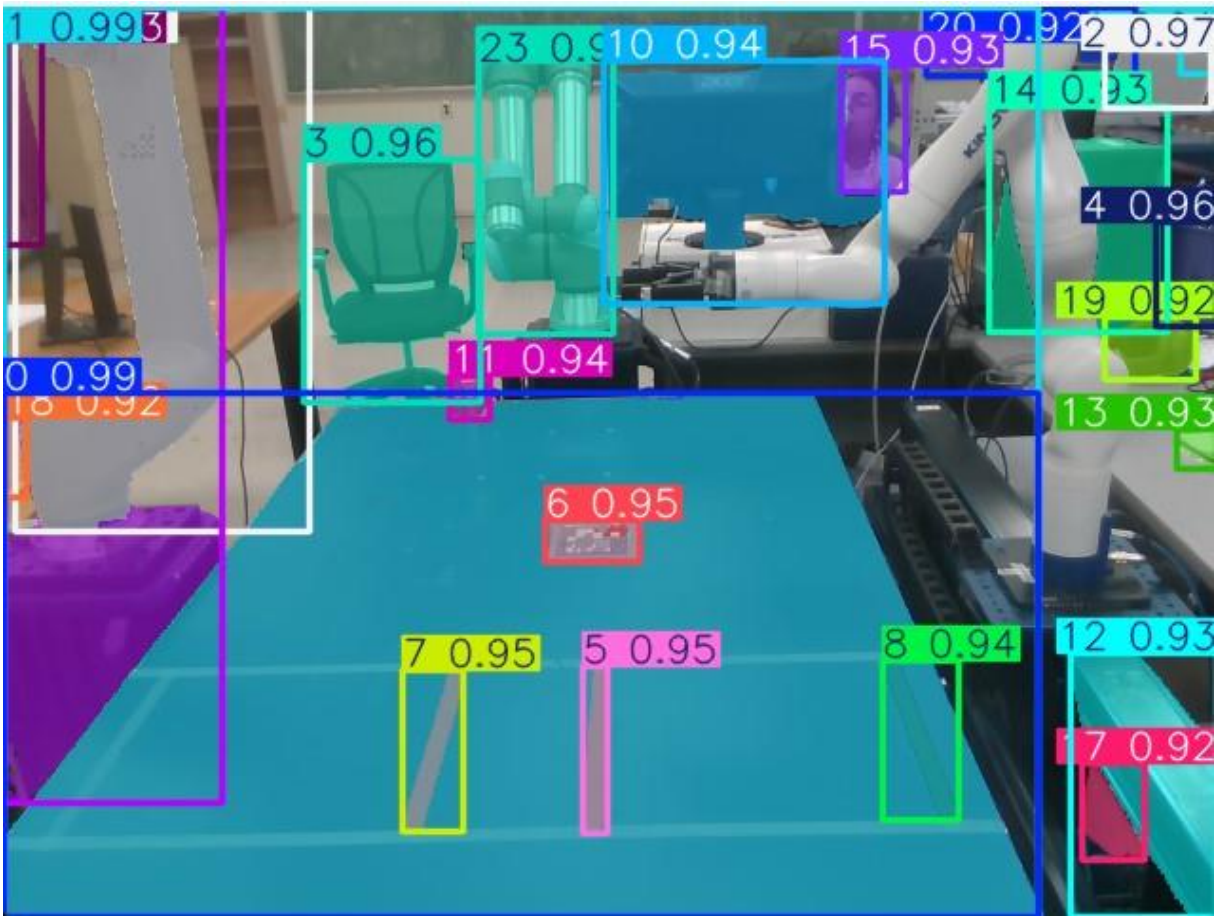
Etude de faisabilité sur L’intégration de la Segmentation à DyVision



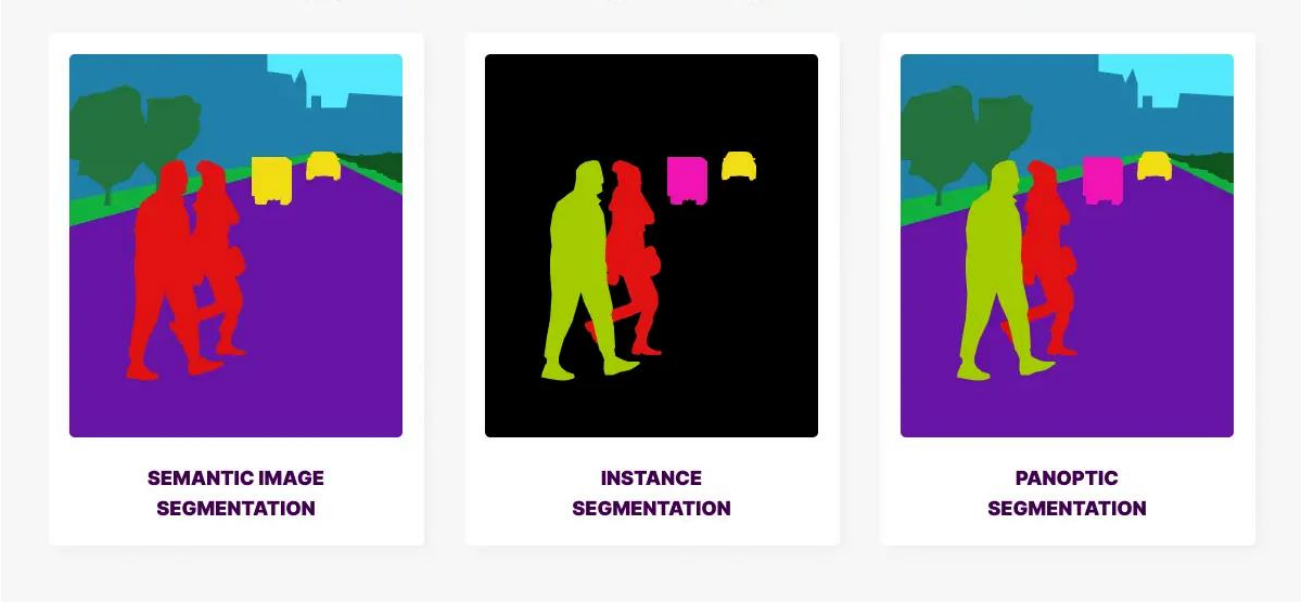
Natanael Jamet

Principe de la segmentation

La segmentation est un procédé qui permet de séparer une image en plusieurs partie dans un but d'identification d'objet.

Il existe plusieurs types de segmentation :

- Sémantique : Les instances d'une même classe ne sont pas différenciées (exemple : un groupe de personne sera segmenté en entier, les personnes individuellement ne seront pas différenciées)
- Instance : les instances de chaque classe sont différenciées. (exemple : chaque personne d'un groupe sera traitée comme des objets séparés)
- Panoptique : attribue une classe à chaque objet et différencie les instances de chaque classe.



Certaines méthodes permettent d'attribuer un label ou un index à chaque objet, permettant un suivi entre deux images d'une vidéo.

Il y a plusieurs méthodes réparties en deux groupes pour effectuer une segmentation :

- Les méthodes classiques : Clustering, intensité, niveau de gris, etc...
- Les méthodes basées sur l'intelligence artificielle : YOLO, SAM, etc...

Objectif de l'intégration de la segmentation dans DyVision

Avec DyVision 2.0, les objets ne sont pas différenciés. On identifie les zones à éviter sans faire de suivi d'un objet au cours des itérations. Pour l'identification des obstacles dans le cas de l'optimisation de trajectoire, cela suffit, mais dans le cas de la prédiction du déplacement il faudrait pouvoir suivre les objets pour récupérer les positions et vitesses par exemple.

Le but est donc d'identifier les objets grâce à une segmentation sur l'image 2D RGB puis de récupérer les nuages de points de chaque objet à partir de l'information de profondeur.

Il faut donc s'intéresser à des segmentations d'instance ou panoptique, pour avoir chaque objet séparément.

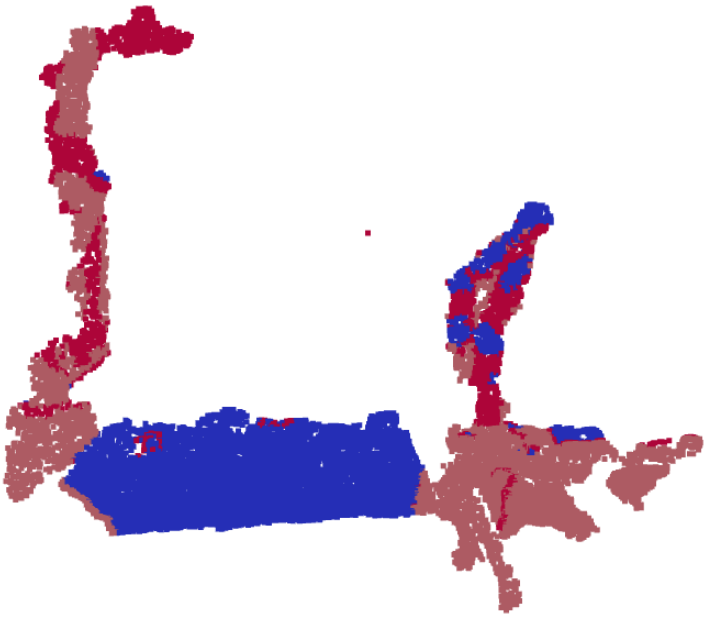
Les tests fait dans ce rapport seront faits sur la zone suivante :

Zones utilisées pour les tests

Essai de différentes méthodes

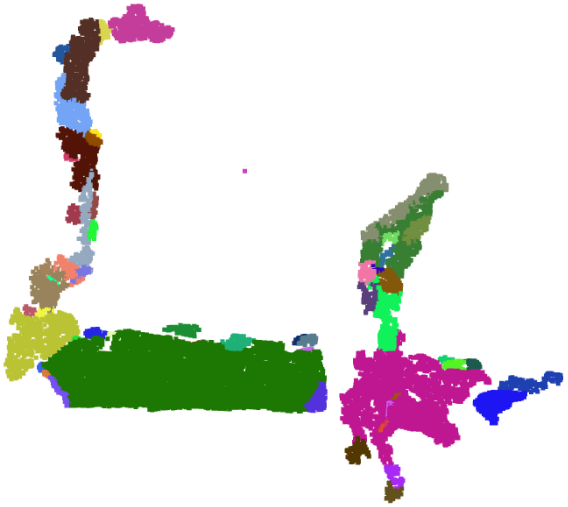
Segmentation K-mean

Il est possible d'appliquer un clustering sur les coordonnées RGB. Cela permet de réunir les points étant de couleur proche :



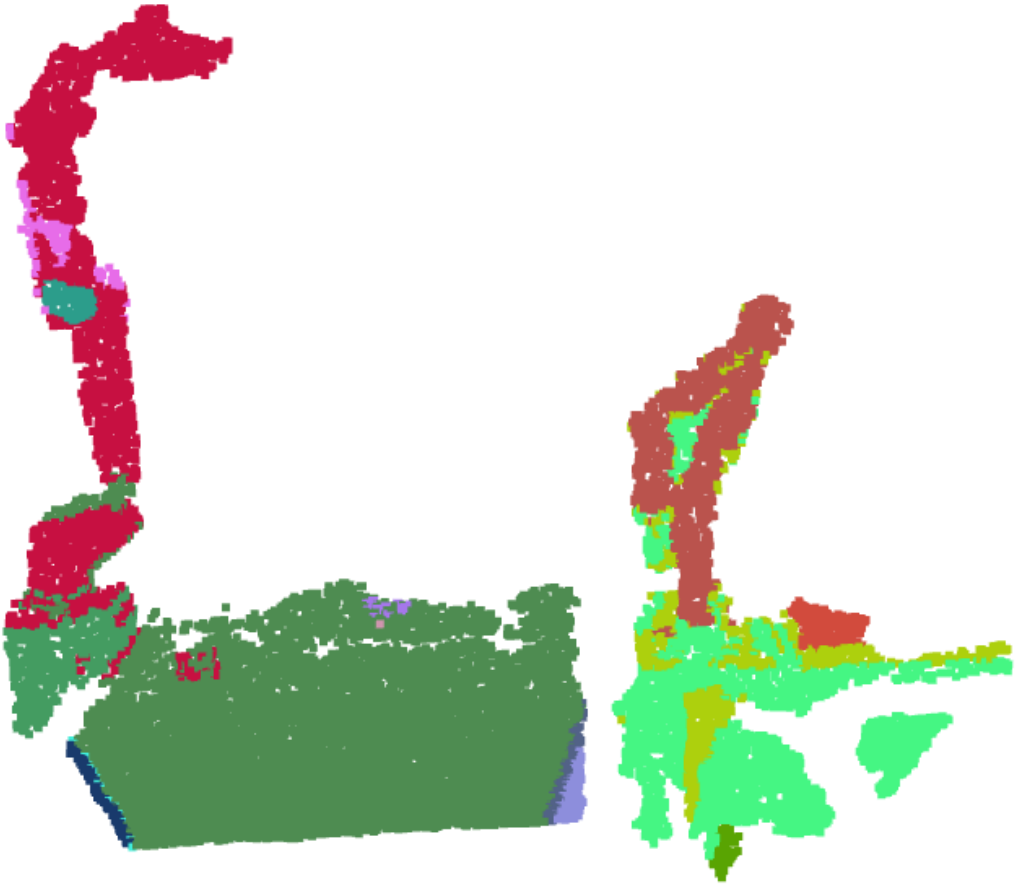
Exemple pour K=3 sur la table sans obstacles

Une fois que le clustering par couleur est fait, on peut appliquer un second clustering sur les coordonnées x, y, z pour séparer les points par critère de distance cette fois ci, en utilisant un clustering DBSCAN :

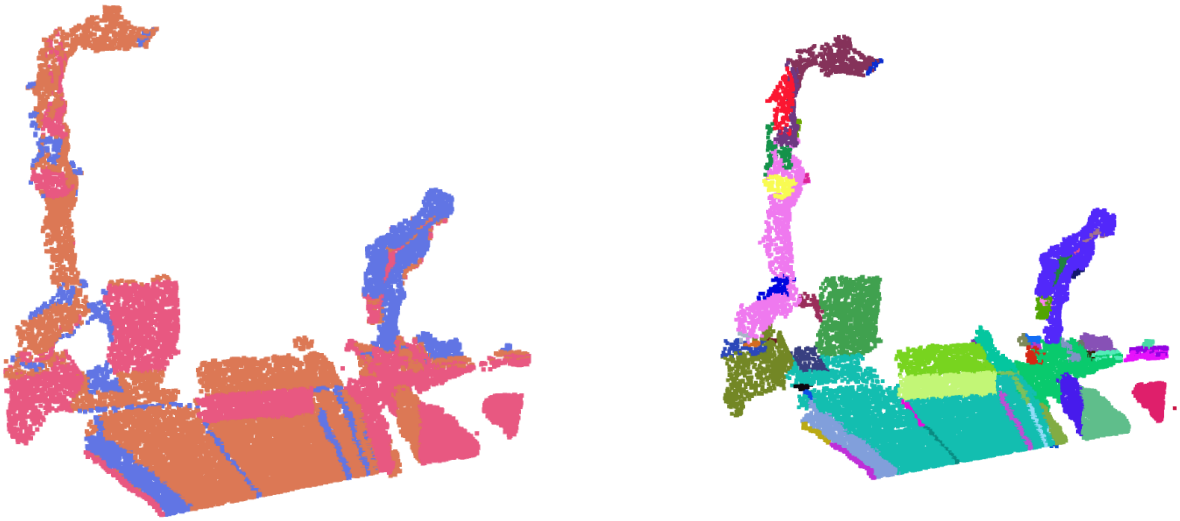


Résultat après clustering xyz

En jouant sur les paramètres on peut changer la taille des objets identifiés :



En ajoutant des objets (de couleurs unies) sur la table on obtient le résultat suivant :



Résultat avec un objet sur la table

Plusieurs choses sont à noter sur ce résultat :

- L'objet, pourtant de couleur uniforme, est reconnus comme deux objets différents. Cela malgré l'utilisation de K=3, une valeur faible qui différencie les couleurs en seulement 3 groupes.
- Il a fallu adapter les paramètres du clustering DBSCAN par rapport à la table vide pour avoir un résultat convenable.

De plus cette méthode ne permet pas directement d'attribuer un index à chaque objet. Pour cela il faudrait ajouter un suivi par similarité.

Je n'ai pas testé la mise en place d'une telle méthode, cependant voici quelques pistes :

- Il est possible de calculer la position du centre de chaque cluster identifié après les 2 clustering, puis, en comparant avec les positions des centres de l'itération précédente, attribuer l'index de l'objet de l'itération précédente le plus proche.

- Il est également possible de faire une identification des caractéristiques géométrique, on pourra alors attribuer l'index de l'objet ressemblant le plus.

Au niveau du temps de calcul, le clustering K-mean est assez rapide cependant le clustering DBSCAN et le suivi par similarité peuvent être très couteux. Il faudra trouver comment réduire les temps d'exécution de ces parties.

SAM2

SAM (Segment Anything Model) est un modèle mis en place par META. Il permet de segmenter et de suivre n'importe quel objet, même des objets sur les quels il n'a pas été entrainé.

Pour les tests que j'ai faits, j'ai utilisé la librairie ultralytics qui permet d'utiliser différents modèles de segmentation.

On charge le modèle avec :

model = FastSAM('FastSAM-s.pt')

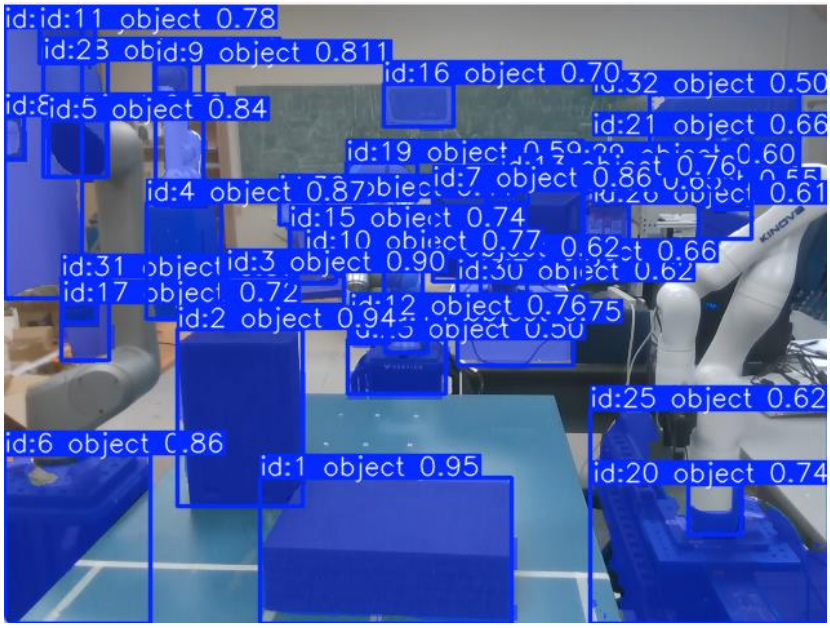
FastSAM est une version plus rapide de SAM avec un peut moins de précision.

On opérer la segmentation avec la ligne :

result=model. track (source=2, imgsz=640, save=False, show=True)

les Paramétres sont:

- Source : l'image segmentée (ici 2 permet de récupérer le flux d'une caméra branchée en USB pour tester)
- Imgsz : un paramètre qui permet de gérer la qualité de l'image (je n'ai pas identifié exactement à quoi il correspond) en le baissant on identifie moins d'objet mais c'est plus rapide
- Show : permet de dire si on veut visualiser le résultat ou non
- Save : permet de de sauvegardé ou non l'image segmentée



Résultat de la segmentation

Les informations sont sous la forme d'une liste d'objets.

Chaque objet est composé de :

- boxes : la boite affichée
- keypoints
- mask : un masque binaire et l'index
- names : le nom de l'objet identifié (si demandé)
- obb: une boite orientée
- orig_img: l'image d'origine
- orig_shape : les dimensions du tableau correspondant à l'image
- path
- probs
- save_dir : le chemin ou est sauvegardé le résultat
- speed: information sur la Vitesse

L'information qui nous intéresse est : mask, elle permet de récupérer un masque binaire, en l'appliquant sur l'image on peut isoler les objets.

Tests appliqués :

Pour tester j'ai appliqué la fonction ci-dessus. Cependant, la mémoire du GPU est vite saturée et arrête le code.

Pour pallier ça, j'ai essayé sur un ordinateur plus puissant, cela ne change rien.

J'ajoute donc les lignes suivantes :

torch. cuda. empty_cache()

os. environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'

puis j'execute le tracking avec :

with torch.no_grad () : pour éviter de calculer les gradients

Cela permet d'éviter la surcharge de la mémoire.

J’ai mis en place le traitement suivant :

Je récupère les valeurs RGB et Profondeur dans deux tableaux numpy, les coordonnées doivent correspondre entre les deux :

```
frames = pipeline.wait_for_frames()
aligned_frames = align.process(frames)
depth_frame = aligned_frames.get_depth_frame()
color_frame = aligned_frames.get_color_frame()

if not depth_frame or not color_frame:
    continue

depth_image = np.asanyarray(depth_frame.get_data())
color_image = np.asanyarray(color_frame.get_data())
```

J’applique ensuite le tracking sur le tableau RGB et récupères l’ensemble des masques binaire :

```
result = model.track(source=color_image, imgsz=226, save=False, show=False)

all_msk = []

#-------- Recup des masques binaires ----------------------------------------
for obj in result:
    if obj.masks is not None:
        for mask in obj.masks:
            mask_tensor = mask.data[0]  # Obtenir le premier masque (Tensor PyTorch)
            mask_np = mask_tensor.cpu().numpy()  # Convertir en numpy array
            mask_resized = cv2.resize(mask_np, (color_image.shape[1], color_image.shape[0]))
            mask_binary = (mask_resized > 0.5).astype(np.uint8)
            all_msk.append(mask_binary)
```

Ce qui me permet d’appliquer ces masques au tableau de Profondeur, pour récupérer les objets 3d un par un :

```
#------- Création des nuages de points pour chaque objet -------------------
point_clouds.clear()

with ThreadPoolExecutor() as executor:
    futures = {executor.submit(process_mask, mask, idx, depth_image, camera_matrix, fixed_cc
    for future in as_completed(futures):
        point_cloud = future.result()
        if point_cloud:
            point_clouds.append(point_cloud)
```

Cela fonctionne, j‘ajoute les options de visualisation pour vérifier le fonctionnement, un objet est attribué à une couleur, il devrait la garder d’une itération à l’autre. Le résultat est donné dans la vidéo jointe au document.

On remarque sur cette vidéo que l’attribution des index n’est pas stable.

Je n’ai pas réussi à identifier d’où ça peut venir, je pense cependant que c’est dû aux différents traitements que j’ai fait pour éviter la surcharge.

En réglant ce problème, il sera possible de récupérer les nuages de chaque objet avec un index attribué.

Le temps de traitements est environ de 200ms pour une carte RTX 3050, avec un ordinateur plus puissant (RTX4070 ti), on obtient un temps de 30ms, ce qui correspond la fréquence de capture de la caméra.