Suivi d'un terrain de handball

Alex Dembélé

Avril 2024

I Introduction

Dans ce projet, nous allons essayer de développer un système permettant de mesurer la distance parcourue par des joueurs de handball via une vidéo prise par une caméra monoculaire. Tout le challenge est d'arriver à situer le terrain dans l'espace 3D et de localiser les joueurs sur ce terrain. Ce travail fait suite à un stage qui traite de ce sujet en utilisant des méthodes de machine learning. Nous allons ici utiliser des méthodes issue de la robotique, notamment en essayant de faire marcher un ORB SLAM pour effectuer la tâche de localisation dans l'espace du terrain. Le but est de comparer les méthodes. Vous pourrez trouver le github associer à ce projet icigithub

II Description du système

Le système étudié est une caméra qui filme automatiquement un match de handball. Il existe déjà des systèmes qui permettent de filmer automatiquement un match. Nous souhaitons améliorer ce système en lui ajoutant des briques qui permettent de tracker les joueurs, repérer le terrain dans l'espace et enfin mesurer la distance parcourue par les joueurs. C'est compliqué d'automatiser cette tâche à cause de plusieurs problème. Tout d'abord, dans un match de handball, les joueurs passent devant les un les autres, se rentrent dedans, sortent du terrain et du champ de la caméra. IL y a donc un problème d'occlusion qui gêne le tracking. Ensuite, le terrain apparaît partiellement à la vidéo. De plus, il y a de nombreuses lignes qui ne sont pas celles du terrain de handball, il est donc dur de repérer le terrain. Il est aussi difficile de se repérer sur ce terrain, il faut donc changer le système de coordonnées pour se placer dans un repère où l'on sait mesurer des distances.

FIGURE 1 – Dispositif

Comme le système est composé de plusieurs briques, qu'il y a une caméra et des moteurs si l'on s'intéresse au système global, alors utiliser ROS est intéressant pour faire communiquer tous ces composants.

II.1 Tracking joueurs

Le tracking des joueurs est fait en utilisant le réseau de neurone yolov8 pour lés détecter. Nous nous intéressons pas en profondeur à ce tracking dans ce projet donc nous en resterons là.

II.2 Repérage du terrain

Cette partie est celle qui nous intéresse. C'est sur cette tâche que nous allons comparer les performances des techniques ML et robotiques. Le but de cette tâche est de situer le terrain ou la partie de terrain, qui apparaît sur l'image de la vidéo, dans un système de coordonnées que l'on connaît afin de pouvoir faire des mesures.

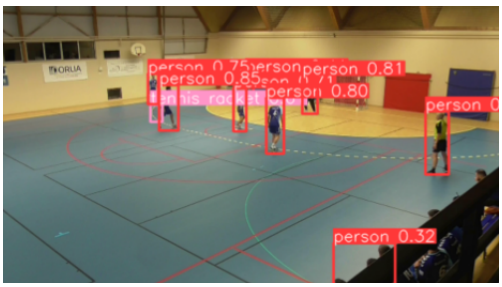

FIGURE 2 – Exemple yolov8

II.3 Mesure de distance

Cette partie sera faiblement abordé car elle nécessite que le tracking et le repérage soient fonctionnels. Elle consiste à fusionner les informations de ces deux parties pour faire la mesure effective des distances parcourues par les joueurs.

II.4 Intérêt du système

Le système est intéressant non seulement pour automatiser la prise de statistique pour le suivi de performances des athlètes, mais aussi pour toutes les méthodes développés. En effet, le passage d'un plan à l'autre et le repérage du terrain pourrait permettre aux industries audiovisuelles se mettre des pubs plus intégrées dans l'environnement. Par exemple, au lieu d'avoir un overlay fixe dans l'écran, on pourrait avoir un overlay fixe sur le terrain. Cela rendrait plus agréable à l'oeil la diffusion de pub ou de message informatif.

III Approche par ML

Dans cette partie nous faisons le repérage du terrain par des techniques de ML. Le but est de déterminer une loss à la main. Cette loss doit être minimiser sur un opérateur, l'homographie qui permet de passer d'un système de coordonnée à un autre. On pourra alors passer des coordonnées de la vidéo à celles d'un terrain standard dont on connaît les dimensions. Cette approche a été faîte en PRE, voici le rapport

Le début de ce rapport permet de se familiariser avec la théorie utiliser dans ce projet. Dans ce PRE, l'opérateur au centre de l'attention était l'homographie, une matrice 3*3 qui permet de relier 2 plans.

IV Approche par SLAM

Dans cette approche, nous allons nous concentrer sur la matrice de projection caméra qui est de taille 3*4 (ou 4*4 selon les utilisations). Nous allons utiliser des techniques de SLAM pour déterminer cette matrice au cours de la vidéo. L'accès à cette matrice nous permettra ensuite de repérer les points clés dans l'espace (joueurs).

Comme annoncé en introduction, l'objectif de ce projet était de faire marcher un orbslam3 sur une vidéo ou avec un enregistrement caméra. Nous allons utiliser le middleware ros pour intégrer orbslam3 dans un pipeline qui permet de réaliser toutes nos tâches. Nous avons aussi fait un programme python simple qui réalise toutes les tâches pour pouvoir faciliter l'intégration.

IV.1 Pipeline

La figure 3 détail le pipeline ROS que nous utilisons lors de ce projet. On commence par acquérir une vidéo soit par la caméra soit en lisant une vidéo déjà enregistré. Ensuite, on envoie les images vers un noeud qui permet de détecter les joueurs avec des bounding box et un noeud qui va faire l'odométrie visuelle (ORBSLAM3) pour reconstituer le mouvement de la caméra. Enfin, on combine le résultat des deux noeuds précédents pour ressituer dans l'espace 3D les joueurs et le terrain.

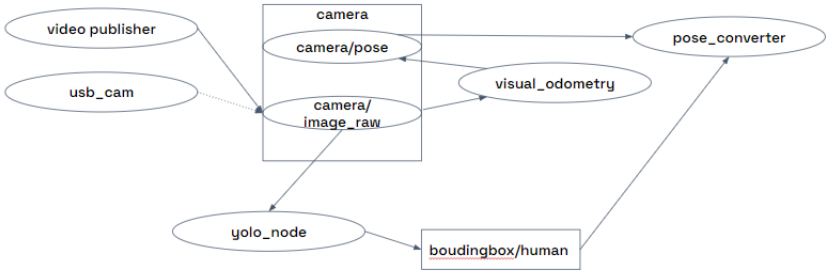

FIGURE 3 – Pipeline

IV.2 Acquisition vidéo

Nous utilisons deux méthodes pour récupérer un flux vidéo, un noeud usb_cam qui permet d'acquérir une vidéo à partir d'une caméra et un noeud video_publisher qui permet de lire et publier une vidéo sur un topic approprié.

IV.2.1 usb_cam

Le noeud usb_cam est un noeud déjà implémenter de ROS, il permet de lire le flux vidéo d'une caméra et de le publier sur le topic /usb_cam/image_raw. Il publie aussi d'autre message sur d'autres topics comme l'image compressé etc..

IV.2.2 video_publisher

Le noeud video_publisher est un noeud que nous avons implémenter en python. Il permet de lire une vidéo avec openCV et de la publier sur le topic désiré. Nous pouvons choisir la taille de l'image et la fréquence de publication.

IV.2.3 Calibration caméra

Afin de traiter les vidéos, il faut connaître certains paramètres e la caméra. Pour les connaître, on fait de la calibration de caméra. Le noeud ROS usb_cam contient un programme pour le faire. Il suffit de filmer un quadrillage (échiquier) et de le faire bouger.

IV.3 Détection joueurs

Comme annoncé précedemment, la détection des joueurs se fait avec un réseau de neurones : yolov8 Nous avons implémenter un noeud ROS en python pour ce faire : yoll_node. Il prend en entrée une image et retourne les coordonnées des boundings box des joueurs sous formes d'une liste applati. Il faudra alors reconstruire la liste des boundings box en regroupant les valeurs par 4.

Cependant, il est apparu un problème lors de l'utilisation de ce noeud, comme le programme est en python et que c'est un réseau de neurones, l'inférence est assez lente. Pour améliorer les performances de ce noeud, il pourrait être judicieux d'essayer de trouver une version compiler en c++ de yolov8. Sinon, il faut essayer de jouer avec la fréquence de l'entrée et de ce noeud pour avoir de la cohérence entre les données.

IV.4 Orbslam3

Nous avons passé le plus de temps de notre projet à essayer de faire marcher un algorithme ORBSLAM3 dans un noeud ROS. Cependant, nous n'avons pas réussi.

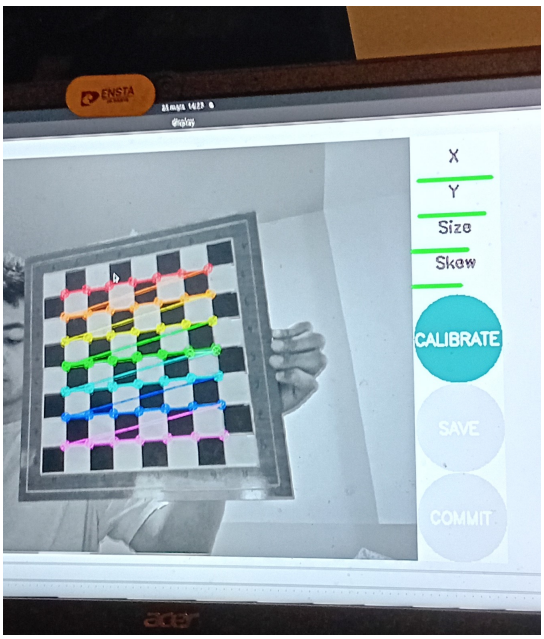

FIGURE 4 – Calibration caméra

IV.4.1 Description

L'algorithme ORBSLAM3 permet de calculer la position de la caméra au cours d'une vidéo. Il fonctionne pour divers type de caméra, dont une caméra monoculaire. Le github officiel est ici.

IV.4.2 Dépendance et compilation normale

Pour fonctionner, ORBSLAM3 à besoin de plusieurs librairies et doit fonctionner sur une certaine version de Ubuntu.

Pour commencer, nous avions un ordinateur en dualboot avec un Ubuntu 22. Nous avons du remplacer ce dualboot par un Ubuntu 20, pour faire marcher ros noetic. Puis, nous avons installer ROS sur cette machine. Ensuite, nous avons du installer OpenCV en c++ sur notre machine. L'installation n'était pas aisé car il y avait un problème de chemin d'inclusions. Nous avons passer une demi-journée à redéfinir proprement tous les chemins d'inclusions pour faire compiler openCV. Puis, il a fallu installer et compiler eigen et pangolin. Enfin, nous pouvons essayer de faire compiler ORBSLAM3. Il a encore fallu redéfinir les chemins openCV. Il y a des problèmes à la compilation avec c++11 qui est utilisé dans les CMakes, il a fallu le remplacer par c++14. Il subsiste des erreurs avec Eigen3 et/ou pangolin. Il sont sûrement -liés à la version de Ubuntu car ce code a été testé sous Ubuntu 16 et 18 et nous sommes sous ubuntu 20. Nous avons réussi à compiler ce code seulement après la présentation orale ... De plus, nous l'avons fait fonctionner uniquement sur des exemples données. Le code fourni en sortie une liste des translations et rotations (sous forme de quaternion) de la caméra. L'interface visuelle n'a pas marché, il doit y avoir un problèeme avec Pangolin. Lorsqu'on lance le code sur certains exemples, notre machine à des fois du mal à mettre les images dans sa RAM et se stoppe à cet endroit dans le code mais ne retourne pas d'erreur...

IV.4.3 Compilation ROS

Nous avons essayé de faire marcher la version ROS du projet. Cela ne compile pas, il y a un problème avec les fichiers ROS du code officiel. C'est sûrement dû au fait que le projet est conçu pour ROS melodic et que nous sommes sous noetic. Il y a des problèmes de versions et de fragment de code manquant. Le fichier build pour ROS est aussi déprécié.

Pour palier à ce problème, j'ai cherché des wrapper ROS de ORBSLAM3. J'ai trouvé plusieurs code promettant de faire marcher le code sous ROS noetic. J'ai notamment trouver un wrapper qui ne nécissitait pas

de compilation. Il contient un launch qui permet de lancer le code. Cela ouvre une fenêtre Rviz configuré pour faire marcher ORBSLAM3. On voit les différents topics servant à faire fonctionner le tout. Cependant, malgré le fait que l'on publie les images sur le bon topic. Aucun calcul n'est fpasser plus d'une semaine à essayer de faire marcher ce wrapper. Mais nous n'avons pas réussi. Nous avons donc chercher d'autres wrappers. Mais nous n'avons réussi à en compiler aucun pour ROS. Il y avait soit des problèmes de versions de ROS/Ubuntu. Soit des wrappers mal fait, soit des problèmes de compilation/ exécution obscure car les fichiers sont mal reliés entre dans le wrapper.

IV.4.4 Docker

Mr Filliat nous a conseillé d'essayer d'utiliser des versions marchant avec Docker. N'ayant plus beaucoup de temps à y consacrer, nous avons rapidement essayé de faire marcher Docker. Nous ne connaissions pas ce logiciel. Il permet de déployer un projet très simplement en une ou deux lignes de commandes. Nous avons donc passer du temps à comprendre Docker mais nous n'avons pas eu le temps de l'utiliser pour faire marcher ORBSLAM3. Nous avons préféré consacrer du temps à nos propres méthodes.

IV.5 Visual Odometry

Comme je n'ai pas réussi à faire fonctionner un package de ORBSLAM3, j'ai essayé de le recoder partiellement en python. Pour ça, je me suis aidé d'un github qui fait de l'odométrie visuelle :github. C'est le noeud visual_odometry. J'avais déjà rapidement essayé ce genre de méthode en utilisant un détecteur KAZE pour faire matcher des points entre deux images puit en estimant l'homographie entre les images en résolvant le système d'équation correspondant ou par SVD.

Dans ce projet, on applique une méthode plus robuste.

IV.5.1 Fonctionnement de l'odométrie visuelle

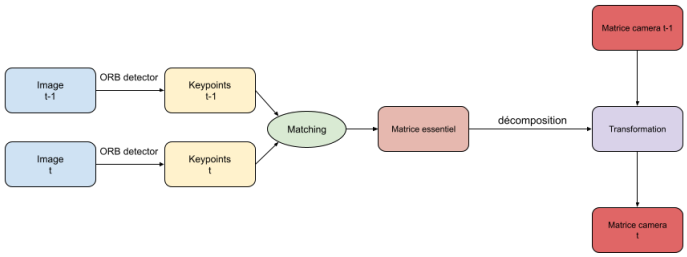

FIGURE 5 – Odométrie visuelle

La figure 5 représente le schéma de fonctionnement de l'odométrie visuelle. Pour estimer le mouvement de la caméra entre deux images consécutives, on utilise un détecteur ORB pour détecter des points d'intérêts entre deux images consécutives. Puis, on estime la transformation entre deux images grâce à la matrice essentielle qui nous permet de connaître la rotation et la translation de la caméra entre deux images. Il existe des méthodes openCV qui permettent de calculer la matrice essentielle à partir de deux nuages de points matchés et des

paramètres intrinsèques de la caméra **findEssentialMat**(), puis de la décomposer pour avoir la rotation et la translation **decomp_essential_mat**() qu'il faut un peu affiner.

Il faut noter que l'on est pas en présence de SLAM car la fermeture de boucle n'est pas traitée.

IV.5.2 Problème des joueurs

Lorsqu'on utilise l'odométrie visuelle tel quelle sur une vidéo d'un match de sport, il pparaît rapidement un problème. Beaucoup de points d'intérêts se situent sur les joueurs qui bougent par rapport au terrain. Ainsi, l'odométrie visuelle peut nous renvoyer des valeurs erronées de déplacement de la caméra car ce déplacement serait en réalité celui des joueurs. Il faut donc masquer les points qui correspondent aux joueurs. On peut utiliser la détection des joueurs faîtes précédemment pour ne pas tenir compte des points d'intérêts qui se trouvent à l'intérieur de bounding box de joueurs.

La figure 6 représente le nouveau schéma d'odométrie visuelle.

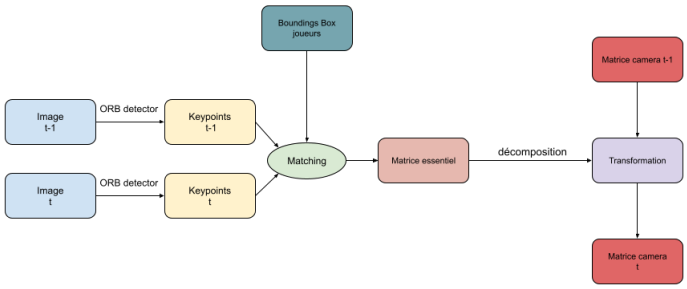

FIGURE 6 – Odométrie visuelle améliorée

IV.5.3 Reconstitution 3D

Pour reconstituer l'environnement en 3D, on utilise la triangulation. La méthode **triangulatePoints**() d'openCV permet de construire un nuage de point 3D à partir de points matchés et de deux position de caméra (matrice de projection). C'est avec cette méthode que 'on inverse la projection pour passer du 2D au 3D.

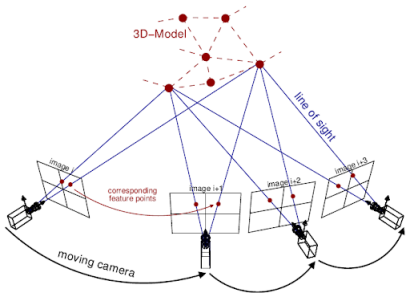

FIGURE 7 – Triangulation

IV.5.4 Modification du code d'origine

Le code d'origine traite le cas d'une séquence d'image et retourne la séquence de pose caméra correspondante. Il prend en entrée la séquence vidéo ainsi que la calibration de la caméra. Nous avons commencé par adapter ce code pour traiter une vidéo. Ensuite, nous avons coder une fonction qui extrait la rotation et les angles d'eulers de la pose de caméra. Cela nous permet d'avoir des informations sur le mouvement de la caméra. Puis, nous avons changer la calibration de la caméra. Ensuite, nous avons ajouter la triangulation pour reconstruire les points 3D ainsi qu'une visualisation de ces points 3D. Après cela, nous avons ajouter le masque qui permet de ne tenir compte que des points d'intérêts qui ne sont pas sur des personnes.

Enfin, nous avons rossifier ce code pour pouvoir l'utiliser dans notre pipeline. Nous avons

IV.6 Résultat avec un programme python

Le programme marche plutôt bien pour faire de l'odométrie visuelle. En effet, nous avons considérer une vidéo où la caméra fait un une rotation de gauche à droite. Lorsqu'on trace la courbe des angles d'eulers, on constate que tous les angles varient, dont un plus que les autres selon le mouvement de la caméra. Ils arrêtent de varier lorsque la caméra s'arrête.

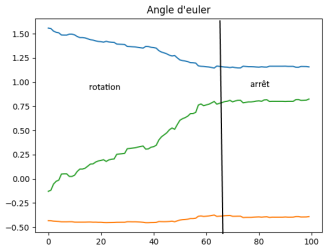

FIGURE 8 – Angle d'Euler

On constate qu'il y a un problème. Les points 3D ne se situent pas sur le terrain. Le terrain a des dimensions 20m*40m avec une hauteur de gymnase de 10m maximum. Or nos nuages de points ont des échelles très variables de 10^2m à 10^6m. Il y a donc un problème dans la méthode. Cela pourrait être l'initialisation de la matrice caméra qui est erronée.

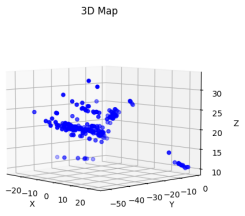

FIGURE 9 – Mauvais nuage de point

Après une longue réflexion sur les causes de ce problème, nous pensons avoir identifié le problème. **Il est quasiment impossible d'inférer de la profondeur avec une caméra mooculaire qui ne fait que des rotations et pas de translation** (ce qui est notre cas). En effet, comme la caméra ne fait que tourner, il n'y a pas de géométrie du triangle, un point est toujours vu sur la même droite reliant ce point au centre optique de la caméra. Cette droite est fixe, il n'y a donc pas de formation de triangle. Il est donc impossible de faire de la triangulation.

En poussant cette réflexion plus loin, on pense aussi que même si l'on avait fait marcher un ORBSLAM3, il aurait eu des problèmes à situer des points dans l'espace et à reconstruire un nuage de point 3D à cause de cette limitation du mouvement de la caméra à une rotation. Ce projet était donc sûrement voué à l'échec ! Il

est possible de faire de la profondeur en utilisant la parallaxe ou des réseaux de neurones, mais nous doutons qu'un ORBSLAM3 utilise ces techniques.

En terme de temps, l'exécution du programme sur une frame met environ 0.44s. C'est assez lent pour faire du temps réel mais faisable si l'on considère uniquement deux images par seconde ce qui est acceptable, on ne risque pas de perdre trop d'information.

IV.7 Résultats avec pipeline ROS

Comme discuté dans la présentation des différents noeuds, il y a des problèmes de fréquence. Les calculs sont trop lents (0.3s par frame) comparés à la fréquence d'entrée de l'image (25-30 fps). Il faut alors trouver le noeud le plus lent et limiter la fréquence d'entrée par rapport à celui-ci. Avec la configuration actuelle, le noeud le plus lent est l'odométrie visuelle. Cela peut-être empiré si l'on rajoute l'étape de masquage des joueurs non astucieusement. On pourrait néanmoins récupérer cette information du noeud de détection des joueurs.

Il n'est pas absurde d baisser la fréquence des images d'entrées. On peut choisir de ne traiter que 3 images par secondes au lieu de 25 ou 30, en ne traitant pas la vingtaine d'autre images. Le tracking serait moins continu, mais la caméra aurait des mouvements un peu plus amples, ce qui pourrait permettre une meilleure situation dans l'espace et moins d'accumulation de dérive.

Dans le pipeline ROS, comme la triangulation n'est pas possible, nous avons essayé d'inverser la matrice de projection pour obtenir les points3D. Mais ce n'est pas une bonne méthode car il est difficile d'accéder aux coordonnées 3D de l'image qui est en 2D.

Nous avons néanmoins essayé de reconstruire un environnement 3D correspondant au terrain de handball et à la position de la caméra à l'aide de nuages de point dans RVIZ.

IV.8 Comparaison

On a pas fait de triangulation en ROS car on a constaté sous un simple programme python que c'était pas une bonne méthode. On s'attache donc à comparer comme annoncer nos résultats entre l'odométrie visuelle et le machine learning.

IV.8.1 Comparaison programme python et pipeline ROS

Tout d'abord, on constate que la version ROS est plus rapide car plus fragmenté, il y a des process qui tourne en même temps. Cependant, cette parallélisation ce fait au dépend de la synchronisation des données (comme toujours en programmation parallèle). Les noeuds mettant des temps différents pour renvoyer un résultat, il se peut que les entrées du dernier noeud ne soit pas celle correspondant à la même image. Cela ne devrait pas poser de problème si les images ne sont trop éloignées dans le temps. Il est aussi possible de synchroniser les informations à l'aide des timestamp ROS.

De plus, le programme python est plus fiable et permet de tirer plus de résultat sans se compliquer la tâche avec ROS. Cependant, pour un code performant, il faudrait passer les codes en c++ et le pipeline ROS serait la meilleure option car il permet de séparer nativement les blocs et visualiser les données avec Rviz.

IV.8.2 Comparaison ML et ORBSLAM3

Dans ce projet nous avons constaté qu'ORBSLAM3 ou une méthode d'odométrie visuelle ne permet pas d'inférer de la profondeur. Elle permettent cependant de très bien tracker les mouvements de la caméra. Avec les méthodes de ML, on arriver à inverser approximativement la projection malgré que ce soit plus long en temps de calcul et moins précis que le tracking caméra par odométrie visuelle. Ainsi, si l'on ne trouve pas de méthode pour inverser la projection caméra une fois qu'on l' a obtenue avec l'odométrie, alors le ML est la voie à suivre. Sinon, il vaut utiliser l'odométrie visuelle car plus rapide et précise. AU final, on pourrait utiliser une combinaison des deux en tirant partie des avantages de chacune des méthodes pour combler les lacunes de l'autre.

V Conclusion

Nous avons passé notre projet à essayer de faire marcher un orbSLAM3, mais nous n'avons pas réussi. Même si cela nous a permis de découvrir pas mal d'aspect de ROS et de package SLAM que nous ne connaissions pas, nous trouvons dommage de ne pas avoir réussi à implémenter quelque chose de fonctionnel. Avec du recul, nous aurions du passer plus de temps à développer nous même un noeud remplaçant ORBSLAM3 et à le débugger/adapter pour avoir un truc qui marchouille un peu plutôt que de ne rien avoir du tout. De plus, s'être rappelé qu'il était quasiment impossible de faire de la profondeur avec une caméra qui fait uniquement de la rotation nous a fait poursuivre dans une mauvaise voie pendant trop longtemps.

Néanmoins, ce projet est très intéressant à développer car il traite de problématiques actuelles dans l'industrie audiovisuelle et robotique.

VI Organisation du projet

Nous avons passé la première séance à formuler le projet, récupérer le matériel et à s'informer . Nous avons passé les deux séances suivantes à installer Ubuntu 20, OpenCV et le reste des librairies utile à ORBSLAM. Puis, jusqu'à une semaine avant la restitution à orale nous avons essayé de faire marcher un noeud ROS ORBSLAM. Cela se décomposait en recherche internet d'un package, renseignement sur le package, puis téléchargement et tentative de compilation/exécution.