Vision-Based Detection, Tracking, and Classification in Occlusion-Prone Autonomous Driving Environments

Final Project

34759 - Perception for Autonomous Systems
Technical University of Denmark
December 05^{th} 2024

Joel Farré Cortes (s240076), Iñaki Zabaleta (s240056), Sergio Monzon (s232515), Kushal Kotagal (s242015) and Jónas Ingi Valdimarsson (s230262)

1 Introduction

The development of autonomous driving systems has gained significant momentum in recent years, with the goal of creating intelligent vehicles capable of navigating complex and dynamic urban environments. A key challenge in this domain lies in enabling accurate perception and understanding of the surroundings, especially in scenarios involving occlusions. Reliable detection, classification, and tracking of objects, such as pedestrians, cyclists, and vehicles, are critical for ensuring safety and robust decision-making.

This project focuses on addressing these challenges through a vision-based approach, leveraging a stereo camera system mounted on a vehicle. The primary objectives include calibrating and rectifying the stereo camera setup, detecting and tracking objects in 3D, and training a machine learning model capable of classifying objects into their respective categories. The work is performed using real-world datasets that include both clear and occluded scenarios, reflecting the complexities of urban environments.

Through this project, we aim to integrate computer vision techniques and machine learning algorithms to create a robust system for object detection and tracking. Additionally, we explore the calibration process and evaluate its impact on the accuracy of rectified stereo images, comparing our results with the provided reference data. The outcomes of this project are expected to contribute to the broader goal of developing reliable and efficient autonomous driving systems.

2 Object Detection and Tracking

2.1 Object Detection and Tracking Under Occlusion

In this section, we discuss the process and methods used to detect objects from the given sequences, determine their 3D position from stereo rectified images, track their movement, and predict their motion behind occlusions. Handling occlusions effectively is crucial in autonomous driving scenarios to ensure accurate object tracking and improve vehicle safety in dynamic environments.

2.2 Approach and Methodology

To achieve the goal of object detection and tracking, three key methods were implemented: object detection, depth calculation, and 3D motion tracking.

Object Detection: Object detection was performed using the YOLO (You Only Look Once) algorithm [5]. YOLO was selected for its simplicity and accuracy, allowing us to detect multiple classes of objects efficiently.

Depth Calculation: Depth calculation was performed by computing the disparity map between the left and right stereo images, followed by applying a formula to determine the depth of detected objects. The required camera parameters—focal length and baseline—were obtained through the calibrated camera matrix (discussed in Section 3) and the provided camera setup, respectively.

3D Motion Tracking: To track the movement of detected objects, the Kalman filter was used, allowing continuous tracking even during occlusions. The Kalman filter was implemented using the `filterpy.kalman` library, which simplified the setup and allowed for an efficient prediction and update cycle. For each detected object, a new Kalman filter instance was created, which estimated the position, size, and velocity of the object. When an object was occluded, the Kalman filter used its prediction capabilities to maintain the object's estimated position until it reappeared. This approach ensured a continuous and smooth tracking experience, even when objects were partially or fully occluded.

2.3 Object Detection Framework

The chosen detection framework for this project was YOLO, specifically version 8. YOLO was selected due to its high accuracy and ease of use compared to other approaches. Initially, a contour-based detection method was considered, but it proved to be inaccurate and introduced unnecessary complexity. YOLO, as a pre-trained model, provided a much more straightforward solution for the requirements of this project. The YOLO model detects objects in an image and provides a bounding box, confidence score, and object class for each detected object. For this implementation, only objects with confidence scores above a set threshold were considered for further processing. Additionally, YOLO classified the detected objects, and this classification was useful for assigning an object ID and storing information for further tracking.

2.4 Depth Calculation

The depth of detected objects was calculated using stereo images. The OpenCV StereoSGBM algorithm was first initialised with the appropriate parameters [2]. These parameters were tuned iteratively to achieve the best results for the stereo images. The process involved calculating the disparity map for each frame by first converting the stereo images to grayscale and then using the `stereo.compute` function to generate the disparity map. This disparity map provides pixel-wise differences between the left and right images, which is crucial for estimating depth. For each detected object, the bounding box coordinates provided by YOLO were used to calculate the 2D coordinates of the center of the object. Using these coordinates, the corresponding disparity value was retrieved from the computed disparity map. If the disparity value was valid, the depth (Z-coordinate) was calculated using the formula:

$$\text{Depth} = \frac{f \times B}{d} \tag{1}$$

where f is the focal length of the camera, B is the baseline distance between the two cameras, and d is the disparity value. This formula allowed us to accurately determine the depth of each detected object in the scene. The calculated depth values were then integrated with the object detection information, enabling 3D tracking of the objects. This depth information was essential for determining the position of objects in the 3D environment and for further tracking.

2.5 Tracking Methodology

To track the motion of detected objects, the Kalman filter was employed [1]. The `filterpy.kalman` Python library simplified the initialisation and use of the Kalman filter model. The model was initialised using the following matrices:

- **F (State Transition Matrix):** This matrix models the relationship between the object's current state and its next state. It includes parameters for position, velocity, and time step (dt) to predict future positions based on current velocity.
- **H (Measurement Matrix):** This matrix maps the predicted state to the observed measurements, which in our case includes position and size of the object. It is used to update the predicted state based on new observations.
- **Q (Process Noise Covariance Matrix):** This matrix represents the uncertainty in the object's movement. The Q matrix was chosen to have relatively higher values for velocity components compared to position components, reflecting the uncertainty in object motion dynamics.
- **R (Measurement Noise Covariance Matrix):** This matrix represents the noise in the sensor measurements. The R matrix was set with higher values for width and height measurements compared to position measurements. This reflects the confidence in detecting object positions, while accounting for a greater uncertainty in the size estimates due to potential occlusions and partial detections.

2.6 Tracking Algorithm

In each new frame, a cost matrix was constructed based on the Mahalanobis distance between predicted positions and new detections. This allowed for the association of detected objects with existing trackers. Assignment of detections to existing trackers was implemented using Python's "linear sum assignment" function from the "scipy.optimize" library.

When an object went behind an occlusion, the Kalman filter's predictions were used to estimate and display the object's bounding box until it reappeared in the frame, ensuring continuity in tracking despite occlusions. The age of each tracker was also monitored, and trackers were deleted if they exceeded a maximum number of consecutive frames without detection. This helped in handling situations where objects left the field of view or were no longer relevant.

The Mahalanobis distance was chosen instead of the Euclidean distance due to its ability to account for the correlations between different state variables, such as position and velocity. Unlike the Euclidean distance, which treats each variable independently, the Mahalanobis distance incorporates the covariance matrix, allowing for a more accurate assessment of how well a detection matches a predicted state [4]. The linear sum assignment was used for assigning detections to trackers because it provides an optimal solution to the assignment problem, ensuring that the overall cost (based on the Mahalanobis distance) is minimised. This algorithm is particularly well-suited for tracking applications where multiple detections and trackers must be matched in a way that minimises discrepancies [3].

2.7 Results and Discussion

The performance of the object detection component of the project was very good. The accuracy and reliability of the YOLO model in identifying different objects were crucial for ensuring that subsequent tracking processes started with strong initial detections.

The depth values calculated from the stereo images were not formally validated against ground truth data. However, the depth estimations generally made intuitive sense based on visual inspection and naked-eye perception of the relative depth of objects in the scene. This suggests that the disparity calculations and depth estimation approach were effective, though further validation would be needed for precise analysis.

Motion prediction of objects when they went behind occlusions was not entirely accurate, but the predictions were generally sufficient to maintain tracking. The Kalman filter's predictions provided a decent estimate of the object's motion, which allowed for re-locking onto the object when it emerged from occlusion. This continuity in tracking despite occlusions was a positive outcome, demonstrating that the Kalman filter's prediction mechanism was functional, though with some limitations.

For cars, the predicted motion was slower than the actual observed speed, while for pedestrians, the predicted motion was often faster than the actual movement. A potential reason for this discrepancy could be that the Kalman filter's constant velocity assumption struggled to capture the dynamic nature of these objects' movements accurately. These results indicate the need for a more adaptive motion model that can better account for acceleration. Overall, object tracking and motion prediction were successfully achieved, but errors were observed, particularly in predicting motion behind occlusions. While the Kalman filter allowed for continuity in tracking during occlusions, improvements could be made to handle varying velocities more effectively, especially for different types of objects with distinct motion behaviors.

2.8 Challenges and Limitations

Ghosting: Sometimes, the object would be undetected and then redetected between frames where it was still visible, resulting in a new ID being assigned. This issue, known as ghosting, could be caused by temporary occlusions, changes in object appearance, or variations in confidence scores that dropped below the detection threshold momentarily. In the scheme of the overall goal, this issue did not significantly impact performance, as the re-identification of objects happened infrequently, and the continuity of tracking was generally maintained without major disruptions.

Constant Velocity Assumption: The Kalman filter's assumption of constant velocity was a significant limitation, particularly in scenarios where object motion deviated significantly from linear paths. This discrepancy highlights the inability of the Kalman filter to account for dynamic motion, leading to errors in tracking and inaccurate bounding box placements, especially during occlusions.

Depth Validation Challenge: When the depth calculations were attempted to be validated against Sequence 2, the translation from camera to world coordinates provided negative depth values. This issue made it challenging to quantitatively validate the accuracy of the depth estimation. Instead, the depth values were validated intuitively based on visual inspection. This introduces an accuracy limitation to the model reducing the confidence in the reliability of the depth values.

3 Image Classification

3.1 Task goal and libraries

The main idea of this section was to be able to create a model of a Machine Learning system which would be able to classify entities in different images into 3 categories: pedestrians, cyclists and cars.

For this the YOLO (You Only Look Once) object detection algorithm was ideal to detect and evaluate objects in a sequence of images. The main goal is to asses the accuracy of the predictions made by the YOLO model by comparing them against the ground truth data of each sequence (*seq_ 01* and *seq_ 02*) provided in the *labels.txt* file.

To achieve this, we are calculating intersection areas, matching predicted and ground truth bounding boxes, an visualizing the results on image frames. YOLO is chosen for this task due to its efficiency and real-time performance capabilities, making it a suitable candidate for detecting objects in video frames or sequential image data.

3.2 What is the code doing?

The task is started with the *load_ ground_ truth* function. It processes the ground truth information provided in a *labels.txt* file. This file contains data such as frame indices, bounding box coordinates, and object types. The function organizes this data into a dictionary, where each frame index maps to a list of objects present in that frame.

The *compute_ area* and *compute_ intersection_ area* functions calculate the area of a bounding box and the intersection area between two boxes. The intersection area is crucial for computing the Intersection over Union (IoU). This measure will be used later to calculate the accuracy between our model and the ground truth.

Since we want to be sure that the IoU calculation is done between the two right matching boxes, we created the function *match_ boxes*. This function pairs predicted bounding boxes with ground truth boxes based on the IoU. Predictions are matched to ground truth objects of the same category with the highest IoU, provided it exceeds a threshold (e.g., 0.5). Unmatched predictions or ground truth boxes are stored for further analysis.

The YOLO model is initialized using a pre-trained weight file. It detects objects in each frame, generating predictions that include bounding boxes, confidence scores, and object categories. Predictions are filtered based on confidence levels and mapped to specific categories such as People, Bike, and Car.

Since YOLO is only able to detect bicycles or pedestrians but not cyclists, there had to be a solution to mix both categories and turn them into cyclists. This is achieved by checking if a bicycle overlaps sufficiently with a pedestrian box. If a match is found, the bounding box of the pedestrian and the bicycle get mixed to include the bicycle, and the object is reclassified as a cyclist. This also is useful so any other bikes that are not currently being ridden and therefore not cyclists, can be ignored.

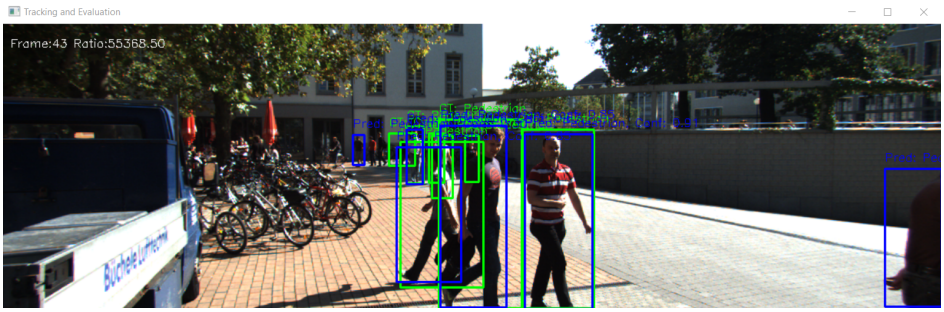
Finally for the user, the ground truth and predicted bounding boxes are drawn on each frame using distinct colors, blue for the predicted boxes and green for the ground truth. Labels and confidence scores are also displayed. The total area of ground truth boxes and the intersection area for matched boxes are computed to calculate the average overlap ratio, which will be used as a kind of loss function to see how far our predictions are from the ground truth.

The script iterates through each frame, visualizing both ground truth and predicted detections, along with metrics like overlap ratios. The processed images are displayed in real-time using OpenCV.

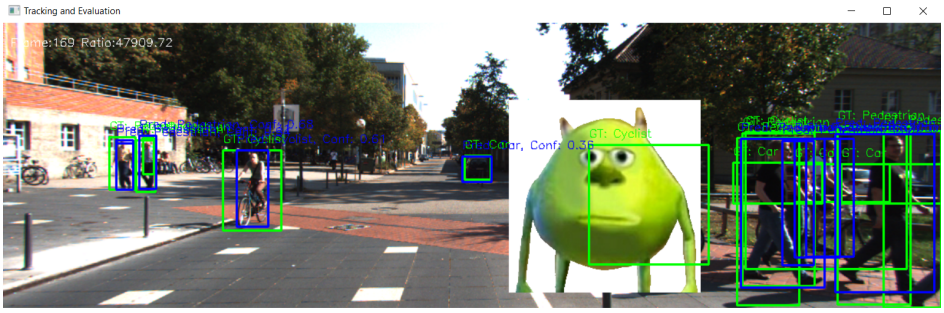
3.3 Achievements

As a final result, our YOLO model is able to do a good job detecting all possible categories.

On the one hand, the model is a bit sensitive to objects in the shadow. There are times that if the confidence is set too high, the model is not able to detect them. On the other hand, if the confidence on the objects is set too low, the model may detect objects that are not considered in the ground truth, which also lowers the accuracy of the model. Defining the perfect confidence is not an easy task since it depends a bit on the sequence and if there are many shadows or light changes in it.



Despite these isolated cases, the model does its task almost perfectly. The boxes it predicts fit very well to the size of the real ones. It can also detect objects at very long distances, which is a plus. I also think we have done a great job in detecting cyclists since the model is not able to do it by itself. Thanks to the union of the pedestrian and bicycle boxes that overlap, we have been able to recreate the boxes of the cyclists and they encompass the subject more accurately than the ground truth provided as seen below. The other benefit of ignoring other bikes can be seen on top which is that we can remove any bikes that are not getting ridden from the detected objects.



4 Camera Calibration and Stereo Rectification

4.1 Introduction to the problem

The aim of this section is to implement stereo rectification in both of the cameras in the vehicle (left camera and right camera). We have achieved this by performing 3 steps:

- Camera calibration: is defined as the process to obtain intrinsic, extrinsic and distortion parameters of a camera to understand how it maps the 3D world onto a 2D image plane.
- Stereo calibration: is the process of determining the relative positions and orientations of two cameras in a stereo setup, as well as their intrinsic and distortion parameters.
- Stereo rectification: is the process of transforming two stereo images (captured from slightly different viewpoints) so that corresponding points in the two images lie on the same horizontal line.

For this section, we have used the library opencv, very useful for computer vision and image processing algorithms.

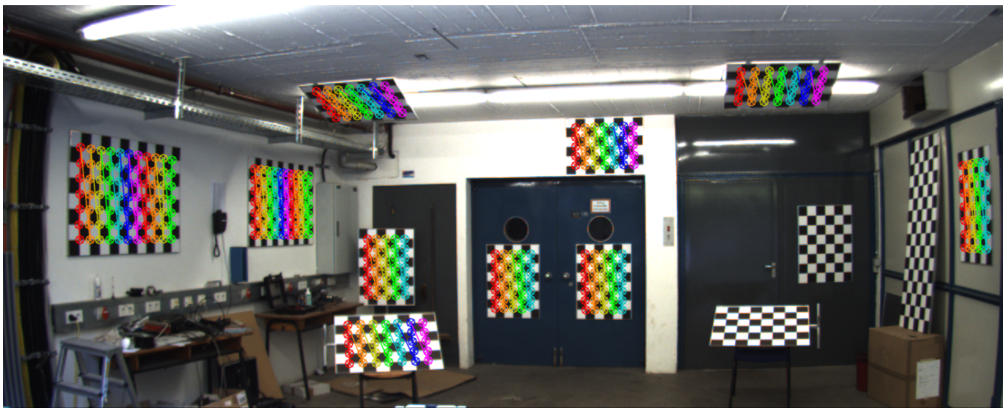
4.2 Steps of the process

Let's dive into each step and see how we have performed them.

1. Obtain the images: The images we have used to perform the camera calibration are the ones provided by the assignment, in the folder 'calib'. In each image, there were 13 chessboards, each of them in a different position, angle and direction.

As we can see, some chessboards are very visible, but some other needed a transformation to be more clear, and so the algorithms can detect the corners of the chessboard more accurate. For each chessboard, we have obtain the number of inner corners it has, so we can use that numbers in posterior function.

2. Camera calibration: Our next step is to perform camera calibration for both cameras. First, we have made some image transformation, so that each chessboard is clearly visible. For each chessboard in each image, we have obtained the window where it is, crop it and apply a factor scale of 2. The final result is an image containing only the chessboard, with a size twice the original. This is done so that the detection focuses on only that specific board, and computes an optimal solution. Then, each chessboard (cropped and expanded) was the input of the opencv function cv2.findChessboardCorners. For this function, our input parameters are each chessboard, the number of inner corners it has and some flags (opencv functions to make the detection more accurate). The idea of that function is to detect the position of all the inner corners. Then, we used the function cv2.cornerSubPix to each of the detected set of inner corners to make the detection more accurate.



As we can see, some of the chessboards were unable to be detected, even with a rough transformation of them, but most of them (always more than 10 out of 13) were detected. After all of the images were passed through this step, and the maximum number of corners were detected, we computed the camera calibration with the function cv2.calibrateCamera. This gives us the parameters ret, cmtx, dist, rvecs, tvecs (being cmtx and dist the camera matrix and the distorsion coefficients, respectively).

This camera calibration was made for both cameras and both set of images.

3. With those 4 parameters (2 camera matrix and 2 distorsion coefficient), our next step is to perform the stereo calibration.

This will be perform by the function cv2.stereoCalibrate. This function gives us the extrinsic parameters on which both of the cameras are. In other words, among the output of the function, we have the rotation and translation to go from one camera to another. This function, as well as the others, only takes grey images, so we needed to convert the RGB image to a grey image.

4. After this function, we have assumed that the rotation and translation of the left camera are the identity matrix and the 0 matrix (so no rotation and translation are required for this camera), and the right camera takes the rotation and translation give by the previous step. So, by applying R and T, the right camera should be alligned with the left camera. Then, the last step is performed. This step is the stereo rectification. That is, apply this transformation to all images, so that it can be used in all of the other sections of this project. This will be done using the sequence images given in the assignment. First we have applied cv2.stereoRectify to take the rectification transform matrix and the projection matrix for each camera. Then, the function cv2.initUndistortRectifyMap is applied to obtain the transformation maps for each camera. This transformation maps are then applied using cv2.remap, so that the final rectified images are obtained. Finally, the final images are cropped with the parameteres roi given in the function cv2.stereoRectify.

   In the following image, we can see 3 images. The first one is the original raw image. The second one is the image after our algorithm has rectified it. Finally, the last image is the rectified provided by the assignment. For example, by looking at the truck (left part), we can assure that the stereo rectification has been applied with a high accuracy.

4.3 Results

Each of the cameras have a rmse error when performing the camera calibration of around 0.21 and 0.20. For general applications, a rmse between 0.2 and 0.5 is consider acceptable. For the stereo calibration, an rmse of around 0.19 is obtained.

Additionally, we have implemented 2 function to compute how similar are the rectified image compute by us and the one given. These algorithms are feature-matching-with-ransac and fourier-similarity. We observe that the first function gives us a similarity of 0.85 and 0.83 for the left and right images, and the second function gives us a similarity of 0.98 and 0.97.

5 Video

Below, we are including the link to a video showing the implementation of our project https://youtu.be/sJpeM7-CwOo

References

[1] FilterPy. *Kalman Filter Documentation*. Accessed: 3 December 2024. URL: https://filterpy.readthedocs.io/en/latest/kalman/KalmanFilter.html.

[2] OpenCV. *StereoSGBM Class Documentation*. Accessed: 3 December 2024. URL: https://docs.opencv.org/3.4/d2/d85/classcv_1_1StereoSGBM.html.

[3] SciPy. *scipy.optimize.linear$_s$um$_a$ssignment*. Accessed: 3 December 2024. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html.

[4] Statistics How To. *Mahalanobis Distance: Definition and Example*. Accessed: 3 December 2024. URL: https://www.statisticshowto.com/mahalanobis-distance/.

[5] Ultralytics. *YOLOv8 Documentation*. Accessed: 3 December 2024. URL: https://docs.ultralytics.com/models/yolov8/.