Simulateur Robotique fonctionnel

Rapport de Magistère
Université Joseph Fourier
23 septembre 1999

GUILLOUD Cyril
SAN SEVERINO Jean Charles

responsables:
SEKHAVAT Sepanta, PISSARD-GIBOLLET Roger et GIRAULT Alain

Introduction

L'idée directrice de ce stage est la création d'un simulateur robotique fonctionnel pour l'équipe SHARP. Il s'agit de modéliser le fonctionnement d'un robot piloté par un programme de contrôle-commande.

Les bases de ce simulateur ont été posées en collaboration avec les équipes BIP, SHARP et les MOYENS ROBOTIQUES.

Le sujet du stage est issu de la demande par l'équipe SHARP d'un simulateur pour expérimenter les programmes de contrôle-commande du robot Cycab, un véhicule électrique. L'équipe SHARP l'utilise pour développer des procédés de planification de mouvement (conduite automatique ou assistance à la conduite).

La première partie de ce stage (octobre 1998 à mai 1999) a été consacrée à l'identification des attentes des différents utilisateurs potentiels et à la recherche d'outils et de techniques nécessaires à la réalisation du simulateur. Cette phase s'est déroulée au sein de l'équipe SHARP en collaboration avec le projet BIP et les MOYENS ROBOTIQUES de l'INRIA. Nous avons été encadrés par trois tuteurs appartenant à ces équipes; respectivement Sepanta Sekhavat, Alain Girault et Roger Pissard-Gibollet.

Durant la deuxième partie (de juin 1999 à septembre 1999) nous avons réalisé le programme du simulateur.

Nous allons tout d'abord présenter les outils utilisés par les équipes concernées par le simulateur. Puis nous expliciterons quelques techniques et outils que nous avons utilisés pour créer ce simulateur. Ensuite nous détaillerons les caractéristiques du simulateur. Nous finirons par discuter des fonctions et évolutions possible du simulateur.

Remerciements

Nous tenons à remercier les trois personnes qui nous ont co-encadré avec beaucoup de patience et de bonne volonté (et il leur en a fallu...) : Sepanta SEKHAVAT, Roger PISSARD-GIBOLLET et Alain GIRAULT. Sans oublier tous les membres des équipes SHARP, BIP et spécialement des MOYENS ROBOTIQUES ainsi que les utilisateurs de la halle robotique de l'INRIA Rhône-Alpes.

Chapitre 1

Outils et techniques

Nous détaillons ici quelques outils et techniques que nous avons utilisés et qu'il est nécessaire d'expliquer pour comprendre la manière dont nous avons programmé le simulateur.

1.1 Bibliothèque OpenGL

L'utilisation fonctionnelle d'un simulateur est conditionnée par différents critères :

- Affichage rapide et agréable des résultats.
- Souplesse et simplicité d'utilisation.
- Disponibilité du programme.

L'utilisation de la bibliothèque graphique OpenGL est un moyen de satisfaire certains de ces objectifs.

1.1.1 Rapidité et visualisation

OpenGL est un ensemble de fonctions d'affichage d'objets 3D. Ces fonctions ont l'avantage de tirer parti des cartes graphiques accélératrices 3D disponibles sur de nombreuses machines. Les fonctions utilisées sont, pour les plus utilisées, codées au niveau matériel sur la carte. Les opérations souvent exécutées pour l'affichage 3D sont donc réalisées extrêmement rapidement par des circuits spécialisés.

La réalisation des calculs d'affichage par des circuits spécialisés permet de gérer des effets de lumière qui facilitent la perception de la scène par rapport à une vue en fil de fer par exemple.

Une caractéristique fondamentale d'OpenGL est l'existence de certaines fonctions qui facilitent considérablement la modélisation des capteurs. Par exemple, il est très facile de créer une nouvelle fenêtre représentant une vue

depuis une caméra dont on aura spécifié les coordonnées et l'orientation. C'est de cette manière que nous représentons les capteurs de type caméra.

Encore une fois l'utilisation de ces fonctions est nettement plus efficace en temps de calcul que la re-définition de nos propres fonctions.

1.1.2 Ouverture

Un aspect d'OpenGL qui nous intéressent également est son « ouverture » : il existe des versions gratuites d'OpenGL (notamment Mesa) et dont les codes sources sont disponibles.

Cette ouverture a plusieurs conséquences :

- Cela a permis un fort développement de ce standard dans le monde de la 3D, de nombreuses cartes accélératrices sont disponibles sur le marché à des prix sans cesse décroissants (dans le monde PC, cette baisse est due essentiellement aux jeux vidéo). Cet aspect est très important : cela signifie que le simulateur sera utilisable sur des machines très diverses et non pas uniquement sur des stations de travail hors de prix.

- Le simulateur peut être utilisé et développé gratuitement ce qui facilite grandement sa diffusion et sa maintenance.

- OpenGL est une bibliothèque très développée et bien documentée, ce qui est indéniablement un atout majeur.

Nous avons également utilisé la bibliothèque Glut, qui est une surcouche fonctionelle à OpenGL, pour la gestion des évènements d'interaction avec l'utilisateur (souris, clavier) et pour le multi-fenêtrage. Glut est une bibliothèque spécifique à chaque système de fenêtrage (X11, Windows, irix, etc.) ce qui permet de concevoir des applications quasi indépendantes (au niveau du code source évidement) du système d'exploitation cible.

Pour plus de détails sur OpenGL et Glut, consulter : [WND99], [rKF99], [J.K96].

1.2 Coordonnées homogènes

Les coordonnées homogènes sont un moyen très pratique et très utilisé pour représenter des objets dans l'espace.

La représentation d'un point se fait par un vecteur colonne à quatre composantes. Les trois premières sont les coordonnées cartésiennes du point tandis que la quatrième est considérée comme un facteur d'échelle égal à 1 en robotique.

$$P_1 = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Une direction (un vecteur) est représentée par quatre composantes dont la quatrième est nulle.

$$\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

En robotique (ainsi qu'en mécanique), on associe à tout élément un ou plusieurs repères. Un changement de repère permet d'exprimer des déplacements relatifs à différents éléments d'un mécanisme articulé. Il faut donc pouvoir passer d'un repère à un autre de manière très simple et efficace en temps de calcul.

Pour faire subir une transformation quelconque (translation + rotation) à un repère R_i qui l'amène sur un repère R_j, on lui applique une matrice ${}^iT_j$ de dimension 4×4, dite de transformation homogène (voir figure 1.1).

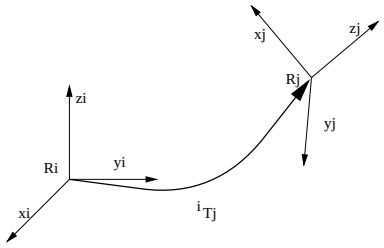

FIG. 1.1 – *Transformation de repère*

$${}^iT_j = \begin{bmatrix} s_x & n_x & a_x & P_x \\ s_y & n_y & a_y & P_y \\ s_z & n_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} {}^is_j & {}^in_j & {}^ia_j & {}^iP_j \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

où ${}^is_j$, ${}^in_j$ et ${}^ia_j$ désignent respectivement les vecteurs unitaires suivant les axes x_j, y_j et z_j du repère R_j exprimés dans le repère R_i et où ${}^iP_j$ est l'origine du repère R_j exprimé dans le repère R_i.

Pour plus de clarté, on représente en général la matrice ${}^iT_j$ sous la forme d'une matrice par blocs :

$${}^iT_j = \begin{bmatrix} & {}^iA_j & & {}^iP_j \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

où ${}^iA_j$ donne la rotation et ${}^iP_j$ la translation voulue.

On voit donc qu'une matrice 4×4 permet de représenter toutes les transformations que nous désirons. Il reste par contre à donner la matrice en fonction des déplacements voulus. Nous utilisons pour cela des fonctions OpenGL de transformation et de positionnement des vues (`gluPerspective`, `gluMultMatrix`, `glRotate*` par exemple). Ces fonctions prennent en paramètres des angles de rotation ou des valeurs de translation et mettent à jour la matrice homogène de transformation correspondante.

Notre simulateur prend donc des valeurs de rotation ou de translation. Celles-ci sont soit des paramètres donnés par l'utilisateur, soit des valeurs retournées par la modélisation physique. Nous codons ensuite ces transformations en matrices homogènes qui sont manipulées par OpenGL pour déplacer les objets simulés.

Une alternative aux coordonnées homogènes est l'utilisation des quaternions. Cette dernière méthode présente des avantages en temps de calcul et en qualité de la simulation. Cependant, les mouvements que nous simulons étant limités à des rotations et une translation, l'utilisation des coordonnées homogènes est suffisante dans notre projet.

1.3 Modèle physique

Notre approche modulaire nous a conduit à ne pas définir de modèle physique à l'intérieur du simulateur. C'est à l'utilisateur de définir lui-même le modèle qu'il veut utiliser. Ceci pour deux raisons : on ne peut pas définir tous les types de modèles existants et un utilisateur désirera toujours disposer de caractéristiques que nous n'avons pas prévues.

L'utilisateur fournira un modèle physique en donnant pour chaque liaison de sa chaîne cinématique des équations auxquelles obéissent les objets définis.

Par exemple dans le cas d'un bras manipulateur à six degrés de liberté (par exemple le Staubli Rx90), chaque articulation est (sauf dans des positions remarquables) soumise à un couple dû à la gravité. Il faut compenser ce couple par une commande du moteur correspondant pour maintenir le robot immobile.

L'utilisateur donnera les informations liées à la gravité dans le modèle physique et les informations liées aux commandes dans son propre programme de contrôle-commande.

Le modèle fourni par l'utilisateur est un fichier de fonctions. Ce fichier est lié au programme et après re-compilation, le simulateur utilise le nouveau modèle.

Le modèle cinématique du Cycab est fourni en annexe C.

Chapitre 2

Environnement du simulateur

Nous exposons dans ce chapitre les programmes et les robots avec lesquels le simulateur fonctionnera.

2.1 ORCCAD

ORCCAD est un environnement logiciel développé par l'INRIA. Il offre une aide à la création de programmes de contrôle-commande de manière graphique. ORCCAD propose des modules de spécification, de compilation et de vérification formelle de ces programmes.

Notre simulateur doit être utilisable en premier lieu avec des programmes de contrôle-commande créés à l'aide d'ORCCAD. Celui-ci génère indifféremment un exécutable destiné aux robots ou un code source en C. Nous devions donc programmer notre simulateur de manière à pouvoir utiliser ces sources. Nous avons donc réalisé une bibliothèque C++. Il est ainsi tout à fait possible de l'utiliser à partir de n'importe quel programme C/C++.

2.2 SIMPARC

SIMPARC est un simulateur pour programmes de contrôle-commande. C'est à ce programme que nous devons proposer une alternative ou plus précisément une simplification.

En effet, SIMPARC simule de manière très fine le fonctionnement du robot cible. Il modélise son architecture au niveau système et matériel (interruptions, bus de données etc.) ce qui n'est guère intéressant dans les premières phases de développement des application auxquelles est destiné notre simulateur. Une lacune de SIMPARC est de ne pas produire un affichage des résultats sous forme graphique : il nécessite un deuxième programme pour interpréter les données produites et créer par exemple une animation en 3D. SIMPARC est cependant indispensable pour tester en profondeur l'integralité du robot.

Nous allons donc tenter de remédier à sa lourdeur au prix de simplifications de modélisation.

Notre objectif premier est de faire tourner une simulation le plus simplement possible. Il faut néanmoins que cette simulation soit réaliste pour pouvoir être exploitée. Nous avons donc éliminé la modélisation de la partie «matériel» et «système» du robot. Nous avons donc seulement conservé la modélisation physique.

Notre simulateur sera utilisé dans les premières phases de développement et ensuite SIMPARC permettra d'affiner les tests.

2.3 Cycab

Le Cycab est un véhicule électrique autonome destiné à devenir un moyen de transport public individuel, c'est-à-dire un véhicule mis à la disposition du public comme moyen de transport (une photo du Cycab est donnée à la figure 2.1).

FIG. 2.1 – *Le Cycab*

Il dispose d'un grand nombre de capteurs pour percevoir des informations extérieures. Ceux qui nous préoccupent sont : une ceinture de 16 capteurs ultrason répartis autour de la carrosserie et une caméra placée à l'avant du véhicule (voir annexe D figure D.1 : la ceinture de capteurs ultrason).

Il est composé de 4 roues indépendantes disposant de leur propre moteur. Chaque roue peut être tournée et contrôlée en vitesse individuellement. Il est ainsi possible par exemple d'appliquer du différentiel aux roues dans les virages de manière logicielle. C'est le programme de contrôle-commande qui se charge de distribuer correctement les ordres aux roues via un bus de com-

munication (bus CAN : control area network). Celui-ci est très performant au niveau de sa fiabilité : il est temps-réel et peu sensible aux parasites.

Pour plus de détails, consulter [BGhMG99].

2.4 Rx90 Staubli

Le Rx90 Staubli est un bras manipulateur à 6 degrés de liberté. Il permet de réaliser des manipulations délicates, répétitives ou dans des conditions extrêmes (voir figure 2.2).

FIG. 2.2 – *Le bras manipulateur Rx90 Staubli*

Il est composé de six segments pouvant réaliser chacun une rotation et est muni d'une pince à son extrémité. Il est contrôlé en appliquant des couples aux articulations.

Le Rx90 et le Cycab sont les deux principaux robots auxquels va s'appliquer notre simulateur dans un premier temps. Par la suite un grand nombre de robots pourront être simulés.

Le modèle physique du Rx90 est fourmi en annexe H.2.

Chapitre 3

Le simulateur

3.1 Caractéristiques du simulateur

Les principales caractéristiques de ce simulateur sont les suivantes :

- **Fonctionnalité :**
  Le but de ce simulateur n'est pas d'être extrêmement perfectionné mais d'être utilisable le plus simplement possible. De nombreux utilisateurs ont besoin de visualiser les résultats des programmes de contrôle de robots (véhicules autonomes ou bras manipulateurs par exemple). Ils ont alors l'utilité d'un simulateur pour tester leurs programmes rapidement et simplement. Il existe déjà des simulateurs très perfectionnés pour modéliser entièrement et très finement un robot (SIMPARC par exemple) mais ceux-ci sont lourds dans leur utilisation et peu adaptés aux premières phases de développement.

- **Adaptabilité :**
  Nous voulons que le simulateur soit le plus facilement adaptable pour pouvoir être utilisé dans de nombreuses configurations et par différentes personnes. Nous l'avons donc programmé de manière modulaire pour pouvoir ajouter ou supprimer des parties qui n'intéresseraient que quelques utilisateurs.

- **Visualisation des résultats :**
  Pour être utilisable, un simulateur doit fournir une vue de l'environnement simulé et de l'évolution du système. Comme les robots utilisés évoluent dans l'espace, nous avons choisi de faire une représentation animée en 3D en utilisant la bibliothèque graphique OPENGL.

- **Portabilité :**
  Les utilisateurs potentiels du simulateur utilisent différents types de machines (UltraSparc, Silicon Graphics, PC) et de systèmes d'exploitation (Solaris, Irix, Linux, Windows). Il est donc indispensable de

créer un simulateur portable. Pour cela nous avons programmé en C++, compilé avec gcc, et utilisé MESA, une implémentation libre d'OPENGL. Tous ces outils sont gratuits et disponibles sur des dizaines d'architectures différentes. Notre simulateur est actuellement compilable sous Solaris, Linux et Irix. Le portage vers d'autres architectures n'est en principe pas un problème pour peu que toutes les bibliothèques que nous avons utilisées soient disponibles (ce qui est le cas sur un très grand nombre de systèmes).

- **Modélisation de capteurs :**
  Le but de ce simulateur est de modéliser des robots agissants en fonction de leur environnement. Pour avoir une certaine conscience du monde qui les entoure, ces robots utilisent des capteur extérocéptifs, c'est-à-dire donnant des informations sur le monde extérieur. Nous avons donc modélisé l'utilisation de tels capteurs. Différents modèles sont utilisables (Ultrason, Laser, Caméra, Caméra linéaire).

  Il est à noter que certains capteurs utilisés couramment par des robots mobiles n'ont pas été traités, comme par exemple le gyroscope. Il sera peut-être utile plus tard de les prendre en considération.

- **Interfaçes avec des programmes de contrôle-commande, API :**
  La principale utilisation de ce simulateur sera en interconnexion avec un logiciel d'aide à la création de programmes de contrôle-commande nommé ORCCAD. ORCCAD génère des fichiers source en C. Nous fournissons donc une API, c'est-à-dire une bibliothèque de fonctions qui pourront être appelées depuis un programme C ou C++ de manière simple. Il sera également possible d'utiliser ce simulateur depuis un quelconque programme écrit en C/C++ ou tout langage acceptant une interface avec du C (c'est-à-dire presque tous).

3.2 Les Capteurs

Les capteurs sont une composante vitale des robots. Ils permettent à ceux-ci d'avoir une perception de leur environnement. Le simulateur que nous programmons va servir à simuler un robot dans un environnement donné. Nous avons donc offert à l'utilisateur la possibilité de gérer différents types de capteurs. Il pourra de plus créer ses propres capteurs.

La modélisation des capteurs est réalisée de différentes manières selon le type de capteur.

3.2.1 Les différents types de capteurs

Caméra

Le modèle de caméra renvoie une image bitmap à l'utilisateur. La responsabilité d'interpréter le résultat est laissée à l'utilisateur, il pourra appliquer un algorithme de vision pour éviter un obstacle par exemple (voir figure 3.1).

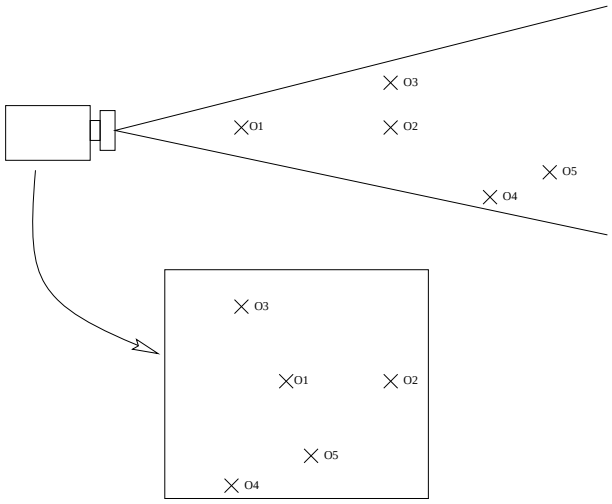

FIG. 3.1 – *Capteur Caméra*

Caméra Linéaire

La caméra linéaire est une caméra composée d'une seule ligne horizontale de n pixels. cette ligne de pixels est une image complète redimenssionnée. En pratique, on utilise une lentille convergente et un cache laissant passer un fin pinceau lumineux pour réduire la vue de la caméra à une ligne (voir figure 3.2).

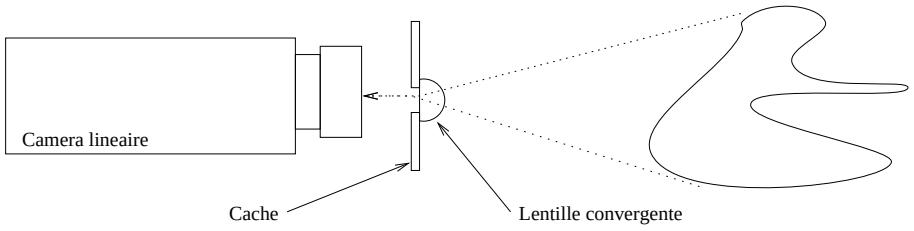

FIG. 3.2 – *Capteur Caméra linéaire*

Une image issue d'une telle caméra est plus simple et plus rapide à traiter qu'une image $n \times m$. Elle permet par exemple de suivre une cible composée de plusieurs lumières. Dans notre simulateur nous allons utiliser une vue $n \times m$ que nous allons réduire à une image bitmap $n \times 1$.

Capteur à ultrasons

Les capteurs ultrasons retournent une information de distance depuis un point donné et dans une direction donnée. La valeur renvoyée sera significative de la distance de l'objet le plus proche dans un cône d'angle précisé par l'utilisateur (voir figure 3.3).

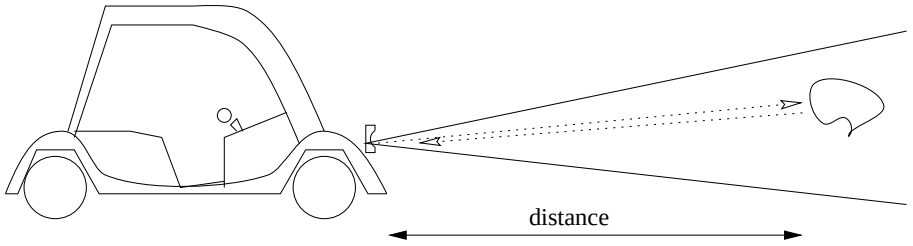

FIG. 3.3 – *Capteur Ultrason*

La distance rendue est peu précise. Elle dépend des matériaux rencontrés, des angles d'incidence, et est peu fiable.

Laser

Le type de capteur Laser est un capteur qui retourne une information de distance. Cette valeur est retournée depuis un point donné et dans une direction précise. Cette information est précise ($\simeq$ 1 cm sur une distance de 12 m) et fiable (voir figure 3.4).

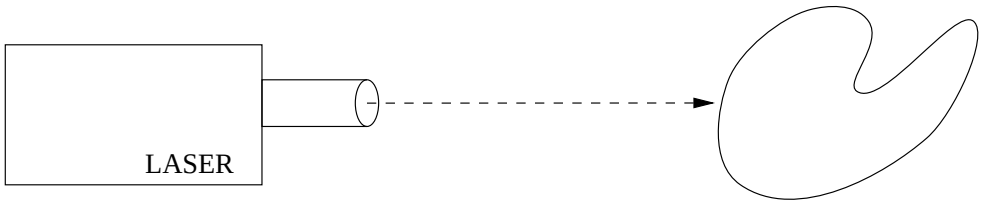

FIG. 3.4 – *Capteur Laser*

Laser à balayage

Le capteur laser à balayage est une variante du laser. Ce capteur sera défini par un angle et un pas de balayage ou un angle et un nombre d'échantillons à retourner : les distances obtenues seront renvoyées dans un tableau à l'utilisateur. Un tel capteur possède à la fois la précision d'un laser et la densité de détection voulue (voir figure 3.5).

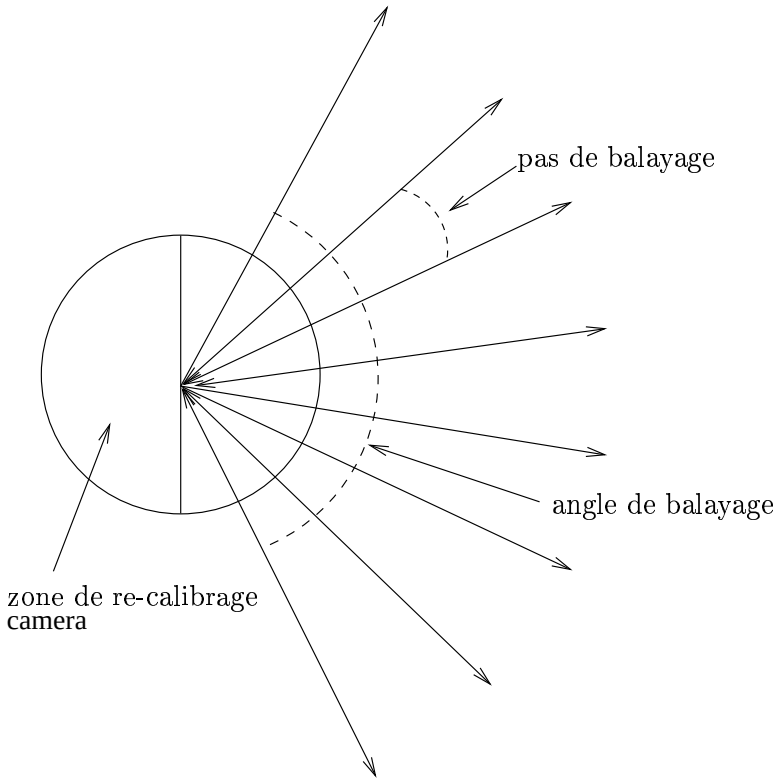

FIG. 3.5 – *Capteur Laser à balayage*

3.2.2 Modélisation des capteurs avec OPENGL

Caméra

Pour les capteurs de type **caméra**, une nouvelle vue est créée pour chaque capteur et la scène est re-dessinée. Cette vue est ensuite affichée dans une fenêtre séparée de la vue principale pour disposer de la vue de la caméra en direct. Les données nécessaires à l'utilisateur lui seront renvoyées dans trois tableaux représentant les composantes *rouge, verte* et *bleue* d'une image bitmap sur laquelle il pourra travailler.

Caméra linéaire

La caméra linéaire est un cas particulier de caméra. Nous créons une nouvelle vue comme pour une caméra, mais seulement une ligne horizontale de pixels est renvoyée à l'utilisateur.

Laser

Pour les capteurs de distance de type **laser**, nous avions prevu de créer une seule vue en mode « orthographique[1] » pour tous les capteurs. Finalement pour des raisons de simplicité, nous avons réutilisé la vue principale qui, elle, est en perspective. Ce mode de vue nous pose cependant un problème supplémentaire. En effet, la fonction de normalisation des distances n'est plus linéaire. Nous avons donc utilisé des fonctions de projection inverse (`gluUnproject`) pour obtenir les bonnes valeurs de distance. Cette vue, dessinée en totalité pour l'utilisateur ne sera pas dessinée en totalité lors de la simulation des capteurs pour ne pas trop ralentir la vitesse du simulateur. Le fait de ne créer qu'une seule vue nous permet d'améliorer la vitesse de calcul des objets en 3D. En effet, il est possible en OpenGL de définir des *listes d'affichage* (display-list) qui sont des formes pré-compilées des objets que nous manipulons. Une fois la liste d'affichage créée, son affichage est notablement plus rapide. Malheureusement il n'est pas prévu dans Mesa la possibilité de partager les listes d'affichage entre plusieurs fenêtres, c'est pourquoi nous utilisons la même fenêtre pour tous les capteurs de distance (qui ne necessitent pas d'affichage).

Cette vue est utilisée pour effectuer un « picking » : on définit une petite zone à un endroit voulu de la vue et une fonction calcule les objets rencontrés et leur distance par rapport à la vue courante. Ces informations sont disponibles grâce à la gestion d'un « Z-buffer » par OpenGL. Un Z-buffer est une image en mémoire de la zone d'affichage où sont stockées les coordonnées en Z des objets affichables afin de déterminer lesquels sont au premier plan et doivent être affichés. Leur distance est donnée normalisée sur l'intervalle $[0\,;\,1]$: 0 est la position du capteur et 1 est la limite maximum de la vue définie. Il suffit alors d'une fonction de conversion pour obtenir la distance du capteur aux objets visibles.

Laser à balayage

Pour les capteurs de type **laser à balayage**, une vue en mode « perspective » est également créée ce qui permet de tenir compte de la diminution apparente de la taille d'un obstacle avec son éloignement. Un objet de petite taille pourra ainsi être détecté seulement s'il est suffisamment proche.

1. Une vue en mode orthographique par opposition à une vue en perspective ne rend pas compte de la réduction de taille des objets en fonction de leur éloignement.

Nous avons quelque problèmes actuellement pour la simulation du laser à balayage. Un tel capteur est capable de réaliser des centaines d'échantillonage de distance en quelques dizaine de millisecondes. Une telle fréquence est très difficile à atteindre avec notre méthode de picking. Il faudra sans doute utiliser une autre méthode pour parvenir à des performances acceptables. Nous avons pensé à lire par exemple les informations de distance directement dans le tampon de profondeur mais il faut alors des traitements supplémentaires pour obtenir les valeurs dans les bonnes unités.

Capteurs à ultrasons

Pour les capteurs à ultrasons, une vue en mode «perspective» est créée et des «plans de coupe» (clipping planes) sont positionnés pour délimiter la zone de détection du capteur (voir figure 3.6).

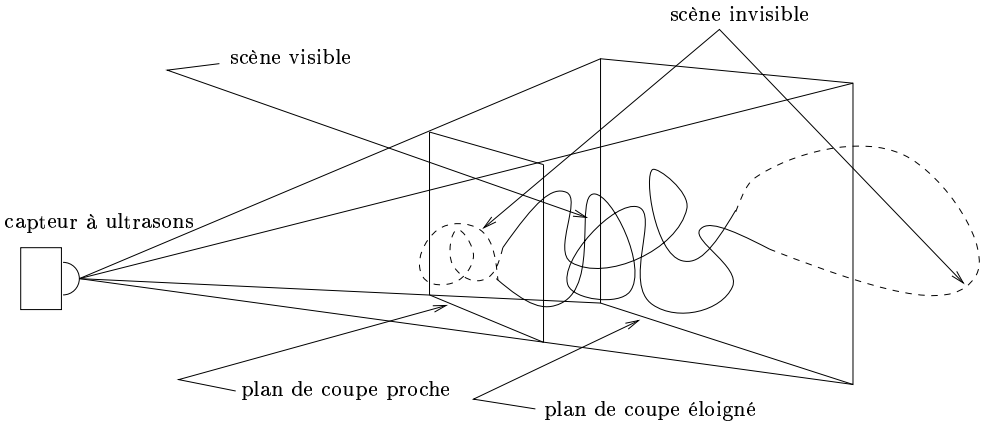

FIG. 3.6 – *Plan de coupe (cliping plane)*

La simulation d'un capteur à ultrasons pose des problèmes très divers :

- Certains objets ne doivent pas êtres pris en compte. Objets trop petits (voir objet 3 figure 3.7), trop éloignés, ayants une trop faible réflexion des vibrations etc. Une manière de remédier à ce problème serait de renvoyer à l'utilisateur une liste de tous les objets rencontrés et à lui de déterminer si tel objet doit être pris en compte. L'inconvénient de cette méthode est encore une fois le coût élevé en temps de calcul.

- Un autre problème que nous pose cette fois notre méthode de simulation est la mauvaise interprétation des distances par le picking OPENGL (voir figure 3.7) l'objet 1 est plus proche du capteur que l'objet 2 au sens de la distance euclidienne. Le picking OPENGL par contre est effectué selon des droites perpendiculaires au capteur. La distance rendue par le picking pour l'objet 1 est alors plus grande que

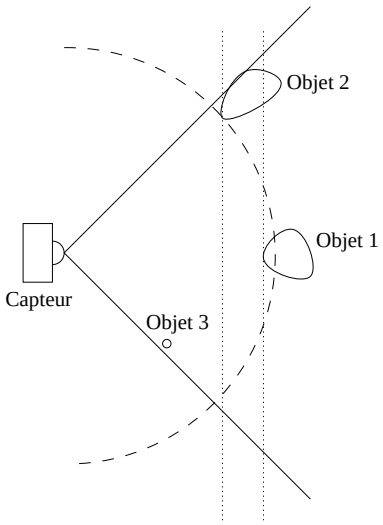

FIG. 3.7 – *Problèmes des capteurs à ultra son*

celle de l'objet 2, ce qui est très imprécis dans le cas de grands angles de détection du capteur. Il est nécessaire de refaire des calculs pour obtenir les véritables distances, ce qui ralentit encore le simulateur.

3.3 Utilisation du simulateur

L'utilisation du simulateur se fera par l'intermédiaire d'une bibliothèque de fonctions. Il s'agit de fonctions d'initialisation d'OpenGL, de fonctions de contrôle de la simulation et de fonctions utilisées pour récupérer les données utiles (valeurs retournées par les capteurs, images données par les caméras etc.)

3.3.1 Contrôles du simulateur

Nous avons prévu différentes options pour la simulation. Le mode par défaut est un affichage de tous les objets sans les repères.

On peut activer les repères. On visualise alors les liens entre les différents objets. Chaque objet possède un repère de couleur différente pour mieux le distinguer. (voir figure 3.8)

Il est également possible de désactiver l'affichage des objets. Ceux-ci sont alors représentés par des points et on distingue plus simplement les repères. L'affichage est alors nettement plus rapide. (voir figure 3.9)

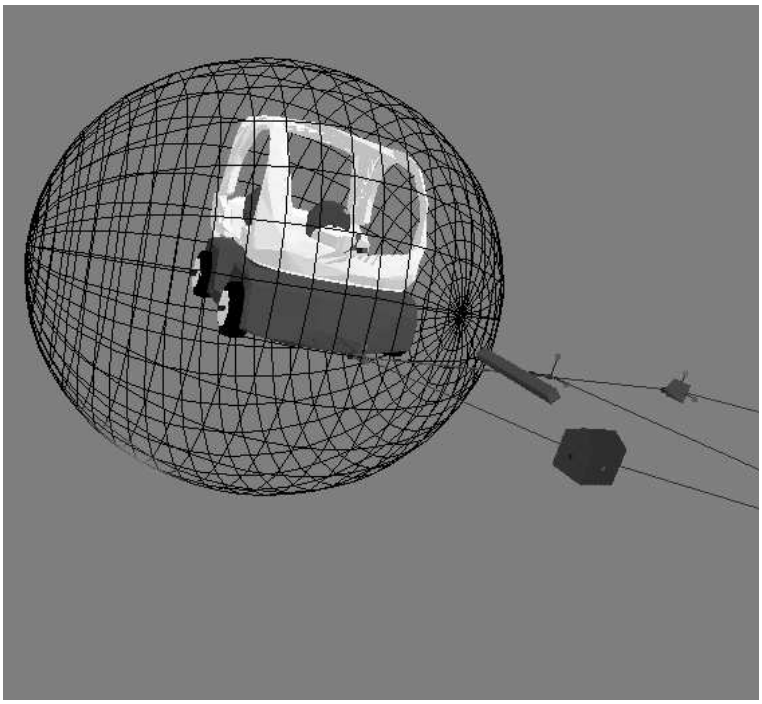

FIG. 3.8 – *Scène avec les repères et les objets*

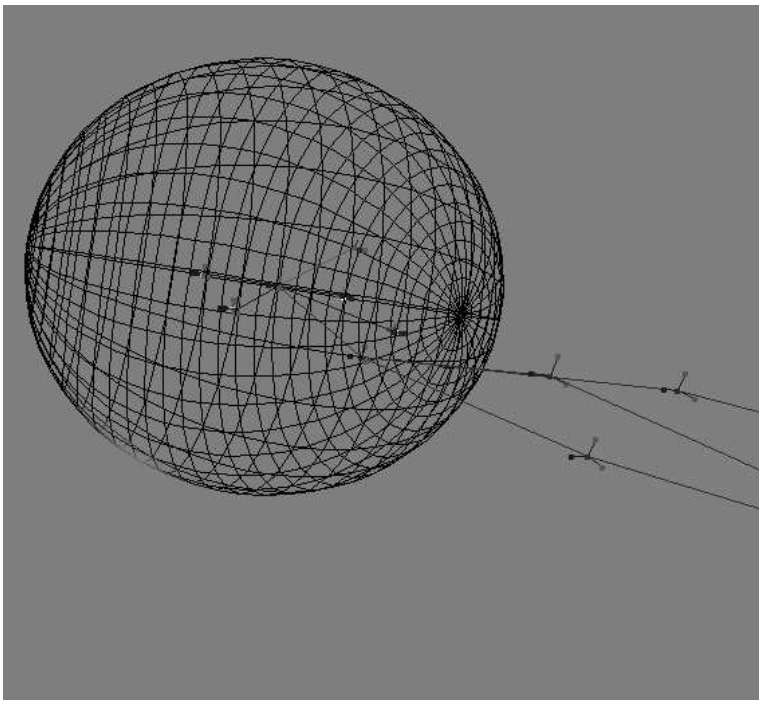

FIG. 3.9 – *Scène avec les repères mais sans les objets*

3.4 Sauvegarde et Chargement

La scène complète sera chargée depuis un fichier au format texte (ASCII).

La structure du fichier est la suivante :

nom de la scène
nombre d'objets dans la scène
couleur de fond de la fenêtre principale
nom de l'objet
type de lien
valeurs initiales des paramètres du lien (nombre variable en fonction du lien)
image (nom de fichier)
nombre de successeurs
liste des successeurs de l'objet
liste des autres objets de la scène

Un exemple de fichier est donné en annexe G. Son utilisation donne la scène produite en annexe E.1

3.5 Image d'un Objet

Le fichier de sauvegarde d'une scène contient pour chaque objet un nom de fichier référençant l'image de celui-ci. L'image d'un objet est donnée par un « maillage » (mesh). Un maillage est une liste de sommets, une liste de faces et une liste de vecteurs donnant la normale de chaque face.

Le format de fichier utilisé est un format texte ASCII (fichier *.asc*). Ce format à la propriété d'être très simple. Il contient une liste de sommets, une liste de faces et quelques autres informations. Il est donc très simple à lire est peut être généré par la plupart des modeleurs 3D du marché (notamment 3DStudioMax). On considère que le fichier image contient un «bon» maillage, c'est à dire une liste de faces triangulaires et orientées correctement. Ce deuxième critère est très complexe à définir. Dans le cas d'une sphère par exemple, il est simple de déterminer si la normale de chaque face pointe vers l'extérieur ou vers l'intérieur. Cela devient beaucoup plus délicat dans le cas d'un volume non fermé (la carrosserie du Cycab) voire non rigide (un drapeau). Nous n'utilisons que des objets rigides et laissons aux soins de l'utilisateur de fournir un «bon» maillage. Par ailleurs, le retournement des faces influe uniquement sur les vues des caméras, les capteurs de distance n'en tiennent pas compte.

Le maillage sera chargé au moment de la lecture de la scène, puis converti en liste d'affichage OpenGL. Un exemple de maillage est donnée en annexe F

Chapitre 4

Structure du simulateur

Note : Ce chapitre est issu d'un rapport de mi-projet que nous avons réalisé après avoir terminé la première partie du simulateur. Il sera utile pour retravailler, si besoin est, sur le simulateur.

Les fichiers peuvent être répartis en 4 groupes : Intégration & Simulation, Modélisation, Visualisation et enfin Représentation & Simulation des capteurs. Certains de ces fichiers représentent des classes avec leurs méthodes, d'autres ne contiennent que des procédures. Nous allons développer ces différentes parties par la suite.

4.1 Définition des Fichiers

4.1.1 Fichier `Main.cc`

Dans celui-ci, nous déclarons les variables globales à tout le programme, puis nous construisons la scène (en faisant appel au constructeur de cette classe) en chargant un fichier de sauvegarede. Nous créons ensuite l'environnement OpenGL (fenêtre et vue) et nous appelons la création des environnements OpenGL propres aux capteurs si nécessaire. C'est également ici qu'on indique les fonctions rattachées à la fenêtre principale. Pour terminer, nous entrons dans la boucle d'exécution d'OpenGL (`glutMainLoop()`). C'est cette boucle qui gère l'appel à l'intégrateur.

4.1.2 Fichiers `Tampon.cc` et `Integration.cc`

Notre première idée concernant la simulation a été de créer trois processus légers (threads) :

- Un pour l'affichage de la scène et la gestion du modèle.
- Un pour la gestion de l'API (contrôle extérieur).
- Un pour l'intégration du modèle.

Pour des raisons de simplicité, nous avons essayé d'intégrer cette simulation dans la boucle « Idle » d'OpenGL (C'est la boucle d'attente d'événement, OpenGL y passe donc dès qu'il n'y a plus de travail de dessin ou de transformation à effectuer). Afin de pouvoir cependant revenir à la première solution dans un but de rapidité, nous avons écrit cette partie du programme en établissant une indépendance maximum entre la simulation et le modèle. Nous pouvons voir la structure choisie sur la figure 4.1.

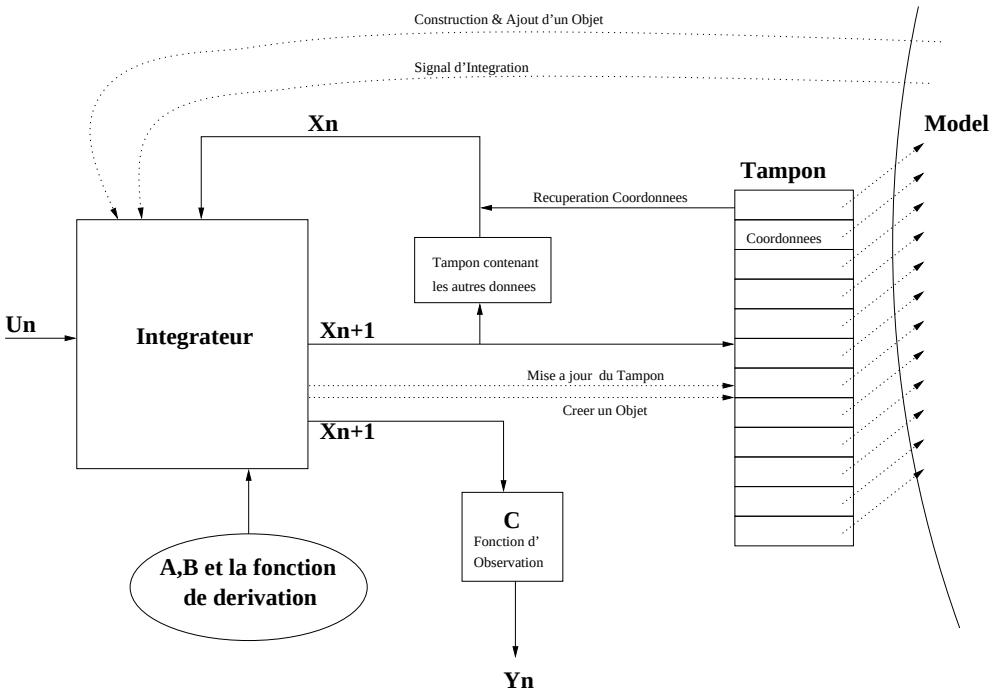

FIG. 4.1 – *Représentation de l'intégrateur*

Cette simulation est divisée en deux fichiers, le premier (`Tampon.cc`) est un tampon entre l'intégration proprement dite et le modèle. C'est ici que nous stockons les coordonnées de tous les objets de la scène. La construction de ce tampon est donc effectuée dynamiquement lors de la construction de la scène via l'intégrateur (de manière à mémoriser les liens). Celui-ci utilise la méthode de Runge-Kuttha à l'ordre 4. Nous estimons que ce niveau de dérivation est suffisant pour la précision que nous désirons atteindre dans la simulation. Il est cependant envisageable d'augmenter son efficacité en implémentant une méthode à pas variable, pour allier l'efficacité de Runge-Kutta à une souplesse lui permettant de mieux s'adapter au type de la simulation à effectuer (par exemple lors de simulations comportant de grandes variations).

L'idée d'une conception dynamique de toutes les parties de notre application a trouvé ses limites dans la programmation de cet intégrateur. En effet, l'intégration nécessite une adaptation au modèle à simuler. Suivant ce modèle, il est nécessaire de prendre en compte, en plus des coordonnées (que nous retrouvons dans tous les modèles), l'accélération ainsi que d'autres paramètres. Dans la figure 4.1, la fonction de dérivation, les paramètres (A et B) du modèle dynamique du robot, la description de la fonction de commande et d'observation ne sont pas définissables, simplement, à partir du modèle graphique (modèle décrit ou chargé dans un fichier). Ces informations sont donc nécessaires lors de la construction de notre intégrateur, et c'est cette raison qui nous a poussé à choisir une solution en deux étapes :

- Dans un premier temps, nous définissons « en dur » toutes les informations difficiles à définir dynamiquement (celles énumérées précédemment). C'est à dire que l'utilisateur va rentrer dans le programme source les données nécessaires avant de re-compiler. La description de la fonction de commande va intervenir dans l'intégrateur afin de connaître les types récupérés et celle du modèle physique interviendra dans la fonction de dérivation. C'est le nombre de paramètres nécessaires à la dérivation ainsi que leurs types qui permettront de construire un tampon supplémentaire si nécessaire. De la même manière, c'est l'utilisateur qui doit définir les variables qu'il souhaite observer en sortie.

- Dans un second temps, nous construisons le Tampon dynamiquement et implémentons la méthode de dérivation à l'aide des informations récupérées avant la compilation. C'est maintenant que les fonctions définies auparavant sont implémentées dynamiquement.

Pour terminer cette description, il faut ajouter que le signal d'intégration est envoyé par la boucle d'exécution, ce signal provoque une mise à jour de toutes les matrices de la scène, mais cela ne provoque pas d'affichage. C'est donc bel et bien le modèle qui décide d'une évolution visuelle de la scène. Cette technique permet d'une part de régler le pas d'intégration afin de garder une bonne précision dans la simulation et d'autre part de pouvoir régler le pas d'affichage afin de garantir une vitesse d'exécution correcte.

4.1.3 L'intégration

Le modèle physique donné par l'utilisateur est du type :

$$\dot{X} = AX + BU$$

où X est la position d'un objet, U une commande et A et B sont des paramètres fonctions du modèle physique du robot. Connaissant la position

au pas n, il faut intégrer cette formule pour obtenir la nouvelle position de l'objet au pas n+1 :

$$X^{n+1} = f(X^n + \dot{X}^n)$$

, où f est une fonction dependant de A et de B. La méthode d'intégration retenue est la méthode de Runge-Kutta à l'ordre 4. Ce type d'intégration est extrêmement courant en physique, ce qui nous a permis de reprendre des algorithmes existants.

Nous ne discuterons pas de l'intérêt d'utiliser l'intégration à l'ordre 4 mais il suffit de savoir que cela permet une bonne approximation tout en conservant un temps de calcul raisonnable. Pour plus de détails, voir [KF92] p.710.

4.1.4 Fichiers `GLfonction.cc`, `Vue` et `Dessin`

Ce fichier regroupe les fonctions OpenGL utilisées pour manipuler les fenêtres, pour gérer les interactions avec l'utilisateur et pour initialiser la boucle OpenGL.

Il contient les fonctions appelées par les évènements OpenGL liés au clavier, à la souris et à la gestion des fenêtres : (`motion()`, `mouse()`, `key()`, `idle()` etc). Des pointeurs vers ces fonctions sont positionnés à l'initialisation d'OpenGL. Chaque évènement OpenGL appelle ensuite la fonction correspondante en lui passant les paramètres utiles.

Pour utiliser les fonctions OpenGL, il faut initialiser un certain nombre de variables (« double tampon d'affichage » (double frame-buffer), mode RGBA ...), toutes les fonctions d'initialisation sont regroupées dans GLfonctions :

- `creerFenetre` : appellée pour initialiser les paramètres de la fenêtre (taille et position de la fenêtre, double buffering, mode de couleurs RGBA etc).
- `initFenetre` : initialisation de la fenêtre (couleur de fond, éclairages, positionnement de la vue).
- `display` : appelée pour afficher la scène dans la fenêtre, lors d'un changement de vue, de la scène ou si la fenêtre redevient visible après avoir été cachée.
- `reshape` : appelée lorsque la fenêtre est redimenssionnée.
- `idle` : appelée lorsque OpenGL ne fait rien. C'est pour l'instant cette fonction qui appelle l'intégrateur.
- `key` : appelée lorsqu'une touche a été pressée.
- `special` : idem que key mais pour les touches speciales (les touches de directions par exemple).

- `mouse` : appelée lors des clicks sur la souris avec le numéro du bouton appuyé passé en paramètre ainsi que la position du curseur.
- `motion` : appelée lors des déplacements de la souris avec un bouton appuyé («drag»).
- `visible` : appelée lors d'un changement de visibilité de la fenêtre (si elle disparait ou si elle apparaît).

`Vue.cc`

Ce fichier regroupe les fonctions utilisées pour calculer le point de vue de l'utilisateur dans la fenêtre principale en fonction des rotations et du zoom effectué avec la souris. Il est constitué d'une classe *CalculVue* disposant de méthodes pour positionner le point de vue désiré.

`Dessin.cc`

Le fichier `Dessin.cc` contient quelques primitives pour dessiner des objets : cube, parallélépipède ... Nous utilisons ces fonctions pour dessiner les objets simples de la scène. Les objets plus compliqués seront chargés par «maillage» depuis des fichiers au format ASCII (*.asc).

4.1.5 Fichiers `Model.cc` et `MesMath.cc`

Le fichier `MesMath.cc` contient la classe *Lien* qui est une structure utilisée comme lien entre deux repères. Elle est définie par une matrice homogène de transformation qui se décompose en une matrice de rotation et une matrice de translation.

Le fichier `Model.cc`, quant à lui, contient toutes les structures utilisées pour définir une scène (point, couleur, image, objet) à l'exception de lien. Sa structure représente les objets sous forme d'arbre comme le montre la figure 4.2

Les pointeurs sur les successeurs sont stockés dans des tableaux dynamiques car nous ne connaissons le nombre de successeurs que lors de la construction de la scène. Cette structure est remplie au fur et à mesure que le fichier de sauvegarde est lu.

4.1.6 Fichier `Capteur.cc`

Pour l'élaboration des capteurs, nous avons prévu de définir plusieurs types. Pour le moment, trois types existent :

- laser

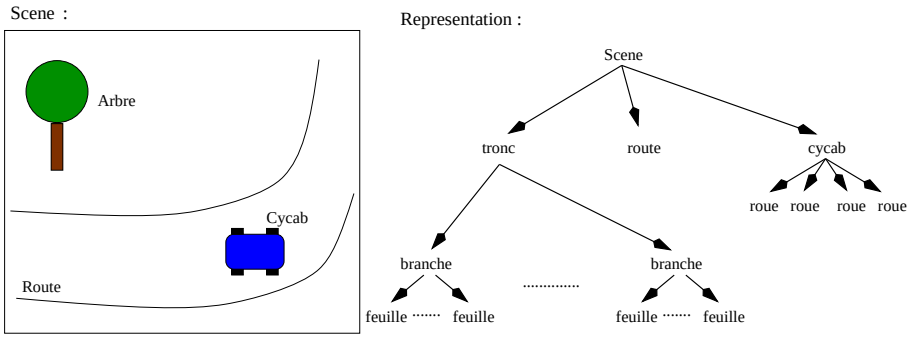

FIG. 4.2 – *Modèlisation d'une scène*

– ultrason

– caméra

Ces différents capteurs nécessitent leur propre environnement OPENGL. Ils peuvent être décomposés en deux groupes. Le groupe nécessitant un environnement d'affichage géré par la bibliothèque GLUT et celui ne nécessitant pas d'affichage (capteurs laser et ultrasons).

Caméra

Dans ce cas, nous créons une fenêtre associée à chaque caméra en procédant de manière similaire à la création du contexte de la fenêtre principale et nous initialisons les déplacements préliminaires à l'affichage afin de simuler la vue de cette caméra.

Autre capteur

Dans ce second cas, nous créons uniquement le contexte OPENGL. De cette manière tous les calculs se rapportant à la lumière ne sont pas effectués (gain de temps) et nous nous basons sur le tampon de profondeur afin de récupérer les informations nécessaires à la simulation.

La simulation dépend du type du capteur. C'est grâce à celui-ci que nous récupérons le bon nombre d'informations pour simuler son comportement et retourner le résultat à l'API. L'introduction de paramètres pour chaque capteur est gérée en donnant le choix à l'utilisateur entre différents capteurs de même type ayant des paramètres différents. La création d'un capteur sur mesure sera prise en charge par l'utilisateur en reproduisant le type désiré et en modifiant les paramètres à son gré.

4.2 Conception des Classes

4.2.1 Classes *Integrateur*, *Tampon* et *Element*

L'intégrateur se décompose en quatre classes (figure 1.3). La classe *caracObjet* est une liste chaînée contenant le nombre de variables de tous les objets de la scène. Elle est construite parallèlement à la construction du *Tampon*. De cette manière, lors de la mise à jour du tampon, nous pouvons avoir avec un parcours parallèle des deux listes, une information sur le type de la donnée à mettre à jour. L'objetScene et l'elemDeb sont des pointeurs sur la tête de liste et l'objetCourant ainsi que l'elemCour sont des pointeurs sur l'instance courante. Les deux autres pointeurs dans la classe *Tampon* ne sont que des variables temporaires. Enfin, les booléens ne servent qu'à indiquer l'état de la construction (des flags).

La figure 4.3 est une représentation de ces classes :

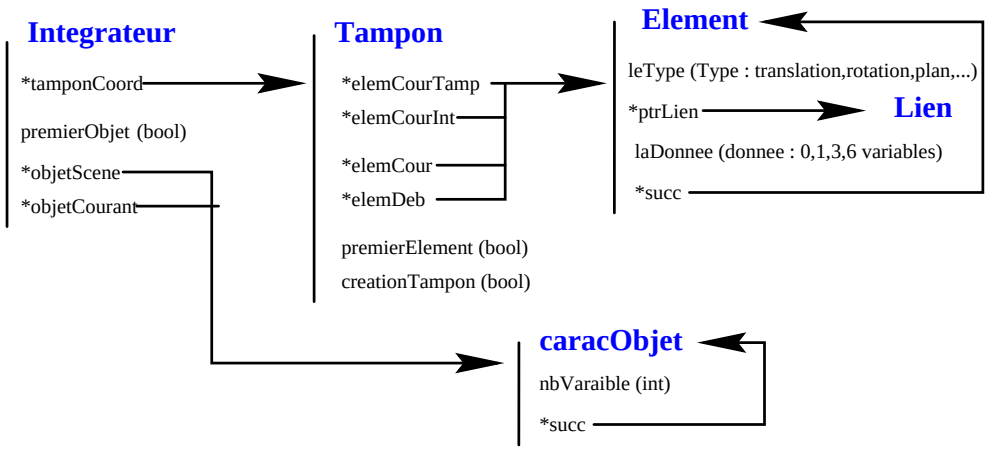

FIG. 4.3 – *Classes Integrateur, Tampon et Element*

4.2.2 Classe CalculVue

La classe *CalculVue* (voir figure 4.4) a pour but la gestion de tous les déplacements s'effectuant avant l'affichage de la scène (le zoom, les rotations...).

Les variables rotateX,....,deplacX,... sont donc les translations et rotations à effectuer depuis la vue actuelle. La variable typeRotation permet de choisir entre le mode « vol » (fly) et le mode « inspection » (inspect). maMatrixVue contient la matrice des rotations effectuées depuis le début. En

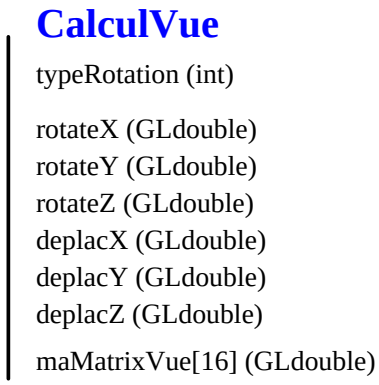

FIG. 4.4 – *Classe CalculVue*

effet, pour avoir les rotations et translations désirées, il nous faut dans un premier temps effectuer les rotations, ensuite nous pouvons faire les translations. Nous chargeons donc cette matrice au début, exécutons les rotations demandées par l'utilisateur, et enfin les translations.

Les méthodes de cette classe ont donc pour objectif la récupération de la matrice de rotation et l'exécution des translations et rotations demandées.

4.2.3 Classe *Lien*

La classe *Lien* (voir figure 4.5) comporte des fonctions d'initialisation : `initMatrice()`, de modification : `modiMatrice()` et de manipulation de matrices. Nous avons défini cinq types de liens, notamment fixe (zéro degré de liberté), rotation (trois degrés de liberté) ou encore libre (six degrés de liberté). À chaque type de lien correspond une initialisation et une modification. Nous passons les paramètres (angles, distances) et la fonction `modiMatrice()` calcule la matrice homogène de transformation correspondante.

Les variables `r`, `d` et `x`, `y`, `z`, alfa, `t`eta, `b`eta représentent les valeurs du déplacement. On initialise suivant le type du lien, qui est indiqué dans le champ `l`eType, certaines de ces variables. À l'aide de celles-ci, on remplit la matrice homogène : `m`atHom ainsi que celles représentant la rotation `r`ota et la translation `t`rans. L'entier `a`ffficheCapteur est un flag indiquant si l'objet représenté est un capteur. Si c'est le cas, la matrice `c`apMat contiendra l'inverse de `m`atHom.

4.2.4 Classe *Scene*, *Objet*, *Couleur*, *Image* et *Point*

La classe *Point* représente les coordonnées d'un point dans le repère auquel il appartient. L'*Image* est composée de `n`bPoints points représentant

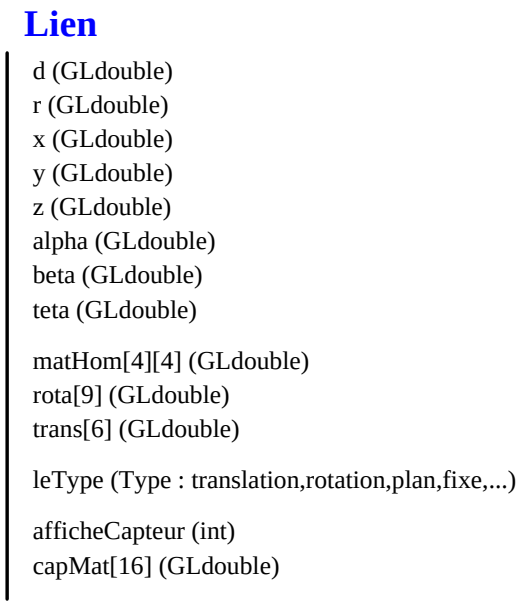

FIG. 4.5 – *Classe Lien*

l'objet. Ces points forment une liste chaînée. La classe *Couleur* est similaire à la classe *Point*, son but est d'indiquer la couleur d'un objet ou d'une scène. Celle-ci, ainsi que tous les autres objets possèdent un champ `nom` qui permet de les identifier. Ces deux structures possèdent une couleur. La scène possède un tableau dynamique de pointeurs sur des objets : `chaine`; sa taille est `nbObjet`. Ces deux champs se retrouvent dans la classe *Objet* par l'intermédiaire des variables `CptSucc` et `TabObjet`. Cette dernière classe possède une image, un pointeur sur son père, un pointeur sur un *Lien* représentant le déplacement entre le repère de l'objet et celui de son père. Elle possède aussi un flag permettant de savoir si cet objet est une fin de chaîne et dans ce cas si c'est un capteur. C'est uniquement dans cette configuration que l'instance possède un pointeur vers la classe *Capteur*.

La figure 4.6 représente la structure d'une scène.

La scène et tous les objets possèdent une fonction d'affichage pour pouvoir tenir compte dynamiquement de la structure de chaque image à afficher.

4.2.5 Classe Capteur

La classe *Capteur* (figure 4.7) possède des méthodes de gestion d'environnement et d'affichage OPENGL. Comme la fonction de gestion des évènements est dirigée par la librairie GLUT, il nous fallait donner toutes les

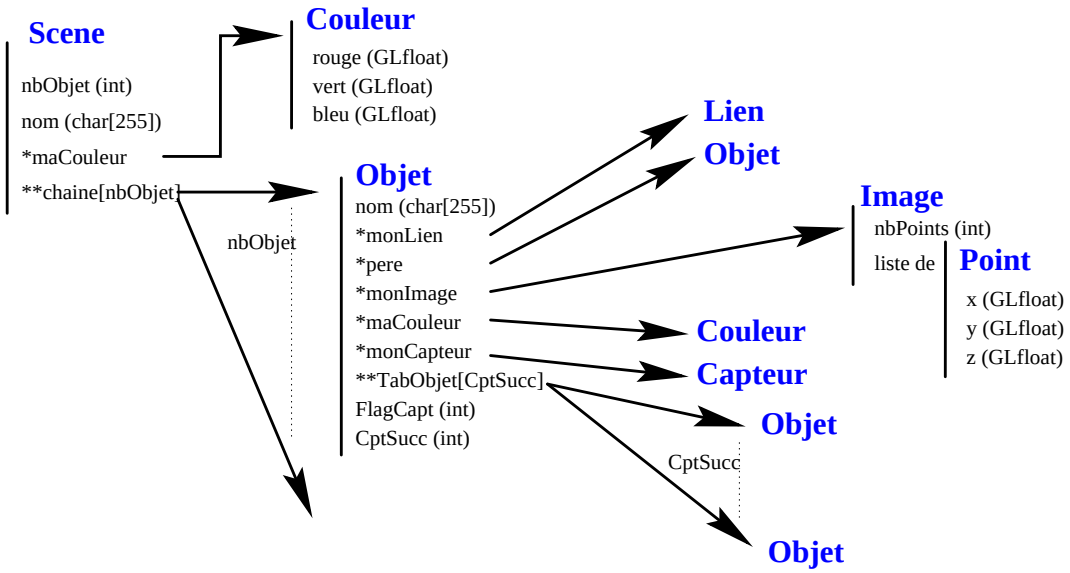

FIG. 4.6 – *les Classes représentant la modèlisation*

fonctions utiles à chaque fenêtre (display, reshape,...). Le problème que nous avons rencontré est le passage en paramètre de ces fonctions. En effet, afin de garder une méthode par capteur, nous les avons introduites dans les classes. De cette manière, nous ne pouvions plus les passer en paramètre comme des fonctions classiques. Nous avons donc décidé de développer une fonction commune à toutes les instances de capteurs, chargée de faire le bon appel suivant l'objet. C'est donc le but des fonctions `leDisplay`, `leDisplayCap` et `leReshape`.

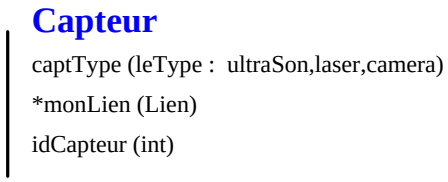

FIG. 4.7 – *Classe Capteur*

Le type du capteur est conservé dans la variable `capType`, le champ `idCapteur` est l'identificateur de la fenêtre d'affichage du capteur. Nous avons aussi un pointeur vers le Lien de l'objet afin de pouvoir récupérer les informations nécessaires à l'affichage. En effet, il est nécessaire de connaître la matrice représentant le déplacement de l'origine à l'objet afin d'appliquer

son inverse avant d'appeler l'affichage de la scène. De cette manière, la vue de la fenêtre représente bien la vue du capteur.

Conclusion

D'un point de vue pratique, nous avons réussi à développer un simulateur pouvant être utilisé de manière relativement simple et pour des usages divers.

Bien que la simulation des capteur ne soit pas achevée, nous avons tout de même bien dégrossi le problème de la simulation à l'aide d'OpenGL ce qui devrait être un bon point de départ pour la poursuite du développement du programme. Nous avons vu quelles étaient les limites de performance d'OpenGL pour le calcul des distances (tout au moins à l'aide de la technique de picking).

Techniquement, ce stage nous a apporté une meilleur connaissance de la conception d'un programme dans son intégralité, des techniques de programmation et de l'utilisation des bibliothèques C++ et OpenGL.

D'un point de vue plus théorique nous avons dû étudier les outils de base de l'analyse numérique (méthode de Runge-Kutta), de l'imagerie 3D (coordonnées homogènes, transformations géométriques) et quelques principes de mécanique (cinématique).

Un autre intérêt de ce stage à été sa polyvalence et la découverte du travail en laboratoire.

Nous avons appliqué des méthodes de nombreux domaines (informatique, mécanique, mathématique) afin de mener à terme ce projet. Ceci nous a permis d'entrevoir les ouvertures de l'informatique à différentes disciplines.

Notre intégration dans différentes équipes de l'INRIA ainsi que la réalisation de ce simulateur en binôme pendant un an ont été très fructueuses tant au niveau du travail en groupe que de la réutilsation de programmes existants. Nous avons pu découvrir le monde de la recherche en informatique qui nous était jusqu'à présent assez lointain.

Nous avons pu ainsi avoir une idée assez précise de nos futurs travaux de recherche en tant qu'étudiants.

Annexe A

Manuel de construction

A.1 Bibliothèques utilisées

Nous avons lié les bibliothèques dynamiquement. De cette manière le simulateur est moins gros et utilise moins de mémoire. En contrepartie, il faut disposer des bonnes bibliothèques pour pouvoir l'exécuter.

Voici la liste des bibliothèques que notre exécutable utilise (données par la commande *ldd* sur un système Solaris). Sur une silicon graphics on aurait pas MesaGL mais GL et quelques autres bibliothèques en moins.

```
lucifer(test1) ldd Exec
        libglut.so =>    /usr/lib/libglut.so
        libMesaGLU.so => /usr/lib/libMesaGLU.so
        libMesaGL.so =>  /usr/lib/libMesaGL.so
        libX11.so.4 =>   /usr/lib/libX11.so.4
        libXext.so.0 =>  /usr/lib/libXext.so.0
        libXmu.so.4 =>   /usr/lib/libXmu.so.4
        libXt.so.4 =>    /usr/lib/libXt.so.4
        libXi.so.5 =>    /usr/lib/libXi.so.5
        libSM.so.6 =>    /usr/lib/libSM.so.6
        libICE.so.6 =>   /usr/lib/libICE.so.6
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libsocket.so.1 =>/usr/lib/libsocket.so.1
        libnsl.so.1 =>   /usr/lib/libnsl.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        libmp.so.2 =>    /usr/lib/libmp.so.2
```

Voici le makefile compilant notre simulateur. il est fait pour compiler sur Solaris. Nous avons également deux autres Makefile semblables pour Linux et Irix.

```
#- commenter la target qui suit pour avoir un make verbeux
```

```
#.SILENT:

#-- Flags
FLAG =
CC = CC
# -DDEBUG
#-- repertoire par defaut des biblio Solaris
R_IRIX = /home/lucifer/guilloud/lib/irix

#-- lib solaris
R_IRIXlib=-L$(R_IRIX)

#-- include solaris
R_IRIXinc=-I$(R_IRIX)/include

#-- repertoire par defaut ou sont stocker les includes specifiques au programme
R_INC = -I./include

#-- repertoire par defaut ou sont stocker les libr specifiques au programme
R_LIB = -L./lib

#-- repertoire ou sont stocker les hearders pris en compte par le makefile
R_INCL = ./include

#-- repertoire ou sont stocker les sources pris en compte par le makefile
R_SRC = ./src

#-- repertoire ou l'on cree l'Exec
R_EXEC  = ./

#-- repertoire ou sont stocker les sources pris en compte par le makefile
R_OSRC = ./Osrc


#-- Option de Purify
#--/usr/local/pure/purify
################################################################################


Prog : $(R_SRC)/Main.cc $(R_OSRC)/GLfonction.o                          \
$(CC)  $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_EXEC)/Exec       \
$(R_SRC)/Main.cc  $(R_OSRC)/Mesh.o $(R_OSRC)/MesMath.o        \
$(R_OSRC)/Tampon.o $(R_OSRC)/Vue.o $(R_OSRC)/Integration.o    \
$(R_OSRC)/CaptBase.o $(R_OSRC)/CaptVideo.o $(R_OSRC)/Camera.o \
$(R_OSRC)/Laser.o $(R_OSRC)/LaserBal.o $(R_OSRC)/Capteur.o    \
$(R_OSRC)/GLfonction.o $(R_OSRC)/Model.o                      \
$(R_IRIXlib) -lglut -lGLU -lGL  -lXmu -lXext -lX11  -lm


#######-------------------------------------------------------------------------------
$(R_OSRC)/GLfonction.o : $(R_SRC)/GLfonction.cc $(R_INCL)/GLfonction.h $(R_OSRC)/Model.o
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc) -o $(R_OSRC)/GLfonction.o \
$(R_SRC)/GLfonction.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/Model.o : $(R_SRC)/Model.cc $(R_INCL)/Model.h $(R_OSRC)/Dessin.o \
$(R_OSRC)/Mesh.o $(R_OSRC)/Integration.o $(R_OSRC)/Capteur.o \
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_OSRC)/Model.o   \
$(R_SRC)/Model.cc -c
```

```
#######-------------------------------------------------------------------------------
$(R_OSRC)/Dessin.o : $(R_SRC)/Dessin.cc $(R_INCL)/Dessin.h
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_OSRC)/Dessin.o  $(R_SRC)/Dessin.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/Mesh.o : $(R_SRC)/Mesh.cc $(R_INCL)/Mesh.h
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc) -o $(R_OSRC)/Mesh.o  $(R_SRC)/Mesh.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/Integration.o : $(R_SRC)/Integration.cc $(R_INCL)/Integration.h \
$(R_OSRC)/Utilisateur.o $(R_OSRC)/Tampon.o \
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc) -o $(R_OSRC)/Integration.o\
$(R_SRC)/Integration.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/Utilisateur.o : $(R_SRC)/Utilisateur.cc $(R_INCL)/Utilisateur.h
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc) -o $(R_OSRC)/Utilisateur.o  $(R_SRC)/Utilisateur.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/Tampon.o : $(R_SRC)/Tampon.cc $(R_INCL)/Tampon.h  $(R_OSRC)/MesMath.o
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc) -o $(R_OSRC)/Tampon.o $(R_SRC)/Tampon.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/MesMath.o : $(R_SRC)/MesMath.cc $(R_INCL)/MesMath.h
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc) -o $(R_OSRC)/MesMath.o  $(R_SRC)/MesMath.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/Capteur.o : $(R_SRC)/Capteur.cc $(R_INCL)/Model.h
$(R_INCL)/Capteur.h $(R_OSRC)/Laser.o \
$(R_OSRC)/Camera.o\
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc) -o $(R_OSRC)/Capteur.o  $(R_SRC)/Capteur.cc -c\


#######-------------------------------------------------------------------------------
$(R_OSRC)/Laser.o : $(R_SRC)/Laser.cc $(R_INCL)/Model.h $(R_INCL)/Laser.h $(R_OSRC)/CaptBase.o
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_OSRC)/Laser.o  $(R_SRC)/Laser.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/CaptBase.o : $(R_SRC)/CaptBase.cc $(R_INCL)/Model.h \
$(R_INCL)/CaptBase.h $(R_OSRC)/Vue.o
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_OSRC)/CaptBase.o  $(R_SRC)/CaptBase.cc -c


#######-------------------------------------------------------------------------------
$(R_OSRC)/Vue.o : $(R_SRC)/Vue.cc $(R_INCL)/Vue.h
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_OSRC)/Vue.o  $(R_SRC)/Vue.cc -c

#######-------------------------------------------------------------------------------
$(R_OSRC)/LaserBal.o : $(R_SRC)/LaserBal.cc $(R_INCL)/Model.h\
$(R_INCL)/LaserBal.h $(R_OSRC)/CaptBase.o
$(CC) $(FLAG) $(R_INC) $(R_SOLinc)  -o $(R_OSRC)/LaserBal.o  $(R_SRC)/LaserBal.cc -c

#######-------------------------------------------------------------------------------
$(R_OSRC)/Camera.o : $(R_SRC)/Camera.cc $(R_INCL)/Model.h $(R_INCL)/Camera.h $(R_OSRC)/CaptVideo.o
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_OSRC)/Camera.o  $(R_SRC)/Camera.cc -c
```

```
#######-------------------------------------------------------------------------------
$(R_OSRC)/CaptVideo.o : $(R_SRC)/CaptVideo.cc $(R_INCL)/Model.h \
$(R_INCL)/CaptVideo.h $(R_OSRC)/CaptBase.o
$(CC) $(FLAG) $(R_INC) $(R_IRIXinc)  -o $(R_OSRC)/CaptVideo.o  $(R_SRC)/CaptVideo.cc -c



###############################################################################

#######-------------------------------------------------------------------------------
net : net1 net2 net3 net4 net5 net6

net1 :
cd include; \
rm *~;\
cd ..

net2 :
cd src; \
rm *~;\
cd ..

net3 :
rm *~;

net4 :
cd save; \
rm *~;\
cd ..

net5 :
cd mesh; \
rm *~;\
cd ..

net6 :
cd model; \
rm *~;\
cd ..

#######-----------------------------------------------------------
clean : clean0 cleanE

clean0 :
cd 0src; \
rm *.o; \
cd ..;

cleanE :
rm Exec;
```

Annexe B

Schéma de l'Intégration

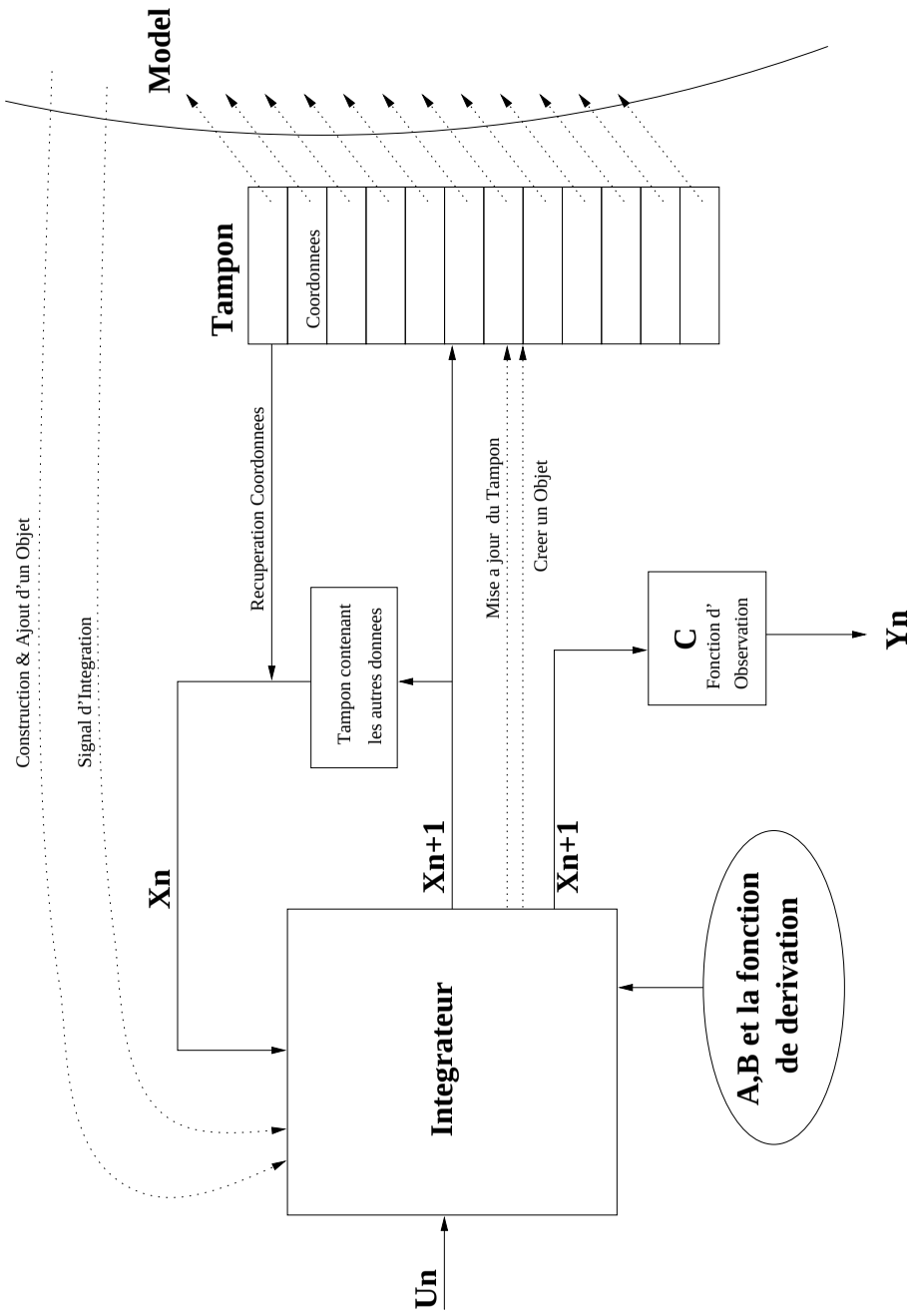

FIG. B.1 – *Integration et Tampon (grand format)*

Annexe C

Modèle cinématique du Cycab

(article de Brieuc DESOUTTER)

C.1 Introduction

Ce court article présente le modèle du Cycab de l'INRIA Rhône-Alpes. Dans une première section est présenté le modèle et ses paramètres et dans une deuxième section, on présente le calcul des vitesses pour les quatres roues.

C.2 Le modèle

Cette section présente le Cycab et son modèle géometrique.

C.2.1 Le Cycab

Le Cycab est un véhicule éléctrique autonome à quatre roues directrices déstiné devenir un moyen de transport publique individuel, i.e. un véhicule individuel mis à la disposition du public comme moyen de transport. Différentes fonctionnalités sont envisagées :

la conduite automatique déchargeant l'utilisateur de cette activité et lui permettant de se consacrer à autre chose ;

la conduite en train qui permet à un opérateur seul de rappatrier plusieurs véhicules ; Le Cycab poursuivant alors une cible infrarouge fixée sur le Cycab de devant ;

la consuite manuelle permettant de piloter le Cycab avec un joystick.

Afin de réaliser ces fonctionnalités, un modèle précis du Cycab est nécéssaire. Nous allons le détailler dans la section qui suit.

C.2.2 Le modèle géométrique

Le modèle du Cycab est présenté sur la figure 1 :

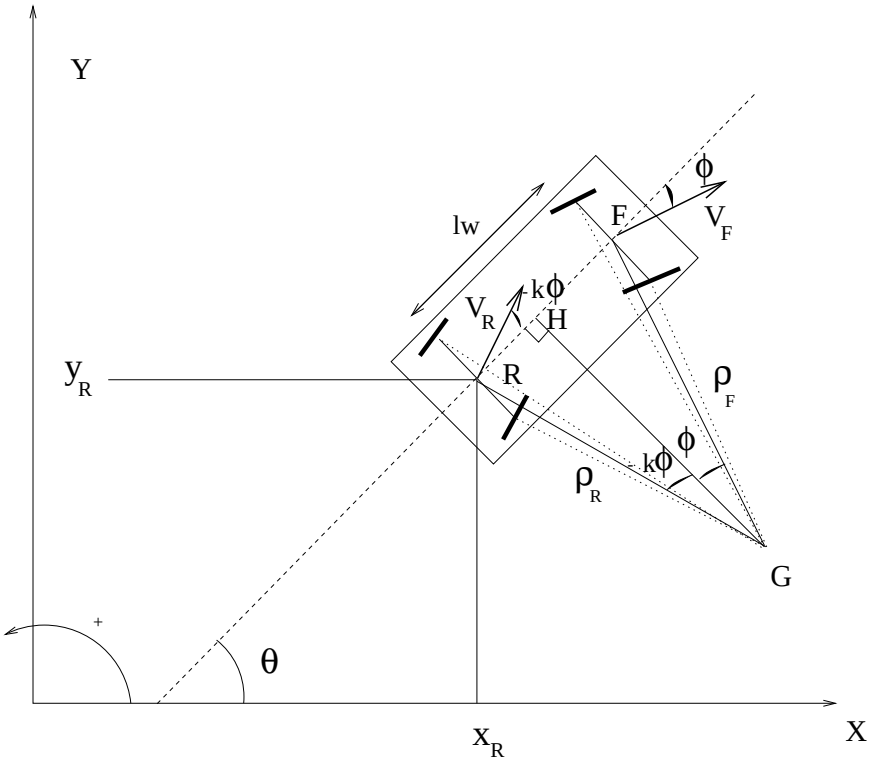

FIG. C.1 – *modèle du Cycab*

À l'aide de ce modèle géométrique, on détermine le modèle cinématique que nous allons détailler dans la section suivante.

C.2.3 Le modèle cinématique

Un calcul géométrique (on ne prend pas en compte le signe des angles) donne :

$$RH = \frac{l_w}{1 + \frac{\tan(\phi)}{\tan(k\phi)}} = l_w \frac{\cos(\phi) * \sin(k\phi)}{\sin(\phi + k\phi)}$$

et

$$HF = \frac{l_w}{1 + \frac{\tan(k\phi)}{\tan(\phi)}} = l_w \frac{\cos(k\phi) * \sin(\phi)}{\sin(\phi + k\phi)}$$

Les rayons de braquages ρ_R et ρ_F sont donc :

$$\rho_R = \frac{RH}{|\sin(k\phi)|} = l_w \frac{\cos(\phi)}{|\sin(\phi + k\phi)|}$$

et

$$\rho_F = \frac{HF}{|\sin(\phi)|} = l_w \frac{\cos(k\phi)}{|\sin(\phi + k\phi)|}$$

Ainsi, la vitesse de rotation instantannée s'écrit :

$$|\dot{\theta}| = \frac{v_R}{\rho_R} = \frac{v_F}{\rho_F}$$

Nous allons maintenant déterminer le signe de $\dot{\theta}$ en considérant que :

$$\frac{d\overrightarrow{RF}}{dt} = \overrightarrow{\Omega} \wedge \overrightarrow{RF}$$

avec, dans le repère inertiel :

$$\frac{d\overrightarrow{RF}}{dt} = \begin{pmatrix} \dot{x_F} - \dot{x_R} \\ \dot{y_F} - \dot{y_R} \\ 0 \end{pmatrix}, \overrightarrow{\Omega} = \begin{pmatrix} 0 \\ 0 \\ \dot{\theta} \end{pmatrix} \text{ et } \overrightarrow{RF} == \begin{pmatrix} l_w * \cos(\theta) \\ l_w * \sin(\theta) \\ 0 \end{pmatrix}$$

D'autre part dans le repère inertiel, les vitesses de R et de F s'écrivent en tenant compte des angles orientés :

$$\begin{cases} \dot{x_R} & = & v_R * \cos(\theta - k\phi) \\ & = & v_F * \frac{\cos(\phi)}{\cos(k\phi)} * \cos(\theta - k\phi) \\ \dot{y_R} & = & v_R * \sin(\theta - k\phi) \\ & = & v_F * \frac{\cos(\phi)}{\cos(k\phi)} * \sin(\theta - k\phi) \end{cases}$$

$$\begin{cases} \dot{x_F} & = & v_F * \cos(\theta + \phi) \\ \dot{y_F} & = & v_F * \sin(\theta + \phi) \end{cases}$$

D'où :

$$\begin{pmatrix} v_F * \left(\cos(\theta + \phi) - \frac{\cos(\phi)}{\cos(k\phi)} * \cos(\theta - k\phi)\right) \\ v_F * \left(\sin(\theta + \phi) - \frac{\cos(\phi)}{\cos(k\phi)} * \sin(\theta - k\phi)\right) \\ 0 \end{pmatrix} = \begin{pmatrix} -l_w \dot{\theta} * \cos(\theta) \\ l_w \dot{\theta} * \sin(\theta) \\ 0 \end{pmatrix}$$

On en déduit alors :

$$\dot{\theta} = v_F * \frac{\sin(\phi + k\phi)}{l_w * \cos(k\phi)}$$

On peut ensuite en déduire la vitesse des 4 roues.

C.3 La conduite manuelle

C.3.1 Les consignes

La conduite manuelle se fait à partir d'un joystick qui fournit une consigne de vitesse de rotation à la roue virtuelle avant soit ψ_F, et une consigne de direction ϕ dont le signe positif est défini par le sens trigonométrique direct (inverse des aiguilles d'une montre) ce qui assure que ϕ et θ sont du même signe. Les consignes sont donc :

$$\psi_F = \frac{v_F}{r} \text{ et } \phi$$

On en déduit directement :

$$\psi_R = \psi_F * \frac{\cos(\phi)}{\cos(k\phi)}$$

C.3.2 Les sorties

Les sorties sont les vitesses de rotation des 4 roues que nous allons calculer. Dans le repère lié au cycab, on a :

$$\overrightarrow{v_{RL}} = \overrightarrow{v_R} + \overrightarrow{L_R R} \wedge \overrightarrow{\Omega}$$

où

$$\overrightarrow{L_R R} = \begin{pmatrix} 0 \\ -\frac{L}{2} \\ 0 \end{pmatrix} \text{ et } \overrightarrow{v_R} = \begin{pmatrix} v_R * \cos(-k\phi) \\ v_R * \sin(-k\phi) \\ 0 \end{pmatrix}$$

D'où :

$$\overrightarrow{v_{RL}} = \begin{pmatrix} v_R * \cos(-k\phi) - \frac{L\dot{\theta}}{2} \\ v_R * \sin(-k\phi) \\ 0 \end{pmatrix}$$

et

$$v_{RL} = \sqrt{v_R^2 + \left(\frac{L\dot{\theta}}{2}\right)^2 - L\dot{\theta} * v_R * \cos(k\phi)}$$

D'où :

$$|\psi_{RL}| = \sqrt{\psi_R^2 + \left(\frac{L\dot{\theta}}{2r}\right)^2 - \frac{L\dot{\theta}}{r} * \psi_R * \cos(k\phi)}$$

De même :

$$v_{RR} = \sqrt{v_R^2 + \left(\frac{L\dot{\theta}}{2}\right)^2 + L\dot{\theta} * v_R * \cos(k\phi)}$$

et :

$$|\psi_{RR}| = \sqrt{\psi_R^2 + \left(\frac{L\dot{\theta}}{2r}\right)^2 + \frac{L\dot{\theta}}{r} * \psi_R * \cos(k\phi)}$$

En remplaçant v_R, ψ_R, $\cos(k\phi)$ respectivement par v_F, ψ_F, $\cos(\phi)$ dans les formules de ψ_{RL} et de ψ_{RR} on obtient les formules de ψ_{FL} et de ψ_{FR}. Le signe des vitesses de chaque roues est déterminé par le signe de la consigne de vitesse : + pour la marche avant, - pour la marche arrière.

Annexe D

Capteurs ultra son

Cette figure représente les 16 capteurs à ultrason disposés sur le cycab de manière à couvrir au mieux la zone de détection désirée. On constate qu'il reste tout de même des zones non couvertes. Une utilisation possible de notre simulateur est de réaliser des essais de positionnement pour déterminer si un programme de contrôle-commande réagit mieux en déplaçant certains capteurs.

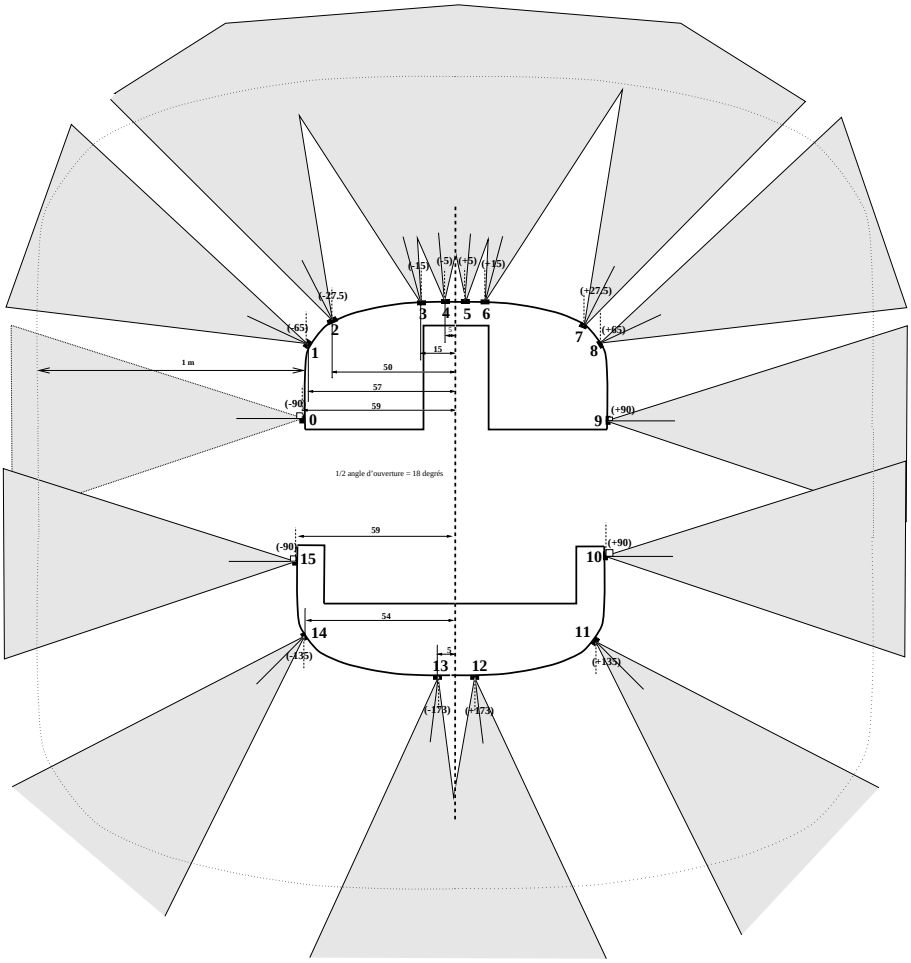

FIG. D.1 – *Ceinture de capteurs Ultra-son*

Annexe E

Capture d'écran du simulateur

La figure E.1 représente la vue du simulateur obtenue en chargeant le fichier d'exemple de l'annexe G.

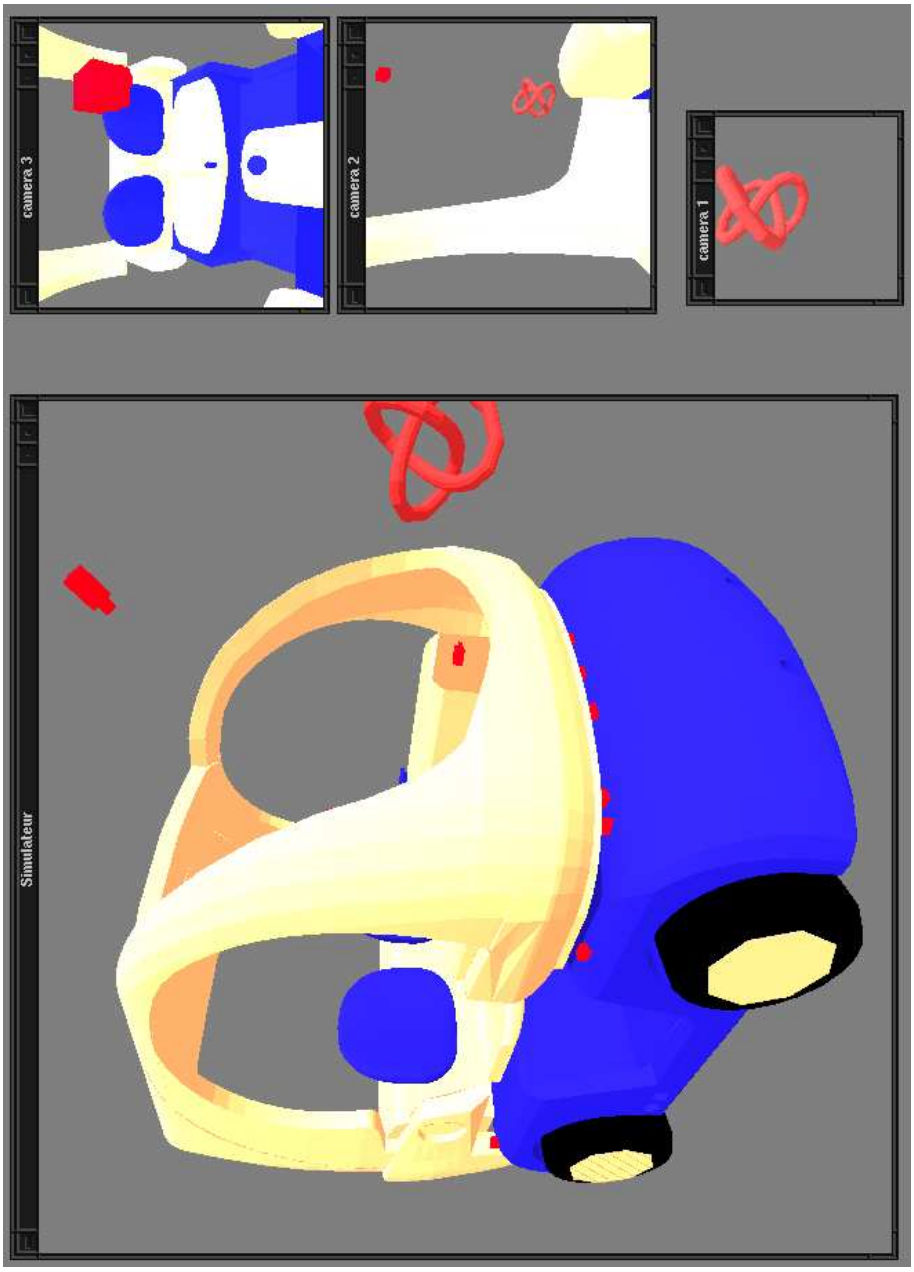

FIG. E.1 – *Capture d'écran du simulateur*

Annexe F

Exemple de maillage

Ce maillage simple est la représentation d'un cube.

```
Named Object: "cube"
Tri-mesh. Vertices: 8     Faces: 12
Vertex list:
Vertex 0: X: 1    Y: 1      Z: 1
Vertex 1: X: 1    Y: 1      Z: -1
Vertex 2: X: 1    Y: -1     Z: 1
Vertex 3: X: 1    Y: -1     Z: -1
Vertex 4: X: -1   Y: 1      Z: 1
Vertex 5: X: -1   Y: 1      Z: -1
Vertex 6: X: -1   Y: -1     Z: 1
Vertex 7: X: -1   Y: -1     Z: -1
Face list:
Face 0:    A:0 B:3 C:1 AB:0 BC:1 CA:1
Smoothing: 2
Face 1:    A:0 B:2 C:3 AB:1 BC:1 CA:0
Smoothing: 2
Face 2:    A:0 B:5 C:4 AB:0 BC:1 CA:1
Smoothing: 3
Face 3:    A:0 B:1 C:5 AB:1 BC:1 CA:0
Smoothing: 3
Face 4:    A:0 B:6 C:2 AB:0 BC:1 CA:1
Smoothing: 6
Face 5:    A:0 B:4 C:6 AB:1 BC:1 CA:0
Smoothing: 6
Face 6:    A:2 B:7 C:3 AB:0 BC:1 CA:1
Smoothing: 5
Face 7:    A:2 B:6 C:7 AB:1 BC:1 CA:0
Smoothing: 5
Face 8:    A:1 B:3 C:5 AB:1 BC:0 CA:1
Smoothing: 6
Face 9:    A:5 B:3 C:7 AB:0 BC:1 CA:1
Smoothing: 6
Face 10:    A:5 B:7 C:4 AB:1 BC:0 CA:1
Smoothing: 7
Face 11:    A:4 B:7 C:6 AB:0 BC:1 CA:1
Smoothing: 7
```

Annexe G

Exemple de fichier de sauvegarde

Annexe H

Exemple de fichiers utilisateur

H.1 Fichier `Utilisateur.cc`

Ce fichier est celui que l'utilisateur doit compléter. Il peut soit définir sa propre fonction de dérivation, soit en appeler une prédéfinie (Dans cet exemple le Rx90).

```
/*****************************************************************/
/**    Nom de la Classe : Utilisateur.cc                        */
/**    Auteur :  San Severino JC                                */
/**              Guilloud  C.                                   */
/**    Sujet : Variables et fonctions definissant               */
/**            le modele physique                                */
/*****************************************************************/
#include "../model/modelfree.cc"
#include "../model/modelrobmob.cc"
#include "../model/modelcycab.cc"
#include "../model/modelrx90.cc"
#include <iostream.h>

/* Variables */
const int nbTamptest = 7;      // rentrer ici le nombre de toutes les variables
                               // utilisees dans la conception des objets

const int nbTampUtil = 0;      // rentrer ici le nombre de variables suplementaires
                               // necessaires a l'integration

const int nbCommX = 2;         // rentrer ici le nombre de variables de commande
                               // necessaires a l'integration


/* Tableau de variables */
// RQ : les variables Xn sont dans un tableau construit par le programme
//      elles sont dans l'ordre de declaration du fichier.
double tabTamponUtil[nbTampUtil];    //Vous devez initialiser ce tableau
double tabCommX[nbCommX];            //Vous devez initialiser ce tableau


/* Fonction de derivation */
/** Fonction derive avec quatre parametres      **/
/**              Xn  : les coordonnees (tableau)**/
/**              tx  : sa taille                **/
```

```
/**              T   : le temps                 **/
/**              res : la derive au pt T de Xn  **/
/**                    en fonction de Uni       **/
void derive (double *Xn, int tx, float tps, double *res) {

  derive_rx90_cin (Xn,tx,tps,res);  // appel de la fonction du RX90
}


/*************************************************************************/
/**********************       Model predifini        *********************/
/**********************  Fonction derivee associee   *********************/
/*************************************************************************/

/* Fonction de derivation du model Avion */
void derive_Avion (double *Xn, int tx, float tps, double *res) {
  // nbTamptest = 6;
  // nbTampUtil = 0;
  // nbCommX = 6;
  // avion : fichier de sauvegarde
  int i;

  for (i=0;i<tx;i++) {
    res[i] = 0;
  }


  tabCommX[0] = 7*cos(tps);
  tabCommX[1] = 0;
  tabCommX[2] = 7*sin(tps);
  tabCommX[3] = 0;
  tabCommX[4] = 0;
  tabCommX[5] = 0;

  model_free(tabCommX,Xn,res,nbCommX,tx);
}

/* Fonction de derivation du model robmob */
void derive_robmob (double *Xn, int tx, float tps, double *res) {
  // nbTamptest = 3;
  // nbTampUtil = 2;
  // nbCommX = 2;
  // robmob : fichier de sauvegarde

  int i;

  // on redefinit Xn en le concatenant a tabTamponUtil
  double leX[(tx+nbTampUtil)];
  for (i=0;i<tx;i++) {
    leX[i] = Xn[i];
  }
  for (i=tx;i<(tx+nbTampUtil);i++) {
    leX[i] = tabTamponUtil[(i-tx)];
  }

  // On cre un tableau resultat de la meme taille
  double leRes[(tx+nbTampUtil)];
  for (i=0;i<(tx+nbTampUtil);i++) {
    leRes[i] = 0;
  }
```

```
  float montps = 4*tps;

  //  On initialise les commandes
  if (montps<10) {
  tabCommX[0] = 0;
  tabCommX[1] = 0;
  } else if (montps<20) {
  tabCommX[0] = 0;
  tabCommX[1] = 1;
  } else if (montps<30) {
  tabCommX[0] = 0;
  tabCommX[1] = -1;
  } else if (montps<40) {
  tabCommX[0] = 1;
  tabCommX[1] = 0;
  } else if (montps<50) {
  tabCommX[0] = 1;
  tabCommX[1] = 1;
  } else if (montps<60) {
  tabCommX[0] = 1;
  tabCommX[1] = -1;
  } else if (montps<70) {
  tabCommX[0] = -1;
  tabCommX[1] = 0;
  } else if (montps<80) {
  tabCommX[0] = -1;
  tabCommX[1] = 1;
  } else if (montps<90) {
  tabCommX[0] = -1;
  tabCommX[1] = -1;
  } else {
    cout << "La simulation est fini \n";
  }

  model_robmob(tabCommX,leX,leRes,nbCommX,(tx+nbTampUtil));

  // On repartit les resultats
  for (i=0;i<tx;i++) {
    res[i] = leRes[i];
  }
  for (i=tx;i<(tx+nbTampUtil);i++) {
    tabTamponUtil[(i-tx)] = leRes[i];
  }
}


/* Fonction de derivation du model RX90 cinematique */
void derive_rx90_cin (double *Xn, int tx, float tps, double *res) {
  // nbTamptest = 6;
  // nbTampUtil = 0;
  // nbCommX = 6;
  // rx90 : fichier de sauvegarde

  int i;

  for (i=0;i<tx;i++) {
    res[i] = 0;
  }

  tabCommX[0] = 0;
  tabCommX[1] = 0;
  tabCommX[2] = 0;
```

```
  tabCommX[3] = 0;
  tabCommX[4] = 0;
  tabCommX[5] = 0;

  if(tps < 60) {
    tabCommX[(int (tps/10))] = 0.4;
  } else if ((tps-60) < 60) {
    tabCommX[(int ((tps-60)/10))] = -0.4;
  } else {
    cout << "La simulation est finie \n";
  }

  model_rx90(tabCommX,Xn,res,nbCommX,tx);
}
```

H.2 Fichier `modelrx90.cc`

Ce fichier représente le modèle du Rx90 et sa fonction d'observation.

```
//
// modelrx90: modele cinematique du Rx90 (6 liens rotoides)
// u  = (dq1 dq2 dq3 dq4 dq5 dq6), rotation en rad/s
// x  = (q1 q2 q3 q4 q5 q6)
// modele: dx = u
//
#include <iostream.h>
#include <math.h>
#define Pi                      3.14159265358979323846
#define RADTODEG(rad)           ((rad)*(180./Pi))
#define DEGTORAD(deg)           ((deg)*(Pi/180.))

// butees articulaires du rx90
#define _RX_JMAX1      DEGTORAD(160.0)
#define _RX_JMAX2      DEGTORAD(137.5)
#define _RX_JMAX3      DEGTORAD(142.5)
#define _RX_JMAX4      DEGTORAD(270.0)
#define _RX_JMAX5      DEGTORAD(120.0)
#define _RX_JMAX6      DEGTORAD(270.0)
#define _RX_JMIN1      (- _RX_JMAX1)
#define _RX_JMIN2      (- _RX_JMAX2)
#define _RX_JMIN3      (- _RX_JMAX3)
#define _RX_JMIN4      (- _RX_JMAX4)
#define _RX_JMIN5      (- DEGTORAD(105.0))
#define _RX_JMIN6      (- _RX_JMAX6)


#define UMIN_rx90 -1.0
#define UMAX_rx90  1.0


void model_rx90(double *u, double *x, double *dx, int usize, int xsize){

  double maxi_ang[6]={_RX_JMAX1,_RX_JMAX2,_RX_JMAX3,
      _RX_JMAX4,_RX_JMAX5,_RX_JMAX6};

  double mini_ang[6]={_RX_JMIN1,_RX_JMIN2,_RX_JMIN3,
      _RX_JMIN4,_RX_JMIN5,_RX_JMIN6};

  int i;
```

```
  for(i=0;i<usize;i++){
    // Saturation sur les commandes
    if (u[i] >  UMAX_rx90)
      u[i] =  UMAX_rx90;
    else
      if (u[i] < UMIN_rx90)
u[i] = UMIN_rx90;
    // Le modele dx = u
    dx[i] = u[i];
    // butees articulaires
    if (x[i] >  maxi_ang[i]) {
      if (dx[i] > 0) {
dx[i] = 0;
      }
    } else if (x[i] < mini_ang[i]) {
      if (dx[i] < 0) {
dx[i] = 0;
      }
    }
  }

  cout << "dx[1] : "<< dx[1] << "   x[1] : " << x[1] << "\n";
}

void obs_rx90(double *x, double *y, int xsize, int ysize){
  int i;

  for(i=0;i<xsize;i++)
    y[i] = x[i];

}
```

Annexe I

Lexique

Voici quelques mots utilisés dans ce rapport méritant une petite explication :

- **tampon de profondeur (depth buffer ou Z-buffer) :** Un tampon de profondeur et une zone tampon faisant correspondre a chaque points du tampon image sa coordonée selon l'axe Z (celui de la profondeur). Il est utilisé pour ordonner les objets de la scène 3D afin de gerer les recouvrements à l'affichage. determiner quels sont les objets les plus proche par une méthode mathématique classique de calcul de distance est beaucoup trop couteux en temps de calcul (et non linéaire).

- **Tampon image (Frame buffer) :** C'est la zone mémoire où sont stockées les informations sur les couleurs des «éléments d'image» (pixels).

- **Liste d'affichage (Display list) :** Une liste d'affichage est une forme pré-compilée d'un ensemble de fonctions OpenGL. Cette pré-compilation s'effectue au premier affichage de la liste et ainsi les affichages suivants sont considerablements accéleré. Les listes d'affichage ont également un énnorme intêret pour l'affichage en mode client-serveur d'OpenGL : la liste est compilée une fois sur le client puis envoyée au serveur. Lors des affichages suivants le serveur disposera déjà de la liste.

- **Maillage (Mesh) :** un maillage est une liste de sommets (définis par trois coordonnées) et de facettes (définies par trois sommets). C'est cette technique qui est utilisée en géneral pour représenter des objets en 3D. Elle est rapide mais peu precise. Une alternative est celle utilisée par les programmes de lancé de rayon : la définition des lieux géometriques des objets.

- **Double tampon (Double buffer) :** L'utilisation d'un double tampon pour l'affichage permet une plus grande fluidité des annimations produites par OpenGL. Un tampon est affiché pendant que l'autre

est rempli. On permute ensuite les deux tampons pour obtenir un affichage sans scintillement (du à l'effacement du tampon visible) de l'image. Cette technique est efficace car le temps de permutation de deux tampon et l'affichage du deuxième est beaucoup plus faible que le temps d'effacement d'un tampon.

Bibliographie

[BGhMG99] Gérard Baille, Phillipe Garnier, hervé Mathieu, and Roger Pissard Gibollet. Le cycab de l'inria rhônes-alpes. Rapport technique de l'INRIA, 1999.

[BIP] BIP. *http://www.inrialpes.fr/bip/*. Site Web de l'equipe BIP.

[DK88] Etienne Dombre and Wisama Khalil. *Modélisation et commande des robots*. éditions Hermès, 1988.

[J.K96] Mark J.Kilgard. *The OpenGL Utility ToolKit (GLUT)*. SGI inc., 1996.

[KF92] Renate Kampf and Chris Freazier. *Numerical Recipes in C*. Cambridge university press, second edition, 1992.

[l'I] l'INRIA. *http://www.inria.fr*. Site Web de l'Institut National de Recherche en Informatique et en Automatique.

[MES] MESA. www.mesa3d.org. Site Web de la bibliotheque graphique Mesa.

[Ope] OpenGL. www.opengl.org. Site Web de la bibliotheque graphique OpenGL.

[rKF99] renate Kampf and Chris Freazier. *OpenGL reference manual*. Addison-Wessley, second edition, 1999.

[Rob] Moyens Robotiques. *http://www.inrialpes.fr/iramr/pub/Welcome.html*. Site Web des Moyens Robotiques de l'INRIA.

[SBE91] Claude Samson, Michel Le Borgne, and Bernard Espiau. *Robot Control, The task function approach*. Oxford Science Publications, 1991.

[SGI] SGI. *http://www.sgi.com*. Site Web de Silicon Graphics.

[SHA] SHARP. *http://www.inrialpes.fr/sharp/*. Site Web de l'equipe Sharp.

[Str99] Bjarne Stroustrup. *C++*. CampusPress reference, 1999.

[WND99] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wessley, second edition, 1999.

Table des figures

Table des matières