# Table des matières

# Introduction

VRcaML est une bibliothèque de fonctions Caml[1] permettant de décrire et de manipuler des objets en trois dimensions, puis de les traduire en VRML[2] pour pouvoir les visualiser utilisant les *browsers* appropriés[3].

Ce document présente à la fois les fonctions de haut-niveau de création et de manipulation d'objets VRcaML, et des fonctions de plus bas niveau, qui permettent une gestion plus fine des scènes 3D décrites et qui peuvent être ignorées (au moins dans un premier temps).

La majorité de ces fonctions concerne les scènes statiques. La partie dynamique de VRcaML est plus restreinte, mais offre déjà des possibilités intéressantes pour l'animation des objets.

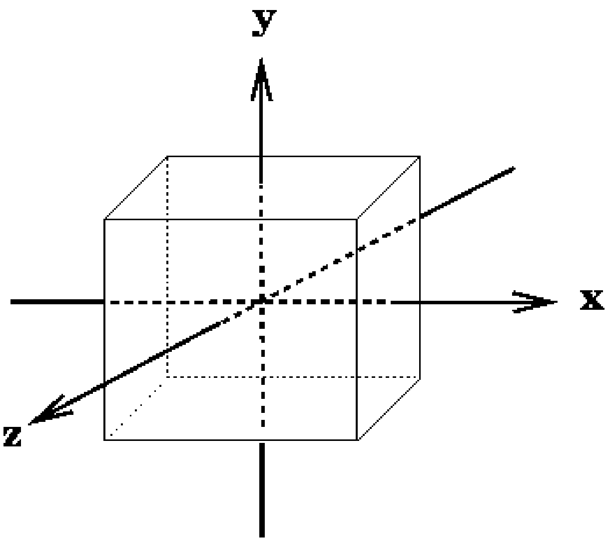
1. Caml-Light 0.7
2. VRML : *Virtual Reality Modeling Language*
3. *Cosmo Player*, *Liquid Reality*, *Community Place*, etc.

# Chapitre 1

# Objets et référentiels

VRcaML est basé sur une architecture arborescente d'objets 3D que l'on peut positionner dans l'espace[1]. Seuls les brouillards et les fonds (*backgrounds*) ne sont pas positionables et doivent être définis hors de l'arborescence (objets dits spéciaux ).

Chaque objet décrit son propre référentiel, définissant sa position, son orientation et son échelle, le tout par rapport à un autre objet. Les transformations géométriques appliquées à un objet VRcaML sont donc en fait subies par le référentiel associé à cet objet. Les référentiels sont, par défaut, orientés selon le schéma suivant[2] :




Les référentiels possèdent un ou plusieurs objets en leur centre (ce sont les référentiels propres à ces objets), mais peuvent aussi posséder des enfants , correspondant aux objets et référentiels placés par rapport à eux. Un objet subit donc toujours une translation ou une rotation par rapport à son père, et non par rapport à son propre référentiel.

Il y a création d'une dépendance entre le père et ses enfants : si on applique une transformation géométrique au père, tous ses enfants la subiront aussi.

1. placement des objets les uns par rapport aux autres, regroupement d'objets pour leur appliquer les mêmes transformations géométriques (translation, rotation et redimensionnement) ou les mêmes propriétés (couleur, texture, etc.), etc.

2. L'utilisateur d'un *browser* VRML est par défaut placé devant un objet, et regarde dans la direction opposée à l'axe des z.

Les objets d'un référentiel peuvent être des objets géométriques simples (sphère, boîte, texte 2D, cone, cylindre) ou complexes (extrusion, grille de niveaux, objets à facettes définis par l'utilisateur), mais aussi des sons, des lumières, des points de vue dans la scène (*view points*), des zones de collision [3], ou encore des ensembles colorés de points et de lignes (*point sets* et *line sets*).

Une scène VRcaML est définie par l'arborescence des objets qu'elle contient, ainsi que par ses brouillards, ses fonds et ses animations.

3. zones dans lesquelles on ne peut pas pénétrer

# Chapitre 2

# Les types de base

Les champs des objets VRcaML contiennent des valeurs plus ou moins structurées, dont certaines peuvent être créées et manipulées indépendamment. Il est déconseillé de créer ces valeurs directement : les fonctions de création lèvent des exceptions quand les valeurs entrées par l'utilisateur n'ont pas de sens. Cela évite les surprises à l'affichage de la scène VRML.

## 2.1 Vecteurs

Il existe deux types de vecteurs en VRcaML : à deux et à trois dimensions.

```
type vect2D == float * float ;;
type vect3D == float * float * float ;;
```

Dans le cas de vecteurs à trois dimensions, les valeurs correspondent, dans l'ordre, aux axes des x, des y et des z des référentiels (*cf.* chapitre 1 page 3). Dans le cas de vecteurs à deux dimensions, les valeurs correspondent à une direction permettant par exemple de décrire une forme 2D [1].

**Création :**

```
new_vect2D : float -> float -> vect2D
new_vect3D : float -> float -> float -> vect3D
```

## 2.2 Rotation

Une rotation dans l'espace est définie par une direction et un angle, donc par un vecteur à trois dimensions et un flottant.

```
type rotation == vect3D * float ;;
```

**Création :**

```
new_rotation : float -> float -> float -> float -> rotation
```

1. voir par exemple la description d'une *cross-section* pour une extrusion, section 3.3.2

## 2.3 Couleur

Les couleurs sont définies en RGB : le premier nombre représente la quantité de rouge, le deuxième celle de vert et le troisième celle de bleu. Comme en VRML, ces valeurs sont comprises entre 0.0 et 1.0. Le noir est donc par exemple représenté par le triplet (0.0, 0.0, 0.0) et le blanc par (1.0, 1.0, 1.0).

```
type color == float * float * float ;;
```

**Création :**

```
new_color : float -> float -> float -> color
```

**Remarque :** il existe un grand nombre de couleurs prédéfinies dans VRcaML. La liste des couleurs disponibles se trouve dans le fichier `COLORS`. On peut utiliser ces couleurs à condition de faire un `include` du fichier `colors.ml`.

## 2.4 Temps

Le temps est représenté par un nombre (réel) de secondes. Il est nécessaire au type `clip` (*cf.* ci-dessous, section 2.6), et sont souvent utiles pour les animations.

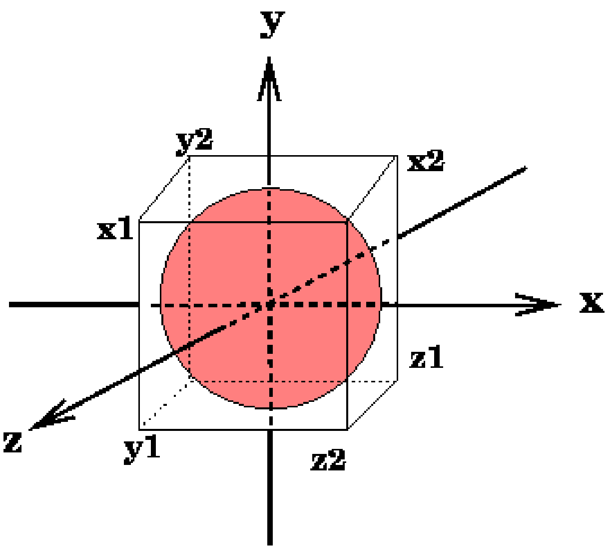
```
type time == float ;;
```

**Création :**

```
new_time : time -> time
```

## 2.5 *Frame* d'un objet

La *frame* d'un objet est la boîte qui l'entoure (*cf.* section 3.1.1 page 10). Elle est calculée approximativement selon la forme de l'objet et les transformations géométriques qu'il a pu subir.



Cette boîte permet de placer facilement les objets les uns par rapport aux autres, sans avoir à calculer à la main leurs coordonnées (*cf.* section 3.1.5 page 13).

```
type frame =
{ width  : float * float ;
  height : float * float ;
  depth  : float * float
} ;;
```

La largeur (*width*) est donnée par le couple (x1,x2), la hauteur (*height*) par (y1,y2) et la profondeur (*depth*) par (z1,z2).

**Création :**

```
new_frame : float -> float -> float -> float -> float -> float -> frame
```

**Remarque :** la frame d'un texte 2D n'est jamais calculée, car sa taille réelle est difficile à déterminer.

## 2.6 *Clip*

Le type `clip` représente soit un clip audio utilisé par le type `sound` (*cf.* section 3.6 page 23), soit une séquence vidéo utilisée pour les textures animées (*cf.* section 3.3.1 page 14). Dans les deux cas, il est nécessaire de donner les instants de début et de fin du clip, ainsi que l'URL correspondant au fichier son ou vidéo[2]. Le booléen `loop` indique si le clip doit être joué en boucle ou non.

```
type clip =
{ clip_loop       : bool ;
  clip_start_time : time ;
  clip_stop_time  : time ;
  clip_urls       : string list
} ;;
```

**Création :**

```
new_clip : time -> time -> string -> clip
new_complete_clip : bool -> time -> time -> string list -> clip
```

## 2.7 Fontes

Le nombre de polices de caractères disponibles est limité à trois familles (*plain*, sérif et sans sérif) et aux quatre styles courants (normal, gras, italique, gras et italique). Par contre, il est possible de faire des effets intéressants en jouant sur le sens de lecture[3].

2. URLs : il est possible de donner des URLs alternatives qui seront utilisées s'il est impossible d'atteindre l'URL principale. C'est pour cela que les champs correspondant à l'URL d'un `vrml_object` sont en fait des listes de chaînes de caractères, la première URL de la liste étant l'URL principale.

3. horizontal, de gauche à droite et de haut en bas par défaut

On peut bien sûr modifier la taille et l'espacement des caractères, mais aussi jouer sur la justification[4]. Il est enfin possible de spécifier une langue (japonais, coréen, etc.).

```
type fontStyle =
{ family        : string ;
  horizontal    : bool ;
  justify       : string list ;
  language      : string ;
  left_to_right : bool ;
  size          : float ;
  spacing       : float ;
  style         : string ;
  top_to_bottom : bool
} ;;
```

**Création simple (famille, taille, style) :**

```
new_font : string -> float -> string -> fontStyle
```

**Création complète (tous les champs dans l'ordre de l'enregistrement) :**

```
new_complete_font :
   string -> bool -> string list -> string -> bool -> float -> float
   -> string -> bool -> fontStyle
```

**Remarque :** toutes les fontes de base (avec les valeurs par défaut pour les champs autres que la famille et le style) sont prédéfinies dans VRcaML.

```
plain_serif
plain_sans_serif
plain_typewriter
bold_serif
bold_sans_serif
bold_typewriter
italic_serif
italic_sans_serif
italic_typewriter
bold_italic_serif
bold_italic_sans_serif
bold_italic_typewriter
```

## 2.8 Image

Une image VRcaML représente un bitmap dont on connaît la taille (largeur et hauteur) et la valeur hexadécimale de chaque pixel. Les pixels sont codés selon le nombre de leurs composants (ou nombre d'octets) :

- 1 : chaque pixel est une valeur hexadécimale sur 1 octet, représentant une intensité[5] ;

4. *cf.* la documentation de référence de VRML [1]
5. `0xFF` est l'intensité maximale d'un point

- 2 : chaque pixel se décompose en une intensité (1er octet) et une opacité (dernier octet) ;
- 3 : chaque pixel est codé en RGB ;
- 4 : chaque pixel comporte son code RGB (trois premiers octets) et son opacité (dernier octet).

```
type image =
{ image_width      : int ;
  image_height     : int ;
  image_components : int ;
  image_pixels     : int list
} ;;
```

**Création (largeur, hauteur, nombre de composants, liste des pixels) :**

```
new_image : int -> int -> int -> int list -> image
```

# Chapitre 3

# Description des objets VRcaML

Ce chapitre décrit un par un les objets qu'il est nécessaire de connaître pour définir des scènes VRcaML, ainsi que leurs fonctions de création et de manipulation de haut-niveau.

Il faut préciser que cette bibliothèque a été conçue pour permettre de créer facilement et rapidement des mondes VRML dans un style purement fonctionnel, et dans une optique de construction par ajout : l'utilisateur créé des objets, qu'il regroupe ou qu'il place les uns par rapport aux autres, et auxquels il applique des propriétés ou des transformations géométriques. Rien n'est prévu pour détruire des objets ou retirer des propriétés, et tout se passe par copie.

Il est fortement déconseillé de créer ou modifier soit même les types VRcaML : il faut toujours utiliser les fonctions de haut-niveau, et éventuellement les fonctions avancées faisant appel à des fonctions de plus bas niveau.

## 3.1 L'objet et son référentiel

### 3.1.1 Principes du type `vrml_object`

Il est utile de connaître la structure du type `vrml_object`, représentant un référentiel, pour bien comprendre les fonctions de manipulation des objets VRcaML.

```
type vrml_object =
{ objects   : vrml_positionable list ;
  children  : vrml_object list ;
  position  : vect3D ;
  rotation  : rotation ;
  scale     : vect3D ;
  frame     : frame ;
  unique    : bool ;
  anchor    : anchor ;
  billboard : billboard ;
  collision : collision } ;;
```

Le référentiel connaît les objets placés en son centre (champ `objects`) ainsi que ses enfants (champ `children`). Il peut être placé dans l'espace (champ `position`), ou subir une rotation (champ `rotation`), toujours par rapport à son père. Il possède une échelle (champ `scale`), qui peut varier lors d'un redimensionnement.

La *frame*, ou *bounding box*, d'un objet (champ `frame`) correspond à la boîte qui l'entoure (*cf.* section 2.5 page 6).

Le champ `unique` n'est utilisé que pour les animations : il est en effet possible d'avoir plusieurs instances d'un objet dans une scène VRcaML. Dans ce cas, toutes les instances subiront les animations en même temps. Mais on peut vouloir représenter plusieurs fois le même objet dans la scène, sans pour autant animer de la même façon toutes ses copies : plusieurs objets de la scène auront exactement la même structure, mais les instances  uniques  seront indépendantes.

**Remarque :** pour rendre un objet unique, il faut utiliser la fonction `make_unique` qui donne un nom à l'objet par un effet de bord. Pour différencier des objets structurellement égaux mais dit  uniques , on utilise la comparaison physique (`==`) : cette méthode peut engendrer des incohérences lors de la traduction si on ne prend pas ses précautions.

Enfin, les objets VRcaML peuvent posséder des propriétés :

- `anchor` : ce champ permet de doter l'objet d'une URL, qui sera activée lorsque l'utilisateur cliquera sur l'objet.
- `billboard` : si l'objet possède un *billboard*, il subira une rotation continue, destinée à toujours présenter l'objet de la même façon à l'utilisateur.
- `collision` : par défaut, l'utilisateur ne peut pas pénétrer dans un objet. Le champ `collision` permet soit de rendre cette opération possible, soit de définir une boîte de  collision  plus grande que l'objet lui-même.

### 3.1.2 Création et manipulation

**Création d'un `vrml_object` :** on ne crée en fait jamais directement de `vrml_object`, puisque les fonctions de créations des objets VRcaML (sphère, extrusion, son, etc.) masquent cette opération. Il est par contre possible d'obtenir un référentiel vide :

```
new_referential : vrml_object
```

**Mise à jour des champs `objects` et `children` :** on peut affecter une nouvelle liste au champ `children`, mais aussi ajouter des enfants ou des objets à un `vrml_object`.

```
add_objects : vrml_object -> vrml_positionable list -> vrml_object
add_object : vrml_object -> vrml_positionable -> vrml_object

set_children : vrml_object -> vrml_object list -> vrml_object
add_children : vrml_object -> vrml_object list -> vrml_object
add_child : vrml_object -> vrml_object -> vrml_object
```

**Mise à jour de la position :** on peut affecter une nouvelle valeur à la position d'un objet ou translater l'objet à partir de sa position courante.

```
set_position : vrml_object -> float -> float -> float -> vrml_object
translation : vrml_object -> float -> float -> float -> vrml_object
```

**Mise à jour de la rotation :** on peut affecter une nouvelle valeur à la rotation d'un objet, ajouter un angle ou bien une rotation complète (direction et angle) à la rotation courante. Cette dernière opération implique la création d'un nouveau référentiel qui encapsule l'objet pour lui faire subir une rotation supplémentaire.

```
set_rotation : vrml_object -> float -> float -> float -> float -> vrml_object
rotation : vrml_object -> float -> vrml_object
add_rotation : vrml_object -> float -> float -> float -> float -> vrml_object
```

**Mise à jour de l'échelle :** on peut affecter une nouvelle valeur à l'échelle d'un objet, ou redimensionner l'objet [1].

```
set_scale : vrml_object -> float -> float -> float -> vrml_object
resize : vrml_object -> float -> float -> float -> vrml_object
```

**Affectation de la *frame* :** il est possible, au besoin, de définir soi-même la *frame* d'un objet. Cette opération est utile lorsque l'on souhaite positionner des objets systématiquement à une certaine distance d'un autre.

```
set_frame : vrml_object -> frame -> vrml_object
```

**Mise à jour de l'URL :** on peut affecter une URL à un objet, ou en ajouter à la liste [2].

```
set_url : vrml_object -> string -> vrml_object
add_url : vrml_object -> string -> vrml_object
```

**Affectation d'un *billboard* :** il faut pour cela indiquer l'axe de rotation du *billboard* (trois dimensions du vecteur).

```
set_billboard : vrml_object -> float -> float -> float -> vrml_object
```

**Utilisation de la collision :** pour pouvoir pénètrer dans les objets de la scène, il faut qu'ils soient *uncollidable* (c'est-à-dire qu'on ne peut pas se cogner dedans). Il est donc possible de rendre un objet *uncollidable* ou *collidable*, mais aussi de spécifier une boîte si l'on souhaite éviter que l'utilisateur ne s'approche trop près d'un objet. Dans ce cas, il faudra préciser le centre et la taille de la boîte (deux vecteurs à trois dimensions).

```
set_collidable : vrml_object -> vrml_object
set_uncollidable : vrml_object -> vrml_object
set_collision : vrml_object -> vect3D -> vect3D -> vrml_object
```

**Remarque :** la détection de collision semble dépendre des *browsers*, et est apparemment assez mal gérée.

### 3.1.3 Regroupement

Le regroupement d'objets VRcaML permet d'appliquer une même transformation ou une même propriété sur plusieurs objets à la fois. Il est par ailleurs possible d'appliquer des propriétés dépendantes d'un type d'objet : par exemple, on ne peut donner une couleur qu'aux objets géométriques normaux (cubes, cones, etc.). L'application d'une couleur à un

1. Attention : dans le cas d'un redimensionnement, les valeurs données seront multipliées aux valeurs existantes.
2. ajout en fin de liste, c'est-à-dire dans l'ordre chronologique de prise en compte des URLs (URL principale en tête).

groupe d'objets reviendra donc à appliquer cette couleur sur tous les objets géométriques qui l'acceptent, et d'ignorer les autres types d'objets[3].

Il est possible de défaire un groupe, afin de recupérer séparément les objets auxquels on aura appliqué les mêmes modifications.

```
group : vrml_object list -> vrml_object
ungroup : vrml_object -> vrml_object list
```

Le regroupement est une opération simple : on crée un nouveau référentiel, qui prend comme enfants les éléments de la liste donnée en argument. Il permet donc aussi de regrouper facilement des objets éparpillés dans la scène et de leur donner un référentiel commun, donc une sorte d'origine absolue, qui pourra par exemple être la racine de l'arborescence de la scène VRcaML.

### 3.1.4 Agrégation

L'agrégation est une opération plus complexe, qu'il n'est pas trop recommandé d'utiliser car elle est irréversible [4] : il s'agit ici de récupérer les objets du champ `objects` d'un `vrml_object`, et de les mettre dans le champ `objects` d'un autre `vrml_object`. Les propriétés des objets absorbés sont perdues (position, collision, etc.). Cette opération peut être utile si l'on souhaite placer des objets au centre d'un même référentiel sans jouer sur les positionnements et les regroupements.

```
agregate_one : vrml_object -> vrml_object -> vrml_object
agregate : vrml_object list -> vrml_object -> vrml_object
```

### 3.1.5 Positionnements relatifs

**A FAIRE !!!!**

## 3.2 Scène 3D

Une scène VRcaML (ou monde) est définie par son titre et les informations supplémentaires que l'auteur peut vouloir donner, par la racine de son arborescence d'objets positionés dans l'espace, par la liste des objets spéciaux qu'elle contient (brouillards et fonds), et enfin par la liste des animations qui peuvent s'y dérouler.

```
type vrml_world =
{ info       : string list ;
  title      : string ;
  tree       : vrml_object ;
  specials   : vrml_special list ;
  animations : animation list
} ;;
```

3. *cf.* la section 3.3.1 page 14 sur les objets géométriques et leurs propriétés, ainsi que la section 4 page 28 sur les fonctions avancées.

4. au sens fonctionnel du terme, car les objets initiaux ne seront pas modifiés. Par contre, le résultat d'une agrégation ne pourra pas être décomposé pour retrouver les objets initiaux

**Création :**

```
new_vrml_world :
 string list -> string -> vrml_object -> vrml_special list -> vrml_world
```

**Manipulation des champs :**

```
set_info : vrml_world -> string list -> vrml_world
set_title : vrml_world -> string -> vrml_world
set_tree : vrml_world -> vrml_object -> vrml_world
set_specials : vrml_world -> vrml_special list -> vrml_world
add_special : vrml_world -> vrml_special -> vrml_world
set_animations : vrml_world -> animation list -> vrml_world
add_animation : vrml_world -> animation -> vrml_world
```

## 3.3 Objets géométriques

Cette section décrit les objets géométriques normaux , c'est-à-dire ceux qui peuvent posséder une texture ou un matériau.

### 3.3.1 Propriétés : texture et materiau

Il existe trois types de textures, qui peuvent être créées et manipulées indépendamment des objets VRcaML :

- *image texture* : il suffit de donner l'URL d'un fichier image (`gif` ou `jpeg` par exemple).
- *pixel texture* : il faut d'abord créer une image VRcaML (type `image`), pour l'affecter à cette texture.
- *movie texture* : les textures animées nécessitent la création d'un clip (type `clip`). Il est par ailleurs possible de préciser la vitesse de défilement du clip vidéo, lors d'une création complète.

```
new_imageTexture : string -> texture
new_pixelTexture : image -> texture
new_movieTexture : clip -> texture
new_complete_movieTexture : float -> clip -> texture
```

La texture par défaut est l'*image texture* : c'est celle qui est appliquée par la fonction `map_texture`. Les 4 fonctions suivantes appliquent la modification de l'élément d'une texture à tous les objets géométriques centrés sur un référentiel, à savoir l'URL d'une *image texture*, l'image d'une *pixel texture*, et le clip ou la vitesse d'une *movie texture*.

```
map_texture : vrml_object -> string -> vrml_object
map_pixelTexture_image : vrml_object -> image -> vrml_object
map_movieTexture_speed : vrml_object -> float -> vrml_object
map_movieTexture_clip : vrml_object -> clip -> vrml_object
```

Pour appliquer les mêmes propriétés à tous les enfants d'un référentiel, il faudra utiliser les fonctions suivantes :

```
map_all_texture : vrml_object -> string -> vrml_object
```

```
map_all_pixelTexture_image : vrml_object -> image -> vrml_object
map_all_movieTexture_speed : vrml_object -> float -> vrml_object
map_all_movieTexture_clip : vrml_object -> clip -> vrml_object
```

Le matériau d'un objet géométrique définit la façon dont l'éclairage de la scène va se refléter sur l'objet.

```
type materialItem =
{ ambient_intensity : float ;
  diffuse_color     : color ;
  emissive_color    : color ;
  shininess         : float ;
  specular_color    : color ;
  transparency      : float
} ;;
```

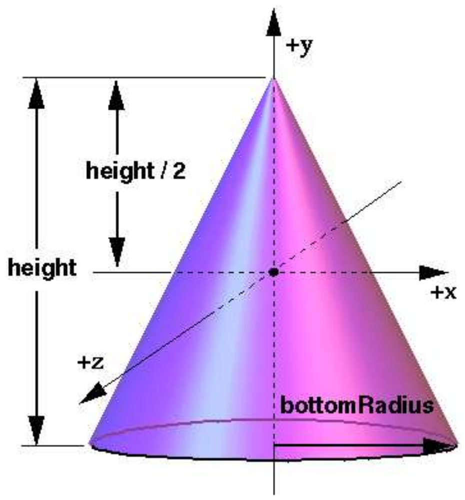
La couleur de l'objet est donnée par le champ `diffuse_color`. L'intensité ambiante (`ambient_intensity`), associée à la couleur, spécifie la façon dont la lumière ambiante se réfléchit à la surface de l'objet. Le champ `emissive_color` exprime la rayonnance de l'objet. Les champs `shininess` et `specular_color` détermine l'existence ou non de points de brillance sur la surface de l'objet. Enfin, le champ `transparency` indique le niveau de transparence d'un objet.

En fait, à part le champ `diffuse_color`, les valeurs par défaut des champs du matériau suffisent amplement pour engendrer des objets géométriques colorés. C'est pour cela que seule la fonction `map_color`, qui modifie la couleur de tous les objets placés au centre d'un référentiel, et la fonction `map_all_color`, qui effectue la même opération sur tous les objets et les enfants d'un `vrml_object`, sont proposées dans les fonctions de haut-niveau. Il faudra utiliser les fonctions avancées pour accéder aux autres champs.

```
map_color : vrml_object -> color -> vrml_object
map_all_color : vrml_object -> color -> vrml_object
```

### 3.3.2 Création et manipulation

**Cone**



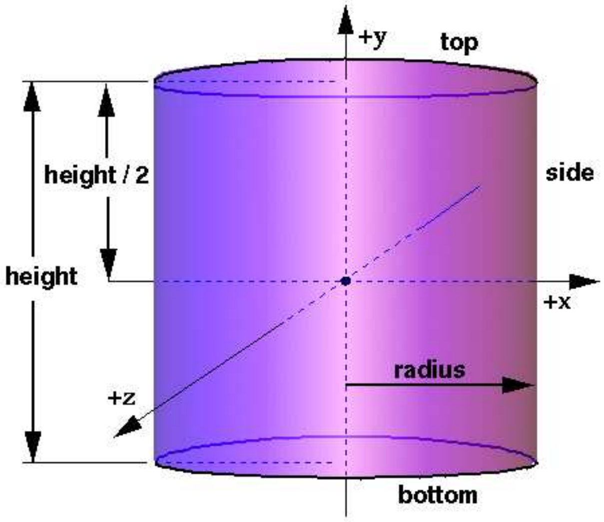
**Type VRcaML :**

```
type cone =
{ cone_bottom_radius : float ;
  cone_height        : float ;
  cone_side          : bool ;
  cone_bottom        : bool
} ;;
```

**Création :** pour créer un cone, il faut donner dans l'ordre son rayon et sa hauteur. Il est possible de spécifier que l'on ne souhaite pas la présence du côté ou du fond du cone, lors d'une création plus complète.

```
new_cone : float -> float -> vrml_object
new_complete_cone : float -> float -> bool -> bool -> vrml_object
```

### Cylindre



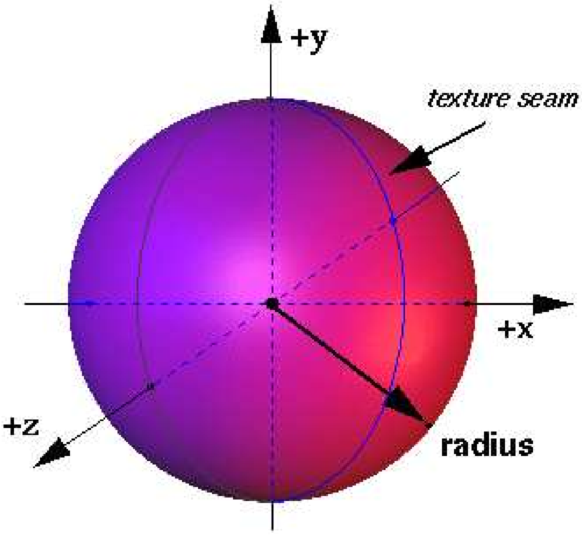

**Type VRcaML :**

```
type cylinder =
{ cylinder_bottom : bool ;
  cylinder_height : float ;
  cylinder_radius : float ;
  cylinder_side   : bool ;
  cylinder_top    : bool
} ;;
```

**Création :** pour créer un cylindre, il faut donner dans l'ordre son rayon et sa hauteur. On peut éventuellement demander à ce que le fond, le haut ou le côté du cylindre ne soient pas présents (création complète).

```
new_cylinder : float -> float -> vrml_object
new_complete_cylinder :
 float -> float -> bool -> bool -> bool -> vrml_object
```

### Sphère



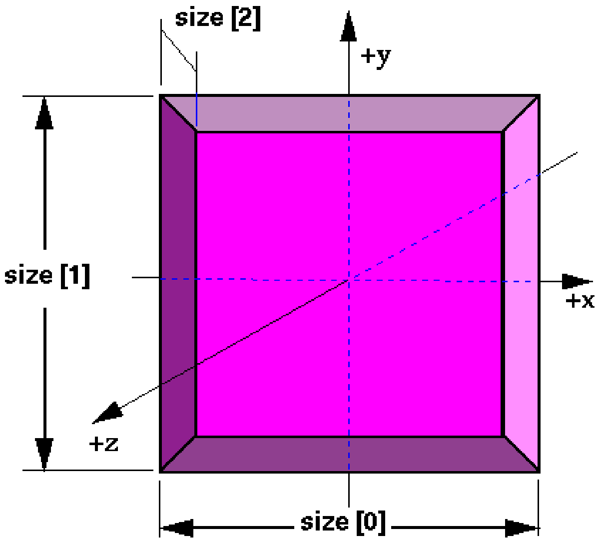

**Type VRcaML :**

```
type sphere = { sphere_radius : float } ;;
```

**Création :** pour créer une sphère, il suffit de donner son rayon.

```
new_sphere : float -> vrml_object
```

### Boîte




**Type VRcaML :**

```
type box = { box_size : vect3D } ;;
```

**Création :** pour créer une boîte, il faut donner les valeurs des côtés pour chaque axe (largeur, hauteur et profondeur, dans cet ordre).

```
new_box : float -> float -> float -> vrml_object
```

Il est possible de créer directement des cubes, en en donnant le côté.

```
new_cube : float -> vrml_object
```

### Texte

**Type VRcaML :**

```
type text =
{ content    : string list ;
  font_style : fontStyle ;
  length     : float list ;
  max_extent : float
} ;;
```

**Création :** pour créer du texte à deux dimensions, il suffit de donner son contenu ligne par ligne (liste de lignes de caractères). Si l'on veut de plus spécifier une fonte particulière (*cf.* section 2.7 page 7), il faudra utiliser la fonction `new_text_with_font`.

Pour une création complète, il faudra indiquer le contenu, la fonte, la longueur de chaque ligne[5] et l'espacement maximal des caractères.

```
new_text : string list -> vrml_object
new_text_with_font : string list -> fontStyle -> vrml_object
new_complete_text :
 string list -> fontStyle -> float list -> float -> vrml_object
```

**Remarque :** c'est le seul objet géométrique dont on ne calcule **jamais** la *frame* ; cela signifie qu'il ne sera pas possible d'utiliser les fonctions de placement relatif avec du texte.

### Objets à facettes définis par l'utilisateur

**Type VRcaML :**

```
type face == vect3D list ;;
type faceSet =
{ faceSet_faces                    : face list ;
  faceSet_texture_coordinate       : textureCoordinate ;
  faceSet_ccw                      : bool ;
  faceSet_convex                   : bool ;
  faceSet_solid                    : bool ;
  faceSet_texture_coordinate_index : int list
} ;;
```

**Création :** ce type est assez complexe, bien qu'il simplifie déjà énormément la création d'objets de ce genre en VRML. Pour une création simple, il suffit de donner la liste des facettes de l'objet, chaque facette étant définie par un ensemble de points dans l'espace.

Pour une création complète, il faudra de plus spécifier les coordonnées des points d'application de la texture, ainsi que l'ordre de description des sommets (champ `faceSet_ccw`, pour *counter-clockwise*), le fait que les facettes soient convexes ou non (champ `faceSet_convex`) et le fait que la forme soit solide ou non (champ `faceSet_solid`).

```
new_faceSet : face list -> vrml_object
```

5. Si cette liste de longueurs est vide, VRML considèrera que les chaînes sont de longueur quelconque. Sinon, il fera en sorte de respecter cette longueur en étendant ou en écrasant le texte.

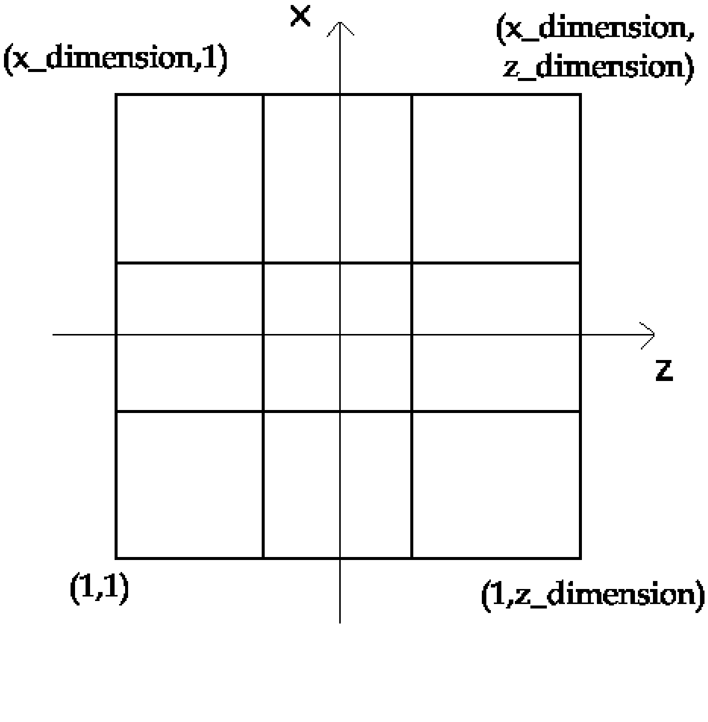
```
new_complete_faceSet :
   face list -> textureCoordinate -> bool -> bool -> bool -> int list
   -> vrml_object
```

### Grille de niveaux

**Type VRcaML :**

```
type elevationGrid =
{ grid_color                : color list ;
  grid_texture_coordinate : textureCoordinate ;
  grid_height               : float list ;
  grid_ccw                  : bool ;
  grid_color_per_vertex     : bool ;
  grid_solid                : bool ;
  grid_x_dimension          : int ;
  grid_x_spacing            : float ;
  grid_z_dimension          : int ;
  grid_z_spacing            : float
} ;;
```

**Création :** les grilles de niveau nécessitent, pour leur création, la liste des points de la grille (hauteurs), la taille de la grille (largeur sur l'axe des x, et longueur sur l'axe des y) et l'espacement des points de la grille sur les deux axes. La liste de hauteurs définie une matrice de (`grid_x_dimension * grid_z_dimension`) :




Pour une création complète, il faudra préciser la liste des couleurs des faces ou des sommets[6], la liste des coordonnées des points d'application de la texture (si texture il y a), la

6. Cette liste est utile si des cases de la même grille sont de couleur différente. Sinon, il suffira d'appliquer une couleur à l'objet, avec `map_color` par exemple.

liste des hauteurs, la largeur et l'espacement entre les points sur l'axe des x, ainsi que la longueur et l'espacement entre les points sur l'axe des z. Les booléens représentent dans l'ordre : l'ordre de description des sommets (champ `faceSet_ccw`), le fait que les couleurs soient appliquées sur les sommets ou sur les cases de la grille (champ `faceSet_color_per_vertex`), et le fait que la grille soit solide ou non (champ `faceSet_solid`).

```
new_elevationGrid : float list -> int -> int -> float -> float -> vrml_object
new_complete_elevationGrid :
   color list -> textureCoordinate -> float list -> bool -> bool -> bool
   -> int -> float -> int -> float -> vrml_object
```

#### Extrusion

**Type VRcaML :**

```
type extrusion =
{ extrusion_begin_cap      : bool ;
  extrusion_ccw            : bool ;
  extrusion_convex         : bool ;
  extrusion_cross_section  : vect2D list ;
  extrusion_end_cap        : bool ;
  extrusion_orientation    : rotation list ;
  extrusion_scale          : vect2D list ;
  extrusion_solid          : bool ;
  extrusion_spine          : vect3D list
} ;;
```

**Création :** pour décrire une extrusion, il faut spécifier une forme à 2 dimensions (*cross-section*) et une trajectoire c'est-à-dire une liste de tranlations dans l'espace décrivant l' épine dorsale (*spine*) de l'objet à créer. La forme 2D suit la trajectoire pour créer une nouvelle forme : à chacun des points de cette trajectoire, il est possible de faire subir une rotation ou un redimensionnement (dans la plan) à la forme 2D pour produire une grande variété de nouvelles formes en 3 dimensions. Il faut savoir que les rotations, les translations et les redimensionnements dont les valeurs sont données dans les champs `extrusion_orientation`, `extrusion_spine` et `extrusion_scale` sont définis les uns par rapport aux autres, dans l'ordre des indices.

```
new_extrusion :
   vect2D list -> vect3D list -> rotation list -> vrml_object
new_extrusion_with_scales :
   vect2D list -> vect3D list -> rotation list -> vect2D list -> vrml_object
new_complete_extrusion :
   bool -> bool -> bool -> vect2D list -> bool -> rotation -> vect2D
   -> bool -> vect3D list -> vrml_object
```

## 3.4 Objets géométriques sans propriété

Il est impossible d'appliquer une texture ou une couleur à un ensemble de points ou à un ensemble de lignes. Ces objets géométriques particuliers sont appelés *unlit geometrics* : en effet, chaque ligne et chaque point possède sa propre couleur, mais les lumières n'ont aucun d'effet sur eux.

### 3.4.1 Ensemble de points colorés

**Type VRcaML :**

```
type point =
{ point_color      : color ;
  point_coordinate : vect3D
} ;;
type pointSet == point list ;;
```

**Création :**

```
new_pointSet : pointSet -> vrml_object
```

### 3.4.2 Ensemble de lignes colorées

**Type VRcaML :**

```
type line =
{ line_point1 : vect3D ;
  line_point2 : vect3D ;
  line_color  : color
} ;;
type lineSet == line list ;;
```

**Création :**

```
new_lineSet : lineSet -> vrml_object
```

## 3.5 Lumières

Les lumières sont aussi des objets que l'on peut positionner dans l'espace. Elles agissent sur les matériaux des objets géométriques normaux .

**Type VRcaML :**

```
type lightType =
  DirectionalLight of directionalLight
| PointLight of pointLight
| SpotLight of spotLight
;;
type light =
{ light_ambient_intensity : float ;
  light_color             : color ;
  light_intensity         : float ;
  light_type              : lightType
} ;;
```

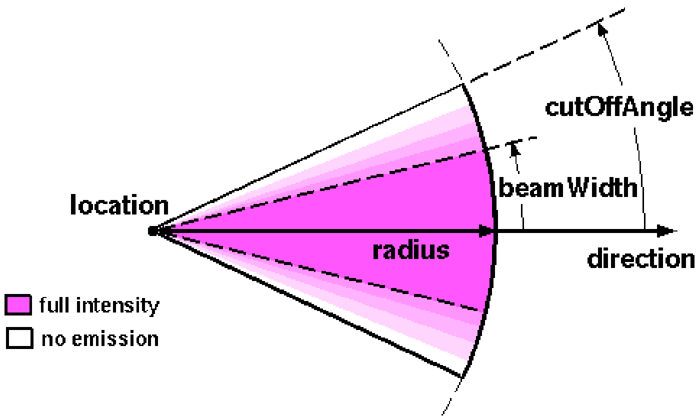
Les fonctions de création de lumières dépendent de leur type (*spot*, point ou lumière directionnelle). Une partie des fonctions de manipulation concerne les champs du type générique `light` et le reste les champs spécifiques à chaque type de lumière.

Les seules fonctions d'accès aux champs disponible sont celles qui semblent les plus utiles, c'est-à-dire celles qui modifient la couleur d'une lumière, quel que soit son type.

```
map_light_color : vrml_object -> color -> vrml_object
map_all_light_color : vrml_object -> color -> vrml_object
```

### 3.5.1 *Spot* de lumière

Le *spot* de lumière éclaire dans une direction, selon le schéma suivant :




**Type VRcaML :**

```
type spotLight =
{ spotLight_attenuation   : vect3D ;
  spotLight_beam_width    : float ;
  spotLight_cut_off_angle : float ;
  spotLight_direction     : vect3D ;
  spotLight_radius        : float
} ;;
```

**Création :** pour créer un *spot*, il suffit de donner sa couleur, son intensité, sa direction et son rayon. Pour une création complète, il faudra indiquer dans l'ordre : la couleur, l'intensité, l'intensité ambiante, l'atténuation, la largeur du rayon lumineux, l'angle de coupure, la direction et le rayon du *spot*.

```
new_spotLight : color -> float -> vect3D -> float -> vrml_object
new_complete_spotLight :
   color -> float -> float -> vect3D -> float -> float -> vect3D -> float
   -> vrml_object
```

### 3.5.2 Lumière directionnelle

Une lumière directionnelle éclaire l'ensemble de la scène de la même façon, dans une direction donnée.

**Type VRcaML :**

```
type directionalLight = { directionalLight_direction : vect3D } ;;
```

**Création :** il suffit de donner sa couleur, son intensité et sa direction. Pour une création plus complète, il faudra la couleur, l'intensité, l'intensité ambiante et la direction.

```
new_directionalLight : color -> float -> vect3D -> vrml_object
new_complete_directionalLight :
 color -> float -> float -> vect3D -> vrml_object
```

### 3.5.3 Point de lumière

Un point de lumière éclaire dans toutes les directions à partir de sa position dans l'espace.

**Type VRcaML :**

```
type pointLight =
{ pointLight_attenuation : vect3D ;
  pointLight_radius      : float
} ;;
```

**Création :** un point de lumière se définit simplement par sa couleur, son intensité et son rayon. Pour une création plus complète, il faudra indiquer la couleur, l'intensité, l'intensité ambiante, l'atténuation et le rayon.

```
new_pointLight : color -> float -> float -> vrml_object
new_complete_pointLight :
   color -> float -> float -> vect3D -> float -> vrml_object
```

## 3.6 Sons

**Type VRcaML :**

```
type sound =
{ sound_direction   : vect3D ;
  sound_intensity   : float ;
  sound_max_back    : float ;
  sound_max_front   : float ;
  sound_min_back    : float ;
  sound_min_front   : float ;
  sound_priority    : float ;
  sound_spatialize  : bool ;
  sound_description : string ;
  sound_pitch       : float ;
  sound_clip        : clip
} ;;
```

**Création :** pour créer un son, il suffit de donner son intensité et le clip qui le définit. Pour une création plus complète, il faudra lui donner une direction, une intensité, les valeurs de `max_back`, `max_front`, `min_back` et `min_front`, une priorité par rapport aux autres sons joués simultanément [7], le fait que le son soit spatial ou non [8], sa description, son *pitch* [9],

7. 1.0 pour une priorité maximale

8. Un son spatial correspond à un point dans l'espace : le son sera plus ou moins audible selon la position du visiteur dans la scène.

9. 1.0 pour que le son soit joué à l'octave normale. Attention : le *pitch* ne peut pas avoir une valeur inférieure ou égale à 0.0.

et enfin le clip qui lui correspond.

```
new_sound : float -> clip -> vrml_object
new_complete_sound :
   vect3D -> float -> float -> float -> float -> float -> float -> bool
-> string -> float -> clip -> vrml_object
```

**Remarque :** les champs `max_back` et `max_front` définissent une ellipse après laquelle le son n'est plus audible (si l'on s'éloigne de sa source), et les champs `min_back` et `min_front` délimitent une ellipse après laquelle le son est à son maximum (si on s'approche encore de la source).

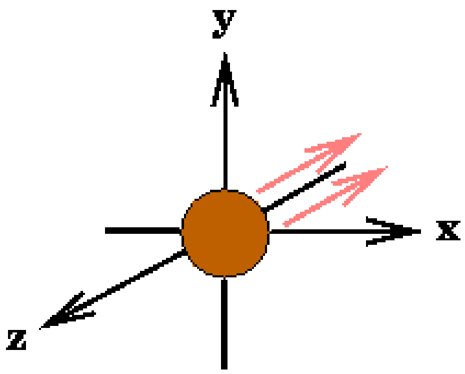
## 3.7 Points de vue

### 3.7.1 Point de vue et navigation

Il est possible de positionner un ou plusieurs points de vue dans la scène, avec éventuellement des propriétés de navigation différentes. Par exemple, on peut placer un *view point* au-dessus d'un objet et n'autoriser que le mode *walk* du *browser*, et un positionner autre point de vue devant un autre objet avec tous les modes de visite possibles (*walk*, *examine* et *fly*, ou éventuellement *none* ce qui implique que le visiteur ne peut pas se mouvoir dans la scène et devra donc se contenter des visites guidées par les animations).

### 3.7.2 Création et manipulation

Il faut savoir qu'un point de vue est par défaut placé de sorte que l'utilisateur regarde droit dans le sens inverse de l'axe des z, c'est-à-dire selon le schéma suivant :




**Type VRcaML :**

```
type viewPoint =
{ viewPoint_field_of_view     : float ;
  viewPoint_jump              : bool ;
  viewPoint_description       : string ;
  navigation_avatar_size      : float list ;
  navigation_headlight        : bool ;
  navigation_speed            : float ;
  navigation_type             : string list ;
  navigation_visibility_limit : float
} ;;
```

**Création :** pour créer un point de vue, il suffit de donner sa description, puis de le positionner dans l'espace comme tout `vrml_object`. Pour une création complète, il faut indiquer son champ de vision en radians[10], le fait que le visiteur puisse être transporté vers le point de vue de façon visible (animée) ou non, une description, la taille de l'*avatar*[11], le fait cet *avatar* possède ou non une lumière au front éclairant systématiquement les objets placés en face du visiteur, la vitesse de navigation et les types de navigation possibles (`WALK`, `EXAMINE`, `FLY`, `NONE`), et enfin la limite de visibilité du visiteur[12].

```
new_viewPoint : string -> vrml_object
new_complete_viewPoint :
   float -> bool -> string -> float list -> bool -> float -> string list
   -> float -> vrml_object
```

## 3.8 Zone de collision

**Type VRcaML :**

```
type collisionBox =
{ collide     : bool ;
  bbox_center : vect3D ;
  bbox_size   : vect3D
} ;;
```

**Création :** une zone de collision est définie par un centre et une taille, ce qui permet de définir des boîtes asymétriques par rapport au centre du référentiel de l'objet.

```
new_collision_box : vect3D -> vect3D -> vrml_object
```

**Remarque :** même remarque que pour le champ `collision` de `vrml_object`, c'est-à-dire que la détection de collision semble dépendre fortement de l'implémentation de chaque *browsers*. L'utilité d'une telle zone est donc apparemment discutable.

## 3.9 LOD

**A FAIRE !!!**

## 3.10 Objets spéciaux

### 3.10.1 Brouillard

**Type VRcaML :**

```
type fog =
{ fog_color            : color ;
  fog_type             : string ;
  fog_visibility_range : float
} ;;
```

10. valeur supérieure à 0.0 et inférieure à $\pi$ (la valeur par défaut correspond à un champ de vision à 45 degrés)

11. objet invisible représentant le visiteur dans la scène

12. 0.0 pour une visibilité infinie, autre valeur positive pour indiquer la distance à partir de laquelle le visiteur ne peut plus voir les objets.

**Création :** un brouillard est complètement défini par sa couleur, son type et la distance à laquelle les objets finissent par être masqués par le *fog*. Il existe deux types de brouillard :

- linéaire (`LINEAR`) : la visibilité diminue linéairement, ce qui donne plutôt une impression de profondeur ;
- exponentiel (`EXPONENTIAL`) : la visibilité diminue de façon exponentielle, ce qui donne un effet de brouillard plus réaliste.

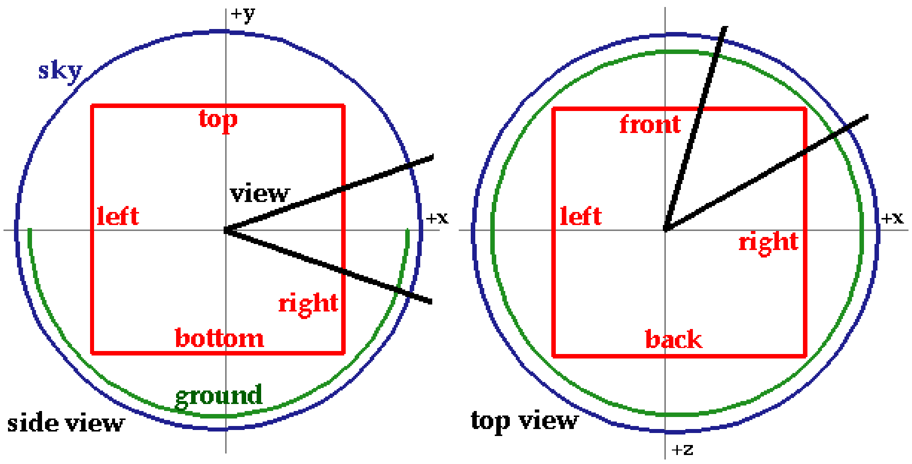
```
new_complete_fog : color -> string -> float -> vrml_special
new_linear_fog : color -> vrml_special
new_exponential_fog : color -> vrml_special
```

**Modification des champs :**

```
setf_fog_color : vrml_special -> color -> vrml_special
setf_fog_type : vrml_special -> string -> vrml_special
setf_fog_visibility_range : vrml_special -> float -> vrml_special
```

### 3.10.2 Fond

L'objet spécial *background* permet de simuler un panorama avec un ciel et un sol, l'ensemble étant placé selon le schéma suivant :




**Type VRcaML :**

```
type background =
{ ground_angle : float list ;
  ground_color : color list ;
  back_url     : string list ;
  bottom_url   : string list ;
  front_url    : string list ;
  left_url     : string list ;
  right_url    : string list ;
  top_url      : string list ;
  sky_angle    : float list ;
  sky_color    : color list
} ;;
```

**Création :** pour créer un fond, il suffit d'une URL, qui sera appliquée dans toutes les directions, et d'une couleur, qui sera affectée à la fois au sol et au ciel. Pour un fond plus complet, il faudra donner la liste des angles[13] correspondant aux cercles concentriques du sol jusqu'au du pôle sud, la liste des couleurs appliquées à ces cercles, les URLs de derrière, dessous, devant, gauche, droite, dessus, et enfin la liste des angles et la liste des couleurs correspondant aux cercles concentriques du ciel jusqu'au du pôle nord.

```
new_background : string -> color -> vrml_special
new_complete_background :
   float list -> color list -> string list -> string list -> string list
-> string list -> string list -> string list -> float list -> color list
-> vrml_special
```

**Remarque :** si le dernier angle du sol ou du ciel est strictement inférieur à $\pi$ (ce qui est souvent le cas), la couleur de la région comprise entre le dernier cercle et le pôle sera interpolée linéairement à partir de la liste des couleurs.

**Modification des champs :**

```
setf_background_ground_angles : vrml_special -> float list -> vrml_special
setf_background_ground_colors : vrml_special -> color list -> vrml_special
setf_background_back_urls : vrml_special -> string list -> vrml_special
setf_background_bottom_urls : vrml_special -> string list -> vrml_special
setf_background_front_urls : vrml_special -> string list -> vrml_special
setf_background_left_urls : vrml_special -> string list -> vrml_special
setf_background_right_urls : vrml_special -> string list -> vrml_special
setf_background_top_urls : vrml_special -> string list -> vrml_special
setf_background_sky_angles : vrml_special -> float list -> vrml_special
setf_background_sky_colors : vrml_special -> color list -> vrml_special
```

13. valeurs comprises entre 0.0 et $\pi$

# Chapitre 4

# Fonctions avancées

Plutôt que de passer par des créations complètes, ou afin de factoriser la modification d'un champ précis d'objets semblables, il est intéressant de pouvoir appliquer des fonctions de mise à jour des champs des types enregistrement de VRcaML à des `vrml_object` qui les masquent.

Il existe, pour pratiquement chaque champ de chaque *record* VRcaML, une fonction de modification. Elles ont toutes une signature du type :

```
vrml_positionable -> 'a -> vrml_positionable
```

Il existe d'autre part des fonctions permettant d'appliquer ce type de fonctions aux éléments des champs `objects` et `children` des `vrml_object` :

```
map_obj :
   vrml_object -> (vrml_positionable -> 'a -> vrml_positionable)
   -> 'a -> vrml_object
map_all :
   vrml_object -> (vrml_positionable -> 'a -> vrml_positionable)
   -> 'a -> vrml_object
```

Voici par exemple la fonction de bas niveau permettant de modifier la couleur d'un objet géométrique :

```
setf_diffuse_color : vrml_positionable -> color -> vrml_positionable
```

Et voici comment sont définies les fonctions `map_color` et `map_all_color` à partir de `setf_diffuse_color` :

```
let map_color object color =
    map_obj object setf_diffuse_color color
;;

let map_all_color object =
    map_all object setf_diffuse_color
;;
```

Voici enfin la liste de toutes les fonctions de type `setf_` utilisables avec `map_obj` et `map_all` :

```
setf_texture_repeatS : vrml_positionable -> bool -> vrml_positionable
```

```
setf_texture_repeatT : vrml_positionable -> bool -> vrml_positionable
setf_texture_type : vrml_positionable -> textureType -> vrml_positionable
setf_texture_center : vrml_positionable -> vect2D -> vrml_positionable
setf_texture_rotation : vrml_positionable -> float -> vrml_positionable
setf_texture_scale : vrml_positionable -> vect2D -> vrml_positionable
setf_texture_translation : vrml_positionable -> vect2D -> vrml_positionable
setf_texture_type_property :
   vrml_positionable -> (textureType -> 'a -> textureType) -> 'a
   -> vrml_positionable
setf_imageTexture_urls : vrml_positionable -> string list -> vrml_positionable
setf_pixelTexture_image : vrml_positionable -> image -> vrml_positionable
setf_movieTexture_speed : vrml_positionable -> float -> vrml_positionable
setf_movieTexture_clip : vrml_positionable -> clip -> vrml_positionable

setf_ambient_intensity : vrml_positionable -> float -> vrml_positionable
setf_diffuse_color : vrml_positionable -> color -> vrml_positionable
setf_emissive_color : vrml_positionable -> color -> vrml_positionable
setf_shininess : vrml_positionable -> float -> vrml_positionable
setf_specular_color : vrml_positionable -> color -> vrml_positionable
setf_transparency : vrml_positionable -> float -> vrml_positionable

setf_cone_radius : vrml_positionable -> float -> vrml_positionable
setf_cone_height : vrml_positionable -> float -> vrml_positionable
setf_cone_side : vrml_positionable -> bool -> vrml_positionable
setf_cone_bottom : vrml_positionable -> bool -> vrml_positionable

setf_cylinder_bottom : vrml_positionable -> bool -> vrml_positionable
setf_cylinder_height : vrml_positionable -> float -> vrml_positionable
setf_cylinder_radius : vrml_positionable -> float -> vrml_positionable
setf_cylinder_side : vrml_positionable -> bool -> vrml_positionable
setf_cylinder_top : vrml_positionable -> bool -> vrml_positionable

setf_text_content : vrml_positionable -> string list -> vrml_positionable
setf_text_font : vrml_positionable -> fontStyle -> vrml_positionable
setf_text_length : vrml_positionable -> float list -> vrml_positionable
setf_text_max_extent : vrml_positionable -> float -> vrml_positionable

setf_elevationGrid_colors :
   vrml_positionable -> color list -> vrml_positionable
setf_elevationGrid_texture_coordinate :
   vrml_positionable -> textureCoordinate -> vrml_positionable
setf_elevationGrid_height :
   vrml_positionable -> float list -> vrml_positionable
setf_elevationGrid_ccw : vrml_positionable -> bool -> vrml_positionable
setf_elevationGrid_color_per_vertex :
   vrml_positionable -> bool -> vrml_positionable
setf_elevationGrid_solid : vrml_positionable -> bool -> vrml_positionable
setf_elevationGrid_x_dimension : vrml_positionable -> int -> vrml_positionable
setf_elevationGrid_x_spacing : vrml_positionable -> float -> vrml_positionable
setf_elevationGrid_z_dimension : vrml_positionable -> int -> vrml_positionable
setf_elevationGrid_z_spacing : vrml_positionable -> float -> vrml_positionable

setf_extrusion_begin_cap : vrml_positionable -> bool -> vrml_positionable
setf_extrusion_ccw : vrml_positionable -> bool -> vrml_positionable
setf_extrusion_convex : vrml_positionable -> bool -> vrml_positionable
```

```
setf_extrusion_cross_section :
   vrml_positionable -> vect2D list -> vrml_positionable
setf_extrusion_end_cap : vrml_positionable -> bool -> vrml_positionable
setf_extrusion_orientation : vrml_positionable -> rotation -> vrml_positionable
setf_extrusion_scale : vrml_positionable -> vect2D -> vrml_positionable
setf_extrusion_solid : vrml_positionable -> bool -> vrml_positionable
setf_extrusion_spine : vrml_positionable -> vect3D list -> vrml_positionable

setf_faceSet_faces : vrml_positionable -> face list -> vrml_positionable
setf_faceSet_texture_coordinate :
   vrml_positionable -> textureCoordinate -> vrml_positionable
setf_faceSet_ccw : vrml_positionable -> bool -> vrml_positionable
setf_faceSet_convex : vrml_positionable -> bool -> vrml_positionable
setf_faceSet_solid : vrml_positionable -> bool -> vrml_positionable
setf_faceSet_texture_coordinate_index :
   vrml_positionable -> int list -> vrml_positionable

setf_light_color : vrml_positionable -> color -> vrml_positionable
setf_light_ambient_intensity : vrml_positionable -> float -> vrml_positionable
setf_light_intensity : vrml_positionable -> float -> vrml_positionable
setf_light_type : vrml_positionable -> lightType -> vrml_positionable
setf_light_type_property :
   vrml_positionable -> (lightType -> 'a -> lightType) -> 'a
   -> vrml_positionable
setf_directionalLight_direction :
   vrml_positionable -> vect3D -> vrml_positionable
setf_pointLight_attenuation : vrml_positionable -> vect3D -> vrml_positionable
setf_pointLight_radius : vrml_positionable -> float -> vrml_positionable
setf_spotLight_attenuation : vrml_positionable -> vect3D -> vrml_positionable
setf_spotLight_beam_width : vrml_positionable -> float -> vrml_positionable
setf_spotLight_cut_off_angle : vrml_positionable -> float -> vrml_positionable
setf_spotLight_direction : vrml_positionable -> vect3D -> vrml_positionable
setf_spotLight_radius : vrml_positionable -> float -> vrml_positionable

setf_sound_direction : vrml_positionable -> vect3D -> vrml_positionable
setf_sound_intensity : vrml_positionable -> float -> vrml_positionable
setf_sound_max_back : vrml_positionable -> float -> vrml_positionable
setf_sound_max_front : vrml_positionable -> float -> vrml_positionable
setf_sound_min_back : vrml_positionable -> float -> vrml_positionable
setf_sound_min_front : vrml_positionable -> float -> vrml_positionable
setf_sound_priority : vrml_positionable -> float -> vrml_positionable
setf_sound_spatialize : vrml_positionable -> bool -> vrml_positionable
setf_sound_description : vrml_positionable -> string -> vrml_positionable
setf_sound_pitch : vrml_positionable -> float -> vrml_positionable
setf_sound_clip : vrml_positionable -> clip -> vrml_positionable

setf_viewPoint_field_of_view : vrml_positionable -> float -> vrml_positionable
setf_viewPoint_jump : vrml_positionable -> bool -> vrml_positionable
setf_viewPoint_orientation : vrml_positionable -> rotation -> vrml_positionable
setf_viewPoint_description : vrml_positionable -> string -> vrml_positionable
setf_navigation_avatar_sizes :
   vrml_positionable -> float list -> vrml_positionable
setf_navigation_headlight : vrml_positionable -> bool -> vrml_positionable
setf_navigation_speed : vrml_positionable -> float -> vrml_positionable
setf_navigation_types : vrml_positionable -> string list -> vrml_positionable
```

```
setf_navigation_visibility_limit :
   vrml_positionable -> float -> vrml_positionable

setf_lod_levels : vrml_positionable -> vrml_object list -> vrml_positionable
setf_lod_ranges : vrml_positionable -> float list -> vrml_positionable
```

# Chapitre 5

# Les animations

**Qu'est ce qu'une animation ?**

Du point de vue de VRcaML, une animation se définit comme le changement d'état d'un objet s'effectuant au cours du temps. Une animation est donc la description des différents états que l'objet va traverser et des instants auxquels il va les traverser. La définition réclame donc la description de ce changement d'état et de son placement dans le temps.

## 5.1 Comment décrire le changement d'état : le type changing

```
type changing =
     Translation of (float list * vect3D list)
   | Rotation of (float list * rotation list)
   | Scale of (float list * vect3D list)
;;
```

Dans un but de simplification et pour une première approche, les changements d'état d'un `vrml_object` ne se font qu'au niveau de ses champs position et orientation. Ces changements (*changings*) se décrivent tous deux comme une liste de couples instants clés * valeurs clés. Les instants clés sont compris entre 0.0 (début du cycle d'animation) et 1.0 (fin du cycle). La chronologie qu'ils décrivent est donc relative et indépendante de la durée réelle de l'animation (qui sera spécifiée plus tard).

À chaque instant clé, une valeur clé fait correspondre la valeur du champ (position ou orientation) qui sera transmise à l'objet. Il est à noter que cette transmission écrasera définitivement la valeur précédente du champ en question. Ceci fait que, de deux `changing` de même type s'appliquant à un même objet au même instant t, l'un des deux l'emportera sur l'autre (de manière aléatoire et arbitraire). De plus, VRML interpolera (linéairement) tous les couples instant * valeur situées entre les points clés spécifiés par le `changing`. Ainsi pour déplacer un objet d'un point à un autre en ligne droite il suffit juste de donner deux valeurs clés, le point de départ et le point d'arrivée.

**Exemple :**

```
Translation ( [ 0.0 ; 0.5 ; 1.0 ] ;
              [ ( 0. , 0. , 0. ) ; ( 0. , 2. , 0. ) ; ( 0. , 0. , 0. ) ] )
```

Au début de l'animation l'objet se situe à l'origine du référentiel où il est placé, puis il va monter le long de l'axe des y jusqu'à atteindre la position y = 2 à la moitié de l'animation. Une fois ce point atteint, il va redescendre jusqu'à sa position initiale qu'il atteindra à la fin de l'animation.

## 5.2 Placement l'animation dans le temps : le type `trigger`

```
type trigger =
     Instant of time
   | Click of vrml_object
   | Enter of (proximityBox * vrml_object)
;;
```

Un *trigger*, ou contrôleur, permet de déclencher et de stopper une animation. Il existe trois familles de `triggers` :

- *instant trigger* : ce *trigger* fonctionne comme un réveil matin, on le règle à une certaine heure (instant t après le début du monde , *i.e* son chargement), et arrivé à cette heure il se déclenche démarrant ou stoppant l'animation qu'il contrôle.
- *click trigger* : ce *trigger* se déclenche lorsque l'utilisateur clique avec son pointeur souris sur l'objet auquel le *trigger* est attaché.
- *proximity trigger* : il y a deux types de *proximity trigger* selon qu'il se déclenche à l'entrée ou à la sortie de la `proximityBox` à laquelle l'objet est attaché.

## 5.3 Le type `animation`

```
type animation =
{ target        : vrml_object ;
  start_trigger : trigger list ;
  stop_trigger  : trigger list ;
  loop          : bool ;
  duration      : float ;
  memory        : bool ;
  movie         : changing list
} ;;
```

Le type `animation` permet de lier ensemble des `changing`, des `trigger` et le `vrml_object` sur lequel l'animation qu'ils décrivent s'appliquera. Le champ `target` désigne l'objet sur lequel l'animation va s'appliquer. Les champs `start_trigger` et `stop_trigger` donnent la liste des *triggers* qui vont respectivement déclencher et arrêter l'animation.

On peut souhaiter qu'une animation boucle sur elle-même jusqu'à ce qu'un des ses `stop_trigger` se déclenche ; le booléen `loop` permet de lui donner cette propriété.

Le champ `duration` spécifie la durée de l'animation ; en effet, si le champ `loop` est à `false`, une animation s'arrêtera d'elle-même une fois un temps égal à sa durée écoulé depuis son début. Et ce même si l'on a affecté un *trigger* au contrôle de son arrêt (bien sûr, si jamais ce *trigger* devait être activé avant qu'un temps `duration` ne soit passé, l'animation s'arrêtera).

Une fois l'animation arrêtée (suite à l'activation d'un de ses *triggers*), et donc l'objet cible figé dans un des états décrits par cette animation, il peut être intéressant que l'animation reprenne, en cas de réactivation, depuis cet état et non depuis l'état initial normal de l'animation (ce qui est le cas par défaut). Le champ `memory` spécifie si l'objet se souviendra de son dernier état au redémarrage de l'animation.

Enfin, le champ `movie` est la liste des `changing` s'appliquant à l'objet au cours de l'animation.

**Remarque :** une fois une animation declenchée, l'activation d'un de ses `start_triggers` n'a aucun effet sur son cours (l'animation ne redémarrera pas).

## 5.4 Les fonctions de base de manipulation des animations

`startOn : animation -> trigger -> animation`

permet de spécifier que le *trigger* donné en argument déclenchera l'animation.

`stopOn : animation -> trigger -> animation`

permet de spécifier que le *trigger* donné en argument arrêtera l'animation.

`initialize : vrml_object -> changing list -> vrml_object`

retourne une instance du `vrml_object` avec des champs `rotation` et `position` correctement initialisés sur les premières valeurs clés des `changing` données en argument.

`create_animation : vrml_object -> changing list -> float -> animation`

retourne une instance du type `animation` avec les valeurs des champs `target`, `movie` et `duration` égales aux valeurs passées en argument.

`add_animation : vrml_world -> animation -> vrml_world`

permet d'ajouter une animation à une scène VRcaML.

# Chapitre 6

# Les fonctions trajectoires

Les fonctions présentées au précédent chapitre permettent de créer à la main des animations, mais l'on aimerait bien avoir des mécanismes plus puissants pour pouvoir engendrer des trajectoires complexes, et pouvoir combiner ces trajectoires pour en créer de nouvelles.

## 6.1 Le type trajectoire

Il a semblé que la façon la plus simple et la plus fonctionnelle de représenter une trajectoire était d'en faire une fonction. En effet, étant donné qu'en VRcaML les trajectoires sont exprimées indépendamment de leur vitesse, et qu'elles consistent en une liste d'association instant clé * valeur clé, il est naturel de fonctionnaliser cette liste d'associations en créant une fonction qui, appliquée à toute fraction (ou instant, *i.e* un flottant compris entre 0.0 et 1.0), donnerait la valeur clé correspondante.

Comme de plus les trajectoires sont en VRML approximées sous forme de valeurs clés entre lesquelles on interpole les autres valeurs, on peut régler la qualité d'approximation en appliquant une fonction prenant en paramètre le grain souhaité et la trajectoire fonctionnalisée et qui fabrique le `changing` correspondant.

```
le type trajectoire : float -> vect3D ou float -> rotation
```

En fait, le type trajectoire n'est pas un type défini réellement dans la bibliothèque VRcaML car c'est un type de fonction, et définir explicitement un type fonctionnel en CAML n'a pas vraiment d'intérêt puisque le mécanisme d'inférence en fait la vérification à notre place.

## 6.2 Quelques exemples de trajectoires fonctionnelles

Tout d'abord des trajectoires de type **translation** :

```
(* trajectoire circulaire de rayon radius, d'axe y avec y = 0 *)
let circle_y radius fraction =
    (radius *. cos (2. *. pi *. fraction), 0. ,
     radius *. sin (2. *. pi *. fraction)) ;;
```

Il est assez facile de cette manière de dériver une trajectoire d'une autre :

```
let spirale radius number h fraction =
```

```
match circle_y radius (fraction *. float_of_int number) with
(x,y,z) -> (x, h *. fraction ,z) ;;
```

De manière logique, l'on voit que `fraction` est le dernier argument des ces nouvelles fonctions trajectoires et que l'on donne les paramètres spécifiant la trajectoire avant celui-ci (par exemple `radius` pour la fonction `circle`). De la même manière pour des trajectoires de type **rotation** (*cf.* ci-dessus) :

```
(* définit une rotation de 360 degrés sur l'axe des y *)
let rotation_y fraction = ( (0.,1.,0.) , fraction *. 2.0 *. pi ) ;;
```

## 6.3 Manipulation des trajectoires fonctionnelles

À partir d'une trajectoire, on engendre un `changing` en donnant le grain (nombre d'étapes intermédiaires désirées) :

```
makeChanging : (float -> 'a) -> int -> (float list * 'a list)
```

Tout l'intérêt de ces trajectoires fonctionnelles est de pouvoir les combiner entre elles très simplement pour pouvoir en créer de nouvelles :

```
reverse_trajectory : (float -> 'a) -> float -> 'a
let reverse_trajectory trajectory fraction = trajectory (1.0 *. fraction );;
```

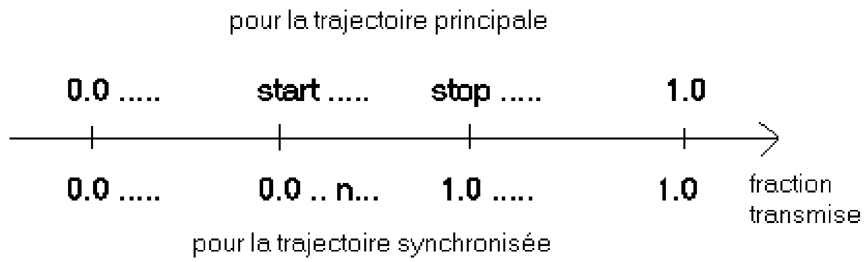
permet d'inverser le sens d'une trajectoire.

```
synchronize_trajectory : float -> float -> ( float -> 'a) -> float -> 'a
```

synchronise une animation pour qu'elle s'écoule relativement à une autre. On peut souhaiter qu'une trajectoire s'applique à un objet pendant une partie du temps d'exécution d'une autre trajectoire. Or une trajectoire n'est *a priori* pas sécable[1], une solution est de recalculer la trajectoire qui doit être exécutée à l'intérieur de l'autre :

- jusqu'au temps `start` : la nouvelle trajectoire transmettra sa valeur initiale (`fraction = 0.0`) ;
- à partir du temps `start` et jusqu'au temps `stop` : elle s'exécutera normalement ;
- et à partir du temps `stop` : elle transmettra sa valeur finale (`fraction = 1.0`).

1. On peut le faire en la divisant à la main en deux animations mais ce n'est pas très élégant




```
repeat_trajectory : int -> (float -> 'a) -> float -> 'a
```

retourne une trajectoire qui est égale au bouclage n fois de la trajectoire donnée en argument.

## 6.4 De l'utilisation des trajectoires pour engendrer des `vrml_object`

On peut en effet réutiliser les trajectoires pour engendrer certains champs des objets VRcaML dits complexes, et plus précisément les *face sets* et les extrusions.

### 6.4.1 Pour les *face sets* :

Une facette étant de type `vect3D list`, il semble facile d'en engendrer avec des trajectoires fonctionnelles. Par exemple à partir d'une trajectoire circulaire, on peut fabriquer facilement des polygones réguliers à n faces du plan.

```
compute_values : int -> (float -> 'a) -> 'a list
```

Cette fonction permet d'engendrer une liste correspondant aux valeurs retournées par la trajectoire pour une application avec une précision passée en paramètre.

### 6.4.2 Pour les Extrusions :

C'est avec les extrusions que les trajectoires prennent tout leur intérêt, car elles permettent d'engendrer à la fois des *shapes* 2D et l'épine dorsale sur laquelle elles seront extrudées.

```
compute_2DShape_x  : ( float -> 'a * 'b * 'c ) -> int -> ( 'a * 'b ) list
compute_2DShape_y  : ( float -> 'a * 'b * 'c ) -> int -> ( 'a * 'b ) list
compute_2DShape_z  : ( float -> 'a * 'b * 'c ) -> int -> ( 'a * 'b ) list
```

Ces fonctions engendrent une liste égale aux valeurs retournées par la trajectoire (de translation) pour une application avec une précision passée en paramètre. Ces valeurs appartiennent à un plan perpendiculaire à un des vecteurs de la base `(i,j,k)`. Ces fonctions permettent donc à partir d'une trajectoire de fabriquer une forme en deux dimensions que l'on pourra affecter au champ `extrusion_cross_section`.

### Engendrer une épine dorsale

Une épine dorsale étant de type `vect3D list` l'application de la fonction `compute_values` suffit à en engendrer une.

### Engendrer une liste d'orientation

De la même manière que pour fabriquer une épine dorsale la fonction `compute_values` suffit.

### Un exemple d'application : fabriquer un ressort

On ne dispose à la base que de la trajectoire de type :

```
circle_y : float -> fraction -> vect3D
(* cercle situé dans le plan Ox,Oz et de centre O *)
```

On veut que le ressort ait n spires donc on va commencer par engendrer une trajectoire circulaire qui boucle n fois à partir de circle :

```
let circle_n r n = repeat_trajectory n (circle_y r);;
```

Le ressort doit avoir une hauteur h donc il faut filtrer la valeur de y de `circle_n` pour la recalculer :

```
let ressort r n h fraction =
match circle_n r n fraction with (x,y,z) -> (x,fraction *. h,z) ;;
```

Et voilà : on a déja l'épine dorsale de prête, il reste à fabriquer la liste d'orientations correspondante[2] :

```
let revolution_orientation n precision fraction =
if (fraction = 0.0 ) then ((0.,0.,0.),0.) else ((0.,1.,0.) ,
(float_of_int n) *. 2.0 *. pi /. float_of_int(precision)) ;;
```

On a dès lors plus qu'à fabriquer effectivement les listes de valeurs avec `compute_values` :

```
let spine = compute_values 50 (ressort 4.0 5 10.0) ;;
let orientation = compute_values 50 (revolution_orientation 5 50) ;;
```

Il nous manque la coupe du fil du ressort et de quoi maintenir l'échelle de cette coupe :

```
let identity fraction = (1.,1.,1.) ;;
let scale = compute_2DShape_y 50 identity ;;
let coupe = compute_2DShape_y 15 (circle_y 1.5) ;;
```

Il suffit alors d'engendrer le ressort lui-même et le tour est joué :

```
let mon_ressort = new_extrusion_bis coupe spine orientation scale ;;
```

2. Attention cependant car dans une extrusion les rotations s'appliquent à la *shape* 2D courante.

# Chapitre 7

# Traduction en VRML

## 7.1 Créer des `vrml_world`

Pour pouvoir traduire une scène en VRML, il faut auparavant avoir empaqueté son arbre d'objets, ses objets spéciaux et ses animations dans une variable de type `vrml_world` :

```
new_vrml_world : string list -> string -> vrml_object ->
                 vrml_special list -> animation list -> vrml_world
set_info : vrml_world -> string list -> vrml_world
set_title : vrml_world -> string -> vrml_world
set_tree : vrml_world -> vrml_object -> vrml_world
set_specials : vrml_world -> vrml_special list -> vrml_world
add_special : vrml_world -> vrml_special -> vrml_world
set_animations : vrml_world -> animation list -> vrml_world
add_animation : vrml_world -> animation -> vrml_world
```

## 7.2 De la difficulté de créer des mondes VRcaML cohérents

Il y a quelques précautions à prendre lorsque l'on construit des mondes 3D avec VRcaML. Car l'utilisation de l'égalité physique ou de l'égalité structurelle pour identifier les objets lors de la phase de traduction peut aboutir à des fichiers `.wrl` incohérents, voire à déclencher la levée d'une exception lors de l'application du traducteur.

Il faut bien engendrer les animations en dernier ou lorsque l'on est sûr de ne pas retoucher aux objets sur lesquels elles s'appliquent. En effet, une différence entre les instances de l'objet situées dans le champ `target` de l'animation et le champ `tree` du `vrml_world` engendreront des incohérences sur le nommage (fait lors de la traduction) du-dit objet.

Ce risque devient vite assez grand lorsque l'on place des objets les uns dans les autres ou lorsque l'on initialise un objet sur les premières valeurs des changing que l'on va lui appliquer. Dans ce dernier cas, l'ordre à suivre est le suivant : créer l'objet, créer les `changing`, initialiser l'objet en fonction des `changing` (fonction `initialize`), puis créer l'animation avec les `changing` et la nouvelle valeur de l'objet, ensuite ne plus appliquer de changement d'état à l'objet et si l'on fait référence à l'objet (pour par exemple le placer dans un autre objet), il faudra bien se servir de l'instance de l'objet obtenue suite à l'initialisation.

## 7.3 Les fonctions de traduction en VRML

```
vrml_world_translate : vrml_world -> string = <fun>
vrml_translate_to : vrml_world -> string -> unit = <fun>
vrml_translate_to_screen : vrml_world -> unit = <fun>
```

## 7.4 Les exceptions levées lors de la traduction

`Failure "Unequals fields scale_list, rotation_list, translation_list"`

Cette exception se lève lorsque les listes contenues dans les champs `rotation_list`, `scale_list` et `translation_list` d'un objet de type `extrusion` ne sont pas de même longueur. Elle est levée au niveau de la fonction de calcul de la frame de l'objet.

`Not_found`

Cette exception, qui fait partie des exceptions standards de Caml, est levée lorsque une incohérence survient entre des instances d'un même objet. Comme il a été dit précédemment, une mauvaise utilisation des fonctions de manipulation des objets peut faire coexister dans un même monde VRcaML des instances non structurellement égales d'un `vrml_object`, ce qui fausse le mécanisme de nommage des objets et lève cette exception.

**Incohérence dans le fichier `.wrl`**

Ce genre d'incohérence est créée de la même manière que l'exception `Not_found` est levée. Un objet peut se retrouver avec un nom égal à la chaîne de caractères vide, ce qui engendre le plantage du *browser* VRML.

# Annexe A

# Exemple du Jussieu Virtuel

## A.1 Le fichier VRcaML

```
(* Jussieu virtuel : un exemple d'utilisation des placements relatifs *)

#open "vrcaml" ;;

let rotonde_cyl =
    translation (map_color (new_cylinder 1.0 5.0) green)
                0.0 (-0.4) 0.0 ;;

let rotonde_box =
    map_color (new_box 2.0 4.0 2.0) gray ;;

let rotonde = group [ rotonde_cyl; rotonde_box ] ;;

let barre1 = map_color (new_box 6.0 4.0 2.0) gray_25 ;;
let barre2 = set_rotation barre1 0.0 1.0 0.0 (pi /. 2.0) ;;

let tour_centrale =
    set_rotation (map_color (new_box 3.0 10.0 3.0) gray_40)
                     0.0 1.0 0.0 (pi /. 4.0) ;;

let sol = map_color (new_box 70.0 0.5 50.0) gray_10 ;;


let b1 = place_left rotonde barre1;;
let b2 = place_right rotonde barre1;;

let section1 = group [ barre1; b1; b2 ] ;;

let b3 = place_behind barre2 rotonde;;
let section2 =
    add_children barre1 [ (add_child b1 b3); (add_child b2 b3) ] ;;

let section3 =
    stick_behind section1
                 (stick_behind section2
                    (stick_front section2
                       (stick_front section2 section2))) ;;
```

```
let section4 = stick_right rotonde (stick_right barre1 section1) ;;

let triangle = stick_left barre1 (stick_behind barre2 rotonde) ;;
let section5 = stick_right triangle section2 ;;

let section6 =
    stick_behind section4
                 (stick_behind section5
                    (stick_front section5
                       (stick_front section5 section5))) ;;

let section7 = stick_left barre1 (stick_right barre1 rotonde) ;;

let section8 = stick_behind barre2 section7 ;;

let section9 =
    stick_behind section7 (stick_behind section8 section8) ;;

let ensemble = stick_left section3 (stick_right section6 section9) ;;

let tour_centrale =
    translation (place_front (place_top tour_centrale sol) ensemble)
                0.0 0.0 (-8.0) ;;

let jussieu = stick_top ensemble sol ;;

let v =
    set_rotation
      (translation
         (place_top (new_viewPoint "Au-dessus du Jussieu virtuel") jussieu)
         0.0 50.0 0.0)
       1.0 0.0 0.0 ((3.0 *. pi) /. 2.0) ;;

let jussieu = group [ jussieu; v; tour_centrale ] ;;

let w = new_vrml_world ["Copyright JBL&NR - GLA97"]
                       "Jussieu virtuel..." jussieu [] [] ;;

vrml_translate_to w "VRML/jussieu.wrl" ;;
```

## A.2 Le fichier VRML engendré

```
#VRML V2.0 utf8
DEF Object_1_Referential Transform {
children [
DEF Object_2_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.4 0.4 0.4 }}
geometry Box { size 3.0 10.0 3.0  }} ]
translation 0.0 5.25 11.1213203436
rotation 0.0 1.0 0.0   0.785398163397 }
DEF Object_3_Referential Transform {
children [
Viewpoint {description "Au-dessus du Jussieu virtuel" } ]
```

```
translation 0.0 55.25 0.0
rotation 1.0 0.0 0.0   4.71238898038 }
DEF Object_4_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.101961 0.101961 0.101961 }}
geometry Box { size 70.0 0.5 50.0  }}
DEF Object_5_Referential Transform {
children [
DEF Object_6_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }}
DEF Object_7_Referential Transform {
children [
DEF Object_8_Referential Transform {
children [
DEF Object_9_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.745098 0.745098 0.745098 }}
geometry Box { size 2.0 4.0 2.0  }} ]}
DEF Object_10_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.0 1.0 0.0 }}
geometry Cylinder { height 5.0 }}]
translation 0.0 -0.4 0.0 }]
translation 4.0 0.0 0.0 }
DEF Object_11_Referential Transform {
children [
USE Object_9_Referential
USE Object_10_Referential ]
translation -4.0 0.0 0.0 }
DEF Object_12_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }} ]}]
translation 0.0 0.0 -16.0 }
DEF Object_13_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }}
DEF Object_14_Referential Transform {
children [
DEF Object_15_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }} ]
translation 0.0 0.0 -4.0
rotation 0.0 1.0 0.0   1.57079632679 }
```

```
USE Object_9_Referential
USE Object_10_Referential ]
translation 4.0 0.0 0.0 }
DEF Object_16_Referential Transform {
children [
USE Object_15_Referential
USE Object_9_Referential
USE Object_10_Referential ]
translation -4.0 0.0 0.0 }]
translation 0.0 0.0 -8.0 }
DEF Object_17_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }}
USE Object_14_Referential
USE Object_16_Referential ]
translation 0.0 0.0 16.0 }
DEF Object_18_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }}
USE Object_14_Referential
USE Object_16_Referential ]
translation 0.0 0.0 8.0 }
USE Object_14_Referential
USE Object_16_Referential ]
translation -12.0 0.0 0.0 }
DEF Object_19_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }}
DEF Object_20_Referential Transform {
children [
DEF Object_21_Referential Transform {
children [
USE Object_9_Referential
USE Object_10_Referential ]
translation 12.0 0.0 0.0 }
DEF Object_22_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }} ]
translation 8.0 0.0 0.0 }
USE Object_8_Referential
USE Object_11_Referential
USE Object_12_Referential ]
translation 0.0 0.0 -16.0 }
DEF Object_23_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
```

```
geometry Box { size 6.0 4.0 2.0  }}
DEF Object_24_Referential Transform {
children [
DEF Object_25_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }} ]
translation -4.0 0.0 0.0 }
USE Object_15_Referential
USE Object_9_Referential
USE Object_10_Referential ]
translation 12.0 0.0 0.0 }
USE Object_14_Referential
USE Object_16_Referential ]
translation 0.0 0.0 -8.0 }
DEF Object_26_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }}
USE Object_24_Referential
USE Object_14_Referential
USE Object_16_Referential ]
translation 0.0 0.0 16.0 }
DEF Object_27_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }}
USE Object_24_Referential
USE Object_14_Referential
USE Object_16_Referential ]
translation 0.0 0.0 8.0 }
USE Object_24_Referential
USE Object_14_Referential
USE Object_16_Referential ]
translation 12.0 0.0 0.0 }
DEF Object_28_Referential Transform {
children [
USE Object_25_Referential
DEF Object_29_Referential Transform {
children [
Shape {appearance Appearance {
material Material { diffuseColor 0.25098 0.25098 0.25098 }}
geometry Box { size 6.0 4.0 2.0  }} ]
translation 4.0 0.0 0.0 }
USE Object_9_Referential
USE Object_10_Referential ]
translation 0.0 0.0 -16.0 }
DEF Object_30_Referential Transform {
children [
USE Object_15_Referential
USE Object_25_Referential
USE Object_29_Referential
```

```
USE Object_9_Referential
USE Object_10_Referential ]
translation 0.0 0.0 -8.0 }
USE Object_15_Referential
USE Object_25_Referential
USE Object_29_Referential
USE Object_9_Referential
USE Object_10_Referential ]
translation 0.0 3.15 0.0 } ] }]}
```

# Bibliographie

[1] http://vrml.sgi.com/moving worlds/. The vitual reality modeling language specification, version 2.0. Technical report, VRML Org., August 1996.

[2] Xavier Leroy. Le système caml special light : modules et compilation efficace en caml. Technical report, INRIA, Novembre 1995.