|  |
| --- |
| Bouve Morgan  Delbergue Julien  Jacquel Olivier |

|  |
| --- |
| Projet robotique ISEN-Lille 2010/2011 |
| Partie Vision |
| Projet robotique |

Sommaire

[Sommaire 1](#_Toc285029940)

[Introduction 2](#_Toc285029941)

[1/ Choix du cadre de travail 3](#_Toc285029942)

[2/ Stéréovision et Calibrations 4](#_Toc285029943)

[3/ Approches possibles pour l'évitement d'obstacles par vision 5](#_Toc285029944)

[A) Détection et mise en correspondance de points d'intérêt 5](#_Toc285029945)

[B) Détection du sol 7](#_Toc285029946)

[4/ Le programme 8](#_Toc285029947)

[A) Conception du module 8](#_Toc285029948)

[B) Architecture prévue 9](#_Toc285029949)

[C) Architecture réelle 10](#_Toc285029950)

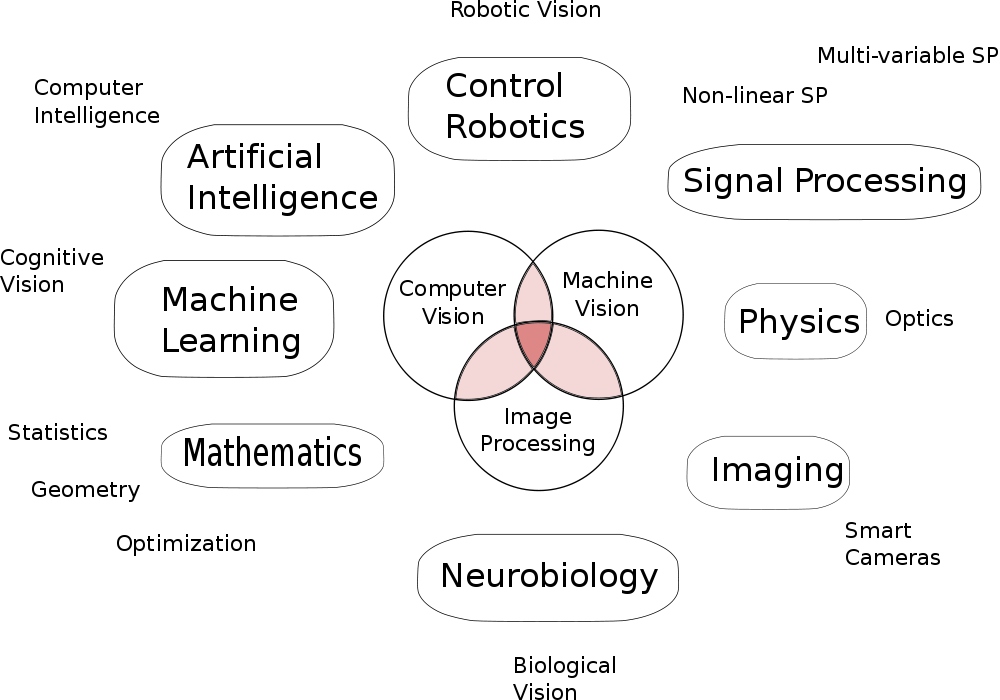
[D) Déroulement du programme 11](#_Toc285029951)

[Z) Compléments 11](#_Toc285029952)

[*Compilation* 11](#_Toc285029953)

[Conclusion 12](#_Toc285029954)

Introduction



La vision par ordinateur est un domaine très vaste de la science, qui fait intervenir de nombreux domaines connexes. La vision en robotique n'est qu'une application de cette science, même s'il elle fait intervenir de multiples compétences, notamment par les contraintes de systèmes embarqués qu'elle apporte. La vision peut par ailleurs faire intervenir une ou plusieurs sources vidéo, comme dans le cas de la stéréovision que nous avons utilisée.

Pour faire de la vision en robotique, il est nécessaire de posséder des notions en traitement d'images. A ce titre les TPs suivis en début d'année, voire l'année dernière, nous ont permis d'acquérir les connaissances nécessaires pour aborder la stéréovision pour notre application d'évitement d'obstacles.

1/ Choix du cadre de travail

Il faut ensuite disposer des outils adéquats, à savoir un framework logiciel de vision, qui nous permet de ne pas partir de 0. Il existe plusieurs bibliothèques de vision, parmi lesquels on peut citer OpenCV, VXL ou encore LTI. Cependant OpenCV est aujourd'hui la plus populaire, ceci notamment grâce à ses meilleures performances. (Un comparatif est réalisé ici : <http://www.aishack.in/2010/07/opencv-vs-vxl-vs-lti-performance-test/>). Nous avons donc pris OpenCV qui semble le choix le plus judicieux.

Ensuite il a fallu choisir le langage dans lequel nous voulions utiliser OpenCV, celui-ci proposant en effet des API C, C++ (et Python). Avant de choisir, il faut d'abord s'assurer que le système cible supporte bien la compilation en C++. Dans notre cas, qu'il s'agisse d'une carte-mère de plusieurs centaines de Mhz avec un OS, ou directement d'un pc portable, le système supporte sans problème le C++. Reste maintenant à faire le meilleur choix entre C et C++. Ayant fait le programme principal du robot en C++, nous avons décidé de continuer avec ce langage. L'autre avantage du C++ est sa modularité et sa réutilisabilité. Ainsi nous gagnons du temps de développement, et il sera normalement possible de réutiliser notre travail pour les projets suivants. Néanmoins, il faut être conscient que le C++ est plus consommateur de ressources que le C, or nous somme dans l'embarqué, cela peut donc être un élément critique. Dans le pire des cas, il est toujours possible à posteriori d'utiliser du C au sein du programme prévu pour le C++, l'inverse n'étant pas vrai. Mais après plusieurs recherches, il est quand même possible de diminuer l'impact de cette baisse de performances due au langage sur le système en utilisant intelligemment le C++. Ainsi quelques règles sont à suivre:

* Éviter de trop utiliser les exceptions
* Réduire les créations/destructions d'objets tant que possible
* Préférer l'utilisation d'objets statiques, c'est-à-dire définis lors de la compilation, consommant donc moins de ressources à l'exécution
* Éviter l'héritage multiple
* En règle générale, faire attention à la gestion de la mémoire, et aux opérations spécifiques au C++ qui requiert en arrière-plan des allocations mémoires plus importantes que du C.

Enfin nous avons adopté la dernière version d'OpenCV sortie à ce jour, la 2.2. Un tutoriel ci-joint a été créé pour la compilation et l'installation des sources sur une distribution Ubuntu (fonctionne avec Ubuntu 10.04, 10.10, non testé avec les versions antérieures)

2/ Stéréovision et Calibrations

Pour faire de l'évitement d'obstacles, il est avant tout intéressant de pouvoir récupérer les informations de distances environnantes, comme le permettent les capteurs classiques. Il existe plusieurs solutions techniques à ce problème, mais la plus efficace, avec des caméras RGB traditionnelles, est la stéréovision. Pour cela, il faut disposer de 2 caméras observant la même zone, sous des angles un peu différents. Ainsi on va pouvoir déterminer les coordonnées 3D des points observés grâce à la triangulation stéréo, mécanisme qui utilise les propriétés de la géométrie épipolaire. Cependant pour se faire, il est nécessaire de pouvoir connaître le matériel utilisé, et notamment être capable de caractériser les imperfections des caméras. Cela permet aussi de rectifier numériquement les images en provenance des caméras. Il faut de même pouvoir caractériser les positions relatives des 2 caméras. (Le chapitre 11 du livre *Learning OpenCV* fourni par M. Palos couvre très bien cette section). Aussi on souhaite pouvoir utiliser ces informations sous forme de matrices pour les différents algorithmes utilisés en traitement d'images. Une fois déterminées, ces matrices ne changent plus et sont sauvegardées pour pouvoir être directement réutilisées par la suite.

**MORGAN**

A) Calibration de chacune des caméras

procédure, TP2....

B) Calibration du système: stéréocalibration

3/ Approches possibles pour l'évitement d'obstacles par vision

A) Détection et mise en correspondance de points d'intérêt

Les 2 caméras forment donc un système permettant la stéréo triangulation. Comme vu en TP, il est donc maintenant possible de déterminer la distance séparant les caméras d'un point situé dans le champ de vision de celles-ci, et donc d'en obtenir ses coordonnées 3D. C'est une approche qui permet la détection des obstacles, et peut être couplée avec d'autres pour fournir plus de précision sur un point spécifié. Or pour cela, il faut sélectionner le point qui nous intéresse dans une image, puis sélectionner son équivalent dans l'image fournie par l'autre caméra. Ce processus est assez simple lorsqu'un humain doit choisir visuellement un unique point sur une image fixe, mais se complique radicalement lorsqu'il faut prendre en compte plusieurs dizaines de points d'intérêt sur chacune des images d'une source vidéo. Dans une optique de systèmes embarqués, il n'y a par ailleurs pas d'interface utilisateur directe pour se faire, il est donc nécessaire d'automatiser cette sélection des points d'intérêt. A cela s'ajoute un problème de mise en correspondance entre les points détectés sur l'image gauche et ceux détectés sur l'image de droite.

Il existe plusieurs algorithmes conçus pour cela, on peut citer notamment le détecteurs d'Harris, SURF ou encore SIFT, sûrement le plus performant à l'heure actuelle. Pourtant celui-ci, bien que très efficace, n'est pas du tout adapté à l'embarqué car il nécessite un temps de calcul beaucoup trop important. Il faut donc trouver un bon compromis entre rapidité et fiabilité. Cela passe par une longue mais importante phase de tests pour trouver le plus adapté à l'application. Afin de simplifier ces changements de détecteurs de points d'intérêt, OpenCV propose justement un système de classes abstraites. Le framework propose ainsi une décomposition en 3 étapes:

* détection de composantes d'intérêt
* extraction de descripteurs de ces composantes
* mise en correspondance des descripteurs

Pour chaque étape, plusieurs algorithmes sont ensuite proposés, et il est possible d'en créer en héritant de la classe générique associée à chaque étape. Ci-dessous un schéma de l'implémentation de ces étapes dans OpenCV 2.2 :

"SIFT"

"SURF"

"BRIEF"

Descriptor extractor

Descriptor matcher

Feature detector

"FAST"

"STAR"

"SIFT"

"SURF"

"MSER"

"GoodFeaturesToTrack"

"HARRIS"

"BruteForce"

"BruteForce-L1"

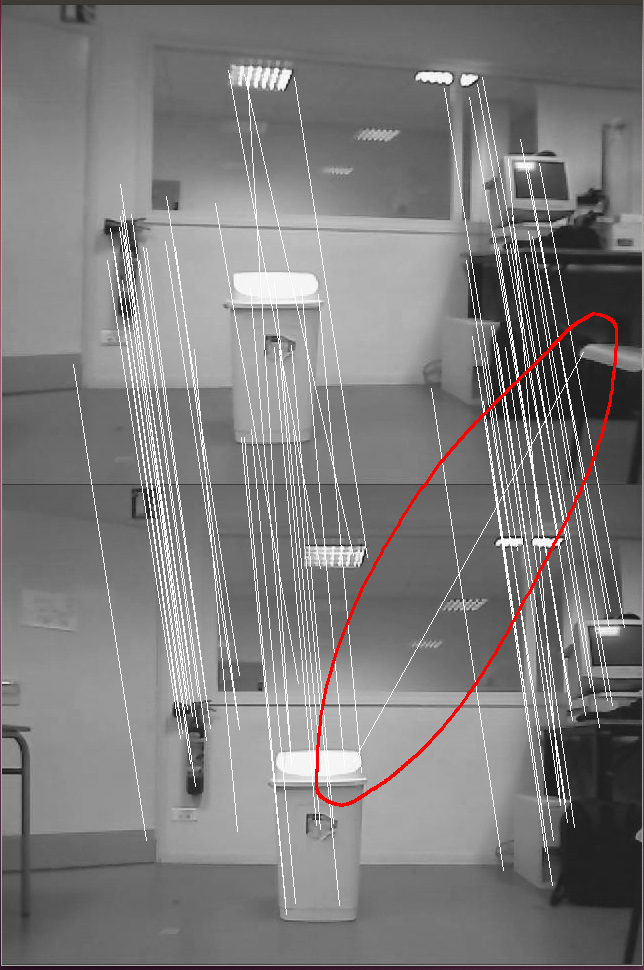
"BruteForce-Hamming"

"BruteForce-HammingLUT"

"FlannBased"

Nous n'avons pas vraiment approfondi cette possibilité de changer de détecteur facilement, en partie parce que nous ne trouvions pas le moyen de changer les paramètres de chaque algorithme avec cette structure très modulaire. Nous avons ensuite préféré nous concentrer sur d'autres approches qui semblaient prometteuses pour la détection d'obstacles, quitte à les coupler par la suite avec un détecteur de points d'intérêt simplifié, pour de meilleurs résultats. Mais il serait utile de se repencher sur la question plus en avant, car cela permettrait sans doute de bon résultats et un moyen efficace pour faire de nombreux tests.

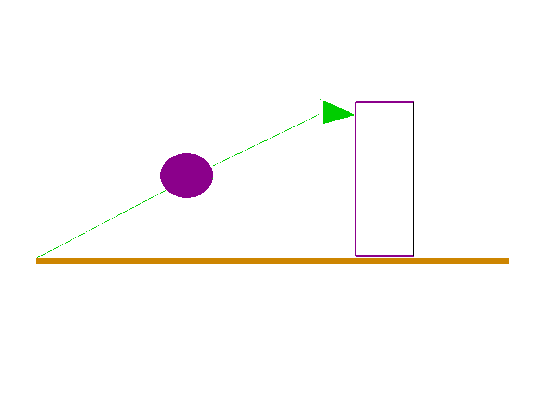
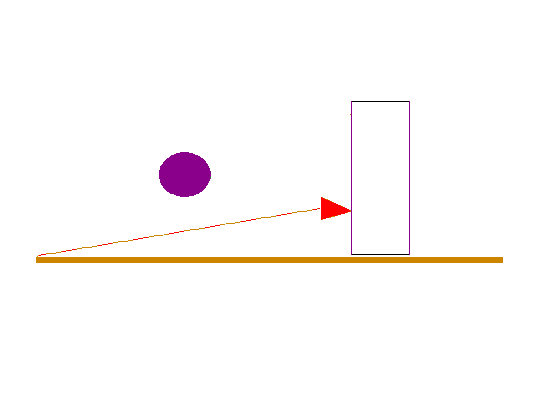
Nous avons tout de même réalisé plusieurs tests, notamment en utilisant un code réduit implémentant le SURF (joint en Annexes). Bien qu'exécuté sur un pc portable, on obtenait un temps de latence assez important (de l'ordre de plusieurs secondes de décalage entre l'image traitée et la réalité). Des réglages sont sans doute à apporter sur les paramètres de détection, mais cela risque de ne pas résoudre le problème, tout du moins pas sans influer sur la qualité du résultat. Par ailleurs, dans la plupart des tests, l'étape de mise en relation produit de mauvaises associations de points. Il faut alors établir un filtre qui évite ces erreurs, en comparant par exemple les positions horizontales/verticales relatives aux points associés. Si un décalage trop important est constaté entre le point de l'image droite et son supposé équivalent dans l'image gauche (au-dessus d'un certain seuil à établir), alors il ne s'agit sûrement pas du même point, et il faut l'ignorer.



Exemple de mauvaise association (SURF)

B) Détection du sol

Une autre piste explorée est la détection et la segmentation du sol. En effet, on peut considérer par simplification que tout ce qui n'appartient pas au sol, zone que le robot peut pratiquer sans crainte, est un obstacle. Ainsi les murs, les obstacles posés au sol, et même les obstacles en hauteur qui entrent dans le champ de vision du robot et obstruent une partie du sol visible seront pris en compte. Il est donc important dans cette approche que les caméras soient situées au sommet du robot.



Mauvaise position : risque de collision avec l'obstacle

Bonne position de la caméra sur le robot

On peut ensuite utiliser les points d'intérêt sur la base des « obstacles » trouvés pour en déterminer la distance. Il faut au préalable pouvoir extraire le sol des images. On recourt pour ce faire à une méthode de segmentation (couleurs, textures, ligne de partage des eaux, graph-cut...). Nous avons réalisé quelques tests, par exemple avec un algorithme de partage des eaux implémenté dans OpenCV : la fonction *floodFill()*. Celle-ci prend en entrée plusieurs paramètres, notamment les coordonnées d'une « graine » (point de départ et référence servant au remplissage), ainsi que divers seuils. On peut éventuellement ajouter un masque, qui sert à limiter la zone de remplissage de l'algorithme. Dans notre application, on a utilisé comme masque une image binaire obtenue avec un détecteur de contours de type Canny. L'algorithme de Canny utilisé est celui implémenté dans OpenCV qui fournit des résultats très satisfaisants. Néanmoins on fera attention aux artefacts qui peuvent apparaître, comme sur l'image si ce dessous :



Détection de contours par Canny :

on observe une tâche au sol

Cela fonctionne plutôt bien sur les images prises à partir des caméras (image ci-dessous), mais les résultats varient beaucoup d'une image à l'autre, car ils dépendent beaucoup de la graine qui change en permanence.

 Plutôt qu'une approche par point échantillon, il serait sans doute préférable d'utiliser plusieurs graines, ou même une zone échantillon. Aussi il serait peut-être judicieux de regarder d'autres méthodes de segmentation basées sur une région référence (par exemple une segmentation couleur par histogrammes). Pour cela il faudrait utiliser un échantillon du sol, par exemple un trapèze pris en bas de l'image (le plus proche possible du robot), comme sur la figure ci-dessous. Cependant nous n'avons pas poursuivi et n'avons pas implémenté cette segmentation par histogrammes dans le programme.

Zone du sol pouvant être prise comme référence

**OLIVIER**

C) La carte de disparité

calculs plutôt rapides ? Permet de faire (simplement?) de la cartographie 2D en même temps?

4/ Le programme

A) Conception du module

Un module de vision générique peut être modélisé comme ceci :

Commands

Parameters

Images

Videos

Communication Protocols, Files

**External Systems**

**Image Sources**

**Vision Core**

Results

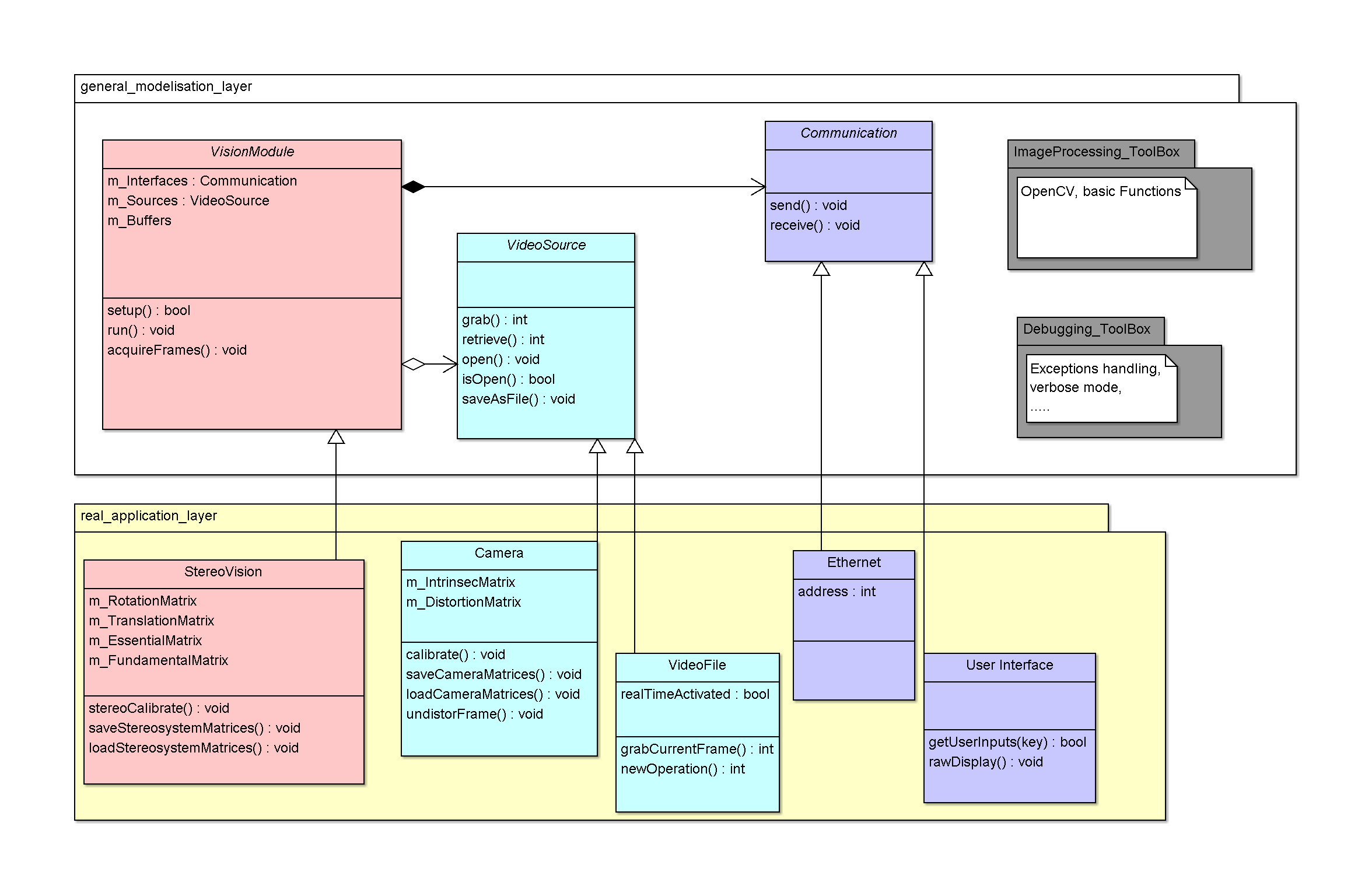
Displays

Communication Protocols, Files

Il est ainsi important de noter que l'affichagee et l'envoi de commandes utilisateur sont des éléments nécessitant une communication entre le module de vision et l'extérieur. Aussi dans notre cas, nous disposons d'un pc monté sur le robot, il n'y a donc pas vraiment de problème. Cependant, dans le cas d'un véritable système embarqué, il n'exsite pas forcément d'interface utilisateur. L'idéal serait donc de pouvoir au choix activer (cas de tests par exemple) ou désactier (cas de fonctionnement réel) cette interface. Ensuite notre application est, pour ce projet, d'éviter les obstacles, mais il faut anticiper des objectifs futurs (détection de personnes, suivi d'un objet, cartographie...), et donc avoir une architecture modulable, flexible et réutilisable tant que possible.

B) Architecture prévue

Nous avons établi un diagramme UML de l'architecture ainsi prévue :



Quelques remarques sur cette modélisation :

* le nombre de sources devrait être variable, car dépent de l'application (monovision, stéréovision, approche multi agents....), de même que le nombre d'interfaces de communication.
* Il est intéressant de pouvoir remplacer dans le programme une caméra par un fichier vidéo, afin de ne pas monopoliser les caméras ou de pouvoir travailler sans prériphériques physiques.
* On peut facilement sauvegarder/charger des matrices de calibration. Ainsi en changeant par exemple les caméras sur le robot, il suffit de recalibrer le tout une seule fois. De même si la position relative des caméras est modifiée, il suffit de relancer une stéréocalibration uniquement.
* L'implémentation d'un protocol Ethernet permettrait de faire communiquer un pc disposant du module de vision avec le programme du robot, même situé sur un autre support matériel (autre pc, carte-mère....)

C) Architecture réelle

Cependant, l'utilisation de classes abstraites implique une plus grande lourdeur et complexité dans la programmation du module. Essentiellement par manque de temps, et aussi de manière à ce que tous les membres du groupe de vision n'ayant pas forcément une bonne connaissance du C++ puissent participer au codage du module, nous avons finalement opté pour une architecture de programme plus simple :

USB

Robot

Camera

Camera

Stereovision

Display

De cette manière on conserve toujours la modularité au niveau de la calibration. De plus, OpenCV propose déjà une classe générique *VideoCapture.* Il suffit donc d'en hériter, on peut alors choisir entre caméra ou fichier vidéo par les constructeurs de la classe (ou par la méthode *open())*. Par ailleurs la communication en USB avec les caméras est transparente pour nous grâce à cette classe.

Il manque cependant une fonctionalité importante au module : un protocol Ethernet pour la communication avec le programme principale du robot. A l'heure actuelle, le module de vision ne renvoit rien au robot.

D) Déroulement du programme

Setup(mode video)

Run

=> diagramme run (disparité comme exemple)

Listes des fonctionalités implémentées dans le programme :

+ On a essayé de respecter les contraintes de l'embarqué, bien qu'on soit directement sur pc. A ce titre il faudra penser à regarder l'utilisation que fait opencv des threads. Apparemment possible de la modifier avec la fonction «

Z) Compléments

### *Compilation*

Le makefile que nous avons utilisé pour le programme de vision est plutôt complet, car il peut servir à la fois pour la compilation manuelle en console, et la compilation directement dans Code Blocks.

Pour cela il faut indiquer à l'IDE quel makefile utiliser :

- Dans l'arborescence des projet de Code Blocks (panel de gauche), faire un clic-droit sur le projet de vision, puis *Properties.*

- Dans l'onglet *Project Settings*, cocher la case « *This is a custom Makefile ».*

*-* Ensuite dans l'onglet *Build targets,* s'assurer que le *« Output filename »* indiqué corresponde bien au nom de l'exécutable spécifié dans le makefile*. (*exemple : *bin/Debug/RobotVisionModule* si le nom de l'exécutable du makefile est *RobotVisionModule)*

Conclusion

Vision n'est à l'heure actuelle pas finie ni parfaite, mais travail bien dégrossi.

L'intérêt d'avoir pensé une telle architecture est qu'elle devrait pouvoir réutilisable pour les années suivantes.

**Annexe:**

**/\***

\* A Demo to OpenCV Implementation of SURF

\* Further Information Refer to "SURF: Speed-Up Robust Feature"

\* Author: Liu Liu

\* [liuliu.1987+opencv@gmail.com](mailto:liuliu.1987+opencv@gmail.com)

\*/

#include <opencv/cv.h>

#include <opencv/highgui.h>

#include <ctype.h>

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include <vector>

using namespace std;

IplImage \*image = 0;

double

compareSURFDescriptors( const float\* d1, const float\* d2, double best, int length )

{

double total\_cost = 0;

assert( length % 4 == 0 );

for( int i = 0; i < length; i += 4 )

{

double t0 = d1[i] - d2[i];

double t1 = d1[i+1] - d2[i+1];

double t2 = d1[i+2] - d2[i+2];

double t3 = d1[i+3] - d2[i+3];

total\_cost += t0\*t0 + t1\*t1 + t2\*t2 + t3\*t3;

if( total\_cost > best )

break;

}

return total\_cost;

}

int

naiveNearestNeighbor( const float\* vec, int laplacian,

const CvSeq\* model\_keypoints,

const CvSeq\* model\_descriptors )

{

int length = (int)(model\_descriptors->elem\_size/sizeof(float));

int i, neighbor = -1;

double d, dist1 = 1e6, dist2 = 1e6;

CvSeqReader reader, kreader;

cvStartReadSeq( model\_keypoints, &kreader, 0 );

cvStartReadSeq( model\_descriptors, &reader, 0 );

for( i = 0; i < model\_descriptors->total; i++ )

{

const CvSURFPoint\* kp = (const CvSURFPoint\*)kreader.ptr;

const float\* mvec = (const float\*)reader.ptr;

CV\_NEXT\_SEQ\_ELEM( kreader.seq->elem\_size, kreader );

CV\_NEXT\_SEQ\_ELEM( reader.seq->elem\_size, reader );

if( laplacian != kp->laplacian )

continue;

d = compareSURFDescriptors( vec, mvec, dist2, length );

if( d < dist1 )

{

dist2 = dist1;

dist1 = d;

neighbor = i;

}

else if ( d < dist2 )

dist2 = d;

}

if ( dist1 < 0.6\*dist2 )

return neighbor;

return -1;

}

void

findPairs( const CvSeq\* objectKeypoints, const CvSeq\* objectDescriptors,

const CvSeq\* imageKeypoints, const CvSeq\* imageDescriptors, vector<int>& ptpairs )

{

int i;

CvSeqReader reader, kreader;

cvStartReadSeq( objectKeypoints, &kreader );

cvStartReadSeq( objectDescriptors, &reader );

ptpairs.clear();

for( i = 0; i < objectDescriptors->total; i++ )

{

const CvSURFPoint\* kp = (const CvSURFPoint\*)kreader.ptr;

const float\* descriptor = (const float\*)reader.ptr;

CV\_NEXT\_SEQ\_ELEM( kreader.seq->elem\_size, kreader );

CV\_NEXT\_SEQ\_ELEM( reader.seq->elem\_size, reader );

int nearest\_neighbor = naiveNearestNeighbor( descriptor, kp->laplacian, imageKeypoints, imageDescriptors );

if( nearest\_neighbor >= 0 )

{

ptpairs.push\_back(i);

ptpairs.push\_back(nearest\_neighbor);

}

}

}

/\* a rough implementation for object location \*/

int

locatePlanarObject( const CvSeq\* objectKeypoints, const CvSeq\* objectDescriptors,

const CvSeq\* imageKeypoints, const CvSeq\* imageDescriptors,

const CvPoint src\_corners[4], CvPoint dst\_corners[4] )

{

double h[9];

CvMat \_h = cvMat(3, 3, CV\_64F, h);

vector<int> ptpairs;

vector<CvPoint2D32f> pt1, pt2;

CvMat \_pt1, \_pt2;

int i, n;

findPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );

n = ptpairs.size()/2;

if( n < 4 )

return 0;

pt1.resize(n);

pt2.resize(n);

for( i = 0; i < n; i++ )

{

pt1[i] = ((CvSURFPoint\*)cvGetSeqElem(objectKeypoints,ptpairs[i\*2]))->pt;

pt2[i] = ((CvSURFPoint\*)cvGetSeqElem(imageKeypoints,ptpairs[i\*2+1]))->pt;

}

\_pt1 = cvMat(1, n, CV\_32FC2, &pt1[0] );

\_pt2 = cvMat(1, n, CV\_32FC2, &pt2[0] );

if( !cvFindHomography( &\_pt1, &\_pt2, &\_h, CV\_RANSAC, 5 ))

return 0;

for( i = 0; i < 4; i++ )

{

double x = src\_corners[i].x, y = src\_corners[i].y;

double Z = 1./(h[6]\*x + h[7]\*y + h[8]);

double X = (h[0]\*x + h[1]\*y + h[2])\*Z;

double Y = (h[3]\*x + h[4]\*y + h[5])\*Z;

dst\_corners[i] = cvPoint(cvRound(X), cvRound(Y));

}

return 1;

}

int main(int argc, char\*\* argv)

{

IplImage\* img1, \* img2, \*grey,\*tmp1,\*tmp2;

char key = ' ';

CvCapture \*capture1=0;

CvCapture \*capture2=0;

// Lecture des webcams

capture1 = cvCaptureFromCAM( 1 );

capture2 = cvCaptureFromCAM( 2 );

if(!cvGrabFrame( capture1 ))

return 20;

CvMemStorage\* storage = cvCreateMemStorage(0);

// cvNamedWindow("Object", 1);

cvNamedWindow("Object Correspond", 1);

static CvScalar colors[] =

{

{{0,0,255}},

{{0,128,255}},

{{0,255,255}},

{{0,255,0}},

{{255,128,0}},

{{255,255,0}},

{{255,0,0}},

{{255,0,255}},

{{255,255,255}}

};

tmp1 = cvQueryFrame( capture1 );

tmp2 = cvQueryFrame( capture2 );

img1 = cvCreateImage( cvGetSize(tmp1), 8, 1 );

//cvCvtColor( tmp1, img1, CV\_RGB2GRAY );

img2 = cvCreateImage( cvGetSize(tmp2), 8, 1 );

//cvCvtColor( tmp2, img2, CV\_RGB2GRAY );

while(key!='q')

{

tmp1 = cvQueryFrame( capture1 );

tmp2 = cvQueryFrame( capture2 );

cvCvtColor( tmp1, img1, CV\_RGB2GRAY );

cvCvtColor( tmp2, img2, CV\_RGB2GRAY );

CvSeq \*objectKeypoints = 0, \*objectDescriptors = 0;

CvSeq \*imageKeypoints = 0, \*imageDescriptors = 0;

int i;

CvSURFParams params = cvSURFParams(500, 1);

double tt = (double)cvGetTickCount();

cvExtractSURF( img1, 0, &objectKeypoints, &objectDescriptors, storage, params );

printf("Object Descriptors: %d\n", objectDescriptors->total);

cvExtractSURF( img2, 0, &imageKeypoints, &imageDescriptors, storage, params );

printf("Image Descriptors: %d\n", imageDescriptors->total);

IplImage\* correspond = cvCreateImage( cvSize(img2->width, img1->height+img2->height), 8, 1 );

cvSetImageROI( correspond, cvRect( 0, 0, img1->width, img1->height ) );

cvCopy( img1, correspond );

cvSetImageROI( correspond, cvRect( 0, img1->height, correspond->width, correspond->height ) );

cvCopy( img2, correspond );

cvResetImageROI( correspond );

vector<int> ptpairs;

findPairs( objectKeypoints, objectDescriptors, imageKeypoints, imageDescriptors, ptpairs );

for( i = 0; i < (int)ptpairs.size(); i += 2 )

{

CvSURFPoint\* r1 = (CvSURFPoint\*)cvGetSeqElem( objectKeypoints, ptpairs[i] );

CvSURFPoint\* r2 = (CvSURFPoint\*)cvGetSeqElem( imageKeypoints, ptpairs[i+1] );

cvLine( correspond, cvPointFrom32f(r1->pt),

cvPoint(cvRound(r2->pt.x), cvRound(r2->pt.y+img1->height)), colors[8] );

}

tt = (double)cvGetTickCount() - tt;

printf( "Extraction time = %gms\n", tt/(cvGetTickFrequency()\*1000.));

cvShowImage( "Object Correspond", correspond );

cvWaitKey(3);

}

// Destruction de la capture

cvReleaseCapture( &capture1 );

cvReleaseCapture( &capture2 );

// cvDestroyWindow("Object");

// cvDestroyWindow("Object SURF");

cvDestroyWindow("Object Correspond");

return 0;

}