

Master 1 - Ingénierie Informatique

Travaux d'Etude et de Recherche

-

GraphoScan

Mémoire Intermédiaire

Thibault CHARPIGNON
Benoît GALLET
Emmanuel HERRMANN
Martin RÉTY

Encadré par : Matthieu EXBRAYAT

6 Février - 13 Mars 2017

Contents

1 Résumé du projet	3
1.1 Présentation	3
1.2 But du projet	3
2 Introduction au domaine	3
3 Analyse de l'existant	4
3.1 Fonctionnalités déjà implémentées	4
3.2 Fréquence d'acquisition limitée	4
3.3 Impossibilité de bouger la feuille	4
3.4 Problèmes divers	5
4 Besoins fonctionnels et non fonctionnels	5
4.1 Augmentation de la cadence d'acquisition	5
4.1.1 Programmation parallèle	5
4.1.2 Complexité	6
4.1.3 Modularité	6
4.1.4 Généralisation	6
4.2 Tracking de la plume	6
4.2.1 Analyse de la complexité	7
4.2.2 Gestion mouvements	7
4.3 Autres axes de travail	7
4.3.1 Zone de capture	7
4.3.2 Changement de la structure	8
4.3.3 Compatibilité Windows - Linux - macOS	8
5 Prototypes et résultats de tests préparatoires	8
6 Planning	9
7 Exemple de fonctionnement	10
8 Architecture	15
9 Algorithmes et structures de données	17
9.1 Parallélisation OpenMP	17
9.2 Export des paramètres de la caméra	17
10 Complexité	18
10.1 Acquisition	18
10.1.1 Acquisition stéréo et enregistrement des vidéos	18
10.1.2 Undistortion	18
10.2 Reconstruction	18

11 Tests de fonctionnement et de validation	18
11.1 Latence	18
11.2 FPS	20
12 Extensions et améliorations possibles	20
12.1 Droits d'auteur	20
12.2 Interface graphique	21
13 Manuel d'utilisation	21
13.1 Installation	21
13.1.1 FlyCap	21
13.1.2 OpenCV	22
13.1.3 OpenGL, glew, glut et glm	22
13.2 Utilisation	23
13.2.1 Acquisition	23
13.2.2 Reconstruction	23

1 Résumé du projet

1.1 Présentation

Ce projet de TER prolonge un travail déjà entamé l'année dernière par deux étudiants de Polytech, consistant à enregistrer en vidéo l'écriture d'un calligraphe, pour pouvoir reconstruire un modèle en 3D du mouvement de la plume. Une structure en bois supporte deux caméras, que l'on peut bouger le long de rails puis fixer à l'aide de vis. Le calligraphe écrit sous cette structure et la feuille est éclairée par des spots lumineux. Il faut alors associer l'image des deux caméras, ce qui n'est pas possible nativement avec le logiciel fourni par le fabricant (FlyCapture de PointGrey) pour faire de l'acquisition vidéo en stéréo, puis reconstituer via OpenGL les mouvements de la plume. Ces mouvements ont été sauvegardés grâce à des algorithmes de tracking, travaillant sur les vidéos enregistrées auparavant.

1.2 But du projet

Ce projet permettra à terme de réaliser une reconstitution 3D des mouvements du calligraphe. On pourra alors lui faire recopier plusieurs textes, provenant de différents lieux et différentes époques, afin de pouvoir comparer les styles d'écriture, définir s'il existait différentes écoles d'écriture, différents styles, etc. De manière plus générale, le projet pourra servir pour beaucoup d'applications par la suite, car le code final se voudra le plus généraliste possible.

2 Introduction au domaine

Le programme en lui-même est entièrement réalisé en C++, et différentes bibliothèques graphiques sont utilisées dans ce projet pour le traitement du flux vidéo, dans le but de faire du tracking sur le résultat, notamment OpenGL et OpenCV. OpenGL est utilisé pour la reconstruction du mouvement de la plume en 3D, tandis qu'OpenCV est plus utilisé pour le traitement de l'image, notamment toutes les opérations faites dessus. De plus, Matlab est utilisé pour effectuer diverses opérations mathématiques sur les images, comme par exemple pour la calibration originelle servant à l'alignement pour la stéréo, ainsi que pour l'enlèvement de la distorsion. En effet, comme les caméras ont un grand angle, les éléments sur le bord de l'image sont courbés, il faut donc les remettre droits pour pouvoir travailler dessus.

3 Analyse de l'existant

3.1 Fonctionnalités déjà implémentées

Le programme tel qu'il nous a été fourni dispose de plusieurs fonctionnalités élémentaires à son bon fonctionnement. Parmi celles-ci, nous pouvons compter :

- Le calibrage du dispositif dans sa globalité (caméras + surface d'écriture)
- La synchronisation des deux caméras pour une reconstitution en trois dimensions des gestes lors de l'écriture

En plus de ces fonctionnalités, existe un dispositif matériel de capture. Ce dernier (Figure 1) est composé d'une structure en bois sur laquelle sont montées deux caméras. Ces dernières sont connectées à l'ordinateur par le biais d'un câble USB 3.0 chacune. Il est également possible de les bouger afin d'en ajuster les réglages.

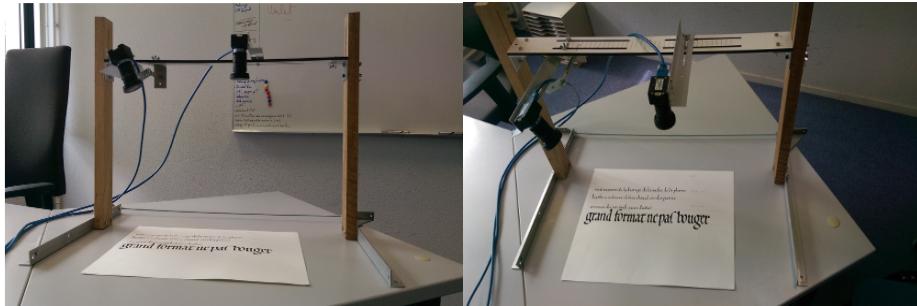


Figure 1: Dispositif de capture stéréo

3.2 Fréquence d'acquisition limitée

Jusqu'alors, le programme ne possédait pas une fréquence d'acquisition suffisante de l'image. En effet, cette dernière n'était que de l'ordre de huit images par seconde. De ce fait, cette valeur ne permet pas une reconstitution précise en trois dimensions des gestes du calligraphe. L'objectif ici est donc de se rapprocher le plus possible de la fréquence maximale d'acquisition des caméras du dispositif, soit trente images par seconde et ainsi gagner en précision lors du traitement.

3.3 Impossibilité de bouger la feuille

Avec la configuration actuelle, le calligraphe a l'impossibilité de bouger la feuille sur laquelle il écrit sous peine de perdre les réglages définis auparavant. Instinctivement, la personne qui écrit peut souhaiter bouger cette feuille et ainsi gagner en confort. Le but serait donc de trouver un moyen de gérer un

changement de position de la feuille sans que cela n'affecte les résultats, en recalculant les réglages par exemple.

3.4 Problèmes divers

Initialement, l'éclairage du dispositif se faisait à l'aide de deux spots disposés de part et d'autre du calligraphe. Le problème majeur d'un tel moyen est la présence d'ombres à certains endroits rendant le traitement des images difficile, ainsi que la chaleur des lampes pouvant se révéler gênante à la longue. Un autre problème à gérer sont les gestes "inutiles" à l'acquisition que le calligraphe peut faire. En effet ce dernier peut par exemple vouloir étendre son bras pour se relaxer, geste qui sera pris en compte par le programme dans la reconstitution.

4 Besoins fonctionnels et non fonctionnels

Une séparation des tâches était déjà effective dans le projet original, sur lequel travaillaient deux personnes : une personne s'occupait de l'acquisition stéréo, pendant que la seconde était sur la modélisation 3D de la plume. Cette séparation a été conservée dans ce TER : GALLET Benoît et HERRMANN Emmanuel s'occupent de la première partie, tandis que CHARPIGNON Thibault et RÉTY Martin ont pris la deuxième.

4.1 Augmentation de la cadence d'acquisition

Naturellement, différentes idées nous sont venues pour augmenter la cadence d'acquisition de la vidéo en stéréo. Nous les détaillons ici, même si par la suite cette liste sera sûrement étoffée. Le programme fonctionne suivant plusieurs étapes : Tout d'abord, une image est capturée à partir des deux caméras, puis les deux images sont encodées dans une vidéo. Une fois la capture finie, la vidéo est traitée afin de régler les problèmes de distorsion. Grâce à cette grande boucle qui capture les images deux par deux (une image par caméra), les vidéos finales commencent et terminent exactement au même moment, et permettent donc d'avoir exactement au même moment la feuille d'écriture filmée sous deux angles différents. La modélisation en 3D sous OpenGL est alors possible. La seule variable est que plus la cadence d'acquisition est élevée, plus il y aura de FPS sur les vidéos finales, et plus la modélisation 3D de la plume sera précise. De plus, notre architecture et notre code doivent être assez robuste, pour que si un jour une troisième voire une quatrième caméra soient rajoutées, le nombre de FPS ne redescende pas drastiquement.

4.1.1 Programmation parallèle

Grâce à la programmation parallèle, qu'elle soit au niveau du CPU avec de l'OpenMP ou des threads, ou au niveau du GPU avec CUDA, nous pensons pouvoir accélérer l'acquisition des images, et donc des FPS sur les vidéos finales.

Nous pensons regarder quelles parties peuvent être faites en parallèle, peut-être est-il possible d'uniquement récupérer une image tous les 3 centièmes de secondes (pour les 30 fps) dans le programme principal, et de faire tous les autres traitements dans des régions parallèles, avec par exemple un thread qui s'occupe d'ajouter la prochaine image à la vidéo, un autre thread qui enlève la distorsion de l'image, etc.

4.1.2 Complexité

Reprendre le code pour en examiner sa complexité est une autre piste envisagée pour augmenter les FPS. Nous pensons séparer cette idée en deux étapes : Tout d'abord regarder la complexité de l'algorithme dans sa généralité, pour se rendre compte s'il y a problème ou non à ce niveau là, et voir les morceaux posant plus problème que le reste, puis faire des tests plus précisément sur ces parties pour voir ce qui ne va pas. Une analyse légèrement différente pourra être effectuée, avec des tests fonctionnels calculant quelle partie prend le plus de temps. Ces tests sont très complémentaires de ceux de complexité, à eux deux ils devraient mettre en exergue les problèmes principaux du code actuel.

4.1.3 Modularité

Outre cette analyse de la complexité, une mise au propre du code devra être effectuée. En effet, tout se trouve dans la fonction `main`, dans deux grandes boucles. Une partie de notre travail sera donc de modulariser cette fonction, de la séparer en plusieurs méthodes afin de gagner en clarté. De plus, l'ajout de l'option `-O2` lors de la compilation permet d'optimiser sensiblement les performances. Nous prévoyons par la suite d'utiliser à la place `-O3` qui compilera les fonctions sur une seule ligne, ce qui permettra de minimiser le coût de cette modularisation du code lors de son exécution.

4.1.4 Généralisation

Sinon, le dernier axe sur lequel travailler sera la généralisation du nombre de caméras. Pour l'instant, tout dans le code est fait pour deux caméras, avec du code dupliqué pour chaque action. La généralisation pour n caméras sera facilitée par la modularisation du code, et permettra par la suite de rajouter une ou plusieurs caméras sans modification majeure, uniquement en changeant quelques `#define`.

4.2 Tracking de la plume

Comme pour la partie sur l'augmentation de la cadence d'acquisition, différents problèmes sont à résoudre pour le tracking. Cette partie permet de traiter les vidéos produites par les caméras et d'en ressortir une trace des mouvements effectués par le calligraphe. L'étudiant de Polytech qui a travaillé sur cette partie a recherché différents algorithmes permettant d'effectuer ce tracking. Son étude se focalise sur deux algorithmes basés sur l'apprentissage de toutes les

apparances observées de l'objet, et d'une estimation des erreurs pour ensuite les éviter :

- Tracking Learning Detection (TLD)
- Kernelized Correlation Filters (KCF)

4.2.1 Analyse de la complexité

Heureusement, l'analyse des deux algorithmes a déjà été faite par l'étudiant, ce qui a montré que dans notre cas l'algorithme KCF est le plus efficace. Son étude est basée sur plusieurs critères, la déviation moyenne des deux vidéos, le nombre de *frames* et le temps de calcul. Seul le premier critère est réellement différent entre les deux méthodes. C'est cette différence qui a orienté son choix vers l'algorithme KCF.

Ici, notre premier axe de recherche sera orienté vers une étude complémentaire de ces algorithmes pour vérifier la véracité de l'analyse précédente. Pour cela nous allons réutiliser les critères d'étude et ensuite essayer d'en trouver d'autres pour confirmer le choix. Dans un second temps il nous faudra rechercher d'autres algorithmes ou méthodes de programmation pour améliorer le tracking.

4.2.2 Gestion mouvements

Bien entendu, le choix des algorithmes n'est pas la seule difficulté, nous faisons face également à des contraintes physiques liées aux mouvements du calligraphe. Par exemple il doit prendre des temps de repos afin de garder sa fluidité d'écriture en faisant des gestes de relaxation du poignet. Ces mouvements ne doivent pas être pris en compte par l'algorithme de tracking afin d'éviter des erreurs sur la représentation du mouvement.

Résoudre ce problème pourrait passer par la sauvegarde à un temps T et à un temps T+1 d'une image de la partie suivie, puis analyser la différence entre les deux images et en ressortir un résultat positif ou négatif. Cela revient à prendre la dernière image où le calligraphe écrit et une autre image qui permettra de voir si le mouvement est la continuité de l'écriture ou un mouvement parasite.

4.3 Autres axes de travail

4.3.1 Zone de capture

En écrivant, le calligraphe doit de temps en temps bouger la feuille pour se repositionner et continuer sa rédaction. L'algorithme actuel ne gère pas ce mouvement, ce qui nécessitait après chaque mouvement de la feuille un nouveau calibrage des caméras et de la zone de capture. Une solution possible pour résoudre ce problème est la mise en place d'un système de cadre pour que le calligraphe sache la zone dans laquelle il peut écrire. Ce cadre pourrait être

un marquage sur la feuille qui délimitera la zone de capture. Nous souhaitons également rechercher d'autre solutions possibles pour résoudre ce problème.

4.3.2 Changement de la structure

Pour le moment le dispositif de capture ne comporte que deux caméras et des angles de prises de vue bien définis. L'ajout d'une caméra et le repositionnement des deux premières peut permettre de rendre plus précise l'acquisition. De cette manière nous aurions à notre disposition des informations supplémentaires pour améliorer la reconstitution du mouvement. Il nous faut donc tester différentes configurations et choisir la meilleure.

Un des facteurs majeurs de la capture d'image est la lumière. En effet, il est important que la feuille soit bien éclairée pour le confort et l'écriture du calligraphe. La structure actuelle ne comporte pas d'éclairage du tout, il était nécessaire d'avoir une lampe d'appoint. Une solution simple est l'ajout d'un panneau LED pour avoir une luminosité uniforme sur toute la feuille.

Tous ces changements devront peut-être être accompagnés d'une refonte totale du dispositif.

4.3.3 Compatibilité Windows - Linux - macOS

A l'origine, les étudiants ont développé tout le code sur Windows et plus particulièrement sur l'IDE Visual Studio (C++). Pour rendre le code réutilisable à l'avenir nous avons comme objectif de pouvoir l'utiliser sur tous les systèmes d'exploitation (Linux/macOS en plus). Pour cela il est nécessaire d'uniformiser le code et de se servir de librairies communes pour standardiser au mieux le projet.

5 Prototypes et résultats de tests préparatoires

Il nous fallait, pour bien prendre en main le projet, tester réellement le dispositif de lancement du logiciel jusqu'à la capture vidéo. Nous avons dû procéder à l'installation de tout l'environnement de travail nécessaire (FlyCap2, OpenCV, OpenGL) et l'acquisition des premières vidéos avec les caméras mises à notre disposition.

Notre première tâche a été de transférer le code initial sous Linux et ainsi le tester directement. Les tests ont été concluants et le premier groupe a pu commencer directement à améliorer le système. Le second, quant à lui, a récupéré le code concernant le tracking mais a rencontré de gros problèmes lors de son passage de Windows vers Linux. Le code n'utilisant pas des fonctions standards, important des librairies en "dur" et étant peu commenté, il est pour le moment impossible de tester le code de l'étudiant qui travaillait sur le tracking l'année précédente. Pour y remédier le second groupe a dû repasser cette partie du

projet sur Windows le temps de bien comprendre les différents problèmes et de les corriger.

6 Planning

Les petites barres sur le diagramme de Gantt (Figure 2) correspondent aux différentes réunions que l'on a eu, suivies des noms des participants. Les plus grandes correspondent quant à elles aux tâches effectuées. Il y en a peu pour le moment car la phase de compréhension et d'installation des composants a été relativement longue.



Figure 2: Diagramme de Gantt de la première partie

7 Exemple de fonctionnement

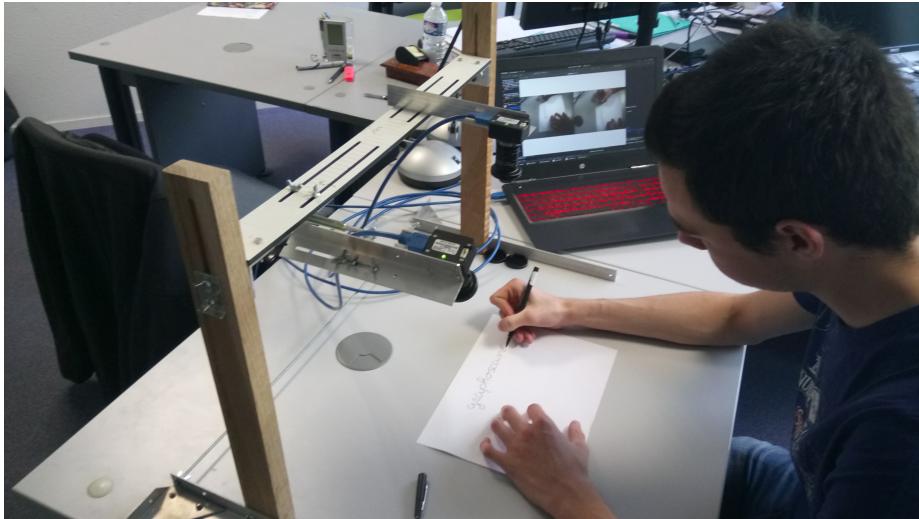


Figure 3: Ecriture et acquisition

Ce dispositif nous permet de capturer les vidéos en direct. Ces deux vidéos sont enregistrées sur ordinateur, pour pouvoir faire de la stéréo : on encode les images en même temps pour avoir exactement les mêmes frames au même moment. On applique ensuite des paramètres d'undistortion pour lisser les bords de l'image. En effet, la caméra ayant un grand angle, l'extérieur de l'image est déformé, ce qui peut apporter des problèmes lors de la reconstruction 3D.

Une fois les deux vidéos récupérées, on peut lancer la reconstruction. La première étape est de retrouver l'écriture sur l'image, pour cela différentes méthodes sont disponibles. Comme sur la figure 5, on peut faire du tracking. Après avoir sélectionné une zone d'intérêt grâce à un ROI (Region Of Interest) selector (figure 4), on essaye de suivre le mouvement. Il suffit d'enregistrer les coordonnées du centre du point pour retrouver le mouvement dessiné.

La seconde technique est celle du HOG (Histogramme de gradient orienté) qui permet de détecter des formes dans une image. Ce qui donne au départ une image comme sur la figure 6, et après nettoyage 7.

L'une ou l'autre de ces techniques peuvent être utilisés pour avoir une suite de coordonnées correspondant au dessin de l'écriture. Une fois que ces deux coordonnées sont récupérées, il existe des fonctions dans OpenCV pour faire des points 3D à partir de points 2D. Il faut pour cela récupérer les paramètres extrinsèques (matrices de rotation et vecteurs de translation, pour passer du repère lié à l'espace de travail au repère lié à la caméra) de la caméra.

Une fois que les coordonnées sont récupérées par l'une ou l'autre méthode, une fonction nous permet de rajouter deux points entre chaque coordonnées pour fluidifier la ligne. Ces coordonnées sont ensuite interprétées par OpenGL

pour permettre de les afficher dans un repère en trois dimensions (figures 8 et 9).

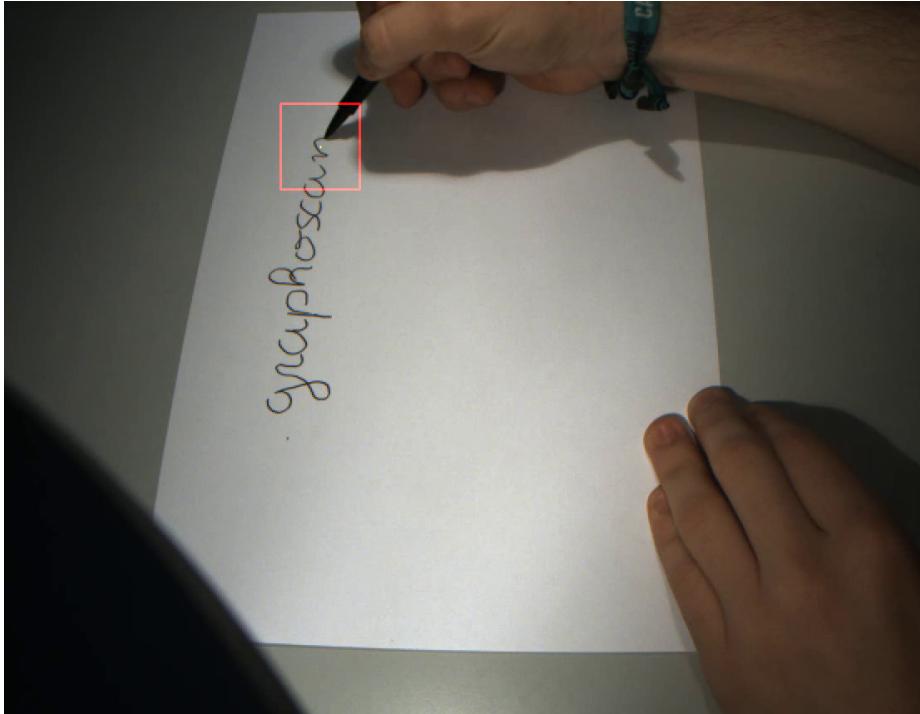


Figure 4: ROI selector

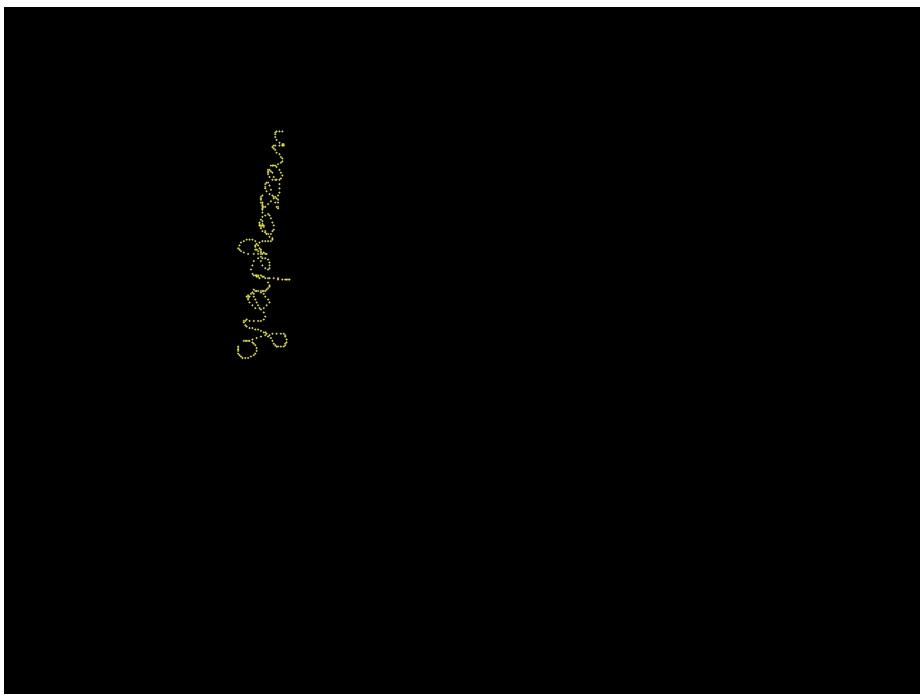
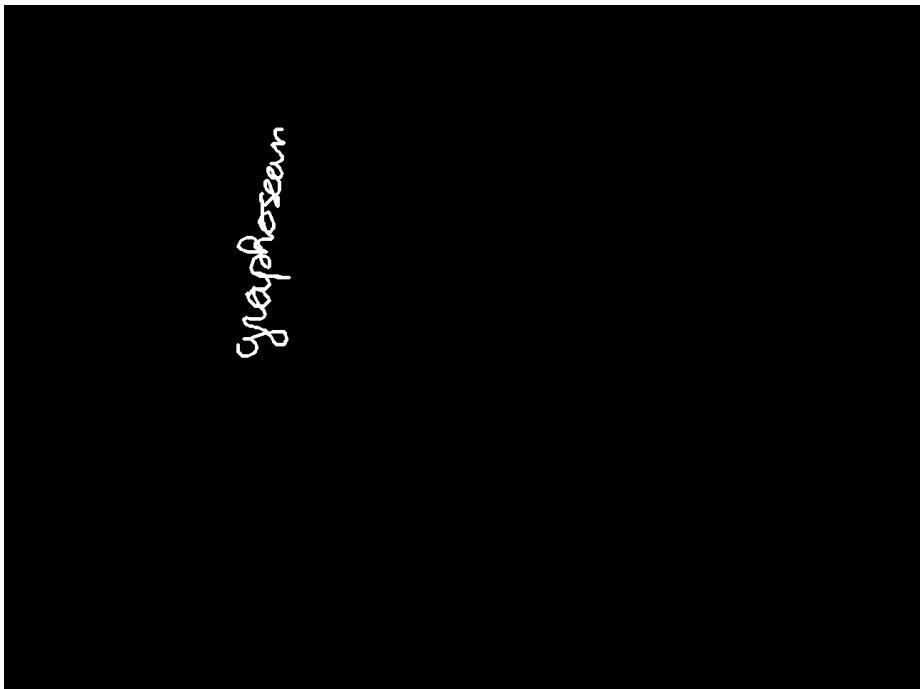


Figure 5: Reconnaissance avec le tracking



Figure 6: HOG brut



13

Figure 7: HOG nettoyé

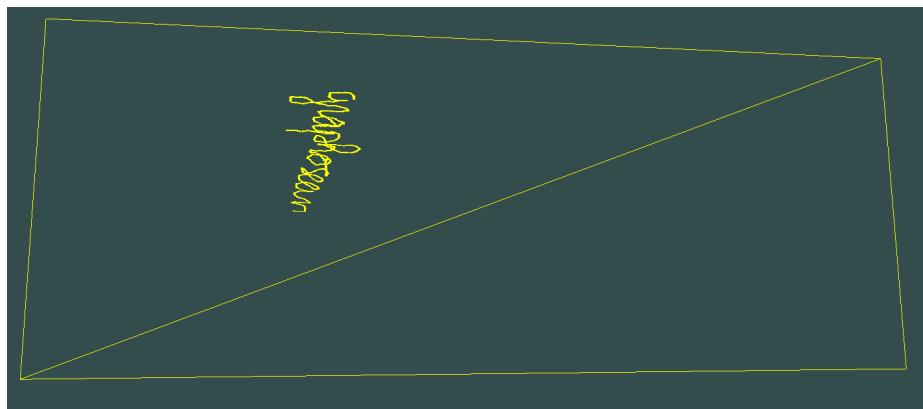


Figure 8: Image 3D - 1



Figure 9: Image 3D - 2

8 Architecture

Le projet est scindé en deux grosse parties, presque indépendantes l'un de l'autre. Comme montré dans la figure 10 , toute l'acquisition des vidéos se fait en parallèle dans la partie acquisition. Cette partie gère aussi l'undistortion des vidéos. Une fois les vidéos créées et traitées correctement, on peut les envoyer à la seconde grosse partie qui est celle du tracking/reconstruction 3D. Cette partie permet de gérer différentes choses, comme des types de tracking ou de la reconstruction en OpenGL. Cette classe fonctionne comme une boîte noire, que l'on peut paramétriser à loisir pour réaliser ce dont on a besoin.

8.1 Acquisition

L'acquisition est elle-même séparée en deux parties. La première est l'acquisition à proprement parler, elle récupère le flux vidéo des deux caméras et permet de les synchroniser. Les vidéos sont encodées en avi, image par image, au rythme de 30 fps, ce qui est la cadence maximum des caméras.

Une fois les vidéos synchrones créées, de l'undistortion est appliquée dessus. En effet, comme les caméras possèdent un grand angle, les bords sont déformés, il faut donc les remettre droits.

8.2 Tracking et reconstruction

Les vidéos créées sont ensuite passées au deuxième programme. Celui-ci peut exécuter différentes fonctions sur les vidéos. Une classe HOG permet de faire de la reconnaissance d'image, et une classe GraphoScan permet de lancer les différents trackings. Pour l'OpenGL, la classe Shader permet de compiler des shaders tandis que OpenGL permet de dessiner les points dans le repère. Enfin, Camera sert à exécuter les différents mouvements de la caméra (zoom, direction, ...).

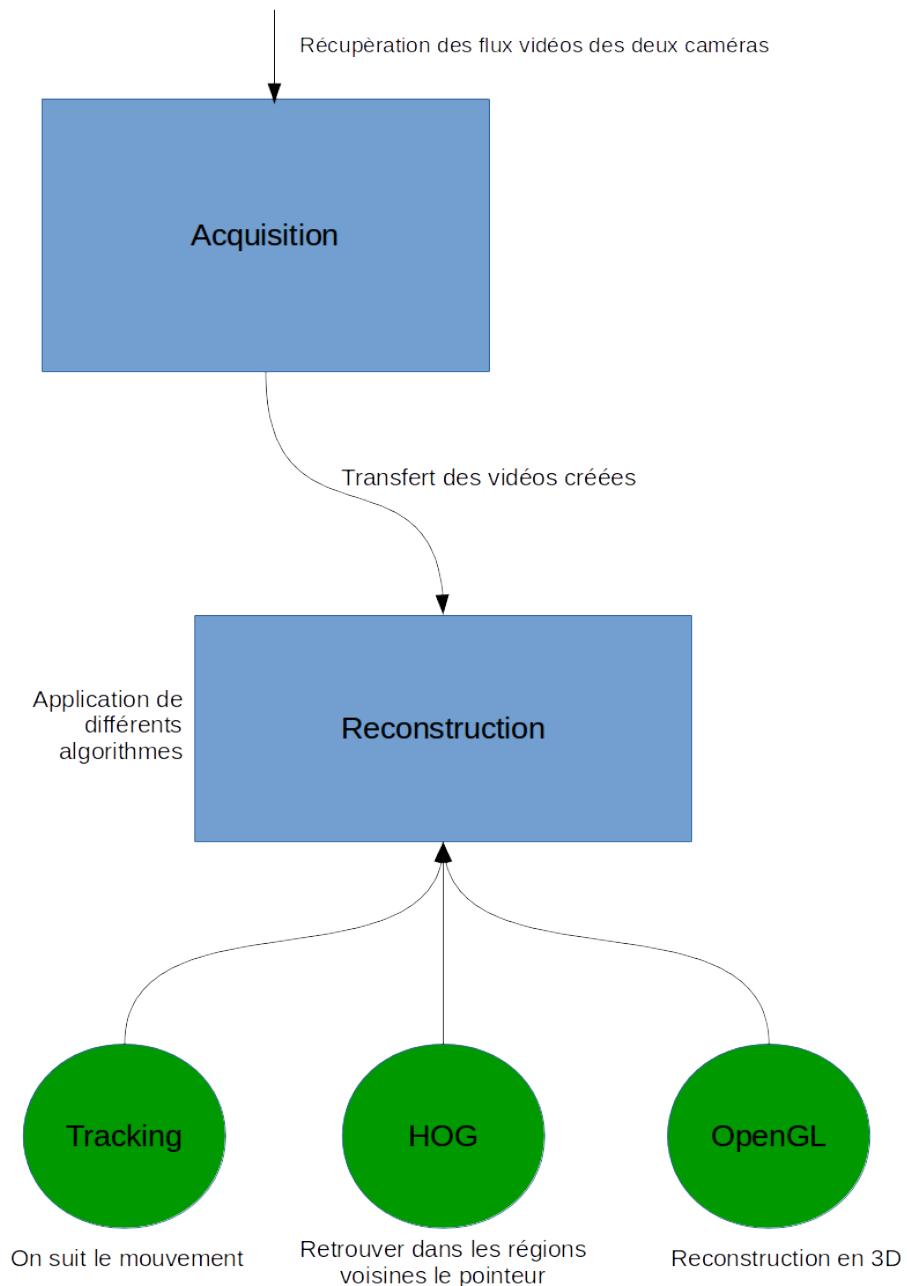


Figure 10: architecture

9 Algorithmes et structures de données

9.1 Parallélisation OpenMP

Afin de pouvoir faire de l'acquisition vidéo en simultané sur plusieurs caméras tout en gardant le plus possible d'images par secondes, nous avons opté pour une parallélisation de l'algorithme d'acquisition grâce à OpenMP. Ainsi nous affectons - dans la mesure du possible - une caméra à un thread en lançant la région parallèle de cette sorte :

```
#pragma omp parallel num_threads(numCameras)
```

Avec `numCameras` le nombre de caméras détectées.

Juste avant de commencer la capture, nous posons une barrière à l'aide de

```
#pragma omp barrier
```

dans le but de déclencher les caméras le plus en même temps possible. À l'intérieur de la boucle d'acquisition, une nouvelle barrière est mise avant chaque récupération du buffer de la caméra. Dans le cas où l'utilisateur souhaite faire un affichage de la capture qu'il est en train d'effectuer, un des threads est choisi via

```
#pragma omp single
```

afin de s'occuper du dit affichage. Enfin, chaque thread écrit la frame dans la vidéo correspondante à la caméra affectée au thread.

9.2 Export des paramètres de la caméra

La récupération des paramètres de la caméra, calculés via MatLab, va nous permettre de faire l'undistortion et la reconstruction 3D. Jusqu'à présent, ces paramètres étaient rentrés en dur dans les programmes, nous avons donc décidé de faire des imports(exports de ces données pour plus de simplicité et de réutilisabilité. Pour ce faire, une fois les paramètres de calibration calculés dans MatLab , on récupère un objet CameraParameters par caméra. Afin de générer les fichiers de configuration nécessaires, il suffit de rentrer deux commandes par caméra :

```
dlmwrite(  \
'*PATH_TO_ACQUISITION*/Calib_camera_*NUM_CAMERA*_Matlab.txt', \
camera*NUM_CAMERA*.IntrinsicMatrix,'delimiter', ' ', \
'precision', 5)
dlmwrite(  \
'*PATH_TO_ACQUISITION*/Calib_camera_*NUM_CAMERA*_Matlab.txt', \
horzcat(camera*NUM_CAMERA*.RadialDistorsion, \
camera*NUM_CAMERA*.TangentialDistorsion), \
'-append', 'delimiter', ' ', 'precision',5)
```

Il faut remplacer `*NUM_CAMERA*` par le nom de l'objet de la caméra correspondante.

10 Complexité

10.1 Acquisition

10.1.1 Acquisition stéréo et enregistrement des vidéos

On assume le fait qu'il y ait n lignes et m colonnes dans une image. Nous proposons deux complexités, une borne min si on considère qu'un pixel de l'image se traite en temps $O(1)$, et une borne max si l'on considère que le pixel se traite en temps $O(3)$ à cause de ses caractéristiques RGB.

Borne min

Récupération de l'image brute (RetrieveBuffer) : $O(n * m)$

Conversion de l'image brute vers RGB (rawImage.Convert) : $O(n * m)$

Création de la matrice de l'image RGB (imageRGB=cv::Mat) : $O(n * m)$

Ecriture d'une image dans la vidéo (outputVideo.write) : $O(n * m)$

Ce qui nous donne un total de temps d'exécution en $O(4(n * m))$.

Borne max

Il suffit de multiplier par trois les valeurs précédentes, ce qui nous donne au total $O(12(n * m))$

Analyse

Nous pouvons donc en conclure que le temps d'exécution de l'algorithme C est : $O(4(n * m)) \leq C \leq O(12(n * m))$

C'est-à-dire qu'il s'exécute globalement en temps linéaire en la taille de l'entrée. Cela dit, il faut l'exécuter à chaque tour de boucle (à chaque acquisition d'image), donc on pourrait dire que sa complexité serait alors de $O(n * m * f)$, f étant le nombre de frames que nous enregistrons. Evidemment, cela vaut pour la version parallèle du code, si on prend en compte la version séquentielle, il faut bien sûr multiplier cette complexité par le nombre de caméras que l'on a.

10.1.2 Undistortion

10.2 Reconstruction

11 Tests de fonctionnement et de validation

Différents tests de validation ont été effectués tout au long du projet afin de s'assurer du bon fonctionnement de l'application. Ces tests pourront être refaits par la suite en décommentant les `#define` nécessaires, et pourront donc être réutilisés par des étudiants reprenant ce projet.

11.1 Latence

La latence est une caractéristique incontournable de notre projet, le client nous ayant précisé plusieurs fois que c'était très important qu'il y ait le moins de décalage possible entre les deux vidéos. Plus le décalage est grand et moins la reconstruction en 3D par la suite sera précise. Nous avons donc mesuré précisément le temps qui sépare deux frames dans le code.

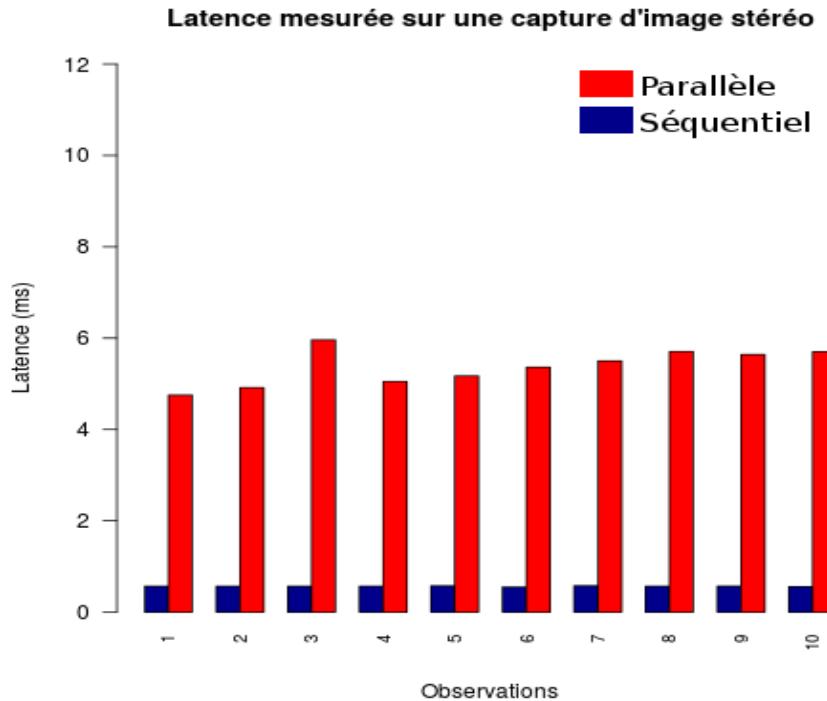


Figure 11: Latence

Ce diagramme en bâton (Figure 11) montre cette latence. Elle est plus importante avec le parallélisme qu'avec le programme séquentiel. Il faudra donc que le client pèse le pour et le contre sur la méthode qu'il préfère utiliser. Avec le parallelisme, le nombre de caméra pourra être augmenté (de façon raisonnable) sans chute de fps, car chaque cœur de la machine s'occupera d'une caméra particulière, par contre, la latence est plus importante(5 à 6 ms) . Avec la version séquentielle du code, la latence est plus que minime (1/2 millième de seconde environ), mais l'ajout d'une à plusieurs caméras fera chuter le nombre de fps. De plus, la latence sera sûrement plus constante dans la région parallèle, car chaque thread s'occupant d'une caméra, celle-ci n'augmentera pas si on ajoute plus de caméras (dans la limite des coeurs disponibles sur la machine). Par contre, séquentiellement, comme on lance les acquisitions les unes après les autres, plus il y a de caméras, plus celles qui sont lancées en dernier auront de décalage avec la première.

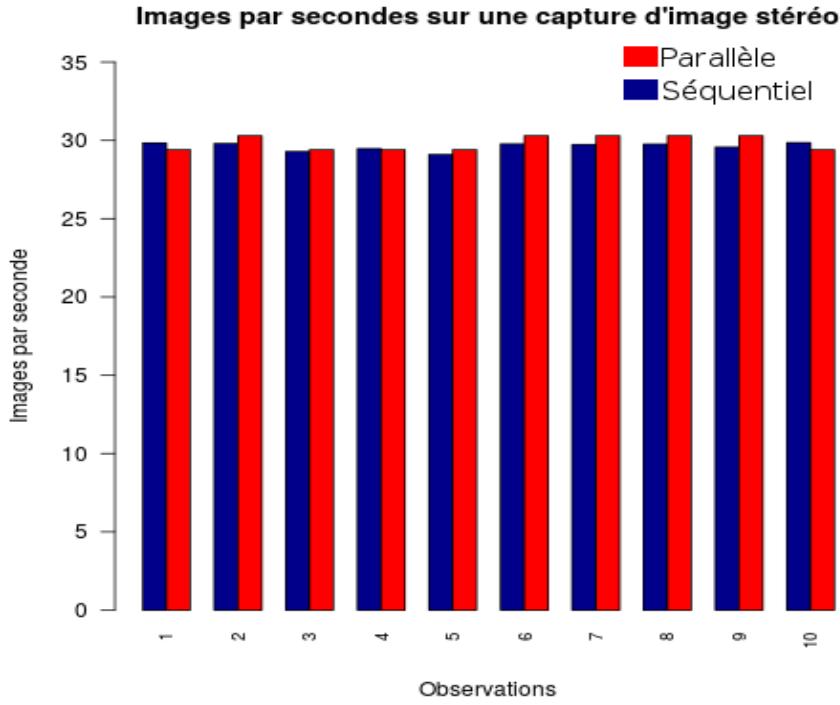


Figure 12: FPS

11.2 FPS

Les FPS étaient un des objectifs principaux de notre TER. En effet, le groupe précédent n’arrivaient qu’à enregistrer des images à la vitesse de 7/8 FPS, ce qui ne permettait pas une reconstruction optimale. Comme on peut le voir sur la Figure 12, nous faisons de l’acquisition à 30 FPS constants, en parallèle comme en séquentiel. Cependant, nous pensons qu’il faut privilégier le code fait en parallèle, car il est plus robuste à l’ajout de caméras futures.

12 Extensions et améliorations possibles

12.1 Droits d'auteur

Au fur et à mesure que nous essayions de comprendre le code de la reconstruction 3D (ce qui a été long car il a été fourni sans doc, commentaires et explications d'utilisation, avec une syntaxe et des imports pour Windows), nous nous sommes rendu compte que sur cette partie plusieurs morceaux de code avaient été copiés sur Internet.

La classe Shader, ainsi que la moitié de la classe Camera provient de http://blog.csdn.net/sinat_26989191/article/details/51205149, un outil pour faire un système solaire en 3D.

La classe HOG est presque entièrement copié collée de la fin de ce programme (<http://lib.csdn.net/snippet/cplusplus/28974>), qui est sous copyright.

Diverses autres parties du code proviennent de copiés-collés de différentes documentations ou exemples, ce qui est dans une certaine mesure moins dérangeant.

Il y a certainement d'autres endroits du projet qui proviennent d'Internet, tout n'a pas été vérifié par manque de temps. Au total, nous estimons qu'au moins 50% du code n'a pas été écrit par le groupe précédent. Cela implique que si ce projet est repris par un groupe futur, il faudra réécrire ces parties du code, ou du moins indiquer clairement les sources et les différents copyrights sur les classes copiées.

12.2 Interface graphique

Une extension possible serait de réaliser une interface graphique, par exemple en Qt. Cela permettrait un maniement beaucoup plus facile de l'application, et rendrait possible son utilisation par des personnes extérieures. En effet, en ce moment tout s'exécute dans le terminal, ce qui peut être moins intuitif. Avec cette interface graphique, on pourrait par exemple choisir de sélectionner une caméra pour voir ce qu'elle enregistre, définir différentes options comme choisir l'algorithme de tracking à utiliser, les sorties vidéos ou images à effectuer, ...

13 Manuel d'utilisation

Pour pouvoir lancer et utiliser cette application, différentes bibliothèques doivent être installées sur l'ordinateur. Toutes les manipulations sont décrites pour un environnement Linux (ou Mac), mais peuvent être transposées assez facilement pour Windows.

13.1 Installation

13.1.1 FlyCap

FlyCap est le framework de PointGrey, le fabricant des deux caméras, permettant de communiquer dans un programme avec les caméras. L'installation est relativement aisée, il suffit d'aller sur le site du constructeur (<https://www.ptgrey.com/support/downloads>) pour télécharger le framework. Il faut alors renseigner le type de caméra (Chameleon3), le modèle (CM3-U3-13S2C-CS) puis le système d'exploitation utilisé. Il faut ensuite sélectionner "Latest FlyCapture2 Full SDK" dans la partie software. Une fois le téléchargement terminé, un README détaille toutes les étapes nécessaires à l'installation, il suffit de les suivre une à une pour avoir FlyCap en état de marche. Pour vérifier si

le logiciel s'est bien installé, il est possible de le lancer dans un terminal via la commande *flycap* pour voir la vue d'une des caméras.

13.1.2 OpenCV

OpenCV doit être installé également, afin de profiter de son traitement d'image et de ses algorithmes de tracking. Il a été décidé d'utiliser OpenCV 3, car celui-ci contient un module optionnel OpenCV_contrib. Ce dernier contient tous les algorithmes de tracking, c'est un extra-module indispensable pour pouvoir exécuter le projet.

Pour installer OpenCV, voici la marche à suivre :

- On se place dans le dossier de travail et on télécharge les dernières versions OpenCV et d'OpenCV_contrib.

```
cd ~/<my_working_directory>
git clone https://github.com/opencv/opencv.git
git clone https://github.com/opencv/opencv_contrib.git
```

- On se place dans le dossier OpenCV téléchargé et on crée un répertoire temporaire pour le build.

```
cd ~/opencv
mkdir build
cd build
```

- On lance la configuration en n'oubliant pas d'inclure l'extra-module. Cette étape peut prendre un certain temps, jusqu'à 1h30 suivant les machines.

```
cmake -D CMAKE_BUILD_TYPE=Release
-DOPENCV_EXTRA_MODULES_PATH=*PATH_TO_FOLDER*/opencv_contrib/modules
-D CMAKE_INSTALL_PREFIX=/usr/local ..
```

- On démarre l'installation. Si on ne possède que quatre cœur, mieux vaut marquer -j4.

```
make -j7
```

- Dans certains cas, il faut rajouter la ligne suivante dans le .bashrc pour indiquer où se trouvent les librairies OpenCV.

```
export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:/usr/local/lib
```

13.1.3 OpenGL, glew, glut et glm

- Il faut commencer par installer OpenGL et Mesa :

```
sudo apt-get install cmake xorg-dev libglu1-mesa-dev
```

- Pour savoir si l'opération s'est bien passée, il faut regarder que les deux fichiers suivants sont bien présents :

```
/usr/include/GL  
/usr/lib/x86_64-linux-gnu/libGL.so
```

- A présent, il faut retourner dans le dossier de travail pour installer GLFW. Commencer par télécharger le code source (<https://sourceforge.net/projects/glfw/files/glfw/3.0.4/glfw-3.0.4.zip/download>), puis exécuter les instructions suivantes :

```
cd glfw-3.0.4  
rehash  
cmake -G "Unix Makefiles"  
make  
sudo make install
```

- Les fichiers suivants doivent maintenant être présents :

```
/usr/local/include/GLFW  
/usr/local/lib/libglfw3.a
```

Pour installer glew : Dans synaptic, installer les packages glew-utils, libglew-dev, libglew1.13, libglewmx1.13

Pour installer glm : Dans synaptic, installer le package libglm-dev.

13.2 Utilisation

Le travail est séparé en deux parties : acquisition et reconstruction, que l'on retrouve dans deux dossiers différents.

13.2.1 Acquisition

Un Makefile est fait pour compiler cette partie. Il permet de compiler le code avec les différentes librairies opencv et flycap. Une fois compilé, il suffit de lancer l'exécutable créé pour lancer l'acquisition stéréo.

13.2.2 Reconstruction

Cette partie nécessite plus de librairies, à cause de la reconstruction OpenGL. Le mode de fonctionnement est donc légèrement différent. Il faut se placer à l'intérieur du répertoire build, et lancer la commande `ccmake ..` pour construire dans le dossier parent l'environnement d'exécution. Il faut ensuite faire la commande `make` pour compiler le programme. Il suffit alors pour lancer l'application de passer en paramètre de l'exécutable les deux vidéos à traiter pour la reconstruction 3D, puis de se laisser guider par les instructions dans le terminal.