Une introduction à...

Didier Verna, *Nadine Richard* - *Avril 98*

(Voir aussi Une introduction à VRML 1.0)

- Note: cette page a été écrite en Avril 1998 pour servir de support de cours aux élèves de l'ENST et du mastère Multimédia de l'école des Beaux Arts. Il n'est pas prévu d'y apporter de quelconques modifications, mise à part la maintenance des liens.
- Le but de ce document est de fournir une premier contact interactif avec VRML 97. À ce titre, il ne constitue pas un manuel de référence du langage (qui serait d'ailleurs inutile puisque la spécification complète est publique), mais il amène progressivement les notions importantes en même temps que les éléments du langage à connaître en priorité.
- Vous pourrez faire vos premiers pas et tester directement le résultat des petits exercices grâce aux formulaires présents un peu partout. Vous devez disposer d'un logiciel de visualisation 3d installé sur votre système. Reportez-vous à la section Installation pour plus d'information.

Sommaire

Installation.

Obtenir un viewer 3d:
Si vous ne disposez pas déjà d'un logiciel de visualisation 3D, le mieux est sans doute de faire un tour au Web3D Repository, qui contient une liste de tels logiciels ainsi que des platformes sur lesquelles ils tournent.

Utiliser les formulaires:

Pour pouvoir utiliser les formulaires de ce document et visualiser directement le résultat de vos exercices, vous devez préciser à votre navigateur quel «helper» utiliser pour les données VRML97 reçues. Pour éviter d'entrer en conflit avec un helper déja existant sur votre site, et dans la mesure où certains logiciels ne supportent pas simultanément VRML 1 et 2, j'ai choisi d'utiliser un type MIME particulier pour ce TP. Ce type est `x-tp/x-vrml97`. Donc, utilisez le menu preferences->applications de votre navigateur pour ajouter votre helper à la liste.

Introduction.

- **VRML (Virtual Reality Modeling Langage)**
  est un langage de modélisation de scènes 3d principalement destiné à être exploité sur le Web. 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 viewer 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.
- **Conventions géométriques:**
  Par convention, les repères spatiaux sont orthonormés directs, et l'utilisateur fait face au plan (X,Y). Donc par defaut, 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'.
  - Tout fichier VRML 97 doit commencer par la ligne suivante qui constitue le header standard:

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

  - Vous pouvez inclure des lignes de commentaire en commençant la ligne par un '#'.
  - Un fichier VRML 97 est composé d'un ensemble de «noeuds» du langage. Un noeud 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 paramêtres sont optionnels, et des valeurs par défaut sont garanties. Si vous ne voulez que les valeurs par défaut, vous êtes cependant obligés de mettre les accolades. Ainsi, pour créer un cube, vous pouvez écrire simplement ceci:

    ```
    Box { }
    ```

    Les noeuds du langage ne peuvent pas s'utiliser n'importe comment et à n'importe quelle place dans le fichier. Il existe en particulier des «noeuds de groupe» qui peuvent contenir certains autres noeuds dans leur paramètres. Nous en verrons certains.
  - Il est possible de réutiliser des noeuds précédemment décrits grâce aux directives **DEF** et **USE**. La directive *DEF* permet d'assigner un nom à noeud particulier. Par exemple, pour définir un «cube» de taille 3, vous pouvez utiliser la syntaxe suivante:

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

    Ensuite, au lieu d'être obligé de réécrire le noeud et l'ensemble de ses paramêtres à chaque fois que vous en avez besoin, vous pouvez simplement écrire:

    ```
    USE cube
    ```

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

Production des objets.

Un objet au sens VRML est compose de deux éléments principaux: une géométrie et une apparence. Ce couple défini ce que l'on appelle une "forme" en VRML. Pour pouvoir utiliser un objet, on est donc obligé de passer par le noeud correspondant, le noeud **Shape**:

Spécification de «Forme»:

- **Shape {**
  appearance app
  geometry geom
  }

Le champ *appearance*, si non nul, contient un noeud *Appearance* qui permet de définir la couleur, les textures etc. Ceci sera vu plus tard, dans la section "Apparence". Le champ *geometry* contient un noeud spécifiant une géometrie d'objet, par exemple les primitives géométriques décrites dans la section suivante.

Les objets décrits en VRML ne sont en fait réellement accessibles que par l'intermédiaire d'un noeud *Shape*. La présence directe de primitive géométriques dans le fichier est susceptible de générer des erreurs.

Primitives géométriques.

VRML 97 reconnait 4 objets de base: boîtes, cônes, cylindres et sphères.
Ces objets sont obtenus grâce à 4 noeuds du langage dont voici la syntaxe. Tous les paramètres présents dans les noeuds sont optionnels, c'est à dire que s'ils sont absents, des valeurs par défaut sont prises (ce sont les valeurs indiquées ci-dessous). Vous remarquerez que ces valeurs par default produisent des objets qui s'étendent de -1 à +1 sur les trois axes).

- **Box { size** 2 2 2 **}**
  Le paramètre **size** doit contenir les dimensions de la boîte selon les trois axes X, Y et Z. Ces dimensions sont des nombres réels positifs.
- **Cone { bottomRadius** 1 **height** 2 **side** TRUE **bottom** TRUE **}**
  Les paramètres **bottomRadius** et **height** définissent la circonférence et la hauteur du cône. Les paramêtres **side** et **bottom** sont des bouléens (valant TRUE ou FLASE) spécifiant si l'on doit tracer le fond et les côtés du cône.
- **Cylinder { radius** 1 **height** 2 **side** TRUE **bottom** TRUE **top** TRUE **}**
  De même que pour le cône, **radius** et **height** spécifient la taille du cylindre, et **top**, **bottom** et **side** determinent les parties du cylindre à tracer.
- **Sphere { radius** 1 **}**
  Le paramètre **radius** détermine le rayon de la sphère.

A l'aide du champ ci-dessous:
Visualiser le cube, puis **remplacez** le par les 3 autres primitives et visualisez le résultat. Faites ensuite la même chose en ajoutant des paramètres optionnels:

Pensez à bien terminer votre dernière ligne de code par quelques retours chariot !

```
#VRML V2.0 utf8

Shape {
  geometry Box { }
}
```

Nettoyer

Visualiser

Apparence des objets.

Le noeud Appearance dont nous avons déja parlé contient deux champs, permettant de définir un "matériau" (couleur de l'objet, réflexion de la lumière) et une texture (image à plaquer sur l'objet). Sa syntaxe est la suivante:

Spécification d'«Apparence»:

- **Appearance {**
  material mat
  texture tex
  }

Les champs *material* et *texture* contiennent des noeuds que nous allons voir immédiatement. Si le champ *material* est nul, aucune lumière n'est prise en compte. Si le champ *texture* est nul, l'objet n'est pas texturé.

Maintenant que nous avons les formes, nous allons donner une apparence aux objets. Il existe deux manières essentielles de modifier l'apparence d'un objet: définir un «matériau» (couleur de l'objet, manière dont il réfléchi la lumière etc.), et définir une «texture» (image calquée directement sur l'objet).

Définir un matériau.

VRML offre un noeud permettant de définir un matériau:

- **Material {**
  ambientIntensity 0.2
  diffuseColor 0.8 0.8 0.8
  specularColor 0 0 0
  emissiveColor 0 0 0
  shininess 0.2
  transparency 0
  }

- Les paramètres sont soit des composantes de couleurs en RVB comprises entre 0 et 1, soit des coefficients également compris entre 0 et 1.
- *diffuseColor* défini la couleur de l'objet, au sens le plus général du terme. Les lumières de la scène influent sur l'apparence de l'objet grâce à cette couleur, en fonction de l'angle de réflection de la lumière sur l'objet.
- *ambientIntensity* correspond à l'influence de la lumière ambiente de la scène sur l'objet. La lumière ambiente est aussi produite par les lumières de la scène, mais ne dépend pas des diverses réflections (lumière isotropique). - *specularColor* défini la couleur des rayons lumineux réfléchis sur la surface de l'objet et renvoyés vers le point de vue de l'utilisateur. C'est ce qui sert à produire les reflets localisés sur des boules métaliques par exemple.
- *emmissiveColor* défini une couleur propre à l'objet, comme s'il était luminescent. C'est à dire qu'en l'abscence de lumière, l'objet emet cette couleur.
- Les champs *shininess* et *transparency* ne nécessitent sans doute pas plus de détail que leur propre nom n'en donne...

À l'aide du champ ci-dessous,
amusez-vous à modifier le matériau du cube. Mais attention, chaque viewer 3d a ses limites, et tous ne supportent pas nécessairement tous les paramètres. Le rendu peut donc être partiellement infidèle.

```
#VRML V2.0 utf8

Shape {
  geometry Cone {}
}

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1 0 0.5
      ambientIntensity 0.3
      specularColor 0.2 0.6 0.2
      emissiveColor 0.2 0 0.1
      shininess 0.5
      transparency 0.3
    }
  }
  geometry Box { }
}
```

Nettoyer
Visualiser

Définir une texture.
VRML offre plusieurs noeuds permettant de définir une texture, voici les 2 principaux:

- **ImageTexture {**
  url u
  }

- *url* contient l'URL d'une image devant servir pour texturer l'objet.

- **PixelTexture {**
  image 0 0 0
  }

Ce noeud permet de définir une image pixel par pixel directement dans le code VRML. Le format du champ image est le suivant:

```
width height size [pixels ...]
```

- *width* et *height* spécifient la taille de l'image (en pixels)
- *size* spécifie le nombre d'éléments composant chaque valeur de pixel:
  - pour une taille de 1, la valeur correspond à une teinte de gris entre 0 et 255.
  - pour une taille de 2, on a l'instensité (teinte de gris) suivie de l'opacité (255 - transparence).
  - pour une taille de 3, on a les trois composantes RGB.
  - pour une taille de 4, on a les 3 composantes RGB suivies de l'opacité.

À l'aide du champ ci-dessous,
À l'aide du champ ci-dessous, changez l'objet texturé ou le noeud de texture et étudiez la façon dont la texture est appliquée par défaut. Notez également l'influence mutuelle des noeuds *Material* et *ImageTexture*.

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    texture PixelTexture {
      image 3 3 3
      0xffffff 0xffffff 0xffffff
      0xff0000 0x00ff00 0x0000ff
      0x111111 0x111111 0x111111
    }
    material Material {
      diffuseColor 1 0 0.5
      ambientIntensity 0.3
      specularColor 0.2 0.6 0.2
      emissiveColor 0.2 0 0.1
      shininess 0.5
      transparency 0.3
    }
  }
  geometry Box { }
}
```

Nettoyer

Visualiser

Transformation et positionnement des objets.

Maintenant que nous savons créer des objets et leur donner une apparence particulière, nous allons pouvoir les transformer et les positionner dans la scène.

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

- **Transform {**
  center 0 0 0
  translation 0 0 0
  rotation 0 0 1 0
  scale 1 1 1
  scaleOrientation 0 0 1 0
  children [... **]**
  }

Ce noeud fait partie de ce qu'on appelle les «noeuds de groupe». Ce sont des noeuds qui ont la propriété de pouvoir contenir d'autres noeuds du langage dans leur paramètres. Ici, le paramètre **children** contient d'autres noeuds (en particulier des *Shape*), auxquels les transformations géométriques seront appliquées.

Les transformations géométriques effectuées par ce noeud sont les suivantes:
- *translation* déplace l'objet selon les axes X Y et Z. - *rotation (x y z a)* tourne de a (en radians) autour du vecteur (x y z).
- *scale* (homothétie) modifie l'échelle selon les 3 axes du repère défini par *scaleOrientation* (cet axe fonctionne de la même manière que le paramètre *rotation*).

À l'aide du champ ci-dessous,
Manipulez les différents paramètres du noeud *Transform*, et notez comment ce noeud permet de séparer les caractéristiques assignées à chaque objet, non seulement les formes géométriques, mais

aussi les apparences..

```
#VRML V2.0 utf8

Transform {
  translation -2 0 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0 0.8 0.1
          transparency 0.2
        }
      }
      geometry Box { }
    }
  ]
}

Transform {
  translation 2 0 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0 0.1 0.8
        }
      }
      geometry Cone { }
    }
  ]
}
```

Nettoyer

Visualiser

Pour mieux comprendre le noeud *Transform* ...

Considérons que les opérations géométriques effectuées se déroulent dans l'ordre suivant:

Transform {		- Translation de -C
translation T		- Rotation de -RS
rotation R		- Homothétie de S
scaleFactor S	<==>	- Rotation de RS
scaleOrientation SR		- Rotation de R
center C		- Translation de C
}		- Tranlsation de T

Lumière !

En général, les viewers s'occupent de mettre une lumière ambiante dans les scènes, et offrent souvent la possibilité de mettre une *head light* c'est à dire une lampe collée sur votre front ! Il peut parfois être utile de définir soi-même ses propres sources de lumière. Il en existe trois :

Source ponctuelle:

Ce noeud défini une source de lumière en un point donné, et qui rayonne de manière isotrope. Voici la syntaxe:

- **PointLight {**
  intensity 1
  color 1 1 1
  location 0 0 0
  radius 100
  attenuation 1 0 0

ambientIntensity 0
}

- *intensity* spécifie l'intensité de la lumière.
- *color* spécifie sa couleur.
- *location* spécifie l'endroit où se trouve la source ponctuelle.
- *radius* spécifie la distance (en mètres) à l'interieur de laquelle cette source lumineuse a un effet sur les objets.
- *attenuation* donne les 3 coéficients d'une équation du second degré donnant l'atténuation de la lumière en fonction de la distance à l'objet éclairé.
- *ambientIntensity* spécifie la quantité de lumière ambiente générée par cette source.

Projecteur:

Ce noeud produit une lumière de type projecteur, dans un cône. La syntaxe est la suivante :

- **SpotLight {**
  intensity 1
  direction 0 0 -1
  color 1 1 1
  location 0 0 0
  radius 100
  attenuation 1 0 0
  ambientIntensity 0
  beamWidth 1.570796
  cutOffAngle 0.785398
  }

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

Source directionnelle:

Ce noeud défini une source de lumière illuminant la scène selon des raies parrallèles. La syntaxe suit:

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

A l'aide du champ ci-dessous:

Testez les différentes sources de lumières existant.
Modifiez éventuellement le *Material* pour l'objet illuminé.

```
#VRML V2.0 utf8

SpotLight {
  location 2 0 0
  direction -1 0 0
}

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0 0.1 0.8
    }
  }
  geometry Sphere { }
}
```

Nettoyer
Visualiser

Animation

- VRML 97 permet d'animer les objets de façon plus ou moins complexe:
  - directement en VRML, en utilisant le **mécanisme d'événements** ;
  - en décrivant des comportements plus fins dans un véritable langage de programmation, par l'intermédiaire des noeuds ***Script***. Ces *Scripts* (en Java, Javascript ou VRMLscript) sont interprétés directement par le viewer VRML.

  Dans ce tutoriel, nous nous en tiendrons aux mécanismes de base d'animation en VRML, et en particulier aux animations mettant en jeu des déplacements d'objets et des changements de couleur. Nous n'aborderons pas la question des *Scripts* car VRWave ne supporte pas encore cette fonctionnalité.

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

- **Les événements**

  Un événement est un message qui contient une valeur d'un certain type (le type du champ considéré). Chaque noeud VRML est composé de champs qui peuvent être:
  - des événements en entrée (*eventIn*);
  - des événements en sortie (*eventOut*);
  - des événements en entrée/sortie (*exposedField*);
  - des champs en lecture seule (*field*).

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

- **Les horloges**

  Un ***TimeSensor*** permet de générer des événements correspondant à des tics d'horloge à intervalles réguliers.

  Définition du *TimeSensor*:

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

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

Les interpolateurs

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

Exemple: le *PositionInterpolator*:

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

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

À l'aide du champ ci-dessous:

Visualisez le cône animé, puis modifiez l'intervalle des horloges et les valeurs des interpolateurs pour transformer l'animation.

```
#VRML V2.0 utf8

DEF CONE Transform {
  children [
    Shape {
      appearance Appearance {
        material DEF CONE_COLOR Material {
          diffuseColor 1.0 1.0 0.0
        }
      }
      geometry Cone {}
    }
  ]
}

DEF POSITION_CLOCK TimeSensor {
  cycleInterval 6
  loop TRUE
  stopTime -1
}

DEF POSITIONS PositionInterpolator {
  key [ 0.0 0.25 0.5 0.75 1.0 ]
  keyValue [ -3.0 0.0 0.0,
              0.0 1.0 0.0,
              3.0 0.0 0.0,
              0.0 1.0 0.0,
             -3.0 0.0 0.0 ]
}

ROUTE POSITION_CLOCK.fraction_changed TO POSITIONS.set_fraction
ROUTE POSITIONS.value_changed TO CONE.translation

DEF COLOR_CLOCK TimeSensor {
  cycleInterval 2
  loop TRUE
  stopTime -1
}

DEF COLORS ColorInterpolator {
```

Nettoyer
Visualiser

Les capteurs

Il existe plusieurs types de capteurs des événements utilisateur, mais les plus intéressants sont le *TouchSensor*, qui capte en particulier les clics souris sur un objet, et le *ProximitySensor*, qui permet de détecter la présence de l'avatar (représentation de l'utilisateur dans la scène) à une certaine distance d'un objet. Grâce à ces capteurs, on peut par exemple cliquer sur une lampe pour l'allumer ou faire s'ouvrir une porte automatiquement dès que l'utilisateur s'en approche.

À l'aide du champ ci-dessous:
Si le viewer VRML le permet, vous pourrez visualisez cette animation simple: en cliquant sur le cône correspondant à la lampe, vous verrez le cube s'éclairer. Remarque: essayez de comprendre pourquoi la lumière s'allume quand on clique, mais s'éteint lorsqu'on relâche le bouton de la souris...

```
#VRML V2.0 utf8

Transform {
  children [
    Shape {
      geometry Cone {
        bottom FALSE
      }
      appearance Appearance {
        material Material {
          diffuseColor 1.0 0.0 0.0
        }
      }
    },
    DEF LIGHT SpotLight {
      on FALSE
      color 1.0 1.0 0.0
    },
    DEF LIGHT_SENSOR TouchSensor {}
  ]
  translation 0.0 3.0 0.0
}

Shape {
  geometry Box {}
```

Nettoyer
Visualiser

Quelques autres noeuds ...

Voici en vrac un certain nombre d'autres noeuds intéressants. Attention, tous ne sont pas nécessairement supportés par le viewer que vous utilisez ...

Ancres www:

- **Anchor {**
  url []
  description ""
  children []
  }

- *url* permet, en cliquant sur un des objet fils, de se connecter à l'adresse indiquée.
- *description* devrait apparaître quelque part dans votre viewer quand vous passez sur un objet cliquable.

Billboard:
Le billboard est un noeud de groupe qui oblige chaque géométrie descendante à rester tournée vers l'utilisateur, quelle que soit sa position. Cela peut être utile pour représenter par exemple un arbre comme une surface plane avec texture, mais qui reste toujours visible sous le même angle.

- **Billboard {**
  axisOfRotation 0 1 0
  children []
  }

- *axisOfRotation* permet de définir l'axe autour duquel les géométries descendantes tournent pour rester orientées vers l'utilisateur.

Collisions:

Par défaut, l'utilisateur n'est pas censé pouvoir pénétrer à l'interieur des objets. Il entre en collision avec. Le noeud *Collision* permet de désactiver ce comportement.

```
Collision {
  collide TRUE
  children []
}
```

- *collide* permet de spécifier si l'utilisateur doit entrer en collision avec les géométries descendantes ou non.

Texte:

```
Text {
  string []
  maxExtent 0.0
}
```

- *string* permet de spécifier un ensemble de chaînes de caractères à afficher dans le plan (local) Z=0.
- *maxExtent* permet de limiter la taille du texte. Si le texte réel dépasse la taille autorisée, il sera compressé.

Un exemple complet.

Un bon bout de code vaut mieux qu'un long discours ... Voici donc en cadeau une bille de clown, avec presque tous les noeuds qu'on a pu voir dans ce tutoriel. Triturez, déformez, cliquez-lui sur le nez ...

```
#VRML V2.0 utf8

# Une lumiere directionnelle a 10h ...
DirectionalLight {
  direction -1 0 -1
}

# Le crane ...
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1 0.8 0.8
    }
  }
  geometry Cylinder { }
}

# Les yeux ...
Transform {
  translation -0.4 0.4 0.8
  children [
    DEF oeuil Shape {
      appearance Appearance {
        material Material {
          diffuseColor  0 0 1
          specularColor 1 1 1
        }
        texture PixelTexture {
          image 4 4 2
          0xffff 0xffff 0xffff 0xffff
          0xffff 0xffff 0xff79 0xffff
          0xffff 0xffff 0xffff 0xffff
          0xffff 0xffff 0xffff 0xffff
        }
      }
      geometry Sphere { radius 0.2 }
    }
  ]
}

Transform {
  translation 0.4 0.4 0.8
  children [ USE oeuil ]
}

# Les oreilles ...
DEF oreilleGauche Transform {
  translation -1.1 0 0
  scale 0.25 1 0.5
  rotation 1 0 0 1.57
```

Nettoyer

Visualiser

Page maintenue par Didier Verna, didier@lrde.epita.fr.