Projet SY31

Traitement de l’image :

Binarisation de l’image :

Définition : Ce que l’on appelle image binaire dans ce rapport est une image qui ne contient que du noir ou du blanc. Notre but est d’isoler l’objet dans cet image pour pouvoir étudier sa taille, forme, sa couleur.

Filtrer la couleur :

Pour détecter l’objet par sa couleur on utilise les valeurs HSV pour déterminer un intervalle de valeur `(Hue, Saturation, Value)` pour chaque couleur. Puis on convertit l’image en image HSV (chaque pixel contient les trois valeurs de HSV) et on utilise la fonction `inRange(img, lowe, upper)` qui renvoi une image binaire. On préfèrera utiliser les valeurs de pixel HSV pour filtrer la couleur plutôt que les valeurs RGB qui sont beaucoup plus sensible à la lumière. Pour pouvoir déterùminer le filtre plus simplement qu’en tâtonnant avec Inkscape j’ai repris le script `range-detector.py` d’un programmeur sur github qui permet de visuellement trouver plus facilement les deux *threshold* de valeurs HSV. Ce programme fait apparaitre deux fenêtres, l’une permet de modifier les valeurs des threshold et l’autre affiche le résultat de la fonction `cv2.inRange(img, lower, upper)` . J’ai modifié le script pour que cela marche avec les valeurs HSV normale (les valeurs S, V de 0 à 100% au lieu de 0 à 255 comme implémenté dans OpenCV).
Exemple d’utilisation sur un screenshot :

Command : `python range-detector.py --filter HSV --image ~/panneauScreen.png`

Détecter les objets sans couleur :

Dans la liste des objets à détecter il y a des objets sans couleurs distincte comme le panneau de signalisation blanc, la vitre et le miroir. Pour détecter le panneau on a fait un filtre HSV mais qui marche pas très bien, en effet il est difficile d’isoler une *range* pour le blanc du panneau dans nos bags, car les murs sont blancs, le t-shirt de Nicolas est blanc. On a donc exploré l’utilisation de la détection de rebord avec OpenCV. OpenCV fournit une fonction magique `Canny` qui utilisent des algorithmes de traitement d’images, comme un réducteur de bruit (Gaussian Filter),un filtre de Sobel pour calculer l’intensité des

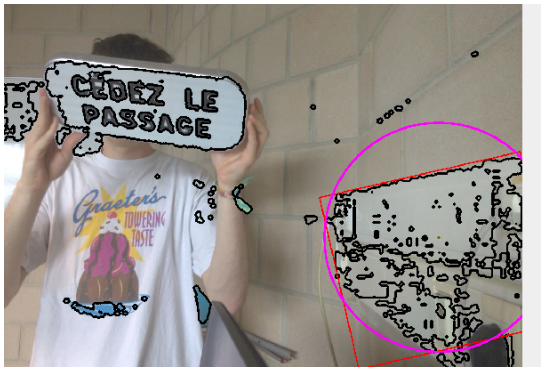
pixels et deux *threshold* pour discriminer les *edges* selon leur intensités. Malheursement, beaucoup trop d'*edges* sont détecter dans l'image ce qui ne rend pas cette technique viable pour nos bag car il devient difficile de trouve l'objet parmi les toutes les *edges.*

Source : Canny Edge Detection Tutorial by Bill Green, 2002 (archive)

Pour améliorer la détection de l'image on a essayé de flouter les endroits non intéressant du bag, avec les murs, le logo du t-shirt, etc avant de passer le l'image à la fonction `Canny` , cependant comme Nicolas bouge beaucoup cela ne marche pas du tout tout le temps et les edges trouver pour le panneau sont mauvaises, ne permettent pas de trouver la forme de l'objet. Donc nous avons laisser tomber cette idée.

Autre technique :

Au final la technique utilisée pour détecter le panneau sera la suivante, en s'inspirant du fait de flouter les bords de l'image on peut appliquer un mask sur l'image BGR pour se débarasser des bords, spécifiquement le bord droit qui nous embête car le support de prise en plastique a la même teinte que notre panneau de signalisation. Pour cela on décide de créer une image totalement noires avec la `np.shape` de notre image BGR. Puis on dessine un cercle blanc là où se trouve notre objet, ce qui crée un masque (image binaire). Ensuite on utilise `np.where` pour fusionner notre masque et l'image en couleur ce qui permet de détecter bien mieux les objets blancs avec le filtre HSV :

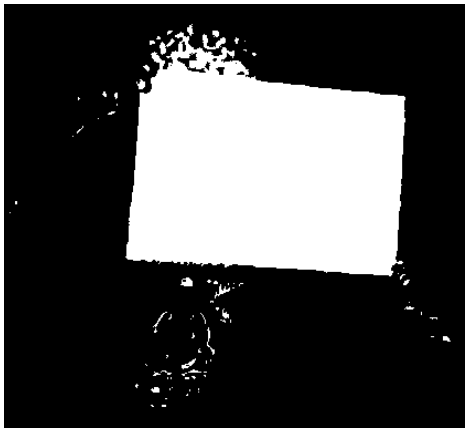


Détection avant le masque

Détection après le masque

Opérations morphologiques :

Pour réduire le bruits des filtres on utilise l' opération morphologique d'*opening,* qui utilise l'algorithme de dilatation (permet d'agrandir le blanc de l'image, dilate l'image), puis d'érosion, qui est l'opération inverse. Ces deux opérations permettent de réduire le bruit de l'image binaire.

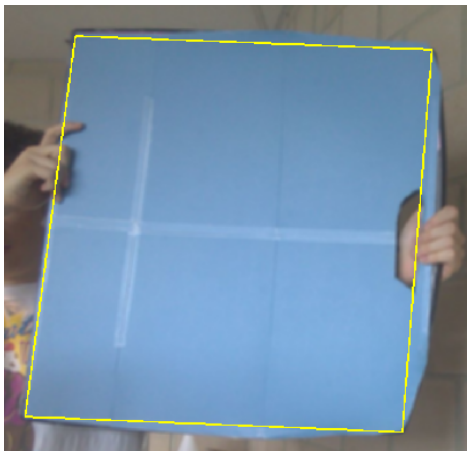


Filtre HSV pour le rouge sans *opening*

Avec *l'opening*

Détection des formes :

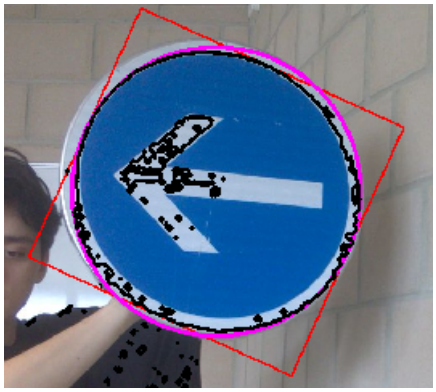
La méthode utilisée pour détecter les formes des objets utilise l'approximation des formes en polynômes par OpenCV. En ayant récupérer le contour de l'objet on peut utiliser la fonction `cv2.approxPolyDP(contour, epsilon, closed = True)` dessus pour simplifier le contour, `epsilon` indique la puissance de l'approximation (réduit le nombre de points possible pour le contour). On utilise un `epsilon` élevé et proportionnel à la taille du contour (avec le périmètre `arcLength()`) pour obtenir un contour très simplifié.
Puis on utilise la fonction `cv2.convexHull` pour réduire encore plus le bruits du contour approximé.
Ainsi on obtient un polygone avec peu de côtés et l'on peut discriminer facilement les rectangles des cercles en regardant ce nombre de côtés. Le nombre de côtés d'un polygone est le nombre de sommets donc il suffit de regarder le nombre de points du contour `len(contour)` , si il est égal à 4 (ou 5 pour le bruit), on a un rectangle, sinon on a un cercle.

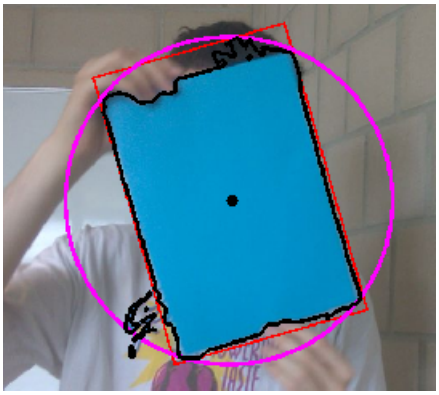


Ce fonctionnement marche bien pour la liste d'objets à détecter, mais n'est pas très robuste, par exemple un losange serait étiqueté "Rectangle". L'avantage c'est que même si l'objet n'est pas droit, est penché la forme sera toujours bien détecté.

Une autre manière de détecter les formes est d'utiliser les fonctions `boundingRect` , `minAreaRect` pour avoir un rectangle orienté et `minEclosingCircle` sur le contour de l'objet, ces fonctions vont créer un rectangle ou un cercle qui contient le contour. Une fois que l'on utilise ces fonctions sur le contour de l'objet on peut calculer l'aire du rectangle orienté obtenue que l'on compare à l'aire du cercle. L'aire la petite correspond à la forme de l'objet (cf images ci-dessous) L'un des problèmes de cette solution est que cela ne marche pas quand les objets sont inclinés (sur leur axe vertical par rapport à la caméra).

En noir, les contours détecter (par la binarisation de l'image
En rouge, le rectangle orienté (obtenu avec `minAreaRect`)
En rose le cercle (obtenu avec `minEnclosingCircle`)





Discrimination grands/petits cartons

Pour ce qui est de distinguer les grand cartons de couleur des petits, nous allons utiliser une des autres sources d'information que nous avons à disposition qui est la distance entre le carton et le robot. En effet, si l'aire du carton sur l'écran est grande et que la distance aussi, c'est probablement que nous avons le grand carton. Et inversement pour le petit carton. Nous allons pour cela évaluer le ratio aire/distance.

En expérimentant un peu avec les bags que nous avions à disposition, nous remarquons que le relation qui unit aire et distance n'est pas linéaire. Cela paraît logique car l'aire augmente quadruple quand la distance double. Sauf que l'on a remarqué ici que même lorsque l'on pondère en utilisant la relation sqrt(aire)/distance, la relation n'est toujours pas parfaitement linéaire (lorsque l'on rapproche le petit carton de la caméra, le ratio tend à monter alors qu'il devrait rester plus ou moins stable.

Nous avons définissons donc un seuil pour le ratio au delà duquel l'objet sera considéré comme gros carton et en deçà duquel on pourra considérer un petit carton.

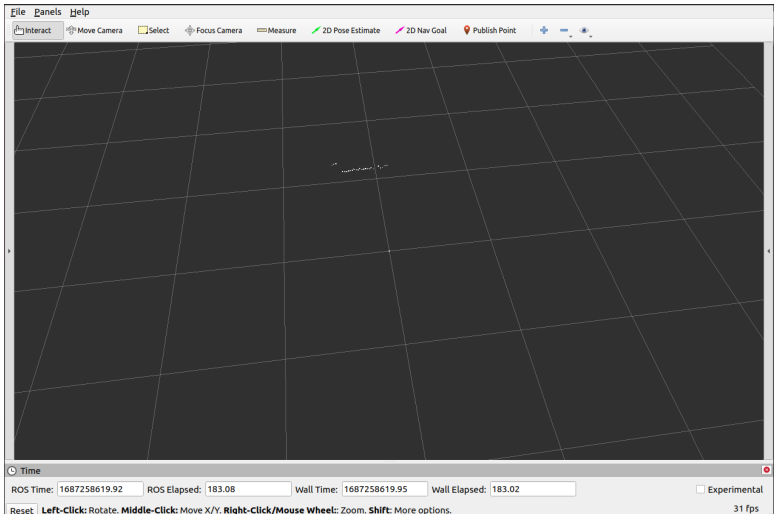
Utilisation LIDAR

Afin de pourvoir distinguer les objets entre eux, nous avons dû chercher quelles caractéristiques pouvaient les différencier compte tenu des capteurs que nous avions. Et l'une d'entre elle est la réflectivité. En effet, certains objets comme le miroir, les panneaux où la vitre sont (en théorie…) beaucoup plus réfléchissants que les cartons que nous avions.

Pour pouvoir récupérer cet indice de réflexion, nous avons utilisé le capteur lidar qui, en plus de récupérer la position des points en 3D autour du robot, permet de récupérer l'information d'intensité de réflexion de ces points.

Ainsi, nous avons récupéré le code que nous avons pu faire en TP puis le modifier pour satisfaire notre besoin. Notre approche pour pouvoir évaluer la réflectivité est assez simple : nous récupérons les points du Lidar qui sont situés devant le robot (là où est situé l'objet à mesurer), puis nous faisons la moyenne de la réflectivité de ces points et regardons si celle-ci se situe au dessus d'un seuil. Si oui, alors l'objet est considéré comme réfléchissant, sinon comme l'inverse.

Pour ce faire, nous avons divisé l'opération en 2 étape nœuds. Un premier nœud est chargé de récupérer tous les points situés dans la zone devant le robot. Pour faire cela, nous avons récupéré le code du TP sur les Lidar et avons ajouté la fonctionnalité suivante : si les points se trouvent dans une section d'angle précise, on l'ajoute à l'ensemble de points à publier, sinon on ne le fait pas. Pour récupérer la valeur de l'angle du capteur Lidar à un point précis, on peut le faire grâce au type spécial donné par ROS, LaserScan. Une fois cet ensemble de points crée, il est publié dans le topic points.



Zone détectée par le robot, ici avec un carton dans la zone de détection

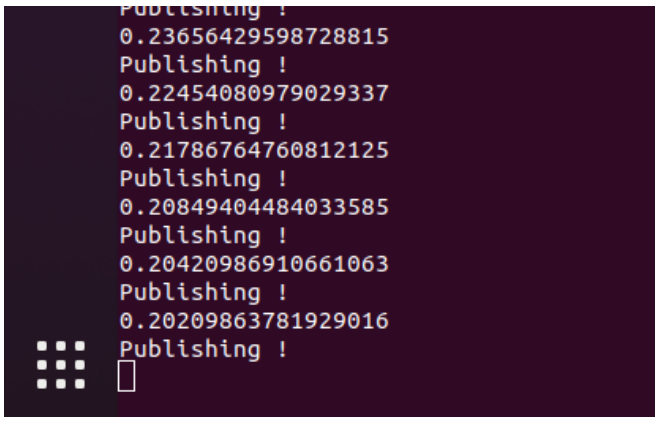
A noter que la zone à détecter est très dépendante du cadre d'expérimentation. Notre zone se situait plus "sur la gauche" compte tenu du fait que sur la droite se situait un mur. Mais dans les bags de nos camarades, cette section d'angle pouvait varier et donc ce paramètre de section d'angle est souvent à réajuster selon l'environnement.

Ensuite, la deuxième partie du traitement LIDAR commence, dans le fichier reflexion_exp.py, on récupère ce topic points et on le traite. Pour ainsi faire, nous avions deux options, soit raisonner par cluster soit directement point par point. Nous avons dans un premier temps essayé la première option mais les résultats n'étants pas probants, nous choisissons de raisonner en points par points.

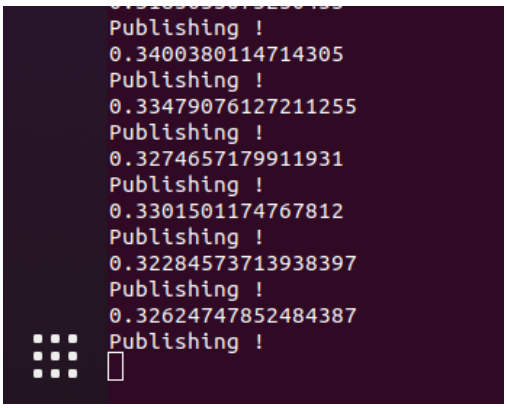
Nous avons d'abord fait une moyenne des points sur la zone à détecter mais les résultats renvoyés n'était pas satisfaisants (la différence d'intensité renvoyée entre le carton et les panneau était à peine visible). Même en pondérant pour essayer d'augmenter cet écart entre les valeurs, ceci n'était pas satisfaisants. Nous avons donc décider de raisonner avec la médiane des points, considérant que notre objet se situera au milieu de la zone de détection. Plus précisément, dans le tableau global des points de la zone, nous avons crée un "mini_cluster" (nommé comme cela dans le code source). En effet, pour la médiane,

nous ne pouvions pas nous contenter d'un point, mais nous devions utiliser un ensemble. Ainsi, ce cluster prend tous les points situés entre [taille du tableau - k] et [taille du tableau + k] où k est un paramètre d'ajustement. Enfin, on fait la moyenne de ce cluster pour avoir notre résultat de réflectivité.

Avec ceci, nous obtenons les résultats suivants :



Réflectivité avec un grand carton



Réflectivité avec le grand panneau

Même si le résultat présenté au dessus semble assez satisfaisant pour pouvoir discriminer les deux types d'objets, on s'est rendu compte que le résultat de réflectivité n'est pas très fiable. Selon que l'on approche ou recule l'objet, la réflectivité change.

On a essayé de pondérer l'intensité avec la distance mesurée en ultrason (ratio intensité/us) mais cela n'était pas probant non plus, la fonction n'étant pas linéaire. Le résultat final est donc mitigé avec le Lidar, ne nous permettant pas d'avoir un résultat fiable pour pouvoir discriminer avec certitude par exemple la vitre ou le miroir des autres objets. A noter que le cadre d'enregistrement de nos bags n'était pas idéal, étant donné que nous avons enregistré ceux-ci à coté d'un mur, ce qui impacte la mesure lidar malgré la réduction de zone de mesure. Nous avions aussi l'impression que le résultat était très variable selon les conditions d'enregistrement, l'éclairage de la pièce, l'orientation d es robots jouant beaucoup sur le résultat (nous avons fait ce constat après avoir testé notre code Lidar sur les bags de nos camarades, ne donnant pas les mêmes ordres de grandeur d'intensité).

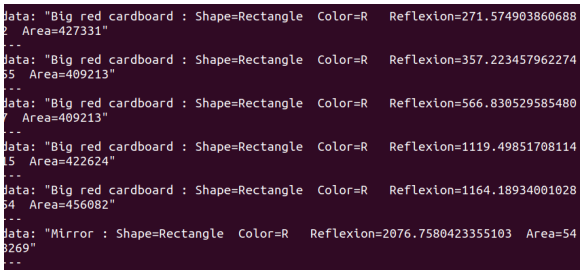
Exploitation des Ultrasons

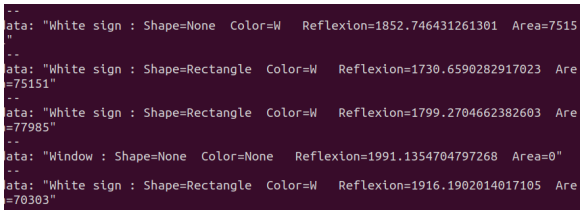
Nous avons pu utiliser le détecteur à ultrason dans ce projet notamment pour discriminer les objets de même nature, cartons de même couleur, comme expliqué dans la première partie. Nous les avons aussi utilisé pour pondérer le résultat de mesure d'intensité comme on vient de l'expliquer mais sans résultat très satisfaisants. Nous nous sommes contentés de cela pour l'exploitation de cette source d'information.

Mise en place et résultats du projet

Pour mettre en place le code final, nous avons utilisé les outils que fournissent ROS. Chaque bout de code envoie les informations d'intensité, de forme, de couleur… dans des topics que le programme final, projet.py, récupère pour ensuite les interpréter. L'interprétation des données entrantes se fait sous la forme d'un "arbre de décision" matérialisées en python par des clauses if/else. Les tests se font sur des seuils (treshold) que nous avons ajusté afin que l'on puisse discriminer les différents objets. Nous avons dans cet arbre esseyé de partir du moins discriminant au plus, c'est à dire qu'on teste d'abord l'intensité de réflexion (peu discriminant) puis la couleur (plus discriminant) puis la forme. Dans la pratique, pour contrer le manque de précision que nous avons rencontré avec le Lidar, nous avons beaucoup plus exploité la détection de couleur et de forme.

Nous avons pu obtenir les résultats ci-dessous :





Au final, notre programme détecte plutôt bien les objets mis à part le miroir et la vitre qui on beaucoup de mal à être détecter avec nos bags, en effet le miroir a parfois une intensité très faible par rapport à la vitre. Le programme est relativement robuste au niveau de certaines manipulations comme lorsque l'on rapproche un objet même s'il est quand même parfois sujet à des erreurs de discernement (il faut tout de même en général garder l'objet à une certaine distance sans trop le bouger pour obtenir un résultat satisfaisant).

A noter que ce programme est assez sensible à l'environnement et qu'il est nécessaire de recalibrer les seuils chaque fois qu'on change de cadre d'enregistrement.