MAUGEY Rémy
De POURQUERY Benjamin
DECOUDRAS Hadrien
BERNARD Jérémi

Rapport de projet technologique

Vison stéréoscopique pour robot

Licence $3^{ème}$ année

27 avril 2017

Chapitre 1

Introduction

L'objectif de ce projet est de développer un outil implémenté sur un robot lui permettant de suivre à un mètre une personne grâce à deux caméras.

Le projet est divisé en deux parties indépendantes, effectuées sur les deux semestres. La première partie a pour but de prendre en main les outils utilisés au cours de l'année, à savoir le langage C++ (à travers Qt) et la bibliothèque OpenCV. Durant cette première partie sera développé un mini programme de traitement d'images.

La deuxième partie a pour objectif de programmer le robot et d'effectuer des simulations grâce au logiciel Unity. En effet, comme le robot n'est pas forcément accessible aux membres du groupe, il est préférable de faire appel à une simulation pour pouvoir tester les algorithmes n'importe quand.

Qt est une bibliothèque logicielle d'interface graphique, nous utiliserons cet outil pour développer l'interface du logicielle.

OpenCV (open computer vision) est un bibliothèque pincipalement d'algorithmes de traitement d'images et de vidéos. Ici elle sera utilisée pour programmer les fonctions du logiciel en rapport avec les images.

1.1 Cahier des charges

Le cahier des charges s'oriente uniquement sur l'objectif du projet, c'est à dire sur la deuxième partie du semestre.

1.1.1 Besoins fonctionnels

Le logiciel doit prendre en entrée deux images au format *PNG* représentant les images gauche et droite de deux caméras. Ces images vont nous permettre de créer une carte de disparité de la scène.
Cette dernière additionnée aux caractéristiques de la caméra va nous permettre de créer une carte de profondeur. Qui sera ensuite utilisée pour permettre au robot de se déplacer.
Il sera cependant nécessaire d'avoir une carte de profondeur de référence de la personne à suivre à travers une phase d'initialisation. Dans ce cas, la première image prise par le robot permettra de créer la première carte de profondeur qui sera considérée comme la référence. Il faut donc que le robot soit bien placé à un mètre de la personne lors du démarrage et que cette dernière soit la seule dans le cadre des caméras.

1.1.2 Besoins non fonctionnels

La carte de disparité est fortement liée aux images d'entrées, de ce fait, les changements de luminosité et d'environnement peuvent poser des problèmes à cause notamment de l'utilisation d'une référence, nécessaire pour reconnaitre la personne.
Le programme doit donc être robuste à ces différents changement.
Le robot doit être aussi capable de suivre la bonne personne même si une autre personne rentre dans le cadre.

Chapitre 2

Théorie et définitions

Comme montré dans la figure 2.1.1, le projet se structure autour d'une notion clef : Pour repérer les jambes d'une personne afin de la suivre, une carte de profondeur doit être calculée. Cette carte est créée grâce à une carte de disparité.

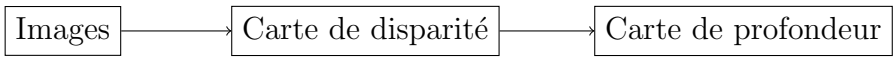

Figure 2.1.1 Fils rouge du projet.

Carte de disparité Un carte de disparité est une image où chaque pixel correspond au déplacement d'un même point dans deux images différentes.

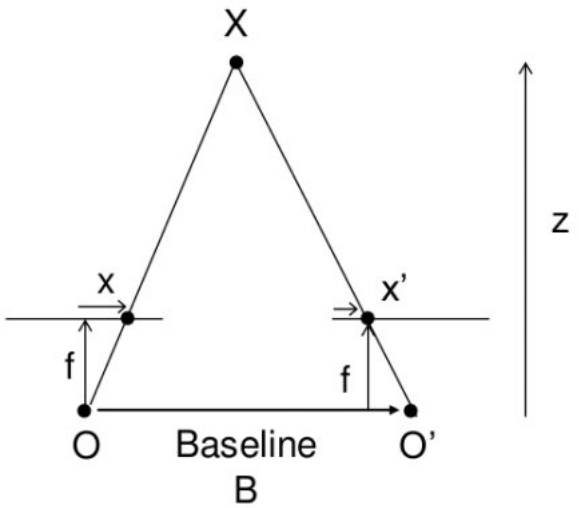

Figure 2.1.2 Schéma du calcul de la carte de disparité. [1]

1. Source : `http://docs.OpenCV.org/3.0-beta/doc/py_tutorials/py_calib3d/py_depthmap/py_depthmap.html`

Comme représentée dans la figure 2.1.2, la valeur du pixel X représente le déplacement du pixel. Les deux images sont représentées par O et O' et la longueur en cm entre elles par la baseline B.

Carte de profondeur La carte de profondeur est une image où chaque pixel représente la distance entre le robot et le point dans le monde réel. Cette distance est exprimable en mètres.
La formule permettant de calculer la carte de profondeur à partir de la carte de disparité est une application du théorème de Thalès.

Figure 2.2.3 Formule de la carte de profondeur.

$$\text{depth} = \frac{\text{baseline} * \text{focal}}{\frac{\text{disp}}{\text{width}} * \text{sensor size}} / \text{référence}$$

Légende :

- baseline : distance entre les centres des deux caméras.
- focal : focal de l'objectif.
- disp : valeur du pixel de la carte de disparité.
- width : largeur de l'image.
- sensor size : taille du capteur.
- référence : distance voulue (dans le cadre de ce projet, en mètres donc 1000).

On peut calculer une carte de profondeur en utilisant la formule ci-dessus sur chaque pixel ou bien utiliser une fonction d'OpenCV :

```
void reprojectImageTo3D(InputArray disp,
                        OutputArray out,
                        InputArray Q,
                        bool handleMissingValues,
                        int ddepth);
```

Problèmes rencontrés L'utilisation de la *reprojectImageTo3D* implique de devoir au préalable calibrer les caméras et rectifier les images, étapes longues et non obligatoires car les caméras fournies sont suffisamment fiables. Donc nous avons décidé de ne pas l'utiliser.

Chapitre 3

Implémentation

3.1 Logiciel Qt

Le premier semestre s'est orienté autour du développement d'un petit programme de traitement d'image. Le programme affiche une fenêtre ainsi que plusieurs menus, l'un d'eux permet d'ouvrir un fichier dans l'un des formats supportés par OpenCV. Le fichier doit nécessairement contenir les deux images juxtaposées pour que notre programme fonctionne. On peut ensuite couper l'image en deux, la flouter grâce à la méthode "blur" d'OpenCV. On peut aussi appliquer à l'image différents algorithmes comme celui de Sobel ou encore celui de Canny tous deux permettent de faire de la détection de contours, présenté sur les figures ci contre.

Figure 3.1.1 Sobel

Figure 3.1.2 Canny

Ces différentes méthodes ont été implémentées dans le but de comprendre

le fonctionnement d'OpenCV, ainsi que de développer et de tester une première esquisse des fonctions que nous utiliserons dans le robot.

Il a fallu créer des méthodes de conversion d'images entre les images de Qt et les images de OpenCV. Deux méthodes ont donc été développées :

```
cv::Mat ImageTools::imageToMat(QImage const& src);
QImage ImageTools::cvMatToImage(const cv::Mat& inMat);
```

Toutes les méthodes en rapport avec le traitement d'image ont été implémentées dans la classe *imagetools*. Au début, nous conservions les images sous forme de QImage ce qui nous obligeait à les convertir à chaque fois que l'on voulait appliquer un filtre. On a fini par réécrire entièrement le programme, où l'on conserve les images sous forme de matrices OpenCV en images membres et la classe *imagetools* sous forme d'un singleton, simplifiant le code.
Le logiciel permet aussi de faire de la reconnaissance de formes grâce aux algorithmes de *flann* et de *surf*. Cependant ce dernier n'est pas open source donc la méthode *flann* est implémenté entre *#define* et n'est pas forcément proposé à l'utilisateur final.
Enfin, le logiciel permet de créer une carte de disparité grâce à l'algorithme *stereosgbm* ou à l'algorithme *stereobm*.

Cette partie nous à permis de prendre en main le C++ ainsi que la bibliothèque OpenCV.

Grâce à ce petit outil, nous avons tester les deux algorithmes permettant de générer des cartes de disparité. Nous avons ainsi retenu *stereosgbm* qui est moins performant mais plus simple à utiliser et crée moins de bruit. Cela nous à permis de choisir des paramètres de *stereosgbm* jugés bons.

Le logiciel Qt prend aussi en charge diverse fonctions propre à Qt et non d'OpenCV tel que le redimensionnement des images, grâce à la méthode cropImage() ou encore la possibilité de zoomer dans l'image.

3.2 Simulation et test

Durant le second semestre, l'objectif était d'implémenter les algorithmes précédent pour les utiliser sur le robot. Le but est de créer une carte de profondeur grâce aux images du robot pour lui permettre de se déplacer en fonction de la personne.
Comme nous n'avons forcément pas accès au robot, nous avons décidé de faire une simulation simple sur le logiciel Unity.

Unity est un outil qui permet dans sa fonction principale de créer des applications 3D en temps réel tel que des jeux vidéo. Il permet grâce à une interface de placer des objets dans une scène et une ou plusieurs caméras. Le logiciel inclus aussi une gestion avancées des interactions physiques entre les objets de la scène, le tout très paramétrable grâce aux scripts. Il existe deux langages de scripts implémentés dans le logiciel, le C# et le Javascript. Durant ce projet nous avons fait le choix d'utiliser le C# car l'API est plus complète que celle du JS. Ce langage de script est à la base un langage de programmation de haut niveau développé par Microsoft dans les années 2000 exclusivement pour la plateforme Windows, mais qui dans ce cas présent est utilisé en temps que langage de scripts grâce à son implémentation libre Mono. Le C# a une syntaxe proche du Java. Mais il est quand même plus souple car il prend en charge les systèmes de "namespace" ainsi que la gestion de la surcharge d'opérateurs. Le langage permet même d'utiliser des pointeurs sous forme de blocs "unsafe" [1] mais son utilisation sous Unity n'est pas convaincante .

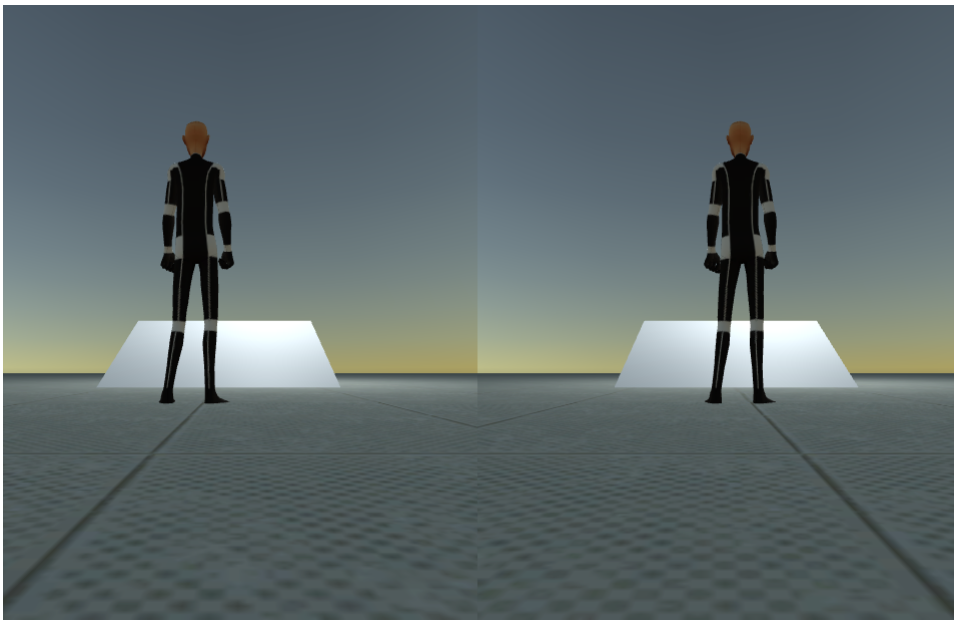

Figure 3.2.1 Image issue de Unity

Problèmes rencontrés Le logiciel est performant pour créer des petites scènes 3D mais il est très difficilement intégrable dans git ce qui pose des problèmes pour travailler simultanément sur la simulation. La version Linux

1. Gestion des pointeurs en C# : `https://msdn.microsoft.com/en-us/library/y31yhkeb.aspx`

de l'éditeur est aussi très instable, nous avons très souvent rencontré des bugs de l'éditeur.

3.3 Création de la bibliothèque

Pour faire la transition entre les algorithmes, l'implémentation du robot et la simulation nous avons décidé de faire une bibliothèque externe contenant tout les algorithmes. Cette bibliothèque est liée statiquement avec le contrôleur du robot et l'interface Unity du contrôleur. Les scripts C# d'Unity se chargent d'appeler les fonctions C++ des contrôleurs et de répercuter les actions sur la scène.
Dans cette bibliothèque, nous avons décidé d'implémenter uniquement la *stereoSGBM*, en effet *stereoBM* produit un résultat légèrement moins bon.

Voici ci-dessous une représentation des interactions entre Unity et la bibliothèque.

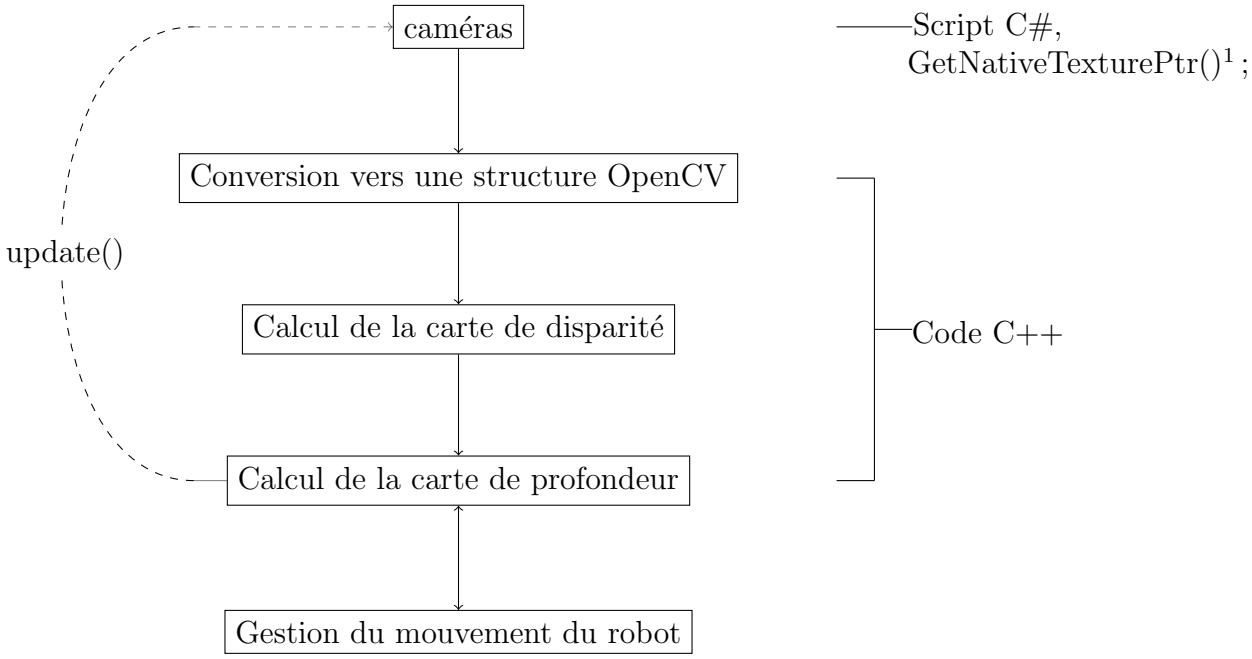

Figure 3.3.1 Interaction entre Unity et la bibliothèque.

1) GetNativeTexturePtr() est une méthode Unity permettant de récupérer le pointeur de resource OpengGl d'une texture. Cela nous a permis de récupérer ce que voient les caméras.

3.3.1 Conversion vers une structure OpenCV

La méthode GetNativeTexturePtr() renvoie un pointeur vers une image au format OpenGL (sous Windows, l'image sera au format directX) dans la mémoire, c'est à dire un id OpenGL. Pour convertir cette donnée vers une matrice OpenCV, il faut dans un premier temps lier cette donnée à une texture OpenGL grâce à la fonction include dans "GL/gl.h" glBindTexture(). Ensuite il suffit de copier les données dans un tableau au au format OpenCV, c'est à dire un taille égale à la taille de la texture et une profondeur de taille 1 octet * le nombre de canaux.
Comme représenté sur la figure 3.3.2, le standard OpenGL est de représenter l'image codée sous forme RGB et dans le sens informatique c'est à dire, le début de l'image se situe est en haut à gauche avec les Y vers la droite et les X vers le bas. Alors que le format OpenCV est de stocker l'image sous format BGR dans le sens mathématique, c'est à dire le début de l'image se situe en bas à droite avec les Y vers la droite et les X vers le haut.

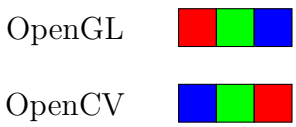

Figure 3.3.2 Représentation d'un pixel OpenGL et OpenCV

Problèmes rencontrés Des problèmes ont été rencontrés lors de l'utilisation de la bibliothèque externe sous Unity. En effet, s'il y a un problème à l'exécution d'une des méthodes de la bibliothèque, l'éditeur s'arrête brusquement sans message d'erreur, rendant le débogage difficile. Nous avons aussi rencontré des problèmes avec l'éditeur au CREMI, en effet pour Rémy et Jérémi l'édtiteur ne se lance même plus. Nous avons aussi des problèmes avec l'exportation des textures des caméras. Il peut arriver que les images récupérées soient déformées et donc unitilisable par nos algorithmes.

Solutions envisagés Nous avons décidé de créer en parallèle de l'intégration à Unity un automate écrit en C++ permettant de tester les algorithmes. Un script C# se charge de prendre une capture d'écran des deux caméras et de les écrire sur le disque. L'automate quant à lui prend en entrée les différentes images et utilise les algorithmes pour produire la carte de disparité et la carte de profondeurs.
L'automate est un petit programme écrit en C++ en utilisant l'outil GNU *gengetopt* pour gérer les arguments de commandes. Par exemple :

```
./automate -i path/to/images -a dist
./automate -i path/to/images -a disp
```

Usage :

–help

–detailed-help

-i input "Le dossier contenant les images à traiter"

-o output "Le dossier qui contiendra les images traitées"

-p préfixe "Préfixe à appliquer aux noms des fichiers à sauvegarder"

-a action "L'action a effectuer sur chaque fichier : disp, sgbm, dist"

Chapitre 4

Le robot

4.1 Caractéristique

Le robot possède deux caméras Blackfly [1] de 1.2 méga-pixel de la marque FLIR®. La taille du capteur est de 6 mm.
La lentille est une Lensagon [2] et possède une focale de 3.5 mm.
Enfin le robot possède une unité de calculs intégrés, c'est un micro ordinateur équipé d'un APU AMD® A10 Micro-6700T Quad core [3] cadencé à 1.2 GHz et équipé du wifi haut débit.

4.2 Résultats

Pour la création de la carte de disparité, nous avons utilisé la fonction *stereosgbm* d'OpenCV. Cette dernière prend de nombreux paramètres :

- minDisparity : valeur minimal possible de la carte de disparité.
  Paramètre choisit : 0
- numDisparities : soustraction entre la valeur maximum et la valeur minimum de la carte de disparité, cette valeur doit être multiple de 16.
  Paramètre choisit : 64
- SADWindowSize : taille des blocs de correspondance, doit être un nombre impaire situé entre 3 et 11.
  Paramètre choisit : 21

1. https://eu.ptgrey.com/blackfly-12-mp-color-gige-poe-aptina-ar0134-16-eu
2. https://www.lensation.de/product/BM3516NDC/
3. http://products.amd.com/en-us/search/APU/AMD-A-Series-Processors/AMD-A10-Series-APU-for-Laptops/A10-Micro-6700T-with-Radeon%E2%84%A2-R6-Graphics/18

- P1 et P2 : contrôle le lissage de la carte de disparité.
  Paramètres choisis : 8 * SADWindowSize * SADWindowSize et 32 * SADWindowSize * SADWindowSize
- disp12MaxDiff : différence maximal pendant la vérification.
  Paramètre choisit : -1
- preFilterCap : valeur de troncature pour les pixels d'image pré-filtrés.
  Paramètre choisit : 31
- uniquenessRatio : marge en pourcentage.
  Paramètre choisit : 0
- speckleWindowSize : Taille maximum de la région du lissage, permet d'éviter le bruit.
  Paramètre choisit : 0
- speckleRange : variation maximal entre chaque composante connecté.
  Paramètre choisit : 2

Nous avons créé une structure prenant en compte tous les paramètres présentés plus haut et nous la passons en argument de la méthode *disparityMap* implémentée dans la bibliothèque.

Figure 4.1.1 Image stéréoscopique de test.

Figure 4.1.2 Carte de disparité

Figure 4.1.3 Carte de profondeur

Pour tester notre implémentation, nous avons pris des photos issues des caméras du robot. La figure 4.1.1 est l'une d'entre elle. Puis nous avons créé une carte de disparité et une carte de profondeur.

4.3 Faire bouger le robot

Il faut à présent utiliser la carte de profondeur pour permettre au robot de prendre une décision. Pour cela, nous avons décider de découper la carte de disparité en trois parties verticalement. Nous calculons ensuite la moyenne de chaque partie. Nous choisissons ensuite la moyenne la plus petite, et le robot se dirige dans cette direction. Si il faut aller à gauche ou à droite, le robot s'arrête et tourne dans la direction voulue. Si il faut aller tout droit le robot arrête de tourner et avance.

Dans la carte de profondeur (figure 4.1.3), chaque pixel représente une distance. Plus la distance est importante plus la valeur du pixel est élevé et donc plus il sera blanc. Dans le cas contraire, plus la distance est proche, plus la valeur du pixel est basse et donc plus le pixel apparaitra sombre.
On peut par exemple voir que dans la figure 4.1.3, les jambes sont plus proches que le mur en arrière plan ; elles sont donc plus sombres.
Il faut noter que lorsqu'on affiche une carte de profondeur, le gradient de gris est entre les valeurs 0 et 1 inclus. Cependant, dans nos cartes de profondeurs les valeurs peuvent aller jusqu'à 20, distance que nous considérons infinie.

Problèmes rencontrés Cette méthode possède comme défaut d'être très imprécise. Si une personne passe devant les objectifs, le robot aura des difficultés à garder sa trajectoire. Le robot ne s'oriente pas tout à fait correctement non plus.

Chapitre 5

Conclusion

Pour conclure, les objectifs fixés par le cahier des charges ont presque tous été atteins. Durant la majorité du temps, nous avons travaillé sur la simulation et nous avons malheureusement pas eu assez de temps pour pouvoir tester l'implémentation sur le robot. De plus Unity ne prend pas en charge les unités de distances donc la simulation ne nous permet pas de dire si le robot fonctionnera correctement.
Il reste cependant encore des problèmes a résoudre. Premièrement, la difficulté de définir des paramètres pour le calcul de la carte de disparité font que le programme est sensible aux fluctuations de luminosité. Une solution apportée serait de trouver dynamiquement des paramètres. Enfin, avec l'implémentation actuel et le fait que faute de temps le programme n'a pas été testé dans des conditions réel, il est difficile à dire comment il se comportera dans ses conditions. Et donc nous ne pouvons pas certifier que le robot suivra la bonne cible si une autre passe devant les objectifs. Dans le futur certaines perspectives d'amélioration pourraient donc être faite.