Introduction à VRML 97

Mathias PAULIN, Maître de conférences,

Sabatier, Toulouse

paulin@irit.fr

Notes de cours à destination des étudiants en Maîtrise dInformatique.

Accessible par : http://www.irit.fr/ACTIVITES/EQ_SYNTHIM/enseignement/

Table des matières.

Introduction

Ce cours s'adresse aux étudiants des formations de l'université Paul Sabatier suivantes :

- Licence-Maîtrise de l'IUP Systèmes intelligents.
- Maîtrise d'Informatique.
- DESS Ingénierie de l'Image Numérique.

Ce cours est accessible en ligne à http://www.irit.fr/ACTIVITES/EQ_SYNTHIM/enseignement/. La version en ligne nécessite un navigateur correctement configuré intégrant un plugin de visualisation de mondes VRML (CosmoPlayer par exemple).

Afin de faciliter la consultation hors ligne de ce cours, une archive intégrant les pages html, les exemples et la spécification officielle du langage est disponible à la même adresse.

L'objectif de ce cours est de présenter les structures de bases du langage ainsi que ses possibilité pour la création et la présentation de mondes virtuels animés. En aucun cas il ne s'agit d'une présentation exhaustive du langage qui pourra être consultée dans le document de spécification http://www.vrml.org/technicalinfo/specifications/vrml97/index.htm.

La hiérarchie de pages html à été créée à l'aide d'un programme de formatage développé pour l'occasion par l'équipe synthèse d'images de l'IRIT travaillant à partir d'un fichier texte formaté.

Présentation du langage, graphe de scène

Structure du langage.

VRML (**V**irtual **R**eality **M**arkup **L**anguage) c'est :

Un langage simple pour décrire les mondes 3D virtuels interactif et multimédia.
Un moyen de réaliser des applications graphiques interactives sur internet.
Un fichier source (texte) VRML, identifié par son extension : **.wrl**
On peut visualiser un monde VRML dans un *navigateur VRML* :
Une application spécifique pour VRML
Un plug-in VRML dans un navigateur HTML
On peut visualiser un fichier VRML depuis un disque local ou depuis Internet
Un monde VRML peut être créer à partir de :
Un éditeur de texte (vi)
Un éditeur de monde VRML (WebSpaceAuthor)
Un programme de modélisation (3DS Max)
Un générateur de formes et de mouvements (Script perl)

Un monde VRML peut être visualisé dans une page HTML:

Pleine page	[boxes.wrl]
Inséré dans la page	[boxes1.html]
Pleine page dans une frame	[boxes2.html]
Inséré dans une frame	[boxes3.html]
Inséré plusieurs fois	[boxes4.html]

Un fichier VRML contient :
L'entête du fichier
#VRML V2.0 utf8
Commentaires - des commentaires et notes
 # Ceci est un commentaire
Noeuds - définition de la scène
Shape {
Champs - Attributs modifiables des noeuds
appearance
Valeurs - Valeur des attributs
Appearance {
plus . . .

Exemple de fichier VRML :

```
#VRML V2.0 utf8
# A Cylinder
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Cylinder {
        height 2.0
        radius 1.5
    }
}
```

#VRML V2.0 utf8
#VRML: Le fichier contiens un texte VRML.
V2.0 : Le texte est conforme à la version 2.0 du langage.
utf8 : Le texte utilise les caractères UTF8.
utf8 est le nom d'un jeu de caractères international.
utf8 est l'acronyme de :
UCS (Universal Character Set) Transformation Format, 8-bit
utf8 code plus de 24000 caractères pour de nombreuses langues.
ASCII en est un sous ensemble

Cylinder {
}
Les noeuds définissent les formes, lumières, sons ... du monde virtuel.
Chaque noeud possède :
Un *type* (**Shape**, **Cylinder**, etc.)
Une paire d'accolades
Zéro ou plus champs entre les accolades
Les nom des types sont *dépendants de la casse*
Chaque mot commence par une majuscule
le reste est en minuscule

Cylinder {
height 2.0
radius 1.5
}
Les champs définissent les attributs des noeuds.
Chaque champs possède :
Un nom (**height**, **radius**, etc.)
Un type de données (float, integer, etc.)
Une valeur par défaut
Les noms de champs sont *dépendants de la casse*
Le premier mot commence par une minuscule
Chaque mot supplémentaire commence par une majuscule
le reste des mots est en minuscule

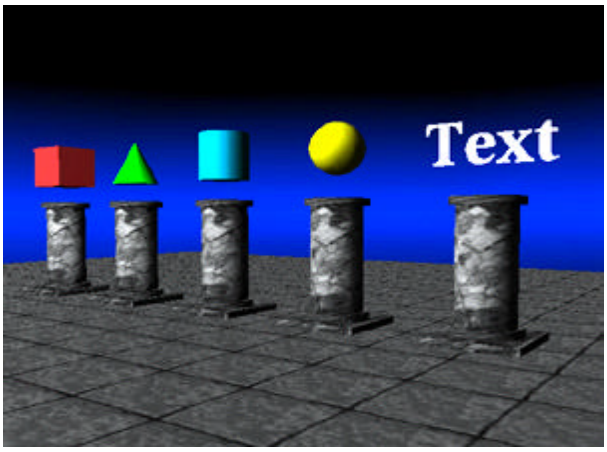
Les différents types de noeuds ont des champs différents
Les champs sont optionnels
La valeur par défaut pour un champ est utilisée s'il est absent
Les champs peuvent être donné dans n'importe quel ordre
L'ordre des champs n'affecte pas l'aspect du noeud
exemples:

nœuds	Champs
Appearance	**appearance**
Material	**material**
ElevationGrid	**radius**
ImageTexture	**coordIndex**
IndexedFaceSet	**textureTransform**

Géométrie et formes primitives

Les noeuds de type *Shapes* sont les briques de base d'un monde VRML.
Les *formes primitives* sont les noeuds *Shapes* standards VRML :

Box	Cone	Cylinder
Sphere	Text	



Shape : Syntaxe

Un noeud **Shape** construit une forme
Un noeud **Shape** possède les attributs suivants :
appearance : SFNode - couleur et texture.
geometry : SFNode - Forme primitive ou structure.

```
Shape {
   appearance . . .
   geometry  . . .
}
```

Shape : Spécification de l'apparence

L'apparence d'un noeud *Shape* est définie par l'attribut *appearance* dont la valeur est un nœud.
Utilisation d'un noeud *Material* pour spécifier une apparence blanche ombrée :

```
Shape {
   appearance Appearance {
      material Material { }
   }
   geometry  . . .
}
```

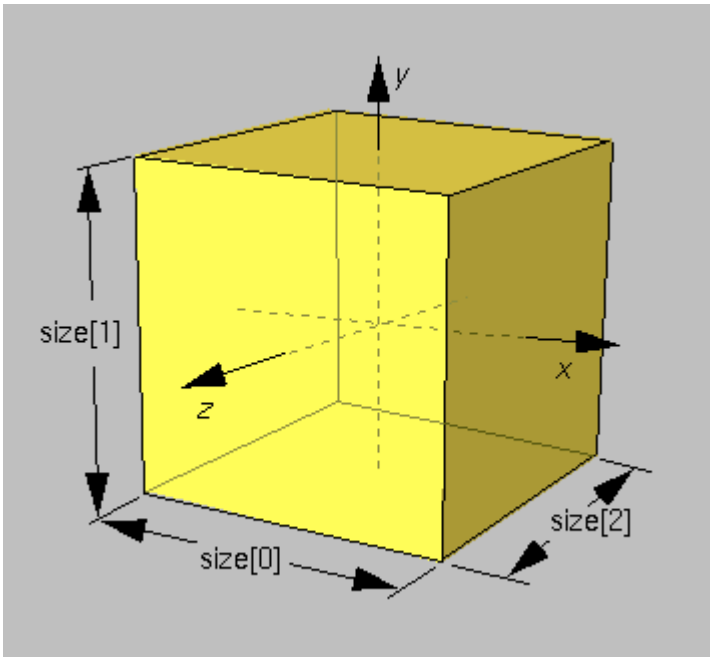
Shape : Spécification de la géométrie

La géométrie d'une forme est construite à partir des noeuds *geometry* :

```
Box       { . . . }
Cone      { . . . }
Cylinder  { . . . }
Sphere    { . . . }
Text      { . . . }
```

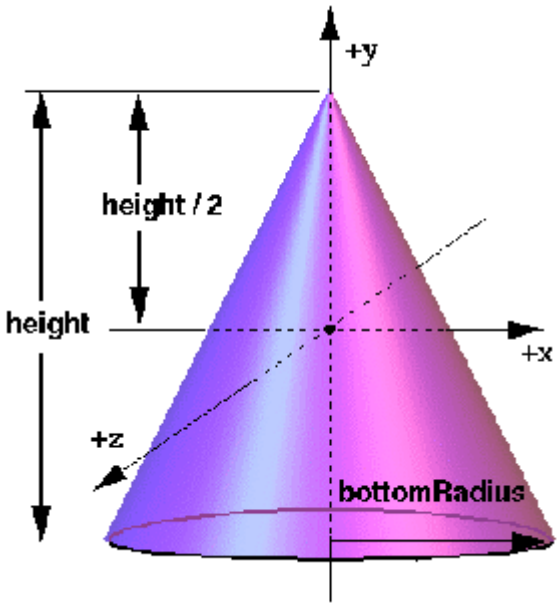
Les champs des noeuds *geometry* spécifient les dimensions de la forme
Ces dimensions sont généralement en mètre

Un noeud **Box** définit un cube centré en (0 0 0) et parallèle aux axes :
size : ***SFVect3f*** (Longueur, Hauteur, Profondeur)



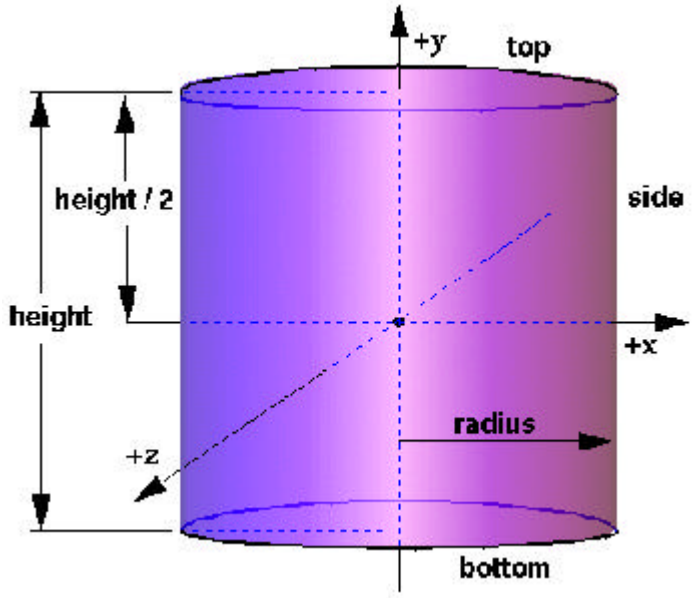

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Box {
      size 2.0 2.0 2.0
    }
}
```

Un noeud **Cone** définit un cône orienté selon l'axe Y et centré en (0 0 0):
height et **bottomRadius** : ***SFFloat*** (hauteur et rayon de base)
bottom et **side** : ***SFBool*** (fermeture du cone)



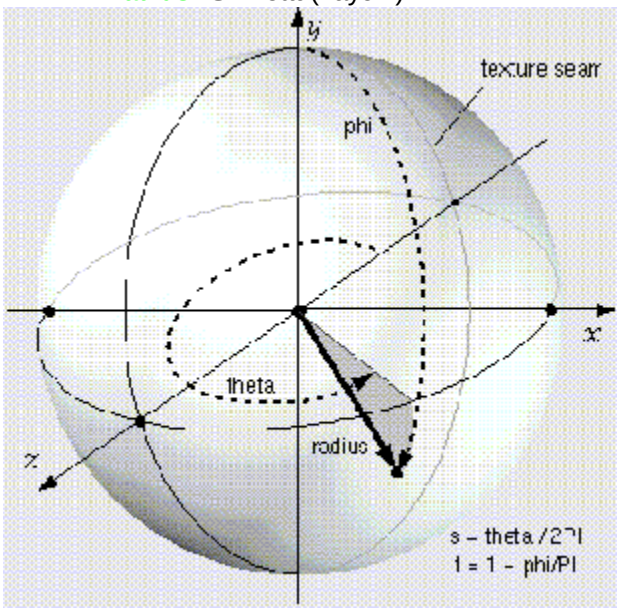

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Cone {
      height 2.0
      bottomRadius 1.0
      bottom TRUE
      side TRUE
    }
}
```

Un noeud **Cylinder** définit un cylindre orienté selon l'axe Y et centré en (0 0 0):
height et **radius** : ***SFFloat*** (hauteur et rayon)
bottom, **top** et **side** : ***SFBool*** (fermeture du cylindre)




```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Cylinder {
      height 2.0
      radius 1.0
      bottom TRUE
      top TRUE
      side TRUE
    }
}
```

Un noeud **Sphere** définit une sphère centrée en (0 0 0):
radius : ***SFFloat*** (rayon)




```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Sphere {
      radius 1.0
    }
}
```

Un noeud **Text** définit un texte dans le plan (XOY) centré en (0 0 0):
string : ***MFString*** (texte à écrire)
fontStyle : ***SFNode*** (Définition du style du texte)
length : ***MFFloat*** (Taille de chaque élément de texte)
maxExtent : ***SFFloat*** (Taille maximale d'un élément de texte)


```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
      string [ "Text", "Shape" ]
      fontStyle FontStyle {
        style "BOLD"
      }
    }
}
```

Un noeud **FontStyle** définit la fonte et le style d'un texte.

family : ***MFString*** - **SERIF**, **SANS** ou **TYPEWRITER**

style : ***SFString*** - **BOLD**, **ITALIC**, **BOLDITALIC** ou **PLAIN**


```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
          family  "SERIF"
          style   "BOLD"
        }
    }
}
```

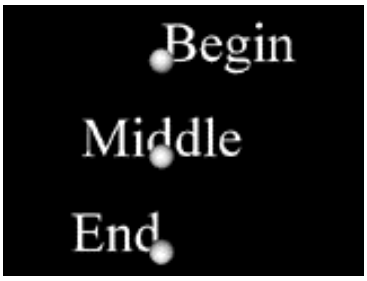
Un noeud **FontStyle** définit la fonte et le style d'un texte.
size : ***SFFloat*** (hauteur des caractères)
spaccing : ***SFFloat*** (Espacement ligne et colonnes)


```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
          size    1.0
          spacing 1.0
        }
```

```
        }
    }
```

Un noeud **FontStyle** définit la fonte et le style d'un texte.

justify : ***MFString*** (justification) - **FIRST**, **BEGIN**, **MIDDLE**, ou **END**

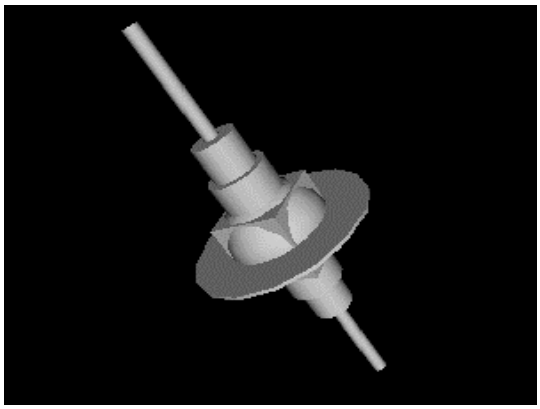



```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
          justify "BEGIN"
        }
    }
}
```

Un noeud **FontStyle** définit la fonte et le style d'un texte.
horizontal : ***SFBool*** (texte horizontal ou vertical)
leftToRight et **topToBottom** : ***SFBool*** (texte de gauche à droite ou de haut en bas)


```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Text {
        string . . .
        fontStyle FontStyle {
          horizontal  FALSE
          leftToRight TRUE
          topToBottom
TRUE
        }
    }
}
```

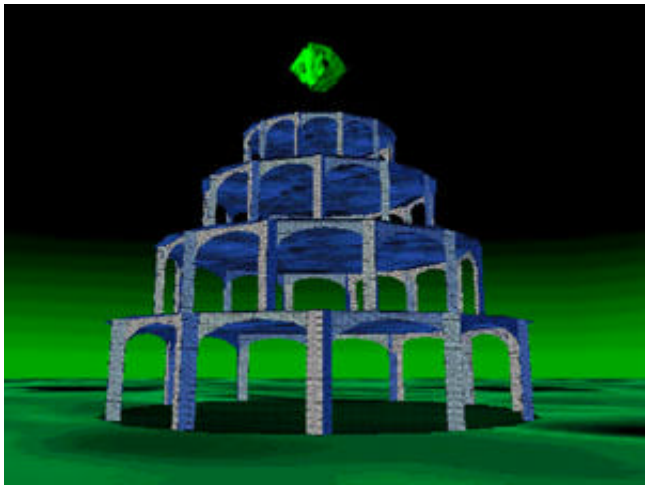
Les formes sont définies centrées sur le monde. Un fichier VRML peut contenir plusieurs formes.
Les formes se recouvrent si elles sont au même endroit.



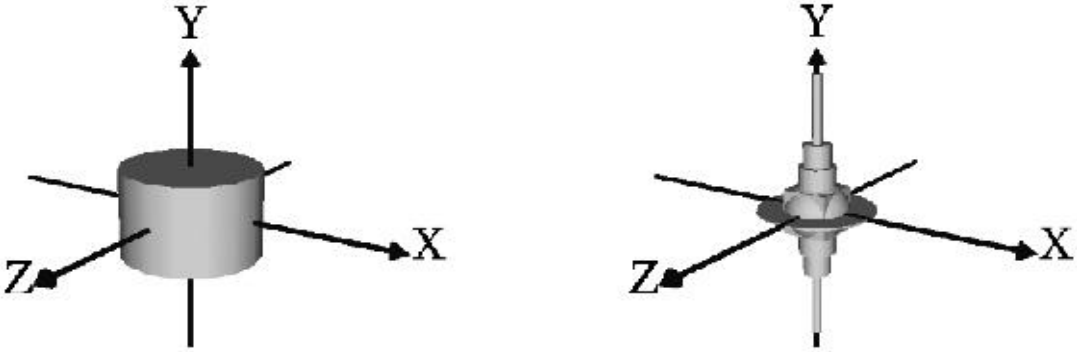
Transformations géométriques

Par défaut, les formes sont construites au centre du monde.
Une *transformation* permet de :

- Déplacer les formes.
- Orienter les formes.
- Redimensionner les formes.



Un fichier VRML définit un monde et son référentiel.
Chaque composant du monde est construit dans le référentiel du monde.
Chaque composant est construit à l'origine du référentiel.



a. Les axes XYZ et une forme simple.

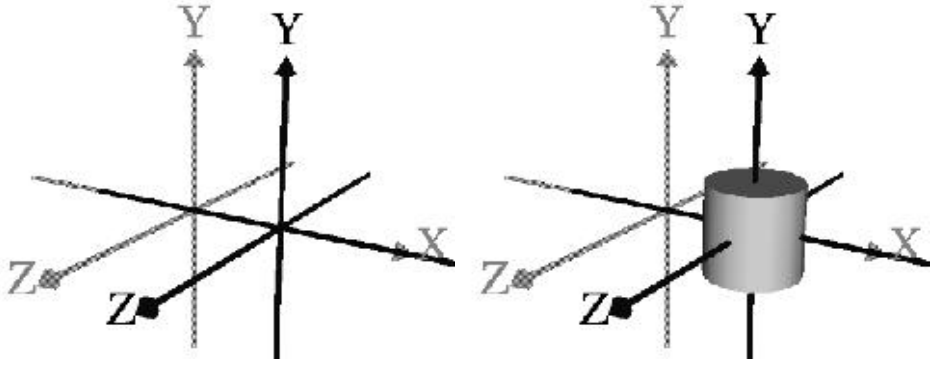
Une *transformation* définit un référentiel qui est

- Positionné
- Orienté
- Redimensionné

par rapport à son référentiel parent.
Les composants définis dans le nouveau référentiel sont positionnés, orientés et par celui-ci.

La définition d'un nouveau référentiel se fait par un noeud **Transform**.

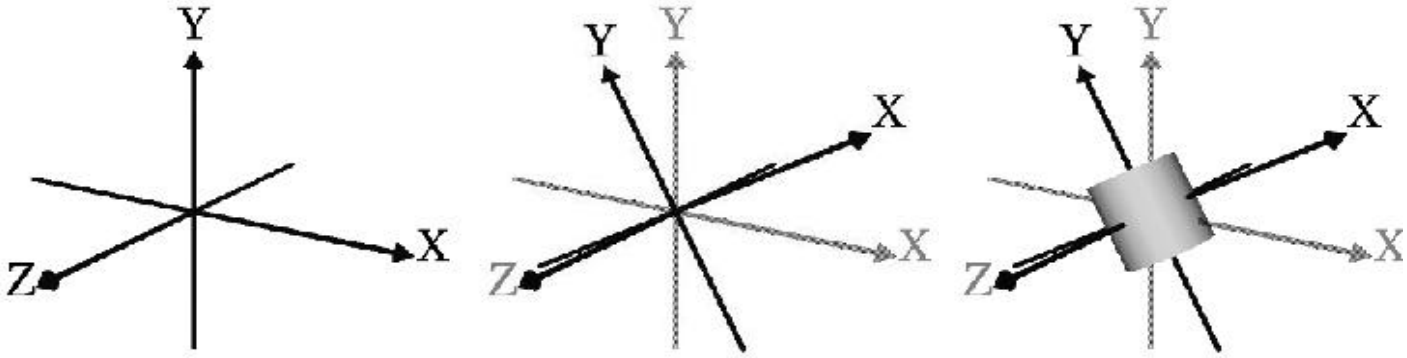
Objet translat é



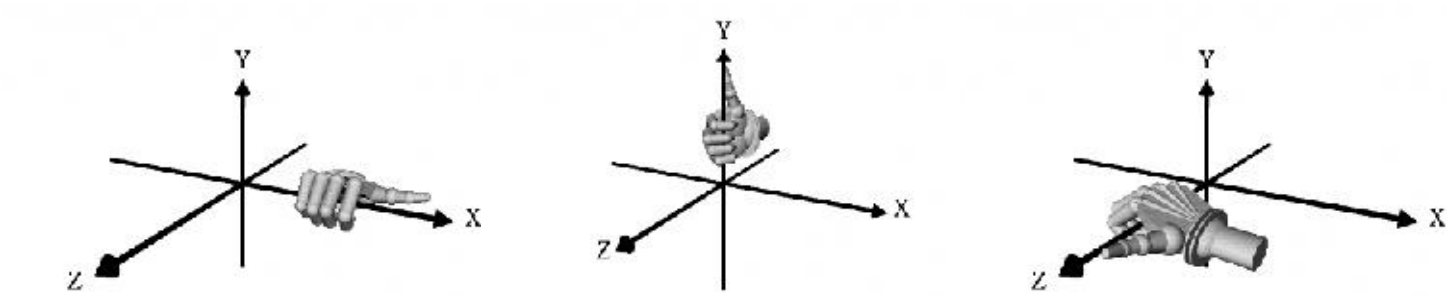
L'attribut **rotation** oriente le système de coordonnées dans R^3
L'angle de rotation est donné en *radians*.

```
Transform {
    #        X   Y   Z    Angle
    rotation 0.0 0.0 1.0  0.52
    children [ . . . ]
}
```

Référentiel du monde



Règle de la main droite pour la définition des rotations.
Ouvrir la main
Lever le pouce et l'orienter dans le sens positif de l'axe de rotation
La direction d'enroulement des doigts est une rotation positive

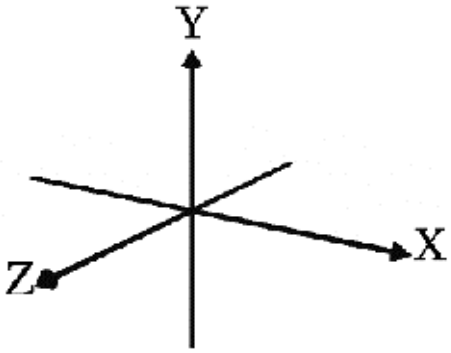


Rotation autour de l'axe X Rotation autour de l'axe Y Rotation autour de l'axe Z

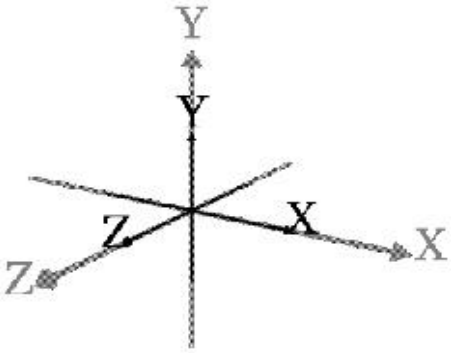
L'attribut **scale** redimensionne le système de coordonnées par un facteur d'échelle sur X, Y et Z

```
Transform {
    #     X   Y   Z
    scale 0.5 0.5 0.5
    children [ . . . ]
}
```

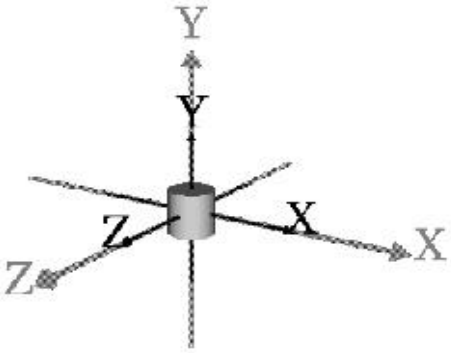
R é f é rentiel du monde



R é f é rentiel redimensionn é



Objet redimensionn é



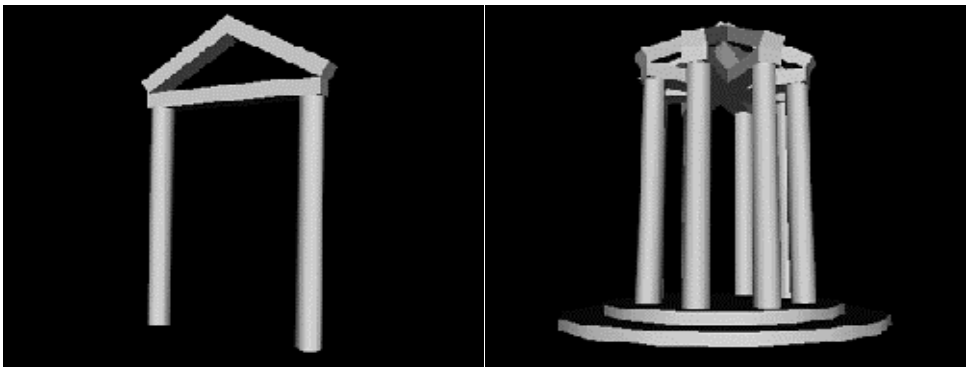
Lorsque l'on veut redimensionner, orienter et translater un référentiel, la combinaison des transformations n'est pas commutative

```
Transform {
    translation 2.0 0.0 0.0
    rotation 0.0 0.0 1.0  0.52
    scale 0.5 0.5 0.5
    children [ . . . ]
}
```

Soit un point ***P*** et un noeud de transformation avec ***C*** (center), ***SR*** (scaleOrientation), ***T*** (translation), ***R*** (rotation), and ***S*** (scale), ***P*** est transformé en ***P'*** par la combinaison (matricelle) suivante :

$\boldsymbol{P'} = T \times C \times R \times SR \times S \times SR^{-1} \times C^{-1} \times \boldsymbol{P}$

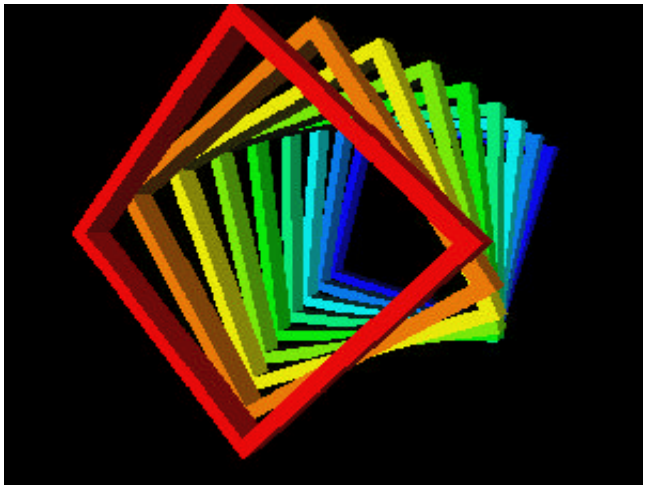
Utilisation des transformations



[arch.wrl] [arches.wrl]

Gestion des matériaux

Par défaut tous les objets ont un matériau blanc opaque.



Le contrôle de l'aspect des objets peut se faire au niveau de :

- Réflexion diffuse
- Intensité ambiante
- Emission
- Transparence
- Brillance
- Texture

L'association d'un matériau à un objet se fait dans un noeud Shape :

```
Shape {
    appearance Appearance {
               }
    geometry   . . .
}
```

Un noeud **Appearance** décrit le matériau (ou la texture) d'un objet par les attributs :

material : ***SFNode*** (Définition du matériau de l'objet).

texture : ***SFNode*** (Définition de la texture de l'objet).

textureTransform : ***SFNode*** (Définition du placement de la texture sur l'objet).

Un noeud **Material** contrôle la couleur de l'objet :

diffuseColor : ***SFColor*** (Couleur diffuse)

emissiveColor : ***SFColor*** (Couleur émise)

transparency : ***SFFloat*** (Transparence)

```
Shape {
    appearance Appearance {
        material Material {
          diffuseColor  0.8 0.8 0.8
          emissiveColor 0.0 0.0 0.0
          transparency  0.0
        }
    }
    geometry . . .
}
```

Un noeud **Material** contrôle aussi la brillance de l'objet :

specularColor : ***SFColor*** (Couleur spéculaire)

shininess : ***SFFloat*** (brillance)

ambientIntensity : ***SFFloat*** (Eclairage ambiant)

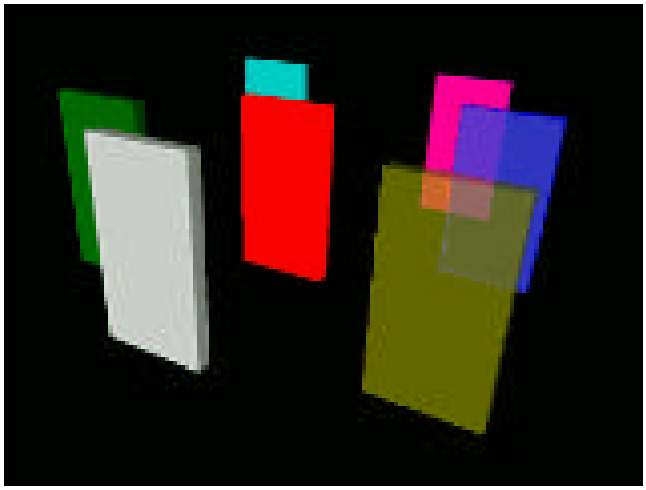
```
Shape {
    appearance Appearance {
        material Material {
          specularColor 0.71 0.70 0.56
          shininess 0.16
          ambientIntensity 0.4
        }
    }
    geometry . . .
}
```

La couleur définit:
un mélange de lumière rouge, verte et bleue
les valeurs vont de 0.0 (pas de couleur) à 1.0 (toute la couleur)

Couleur	Rouge	Vert	Bleu	Résultat
Blanc	1.0	1.0	1.0	
Rouge	1.0	0.0	0.0	
Jaune	1.0	1.0	0.0	
Cyan	0.0	1.0	1.0	
Marron	0.5	0.2	0.0	

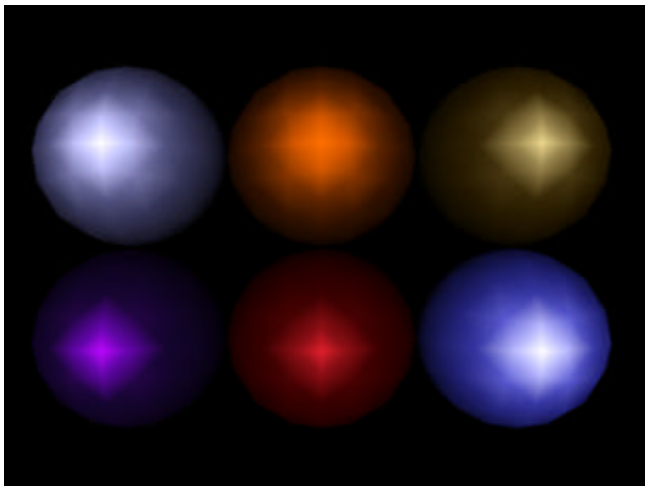
Exemple de formes colorées

```
Shape {
  appearance Appearance {
    material Material {
      diffuseColor  0.2 0.2 0.2
      emissiveColor 0.0 0.0 0.8
      transparency  0.25
    }
  }
  geometry Box {
    size 2.0 4.0 0.3
  }
}
. . .
```



Définition de matériaux brillants

Description	Intensité ambiante	Couleur diffuse	Couleur spéculaire	Brillance
Aluminum	0.30	0.30 0.30 0.50	0.70 0.70 0.80	0.10
Cuivre	0.26	0.30 0.11 0.00	0.75 0.33 0.00	0.08
Or	0.40	0.22 0.15 0.00	0.71 0.70 0.56	0.16
Bleu métalique	0.17	0.10 0.03 0.22	0.64 0.00 0.98	0.20
Rouge métalique	0.15	0.27 0.00 0.00	0.61 0.13 0.18	0.20
Plastique bleu	0.10	0.20 0.20 0.71	0.83 0.83 0.83	0.12



Objets complexes

Les objets complexes peuvent être difficiles à obtenir par mélange de formes simples :
- Terrains
- Animaux
- Plantes
- Véhicules ...

Au lieu de construire ces objets à partir de formes primitives, on les construits à partir de formes atomiques :
- Points
- Lignes
- Faces

La construction d'objet à partir de coordonnées se fait en 2 étapes :
- Placement de points par définition de coordonnées 3D.
- Connexion de ces points pour former des objets.

La définition de coordonnées est relative au référentiel courant et se fait dans un noeud **Coordinate**.

La connexion de ces coordonnées est faite par un noeud de géométrie parmi les suivants :

PointSet
IndexedLineSet
IndexedFaceSet

Un noeud **Coordinate** contient une liste de coordonnées nécessaire à la construction d'un objet :

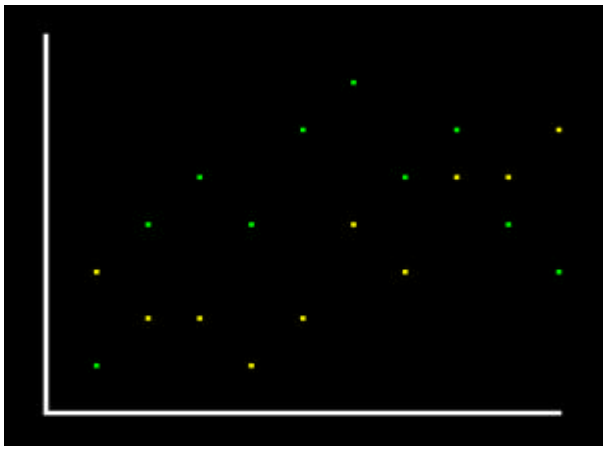
point : ***MFVect3f*** (Liste des coordonnées)
Les coordonnées (de R^3) sont séparées par une virgule

```
Coordinate {
    point [
#       X   Y   Z
        2.0 1.0 3.0,
        4.0 2.5 5.3,
        . . .
    ]
}
```

Un noeud **PointSet** définit un nuage de points dans R^3 :

coord : ***SFNode*** (Noeud de définition des coordonnées)
Un point est placé à chaque coordonnée

```
Shape {
    appearance Appearance { . . . }
    geometry PointSet {
      coord Coordinate {
        point [ . . . ]
      }
    }
}
```

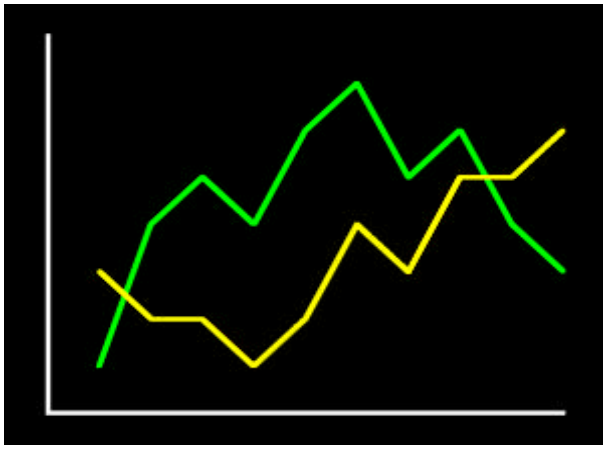


Un noeud **IndexedLineSet** définit un ensemble de polylignes construites sur des points :

coord : ***SFNode*** (Noeud de définition des coordonnées des points)

coordIndex : ***MFInt32*** (Noeud de définition des polylignes)

```
Shape {
    appearance Appearance { . . . }
    geometry  IndexedLineSet {
      coord Coordinate {
         point [  . . .  ]
      }
      coordIndex [ . . . ]
   }
}
```



Les coordonnées d'un noeud **Coordinate** sont implicitement numérotées à partir de 0
Pour construire un ensemble de polylignes :
Construire un ensemble de coordonnées
Relier ces coordonnées entre elles en utilisant leurs index
Un segment de droite est tracé entre chaque paire de sommet
Un index de -1 indique une fin de polyligne
La polyligne n'est pas automatiquement fermée

coordIndex [1, 0, 3, 8, -1, 5, 9, 0]

`1, 0, 3, 8,`	**Trace une ligne entre 1, 0, 3 et 8**
`-1,`	**Fin de ligne, debut de la suivante**
`5, 9, 0`	**Trace une ligne entre 5, 9 et 0**

Un noeud **IndexedFaceSet** définit un objet à partir d'un ensemble de polygones :

coord : ***SFNode*** (Noeud de définition des coordonnées des points)

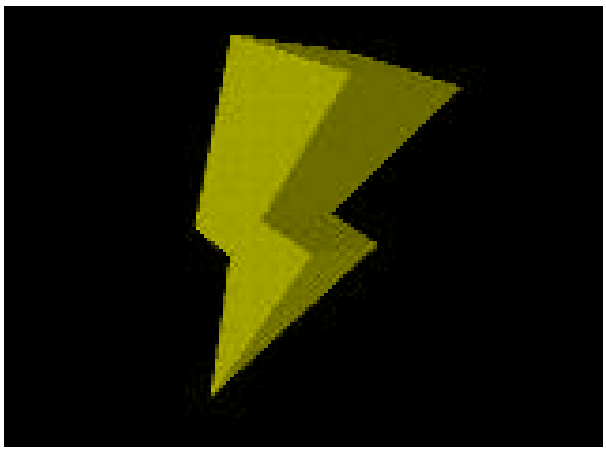
coordIndex : ***MFInt32*** (Noeud de définition des polygones)

solid : ***SFBool*** (Indicateur de fermeture de l'objet)

ccw : ***SFBool*** (Indicateur d'ordre des sommets des polygones)

convex : ***SFBool*** (Indicateur de convexité des polygones)

```
Shape {
    appearance Appearance { . . . }
    geometry IndexedFaceSet {
      coord Coordinate { . . . }
      coordIndex [ . . . ]
      solid TRUE
      ccw TRUE
      convex TRUE
    }
}
```



Un polygone est tracé pour chaque séquence d'index de coordonnées

l'index **-1** sépare les polygones
Les polygones sont automatiquement fermés
Un objet solide est un objet dont on ne voit *jamais* l'interieur

Si **solid** vaut TRUE, les faces arrières ne sont pas tracées.
Les sommets des faces avant d'un polygone sont ordonnés dans le sens

Si **ccw** vaut FALSE, l'ordre est inversé.
Les polygones sont supposés *convexes*

Si **convex** vaut FALSE, ils sont automatiquement subdivisés en triangle.

Un noeud **Material** définit un matériau applicable à la totalité d'un objet.

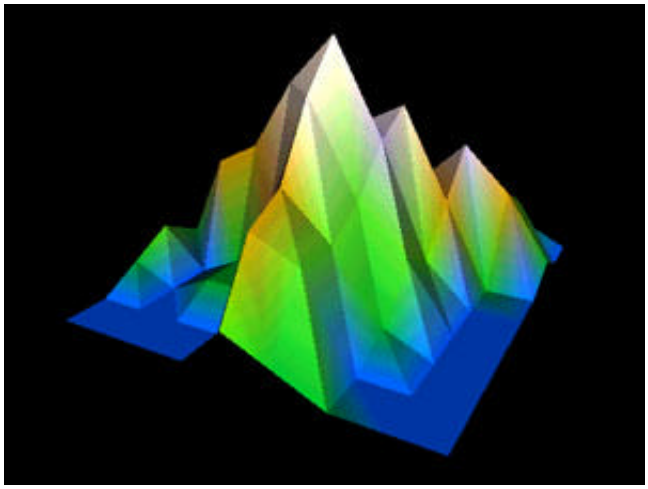
On peut fixer la couleur de partie d'un objet en utilisant un noeud **Color**.

Un noeud **Color** contient un ensemble de triplet RGB.

```
Color {
    color [ 1.0 0.0 0.0, . . . ]
}
```

Un noeud **Color** peut être utilisé comme valeur du champ **color** des noeuds de géométrie **IndexedFaceSet**, **IndexedLineSet**, **PointSet** et **ElevationGrid**.

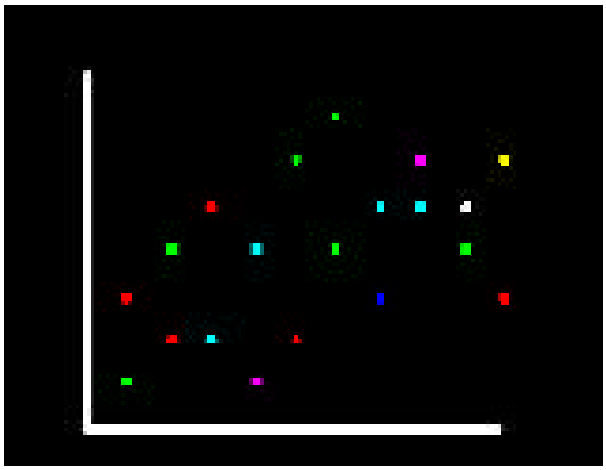
Les couleurs définies dans un noeud **Color** remplacent le matériau associé à l'objet.
Les couleurs peuvent être associées à :
Chaque point, ligne ou face d'un objet.
Chaque coordonnée d'une ligne ou d'une face d'un objet.



L'association de couleur à un noeud **PointSet** est directe :

L'attribut **color** définit les couleurs.
Les couleurs sont associées à chaque point, dans l'ordre.

```
Shape {
    appearance Appearance {
                . . . }
    geometry PointSet {
      coord Coordinate {
            . . . }
      color Color { . . . }
   }
}
```

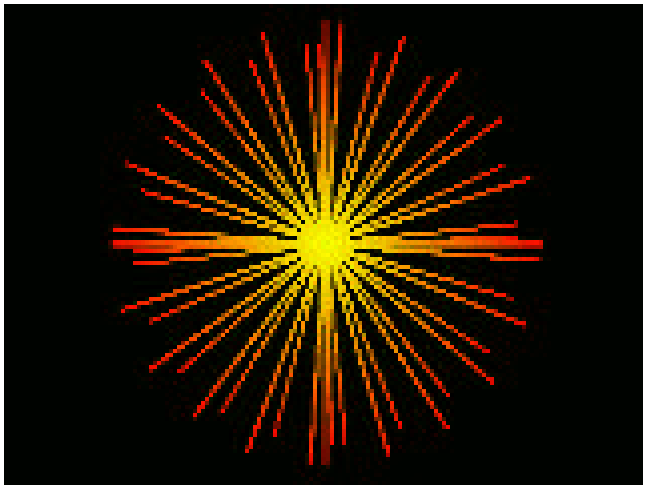


L'association de couleurs à un noeud **IndexedLineSet** est contrôlée par un tableau d'index :

L'attribut **colorIndex** définit les index de couleurs. Si cet attribut est absent, les index

L'attribut **colorPerVertex** indique si les couleurs sont attribuées aux polylignes ou aux sommets

```
Shape {
    appearance Appearance {
            . . . }
    geometry IndexedLineSet {
      coord Coordinate {
          . . . }
      coordIndex [ . . . ]
      color Color { . . . }
      colorIndex [ . . . ]
      colorPerVertex TRUE
    }
}
```

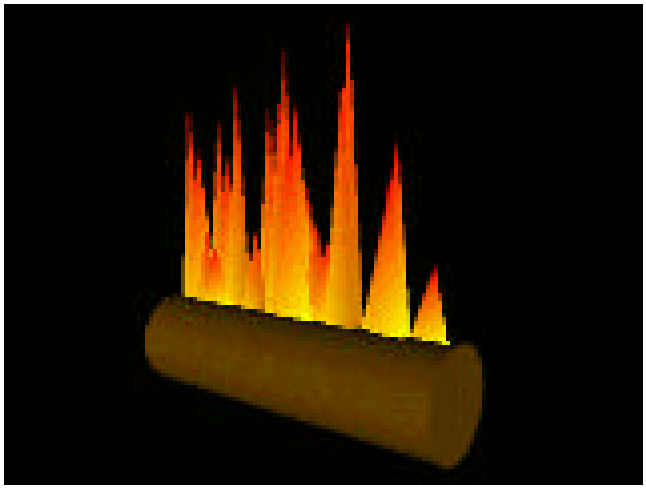


L'association de couleurs à un noeud **IndexedFaceSet** est contrôlée par un tableau d'index :

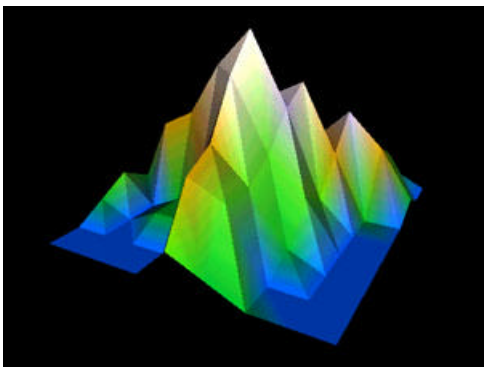
L'attribut **colorIndex** définit les index de couleurs. Si cet attribut est absent, les index

L'attribut **colorPerVertex** indique si les couleurs sont attribuées aux faces ou aux sommets

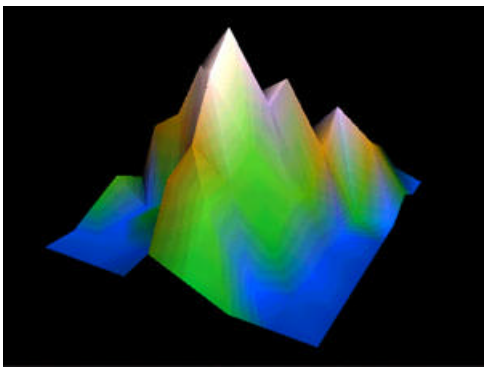
```
Shape {
    appearance Appearance {
            . . . }
    geometry IndexedFaceSet {
      coord Coordinate {
          . . . }
      coordIndex [ . . . ]
      color Color { . . . }
      colorIndex [ . . . ]
      colorPerVertex TRUE
    }
}
```



Lors du calcul de l'ombrage sur un objet, ses faces sont souvent visibles.
Pour créer un objet lisse, on peut utiliser de (très) nombreuses faces :
Cette solution est trop coûteuse pour une utilisation réelle.
On utilise plutôt un ombrage avec lissage pour créer l'impression d'un objet lisse.



Sans lissage

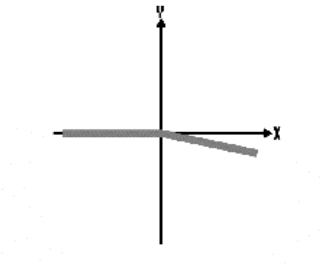


Avec lissage

On peut autoriser le lissage en utilisant l'attribut **creaseAngle** pour les noeuds :

IndexedFaceSet
ElevationGrid
Extrusion

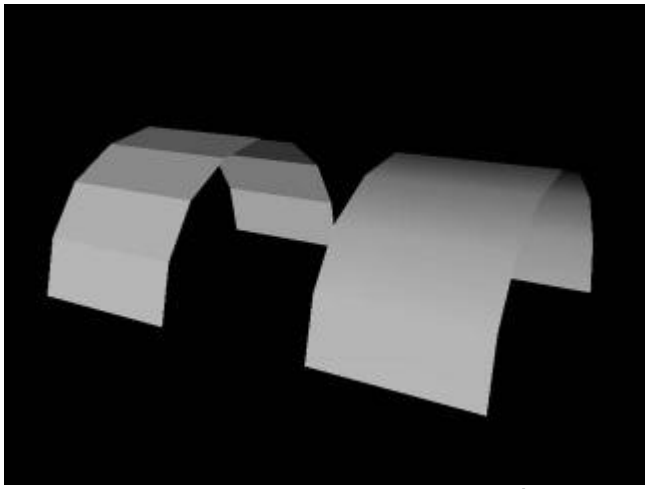
l'attribut **creaseAngle** définit un seuil de lissage entre deux faces.



Si l'angle des faces est supérieur à l'angle **creaseAngle**, pas de lissage

Si l'angle des faces est inférieur à l'angle **creaseAngle**, lissage

Exemple de lissage



A gauche, creaseAngle = 0 (Facettisé),
A droite, creaseAngle = 1.571 (Lisse)

Le contrôle de l'ombrage peut aussi se faire à l'aide d'un vecteur normal.
Le navigateur VRML calcule automatiquement les normales.

Un noeud **Normal** permet de spécifier ses propres normales.

Les normales peuvent être précisées dans un noeud **IndexedFaceSet** ou **ElevationGrid**.

Un noeud **Normal** contient la liste des vecteur normaux aux faces ou au sommets.

```
Normal {
    vector [ 0.0 1.0 0.0, . . . ]
}
```

L'association de normales à un noeud **IndexedFaceSet** est contrôlée par un tableau d'index :

L'attribut **normalIndex** définit les index des normales. Si cet attribut est absent, les index

L'attribut **normalPerVertex** indique si les normales sont attribuées aux faces ou aux sommets

```
Shape {
    appearance Appearance { . . . }
    geometry IndexedFaceSet {
      coord Coordinate { . . . }
      coordIndex [ . . . ]
      normal Normal { . . . }
      normalIndex [ . . . ]
      normalPerVertex TRUE
   }
}
```

Groupement d'objets

VRML ne définit que des formes simples :

- Cube
- Sphère
- Cône
- Cylindre

Pour combiner les formes simples en formes complexes, VRML définit les noeuds suivants :

```
Group       { . . . }
Switch      { . . . }
Transform   { . . . }
Billboard   { . . . }
Anchor      { . . . }
Inline      { . . . }
```

Un noeud **Group** définit un assemblage élémentaire de noeuds :

children : ***MFNode*** (Liste des noeuds groupés)

Chaque noeud du groupe est affiché.

```
Group {
    children [ . . . ]
}
```

Un noeud **Switch** définit un assemblage à choix multiples de noeuds :

children : ***MFNode*** (Liste des noeuds groupés)

whichChoice : ***SFInt32*** (Noeud à afficher)

Un seul noeud du groupe est affiché.
Le choix du noeud affiché se fait par l'attribut whichChoice.
Les noeuds sont numérotés à partir de 0
La valeur -1 permet de n'afficher aucun noeud

```
Switch {
    whichChoice 0
    choice [ . . . ]
}
```

Un noeud **Billboard** définit un assemblage de noeuds dans un référentiel rotatif :

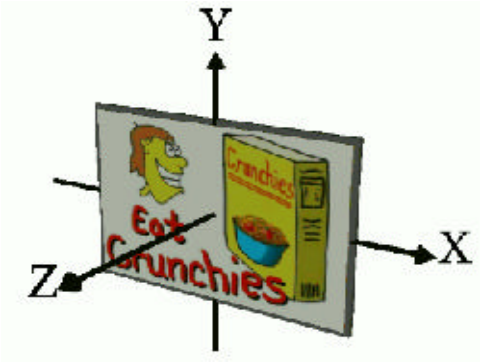
children : ***MFNode*** (Liste des noeuds groupés)

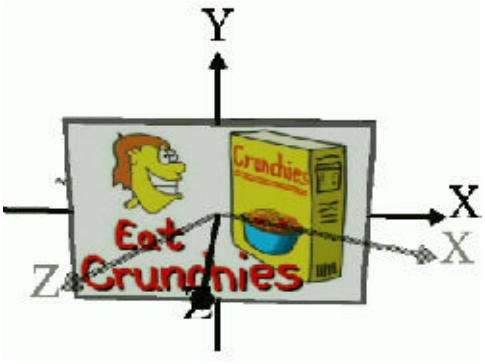
axisOfRotation : ***SFVect3f*** (axe de rotation du référentiel)

Chaque noeud du groupe est affiché.
Le référentiel fait toujours face à l'observateur.

```
Billboard {
    axisOfRotation 0.0 1.0 0.0
    children [ . . . ]
}
```

L'axe de rotation définit la contrainte à appliquer sur le référentiel du **Billboard**:
Il est similaire à l'attribut rotation d'un noeud **Transform** mais sans angle (calculé automatiquement)





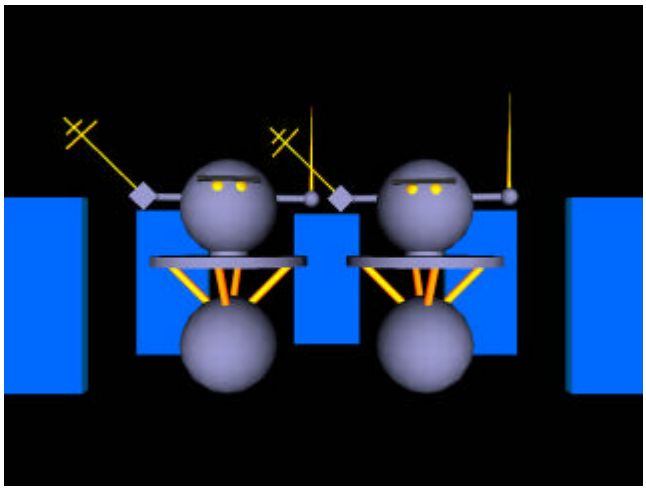
Le Billboard tourne automatiquement

L'axe de rotation limite l'orientation du **Billboard** :
Un axe de rotation nul supprime toute contrainte sur la rotation du billboard :

Orientation autour de	Axe
Axe X	**1.0 0.0 0.0**
Axe Y	**0.0 1.0 0.0**
Axe Z	**0.0 0.0 1.0**
Tout axe	**0.0 0.0 0.0**

Le robot de droite est contraint sur l'axe Y
Le robot de gauche n'est pas contraint

```
Billboard {
    # Y-axis
    axisOfRotation 0.0 1.0 0.0
    children [
      Shape { . . . }
      Shape { . . . }
      Shape { . . . }
      . . .
    ]
}
```



Un noeud **Anchor** définit un groupe réagissant comme une balise hyper-texte (hyper-monde) :

children : ***MFNode*** (Liste des noeuds groupés)

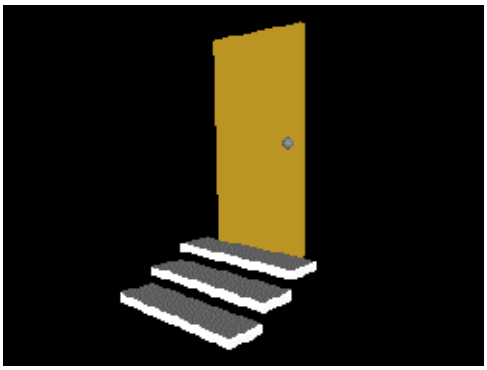
url : ***MFString*** (Liens hyper-texte)

description : ***MFString*** (Description des liens)

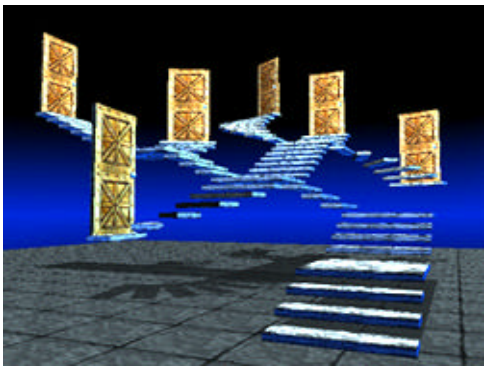
Chaque noeud du groupe est affiché.
Cliquer sur un noeud active la balise (chargement du lien).

```
Anchor {
    url "stairwy.wrl"
    description "Twisty Stairs"
    children [ . . . ]
}
```

Suivez les liens ...



Cliquer sur la porte pour aller vers ...



... Le chemin des marches

Un noeud **Inline** crée un groupe à partir du contenu d'un autre fichier VRML :

url : ***MFString*** (Nom du ficher a charger)
Chaque noeud du groupe est affiché.

```
Inline { url "table.wrl" }
. . .
Transform {
    translation -0.95 0.0 0.0
    rotation 0.0 1.0 0.0 3.14
    children [
        Inline { url "chair.wrl" }
    ]
}
```

Exemple de noeud Inline

Nommage des noeuds

Si plusieurs formes d'un monde partagent les mêmes informations, il est intéressant de ne pas les dupliquer :
Afin de diminuer la taille des fichiers VRML définissant de tels mondes il est intéressant de :
une seule fois l'information
Utiliser plusieurs fois cette information

Pour cela, les Mots clés **DEF** et **USE** du langage VRML doivent être utilisés :

DEF permet de nommer un noeud de manière à pouvoir le réutiliser:

DEF est toujours en majuscule
Tout type de noeud peut être nommé
Un nom doit être unique dans un fichier VRML

DEF est l'équivalent VRML de la définition de constantes typées en C

```
Shape {
    appearance Appearance {
        material DEF RedColor Material {
            diffuseColor 1.0 0.0 0.0
        }
    }
    geometry . . .
}
```

Le mot clé **USE** permet d'utiliser un noeud précédemment nommé:

USE est toujours en majuscule
On ne peut utiliser un noeud nommé que dans le même fichier VRML
La réutilisation d'un noeud nommé est appelée une instance
Chaque instance d'un noeud partage la **même** définition

```
Shape {
    appearance Appearance {
        material USE RedColor
    }
    geometry . . .
}
```

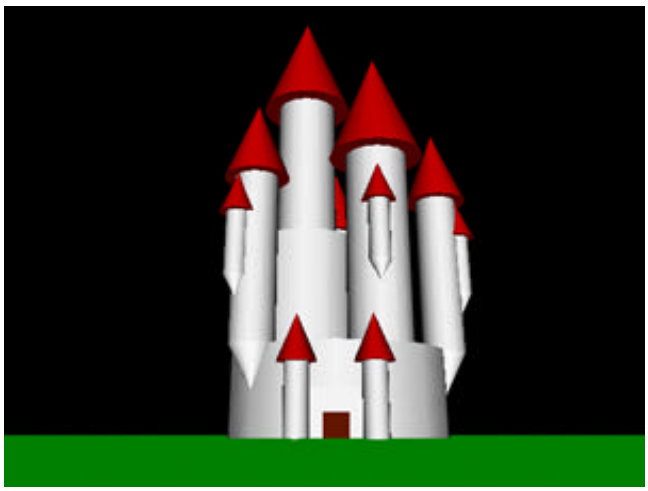
Le nommage de noeud VRML est utile afin de :
- typer les formes
- Réduire la taille des fichiers VRML
- Permettre des modifications rapide de noeuds identiques
- Augmenter la vitesse d'analyse d'un fichier VRML

Le nommage est nécessaire pour l'animation

Exemples

Des noeuds **Cylinder** définissent les tours.

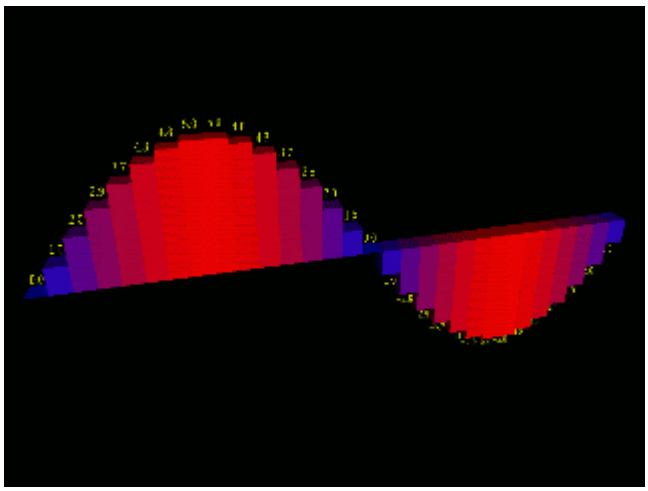
Des noeuds **Cone** définissent les toits et les bas de tours.



Des noeuds **Box** définissent les barres.

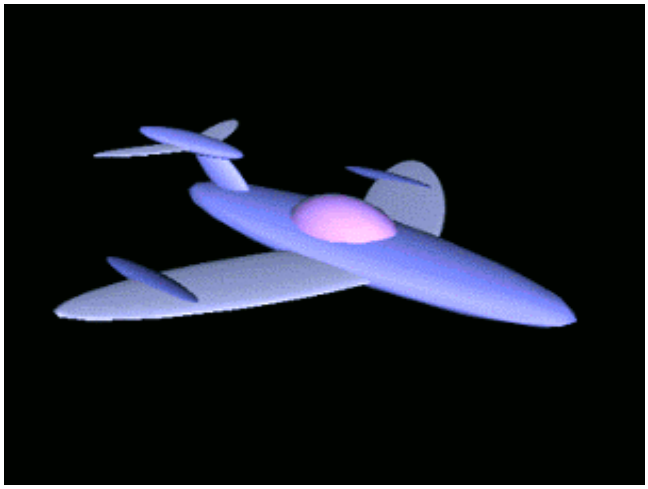
Des noeuds **Text** définissent les étiquettes.

Des noeuds **Billboard** permettent de contraindre la position des étiquettes par rapport à l'observateur.



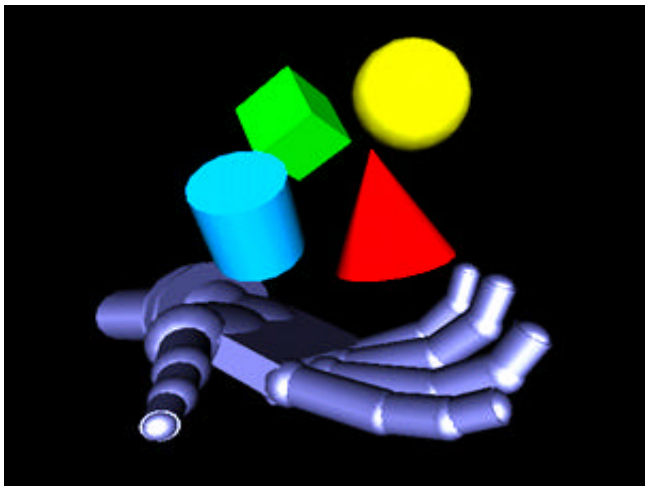
Des noeuds **Sphere** définissent les parties de l'avion.

Des noeuds **Transform** redimensionnent les sphères pour contrôler l'aspect de l'avion.

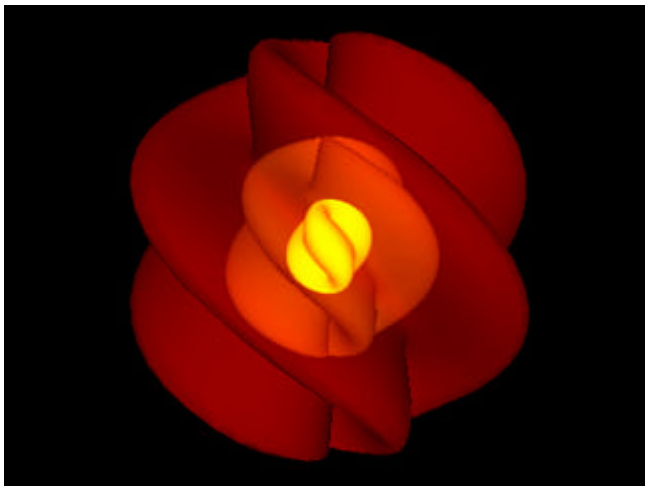


Des noeuds **Cylinder** et **Sphere** définissent la main.

Des noeuds **Transform** articulent la main.



Des noeuds **IndexedFaceSet** définissent la surface.



Eclairage, textures et environnement

Placement des caméras et contrôle de la navigation

Par défaut, l'observateur est placé en (0.0 0.0 10.0) et regarde le long des Z négatifs
La modification de la position de l'observateur se fait soit par des commandes de navigation soit par l'introduction de noeuds **Viewpoint**
Plusieurs points de vues peuvent être définis par un noeud **Viewpoint** permettant de :
Sélectionner la position initiale de l'observateur.
Définir un ensemble de points de vue dans la scène.
Nommer les caméras pour faciliter la navigation.

Un noeud **Viewpoint** définit un point de vue identifié par son nom :

position : ***SFVec3f*** (Position de la caméra)

orientation : ***SFRotation*** (Orientation de la caméra dans le référentiel courant)

fieldOfView : ***SFFloat*** (Angle d'ouverture de la caméra en radians)

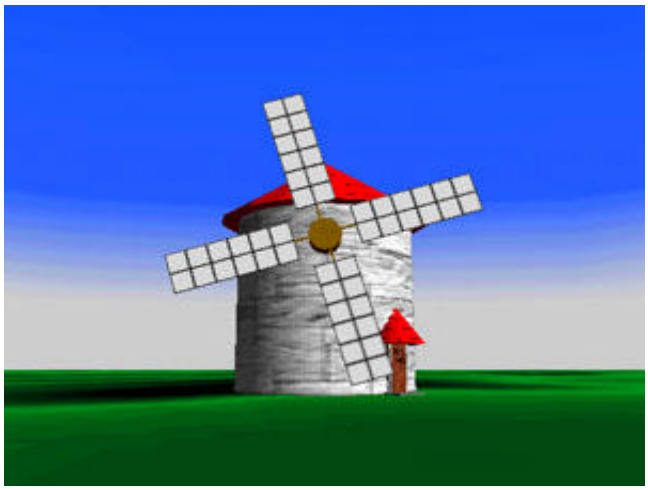
description : ***SFString*** (Nom de la caméra)

jump : ***SFBool*** (Indicateur d'animation lors du changement de caméra)

```
Viewpoint {
    position    0.0 0.0 10.0
    orientation 0.0 0.0 1.0 0.0
    fieldOfView 0.785
    jump TRUE
    description "Vue initiale"
}
```

Un noeud **Viewpoint** est affecté par les noeuds **Transform** se trouvant plus haut dans l'arbre VRML.

Le premier noeud **Viewpoint** trouvé dans le fichier VRML définit la caméra par défaut.



Chaque monde virtuel ayant sa spécificité, sa visualisation peut être différente :

- *Walk through* dans les environnements architecturaux
- *Fly through* pour les scènes aériennes
- *Examine* dans les applications CAO

Le type de navigation peut être indiqué dans le fichier VRML.
La taille et la vitesse de l'avatar de l'observateur est modifiable.

Un noeud **NavigationInfo** permet de préciser les conditions d'observation du monde :

type : ***MFString*** (type de navigation)

avatarSize : ***MFFloat*** (Taille de l'avatar)

speed : ***SFFloat*** (Vitesse de déplacement)

headlight : ***SFBool*** (Lumière frontale)

```
NavigationInfo {
    type       [ "WALK", "ANY" ]
    avatarSize [ 0.25, 1.6, 0.75 ]
    speed      1.0
    headlight  TRUE
}
```

Il y a cinq types de navigation définis par le langage VRML :

WALK : déplacement sensible à la gravité.

FLY : déplacement insensible à la gravité.

EXAMINE : Examen d'un objet en tournant autour.

NONE : déplacement géré par le monde et non par l'utilisateur.

ANY : autorise l'utilisateur à choisir son mode de navigation.

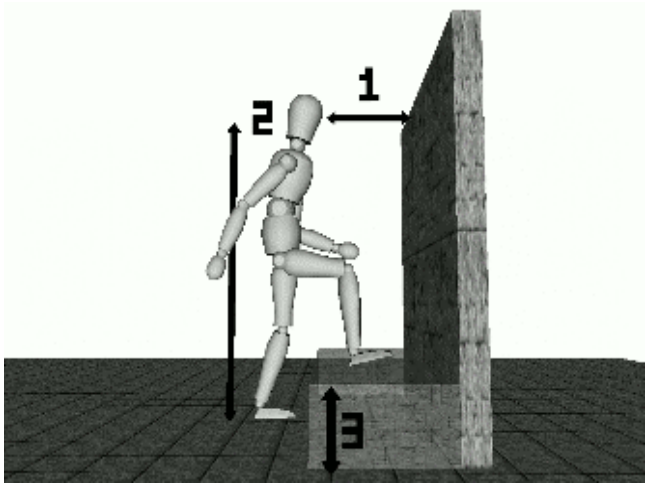
Par défaut, une source de lumière est placée à la position de l'observateur :
Elle éclaire ce qui est devant l'observateur.

noeud **NavigationInfo** .

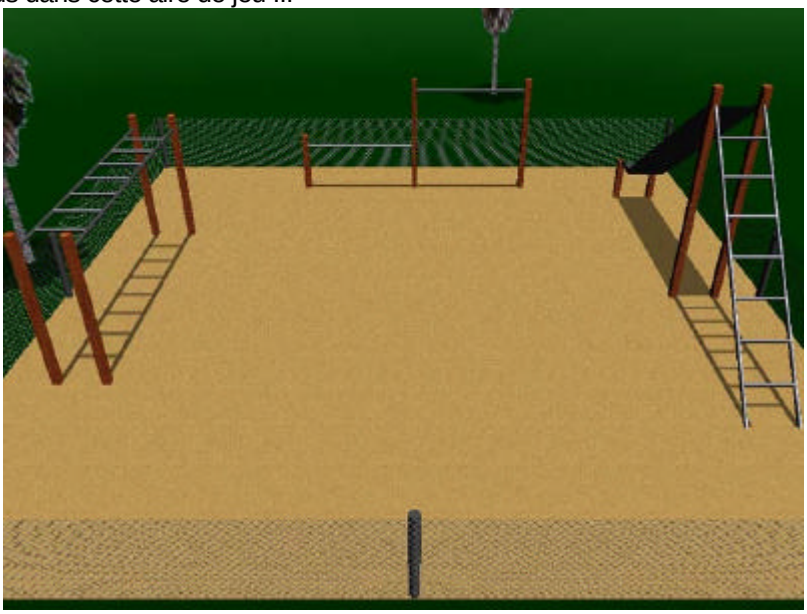
Elle peut être allumée ou éteinte par l'utilisateur.

Le contrôle de l'observateur peut être fait en paramétrant son avatar par 3 valeurs:

La première est la distance limite à un obstacle.
La seconde est la hauteur par rapport au sol ou doit être maintenue la position d'observation.
La troisième est la taille maximale d'un objet par dessus lequel peut passer l'avatar.



Déplacez vous dans cette aire de jeu ...



Gestion de l'éclairage

Par défaut, une seule source de lumière éclaire la scène : la *headlight*
Afin de rendre le monde virtuel plus réaliste, de nombreuses sources de lumière peuvent être

- pour simuler le soleil, un éclairage lointain ...
Spots lumineux - pour simuler les éclairage locaux directionnels
Lumières ponctuelles- pour simuler les éclairage locaux omnidirectionnels
Les sources de lumière peuvent être positionnée, orientées et colorées.
Les sources de lumière ne génèrent pas d'ombres portées.

Les différentes sources de lumière possèdent les attributs suivants :

on : ***SFBool*** (Source allumée ou pas)
intensity : ***SFFloat*** (Intensité de la source)
ambientIntensity : ***SFFloat*** (Intensité ambiante de la source)
color : ***SFColor*** (Couleur de la source)

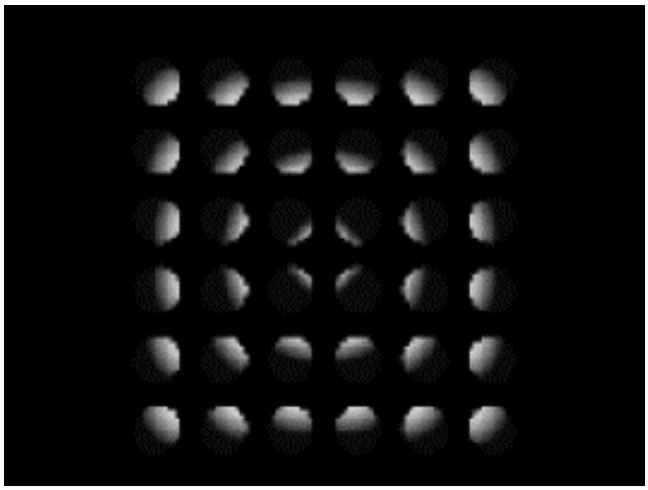
Les noeuds **PointLight** et **SpotLight** possèdent les attributs suivants :

location : ***SFVect3f*** (Position de la source dans le repère courant)
radius : ***SFFloat*** (Distance d'éclairage maximale)
attenuation : ***SFVect3f*** (Atténuation suivant la distance)

Les noeuds **DirectionalLight** et **SpotLight** possèdent l'attribut suivant :

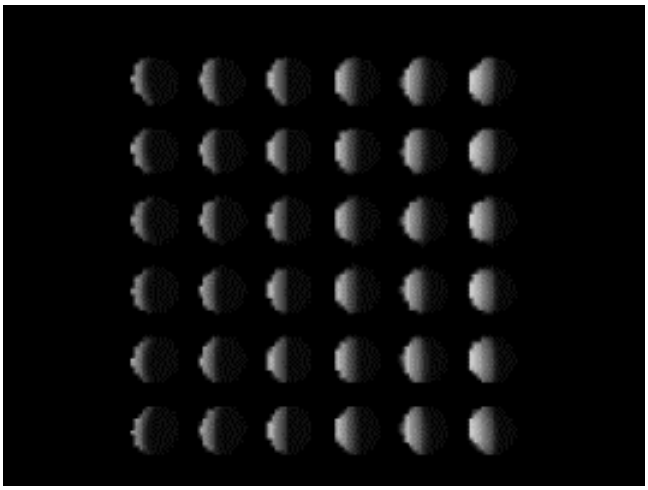
direction : ***SFVect3f*** (Direction d'illumination de la source dans le repère courant)

Un noeud **PointLight** éclaire de manière radiale à partir d'un point.



```
PointLight {
    location 0.0 0.0 0.0
    intensity 1.0
    color 1.0 1.0 1.0
}
```

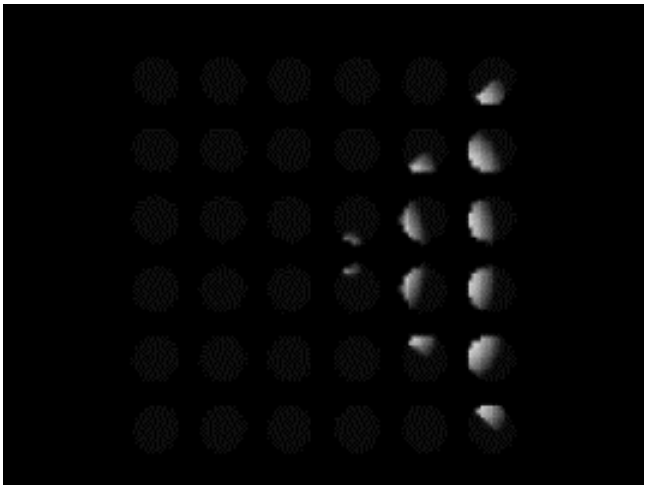
Un noeud **DirectionalLight** éclaire dans une direction donnée depuis un point situé à l'infini.



```
DirectionalLight {
    direction 1.0 0.0 0.0
    intensity 1.0
    color 1.0 1.0 1.0
}
```

Un noeud **SpotLight** éclaire à partir d'un point, dans une direction et dans un cône

cutOffAngle : ***SFFloat*** (Ouverture externe du cône)
beamWidth : ***SFFloat*** (Ouverture interne du cône)
Entre les cônes interne et externe, l'intensité diminue



```
SpotLight {
    location  0.0 0.0 0.0
    direction 1.0 0.0 0.0
    intensity 1.0
    color 1.0 1.0 1.0
    cutOffAngle 0.785
}
```

Le modèle d'éclairage devant être implémenté par un navigateur VRML est dérivé du modèle de Phong.
Une implémentation idéale d'un navigateur VRML doit évaluer l'équation suivante en chaque point d'une surface :

$$I_{rgb} = (1 - f_0) \times I_{Frgb} + f_0 \times (O_{Ergb} + SUM(attenuation_i \times spot_i \times I_{Lrgb} \times (ambient_i + diffuse_i + specular_i)))$$

Pour plus de détail sur le modèle d'éclairage à utiliser, voyez ici

Placement et contrôle des textures

Afin d'augmenter le réalisme des mondes VRML, plusieurs possibilités existent :

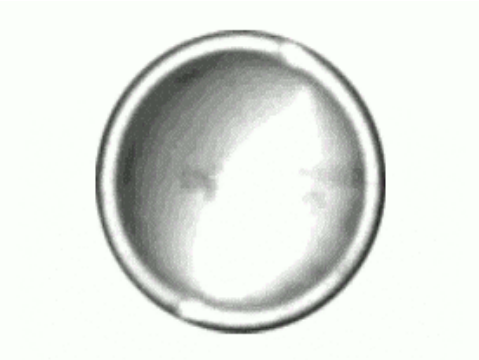
- Modélisation de tous les détails par des faces colorée
- Utilisation de photo du monde réel dans le monde virtuel

La modélisation de tous les détails par des faces colorée n'est pas envisageable :

- Le fichier VRML serait trop volumineux.
- Le temps de tracé serait trop important

L'utilisation de photo du monde réel comme matériau des objets du monde virtuel permet de:

- Modéliser toute apparence complexe de manière simple
- Accélérer les temps de rendu et d'affichage



Très peu de navigateurs gèrent la vidéo.

L'association d'une texture à un objet se fait par la définition de l'attribut **appearance** d'un noeud **Shape** :

texture : ***SFNode*** (Identification de la texture) .

textureTransform : ***SFNode*** (Placement de la texture) .

```
Shape {
    appearance Appearance {
      material Material { . . . }
      texture ImageTexture { . . . }
      textureTransform TextureTransform { . . . }
   }
    geometry . . .
}
```

Un noeud **ImageTexture** définit l'image à utiliser pour le plaquage :

url : ***MFString*** (URL du fichier image)

repeatS : ***SFBool*** (Bouclage de la texture sur *s*)

repeatT : ***SFBool*** (Bouclage de la texture sur *t*)

```
Shape {
    appearance Appearance {
        material Material { }
        texture ImageTexture {
         url "wood.jpg"
      }
    }
    geometry . . .
}
```

Un noeud **PixelTexture** définit l'ensemble de pixels à utiliser pour le plaquage :

image : ***SFImage*** (Définition en ligne de l'image)

repeatS : ***SFBool*** (Bouclage de la texture sur *s*)

repeatT : ***SFBool*** (Bouclage de la texture sur *t*)

```
Shape {
    appearance Appearance {
        material Material { }
        texture PixelTexture {
         image 2 1 3
            0xFFFF00 0xFF0000
      }
    }
    geometry . . .
}
```

Un noeud **MovieTexture** définit la vidéo MPEG-1 à utiliser pour le plaquage :

url : ***MFString*** (Définition de la vidéo à charger)

repeatS : ***SFBool*** (Bouclage de la texture sur *s*)

repeatT : ***SFBool*** (Bouclage de la texture sur *t*)

speed : ***SFFloat*** (Vitesse de défilement du film)

startTime : ***SFTime*** (Temps de départ)

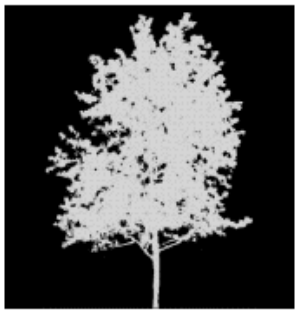
stopTime : ***SFTime*** (Temps de fin)

loop : ***SFBool*** (Jouer le film en boucle)

```
Shape {
    appearance Appearance {
        material Material { }
        texture MovieTexture {
         url "movie.mpg"
         loop TRUE
```

Arbres, nuage, feu ...

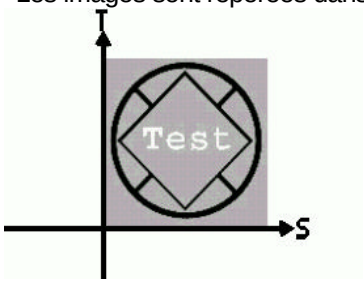
Partie couleur d'une texture d'arbre.



Partie transparente (en noir) d'une texture d'arbre

Exemple de texture transparente

Par défaut, les textures se plaquent sur l'intégralité de l'objet.
Le placement précis d'une texture sur un objet nécessite la définition des transformations à effectuer sur l'image et les correspondances entre les cordonnées de l'objet et les coordonnées dans l'image.
Les images sont repérées dans le *système de coordonnées de textures*



la direction *S* est horizontale la direction *T* est verticale (0,0) en bas à gauche (1,1) en haut à droite

Un noeud **TextureCoordinate** définit un ensemble de coordonnées dans l'espace texture.

```
TextureCoordinate {
    point [ 0.2 0.2, 0.8 0.2, . . . ]
}
```

Ce noeud est utilisé comme valeur de l'attribut **texCoord** des noeuds **IndexedFaceSet** et **ElevationGrid**.
L'association entre les sommets de l'objet et ses coordonnées de texture se fait de la même manière que pour ses coordonnées cartesienne, par la définition d'un index.

Utilisation du champ **texCoordIndex** pour le noeud **IndexedFaceSet**.
Correspondance automatique pour le noeud **ElevationGrid**.

Le noeud **TextureTransform** permet de définir les transformations a effectuer sur la texture avant le plaquage :

translation permet de déplacer le repère texture

rotation permet d'orienter le repère texture

scale permet de redimensionner le repère texture

```
Shape {
    appearance Appearance {
        material Material { . . . }
        texture ImageTexture { . . . }
        textureTransform TextureTransform {
         translation 0.0 0.0
```

```
            rotation    0.0
            scale       1.0 1.0
        }
    }
}
```

Le noeud **TextureTransform** permet de définir les transformations a effectuer sur la texture avant le plaquage :

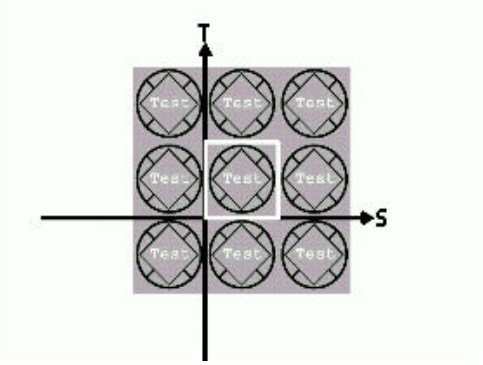
translation permet de déplacer le repère texture

rotation permet d'orienter le repère texture

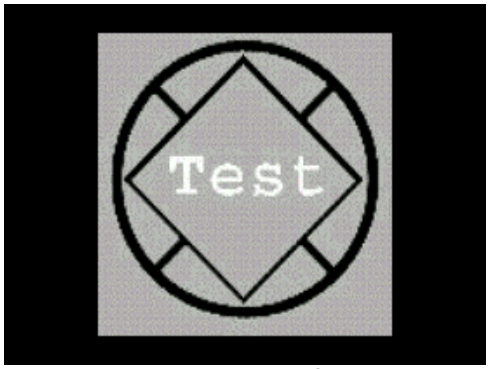
scale permet de redimensionner le repère texture

```
Shape {
    appearance Appearance {
        material Material { . . . }
        texture ImageTexture { . . . }
        textureTransform TextureTransform {
            translation 0.0 0.0
            rotation    0.0
            scale       1.0 1.0
        }
    }
}
```

Exemple de texture sans transformation

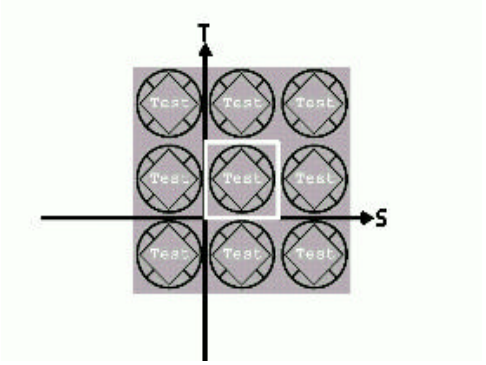


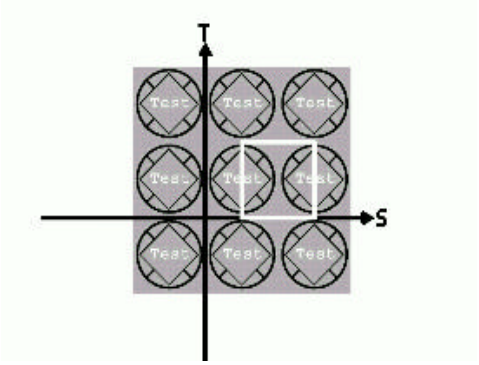
Texture dans l'espace texture



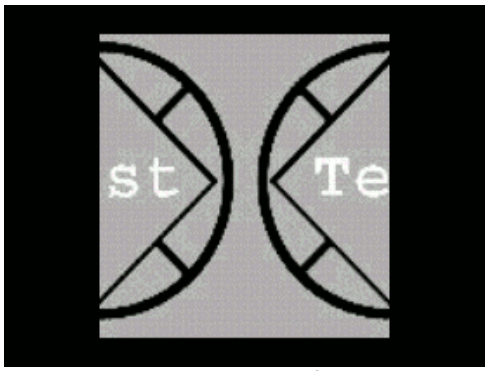
Texture plaquée

Exemple de texture avec translation



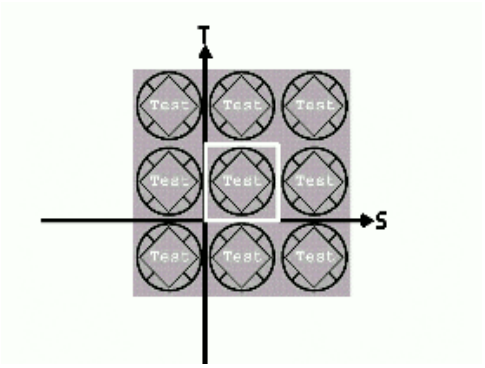


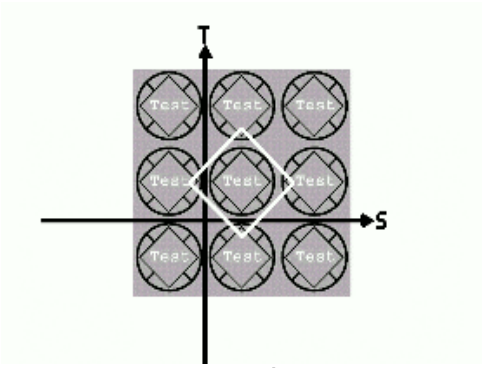
Texture dans l'espace texture



Texture plaquée

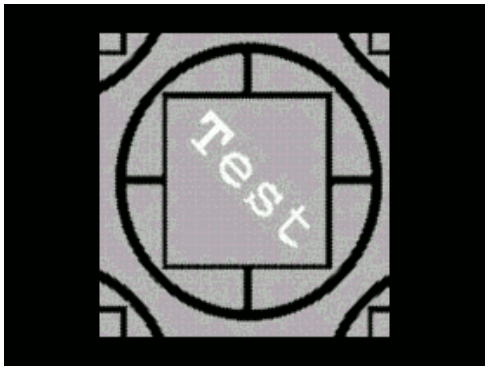
Exemple de texture avec rotation



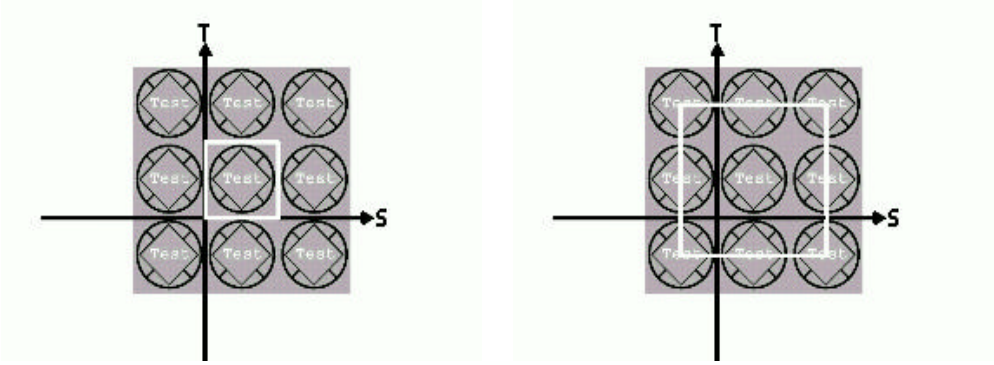


Texture dans l'espace texture

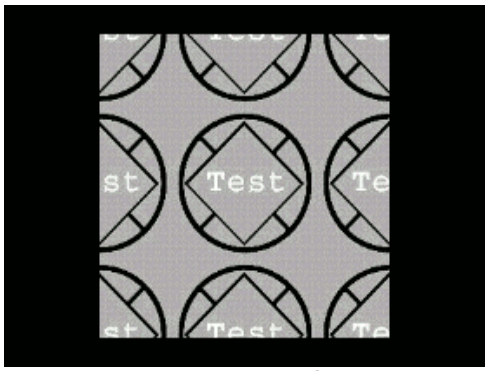
Rotation du repère texture



Exemple de texture avec mise à l'échelle



Texture dans l'espace texture



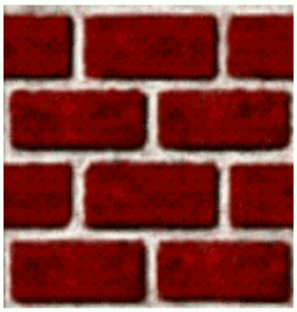
Texture plaquée

Exemple de définition de coordonnées de texture

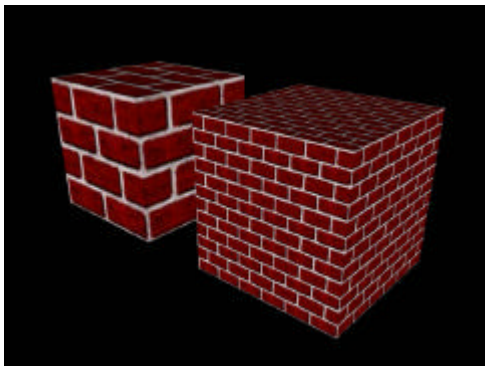
Image du biscuit

Biscuit coupé ...

Exemple de définition de coordonnées de texture



Texture initiale



Les objets du monde virtuel forment l'avant plan.
La définition d'un environnement autre que le vide intersidéral passe par l'utilisation de noeuds **Background**
L'environnement permet de définir :
La couleur du ciel et du sol.
Des images panoramique pour les décors de montagnes, ville ...
L'utilisation d'imposteurs pour l'environnement est plus efficace que leurs définition géométrique.

La définition d'un environnement entraîne la création de trois formes spéciales :
Une sphére pour le ciel.
Une hémi-sphére pour le sol à l'intérieur de la
Un boite panoramique pour l'environnement à l'intérieur de
La sphère céleste et la demi-sphère terrestre sont colorée avec un gradient.
La boite panoramique est texturée avec six images.
L'observateur peut regarder en l'air, en bas, à droite ou à gauche pour voir un autre aspect de l'environnement.
L'observateur ne peut pas s'approcher de l'environnement.

Un noeud **Background** définit l'environnement :

skyColor et **skyAngle** définissent le gradient de couleur du ciel.

groundColor et **groundAngle** définissent le gradient de couleur du sol.

```
Background {
    skyColor    [ 0.1 0.1 0.0, . . . ]
    skyAngle    [ 1.309, 1.571 ]
    groundColor [ 0.0 0.2 0.7, . . . ]
    groundAngle [ 1.309, 1.571 ]
}
```

Gestion du gradient pour le ciel :

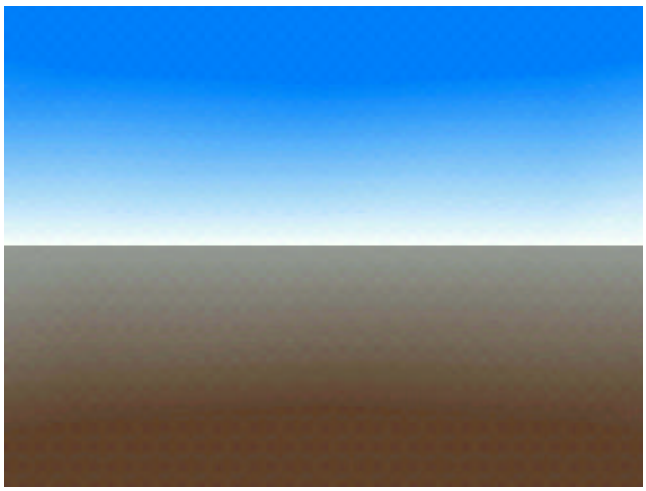
La première couleur de **skyColor** définit la couleur au pôle nord.

Les suivantes définissent les couleurs aux angles donnés dans **skyAngle**.
La valeur maximale pour les angles est 3,14159265358979323846264338327 95 radians
La dernière couleur donnée est utilisée jusqu'au pôle sud.
Gestion du gradient pour le sol :

La première couleur de **groundColor** définit la couleur au pôle sud.

Les suivantes définissent les couleurs aux angles donnés dans **groundAngle**.
La valeur maximale pour les angles est PI / 2.0 radians
Après la dernière couleur, le reste de l'hémi-sphère est transparent.

```
Background {
    skyColor [
        0.0 0.2 0.7,
        0.0 0.5 1.0,
        1.0 1.0 1.0
    ]
    skyAngle [ 1.309, 1.571 ]
    groundColor [
        0.1 0.10 0.0,
        0.4 0.25 0.2,
        0.6 0.60 0.6,
    ]
    groundAngle [ 1.309, 1.571 ]
}
```

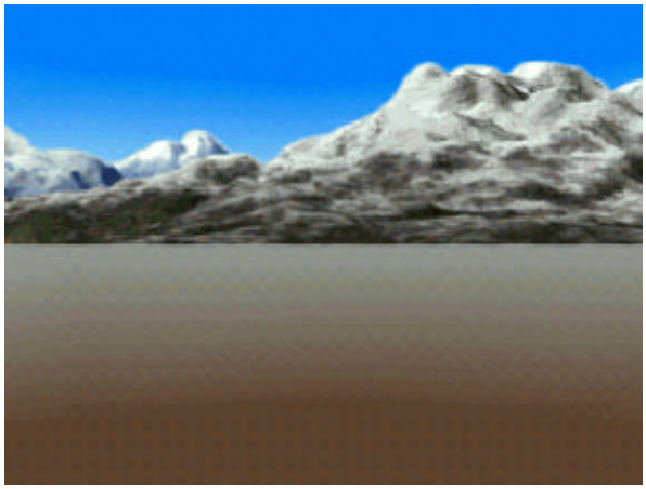


Un noeud **Background** définit aussi le panorama :

frontUrl, **backUrl**, etc... définissent les images constituant le panorama.

```
Background {
    . . .
    frontUrl  "mountns.png"
    backUrl   "mountns.png"
    leftUrl   "mountns.png"
    rightUrl  "mountns.png"
    topUrl    "clouds.png"
    bottomUrl "ground.png"
}
```

```
Background {
    skyColor [
        0.0 0.2 0.7,
        0.0 0.5 1.0,
        1.0 1.0 1.0
    ]
    skyAngle [ 1.309, 1.571 ]
    groundColor [
        0.1 0.10 0.0,
        0.4 0.25 0.2,
        0.6 0.60 0.6,
    ]
    groundAngle [ 1.309, 1.571 ]
    frontUrl "mountns.png"
    backUrl  "mountns.png"
    leftUrl  "mountns.png"
    rightUrl "mountns.png"
    # no top or bottom images
}
```



Animation et sensor

Introduction à l'animation

Des noeuds tels que les **Billboard** ou **Anchor** possède leurs propres comportement.
On peut ajouter un comportement à toute forme afin de l'animer.
Pour cela, nous avons besoin d'un mécanisme de séquencement, le temps, et de réponse à des événements pour permettre l'interaction entre objets.
noeuds peuvent appartenir à un circuit d'animation :
Les noeuds peuvent envoyer et recevoir des événements.
Des routes d'animation connectent les noeuds.
Un événement est un message envoyé entre noeuds contenant :
Une information numérique. (une translation, une rotation ...)
Une marque temporelle indiquant la date à laquelle l'événement à été envoyé.

Pour construire un circuit d'animation, il faut :
Un noeud qui envoie un événement :
Le noeud doit être nommé par DEF.
Un noeud qui recoie un événement :
Le noeud doit être nommé par DEF.
Une route les reliant :
Pour faire pivoter une forme :

Connecter un noeud envoyant un événement *rotation* au champ **rotation** d'un noeud **Transform**.
Pour faire clignoter une forme :

Connecter un noeud envoyant un événement *color* au champ **diffuseColor** d'un noeud **Material**.

Chaque noeud possède des champs, des entrées et des sorties :
des champs pour stocker l'information :
field.
des entrées :
eventIn.
des sorties :
eventOut.
Un attribut *exposedField* représente les trois à la fois.

Un noeud **Transform** possède les entrées suivantes :

set_translation

set_rotation

set_scale

Un noeud **Material** possède les entrées suivantes :

set_diffuseColor

set_emissiveColor

set_transparency

Un noeud **OrientationInterpolator** possède la sortie suivante :

value_changed pour envoyer des valeurs de rotation.

Un noeud **TimeSensor** possède la sortie suivante :

time pour envoyer des valeurs de temps.

L'instruction **ROUTE** relie deux noeuds par :
Le nom de l'émetteur et de sa sortie.
Le nom du récepteur et de son entrée.

```
ROUTE Emetteur.value_changed
   TO Recepteur.set_rotation
```

ROUTE et **TO** doivent être en majuscules.

La plupart des noeuds possèdent des champs de type *exposedField*
Si le champ *exposedField* se nomme xxx alors :

set_xxx est une entrée sur le noeud.

xxx_changed est une sortie du le noeud.

Le préfixe **set_** et le suffixe **_changed** sont optionnels.

```
DEF Touch TouchSensor { }
DEF Timer1 TimeSensor { . . . }
DEF Rot1 OrientationInterpolator { . . . }
DEF Frame1 Transform {
    children [
        Shape { . . . }
    ]
}

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

Contrôle du temps

Le contôle du temps se fait par l'utilisation d'un noeud **TimeSensor** fonctionnant comme un chronomètre :
on peut démarrer et arrêter le chrono.
Le **TimeSensor** génère des valeurs de temps absolue et fractionelles :
Les valeurs absolue sont des valeurs entière en secondes depuis le 1er janvier 1970 à 00h00.
Utile pour programmer des événements à des dates précises
Les valeurs fractionnaires sont des valeurs réelles entre 0.0 et 1.0.
Lorsque le timer démarre, il émet la valeur 0.0
A la fin d'un cycle il émet la valeur 1.0

Le nombre de valeurs émises est controlé par l'attribut **cycleInterval**
Le **TimeSensor** peu boucler indéfiniment ou faire uniquement un cycle :

Un noeud **TimeSensor** génère des événements temporels :

startTime et **stopTime** : quand démarrer et s'arrêter

cycleInterval : longueur du cycle

loop : exécution en boucle ou non.

```
TimeSensor {
    cycleInterval 1.0
    loop FALSE
    startTime 0.0
    stopTime 0.0
}
```

Pour créer un timer infini :

loop TRUE

stopTime <= **startTime**

Pour créer un timer fini :

loop TRUE
stopTime > **startTime**
Pour créer un timer d'un cycle :
loop FALSE
stopTime <= **startTime**

Exemples