL.gl.glPopMatrix();
```

LA CLASSE CylinderNodeOpenGL

Le nœud *Cylinder* spécifie un cylindre couvert, centré en (0, 0, 0) dans le système de coordonnées local et avec un axe central orienté le long de l'axe des Y. Par défaut, le cylindre a une taille de –1 vers +1 dans les trois dimensions. Le champ *radius* spécifie le rayon du cylindre et le champ *height* spécifie la hauteur du cylindre le long de l'axe central. Les champs *radius* et *height* devront être supérieurs à 0. La figure suivante représente le nœud *Cylinder*. Le cylindre possède trois parties :

- le coté (*side*),
- le haut (*top*, Y = + *height*/2)
- le bas (*bottom*, Y = -*height*/2)

Chacune de ces parties est représentée par un booléen qui indique si la partie existe (*TRUE*) ou n'existe pas (*FALSE*). Les parties n'existant pas ne seront pas représentées sur le rendu de la figure et ne seront donc pas éligibles pour des tests d'intersection tels que l'activation d'un senseur.

Le nœud VRML de *Cylinder* est par défaut le suivant :

```
Cylinder {
     field SFBool   bottom TRUE
     field SFFloat  height 2
     field SFFloat  radius 1
     field SFBool   side      TRUE
     field SFBool   top  TRUE
}
```

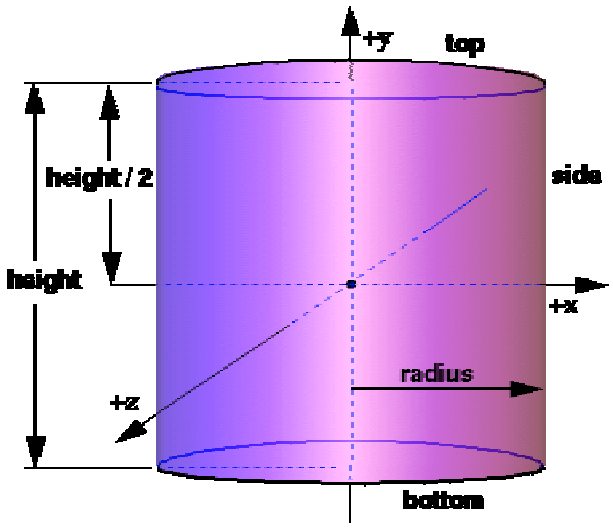


Fig. 28 – Cylindre [VRM97]

On teste l'état des booléens top, *bottom* et *side* :

```
top    = node.getTop();
side   = node.getSide();
bottom = node.getBottom();
radius=node.getRadius();
height=node.getHeight();
```

On sauvegarde la matrice de transformation :

```
sgGL.gl.glPushMatrix();
```

Génération d'un quadrique :

```
qobj = sgGL.glu.gluNewQuadric();
sgGL.glu.gluQuadricDrawStyle(qobj,  CanvasOpenGL.GLU_FILL);
sgGL.glu.gluQuadricNormals(qobj,  CanvasOpenGL.GLU_SMOOTH);
```

Translation du cône d'une demi hauteur :

```
sgGL.gl.glTranslatef(0.0f, height/2, 0.0f);
```

Rotation de 90 degrés :

```
sgGL.gl.glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
```

Si le booléen *side* est vrai, dessiner le cylindre à l'aide d'une figure cylindrique dont les deux bases on le même rayon *radius*, et une hauteur *height* :

```
if (side) {
                    sgGL.glu.gluCylinder(qobj,radius,radius,height,16,1);

};
```

Si le booléen *bottom* est vrai, dessiner la base du cylindre:

```
if (bottom) {
        sgGL.glu.gluQuadricOrientation(qobj, CanvasOpenGL.GLU_INSIDE);
        sgGL.glu.gluDisk(qobj,0,radius,16,1);

}
```

Si le booléen *top* est vrai, dessiner le haut et le translater de la hauteur du cylindre (height):

```
if(top) {
          sgGL.glu.gluQuadricOrientation(qobj, CanvasOpenGL.GLU_OUTSIDE);
        sgGL.gl.glTranslatef(0.0f, 0.0f, height);
        sgGL.glu.gluDisk(qobj,0,radius,16,1);
}
```

Restaurer la matrice de transformation :

```
sgGL.gl.glPopMatrix();
}
```

LA CLASSE SphereNodeOpenGL

Le nœud *Sphere* spécifie une sphère centrée en (0, 0, 0) dans le système de coordonnées local. Le champ *radius* spécifie le rayon de la sphère et devra être supérieur à 0.

```
Sphere {
  field SFFloat radius  1
}
```

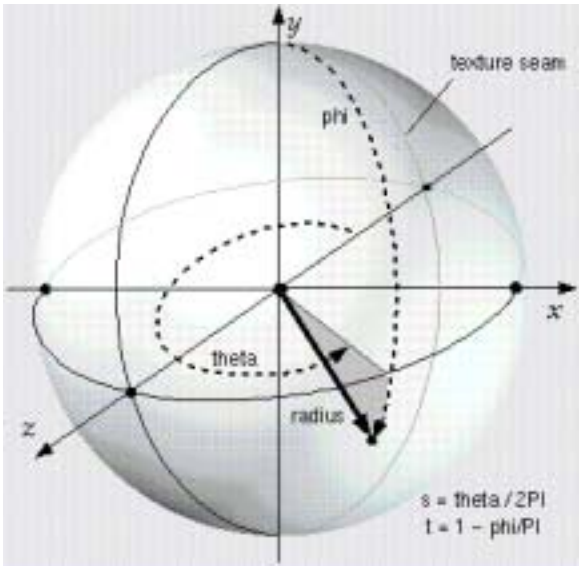


Fig. 29 – Sphère [VRM97]

LA CLASSE IndexedFaceSetNodeOpenGL

Cette classe, parmi les plus importantes de l'extension du logiciel *CyberVRML97*, permet l'affichage de figures à base de polyèdres.

Le nœud *IndexedFaceSet* représente une forme 3D composée de facettes (polygones) à partir de sommets listés dans le champ *Coord*. Celui-ci contient un nœud *Coordinate* qui définit les sommets 3D référencés par le champ *coordIndex*. Le nœud *IndexedFaceSet* utilise les sommets se son champ *coordIndex* pour définir les faces polygonales en indexant les coordonnées dans le nœud *Coordinate*.
Un index égal à –1 indique la fin de la face courante et le début de la suivante. La dernière face peut se terminer par un index égal à –1. Si l'index le plus grand dans le champ *coordIndex* est N, le nœud *Coordinate* contiendra N+1 coordonnées (indexées de 0 à N).

Chaque face du nœud *IndexedFaceSet* aura, conformément à la spécification:

- au moins trois sommets distincts,
- des sommets définissant un polygone dans le même plan.

Le nœud *IndexedFaceSet* est spécifié dans le système de coordonnées local et est affecté par les transformations de ses nœuds ancêtres.

Si le champ *color* est différent de *NULL*, il contiendra un nœud *Color* dont les couleurs seront appliquées aux sommets ou faces de *IndexedFaceSet* comme suit :

Si le champ *colorPerVertex* est égal à *FALSE*, les couleurs seront appliquées à chaque face, de la façon suivante :

- si le champ *colorIndex* n'est pas vide, alors une seule couleur est utilisée pour chaque face de *IndexedFaceSet*. Il y aura au moins autant de sommets dans le champ *colorIndex* qu'il y aura de faces dans *l'indexedFaceset*. Le champ *colorIndex* ne devra contenir aucune valeur négative.
- si le champ *colorIndex* est vide, alors les couleurs du noeud *Color* seront appliquées à chaque face du nœud *IndexedFaceSet* dans l'ordre. Il y aura au moins autant de couleurs dans le nœud *Color* qu'il y a de faces.

Si le champ *colorPerVertex* est égal à *TRUE*, les couleurs seront appliquées à chaque sommet, de la façon suivante :

- si le champ *colorIndex* n'est pas vide, les couleurs seront appliquées à chaque sommet de *IndexedFaceSet* exactement de la même façon que le champ *coordIndex* est utilisé pour choisir les coordonnées de chaque sommet du nœud *Coordinate*. Le champ *colorIndex* contiendra au moins autant d'indices que le champ *coordIndex* et contiendra des marqueurs de fin de face (-1) exactement aux mêmes endroits que dans le champ *coordIndex*.
- si le champ *colorIndex* est vide alors le champ *coordIndex* sera utilisé pour choisir les couleurs dans le nœud Color.

Si le champ *color* est égal à *NULL*, la géométrie sera rendue normalement en utilisant le nœud *Material* défini dans le nœud *Appearance*.

Si le champ *normal* est différent de *NULL*, il contiendra un nœud *Normal* dont les normales seront appliquées aux sommets ou faces de *IndexedFaceSet* exactement de la même manière que celle qui a été décrite ci-dessus pour appliquer les couleurs aux sommets ou faces, en remplaçant respectivement *colorPervertex* par *normalPervertex* et *colorIndex* par *normalIndex*.

Si le champ normal est égal à *NULL*, le navigateur générera alors automatiquement des normales en utilisant le paramètre *creaseAngle* pour déterminer si les normales doivent être lissées à travers des sommets partagés.

Voici la définition VRML du nœud *IndexedFaceSet* :

```
IndexedFaceSet {
eventIn MFInt32 set_colorIndex
eventIn MFInt32 set_coordIndex
eventIn MFInt32 set_normalIndex
eventIn MFInt32 set_texCoordIndex
exposedField SFNode color NULL
exposedField SFNode coord NULL
exposedField SFNode normal NULL
exposedField SFNode texCoord NULL
field SFBool ccw TRUE
field MFInt32 colorIndex []
field SFBool colorPerVertex TRUE
field SFBool convex TRUE
field MFInt32 coordIndex []
field SFFloat creaseAngle 0
field MFInt32 normalIndex []
field SFBool normalPerVertex TRUE
field SFBool solid TRUE
field MFInt32 texCoordIndex []
}
```

Exemple :

```
#VRML V2.0 utf8
#- KDO - Cours VRML : IndexedFaceSet
NavigationInfo {
type "EXAMINE"
headlight FALSE
}
     Shape {
          appearance Appearance {
               material Material {}
          }
     geometry IndexedFaceSet {
          solid FALSE
               coord Coordinate {
                    point [
                         -2 -2  2, -2 2  2, 2  2  2, 2 -2  2,
                         -2 -2 -2, -2 2 -2, 2  2 -2, 2 -2 -2
                    ]
               }
               coordIndex [
                    0 1 2 3 -1, 7 6 5 4 -1,
                    4 5 1 0 -1, 3 2 6 7 -1,
                    3 7 4 0 -1, 1 5 6 2
               ]
               colorPerVertex TRUE
               color Color {
                    color [
                         0 0 1, 0 1 0, 0 1 1, 1 0 0,
                         1 0 1, 1 1 0, 1 1 1, 0 1 0
```

```
                    ]
                }
        }
}
```

Dans le nœud *Coordinate*, on a défini huit sommets, numérotés de 0 à 7, dont les coordonnées sont les suivantes :

INDEX	COORDONNEE
0	-2 -2 2
1	-2 2 2
2	2 2 2
3	2 -2 2
4	-2 -2 –2
5	-2 2 –2
6	2 2 –2
7	2 -2 –2

Le nœud *CoordIndex* fait référence aux index défini dans le tableau ci-dessus :

0 1 2 3 –1 : définition d'une face.

0 est l'index du sommet -2 -2 2
1 est l'index du sommet -2 2 2
2 est l'index du sommet 2 2 2
3 est l'index du sommet 2 -2 2
-1 = fin de la face.

On génère donc un polygone à quatre sommets, dont les coordonnées sont celles qui ont été énoncées ci-dessus. La même opération est faite avec la facette dont les index sont 7 6 5 4, jusqu'à parcourir tous les éléments de *coordIndex*.
Le code OpenGL généré serait alors pour la première face :

```
glBegin(GL_POLYGON) ;
glVertex3f(-2.0 ,-2.0, 2.0) ;
glVertex3f(-2.0 , 2.0, 2.0) ;
glVertex3f( 2.0 , 2.0, 2.0) ;
glVertex3f( 2.0 ,-2.0, 2.0) ;
glEnd() ;
etc.
```

Le champ *colorPerVertex* étant égal à *TRUE*, on va alors générer une couleur par sommet. Comme le vecteur *colorIndex* est vide, c'est donc le champ *coordIndex* qui va être utilisé pour choisir les couleurs dans le nœud *Color*.

INDEX	COORDONNEE	COULEUR
0	-2 –2 2	0 0 1
1	-2 2 2	0 1 0
2	2 2 2	0 1 1
3	2 -2 2	1 0 0
4	-2 -2 –2	1 0 1
5	-2 2 –2	1 1 0
6	2 2 –2	1 1 1
7	2 -2 –2	0 1 0

S'il y avait dans cet exemple un vecteur *normalIndex*, on aurait procédé exactement de la même façon pour générer des vecteurs normaux par sommet. En l'absence de ce vecteur, une normale par face est générée.

Finalement, le code OpenGL généré pour une face est le suivant :

```
gl.glBegin(GL_POLYGON);
glColor3f(0.0,0.0,1.0);
glNormal3f(0.0,0.0,1.0) ;
glVertex3f(-2.0,-2.0,2.0);
glColor3f(0.0,1.0,0.0);
glVertex3f(-2.0,2.0,2.0);
glColor3f(0.0,1.0,1.0);
glVertex3f(2.0,2.0,2.0);
glColor3f(1.0,0.0,0.0);
glVertex3f(2.0,-2.0,2.0);
glEnd();
```

Présentation du calcul des vecteurs normaux à un polygone

Rappel sur le produit vectoriel

Nous savons que le produit vectoriel W=UxV de deux vecteurs U et V est un vecteur perpendiculaire à U et V ayant pour longueur :

$$\left|\overrightarrow{W}\right| = \left|\overrightarrow{U}\right| \cdot \left|\overrightarrow{V}\right| \cdot \sin\alpha$$

où α est l'angle (<π) entre U et V. Le trièdre (U, V, W) est dans le sens direct (fig. 30).

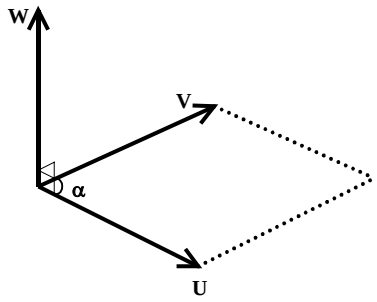


Fig. 30 – Produit vectoriel

Dans un repère cartésien direct (x, y, z), désignons les composantes des vecteurs U, V, W par :

$$\overrightarrow{U}\begin{Bmatrix} x_1 \\ y_1 \\ z_1 \end{Bmatrix} \qquad \overrightarrow{V}\begin{Bmatrix} x_2 \\ y_2 \\ z_2 \end{Bmatrix} \qquad \overrightarrow{W}\begin{Bmatrix} x_3 \\ y_3 \\ z_3 \end{Bmatrix}$$

Les composantes de W = U x V sont par définition données par :

$$\overrightarrow{W}\begin{Bmatrix} x_3 = y_1 z_2 - z_1 y_2 \\ y_3 = z_1 x_2 - x_1 z_2 \\ z_3 = x_1 y_2 - y_1 x_2 \end{Bmatrix}$$

Calcul du vecteur normal

Nous présentons ci-dessous le détail du calcul d'un vecteur normal à une surface. Considérons le triangle formé par les sommets P_1, P_2, P_3. Nous allons calculer le vecteur normal aux vecteurs $V_1=P_1P_2$ et $V_2=P_1P_3$. Considérons également un repère de centre O :

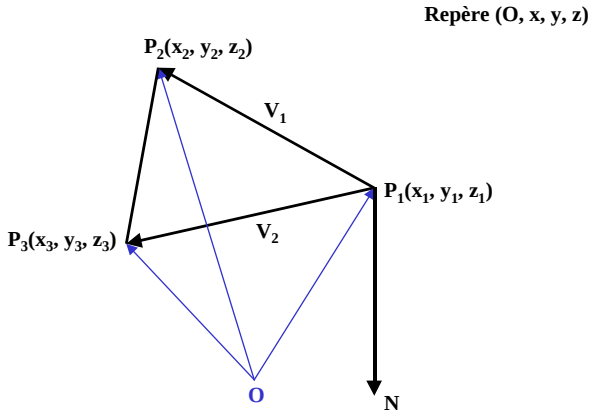


Fig. 31 - Calcul du Vecteur normal $N = P_1P_2 \times P_1P_3$

On note :

$$\overrightarrow{OP_1}\begin{Bmatrix} x_1 \\ y_1 \\ z_1 \end{Bmatrix} \qquad \overrightarrow{OP_2}\begin{Bmatrix} x_2 \\ y_2 \\ z_2 \end{Bmatrix} \qquad \overrightarrow{OP_3}\begin{Bmatrix} x_3 \\ y_3 \\ z_3 \end{Bmatrix}$$

$$\overrightarrow{V_1} = \overrightarrow{P_1P_2} = \overrightarrow{P_1O} + \overrightarrow{OP_2}$$

$$= -\overrightarrow{OP_1} + \overrightarrow{OP_2}$$

$$= \overrightarrow{OP_2} - \overrightarrow{OP_1}$$

qui a pour composantes :

$$\begin{Bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{Bmatrix}$$

D'où

$$\overrightarrow{V_1} = \overrightarrow{P_1P_2} \begin{Bmatrix} X_1 = x_2 - x_1 \\ Y_1 = y_2 - y_1 \\ Z_1 = z_2 - z_1 \end{Bmatrix}$$

De même pour les composantes de P_1P_3 (vecteur V_2) = $OP_3 - OP_1$:

$$\overrightarrow{V_2} = \overrightarrow{P_1P_3} \begin{Bmatrix} X_2 = x_3 - x_1 \\ Y_2 = y_3 - y_1 \\ Z_2 = z_3 - z_1 \end{Bmatrix}$$

Alors, d'après le rappel ci-dessus, on peut calculer les composantes de $N = V_1 x V_2$:

$X_3 = Y_1Z_2 - Z_1Y_2 = (y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_1)$
$Y_3 = Z_1X_2 - X_1Z_2 = (z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_1)$
$Z_3 = X_1Y_2 - Y_1X_2 = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$

D'autre part, pour que N soit de longueur 1, on divise chacune de ses composantes par sa longueur, on obtient alors un vecteur normal unitaire :

$$n = \frac{N}{|N|} = \frac{N}{\sqrt{X_3^2 Y_3^2 Z_3^2}}$$

LA CLASSE TimerThread

La classe *TimerThread* dérive directement de la classe Java *Thread*. Elle permet de gérer une thread d'animation, à travers l'utilisation du nœud VRML *TimeSensor* :

En fonction des champs suivants :
- *cycleInterval*, qui spécifie le nombre de secondes pendant lesquelles l'horloge renvoie des événements,
- *enabled*, qui active ou désactive le détecteur,
- *loop*, qui autorise ou interdit le bouclage de l'horloge,
- *startTime*, qui spécifie quand l'horloge doit démarrer,
- *stopTime*, qui spécifie quand l'horloge doit s'arrêter,

les événements en sortie (*eventOut*) suivants sont mis à jour dans la classe *TimerThread* , de la façon suivante :

- *cycleTime*, renvoie l'heure en début de cycle :
  ```
  node.sendEvent(node.getCycleTimeField());
  ```
- *fraction_changed*, renvoie la fraction de temps comprise entre 0 et 1 :
  ```
  node.sendEvent(node.getFractionChangedField());
  ```
- *isActive,* renvoie l'état de l'horloge :
  ```
  node.sendEvent(node.getIsActiveField());
  ```
- *Time,* renvoie l'heure de chaque fraction :
  ```
  node.sendEvent(node.getTimeField());
  ```

Ces événements de type *eventOut* permettront de mettre à jour les instructions *ROUTE* indispensables aux animations et présents à la fin des fichiers VRML.

LA CLASSE j4

C'est la classe de départ du navigateur VRML : c'est à partir de celle-ci que l'on pourra charger une scène VRML. Pour cela, on crée d'abord un graphe de scène dans CyberVRML97 :

```
SceneGraph sceneGraph = new SceneGraph() ;
```

On charge ensuite le fichier VRML dans ce graphe de scène :

```
sceneGraph.load(file0) ;
```

On instancie ensuite un graphe de scène SceneGraphOpenGL (de l'interface) avec l'instance de SceneGraph :

```
SceneGraphOpenGL sceneGraphOpenGL = new SceneGraphOpenGL(sceneGraph) ;
```

Le chargement d'une scène dans le navigateur se fait de la manière suivante :

java j4 <nom_du_fichier.wrl>

CONCLUSION

Pour conclure, nous allons voir quelles fonctionalités manquent par rapport à la norme VRML97, celles qui pourraient être ajoutées au navigateur, puis les améliorations qui seraient souhaitables d'apporter à celui-ci. Je présenterai alors les avantages et limites de l'intégration du langage Java dans cette étude. Nous terminerons enfin par une vue globale des techniques 3D sur le Web.

Ce qu'il manque par rapport à la norme VRML97 :

- Dans le développement de l'interface OpenGL, implémentation :

  - de la gestion des textures en général
  - des points de vue (*ViewPoint*)
  - de la détection des collisions
  - de capteurs particuliers tels que *CylinderSensor*, *SphereSensor*, *PlaneSensor*
  - de la lumière directionnelle (*DirectionalLight*)
  - de la représentation filaire (*IndexedLineSet*)
  - de la gestion du champ *creaseAngle* du nœud *IndexedFaceSet*
  - du niveau de détail (*Level of detail, LOD*)
  - des surfaces accidentées (*ElevationGrid*)
  - de nœuds particuliers tels que *Anchor, Background* ou *Billboard*

- Dans le développement VRML, implémentation :

  - de l'interface E.A.I. (*External Authoring Interface*)
  - des scripts JavaScript et VRMLscript
  - des prototypes externes (*EXTERNPROTO*)

Ce qu'il manque au navigateur :

Il faudrait ajouter au navigateur un menu permettant de sélectionner un rendu filaire ou facetté, de changer le point de vue, de revenir à la position initiale de la scène, de passer du mode avec ou sans lumière, de changer la vitesse de navigation, d'activer un zoom, etc. Il faudrait également implémenter le son et le faire fonctionner sous forme d'applet.

Les améliorations qui seraient souhaitables d'apporter au navigateur visent principalement à augmenter la vitesse d'exécution. Cette amélioration concerne des scènes VRML conséquentes en terme de complexité (nombre de nœuds VRML important, notamment l'utilisation "intensive" du nœud *IndexedFaceSet*). Pour cela, plusieurs actions sont nécessaires. Les plus importantes sont les suivantes :

- Prise en compte de la possibilité d'utiliser le mode retenu d'OpenGL au moyen de listes d'affichage, ce qui ne peut être fait de façon globale pour toute la scène. En effet, il n'est pas possible, sous peine de ne plus voir fonctionner les animations, de n'utiliser qu'une liste d'affichage globale pour faire cela. Il faut définir une liste d'affichage au niveau de chaque nœud, ce qui est relativement complexe à mettre en oeuvre.

- Faire fonctionner le navigateur avec le jdk 1.4 qui comporte notamment une amélioration de l'implémentation des *JNI* (*Java Native Interface*).

Avantages et limites de l'intégration du langage Java dans cette étude

L'idée des bindings Java utilisant l'interface *JNI* est d'écrire une petite partie de code C ou C++ (le lanceur de la *JVM* Win32) et d'écrire la majeure partie du code (la portion applicative) en Java [FRI99]. Le choix de Java à travers une interface Java-OpenGL permet d'augmenter les possibilités de Java telles que la portabilité (écrire une fois, exécuter partout), le *multi-threading* intégré et les possibilités de programmation réseau.

Les avantages sont les suivants :

- Simplicité

Le langage Java évite les caractéristiques les plus complexes des langages C et C++. Parmi celles-ci on trouve l'héritage multiple, destructeurs, surcharge d'opérateurs (Java propose le concept d'interface pour compenser son manque de l'héritage multiple). La simplicité de Java permet un apprentissage plus rapide.

- Fiabilité

Le langage Java ne propose pas le concept de pointeur. Le mauvais usage de cette caracteristique entraîne souvent de mauvaises terminaisons de programmes ou des comportements imprévisibles. Java a été conçu pour être fiable et sûr.

- Large variété d'API libres d'utilisation

Sun [SUN02] a mis en circulation une large variété d'*API* pour Java. Elles sont conçues pour offrir des fonctionalités d'un point de vue portable pour Win32, Unix et d'autres systèmes d'exploitation.

Par rapport à tous ces avantages, Java présente des inconvénients :

- Performance d'exécution

Les machines virtuelles abaissent la performance. La technique des instructions Java interprétées est de loin la plus lente car elle interprète chaque instruction machine Java. Ceci est le format d'exécution de base supporté par toutes les Machines Virtuellles Java (*JVM*). Cependant, l'apparition du compilateur « *Just In Time*[20] » (*JIT*, cette partie de la *JVM* qui compile les instructions Java – connues sous le nom de *bytecode* – en instructions machines

[20] *JIT* pour « *Just in time* » veut dire essentiellement que les instructions machine d'un programme sont compilées à l'exécution en code machine natif. Cela produit des améliorations de performances importantes par rapport à la méthode d'exécution du code machine interprété. Une bonne *JVM* basée sur le *JIT* peut exécuter du code à plus de 80-90% des applications compilées C/C++.

natives durant l'exécution) entraînera rapidement la disparition du problème de performance. La plupart des *JVM* supportées par les vendeurs sont basées sur le *JIT*, bien que la qualité des améliorations concernant la rapidité peut varier largement.

- Différentes Machine Virtuelles Java

Il existe plusieurs *JVM* bien connues. Presque tous les *IDE* Java sont en correspondance avec une *JVM* particulière. Le code que l'on développe a besoin de fonctionner de la même façon sur chacune d'elles.

- Le support de Java est parfois incomplet

Un des problèmes majeurs est que chaque *JVM* n'exécute pas toujours correctement le code Java ou ne supporte pas les caracteristiques dont on a besoin. Par exemple, le support Java de Netscape paraît être assez pauvre et instable.

- L'utilisation de Java comme une applet réduit les performances [GLD02]

Il apparaît que l'utilisation de Java comme une applet dans un navigateur *HTML* réduit considérablement les performances car l'affichage passe souvent par l'*API* de fenêtrage du navigateur au lieu de passer directement par l'*API* d'affichage de votre système d'exploitation. Ceci est particulièrement vrai avec *Netscape 6.0* qui divise les performances de Java par deux (en utilisant le plugin Java) comparé à l'*appletViewer*.

- Différence de rapidité par rapport à du code C/C++

Il semblerait qu'il y ait quand-même une différence dans la vitesse d'exécution en comparaison avec du code C ou C++ (comparaison faite avec des navigateurs développés en C/C++). Il faut par ailleurs être prudent sur cette comparaison C/Java qui ne pourra être sérieusement validée dans notre cas qu'après la mise en oeuvre des listes d'affichage comme expliqué précédemment.

- Différences d'implémentations OpenGL

Un code Java écrit avec une version d'OpenGL peut échouer avec une version différente. Par exemple, un code fonctionnant parfaitement sous Mesa peut ne pas fonctionner sous OpenGL Microsoft ou bien du code sous Linux peut ne pas fonctionner sous Irix.

Je n'ai pas envisagé dans cette étude le développement intégral d'un logiciel de gestion du graphe de scène VRML97, car le temps imparti était trop court. C'est pourquoi j'ai utilisé *CyberVRML97*, un des seuls logiciels ouverts permettant de faire cela en Java. En ce qui concerne l'utilisation de la bibliothèque OpenGL en Java, il n'aurait pas été raisonnable d'écrire mes propres bindings Java-OpenGL. Le logiciel *Magician* paraissait assez attrayant de par ses caractéristiques (voir par. 2.2.11) mais celui-ci comporte une licence globale d'environ 50.000 F (7620€) pour l'utilisation avec les codes source. C'est pourquoi mon choix s'est porté sur *GL4Java* qui présente l'avantage de supporter aujourd'hui 100% des fonctionalités d'OpenGL à travers Java et permet l'installation des bindings sur diverses plate-formes (voir par. 2.3.7).

VRML souffre du manque de qualité et de rapidité des navigateurs que l'on peut trouver sur le marché. Il existe peu de navigateurs VRML97 et ceux qui existent n'implémentent qu'une partie des fonctionalités décrites dans les spécifications. Actuellement, VRML a des difficultés à s'imposer et à prendre l'essor qui lui était promis.

Bien entendu, en marge de VRML, il existe des plugins 3D pour le Web tels que Metastream de Metacreation dont le format est libre d'utilisation, MendelBox de Mendel3d qui permet un rendu 3D de très bonne qualité. Il existe également des solutions telles que Macromédia Flash qui propose un plugin permettant l'utilisation d'animation vectorielle avec une interactivité plus évidente (déplacement d'images selon l'emplacement du pointeur de la souris).

Cette étude a permis de combiner le langage VRML et la bibilothèque graphique OpenGL à travers le langage Java, ceci ayant entre autre intérêt de masquer au programmeur les détails d'implémentation d'OpenGL, lui autorisant à se concentrer sur son sujet principal, à savoir une utilisation adéquate des fonctions OpenGL permettant d'améliorer le rendu des scènes VRML, en augmentant ainsi sa créativité.

TABLE DES MATIERES

BIBLIOGRAPHIE

Ouvrages :

[COU98] Jean-Pierre Couwenbergh. – *La Synthèse d'Images*. – Marabout : 1998.

[WOO99] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner. – *OpenGL 1.2*. – Campus Press : 1999.

Foley, Van Dam, Feiner, Hughes: « *Computer Graphics: Principles and Practice* » - Addison Wesley - 1999

Articles cités [21]**:**

[BIL99] Bill DAY « *3D Graphics programming in Java, Part 3 : OpenGL* » – mai 1999
http://www.javaworld.com/javaworld/jw-05-1999/jw-05-media_p.html

[FRI99] Geoff FRIESEN *« The Win32 Java Hybrid – Blending Java and Win32 to make Microsoft OS applications »*
http://www-106.ibm.com/developerworks/java/library/java-hybrid/index.html?dwzone=java

[GOR98] Rob GORDON « *Essential JNI, Java Native Interface* » - 1998
http://cstbtech.bcit.ca/BTechPracticum1/doc/design.html

[JAN01] Nicolas JANEY- LIFC, Département d'Informatique – Infographie – Université de Franche-Comté
http://raphaello.univ-fcomte.fr/IG/

[NEY98] Fabrice NEYRET *« OpenGL : Principes et astuces »* – juillet 1998
http://www-imagis.imag.fr/Membres/Fabrice.Neyret/doc/opengl.html

[VER98] Didier VERNA, Nadine RICHARD *« Une introduction à VRML97 »* - avril 1998
http://www.lrde.epita.fr/~didier/comp/teaching/vrml97.php

[21] Tous les sites Web référencés ont fait l'objet d'une consultation au 15 juin 2002

Ressources sur le Web

Tutoriaux VRML :

Tutorial de Dave NADEAU
http://www.sdsc.edu/~nadeau

[KDO98]Groupe VRML Francophone – 1998
http://kdo.chez.tiscali.fr/vrml/start.htm

Ressources VRML :

The Web3D repository
http://www.web3d.org/vrml/vrml.htm

[VRM97] Norme VRML97
http://www.web3d.org/Specifications/VRML97

Ressources OpenGL :

Site officiel d'OpenGL
http://www.opengl.org

Silicon Graphics
http://www.sgi.com

Ressources X3D :

[X3D] *Extensible 3D Graphics Working Group*
http://www.web3d.org/x3d.html

Ressources Java :

The source for Java Technology
http://java.sun.com

Ressources JavaCC :

[JCC] JavaCC (*Java Compiler Compiler*)
http://www.webgain.com/products/java_cc/

Ressources Java3D :

[J3D] Java3D *Sun* :
http://java.sun.com/products/java-media/3D/

The Java3D Community Site
http://www.j3d.org

Logiciels 3D *Open Source*:

[SGO99] Site Web de Sven GOETHEL (logiciel *GL4Java*)
http://www.jausoft.com

[GLD02] Liste de diffusion de *GL4Java*:
http://gl4java.sourceforge.net

[ZIE01] GL4Java pour *Macintosh*
http://www.gerardziemski.com

[SKO99] Site Web de Satoshi KONNO (logiciel *CyberVRML97*)
http://www.cybergarage.org

Projet OpenVRML :
http://www.openvrml.org/

***Bindings* Java-OpenGL :**

[JOG] Site Web de JOGL
http://www.pajato.com/jogl/

[JSP] Site Web de JSPAROW
http://home.att.ne.jp/red/aruga/jsparrow/

[MAG] Site Web distributeur de *MAGICIAN*
http://www.symbolstone.org

Plugins VRML :

Site Web de *Computer Associates*
http://www.cai.com/cosmo

Divers :

[SUN02] Site Web de *Sun Microsystems Inc.*
http://www.sun.com

[CED02] Site Web du CEDRIC (Laboratoire de Recherche en Informatique du Cnam)
http://cedric.cnam.fr

Mémoires d'ingénieur :

[LEG99] J.M. LE GALLIC. *« La visualisation des données géographiques – L'approche VRML-Java »*. Mémoire d'ingénieur CNAM, juin 1999.

[TOP01] A. TOPOL. *« VRML : étude, mise en œuvre et applications »*.
Mémoire d'Ingénieur CNAM, juillet 2001.
Fichier PDF téléchageable sur **http://cedric.cnam.fr/~topol/memInge.pdf**

ANNEXES

ANNEXE 1 : Combinaison d'objets - Page 10

```
#VRML V2.0 utf8
#-----------------------------------------------
# Description  : Fichier à inclure pour cours
# Auteur       : KDO


#---------------[ Le Cube ]----------------
Transform {
        translation -4 0 0
        rotation 0.5 0.5 0.5 1.7
        children [
                Shape {
                        geometry Box {}
                        appearance Appearance {
                                material Material {
                                        ambientIntensity 0.3
                                        diffuseColor 0 0.8 0.2
                                }
                        }
                }
        ]

}

#---------------[ La Sphère ]--------------
Transform {
        translation 0 0 -4
                children [
                        Shape {
                                geometry Sphere {}
                                appearance Appearance {
                                        material Material {
                                                ambientIntensity 0.3
                                                diffuseColor 0.5 0 0.8
                                        }
                                }
                        }
                ]
}
```

```
#----------------[ Le Cone ]----------------
Transform {
        translation 4 0 0
        rotation 5 0 1 0.75
                children [
                        Shape {
                                geometry Cone {}
                                appearance Appearance {
                                        material Material {
                                                ambientIntensity 0.3
                                                diffuseColor 0.8 0 0
                                        }
                                }
                        }
                ]
}
```

```
#VRML V2.0 utf8

PointLight {
ambientIntensity 0.5
location   0 1 0
}

Transform {
  translation 0 0 3
  children [
    DEF leg Shape {
      geometry Cylinder { radius .2
                                  height .5 }
      appearance Appearance {
            material Material { diffuseColor .2 .1 0 }
      }
    }
  ]
}

Transform {
  translation 0 0 -3
  children [
    USE leg
  ]
}

Transform {
  translation -2 0 -3
  children [
    USE leg
  ]
}

Transform {
  translation -2 0 3
  children [
    USE leg
  ]
}

Transform {
  translation -1 .3 0
  rotation 0 0 1 .1
  children [
    Shape {
      geometry Box { size 2.5 .5 7 }
      appearance Appearance {
            material Material { diffuseColor .1 .2 .5 }
      }
    }
  ]
}
```

```
Transform {
  translation -1 .5 3.5
  rotation 0 0 1 .1
  children [
    DEF arm Group {
      children [
            Shape {
          geometry Box { size 2.5 .3 .3 }
          appearance Appearance {
                material Material { diffuseColor .1 .2 .5 }
          }
        }
        Transform {
          translation 0 .2 0
          rotation 0 0 1 -1.5
          children [
                Shape {
            geometry Cylinder { radius .2
                                        height 3 }
            appearance Appearance {
              material Material { diffuseColor .1 .2 .5 }
            }
          }
        ]
      }
    ]
  }
 ]
}

Transform {
  translation -1 .7 3.5
  rotation 0 0 1 -1.4
  children [
    DEF woodarm Shape {
      geometry Cylinder { radius .1
                                  height 3.2 }
      appearance Appearance {
        material Material { diffuseColor .2 .1 0 }
      }
    }
  ]
}

Transform {
  translation -1 .7 -3.5
  rotation 0 0 1 -1.4
  children [
    USE woodarm
  ]
}

Transform {
  translation -1 .5 -3.5
  rotation 0 0 1 .1
  children [
    USE arm
  ]
}
```

```
Transform {
  translation -2.2 .8 0
  rotation 0 0 1 .4
  children [
    Shape {
      geometry Box { size .3 1.5 6.5 }
      appearance Appearance {
            material Material { diffuseColor .1 .2 .5 }
      }
    }
  ]
}

Transform {
  translation -1 .4 0
  rotation 0 0 1 .1
  children [
    Shape {
      geometry Box { size 2.5 .5 6.5 }
      appearance Appearance {
            material Material { diffuseColor .1 .2 .5 }
      }
    }
    Transform {
      translation 1.15 0 0
      rotation 1 0 0 1.570796327
      children [
        Shape {
              geometry Cylinder { radius .25
                                  height 6.5 }
              appearance Appearance {
                material Material { diffuseColor .1 .2 .5 }
              }
            }
      ]
    }
  ]
}
```

```
#VRML V2.0 utf8
Transform {
  translation 0 0 -2
  rotation 0.7 0.7 -0.12 0.49
  children [
    DEF TS1 TouchSensor {}
    Shape {
      appearance Appearance {
        material DEF BoxColor Material {diffuseColor 1 0 0}
      } # End Appearance
      geometry Box {}
    }
    Transform {
      translation 0 2 0
      children [
            DEF TS2 TouchSensor {}
        Shape {
          appearance Appearance {
            material DEF SphereColor Material {diffuseColor 0 0 1}
          } # End Appearance
          geometry Sphere {}
        }
      ]
    }
  ]
}


DEF ColorScript1 Script {
          url "ChangeColor1.class"
          eventIn SFBool clicked1
          eventOut SFColor newColor1
          field SFBool on1 FALSE
}

DEF ColorScript2 Script {
          url "ChangeColor2.class"
          eventIn SFBool clicked2
          eventOut SFColor newColor2
          field SFBool on2 FALSE
}

ROUTE TS1.isOver TO ColorScript1.clicked1
ROUTE TS2.isActive TO ColorScript2.clicked2
ROUTE ColorScript1.newColor1 TO BoxColor.set_diffuseColor
ROUTE ColorScript2.newColor2 TO SphereColor.set_diffuseColor
```

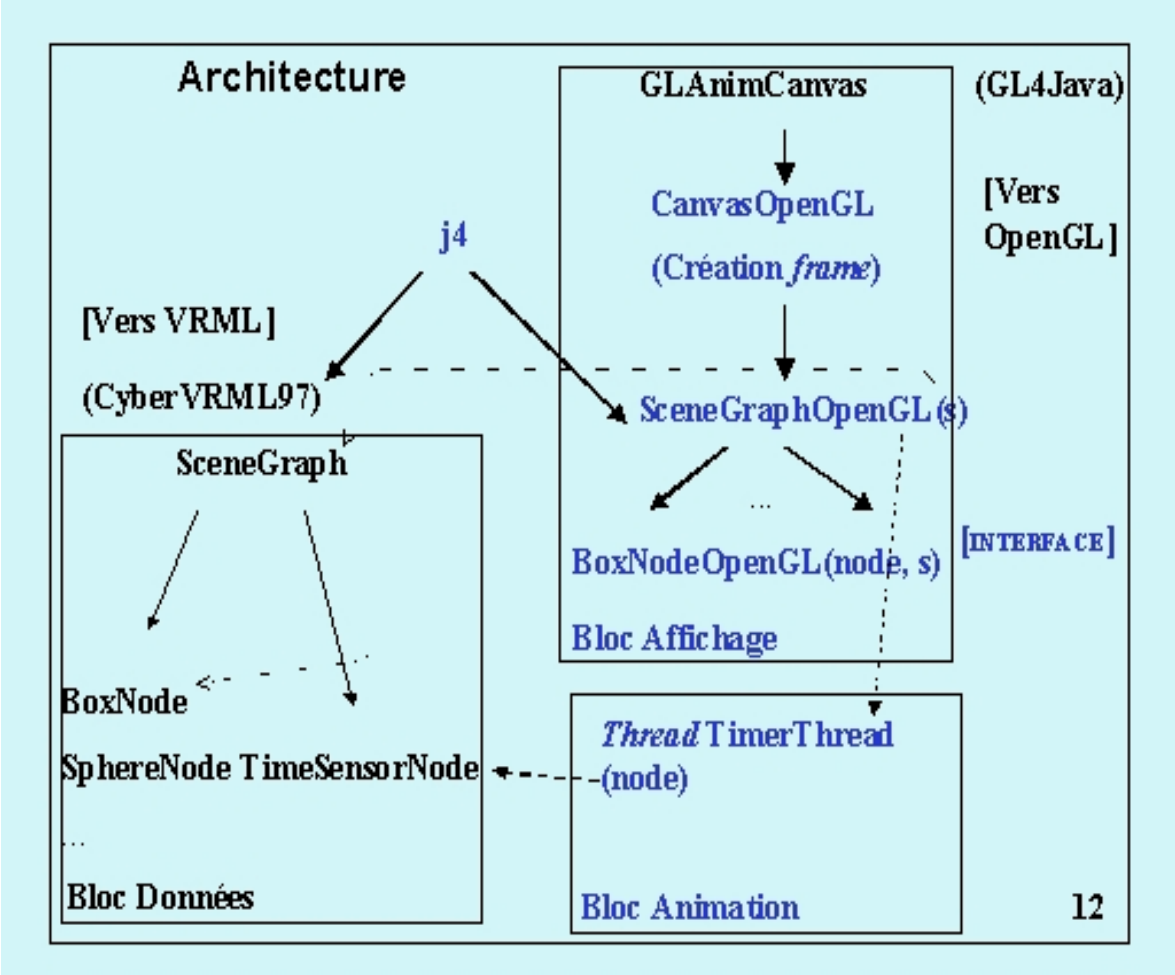
```
public class ChangeColor1 extends Script {
private SFColor newColor1;
private SFBool on1;
float red[] = {1,0,0};
float blue[] = {0,0,1};
public void initialize() {
        System.out.println("Init");
        newColor1 = (SFColor)getEventOut ("newColor1");
        on1 = (SFBool)getField("on1");
}
        public void processEvent(Event e) {
                if (e.getName().equals("clicked1")) {
                  System.out.println("clicked1");
                  ConstSFBool v = (ConstSFBool)e.getValue();
                  if (v.getValue()) {
                        if(on1.getValue()) {
                                newColor1.setValue(red);
                        }
                        else {
                          newColor1.setValue(blue);
                        }
                     on1.setValue(!on1.getValue());
                  }
                }
        }
}

import cv97.*;
import cv97.field.*;
import cv97.node.*;
public class ChangeColor2 extends Script {
private SFColor newColor2;
private SFBool on2;
float red[] = {1,0,0};
float blue[] = {0,0,1};
public void initialize() {
        System.out.println("Init");
        newColor2 = (SFColor)getEventOut ("newColor2");
        on2 = (SFBool)getField("on2");
}
        public void processEvent(Event e) {

           if (e.getName().equals("clicked2")) {
             System.out.println("clicked2");
                ConstSFBool v = (ConstSFBool)e.getValue();

                if (v.getValue()) {
                        if(on2.getValue()) {
                                newColor2.setValue(blue);
                        }
                        else {
                          newColor2.setValue(red);
                        }
                     on2.setValue(!on2.getValue());
                  }
                }
        }
}
```

ANNEXE 4 : Schéma d'architecture



- La partie Bloc Données représente les classes de *CyberVRML97* et permet donc d'accéder au graphe de scène VRML
- La partie Bloc Affichage représente une partie de l'interface VRML-OPENGL qui hérite des classes de *GL4Java* (c'est-à-dire de *GLAnimCanvas*) ce qui permet d'accéder aux fonctions OpenGL depuis Java.
- En lançant *j4*, on instancie un graphe de scène *SceneGraph* du Bloc Données et un graphe de scène *SceneGraphOpenGL(s)* du Bloc Affichage : ce dernier est lancé en lui passant l'instance *s* de *SceneGraph*.
- De la même manière, lorsque l'on crée un nœud VRML dans le Bloc Affichage, il reçoit les instances *s* et *node* de *SceneGraph* : par exemple le nœud *BoxNodeOpenGL* recevra l'instance de *BoxNode*. C'est en remontant (via les constructeurs) dans *CanvasOpenGL* que l'on va créer une fenêtre *Canvas* de Java dans laquelle on représentera la scène VRML
- Enfin, le Bloc Animation contient une classe *TimerThread* qui est activée sur la présence dans *SceneGraph* d'un nœud *TimeSensor* : une thread est lancée pour chaque occurrence de ce nœud.
- Cette architecture garantie l'indépendance de l'interface VRML-OPENGL, à savoir qu'aucune modification n'a été réalisée dans ces deux logiciels (sauf déclaration du package cv97.opengl dans *GLAnimCanvas*).

ANNEXE 5 : Principales classes de l'interface

```
/*****************************************************************
*
*          Navigateur Java-OpenGL
*          Copyright (C)  Jean-Marie ABISROR
*          (http://membres.lycos.fr/sysnet)
*          File : j4.java
* Note:  Ce logiciel utilise les paquetages de CyberVRML97 for Java
*          developpe par Satoshi Konno
*
*****************************************************************/
import cv97.*;
import cv97.opengl.*;
import cv97.node.*;
import java.applet.*;
import java.security.*;

public class j4 extends Applet{
          String param1=null;

   public void LoadSceneGraph(String file0, String file1) {
      Node node=null;
      SceneGraph sceneGraph = new SceneGraph();
      //SceneGraph sceneG = new SceneGraph();       // Double fenetrage
      //sceneGraph.setOption(SceneGraph.USE_PREPROCESSOR);
      //sceneGraph.setOption(SceneGraph.NORMAL_GENERATION);
      sceneGraph.load(file0);
      //sceneG.load(file1);                         //Double fenetrage
      //sceneGraph.startSimulation();
      //sceneGraph.update();
      SceneGraphOpenGL sceneGraphOpenGL = new SceneGraphOpenGL(sceneGraph);
     //SceneGraphOpenGL sceneGraphOpen = new SceneGraphOpenGL(sceneG);        //Double fenetrage
     //node.level=2;
    //sceneGraph.Afficher();
    }

      public static
      void main(String args[]) {
        //String arg0 = args[0];
        //String arg1 = args[1];

         j4 C2 = new j4();
                    if (args.length < 1)
                System.out.println("Usage : "+args[0]+"<filename>");

                    else {
                if (args.length < 2) {

                C2.LoadSceneGraph(args[0],null);
                }

          else
                    C2.LoadSceneGraph(args[0],args[1]);

          }

          }
```

```
	public void init() {
		param1=getParameter("parametre1");
		System.out.println("Valeur du paramètre:" +param1);
		j4 C2 = new j4();
		C2.LoadSceneGraph(param1,null);
	}
}
```

```
/*************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*               File : SceneGraphOpenGL.java
*
*************************************************************/

package cv97.opengl;
import cv97.*;
import cv97.opengl.*;
import cv97.node.*;

public class SceneGraphOpenGL extends CanvasOpenGL {
private RootNode mRootNode = null;
Node node=null;
boolean firstPass = true;

        public SceneGraphOpenGL(SceneGraph sg) {
                super();
                mRootNode = sg.getRootNode();
                node=mRootNode;
                setSGGL(this, sg);
        }



    public void display() {
   //   System.out.println("SceneGraphOpenGL.display");

                        if (firstPass) {
                                // Creation de 4 vecteurs pour memoriser
                                // Script, Interpolator, Sensor et Pile
                                vectorOpenGL = new VectorOpenGL();
                         }

/*      if( !glj.gljMakeCurrent()) {
                System.out.println("SceneGraphOpenGL: problem in use() method");
        return;
                }
*/

        gl.glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        gl.glPushMatrix();
        gl.glTranslatef(posX,posY,posZ);
        gl.glRotatef(rotX, 1.0f, 0.0f, 0.0f);
        gl.glRotatef(rotY, 0.0f, 1.0f, 0.0f);

        display((Node)mRootNode);
        firstPass = false;
        gl.glPopMatrix();
        gl.glFlush();
         glj.gljSwap();
        //glj.gljCheckGL();
      }


        public void display(Node node) {
         boolean traite = false;
       int i;
       VectorElementOpenGL sensorElement = null;

                        if (node==null)
                           System.out.println("Pb: node=NULL");
                        if (firstPass) {
                                for (i=0; i<node.getNChildNodes(); i++) {
```

```
                                        if (node.getChildNode(i).isTouchSensorNode()) {
                                        //Empiler
                                        ItemVectorOpenGL itemPile;
                                        itemPile = new ItemVectorOpenGL();
                                        itemPile.nodeType   = Constants.touchSensorTypeName;
                                        itemPile.nodeName  = node.getChildNode(i).getName();

                                        vectorOpenGL.addPile(itemPile);
                                        }
                                }
                        }


        for (i=0; i<node.getNChildNodes(); i++) {
         Node child = node.getChildNode(i);
                        Field thisField , nodeField;
           if (child.isTimeSensorNode()) {
                        if (firstPass) {
                                TimerThread timerThread = new
TimerThread("TimerThread",(TimeSensorNode)child);
                                timerThread.start();
                        }
                        else
                                traite = true;
           }
                          if (child.isNavigationInfoNode()) {
                                if (firstPass) {
                                   NavigationInfoNodeOpenGL NavigationInfo = new
NavigationInfoNodeOpenGL();        //dynamique
                                   NavigationInfo.display((NavigationInfoNode)child,this);     //dynamique
                                   //NavigationInfoNodeOpenGL.display((NavigationInfoNode)child,this);
                                 }
                          }
           if (child.isInlineNode()) {
             if (firstPass) {
                                InlineNodeOpenGL Inline = new InlineNodeOpenGL(); //dynamique
                                Inline.display((InlineNode)child,this,sg);                      //dynamique
                                Inline.loadURL((InlineNode)child,sg);
                                //InlineNodeOpenGL.display((InlineNode)child,this,sg); //statique
             }
           }

           if (child.isPointLightNode()) {
           PointLightNodeOpenGL PointLight = new PointLightNodeOpenGL();      //dynamique
           PointLight.display((PointLightNode)child,this);                                  //dynamique
                        //PointLightNodeOpenGL.display((PointLightNode)child,this);
           //statique
               traite = true;
           }

           if (child.isShapeNode()) {
             if (firstPass) {
//                              if (vectorOpenGL.vectorPile.size() > 0) {
                                                //creer un nouvel element VectorElement
                                                sensorElement = new VectorElementOpenGL();
                                                for (int k=0; k<vectorOpenGL.vectorPile.size();k++) {

                                ItemVectorOpenGL itemPile = vectorOpenGL.getPileElement(k);

                                        //creer l'element recepteur renseigné avec l'element de la pile

                                ItemVectorOpenGL item = new ItemVectorOpenGL();
                                item.nodeType = itemPile.nodeType;
                                tem.nodeName = itemPile.nodeName;

           System.out.println("item.nodeType="+item.nodeType);

           System.out.println("item.nodeName="+item.nodeName);
```

```
                                                                    //ecrire sur vecteur element
                                                                    sensorElement.addElement(item);
                                                }
                                                                    vectorOpenGL.addSensor(sensorElement);
            }

                                    if (displayMode == gl.GL_SELECT)
                                                gl.glPushName(shapeCounter++);
                        ShapeNodeOpenGL Shape = new ShapeNodeOpenGL();              //dynamique
                        Shape.display((ShapeNode)child,this);                       //dynamique
                        //ShapeNodeOpenGL.display((ShapeNode)child,this);           //statique
                                    if (displayMode == gl.GL_SELECT)
                                                gl.glPopName();
                traite = true;
            }

            if (child.isTransformNode()) {
                        TransformNodeOpenGL Transform = new TransformNodeOpenGL();  //dynamique
                        Transform.display((TransformNode)child,this);               //dynamique
                        //TransformNodeOpenGL.display((TransformNode)child,this);   //statique
              traite = true;
            }

              if (!traite)
                   display(child);
           }

              if (firstPass) {
                        for (i=0; i<node.getNChildNodes(); i++) {
                          if (node.getChildNode(i).isTouchSensorNode()) {
                            //Dépiler
                                    vectorOpenGL.deletePile();
                        }
                        }
            }

    }

}
```

```
/*****************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*               File : CanvasOpenGL.java
*
*****************************************************************/

package cv97.opengl;
import cv97.*;
import cv97.node.*;
import cv97.field.*;
import cv97.route.*;
import java.awt.*;
import java.awt.event.*;
import gl4java.*;
import gl4java.awt.*;
import gl4java.GLContext;
import gl4java.GLFunc;
import gl4java.GLEnum;
import gl4java.utils.glut.*;

public class CanvasOpenGL extends GLAnimCanvas implements
                                                KeyListener , MouseMotionListener {
VectorOpenGL vectorOpenGL;
SceneGraphOpenGL sgGL;
SceneGraph sg;
int displayMode;
int shapeCounter;
int w, h;
Frame frame;
Point point = null;
GLUTFunc glut = null;
float rotX, rotY, rotZ;
float posX, posY, posZ;
float light[] = { 0.0f, 0.0f, 1.0f, 0.0f };
int base_x;
int base_y;
int hits=0;
int [] selectBuf = new int[2048];
int [] viewport = new int[4];
int oldHits = 0;
int oldIndex[];

                public CanvasOpenGL() {
                                this(500,500);
                }

                public CanvasOpenGL(int w, int h) {
                super(w,h);
                 System.out.println("Appel constructeur CanvasOpenGL");
                frame = new Frame("OpenGL");
                frame.setSize(w,h);
                frame.add("Center",this);
                //frame.getContentPane().add(this, BorderLayout.CENTER);
                frame.setVisible(true);
                 addKeyListener(this);
                frame.addMouseListener(this);
                addMouseMotionListener(this);
                frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                System.exit(0);
                 }
                });
```

```
            GLContext.gljNativeDebug = false;
            GLContext.gljClassDebug  = false;
            rotX = 0.0f;
            rotY = 0.0f;
            rotZ = 0.0f;
            posX = 0.0f;
            posY = 0.0f;
            posZ = -10.0f;
            this.w = w;
            this.h = h;
            }


          public void setSGGL(SceneGraphOpenGL sgGL, SceneGraph sg) {
            this.sgGL = sgGL;
            this.sg = sg;
          }
            public void preInit() {
                        doubleBuffer = true;
                        stereoView = false;
                        //stencilBits = 8;
                        //setUseFpsSleep(true);
                        //setUseRepaint(true);
                        createOwnWindow = true;
            }


            public void init() {

                System.out.println("Appel a init de CanvasOpenGL");

              //super.init();
              start();


          reshape(getSize().width, getSize().height);
          glut = new GLUTFuncLightImpl(gl, glu);
          gl.glClearColor(0.0f,0.0f,0.0f,0.0f);
          gl.glShadeModel(GL_SMOOTH);                       // GL_SMOOTH ou GL_FLAT
          gl.glEnable(GL_DEPTH_TEST);
          gl.glPolygonMode(GL_FRONT, GL_FILL);              // GL_FILL ou GL_LINE ou GL_POINT
          gl.glLightfv(GL_LIGHT0, GL_POSITION, light);
          gl.glEnable(GL_LIGHT0);
          gl.glEnable(GL_LIGHTING);
        }

         public void reshape(int w, int h) {
          gl.glViewport(0, 0, w, h);
          gl.glMatrixMode(GL_PROJECTION);
          gl.glLoadIdentity();
          if (w>h)
            glu.gluPerspective(45.0f, (float)w/(float)h, 1.0f, 10000.0f);
          else
            glu.gluPerspective(45.0f, (float)h/(float)w, 1.0f, 10000.0f);
//          gl.glScalef(0.25f,0.25f,0.25f);
          gl.glMatrixMode(GL_MODELVIEW);

                this.w = w;
                this.h = h;
        }
```

```
        public void keyTyped(KeyEvent e) {
        }
        public void keyPressed(KeyEvent e) {
          int code = e.getKeyCode();

          switch (code) {
            case KeyEvent.VK_Q:
              rotX += 5.0;
              break;
            case KeyEvent.VK_W:
              rotX -= 5.0;
              break;
            case KeyEvent.VK_A:
              rotY += 5.0;
              break;
            case KeyEvent.VK_Z:
              rotY -= 5.0;
              break;
            case KeyEvent.VK_LEFT:
                        posX-=2.0f;
                        break;
                      case KeyEvent.VK_RIGHT:
                        posX+=2.0f;
                        break;
            case KeyEvent.VK_UP:
                        posY+=5.0f;
                        break;
                      case KeyEvent.VK_DOWN:
                        posY-=5.0f;
                        break;
                      case KeyEvent.VK_PAGE_UP:
              //System.out.println("Z="+posZ);
                        posZ+=20.0f;
                        break;
                      case KeyEvent.VK_PAGE_DOWN:
              //System.out.println("Z="+posZ);
                        posZ-=20.0f;
                        break;
          }

}

        public void keyReleased(KeyEvent e) {
                }


    public void mousePressed(MouseEvent e) {
                //System.out.println("Mouse pressed");
                base_x = e.getX();
                base_y = e.getY();
                try {
                                test_select(base_x, base_y, "pressed");
                }
           catch (NullPointerException npe)
           {
                /*npe.printStackTrace()*/
           }
      }
```

```
  public void mouseReleased(MouseEvent e) {
            //System.out.println("Mouse released");
            base_x = e.getX();
            base_y = e.getY();
            try {
                        test_select(base_x, base_y, "released");
            }
     catch (NullPointerException npe)
     {
            /*npe.printStackTrace()*/
     }
  }


  public void mouseMoved(MouseEvent e) {
    //System.out.println("Mouse moved");
            int x = e.getX();
            int y = e.getY();
            try {
                         test_select(x, y, "over");
     }
     catch (NullPointerException npe)
     {
            /*npe.printStackTrace()*/
     }
  }

  public void mouseDragged(MouseEvent e) {
    //System.out.println("Mouse dragged");

          int x = e.getX();
          int y = e.getY();

     if (e.getX() > base_x + 50) {
            tourner_droit();
     }
     if (e.getX() < base_x - 50) {
            tourner_gauche();
     }

     if (e.getY() > base_y + 5) {
            eloigner();
     }
     if (e.getY() < base_y - 5) {
            rapprocher();
     }

  }


  public void eloigner() {
            for (int i=1; i<2; i++) {
                        posZ-=3.0f;
            }
  }

  public void rapprocher() {
            for (int i=1; i<2; i++) {
                        posZ+=3.0f;
            }
}

  public void tourner_gauche() {
                        posX+=1.0f;
}
```

```
  public void tourner_droit() {
                        posX-=1.0f;
}

  public void test_select(int x, int y, String event) {
    int indexObject;
    int newIndex[];
            gl.glGetIntegerv(gl.GL_VIEWPORT,viewport);
            gl.glSelectBuffer(2048,selectBuf);
            gl.glRenderMode(gl.GL_SELECT);
            displayMode = gl.GL_SELECT;
            gl.glInitNames();
            gl.glMatrixMode(gl.GL_PROJECTION);
            gl.glPushMatrix();
            gl.glLoadIdentity();
            glu.gluPickMatrix(x, viewport[3] - y, 3.0, 3.0, viewport);

            if (w > h)
      glu.gluPerspective(45.0f, (float)w/(float)h, 1.0f, 10000.0f);
     else
      glu.gluPerspective(45.0f, (float)h/(float)w, 1.0f, 10000.0f);

            shapeCounter = 0;
            sgGL.display();
            gl.glMatrixMode(gl.GL_PROJECTION);
            gl.glPopMatrix();
            gl.glMatrixMode(gl.GL_MODELVIEW);
            hits = gl.glRenderMode(gl.GL_RENDER);
            displayMode = gl.GL_RENDER;
            System.out.println("hits="+hits);
            int cpt = 0;
            int names = 0;
            newIndex = new int[hits];
            for (int i=0; i < hits; i++) {

                        names = selectBuf[cpt];
//                      System.out.println("Number of names for hit = " + names);
                        cpt++;
//                      System.out.println("z1 is " + (float)selectBuf[cpt]/0x7fffffff);
                        cpt++;
//                      System.out.println("z2 is " + (float)selectBuf[cpt]/0x7fffffff);
                        cpt++;
                        indexObject = selectBuf[cpt];
                        System.out.println("index = " + indexObject);
                        cpt++;
                        boolean changed = true;
                        int j;
                        if (event.equals("over")) {
                                    for (j=0;j<oldHits;j++)
                                                if (oldIndex[j] == indexObject) changed = false;
                        }

                        newIndex[i] = indexObject;
                        if (changed == true) {
                          // Recuperation des TS associés a la Shape trouvée
                          VectorElementOpenGL ve = vectorOpenGL.getSensor(indexObject);
                          for (j=0;j<ve.size();j++) {
                         ItemVectorOpenGL item = ve.getElement(j);
                         System.out.println("\t"+item.nodeName);
                         Route route = null;
                         System.out.println("event="+event);
                         if (event.equals("pressed"))
                                    route = getFirst2Routes(item.nodeName, "isActive", "touchTime");
                        if (event.equals("released"))
                                     route = getFirstRoute(item.nodeName, "isActive");
                        if (event.equals("over"))
```

```
                                        route = getFirstRoute(item.nodeName, "isOver");

                                while (route != null) {
                    // Afficher les routes activées
                                        Node  fromNode      = route.getEventOutNode();
                                        Field f1        = route.getEventOutField();
                                        Node  toNode        = route.getEventInNode();
                                        Field f2        = route.getEventInField();
                                        System.out.println("\t\tROUTE " + fromNode.getName() + "." + f1.getName() + " TO "
+ toNode.getName() + "." + f2.getName());

                                        if (toNode.isScriptNode())
                                          ((ScriptNode)toNode).processEvent(f2);
                                else {

                                         TouchSensorNode touchSensor = (TouchSensorNode)fromNode;

                                        if (event.equals("pressed")) {
                                                ConstSFBool isActive = touchSensor.getIsActiveField();
                                                ConstSFTime touchTime = touchSensor.getTouchTimeField();
                                                isActive.setValue(true);
                                                touchTime.setValue(sg.getCurrentTime());
                                                touchSensor.sendEvent(touchTime);
                                                touchSensor.sendEvent(isActive);
                                                        }

                                        if (event.equals("released")) {
                                                ConstSFBool isActive = touchSensor.getIsActiveField();
                                                isActive.setValue(false);
                                                touchSensor.sendEvent(isActive);
                                                        }


                                        }

                    if (event.equals("pressed"))
                       route = getNext2Routes(item.nodeName, "isActive", "touchTime", route.next());
                    if (event.equals("released"))
                       route = getNextRoute(item.nodeName, "isActive", route.next());
                                if (event.equals("over"))
                       route = getNextRoute(item.nodeName, "isOver", route.next());
                                }
                               }
                    }
                      }
                      oldIndex = newIndex;
                      oldHits = hits;
  }


                      public Route getNextRoute(String node, String event, Route beginRoute) {
                                Route route;
                                for (route = beginRoute; route != null; route = route.next()) {
                                        Node fromNode = route.getEventOutNode();
                                        Field f1      = route.getEventOutField();
                                                if (fromNode.getName() == node)
                                                        if (f1.getName() == event) break;
                                }

                                return route;
                      }

                      public Route getFirstRoute(String node, String event) {
                                return getNextRoute(node, event, sg.getRoutes());
                      }

                      public Route getNext2Routes(String node, String event1, String event2, Route beginRoute) {
                                Route route;
```

```
                for (route = beginRoute; route != null; route = route.next()) {
                        Node fromNode = route.getEventOutNode();
                        Field f1      = route.getEventOutField();
                                if (fromNode.getName() == node) {
                                        if (f1.getName() == event1) break;
                                        if (f1.getName() == event2) break;
                                }
                }

                return route;
        }

        public Route getFirst2Routes(String node, String event1, String event2) {
                return getNext2Routes(node, event1, event2, sg.getRoutes());
        }

        public void ReInit()
        {

        }

}
```

```
/*****************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*               File : ShapeNodeOpenGL.java
*
*****************************************************************/

package cv97.opengl;
import cv97.node.*;
public class ShapeNodeOpenGL {
  public ShapeNodeOpenGL() {
  }

  public void display(ShapeNode node, SceneGraphOpenGL sgGL) {
   Node appearance, geometry;

 /*   System.out.println("Appel a ShapeNodeOpenGL.display");                                    */

    appearance = (Node)(node.getAppearanceField().getValue());          // Appearance
    geometry   = (Node)(node.getGeometryField().getValue());            // Box , Cylinder , Cone , PointSet

    // Traitement de l'apparence

    if (appearance!=null) {
                if (appearance.isAppearanceNode()) {
                                Node material = (Node)(((AppearanceNode)appearance).getMaterialField().getValue());
// Material
                                Node texture = (Node)(((AppearanceNode)appearance).getTextureField().getValue());    //
Texture
                                //Node texturetransform =
(Node)(((AppearanceNode)appearance).getTextureTransformField().getValue());    // TextureTransform

      if (material != null) {
        MaterialNodeOpenGL Material = new MaterialNodeOpenGL();       //dynamique
        Material.display((MaterialNode)material,sgGL);                          //dynamique
        //MaterialNodeOpenGL.display((MaterialNode)material,sgGL);      //statique
      }
      if (texture != null) {
        /* A implementer
        if (texture.isImageTextureNode()) {
          ImageTextureNodeOpenGL.display((ImageTextureNode)texture,sgGL);
        }
        if (texture.isMovieTextureNode()) {
          MovieTextureNodeOpenGL.display((MovieTextureNode)texture,sgGL);
        }
        if (texture.isPixelTextureNode()) {
          PixelTextureNodeOpenGL.display((PixelTextureNode)texture,sgGL);
       }
        */
      }
    }
   }

    // Traitement de la géométrie
    if (geometry.isSphereNode()) {
                SphereNodeOpenGL Sphere = new SphereNodeOpenGL(); //dynamique
                Sphere.display((SphereNode)geometry, sgGL);                 //dynamique
                //SphereNodeOpenGL.display((SphereNode)geometry, sgGL);             //statique
    }

    if (geometry.isBoxNode()) {
                BoxNodeOpenGL Box = new BoxNodeOpenGL();                        //dynamique
                Box.display((BoxNode)geometry, sgGL);                           //dynamique
                //BoxNodeOpenGL.display((BoxNode)geometry, sgGL);       //statique
    }
```

```
    if (geometry.isCylinderNode()) {
                CylinderNodeOpenGL Cylinder = new CylinderNodeOpenGL();          //dynamique
                Cylinder.display((CylinderNode)geometry, sgGL);                  //dynamique
                //CylinderNodeOpenGL.display((CylinderNode)geometry, sgGL);      //statique
    }

    if (geometry.isConeNode()) {
                ConeNodeOpenGL Cone = new ConeNodeOpenGL();                      //dynamique
                Cone.display((ConeNode)geometry, sgGL);                          //dynamique
                //ConeNodeOpenGL.display((ConeNode)geometry, sgGL);  //statique
    }

    if (geometry.isPointSetNode()) {
    }

    if (geometry.isIndexedFaceSetNode()) {
                IndexedFaceSetNodeOpenGL IndexedFaceSet = new IndexedFaceSetNodeOpenGL();
                //dynamique
                IndexedFaceSet.display((IndexedFaceSetNode)geometry, sgGL);
                //dynamique
                //IndexedFaceSetNodeOpenGL.display((IndexedFaceSetNode)geometry, sgGL);     //statique
    }

  }
}



/*****************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*
*               File : MaterialNodeOpenGL.java
*
*****************************************************************/

package cv97.opengl;
import cv97.node.*;

public class MaterialNodeOpenGL {
        public void display (MaterialNode node,SceneGraphOpenGL sgGL) {
        /* System.out.println("Appel Methode display de MaterialNodeOpenGL");         */
        float transparency[] = new float[4];
        transparency[0] = transparency[1] = transparency[2] = transparency[3] = node.getTransparency();
        float ambientIntensity[] = new float[4];
        ambientIntensity[0]=ambientIntensity[1]=ambientIntensity[2]=node.getAmbientIntensity();
        ambientIntensity[3]=1;
        float shininess[]             = new float[1]; node.getShininess();
        float diffuseColor[]        = new float[4]; node.getDiffuseColor(diffuseColor);
        diffuseColor[3]             = 1.0f;
        float specularColor[]     = new float[4]; node.getSpecularColor(specularColor);
        specularColor[3]          = 1.0f;
        float emissiveColor[]     = new float[4]; node.getEmissiveColor(emissiveColor);
        emissiveColor[3]          = 1.0f;

        sgGL.gl.glMaterialfv(CanvasOpenGL.GL_FRONT, CanvasOpenGL.GL_AMBIENT, ambientIntensity);
        sgGL.gl.glMaterialfv(CanvasOpenGL.GL_FRONT, CanvasOpenGL.GL_SHININESS, shininess);
        sgGL.gl.glMaterialfv(CanvasOpenGL.GL_FRONT, CanvasOpenGL.GL_DIFFUSE,  diffuseColor);
        sgGL.gl.glMaterialfv(CanvasOpenGL.GL_FRONT, CanvasOpenGL.GL_SPECULAR, specularColor);
        sgGL.gl.glMaterialfv(CanvasOpenGL.GL_FRONT, CanvasOpenGL.GL_EMISSION, emissiveColor);
        sgGL.gl.glEnable(sgGL.gl.GL_COLOR_MATERIAL);
        sgGL.gl.glColorMaterial(sgGL.gl.GL_FRONT_AND_BACK, sgGL.gl.GL_DIFFUSE);
        sgGL.gl.glColor3f(diffuseColor[0], diffuseColor[1], diffuseColor[2]);

  }
```

```
}

/*****************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*
*               File : BoxNodeOpenGL.java
*
*****************************************************************/

package cv97.opengl;
import cv97.node.*;

public class BoxNodeOpenGL {
               public void display(BoxNode node,SceneGraphOpenGL sgGL) {
 /*             System.out.println("Appel Methode display de BoxNodeOpenGL");                    */
               float xsize,ysize,zsize;
               float xmin,xmax,ymin,ymax,zmin,zmax;
               xsize = node.getX();
               ysize = node.getY();
               zsize = node.getZ();

               if (xsize>0)  xmin = -xsize/2.0f; else
                                       xmin =  xsize/2.0f;
               if (xsize<0)  xmax = -xsize/2.0f; else
                                       xmax =  xsize/2.0f;
               if (ysize>0)  ymin = -ysize/2.0f; else
                                       ymin =  ysize/2.0f;
               if (ysize<0)  ymax = -ysize/2.0f; else
                                       ymax =  ysize/2.0f;
               if (zsize>0)  zmin = -zsize/2.0f; else
                                       zmin =  zsize/2.0f;
               if (zsize<0)  zmax = -zsize/2.0f; else
                                       zmax =  zsize/2.0f;

                // Avant
               sgGL.gl.glBegin(sgGL.GL_POLYGON);
               sgGL.gl.glNormal3f(0.0f, 0.0f, -1.0f);
               sgGL.gl.glVertex3f(xmin,ymin,zmin);
               sgGL.gl.glVertex3f(xmax,ymin,zmin);
               sgGL.gl.glVertex3f(xmax,ymax,zmin);
               sgGL.gl.glVertex3f(xmin,ymax,zmin);
               sgGL.gl.glEnd();

               // Gauche
               sgGL.gl.glBegin(sgGL.GL_POLYGON);
               sgGL.gl.glNormal3f(-1.0f, 0.0f, 0.0f);
               sgGL.gl.glVertex3f(xmin,ymin,zmin);
               sgGL.gl.glVertex3f(xmin,ymax,zmin);
               sgGL.gl.glVertex3f(xmin,ymax,zmax);
               sgGL.gl.glVertex3f(xmin,ymin,zmax);
               sgGL.gl.glEnd();

               // Droite
               sgGL.gl.glBegin(sgGL.GL_POLYGON);
               sgGL.gl.glNormal3f(1.0f, 0.0f, 0.0f);
               sgGL.gl.glVertex3f(xmax,ymin,zmin);
               sgGL.gl.glVertex3f(xmax,ymax,zmin);
               sgGL.gl.glVertex3f(xmax,ymax,zmax);
               sgGL.gl.glVertex3f(xmax,ymin,zmax);
               sgGL.gl.glEnd();
```

```
            // Bas
            sgGL.gl.glBegin(sgGL.GL_POLYGON);
            sgGL.gl.glNormal3f(0.0f, -1.0f, 0.0f);
            sgGL.gl.glVertex3f(xmin,ymin,zmin);
            sgGL.gl.glVertex3f(xmax,ymin,zmin);
            sgGL.gl.glVertex3f(xmax,ymin,zmax);
            sgGL.gl.glVertex3f(xmin,ymin,zmax);
            sgGL.gl.glEnd();

            // Haut
            sgGL.gl.glBegin(sgGL.GL_POLYGON);
            sgGL.gl.glNormal3f(0.0f, 1.0f, 0.0f);
            sgGL.gl.glVertex3f(xmin,ymax,zmin);
            sgGL.gl.glVertex3f(xmax,ymax,zmin);
            sgGL.gl.glVertex3f(xmax,ymax,zmax);
            sgGL.gl.glVertex3f(xmin,ymax,zmax);
            sgGL.gl.glEnd();

            // Arriere
            sgGL.gl.glBegin(sgGL.GL_POLYGON);
            sgGL.gl.glNormal3f(0.0f, 0.0f, 1.0f);
            sgGL.gl.glVertex3f(xmin,ymin,zmax);
            sgGL.gl.glVertex3f(xmax,ymin,zmax);
            sgGL.gl.glVertex3f(xmax,ymax,zmax);
            sgGL.gl.glVertex3f(xmin,ymax,zmax);
            sgGL.gl.glEnd();
    }
}
```

```
/*************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*               File : TimerThread.java
*
*************************************************************/

package cv97.opengl;
import cv97.*;
import cv97.opengl.*;
import cv97.node.*;

public class TimerThread extends Thread {
TimeSensorNode mynode = null;
int delay;
   public TimerThread(String str, Node node) {
      super(str);
      mynode = (TimeSensorNode)node;
   }
   public void setNode(Node node) {
      mynode = (TimeSensorNode)node;
   }
   public void run() {
               int i=0;
               setPriority(5);
        while (true) {
               long tm=System.currentTimeMillis();
                 display((TimeSensorNode)mynode);
         //System.out.println(getName()+" "+i);
         tm+=delay;
         try {
             Thread.sleep(Math.max( 3, tm - System.currentTimeMillis()));
         } catch (InterruptedException e) {
               System.out.println("Interupt");
               break;}
      }
   }
               public void display(TimeSensorNode node) {
                //System.out.println("Appel Methode display de TimerThread");
                //Cette partie est inspiree de CyberVRML97 (nœud TimeSensor)
                              double currentTime = 0;
                              double startTime = node.getStartTime();
                              double stopTime = node.getStopTime();
                              double cycleInterval = node.getCycleInterval();
                              boolean bActive          = node.isActive();
                              boolean bEnable          = node.isEnabled();
                              boolean bLoop            = node.isLoop();

               node.setIsActive(bActive);
               node.setEnabled(bEnable);
               node.setLoop(bLoop);

                              if (currentTime == 0)
                                            currentTime = node.getCurrentSystemTime();

                              // isActive
                              if (bEnable == false && bActive == true) {
                                            node.setIsActive(false);
                                            node.sendEvent(node.getIsActiveField());
                                            System.out.println("bEnable=false&&bActive=true");
                                            return;
                              }
```

```
                        if (bActive == false && bEnable == true) {
                                if (startTime <= currentTime) {
                                        if (bLoop == true && stopTime <= startTime)
                                                bActive = true;
                                        else if (bLoop == false && stopTime <= startTime)
                                                bActive = true;
                                        else if (currentTime <= stopTime) {
                                                if (bLoop == true && startTime < stopTime)
                                                        bActive = true;
                                                else if (bLoop == false && startTime <
(startTime + cycleInterval) && (startTime + cycleInterval) <= stopTime)
                                                        bActive = true;
                                                else if (bLoop == false && startTime < stopTime &&
stopTime < (startTime + cycleInterval))
                                                        bActive = true;
                                        }
                                }
                                if (bActive) {
                                        node.setIsActive(true);
                                        node.sendEvent(node.getIsActiveField());
                                        node.setCycleTime(currentTime);
                                        node.sendEvent(node.getCycleTimeField());
                                }
                        }

                        currentTime = node.getCurrentSystemTime();

                        if (bActive == true && bEnable == true) {
                                if (bLoop == true && startTime < stopTime) {
                                        if (stopTime < currentTime)
                                                bActive = false;
                                }
                                else if (bLoop == false && stopTime <= startTime) {
                                        if (startTime + cycleInterval < currentTime)
                                                bActive = false;
                                }
                                else if (bLoop == false && startTime < (startTime + cycleInterval) &&
(startTime + cycleInterval) <= stopTime) {
                                        if (startTime + cycleInterval < currentTime)
                                                bActive = false;
                                }
                                else if (bLoop == false && startTime < stopTime && stopTime < (startTime +
cycleInterval)) {
                                        if (stopTime < currentTime)
                                                bActive = false;
                                }

                                if (bActive == false) {
                                        node.setIsActive(false);
                                        node.sendEvent(node.getIsActiveField());
                                }
                        }

                        if (bEnable == false || node.isActive() == false) {
                                return;
                        }
                        // fraction_changed
                        double  fraction = (currentTime - startTime) % cycleInterval;
                        if (fraction == 0.0 && startTime < currentTime)
                                fraction = 1.0;
                        else
                                fraction /= cycleInterval;
                        node.setFractionChanged((float)fraction);
                        node.sendEvent(node.getFractionChangedField());
                        //System.out.println("Envoi evenement fraction_changed"+fraction);
                        // cycleTime
```

```
                        double      cycleTime   = node.getCycleTime();
                        double      cycleEndTime            = cycleTime + cycleInterval;
                        while (cycleEndTime < currentTime) {
                                //System.out.println("Boucle cycleTime");
                                node.setCycleTime(cycleEndTime);
                                cycleEndTime += cycleInterval;
                                node.setCycleTime(currentTime);
                                node.sendEvent(node.getCycleTimeField());
                        }
                        // time
                        node.setTime(currentTime);
                        node.sendEvent(node.getTimeField());
                }
}
```

```
/*************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*
*               File : VectorOpenGL.java
*
*************************************************************/

package cv97.opengl;

import java.util.Vector;
import cv97.*;
import cv97.opengl.*;
import cv97.node.*;

public class VectorOpenGL {

Vector      vectorPile,
            vectorScript,
            vectorSensor,
            vectorInterpolator;

      public VectorOpenGL () {
       vectorPile                   = new Vector();
       vectorScript                 = new Vector();
       vectorSensor                 = new Vector();
       vectorInterpolator = new Vector();
      }

            public void addPile(ItemVectorOpenGL item) {
                        vectorPile.addElement(item);
            }

            public void deletePile() {
                        vectorPile.removeElementAt(vectorPile.size()-1);
            }

            public int getPileCapacity() {
                        return vectorPile.capacity();
            }

            public ItemVectorOpenGL getPileElement(int index) {
                         return (ItemVectorOpenGL)vectorPile.elementAt(index);
            }

            public void addScript(VectorElementOpenGL element) {
```

```
                vectorScript.addElement(element);
        }

        public void addSensor(VectorElementOpenGL element) {
                vectorSensor.addElement(element);
        }

        public VectorElementOpenGL getSensor(int index) {
                //System.out.println("getSensor index = "+index);
                return (VectorElementOpenGL)vectorSensor.elementAt(index);
        }


        public void addInterpolator(VectorElementOpenGL element) {
                vectorInterpolator.addElement(element);
        }


}

/*****************************************************************
*
*       Navigateur Java-OpenGL
*       Copyright (C)  Jean-Marie ABISROR
*
*       File : VectorElementOpenGL.java
*
*****************************************************************/

package cv97.opengl;
import java.util.Vector;
import cv97.*;
import cv97.opengl.*;
import cv97.node.*;

public class VectorElementOpenGL {
Vector  vectorElement;

    public VectorElementOpenGL() {
         vectorElement = new Vector();
    }

        public void addElement(ItemVectorOpenGL item) {
                vectorElement.addElement(item);
        }

        public int size() {
                return vectorElement.size();
        }

        public ItemVectorOpenGL getElement(int index) {
                return (ItemVectorOpenGL)vectorElement.elementAt(index);
        }

}
```

```
/*****************************************************************
*
*               Navigateur Java-OpenGL
*               Copyright (C)  Jean-Marie ABISROR
*
*               File : ItemVectorOpenGL.java
*
*****************************************************************/

package cv97.opengl;
import cv97.*;
import cv97.opengl.*;
import cv97.node.*;

public class ItemVectorOpenGL {
  String nodeType;
  String nodeName;
  Object nodeOpenGL;


  public ItemVectorOpenGL() {}
}
```