Chapitre IX : VRML et la réalité virtuelle

IX.1 – Le concept de réalité virtuelle

Depuis quelques années, la recherche concernant l'interface homme-machine a suscitée beaucoup d'attention auprès de la presse et de l'industrie. Toute une technologie, connue sous l'appellation de "Réalité Virtuelle" est apparue. Cette technologie se donne pour but d'interagir avec les sens humains: la vue, l'ouï e, le toucher..

Le terme "Réalité Virtuelle" a été introduit par Jaron Lanier, fondateur de VPL Research en 1989. Il fait suite aux travaux sur les réalités artificielles (Myron Krueger, 1970), le cyberspace (William Gibson, 1984), et plus récemment sur les environnements virtuels et les mondes virtuels ...

Avec les techniques modernes de synthèse d'images, nous sommes aujourd'hui capables de créer des images d'un réalisme surprenant. Mais, aussi réalistes soient-elles, ces images n'autorisaient pas l'homme à y pénétrer ...

La Réalité Virtuelle (VR) a pour objectif de permettre aux hommes d'accéder à des mondes artificiels en trois dimensions et d'interagir avec les objets de ces mondes. Elle est basée sur les techniques de la synthèse d'images en temps réel auxquels s'ajoute quatre principes :

- l'immersion physique (casque, gants, retour d'efforts, etc.),
- l'immersion mentale (amène l'utilisateur à "plonger" dans l'environnement 3D),
- l'interactivité,
- la navigation.

IX.1.a : La réalité virtuelle immersive

La mise en oeuvre de ces principes d'immersion impliquent généralement l'utilisation de technologies et de périphériques spécifiques.

Les périphériques d'immersion

Les casques d'immersion (ou Head-mounted display : HMD) furent les premiers périphériques développés spécifiquement pour la réalité virtuelle. Le premier fut conçu par Evans et Sutherland en 1965. Le premier casque commercial fut celui de la société VPL Research en 1989.

Un casque possède généralement deux optiques dotées chacune d'un écran miniature (LCD ou tube cathodique pour les plus anciens) reproduisant ainsi une image stéréoscopique. Un capteur mesure également la position et l'orientation de la tête pour fournir à l'ordinateur les informations nécessaires à l'ajustement des images en fonction des mouvements de l'utilisateur.

Le casque est généralement utilisé avec des interfaces complémentaires comme des gants (gloves), joysticks, etc., pour permettre une interaction avec les objets du monde virtuel.

Figure 1 : Les périphériques de la réalité virtuelle

Les inconvénients de cette approche sont les suivants :

- ils ne conviennent qu'à une seule personne à la fois,
- ils sont extrêmement coûteux si l'on recherche une bonne qualité,
- ils demeurent lourds et encombrants,
- ils sont fatigants pour les yeux,
- ils peuvent rendre "malade"

Le BOOM : Binocular Omni-Orientation Monitor

Il fut développé par Fakespace. Il comprend un système de vision binoculaire associé à des bras de manipulation articulés. C'est un système lourd mais qui permettait une première forme d'interactivité.

Le système CAVE

Le système CAVE (Cave Automatic Virtual Environment) a été développé initialement à l'université de l'Illinois à Chicago. L'utilisateur n'a plus de casque, mais se trouve dans une salle cubique équipée de projecteurs stéréoscopiques. L'utilisateur peut alors porter des lunettes simples (rouge/verte) pour obtenir l'effet de relief. De plus, plusieurs personnes peuvent participer en même temps à l'expérience immersive.

Les simulateurs dédiés

Il est ensuite possible de construire des systèmes intégrés informatiques/mécaniques qui deviennent de véritables simulateurs pour reproduire au plus près les impressions ressenties dans le système réel. Les applications de ces systèmes onéreux sont principalement l'entraînement et le divertissement. La figure 2 les illustre, ainsi que le système CAVE.

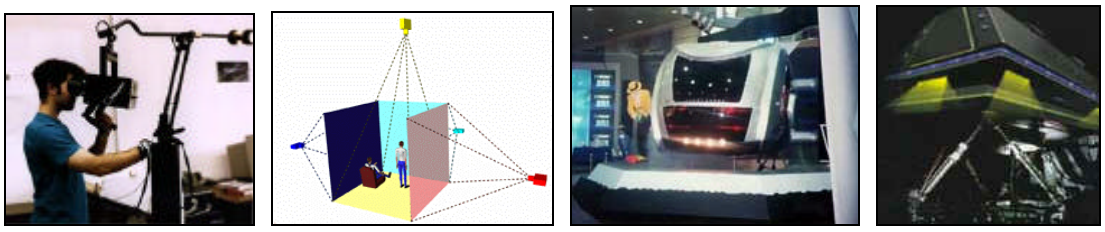

Figure 2 : Les système BOOM et CAVE et les simulateurs dédiés

IX.1.b : La réalité virtuelle non immersive

L'évolution de la réalité virtuelle est fonction de la technologie :

- en 1995, une machine produisant un million de polygones texturés par seconde coûtait plus d'un million de francs ;
- en 1996, elle ne coûtait plus que 200.000 Francs ;
- en 1997, cette performance était accessible sur un PC avec une carte graphique à 2.000 Francs ;
- en 2000, tous les PC sont équipés en standard de capacités bien supérieures.

L'apparition de cette potentialité technologique sur les ordinateurs de bureau a favorisé le développement de technologies logicielles accessibles aux développeurs d'applications multimédia.

Ces technologies sont souvent nommées ***non-immersives*** du fait qu'elle ne sont pas fondées sur une immersion totale de l'utilisateur. En effet, elles n'utilisent pas d'interfaces spécifiques, ce qui limite la sensation d'immersion, mais, en contrepartie, elles sont accessibles sur n'importe quel ordinateur.

Les deux technologies les plus répandues sont Quicktime VR d'Apple et VRML de Silicon Graphics.

Quicktime VR

QuickTime VR est une technologie développée par Apple, basée sur QuickTime où VR signifie Virtual Reality (réalité virtuelle). QuickTime est un ensemble de programmes qui permet de visualiser des films et des animations accompagnées de son.

QuickTime VR permet lui de visualiser des panoramas à 360°, construits à partir d'une série de photos prise d'un même point. Dans un panorama QuickTime VR, on peut donc tourner sur soi-même pour observer tout le panorama, et les images obtenues ont le même rendu que des photos numérisées.

Mais QuickTime VR permet aussi un autre type de vue : au lieu de faire tourner l'utilisateur sur lui-même, on peut faire tourner un objet 3D pour l'observer sous tous les angles. A partir de ces deux types de vues, on construit une scène complète. Une telle scène est composée de panoramas pris depuis plusieurs endroits différents.

Tous ces panoramas sont ensuite reliés par des liens "cliquables" pour pouvoir se déplacer à l'intérieur de la scène.

On peut ainsi visiter entièrement un site, en ayant une vue à 360° depuis tous les endroits où les vues ont été prises. Ce type de scène permet donc une visite virtuelle d'un site existant, avec un rendu photographique.

Une scène panoramique QTVR est créée à partir de photographies (avec une lentille 15mm) ou d'images de synthèse (avec un logiciel 3D). Pour la réalisation d'un panorama QTVR, il faut prendre une série de photographies en tournant l'appareil de 30° pour chaque prise de vue. Si on joint les photographies, directement dans un logiciel de retouche photo, pour réaliser un panorama, il y aura des joints désagréables (comme le démontre la première image de la figure 3).

Pour réussir à faire un panorama il faut utiliser l'ensemble disponible chez Apple (QuickTime VR Authoring Tool suite), qui permet de retoucher chacune des images pour créer une image *cylindrique* déformée ; les formes deviennent courbes et tout le paysage devient incurvé (comme le démontre la deuxième image de la figure 3).

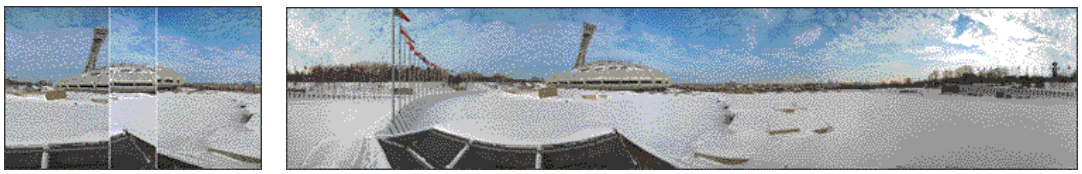

Figure 3 : Réalisation d'un panorama en QuickTime VR

L'ensemble de la technologie Quicktime VR est accessible sur le site Internet d'Apple : http://www.apple.com/quicktime/qtvr/index.html

VRML

VRML (Virtual Reality Modeling Language) est un langage informatique développé par un ensemble de chercheurs à l'initiative de la société Silicon Graphics.

VRML a été conçu pour permettre l'intégration d'environnements virtuels sur le Web. A la place de page HTML, les scènes VRML décrivent des mondes en trois dimensions dans lesquels l'utilisateur peut se déplacer, interagir et cliquer sur des hyperliens.

La version courante de VRML est VRML 2.0 qui est devenu un standard ISO/IEC sous le nom de VRML97

IX.2 – Présentation de VRML

Nous reprenons dans ce paragraphe, et nous tenons à les remercier de ce fabuleux travail, les pages didactiques de Groupe VRML Francophone (http://webhome.infonie.fr/kdo/vrml/index.htm)
Nous avons de plus intégré à ce cours des éléments issu de notre expérience propre. Pour de plus amples informations, on pourra se référer au site officiel du consortium VRML : http://www.vrml.org

IX.2.a : Historique

En 1993 Mark Pesce et Tony Parisi développèrent une interface 3D pour le web, laquelle incorporait une grande partie des recherches sur la réalité virtuelle et sur les réseaux (les liens hypertextes par exemple). Ils communiquèrent ces innovations à Tim Berners-Lee, l'inventeur du langage HTML, et Pesce fut invité à présenter une communication à la première conférence internationale sur le World Wide Web à Genève. Lors d'une réunion à propos des interfaces de réalité virtuelle les participants se mirent d'accord sur la nécessité de définir un langage commun permettant de décrire des scènes 3D et des hyperliens, tout comme le HTML, mais pour des applications de réalité virtuelle. Le terme ***Virtual Reality Modeling Language*** (VRML) fut adopté, et le groupe (présidé par Pesce et Brian Behlendorf, du magazine WIRED) commença à travailler sur la spécification du langage VRML immédiatement à la suite de cette conférence.

Avec l'aide du magazine WIRED, Behlendorf mit en place un système de mailing list pour faciliter les discussions sur la spécification de VRML. Le résultat dépassa leurs espérances, en une semaine ils obtinrent plus d'un millier de participants. Rapidement ils adoptèrent un ensemble de contraintes pour VRML, et se mirent en quête de technologies pouvant être adaptées pour les satisfaire.

Les membres proposèrent plusieurs candidats au titre du langage VRML, et après pas mal de délibérations, un consensus s'établit autour du langage ***Open Inventor*** de Silicon Graphics Inc (SGI). Le Format des fichiers texte Inventor permet la descriptions de scènes 3D complètes avec rendu des objets sous forme de polygones, de traiter l'éclairage, les textures et d'autres effets réalistes. Il possédait toutes les qualités attendues par les professionnels, une large diffusion, et de nombreux outils pour le développement.

Un sous-ensemble du format de fichier Inventor, possédant des extensions pour le réseau, est à la base de VRML. Gavin Bell de Silicon Graphics a adapté le format Inventor pour VRML. SGI a mis ce format dans le domaine publique et fournit un parser toujours dans le domaine publique pour faciliter le développement d'outils et de visualiseurs VRML.

VRML est conçut de manière à satisfaire 3 critères:

- Indépendance de la plate-forme (Windows, MAC OS, UNIX etc..)
- Extensibilité
- Travailler avec une faible bande passante (modem 14.4 kBps)

Sans être une extension de HTML, VRML en reprend les concepts. C'est un langage commun de description de scènes graphiques 3D et de gestion des hyperliens au même titre que l'est HTML pour les formats de textes et d'images des pages Web. Il ajoute une troisième dimension au World Wide Web.

VRML en est aujourd'hui à sa version 2 (1997)

Opérationnelle à partir de septembre 95, la version 1.0 a permis, à partir d'une page HTML, de référencer une scène ou un objet de cette scène en VRML. En cliquant, on pouvait directement accéder à un objet (en "activant le lien" vers celui-ci). De même, à partir d'un objet, on pouvait activer un autre lien et ainsi visualiser une autre scène, un texte, ou encore manipuler un autre objet. S'il était possible de manipuler des scènes statiques, on ne pouvait pas encore parler véritablement d'interaction au sens de la réalité virtuelle définie par les trois "I" : Imagination, Interaction, Immersion, dont la caractéristique essentielle est de plonger l'utilisateur dans la scène. La version proposée alors était assez limitée car elle n'intégrait pas les interactions, le son, ou la présence de plusieurs utilisateurs. Elle utilisait, en effet, davantage le "ML" (Langage de Modélisation) que le "VR" (Réalité Virtuelle proprement dite) du sigle VRML.

En 1997, la version 2.0 de VRML a offert un degré d'interactivité supérieur. Tout en conservant une compatibilité avec la précédente, celle-ci a été enrichie de fonctionnalités multimédia (son, vidéo, animation) en attribuant des comportements aux objets représentés et en gérant les interactions entre objets. Par ailleurs, une vraie programmation client/serveur a vu le jour, rendue possible par l'interaction avec Java.

IX.2.b : Installation et utilisation

Le VRML est reconnu par la plupart des modeleurs 3D et des navigateurs Internet. On peut écrire des fichiers VRML en utilisant un éditeur de texte ou un modeleur 3D. On peut les visualiser soit avec un navigateur VRML soit avec un ***module externe*** (plug-in) VRML sur un navigateur HTML

Les deux plug-in les plus connus sont CosmoPlayer de Silicon Graphics et WorldView de Platinum. Une fois installés, il suffit d'ouvrir les fichiers ".wrl" dans un navigateur. L'interface – plus ou moins complexe – permet de naviguer dans la scène, de tourner, de ce rapprocher des objets, de suivre un cheminement particulier, etc. La figure 4 présente l'interface de CosmoPlayer, insérée dans Netscape Navigator, sous l'environnement Silicon Graphics et avec Internet Explorer sous Windows.

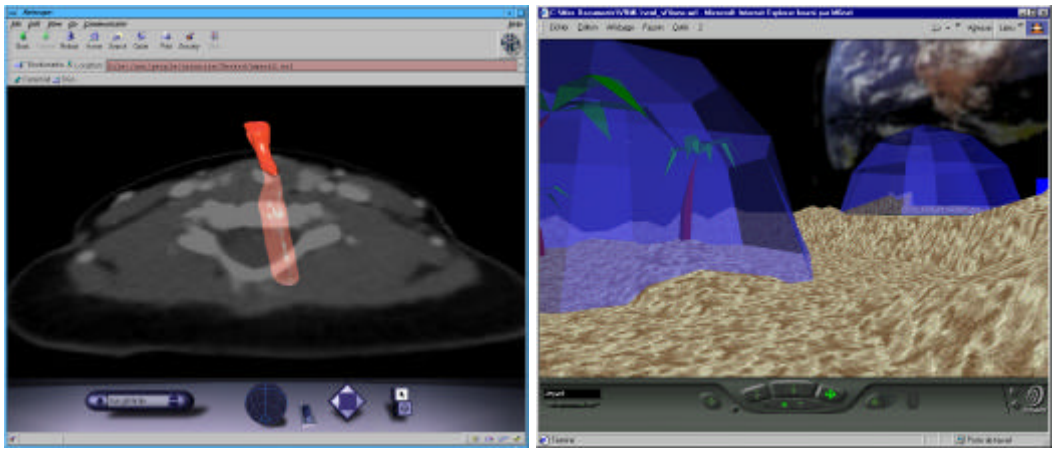

Figure 4 : Les interfaces de Navigation CosmoPlayer

Attention : VRML est un langage de programmation Orienté-Objet et, de ce fait, la plupart des remarques relatives à la rigueur lors du développement logiciel sont ici bien évidemment valides !

IX.2.c : Les bases de VRML

VRML est un langage ***descriptif*** et pas un langage de programmation. Cela signifie que vous pouvez décrire des environnements virtuels dans un fichier texte, mais que vous ne vous occupez pas de la manière dont cette description est ensuite utilisée pour générer les images sur l'écran. Tout le niveau mise en oeuvre est à la charge du visualiseur 3D. Tandis que la version 1.0 du langage ne permettait que la description d'environnements statiques, VRML 97 (petite mise à jour de VRML 2.0) permet en plus de décrire dans une certaine mesure des comportements dynamiques (animations) et plus d'interaction entre l'utilisateur et la scène. dans laquelle on regarde et ce que l'on voit. Mais ce n'est pas tout, la caméra peut être "penchée", en contre plongée, ou dans une position arbitraire : elle indique donc l'***angle*** avec lequel on observe les objets.

Conventions géométriques:

Par convention, les repères spatiaux sont orthonormés directs, et l'utilisateur fait face au plan (*X*,*Y*). Donc par défaut, on a l'axe *X* vers la droite, l'axe *Y* vers le haut, et l'axe *Z* vers soi.

Structure d'un fichier VRML 97:

Un fichier VRML 97 est un fichier texte, en général d'extension `".wrl"` ou `".vrml"`.
Tout fichier VRML 97 doit commencer par la ligne suivante qui constitue l'en-tête (*header*) standard:

```
#VRML V2.0 utf8
```

On peut inclure des lignes de commentaire en commençant la ligne par un `#'.

Un fichier VRML 97 est composé d'un ensemble de ***nœuds*** du langage. Un nœud du langage s'écrit sous la forme

```
MOT_CLÉ { PARAMÈTRES ... }
```

et permet de produire des formes ou des transformations géométriques, spécifier l'apparence des objets, rajouter du texte ... Tous les paramètres sont optionnels, et des valeurs par défaut sont garanties. Si on ne veut que les valeurs par défaut, il demeure obligatoire de mettre les accolades. Ainsi, pour créer un cube, on peut écrire simplement ceci:

```
Box { }
```

Les nœuds du langage ne peuvent pas s'utiliser n'importe comment et à n'importe quelle place dans le fichier. Il existe en particulier des ***nœuds de groupe*** qui peuvent contenir certains autres nœuds dans leur paramètres. Nous en verrons certains.

Il est possible de réutiliser des nœuds précédemment décrits grâce aux directives `DEF` et `USE`. La directive `DEF` permet d'assigner un nom à nœud particulier. Par exemple, pour définir un "cube" de taille 3, on peut utiliser la syntaxe suivante:

```
DEF cube Box { size 3 3 3 }
```

Ensuite, au lieu d'être obligé de réécrire le nœud et l'ensemble de ses paramètres à chaque fois que l'on en a besoin, il suffit simplement d'écrire:

```
USE cube
```

Note: L'appel à `DEF` exécute réellement le nœud en question. Il ne s'agit donc pas seulement d'une définition, car le nœud défini prend effet immédiatement.

IX.3 – Les primitives de forme simple en VRML

IX.3.a : Un premier exemple simple

L'objectif de ce premier exemple simple, dénué de toute prétention esthétique est de créer une scène comportant un cube. Ce n'est pas très beau, mais ... ça marche

```
#VRML V2.0 utf8
#-KDO - Première scène en VRML
# Juste un cube, blanc... pour les GBM
Shape {
      geometry Box {
            size  2.000 2.000 2.000
      }
}
```

Voici l'analyse de ce script :

Comme nous l'avons vu précédemment, un fichier VRML 2.0 commence TOUJOURS par cette ligne :

```
#VRML V2.0 utf8
```

Il s'agit d'un commentaire destiné à informer le navigateur sur la version de VRML utilisée.
Le caractère # est un indicateur de commentaire. Les lignes 2 et 3 sont donc des commentaires.

L'instruction `Shape` permet de commencer un ***bloc***, dans lequel une forme géométrique est définie.

Un bloc commence par une accolade ouvrante `{` et finit par une accolade...fermante `}`

L'instruction `geometry` précise quelle forme sera affichée, en l'occurrence une forme `Box`, c'est à dire un parallélépipède.

Dans le bloc définit pour Box, apparaît le paramètre `size` qui permet de définir la taille en largeur, hauteur et profondeur. Nous avons défini ici un cube de 2.0 unités de coté. La figure 5 illustre le résultat de ce script.

Figure 5 : Hello "cube" – le premier exemple VRML

Note : On considère qu'en VRML, l'unité de mesure est le mètre. Cela permet aux programmeurs de concevoir des objets dont l'échelle est cohérente entre eux. Mais ceci est une vue de l'esprit, vous êtes libres de penser qu'une largeur de 100, vaut cent mètres...ou 100 Km…

IX.3.b : Le bloc SHAPE

Le bloc `Shape` permet d'introduire une forme dans une scène. Cette forme sera définie par 2 champs permettant de déclarer 2 blocs.

```
Shape {
      exposedField SFNode appearance NULL
      exposedField SFNode geometry NULL
}
```

Le bloc `Appearence` qui permet de définir l'aspect visuel de la forme, et le bloc `Geometry` qui permet de définir le type de forme.

Note : Par défaut, la création d'un objet s'effectue aux coordonnées 0 0 0. Le centre de l'objet se situant à ces coordonnées. Nous verrons par la suite comment placer un objet en modifiant ses coordonnées.

Nous allons donc à présent étudier le bloc `Geometry`, et les formes simples (`Box`, `Cone`, `Cylinder`, `Sphere`), avant d'aborder ultérieurement les formes complexes. Le bloc `Appearance` sera lui aussi étudié plus tard, mais les exemples immédiats l'utiliseront pour colorer les objets.

IX.3.c : Le bloc 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 :

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

Voici un exemple simple de son utilisation (le résultat est illustré sur la figure 6):

```
# VRML V2.0 utf8
#-KDO - 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
      }
}
```

Figure 6 : Box

IX.3.d : Le bloc CONE

Le bloc `Cone` permet de définir un cône selon le formalisme suivant :

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

La signification des champs est la suivante :

- `bottomRadius` Définit le rayon de la base du cône, par défaut 1
- `height` Définit la hauteur du cône, par défaut 2 unités
- `side` Autorise l'affichage des cotés du cône, par défaut TRUE (VRAI)
- `bottom` Autorise l'affichage de la base du cône, par défaut TRUE

Voici un exemple de son utilisation :

```
# VRML V2.0 utf8
#-KDO - 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
      }
}
```

La figure 7 illustre cet exemple

Figure 7 : Cone

IX.3.d : Le bloc CYLINDER

Le bloc `Cylinder` permet de définir un cylindre selon le formalisme suivant :

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

La signification des champs est la suivante :

`Bottom`	Autorise l'affichage de la base du cylindre, par défaut TRUE (VRAI)
`height`	Définit la hauteur du cylindre, par défaut 2 unités
`radius`	Définit le rayon de la base du cylindre, par défaut 1
`side`	Autorise l'affichage des cotés du cylindre, par défaut TRUE
`top`	Autorise l'affichage du sommet du cylindre, par défaut TRUE

Voici un exemple de son utilisation :

```
# VRML V2.0 utf8
#-KDO - Shape : Cylinder
# Un cylindre mauve creux
Shape {
      appearance Appearance {
            material  Material {
                  ambientIntensity 0.4
                  diffuseColor  1.0 0.2 1.0
            }
      }
      geometry Cylinder {
            radius  2.2
            height 2.5
            bottom FALSE
            top  FALSE
      }
}
```

La figure 8 illustre cet exemple

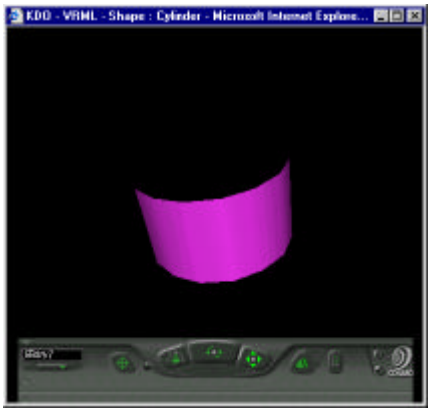

Figure 8 : Cylinder

IX.3.e : Le bloc SPHERE

Le bloc `Sphere` permet de définir une sphère. Il ne contient qu'un seul champ (`radius`), permettant de définir le rayon de la sphère.

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

Voici un exemple de son utilisation :

```
# VRML V2.0 utf8
#-KDO - Shape : Sphere
# Une sphère bleue
Shape {
      appearance Appearance {
            material  Material {
                  ambientIntensity 0.4
                  diffuseColor  0.3 0.3 1.0
            }
      }
      geometry Sphere {
            radius  2.2
      }
}
```

La figure 9 illustre cet exemple.

Figure 9 : Sphere

IX.4 – La couleur et les textures

IX.4.a : Le bloc APPEARANCE

Nous avons vu au paragraphe IX.3, le bloc `Shape` qui permet d'introduire une forme dans une scène. Cette forme étant définie par 2 champs permettant de déclarer 2 blocs. Le bloc

Appearance qui permet de définir l'aspect visuel de la forme et que nous allons voir, et le bloc Geometry qui permet de définir le type de forme que nous avons déjà vu. Le bloc Appearance a la syntaxe suivante :

```
Appearance {
      exposedField SFNode material NULL
      exposedField SFNode texture NULL
      exposedField SFNode textureTransform NULL
}
```

Le bloc Appearance va nous permettre de contrôler l'aspect visuel de la forme Shape auquel il appartient. Ses champs sont les suivants :

- material Permet de déclarer un bloc Material
- texture Permet de déclarer un bloc Texture
- textureTransform Permet de déclarer un bloc TextureTransform

Note : Les 3 champs sont optionnels, mais au moins l'un d'entre eux doit être présent si un bloc Appearance est déclaré.

IX.4.b : Le bloc MATERIAL

Le bloc Material permet de modifier l'aspect visuel de la forme associée . On peut ainsi modifier, la couleur, la luminosité propre de l'objet, sa transparence etc... Il possède la syntaxe suivante :

```
Material {
      exposedField SFFloat ambientIntensity 0.2
      exposedField SFColor diffuseColor 0.8 0.8 0.8
      exposedField SFColor emissiveColor 0 0 0
      exposedField SFFloat shininess 0.2
      exposedField SFColor specularColor 0 0 0
      exposedField SFFloat transparency 0
}
```

La signification des 6 champs possibles est la suivante :

- ambientIntensity Permet de modifier la luminosité propre de l'objet
- difuseColor Permet de donner une couleur à l'objet
- emissiveColor Permet de définir la couleur de rayonnement (lueur)
- shininess Permet de définir le degré de brillance de l'objet
- specularColor Permet de définir la couleur du spot de brillance
- transparency Permet de définir le degré de transparence de l'objet

En voici un exemple, sur une sphère.

```
# VRML V2.0 utf8
#-KDO - Shape - Appearance: Material
# Une sphère verte brillante
Shape {
```

```
      appearance Appearance {
            material  Material {
                  ambientIntensity 0.4
                  diffuseColor  0.2 0.7 0.2
                  shininess 1.0
                  specularColor 0.3 0.3 0.3
            }
      }
      geometry Sphere {
            radius  1.5
      }
}
```

La figure 10 illustre cet exemple.

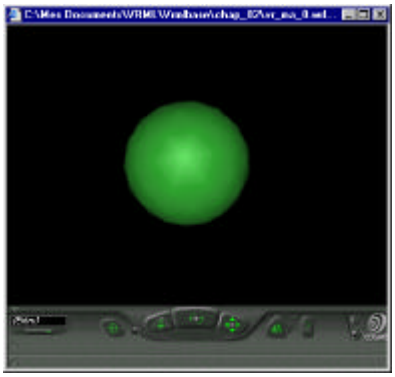

Figure 10 : Une sphère brillante

IX.4.c : Les textures

Il existe plusieurs façon de définir et d'appliquer une texture à un matériau. La première méthode est de définir une texture en mode point :

Le bloc PIXELTEXTURE

Le bloc `PixelTexture` permet de texturer un objet en lui appliquant un motif défini par un ensemble de pixels. Sa syntaxe est la suivante :

```
PixelTexture {
      exposedField SFImage image 0 0 0
      field SFBool repeatS TRUE
      field SFBool repeatT TRUE
}
```

La signification des champs est définie par :

- `image` Définition du motif : largeur hauteur résolution.
  résolution :
  1 = 8 bits - Echelle de gris, 256 niveaux
  2 = 8 bits en échelle de gris + 8 bits de transparence
  3 = 24 bits RGB
  4 = 24 bits RGB + 8 bits de transparence

- `repeatS` Autorise/interdit la répétition horizontale du motif. Voir la définition du bloc `TextureTransform`
- `repeatT` Autorise/interdit la répétition verticale du motif. Voir la définition du bloc `TextureTransform`

Voici un exemple de son utilisation :

```
# VRML V2.0 utf8
#-KDO - Shape - Appearance - PixelTexture
# Une sphère texturée avec un damier Bleu-Jaune
Shape {
      appearance Appearance {
            material  Material {
                  ambientIntensity 0.4
                  diffuseColor  0.3 0.3 1.0
            }
            texture PixelTexture {
                  image 2 2 3
                  0x0000FF 0xFFFF00
                  0xFFFF00 0x0000FF
            }
      }
      geometry Sphere { }
}
```

On défini donc ici une image de 2 par 2 pixels. Les 4 pixels suivant donc, codés ligne par ligne, en hexadécimal en mode 3 : RVB selon le formalisme VRML:

BLEU (0, 0, 255)	JAUNE (255, 255, 0)
JAUNE (255, 255, 0)	BLEU (0, 0, 255)

La figure 11 illustre la résultat. Comme le motif n'a pas subi de transformation d'échelle, il est étendu, étalé (mais non répété) sur toute la surface. On notera la puissance de ce procédé.

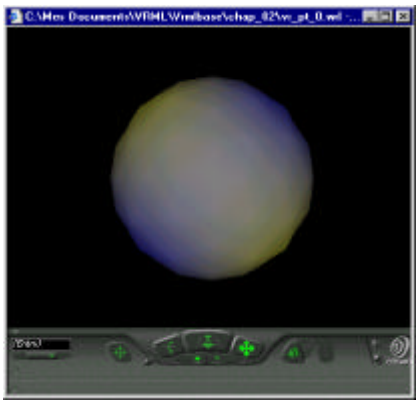

Figure 11 : Utilisation de `PixelTexture` pour le rendu d'une sphère

Le bloc IMAGETEXTURE

Le bloc `ImageTexture` permet d'utiliser une image comme texture d'un objet. : Les formats graphiques supportés par les visualiseurs VRML sont le GIF, le JPEG et le PNG. Seul le

JPEG ne supporte pas la transparence. Le PNG supporte plusieurs niveaux de transparence. La syntaxe est la suivante :

```
ImageTexture {
      exposedField MFString url []
      field SFBool repeatS TRUE
      field SFBool repeatT TRUE
}
```

Voici la signification des divers champs :

- `url` Adresse de l'image réalisant la texture. Ce champ peut contenir plusieurs adresses qui seront parcourues par ordre décroissant de préférences.
- `repeatS` Autorisation de la répétition horizontale de l'image. S'utilise en conjonction avec le bloc `TextureTransform`.
- `repeatT` Autorisation de la répétition verticale de l'image. Même remarque que pour `repeatS`

Ne dérogeons pas à la règle, voici un petit exemple :

```
# VRML V2.0 utf8
#-KDO - Shape - Appearance - ImageTexture
# Une sphère avec le logo VRML RING
Shape {
      appearance Appearance {
            material  Material {
                  ambientIntensity 0.4
                  diffuseColor  0.3 0.3 1.0
            }
            texture ImageTexture {
                  url  "/rsc/vrmlring.jpg"
            }
      }
      geometry Sphere {
            radius  2.2
      }
}
```

La structure hiérarchique du langage VRML apparaît très bien ici. On commence par déclarer un bloc Shape, pour dire que l'on veut dessiner une forme, ensuite à l'intérieur de ce bloc on utilise un bloc `Appearance` pour dire quel sera l'aspect visuel de notre forme. On introduit alors un bloc `Material` qui va permettre de spécifier l'intensité lumineuse propre à l'objet ainsi qu'une couleur par défaut. Toujours à l'intérieur du bloc `Appearance`, on déclare une texture utilisant une image avec le bloc `ImageTexture`. On referme le bloc `Appearance`, et le bloc `Sphere` vient préciser quelle forme on veut dessiner.

Le bloc TEXTURETRANSFORM

Le bloc `TextureTransform` permet d'appliquer des transformations à une texture : rotation, décalage et facteur d'échelle. La syntaxe est la suivante :

```
TextureTransform {
      exposedField SFVec2f center 0 0
```

```
      exposedField SFFloat rotation 0
      exposedField SFVec2f scale 1 1
      exposedField SFVec2f translation 0 0
}
```

Les champs sont les suivants :

- `center` Permet de spécifier le point à partir duquel auront lieux les transformations. Par défaut (0 0) le coin gauche inférieur. Le coin droit supérieur est 1 1.
- `rotation` Permet de spécifier un angle de rotation (en radians) de la texture par rapport aux coordonnées définies par center
- `scale` Permet d'appliquer un facteur d'échelle horizontal et vertical. La taille sera divisée par ce facteur sur l'axe considéré. Si les champs repeatS et repeatT de la texture sont à TRUE (par défaut) alors l'image sera répétée pour remplir la surface de la face considérée.
- `translation` Permet de spécifier un décalage de l'image horizontal et vertical (S et T). Valeurs comprises entre 0 et 1

Voici un exemple utilisant `TextureTransform` :

```
# VRML V2.0 utf8
#-KDO - Shape - Appearance - TextureTransform
# Une boite texturée avec le logo VRML RING
# Texture répétée 2 fois sur les 2 directions
Shape {
      appearance Appearance {
            material  Material {
                  ambientIntensity 0.4
                  diffuseColor  0.3 0.3 1.0
            }
            texture ImageTexture {
                  url  "../rsc/vrmlring.jpg"
            }
            textureTransform TextureTransform {
                  scale 2 2
            }
      }
      geometry Box { }
}
```

La figure 12 présente les deux exemples de texture utilisant une image. Ce petit programme se sert d'une image pour recouvrir le cube. Dans le bloc `TextureTransform`, nous appliquons un facteur d'échelle horizontal et vertical (S et T) de 2, ce qui divise par 2 la taille de notre image sur les 2 axes. Comme `repeatS` et `repeatT` d' `ImageTexture` sont à TRUE (pas déclarés, donc valeur par défaut), l'image est répétée 2 fois sur les 2 axes, et cela pour chaque face de notre cube. Par ailleurs on remarquera que le bloc `Box` est vide, donc que nous utilisons les valeurs par défaut du paramètre size, soit 2 2 2.

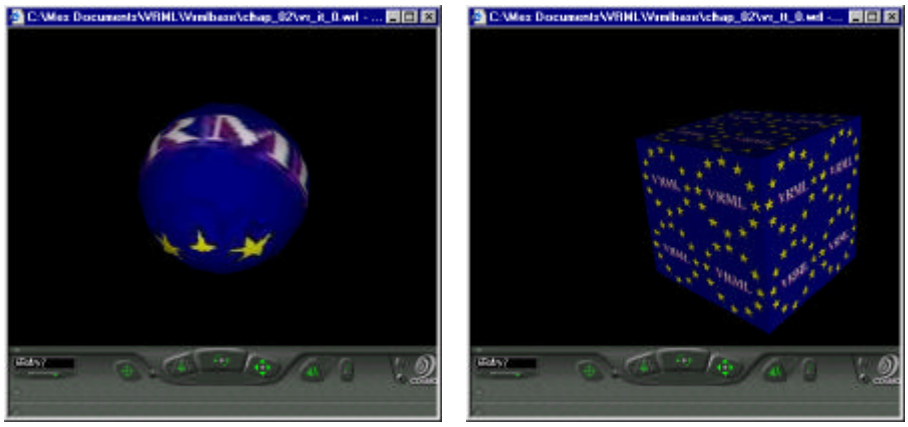

Figure 12 : Utilisation de textures issues d'images

IX.5 : Positionner les objets et naviguer dans la scène

Nous avons vu dans les paragraphes précédents comment créer des formes simples, les colorer ou leur affecter une image comme texture. Jusqu'à présent nos scènes ne comportaient qu'un seul objet dont la création s'effectuait aux coordonnées 0 0 0. Dans ce chapitre nous allons apprendre à positionner un objet aux coordonnées que l'on désire, et de ce fait nous construirons une scène comportant plusieurs objets. Cette scène sera par la suite utilisée comme scène de base pour l'étude de différentes instructions. Ce paragraphe va de plus introduire la création des caméras (ou points de vue) différentes et de paramétrer le logiciel de navigation (dans la limite des options présentes).

IX.5.a : La représentation des axes en VRML

Le système de coordonnées utilisé en VRML, qui est un système cartésien orthogonal. On notera que pour placer un objet en profondeur dans la scène il faut lui attribuer une coordonnée négative sur l'axe Z, puisque celui-ci est croissant dans la direction de l'utilisateur.

IX.5.b : Le bloc TRANSFORM

Le bloc `Transform` permet de positionner, de mettre à l'échelle et de faire effectuer des rotations à un bloc ou a un ensemble de blocs . Sa syntaxe est la suivante :

```
Transform {
      eventIn MFNode addChildren
      eventIn MFNode removeChildren
      exposedField SFVec3f center 0 0 0
      exposedField MFNode children []
      exposedField SFRotation rotation 0 0 1 0
      exposedField SFVec3f scale 1 1 1
      exposedField SFRotation scaleOrientation 0 0 1 0
      exposedField SFVec3f translation 0 0 0
      field SFVec3f bboxCenter 0 0 0
      field SFVec3f bboxSize -1 -1 -1
}
```

Ses champs se distinguent de la façon suivante :

- `addChildren` Evènement. Permet d'ajouter un bloc enfant
- `removeChildren` Evènement. Permet de supprimer un bloc enfant
- `center` Permet de spécifier un centre pour l'opération de changement du facteur d'échelle
- `children` Contient le ou les blocs subissant la transformation
- `rotation` Permet d'effectuer une rotation en donnant un vecteur (*x*,*y*,*z*) et un angle en radians
- `scale` Permet de modifier le facteur d'échelle
- `scaleOrientation` Permet de spécifier un vecteur de rotation pour l'opération de mise à l'échelle
- `translation` Permet de positionner un objet par rapport au point (0 0 0)
- `bboxCenter` Permet de spécifier le centre d'une boite englobante
- `bboxSize` Permet de créer et de spécifier la taille d'une boite englobante invisible

Voici un exemple illustrant l'emploi de `Transform` :

```
#VRML V2.0 utf8
#-------------------------------------------------
# Description  : Fichier à inclure pour cours
# Auteur       : KDO
#-------------------------------------------------
#------------------[ Le Sol ]------------------
Transform {
      translation 0 -1.5 0
      children [
            Shape {
                  geometry Box { size 80 0.01 80 }
                  appearance Appearance {
                        material Material {
                              ambientIntensity 0.5
                        }
                        texture PixelTexture {
                              image 2 2 3
                              0x0000FF 0xEEEEEE
                              0xEEEEEE 0x0000FF
                        }
                        textureTransform TextureTransform {
                              scale 9 9
                        }
                  }
            }
      ]
}
#---------------[ 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 0 0 1 0.75
      children [
            Shape {
                  geometry Cone {}
                  appearance Appearance {
                        material Material {
                              ambientIntensity 0.3
                              diffuseColor 0.8 0 0
                        }
                  }
            }
      ]
}
```

Comme nous pouvons le constater, positionner un ou plusieurs objets ne complique pas vraiment les choses. Il suffit de spécifier la position avec le champ translation, et d'insérer notre objet dans le champ `children`. En ce qui concerne les rotations, on remarquera qu'il n'est pas aisé de manipuler un vecteur et un angle en radians.... La figure 13 illustre le résultat de cet exemple.

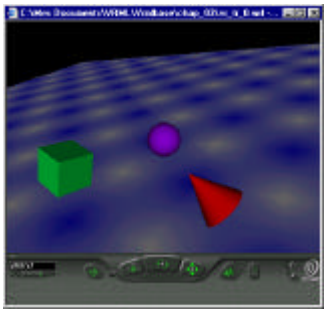

Figure 13 : Plusieurs objets crées et déplacés grâce à `Transform`

IX.5.c : Paramétrer la navigation

Le bloc NAVIGATIONINFO

Le bloc `NavigationInfo` permet de paramétrer certaines options du visualiseur. Sa syntaxe est la suivante :

```
NavigationInfo {
        eventIn SFBool set_bind
        exposedField MFFloat avatarSize [0.25, 1.6, 0.75]
        exposedField SFBool headlight TRUE
        exposedField SFFloat speed 1.0
        exposedField MFString type "WALK"
        exposedField SFFloat visibilityLimit 0.0
        eventOut SFBool isBound
}
```

La signification des champs est la suivante :

- `set_bind` Evènement. Permet d'activer / désactiver le bloc
- `avatarSize` Permet de spécifier la taille de l'utilisateur. Utilisé pour la détection de collision et pour le suivi de terrain.
- `headlight` Active / inhibe l'éclairage frontal.
- `speed` Vitesse de déplacement
- `type` Type de navigation (WALK, FLY, EXAMINE, NONE)
- `visibilityLimit` Détermine la distance jusqu'à laquelle l'utilisateur peut voir
- `isBound` Evènement. Renvoie l'état actif (TRUE) / inactif (FALSE) lors d'un `set_bind`

Le bloc VIEWPOINT

Le bloc `Viewpoint` permet de créer une caméra, de la positionner, de l'orienter etc. sa syntaxe est la suivante :

```
Viewpoint {
      eventIn SFBool set_bind
      exposedField SFFloat fieldOfView 0.785398
      exposedField SFBool jump TRUE
      exposedField SFRotation orientation 0 0 1 0
      exposedField SFVec3f position 0 0 10
      field SFString description ""
      eventOut SFTime bindTime
      eventOut SFBool isBound
}
```

La signification des champs suit :

- `set_bind` Evènement. Permet d'activer (TRUE) ou désactiver (FALSE) le Viewpoint
- `fieldOfView` Permet de modifier l'angle de vue (en radians)
- `jump` Permet de choisir une transition lente (TRUE) ou immédiate (FALSE) entre 2 points de vue.

- `orientation` Permet d'orienter la caméra selon un vecteur et un angle
- `position` Permet de positionner la caméra aux coordonnées x y z
- `description` Label, apparaissant dans le menu de positionnement de certains visualiseurs
- `bindTime` Evènement. Renvoie le temps actuel lors de la réception d'un `set_bind`
- isBound Evènement. Renvoie l'état activé (TRUE) / inactif (FALSE) après un `set_bind`

Voici un exemple :

```
# VRML V2.0 utf8
#- KDO - Viewpoint
Viewpoint {
      position    0 1 16
      description "Position 1"
      fieldOfView 0.55
}
Viewpoint {
      position    16 1 -1
      orientation 0 1 0 1.5
      description "Position 2"
      fieldOfView 0.55
}
Inline {
      url "vr_tr_0.wrl"
}
```

Figure 14 : Utilisation de `Viewpoint`

Nous avons déclaré 2 caméras, offrant ainsi 2 points de vue différents. Avec Cosmo Player 2.0 par exemple, il est possible de passer de l'un à l'autre en utilisant le petit bouton à gauche du visualiseur (Figure 14). Avec de nombreux Viewpoints, il est possible de réaliser ainsi de véritables cheminements dans la scène.

La scène visualisée, est la scène décrite pour le cours sur le bloc `Transform`, que l'on a inclus dans le fichier avec l'instruction `Inline`.

IX.6 : L'éclairage et l'environnement

Pour l'étude de ce paragraphe, nous utiliserons la scène définie dans le paragraphe IX.4 lors de l'étude du bloc `Transform`. Le fichier sera inclus dans une autre scène, nous permettant ainsi de comparer les diverses modifications d'éclairage et d'environnement. Cette scène type est contenue dans le fichier `vr_tr_0.wrl`.

IX.6.a : L'éclairage neutre

La scène suivante affiche notre scène type d'une façon neutre par rapport aux éclairages et à l'environnement.

```
#VRML V2.0 utf8
# Description  : Fichier exemple
# Auteur       : KDO
#--------[ Paramétrage Browser ]---------
NavigationInfo {
      headlight   TRUE
      type        "EXAMINE"
}
#------[ Définition du Viewpoint ]-------
Viewpoint {
      position    0 1 16
      description "Position de départ"
      fieldOfView 0.35
      jump TRUE
}
Inline {
      url "../chap_03/vr_tr_0.wrl"
}
```

Le fait que `headlight` soit à TRUE, permet d'éclairer la scène avec un éclairage frontal.

IX.6.b : L'éclairage Directionnel

Le bloc `DirectionalLight` permet de créer une source lumineuse directionnelle, non localisée. Sa syntaxe est la suivante :

```
DirectionalLight {
      exposedField SFFloat ambientIntensity 0
      exposedField SFColor color 1 1 1
      exposedField SFVec3f direction 0 0 -1
      exposedField SFFloat intensity 1
      exposedField SFBool on TRUE
}
```

La signification des champs est la suivante :

- `ambientIntensity` Détermine de combien la lumière participe à l'éclairage total
- `color` Couleur de la lumiere
- `direction` Vecteur de direction

- `intensity` Intensité de la source lumineuse
- `on` Activation / Désactivation de la source (activée par défaut TRUE)

Voici un exemple d'utilisation :

```
#VRML V2.0 utf8
#- KDO - Cours VRML : DirectionalLight
#----------[ Paramétrage Browser ]
NavigationInfo {
      headlight  FALSE
      type       "EXAMINE"
}
#----------[Vue initiale ]
      Viewpoint {
            position    0 1 28
            description "Position de départ"
            fieldOfView 0.35
            jump TRUE
}
#----------[ La Lumière ]
DirectionalLight {
      ambientIntensity 0.8
      direction    0 -1 0
}
#----------[ Les Objets ]
Inline {
      url "../chap_03/vr_tr_0.wrl"
}
```

Le résultat apparaît sur la figure 15

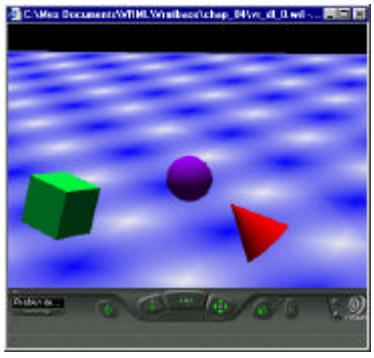

Figure 15 : Utilisation d'une lumière directionnelle

Nous avons défini une lumière directionnelle éclairant dans la direction du bas (vecteur Y de direction = -1). Ce type de lumière est surtout utilisé pour réaliser un éclairage global de type lumière du jour.

IX.6.c : L'éclairage ponctuel

Le bloc `PointLight` permet de créer une source lumineuse localisée. Cette source est ***omnidirectionnelle***.

Sa syntaxe est la suivante :

```
PointLight {
      exposedField SFFloat ambientIntensity 0
      exposedField SFVec3f attenuation 1 0 0
      exposedField SFColor color 1 1 1
      exposedField SFFloat intensity 1
      exposedField SFVec3f location 0 0 0
      exposedField SFBool on TRUE
      exposedField SFFloat radius 100
}
```

La signification des champs est la suivante :

- `ambientIntensity` Détermine de combien la lumière participe à l'éclairage total
- `attenuation` Permet de spécifier une direction d'atténuation
- `color` Couleur de la source lumineuse
- `intensity` Intensité de la lumière. Entre 0 et 1 inclus
- `location` Position de la source
- `on` Source active / inactive. Active par défaut
- `radius` Portée de la source lumineuse

Voici un exemple d'utilisation :

```
#VRML V2.0 utf8
#- KDO - Cours VRML : PointLight
#----------[ Paramétrage Browser ]
NavigationInfo {
      headlight  FALSE
      type       "EXAMINE"
}
#----------[Vue initiale ]
Viewpoint {
      position    0 1 28
      description "Position de départ"
      fieldOfView 0.35
      jump TRUE
}
#----------[ La Lumière ]
PointLight {
      ambientIntensity 0.8
      location   0 1 0
}
#----------[ Les Objets ]
Inline {
      url "../chap_03/vr_tr_0.wrl"
}
```

IX.6.d : L'éclairage par spot

Le bloc `SpotLight` permet de créer une source lumineuse localisée et directionnelle de type spot. Sa syntaxe est la suivante :

```
SpotLight {
```

```
      exposedField SFFloat ambientIntensity 0
      exposedField SFVec3f attenuation 1 0 0
      exposedField SFFloat beamWidth 1.570796
      exposedField SFColor color 1 1 1
      exposedField SFFloat cutOffAngle 0.785398
      exposedField SFVec3f direction 0 0 -1
      exposedField SFFloat intensity 1
      exposedField SFVec3f location 0 0 0
      exposedField SFBool on TRUE
      exposedField SFFloat radius 100
}
```

La signification des champs est un peu plus technique :

- `ambientIntensity` Détermine de combien la source contribue à l'éclairage total
- `attenuation` Vecteur définissant la direction de l'atténuation
- `beamWidth` Angle définissant le cône intérieur dans lequel la lumière est maximum voir figure 16
- `color` Couleur de la source lumineuse
- `cutOffAngle` Angle définissant le cône total de lumière. Inférieur ou égale à 180° soit environ 1.57 radians.
- `direction` Vecteur de direction de la source
- `intensity` Intensité lumineuse de la source
- `location` Position de la source
- `on` Permet d'activer / désactiver la source
- `radius` Portée maximum des rayons lumineux

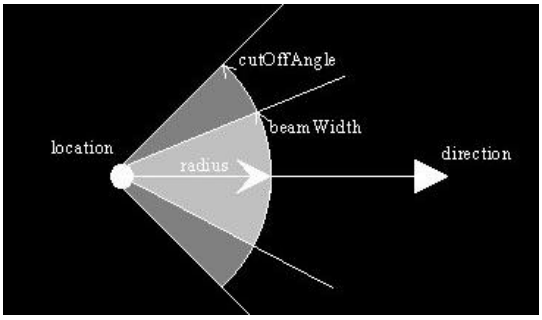

Figure 16 : La lumière de type SPOT

IX.6.e : L'arrière-plan

Le bloc `Background` permet de créer un arrière plan sphérique pour le ciel et le sol, et / ou de créer un panorama cubique englobant la scène, dont chacune des faces du cube peut recevoir une image. Sa syntaxe est la suivante :

```
Background {
      eventIn SFBool set_bind
      exposedField MFFloat groundAngle []
      exposedfield MFColor groundColor []
      exposedField MFString backUrl []
      exposedField MFString bottomUrl []
```

```
      exposedField MFString frontUrl []
      exposedField MFString leftUrl []
      exposedField MFString rightUrl []
      exposedField MFString topUrl []
      exposedField MFFloat skyAngle []
      exposedField MFColor skyColor [0 0 0]
      eventOut SFBool isBound
}
```

et voici la signification des champs :

- `skyAngle` Sert à spécifier une série d'angles associés à une série de couleurs dans `skyColor`
- `skyColor` Contient une série de couleurs, associée à `skyAngle`. Le nombre de couleurs doit être égale au nombres d'angles plus 1 (voir exemple)
- `groundAngle` Sert à spécifier une série d'angles associés à une série de couleurs dans `groundColor`
- `groundColor` Contient une série de couleurs, associée à `groundAngle`. Le nombre de couleurs doit être égale au nombres d'angles plus 1
- `backUrl` Adresse de l'image placée à l'arrière
- `bottomUrl` Adresse de l'image placée au sommet
- `frontUrl` Adresse de l'image placée à l'avant
- `leftUrl` Adresse de l'image placée à gauche
- `rightUrl` Adresse de l'image placée à droite
- `topUrl` Adresse de l'image placée en haut
- `set_bind` Evènement. Permet d'activer / désactiver l'arrière plan
- `isBound` Evènement. Renvoie l'état du Background après un set_bind

IX.6.f : Flou artistique

Le bloc `Fog` permet de créer un brouillard recouvrant la scène. Il permet donc de limiter la vision d'objets lointains ou/et de créer un effet dramatique. L'utilisation de l'option EXPONENTIAL donne un résultat plus réaliste. La syntaxe est la suivante :

```
Fog {
      exposedField SFColor color 1 1 1
      exposedField SFString fogType "LINEAR"
      exposedField SFFloat visibilityRange 0
      eventIn SFBool set_bind
      eventOut SFBool isBound
}
```

La signification des champs est :

- `color` Définit la couleur du brouillard
- `fogType` Type de brouillard : LINEAR ou EXPONENTIAL
- `visibilityRange` Distance d'occultation totale. Si 0, pas de brouillard
- `set_bind` Evènement. Permet d'activer / désactiver le bloc
- `isBound` Evènement. Reçut lors d'un `set_bind`

IX.7 : Les formes complexes

Jusqu'à présent, nous n'avons fait qu'utiliser des formes simples. Nous abordons ici les formes complexes qui, mis à part `PointSet` et éventuellement `IndexedLineSet`, semblent difficilement réalisables à la main.

L'utilisation d'utilitaires devient une étape obligatoire, dès lors que l'on aborde la création d'objets complexes. On notera que si les utilitaires ou modeleurs actuellement disponibles savent très bien gérer `IndexedFaceSet`, il en va tout autrement en ce qui concerne les autres primitives complexes...

Nous ne nous focaliserons ici que sur les objets "classiques" de l'imagerie 3D, à savoir les points, les arêtes et les faces. D'autres objets plus complexes peuvent être utilisés. On se reportera par exemple au site du Groupe VRML Francophone pour plus d'information.

IX.7.a : Les points

Le bloc `PointSet` permet de tracer un ensemble de points dont les coordonnées sont introduites à l'aide du sous bloc `Coordinate` et la couleur avec le sous bloc `Color`. Le bloc `PointSet` doit être déclaré dans un bloc `Shape`. Sa syntaxe est la suivante :

```
PointSet {
      exposedField SFNode color NULL
      exposedField SFNode coord NULL
}
```

La signification des champs est :

- `coord` Permet de déclarer un sous bloc `Coordinate` dans le quel seront spécifiées les coordonnées des points à afficher.
- `color` Permet de déclarer un sous bloc `Color` dans lequel seront déclarées les couleurs respectives de chaque point figurant dans le bloc `Coordinate`. Le nombre d'éléments doit être identique dans les deux blocs. Ce bloc est facultatif si l'on a déclaré préalablement un bloc `Material`.

Voici un exemple illustrant cet usage :

```
#VRML V2.0 utf8
#- KDO - Cours VRML : PointSet
NavigationInfo {
      type "EXAMINE"
}
DirectionalLight {
      ambientIntensity 1
}
Shape {
      geometry PointSet{
            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
                  ]
            }
            color Color {
                  color [
                        1 1 1, 1 1 1, 1 1 1, 1 1 1,
                        1 1 0, 1 1 0, 1 1 0, 1 1 0
                  ]
            }
      }
}
```

IX.7.b : Les lignes

Le bloc `IndexedLineSet` permet de tracer un ensemble de lignes. Sa syntaxe est la suivante :

```
IndexedLineSet {
      eventIn MFInt32 set_colorIndex
      eventIn MFInt32 set_coordIndex
      exposedField SFNode color NULL
      exposedField SFNode coord NULL
      field MFInt32 colorIndex []
      field SFBool colorPerVertex TRUE
      field MFInt32 coordIndex []
}
```

Les différents champs de ce bloc sont :

- `set_colorIndex` Événement. Permet de modifier `colorIndex`
- `set_coordIndex` Événement. Permet de modifier coordIndex
- `color` Permet de déclarer un sous bloc `Color` dans lequel seront déclarées les couleurs respectives de chaque portions définies par `coordIndex`.
- `coord` Permet de déclarer un sous bloc `Coordinate` dans le quel seront spécifiées les coordonnées des points à relier par une ligne
- `colorIndex` Permet de spécifier quelle couleur définie dans le bloc `Color` sera appliquée à chaque portion définie par `coordIndex` . Le nombre de valeurs doit être identique à celui des portions définies.
- `colorPerVertex` Permet d'effectuer un dégradé de couleurs entre des segments non convexes différents.
- `coordIndex` Permet de spécifier quels points définis dans le bloc `Coordinate` seront reliés ensembles. La fin d'une portion est identifiée par la valeur -1

Voici un exemple d'utilisation de `IndexedLineSet` et l'illustration correspondant en figure 17.

```
#VRML V2.0 utf8
#- KDO - Cours VRML : IndexedLineSet
#- Exemple 1
NavigationInfo {
      type "EXAMINE"
}
```

```
Shape {
      geometry IndexedLineSet {
            colorPerVertex 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 0 -1, 4 5 6 7 4 -1
                  0 4 -1, 1 5 -1, 2 6 -1, 3 7 -1
            ]
            color Color {
                  color [
                        0 0 1, 0 1 0, 1 1 0
                  ]
            }
            colorIndex [
                  0, 1, 2, 2, 2, 2
            ]
      }
}
```

Les points définis dans le bloc `Coordinate` forment un cube. A l'aide de `coordIndex`, nous traçons tout d'abord la face la plus proche en demandant de relier les points 0 1 2 3 0 (la valeur d'index du premier point définit dans le bloc `Coordinate` étant 0). Ceci constituant notre première portion, nous insérons la valeur -1 pour la terminer. Puis successivement nous traçons la portion constituant la face arrière (4 5 6 7 4), et les 4 portions permettant de relier les 2 faces.
Nous définissons ensuite dans le bloc `Color` 3 couleurs (bleu, vert et jaune). La première couleur étant désignée par la valeur 0 dans `colorIndex`.
Enfin dans `colorIndex` nous demandons d'affecter la couleur d'index 0 (bleu) à la première portion (la face avant du cube), la valeur d'index 1 (vert) à la deuxièmes portion (face arrière) et la valeur 2 (jaune) aux 4 dernières portions.

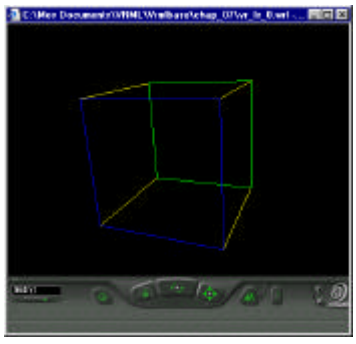

Figure 17 : `IndexedLineSet`

IX.7.b : Les faces

Le bloc `IndexedFaceSet` permet de réaliser des objets, constitués de facettes, aussi complexes qu'on le désire. La syntaxe est évidemment un peu plus complexe que pour `IndexedLineSet` :

```
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 []
}
```

Voici le détail de la signification des champs :

- `set_colorIndex` Événement. Permet de modifier `colorIndex`
- `set_coordIndex` Événement. Permet de modifier `coordIndex`
- `set_normalIndex` Événement. Permet de modifier `normalIndex`
- `set_texCoordIndex` Événement. Permet de modifier `texCoordIndex`
- `color` Permet de déclarer un sous bloc `Color` dans lequel seront déclarées les couleurs respectives de chaque faces définies par `coordIndex`.
- `coord` Permet de déclarer un sous bloc `Coordinate` dans le quel seront spécifiées les coordonnées des points constituant une face.
- `normal` Permet de déclarer un sous bloc Normal dans lequel on définira des vecteurs représentant la normale à la surface. Seul un utilitaire peut réaliser ce genre de calculs.
- `texCoord` Permet de déclarer un sous bloc `TextureCoordinate` dans lequel on définira des points de contrôle pour l'application d'une texture.
- `ccw` Permet de spécifier si les points définissant une face le sont dans le sens contraire des aiguilles d'une montre (TRUE) ou dans le sens des aiguilles d'une montre ou dans le désordre (FALSE)
- `colorIndex` Permet de spécifier quelle couleur définie dans le bloc `Color` sera appliquée à chaque face définie par `coordIndex` . Le nombre de valeurs doit être identique au nombre de faces.
- `colorPerVertex` Permet d'associer une couleur par point définissant l'objet. Si TRUE alors dégradé de couleurs.
- `convex` Permet de préciser si l'afficheur VRML doit considérer la surface comme convexe (fermée) ou non. La valeur FALSE ralentit l'afficheur qui doit calculer alors la fermeture de la surface.
- `coordIndex` Permet de spécifier quels points définis dans le bloc `Coordinate` seront reliés ensembles. Les différentes faces sont séparées par la valeur -1
- `creaseAngle` Permet d'adoucir l'angle formé par les faces en dessous d'une valeur d'angle spécifiée

- `normalIndex` Permet de spécifier quels vecteurs définis dans le bloc Normal sont utilisés
- `normalPerVertex` Permet d'associer un vecteur par point définissant l'objet
- `solid` Si FALSE, permet de dessiner les 2 cotés d'une face
- `texCoordIndex` Similaire à `coordIndex`, mais appliqué aux coordonnées de la texture déclarées dans `texCoord`.

```
#VRML V2.0 utf8
#- KDO - Cours VRML : IndexedFaceSet
NavigationInfo {
      type "EXAMINE"
}
Shape {
      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, 4 5 6 7 -1,
                  0 1 5 4 -1, 7 6 2 3 -1,
                  0 4 7 3 -1, 1 5 6 2
            ]
            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
                  ]
            }
      }
}
```

La figure 18 illustre le résultat :

Nous avons, dans le bloc `Coordinate`, défini les coordonnées de 8 points correspondant aux 8 coins d'un cube. Ensuite, dans `coordIndex`, nous avons construit les 6 faces du cube. Le bloc `Color` contient la définition des 8 couleurs associées aux 8 points définis dans `Coordinate`, puisque `colorPerVertex` est à TRUE (valeur par défaut) et donc ces couleurs seront affichées en dégradé.

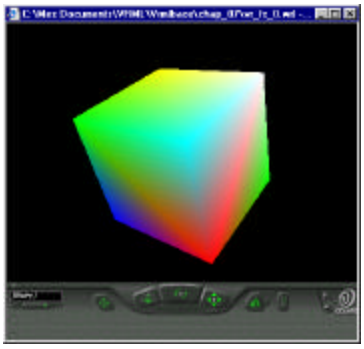

Figure 18 : Le Cube version IndexedFaceSet

Il existe de nombreuses autres fonctionnalités relevant plus de la partie "réalité virtuelle" de VRML. Nous nous sommes focalisés ici sur l'aspect description de scène. Pour plus d'information, on se réfèrera au site officiel du consortium VRML :

http://www.vrml.org

ou au site du Groupe VRML francophone :

http://webhome.infonie.fr/kdo/vrml/index.htm