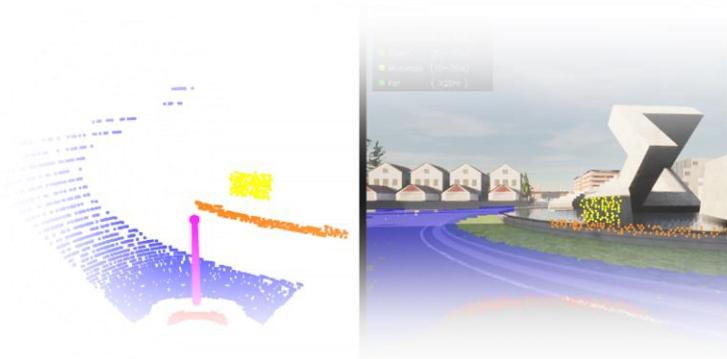


CAMERA VS LIDAR

ΥΠΟΛΟΓΙΣΤΙΚΗ ΓΕΩΜΕΤΡΙΑ &
Εφαρμογές 3Δ Μοντελοποίησης



Γέροντας Νικόλαος - Α.Μ.: 1092813

Περιεχόμενα

Μέρος Α:	2
i) Ανιχνεύστε τα όρια του δρόμου και χρωματίστε διαφορετικά τις 2 λωρίδες κυκλοφορίας όπως και τις περιοχές εκτός του δρόμου.	2
Επίλυση με μηχανική μάθηση:	2
Επίλυση με monolen κάμερα και GrabCut τεχνικές:	3
Επίλυση με stereo κάμερα και disparity:	6
ii) Ανιχνεύστε τα αντικείμενα/εμπόδια που βρίσκονται στο δρόμο, πχ αυτοκίνητα, πεζοί, άλλα εμπόδια. Δε χρειάζεται να κάνετε κατηγοριοποίηση.	9
Αρχική υλοποίηση	9
Επίλυση με Yolo και βελτίωση DBSCAN με παράλληλη επεξεργασία:	11
iii) Υπολογίστε διάνυσμα κίνησης του αυτοκινήτου προς τη σωστή κατεύθυνση. Αν υπάρχουν εμπόδια απεικονίστε έναν κύκλο.	13
Μέρος Β:	15
i) Χρησιμοποιήστε το 3D point cloud για να ανιχνεύσετε το δρόμο και τα όριά του. Κατηγοριοποιήστε και χρωματίστε διαφορετικά τα αντίστοιχα σημεία.	15
ii) Ανιχνεύστε τα αντικείμενα/εμπόδια που βρίσκονται στο δρόμο, πχ αυτοκίνητα, πεζοί, άλλα εμπόδια. Δε χρειάζεται να κάνετε κατηγοριοποίηση.	17
iii) Υπολογίστε διάνυσμα κίνησης του αυτοκινήτου προς τη σωστή κατεύθυνση. Αν υπάρχουν εμπόδια απεικονίστε έναν κύκλο.	19
Ερωτήματα Aiv και Biv - Τοίχος	21
Σχολιασμός αποτελεσμάτων	23
Real Time Simulation	24
Οδηγίες εγκατάστασης	25
Βιβλιογραφία	26

Github link: <https://github.com/Nick-744/Camera Vs Lidar 3D/tree/main>

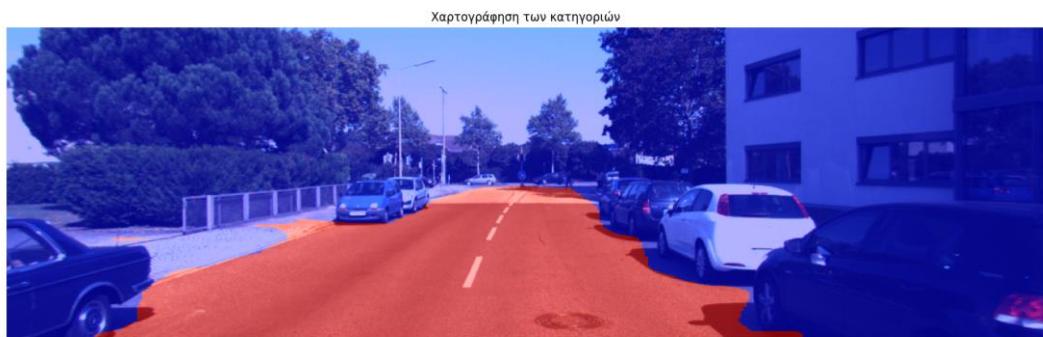
Μέρος Α:

i) Ανιχνεύστε τα όρια του δρόμου και χρωματίστε διαφορετικά τις 2 λωρίδες κυκλοφορίας όπως και τις περιοχές εκτός του δρόμου.

Το συγκεκριμένο ερώτημα υλοποιήθηκε με πολλούς και διαφορετικούς τρόπους. Η 1^η προσπάθεια ανίχνευσης του δρόμου πραγματοποιήθηκε χρησιμοποιώντας μηχανική μάθηση. Αναλυτικότερα:

Επίλυση με μηχανική μάθηση:

Χρησιμοποιώντας **SCNN** (Sequential Convolutional Neural Network), συγκεκριμένα το DeepLabV3 model με MobileNetV3-Large ως backbone, είχα τα εξής αποτελέσματα [ενδεικτικά]:

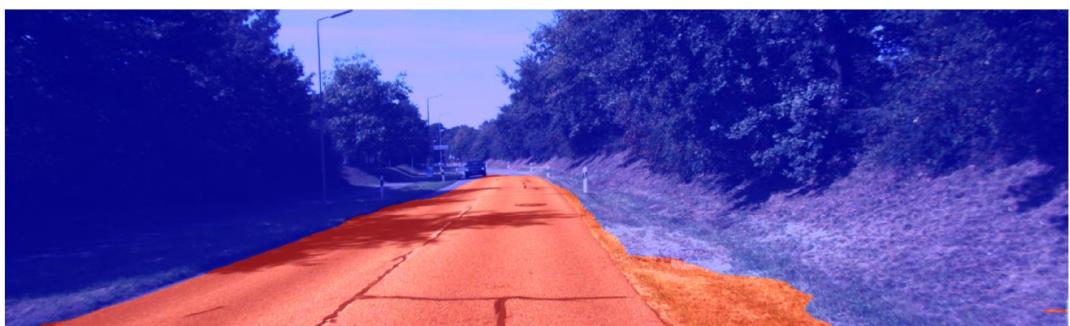


Επεξεργασία εικόνας um_000004.png σε 0.20 δευτ.

Αρχείο: abandoning_ml_for_cv\deeplab_v3_p\inference.py



Επεξεργασία εικόνας um_000008.png σε 0.20 δευτ.



Επεξεργασία εικόνας um_000041.png σε 0.22 δευτ.

Φυσικά, το μοντέλο τροποποιήθηκε κατάλληλα έτσι ώστε να μπορεί να κάνει διαχωρισμό μόνο μεταξύ του δρόμου και της υπόλοιπης εικόνας. Η υλοποίηση βρίσκεται στο αρχείο:

abandoning_ml_for_cv\deeplab_v3_p\my_model.py

Συγκεκριμένα, άλλαξα το τελευταίο layer του με 1 Conv2d που έχει μόνο 2 εξόδους (κλάσης) και όρισα το kernel size να έχει την τιμή 1 ώστε το νευρωνικό να μπορεί να πετύχει pixel-wise αναγνώριση! Το input size της εικόνας έχει οριστεί σε 368x368 ώστε να υπάρχει ισορροπία μεταξύ της πληροφορίας που δίνουμε και της ταχύτητας της αναγνώρισης.

Παράλληλα, λόγω του ότι το KITTI dataset δεν είχε ικανοποιητικό πλήθος δεδομένων για εκπαίδευση νευρωνικού (τουλάχιστον, βάση της εμπειρίας μου), χρησιμοποίησα το CARLA simulation μαζί με το αρχείο *abandoning_ml_for_cv\carla_sim\weather_traffic_dataset_combo.py* για να εξάγω KITTI-like δεδομένα εκπαίδευσης. Ενδεικτικά:



Η παραπάνω μάσκα είναι εφικτό να υπάρξει βάση της κατηγοριοποίησης που προσφέρει το CARLA.

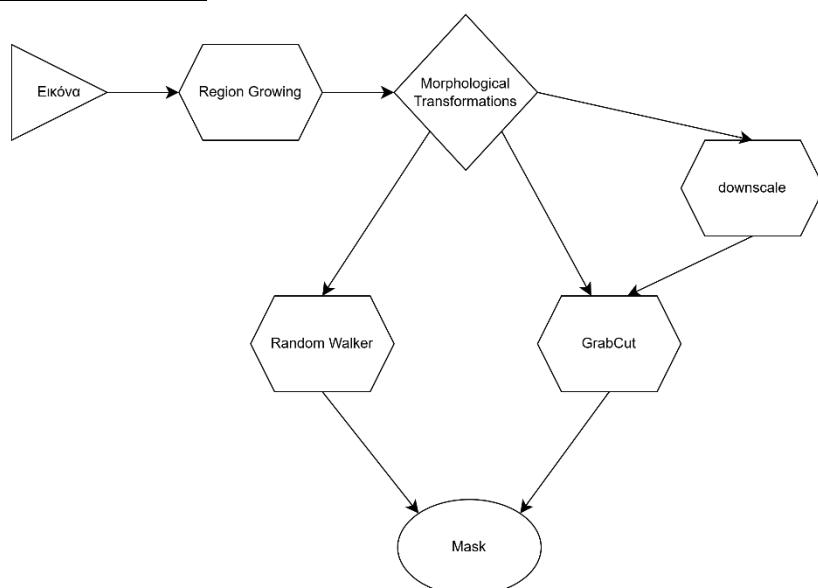
Συλλέγοντας 6000 segmentation παραδείγματα (δηλαδή, 12000 εικόνες συνολικά) και εκπαιδεύοντας το νευρωνικό για 60 εποχές με batch size = 16 (6 ώρες εκτέλεση), προέκυψε ένα δίκτυο που είχε ως αποτελέσματα αυτά που είδαμε στην αρχή της υποενότητας.

Επίλυση με monolen κάμερα και GrabCut τεχνικές:

Αρχείο: *pure_cv_approach\Ai_road_finder\Ai_region_growing.py*

Το pipeline που ακολούθησα για τον συγκεκριμένο τρόπο επίλυσης, είναι:

[1]. Εξαγωγή της μάσκας του δρόμου:



Διάγραμμα της συνάρτηση *my_road_detection*

Αναλυτικότερα, μετά τα Morphological Transformations υπάρχουνε 3 επιλογές: GrabCut, fast GrabCut (μέσω downscaled εικόνας) και Random Walker. Το πιο ισορροπημένο μεταξύ ταχύτητας και αποτελέσματος είναι το **GrabCut fast**.

[2]. Εύρεση του κυρτού περιβλήματος της μάσκας:

Αμέσως μετά, η παραγόμενη μάσκα του δρόμου δίνεται ως είσοδος στη συνάρτηση `find_mask_vertices`, η οποία επιστρέφει τις κορυφές της μάσκας. Οι κορυφές αυτές χρησιμοποιούνται από τον αλγόριθμο Graham Scan για τον υπολογισμό του κυρτού περιβλήματος της περιοχής του δρόμου.

[3]. Διαχωρισμός των 2 λωρίδων:

Η συνάρτηση `lane_separation` εφαρμόζει Canny (edge detector) στην εικόνα εισόδου και στη συνέχεια χρησιμοποιεί Hough Line Transform. Από τις γραμμές που βρέθηκαν, όποια έχει το καλύτερο score βάσει της τοποθέτησής της ως προς το κέντρο της μάσκας, επιστρέφεται ως η διαγράμμιση του δρόμου.

[4]. Τεμαχισμός του κυρτού περιβλήματος:

Βάσει των 2 προηγούμενων αποτελεσμάτων [**κυρτό περιβλήμα** και **διαχωριστική γραμμή**], η συνάρτηση `split_convex_hull` τεμαχίζει το κυρτό περιβλήμα σε 2 μέρη, την αριστερή και δεξιά λωρίδα.

Αποτελέσματα με GrabCut fast [ενδεικτικά]:



Διάρκεια εκτέλεσης: 0.96 sec

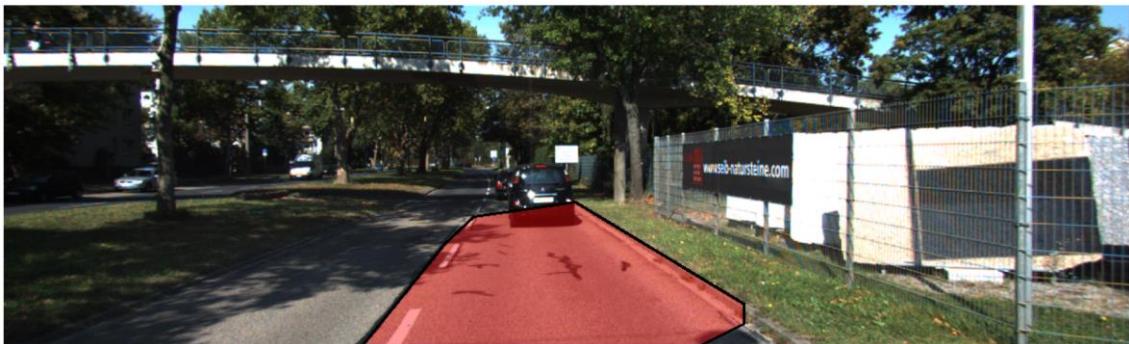


Διάρκεια εκτέλεσης: 1.00 sec

Αποτελέσματα με GrabCut [ενδεικτικά]:



Διάρκεια εκτέλεσης: 8.85 sec



Διάρκεια εκτέλεσης: 5.59 sec

Δεν βρέθηκε αξιόπιστη διαχωριστική γραμμή! Απλός σχεδιασμός δρόμου...

Παρατηρείται ότι η παρουσία σκιάς συχνά επηρεάζει αρνητικά την ποιότητα της αναγνώρισης του δρόμου. Επιπλέον, ο χρόνος εκτέλεσης είναι σημαντικά αυξημένος - έως και 5 έως 10 φορές μεγαλύτερος σε σύγκριση με τη μέθοδο GrabCut fast!

Random Walker αποτέλεσμα:



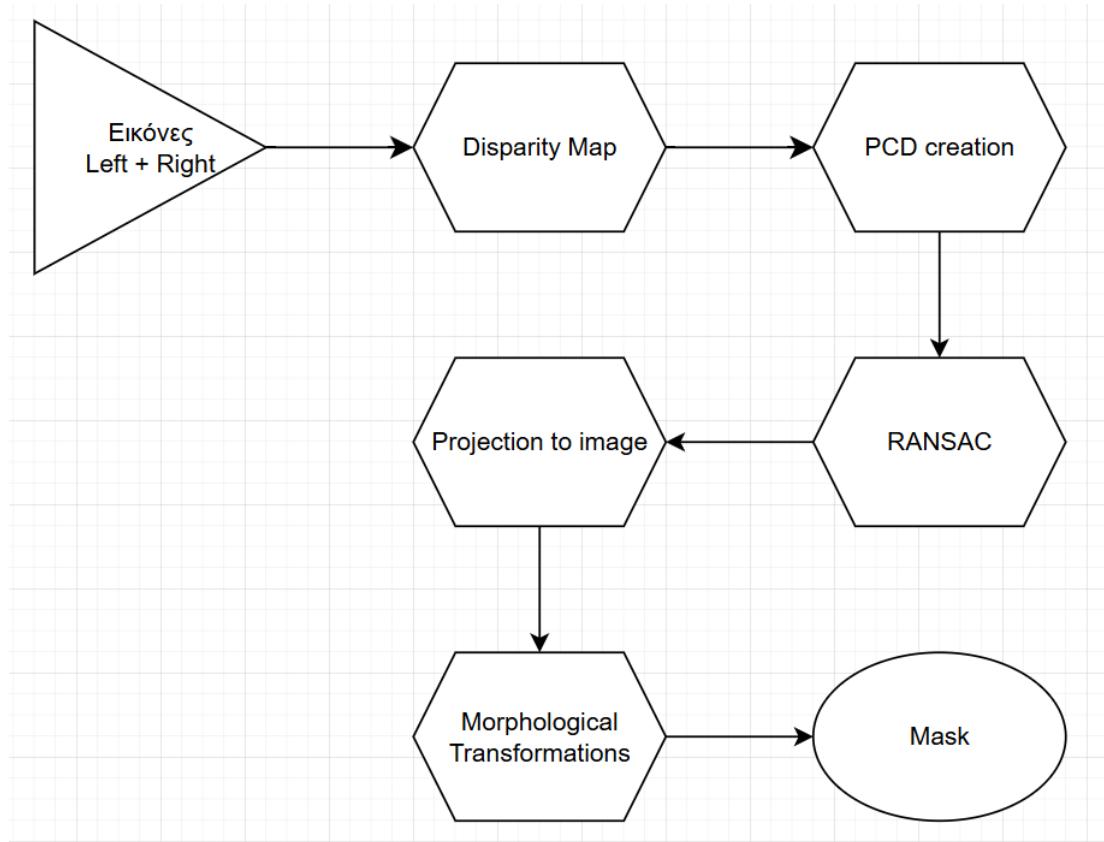
Διάρκεια εκτέλεσης: 0.27 sec

Δεν βρέθηκε αξιόπιστη διαχωριστική γραμμή! Απλός σχεδιασμός δρόμου...

Παρατηρούμε ότι αποτελεί την πιο γρήγορη αναγνώριση από τις 3 επιλογές, αλλά με σχετικά μέτρια αποτελέσματα. Αξίζει να σημειωθεί ότι η τεχνική Random Walker αξιοποιεί και τη συνάρτηση compute_c1_channel για την αντιμετώπιση των προβλημάτων που προκαλούνται από την παρουσία σκιών.

Επίλυση με stereo κάμερα και disparity:

Λόγω των προβλημάτων που εμφάνισε η επίλυση με μία μόνο κάμερα (αδυναμία εύρεσης δρόμου σε περιοχές με σκιά **και**, **κυρίως**, **ο μεγάλος χρόνος εκτέλεσης**), κατέφυγα στην επίλυση με stereo camera setup (που προσφέρει το KITTI dataset) και χρήση disparity map. Το pipeline που ακολούθησα τελικά για τον συγκεκριμένο τρόπο επίλυσης, είναι:



➤ **Disparity Map:** Χρησιμοποίησα την build-in συνάρτηση StereoSGBM_create της βιβλιοθήκης opencv-python.

➤ **PCD creation:** Η συνάρτηση point_cloud_from_disparity υπολογίζει για κάθε έγκυρο pixel (u, v) με γνωστή ανομοιότητα d , τις 3D συντεταγμένες (**X**, **Y**, **Z**) του αντίστοιχου σημείου στον χώρο, χρησιμοποιώντας τις εξισώσεις:

$$Z = f \cdot \frac{T_x}{d} \quad X = \frac{(u - c_x)}{f} \cdot Z \quad Y = \frac{(v - c_y)}{f} \cdot Z$$

f : focal length

T_x : baseline

d : disparity

c_x, c_y : κέντρο προβολής της κάμερας

➤ **RANSAC:** Χρησιμοποίησα την build-in συνάρτηση segment_plane της open3d βιβλιοθήκης.

➤ **Projection to image:** Η συνάρτηση project_points_to_mask υλοποιεί την αντίστροφη διαδικασία της point_cloud_from_disparity, προβάλλοντας τα 3D σημεία πίσω στο επίπεδο της εικόνας!

➤ **Morphological Transformations:** Επεξεργασία της μάσκας με εργαλεία της OpenCV για καλύτερη εμφάνιση.

Αποτελέσματα:



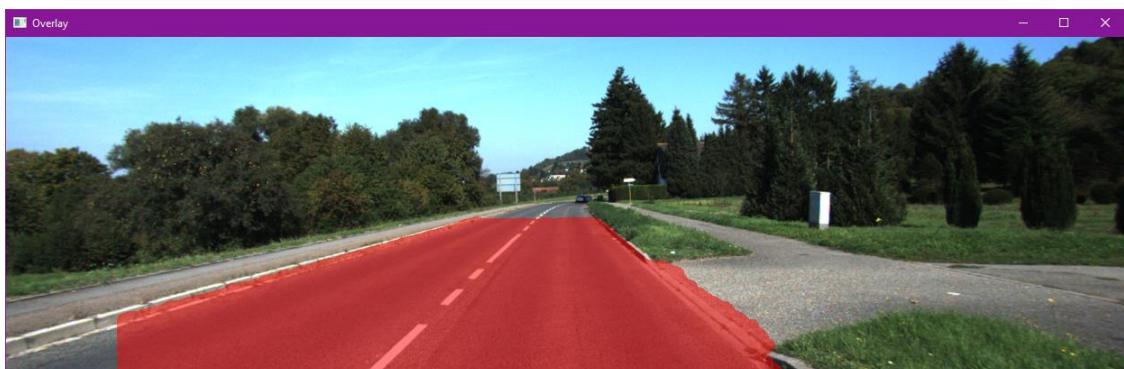
Διάρκεια εκτέλεσης: 0.13 sec



Διάρκεια εκτέλεσης: 0.15 sec



Διάρκεια εκτέλεσης: 0.11 sec



Διάρκεια εκτέλεσης: 0.10 sec

Αρχείο: *pure_cv_approach\Ai_road_finder\Ai_from_disparity.py*

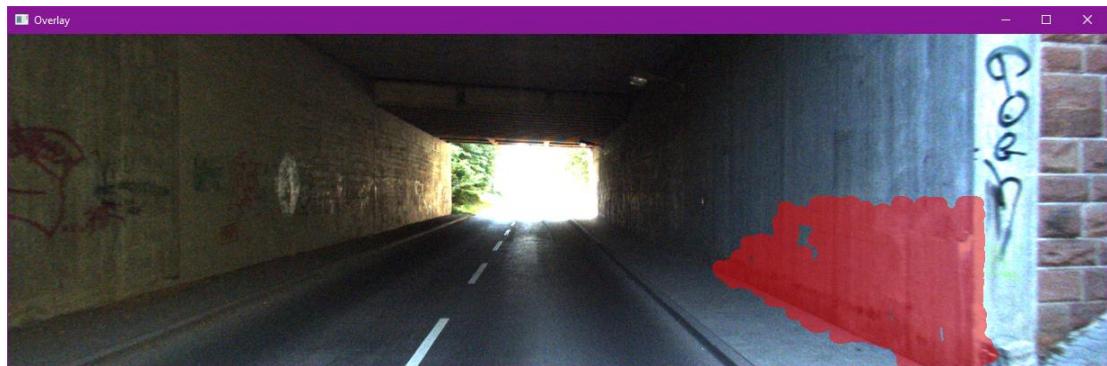
Αποτυχημένες περιπτώσεις:



Διάρκεια εκτέλεσης: 0.13 sec



Διάρκεια εκτέλεσης: 0.10 sec



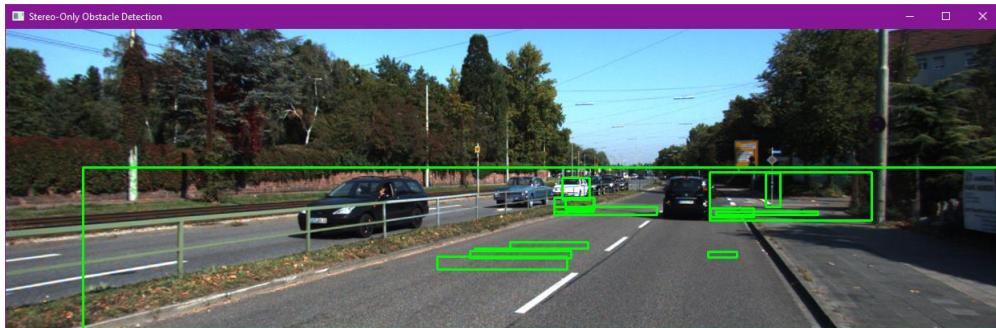
Διάρκεια εκτέλεσης: 0.39 sec

Με το συγκεκριμένο pipeline, παρατηρούμε ότι η ταχύτητα επεξεργασίας πρακτικά δεκαπλασιάστηκε σε σχέση με την μέθοδο GrabCut fast και το εύρος των σωστών αποτελεσμάτων είναι σαφώς μεγαλύτερο! Αποτελεί την τελική υλοποίησή μου για το ερώτημα Ai.

ii) Ανιχνεύστε τα αντικείμενα/εμπόδια που βρίσκονται στο δρόμο, πχ αυτοκίνητα, πεζοί, άλλα εμπόδια. Δε χρειάζεται να κάνετε κατηγοριοποίηση.

Η **αρχική υλοποίηση** στο παρόν ερώτημα έγινε με χρήση του αλγορίθμου DBSCAN. Συγκεκριμένα, έχοντας ως βάση το pipeline που περιεγράφη στην υποενότητα [Επίλυση με stereo κάμερα και disparity](#), προστέθηκαν επιπλέων οι εξής συναρτήσεις για την ανίχνευση αντικειμένων:

filter_by_height: βάση της εξίσωσης επιπέδου για τον δρόμο που προκύπτει μέσω RANSAC, φιλτράρουμε τα σημεία των εμποδίων (obstacle_pts) βάση ύψους. Η διαδικασία αυτή είναι απαραίτητη λόγω ατελειών στο point cloud! **Παράδειγμα αποτελέσματος χωρίς την εφαρμογή του φίλτρου ύψους**:



Χρόνος εκτέλεσης: 8.35 sec

Παράλληλα, εφαρμόζοντας το φίλτρο, μειώνουμε τον αριθμό των σημείων που πρέπει να επεξεργαστούν, άρα και τον χρόνο εκτέλεσης.

cluster_obstacles_dbSCAN: η βασική συνάρτηση του ερωτήματος για την εύρεση των εμποδίων. Χρησιμοποιήθηκε η DBSCAN συνάρτηση της βιβλιοθήκης sklearn.cluster ως υλοποίηση.

group_clusters: ομαδοποιεί τα clusters που έχουνε ίδιο label. Αγνοεί όσα δεν έχουνε κατηγοριοποιηθεί.

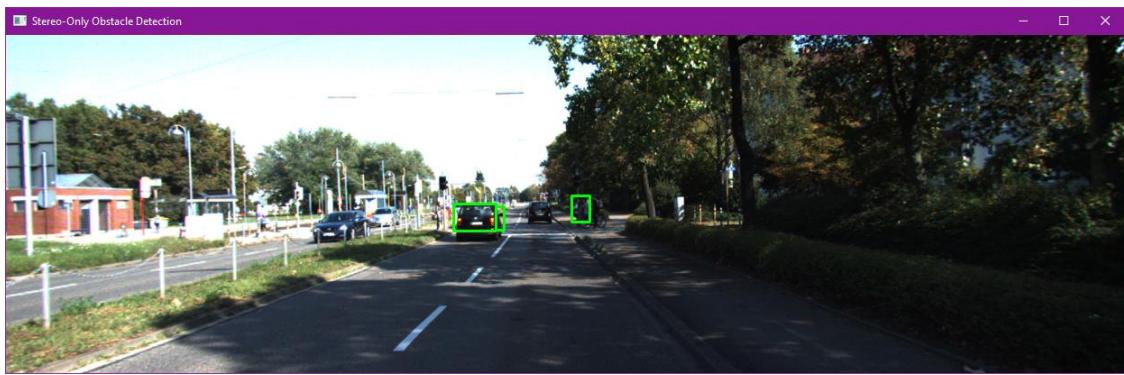
is_obstacle_on_road ή **filter_boxes_by_road_mask**: η κύρια διαφορά μεταξύ τους είναι ότι η 1^η συνάρτηση εντοπίζει τα clusters που βρίσκονται πάνω στον δρόμο με 3D επεξεργασία (προβολή των σημείων κάθε cluster πάνω στο επίπεδο και έλεγχος αν βρίσκονται εντός των ορίων του), ενώ η 2^η απλά ελέγχει αν **τα πλαίσια προβολής των clusters στην εικόνα** επικαλύπτονται σε ποσοστό άνω ενός ορίου με την μάσκα του δρόμου. Η μέθοδος της 2^{ης} συνάρτησης δουλεύει σαφώς γρηγορότερα και έχει καλύτερα αποτελέσματα!

Παραδείγματα από την εκτέλεση:

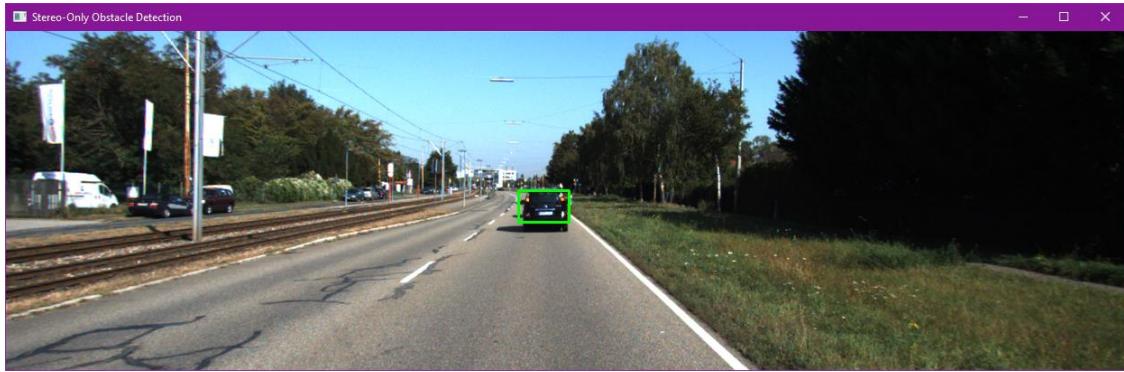


Χρόνος εκτέλεσης: 4.27 sec

Αρχείο: `pure_cv_approach\Aii_object_detection\stereo_obstacle_detection_deprecated.py`



Χρόνος εκτέλεσης: 6.00 sec



Χρόνος εκτέλεσης: 3.24 sec



Χρόνος εκτέλεσης: 3.77 sec



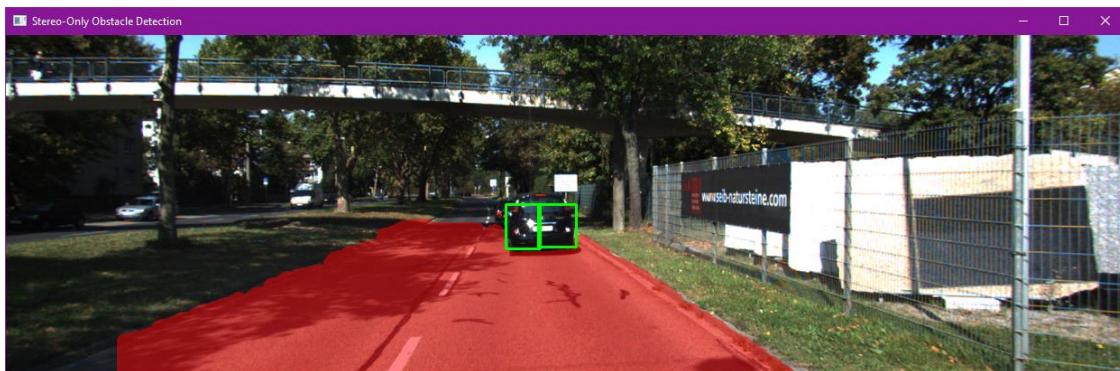
Χρόνος εκτέλεσης: 3.48 sec

Παρατηρείται ότι οι χρόνοι επεξεργασίας είναι απαγορευτικοί για επεξεργασία σε πραγματικό χρόνο, επομένως το συγκεκριμένο ερώτημα αντιμετωπίστηκε και με εναλλακτική προσέγγιση.

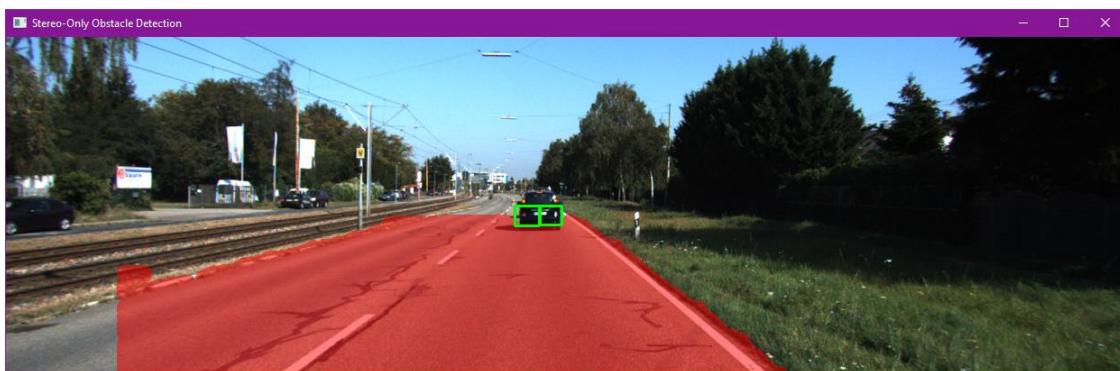
Επίλυση με Yolo και βελτίωση DBSCAN με παράλληλη επεξεργασία:

Αρχείο: *pure_cv_approach\Aii_object_detection\Aii_obj_detection_current.py*

Στηριζόμενος στην [αρχική υλοποίηση](#), χρησιμοποίησα την βιβλιοθήκη joblib για να γίνει παράλληλη επεξεργασία clusters ώστε να μειωθεί ο χρόνος εκτέλεσης! **Παραδείγματα αποτελεσμάτων:**

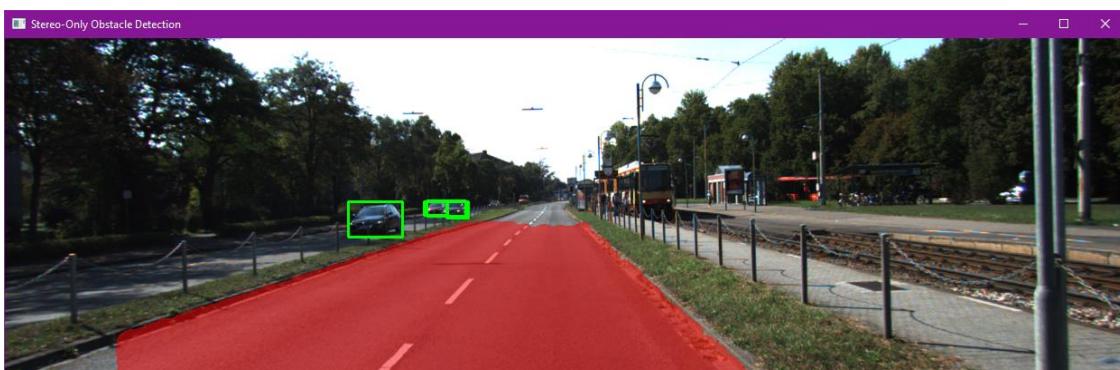


Χρόνος εκτέλεσης: 0.89 sec



Χρόνος εκτέλεσης: 0.42 sec

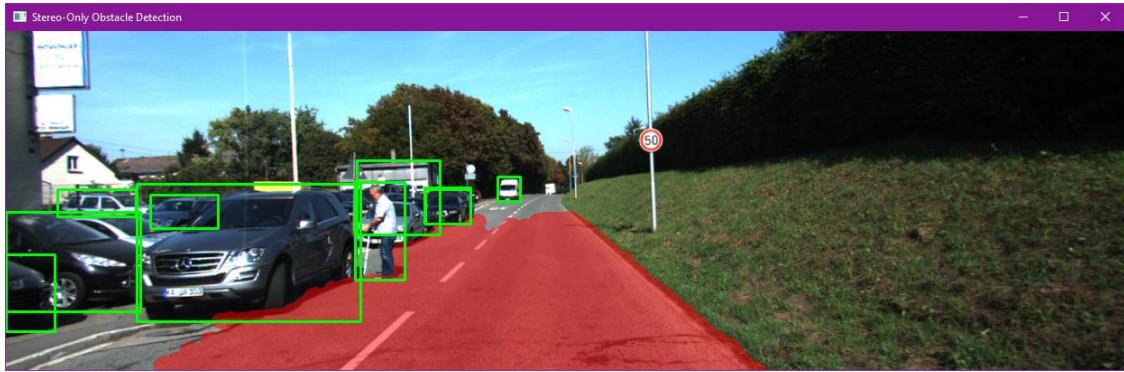
Τέλος, δοκίμασα το [προ-εκπαιδευμένο νευρωνικό δίκτυο YOLO](#), που **αποτέλεσε κιόλας την τελική υλοποίησή μου για το ερώτημα Aii** (λόγω της ταχύτητας επεξεργασίας που προσφέρει):



Χρόνος εκτέλεσης: 0.26 sec



Χρόνος εκτέλεσης: 0.28 sec



Χρόνος εκτέλεσης: 0.32 sec



Χρόνος εκτέλεσης: 0.37 sec



Χρόνος εκτέλεσης: 0.30 sec

Υπάρχουν 2 επιλογές φόρτωσης του YOLO, ένας με `torch.hub.load` και ένας με `yolov5.load`. Ο 2^{ος} τρόπος υπάρχει επειδή θέλω το πρόγραμμα μου να μπορεί να εκτελεστεί και σε Python 3.7 έκδοση!

iii) Υπολογίστε διάνυσμα κίνησης του αυτοκινήτου προς τη σωστή κατεύθυνση. Αν υπάρχουν εμπόδια απεικονίστε έναν κύκλο.

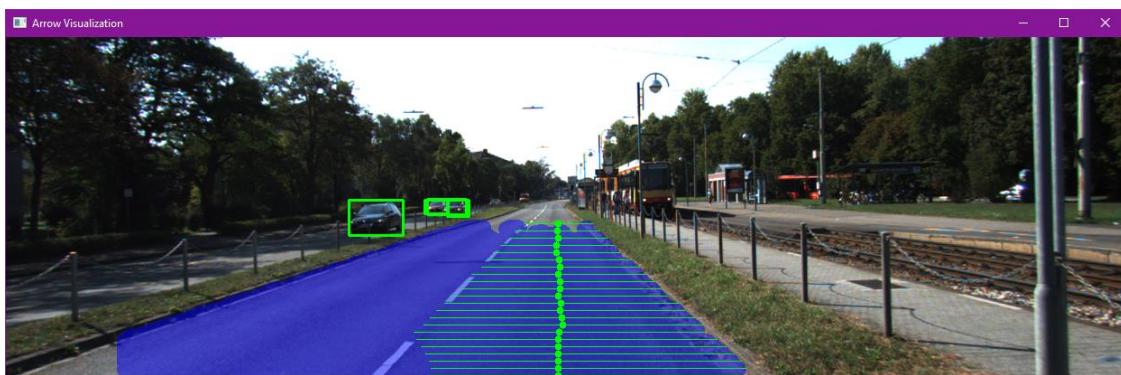
Αρχείο: pure_cv_approach\Aiii_arrowMove.py

Επεκτείνοντας την υπάρχουσα υλοποίηση με τη συνάρτηση **draw_arrow_right_half**, επιτυγχάνεται η επίλυση του ερωτήματος Aiii. Πιο συγκεκριμένα, η συνάρτηση αυτή:

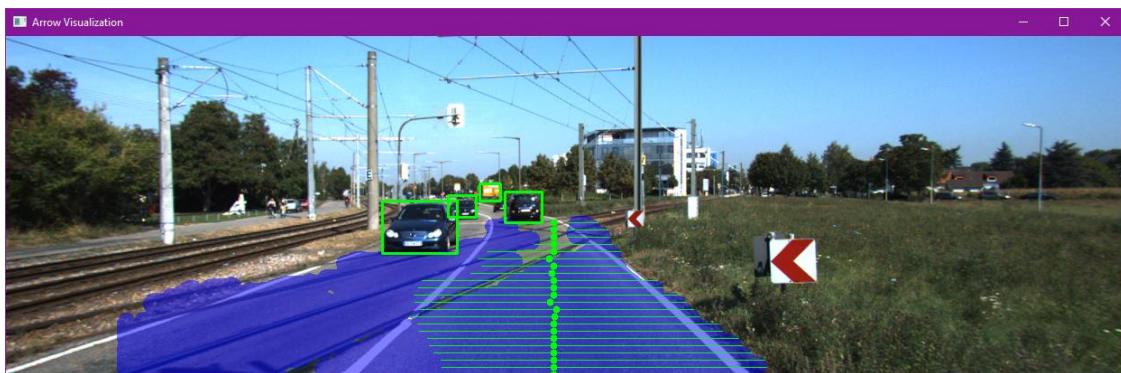
- ✓ εντοπίζει επαναληπτικά τα κεντρικά σημεία του δεξιού τμήματος του δρόμου (σε επίπεδα βάθους),
- ✓ σχεδιάζει διαδοχικά σημεία (κουκίδες) προς «τα μπροστά» στην εικόνα,
- ✓ και χρωματίζει την πορεία πράσινη ή κόκκινη, ανάλογα με το αν εντοπίζεται σύγκρουση με κάποιο **εμπόδιο (πράσινο πλαίσιο)**.

Προαιρετικά εφαρμόζεται φίλτρο (Remove Jumps filter) για αποφυγή θορύβου στη μάσκα, ενώ μπορεί να σχεδιαστεί και το πλήρες πλάτος του δρόμου.

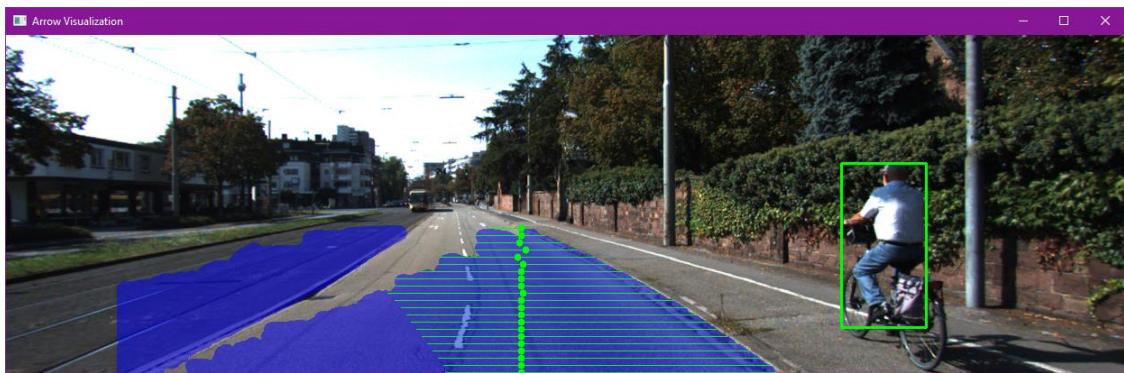
Αποτελέσματα:



Διάρκεια εκτέλεσης: 0.32 sec



Διάρκεια εκτέλεσης: 0.24 sec



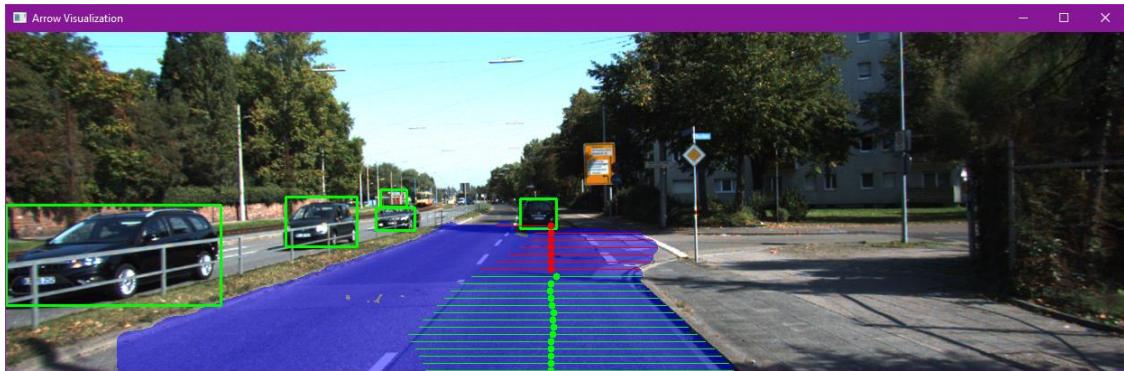
Διάρκεια εκτέλεσης: 0.30 sec



Διάρκεια εκτέλεσης: 0.25 sec



Διάρκεια εκτέλεσης: 0.29 sec

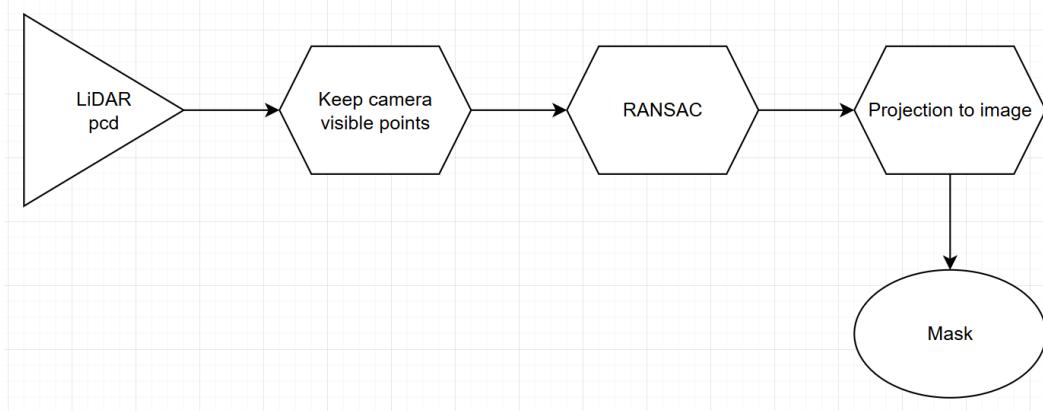


Διάρκεια εκτέλεσης: 0.31 sec

Μέρος Β:

i) Χρησιμοποιήστε το 3D point cloud για να ανιχνεύσετε το δρόμο και τα όριά του.
Κατηγοριοποιήστε και χρωματίστε διαφορετικά τα αντίστοιχα σημεία.

Το pipeline της επίλυσής μου:



Παρατηρούμε ότι αρκετά στάδια του παραπάνω pipeline - συγκεκριμένα τα RANSAC και Projection to image - αντιστοιχούν σε τεχνικές που είδαμε και στην υποενότητα [Επίλυση με stereo κάμερα και disparity!](#) Η μόνη **νέα τεχνική** που εφαρμόζεται, ειδικά για την καλύτερη αξιοποίηση των δεδομένων LiDAR, είναι η **Keep camera visible points**, η οποία υλοποιείται μέσω του helper `project_lidar_to_image` και της συνάρτησης `filter_visible_points`. Με λίγα λόγια:

- Η 1^η συνάρτηση μεταφράζει κάθε 3D σημείο του LiDAR σε pixel της κάμερας, απορρίπτοντας όσα είναι πίσω από τον φακό.
- Η 2^η χτίζει πάνω σε αυτό και κρατά αποκλειστικά τα σημεία που πράγματι «πέφτουν» μέσα στην εικόνα, ώστε το pipeline να δουλέψει τελικά μόνο με ό,τι όντως βλέπει η κάμερα.

Αυτό είναι εφικτό, καθώς το KITTI dataset παρέχει τους πίνακες `Tr_velo_to_cam` και `P2` στα calibration αρχεία του.

Έτσι, με αυτή την τεχνική, ο RANSAC πετυχαίνει πιο γρήγορη και ακριβή ανίχνευση του δρόμου. Η ιδέα αυτής της τεχνικής ήρθε από το optimization **View Frustum Culling** που εφαρμόζει η CD Projekt Red στο παιχνίδι Witcher 3.

Αποτελέσματα:



Διάρκεια εκτέλεσης: 0.06 sec / um_000044



Διάρκεια εκτέλεσης: 0.10 sec / uu_000043



Διάρκεια εκτέλεσης: 0.06 sec / uu_000052



Διάρκεια εκτέλεσης: 0.08 sec / uu_000056



Διάρκεια εκτέλεσης: 0.09 sec / uu_000072

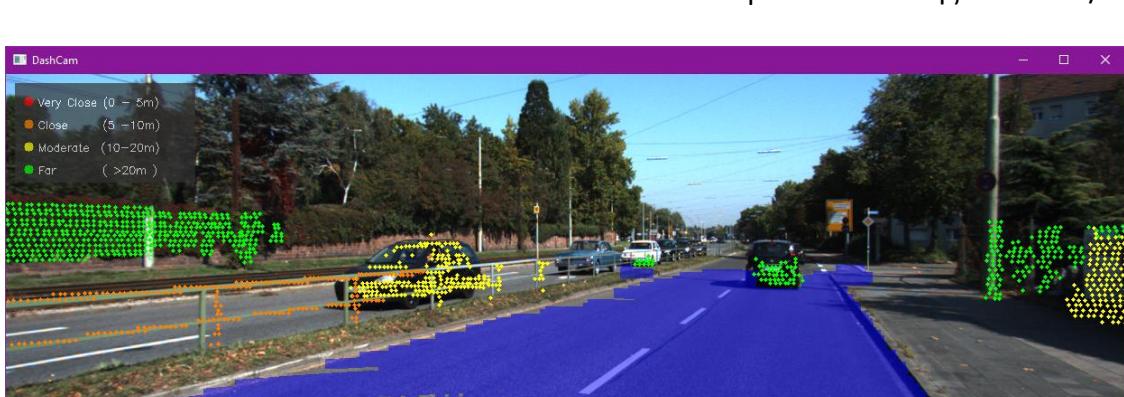
Αρχείο: pure_cv_approach\part_B\Bi_road_detection_pcd.py

ii) Ανιχνεύστε τα αντικείμενα/εμπόδια που βρίσκονται στο δρόμο, πχ αυτοκίνητα, πεζοί, άλλα εμπόδια. Δε χρειάζεται να κάνετε κατηγοριοποίηση.

Για την επίλυση του ερωτήματος εφαρμόστηκε λογική παρόμοια με εκείνη της υποενότητας [Αρχική υλοποίηση](#). Αρχικά, χωρίζουμε τα σημεία του LiDAR σε σημεία δρόμου και μη με την βοήθεια της συνάρτησης `my_road_from_pcd_is` που υλοποιήθηκε στο Bi ερώτημα. Ακολουθεί φίλτραρισμα κατά ύψος (`filter_by_height` συνάρτηση) και DBSCAN. Τέλος, η μάσκα του δρόμου προβάλλεται πάνω στην εικόνα μέσω της `overlay_mask`, ενώ τα clusters αποδίδονται με κουκίδες (διαφορετικού χρώματος ανάλογα με την απόσταση που έχουνε από την κάμερα) χρησιμοποιώντας τη συνάρτηση `visualize_results!`

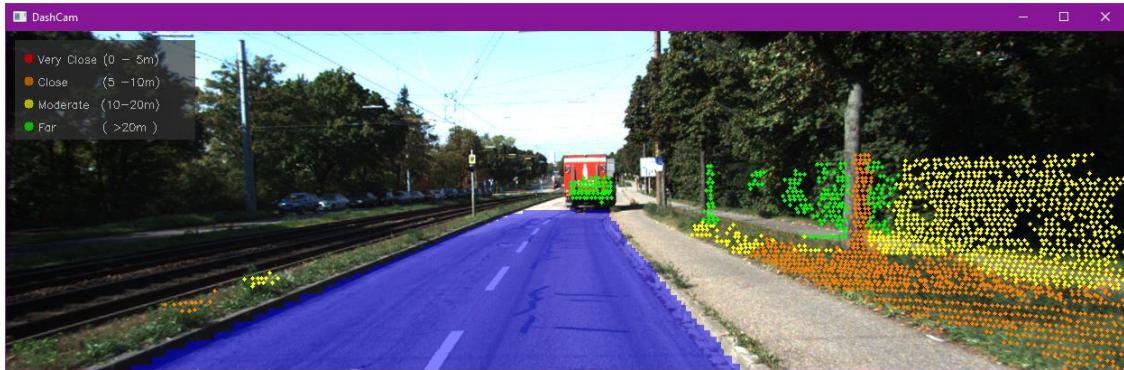
Αρχείο: `pure_cv_approach\part_B\Bii_LiDAR_obstacle_detect.py`

Αποτελέσματα:





Διάρκεια εκτέλεσης: 0.19 sec / um_000019



Διάρκεια εκτέλεσης: 0.18 sec / um_000060



Διάρκεια εκτέλεσης: 0.20 sec / um_000072



Διάρκεια εκτέλεσης: 0.19 sec / um_000010

iii) Υπολογίστε διάνυσμα κίνησης του αυτοκινήτου προς τη σωστή κατεύθυνση. Αν υπάρχουν εμπόδια απεικονίστε έναν κύκλο.

Η επίλυση στο ερώτημα αυτό συνδυάζει τις υλοποιήσεις των ερωτημάτων Bi και Bii, ενσωματώνοντας επιπλέον πληροφορία για το διάνυσμα κίνησης. Συγκεκριμένα, με την βοήθεια της συνάρτησης **prepare_processed_pcd**, έχουμε:

- ✓ Εύρεση του δρόμου.
- ✓ Dotted αναπαράσταση των εμποδίων.
- ✓ Βέλος προσανατολισμού (δημιουργείται από την συνάρτηση **cast_arrow_along_camera**).

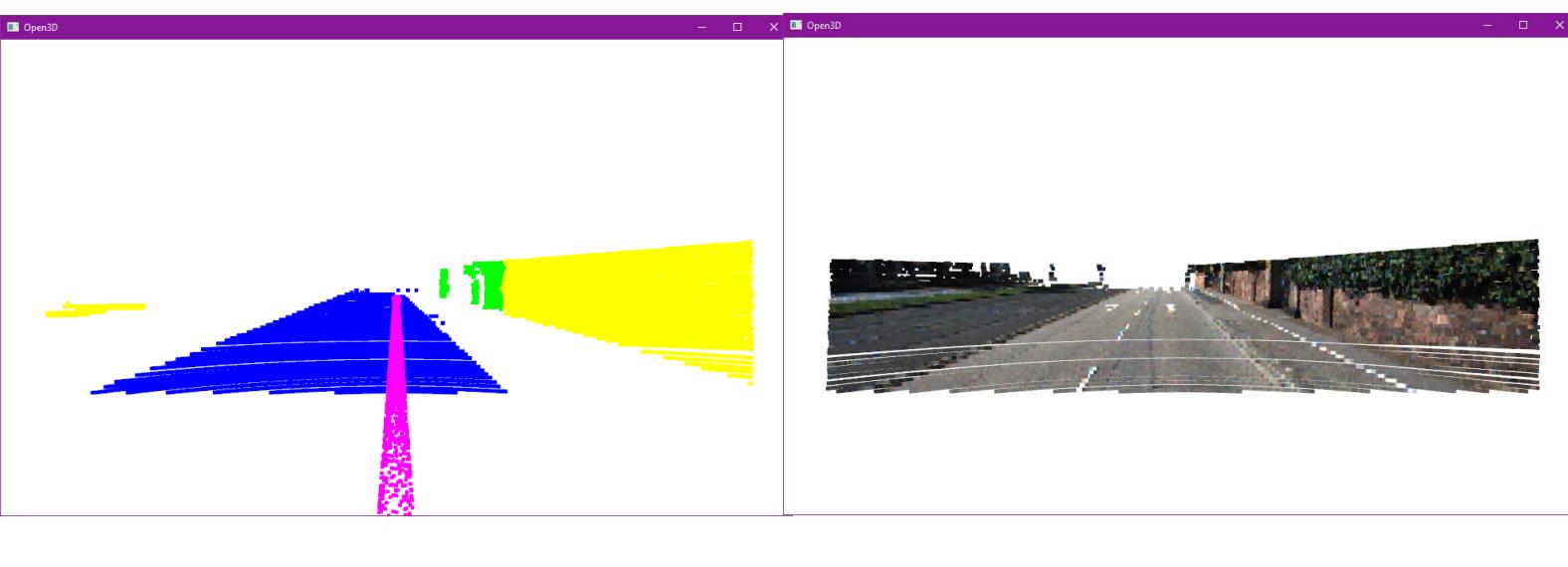
Αναλυτικότερα, η συνάρτηση **cast_arrow_along_camera** προβάλει 1 3D βέλος από το LiDAR origin προς την κατεύθυνση που κοιτά η κάμερα. Αν το βέλος συναντήσει εμπόδιο, τότε σταματά εκεί η επέκτασή του και αλλάζει το χρώμα του σε κόκκινο, αλλιώς το βέλος φτάνει μέχρι το max length που έχουμε περάσει σαν παράμετρο στην συνάρτηση. Φυσικά, έχει εφαρμοστεί κατάλληλη μετατόπιση του βέλους στον Z άξονα για καλύτερη εμφάνιση!

Παράλληλα, δημιουργήθηκε μία συνάρτηση (**project_colors_on_pcd**) που προβάλει τα χρώματα της εικόνας στο pcd του LiDAR (καθώς ήθελα το ερώτημα αυτό να έχει διαφορετική εμφανισιακά λύση από το αντίστοιχο Aiii της κάμερας). Πατώντας Space γίνεται εναλλαγή της προβολής [**raw/processed**].

Τέλος, υπάρχει η δυνατότητα προβολής είτε από Bird's-Eye View (BEV), είτε από οπτική γωνία που αντιστοιχεί στην κάμερα του KITTI.

Αποτελέσματα

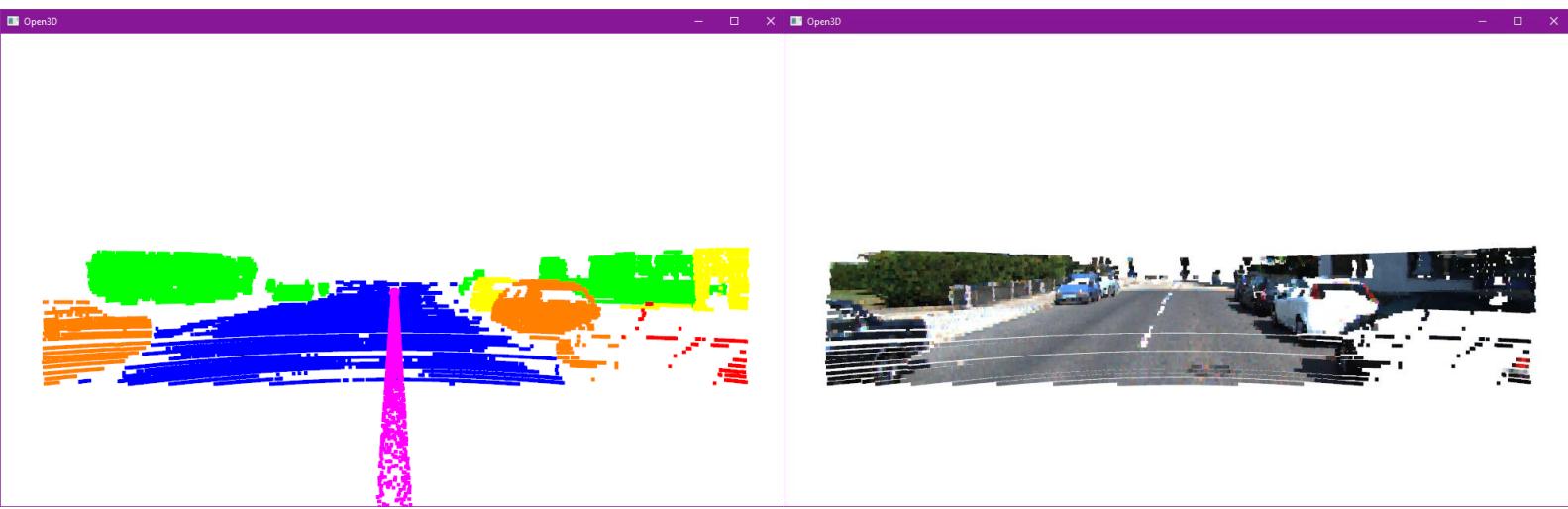
Χρόνος εκτέλεσης: 0.30 sec



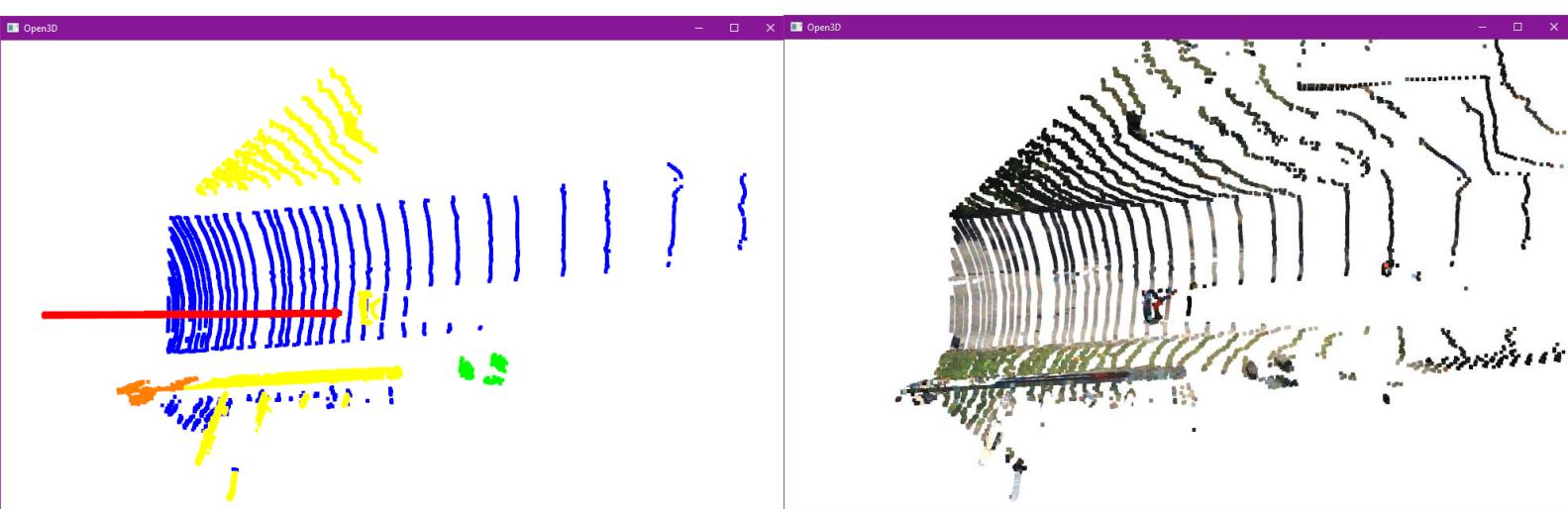
processed

raw

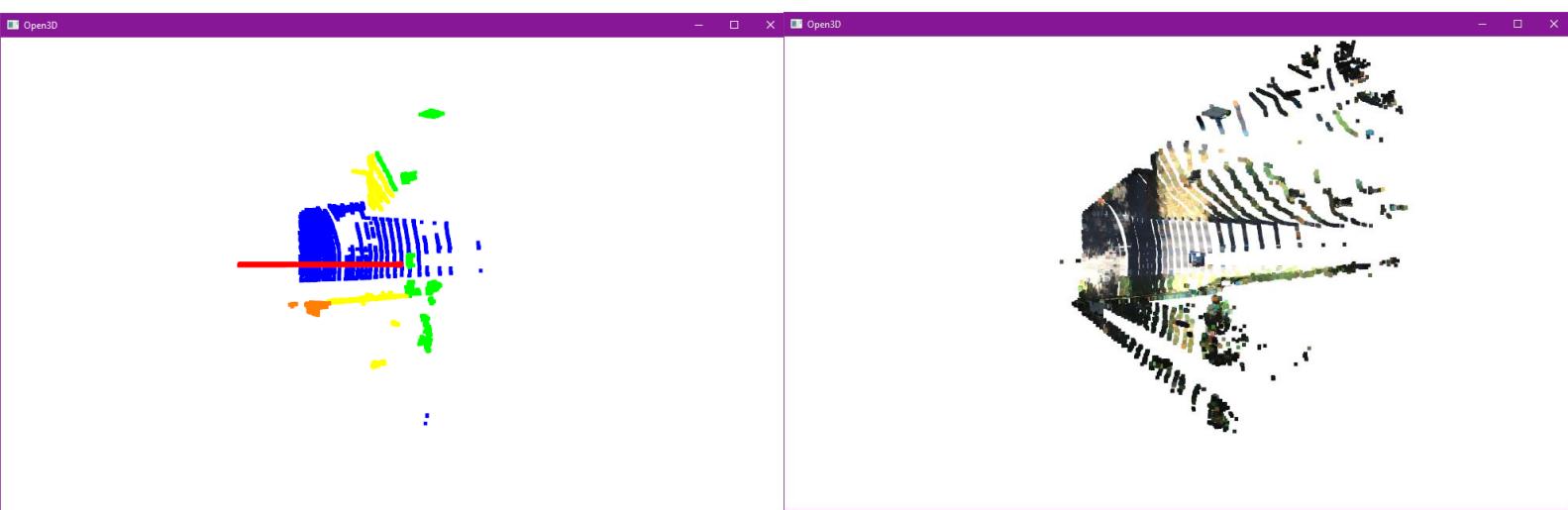
Αρχείο: *pure_cv_approach\Biii_LiDAR_arrowMove.py*



Χρόνος εκτέλεσης: 0.33 sec



Χρόνος εκτέλεσης: 0.36 sec



Χρόνος εκτέλεσης: 0.32 sec

Ενδεικτική εκτέλεση του κώδικα σε βίντεο:
<https://youtu.be/DXkgFZ9ZrB4>

Ερωτήματα Αίν και Βίν - Τοίχος

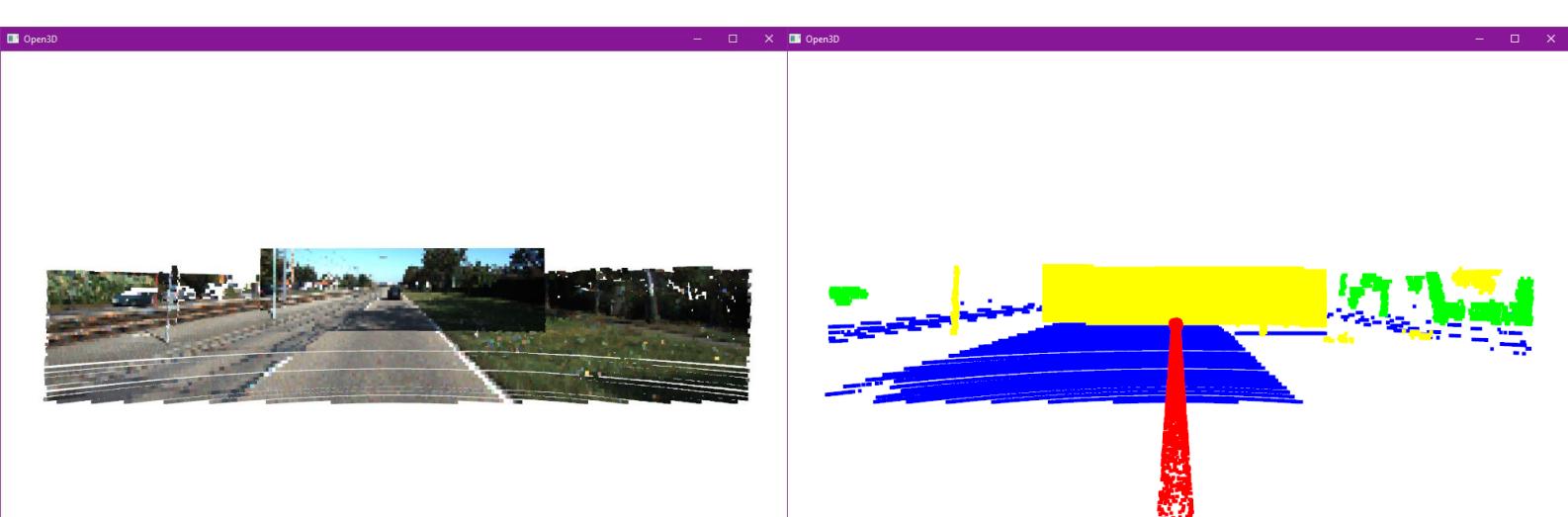
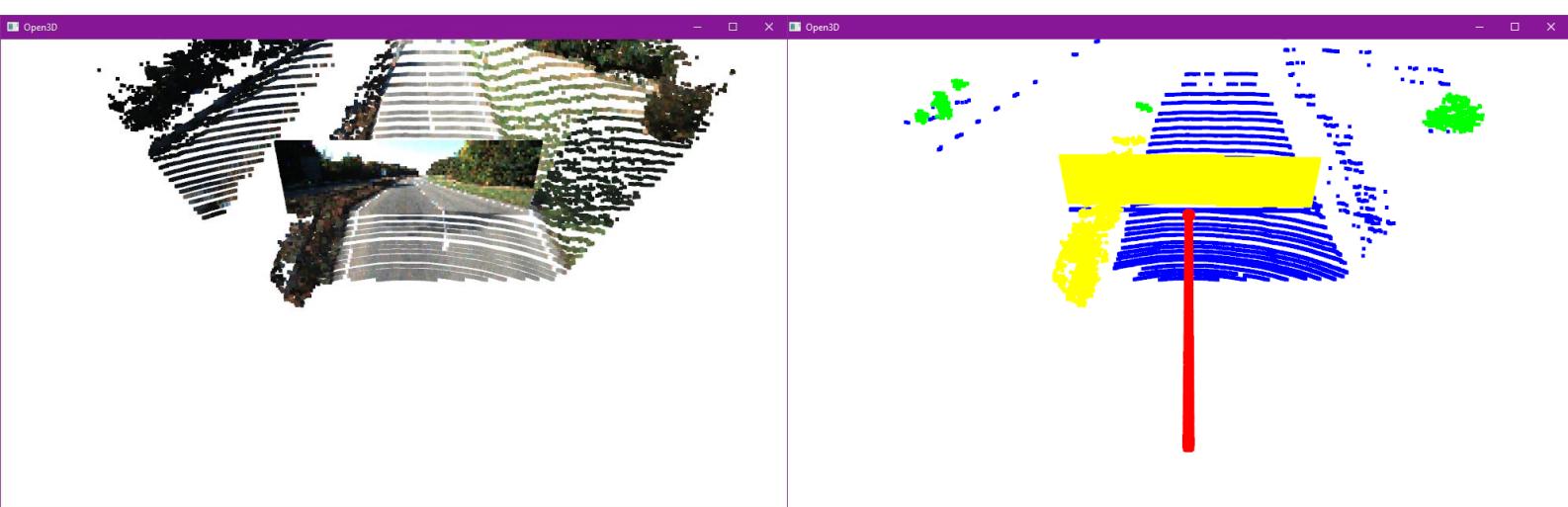
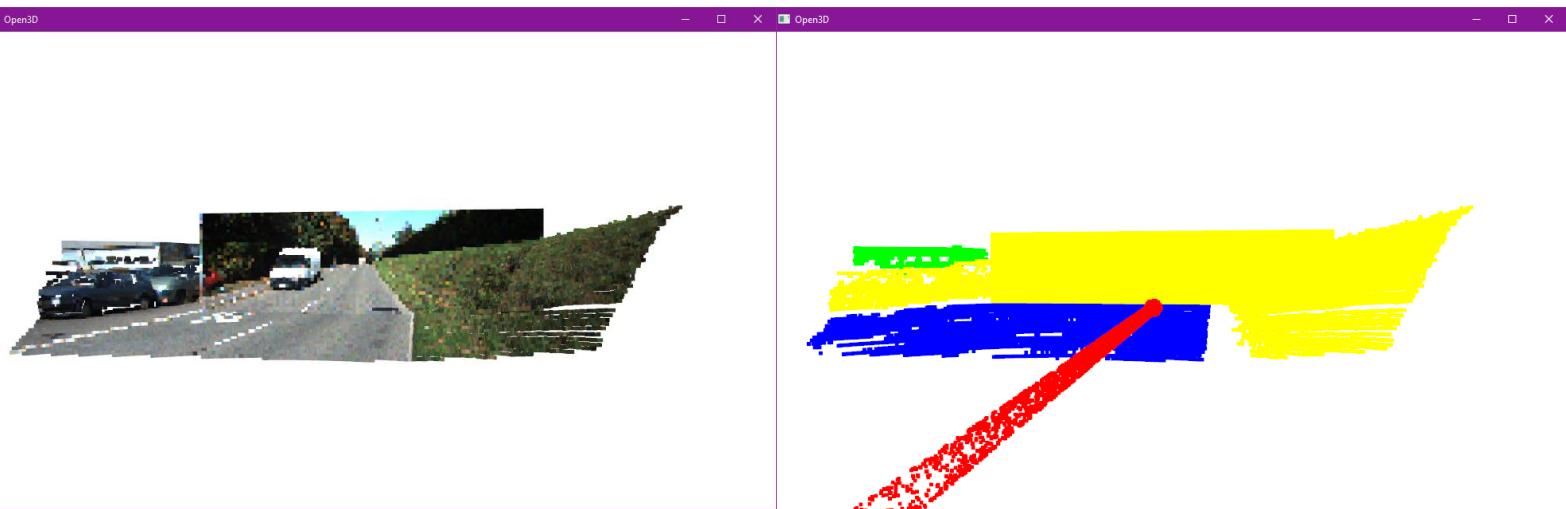
Αίν) Κατασκευάστε ένα επίπεδο κάθετα στην επιφάνεια του δρόμου, πάνω στο οποίο θα προβάλετε (απεικόνιση υφής) την εικόνα που βλέπει η κάμερα λίγα μέτρα πριν τον τοίχο.

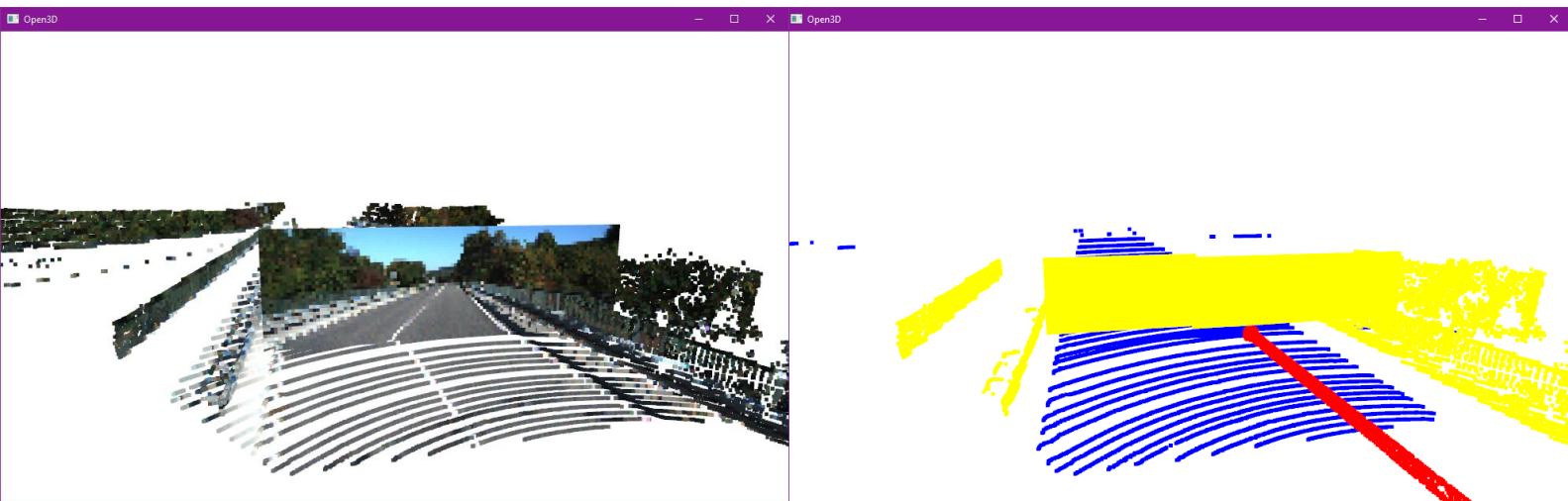


-> Camera Wall Test

Διάρκεια εκτέλεσης: 0.35 sec
Διάρκεια εκτέλεσης: 0.33 sec
Διάρκεια εκτέλεσης: 0.25 sec
Διάρκεια εκτέλεσης: 0.26 sec

Βιν) Χρησιμοποιήστε το περιβάλλον του ερωτήματος A.iv) και εκτελέστε τα ερωτήματα ii), iii), vi), vii) για την περίπτωση αυτή.





-> LiDAR Wall Test

Χρόνος εκτέλεσης: 0.74 sec

Χρόνος εκτέλεσης: 0.51 sec

Χρόνος εκτέλεσης: 0.52 sec

Χρόνος εκτέλεσης: 0.50 sec

Σχολιασμός αποτελεσμάτων

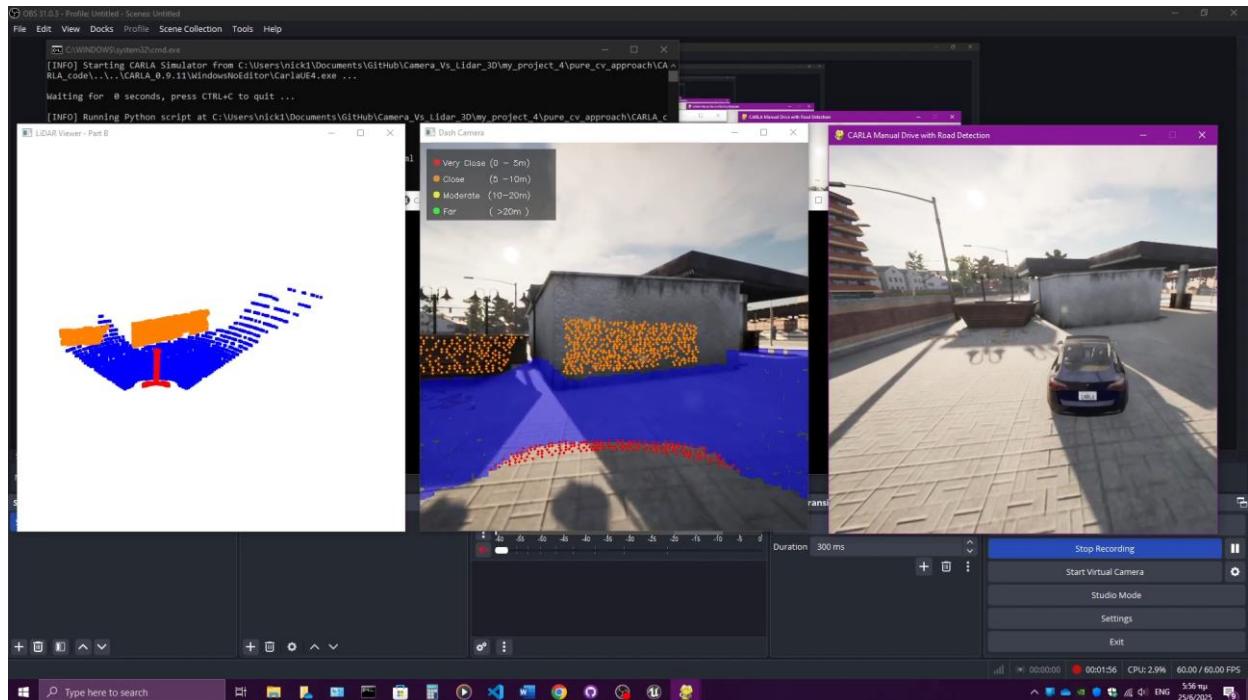
Παρατηρούμε ότι το **setup** της κάμερας μπερδεύτηκε/απέτυχε στις 2 από τις 4 περιπτώσεις και θα έπεφτε πάνω στον τοίχο (οι 2 που απλά δεν βρίσκει δρόμο μέσα στον τοίχο και δείχνει κανονικό βέλος κίνησης, πιθανός οφείλονται σε κακό edit των φωτογραφιών). Αντίθετα, το **setup** του LiDAR εντόπισε και στις 4 περιπτώσεις την παρουσία εμποδίου μπροστά, χρωματίζοντας έτσι το βέλος του κόκκινο ως ένδειξη κινδύνου/αντικειμένου!

Real Time Simulation

BONUS: Τρέξετε τις υλοποιήσεις σας ακολουθιακά ώστε να φαίνονται οι λύσεις για διαδοχικά καρέ και να παρουσιάζονται συνεχόμενα (βίντεο).

Για το ερώτημα BONUS χρησιμοποιήθηκε το **CARLA Simulation**, όπου υλοποιήθηκαν όλα τα ερωτήματα και εκεί (εκτός αυτών με τον τοίχο λόγω compilation προβλημάτων με την Unreal Engine έκδοση της CARLA). Ακολουθεί βίντεο για τα ερωτήματα Bι, Bii και Biii με χειρισμό από παίχτη:

<https://youtu.be/pzfL9UhPmQs>



Χειρισμός



Καθώς το τέσταρα, ανακάλυψα
ότι έχει χαλάσει η σκανδάλη L2
στο χειριστήριό μου...

<https://youtu.be/frrVAOI6k38> - Εκτέλεση CARLA για τα ερωτήματα κάμερας

Οδηγίες εγκατάστασης

Χρηζανται οι βιβλιοθήκες:

numpy
opencv-python
matplotlib
open3d
scikit-image
scikit-learn
torch
torchvision
joblib
(δηλαδή, το αρχείο: **general_requirements.txt**)

+ CARLA_0.9.11\WindowsNoEditor\PythonAPI\examples\requirements.txt,

για το CARLA (έκδοση 9.11) απαιτείτε Python 3.7!

<https://github.com/carla-simulator/carla/releases>

Βιβλιογραφία

- [1]. <https://towardsdatascience.com/tutorial-build-a-lane-detector-679fd8953132/>
- [2]. <https://github.com/amusi/awesome-lane-detection/tree/master>
- [3]. <https://sh-tsang.medium.com/review-deeplabv3-atrous-separable-convolution-semantic-segmentation-a625f6e83b90>
- [4]. <https://medium.com/@r1j1nghimire/semantic-segmentation-using-deeplabv3-from-scratch-b1ff57a27be>
- [5]. <https://github.com/Unity-Technologies/com.unity.perception>
- [6]. https://universe.roboflow.com/road-segmentation/road_segmentationv1.1
- [7]. <https://carla.org/>
- [8]. <https://www.youtube.com/@FullSimDriving/videos>
- [9]. <https://sagnibak.github.io/blog/how-to-use-carla/>
- [10]. https://medium.com/@VK_Venkatkumar/segmentation-traditional-deep-learning-approaches-edd50a3308b3
- [11]. <https://medium.com/@mansibagohil1512/region-growing-an-inclusive-overview-from-concept-to-code-in-image-segmentation-f12c5ba50709>
- [12]. https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html
- [13]. <https://medium.com/data-science/a-hands-on-application-of-homography-ipm-18d9e47c152f>
- [14]. <https://www.geeksforgeeks.org/what-are-some-popular-algorithms-used-for-image-segmentation-and-how-do-they-differ-in-their-approach/>
- [15]. <https://hari-sikchi.github.io/papers/LaneDetection.pdf>
- [16]. <https://github.com/VainF/DeepLabV3Plus-Pytorch>
- [17]. <https://www.cityscapes-dataset.com/dataset-overview/#features>
- [18]. <https://arxiv.org/pdf/2501.07245v1>
- [19]. <https://arxiv.org/pdf/2007.10743>
- [20]. <https://github.com/jamohile/stereoscopic-point-clouds?tab=readme-ov-file>
- [21]. <https://medium.com/analytics-vidhya/depth-sensing-and-3d-reconstruction-512ed121aa60>
- [22]. https://github.com/jedeschaud/kitti_carla_simulator
- [23]. <https://wambitz.github.io/tech-blog/carla/python/c++/simulation/autonomous-vehicles/2024/09/29/carla-win11.html>