SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta elektrotechniky a informatiky

Ústav robotiky a kybernetiky

Počítačové videnie a spracovanie obrazu

Kinect

Tomáš Majtner

Danil Zlobin

Robotika a kybernetika

25.4.2024

Zadanie

1. Vytvorenie mračna bodov pomocou Kinect v2 pre testovanie. Nájdite online na webe mračno bodov popisujúce väčší priestor pre testovanie algoritmov a načítajte mračno dostupného datasetu.

2. Pomocou knižnice (open3d - python) načítate vytvorené mračno bodov a zobrazíte.

3. Mračná bodov očistite od okrajových bodov. Pre tuto úlohu je vhodne použit' algoritmus RANSAC.

4. Segmentujete priestor do klastrov pomocou vhodne zvolených algoritmov (K-means, DBSCAN, BIRCH, Gausian mixture, mean shift ...). Treba si zvolit' aspoň 2 algoritmy a porovnat' ich výsledky.

5. Detailne vysvetlite fungovanie zvolených algoritmov.

6. Vytvorte dokumentáciu zadania: • popis implementovaných algoritmov, • grafické porovnanie výstupov, • vysvetlite rozdiel v kvalite výstupov pre rozdielne typy algortimov

Využité technológie

Kinect v2 – kamera na vyhotovenie mračna bodov

Freenect – knižnica na prácu s kinectom

Open3d – knižnica na zobrazenie mračna bodov

RANSAC – algoritmus pre elimináciu šumov a parazitných bodov

K-means – algoritmus pre klasifikáciu bodov do skupín

DBSCAN – algoritmus pre klasifikáciu bodov do skupín (na porovnanie)

Visual studio Code – IDE využité pri riešení zadania

Python – programovací jazyk využitý pri riešení zadanie

Git – version control

Úloha 1. Vytvorenie mračna bodov

Táto úloha pozostávala z troch krokov :

1. Vytvorenie hĺbkovej snímky
2. Úprava nasnímaných bodov
3. Zkombinovanie hlbky a farby

1. Vytvorenie hĺbkovej snímky

Na vytvorenie hĺbkovej snímky sme použili knižnicu freenect2. Po inicializácií zariadenia sa nasnímajú 2 typy snímkov. Farebné a hĺbkové snimký sa následne uložia pre dalšie spracovanie.

```
device = Device()
frames = {}
with device.running():
    for type_, frame in device:
        frames[type_] = frame
        if FrameType.Color in frames and FrameType.Depth in frames:
            break
```

2. Úprava nasnímaných bodov

RGB a hĺbkové snímky sa následne zkalibrovali a uložili cez funkciu registration.

```
rgb, depth = frames[FrameType.Color], frames[FrameType.Depth]
undistorted, registered, big_depth = device.registration.apply(
    rgb, depth, with_big_depth=True)
```

3. Zkombinovanie hĺbky a farby

Ako posledné sa pomocou funkcie write zkombinovali snímky bodov z informáciami o hlbke a farbe a vytvorili .pcd mrak bodov.

```
with open('output.pcd', 'wb') as fobj:
    device.registration.write_pcd(fobj, undistorted, registered)

with open('output_big.pcd', 'wb') as fobj:
   device.registration.write_big_pcd(fobj, big_depth, rgb)
```

Úloha 2. Načítanie a zobrazenie mračna bodov

V tejto úlohe sme využili knižnice open3d a nupmy na prácu s mračnom bodov. V procese sme vyfiltrovali vsetky body, ktoré obsahujú nečíselné hodnoty.

```
def filter_nan(pcd):
    pcd_points = np.array(pcd.points)
    pcd_colors = np.array(pcd.colors)
    row_selector = ~np.isnan(pcd_points).any(axis=1)
    if row_selector.all():
        return pcd, 0
    pcd_points = pcd_points[row_selector, :]
    pcd_colors = pcd_colors[row_selector, :]

    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(pcd_points)
    pcd.colors = o3d.utility.Vector3dVector(pcd_colors)
    return pcd, (~row_selector).sum()
```

V tejto funkcií sme najprv zkonvertovali dáta z point cloudu na NumPy polia. Následne sa identifikovali všetky body v poliach, ktoré obsahovali nečíselný element. A nakoniec sme aktualizovali point cloud z vyfiltrovaných dát.

Zobrazenie mračna bodov prebehlo pomocou *read_point_cloud* funkcie, kde sa po vyfiltrovaní následne dáta zobrazia pomocou funkcie *draw.*

```
if __name__ == "__main__":
    pcd = o3d.io.read_point_cloud(sys.argv[1])
    pcd, nan_count = filter_nan(pcd)
    print(pcd)
    print(np.array(pcd.points))
    print(f"Deleted {nan_count} NaN points")
    o3d.visualization.draw([pcd])
```

3. Očistenie od okrajových bodov pomocou algoritmu RANSAC

Táto úloha pozostávala z týchto bodov :

1. Načítanie a preprocessing point cloudu
2. Segmentácia rovín pomocou RANSAC
3. Uloženie výsledkov a zobrazenie

1. Načítanie a preprocessing point cloudu

Na načítanie sme použili funkciu open3d *read_point_cloud* a pri preprocessingu sme využili našu funkciu *filter_nan* na vyfiltrovanie NaN hodnôt a odhad vzdialenosti susedných bodov.

```
pcd = o3d.io.read_point_cloud(file)
pcd, _ = filter_nan(pcd)

# estimating normals, used just for visualisation of pcd
pcd.estimate_normals(search_param=o3d.geometry.KDTreeSearchParamHybrid(radius=0.1, max_nn=16),
 fast_normal_computation=True)
```

2. Segmentácia rovín pomocou RANSAC algoritmu

RANSAC algoritmus iteratívny algoritmus používaný na odstraňovanie outlierov a odhad parametrov matematického modelu z datasetu obsahujúceho zašumené dáta. Dôležité sú vhodné hodnoty vzdialenostného tresholdu, maximálneho počtu rovín a hodnota n, ktorá definuje opčet bodov potrebných na definovanie roviny. RANSAC funguje na princípe porovnania vzdialenosti, čo znamená, že ak je bod od roviny vzdialeny menej ako treshold možeme ho považovať za súčasť danej roviny.

```
# cycle to filter out the noise points.
# after interations 'rest will contain the outliers of the planes
for i in tqdm(range(max_planes)):
    colors = plt.get_cmap("tab20")(i)
    # ransac pass on the rest of pcd
    _, inliers = rest.segment_plane(distance_threshold=distance_thr,ransac_n=3,num_iterations=1000)
    # saving the segment found
    segments.append(rest.select_by_index(inliers))
    segments_orig.append(o3d.geometry.PointCloud(segments[i]))
    # coloring segments to different colors
    segments[i].paint_uniform_color(list(colors[:3]))
    # selecting only the points that have not been choosen by ransac
    rest = rest.select_by_index(inliers, invert=True)
```

3. Uloženie výsledkov a zobrazenie

Všetky roviny uložene do jednotlivých segmentov musíme nasledne spojit do jedného point cloudu.

```
filtered_orig = segments_orig[0]
for s in segments_orig:
    filtered_orig += s

# re-color noisy point cloud, just to save it in red
rest_red = o3d.geometry.PointCloud(rest)
rest_red.paint_uniform_color([1,0,0])

# writing results down
o3d.io.write_point_cloud(os.path.join(script_folder, "results", name), filtered)
```

4. Segmentujete priestor do klastrov pomocou algoritmov k-means a DBscan

DBscan algoritmus

Algoritmus DBscan rozdeluje body na :

Core point

- hlavný bod, ktorý je obklopený minimalne MinPts množstvom susedných bodov v radiuse epsilon

Border point

- je to hraničný bod, ktorý nie je core point ale je v rozsahu epsilon od core pointu

Noise point

- je to bod,ktorý nie je ani core a nie je ani v rozsahu epsilon od core pointu, nepatrí nikam a považuje sa za šum

Kroky algoritmu :

1. Inicializácia
   - Výber parametrov epsilon a minPts
   - Nastavenie všetkých bodov na nenavštívené
2. Dosiahnuteľnosť bodov
   - Ak bol bod navštívený, presuň sa na další bod
   - Nastav bod ako navštívený
   - Nájdi všetky body v rozsahu epsilon a označ ich ako susedov
   - Pokiaľ má bod aspoň minPts susedov (vrátane seba) je core point
   - Vytvorí sa nový cluster pre bod a rekurzívne sa expanduje pridaním všetkých bodov, ktoré su dosiahnuteľné
3. Clusterizacia
   - Proces sa opakuje pre všetky nenavštívené body datasetu
   - Pokiaľ bod nie je dosiahnuteľný zo žiadneho core pointu je označený za šum

```
DBScan_eps = 0.03
# Clustering using DBSCan
labels = np.array(pcd.cluster_dbscan(eps=DBScan_eps, min_points=5))
mlabel = labels.max()
print(f"DBScan: {mlabel + 1} clusters")
colors = plt.get_cmap("tab20")(labels / (mlabel if mlabel > 0 else 1))
colors[labels < 0] = 0
pcd.colors = o3d.utility.Vector3dVector(colors[:, :3])
os.makedirs(os.path.join(script_folder, "results"), exist_ok=True)
name = os.path.basename(file).replace(".pcd", "_dbscan_clusters.pcd")
o3d.io.write_point_cloud(os.path.join(script_folder, "results", name), pcd)
o3d.visualization.draw_geometries([pcd])
```

K-means algoritmus

Algoritmus strojového učenia bez učitela, ktorý narozdiel od DBscan algoritmu má dopredu určený počet clustrov, ktoré sú okoloce centroidu.

Kroky algoritmu :

1. Inicializácia
   - Vhodne si vybrať množstvo clustrov, ktoré chceme identifikovať
   - Inicializácia centroidov čisto náhodne, ktoré budú reprezentovať stred clustrov
2. Priradenie bodov ku clustrom
   - Vypočítanie Euclidovskej vzdialenosti medzi bodom a každým centroidom
   - Priradenie bodu ku clustru, ktorého centroid je najbližšie
3. Aktualizácia centroidov
   - Po tom čo sú všetky body priradené ku clustrom sa centroidy zmenia aby presnejšie vyjadrovali stred clustru
4. Opakovanie bodov 2. a 3.
   - Opakovanie priradenia a aktualizácie centroidov pokiaľ nenastane konvergencia, čo je buď nedostatočná zmena centrodiov alebo dosiahnutie určitého počtu iterácií.

```
# Amount of clusters for K-Means
KMeans_n = 10
# Clustering using K-Means
kmeans = KMeans(n_clusters=KMeans_n, random_state=0, n_init="auto")
print(f"K-Means: set {KMeans_n} clusters")
# Taking points from pcd
pcd_npdata = np.array(pcd.points)
kmeans = kmeans.fit(pcd_npdata)
colors_km = plt.get_cmap("tab20")(kmeans.labels_ / KMeans_n)
# coloring points
pcd.colors = o3d.utility.Vector3dVector(colors_km[:, :3])
name = os.path.basename(file).replace(".pcd", "_kmeans_clusters.pcd")
o3d.io.write_point_cloud(os.path.join(script_folder, "results", name), pcd)
```

Porovnanie RANSAC vs. DBscan vs. K-means

Účel:

- RANSAC: Robustné odhadovanie modelov v prítomnosti odľahlých hodnôt (outlierov).
- DBSCAN: Zhlukovanie založené na hustote pre priestorové dáta.
- K-means: Rozdelenie dát do k zhlukov na základe podobnosti.

Spracovanie šumu a odľahlostí:

- RANSAC: Identifikuje a ignoruje odľahlé hodnoty počas odhadovania modelu.
- DBSCAN: Dokáže detegovať a označiť odľahlé hodnoty ako noise body.
- k-means: Je citlivý na šum a odľahlé hodnoty; zvyčajne ich priradí k najbližšiemu zhluku.

Tvar zhlukov:

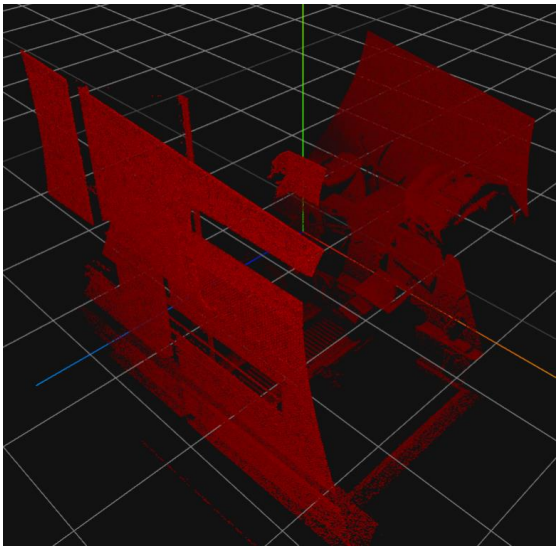
- RANSAC: Typicky sa používa na prispôsobovanie lineárnych alebo geometrických modelov (priamky, roviny).
- DBSCAN: Dokáže identifikovať zhluky s ľubovoľným tvarom.
- k-means: Predpokladá, že zhluky sú sférické a môže mať problémy s nelineárnymi alebo nepravidelnými tvarmi zhlukov.

Požiadavky na parametre:

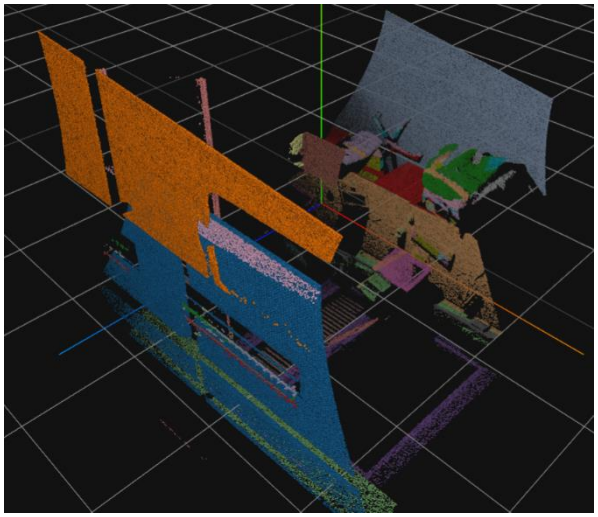
- RANSAC: Vo všeobecnosti si vyžaduje určenie parametrov súvisiacich s prispôsobovaním modelu.
- DBSCAN: Vyžaduje nastavenie parametrov ako ε (polomer okolia) a MinPts (počet bodov).
- k-means: Vyžaduje určenie počtu zhlukov (k) vopred a voľbu vhodnej metódy inicializácie centroidov.

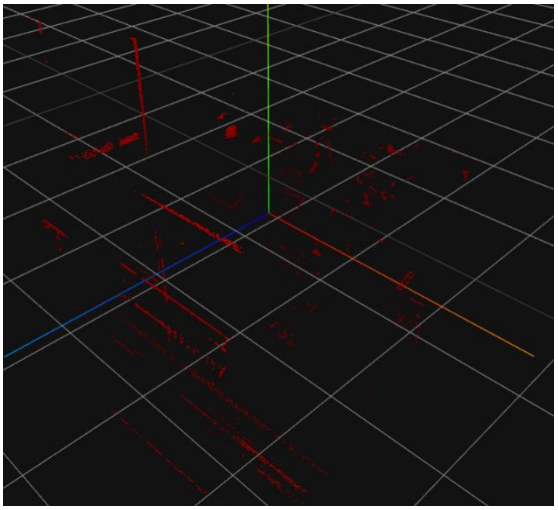
Porovnanie výsledkov

1. raw data

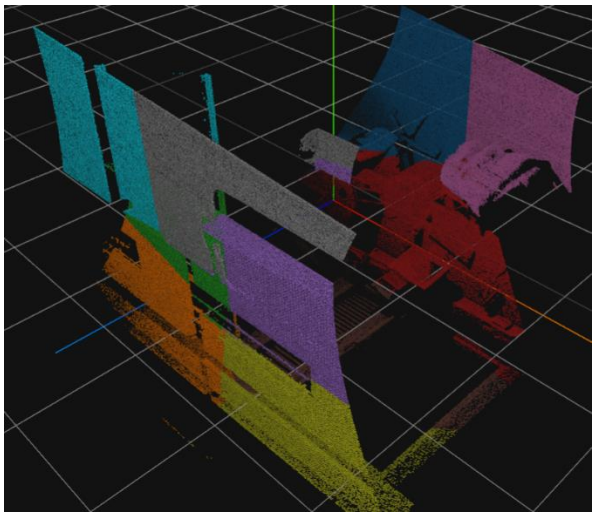


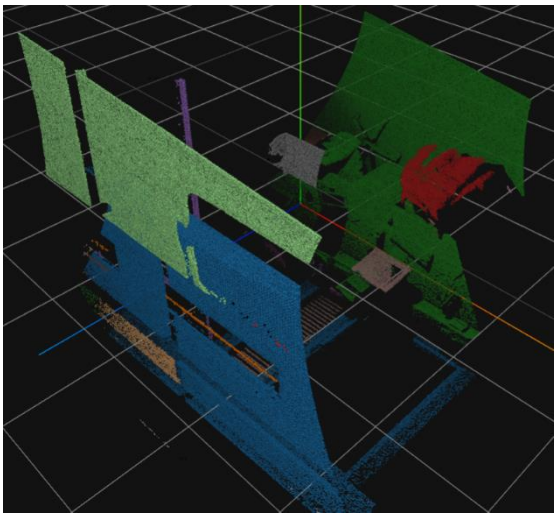
2. RANSAC



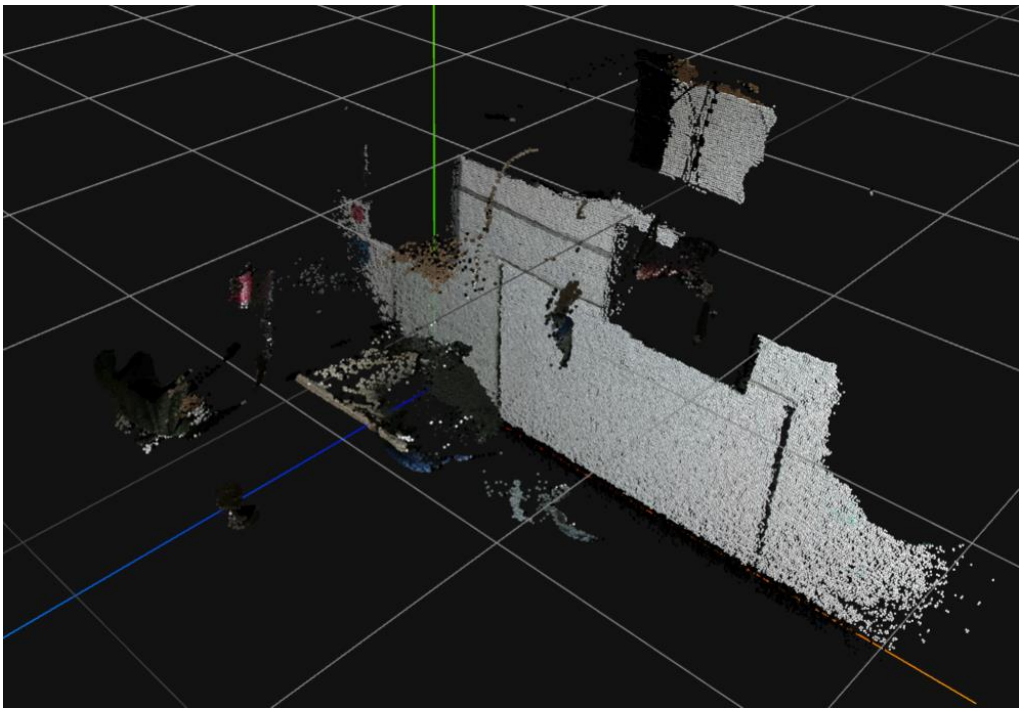


3. K- means vs DBscan

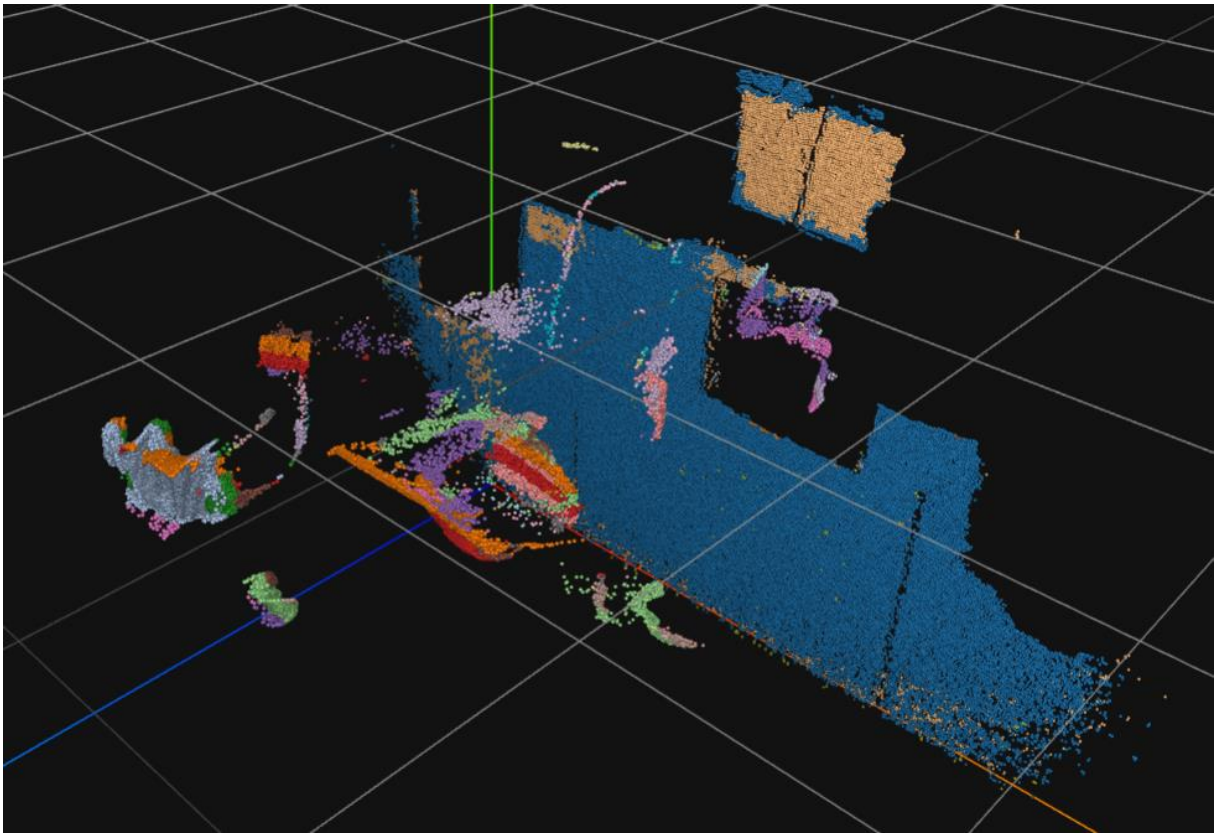




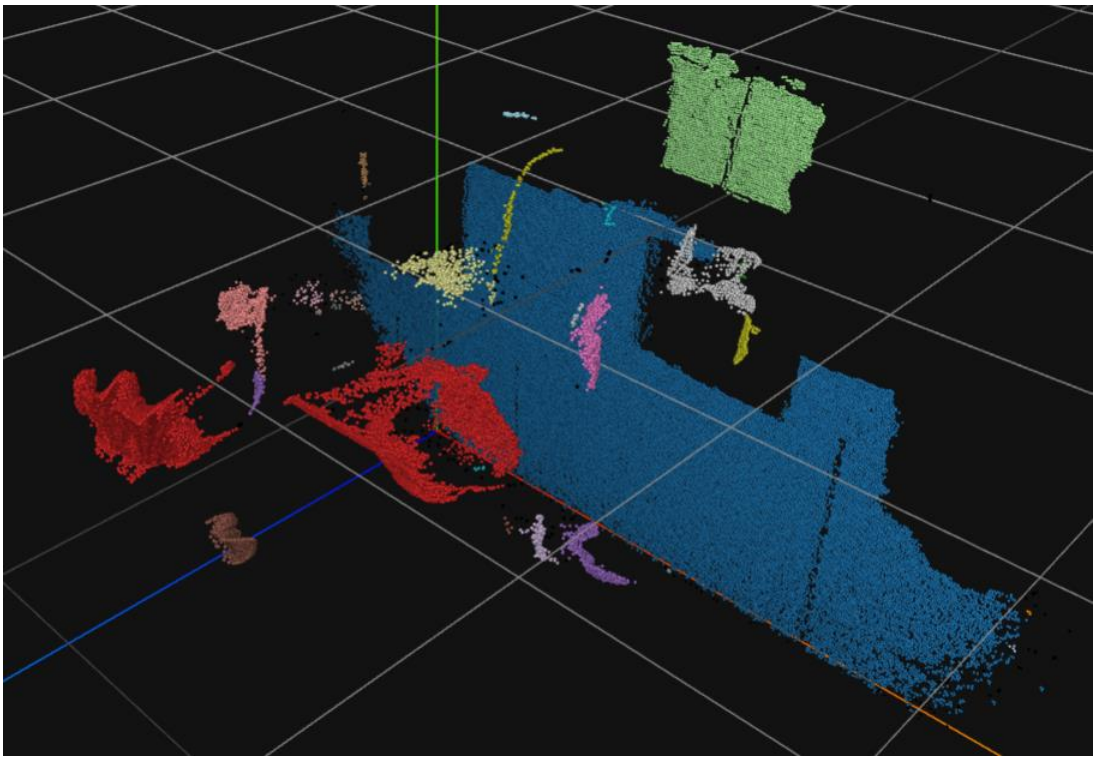
1. Raw data



2. RANSAC



3. DBscan



4. K-means