

Projet Technologique

VISION STÉRÉOSCOPIQUE

Geoffrey MEILHAN
Mohamed ALAMI
Kenji FONTAINE

27 avril 2017

Table des matières

1	Description du projet	3
2	Domaine : Vision stéréoscopique	3
3	Cahier des charges	3
3.1	Besoins fonctionnels	3
3.2	Besoins non fonctionnels	4
4	Architecture du code : partie QT et OpenCV	5
5	Architecture du code : partie Unity	6
6	Tests	6
7	Problèmes rencontrés	6
8	Conclusion	7

1 Description du projet

La stéréoscopie est un ensemble de techniques visant à créer ou améliorer la perception de relief à partir de deux images planes.

Le projet consiste à développer un module permettant d'évaluer une distance à partir de deux images planes en entrée. Ce projet prendra la forme d'une implémentation du module sur un système mobile à roues. L'objectif final étant de concevoir un robot suiveur, capable de suivre une personne à une distance donnée.

2 Domaine : Vision stéréoscopique

Aujourd'hui devenu peu coûteux et peu encombrant, les systèmes de vision stéréoscopiques sont de plus en plus répandus.

On trouve de nombreux domaines d'application dont les suivants :

- Système de freinage automatique chez les voitures (Toyota par exemple).
- Détection d'obstacle chez les voitures autonomes.
- Reconnaissance d'objets chez les robots.
- La réalité virtuelle

3 Cahier des charges

3.1 Besoins fonctionnels

Cette section décrit les besoins fonctionnels qui doivent être remplis par le module créé.

OpenCV et QT

Cette première partie décrit les besoins fonctionnels relatifs à la partie OpenCV et QT du projet.

Split

Le module doit pouvoir séparer verticalement une image en deux images de même taille.

Détection de bords

Le module doit pouvoir détecter et mettre en valeur les bords d'une image.

Carte de disparité

Le module doit pouvoir calculer une carte de disparité à partir de deux images sources.

Carte de profondeur

Le module doit pouvoir, à partir d'une carte de disparité, calculer une carte de profondeur.

Conversion QImage/cv : :Mat

Le module doit pouvoir convertir une image d'un format à l'autre.

Unity

Cette deuxième partie décrit les besoins fonctionnels relatifs à la partie Unity du projet.

Simulation

L'environnement Unity doit permettre d'obtenir des images stéréoscopiques.

Robot

Carte de profondeur

Le robot doit pouvoir calculer une carte de profondeur à partir de deux images en entrée en utilisant le module créé.

Estimer une distance

Le robot doit pouvoir obtenir une distance à partir d'une carte de profondeur.

Déplacement

Le robot doit pouvoir avancer ou reculer en fonction d'une distance estimée.

3.2 Besoins non fonctionnels

OpenCV et QT

Robuste

Les distances estimées doivent être ordonnées de la même façon que les distances réelles, sous des conditions variables de luminosité, ombres ou mouvements.

Rapide

L'estimation de la distance doit être suffisamment rapide afin de suivre une personne marchant à vitesse normale (5km/h). Par manque de test, la rapidité n'est pas quantifiée.

Unity

Automatisation

L'acquisition d'image de synthèse doit être automatisable. Pour pouvoir prendre plusieurs images d'un seul coup par exemple.

Robot

Rapide

L'estimation de la distance doit être suffisamment rapide afin de suivre une personne marchant à vitesse normale (5km/h). Par manque de test, la rapidité n'est pas quantifiée.

4 Architecture du code : partie QT et OpenCV

convert.cpp convert.h

Ces fichiers permettent de convertir des images stockées sous le format d'openCV (cv : :Mat) en image sous format QT(QImage) et inversement. Il existe deux fonctions principales : **mat2QImage** et **qImage2Mat**.

edit.cpp edit.h

Ces fichiers forment le "core" de notre module. Sont incluent la plupart des algorithmes utilisés lors des calculs de carte de disparité ou de profondeur, de détection de bords, de détections de points d'intérêt, etc...

split : Cette fonction permet de séparer verticalement une image en entrée, en deux images de même taille. Elle prend en entrée une image de type QImage. Cette QImage sera coupée en deux et chaque partie ainsi obtenue sera respectivement stockée dans les cv : :Mat dont les adresse sont données en paramètres.

sobel : Cette fonction permet de détecter les bords d'une image. Elle prend en entrée une image de type cv : :Mat. la fonction va effectuer sur l'image source une détection de bords, puis stockera le résultat ainsi obtenu dans une autre image cv : :Mat dont l'adresse est donnée en tant que paramètre.

surf et surfmatch : Ces deux fonctions permettent d'effectuer une détection de points d'intérêts. La fonction **surf** prend une seule image de type cv : :Mat en entrée. Elle va détecter les points d'intérêt sur l'image en entrée puis les mettre en valeur. Le résultat obtenu sera stockée dans une autre image cv : :Mat dont l'adresse est donnée en tant que paramètre.

La fonction **surfMath** fait la même chose mais avec 2 images en entrée, toujours de type cv : :Mat. Elle va, en plus de la détection de points d'intérêts, effectuer une mise en correspondance des points d'intérêts entre eux. Le résultat obtenu sera stocké dans une cv : :Mat de destination dont l'adresse est en paramètre de la fonction.

dispMap : Cette fonction permet de calculer une carte de disparité à partir de deux images cv : :Mat en entrée. Elle fait appel à l'algorithme **StereoBM** de la bibliothèque **OpenCV**. La carte de disparité ainsi obtenue est stockée dans une image cv : :Mat dont l'adresse est donnée en paramètre.

main.cpp

Il s'agit du code exécuté lors du lancement du programme. On distingue 2 parties. Une première partie avec GUI correspond au code généré par défaut par QT. Elle permet de lancer le programme avec l'interface graphique.

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

Une deuxième partie permet d'exécuter des parties différentes du code en fonction de l'argument rentré lors de l'appel du programme. Si l'appel au programme se fait par la commande :

```
$ ./projet 0
```

Alors le programme va traiter toutes les images dans les sous-dossiers écrits en dur dans le code, dans ce cas précis il récupère les images dans le dossier source/cam et écrit dans le dossier result/cam avec l'appel suivant :

```
for (int j = 0 ; j < 3 ; j++) {  
    for (int i = 0 ; i < 5 ; i++) {  
        std::ostringstream ossI, ossO;  
        ossI << "source/cam" << j << "-dist" << i << ".png"; //path fichier entrant  
        ossO << "result/cam" << j << "-dist" << i << ".png"; //path fichier sortant
```

Le nombre d'images traitée est $i*j$ et est rentré en dur dans le code. Cette partie a été écrite pour faciliter le traitement d'un grand nombre d'images, afin de tester les cartes de profondeur sur diverses situations.

Une troisième partie très similaire à celle décrite plus haut, à la différence que les images en entrée sont au nombre de deux et ont déjà été split. L'implémentation est la même avec la quantité d'images écrite en dur dans le code, tout comme les dossiers contenant les images. L'appel se fait par :

```
$ ./projet 1
```

Enfin un dernier appel au programme peut être fait permettant de tester plusieurs valeurs de SAD et de ... pour un image rentré dans le code. Celà nous a permis de faire des tests sur les valeurs optimales de StereoBM sur une certaine image, avant notre implémentation des trackbars. L'appel se fait par :

```
$ ./projet 2
```

5 Architecture du code : partie Unity

But in the end

6 Tests

- ouvrir une image - détection de bords - split - depth/disp map - unity (scripts) -

I didn't went to Japan

Nous n'avons pas écrit de fonction permettant de tester les différentes fonctionnalités de notre programme, même si on peut assimiler les différents appel à notre programme avec les arguments "0", "1" ou "2" comme des tests pour les fonctionnalités suivantes :

- Ouvrir une image
- Convertir une QImage en cvMat et l'opération inverse
- Split une image en deux images de même résolution afin de les traiter
- Obtenir une carte de disparité
- Ouvrir une nouvelle fenetre

Nous avons cependant testé toutes les fonctionnalités que notre programme est censé contenir dans le cahier des charges.

7 Problèmes rencontrés

Installation OpenCV

Lors des premières séances, nous avons eu des difficultés à installer OpenCV. En effet, utilisant les trois OS majeures (Windows, Mac, Linux), la procédure d'installation n'était pas la même et nous

n'avons pas pu nous aider les uns les autres facilement. Ceci a retardé un peu l'implémentation des fonctions OpenCV, et nous avons même abandonné l'installation sous Windows.

Carte de disparité

Afin de traiter la carte de disparité, nous avons dans un premier temps utiliser l'algorithme StereoBM. L'appel à celui-ci sur les images tests fournis par l'enseignant nous retournait un résultat satisfaisant lorsque nous n'utilisions aucun paramètre. Nous avons donc garder cette implémentation naïve(pas sur pour cet adjectif) de StereoBM.

Lorsque nous en avons du traiter les images issues du robot, les valeurs par défaut de cet algorithme ne nous donnait pas de résultat concluant. Nous avons donc eu besoin d'implémenter une variation des paramètre de StereoBM dans notre code, afin de mieux contrôler le résultat. Après avoir essayer pendant plusieurs séances d'améliorer les cartes de profondeur avec cette méthode, nous avons abandonner au profit de l'utilisation de StereoSGBM. Ces deux algorithmes étant similaire dans leur implémentation, nous n'avons pas eu besoin de beaucoup de temps pour adapter notre programme. Nous avons très rapidement obtenu un meilleur résultat avec beaucoup moins d'effort.

Conversion d'images

Notre conversion d'images a encore des problèmes de source inconnues. En effet, lors de l'appel du split depuis l'interface graphique, les images ne sont pas traitées et la fonction split affiche deux fois l'image de gauche. Cependant le split est le seul moment où nous avons remarqué ce comportement.

8 Conclusion

Grace à ce projet nous avons pu en apprendre plus sur un domaine qui nous était peu connu en la reconnaissance visuelle, ainsi qu'améliorer nos connaissances en C++.