Gestion d'une plate-forme de vidéo-surveillance à usage robotique: modélisation d'un environnement dynamique

Eric Boniface

Conservatoire National des Arts et Métiers

Centre régional Rhône-Alpes

Centre d'Enseignement de GRENOBLE

Mémoire présenté par

Eric Boniface

en vue d'obtenir

Le Diplôme d'Ingénieur C.N.A.M.
en Informatique

Gestion d'une plate-forme de vidéosurveillance à usage robotique. Modélisation d'un environnement dynamique

Soutenu le 30 mars 2006

Jury

Présidente :	Mme Véronique Donzeau-Gouge
Membres :	M. Eric Gressier
	M. Thierry Fraichard
	M. Jean-Pierre Giraudin
	M. Christian Laugier
	M. André Plisson

Conservatoire National des Arts et Métiers

Centre régional Rhône-Alpes

Centre d'Enseignement de GRENOBLE

Mémoire présenté par

Eric Boniface

en vue d'obtenir

Le Diplôme d'Ingénieur C.N.A.M.
en Informatique

Gestion d'une plate-forme de vidéosurveillance à usage robotique. Modélisation d'un environnement dynamique

---

Les travaux relatifs au présent mémoire ont été effectués sous la direction de Thierry Fraichard, au sein de l'équipe de recherche e-Motion, dirigée par Christian Laugier.
e-Motion est intégrée au laboratoire Gravir, UMR (Unité Mixte de Recherche) de l'INRIA, du CNRS, de l'INPG et de l'UJF.
Gravir - GRAphics, VIsion and Robotics - regroupe plusieurs équipes, dans les domaines de la synthèse d'image, de la vision et de la robotique.

Remerciements

Le diplôme d'ingénieur CNAM est une épreuve souvent longue et nécessitant une bonne dose de motivation. L'aboutissement est possible grâce au soutien de nombreuses personnes. Il n'est pas évident de toutes les remercier en quelques lignes.

Mes premiers remerciements s'adressent à Monsieur Christian Laugier, directeur de l'équipe e-Motion et professeur de robotique pour le CNAM Grenoble, pour m'avoir proposé ce sujet. Il m'a aussi accordé beaucoup de son temps par rapport à mes réflexions et prises de renseignements sur une poursuite éventuelle d'étude.

Il en va de même pour Monsieur Thierry Fraichard, chargé de recherche à l'INRIA, mon tuteur lors de ce stage. Il a su faire preuve de patience et m'a donné de nombreux conseils tant sur le stage et le mémoire que sur mes interrogations diverses.

Je tiens à exprimer ma gratitude à toutes les personnes du CNAM pour leur aide au long de ces années, entre autres Monsieur André Plisson, directeur du centre d'enseignement de Grenoble et Monsieur Jean-Pierre Giraudin, professeur à l'Université Pierre-Mendès France de Grenoble et responsable du cycle ingénieur CNAM en informatique à Grenoble. Ils m'ont tous les deux apporté une aide précieuse dans mon parcours.

Je remercie Madame Véronique Donzeau-Gouge, professeur au CNAM de Paris, qui me fait l'honneur de présider le jury, ainsi que Monsieur Eric Gressier, professeur au CNAM de Paris, membre de ce jury.

Mes remerciements vont aussi à l'équipe e-Motion avec qui j'ai passé une très bonne année qui m'a permis d'apprendre de nombreuses choses dans la bonne humeur (ha, les parties d'Awalé ou de ping-pong). Message pour le club de robotique : bonne chance pour 2006 !)

Mes remerciements vont aussi à mes camarades de l'AIPST, l'Association des Ingénieurs de la Promotion Supérieure du Travail-CUEFA/CNAM. Christine, Albino, JB, Thierry et ceux qui nous ont rejoint, merci de votre aide pour mon mémoire et de votre confiance pour l'association.

Ce rapport est aussi passé dans les mains de mes meilleurs amis qui ont pu y consacrer un peu de leur temps et je les en remercie !

Enfin, je tiens à exprimer mes plus tendres remerciements à ma femme et mon fils qui m'ont supporté pendant mon cursus. Je sais que je n'ai pas toujours été facile à vivre et ils ont su faire preuve de patience, tout en m'apportant leur amour. Merci à vous !

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Contexte : modélisation de l'environnement

L'automobile et - de manière plus large - le transport routier ont évolué rapidement ces dernières années, grâce aux avancées technologiques en matière d'électronique, d'informatique et de robotique.

Qui n'a pas révé un jour d'avoir une voiture se conduisant toute seule, se garant aussi de manière autonome, éliminant ainsi la joie toute relative de faire un créneau ?

Nous parlons bien là de la conduite automatique d'un véhicule, vaste sujet de recherche depuis plusieurs années. Diverses solutions ont déjà été envisagées voire testées, telles que la conduite en convoi (projet Praxitèle, INRIA) utilisant des voies routières dédiées à des files de véhicules automatisés, ou bien la conduite automatique sur des sites protégés (projet ParkShuttle par exemple), pour ne citer que ceux là.

Pour que le véhicule puisse se déplacer de manière automatique, il va devoir être capable de se localiser dans l'espace dans lequel il évolue et avoir connaissance de son environnement, grâce à une modélisation de ces différents éléments.

Pour se localiser, le véhicule automatique doit posséder des « sens », qui vont ainsi l'aider à savoir où il est. Ces « sens » vont aussi lui permettre de « voir » ce qui l'entoure : obstacles fixes, obstacles mobiles, détection de repères connus ou non, ...

Le véhicule que nous allons utiliser, le Cycab, possède plusieurs types de capteurs : intéroceptifs, proprioceptifs ou extéroceptifs. Les données intéroceptives nous fournissent des informations propres au fonctionnement interne du véhicule, comme par exemple la charge de la batterie. Le deuxième type permet d'obtenir des informations sur la condition du robot par rapport à son environnement, comme par exemple, le déplacement relatif effectué par rapport à une origine. Les capteurs extéroceptifs permettent d'avoir connaissance du monde extérieur au véhicule, on retrouve par exemple des caméras ou des capteurs laser.

Les capteurs peuvent être soit embarqués dans le véhicule, c'est le cas du laser qui équipe le Cycab ou bien débarqués, comme par exemple des caméras extérieures posées sur le parking. Une utilisation conjointe des deux types de capteurs est possible mettant en jeu dans ce cas une fusion des données de perception.

Grâce à ces capteurs, nous allons être capable de modéliser l'environnement, ce qui revient à dire construire un modèle incluant différents types d'objets : mobiles, immobiles, connus ou inconnus a priori. Le véhicule robot va ainsi pouvoir recevoir toutes les informations pertinentes pour son déplacement dans le contexte d'évolution. La localisation couplée à la modélisation va permettre de planifier et prendre des décisions pour aboutir à la conduite automatique.

Les travaux présentés dans ce rapport sont rattachés au projet Parknav, que nous présenterons plus loin, dont l'objectif est l'automatisation de la conduite d'un véhicule se déplaçant dans un environnement dynamique équipé d'un système de perception.

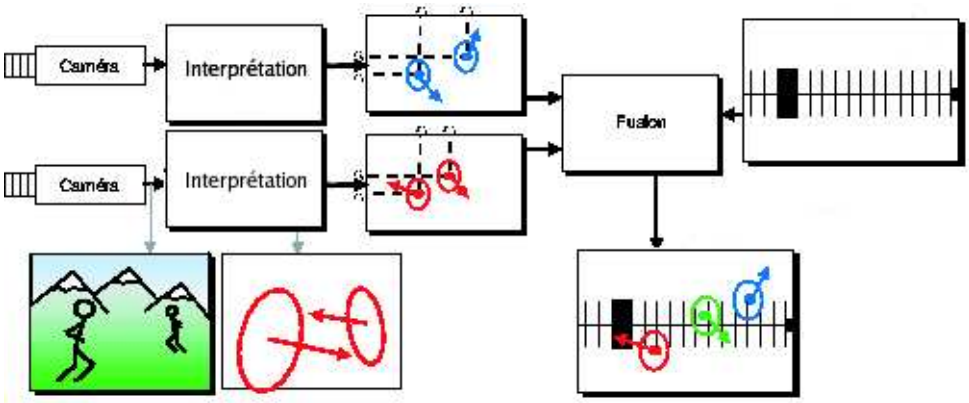

Fig. 1.1 – Principe de l'architecture

Parkview : la plate-forme expérimentale

Le but de cette plate-forme est de fournir les informations en temps réel sur l'environnement, en l'occurrence le parking arrière de l'INRIA à Montbonnot, qui a été équipé de caméras vidéo. Parkview est l'association d'éléments divers tant matériels (caméras, serveurs,. . .) que logiciels (serveur de cartes, outils divers, . . .).

Ce système de perception débarqué a été présenté dans les rapports d'activité du projet [Fra03] [Fra04] (voir aussi la réponse à l'appel à proposition [Fra02]). Sa mise en place a démarré lors du stage ingénieur CNAM de Frédéric Hélin [Hel03] en 2002.

En plus de ces aspects, Parkview a aussi un rôle d'outil d'expérimentation, permettant ainsi à d'autres équipes de tester leurs travaux dans le domaine de la vision ou de la robotique.

Le serveur de cartes

La modélisation de l'environnement est le rôle du serveur de cartes dynamiques. Ce dernier doit permettre la reconstruction du contexte en fournissant une carte générée en temps réel. Il s'agit de l'application centrale de Parkview, partie intégrante du projet Parknav, cœur de la centralisation et du traitement des informations reçues des capteurs (principalement les caméras).

Ce serveur devra permettre à tout type de client de récupérer en temps réel la « photo » de l'environnement, autrement dit une reconstruction de la scène dynamique, avec ses différents éléments constitutifs.

Le principe est repris par la figure 1.1 : le signal de capteurs débarqués - principalement des caméras - est traité afin d'interpréter ce qui se passe dans l'environnement. Ensuite, il y a un processus de fusion des données, nécessaire car nous utilisons plusieurs capteurs nous retournant des informations complémentaires. Enfin, la modélisation de l'environnement est réalisée en combinant le résultat de cette fusion avec les éléments connus a priori sur l'environnement.

C'est cette chaîne de traitement que nous allons présenter tout au long de ce rapport.

Travaux réalisés

Cette année d'étude a porté sur la remise à plat de l'architecture de Parkview, sur les aspects logiciels et matériels. L'axe le plus important étant le développement complet du serveur de cartes, fournissant les services attendus par le projet Parknav.

Les travaux réalisés dans [Hel03] ont pu servir de base à l'étude présentée dans ce document, cependant, ils n'ont pas été déployés et maintenus ne suivant pas ainsi l'évolution des éléments de la plate-forme.

Dans les deux années qui ont suivi la mise en place, plusieurs évolutions ont eu lieu sur le matériel et sur les logiciels, les partenaires du projet Parknav ayant développé de nouveaux outils et applications. Par conséquent, il a fallu revoir complètement les principes de l'infrastructure logicielle proposée dans [Hel03].

Après un inventaire de l'existant, le point principal fut le développement du nouveau serveur de cartes, respectant les contraintes temps-réel, l'intégration des applications de nos partenaires, les capteurs, ... La mise au point du serveur a nécessité aussi la création d'un outillage logiciel adéquat afin d'assurer un fonctionnement correct.

En parallèle, un ensemble d'actions ont du être menées sur la partie matérielle soit pour répondre à des problèmes techniques, soit pour suivre les évolutions des technologies.

Plan de lecture

Cette étude s'inscrit dans le cadre de l'équipe e-Motion et comme nous l'avons vu plus haut dans le contexte particulier de Parknav.

Le prochain chapitre présente le contexte global, ce sera l'occasion de présenter l'équipe e-Motion, de voir plus en détail le projet Parknav ainsi que les partenaires avec lesquels nous avons travaillés. La plate-forme Parkview sera présentée ainsi que ses objectifs.

Le chapitre 3 aborde le cahier des charges pour les travaux sur la plate-forme et le serveur de cartes. En repartant des objectifs de ces deux éléments, ce chapitre pose les besoins, contraintes et points importants à intégrer dans le développement du serveur et les impacts éventuels sur l'infrastructure.

Comme nous l'avons vu, la plate-forme a été initialisée en 2002 et a depuis évolué. Le chapitre 4 présente son état des lieux au démarrage du stage, les composants en place et les contributions des différents intervenants depuis le démarrage. Ensuite la voiture automatique fera l'objet d'un chapitre dédié (chapitre 5) car ses nombreuses spécificités méritent un chapitre à part.

L'état des lieux effectué, le chapitre 6 présente une synthèse de nos apports et réalisations qui seront abordés en détail dans les chapitres suivants.

Nous présenterons les travaux sur la plate-forme elle-même, les mises à niveau ou ajout de matériel, dans le chapitre 7.

Les développements du serveur de cartes représenteront le chapitre le plus long de ce document. Effectivement, le chapitre 8 abordera tous les aspects qui ont conduit à la réalisation du serveur : le prototype, les modules, les aspects mise au point et mesures de performance.

Enfin nous aborderons dans la conclusion une synthèse du travail effectué ainsi que des points restants en suspens sur la plate-forme. Pour terminer, nous apporterons un commentaire personnel sur la réalisation de ce stage.

Chapitre 2

Contexte du stage

Ce mémoire est le résultat d'un stage d'ingénieur CNAM, dernière étape pour l'obtention du titre d'ingénieur du Conservatoire National des Arts et Métiers. Pendant un an, ce fut l'occasion de travailler au sein d'une équipe de l'INRIA, l'Institut National De Recherche en Informatique et en Automatique. Ce chapitre va présenter cette équipe, le projet sur lequel porte l'étude et de manière générale le contexte.

2.1 Le CNAM

Fig. 2.1 – Le CNAM

Cette section ne fera qu'une présentation rapide du CNAM, le Conservatoire National des Arts et Métiers, car nous ne souhaitons pas faire un copier/coller du site Web de l'organisme[1].

Composé de plus de 150 centres d'enseignement, dont certains hors de France, le CNAM permet à tout un chacun de se former tout au long de sa vie active, que ce soit pour l'obtention d'un diplôme ou simplement de nouvelles compétences.

Le centre de Grenoble[2] et ses 4 antennes (Annecy (74), Chambéry (73), l'Isle d'Abeau (38) et Valence (26)) figurent dans le palmarès du nombre de diplômés ingénieurs sortant chaque année. Ce centre d'enseignement, créé en 1974, a su faire reconnaître les diplômes dispensés par le CNAM auprès des entreprises de la région Dauphiné-Savoie. Il s'agit d'un acteur important pour ne pas dire incontournable dans le domaine de la formation pour la région.

[1] http ://www.cnam.fr. Dernière consultation : 17-nov-05

[2] http ://www.cnam.grenoble.free.fr/. Dernière consultation : 17-nov-05

2.2 L'INRIA

Fig. 2.2 – L'INRIA

En quelques mots - le site Web de l'INRIA[3] fourmille d'informations sur ce qu'est l'INRIA, il s'agit d'un institut public de recherches fondamentales et appliquées dans les domaines des STIC[4].

Composé de 6 unités de recherche ou UR, dont celle de Montbonnot, l'INRIA a de forts partenariats avec des universités, des grandes écoles ainsi que le CNRS, travaillant de concerts dans plus de 120 « projets » ou équipes de recherche. Un autre point fort de son fonctionnement porte sur les liens avec le monde industriel et la création d'entreprises dans le domaine des STIC, ce depuis maintenant 20 ans[5]. Le site de Montbonnot ne déroge pas à la règle, grâce à la création de plusieurs « start-up ».

Fig. 2.3 – L'UR de Montbonnot

Cette UR est organisée en cinq pôles de recherche dont les systèmes cognitifs ou bien les systèmes numériques. L'équipe e-Motion, dans laquelle s'inscrit ce travail, fait partie du pôle des systèmes numériques.

[3]http ://www.inria.fr. Dernière consultation : 17-nov-05
[4]Sciences et Technologies de l'Information et de la Communication
[5]http ://www.inria.fr/valorisation/20ansdestartup/index.fr.html. Dernière consultation : 17-nov-05

2.3 L'équipe e-Motion

FIG. 2.4 – L'équipe e-Motion

Comme dit ci-dessus, le stage s'est déroulé dans l'équipe de recherche e-Motion[6] - anciennement Sharp et Cybermove - dirigée par Christian Laugier.

Elle fait partie de l'unité mixte de recherche (UMR) GRAVIR - GRAphics, VIsion and Robotics - qui regroupe plusieurs équipes, dans les domaines de la synthèse d'image, de la vision et de la robotique. Cette UMR inclut l'INRIA, le Centre National de la Recherche Scientifique[7], l'Institut National Polytechnique de Grenoble[8] et l'Université J. Fourier[9].

FIG. 2.5 – Les partenaires de l'équipe e-Motion

e-Motion fait partie du pôle des systèmes numériques, ses axes de travaux portant sur la géométrie et les probabilités pour le mouvement et l'action. L'équipe développe des méthodes et des modèles afin de construire des systèmes artificiels ayant des capacités de perception, de décision et d'action, suffisamment évolués et robustes pour pouvoir opérer dans un environnement dynamique et ouvert.

L'application principale de ces recherches porte sur l'introduction de systèmes robotisés évolués et sécurisés dans notre espace de vie, avec l'objectif le confort et la sécurité des personnes dans leur quotidien. Les systèmes robotisés sont, par exemple, dans le domaine des transports avec de futures voitures « robotisées » ou bien des robots d'assistance ou d'intervention (tâches domestiques, actions militaires, . . .).

A signaler que l'équipe a donné lieu à la création de la start-up ProBayes[10]®, mettant ainsi sur le marché le fruit des recherches sur le thème du bayésien.

[6]http ://emotion.inrialpes.fr. Dernière consultation : 2-dec-05.
[7]http ://www.cnrs.fr/ Dernière consultation : 2-dec-05.
[8]http ://www.inpg.fr/ Dernière consultation : 2-dec-05.
[9]http ://www.ujf-grenoble.fr/ Dernière consultation : 2-dec-05.
[10]http ://www.probayes.com/ Dernière consultation : 2-dec-05.

Ces différents travaux sont intégrés à plusieurs projets de recherche, dont le projet français Parknav.

2.4 Le projet Parknav

2.4.1 Présentation générale

Parknav[11], fait partie de Robea[12] - Robotique et Entités Artificielles, programme interdisciplinaire du CNRS. Le projet devait se dérouler d'octobre 2002 à septembre 2005.

L'objectif de Parknav est "[...] l'automatisation de la conduite d'un véhicule évoluant au milieu d'obstacles mobiles [...] dans un site équipé d'un système de perception à base de caméras." [Fra03].

Nous retrouvons les points suivants :

- conduite automatique : le véhicule devra être capable de se déplacer de manière autonome dans son environnement,
- environnement dynamique : le contexte d'évolution de la voiture est dynamique, autrement dit, de nombreux obstacles mobiles peuvent interagir avec notre véhicule. Ces obstacles peuvent être des piétons ou d'autres véhicules par exemple,
- site équipé d'un système de perception : le contexte de navigation est « imposé » par Parknav, en l'occurrence un site équipé de caméras pour la perception de l'environnement et sa reconstruction informatique.

Ce qui semble facile de prime abord pose en fait plusieurs problèmes scientifiques, sur lesquels porte le projet.

2.4.2 Problématique scientifique

Interprétation de scènes complexes dynamiques

L'idée est de pouvoir détecter, identifier et suivre des obstacles mobiles dans l'environnement d'analyse, par le biais de matériels - des caméras - et de logiciels, ce en temps réel et de manière la plus robuste possible.

En combinant ces observations avec des données statiques sur l'environnement, connues a priori, et en utilisant les capteurs propres au véhicule, il est alors possible de reconstruire l'environnement, sous la forme d'une carte dynamique de ce dernier.

Planification de trajectoire

En s'appuyant sur l'environnement modélisé, il va être possible de procéder à la planification de mouvements afin de faire évoluer le véhicule en tenant compte des obstacles et des incertitudes.

[11] http ://emotion.inrialpes.fr/parknav. Dernière consultation : 17-nov-05

[12] http ://www.laas.fr/robea/index.html. Dernière consultation : 11-juil-05.

Cette opération de planification impose que l'interprétation soit la plus performante et robuste possible. Le temps réel est de mise.

Notre problématique

Dans toute l'étude qui suit, nous nous concentrerons uniquement sur la première partie : la plate-forme de perception et la modélisation de l'environnement pour l'interprétation des scènes. La conduite automatique n'étant pas la problématique traitée ici.

2.4.3 Partenaires

Nous pouvons remarquer que Parknav regroupe plusieurs disciplines différentes dont la vision et la robotique. C'est pour cela que plusieurs partenaires sont impliqués dans le projet afin d'aboutir à la planification réactive de mouvement en fonction des informations fournies par la perception embarquée et débarquée.

Les points vus dans le paragraphe précédent impliquent des domaines de compétences variés répartis chez les différens partenaires (cf. tableau 2.1).

Partenaires	Domaine Parknav
e-Motion, GRAVIR	Ses axes portent sur la planification de mouvement en environnement fortement dynamique et gestion de la plate-forme de perception
RIA, Laas-CNRS	La planification de mouvements, mais en environnement faiblement dynamique
PRIMA, GRAVIR	La détection et le suivi de personnes (tracker), véhicules et obstables de manière générale
MOVI, GRAVIR	Outils de calibration des caméras et identification des obstacles mobiles
LAGADIC, IRISA	L'interprétation de scènes avec des caméras embarquées de type orientables (pan-tilt).

Tab. 2.1 – Partenaires du projet Parknav

Dans la suite de ce rapport, nous parlerons régulièrement de *tracker* ou de *tracking*. Un tracker est un logiciel permettant de détecter et de suivre un objet en mouvement sur un flux vidéo.

Nous serons amenés dans la suite à travailler de concert avec ces différentes équipes que ce soit en tant que fournisseur de services, d'intégrateur de leurs développements ou bien en tant que coordinateur.

2.5 Parkview

La plate-forme de perception ParkView a été initiée en 2002 et a pour objectif principal de recueillir des informations sur l'environnement, de les centraliser, de les rendre disponibles voire d'appliquer différents traitements sur les données. Elle s'inscrit totalement dans le projet Parknav, afin de fournir le site équipé tel que vu dans la section 2.4.1 (page 8).

A noter que d'autres projets pourront aussi bénéficier de cette infrastructure pour des besoins différents de ceux exprimés par ParkNav. Les projets Puvame[13] ou Mobivip[14] sont des exemples de projets de l'équipe e-Motion potentiellement utilisateurs de la plate-forme. Les équipes partenaires, entre autres celles travaillant sur la vision, sont aussi utilisatrices de Parkview afin de recueillir des vidéos de test.

Le site qui a été choisi est le parking arrière de l'INRIA, dont quelques photos donnent un aperçu dans la figure 2.6.

Fig. 2.6 – Le parking arrière de l'INRIA

L'un des points importants de ce projet est la notion d'environnement dynamique, qui sous entend que nous pouvons avoir de nombreux obstacles mobiles, évoluant à des vitesses variées. Le parking nous permet tout à la fois d'avoir ce type d'environnement - avec les heures d'arrivée et de départ du personnel - tout en ayant la possibilité de bloquer certaines parties du parking afin de pouvoir effectuer des tests.

[13]http ://emotion.inrialpes.fr/puvame/ Dernière consultation : 17-nov-05.
[14]http ://www-sop.inria.fr/mobivip/ Dernière consultation : 17-nov-05.

L'environnement sera constitué des différents éléments du parking : objets mobiles ou immobiles à un instant t (voitures, piétons, cyclistes, ...), places de parking, trottoirs, bâtiments, ... L'ensemble de ces éléments est ce que nous appelons *le contexte de scènes dynamiques*.

Différentes caméras sont réparties sur le parking en tant que capteurs de perception. Elles permettent via des logiciels adaptés, présentés par la suite (chapitre 4, page 17), de détecter les objets mobiles. Les informations retournées par la plate-forme doivent aider à la modélisation de l'environnement en temps réel.

La suite de ce rapport va présenter les différents éléments constitutifs de Parkview, la plate-forme, leurs évolutions durant le stage et les travaux réalisés. L'accent sera mis sur le développement de l'outil logiciel principal : le serveur de cartes.

Chapitre 3

Cahier des charges

Ce chapitre va présenter le cahier des charges utilisé pour le développement du serveur et les évolutions de la plate-forme en général. Nous aborderons les objectifs exacts de ce travail de mémoire, les besoins et contraintes liés au contexte.

3.1 Besoins et contraintes

Plusieurs réunions avec Thierry Fraichard et les intervenants sur le projet ont permis de mettre en évidence les besoins et contraintes des développements à réaliser.

3.1.1 Contraintes liées à ParkNav et son serveur

Partenariat

Ce projet est un partenariat entre plusieurs équipes travaillant sur la vidéo ou la robotique. Chacune a en charge une partie dédiée en fonction de son domaine de prédilection, ainsi les logiciels de traitement du flux vidéo seront fournis par l'équipe Prima.

Il s'agit d'un tracker qui permettra de suivre les cibles, c'est-à-dire, un objet en mouvement constitué entre autres par ses coordonnées, sa vitesse, un identifiant unique. Ce logiciel est communiquant, capable de fournir par le biais du réseau les informations concernant les cibles.

Il est important de noter que nous sommes dépendant de l'équipe Prima pour le développement ou les évolutions de ce(s) logiciel(s), d'où la notion de contraintes.

Le Cycab

Le Cycab - abordé plus en détail dans le chapitre 5 - est le véhicule autonome utilisé pour notre plate-forme. Il sera utilisé à la fois en tant que client de l'architecture, ce qui permettra de faire de la conduite automatique. Mais aussi en tant que fournisseur de données ou capteur d'entrée au même titre que le couple caméra/tracker par exemple. Dans ce mode, le Cycab nous fournira sa position en temps réel dans l'environnement.

Performances et maintenabilité

La génération du modèle dynamique de l'environnement devra être faite en temps réel, surtout si nous tenons compte du fait qu'un robot mobile devra être client de la plate-forme, comme indiqué précédemment. Des outils de mesures de performances peuvent être étudiés afin de valider ce point. Le code doit être facilement extensible pour pouvoir rajouter de nouveaux modules ou fonctionnalités.

Devant le volume de données à traiter et le modèle client/serveur de l'architecture, l'utilisation d'outils de type profiling ou de mise en évidence de fuites mémoires, par exemple, sera un réel plus. La notion de robustesse est un critère important car le serveur de cartes doit pouvoir fonctionner sans interruption.

Démonstrations

Une première version doit être fonctionnelle assez rapidement, car de nombreuses démonstrations de la plate-forme sont prévues dans les mois suivant le démarrage du stage pour le rapporteur du projet, des industriels, d'autres universités, ...

Par exemple, fin février 2005 une première version devait être disponible, que ce soit sous la forme d'un prototype ou sur l'architecture finale. Il faut que la plate-forme soit « manipulable » facilement, rapidement et avec le plus de souplesse possible. Il est important de perdre le moins de temps possible à chaque fois en préparation.

3.1.2 Contraintes liées à Parkview

La plate-forme elle-même doit être constituée des éléments généraux listés dans le tableau 3.1, induits par les spécifications faites dans Parknav ([Fra02], [Fra03]).

Matériel	Le parking arrière de l'INRIA avec ses composants statiques (arbres, lampadaires, trottoirs, ...)
	Des capteurs, comme des caméras vidéo couvrant le parking
	Un ou plusieurs PC pour les logiciels
Logiciel	Le serveur de cartes et ses composants
	Les outils de gestion des caméras (Les caméras doivent être configurées et calibrées pour un fonctionnement correct)
	Les logiciels de *tracking* des partenaires

Tab. 3.1 – Eléments généraux de Parkview

L'usage de caméras est préconisé dans [Fra02] comme capteurs d'entrée. Le signal vidéo (analogique) devra être reçu par des PC tournant sous GNU/Linux grâce à des cartes d'acquisition. Une extension du nombre de caméras est possible, ce qui veut dire que l'infrastructure doit être extensible. Plusieurs types de caméras peuvent être utilisés : liaisons filaires ou sans fil, caméras orientable (pan-tilt) ou fixe, ...

En plus du flux caméra, des données en provenance d'une voiture type Cycab ou bien de tout engin équipé de capteurs adéquats pourront être gérées. De manière générale, il faut que l'architecture développée permette l'intégration de nouveaux flux d'entrée, le plus simplement possible, ce qui revient de nouveau à parler d'architecture ouverte et extensible, de type multi capteurs.

Il ne devra pas y avoir de point d'étranglement matériel faisant chuter les performances, entre autres les aspects réseau. Mais l'étude faite dans [Hel03] montre clairement que ce n'est pas un souci, la bande passante réseau étant supérieure à nos besoins.

La plate-forme est aussi utilisée pour d'autres projets que Parknav ou bien par d'autres équipes : Puvame, Mobivip, besoin de vidéo de tests, ... Elle doit permettre de faciliter les expérimentations et recherches dans les domaines de la vision et de la robotique, ce qui peut nécessiter d'être capable de récupérer les données à différents niveaux de traitement (des données brutes des capteurs jusqu'à la carte de l'environnement).

L'architecture développée devra être ouverte pour permettre d'intégrer de nouvelles caméras, de nouveaux tracker ou de nouveaux composants avec un minimum d'efforts.

3.1.2.1 Aspects logiciels à développer

Une fois le flux des capteurs d'entrée acquis, l'objectif logiciel est de construire la carte dynamique en temps réel, représentant l'environnement du parking : le serveur de cartes sera en charge de cette partie. Les différents modules à créer devront pouvoir être lancés depuis n'importe quelle machine - véhicule compris - ce qui implique un développement de type client/serveur.

Comme nous l'avons dit plus haut, il faut que la plate-forme soit facilement extensible et configurable. La possibilité de pouvoir changer la configuration des programmes au lancement voire en cours de fonctionnement serait un vrai plus - et n'implique pas une recompilation à chaque fois.

Types de traitement Le serveur de cartes devra être capable d'implémenter plusieurs types d'algorithmes, comme par exemple pour réaliser l'association de données ou la fusion des capteurs qui est rendue nécessaire par l'utilisation de plusieurs caméras ou capteurs d'entrée en général.

Il faudra mettre l'accent sur les performances de ces modules, car ils ne doivent pas pénaliser les performances globales du système. Des mesures seront à envisager afin de valider cet aspect crucial. Ces modules peuvent nécessiter l'usage de librairies dédiées aux calculs mathématiques (par exemple une librairie spécialisée dans les calculs probabilistes).

Applications clientes Un serveur n'a d'intérêt que s'il posséde des clients. Le premier à réaliser sera un client graphique afin de pouvoir visualiser la carte en temps réel ; ceci permettra de valider les traitements en amont et la bonne interopérabilité des différents composants. Il faudra au minimum une version avec affichage 2D.

D'autres applications utilisatrices de la carte dynamique pourront voir le jour en fonction du besoin des projets. Le Cycab ou d'autres robots mobiles devront s'interfacer avec le serveur de cartes, ce afin d'adapter leur comportement en fonction des changements de l'environnement du parking, objectif du projet ParkNav.

Pratiques à l'INRIA

La majorité des machines présentes à l'INRIA tournent sous GNU/Linux, c'est sous ce système d'exploitation que sera fait le développement. D'autre part, de nombreux projets sont développés en C++ [Sil98] et [Str97], alliant robustesse et performance en utilisant le paradigme objet.

Le développement sera fait potentiellement par plusieurs intervenants, aussi il est recommandé d'utiliser un système de gestion de versions, afin de permettre d'une part un développement collaboratif et d'autre part, de pouvoir faire des retours arrières simples en cas de probléme. L'équipe e-Motion utilise déjà un outil, en l'occurrence CVS.

Documentation

Un projet n'est pérenne que s'il est facilement maintenable et évolutif ; la documentation est un point important pour aboutir à ces deux qualités.

Il faudra mettre à disposition des équipes différents documents explicitant le fonctionnement des logiciels développés ou matériels mis en place. Ils pourront prendre différentes formes, que ce soit papier ou électronique (pages Web, par exemple sur le site Parkview[1]).

Une documentation de type doxygen[2] (voir l'annexe D, page 129) sera aussi un réel plus et impose l'écriture de commentaires structurés dans le code source.

[1] http ://emotion.inrialpes.fr/parkview/

[2] http ://www.stack.nl/~dimitri/doxygen/ Dernière consultation : 17-nov-05.

Chapitre 4

Parkview en novembre 2004

L'état des lieux a été l'occasion de faire une remise à plat complète de l'existant : quels furent les apports depuis la création de la plate-forme ? quels sont les problèmes sur le matériel non encore résolus ? est-ce qu'il y avait des évolutions prévues, non encore déployées ? quel est le statut des développements réalisés dans [Hel03] ? ...

Ce chapitre va présenter les différents éléments constitutifs de cette infrastructure au démarrage du stage, ainsi que les choix technologiques qui ont été faits lors des deux premières années du projet ParkNav. Nous allons voir d'une part que la plate-forme a évolué depuis sa création, et d'autre part, qu'un ensemble d'outils est déjà présent pour pouvoir faire le développement nécessaire, que ce soit matériel ou logiciel.

4.1 Le parking

Comme nous l'avons mentionné, l'environnement de Parkview est le parking arrière de l'INRIA. La figure 4.1 permet d'avoir plusieurs vues avec les différents bâtiments du parking (garage à vélo, halle robotique, ...).

Courant 2004, l'ingénieur expert travaillant pour le projet ParkNav a pu procéder à un relevé topographique du parking, en partant d'un plan Autocad© fourni par le service des S.G.[1]. De là, l'ingénieur a pu créer un fichier au format XML détaillé dans l'annexe A, figure A.1. En résumé, il s'agit d'un format dit 2D et demi (2.5D) dans la mesure où nous avons les coordonnées dans le plan du parking plus une information de hauteur pour l'objet ou le bâtiment décrit. Ce fichier nous sera très utile principalement pour « dessiner » le parking, la figure 4.2 nous donne un exemple de représentation 2D du parking.

[1] Services Généraux

FIG. 4.1 – Parkview : le parking arrière de l'INRIA

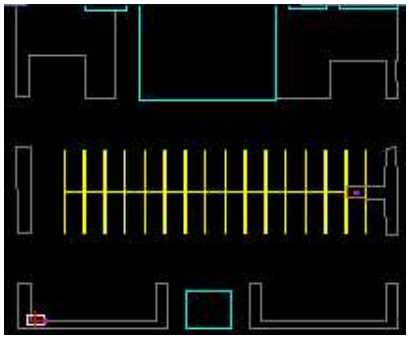

FIG. 4.2 – Exemple de représentation du parking

A noter que le fichier intègre une sémantique permettant de différencier les différents types d'objets statiques et ainsi de les représenter par des couleurs différentes.

4.2 Le matériel

Au démarrage de la plate-forme [Fra03] et [Hel03], il n'y avait que deux caméras en fonction (marque JVC, [JVC01]) avec un PC traitant le flux vidéo ; ceci constituait une première base de travail, cependant insuffisante pour répondre à la problèmatique de construction de l'environnement dynamique.

Pour répondre à différents besoins tels qu'une meilleure couverture du parking, le besoin de puissance supplémentaire au niveau des machines, l'évolution de l'infrastructure matérielle depuis [Hel03] (nouveaux serveurs, nouvelles caméras), etc. Courant 2003 et 2004, un ingénieur expert recruté sur le projet a pu installer 5 nouvelles caméras et un PC, en suivant les recommandations faites dans [Hel03].

Sur ces 5 caméras, nous pouvons noter la présence d'une pan-tilt ou caméra orientable, permettant de couvrir un angle de 180° via un pilotage à distance, ainsi que l'installation de caméras avec transmission sans fil.

Début novembre 2004 ([Fra04]), Parkview était constituée des éléments matériels listés dans le tableau 4.1.

2 PC sous GNU/Linux : machine Parkview et Carolus sous RedHat 9©
3 cartes d'acquisition sur Parkview
4 cartes sur la machine Carolus
4 caméras fixes et filaires réparties sur la halle robotique
2 caméras fixes sans fil sur le garage à vélos
une caméra fixe filaire de type Pan Tilt sur la halle

Tab. 4.1 – Eléments matériels de Parkview

Ces éléments matériels permettent ainsi de couvrir tout le parking. La situation est schématisée dans la figure 4.3, avec les différentes caractéristiques de l'équipement.

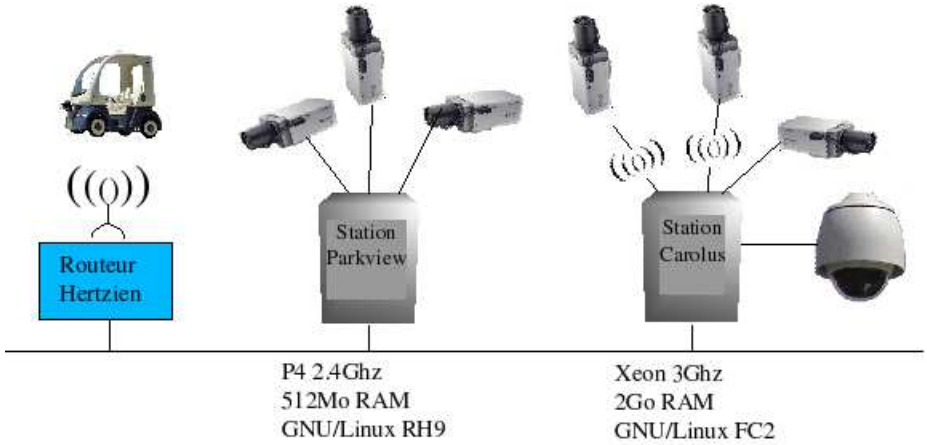

Fig. 4.3 – Infrastructure Parkview en novembre 2004

En plus des caméras d'origine, des JVC de référence TK-C1481 [JVC01][2], 4 caméras SONY DC598P[3] ont été installées. Le changement de modèle est du à l'obsolescence des JVC, dont la commercialisation a été arrêtée, mais les caractèristiques entre les deux modèles sont très proches.

Toutes les caméras sont équipées d'objectif grand-angle, ce qui a l'avantage de couvrir une surface plus importante, mais qui présente l'inconvénient de renvoyer des images avec distorsion, qu'il faudra bien sûr corriger. Cette distorsion s'explique par le fait que les angles d'incidence des rayons lumineux ne sont pas conservés lorsqu'ils passent par le centre optique de la caméra. La figure 4.4 montre l'impact sur une mire de test. La correction de distorsion va permettre d'obtenir une image perspective telle que fournie par un objectif classique.

Fig. 4.4 – Distorsion sur une mire

L'ajout d'une nouvelle machine permet maintenant de répartir la charge de traitement des flux vidéo entre 2 machines dédiées, sachant que les traitements effectués sont consommateurs en ressources système (mémoire et processeurs).

L'emplacement des 6 caméras fixes a été choisi de telle sorte que tout le parking arrière soit couvert en permanence, tout du moins la partie entre le garage à vélo et la halle robotique (voir les figures 4.5 et 4.6).

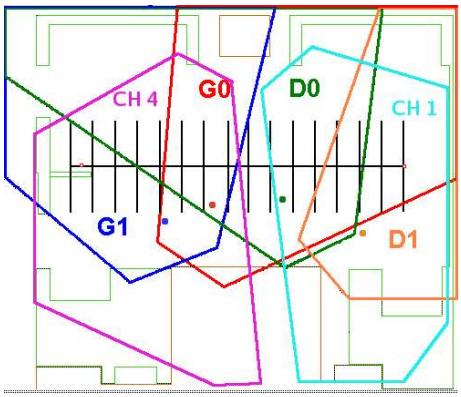

Fig. 4.5 – Zône de couverture des caméras

[2]http ://www.claravision.fr/Materiel/jvc/jvc.php. Dernière consultation : 17-nov-04
[3]http ://www.infodip.com/pages/sony/camera/ssc-dc598p.html. Dernière consultation : 17-nov-04

FIG. 4.6 – 3 caméras sur la halle robotique

Une fois positionnées, il a fallu calibrer les caméras, autrement dit, trouver la matrice qui permet de projeter l'image obtenue sur le plan du parking, c'est ce que l'on appelle la transformation homographique [Hel03].

La caméra pan-tilt n'est pour le moment pas utilisée dans la chaîne de traitement, car les logiciels de tracking ne savent pas contrôler cette caméra.

Cependant, elle sert régulièrement pour des prises de vue avec suivi de voitures par exemple, ce qui permet de générer des vidéos pour traitement a posteriori.

A noter que lors du démarrage du stage, nous avions un problème d'invasion de ... fourmis dans certaines caméras. Effectivement, étant au début de l'hiver, le caisson chauffé mais non étanche - ce fut une découverte - a permis à ces insectes de se réfugier au chaud dans le boîtier. N'ayant pas de moyen pacifique de se débarrasser de ces charmants insectes, il a fallu pulvériser un insecticide, ce qui a permis de régler le problème.

4.3 Le principe retenu

Un point important est le choix des éléments qui constituent le modèle à créer. Un découpage en trois types a été choisi :

- éléments statiques : tout ce qui est immobile (trottoirs, bâtiments, ...),
- éléments mobiles : les objets en mouvement,
- éléments semi-statiques : les objets immobiles à un instant t (exemple d'une voiture garée).

La figure 4.7 montre l'association de ces différents éléments pour créer le modèle du monde.

La figure 4.8 est une vue plus détaillée de la figure vue en introduction (cf. 1.1. Elle est tirée du premier rapport d'activité de ParkNav (fin de la première année) [Fra03] et montre le principe général retenu dès le début du projet.

De ces choix découle la connaissance a priori du parking, en l'occurrence toutes les parties immobiles tels que bâtiments, trottoirs, lampadaires, places de parking, ...

Les premiers maillons de la chaîne sont les capteurs en entrée : caméras ou tout autre capteur (le Cycab par exemple, voir chapitre 5). Les données sont alors traitées par un logiciel de type tracker, qui va analyser le flux en provenance des caméras, pour pouvoir détecter les objets mobiles.

Différents calculs seront faits sur les données (entre autres la correction de distorsion induite par les objectifs grand angle des caméras, voir la section 7.3).

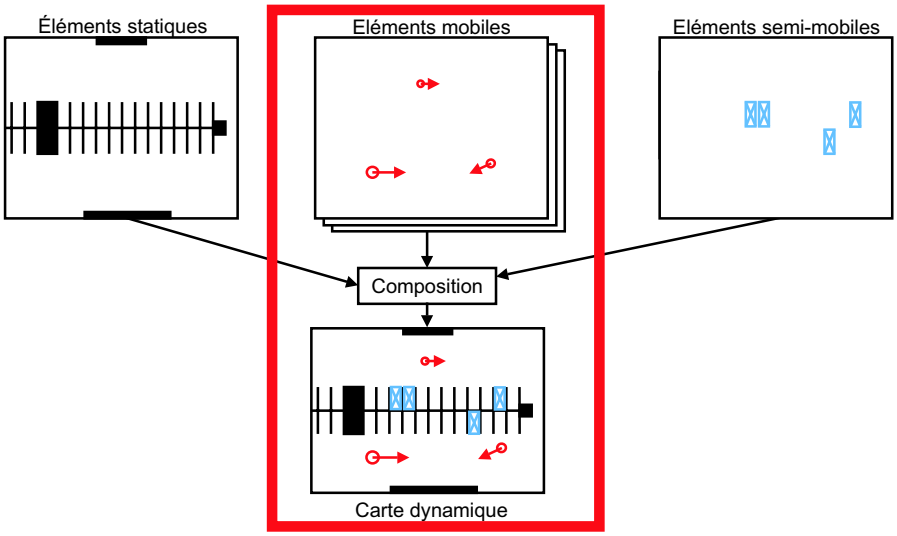

FIG. 4.7 – Carte dynamique composite [Fra03]

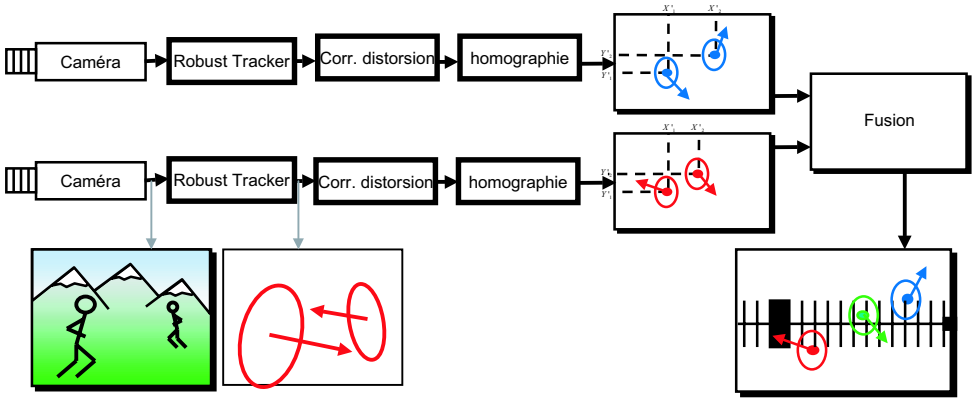

FIG. 4.8 – L'architecture globale retenue

Afin de traiter les données brutes, nous allons présenter maintenant les différents outils des partenaires du projet.

4.4 Les trackers

Leur rôle principal pour Parkview est de détecter, voire suivre, des cibles en mouvement dans un flux vidéo, puis de retourner par le réseau les informations relatives à ces cibles (coordonnées dans l'image, date et heure d'acquisition - timestamp, matrice de covariances, ...).

L'un des partenaires du projet ParkNav est l'équipe Prima. Cette dernière a fourni lors de la création de la plate-forme deux trackers que nous allons étudier dans ce qui suit, l'un nommé PrimaBlue et l'autre Primalab.

Il est à noter qu'il n'existait en novembre 2004 aucune documentation sur ces deux logiciels, d'où l'idée de faire un guide succinct d'installation et d'utilisation disponible en annexe B.

4.4.1 PrimaBlue

PrimaBlue est fourni sous la forme d'un code binaire avec des fichiers de configuration. Ceci veut dire que nous le prenons « tel quel », sans avoir la possibilité de le modifier directement.

Il s'agit d'un dérivé du logiciel Blue Behaviour© de la société BlueEyeVideo®. Cependant, bien que ce dernier évolue rapidement, le développement de PrimaBlue est stoppé. La version que nous avons utilisée pendant ce stage est celle figée de novembre 2004.

4.4.1.1 Fonctionnalités

Le tableau 4.2 liste les fonctionnalités de base de ce logiciel, telles que décrites dans [Hel03].

Un code binaire peut traiter plusieurs caméras.
Possibilité de lire une vidéo pré-enregistrée
Export des observations via le réseau (socket TCP)
Support du multi-caméras (définition de zones de recouvrement)
Export des données dans un fichier type XML
Ajout d'un paramètre pour normaliser l'intensité

Tab. 4.2 – Fonctionnalités de PrimaBlue

Le logiciel intègre le support du multi-caméras qui permet de définir des zones de recouvrement entre les caméras, pour ensuite faire le suivi des objets d'une caméra à l'autre. Durant cette étude, nous allons implémenter nos propres algorithmes d'association et de fusion de capteurs, rendant inutile le support du multi-caméras.

D'autre part, le paramètre de normalisation d'intensité a été rajouté afin de contourner les problèmes de variation lumineuse, point que nous aborderons un peu plus loin dans ce chapitre.

PrimaBlue permet de ne lancer qu'un seul programme pour pouvoir gérer plusieurs caméras sur une même machine, avec un maximum de 3 caméras sur une machine moyenne, et jusqu'à 4 sur une machine avec plus de puissance mémoire et processeur (bien entendu, il faut aussi une carte d'acquisition par caméra pour gérer le flux vidéo).

De même, PrimaBlue est normalement capable de calculer les coordonnées dans le référentiel du parking, mais nous préférons faire nos propres calculs afin de mieux maîtriser ce qui est fait. L'utilisation de PrimaBlue est détaillée en annexe B, afin de permettre à tout utilisateur de pratiquer l'outil.

La figure 4.9 est une capture d'écran du logiciel.

FIG. 4.9 – PrimaBlue

4.4.1.2 Suivi de cibles

La fonctionnalité de base du logiciel est la détection et le suivi d'objets en mouvement. La qualité de cette détection conditionne tout le reste de la chaîne de traitement de Parkview.

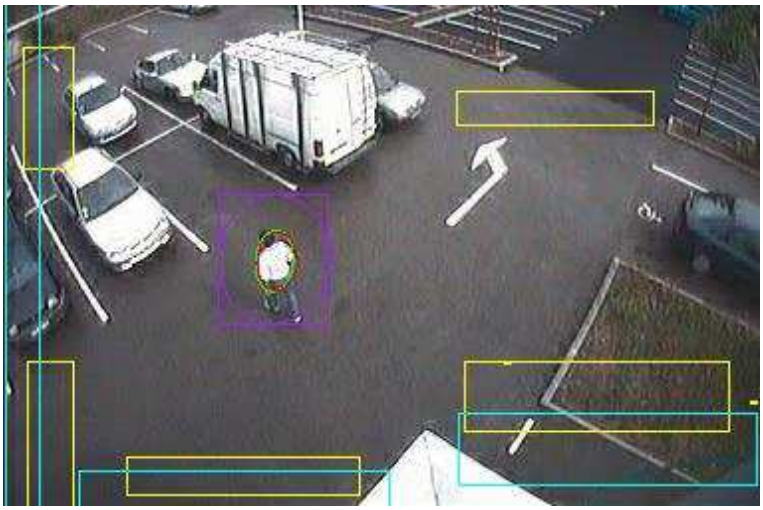

FIG. 4.10 – Flux vidéo avec zones de détection

La prise en charge d'une cible se produit lorsqu'un objet mobile traverse une zone de détection définie initialement dans l'image (zones jaunes sur la figure 4.10). A partir de là, PrimaBlue va faire le *tracking* ou suivi de la cible, jusqu'à ce qu'elle passe par une zone de fin de détection (zones bleues dans la figure 4.10) ou bien sorte de l'image.

Le suivi s'appuie sur différentes techniques de prédiction, principalement utilisant un filtre de Kalman [WB04] combinant les données à t-1 et celles observées à t. Cette technique permet d'avoir des coordonnées de positionnement (dans l'image) précises de l'objet.

Lors de ce suivi, le logiciel enverra les informations de la cible trackée (cf. tableau 4.3). Nous pouvons aussi distinguer sur la figure des ellipses et rectangles englobants qui ont des significations particulières. La plus significative pour nous est l'ellipse verte qui correspond à la cible en cours de suivi.

Libellé	Définition	Unité	Taille (Octets)
$Taille$	Taille du message	int	2
$Canal$	Identifiant de la caméra	int	1
$Timestamp$	Date de la mesure	$mseconde$	18
x_{img}	Abscisse de la cible dans l'image	int	4
y_{img}	Ordonnée de la cible dans l'image	int	4
xx	Param. ellipse englobante - x	$float$	4
yy	Param. ellipse englobante - y	$float$	4
xy	Param. ellipse englob. - angle	$float$	4
Id	Identificateur de la cible	$string$	variable
Total			$Taille$

TAB. 4.3 – Flux d'informations envoyé par *PrimaBlue*

4.4.1.3 Export réseau

L'une des fonctionnalités qui nous est nécessaire est la transmission des observations par le réseau, condition sine qua non pour une intégration avec le serveur de cartes.

PrimaBlue permet d'exporter les données via une connexion classique de type socket TCP. Le flux de données est repris dans le tableau 4.3. Par défaut, pour un objet mobile à faible vitesse, tel qu'un piéton, nous avons une fréquence d'envoi de l'ordre de 5 Hz.

4.4.1.4 Installation

PrimaBlue comporte plusieurs dépendances logicielles telles que des librairies à installer sur la machine. D'autre part, le programme lui-même n'est pas installé sur l'une des machines, mais sur le compte de l'utilisateur. Nous n'allons pas aborder ici le détail des prérequis, cependant l'annexe B contient le guide d'installation et d'utilisation de ce tracker.

4.4.2 PrimaLab

Connu aussi sous le nom d'Imalab, Primalab est un logiciel développé dans l'équipe Prima, mais contrairement au tracker vu précédemment, celui-ci est en plein développement, de nouvelles versions sortant régulièrement.

4.4.2.1 Fonctionnalités

Au démarrage du stage, nous avions la version 2 de Primalab. Elle permettait de faire une détection et un suivi de cibles avec un niveau de performance équivalent à celui de PrimaBlue. Il faut lancer un binaire Primalab par caméra à traiter, avec un maximum égal à celui de Primablue à savoir 3 caméras sur une machine moyenne, et jusqu'à 4 sur une machine avec plus de puissance mémoire et processeur.

Par contre, le gros inconvénient de cette version est l'inexistance d'un module de communication : Primalab v2 ne sait pas communiquer les informations de tracking sur le réseau. Du à cette lacune, cette version était peu utilisée, contrairement à PrimaBlue, ce qui explique que nous ne rentrerons pas dans les détails de ce tracker dans ce chapitre.

4.4.2.2 Installation

Tout comme pour PrimaBlue, un guide rapide pour l'installation de ce logiciel est disponible en annexe B (section B.3).

4.5 Logiciels

Les développements décrits dans [Hel03] n'ont pas été achevés et maintenus. Cependant, ils ont permis d'orienter les travaux faits fin 2004 afin de redévelopper le serveur de cartes dynamiques. Les développements faits fin 2002 et courant 2003 [Hel03] ont permis de tester le traitement d'un flux vidéo en entrée (caméra ou vidéo pré-enregistrée).

A l'époque, il n'y avait qu'un seul serveur disponible, aussi le logiciel développé, constitué de différents modules, ne tournait que sur cette machine. Le principe retenu pour la communication entre ces modules étaient une mémoire partagée. Nous verrons par la suite que ce choix a dû être remis en cause, car nous sommes passés à une architecture distribuée.

Au début du stage, le développement d'un prototype démarrait tout juste. Son rôle était de voir de manière rapide la faisabilité ainsi que les contraintes et problèmes potentiels de la future plate-forme logicielle. Le cahier des charges de ce prototype était relativement simple, puisqu'il ne s'agissait de traiter qu'une caméra et le Cycab en tant que capteurs d'entrée, sans traitement particulier sur les données reçues (association, fusion ou autre). Il fallait aussi faire un affichage en deux dimensions du résultat obtenu, afin de pouvoir valider les traitements effectués.

Cette architecture est représentée par la figure 4.11 avec PrimaBlue en tant que *tracker* et le Cycab en tant que capteur d'entrée.

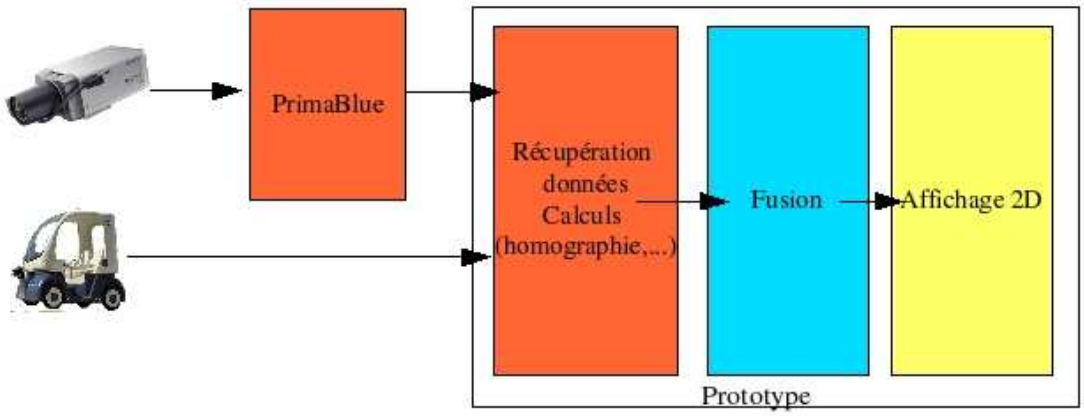

Fig. 4.11 – Architecture simplifiée du prototype

Nous avons consacré nos efforts dans un premier temps sur la finalisation de ce prototype, véritable base du développement du serveur de cartes.

Cette étape ne sera pas décrite en détail ici, car de nombreux points vus lors de ce développement ont été repris dans la nouvelle architecture ou complétement ré-écrits. Par conséquent, ils seront détaillés dans le chapitre 8.

4.6 Résumé

Au démarrage du stage, l'infrastructure matérielle est en place et permet de démarrer des développements en rapport avec le serveur de cartes. En plus des éléments que nous avons vus ici, nous avons aussi une voiture devant servir à terme pour la conduite automatique. Le développement d'un premier prototype - détaillé plus loin dans ce rapport (cf. 8.1) - nous a permis de comprendre les différents éléments matériels et logiciels de la plate-forme ainsi que les besoins réels.

Chapitre 5

Le Cycab

Ce chapitre va nous permettre de présenter le Cycab, le véhicule autonome utilisé à l'INRIA pour les tests de navigation par exemple.

5.1 Présentation du véhicule

Fig. 5.1 – Photo du Cycab

Le Cycab est un véhicule électrique commercialisé par la société Robosoft[1], basée à Toulouse, sur une idée initiale de l'INRIA.

L'objectif de ces petites voitures était de les destiner à une utilisation en libre-service dans des zones délimitées, en complément des transports en commun, telles que les zones de centre ville, d'où l'origine du nom : City Cab ou Cyber Cab[Her03b]. Dans ces dernières, l'idée d'une « voiturette » utilisable à la demande, permettant d'aller d'une station - ouverte 24h sur 24 - à une destination finale semble être la solution optimale en terme de rentabilité et de mise en œuvre.

A ce jour, les Cycab sont surtout utilisés en tant que voiture de tests pour les différents laboratoires de robotique, dont l'INRIA, avec toujours l'objectif, à terme, de flotte de véhicules en libre-service.

5.1.1 Caractéristiques du Cycab

De [INR98], [Mat03] et [BGMPG99], nous pouvons tirer les informations techniques principales listées dans le tableau 5.1.

[1] http ://www.robosoft.fr/. Dernière consultation : 22-juin-05

	Valeurs
Longueur	1,90 m
Largeur	1,20 m
Poids	300 Kg
Vitesse Max	30 Km/h
Trains de direction indépendants	2
Ordinateur	PC embarqué, GNU/Linux RTAI[a]
Réseau	sans fil[b]
Autonomie	2 h
Mode de conduite	Auto ou Manuel
Interaction	Ecran tactile et joystick

[a]module temps réel
[b]Réseau type 802.11 non compatible avec le Wifi « classique »

TAB. 5.1 – Caractéristiques principales du Cycab

La partie bas niveau est gérée par un logiciel nommé Syndex[2], véritable système d'exploitation traitant les noeuds matériels et les communications sur le bus interne CAN[3] du Cycab.

La configuration du robot sera déterminée à tout instant par le triplet (x, y, θ), autrement dit, la position du robot dans le repère du parking et l'orientation du robot.

5.1.2 Les Cycab de Montbonnot

A Montbonnot, il y a deux Cycab en état de marche sous la responsabilité des SED[4].

Plusieurs personnes ont contribué à développer des outils pour le bon fonctionnement du Cycab - s'appuyant sur le Syndex, mais il faut noter que Cédric Pradalier et Christophe Braillon sont les principaux contributeurs pour les logiciels haut niveau (au dessus du Syndex), comme décrit dans [Bra03], [Pra01], [PS02], [Her03a].

Pour différentes raisons de sécurité, la vitesse maximum a été descendue à 20 Km/h, ce qui est largement suffisant pour nos expérimentations.

Plusieurs projets de l'équipe e-Motion utilisent le Cycab, nous pouvons citer Puvame, Mobivip et bien sûr ParkNav.

Dans le cadre du projet ParkNav, le Cycab nous permet d'avoir un véhicule à conduite automatique, instrumenté de telle sorte que nous pouvons lui envoyer les ordres nécessaires à une bonne navigation ; ce, tout en récupérant sa position, comme nous allons le voir plus loin.

5.2 Les capteurs proprioceptifs/extéroceptifs

Les Cycab sont pourvus de capteurs proprioceptifs, à savoir des encodeurs incrémentaux fixés sur les roues et retournant les informations sur les déplacements, informations entachées d'erreur d'après les expérimentations de [Pra01]. Ce principe s'appelle l'odométrie. Le Cycab inclut aussi un gyroscope retournant l'orientation de la voiture.

[2]http ://www-rocq.inria.fr/syndex/. Dernière consultation : 24-nov-05.
[3]Controller Area Network
[4]Support Expérimentations et Développement logiciel

Du point de vue extéroceptif, les voitures possèdent un capteur laser Sick ou télémètre à balayage, de référence LMS-219, fixé à l'avant du véhicule. [Pra01] donne une description détaillée de ce capteur. Il permet d'obtenir un couple de mesures tous les demi-degrés sur un plan horizontal de 180° devant le Cycab.

Les informations en question sont la distance mesurée et l'intensité du faisceau réfléchi ; cette dernière valeur étant très significative lors de l'utilisation de balises avec matériaux réfléchissants (catadioptre) ou dans le cas d'un impact sur un phare de voiture par exemple.

La portée (les mesures sont fiables jusqu'à 20m) et la précision (à une telle distance l'erreur est de 7cm d'après le constructeur) en font un outil très performant pour procéder à la localisation du robot.

L'avantage du Sick par rapport à d'autres solutions (ultra-sons, infra-rouges, caméras, ...) est sa grande précision dans les mesures de distance et d'angles, avec une portée importante.

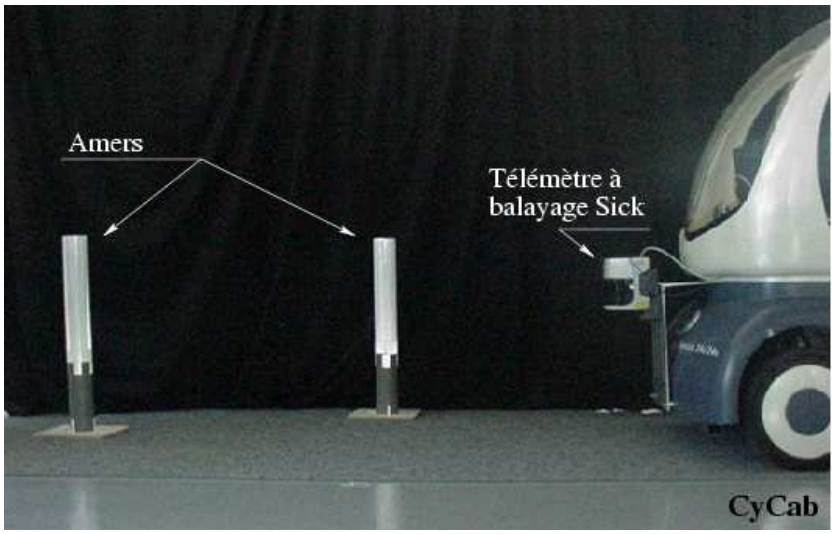

Fig. 5.2 – Le Cycab et deux balises pour la localisation

Nous disposons aussi d'un proximètre à ultra-sons, mais qui n'est pas à ce jour utilisé sur les voitures.

La figure 5.2 représente le Cycab avec 2 balises identiques à celles que nous utilisons pour nos démonstrations, pour la localisation de la voiture. Nous pouvons voir aussi à l'avant de la voiture le télémètre laser. Les amers ou balises sont des cylindres en PVC, sur lesquels ont été collés des surfaces hautement réfléchissantes (catadioptres). Ils mesurent 15 cm de diamètre pour 1 mètre de haut [Pra01].

5.3 SLAM : localisation relative

Sans rentrer dans le détail de ce qui a été fait [Pra01], [Pra04] et [Bra03], nous pouvons dire que le Cycab principal embarque un module de planification de mouvements avec évitement d'obstacles.

Le fonctionnement est schématisé dans la figure 5.3, qui met en évidence une localisation relative du Cycab par rapport aux balises.

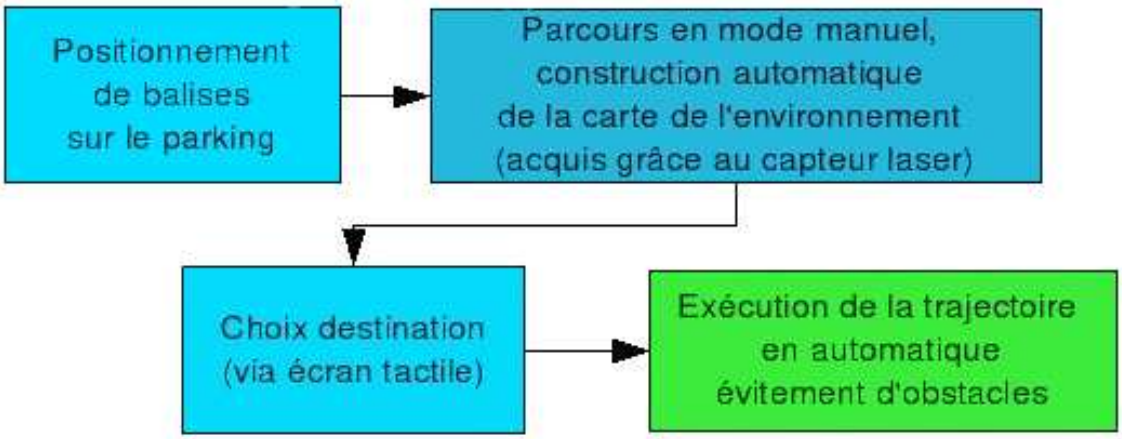

FIG. 5.3 – Principe localisation relative

Cette procédure paraît extrêmement simple, cependant la vraie difficulté est d'obtenir la position du Cycab à tout instant par rapport à la carte de l'environnement construite au préalable. C'est tout l'enjeu de la localisation, relative dans ce cas.

Pour que la planification et la réalisation de la trajectoire soit efficace, il faut une localisation optimale. Un problème connu dans la robotique mobile est celui du SLAM (Simultaneous Localization and Map building), autrement dit, la capacité de pouvoir se localiser tout en construisant la carte de l'environnement.

[Pra01] nous montre que pour réaliser cette tâche, il est nécessaire de fusionner les données du télémètre laser avec les capteurs proprioceptifs, en l'occurrence l'odométrie, qui permet d'obtenir le déplacement relatif du véhicule.

En maintenant au fur et à mesure de la progression du robot la carte de l'environnement à jour avec les balises, et en fusionnant les différents capteurs, le robot est non seulement localisé, mais il peut aussi faire des corrections dans sa trajectoire en fonction d'éléments extérieurs (obstacles).

Le SLAM est utilisé pour faire la reconnaissance des points d'impacts, sur le principe suivant :
- détection de 3 balises,
- calcul de l'aire du triangle,
- recherche en mémoire d'une aire égale,
- si un triangle de même aire existe, vérification des 3 côtés,
- si concordance, validation de la détection.

Dans ce mode de fonctionnement, le Cycab n'est pas interfacé avec Parkview, l'espace de travail est robot-centré et la voiture est autonome.

5.4 Le simulateur

Sortir le Cycab à chaque opération est lourd, contraignant voire impossible suivant la charge des batteries. C'est pour cela qu'un simulateur a été développé au sein de l'équipe afin de reproduire à l'identique le fonctionnement du Cycab :
- déplacement du Cycab via le clavier,
- retour de l'odométrie virtuelle,
- gestion d'un Sick virtuel identique au Sick réel,
- représentation à l'écran de l'environnement,
- visualisation des impacts du Sick,
- possibilité d'animer des objets dans l'environnement,
- envoi d'ordres pour conduite automatique,
- ...

Ce simulateur nous sera très utile tout au long du stage avant de tester en grandeur réelle nos programmes.

5.5 Résumé

Le Cycab est un véhicule très utile par rapport à notre besoin : capteurs précis, maniabilité, possibilité de le commander en automatique, ...

Le seul inconvénient à noter est une certaine lourdeur lorsque nous devons procéder à des tests en extérieur, entre autres, sur la sortie des balises. Cependant pour palier cela, le simulateur est un outil très efficace, reproduisant le comportement du robot et de ses capteurs.

Chapitre 6

Présentation de notre contribution

Après cette phase d'état des lieux, nous allons introduire dans ce chapitre les différents développements ou travaux décrits dans la suite de ce rapport. Nous allons introduire les problématiques que nous souhaitons aborder et les solutions proposées.

6.1 Préambule

Il est important de noter que les travaux qui vont être détaillés par la suite sont le fruit d'une ré-ingénierie d'un système existant, tant d'un point de vue matériel que logiciel. D'autre part, de nouveaux besoins sont apparus suite à l'étude [Hel03] induits, entre autres, par l'évolution du parking et par l'ajout de nouveaux matériels.

6.2 Travaux sur la plate-forme

Il n'y a pas eu d'évolution synchronisée entre la partie matérielle et la partie logicielle avant le stage, ce en grande partie car le serveur de cartes n'a pas été achevé. En conséquence, suite à l'état des lieux, la conclusion était que nous devions redévelopper de zéro un serveur de cartes avec ses modules. L'analyse des besoins et du cahier des charges de ce redéveloppement a permis de mettre en évidence des impacts sur l'infrastructure matérielle et sur les logiciels fournis par nos partenaires. Le chapitre 7 présentera les évolutions que nous avons apportées.

6.2.1 Mise à jour du matériel

Les machines de Parkview sont gérées par le service des M.I.[1]. Pour pouvoir conserver le support de cette équipe, il était nécessaire d'aligner les machines avec leurs recommandations, d'où une migration du système d'exploitation.

[1] Moyens Informatiques

D'autre part, nous avions besoin de plus de « puissance » pour faire tourner nos programmes et les trackers, c'est pour cela qu'en plus de rajouter une nouvelle machine, nous avons aussi augmenté la capacité mémoire de chacune d'elle.

6.2.2 Les caméras sans fil

Une partie du parking est traitée par des caméras sans fil, fixées sur le garage à vélo. Le sans fil a permis d'éviter de tirer des câbles entre les deux bâtiments principaux. Cependant, la liaison vidéo obtenue est de mauvaise qualité, impactant directement la chaîne de traitement (fausses cibles dans les trackers). Afin de résoudre ce problème, nous avons mené une étude complète sur cette liaison sans fil qui non seulement a permis de prouver que le matériel n'était pas adapté, mais d'autre part nous a fourni les informations nécessaires au remplacement de certains composants.

6.2.3 Calibration des caméras

L'utilisation des caméras positionnées en hauteur et équippées d'un objectif grand angle implique que nous obtenons des images déformées (distorsion). D'autre part, les observations faites sur ces images sont positionnées dans le repère image et non pas dans le repère parking.

Il a fallu procéder de nouveau à la calibration des caméras, autrement dit au calcul des coefficients propres à chaque caméra afin de pouvoir effectuer la correction de distorsion, ainsi que les projections homographiques (cf. figure 6.1).

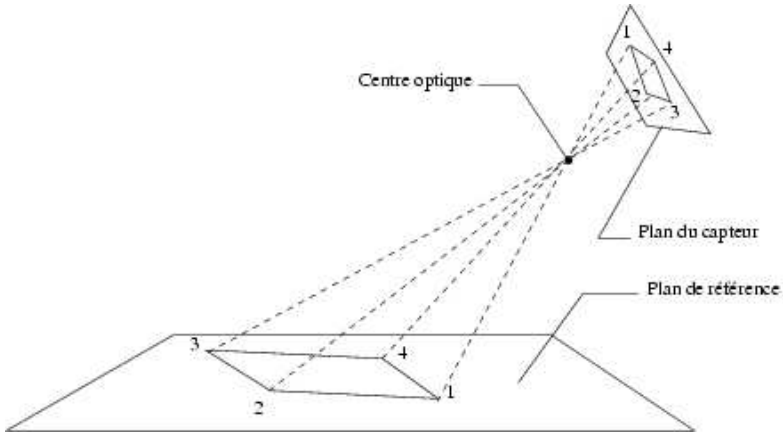

Fig. 6.1 – Projection centrale

6.2.4 Les *trackers*

Comme nous l'avons vu dans l'état des lieux, nous avions deux logiciels de tracking au démarrage du stage. Par contre, nous n'avions pas de données de comparaison entre les deux logiciels. C'est pourquoi nous avons procédé à une campagne de tests afin de mettre en concurrence les deux trackers.

L'équipe Prima nous a fourni à plusieurs reprises de nouvelles versions de PrimaLab, versions qu'il a fallu tester afin de voir si nous pouvions - ou non - les intégrer dans notre architecture. Au final, nous n'avons utilisé que PrimaBlue dans notre chaîne de traitement.

6.2.5 La localisation absolue

Le Cycab se branche sur le serveur de cartes afin de lui retourner sa position, étape nécessaire pour la modélisation de notre environnement. Une localisation absolue a été développée courant 2004 sur le Cycab, permettant d'obtenir sa position dans le plan du parking, ce en utilisant 2 balises de référence au début de chaque manipulation.

Nous avons très vite observé que cette localisation n'était pas fiable, car apportant une erreur de positionnement importante. C'est pour cela que nous nous sommes penchés sur un nouveau principe de localisation absolue qui utilise des balises fixes connues implantées sur tout le parking.

6.3 Le serveur de cartes

6.3.1 Le prototype

L'architecture que nous souhaitions mettre en place, comporte de nombreux éléments techniques qu'il faut pouvoir valider avant de démarrer le développement complet du serveur.

Le prototype - dont le développement démarrait tout juste au début du stage - a permis de valider un ensemble d'hypothèses : intégration du Cycab en tant que capteur, client graphique, utilisation de plus d'une caméra, ...

6.3.2 La nouvelle architecture

Une fois les hypothèses validées via le prototype, il fallait définir la future architecture. Nous sommes repartis du cahier des charges et des besoins énoncés afin de créer une architecture qui soit tout à la fois simple, ouverte, facilement maintenable et évolutive.

Nous avons ainsi décidé de créer une architecture de type client/serveur avec différents niveaux. Bien entendu, l'objectif premier est de satisfaire aux attentes du projet Parknav, mais nous avons intégré dès le début le fait que d'autres équipes ou projets pouvaient être utilisateurs de la plate-forme dans sa globalité.

La création du serveur de cartes a eu lieu en parallèle des changements d'ordre matériel. Dans la suite de ce rapport, l'expression « serveur de cartes » représente l'ensemble des composants, à savoir : trackerConnector, mapServer, client graphique. La figure 6.2 schématise l'architecture que nous allons détailler. A noter que ce schéma représente quatre capteurs en entrée, composés d'un *tracker* ou d'un Cycab accompagné d'un *trackerConnector*.

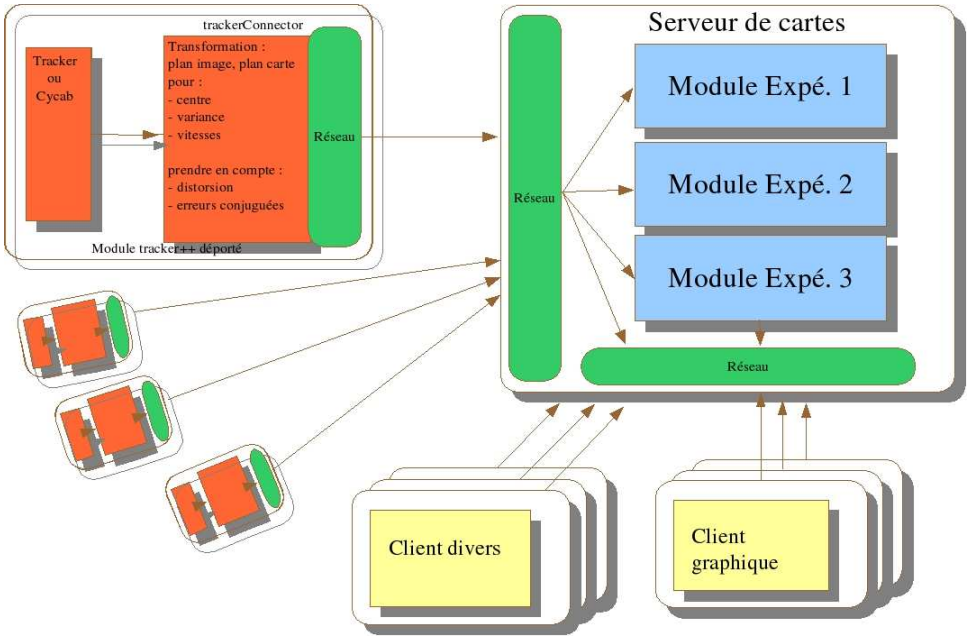

FIG. 6.2 – Nouvelle architecture

6.3.3 Modules communs

Les différents modules du serveur de cartes utilisent des fonctionnalités très proches voire redondantes. C'est pour cela que nous avons créés des fonctions communes, sortes de librairies de code, pour la gestion des communications, des traces d'information, ... Ces modules n'apparaissent pas dans la figure 6.2, car cette dernière est une vision macroscopique de l'architecture. Ils sont intégrés dans chaque composant principal (*trackerConnector*, *mapServer*, client graphique).

6.3.4 Le trackerConnector

Comment rendre interopérable n'importe quel tracker ou bien le Cycab avec notre serveur de cartes ? C'est le but du *trackerConnector* qui fournit un moyen d'interfacer un logiciel quelconque avec le reste de l'architecture. Suivant les cas, ce module effectuera des calculs sur les données qu'il recevra ; par exemple dans le cas de Primablue, c'est lui qui procède à la correction de distorsions, ainsi qu'à la projection homographique.

6.3.5 Le mapServer

Une fois les données collectées par notre trackerConnector, il faut qu'un composant central traite toutes ces informations pour créer la modélisation de l'environnement. Il s'agit bien entendu du serveur de cartes ou mapServer. Ce module central de l'architecture va pouvoir collecter les données en provenance des trackerConnector pour éventuellement appliquer des traitements (fusion, association, ...). En sortie, il fournira la modélisation du parking avec les différents éléments vus dans la section 4.3.

6.3.6 Le client graphique

Comment valider que le modèle créé soit valide et qu'ainsi la chaîne tracker-trackerConnector-mapServer fonctionne correctement ? Dans un premier temps, nous avons simplement récupéré la carte dans un fichier texte avec l'inconvénient de la lourdeur pour valider les données. D'où le développement d'un client graphique permettant d'avoir une visualisation des différents éléments de la carte en temps réel.

6.3.7 Mise au point

Les développements mis en jeu ci-avant impliquent une certaine complexité dans le codage et la mise au point des programmes. Afin de pouvoir optimiser notre code ou tout simplement pour corriger des erreurs de codage, nous avons utilisé plusieurs outils dont un permettant de détecter toute manipulation interdite de la mémoire. Il est clair que ce produit nous a été très utile dans la mise au point de toute l'infrastructure logicielle.

6.4 Le site Web

Faire connaitre notre travail et la plate-forme est un point important dans la vie d'un projet. Le site Web de Parkview a été créé dans cette optique, aussi pour pouvoir mettre à disposition des documents/fichiers en rapport avec Parkview. Une refonte du site Web a permis aussi de tester quelques outils permettant de monter un site rapidement, en l'occurrence pmWiki[2].

[2] http ://www.pmwiki.org. Dernière consultation : 15-oct-05.

Chapitre 7

Travaux sur la plate-forme

Ce chapitre va se consacrer aux travaux réalisés en dehors du serveur de cartes, cette partie sera abordée dans un chapitre dédié (cf. chapitre 8).

7.1 Evolution des PC

7.1.1 GNU/Linux

Depuis fin 2004, le système de référence aux M.I.[1] est Fedora Core 2 (FC2), une distribution GNU/Linux gratuite sponsorisée par la société RedHat®. Suite à l'état des lieux, nous avons dû migrer les deux serveurs sous cette évolution totalement libre de la distribution RedHat©.

Une première machine a été migrée afin de dérouler un ensemble de tests pour valider le bon fonctionnement des logiciels (tracker, prototype, etc...). Cette migration nous a permis aussi de résoudre quelques conflits entre les logiciels que nous utilisons et la version précédente du système d'exploitation (RedHat v9©), comme par exemple des problèmes d'accès à des segments de mémoire partagée dans PrimaBlue.

FC2 intégre plusieurs évolutions intéressantes dont un nouvel outil de gestion de packages : yum[2]. Ce dernier est très proche du célèbre outil de la distribution Debian, à savoir APT[3] et permet ainsi de maintenir le système à jour de manière souple.

7.1.2 Nouvelle machine

Les machines existantes sont destinées à traiter le flux vidéo et à faire tourner les trackers. Il nous fallait une machine dédiée au serveur de cartes, afin d'avoir une répartition de charge en ressources systèmes optimale. La machine Bistre intègre un Pentium®4 à 1.5Ghz, avec 512Mo de RAM tournant sous FC2.

7.1.3 Mémoire

Les machines Parkview et Bistre ne comportaient que 512Mo de mémoire, ce qui est peu au vue de la consommation d'un tracker ou des besoins des modules du serveur de cartes. Aussi, nous avons procédé à l'augmentation de la mémoire pour arriver à 1Go par machine.

La figure 7.1 représente l'infrastructure matérielle après ces différents changements.

[1] Moyens Informatiques : l'équipe qui centralise la maintenance et les installations des PC
[2] http ://www.fedorafaq.org/#installsoftware
[3] http ://www.debian.org/doc/manuals/apt-howto/index.en.html

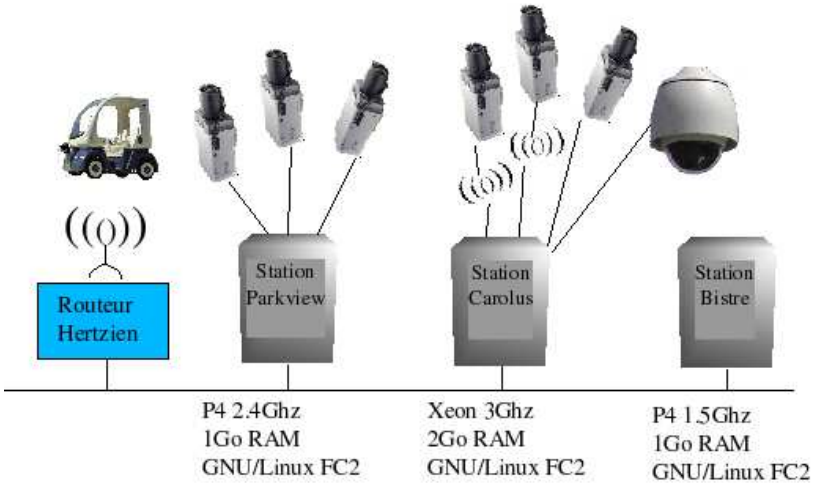

FIG. 7.1 – Infrastructure finale

7.2 Les caméras sans fil

7.2.1 Problème de transmission

Nous avons rencontré de gros problèmes de transmission du flux vidéo avec les caméras sans fil, l'image comportait de nombreux parasites, ce malgré l'utilisation d'équipements d'amplification (pré-amplificateur et antennes). L'impact de ces parasites est la détection de fausses cibles dans les *trackers*, perturbant ainsi toute la chaîne de traitement et se rajoutant aux fausses cibles potentielles retournées par le tracker.

7.2.2 Inventaire du matériel

Pour revenir sur le matériel en place, nous avons dans la chaîne plusieurs éléments listés dans le tableau 7.1 qui apportent, au niveau puissance, soit des gains, soit des pertes.

Eléments	Gain/Perte
Caméra	Pas d'impact direct
Liaison Carte–Antenne (< 10 cm)	Insignifiant
22 m en espace ouvert	-66,89 dB
Antennes omnidirectionnelles sur la halle	15 dBi (Il s'agit de décibels isotropiques, gain de puissance par rapport à une antenne isotropique)
Cable Antenne–Pré-ampli	-4,4 dB
Pré-ampli	+26 dB (NF : 0,68 dB)

TAB. 7.1 – Chaîne de liaison des caméras sans fil

Le premier point à noter est l'utilisation d'antennes omnidirectionnelles (voir figure 7.2) avec un gain d'amplification important pour une transmission sur une distance de 22 m. Le problème avec ce genre de matériel est que leur zone de couverture optimale n'est pas la région proche, en d'autres termes, les environs de l'INRIA seraient mieux couverts que les 22 mètres de parking que nous utilisons.

FIG. 7.2 – Antenne omnidirectionnelle

7.2.3 Calcul puissance

En partant de [ADFF04], nous pouvons calculer les différents éléments de la transmission afin d'effectuer le bilan de la liaison.

La carte émettrice ou plus exactement l'amplificateur de la caméra a une puissance de 50 mW, ce qui correspond d'après la formule suivante à 17 dBm (décibels « milliwatts »).

$$dBm = 10\log(\ \frac{P}{0.001}\ ) \tag{7.1}$$

Le câble entre la caméra et l'amplificateur de la caméra mesure environ 10 cm. Sachant qu'il s'agit d'un câble ordinaire, nous prendrons les caractéristiques d'un câble coaxial type Ethernet (RG58) :

$$Atténuation = 1dB\ /\ mètre \tag{7.2}$$

$$dBm = \frac{(\ -0.010 \times Atténuation \times 1000)}{1000} \tag{7.3}$$

La puissance rayonnée à la sortie de l'antenne (encore appelée **PIRE** ou puissance isotrope rayonnée équivalente) peut maintenant être calculée, sachant que nous rajoutons environ 1 dB de perte due aux connecteurs.

$$Puissance_{rayonnée} = puissance\ émetteur\ -\ perte\ cble\ +\ gain\ antenne \tag{7.4}$$

$$P_{rayonnée} = 15.99\ dBm \tag{7.5}$$

$$= 40\ mW \tag{7.6}$$

Le calcul des pertes en espace ouvert est réalisé d'après la formule de Friis, il s'agit de calculer la perte engendrée par la propagation en espace libre [WJL03].

La formule donnant le ratio entre la puissance à la source et la puissance à la réception est :

$$\frac{P_r}{P_t} = G_t\ G_r\ (\ \frac{\lambda}{4\pi d}\ )^2\ ,\ avec\ \lambda = \frac{c}{f} \tag{7.7}$$

De là, nous pouvons calculer la perte due à la propagation :

$$Perte = 10\log\ \frac{P_r}{P_t} = 10\log\ G_t\ +\ 10\log G_r\ -\ 20\log f\ -\ 20\log d\ +\ K \tag{7.8}$$

$$K = 20\log\frac{3\times 10^8}{4\pi} = 147,56 \tag{7.9}$$

Dans le cas où les antennes sont à gain unitaire (isotropique), nous pouvons définir une formule par défaut de perte due à la propagation.

$$Perte_{défaut}\ =\ -32,44\ -20\log\ f_{MHz} - 20\log\ d_{Km} \tag{7.10}$$

avec,
- $Perte$ est l'atténuation en dB,
- $Perte_{défaut}$ est l'atténuation en dB lorsque les antennes sont à gain unitaire,
- P_t est la puissance en transmission,
- P_r est la puissance sur l'antenne de réception,
- G_t concerne le gain de l'antenne de transmission,
- G_r concerne le gain de l'antenne de réception,
- λ est la longueur d'onde,
- c : la célérité de la lumière,
- d : distance en Km,
- f : fréquence en MHz (2400 dans notre cas).

Dans notre cas, en appliquant la formule de Friis :

$$Perte = -32,44 - 20\log 2400 - 20\log 0,022 = -66,89\ dB \tag{7.11}$$

Nous voyons ici la relation entre la perte et la distance, sachant que de nombreux paramètres peuvent altérer les valeurs théoriques obtenues (obstacle, réflexion des ondes, ...), en pratique, les obstacles et réflexions diverses augmentent cette atténuation...

Du côté réception, autrement dit sur la halle robotique, nous avons pris comme hypothèse de travail un câble de très bonne qualité avec une atténuation de 0,22 dB/m (pour une fréquence de 2,45 Ghz), sur une longueur de 20 m environ.

Ce qui nous donne :

$$Perte_{cble} = \frac{(\ -20\times 0,22\times 1000)}{1000} = -4,4\ dBm \tag{7.12}$$

7.2.4 Bilan de la liaison

Une fois ces différens calculs effectués, nous pouvons faire le bilan de la liaison en l'état, partant de la caméra jusqu'à l'antenne extérieure.

En l'occurrence, il faut pour que le système fonctionne correctement que :

$$Total_{Emission} + Total_{Propagation} + Total_{Réception} > 0 \tag{7.13}$$

or :

$$Total_{Emission} + Total_{Propagation} + Total_{Réception} \tag{7.14}$$
$$= (\ 30 - 1,01 + 0\ ) - 66,89 + (\ 15 - 4,4\ ) \tag{7.15}$$
$$= -27,3\ dB \tag{7.16}$$

Nous voyons que théoriquement en utilisant uniquement le matériel « extérieur » le signal ne devrait même pas arriver jusqu'à l'antenne. Nous utilisons aussi un pré-amplificateur relié à l'antenne ; ce dernier fournit un gain de 26 dB avec un rapport signal/bruit très faible. En reprenant le bilan 7.16, nous arrivons à :

$$Total_{Emission} + Total_{Propagation} + Total_{Réception} \tag{7.17}$$
$$= -27,3 + 26 \tag{7.18}$$
$$= -1,3dB \tag{7.19}$$

Ce résultat explique que nous obtenons bien un signal vidéo, mais très perturbé et fluctuant en fonction des conditions climatiques.

7.2.5 Résolution du problème

Afin de résoudre nos soucis de performances (et pouvoir utiliser les caméras !), 2 points principaux sont à revoir dans notre infrastructure (d'après le tableau 7.1) :

- utiliser uniquement des antennes directionnelles (sur la halle robotique et sur les caméras),
- remplacer le module émetteur des caméras, par un modèle plus puissant.

Après discussion avec les SED[4], il ressort que les antennes fixées sur les caméras sont normalement préconisées pour un usage intérieur, et par conséquent ne conviennent pas à notre besoin, d'où la préconisation de changements d'antennes.

Cependant, il est bon de noter que la règlementation française impose des limites d'émission, comme indiqué sur le site de l'ART[5]. En l'occurrence, la puissance autorisée en extérieur pour le réseau 2,4 Ghz est de 100 mW. Dans notre cas, le PIRE[6] actuel est de 40 mW (cf. équation 7.6), ce qui nous laisse de la marge.

Afin d'essayer de résoudre ce problème, nous avons dans un premier temps testé une liaison filaire directe. Le résultat fut bien entendu que les caméras fonctionnent correctement en filaire (il aurait été étonnant que deux caméras soient tombées en panne).

[4]Service Support Expérimentation et Développement
[5]http ://www.art-telecom.fr/dossiers/rlan/index-d-rlan.htm. Dernière consultation : 6-juil-05.
[6]Rappel : puissance isotrope rayonnée équivalente

Nous avons pris contact avec la société Infracom®[7] afin d'étudier ce qui est faisable. Ils ont confirmé le diagnostic réalisé et nous leur avons commandé le matériel suivant (cf. figure 7.3) :
- émetteur vidéo 100 mw (référence minitx24-100),
- antenne directive type *patch* (référence 248080).

FIG. 7.3 – Emetteur 100mW et nouvelle antenne

A la fin du stage, le matériel n'était pas encore arrivé.

7.3 Calibrage des caméras

Comme nous l'avons vu dans le chapitre précédent, il est nécessaire de procéder au calibrage des caméras (voir aussi [Hel03]). Cette opération permet de récupérer les paramètres intrinsèques (focale, centre optique, ...) mais aussi extrinsèques (matrice d'homographie), qui nous permettront de faire les projections sur le plan du parking des observations faites dans le plan de la caméra.

7.3.1 Logiciel

Pour obtenir ces paramètres, nous avons utilisé un logiciel développé en interne : *CameraCalibration*. Il utilise la librairie OpenCV[8] dédiée au traitement des images[9] pour pouvoir effectuer un ensemble d'opérations sur une image issue de la caméra que l'on traite. Le calibrage est de type multi-plan, autrement dit, il y a utilisation d'une mire (un damier noir et blanc) présentée à la caméra suivant une inclinaison arbitraire.

L'application *CameraCalibration* comporte deux onglets distincts : l'un pour les paramètres intrinsèques (focal, centre optique, ...), l'autre pour les paramètres extrinsèques, autrement dit la matrice d'homographie.

Les figures 7.4 et 7.6 (page 46) montrent l'interface utilisateur de cette application, développée en C++ avec la librairie graphique Qt[10].

[7] http ://online.infracom-france.com/. Dernière consultation : 27-nov-05.
[8] http ://www.intel.com/technology/computing/opencv/. Dernière consultation : 15-nov-05.
[9] Pour information, OpenCV intègre de quoi développer un tracker, voir le manuel de référence inclut dans [Int03]
[10] http ://www.trolltech.com/. Dernière consultation : 6-juil-05.

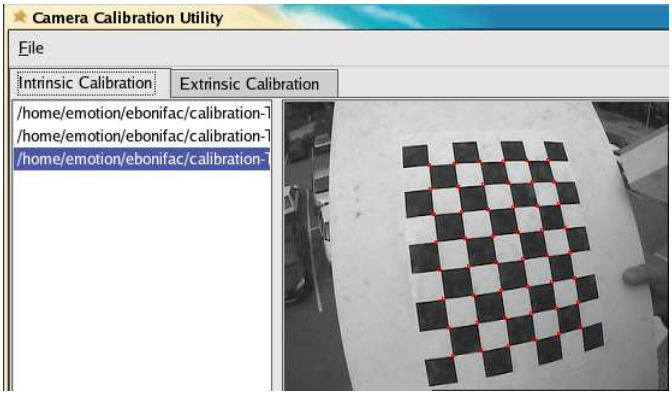

FIG. 7.4 – Calibrage : paramètres intrinsèques

L'application va sauvegarder les paramètres dans un fichier au format XML, comme celui de la figure 7.5.

```
<?xml version = '1.0' encoding = 'UTF−8'?>
<cameraparameters label="Right1" >
        <intrinsic width="384" height="288" fx="240.29" fy="242.17" cx="183.98"
        cy="139.648" k1="−0.421635" k2="0.254622" p1="−0.00372892" p2="0.002574" />
        <homography a11="−9.05409" a12="33.3147" a13="277.117" a21="11.7073"
        a22="56.7613" a23="−1754.04" a31="0.000248907" a32="0.0191073" a33="1"/>
</cameraparameters>
```

FIG. 7.5 – Stockage des paramètres de la caméra

Nous retrouvons la taille de l'image utilisée (qui est aussi celle de nos vidéos), une partie avec les paramètres intrinsèques et enfin la matrice d'homographie.

7.3.2 Récupération des paramètres intrinsèques

Comme nous pouvons le voir sur la figure 7.4 (page 45), le programme utilise l'image d'un damier de dimensions connues ; ce principe est lié à OpenCV qui inclut les primitives nécessaires pour la détermination des coins du damier.

Les paramètres intrinsèques représentent les caractéristiques « internes » de la caméra - comme le nom le laisse entendre - , voir le tableau 7.2.

Distance de focale ou focale (paramètres fx, fy)
Coordonnées du centre optique (cx, cy)
Taille d'un pixel
Vecteur de coefficients de distorsion (k1, k2, p1, p2)

TAB. 7.2 – Paramètres intrinsèques

Ces paramètres vont nous permettre de calculer la correction de distorsion, due aux objectifs grand angle.

7.3.3 Récupération des paramètres extrinsèques

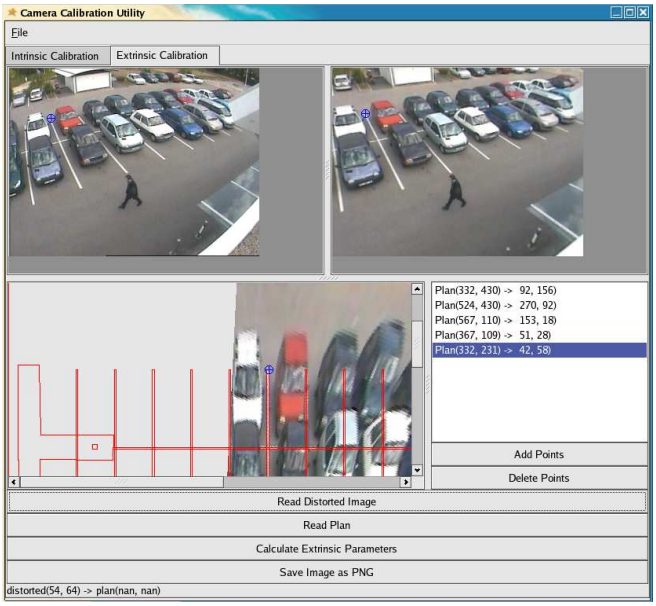

Fig. 7.6 – Calibrage : paramètres extrinsèques

Il s'agit des paramètres liés au positionnement de la caméra dans le monde, en l'occurrence, la matrice d'homographie, que l'on peut détailler comme étant la combinaison d'une matrice de rotation et un vecteur de translation.

Cette dernière va nous permettre d'effectuer la projection des coordonnées du plan image, vers le plan du parking, ce dont nous avons besoin bien entendu.

Dans [Hel03], le chapitre 3 aborde la transformation homographique et explique d'une part les principes et les formules mathématiques. La figure 6.1 (chapitre 6) - tirée de [Hel03] - reprend le schéma de la projection centrale ou projection via le centre optique.

En résumé, nous pouvons dire que pour pouvoir effectuer la transformation du plan de la caméra vers le plan du parking, il nous faut trouver les coefficients de l'homographie, qui dépendent de la position de la caméra.

Le logiciel que nous utilisons permet de calculer ces coefficients, via l'onglet 'Extrinsic Calibration' (cf. la figure 7.6).

Le principe est le suivant :
- Il faut d'abord sélectionner une image prise avec la caméra à calibrer.
- Le logiciel affiche automatiquement la même image sans la distorsion.
- Il faut charger le plan du parking (fichier XML).
- Puis, il faut sélectionner au moins 4 points de l'image du parking et trouver la correspondance sur le plan schématisé (les points ne doivent pas être tous alignés).
- Une fois fait, le calcul peut avoir lieu (bouton 'Calculate Extrinsic Parameters').
- Si tout se passe bien, une image transformée est affichée sur le plan schématique.
- La dernière étape consiste à sauvegarder les paramètres dans un fichier.

Le calcul des coefficients est réalisé grâce à la librairie OpenCV dont nous avons parlé plus haut.

7.4 Les trackers

7.4.1 Comparatif

Nous avons procédé à une campagne de tests afin d'éprouver les deux trackers, sur la machine Carolus, la plus puissante des machines en place (présentée dans la figure 4.3, chapitre 4).

Lors des expérimentations qui ont eu lieu en 2003 et 2004, les ingénieurs en place ont clairement mis en évidence qu'il y avait des problèmes dûs au fonctionnement en extérieur.

Il faut bien insister sur le fait que la fiabilité de ces logiciels de détection conditionne toute la chaîne de traitement de Parkview.

Dans un premier temps, nous avons mesuré la facilité d'utilisation des deux logiciels (tableau 7.3). L'un des points à noter et qu'il n'y a pas de documentation de ces produits, ce qui implique que nous ne sommes pas capables de positionner les paramètres de chaque logiciel de manière optimale, sans l'aide de l'équipe Prima.

	PrimaLab	PrimaBlue
O.S.	Linux	Linux
Serveur Vidéo	Non	Oui
Paramètres	23	35
Interface	N/A	Compréhensible
Communication	N/A	Socket TCP
Langage devpt	Scheme	C++
Caméras/session	1	3 à 4
Sessions simul.	3	1
Documentation	Générale. Pas sur les paramètres	Non
Playback vidéo	Oui	Oui

Tab. 7.3 – Facilité d'utilisation des trackers

Comme vu précédemment, nous pouvons utiliser un maximum de 3 à 4 caméras par machine et seul PrimaBlue est communiquant (le comparatif a été fait avec la version 2 de PrimaLab).

Concernant le comparatif de tracking, nous avons établi un ensemble de vidéos de tests représentant différents cas de figures (tableau 7.4).

Scénarios
Un objet mobile du début jusqu'à la fin
Un objet mobile qui s'arrête
Un objet immobile qui démarre
Un objet mobile qui s'arrête puis redémarre
Un piéton se cache derrière une voiture
Un piéton passe derrière une voiture, il est partiellement caché
Véhicule qui s'arrête, un piéton sort du véhicule
Véhicule qui s'arrête, un piéton sort du véhicule et le véhicule redémarre
2 piétons qui marchent ensemble puis se séparent
1 piéton qui traverse
2 piétons qui se croisent
2 piétons qui se suivent et qui croisent un 3ème piéton
2 piétons qui marchent ensemble et qui croisent un 3ème piéton
3 piétons qui marchent ensemble et qui se séparent

TAB. 7.4 – Scénarios de tests

Le résultat est que dans les deux cas nous sommes fortement tributaires des conditions de l'environnement à un instant t : luminosité (variation brutale, ombre, ...), pluie, ... Nous avons régulièrement des détections de fausses cibles, qui bien sûr sont pénalisantes pour la modélisation de l'environnement. Pour « défendre » les deux logiciels, il faut noter qu'il s'agit d'un problème récurrent en robotique et en vision. Ainsi, le « jouet » Aibo de Sony© n'arrive pas à détecter un ballon qui est à moitié dans l'ombre.

A noter que PrimaBlue peut utiliser un paramètre nommé *normalizeIntensity* qui permet d'être moins sensible aux variations de luminosité. L'utilisation de cette option apporte un gain réel et, par conséquent, le paramètre sera utilisé suite à ces tests. Le seul souci c'est que ce paramètre non seulement n'est pas géré par l'interface graphique mais en plus il n'est pas activé par défaut ! Le simple fait de sauvegarder un fichier de configuration écrase ce paramètre.

Dans tous les cas, nous voyons que la problématique des fausses cibles ou bien des pertes de cible, implique d'avoir un module intégré au serveur de cartes capable de distinguer les « vraies » cibles par rapport aux données reçues. Ce module fera de l'association ou de la fusion. Ceci va induire des choix quand au développement de ce serveur.

En résumé, les deux logiciels sont loin d'être exempts de défauts et bugs, et globalement ont des performances équivalentes. Cependant, l'avantage réel de PrimaBlue est son mode communiquant qui fait défaut à PrimaLab v2.

7.4.2 Evolution de PrimaLab

Comme dit dans le paragraphe 4.4, Primalab, contrairement à PrimaBlue, évolue fréquemment. Ainsi durant le stage, nous sommes passés de la version 2 à la version 3, avec une refonte complète de l'architecture interne du logiciel.

7.4.2.1 Fonctionnalités

La détection de cible a été revue et améliorée, rendant ainsi le logiciel plus performant que PrimaBlue. D'autre part, l'architecture interne a aussi sensiblement évoluée, permettant l'ajout de module divers (type plugin ou greffon en bon français). Un module de communication a été développé par l'équipe afin de palier au manque principal de ce logiciel.

7.4.2.2 Installation

L'installation de PrimaLab est détaillée en annexe, cf. annexe B (section B.3).

7.4.2.3 Utilisation

Malheureusement, nous n'avons pas pu intégrer cette nouvelle version dans notre architecture car nous avons été confronté à de nombreux problèmes techniques lors des tentatives d'intégration. Entre autres, le module de communication n'a pu être activé et testé. Par ailleurs, plusieurs versions successives nous ont été livrées en fin de stage, ne nous laissant pas le temps d'avoir une version stabilisée à intégrer.

Ceci implique que seul le logiciel PrimaBlue a réellement été utilisé durant cette année.

7.5 Localisation absolue

Nous avons présenté dans le chapitre sur le Cycab (cf. chapitre 5) une localisation relative s'appuyant sur le principe du SLAM. Cette localisation permet d'obtenir la position du Cycab par rapport à un emplacement initial, mais dans notre cas, nous avons besoin de connaître la position du robot dans la plan du parking, autrement dit une localisation absolue.

Fernando De-La-Rosa, ingénieur expert sur le projet Puvame, a intégré courant 2004 une évolution à la localisation relative en implémentant une localisation absolue dans le plan du parking. Le fonctionnement est schématisé dans la figure 7.7. Fernando a appelé ce mode « SLAMinMap » ou SLAM dans le plan (du parking).

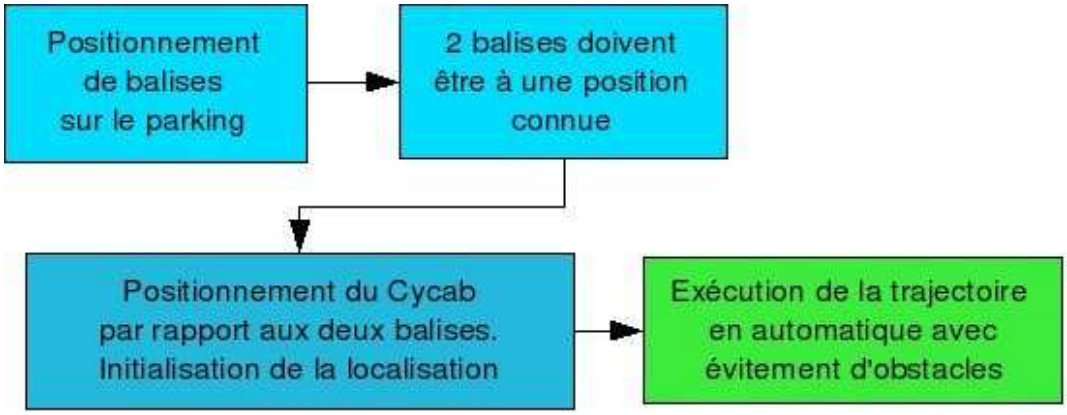

FIG. 7.7 – Principe de localisation absolue

L'étape de localisation par rapport aux deux points de référence est vraiment la clé de la bonne localisation par la suite. L'odométrie joue ici un rôle important car il n'y a pas de construction de la carte de l'environnement au préalable. C'est la limite de cette méthode, car les capteurs proprioceptifs induisent une erreur pouvant s'accumuler au cours de la trajectoire (glissement des roues par exemple).

Ce module de localisation nous permet d'interfacer le Cycab avec Parkview et ainsi devenir fournisseur d'informations pour le serveur de cartes. Durant le stage, c'est cette localisation absolue qui a été utilisée principalement. La figure 7.8 montre le résultat de la localisation du Cycab en utilisant le prototype qui sera décrit dans la section 8.1. Nous voyons l'emplacement des deux balises de référence sur les angles du trottoir et la représentation graphique du Cycab dans le client.

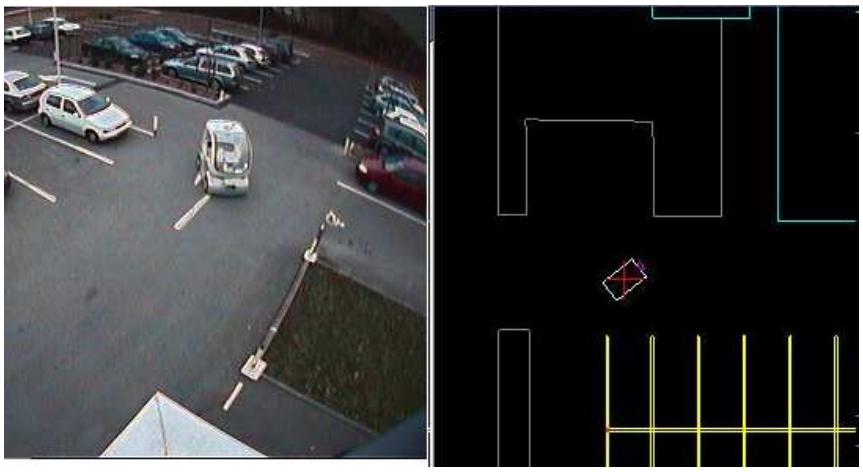

FIG. 7.8 – Prototype : localisation du Cycab

7.5.1 Le problème

Cependant, cette localisation est franchement perfectible : au bout de 10 mètres de déplacement, nous avons une erreur d'environ 1 m, ce qui est suffisant pour des tests uniquement de localisation, mais n'est pas acceptable pour effectuer de l'évitement d'obstacles ou de la conduite automatique.

D'autre part, les balises sont positionnées à chaque manipulation, entre autres, deux de ces plots sont déposés sur des points précisément identifiés au niveau du plan parking (angle de l'un des trottoirs). L'un des inconvénients de cette méthode est la lourdeur du positionnement des balises lors de chaque manipulation : perte de temps, problème avec le vent, imprécision du positionnement, ...

7.5.2 La solution retenue

Nous avons décidé d'étudier un nouvel algorithme de localisation absolue s'appuyant sur une carte de balises fixes, carte qui est connue a priori. Autrement dit, nous avons équipé le parking de balises fixes ce que montre la figure 7.9.

Les croix représentent des poteaux à installer, les carrés sont des bandes réflectrices à poser sur des poteaux existants ou sur des angles de bâtiment.

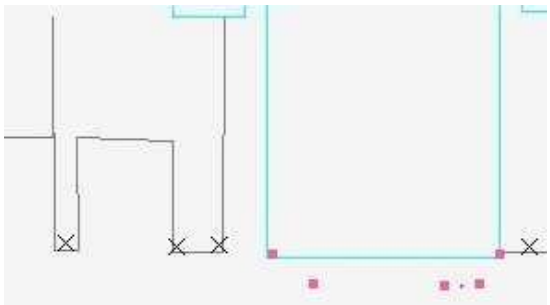

Fig. 7.9 – Equipement (partiel) du parking en balises

Une fois ces balises implantées, il faut faire intervenir un géomètre afin de prendre les coordonnées exactes de chacune d'elle, dans le plan du parking. En ayant ces données précises, le Cycab sera capable de se localiser à tout instant de manière très performante.

7.5.3 Caractéristiques des balises

Par rapport aux spécifications du laser Sick (0,5 degrés entre deux rayons), le choix pour les poteaux s'est porté sur des piquets en bois d'un diamètre de 10 cm, nous permettant de "toucher" la balise dans 100% des cas sur une distance comprise entre 10 m et 13 m.

D'autre part, le positionnement a été choisi pour qu'il y ait toujours au moins 3 balises visibles par le Sick, condition indispensable pour une bonne localisation de la voiture que ce soit en localisation dans le referentiel parking ou bien pour la méthode de Cédric Pradalier [Pra01]. Il est important aussi d'éviter les symétries au niveau de la disposition des balises.

7.5.4 L'existant et notre proposition

Après une petite investigation sur l'existant, nous avons étudié le rapport d'un stagiaire qui est intervenu dans l'équipe en 2005 - Chong Chin Hui. Il a travaillé sur l'étude d'une nouvelle localisation pour le Cycab [CH05], s'appuyant sur le principe du SLAM (voir la section 5.3). Il intègre dans son étude la connaissance de 3 balises fixes à minima lui permettant de ce repérer dans l'ensemble des balises connues. Ensuite, ce repérage est couplé avec l'odométrie pour pouvoir calculer les coordonnées absolues du Cycab.

L'approche que nous proposons est différente, dans la mesure où nous allons fournir au Cycab la liste des balises connues, qui ont été installées sur le parking. Nous pourrons ainsi nous passer de l'odométrie, peu fiable. La principale action de la localisation est de détecter et de reconnaître les balises connues. Ensuite, il est facile d'obtenir une matrice de transformations permettant de calculer les coordonnées dans le plan du parking.

La figure 7.10 montre le principe de la concordance des balises. Il est possible de reprendre une grande partie du programme développé par Chong Chin Hui, même si en l'état il est inutilisable principalement à cause d'une partie graphique très lourde (avec blocage du PC).

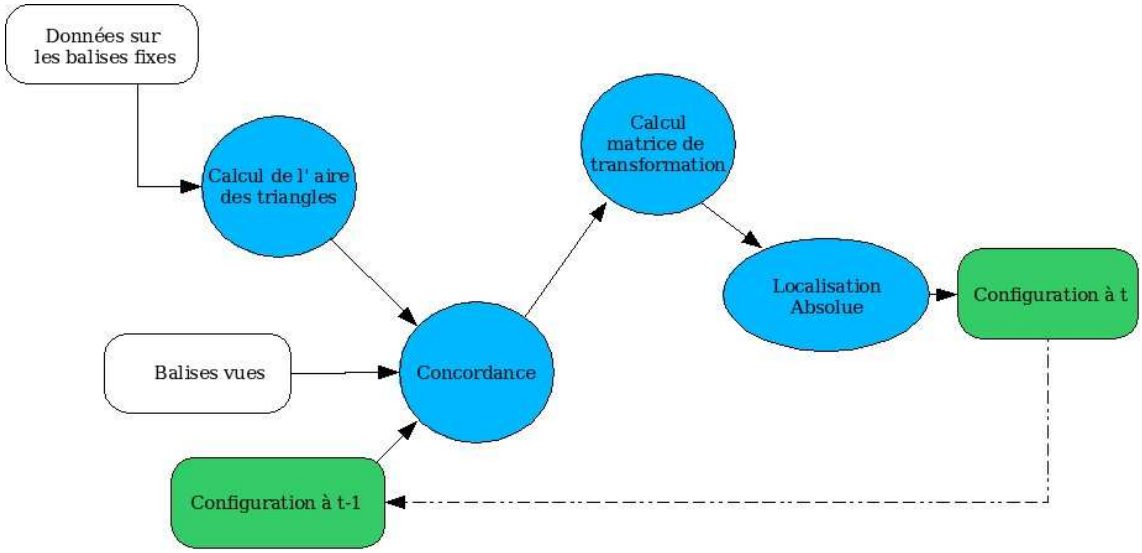

FIG. 7.10 – Nouvelle localisation : concordance des balises

Dans notre cas, nous n'avons pas besoin de l'interface graphique proposée, car nous souhaitons avoir un module s'intégrant avec notre architecture, qui inclut déjà un client graphique. Notre proposition par rapport à ce travail est de supprimer la gestion graphique et de faire évoluer les traitements afin d'intégrer la liste des balises dès le démarrage du programme.

Avant de passer au Cycab directement, nous devons développer ce nouveau module avec le simulateur. Pour ce faire, il va falloir le faire évoluer pour pouvoir intégrer ce module (et d'autres à venir) de manière plus facile qu'aujourd'hui. Ce que nous souhaitons réaliser est présenté dans la figure 7.11.

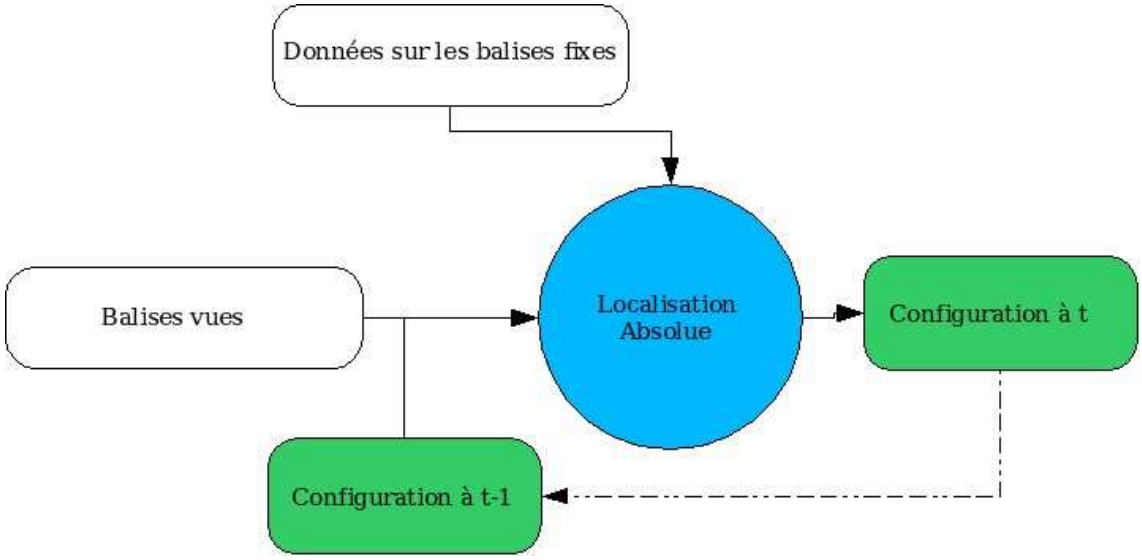

FIG. 7.11 – Principe de la nouvelle localisation

7.5.5 Intégration avec le simulateur

Le simulateur, comme nous l'avons vu dans la section 5.4, est un outil performant reproduisant les comportements du Cycab et de ses capteurs. Il est possible de développer des modules se « branchant » sur le simulateur afin de tester de nouveaux algorithmes, comme par exemple l'algorithme de localisation.

Le simulateur a subi une profonde refonte en 2005, principalement avec l'intégration d'un moteur graphique en 3 dimensions permettant d'avoir une représentation de l'environnement plus « avenante ». Cependant, le cœur de l'architecture est restée très proche de l'existant qui malheureusement, est complexe à appréhender. Le développement d'un module externe est lourd et nécessite un investissement important dans la lecture du code du simulateur.

L'idée serait de modifier le simulateur afin de pouvoir intégrer des modules extérieures plus facilement, comme sur le principe de *plugin* ou greffon en français. A l'heure actuelle, le simulateur communique ses données via des segments de mémoire partagée, il faudrait utiliser uniquement des sockets pour l'envoi vers l'extérieur. Ceci nous permettrai d'ajouter tout programme très facilement, même de manière déportée. Bien entendu, ce mode de fonctionnement devra être aussi porté sur le Cycab réel, afin d'avoir un comportement identique et rendant les modules portables.

7.5.6 Poursuite de l'étude

Faute de temps et devant la complexité du simulateur, nous n'avons pas pu achever intégralement cette partie.

Les différents points restants à développer sont :
- évolution du simulateur pour communication par socket,
- création d'une API de connexion au simulateur,
- reprise du code de Chon pour intégration des balises fixes et suppression de la partie graphique,
- intégration dans le nouveau code de l'API de communication,
- interfaçage du module de localisation avec le serveur de cartes,
- évolution du serveur de cartes si nécessaire.

7.6 Résumé

La plate-forme Parkview a évolué pendant le stage afin de rentrer en conformité avec les consignes de l'INRIA, ou pour correspondre aux besoins des développements comme le serveur de cartes, ou bien encore pour résoudre des problèmes techniques. Tout n'est pas terminé, ainsi l'installation du nouveau matériel pour les caméras sans fil reste à faire. De même, les développements pour la localisation absolue sont à démarrer, afin d'utiliser la carte des balises que nous avons implémentées sur le parking.

Chapitre 8

Le serveur de cartes

Parkview pour être pleinement opérationnel doit être doté d'un logiciel ou d'un ensemble de logiciels permettant de traiter le flux vidéo : c'est le but du serveur de cartes dynamiques et de ses composants annexes.

Ce chapitre va lister les différents modules, leurs spécificités, les choix de conception, les difficultés et autres aspects techniques pertinents.

8.1 Le premier prototype

Comme indiqué en section 4.5, les premiers travaux de développement avaient pour objectif la réalisation d'un prototype.

Il s'agit d'un programme monolythique, "tout-en-un", comprenant les parties listées dans le tableau 8.1 et représentées sur la figure 4.11.

Une partie cliente s'interfaçant avec un tracker PrimaBlue pour un seul flux vidéo
Une partie cliente pour le Cycab
Un module de transformation de données
Un module d'affichage en deux dimensions

Tab. 8.1 – Fonctionnalités du prototype

Développer un tel prototype représente plusieurs intérêts :
- apprentissage des techniques nécessaires (C++, graphisme, vision, caméra, ...),
- valider certaines hypothèses de travail (connexion au tracker, performance, ...),
- pouvoir tester rapidement nos solutions,
- avoir dès que possible un outil de démonstrations.

Ce programme nous a donné l'occasion de tester la mise en place de fichiers de configurations dans un format XML. Pourquoi le choix de l'XML ? Parce que la syntaxe permet d'avoir une codification claire des différents champs du fichier, pour des fichiers de configuration par exemple. D'autre part, le traitement de ce type de fichiers est quelque chose d'aisé.

Dans un deuxième temps, nous avons rajouté une gestion sommaire de deux flux vidéos en parallèle, ce afin de voir comment se comporte notre programme en terme de performance. Les résultats furent très bons, nous permettant ainsi de valider le principe d'une utilisation multi caméras sans souci.

Nous avons pu aussi tester rapidement un ensemble d'algorithmes que ce soit dans l'intégration du Cycab, sur le traitement de flux vidéos, ou tout simplement sur le rendu graphique à prévoir à l'avenir. Très vite nous avons atteint les limites de cette architecture mono-bloc : codage en dur d'un ensemble d'informations, la moindre modification impliquait de recompiler tout le programme, l'ajout d'un autre flux/capteur était très lourd, ...

Ce qui nous a conduit à la nouvelle architecture.

8.2 La nouvelle architecture

8.2.1 Revue du cahier des charges

En repartant du cahier des charges vu au chapitre 3, nous pouvons extraire les caractéristiques et fonctionnalités essentielles de notre serveur de cartes (et de ses modules annexes), tableau 8.2.

Construction de la carte dynamique
Architecture de type client/serveur
Données en temps-réel
Evolutivité et ouverture
Interfaçage avec les trackers actuels
Réception de données du Cycab
Au minimum, un client graphique 2D
Structure de données utilisée par tous les programmes
Rejouer des scénarios type (simulateur, ...)

TAB. 8.2 – Fonctionnalités nécessaires du mapServer

Développement en C++, sous GNU/Linux
Utilisation d'un outil de versionning[a]

[a]Gestion de versions

TAB. 8.3 – Contraintes d'implantation

En plus, nous proposons d'implémenter des fichiers de configuration, permettant de sélectionner au lancement de chaque module les options de base (comme par exemple le niveau de trace souhaité).

8.2.2 La carte dynamique

Structure de la carte dynamique

Nous avons abordé dans le cahier des charges les différents éléments nécessaires à la carte modélisant le parking. Elle doit contenir les éléments suivants, tels qu'ils ont été définis dans [Fra03] et que l'on retrouve aussi dans la figure 4.7 :
- les données statiques du parking (le plan),
- les cibles en mouvement,
- les informations sur les objets dits semi-statiques.

Données statiques

Le modèle doit inclure le plan du parking afin de pouvoir représenter graphiquement ce dernier, mais aussi pour tout calcul de coordonnées.

Comme indiqué dans la section 4.1, le plan a été construit courant 2004 par l'ingénieur expert en place. Cependant, le parking a été agrandi depuis et le plan ne tenait pas compte de cette évolution. Les SED nous ont fourni un plan complet réalisé sous Autocad© ce qui nous a permis d'étendre notre représentation XML existante.

L'extension n'est pas couverte par les caméras de Parkview. Cette région étant peu utilisée par les automobilistes, elle est idéale pour pouvoir procéder à des expérimentations avec le Cycab par exemple. La dernière version en date du plan XML est disponible dans le dépôt de l'outil de gestion de sources, ainsi que sur le site Web de Parkview[1].

Les cibles mobiles

Une cible mobile est représentée par un ensemble de données nécessaires aux traitements, évoluant dans le temps. Nous pouvons citer au minimum : les coordonnées dans le plan image, celles dans le plan absolu du parking, le type de tracker/détecteur ayant fait l'acquisition, l'heure de détection (précision à la milliseconde), une vitesse de déplacement. Un identifiant de cibles peut compléter ces données et permettre avec l'information sur le type de tracker/détecteur de connaître l'origine de la cible : caméra, Cycab, autre, ...

Les semi-statiques

Un objet semi-statique est un corps immobile à un instant t qui peut à t+1 devenir mobile. Une voiture garée est un très bon exemple. Dans un premier temps, ces futures cibles ne seront pas gérées et pourront faire l'objet d'un autre sujet de stage.

Documentation, fiabilité

La structure de données doit être documentée et validée avec les différents intervenants sur les projets liés à Parkview, ceci afin de faciliter la réutilisabilité des données et l'interfaçage avec le serveur de cartes.

[1] http ://emotion.inrialpes.fr/parkview. Dernière consultation : 2-dec-05

Il est important d'insister sur la fiabilité et la précision de ces informations, car elles serviront à terme à faire du calcul de trajectoires sur des robots mobiles, tel que le CyCab.

Schémas de l'architecture

L'architecture choisie est de type client/serveur, plus exactement 3 tiers ou 3 couches (la figure 8.1 montre une architecture 3 tiers type avec une base de données). La différence avec la figure citée est que nous n'aurons pas de base, mais des informations fournies en temps réel par les capteurs.

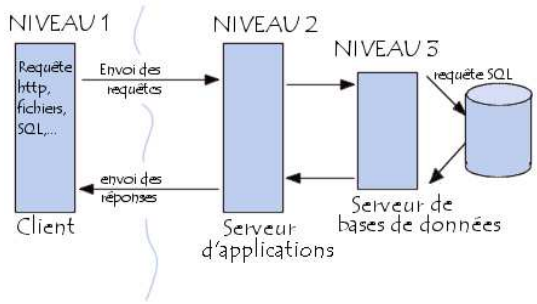

FIG. 8.1 – Architecture 3 tiers type (http ://www.commentcamarche.net

Dans notre cas, le tiers de niveau 3, souvent appelé « backend » dans la littérature, pousse les données, contrairement à une architecture 3 tiers classique où c'est le niveau contenant la logique applicative qui demande les informations.

La figure 8.2 schématise notre architecture cible de manière macroscopique, tandis que la figure 8.3 représente le diagramme de composants macroscopiques de cette nouvelle architecture.

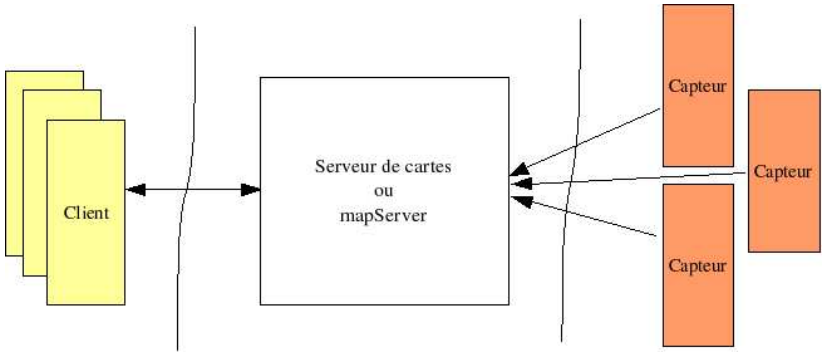

FIG. 8.2 – Schéma macroscopique de la nouvelle architecture

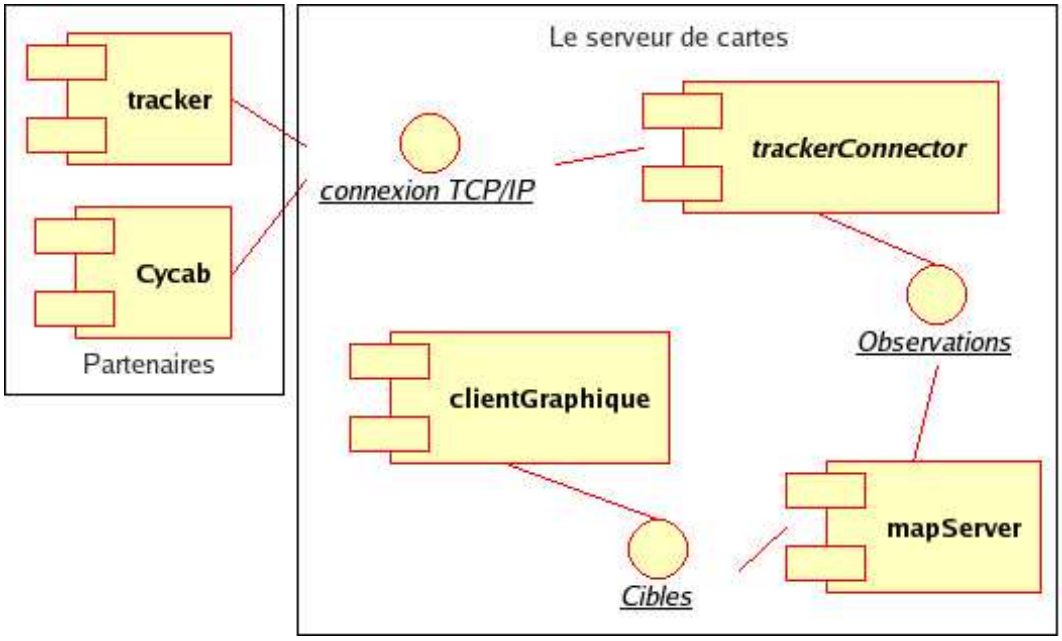

FIG. 8.3 – Diagramme des composants macroscopiques

En « zoomant » sur ce schéma, nous obtenons la représentation de la figure 8.4 (reprise de la 6.2).

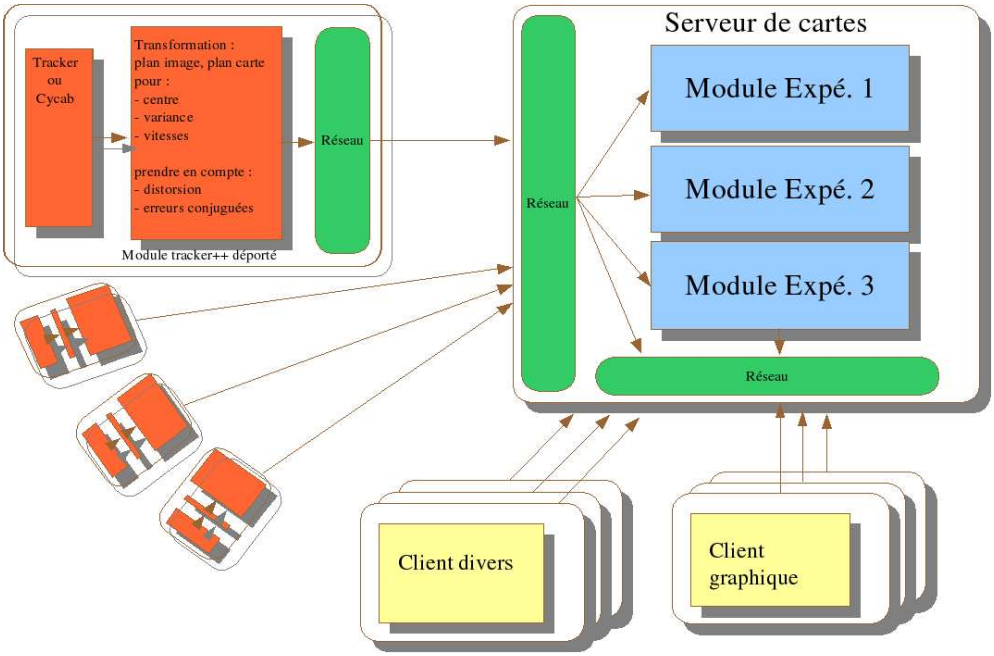

FIG. 8.4 – Nouvelle architecture détaillée

La suite de ce chapitre porte sur les choix technologiques et la description détaillée de chaque module. En préambule nous pouvons dire que :

- Le *trackerConnector* est un module permettant d'interfacer un tracker ou un Cycab avec le *mapServer*. Ce composant effectue les transformations en amont (formatage de données, distorsion/homographie le cas échéant, ...).
- Le *mapServer* inclut différents modules permettant de faire de l'association, de la fusion, tout module d'expérimentation ou bien un envoi de données brutes non transformées.
- Les clients - entre autres graphiques - récupèrent leurs informations du serveur de cartes.

8.2.2.1 Pourquoi cette architecture

Si l'on regarde notre besoin, nous voyons que nous devons recevoir des informations de différents capteurs, les envoyer à un serveur de cartes, pour pouvoir les transférer à des clients divers.

D'autre part, chaque étage ou niveau doit être évolutif, nous devons par exemple être capable de traiter de nouveaux capteurs, ou bien pouvoir envoyer les cartes à de nouveaux types de clients, répartis sur le réseau de l'INRIA, voire sur un site Web (c'est un exemple). Cette architecteure permet de modulariser de manière souple et libre chaque niveau.

Dans notre cas, les 3 couches de l'architecture sont composées des éléments suivants :

- un module tracker déporté ou encore module tracker++,
- le serveur de cartes lui-même,
- des clients interrogeant le serveur de cartes.

Chacun des modules sera détaillé dans les sections suivantes, cependant voici une rapide introduction des fonctionnalités de chacun :

Le trackerConnector Notre infrastructure est alimentée par différents capteurs (caméras, Cycab, ...) qui envoient leurs informations à un module que nous avons développé, éventuellement en passant par un programme intermédiaire (par exemple les trackers) fournit par une autre équipe. L'objectif de ce composant est de préparer les données pour envoi au *mapServer*.

Le *mapServer* Cœur de l'infrastructure, toutes les données recueillies sont centralisées, éventuellement traitées, puis envoyées aux différents clients connectés. Il devra être ouvert afin de pouvoir intégrer différents modules de traitements expérimentaux : association et/ou fusion de données, autres représentations de l'environnement, ...

Nous avons implémentés deux modules expérimentaux : le jPDA[2] et le BOF[3] [Cou03]. Seul le jPDA est en rapport avec notre projet et Parknav, puisqu'il s'agit d'un algorithme permettant de faire de l'association de données, en s'appuyant sur les observations à t-1 et celles à t. Cet algorithme est abordé dans le paragraphe 6.2.

Les clients Ils seront demandeurs de la carte de l'environnement, afin soit de l'afficher (cas du client graphique), ou bien dans le cas d'un robot mobile, pour adapter son comportement en fonction des éléments détectés (changement de trajectoire par exemple).

8.2.2.2 Choix technologiques

En dehors des choix « imposés » à savoir le langage C++ et GNU/Linux, nous sommes repartis du cahier des charges pour exprimer les contraintes que nous avons, et nous avons ainsi mis en évidence plusieurs éléments, listés dans le tableau 8.4.

[2] joint probabilistic data association

[3] Bayesian Occupancy Filter ou Grille d'occupation bayésienne

1. Un transfert réseau très rapide
2. Une librairie de « parsing »[a] XML
3. Certains calculs mathématiques imposent l'usage de librairies dédiées
4. une API[b] permettant de développer un client graphique
5. des API spécifiques pour les modules du *mapServer*

[a]Interprétation d'un fichier XML
[b]Application Programming Interface, ensemble d'outils, de fonctions permettant de construire une application

TAB. 8.4 – Choix technologiques : conditions à vérifier

Protocole Réseau (tableau 8.4, point 1) L'interfaçage avec les *trackers* ou tout capteur en entrée (le Cycab par exemple) dépendra des moyens de communication dont dispose le capteur en question.

Par exemple, dans le cas de PrimaBlue, comme nous l'avons indiqué dans le tableau 4.2, il est capable d'envoyer le flux d'observations par le biais d'une socket TCP classique.

Concernant les communications au sein de notre architecture, nous sommes partis sur le protocole UDP - décrit dans la RFC768[4,5] et sur le Web [_S03]. UDP a le grand avantage, par rapport à TCP [_S04], de la rapidité du flux grâce à son mode non connecté (et à l'absence de contrôle d'erreur).

Cependant, ce choix implique quelques travers, comme par exemple la non garantie d'arrivée d'un packet réseau. Ceci n'est pas un handicap, car nous sommes dans un environnement temps-réel où, de toute façon, nous ne pouvons pas nous permettre d'avoir des retransmissions de paquets en cas d'échec, il est préférable d'attendre le paquet suivant.

Nous avons choisi la librairie eNet créée initialement pour le jeu Cube (voir le site Web officiel[6]). Ce jeu devait pouvoir envoyer des données très rapidement avec une latence minimale.

Pour ce faire, eNet s'appuie sur la couche UDP, tout en proposant des fonctionnalités propres à TCP ou bien innovantes, comme listées dans le tableau 8.5.

Librairie gratuite avec libre accès aux sources
Basée sur UDP (rapidité)
Implémentation d'un mode *reliable* (arrivée garantie)
Séquencement des paquets respectés
Taille de paquets quelconque
Portable (Unix, Win32, PocketPC,...)

TAB. 8.5 – Fonctionnalités de eNet

L'une des limites d'eNet - comme de nombreux « petits » projets de logiciels libres[7] - est le manque de documentation ; nous avons eu de nombreux déboires avec cette librairie, mais grâce au forum et en consultant le code source, nous sommes arrivés à obtenir le résultat escompté. Cependant, ceci a eu un impact sur le temps de développement total du projet.

[4]Requests for Comments : spécifications officielles des normes, standards et implémentations
[5]http ://www.frameip.com/rfc/rfc768.php
[6]http ://enet.bespin.org/ Dernière consultation : 20-juil-05.
[7]Licence MIT, voir annexe C

Il est à noter qu'en plus d'eNet, nous utiliserons aussi la couche TCP pour un module de configuration à distance du *mapServer*.

Traitement XML (tableau 8.4, point 2) Nous avons décidé d'utiliser un format décrit en XML pour les flux de communications inter-programmes, sachant que nous utilisons déjà une librairie dédiée pour interpréter ce type de format.

Nous nous sommes interrogés à plusieurs reprises au démarrage des spécifications sur ce choix, craignant d'avoir des pertes de performance dues au formatage des données en XML.

La problématique XML a dans un premier temps trouvé réponse par la librairie Xerces-C++ [Apa04]. Cette librairie Open Source[8] n'a pas été retenue à cause de sa lourdeur (taille des librairies), de sa complexité de mise en œuvre et du manque de documentation.

Le choix final s'est porté sur la libXML2. Cette librairie a été initialement développée pour le projet GNOME sous GNU/Linux. Elle est en fait utilisable dans tout programme nécessitant un parser XML. Bien que construite en C, elle est intégrable en C++ sans problème et est parfaitement multi plates-formes (GNU/Linux, Unix, Windows, CygWin, MacOS, MacOS X, RISC Os, OS/2, VMS, QNX, MVS, ...).

Elle est gratuite, avec libre accès aux sources[7]. Utilisée abondamment dans l'environnement GNOME, elle bénéficie d'un nombre conséquent d'utilisateurs. Des exemples de code seront donnés dans la suite de ce chapitre et montreront la simplicité de son utilisation.

Librairie mathématique (tableau 8.4, point 3) La librairie mathématique abordée au point 3 du tableau 8.4 est la GNU Scientific Library[9] ou GSL.

Tout comme les librairies ci-dessus, elle est gratuite avec libre accès aux sources[10] . Elle couvre de nombreux domaines, tels que les nombres complexes, les matrices, les histogrammes, les calculs statistiques, ...

Un point important à noter pour la GSL est la licence utilisée : la GPL ou GNU General Public License. Il s'agit d'une licence dite « virale » dans la mesure où tout logiciel l'utilisant se doit d'être GPL à son tour. A priori, ceci n'est pas un problème dans notre contexte - laboratoire de recherches, mais dans le cas où il nous faudrait une licence plus permissive pour notre plate-forme, il faudra se tourner vers d'autres solutions (non étudiées dans ce rapport).

API graphique (tableau 8.4, point 4) Notre choix s'est porté sur la librairie ou plutôt la spécification OpenGL[11] [WNDS03], en l'occurrence sur Mesa[12], qui est une implémentation Open Source[7] d'OpenGL.

L'avantage de cette librairie est qu'elle permet de créer du code qui sera correctement géré par le matériel optimisé OpenGL (toutes les dernières cartes graphiques). Cette spécification est abondamment utilisée dans les domaines des jeux vidéos, gros consommateurs en ressources graphiques. Elle permet aussi bien de faire de la 3D que de la 2D, ce qui est notre cas dans un premier temps.

Pour information, nous avons procédé à quelques tests comparatifs du client graphique (voir section 8.6), avec un driver XWindows optimisé pour OpenGL et sans optimisation. Les résultats du tableau 8.6 se passent de commentaires.

[8] Licence Apache, voir annexe C, page 127
[9] http ://www.gnu.org/software/gsl/. Dernière consultation : 24-juin-05
[10] Licence GPL, voir annexe C
[11] http ://www.opengl.org/. Dernière consultation : 24-juin-05
[12] http ://www.mesa3d.org/. Dernière consultation : 23-juin-05

	FPS Moyen[a]
Sans optimisation	40
Avec	200

[a]Frame per second ou image par seconde

TAB. 8.6 – Impact d'un driver

OpenGL est spécifiée de telle sorte qu'elle est indépendante du système d'exploitation et des gestionnaires de fenêtres, ce qui lui permet d'être multi plates-formes. OpenGL et les librairies annexes existent sur de nombreux matériels, dont les Pocket PC, ce qui ouvre les possibilités de développement. En plus de Mesa, nous utilisons aussi freeglut[13] qui est une implémentation libre[7] d'OpenGL Utility Toolkit (GLUT). Cette librairie a pour but de faciliter la gestion des fenêtres et des périphériques d'entrée (type clavier, souris).

OpenGL ne contient aucune commande permettant d'ouvrir des fenêtres ou bien de gérer des événements d'entrée type clavier ou souris, c'est là qu'intervient GLUT. Il simplifie cette gestion interactive tout en proposant en plus des fonctions de création d'objets en 3D, telle qu'une sphère, ...

Particularités des modules (tableau 8.4, point 5) Différents modules d'expérimentations seront intégrés dans le serveur de cartes, en fonction des besoins. Nous pouvons citer d'ores et déjà le module qui sera en charge de faire de l'association de données, le jPDA[14].

Chacun de ces modules utilisera des librairies spécifiques. Par exemple, le jPDA nécessite la librairie ProBT© pour pouvoir utiliser des composants probabilistes tels que les filtres de Kalman [WB04].

8.2.3 Données pour caractériser une observation ou une cible

Les trackers et le Cycab envoient leurs données dans un format qui leur est propre.

De notre côté, nous avons besoin d'un format de données défini pour toute l'architecture à développer. Comme nous l'avons dit dans la section précédente, nous sommes partis sur un formalisme type XML. Nous allons maintenant voir quels types de valeurs nous allons prendre et comment elles seront formatées.

Le tableau 8.7 présente la structure que nous avons appelée commData (pour communication de données).

Certains paramètres seront (re)calculés par nos programmes. Ainsi le premier module de notre chaîne, le trackerConnector, calculera :

- x, y,
- vx, vy,
- covXX, covYY, covXY,
- tv (dans le cas du Cycab qui ne fournit pas d'horodatage des données).

Cette structure de données commData va être transformée en une trame XML pour pouvoir être envoyée au serveur de cartes.

L'exemple figure 8.5 montre une trame pour une observation. De cette trame, nous pouvons déduire qu'il s'agit d'une observation provenant d'un tracker PrimaBlue (tracker_id = 0), en sortie du trackerConnector (les coordonnées x,y et les vitesses vx,vy ont été calculées). Nous savons aussi que pour ce tracker, la caméra utilisée était la numéro 2, mais ceci ne nous dit pas qu'elle est la caméra exacte sur les 6 que nous utilisons (cette information n'est pas pertinente une fois que nous sommes au niveau du serveur de cartes).

[13]http ://freeglut.sourceforge.net/index.php Dernière consultation : 24-juin-05
[14]Joint Probabilistic Association Data

Valeurs	Types	Descriptions
cible_id	string	« cycab » si le capteur est le cycab, sinon un identifiant défini par le tracker
camera_id	int	Numéro de caméra pour appliquer les bonnes transformations
tracker_id	int	0 = BEV, 1 = Cycab, 2 = PrimaLab, ...
xi, yi	double	Coordonnées dans le plan image
x, y	double	Coordonnées dans le plan parking
vx, vy	double	Vitesse
covXX, covYY, covXY	double	Matrice de covariances donnant une indication de taille de cible
theta	double	Angle du Cycab
vtrans	double	Vitesse du Cycab
phi	double	Angle de braquage
tv	struct timeval	Horodatage (msec)

TAB. 8.7 – Structure de données de l'architecture

```
<?xml version = '1.0' encoding = 'UTF−8'?>\
<observation cible_id="[WO:30078]" tracker_id="0" camera_id="2"
    timeSeconds="1119010015" timeUSeconds="941000" xi="352"
    yi="161" x="30.7724" y="5.01281" vx="0.17484" vy="0.0284814"
    cxx="0.240474" cyy="0.463959" cxy="72.0489"theta="0"
    vtheta="0" phi="0" vtrans="0" />
```

FIG. 8.5 – Exemple de trame XML commData

Gestion des sources En dehors des besoins identifiés dans le tableau 8.4, nous faisons aussi quelques choix liés au contexte de l'INRIA ou aux compétences des différentes personnes travaillant sur le projet.

Compilation La compilation est gérée par un outil nommé *scons*[15] [Kni04].

Il s'agit d'un remplaçant du classique *make* et de son *Makefile*, qui a le gros avantage d'utiliser des scripts développés en Python, ce qui permet de bénéficier de ce langage puissant pour pouvoir gérer toutes les sources de manière efficace.

[15] http ://www.scons.org. Dernière consultation : 23-aou-05

Il est par exemple très facile de détecter si toutes les librairies nécessaires à la compilation sont installées ou non sur la machine, et le cas échéant, d'afficher un message prévenant l'utilisateur.

Les sources La gestion des sources à l'INRIA est faite principalement avec l'outil CVS[16], alias Concurrent Versions System. Il s'agit certainement de l'outil de versionning le plus utilisé actuellement, entre autres, parce que :
- il est gratuit,
- facile à installer (que ce soit la partie serveur ou bien la partie cliente),
- il existe de nombreuses interfaces graphiques offrant un grand confort d'utilisation,
- multi plates-formes,
- la plupart des logiciels d'édition propose une gestion CVS,
- il permet un développement en équipe (gestion de conflits, de verrous, ...).

La marche à suivre pour récupérer les sources avec CVS est décrite sur le site Web de Parkview, vous pouvez aussi la retrouver dans l'annexe E. Dans cette annexe, vous retrouverez la figure E.1 qui représente l'arborescence des fichiers créée pour ces développements.

Les diagrammes de classes que nous avons dans la suite de ce rapport, utilisent les mêmes conventions que ceux générés par doxygen, vous en trouverez la légende dans le tableau 8.8.

Symboles	Description
+	Membre ou méthode public
#	Protégé
-	Privé

TAB. 8.8 – Légende pour les diagrammes de classe (format doxygen)

8.3 Modules communs

Les différentes applications que nous avons développées utilisent soit du code spécifique à un module (par exemple les informations de caméra ne sont pertinentes que pour le trackerConnector), soit du code commun et réutilisable par tous.

Le but de cette section est de voir les modules de code communs, avant de rentrer dans le détail de chaque application. Ces différents modules sont tous dans le répertoire *src* sous la racine de Parkview (voir figure E.1).

8.3.1 Les traces

Lorsque nous sommes amenés à développer une application, il faut très rapidement mettre en place un système de traces, pour pouvoir *debugger* le déroulement des programmes. C'est le but du module « traces » constitué des fichiers sources *traces.hpp*, *traces.cpp* et *oformstream.hpp*.

Ces programmes sont assez simples et ne font qu'implémenter des fonctions d'horodatage, soit uniquement horaire (rôle de la fonction *std : :string timestamp ()*), soit complet sous le format $YYYY-MM-DDHH:MM:SS$ (rôle de la fonction *std : :string fulltimestamp()*), ainsi qu'un ensemble de macros d'affichage (figure 8.6).

[16]https ://www.cvshome.org/. Dernière consultation : 23-aou-05

```
#ifndef TRACE_DEBUG_N1
#define TRACE_DEBUG_N1( s...) \
        if (debug_level>=1) \
                do {    printf("-I- "s ); \
                        printf("\n"); \
                        fflush (stdout); \
                } while(0)
#endif // TRACE_DEBUG_N1
```

FIG. 8.6 – Traces : exemple de code pour les traces de niveau 1

Cette macro utilise la variable « debug_level » que nous retrouverons dans tous les programmes de l'architecture. Suivant la valeur de cette variable, le texte demandé sera affiché (debug_level >= 1) ou non.

A noter que nous avons dû créer une fonction similaire à la classique sprintf du C/C++. Effectivement, afin d'utiliser du code le plus standard par rapport au C++ (cf. [Sil98] et [Str97]), nous utilisons des variables de type *string*, qui ont le grand avantage d'inclure des destructeurs gérant correctement la récupération de l'espace mémoire après utilisation (contrairement au type *char* classique du C).

Or, il n'y a pas de fonction de formatage comme sprintf avec ce type de variable. C'est le rôle du module oformstream.

8.3.2 « Parser » de configuration

Ce module a pour objectif de fournir une classe de base pour la lecture des fichiers de configuration XML, fichiers permettant de modifier le comportement d'un programme au démarrage de celui-ci. Tous les programmes (*trackerConnector*, *mapServer*, *gclientMapServer* - le client graphique) utilisent ces fichiers de configuration.

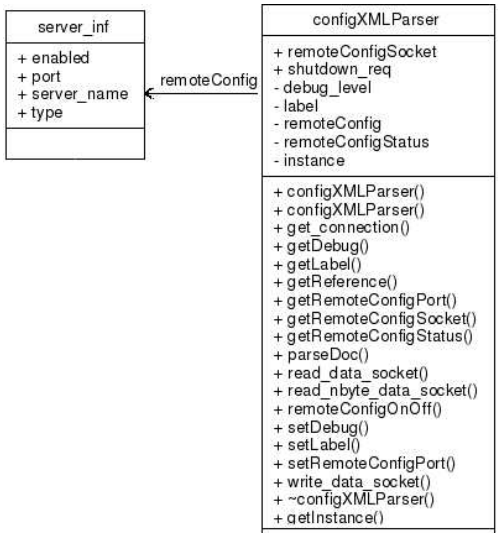

FIG. 8.7 – Classe configXMLParser

Cette classe intègre deux types de fonctionnalités : la gestion des fichiers de configuration et la gestion de la configuration à distance dans le cas où cette option est activée.

La structure *server_inf* permet d'indiquer tous les paramètres définissant un serveur : nom ou adresse ip de la machine, le port réseau à utiliser, un type de serveur (surtout utilisé pour indiquer la nature du tracker) et enfin, un paramètre permettant d'activer ou non le serveur en question.

Méthodes implémentées Comme dit précédemment, cette classe incorpore deux parties différentes : la gestion de configuration et la configuration à distance.

8.3.2.1 Gestion de configuration

L'objectif est de pouvoir « parser » un fichier XML et définir les champs de manière appropriée. Le tableau 8.9 recense les principales méthodes de cette classe. Vous retrouverez aussi sur la figure 8.8 le graphe des méthodes principales.

FIG. 8.8 – Graphe des méthodes de parsing

Méthodes	Description
configXMLParser( )	Constructeur de la classe qui initialise les différentes variables et la structure *server_inf* pour une configuration à distance.
void **setLabel**(const char * aLabel)	Permet de donner un nom à cette configuration.
const char ***getLabel**( )	Retourne le nom défini avec setLabel.
void **setDebug**(int aLabel)	Positionne la variable *debug_level* pour obtenir le niveau de traces voulu.
int **getDebug**( )	Retourne la valeur courante de la variable *debug_level*.
void **parseDoc**(const char *docname, const char *rootString)	Cette méthode démarre le « parsing » du fichier XML. Elle récupère la déclaration XML puis l'élément racine.
virtual void **getReference**(xmlDocPtr doc, xmlNodePtr current)	Appelée par parseDoc, c'est la méthode qui récupère et traite tous les éléments fils de la racine.

TAB. 8.9 – Méthodes de la classe configXMLParser

parseDoc permet de construire l'arbre des noeuds suivant un principe proche de DOM[17] [YoL05], [Vei05].

La méthode *getReference* est déclarée en tant que fonction virtuelle pure (utilisation du mot clé *virtual* et la déclaration est suivie par un « 0 » comme illustrées dans la figure 8.9). Ceci veut dire que tous les programmes ayant besoin de cette classe, devront implémenter cette fonction. Si un module doit traiter un nouveau type de fichier XML, il faudra ré-écrire cette méthode.

$$\boxed{virtual\ void\ getReference(xmlDocPtr\ doc,\ xmlNodePtr\ current)\ =\ 0;}$$

FIG. 8.9 – Fonction virtuelle pure

[17] Document Object Model

La figure 8.10 représente un extrait de l'une des implémentations de la fonction *getReference*.

```
...
current = current->xmlChildrenNode;

while ( current != NULL ) {
    if ( ! xmlStrcmp( current->name, (const xmlChar*) "general" )) {

      // If we have to compute distortion or not
      valeur = xmlGetProp (current, (const xmlChar*)"distort");
      if ( valeur != NULL ) {
          setDistort ( (char*)valeur );
          // Do not forget this to clean up memory !!
          xmlFree( valeur );
      }
    ...
    }
    // Next children node
    current = current->next;
}
```

FIG. 8.10 – Exemple d'implémentation de la fonction *getReference*

Dans cet exemple, nous voyons qu'il y a une boucle balayant tous les noeuds fils du fichier. Puis pour un noeud donné (par exemple ici « general » nous allons traiter les attributs qui nous intéressent ; ici « distort » qui représente un drapeau pour savoir si nous devons calculer la distorsion ou non.

8.3.2.2 Configuration à distance

Nous souhaitons, pour au moins un programme (le serveur de cartes), pouvoir utiliser un outil de configuration à distance. Les couches de bas niveau sont implémentées dans la classe configXMLParser : activation du serveur, acceptation des connexions, écriture et lecture de données (cf. tableau 8.10).

Méthodes	Description
void **setRemoteConfigPort**(int port)	Définie le numéro de port sur lequel écouter pour la configuration à distance.
int **getRemoteConfigPort**( )	Retourne le numéro de port en écoute.
bool **getRemoteConfigStatus**( )	Renvoie le statut courant du serveur de configuration à distance.
int **getRemoteConfigSocket**( )	Le numéro de la socket est renvoyé.
int **get_connection**(int s)	Si un client veut établir la connexion, cette méthode acceptera la demande.
int **write_data_socket**(int s, char *buf)	Envoi de données vers le client.
int **read_nbyte_data_socket**(int remote_socket, char *buf, int n)	Lecture de n octets en provenance du client.
int **read_data_socket**(int remote_socket)	Lecture d'un nombre quelconque de caractères du client.

TAB. 8.10 – Méthodes pour la configuration à distance

Pour une utilisation de ces méthodes, voir la section 8.5.8. Elle présente le développement d'un « petit » client en Python permettant de contrôler le fonctionnement du serveur de cartes.

8.3.3 Classes de communication

En dehors de la configuration à distance, la majorité des objets de communication utilise le protocole eNet. Pour ce faire, nous avons développé des classes de type client et serveur que nous allons détailler dans cette partie, après avoir abordé le format exact du flux de communication.

8.3.3.1 Principes mis en œuvre

Comme nous l'avons expliqué plus haut, les observations ou cibles sont envoyées dans un format XML. En fait nous n'envoyons pas directement les trames XML, nous avons défini un protocole de communication, comportant différents types de message suivant la nature de l'échange, la structure générale est reprise dans le tableau 8.11.

Libellé	Définition	Unité	Type (octet)
Taille	Taille du message	-	4
Type de message	Indique la nature de la transmission	$MsgType$	1
Num. client	Numéro du client destinataire	int	1
Données	Les données à envoyer	Optionnel	Variable

TAB. 8.11 – Format d'un message Parkview

Tous les échanges dans l'architecture Parkview vont être structurés comme indiqué, autrement dit : les flux de connexion/déconnexion, les observations, les cibles, les données statiques, ... Comme indiqué dans le tableau, les données sont optionnelles, autrement dit, certains types de message peuvent se limiter à l'en-tête constitué de la taille, du type de message et d'un numéro de client.

Le type de données MsgType est une énumération des différents types que nous avons implémentés, en voici un extrait (cf tableau 8.12).

Type	Description
CONNECTOR_HELLO	Message envoyé par le *trackerConnector* pour se présenter au serveur de cartes.
GCLIENT_HELLO	Idem pour un client graphique.
CONNECTOR_BYE	Le *trackerConnector* se déconnecte du serveur de cartes.
CLIENT_BYE	Idem pour un client.
MS_HELLO	Le serveur de cartes confirme qu'il s'agit bien de lui.
MS_BYE	Le serveur de cartes s'arrête et informe les clients.
TC_OBS	Le *trackerConnector* envoie au *mapServer* les observations.
MS_TARGET	Envoi des cibles aux clients.
STATIC_MAP	Le serveur de cartes envoie le plan aux clients.
STATIC_GEOMETRY	Les clients peuvent avoir besoin de la géométrie des objets (le Cycab par exemple pour pouvoir le dessiner).

TAB. 8.12 – Types de messages Parkview

Le message CLIENT_BYE est un exemple de type sans données. A contrario, STATIC_MAP va contenir un volume de données important, puisqu'il y a envoi de tout le plan du parking dans un format XML, soit plus de 13 Ko d'informations.

8.3.3.2 Classe de base

La figure 8.11 représente la classe PVCommBase qui implémente les méthodes et membres de base, utiles à la fois pour la partie cliente et pour les serveurs.

FIG. 8.11 – Classe de base de communication

Les méthodes de cette classe (tableau 8.13) définissent les primitives d'envoi d'un message, ainsi que la récupération des messages dans la file de réception.

Méthodes	Description
PVCommBase(const char *hostname, int port)	Constructeur qui va initialiser le module eNet et positionner un nom de serveur et un numéro de port.
int **nbActivePeers**( )	Retourne le nombre de partenaires connectés.
void **sendPacket**(int client_num, ENETPacket *aPacket)	Fonction utilisant le bas niveau eNet pour l'envoi d'un paquet à un client donné ou bien à tous les clients (si client_num = -1)
void **sendMessage** (int client_num, int typeMsg, ENetPacketFlag flag, const char * aMessage, unsigned int msgSize)	Construction d'un message typeMsg pour un client donné (ou tous).
void **sendMessage** (int client_num, int typeMsg, ENetPacketFlag flag, const char * aMessage, unsigned int msgSize, int param1)	Construction d'un message typeMsg pour un client donné (ou tous), avec du texte et un paramètre entier.
MessageList & **getMessage**( )	Retourne la liste des messages en attente dans la file

TAB. 8.13 – Méthodes de PVCommBase

La structure *MessageList* est définie de la sorte : *typedef deque<string> MessageList ;*. Le type C++ *deque* est similaire à un vecteur dans lequel il est possible de faire des insertions rapides en début ou fin de chaîne.

8.3.3.3 Le client eNet

Dérivant de la classe PVCommBase (voir la figure 8.12), l'objectif est d'implémenter tout ce qui est nécessaire pour gérer un client de communication. Le client rajoute quelques méthodes importantes par rapport à PVCommBase (tableau 8.14).

Méthodes	Description
virtual bool **processEvents**(int duration)	Fonction appelée à fréquence régulière pour gérer les évènements eNet (comme la réception de messages ou la déconnexion).
void **shutdown**(int sig_recu)	Lorsque le client s'arrête, il y a envoi d'un message (CLIENT_BYE par exemple) au serveur de cartes.
void **helloMsg**(char * hostname, int port, int typeMsg)	Le client annonce au serveur qui il est.
virtual bool **process** (ENetPacket *packet, int sender) = 0	Fonction virtuelle pure, traitant les messages reçus.

TAB. 8.14 – Méthodes importantes du client

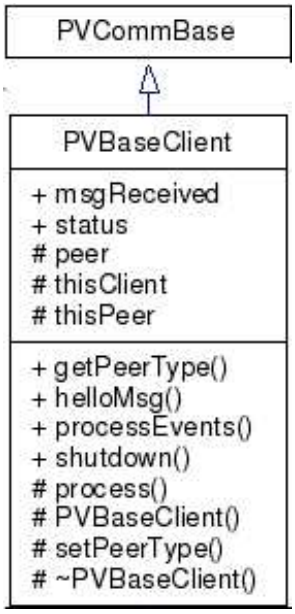

FIG. 8.12 – Héritage de la classe client PVBaseClient

La fonction *process* est une fonction virtuelle pure (comme vu à la figure 8.9), ce qui veut dire que chaque programme utilisant cette classe doit implémenter sa propre fonction. Ceci est logique puisque son but est de traiter les message entrants, dans le format Parkview, et de déclencher les actions souhaitées. Il est important de noter que la méthode *processEvents* doit être appelée à fréquence régulière (minimum 1 Hz) ; sans cela, la connexion échouera pour dépassement de temps.

8.3.3.4 La partie serveur

Dérivant de la classe PVCommBase, l'objectif est d'implémenter tout ce qui est nécessaire pour créer et gérer un serveur de communication.

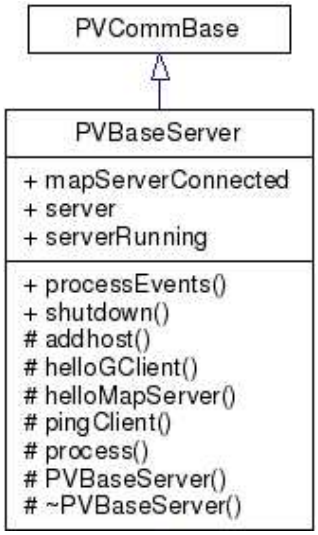

FIG. 8.13 – Graphe de la classe PVBaseServer

Les méthodes utilisées dans cette classe (voir tableau 8.15) sont assez proches de celles du client. Nous retrouvons par exemple les méthodes *processEvents* et *process*.

Méthodes	Description
virtual bool **processEvents**(int duration)	Fonction appelée à fréquence régulière pour gérer les évènements eNet (comme la réception de messages).
void **shutdown**(int sig_recu)	Lorsque le serveur s'arrête, il y a envoi d'un message (MS_BYE par exemple) à tous les clients connectés.
virtual bool **process** (ENetPacket *packet, int sender) = 0	Fonction virtuelle pure, traitant les messages reçus.
int **addhost**(client host)	Lorsqu'un nouveau client se connecte, ses références sont rajoutées dans la liste des clients.

TAB. 8.15 – La classe PVCommServer

8.3.4 Le simulateur

Une telle architecture nécessite de nombreux tests, non seulement pour valider, recetter les nouveaux développements, mais aussi pour débugger l'infrastructure. Le souci est que chaque test réclame les composants suivants a minima : un tracker, un tracker Connector, un serveur de cartes et un client graphique.

PrimaBlue sait travailler sur des vidéos pré-enregistrées, mais son lancement est lourd et nécessite une installation sur la machine cliente, sans parler des impacts sur les performances globales de la machine.

Sortir le Cycab à chaque fois est encore plus contraignant, cela nécessite de démarrer la voiture, de sortir les balises et de faire le positionnement initial. Il existe un simulateur du Cycab, outil très performant simulant toutes les options et comportements de la voiture. Cependant, la version actuelle ne permet pas un positionnement dans le plan du parking, indispensable pour nos traitements. Toujours dans l'optique de faciliter les démonstrations et la mise au point des programmes, nous avons développé un mode simulateur permettant de rejouer des fichiers générés lors d'un scénario réel (ou bien par exemple en rejouant des vidéo pré-enregistrées avec PrimaBlue). Ces fichiers peuvent être créés à différents points de la chaîne de traitement, que ce soit au niveau du *trackerConnector*, du *mapServer*, voire du client graphique (figure 8.14). A chaque fois, il y a écriture des données en cours (observations ou cibles) dans un fichier au format XML.

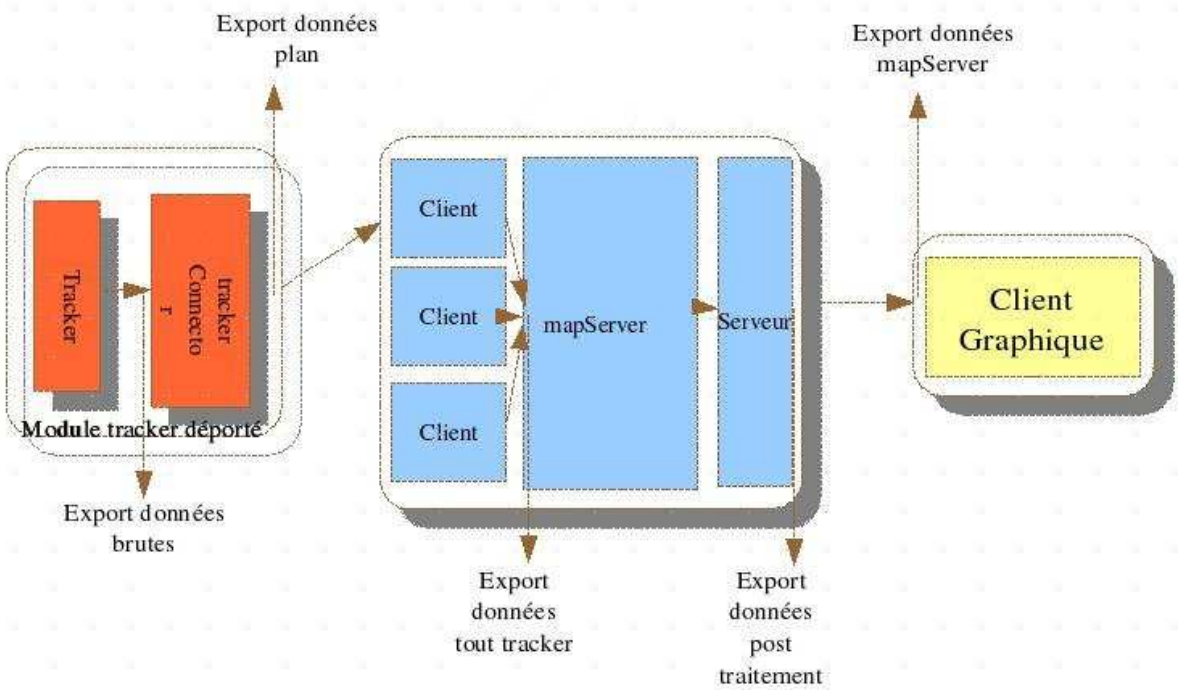

FIG. 8.14 – Export de données.

L'objectif du simulateur est de pouvoir lire ce type de fichiers et le rejouer à l'identique, éventuellement en boucle. La classe *PVSimulBase* (figure 8.15) implémente les méthodes de base qui seront utilisées par les programmes fournissant ce service.

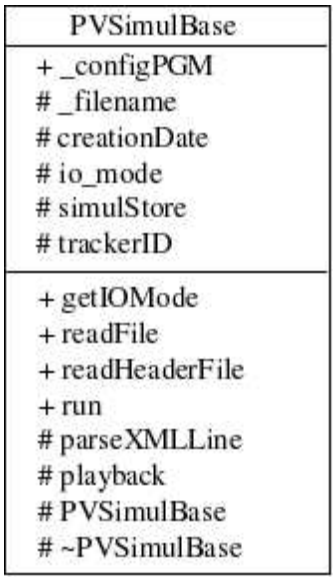

FIG. 8.15 – PVSimulBase : classe de base du simulateur

Description des méthodes et membres de *PVSimulBase*

La figure 8.16 montre la définition de la structure *playBackRecords*, qui contient des enregistrements de type commData, la structure que nous avons définie. La clé de cette structure est de type double et contiendra les heures de chaque observations prises (le *timestamp*).

```
typedef map< double, commData, ltfloat > playBackRecords;
```

FIG. 8.16 – Structure de sauvegarde des données du fichier

Le tableau 8.16 met en évidence une fonction virtuelle pure, *playback*, qui devra être implémentée dans tous les programmes fournissant un service de type simulateur.

Méthodes/Membres	Description
string _filename	Le nom du fichier à traiter.
string creationDate	La date de création du fichier est stockée dans l'en-tête de ce dernier.
int io_mode	Détermine si les données sont en sortie (=1) ou en entrée (=0) du programme générateur.
playBackRecords simulStore	Structure pour le stockage des trames XML.
int trackerId	Identifie la nature du capteur (PrimaBlue, Cycab, ...).
int getIOMode( )	Accesseur pour récupérer le mode du fichier (entrée ou sortie).
int readFile( )	Méthode pour la lecture du fichier à rejouer. Toutes les trames lues seront stockées dans simulStore.
int readHeaderFile( )	Traite la première ligne du fichier pour récupérer ses paramètres (trackerId, io_mode, ...).
int run( )	Boucle principale du simulateur qui rejoue les trames.
void parseXMLLine(const char *ligne, commData *cdTemp)	Méthode qui traite une ligne XML du fichier et stocke les valeurs dans une structure commData.
virtual bool playback(int io_mode, int sleepDuration, commData simulData)=0	Méthode virtuelle (cf. figure 8.9), cœur du rejeu.

TAB. 8.16 – Méthodes et membres de la classe PVSimulBase

Comme nous l'avons dit ci-dessus, un fichier (exemple dans la figure 8.17) contiendra les données exportées par le programme, dans un format XML, ainsi qu'une ligne d'en-tête donnant les renseignements suivants :

- le type de capteur/tracker (PrimaBlue, Cycab, ...),
- le point d'export (en entrée du programme générateur ou en sortie),
- la date de création du fichier d'export.

```
cycab,OUT,2004−05−04 13:42
<?xml version = '1.0' encoding = 'UTF−8'?>
<observation cible_id="cycab" tracker_id="1" camera_id="0" timeSeconds="1115206823"
   timeUSeconds="271753" xi="0" yi="0" x="41" y="6" vx="0" vy="0" cxx="0"
   cyy="0" cxy="0" theta="3.1415" vtheta="0" phi="0" vtrans="0" />
```

Fig. 8.17 – Lignes d'un fichier d'export pour un Cycab, en sortie

Le rejeu de ces données pose plusieurs problèmes : comment synchroniser le rejeu pour deux exports faits en même temps (par exemple, l'export de la sortie d'un connecteur sur PrimaBlue et celui d'un connecteur sur le Cycab) ? De même comment relire des trames de deux exports faits à des instants différents (scénarios différents) ?

Dans le cas du rejeu d'un seul fichier, il n'y a pas vraiment de problème, puisqu'il suffit de rejouer toutes les trames avec l'écart de temps de chacune d'elle. Dès qu'il y a deux fichiers ou plus, soit ils sont joués « en même » temps, autrement dit il faut appliquer le même principe que ci-dessus, soit nous maintenons l'écart entre les fichiers.

Supposons que le premier fichier contienne des données, issues d'un trackerConnector, en provenance de PrimaBlue et exportées à 14 :05 :00. Le second fichier a été créé à 14 :10 :00 d'un trackerConnector avec un Cycab en entrée. Nous voulons rejouer ces données, ce qui veut dire lancer deux trackers, en conservant l'écart entre les deux temps.

Nous avons développé un outil permettant de lancer « simultanément » plusieurs modules (trackerConnector ou serveur de cartes par exemple) en mode simulation, chacun avec un fichier : le « magnétoscope ». Le principe est repris dans la figure 8.18, avec l'exemple d'un lancement de deux trackerConnector en même temps (en fait, avec un écart infime de ∂t).

Nous calculons au préalable l'écart entre les premières observations de chaque fichier ($\triangle Obs$).

Nous avons pris la décision de limiter cette valeur à un maximum de 30 mn. Si l'écart entre les deux fichiers dépasse cette limite, alors ils seront rejoués en parallèle.

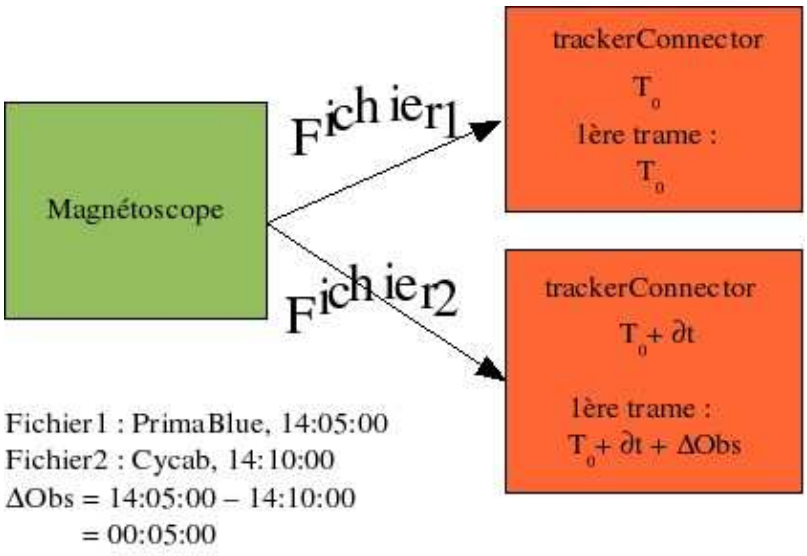

FIG. 8.18 – Principe et exemple du magnétoscope

8.3.5 Les outils

Nous avons aussi développé quelques fonctions d'outillage afin de rendre des services spécifiques, tels que :

- purge de structure STL[18] type *map*,
- purge des séquences,
- compteurs pour les mesures de performance,
- gestion de « timer ».

8.3.5.1 Purge des structures évoluées

Lorsque l'on développe en C++ « moderne », très vite nous sommes amenés à utiliser la STL et des structures de type *string*, *map*, *sequence*, ...

La gestion en est globalement simplifiée et automatisée. Ainsi avec une variable de type *string* (chaîne de caractères), il n'est pas nécessaire de connaître la taille initiale, ni de se préoccuper de l'allocation/désallocation de la mémoire.

Le problème est que les structures plus complexes type *map* ou *sequence*, peuvent contenir des pointeurs vers d'autres types de variable ; dans ce cas, le programme à l'exécution, ne saura pas comment nettoyer la mémoire.

C'est pour cela que nous avons développé un *template* ou patron C++ [Sil98], [Str97] permettant de nettoyer ce type de variable (figure 8.19).

[18]Standard Template Library. Bibliothèque fournissant des structures évoluées.

```
/** A function template to completely purge any STL map.
 * @author e-Motion / @param c The map to be purged.
 */
template<class Map> void purgeMap(Map& c) {
        typename Map::iterator i;
        for(i = c.begin(); i != c.end(); i++) {
                delete i->second;
                i->second = 0;
        }
        c.clear();
}
```

FIG. 8.19 – Template de nettoyage d'objets complexes (*map*) de la STL

L'avantage de ce patron est qu'il va pouvoir faire le nettoyage de la structure quel que soit son contenu. Ceci permet d'éviter le phénomène de fuite mémoire lié à ce nettoyage.

8.3.5.2 Compteur de performance

Afin de pouvoir mesurer la performance d'un traitement, nous avons implémenté une classe *PV_PerfCounter*, définie comme indiqué sur la figure 8.20.

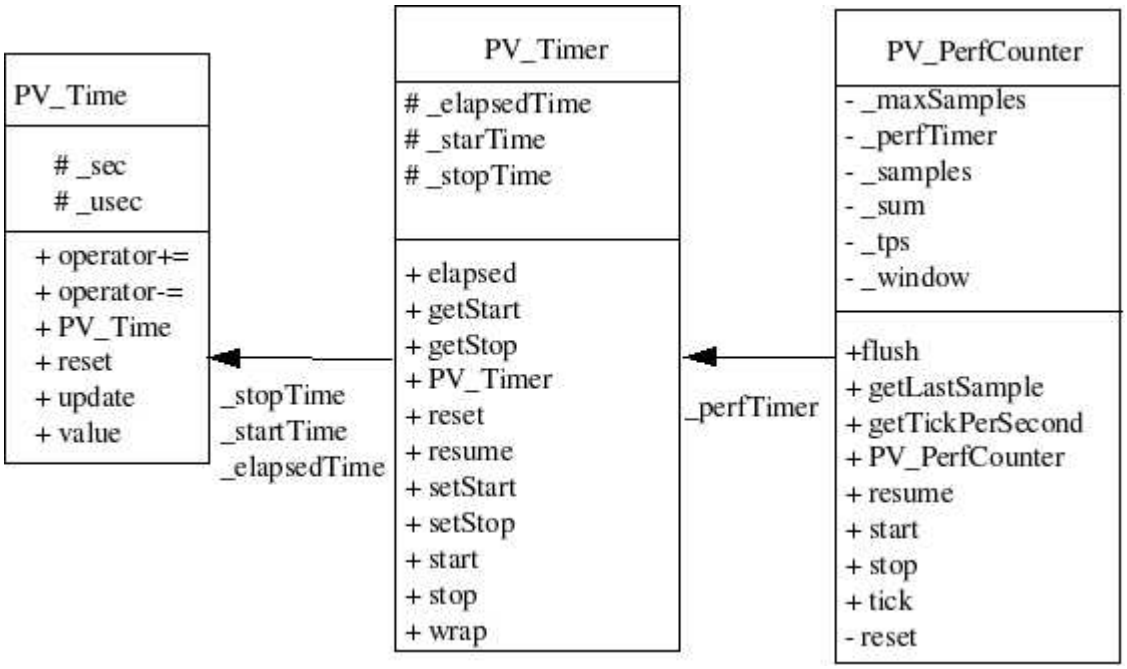

FIG. 8.20 – Schéma de la classe PV_PerfCounter

Le principe de cette classe est décrit dans la figure 8.21.

Initialisation du compteur (PV_PerfCounter compteur(nombre echantillons))
Tant que (le programme tourne) **faire**
  Si (évènement à mesurer) **Alors**
    Incrément compteur, méthode tick
  Fin Si
  Si (affichage statistique) **Alors**
    Appel à getTickPerSecond
  Fin Si
Fait

FIG. 8.21 – Algorithme du compteur de performance

Ce module a par exemple été utilisé pour le client graphique, afin de mesurer le nombre de messages reçus.

8.4 Le *trackerConnector*

Le premier module que nous avons développé est le *trackerConnector* ou Connecteur. Ce nom fait référence aux architectures n-tiers et aux serveurs d'applications. Effectivement, dans ce type de serveur, il est important d'avoir une bonne interopérabilité, autrement dit, de pouvoir interconnecter des applications hétérogènes (systèmes d'exploitation ou modes différents de communication) ce qui se fait par ce que l'on appelle des connecteurs.

Dans notre cas, le *trackerConnector* doit pouvoir dialoguer avec tout tracker, le Cycab ou bien tout autre périphérique d'entrée. C'est le premier maillon de notre architecture (cf. figures 6.2 et 8.22).

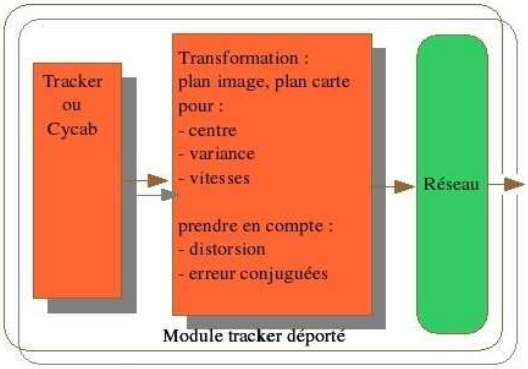

FIG. 8.22 – Le trackerConnector

8.4.1 Spécifications et principes

PrimaBlue, PrimaLab, le Cycab sont autant de capteurs qu'il faut interfacer avec le serveur de cartes.

Dans certains cas, les données seront simplement reformatées en XML, en tenant compte des données telles que décrites dans le tableau 8.7. Dans d'autres cas, PrimaBlue par exemple, il faudra effectuer différents calculs (transformations homographiques, ...). Une fois fait, les données seront envoyées au serveur de cartes via le protocole et le format choisis.

Ce module pourra être lancé plusieurs fois sur une même machine, en fonction des besoins, que ce soit sur la machine où tourne le serveur de cartes ou bien totalement déporté sur un autre système.

Détail de l'implémentation

L'enchaînement des opérations est représenté par l'algorithme global de la figure 8.23.

1. Lecture de la ligne de commande et du fichier de configuration le cas échéant
2. **Si** (demande d'export en entrée) **Alors**
   | Création du fichier
   Fin Si
3. **Si** (demande d'export en sortie) **Alors**
   | Création du fichier
   Fin Si
4. **Si** (demande de fichier de traces) **Alors**
   | Création du fichier
   Fin Si
5. Mise en place interception des signaux
6. Création du serveur eNet
7. **Si** (non simulation) **Alors**
   | **Selon que**
   | | tracker = "BEV" : Lancer la gestion pour PrimaBlue
   | | tracker = "CYCAB" : Lancer la gestion pour le Cycab
   | | autre : Tracker non défini
   | **Fin Selon que**
   Sinon
   | Lancer la simulation
   Fin Si
8. Arrêter le serveur
9. Fermeture de tous les fichiers

Fig. 8.23 – Algorithme global du *trackerConnector*

Comme l'on voit dans cet algorithme, plusieurs variables vont être définies soit par défaut, soit par le biais d'un fichier de configuration (cf. section 8.4.3).

Une variable *tracker* va permettre de savoir si nous devons nous connecter à un logiciel PrimaBlue ou bien à un Cycab, ou tout autre tracker à venir. De même, une variable va permettre de savoir si nous sommes dans un mode simulation ou bien dans un mode « normal ».

8.4.2 Ligne de commande

La ligne de commande pour lancer un *trackerConnector* est la suivante :

trackerConnector [-h] [–ofile-in filename] [–ofile-out filename] [–config config.xml] [–simfile filename] [–simdelta delta_obs]

TAB. 8.17 – Ligne de commande du trackerConnector

Le tableau 8.18 recense les options de la ligne de commande. Les paramètres entre crochets sont optionnels et si aucun n'est passé à la commande *trackerConnector*, alors le fichier config.xml sera lu (s'il n'existe pas, le programme ne se lancera pas !).

	Description
h	Affiche la syntaxe de lancement du *trackerConnector*
config config.xml	Fichier de configuration. Par défaut, config.xml
ofile-in filename	Sauvegarder les données en entrée du *trackerConnector* dans *filename*
ofile-out filename	Idem mais en sortie du *trackerConnector* (post calculs)
simfile filename	Mode simulation et rejeu des données du fichier *filename*
simdelta delta_obs	(Simulation) Temps à attendre avec le rejeu de la première trame.

TAB. 8.18 – Paramètres de la ligne de commande du *trackerConnector*

8.4.3 Fichier de configuration

Le fichier de configuration présenté à la figure 8.24 est un exemple de ce qui est faisable en matière de configuration. Il est à noter que le format est en XML.

```
<?xml version = '1.0' encoding = 'UTF−8'?>
<bevConnector label="tC" >
    <general distort="false" simu="false" simulLoop="on" simulFile="bev1.dump"
        ltracker="traces.txt" debug="0" />
    <mapServer servername="localhost" port="6500" />
    <tracker servername="localhost" port="1500" type="bev" enabled="true" />

    <camera id="0" file="cameraRight1−cm.xml" />
    <camera id="1" file="cameraLeft0−cm.xml" />
    <server port="6600" enabled="true" />
</bevConnector>
```

FIG. 8.24 – Exemple de fichier de configuration *trackerConnector*

Quelques commentaires sur ces valeurs :
- Avec un niveau de traces maximum (debug=3), il y aura une baisse sensible des performances, car le *trackerConnector* sera très verbeux.
- La section mapServer permet de se connecter au serveur de cartes.
- La partie *tracker* est utilisée pour dialoguer avec le tracker ou le Cycab.
- « server » fait référence à la partie serveur eNet du *trackerConnector* à lancer.
- Il faut bien vérifier avant de lancer le *trackerConnector* que le numéro de port de la section « server » soit bien libre au niveau système.
- Les fichiers des caméras doivent exister. Sinon ou si les valeurs sont fausses, le calcul de projection dans le plan sera faux (et du coup tout le reste de la chaîne de traitement).

Plusieurs parties sont détaillées dans le tableau 8.19.

Sections	Paramètres	Descriptions
general	distort	Indique si le calcul de distorsion doit être fait ou non
	simu	Si à « true » le mode « magnétoscope » est effectif
	simuFile	Indique le fichier à rejouer (si simu=« true »
	simuLoop	Si à « true » les données sont rejouées en boucle
	ltracker	Donne le nom éventuel du fichier de trace
	debug	0 = pas de trace, 3 = très verbeux
mapServer	servername	Le nom ou l'adresse IP du serveur de cartes
	port	Le port réseau du serveur de cartes
tracker	servername	Le nom (ou l'IP) de la machine où s'exécute le tracker
	port	Le port réseau du tracker
	type	Indique le type de tracker (BEV, cycab, ...)
	enabled	Si « false » : la ligne non prise en compte[a]
server	port	Le port réseau sur lequel eNet va écouter
	enabled	« false » : pas de serveur eNet ; pas d'application !
camera	id	Le numéro de canal indiqué dans PrimaBlue
	file	Fichier des paramètres intrinsèques/extrinsèques

TAB. 8.19 – Champs du fichier de configuration *trackerConnector*

[a]Il faut une ligne valide, sinon le programme ne se lance pas

Programme de lecture du fichier de configuration

Afin de pouvoir lire ces fichiers, nous avons implémenté les fonctions nécessaires dérivant de la classe *configXMLParser* (définie en 8.3.2).

La figure 8.25 nous montre que cette classe fille comporte de nombreuses méthodes. Nous n'aborderons pas ici le détail complet, sachant que la documentation Doxygen (voir annexe D) inclut tout le détail.

Cependant, nous pouvons extraire de ce schéma 3 types de méthodes liées à :
- la gestion des caméras,
- la gestion de la partie tracker/Cycab,
- la configuration du serveur eNet pour dialoguer avec le serveur de cartes.

FIG. 8.25 – Classe configConnector

8.4.4 La gestion des caméras

Nous avons vu au paragraphe 7.3.1 qu'il est nécessaire de calibrer les caméras pour pouvoir les utiliser correctement. De ce calibrage ressort deux types de paramètres : les données intrinsèques et extrinsèques.

Le *trackerConnector* va avoir besoin de ces informations pour faire les différents calculs, tels que la projection dans le plan du parking (homographie) ou bien la correction de distorsion. Nous utilisons la technique d'OpenCV, la construction d'une matrice incluant les coordonnées après correction, ce pour la dimension de l'image (dans notre cas 384x288). Cette opération sera faite à l'initialisation du module concerné, cette matrice nous fournissant un accès rapide pour la correction de coordonnées.

Nous avons implémenté une classe afin de décrire une caméra, classe représentée dans la figure 8.26.

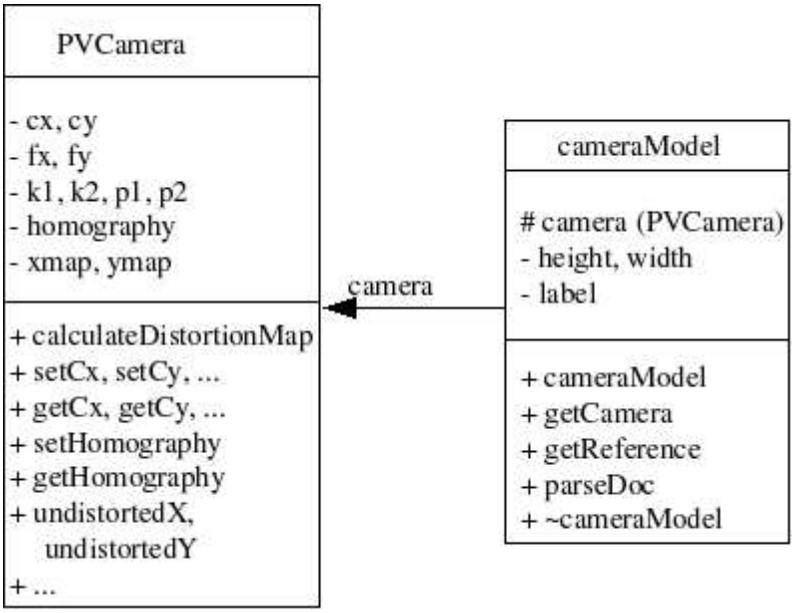

FIG. 8.26 – Définition d'une caméra

Toutes les méthodes et les membres de ces classes ne sont pas représentés dans ce schéma, afin de ne pas alourdir le rapport, seuls les plus importants sont conservés et décrits dans le tableau 8.20. Le type de données homographyMatrix est une matrice de type doubles 3x3.

Méthodes/Membres	Description
double **fx, fy, cx, cy, . . . , p2**	Paramètres intrinsèques de la caméra (cf. 7.2).
int * **xmap, ymap**	Matrice d'entiers pour les coordonnées sans distorsion.
PVCamera( )	Initialise les variables, appelle *calculateDistortionMap*
void **calculateDistortionMap** ( )	Calcul des valeurs pour la correction de la distorsion
void **setCx, . . .** (const double valeur)	Mutateur, modifie les variables privées[Sil98]
double **getCx, getCy, . . .** ( )	Accesseur (retourne la valeur d'une variable privée)
void **setHomography**(homographyMatrix & valeur)	Initialise la matrice d'homographie d'après les données du fichier XML
homographyMatrix ***getHomography**()	Retourne la matrice d'homographie.

TAB. 8.20 – Méthodes et membres de la classe PVCamera

La classe *cameraModel* hérite de *PVCamera* et rajoute les méthodes nécessaires pour pouvoir *parser* le fichier XML contenant tous les paramètres.

La signature du constructeur de cette classe est représentée dans la figure 8.27. Nous voyons que le nom de fichier est passé en paramètre au constructeur. Ce dernier va faire appel à des méthodes similaires à ce que nous avons déjà vu (cf. 8.3.2) : *parseDoc* et *getReference*.

Ces deux méthodes vont mettre à jour l'objet camera de type *PVCamera* (via les mutateurs définis dans *PVCamera*).

```
cameraModel(const char *xmlConfigFile);
```

FIG. 8.27 – Constructeur de la classe cameraModel

8.4.4.1 Point important

Il faut absolument pouvoir faire le lien entre le flux vidéo que nous recevons et la bonne définition des paramètres de la caméra, car si il y a une erreur à ce niveau les calculs à venir type projection dans le plan (homographie) seront totalement faux et toute la chaîne de traitement le sera aussi.

8.4.5 Mode PrimaBlue

Pour poursuivre sur le point précédent, PrimaBlue permet de donner un numéro de canal à un flux vidéo (qu'il soit direct, autrement dit le flux d'une caméra, ou bien pré-enregistré). Ce numéro doit apparaître dans le fichier de configuration XML du *trackerConnector* (voir le tableau 8.19) avec bien évidemment le fichier correct de paramètres.

Le *trackerConnector*, dans son mode PrimaBlue (voir figure 8.23, ligne 7), fonctionne comme indiqué dans la figure 8.28. A noter l'instanciation des objets de type cameraModel vu plus haut, d'après le fichier de configuration.

```
Initialisation du client socket TCP
Instanciation des objets caméra
Tant que ( non connecté au serveur de cartes et pas d'arrêt demandé ) faire
    | Tentative de connexion au serveur
Fait
Si ( pas d'arrêt demandé ) Alors
    | Tant que ( connexion et pas d'arrêt demandé ) faire
    |     | Lecture de la socket TCP
    |     | Si ( réception données ) Alors
    |     |     | Transformation des données ( distorsion, homographie, ... )
    |     |     | Formatage en XML ( cf. tableau 8.7 )
    |     |     | Envoi au serveur de cartes
    |     | Fin Si
    | Fait
Fin Si
```

FIG. 8.28 – Algorithme du mode Primablue du *trackerConnector*

Ces objets seront stockés dans une structure de type *map* (figure 8.29).

```
map< int , cameraModel * >          cameraList;
```

FIG. 8.29 – Stockage de la liste des objets cameraModel

L'avantage de cette structure est qu'il suffit de récupérer le numéro de canal des données de PrimaBlue (données « canal » dans le tableau 4.3) pour avoir un accès immédiat aux paramètres de la caméra, le champ *int* représentant ce numéro de canal.

8.4.5.1 Transformation des données

Après avoir reçu une trame binaire de PrimaBlue, il faut dans un premier temps découper cette trame pour pouvoir collecter information par information.

Dans un deuxième temps, il faut appliquer les transformations dont nous avons déjà parlées (cf. 8.4.4 et 7.3).

Comme indiqué dans le paragraphe précédent, la trame contient le numéro de canal, il suffit alors de récupérer les données de la caméra et d'appliquer les transformations adéquates. La plus importante est la projection dans le plan.

Pour rappel, nous récupérons un couple de coordonnées (x_img, y_img) dans le repère image, ainsi que la covariance qui nous donne une indication de taille. Nous avons pris le parti de faire une première projection des coordonnées obtenues sur le sol.

La raison en est que PrimaBlue nous retourne le centre de l'ellipse de détection (ellipse verte dans la figure 4.10). Dans le cas d'un piéton ceci correspond dans la majorité des cas au centre de gravité et projeter directement ce point sur le plan du parking génère une erreur, liée à la distance de la cible à la caméra.

Pour récupérer les coordonnées au sol dans le plan image, nous effectuons un premier calcul de correction verticale et horizontale :

$$correctionVerticale = \sqrt{cov_{\ en\ y}}\ ;\ correctionHorizontale = \sqrt{cov_{\ en\ x}}$$

Puis nous calculons le nouvel y :

$$y\_img = y\_img + correctionVerticale$$

D'autre part, nous devons aussi « projeter la covariance » dans le plan du parking, afin d'obtenir une estimation de la taille de l'objet dans ce plan. Pour ce faire, nous prenons deux autres points (x_bis, y_bis) et (x_ter, y_ter) définis comme suit :

$$\begin{aligned} x\_bis &= x\_img \\ y\_bis &= y\_img + 2 * correctionVerticale \\ x\_ter &= x\_img + correctionHorizontale \\ y\_ter &= y\_img + correctionVerticale \end{aligned}$$

De là, nous pouvons appliquer la transformation homographique sur les 3 points (x_img, y_img), (x_bis, y_bis) et (x_ter, y_ter). La figure 8.30 représente les calculs pour le point principal. Les calculs pour les autres points sont identiques aux variables près.

$$\begin{pmatrix} x\_plan \\ y\_plan \\ z\_plan \end{pmatrix} = \begin{pmatrix} homography[0,0] & homography[0,1] & homography[0,2] \\ homography[1,0] & homography[1,1] & homography[1,2] \\ homography[2,0] & homography[2,1] & homography[2,2] \end{pmatrix} \times \begin{pmatrix} x\_img \\ y\_img \\ 1 \end{pmatrix}$$

FIG. 8.30 – Transformation homographique du point observé

Homography est la matrice d'homographie (matrice 3x3) pour la caméra en cours. Une fois les 3 points projetés sur le plan du parking, nous calculons la nouvelle covariance.

$$\begin{pmatrix} cov_{en\ x} \\ cov_{en\ y} \end{pmatrix} = \begin{pmatrix} (x\_bis\_plan - x\_plan)^2 + (y\_bis\_plan - y\_plan)^2 \\ (x\_ter\_plan - x\_plan)^2 + (y\_ter\_plan - y\_plan)^2 \end{pmatrix}$$

FIG. 8.31 – Transformation de la covariance

8.4.6 Mode Cycab

Dans ce mode (figure 8.23, ligne 7), le *trackerConnector* fonctionne de manière similaire à celle décrite pour PrimaBlue (figure 8.28). Afin de se connecter au Cycab, nous avons utilisé les librairies développées par Cédric Pradalier lors de sa thèse et son DEA (respectivement [Pra04], [Pra01]). Elles nous apportent différentes fonctions permettant d'interroger le Cycab (exemple du code de récupération dans la figure 8.32).

```
// Waiting for Cycab's update
cycab->waitUpdate();

// Get the current Cycab's state
cycab->getState(&rec,&dsl,&dsr,&lphi,&ltime);

// We define the measure's timestamp
gettimeofday(&tv,NULL);
cycab_temps->tv_sec = tv.tv_sec;
cycab_temps->tv_usec = tv.tv_usec;

if (cycab->getPos(&x,&y,&theta)) {
    assert( NULL != px ); *px = x;
    assert( NULL != py ); *py = y;
    assert( NULL != otheta ); *otheta = theta;
    assert( NULL != vtrans ); *vtrans = (dsl + dsr)/2.0;
    assert( NULL != phi ); *phi = lphi;
}
```

FIG. 8.32 – Code pour l'interrogation du Cycab

La différence principale dans ce mode porte sur la transformation des données, qui se résume à rajouter l'heure d'acquisition et à formater les informations suivant notre protocole.

8.4.7 la partie serveur

Quelque soit le mode, nous devons lancer un serveur eNet qui communiquera avec le serveur de cartes. Nous avons implémenté un objet *bcServer*, tel que décrit dans la figure 8.33, qui hérite de l'objet *PVBaseServer* (cf. 8.3.3.4).

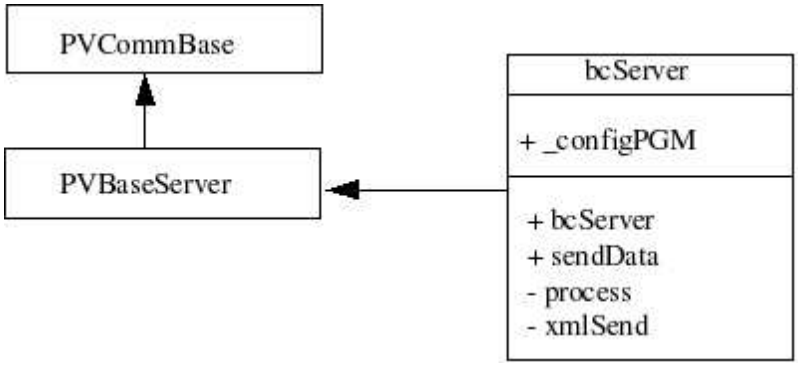

FIG. 8.33 – La classe du serveur eNet du *trackerConnector*

Description des méthodes de *bcServer* Le tableau 8.21 liste les différentes fonctions implémentées dans cette classe. Entre autres, la méthode *process* gère les messages et évènements en provenance du serveur de cartes, comme par exemple, une connexion réussie, ou bien un arrêt du serveur de cartes.

Méthodes	Description
void sendData (commData bevData, int type_tracker)	Ajoute quelques champs nécessaires à la chaîne de traitement et appelle xmlSend.
void xmlSend (commData data)	Formate les données en XML (cf. 8.5).
bool process (ENetPacket *packet, int sender)	Gestion des messages en provenance du serveur de cartes.

TAB. 8.21 – Méthodes et membres de la classe bcServer

8.4.8 Mode Simulateur

Le simulateur, présenté en 8.18, a été implémenté dans le *trackerConnector*. Nous avons deux paramètres sur la ligne de commande permettant de donner le delta de démarrage du rejeu ($\triangle Obs$) et le nom du fichier à rejouer.

8.4.9 Résumé

Le *trackerConnector* est l'un des premiers éléments de notre chaîne de traitement. Il a pour rôle principal de rendre un logiciel tiers compatible avec notre architecture. De part sa nature même, il doit être évolutif, afin de pouvoir interconnecter tout nouveau tracker, logiciel voire périphérique en entrée. Pour ce faire, nous avons mis en place des mécanismes de communications simples, comme par exemple des sockets TCP/IP, qui facilitent les échanges avec l'extérieur. Sa modularité permet aussi de créer très rapidement la partie du dialogue avec un nouveau tracker.

Il doit aussi être performant et rapide, ce qui veut dire que les transformations sur les données doivent être le plus efficace possible. La mise au point avec les outils comme Valgrind (cf. section 8.7.1) ont permis d'optimiser le code et surtout l'utilisation de la mémoire.

8.5 Le *mapServer*

Le *mapServer* est le cœur de notre architecture. Il a pour fonction de « récolter » les données en provenance des *trackerConnector*.

8.5.1 Spécifications

Effectivement comme le montrent les figures 6.2 et 8.34, le serveur de cartes est central : il reçoit les données des trackerConnector et envoie aux clients. Le mapServer gèrera les données avec le format *commData* défini dans la présentation de la nouvelle architecture (section 8.2.3).

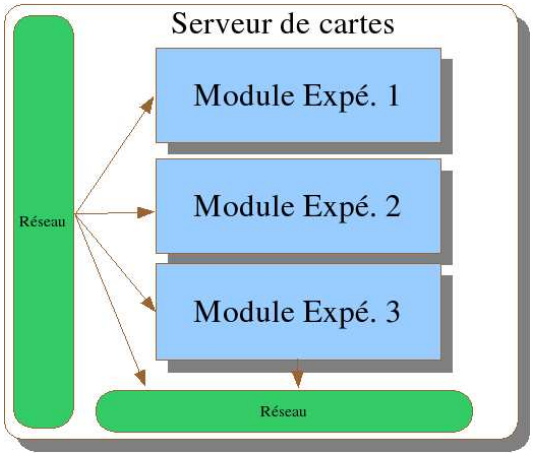

Fig. 8.34 – Le mapServer

Ce module devra permettre de tester et d'implémenter des modules que ce soit pour l'association, la fusion ou tout autre traitement sur les données. Un mode simulateur pourra être implémenté en partant de la classe *PVSimulBase* (cf. 8.3.4).

8.5.2 Principe

L'enchaînement des opérations est représenté par l'algorithme de la figure 8.35.

1. Lecture de la ligne de commande et du fichier de configuration le cas échéant
2. **Si** (demande d'export en entrée) **Alors**
   | Création du fichier
   Fin Si
3. **Si** (demande d'export en sortie) **Alors**
   | Création du fichier
   Fin Si
4. **Si** (demande de fichier de traces) **Alors**
   | Création du fichier
   Fin Si
5. **Si** (configuration à distance) **Alors**
   | Thread de gestion de la configuration à distance
   Fin Si
6. Mise en place interception des signaux
7. Création du serveur eNet (thread)
8. **Tant que** (fin non demandée) **faire**
   | 9. Création du serveur eNet pour les trackerConnector
   | 10. Attente connexion d'un trackerConnector
   | 11. Lancement d'un thread pour gérer le trackerConnector
   Fait
12. Arrêter les threads
13. Fermeture de tous les fichiers

FIG. 8.35 – Algorithme global du mapServer

Les interceptions de signaux permettent de « nettoyer » les fichiers, la mémoire, ... lorsque l'utilisateur sort du serveur via un *control+c*.

8.5.3 Ligne de commande

La ligne de commande pour lancer un mapServer est la suivante :

mapServer [-h] [–outfile-out outputfilename] [–outfile-in filename] [–config config.xml]

TAB. 8.22 – Ligne de commande du mapServer

Le détail des options est listé dans le tableau 8.23.

	Description
h	Affiche la syntaxe de lancement du mapServer
config config.xml	Un fichier de configuration au format XML doit être utilisé. Si ce paramètre n'est pas utilisé, le programme prendra un fichier config.xml
outfile-in filename	Indique qu'il faut sauvegarder les données en entrée du mapServer dans le fichier filename
outfile-out filename	Idem mais en sortie du serveur de cartes (post calculs)

TAB. 8.23 – Options de la ligne de commande du mapServer

Les paramètres indiqués entre crochets sont optionnels. Si aucun fichier de configuration n'est passé à la commande mapServer, alors le fichier *config.xml* sera lu (ce qui implique qu'il existe !).

8.5.4 Fichier de configuration

Tout comme le programme précédent (cf. section 8.4.3), les options d'exécution seront définies dans un fichier de configuration au format XML (figure 8.36).

```
<?xml version = '1.0' encoding = 'UTF−8'?>
<mapServer label="mapServer" >
   <general simu="false" log="mapServer.log" asso="true" bof="false" debug="0"
   staticMapFileName="ParkingPlan.xml" staticTargetsGeometry="targets.xml" />

   <remoteConfig port="6501" enabled="true" />
   <serverConnector port="6500" enabled="true" />

   <server port="7700" enabled="true" />
</mapServer>
```

FIG. 8.36 – Exemple de fichier de configuration mapServer

Le tableau 8.24 liste les paramètres que nous n'avons pas encore rencontrés.

Sections	Paramètres	Descriptions
general	log	Donne le nom éventuel du fichier de trace
	asso	Si valeur « true » le module jPDA sera actif.
	bof	Si valeur « true » le module BOF sera actif.
	staticMapFileName	Indique le nom du fichier contenant le plan du parking.
	staticTargetsGeometry	Fichier contenant les descriptions géométriques des objets (cf. annexe A).
remoteConfig	-	Description pour la configuration à distance.
serverConnector	-	Serveur eNet en attente de connexion de trackerConnector.
server	-	Partie serveur pour les clients.

TAB. 8.24 – Champs du fichier de configuration trackerConnector

Nous pouvons noter dans cette configuration deux paramètres permettant d'activer ou non les modules (jPDA, BOF, ...). Si aucun de ces modules est activé (asso = « false », bof = « false »), le serveur de cartes enverra les données brutes sans transformation.

Nous avons aussi deux paramètres permettant d'indiquer les fichiers de description du plan et des objets géométriques. Le serveur de cartes enverra à chaque client qui se connecte ces informations (la description des fichiers est disponible dans l'annexe A).

8.5.5 Gestion des observations

La partie "réception" du serveur de cartes, autrement dit le *thread* lancé à la connexion d'un trackerConnector, crée un objet héritant de la classe *PVCommClient* (cf. section 8.3.3.3). Cet objet gère trois types de messages :

- CONNECTOR_HELLO,
- CONNECTOR_BYE,
- TC_OBS.

Les deux premiers messages concernent la connexion/déconnexion d'un trackerConnector. Le dernier est celui qui nous intéresse ici : la réception des observations. Chaque trackerConnector va transmettre les observations au fil de l'eau à son *thread* dédié, dans le format XML que nous avons vu précédemment, encapsulé dans notre protocole. Chaque observation XML reçue est stockée dans une structure *messageList* (voir la définition section 8.3.3.2).

Cette structure unique est partagée par tous les processus de réception, ce qui pose le problème des accès multiples. Afin de gérer ceci, nous avons utilisé un système de verrou ou *mutex* classique en programmation multi-threadée[19].

8.5.6 La structure mapServerBase

Dès qu'un trackerConnector a mis à jour la structure *messageList*, en l'occurrence une liste d'observations, l'objet carte est lui aussi actualisé. Cet objet est une instance de la classe *mapServerBase* représentée dans la figure 8.37. Ce diagramme représente la classe de base ainsi que la classe fille *mapServerJPDA* utilisée plus particulièrement pour le module expérimental JPDA (voir la section suivante).

[19] http ://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html. Dernière consultation : 20-mai-05.

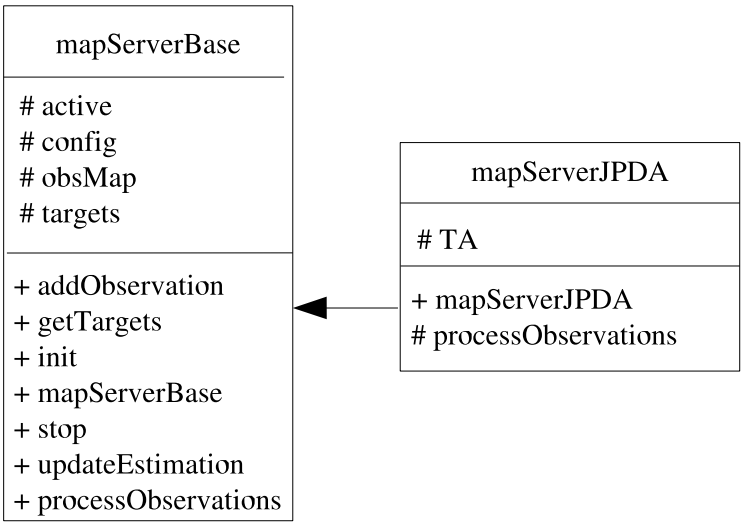

FIG. 8.37 – Classe MapServer

Le tableau 8.25 présente les membres et méthodes de la classe de base.

Méthodes/Membres	Description
active	Est-ce que cet objet est actif ou non ?
config	Pointeur vers l'objet de configuration du mapServer (cf. section 8.3.2).
obsMap	Structure C++ de type *map* pour stocker les observations en cours.
targets	Liste de cibles après traitement des observations.
mapServerBase( )	Constructeur. Initialise le mutex pour la gestion de l'exclusion mutuelle des *threads*.
void **init** (configMapServer *)	Initialise la configuration, déclare l'objet actif et nettoie la liste des observations.
void **stop**( )	Désactive l'objet et purge la liste des observations.
void **addObservation** (commdata &)	Stocke l'observation que nous venons de recevoir dans la structure obsMap.
targetList& **getTargets**( )	Accesseur retournant la liste en cours des cibles calculées.
void **updateEstimation**( )	Prépare les structures contenant les observations avant d'appeler processObservations.
virtual void **processObservations**(obsMap)=0	Fonction virtuelle traitant les observations afin d'obtenir des cibles (voir section suivante).
std : :ostream& operator< < (std : :ostream&, const targetList&)	Redéfinition de l'opérateur de flux sortant afin de générer les trames XML depuis la liste de cibles.

TAB. 8.25 – Méthodes et membres de la classe mapServerBase

Comme nous le voyons, il y a une méthode virtuelle pure (*processObservations*), impliquant qu'il nous faut absolument définir une classe héritant de celle de base et redéfinissant la méthode en question.

Le processus de traitement des observations fonctionne de la façon suivante :

1. Le mutex/verrou est posé, afin d'interdire la mise à jour suite à réception d'observations
2. La liste des observations non encore traitées (obsMap) est dupliquée
3. obsMap est purgée
4. Le mutex/verrou est libéré, le serveur stocke de nouveau les observations venant des trackerConnector
5. La méthode virtuelle processObservations est appelée et effectue son traitement (en fonction de l'implémentation choisie)

8.5.7 Les modules expérimentaux

L'un des objectifs du *mapServer* est de permettre de tester des modules expérimentaux. Normalement, il s'agit de développer au moins une classe héritant de la classe de base vue dans le paragraphe précédent. Cette nouvelle classe devant implémenter la méthode virtuelle *processObservations*. Le premier module que nous avons mis en place est le jPDA, qui signifie Joint Probabilistic Data Association ou association de données à probabilités jointes.

Le second module, en cours d'implémentation, est le BOF ou Bayesian Occupancy Filter, autrement dit, un filtre à grille d'occupation bayésienne. Il ne sera pas détaillé dans ce rapport car sortant du but de Parkview et ParkNav. Cependant, il est intéressant de noter la possibilité d'intégrer des modules non rattachés au projet initiateur de la plate-forme.

Le jPDA

Comme l'indique son nom, l'objectif du jPDA est de faire de l'association pour les données reçues, les observations.

La figure 8.38 représente la manière dont 2 cibles vont être vues dans une zone de recouvrement. Ceci est le propre d'une perception multi-caméras, avec zone de recouvrement : une même cible peut avoir plus d'une observation dans la chaîne de traitement, chacune étant entachée d'une erreur (détection, transformation homographique, . . .).

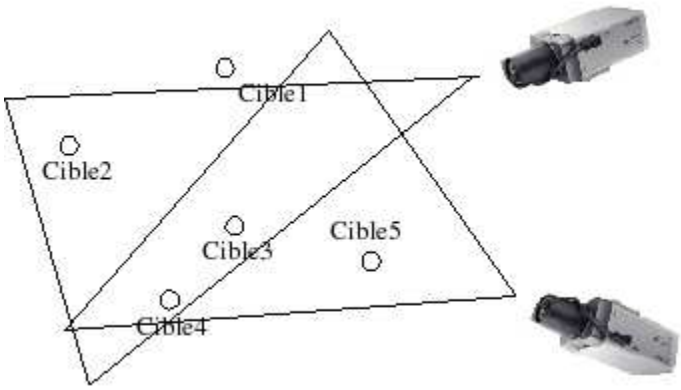

FIG. 8.38 – Perception avec recouvrement en multi-capteurs

Le jPDA a pour rôle de récupérer toutes les observations reçues entre deux traitements et d'en « déduire » les cibles correspondantes.

La méthode d'entrée de ce module doit être *processObservations*, qui prend en paramètre la liste de toutes les observations non encore traitées, et en sortie cette méthode aura rempli la structure *targets*.

Le principe du jPDA est le suivant [DWDA03] :

1. A un instant k, nous avons T cibles dont l'état X^k est connu. Soit $X^k = \{x_1^k, \ldots, x_T^k\}$ l'état des cibles à l'instant k.
2. Au même instant k, nous recevons m_k observations, soit : $Z^k = \{x_1(k), \ldots, x_{m_k}(k)\}$.
3. Prédiction du prochain état d'une cible en calculant (filtrage Bayésien) $p(x_i^k|Z^{k-1})$
4. A l'arrivée de nouvelles observations, l'état est actualisé en utilisant $p(x_i^k|Z^k)$.

Ce module conserve à tout instant la configuration des cibles calculées dans des structures internes.

Il est à noter que le paramètrage du module jPDA met en œuvre plusieurs variables/valeurs qui rendent ce programme délicat à configurer correctement (voir la documentation du module - non incluse dans ce rapport).

8.5.8 Configuration à distance

Comme indiqué précédemment, le *trackerConnector* et le *mapServer* sont configurables lors du lancement, via un fichier de configuration. Cependant, dans le cas du *mapServer*, il peut être intéressant de modifier son comportement en cours de fonctionnement (ou « online »). Pour ce faire, nous avons implémenté différentes méthodes dans la classe configXMLParser, comme indiqué dans la section 8.3.2.2.

Du côté du serveur de cartes, nous avons développé un thread (ou processus léger) lancé au démarrage du *mapServer*, si l'utilisateur l'a demandé (paramètre remoteConfig.enabled à « true »). Ce thread crée la socket TCP, en fonction des paramètres du fichier de configuration, puis attend les connexions entrantes.

Nous avons développé un client expérimental en Python (cf. la figure 8.39), nous permettant :

- d'activer ou non l'un des modules,
- d'arrêter le serveur de cartes,
- de changer le niveau de traces (paramètre « debug »).

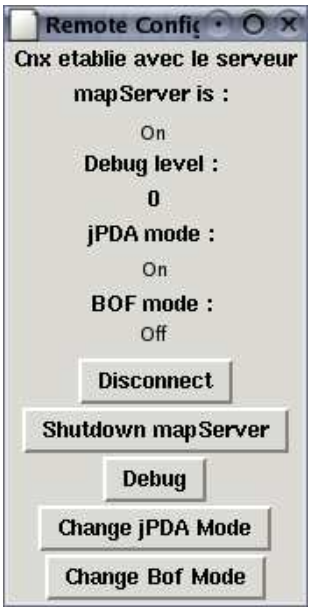

FIG. 8.39 – Client pour la configuration à distance

8.5.9 Serveur pour les trackerConnector

La gestion d'un *trackerConnector* est un peu particulière : un serveur eNet attend les connexions entrantes d'un trackerConnector, pour pouvoir ensuite lancer un client eNet. La figure 8.40 représente les échanges principaux entre un trackerConnector et le mapServer.

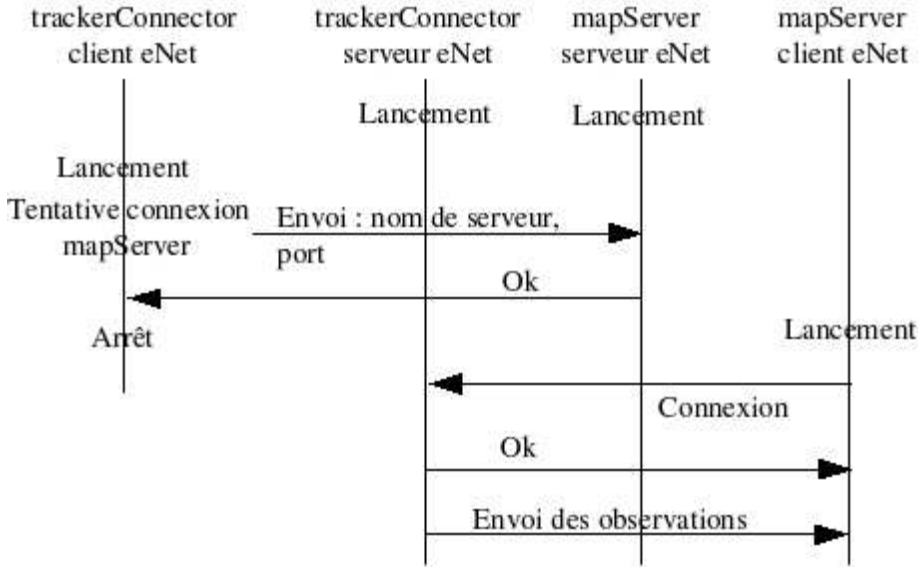

FIG. 8.40 – Connexion entre le trackerConnector et le mapServer

Le serveur eNet tourne en permanence dans l'attente d'un nouveau trackerConnector, il est basé sur la classe *eNetServer* décrite dans la prochaine section. A chaque connexion, le mapServer lancera un nouveau thread ou processus léger, qui prendra en charge le dialogue avec ce nouveau fournisseur de données.

8.5.10 Serveur pour les clients

Nous allons présenter ici la partie serveur du *mapServer* ainsi que le mode de fonctionnement avec les clients. Il s'agit bien entendu d'un serveur eNet, géré grâce à une classe qui hérite de *PVBaseServer* (cf. section 8.3.3.4) : la classe *eNetServer* décrite dans la figure 8.41.

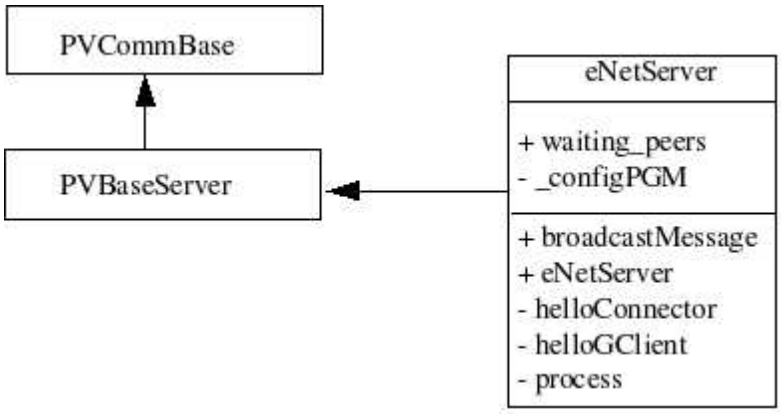

FIG. 8.41 – Classe eNetServer

Connexion d'un client

Lorsqu'un client se connecte au serveur, la méthode *helloGClient* est appelée. Elle va avoir pour rôle de :

- confirmer la bonne connexion (Envoi d'un message MS_CONN_OK),
- envoyer la carte statique du parking (le fichier décrit dans l'annexe A),
- envoyer la description des objets (décrit dans la même annexe A).

Seulement après la réception de ces éléments, le client pourra commencer son travail.

Transmission des données

Les données, en l'occurrence ici les cibles (sauf si le module jPDA est désactivé, auquel cas nous parlons d'observations), sont envoyées suivant le format XML que nous avons établi pour toute la chaîne de traitement (cf. figure 8.5).

Nous avons décidé de faire un envoi à tous les clients en même temps, un mode *broadcast*. C'est la fonction de la méthode *broadcastMessage* ; ainsi, tous les clients graphiques recevront la même carte en temps réel. Si les modules BOF et jPDA sont activés, les clients recevront les deux jeux de données en même temps. Il faut cependant noter que tout est en place pour pouvoir faire de l'envoi différencié client par client, permettant ainsi d'avoir des clients de types différents.

Le Cycab par exemple en tant que client sera surtout demandeur de la carte, autrement dit, de la sortie du module jPDA. La géométrie des objets n'est pas importante dans ce cas.

8.5.11 Résumé

Le *mapServer* est l'élément central de notre architecture, celui qui effectue la fusion des informations provenant des *trackerConnector*. Il a pour rôle principal de modéliser l'environnement avec ses différents éléments. Il permet aussi de tester des modules expérimentaux, comme le module jPDA que nous avons présenté.

Tout comme le *trackerConnector*, la mise au point avec Valgrind fut un passage important afin d'optimiser le code. L'ajout d'une configuration à distance dynamiquement est un plus, qui nous permet de modifier le comportement du *mapServer* sans devoir l'arrêter.

8.6 Le client graphique

Il nous a fallu très rapidement développer un moyen d'afficher le résultat de la chaîne de traitement, c'est là qu'intervient le client graphique.

8.6.1 Spécifications et principes

Le client graphique doit se connecter au *mapServer* aussi bien en local (sur la même machine que le serveur) que de n'importe quelle machine. Il a pour premier rôle de recevoir les données envoyées par ce dernier : la carte dynamique de l'environnement.

Il doit afficher cette carte, a minima, dans un environnement 2D, en incluant les éléments fournis par le serveur de cartes : plan statique du parking, cibles, voire - si disponibles - les éléments semi-statiques. La partie traitement graphique doit être performante afin de respecter l'aspect temps réel de la plate-forme. Par ailleurs, il faut intégrer la possibilité d'un affichage type 3D.

Comme nous l'avons vu dans le paragraphe présentant la nouvelle architecture et les choix technologiques, section 8.2.2.2, nous utilisons l'API graphique OpenGL sous la forme de la librairie Mesa, accompagnée de GLUT pour la gestion simplifiée des fenêtres, évènements, etc.

Algorithmes généraux

L'enchaînement des opérations est représenté par l'algorithme de la figure 8.42.

1. Lecture de la ligne de commande et du fichier de configuration le cas échéant
2. **Si** (demande d'export en entrée (simulateur)) **Alors**
   | Création du fichier
   Fin Si
3. **Si** (demande de fichier de traces) **Alors**
   | Création du fichier
   Fin Si
4. Mise en place interception des signaux
5. Préparation de l'environnement graphique
6. Création du client eNet
7. **Si** (Connexion au mapServer) **Alors**
   | Lancement du thread de gestion graphique
   Fin Si
8. Arrêter le client
9. Fermeture de tous les fichiers

Fig. 8.42 – Algorithme global du client graphique

La partie gestion graphique est bien entendue le cœur de ce programme. Nous la détaillons dans l'algorithme figure 8.43.

Tant que (Nous ne recevons pas de demande d'arrêt) **faire**
  Vérification évènement eNet
  Si (Réception message MS_TARGET) **Alors**
    Verrouillage du stockage des cibles
    Parsing du message
    Déverrouillage du stockage
  Fin Si
Fait

FIG. 8.43 – Algorithme réception des cibles

Le parsing du message s'appuie sur des méthodes similaires à ce que nous avons déjà rencontrées : *parseDoc*, *getReference*. Elles vont remplir un objet de type *targetList*, à savoir une pile d'enregistrements *commData* (le format défini sur toute l'architecture).

Chaque message que nous recevons du serveur de cartes contient toutes les cibles en cours dans l'environnement. L'algorithme 8.44 présente le principe du traitement de la pile des cibles.

[Cette boucle agit tant qu'il y a des cibles à traiter]
Tant que (Cibles dans targetList) **faire**
  Si (La cible n'est pas le Cycab) **Alors**
    Affiche un « ovni »
  Sinon
    Dessine le Cycab
  Fin Si
Fait

FIG. 8.44 – Algorithme affichage des cibles

Détail de l'implémentation

La partie communication eNet est très similaire à ce que nous avons eu précédemment, si ce n'est que le client va recevoir plus d'informations suite à la connexion (voir le tableau 8.12) : plan statique du parking (message *STATIC_MAP*), données géométriques des objets (message *STATIC_GEOMETRY*).

Par la suite, il recevra soit les cibles comme résultantes du module jPDA ou d'un envoi des données brutes (message *MS_TARGET*), soit des données du module BOF[Cou03] - en cours d'implémentation lors du stage - (message *MS_BOF_DATA*). Cette gestion du client est réalisée par une classe dérivée de *PVCommClient* (section 8.3.3.3) que nous ne détaillerons pas ici, car identique à ce que nous avons déjà vu.

La vrai difficulté de ce programme est la gestion graphique, plus exactement, la représentation du monde en 2D avec ces différents éléments : plan statique du parking, objets en mouvement, Cycab, ...

8.6.2 Ligne de commande

La ligne de commande pour lancer un client graphique est représentée figure 8.26.

gclientMapServer [-h] [–config config.xml] [–dumpfile dump.file]

TAB. 8.26 – Ligne de commande du gclientMapServer

Le tableau 8.27 recense les différentes options.

	Description
–h	Affiche la syntaxe de lancement du mapServer.
–config config.xml	Un fichier de configuration au format XML doit être utilisé. Si ce paramètre n'est pas utilisé, le programme prendra un fichier config.xml.
–dumpfile dump.file	Indique qu'il faut sauvegarder les données en entrée du client dans le fichier dump.file.

TAB. 8.27 – Options du client graphique

Les paramètres entre crochets sont optionnels, le fichier de configuration par défaut étant config.xml (qui doit exister dans ce cas). Il est possible d'exporter les données telles qu'elles arrivent sur le client graphique, ceci afin de valider la bonne réception ou le bon déroulement des traitements en amont, ou pour pouvoir rejouer en mode simulation des données déjà reçues.

8.6.3 Fichier de configuration

Comme les autres programmes de la chaîne de traitement, celui-ci utilise un fichier de configuration pour définir un ensemble de paramètres. Il est très similaire à ce que nous avons vu pour le *mapServer* ou le *trackerConnector*.

```
<?xml version = '1.0' encoding = 'UTF−8'?>
<gclientMapServer label="gclientMapServer" >
   <general wtitle="mapServer Graphical Client" width="640" height="480"
       log="gclientMapServer_blanc.log" background_color="#FFFFFF" debug="0" />
   <webexport export="off" path="/home/wwwemotion/pub/parkview/images" />
   <server servername="blanc" port="7700" enabled="true" />
</gclientMapServer>
```

FIG. 8.45 – Exemple de fichier de configuration gclientMapServer

Le tableau 8.28 liste les paramètres que nous n'avons pas encore rencontrés.

Sections	Paramètres	Descriptions
general	wtitle	Le nom de la fenêtre graphique.
	width, height	Taille par défaut de la fenêtre.
	background_color	Couleur du fond.
webexport	export	Si valeur « on », il y aura une copie d'écran en format BMP (bitmap).
	path	Chemin où stocker l'image bitmap.
server	-	Informations pour se connecter au serveur.

TAB. 8.28 – Champs du fichier de configuration gclientMapServer

Nous pouvons noter dans cette configuration différents paramètres pour la configuration de la fenêtre graphique. D'autre part, nous avons aussi une section *webexport* qui sera décrite un peu plus loin dans ce chapitre (section 8.6.5).

Les phases d'échange avec le *mapServer* sont listées dans la figure 8.46.

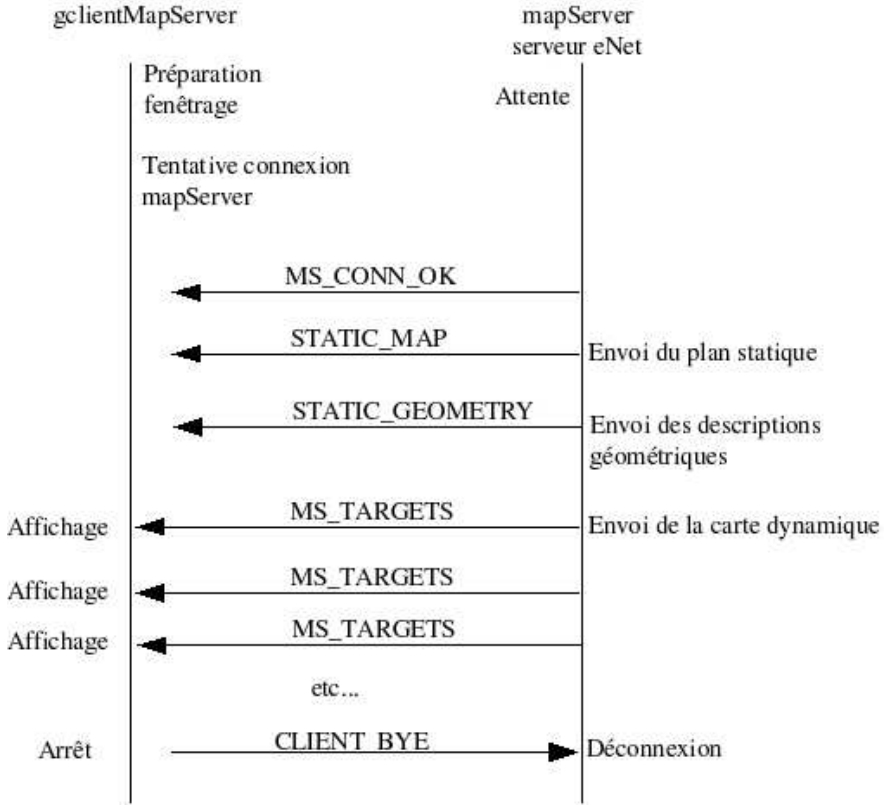

FIG. 8.46 – Phase d'échanges client graphique - serveur de cartes

8.6.4 La représentation du monde

Véritable difficulté de ce client graphique, car il n'est pas évident de devoir représenter un monde, même en 2D, avec des objets mobiles et de natures différentes. Il existe des moteurs graphiques tels que ceux des jeux vidéos intégrant tout ce qu'il faut pour gérer des mondes virtuels, mais ils sont en général orientés 3D, et apportent une certaine lourdeur dans la mise en œuvre, dans le code et dans le poids final de l'exécutable. Voici quelques moteurs libres (liste non exhaustive) : Irrlicht[20], Crystal Space 3D [21], OGRE[22], ...

Nous avons pris le parti de développer une couche graphique « légère » en utilisant les primitives d'OpenGL et les outils fournis avec.

Méthodes d'affichage

Les commandes de dessin d'OpenGL, comme vues à la section 8.2.2.2 , se résument à la création de primitives géométriques [WNDS03] (points, lignes et polygones) et la librairie ne fournit pas de mécanisme d'interactivité (ouverture de fenêtre, gestionnaire d'événements clavier, souris, ...).

GLUT quant à lui, apporte non seulement la gestion de l'interface hommme-machine, mais aussi des routines de dessin complexe (sphère, tore, théière). Cette librairie fonctionne avec le principe de *callback* d'événements.

Autrement dit, le point central de GLUT est une boucle de traitement sans fin (méthode *glutMainLoop*), interrompue uniquement sur réception d'un événement. Dans ce cas, suivant la nature de l'événement (par exemple la fermeture de la fenêtre), la fonction *callback* définie est appelée, pour ensuite revenir dans la boucle principale (sauf si l'événement est une demande de sortie du programme).

Afin de correspondre à notre besoin, nous avons implémenté des classes encapsulant les primitives GLUT et OpenGL. Nous allons maintenant aborder cette surcouche graphique. Le schéma 8.47 représente les deux classes *PV_Window* et *PV_GlutWindow* ainsi que leur héritage.

[20] http ://irrlicht.sourceforge.net/. Dernière consultation : 20-juil-05.
[21] http ://www.crystalspace3d.org/. Dernière consultation : 20-juil-05.
[22] http ://www.ogre3d.org/. Dernière consultation : 20-juil-05.

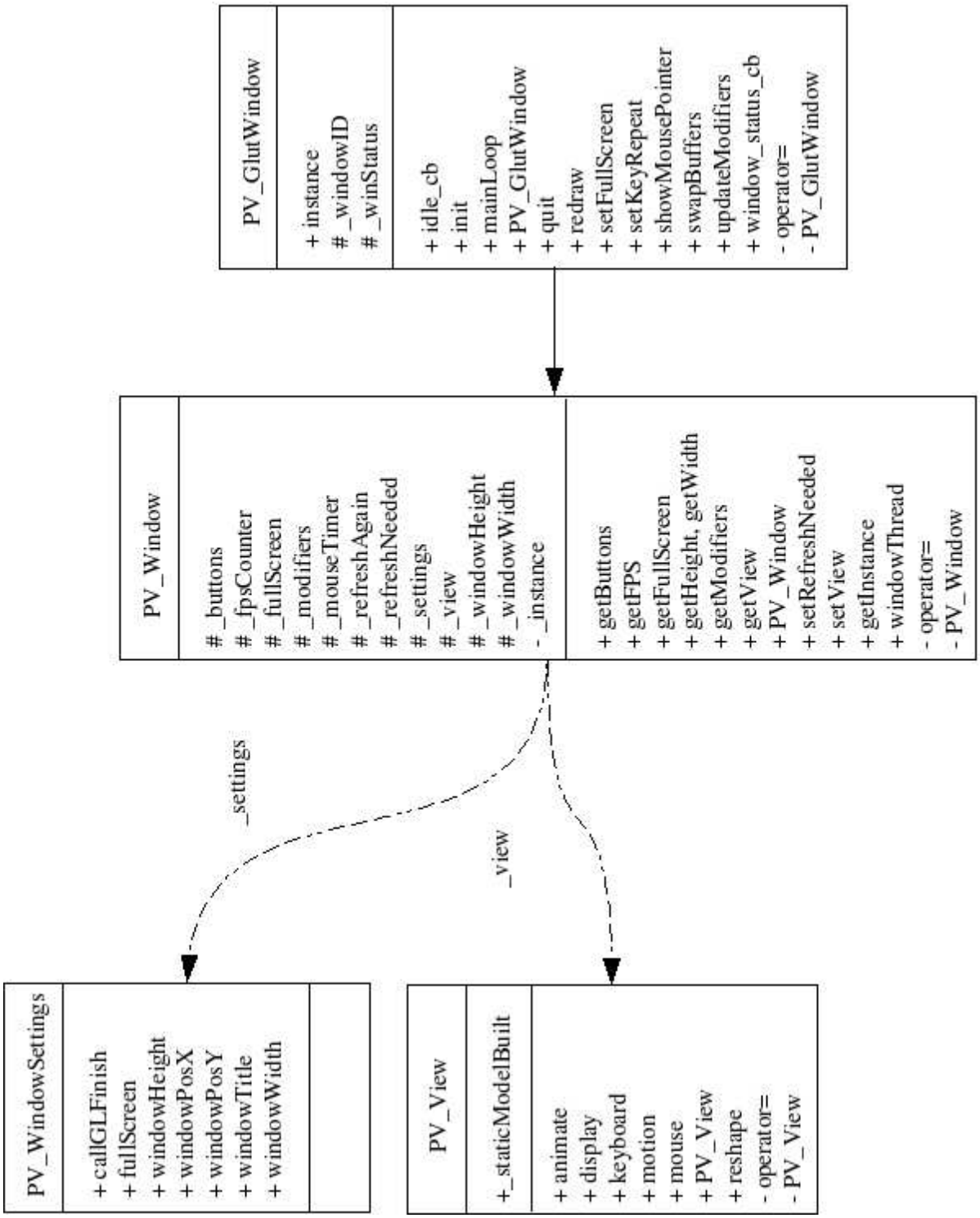

FIG. 8.47 – Classes encapsulant OpenGL et GLUT

L'objet *PV_WindowSettings* permet de définir les paramètres de la fenêtre, voir tableau 8.29.

Membres	Description
callGLFinish	Booléen indiquant une demande d'arrêt du programme.
fullScreen	Booléen pour basculer en mode plein écran (non utilisé pour l'instant).
windowHeight, windowWidth	Dimension de la fenêtre.
windowPosX, windowPoxY	Coordonnées du coin supérieur gauche.
windowTitle	Titre de la fenêtre.

TAB. 8.29 – Classe PV_WindowSettings

La classe *PV_GlutWindow* encapsule les appels aux primitives GLUT en rajoutant nos propres gestionnaires d'événements (*callback*). Pour chaque *callback*, il y a appel des méthodes GLUT et au final appel à une primitive implémentée dans *PV_Window*.

La figure 8.48 reprend pour exemple la définition de la méthode *mainLoop*, qui est la boucle principale.

```
void PV_GlutWindow::mainLoop( void ) {
        // Open window (this creates an OpenGL context)
        // GLUT_DEPTH : Window with a depth buffer
        // GLUT_RGB: an RGBA mode window; GLUT_DOUBLE: Double buffered
        // GLUT_ALPHA : an alpha component to the color buffer
        glutInitDisplayMode( GLUT_DEPTH | GLUT_RGB |
                GLUT_DOUBLE | GLUT_ALPHA );

        // Sets the window size and position
        glutInitWindowSize( _settings->windowWidth, _settings->windowHeight );
        glutInitWindowPosition( _settings->windowPosX, _settings->windowPosY );
        // Create the window with its title
        _windowID = glutCreateWindow( _settings->windowTitle.c_str() );

        // Don't register idle and display callbacks now, wait for the 1st reshape
        // event. See first_reshape_callback ().
        glutReshapeFunc( first_reshape_callback );
        glutMouseFunc( mouse_callback );
        glutMotionFunc( motion_callback );
        glutPassiveMotionFunc( motion_callback );
         ...
        // Give up control to GLUT
        glutMainLoop();
}
```

FIG. 8.48 – PV_GlutWindow : méthode *mainLoop*

Dans un premier temps, elle initialise les différents paramètres d'affichage et de gestion des évènements. La classe *PV_Window* représente le bas niveau de notre implémentation, pour la gestion de notre fenêtre d'affichage.

Afin de s'assurer qu'il n'y a qu'une seule instance de ces classes, nous avons défini une variable statique, un singleton, *_instance* permettant de valider cette unicité (listing 8.49).

```
if ( _instance ) {
    TRACE_DEBUG( "-E- Attempt to build a second mapView2D object" );
    abort();
}
// Pointer to this object for GLUT callbacks
_instance = this;
```

FIG. 8.49 – Une seule instance d'affichage autorisée

La surcouche graphique, que nous avons abordée ci-dessus, permet de gérer un affichage 2D, mais elle reste très générique et n'a pas de notion de type d'objets, de Cycab ou autres. Pour ce faire, *PV_Window* inclut une autre classe nommée *PV_View*. Cette dernière est une classe abstraite car contenant au moins une méthode virtuelle pure [Sil98].

Nous avons implémenté une classe *mapView2D* qui hérite de *PV_View* (figure 8.50) et qui est spécifique à notre besoin ; cette classe va prendre en charge la gestion du plan statique, des descriptions géométriques des objets, ... Pour ce faire elle va utiliser différents types d'objets (cf. diagramme des classes figure 8.57) pour :

- le plan statique -> classe *planVizualisation*,
- les cibles -> classe *targetsVizualisation*.

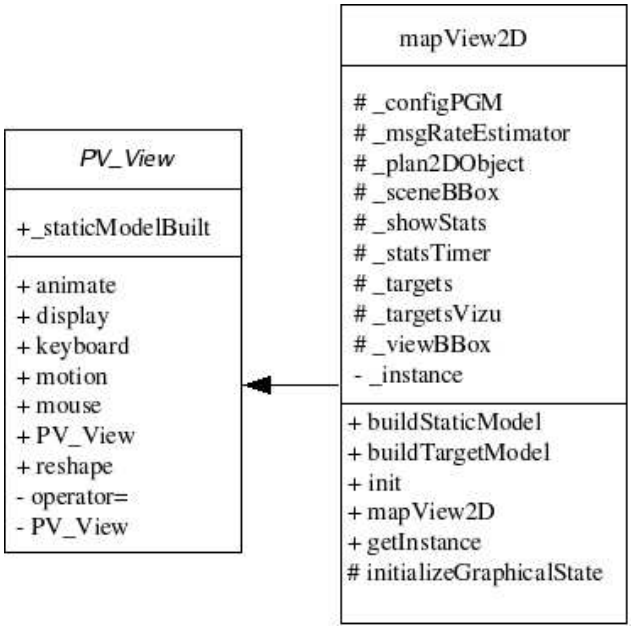

FIG. 8.50 – Classes spécialisées *PV_View* et *mapView2D*

8.6.5 Le *WebExport*

Nous avons testé une fonctionnalité qui nous paraissait intéressante : pouvoir obtenir en temps réel un affichage du plan via une page Web. Autrement dit, obtenir l'équivalent d'une copie de l'écran OpenGL et l'envoyer dans un fichier de format image.

Ce fichier est stocké dans l'arborescence du site Web de Parkview[23] et affichable via une page écrite en PHP.

[23] http ://emotion.inrialpes.fr/parkview. Dernière consultation : 2-dec-05.

L'opération d'export est facile à réaliser avec quelques primitives OpenGL et permet de générer à la volée un fichier type *BMP* ou *bitmap*, comme nous allons le voir.

```
void mapExportWeb(int view, char *path, char *filename )
{
  GLubyte * tempSnapshot;
  glutSetWindow(view);
  unsigned int dx = glutGet(GLUT_WINDOW_WIDTH);
  unsigned int dy = glutGet(GLUT_WINDOW_HEIGHT);
```

FIG. 8.51 – Fonction d'export Web. Initialisation

La figure 8.51 montre la fonction d'export avec ses différents paramètres (tableau 8.30). La première action est de récupérer la configuration de la fenêtre.

Paramètres	Description
int view	Le numéro de la fenêtre OpenGL.
char *path	Le chemin de sauvegarde du fichier.
char *filename	Le nom du fichier généré.

TAB. 8.30 – Paramètres de la fonction mapExportWeb

```
glReadPixels(0,0,dx,dy,GL_RGB, GL_UNSIGNED_BYTE,tempSnapshot);
```

FIG. 8.52 – Lecture des pixels du rectangle.

Toute la récupération des données repose sur la fonction OpenGL *glReadPixels*, dont la syntaxe d'appel est indiquée dans la figure 8.52 et le tableau 8.31.

Paramètres	Description
0,0,dx,dy	Définit le rectangle à sauvegarder. Ici, toute la fenêtre.
GL_RGB	Stocke le composant chromatique rouge, puis le vert et le bleu.
GL_UNSIGNED_BYTE	Les données sont sous la forme d'entiers 8 bits non signés.

TAB. 8.31 – Paramètres de la fonction glReadPixels

Cependant, l'image générée est volumineuse, ceci à cause du format BMP, et ne peut en aucun cas permettre un affichage rapide - encore moins temps réel - pour un internaute en dehors de l'intranet de l'INRIA.

L'idée peut-être maintenue, mais il faudrait alors voir pour convertir cet export dans un format compressé, tel que PNG[24] (Portable Network Graphics) ou bien JPEG[25] (Join Photographic Experts Group)(PNG convenant mieux car l'image a peu de couleurs).

[24]http ://www.libpng.org/pub/png/. Dernière consultation : 19-mai-05.
[25]http ://ou800doc.caldera.com/en/jpeg/libjpeg.txt. Dernière consultation : 19-mai-05.

Ceci doit être fait en gardant à l'esprit de ne pas impacter les performances du client graphique, ou alors en développant un client dédié qui ne ferait que cela. Ceci pourrait faire l'objet d'un travail futur.

8.6.6 Résumé

Le client graphique fut très vite un outil indispensable afin de vérifier et valider les traitements que nous avons développés. Son développement fut délicat car nous voulions avoir un composant extensible (par exemple vers un affichage en 3D) ce qui pose de nombreuses contraintes dans la représentation du monde.

Cependant, il fut abondamment utilisé que ce soit lors de la mise au point des programmes ou bien lors des démonstrations de la plate-forme. S'appuyant sur des librairies d'affichage classiques et bas niveau, les performances sont très bonnes et permettent d'envisager de développer des extensions ou d'autres types de clients graphiques (comme par exemple pour des assistants personnels ou PDA).

8.7 Mise au point et mesure de performance

Lors du développement d'une application client/serveur avec échanges importants de messages (voire massifs suivant les cas), la partie mise au point, analyse du code est importante, entre autres pour éviter les fuites mémoire souvent causes de défaillance d'un programme. Dans notre cas, nous avons cherché des outils gratuits permettant de faire ce travail.

Nous avons finalement utilisé valgrind [26] comme outil principal et nous avons aussi mis en place plusieurs outils de mesure.

8.7.1 Valgrind

8.7.1.1 Présentation

Valgrind est un ensemble d'outils Open Source permettant de suivre le déroulement d'un programme et de retourner tout problème lié à l'exécution, à la mémoire, ... Valgrind va nous permettre d'extraire différentes anomalies :

- accès à une zone mémoire non prévue,
- utilisation d'une variable avant son initialisation,
- fuite mémoire (non libération de mémoire par exemple),
- arrêt du programme (ou *segmentation fault*).

8.7.1.2 Lancement

Pour pouvoir fonctionner correctement, il faut que le programme à tester soit compilé avec l'option « -g », qui active les options de débugage. A noter que les options d'optimisation, types « -O1, -O2, ... » ne sont pas recommandées, car elles perturbent les outils de Valgrind.

La figure 8.53 montre le shell script permettant de lancer le mapServer via Valgrind.

[26] http ://www.valgrind.org Dernière consultation : 2-oct-05.

```
#!/bin/sh
valgrind --logsocket=127.0.0.1:12345 \
        --suppressions=mapServer.supp \
        --error-limit=no \
        --leak-check=full \
        mapServer -c config_bistre.xml
```

FIG. 8.53 – Exécution de Valgrind

Avec les paramètres fournis en ligne de commande, l'outil va envoyer les informations sur une socket. Un programme fourni va pouvoir écouter sur cette socket et stocker les informations. D'autre part, il est possible de dire à Valgrind de ne pas gérer/*trapper* certaines erreurs, c'est le rôle de l'option --suppressions. La dernière ligne représente la commande à exécuter réellement avec les paramètres éventuelles.

8.7.1.3 Résultats

Les outils de Valgrind sont très verbeux et permettent ainsi d'avoir un maximum de précision sur les erreurs rencontrées. Le listing 8.54 donne un exemple d'une erreur sur l'utilisation abusive de la commande *delete*.

```
(1) ==16693== Invalid free() / delete / delete []
(1) ==16693== at 0x1B904AC1: operator delete[](void*) (vg_replace_malloc.c:161)
(1) ==16693== by 0x80551F2: configConnector::~configConnector()...
(1) ==16693== by 0x80505EC: manageCycab(configConnector*, bcServer*) ...
(1) ==16693== by 0x8058036: main (main.cpp:134)
(1) ==16693== Address 0x1BAC5F68 is 0 bytes inside a block of size 11 free'd
(1) ==16693== at 0x1B904AC1: operator delete[](void*) (vg_replace_malloc.c:161)
(1) ==16693== by 0x80551F2: configConnector::~configConnector() ...
(1) ==16693== by 0x80505EC: manageCycab(configConnector*, bcServer*) ...
(1) ==16693== by 0x8058036: main (main.cpp:134)
```

FIG. 8.54 – Exemple de résultat

Cet outil nous a permis d'optimiser notre code et aussi de résoudre plusieurs conflits/plantages rencontrés.

8.7.2 PerfCounter

Comme vu à la section 8.3.5.2 nous avons créé un ensemble de fonctions nous permettant de mesurer les temps d'exécution de nos programmes. Ces fonctions nous ont permis de mettre en évidence, entre autres, la différence de comportement du client graphique lorsqu'un pilote ou *driver* optimisé est utilisé ou non.

8.7.3 Résumé

Valgrind est un outil très performant et utile pour des développements complexes. Les rapports générés permettent d'obtenir de nombreuses informations et ainsi aident à améliorer le code. Nous l'avons beaucoup utilisé, ce qui nous a permis d'optimiser des portions entières de code.

8.8 Synthèse

8.8.1 Résumé

Nous avons présenté dans ce chapitre le serveur de cartes et ses différents modules : le *trackerConnector*, le *mapServer* et le client graphique. Ces composants logiciels sont le cœur de notre infrastructure et permettent d'obtenir une modélisation de l'environnement utilisé. Cette plate-forme logicielle correspond au cahier des charges dans la mesure où elle est ouverte - il est facile de rajouter des extensions - et performante - les tests ont démontré un traitement en temps réel des données.

Le *trackerConnector* permet d'avoir cette ouverture de l'infrastructure, puisqu'il rend compatible un logiciel tel qu'un *tracker* avec notre protocole de communication et notre format XML. Le cas échéant, il est capable de procéder à des transformations de données.

Le *mapServer* quant à lui collecte les données des *trackerConnector*, effectue différents opérations sur ces informations afin d'obtenir en sortie la carte de l'environnement à l'instant t, autrement dit, un ensemble de cibles.

Finalement, le client graphique permet de visualiser de manière très efficace le résultat des traitements effectués en amont. Sa mise en œuvre fut assez complexe de part la difficulté de dessiner tous les éléments que nous devons gérer, mais le résultat est très satisfaisant.

8.8.2 Quantification du développement

En terme de volumétrie, la globalité du développement porte sur plus de 16500 lignes (commentaires inclus) et plus de 60 classes au total. Certaines portions de code ont été adaptées du premier prototype, d'autres nous ont servi de matière première pour développer nos propres classes (une partie des classes de gestion graphique). Nous avons aussi utilisé des librairies déjà existantes dans l'équipe comme par exemple celles du Cycab pour la partie communication avec la voiture ou les en-têtes C++ de ProBT© pour le module jPDA. Ces modules externes ne sont pas pris en compte dans le calcul du nombre de lignes.

8.8.3 Diagrammes de classe

Afin de ne pas surcharger ce mémoire, nous n'avons pas détaillé toutes les classes mises en jeu pour chaque module, cependant, les figures qui suivent représentent les diagrammes de classes simplifiés, c'est-à-dire sans les méthodes et sans les membres, de chaque composant du serveur de cartes.

Le diagramme de classes 8.55 représente le trackerConnector dans une version épurée, car seules les classes importantes sont représentées. Nous retrouvons au centre les classes pour la configuration du programme. Nous voyons aussi les parties communication et traitement des données.

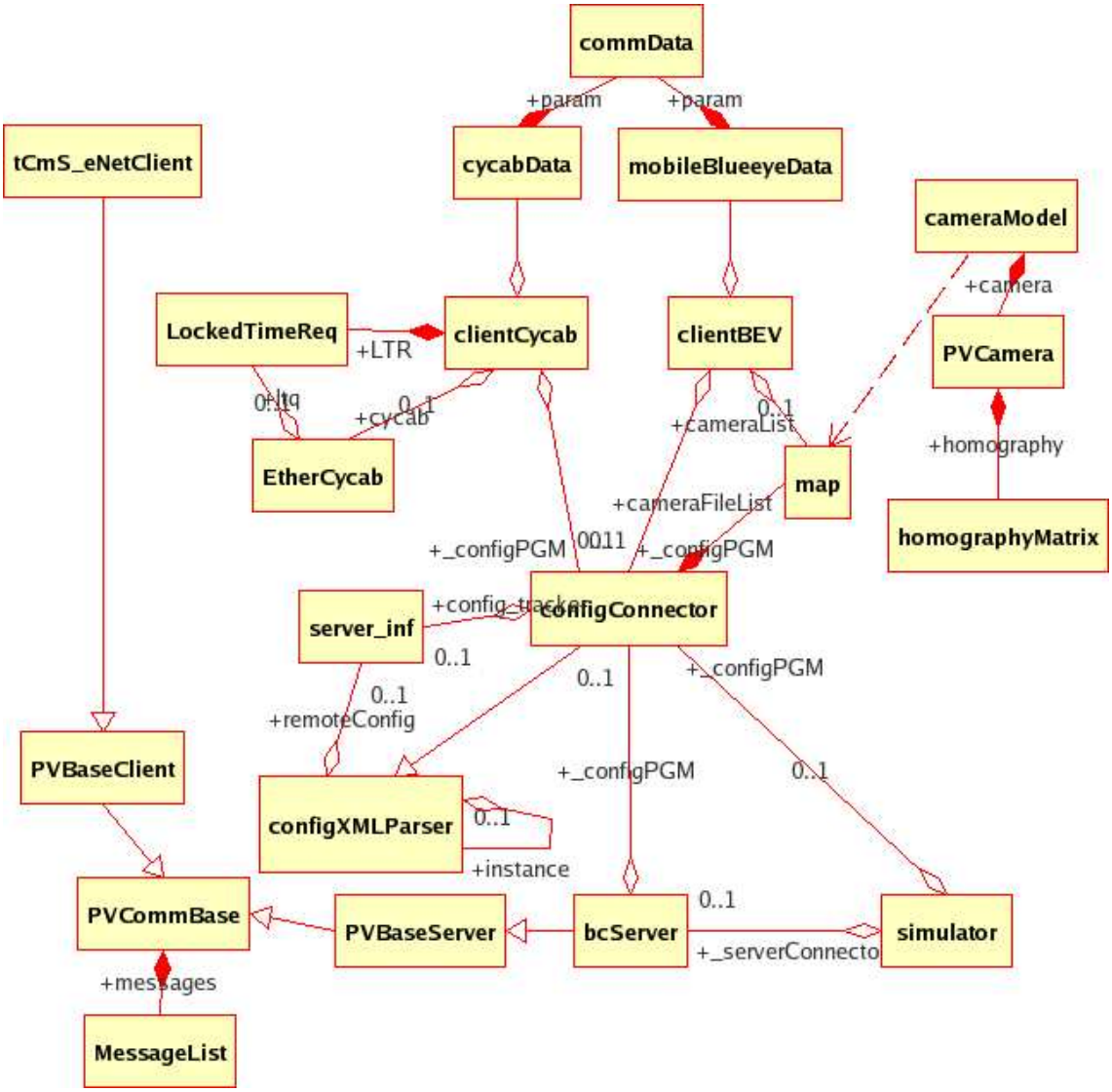

FIG. 8.55 – Diagramme de classes du *trackerConnector*

De même pour le mapServer (figure 8.56), seules les classes les plus importantes sont affichées. A noter les classes relatives au jPDA.

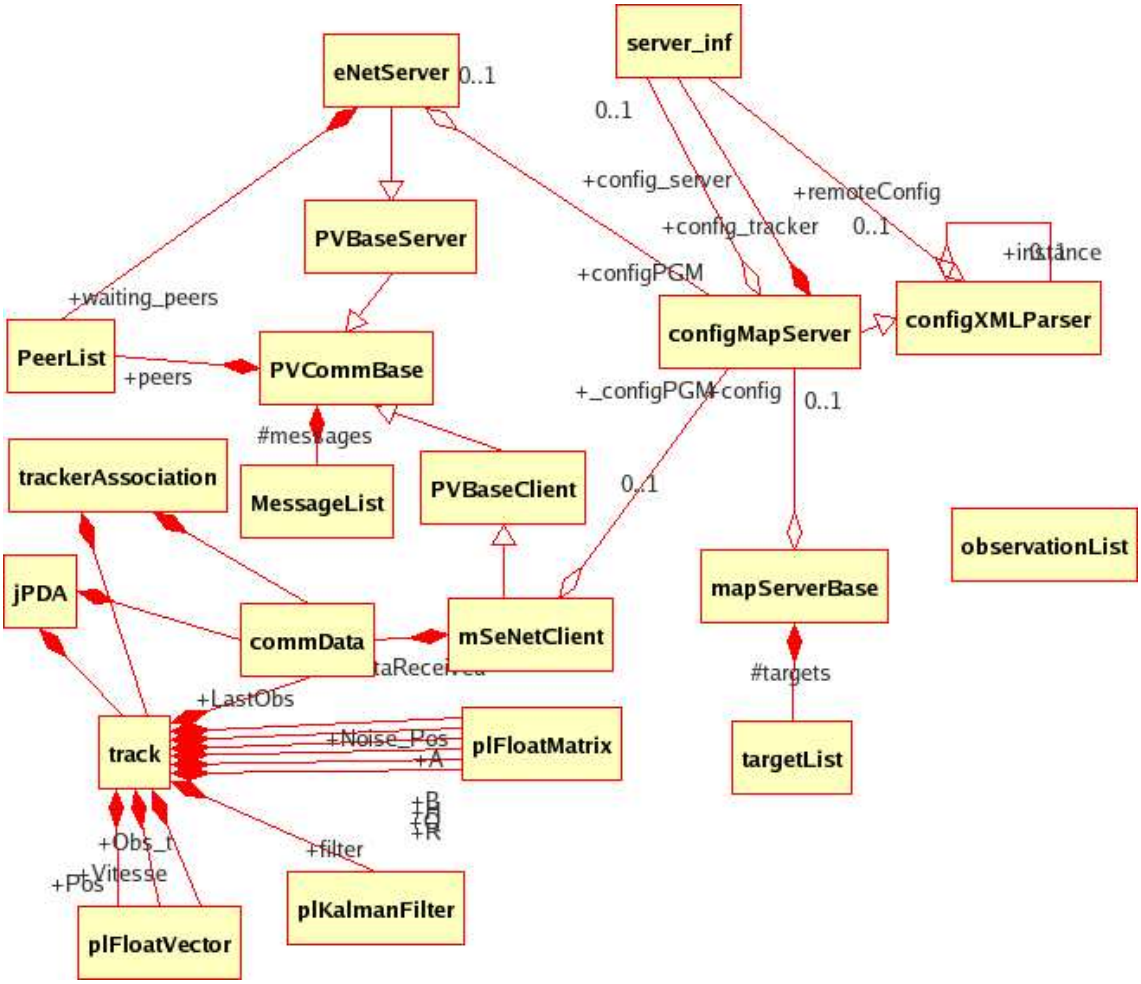

FIG. 8.56 – Diagramme de classes du *mapServer*

Comme nous l'avons dit ci-dessus, le client graphique est la partie la plus complexe de l'architecture en terme de modélisation objets et de classes, ce que montre la figure 8.57. Ce diagramme comporte des classes déjà rencontrées, telles que la partie communication ou bien la partie configuration. Par contre, la complexité principale réside dans les classes nécessaires à la représentation du monde. Elles sont toutes rattachées à *mapView2D* que nous avons abordé dans la section 8.6.4 (cf. figure 8.50).

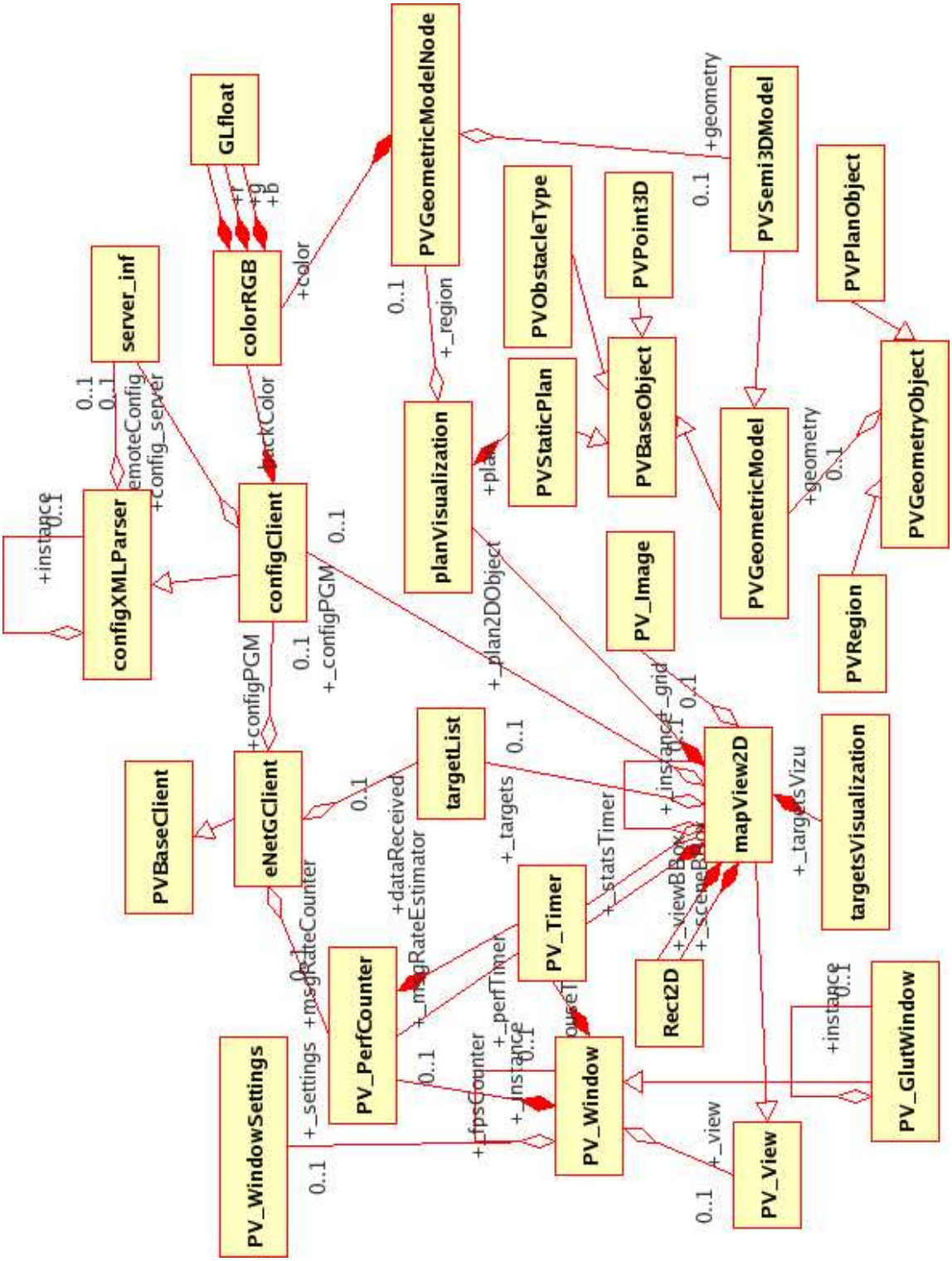

FIG. 8.57 – Diagramme de classes du client graphique

Chapitre 9

Conclusion

La robotique fascine depuis des années, rendue attrayante grâce à la littérature et au cinéma, des *Robots* d'Asimov aux robots de *la Guerre des Etoiles*.

Cependant, nous sommes loin de cette vision futuriste. La robotique est une science jeune à des années lumière des romans ou des films. Sa principale caractéristique est son aspect multi-disciplinaire : l'informatique, les mathématiques, la mécanique, la vision, ... Ce stage a permis d'effectuer différents travaux sur ces thèmes.

9.1 Travaux effectués

Notre problématique principale est la modélisation de l'environnement dans lequel va évoluer un véhicule robot, à savoir le parking arrière de l'INRIA. Les développements précédents [Hel03] n'ayant pas été maintenus, il a fallu repartir de zéro afin d'intégrer les évolutions technologiques, les nouvelles contraintes liées au projet, ...

Pour ce faire, et afin de répondre aux besoins du projet Parknav, il a fallu créer un serveur de cartes s'appuyant sur la plate-forme expérimentale, répondant au nom de Parkview. Nous avons pu maintenir cette plate-forme matérielle, la faire évoluer en fonction des nouveaux besoins ou tout simplement répondre à des contraintes d'exploitation. Les développements réalisés ont permis d'obtenir un outil capable de modéliser en temps réel l'environnement de la plate-forme.

Parknav c'est aussi l'association de plusieurs partenaires, travaillant sur la vision, les détections d'objet, ... Notre infrastructure est l'addition de nos développements et de nos réalisations.

Au final, véritable travail d'équipe, nous avons là un outil d'expérimentation très ouvert et accessible, facilement évolutif. Les diagrammes présentés dans la section 8.8 montrent que le travail réalisé fut important, fastidieux et souvent complexe.

9.2 Méthodes de travail

Le travail d'un ingénieur, surtout sur un projet de développement, nécessite de l'organisation et de la rigueur pour que le projet respecte les contraintes imposées par la maîtrise d'ouvrage (délai, risque, contraintes techniques, ...).

Dans le cadre de ce travail, nous avons essayé d'appliquer différentes techniques de génie logiciel que nous allons aborder rapidement ici.

Dans un premier temps, nous avons mis en place un outil d'organisation d'idées, suivant le principe du « Mind Mapping » ou de cartes heuristiques[1]. Cette technique permet de représenter graphiquement et simplement des idées et les liens qui existent entre elles. L'outil utilisé se nomme *freemind*[2] (cf. annexe F).

En plus de cela, nous avons utilisé un tableau de bord de gestion du stage (voir annexe F). Nous avons découpé le stage en différents lots macroscopiques, représentants les grandes étapes du stage ou bien les modules principaux de développement, comme par exemple : le lot apprentissage des techniques ou le lot *trackerConnector*. Ce tableau de bord nous a permis de créer un planning sur toute l'année avec un calcul de charges qui furent réévaluées au fur et à mesure de l'avancement. Un outil libre de planification - *ganttProject*[3] sorte de MS Project© - nous a permis de représenter graphiquement (diagramme de Gantt[4]) les différents charges.

Des points réguliers avec Thierry Fraichard ont permis de cadrer l'avancement du projet, de débatter sur les choix technologique, etc.

A chaque étape de codage des différents modules ou à chaque investigation technique, nous avons aussi procédé à la validation ou recette du code créé. Pour ce faire, nous avons identifié des scénarios de tests nous permettant d'avoir un niveau de validation suffisant (exemple du tableau 7.4).

L'INRIA défend le principe du logiciel libre via différentes actions ou partenariats (voir le consortium Object Web[5] par exemple). Nous avons pris le parti de développer en utilisant principalement des logiciels dits Open Source, d'une part pour la gratuité de ces produits, d'autre part, pour l'accès aux sources nous permettant ainsi d'adapter le cas échéant les programmes ou bien de mieux comprendre les mécanismes internes. Les logiciels libres ont aussi l'avantage d'être fournis par des communautés souvent très dynamiques, ce qui permet de résoudre rapidement les problèmes éventuels.

La documentation d'un développement de cet ampleur est un gage de pérennité pour les programmes ; ils peuvent être maintenus plus facilement. Pour ce faire, nous avons mis en place une documentation automatique via l'outil Doxygen (produit libre). Il nécessite de la rigueur mais permet d'obtenir une documentation très fonctionnelle.

Enfin, nous avons aussi utilisé un outil libre pour la modélisation de nos classes et la représentation des liens inter-objets (logiciel *umbrello*, abordé dans l'annexe F). Ce type d'outil permet d'avoir une vision claire de l'objectif à atteindre et aussi de partager sa vision avec les membres de l'équipe.

9.3 Le futur de Parkview

La vie de Parkview et du serveur de cartes ne doit pas s'arrêter à la fin de ce stage ou de ce document. La plate-forme doit encore évoluer surtout d'un point de vue logiciel et de nombreux axes de travaux futurs sont apparus.

Ainsi, l'une des priorités va être d'interconnecter le Cycab, en tant que client, avec le serveur de cartes. Ceci permettra à e-Motion de pouvoir développer de nouveaux programmes de navigation automatique, tenant compte de la carte dynamique du parking. En parallèle, le développement de la nouvelle localisation permettrait d'avoir une précision supplémentaire sur la position du Cycab.

Les objets en mouvement ne sont pas caractérisés, autrement dit, nous ne faisons pas la différence entre un piéton, un vélo ou une voiture. Il serait intéressant de pouvoir intégrer ce genre de mécanisme, qui permettrait de rendre encore plus précis la carte de l'environnement et les décisions qui en découlent. Les partenaires de e-Motion ont des sujets de recherche en cours sur ce thème.

[1] http ://fr.wikipedia.org/wiki/Mind_mapping. Dernière consultation : 2-jan-06.
[2] http ://freemind.sourceforge.net. Dernière consultation : 20-dec-05.
[3] http ://ganttproject.sourceforge.net/fr/. Dernière consultation : 2-jan-06.
[4] http ://fr.wikipedia.org/wiki/Gantt. Dernière consultation : 2-jan-06.
[5] http ://www.objectweb.org. Dernière consultation : 20-dec-05.

De nouveaux modules de fusion et d'association pourraient faire l'objet de tests et de développements de clients graphiques multi-plates-formes

Le parking n'est pas totalement couvert, entre autres, l'extension réalisée entre 2003 et 2004. Rajouter des caméras implique un minimum de travail pour pouvoir les intégrer dans la plate-forme (calibrage, calcul des paramètres d'homographie, tests).

En résumé, cette plate-forme devrait encore voir de nombreux stagiaires la faire évoluer.

9.4 Bilan personnel

Ce stage m'a permis de mettre à profit mes acquis dans la gestion de projets et mes compétences techniques au niveau informatique.

Au sein de mon entreprise, je suis amené à gérer des projets de tailles diverses et à mettre en place des outils de gestion et de suivi. Ce stage m'a permis de tester ce type de démarche avec des outils simples et gratuits.

Ce fut aussi l'occasion de découvrir des méthodes ou technologies que je ne connaissais pas telles que la vision, la mécanique, la robotique, ... et de pouvoir en un an acquérir de nombreuses compétences nouvelles.

Mais par dessus tout, ce fut l'occasion de travailler avec des personnes passionnantes et passionnées, venant de tous horizons, sur un sujet qui me fascine depuis mon enfance.

Annexes

Annexe A

Le plan du parking

A.1 Plan du Parking

Nous allons aborder un peu plus en détail le fichier XML décrivant le plan du parking, qui sera envoyé au client graphique par le serveur de cartes.

Il sera utilisé pour représenter le parking en 2D ou 2.5D.

Effectivement, ce fichier nous fournit des coordonnées au format 2.5D, comme décrit dans la section 4.1.

L'exemple qui suit ne prend qu'un extrait du fichier. Les mesures sont ici en mètre, sachant que nous avons un autre fichier avec des mesures faites en centimétre.

A.1.1 Analyse d'un extrait du fichier

En-tête

Les premières lignes de la figure A.1 comportent l'en-tête classique d'un fichier XML, puis le tag principal « staticplan » et un paramètre indiquant que nous sommes dans un format 2.5D. Ce dernier champ permet d'anticiper un futur format 3D.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<!-- Example definition file for a static plan.
        Attribute default values are shown between braces in the comments
        Mandatory attributes are indicated by a *  -->

<!--Root object.
     Attributes: format specifies if we use a 2.5D definition
           or a full 3D definition.-->
<staticplan format="2.5">
```

Fig. A.1 – Début fichier plan

Objets types

Cette section du fichier - figure A.2 - définit un ensemble d'objets types, tels que les places de parking (« PlaceMark ») ou bien les trottoirs (« Sidewalk »).

A chaque objet nous donnons aussi une couleur par défaut, qui pourra être redéfinie dans le programme d'affichage.

```
<!−−objecttypes are used to integrate and validate semantic information
    Attributes
     name∗ : the mean associated to the semantic context.
     virtual [ false ]: is it a physical or virtual object.
     −−>
<objecttype name="PlaceMark" color="#FFFF00"/>
<objecttype name="ParkingPlace" virtual = "true"/>
<objecttype name="Building" color="#00FFFF"/>
<objecttype name="Sidewalk" color="#808080"/>
<objecttype name="Lamp" color="#FF00FF"/>
```

Fig. A.2 – Plan : objets types

Région principale

Le plan comporte une région principale (cf. A.3), la partie du parking que nous traitons. Cette zone est définie par un nom, une couleur par défaut et les coordonnées des quatre coins du parking (la forme pouvant être quelconque).

Nous venons de délimiter le plan à gérer.

```
<!−−A static plan may have many regions, for example, stages of a parking building
        Attributes:
        label [""]: The unique identifier of the region −−>
<region label="MainParking" color="#FFFFFF">
<!−−A point in the region.
        Attributes:
        x∗, y∗      : The point coordinates as real numbers.
        label [""] : A text tag for the point.−−>
<point x="−1.30" y="−1" />
<point x="−1.30" y="40" />
<point x="45" y="40" />
<point x="45" y="−1" />
```

Fig. A.3 – Plan : la région étudiée

Un objet

Dans l'exemple A.4, nous définissons - partiellement, car il manque la fin de la déclaration - un objet de type trottoir (« Sidewalk »), auquel nous attribuons un label (en l'occurrence ici, il s'agit du trottoir situé à droite du garage à vélo).

```
<!−−Objects populating the environment.
    Attributes:
      type∗       : An object type previously defined using objecttype.
      height   [0]: The height of the object.
      label   [""] :  A text tag for the object. −−>
<object type = "Sidewalk" label = "Right−side−of−Bike−parking"
   height = "0.15">
  <point x="−1.606" y="−1" />
  <point x="16.65" y="−1" />
  <point x="16.65" y="4.50" />
  <point x="15.33" y="4.50" />
  <point x="15.33" y="0" />
  <point x="0" y="0" />
  <point x="0" y="4.50" />
  <point x="−1.5995" y="4.4769" />
  <point x="−1.5801" y="0" /> <!−− EBO − 16−mar−05 −−>
  <point x="−15.0434" y="0.0144" /> <!−− EBO − 16−mar−05 −−>
```

Fig. A.4 – Plan : exemple d'un objet

Comme nous l'avons vu, nous sommes dans un format 2.5D, autrement dit nous devons retrouver l'ensemble des coordonnées (x,y) des points définissant l'objet plus une indication de hauteur.

C'est le cas ici avec le paramètre height.

A.2 Description des objets

Le client graphique a besoin de connaître aussi la description géométrique des objets pour pouvoir les représenter graphiquement.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<targets format="2.5">
```

Nous sommes toujours dans un format type 2.5D (x, y, hauteur).

```
<class name="default" height="2.0">
  <circle radius="0.5"/>
</class>
```

Ceci définit la forme par défaut des objets : un cercle.

Le Cycab est modélisé par un polygone tel que définit en A.5 (à savoir un rectangle pour la caisse du Cycab et un autre rectangle pour schématiser le laser Sick).

```
<class name="cycab" height="1.5">
  <polygon>
    <point x="1.65" y="0.6"/>
    <point x="1.65" y="0.2"/>
    <point x="1.85" y="0.2"/>
    <point x="1.85" y="−0.2"/>
    <point x="1.65" y="−0.2"/>
    <point x="1.65" y="−0.6"/>
    <point x="−0.35" y="−0.6"/>
    <point x="−0.35" y="0.6"/>
  </polygon>
</class>
```

Fig. A.5 – Géométries : le Cycab

Nous avons plusieurs types d'objets (figure A.6) : des Cycab de couleurs différentes, l'objet piéton ou vélo, ainsi qu'un type non défini (Objet ou Véhicule Non Identifié).

```
<object label="cycab−red" type="cycab" color="#FF0000"/>
<object label="cycab−blue" type="cycab" color="#0000FF"/>
<object label="pedestrian" type="default" color="#0000FF"/>
<object label="bicycle" type="default" color="#FF0000"/>
<object label="ovni" type="default" color="#FF00FF"/>
```

Fig. A.6 – Géométrie : les types d'objets

Toutes ces caractéristiques seront envoyées au client graphique dans la phase de connexion. Il les utilisera pour la représentation 2D des objets.

Annexe B

Guide d'utilisation des trackers

B.1 Installation de PrimaBlue

Comme indiqué dans la section 4.4.1.4, PrimaBlue a un ensemble de prérequis pour pouvoir fonctionner correctement.

Tout d'abord, ce logiciel étant dérivé de BlueBehaviour©, il nécessite une licence spécifique à la machine où il devra tourner. Aussi, il faudra demander à l'équipe Prima de générer une licence pour la machine hôte. Il s'agira d'un fichier nommé *license.txt*, contenant une clé binaire calculée d'après l'adresse MAC de la machine. Ce fichier devra être stocké sur la machine elle-même dans un répertoire /etc/BEV. Sans ce fichier, l'application ne sera pas fonctionnelle.

Concernant les librairies, le *tracker* a besoin des *svideotools* - nécessaires aussi à PrimaLab. Il s'agit d'utilitaires permettant de traiter un flux vidéo, que ce soit direct (caméra) ou pré-enregistré. PrimaBlue nécessite plus particulièrement le logiciel *svideod-mpeg_play*, afin de pouvoir rejouer des vidéos pré-enregistrées. Ces outils sont en général dans le répertoire /usr/local/bin (sur les machines Carolus et Parkview).

Les fichiers de configuration de PrimaBlue sont dans un format XML qui nécessite un schéma pour valider le contenu [W3C05]. Ce dernier doit se trouver dans le répertoire /etc/BEV de la machine et porte le nom de blueEyeConfig.xsd (voir avec l'équipe Prima pour le récupérer).

Toujours concernant XML, la version du tracker, que nous avons, utilise la librairie Xerces-C++ du projet Apache XML[1]. Mais PrimaBlue n'est pas compatible avec les versions récentes. Il lui faut absolument la version 1.7, disponible sur le site Web de Parkview.

B.2 Utilisation de PrimaBlue

PrimaBlue se lance via un shell script nommé exec-BlueEye.

Ains en supposant que le tracker soit installé dans la *home directory* du compte ebonifac, il faudra lancer :

```
\~ebonifac/Trackers/Blueeye/exec-BlueEye
```

Ceci lancera en fait *Fedora_blueBehaviour*.

Ensuite, il faut rajouter un canal vidéo à gérer - peut être un flux pré-enregistré :

```
Video channels->add
```

Le champ *caption* permettra de faire le lien avec le numéro fourni au connecteur (il faut 1 chiffre).

[1] http ://xml.apache.org/xerces-c/. Dernière consultation : 26-juil-05.

Choix de la résolution vidéo :

```
Input->Video : resolution 384x288
```

Comme nous l'avons dit dans le rapport, il est possible de mettre une résolution supérieure, en théorie, afin d'obtenir de meilleurs résultats. Cependant, les tests ont montré que la machine hôte doit dans ce cas être très puissante, le logiciel utilisant beaucoup de ressources systèmes.

B.2.1 Vidéo pré-enregistrée

Pour pouvoir rejouer une vidéo pré-enregistrée, il faut cocher la case *MpegPlayerGUI*, puis faire *selectMpegFile*.

Il faut ensuite charger un fichier de configuration, qui contiendra, entre autres, les zones de détection : sélectionner *Configuration file->load* en bas de la fenêtre.

Par exemple, ẽbonifac/Trackers/Blueeye/Parknav_fev05/.

Cliquer sur *Run* pour lancer le playback de la vidéo.

IMPORTANT : le programme va lire la vidéo jusqu'au bout puis boucler. Cependant, les vidéos générées avec nos outils ne fonctionnent pas en boucle et cause le plantage du tracker dans ce cas.

Pour résoudre ce problème, il est possible d'utiliser un utilitaire nommé avidemux qui permet de corriger les fichiers MPEG. Voir le site de l'outil afin d'obtenir plus d'informations. Pour arrêter le playback de la vidéo, faire *Input->Video->Stop*, il ne faut pas toucher au contrôle du logiciel de lecture.

B.2.2 Vidéo directe

Il faut, dans un premier temps, choisir le périphérique ou *device* lié à la caméra voulue : *input->video->device*. Choisir parmi les */dev/video** (4 sur Carolus, 3 sur Parkview).

Le démarrage du traitement vidéo se fait en cliquant sur le bouton *Run*, ce qui va lancer l'utilitaire svideotools.

Comme précédemment, il faut choisir le fichier de configuration, puis lancer le tracker.

B.2.3 Sauvegarde de la configuration

Une fois que tout les trackers tournent, settings->SAVE

Export->NetworkConfig->SendData, puis activate

Editer config trackerConnector suivant video channels

Lancer le trackerConnector

Dans la fenetre video, LMB fait snapshot dans Blueeye/video/

settings->load demande 1 nom de fichier d'events Parknav_fev05/eventParkview Apres avoir sauve 1 fichier de conf, ajouter a la main dans la section tracker (en bas) <normalizeIntensity value="true">

B.3 PrimaLab

A noter que Primalab est basé sur Ravi, qui signifie Robotique, Apprentissage, Vision, Intégration [LZ99].

Cette plate-forme est un système intégré C++-Scheme[2]

L'avantage de ce duo C++-Scheme est d'offrir une grande facilité d'extension du logiciel Primalab de base.

Le tracker Primalab v2 et v4 s'appuient sensiblement sur les mêmes composants (tableau B.1).

[2]Introduction à Scheme : http ://users.info.unicaen.fr/m̃arc/ens/deug/intro/intro.html

Ravi
PrimaVision
SvideoTools
Imalab
Le tracker lui-même

TAB. B.1 – Les composants de Primalab

Les logiciels cités s'installent dans l'ordre du tableau. En dehors de svideotools, tous les logiciels sont fournis et développés par l'équipe Prima. Svideotools quant à lui est un serveur vidéo qui permet de rejouer des films pré-enregistrés, il est téléchargeable sur le site Web de Parkview[3].

Il a fallu aussi rajouter un module de communication[4], non fourni en standard dans les paquets d'installation de PrimaLab.

Ce module va nous permettre d'interconnecter le tracker à notre architecture, via un *trackerConnector* (voir pour le détail la section 8.4). Il s'appuie sur la librairie GNU CommonCPP pour proposer les services réseaux nécessaires [GNU05].

[3]http ://emotion.inrialpes.fr/parkview/documents/svideotools.tgz
[4]Disponible aussi sur le site Web

Annexe C

Licences des logiciels utilisés

GPL :
Apache License 2.0 :
MIT : http ://www.opensource.org/licenses/mit-license.html
Tiré du site : http ://fplanque.net/
"Combien existe-t-il de licences Open-Source ?

Beaucoup !

De très nombreux auteurs ont décidé qu'une licence existante (notamment la licence GPL et son aspect viral) ne correspondait pas à leurs impératifs et ont créé la leur, souvent en modifiant une licence existante. Il faut savoir aussi que certaines licences se prétendent "open source" sans l'être vraiment.

Une liste de référence est fournie par l'« Open Source Initiative » ou OSI.

Pour avoir le label « Open Source », une licence doit vérifier les 10 points de la définition de l'OSI[1].

Le problème actuel est l'abondance de licences dites Open Source, sans parler de celles qui sont dites libres, mais ne vérifiant pas les règles de l'OSI !

La figure C.1 montre la répartition des licences les plus répandues. Nous pouvons noter, sans grande surprise, que la licence GPL est la plus ancrée dans le monde de l'Open Source.

Cette licence est dire « virale » car elle contamine les programmes qui intègre le code GPL. Autrement dit, si vous incluez dans votre code source, des bouts de code sous licence GPL, vous êtes obligés de passer tout votre programme sous cette licence. Ceci n'est pas très génant pour un développement libre, mais peut l'être pour une entreprise souhaitant réutiliser du code existant. C'est pour cela, que d'autres licences ont fait leur apparition, plus permissives que la GPL : LGPL, MPL, ...

[1] http ://www.opensource.org/index.php. Dernière consultation : 1-dec-2005.

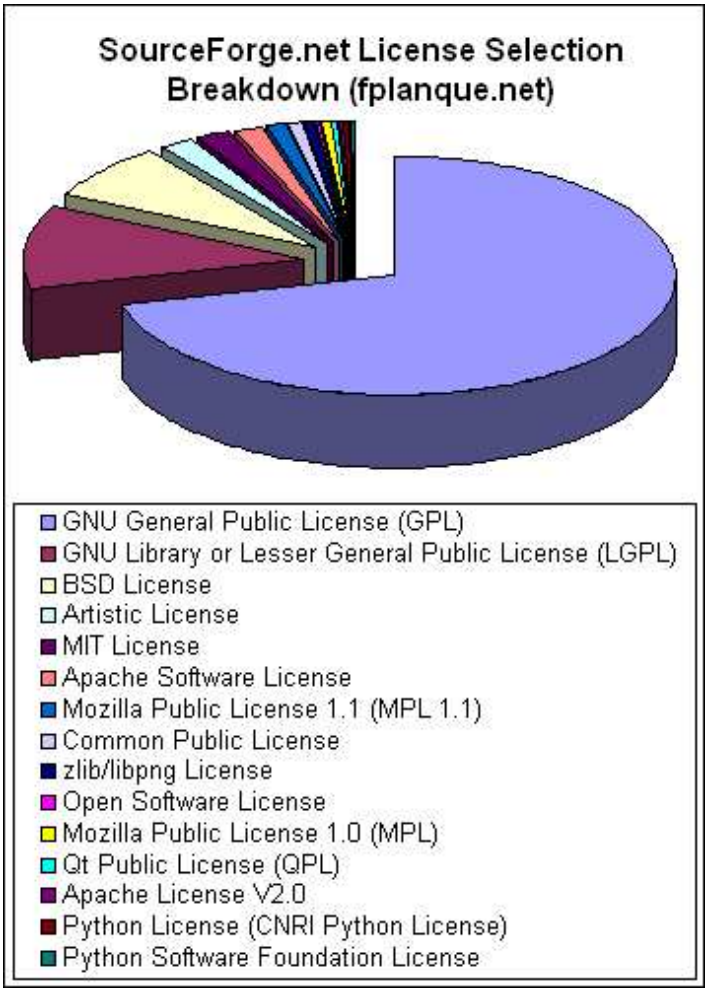

FIG. C.1 – Répartition des licences. Sources : http ://fplanque.net

Le site *http ://fplante.net* classe les licences en 3 grandes catégories permettant d'y voir un peu plus clair.

1. Les licences autorisant à basculer en closed source à tout moment (la seule condition étant de mentionner les auteurs et leur copyright) :
   (a) BSD License,
   (b) MIT License,
   (c) X.Net License,
   (d) autres licences dites "permissives".
2. Les licences obligeant à garder le code en open source en cas de modification, mais autorisant la combinaison avec du code closed source :
   (a) MPL (Mozilla Public License),
   (b) LGPL (GNU Lesser General Public License).
3. Les licences n'autorisant aucune concession par rapport au caractère open source ou à la combinaison avec du code closed source :
   (a) GPL (GNU General Public License).

Annexe D

Doxygen

La documentation qui suit est un extrait d'un tutorial sur Doxygen traduit pour l'occasion de l'anglais[1].

A moins que vous n'utilisiez déjà un style de commentaires que Doxygen comprend, la documentation de votre code sera incompréhensible par l'outil. Aussi, que faire ?

D.1 Pourquoi un système de documentation automatique ?

Il existe de nombreuses solutions pour documenter un code, mais la documentation dite automatique représente plusieurs avantages.

- La documentation est toujours à jour : il est plus facile de modifier un commentaire dans le code, que de lancer votre traitement de texte pour changer un document,
- Réutilisation de vos commentaires,
- Formatage automatique : avec de simples balises, vous pouvez créer une documentation au look professionnel, avec une gestion efficace des liens hypertextes,
- Le fait de mettre les commentaires dans le code augmente la facilité de maintenance.

D.2 Pourquoi utiliser doxygen ?

Doxygen est un outil très répandu, surtout chez les développeurs habitués aux outils gratuits de type Open Source.

- C'est un logiciel OpenSource, gratuit,
- Il est multi-plates-formes,
- La configuration est étendue et permet de réaliser des documents dans de nombreux formats.

[1] http ://www.codeproject.com/tips/doxysetup.asp

D.3 Commentaires avant les classes et structures

Il suffit d'utiliser « /// » pour le bloc de commentaires, comme dans D.1.

```
/// SNAPINFO is a structure ...
///
struct SNAPINFO
{
   // ...
}
```

Fig. D.1 – Doxygen : commentaire de structure ou classe

D.4 Commentaire succinct dans la déclaration des membres

Utiliser « /// » dans le cas où il n'y a qu'une seule ligne avant le membre. Sinon, « ///< » permet de commenter un membre sur la même ligne.

```
struct SNAPINFO
{
    /// Init, using a pre-change and a post-change rectangle
    void Init(RECT const \& oldRect, RECT const \& newRect, DWORD snapwidth);

    void SnapHLine(int y); ///< Snap to a horizontal line
}
```

Fig. D.2 – Doxygen : commentaires sur les membres

D.5 Ajouter une description détaillée sur l'implémentation d'une méthode

Comme pour les classes, /// vous permet de faire un bloc de commentaires (figure D.3).

```
/// Initializes the structure with an old an a new rectangle.
/// use this method if the window is about to be moved or resized (e.g. in
void Init(RECT const & oldRect, RECT const & newRect, DWORD snapwidth) {
   // ...
}
```

Fig. D.3 – Doxygen : commentaires sur une méthode

Annexe E

Arborescence du projet (CVS)

Afin de récupérer les sources du serveur de cartes, il faut procéder de la façon suivante :

- Authentification : cvs -d :pserver :votrelogin@indigo.inrialpes.fr :/sharp/repository login Mettre votre mot de passe habituel.
- Récupération : cvs -z3 -d :pserver :votrelogin@indigo.inrialpes.fr :/sharp/repository co -P parkview

Ceci créera un répertoire *parkview* avec toute l'arborescence du projet, que vous retrouvez détaillée dans la figure E.1.

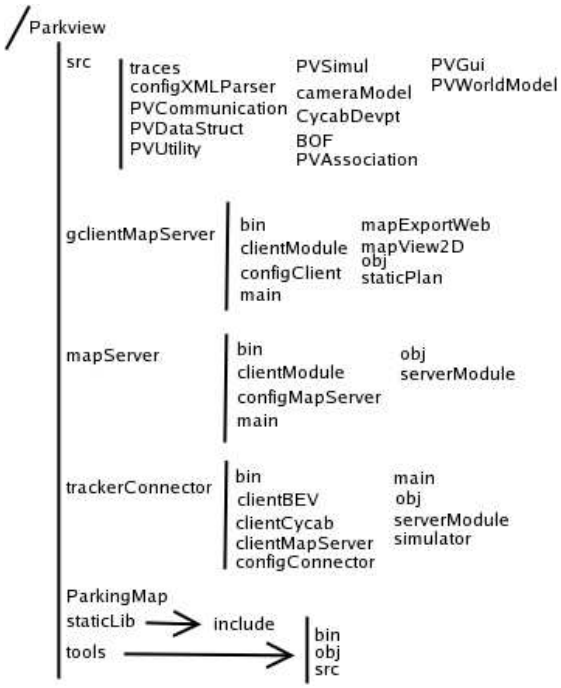

Fig. E.1 – L'arborescence du projet

Bien que les répertoires des différents modules (trackerConnector, mapServer, gclient) soient tous dans la même arborescence, il s'agit bien de programmes différents, avec au final un binaire par module.

Explications

Le répertoire *src* contient les différents modules communs ou bien externes aux différents programmes. *ParkingMap*, *staticLib* et *tools* contiennent différents fichiers ou programmes nécessaires à la compilation (par exemple, *staticLib* inclut la librairie ProBT© pour le module jPDA.

gclientMapServer est le répertoire du client graphique incluant le binaire *bin*, les parties spécifiques du client réseau et de l'outil de configuration, ainsi que les codes pour la gestion de l'affichage.

mapServer contient toute l'arborescence du serveur de cartes. Nous retrouvons le répertoire du binaire *bin*, ainsi que les parties spécifiques des modules.

trackerConnector est le répertoire du module d'interface pour les trackers et le Cycab.

Annexe F

Tableau de bord de gestion du stage

Le stage s'est déroulé avec une autonomie très importante. Cependant, afin de faire le suivi des différentes tâches, j'ai mis en place un document de gestion par lots du projet.

La figure F.4 représente le calendrier avec les différentes actions/tâches qui ont été accomplies.

A noter que deux mois ont été consacrés à une partie DEA. A ce moment, il était prévu une poursuite d'étude, mais pour des raisons personnelles ceci n'a pas pu ce faire, d'où l'abandon de cette partie.

Le tableau F.1 quant à lui présente un exemple de lot, en l'occurrence le client graphique.

N°	Description	Type	Qui	Valideur	Charge
7	Lot N° 7 : Client graphique				
7.1	Cahier des charges	Documents	EBO	TFR	0.5
7.2	Spécifications	Documents	EBO	TFR	2
7.3	Réalisation	Devpt	EBO	TFR	
	Investigation moteur graphique	Devpt	EBO		3
	auto-formation OpenGL	Devpt	EBO		5
	Client eNET	Devpt	EBO		3
	Gestion graphique GL	Devpt	EBO		15
	Refonte client	Devpt	EBO		10
7.4	Doxygen	Documents	EBO		3

Tab. F.1 – Exemple de lot : le client graphique

Une fois les différents lots créés et les charges calculées, il a été possible de créer les tâches concernées dans un logiciel Open Source de planification répondant au nom de ganttProject[1]. Les figures F.1 et F.2 montrent respectivement le diagramme de Gantt pour les premières étapes du stage et pour le développement du client graphique.

[1]http ://ganttproject.sourceforge.net. Dernière consultation : 2-jan-06.

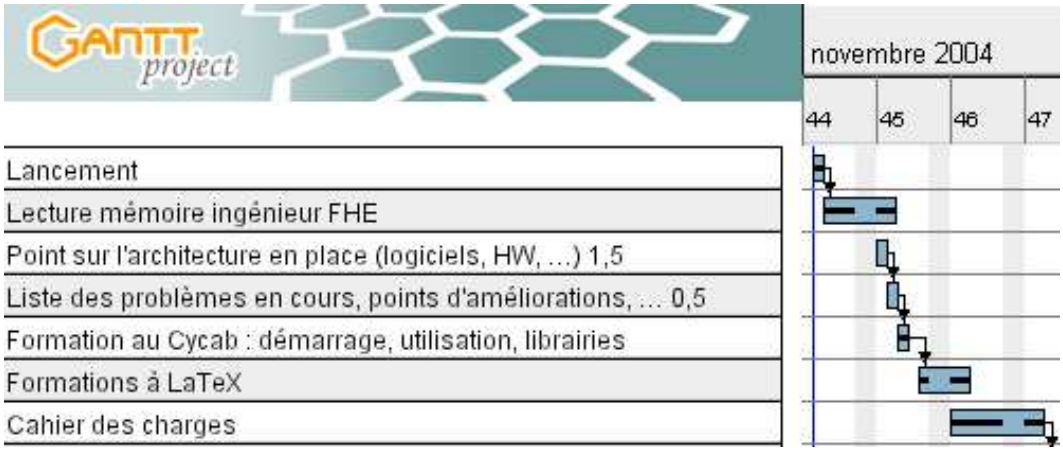

Fig. F.1 – Diagramme de Gantt pour le démarrage du stage

Fig. F.2 – Diagramme de Gantt pour le développement du client

J'ai utilisé aussi un outil de « mind mapping »(freemind), qui permet de gérer facilement ses idées, voire d'en faire le suivi. La figure F.3 est un exemple pour l'année d'étude, avec un découpage en 5 parties.

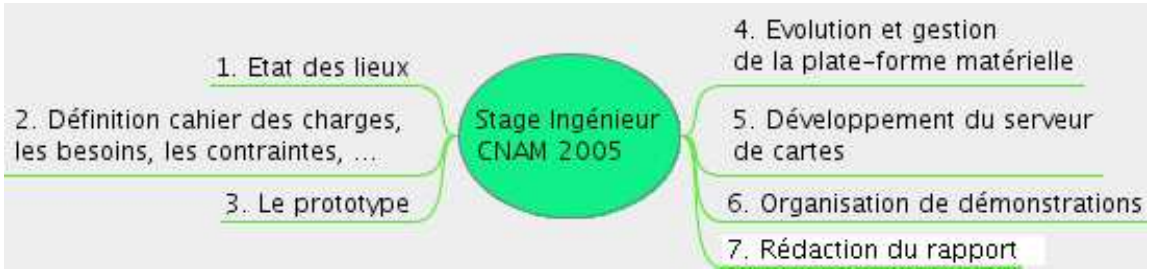

Fig. F.3 – Découpage du stage (outil de MindMapping)

Enfin, la modélisation objet a été facilitée grâve à l'outil *Umbrello*[2].

[2]http ://uml.sourceforge.net. Dernière consultation : 20-dec-05.

November		December		January		February	
1 L		1 M	Evaluation tech. des trackers	1 S		1 M	Etude eNet
2 M	Lancement	2 J		2 D		2 M	
3 M	Etat des lieux	3 V		3 L	Prototype	3 J	
4 J		4 S		4 M		4 V	
5 V		5 D		5 M	Test client Cycab	5 S	
6 S		6 L	Evaluation tech. des trackers	6 J		6 D	
7 D		7 M		7 V		7 L	Tracker Connector / eNet / Démo
8 L	Etat des lieux	8 M		8 S		8 M	
9 M		9 J		9 D		9 M	
10 M		10 V		10 L	Test client Cycab	10 J	
11 J		11 S		11 M		11 V	
12 V	Etat des lie	12 D		12 M		12 S	
13 S		13 L	Choix du tracker	13 J		13 D	
14 D		14 M		14 V		14 L	Tracker Connector / Serveur de cartes
15 L	Etat des lieux	15 M	Le prototype	15 S		15 M	
16 M		16 J		16 D		16 M	
17 M		17 V		17 L	Réglage affichage 2D / Démo	17 J	
18 J	Cahier des charges	18 S		18 M		18 V	
19 V		19 D		19 M		19 S	
20 S		20 L	Le prototype	20 J		20 D	
21 D		21 M		21 V		21 L	Tracker Connector / mapServer / Démo
22 L	Cahier des charges	22 M		22 S		22 M	
23 M		23 J		23 D		23 M	
24 M		24 V		24 L	Etude Serveur de Cartes	24 J	
25 J		25 S		25 M		25 V	
26 V		26 D		26 M		26 S	
27 S		27 L	Le prototype	27 J		27 D	
28 D		28 M		28 V		28 L	Serveur de cartes
29 L	Spec	29 M		29 S			
30 M		30 J		30 D			
		31 V		31 L	Fusion		

Fig. F.4 – Tableau de suivi

Bibliographie

[ADFF04] Pierre-Yves Aimon, Nathalie Dumas, Laurent Fallet et Samy Fouilleux : WIFI : Etude théorique & projet ROBI. Mémoire de D.E.A., INSA, http ://asi.insa-rouen.fr/ lfallet/docs/sem3/ROBI_WiFi.pdf, 2 janvier 2004.

[Apa04] Apache Software Foundation : Xerces-C++. Internet, http ://xml.apache.org/xerces-c/, avril 2004. Dernière consultation : 20 juin 2005.

[BGMPG99] G. Baille, Ph. Garnier, H. Mathieu et R. Pissard-Gibollet : Le cycab de l'inria Rhône-Alpes. Rapport technique 229, Institut National de la Recherche en Informatique et en Automatique, Montbonnot (FR), avril 1999.

[Bra03] Christophe Braillon : Evitement d'obstacles et suivi de trajectoires. Rapport technique, INRIA Rhône-Alpes, 26 septembre 2003.

[CH05] Chong Chin Hui : Technical report on cycab localization. Technical report, Institut National de la Recherche en Informatique et en Automatique, Montbonnot (FR), 2005.

[Cou03] Christophe Coué : *Modèle bayésien pour l'analyse multimodale d'environnements dynamiques et encombré : application à l'assistance à la conduite en milieu urbain.* Thèse de doctorat, INPG, http ://www.inrialpes.fr/sharp/people/coue/Publis/coue :these.pdf, décembre 2003.

[DWDA03] Dirk Schulz, Wolfram Burgard, Dieter Fox et Armin B. Cremers : People Tracking with a Mobile Robot Using Sample-based Joint Probabilistic Data Association Filters. *IJRR*, février 2003.

[Fra02] Thierry Fraichard : Réponse à l'appel à proposition Robea 2002. http ://emotion.inrialpes.fr/parknav/proposition.pdf, 11 juin 2002. Dossier complêt.

[Fra03] Thierry Fraichard : Rapport d'activité ParkNav à un an. Rapport technique, INRIA, http ://emotion.inrialpes.fr/parknav/parknav-ra-oct-03.pdf, octobre 2003.

[Fra04] Thierry Fraichard : Rapport d'activité du projet ParkNav à deux ans. Rapport technique, INRIA, http ://emotion.inrialpes.fr/parknav/parknav-ra-oct-04.pdf, octobre 2004.

[GNU05] GNU.org : Common C++. Internet, http ://www.gnu.org/software/commoncpp/, mars 2005. Dernière consultation : 5/07/2005.

[Hel03] F. Helin : Développement de la plate-forme expérimentale Parkview pour la reconstruction de l'environnement dynamique. Mémoire de fin d'études, Conservatoire Nat. des Arts et Métiers, Grenoble (FR), 2003.

[Her03a] J. Hermosillo : *Planification et exécution de mouvements pour un robot bi-guidable : une approche basée sur la platitude différentielle.* Thèse de doctorat, Inst. Nat. Polytechnique de Grenoble, Grenoble (FR), June 2003.

[Her03b] HERVÉ MATTHIEU : Linux temps réel pour le contrôle des robots de l'INRIA Rhône-Alpes. 4èmes Journées de Grenoble Linux Embarqué et Temps Réel, 27 novembre 2003.

[INR98] INRIA ROCQUANCOURT : Fiche technique du Cycab. Rapport technique, INRIA, http ://www-rocq.inria.fr/praxitele/fiche-tech.html, octobre 1998.

[Int03] INTEL : OpenCV Library Reference manual. Internet, http ://sourceforge.net/project/showfiles.php?group_id=22870&package_id=16948. Dernière Consultation : 15/07/2005., 2003.

[JVC01] JVC : *Coulour Video Camera TK-C1481.* Victor Compagny of Japan, Limited, Japan, 2001. Instructions Book.

[Kni04] Steven KNIGHT : SCons User Guide 0.96. Internet, http ://www.scons.org/doc/HTML/scons-user/book1.html, août 2004. Dernière consultation : 23/06/05.

[LZ99] Augustin LUX et Bruno ZOPPIS : Gestion mémoire multi-stratégie pour une plate-forme multi-languages. Rapport technique, INRIA, http ://pauillac.inria.fr/ weis/j-fla99/ps/zoppis.ps, février 1999.

[Mat03] Hervé MATHIEU : Linux Temps Réel pour le contrôle des robots de l'INRIA Rhône-Alpes. Rapport technique, INRIA, http ://www.irisa.fr/Cycab/Presentation/Presentation.html, novembre 2003.

[Pra01] Cédric PRADALIER : Conception d'un système de localisation pour un robot mobile : utilisation d'un télémètre laser et placement d'amers dans l'environnement. Mémoire de D.E.A., INPG, juin 2001. DEA.

[Pra04] Cédric PRADALIER : *Navigation intentionnelle d'un robot mobile.* Thèse de doctorat, INPG, septembre 2004.

[PS02] C. PRADALIER et S. SEKHAVAT : Concurrent matching, localization and map building using invariant features. septembre-octobre 2002.

[_S03] _SEBF : Entête UDP. Internet, http ://www.frameip.com/entete-udp/, 16 novembre 2003.

[_S04] _SEBF : Entête TCP. Internet, http ://www.frameip.com/entetetcp/, 12 mars 2004. Dernière consultation : 20/06/05.

[Sil98] Nino SILVERIO : *Langage C++.* Eyrolles, février 1998. Programmation et exercices corrigés.

[Str97] Bjarne STROUSTRUP : *The C++ programming language (3th ed.).* Addison-Wesley Longman Publishing Co., Inc., 1997.

[Vei05] Daniel VEILLARD : The XML C parser and toolkit of Gnome. Internet, http ://www.xmlsoft.org/, avril 2005. Dernière consultation : 28/06/2005.

[W3C05] W3C : Introduction to XML Schema. Internet, http ://www.w3schools.com/schema/schema_intro.asp, 2005. Dernière consultation : 6/07/2005.

[WB04] Greg WELCH et Gary BISHOP : An Introduction to the Kalman Filter. Rapport technique, University of North Carolina, http ://www.cs.unc.edu/ welch/kalman/index.html, 1 mars 2004.

[WJL03] Y. Wang, X. Jia et H.K. Lee : An indoors wireless positioning system based on wireless local area network infrastructure. Rapport technique, The University of New South Wales Sydney, Australie., http ://www.gmat.unsw.edu.au/snap/publications/wangy_etal2003a.pdf, juillet 2003.

[WNDS03] Mason Woo, Jackie Neider, Tom Davis et Dave Shreiner : *OpenGL 1.2 Guide officiel.* 27 janvier 2003. Le guide officiel pour l'apprentissage et la maîtrise d'OpenGL 1.2.

[YoL05] YoLinux.com : XML and Gnome LibXML2. Internet, http ://www.yolinux.com/TUTORIALS/GnomeLibXml2.html, mars 2005. Dernière consultation : 27/06/2005.

NOM : Boniface
Prénom : Eric

Sujet : Gestion d'une plate-forme de vidéosurveillance à usage robotique. Modélisation d'un environnement dynamique.

Mémoire d'ingénieur C.N.A.M., Grenoble
soutenu le 30 mars 2006

Parkview est une infrastructure expérimentale créée pour les besoins du projet ParkNav lors du stage de Frédérique Helin, courant 2003. L'objectif est de fournir une plate-forme ouverte, utilisant un serveur de cartes logiciel capable de modéliser l'environnement en temps réel. Les applications utilisatrices de ce serveur vont des tests de modules expérimentaux, jusqu'à la conduite automatique (non couverte dans cette étude).

Avant d'aborder le détail de l'existant et des travaux réalisés, nous introduisons le contexte complet de ce mémoire d'ingénieur CNAM. L'INRIA puis l'équipe e-Motion sont présentées afin de voir le contexte organisationnel. Puis, le projet à l'origine de la création de la plate-forme est abordé, ainsi que Parkview elle-même et le serveur de cartes.

Parknav et les utilisations potentielles de Parkview induisent des besoins ou des contraintes particuliers : traitement en temps réel, flux vidéo multi-caméras, dynamique des scènes, ... Nous avons décrit ces différents besoins et contraintes - logicielles et matérielles - qui conditionnent les développements réalisés. Depuis sa création, la plate-forme a évolué par l'ajout de matériel, l'évolution des logiciels des partenaires du projet et la réalisation de programmes de tests. L'une des premières étapes du stage fut de réaliser l'inventaire de l'existant et le point sur les problèmes en cours.

Dans cet inventaire, un acteur important est la voiture électrique, le Cycab. Véritable robot roulant, il est possible de lui faire exécuter des ordres en mode automatique ou bien en mode manuel. Equipée de capteurs tel qu'un laser Sick ou bien un gyroscope, elle nous retourne de manière très précise sa position relative, voire absolue via des algorithmes de conversion. C'est à la fois un fournisseur et un client de l'architecture développée.

Une fois cet inventaire effectué, nous nous sommes penchés sur la proposition d'une nouvelle architecture que nous avons décrite avec ses différents éléments. Le principe étant d'expliquer ce que nous souhaitons faire - autrement dit le problème à résoudre - et la solution proposée. Cette présentation de notre contribution est l'occasion d'introduire un ensemble de termes ou de principes qui seront abordés dans la suite du mémoire. Les travaux sur la plate-forme matérielle sont dans un premier temps étudiés, pour ensuite rentrer dans le détail du développement du serveur de cartes.

Le matériel évolue au cours du temps, de nouveaux capteurs sont commercialisés et le matériel en place s'use. Aussi, plusieurs actions ont du être prises pour maintenir l'infrastructure matérielle à jour et en état. Différents problèmes techniques - tels que les difficultés de transmission des caméras sans fil - nous ont conduit à revoir l'installation en place. Notre nouvelle architecture nécessite aussi de revoir le parc des machines afin de pouvoir faire fonctionner les programmes correctement.

Le serveur de cartes est au cœur de l'architecture logicielle, il a pour rôle de modéliser l'environnement en intégrant les différents éléments statiques, mobiles, voire semi-statiques. Il a été découpé en trois grands composants : le *trackerConnector* qui permet d'interfacer un logiciel extérieur avec notre architecture, le *mapServer* qui réalise la modélisation de l'environnement en fonction des données reçues des *trackerConnector*. Et enfin le client graphique qui permet de visualiser le résultat des différents traitements sur les données reçues. Le développement de ce serveur de cartes a nécessité un réel travail d'équipe pour aboutir à un ensemble fonctionnel et efficace.

Mots clés : Environnement dynamique - Détection et suivi de cibles - Transformations Homographiques - Distorsion optique - Cycab - Multi-capteurs et fusion - Serveur de cartes - Temps réel - Client/serveur.

Keywords : Dynamic scene - Target detection and tracking - Homography - Optical Distortion - Cycab - Multiple sensors, data merging - Map server - Real time - Client/server model.