TU WIEN Informatics

CycleSafely Mobile

Replacing the object detector in the CycleSafely pipeline with a real-time YOLO-based BEV model and evaluating its performance

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Bernhard Bayer
Matrikelnummer 01228892

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass. Mag.rer.soc.oec. Stefan Ohrhallinger, PhD

Wien, 21. März 2026

Bernhard Bayer	Michael Wimmer

TU WIEN Informatics

CycleSafely Mobile

Replacing the object detector in the CycleSafely pipeline with a real-time YOLO-based BEV model and evaluating its performance

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Bernhard Bayer
Registration Number 01228892

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Mag.rer.soc.oec. Stefan Ohrhallinger, PhD

Vienna, March 21, 2026

Bernhard Bayer	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Bernhard Bayer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 21. März 2026

Bernhard Bayer

Kurzfassung

CycleSafely Mobile ist eine Anwendung für Smartphones, welche die Sicherheit im Straßenverkehr für Fahrradfahrer erhöhen soll. Objekte wie Autos, Fahrradfahrer und Fußgänger werden in LiDAR-Punktwolken erkannt. Anschließend werden die Objekte verfolgt und es wird ihre zukünftige Position bestimmt. Dies erlaubt es, frühzeitig Kollisionen vorherzusagen und so den Fahrradfahrer zu warnen. In dieser Arbeit wird der Objektdetektor in der CycleSafely-Anwendung durch eine effizientere Alternative ausgetauscht. Während der bereits vorhandene Detektor SFA3D eine gute Erkennungsgenauigkeit erzielt, ist er für Echtzeitanwendungen auf Mobilgeräten zu rechenintensiv. Deshalb wird ein YOLOv11n-OBB-basierter Ansatz verwendet, welcher mit Bird's-Eye-View-Darstellungen (BEV) von LiDAR-Daten arbeitet, die in einem separaten Vorverarbeitungsschritt erstellt wurden. Die Laufzeitleistung der CycleSafely Mobile-Pipeline wird auf einem Smartphone anhand von LiDAR-Daten aus einer einzigen Sequenz der KITTI-Rohdatenaufzeichnungen [8] evaluiert. Die vorgeschlagene Methode führt zu einer deutlichen Steigerung der Laufzeitleistung und erreicht auf einem Mittelklasse-Smartphone bei einer BEV-Auflösung von 320 Pixeln bis zu 42 ms pro Frame ($\approx 23,8$ FPS). Gleichzeitig wird auf dem KITTI-Validierungsdatensatz eine mittlere durchschnittliche Präzision (mAP@0,5) von 0,74 erzielt. Bei höheren BEV-Auflösungen, wie z. B. 640 Pixeln, erreicht die Methode bis zu 0,90 mAP@0,5 bei etwa 168 ms pro Frame (≈ 6 FPS). Dies verdeutlicht den Kompromiss zwischen Genauigkeit und Laufzeitleistung. Diese Ergebnisse zeigen, dass 2D-BEV-basierte Objekterkennungsansätze eine praktikable Lösung für mobile Echtzeitanwendungen darstellen.

Abstract

CycleSafely Mobile is a smartphone application designed to improve road safety for cyclists. Objects such as cars, cyclists, and pedestrians are detected in LiDAR point clouds. The objects are then tracked, and their future positions are determined. This allows for early collision prediction, thus warning cyclists. In this work, the object detector in the CycleSafely application is replaced with a more efficient alternative. While the existing SFA3D detector achieves good detection accuracy, it is too computationally intensive for real-time applications on mobile devices. Therefore, a YOLOv11n-OBB-based approach is employed, which operates on bird's-eye view (BEV) representations of LiDAR data generated in a separate preprocessing step. The runtime performance of the CycleSafely Mobile pipeline is evaluated on a smartphone using LiDAR data from a single sequence of the KITTI raw data recordings [8]. The proposed method leads to a significant increase in runtime performance, achieving up to 42 ms per frame ($\approx$ 23.8 FPS) on a mid-range smartphone at a BEV resolution of 320, while reaching a detection performance of 0.74 mean Average Precision (mAP@0.5) on the KITTI validation split. At higher BEV resolutions, such as 640, the method achieves up to 0.90 mAP@0.5 at around 168 ms per frame ($\approx$ 6 FPS), illustrating the trade-off between accuracy and runtime performance. These results demonstrate that 2D BEV-based object detection approaches represent a viable solution for real-time mobile applications.

Contents

CHAPTER 1

Introduction

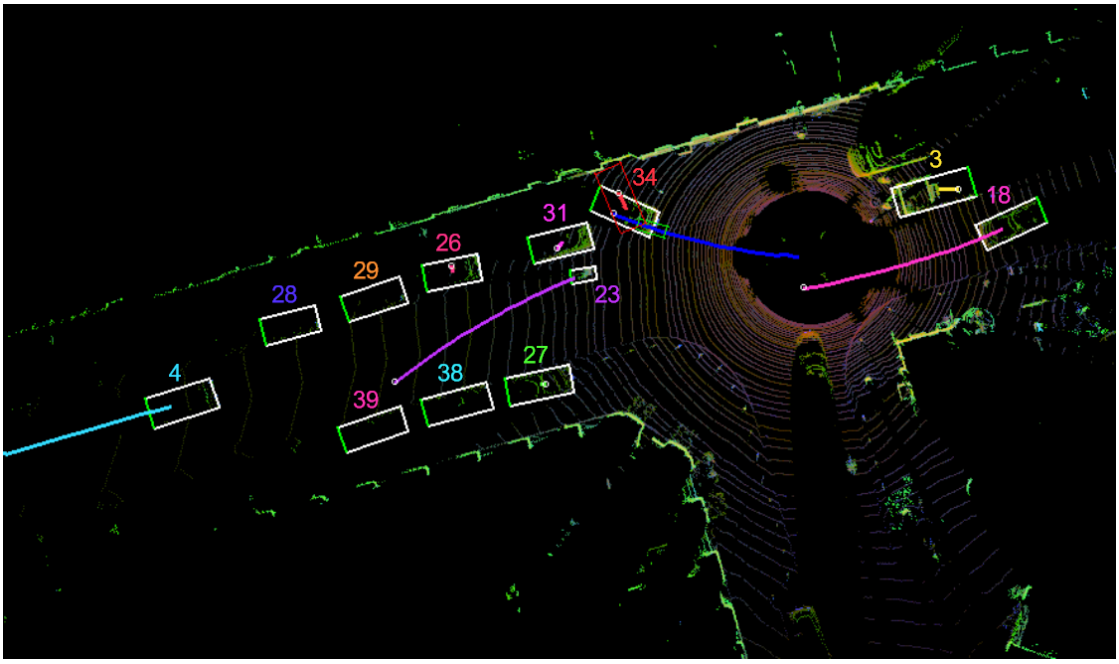


Figure 1.1: The image shows the visualization of the CycleSafely Mobile application. A point cloud is converted into a bird's-eye view map. Detected cars are visualized using bounding boxes. The lines show the predicted paths of the cars. The red number 34 shows a predicted collision.

Object detection and machine learning are important and widely studied research areas. Object detection and tracking are widely used in applications such as smartphones, security cameras, cashierless stores, and autonomous driving.

While some tasks do not require fast object detection, many other tasks depend on fast and accurate object detection. A prominent example is autonomous driving, where critical calculations and decisions based on detected objects on the road, such as cars, pedestrians, and cyclists, are executed.

Object detection can be done using an RGB camera image as input (Monocular RGB image vehicle detection), but a popular way to detect cars and other vehicles on the road is the use of a LiDAR scanner. The LiDAR scanner uses a laser to generate a point cloud consisting of up to millions of points of the surrounding area. Machine learning is then used to detect various objects in the point cloud, usually vehicles, bicycles, and pedestrians.

While autonomous vehicles can rely on powerful onboard hardware to perform object detection, this is not feasible for cyclists. Cyclists primarily rely on their smartphones as their main computing platform. However, smartphones are limited in both processing power and sensing capabilities. Although many modern smartphones are equipped with time-of-flight (ToF) sensors for depth perception, their effective range is typically limited to approximately 5 to 10 meters. This range is insufficient for reliable object detection in real-world traffic scenarios. Therefore, in practical applications, the system would need to be complemented by an external LiDAR sensor to ensure sufficient spatial coverage.

This thesis focuses on finding and implementing a fast object detection model, which preferably is able to do real-time object detection on an edge device, like a mobile phone. The goal is to achieve at least 10 frames per second (FPS). FPS describes how many times the object detection model is able to perform inference in one second.

The model is then integrated into the existing CycleSafely Mobile pipeline [10]. The CycleSafely Mobile application tries to predict collisions with the user by detecting cars in a point cloud. After tracking the detected objects, their trajectories are predicted. If a collision is detected, the time until the collision is displayed. The main challenge is to achieve efficient object detection on an edge device, such as a smartphone, at an acceptable speed, so real-time predictions can be made.

The runtime performance of the CycleSafely Mobile pipeline is evaluated on a smartphone. Since the model itself cannot process raw LiDAR data, the LiDAR data is converted into a bird's-eye view (BEV) representation in a pre-processing step. The LiDAR data used for evaluation are not read directly from a LiDAR sensor, but originate from a single sequence of the KITTI raw data recordings [8].

CHAPTER 2

Related Work

In this chapter, we will examine various approaches to object detection in LiDAR data and identify which datasets are frequently used in autonomous driving research. We will focus particularly on approaches suitable for real-time applications and therefore for mobile deployment.

2.1 Object Detection

In this section, we will first discuss various LiDAR-based 3D detection methods, and then the 2D object detector YOLO.

2.1.1 LiDAR-based 3D detection

This section reviews common methods for detecting objects in LiDAR data. These can generally be divided into three main categories: voxel-based methods, point-based methods, and bird's-eye view (BEV) based methods.

Voxel-based methods

In voxel-based methods, points from the point cloud are converted into a 3D grid consisting of cubes called voxels. Multiple points are grouped within a single voxel. This allows for efficient application of 3D convolutions. If the voxel size is too large, this leads to a loss of important spatial information. On the other hand, small voxels can lead to increased memory consumption and computational cost. An example of a voxel-based model is SECOND [27].

Point-based methods

Point-based models process point cloud data directly, without converting it into another structure. Unlike voxels, where information is lost through quantization, here the original geometric information of the data is preserved. This enables precise 3D localization and shape recognition. However, due to their high computational complexity, these models are not suitable for real-time applications. Examples of point-based methods are PointNet [4] and PointNet++ [15].

BEV methods

In the bird's-eye view (BEV) method, the space of the point cloud is divided into a grid. Points within each grid cell are aggregated into various features, such as point density, maximum height, or point intensity, often encoded as their own channel. This generates a 2D top-down representation of the environment, which allows the application of 2D CNNs. Because 2D convolutions are used, which are very fast, these models are suitable for real-time applications. One disadvantage of this representation is that some vertical information is compressed, which can lead to a reduction in the accuracy of 3D height estimation. Examples of BEV-based models are PointPillars [12], Complex-YOLO [18], YOLO3D [1] and SFA3D [5].

YOLO3D [1] extends the 2D image detector YOLOv2 to predict 3D bounding boxes from BEV representations of LiDAR point clouds. For BEV representation, a height map and a density map are used. A BEV resolution of 608×608 is used, along with a range of 30.4 m to the left and right and 60.8 m forward.

Complex-YOLO [18] is also an extension of YOLOv2. To further improve orientation estimation, an Euler Region Proposal Network is used to predict the yaw angle of an object. A BEV map, encoded in height, intensity, and density, is used, covering an area of $80\,\text{m} \times 40\,\text{m}$ in front of the sensor with a resolution of 1024×512.

SFA3D [5] is a real-time BEV-based object detector. It is designed for use in autonomous driving scenarios. Similar to previous approaches, point cloud data is converted into a BEV representation. The BEV map, encoded in intensity, height, and density, is used to predict 3D bounding boxes of traffic participants. The BEV map has a resolution of 608×608 and covers an area of 25 m to the left and right and 50 m forward. The model achieves real-time performance on desktop GPUs and is used as an object detector in the CycleSafely pipeline. However, the high computational requirements still pose challenges for deployment on mobile devices.

Many LiDAR-based models achieve high detection accuracy. However, many models, even those advertised as real-time, are unsuitable for mobile hardware due to their computational demands. Models like SFA3D still rely on relatively large backbone networks, limiting their performance on mobile devices. Therefore, a lightweight object detector designed for fast inference, such as YOLO-based architectures, presents a promising alternative for mobile applications.

2.1.2 YOLOv11

The YOLO (You Only Look Once) family of models is a widely used real-time 2D object detection framework. It is capable of detecting 2D objects using RGB images. Since BEV maps often consist of multiple channels, such as density, height, and intensity, they can be treated similarly to RGB images and used directly as input for YOLO for both training and detection. Because of its flexibility to be easily trained on custom datasets, it is used in many different areas, such as medical image analysis, retail monitoring, and autonomous driving applications.

YOLO, in the variant YOLOv11 [11], is available in different model sizes and supports multiple tasks, including object detection, oriented bounding box detection (OBB), instance segmentation (SEG), pose estimation, and classification (CLS). The models come in five different sizes: n, s, m, l, and x. These sizes differ in the number of parameters the model has. While size YOLOv11n has 2.6 million parameters, size YOLOv11x has 59.6 million parameters. While the largest model is designed for desktop hardware and high accuracy, the smaller models are specifically designed for less powerful hardware. For resource-constrained hardware such as mobile devices, smaller variants such as YOLOv11n are especially suitable. Although they are smaller, they still maintain good accuracy.

The YOLOv11n-OBB variant combines lightweight architecture with support for oriented bounding boxes, making it particularly suitable for applications such as vehicle detection in bird's-eye view representations on mobile devices.

Table 2.1 summarizes key characteristics of BEV-based object detection approaches discussed in this chapter.

Table 2.1: Comparison of object detection approaches relevant to this work.

Model	Input Representation	Bounding Boxes	Real-time	Dataset
YOLO3D	BEV LiDAR map	3D OBB	GPU	KITTI
Complex-YOLO	BEV LiDAR map	3D OBB	GPU	KITTI
SFA3D	BEV LiDAR map	3D OBB	GPU	KITTI
YOLOv11n-OBB	RGB / BEV image	2D OBB	Mobile	Custom

2.2 Datasets

To train a machine learning model, it is important to choose an appropriate dataset. There are many datasets of varying quality for autonomous driving. In the object detection research community for autonomous driving, a few datasets have become widely established, such as the KITTI dataset [9], the Waymo Open Dataset [24], the NuScenes dataset [3], the ONCE dataset [13], and the Argoverse2 dataset [25]. In this work, both the Waymo Open Dataset and the KITTI dataset were initially considered. However, the KITTI dataset was ultimately selected due to its more manageable size

and its widespread use in related work, enabling direct comparison with approaches such as SFA3D. Additionally, licensing restrictions of the Waymo Open Dataset limit the redistribution of trained models, which further motivated the focus on KITTI.

2.2.1 KITTI Dataset

The KITTI Vision Benchmark Suite, released by the Karlsruhe Institute of Technology, is widely used in autonomous driving research. The suite provides data for a variety of tasks, including stereo, optical flow, scene flow, depth prediction and completion, visual odometry/SLAM, object detection, tracking, road/lane detection, semantic segmentation, and semantic instance segmentation.

This work primarily utilizes the object detection benchmark [9], hereafter referred to as the KITTI dataset, which supports evaluation in 2D, 3D, and bird's-eye view (BEV). It consists of 7,481 training samples and 7,518 test samples. For each training sample, the dataset provides stereo RGB images as well as a corresponding point cloud. In total, it contains 80,256 labeled objects, each annotated with a 2D bounding box in image coordinates and additional 3D information.

In addition, the suite provides recorded sequences, referred to as raw data [8]. These sequences include grayscale and color stereo images, 3D Velodyne point clouds, GPS/IMU measurements, calibration files, and 3D object tracklet annotations. The data is available in both raw (unsynchronized and unrectified) and processed (synchronized and rectified) formats. The sequences are categorized into different environments, including city, residential, road, campus, and person.

In this work, the processed sequence `2011_09_26_drive_0005` from the city category, consisting of 154 frames, serves as test data in the CycleSafely Mobile application.

2.2.2 Waymo Open Dataset

The Waymo Open Dataset [24] is divided into three sub-datasets: the Perception Dataset with sensor data and labels for 2030 segments, the Motion Dataset with object trajectories and corresponding 3D maps for 103,354 segments, and the End-to-End Driving Dataset with images covering 360 degrees for 5000 segments. The dataset is subject to licensing restrictions, which limit the redistribution of trained models and dataset content.

CHAPTER 3

Implementation

In this chapter, we discuss how the KITTI Dataset was prepared for training the YOLO11-OBB model, how the model was trained, and how YOLO was integrated into the CycleSafely Mobile pipeline.

The implementation of the CycleSafely Mobile pipeline is available in the repository `https://gitlab.cg.tuwien.ac.at/stef/cyclesafelymobile`. The mobile application is developed using the Flutter framework, with Dart as the primary programming language. Components related to dataset preparation and model detection evaluation, which are discussed in later sections, are implemented in Python.

3.1 Overview of the CycleSafely Pipeline

The CycleSafely Mobile pipeline [10] consists of four components: the object detector, the object tracker, the trajectory predictor, and the collision detector. The object detector uses a point cloud from a LiDAR sensor as input. In the next step, the object detector identifies objects within the point cloud, such as cars, pedestrians, and cyclists. Detected objects are assigned an oriented bounding box. These bounding boxes also encode the object orientation. The oriented bounding boxes are then passed to the object tracker, which assigns an ID to each bounding box. The trajectory predictor estimates the future path of the detected objects. Based on the predicted trajectories, the collision detector determines whether a collision will occur. Figure 3.1 illustrates the pipeline as a flowchart.

Figure 3.2 shows the CycleSafely application. Figure 3.2a displays the various settings. The user can select the detection model, along with which delegate settings, such as GPU delegate, XNNPACK, or NNAPI, should be used. Figure 3.2b shows the running pipeline. Detected objects are visualized using bounding boxes. The arrows indicate the predicted path. Figure 3.2c shows a predicted collision in 2 seconds.

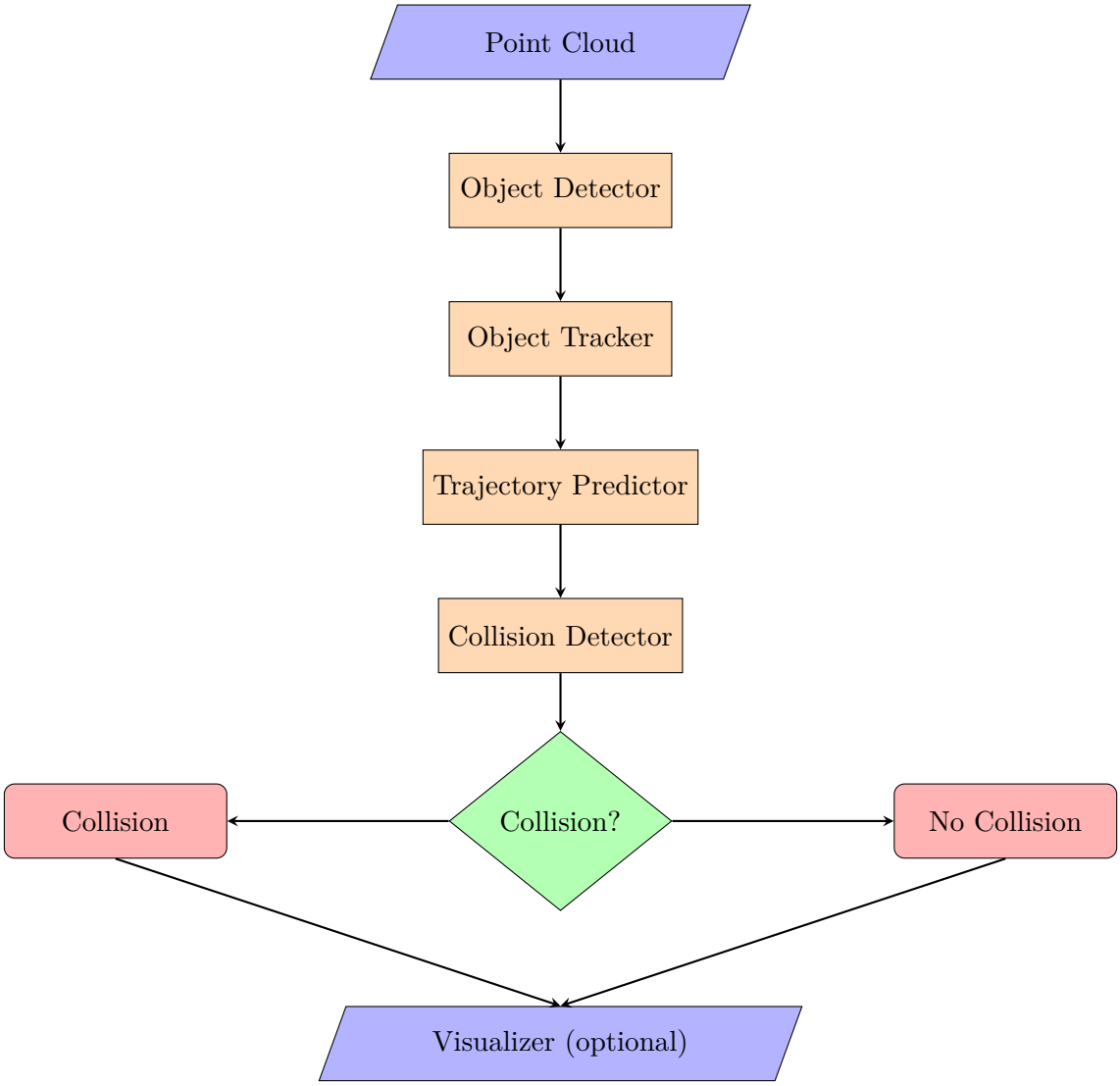
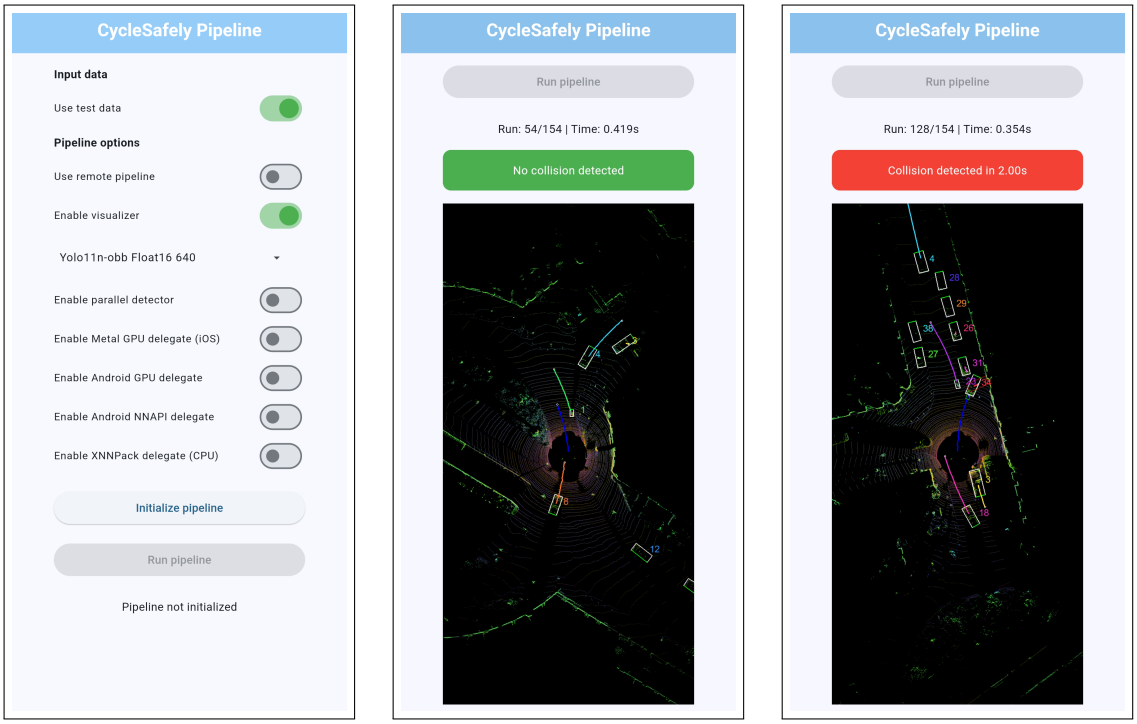


Figure 3.1: The CycleSafely pipeline: A point cloud serves as input for the object detector. Detected oriented bounding boxes are passed to the object tracker, which assigns an ID to each box. The trajectory predictor attempts to determine their future position. Based on this predicted position, the collision detector decides whether a collision will occur.

3.2 Initial YOLO11-OBB model training test

YOLOv11n with oriented bounding boxes (OBB) was chosen as the model because of its small size and fast speed on mobile devices. Initial experiments were conducted using a GitHub project [2] which uses the Waymo Open Dataset to train a YOLOv8-OBB model. In the project, the Waymo LiDAR data is converted into BEV maps, which are then used to train the YOLO model. The project was used to train a YOLOv11n-OBB model



(a) CycleSafely main screen with settings.

(b) Running pipeline. No collision is detected.

(c) Running pipeline: A collision is detected in 2 seconds.

Figure 3.2: The CycleSafely app: Showing the settings and the running pipeline.

and to validate the feasibility of training YOLO models on BEV representations.

3.3 YOLO11-OBB model training

The first step is to determine a YOLO compatible input resolution. Due to the YOLO architecture using a maximum stride of 32, the input resolution must be divisible by 32. Empirical evaluation showed that stable execution with the Android TensorFlow Lite GPU delegate requires the input resolution to be divisible by 64. Resolutions divisible by 32 but not by 64 led to a native GPU delegate deadlock, likely caused by incompatible spatial dimensions in the final stride-32 feature map.

3.3.1 Dataset preparation

The KITTI dataset [9] was used for training and evaluation. It contains a training set with labeled data and a separate unlabeled test set. The training set comprises 7,481 samples, each consisting of a LiDAR point cloud and a corresponding annotation file. These samples were sequentially split into 6,000 training samples and 1,481 validation samples, following the split used by SFA3D. Since the YOLO training dataloader does

not support LiDAR data, the LiDAR data was converted into BEV maps. The BEV map covers an area of 50 m × 50 m, with 25 m to the left, 25 m to the right, and 50 m forward.

As the BEV map size, different pixel resolutions were tested, starting with 640 × 640. Later, other BEV map sizes were also tested, like 320 × 320, 512 × 512, or 1024 × 1024.

Like in Complex-YOLO and SFA3D, the BEV map is encoded using intensity, height, and density channels. To compute these channels, the x and y coordinates of the LiDAR points in the point cloud are discretized into a 2D grid based on the resolution and defined spatial area of the BEV map. Each point is assigned to exactly one cell in the BEV map. For each grid cell, the height, intensity, and density features are computed as follows: The height is calculated using the z-value of the highest point within the cell. This value is normalized to the range $[0, 1]$ using a predefined height range of $z_{\min} = -2.73\,\text{m}$ and $z_{\max} = 1.27\,\text{m}$.

$$h = \frac{z_{\max}^{\text{cell}} - z_{\min}}{z_{\max} - z_{\min}} \tag{3.1}$$

The intensity value is taken from the point with the maximum height within the cell. The density is calculated using the number of LiDAR points (N) within the cell and normalized using logarithmic scaling using the equation:

$$d = \min\left(1.0, \frac{\log(N+1)}{\log(64)}\right) \tag{3.2}$$

This logarithmic scaling, which follows common practice in related work such as Complex-YOLO [18], YOLO3D [1], and SFA3D [5], helps compress large point counts while preserving differences in sparsely populated regions. The resulting BEV map is then saved as a PNG image for training. Due to an OpenCV default setting, which saves RGB images as BGR, the BEV maps were saved as a density-, height-, and intensity map.

Table 3.1 shows the KITTI dataset label format, and Table 3.2 shows the YOLO-OBB label format. The labels in the KITTI dataset contain a bounding box for different objects. However, the bounding box coordinates are only available for 2D camera images and not for BEV maps. Therefore, the labels first had to be transformed into the correct coordinate system. To do this, the bounding boxes contained in the labels were first transformed from camera coordinates into world coordinates, and then into pixel coordinates for the BEV map. Since training was only performed with the object classes *car*, *cyclist*, and *pedestrian*, the remaining objects were filtered out of the labels. In particular, the class *car* includes the KITTI categories *Car* and *Van*, while the class *pedestrian* includes *Pedestrian* and *Person_sitting*. Other categories such as *Truck*, *Tram*, and *Misc* were excluded. Objects that were outside the range of the BEV map were also filtered out. The resulting bounding boxes were then saved in the format compatible with the YOLO format [22].

The BEV map PNG files and the corresponding label TXT files are then saved in a training or validation folder. Figure 3.3 shows the new dataset folder structure.

Table 3.1: Label format of the KITTI Object Detection dataset

Column	Field	Type	Description
Column 0	type	String	Object class (e.g., Car, Pedestrian, Cyclist)
Column 1	truncated	Float [0,1]	Fraction of object extending beyond image boundaries
Column 2	occluded	Integer [0–3]	Occlusion level (0=fully visible, 1=partly occluded, 2=largely occluded, 3=unknown)
Column 3	alpha	Float (rad)	Observation angle of the object
Column 4-7	2D bounding box	Float (px)	The top corners (x1, y1) and bottom corners (x2, y2) of the 2D bounding box pixel coordinates
Column 8-10	dimensions	Float (m)	3D object dimensions (height, width, length)
Column 11-13	location	Float (m)	Center location (x, y, z) of the 3D object in camera coordinates
Column 14	rotation_y	Float (rad)	Rotation around the Y-axis in camera coordinates (yaw)

Table 3.2: YOLO11-OBB Label Format

Column	Field	Type	Description
Column 0	class_id	int	Class index (starting from 0)
Column 1-2	x1, y1	float	First corner of the bounding box (normalized)
Column 3-4	x2, y2	float	Second corner of the bounding box (normalized)
Column 5-6	x3, y3	float	Third corner of the bounding box (normalized)
Column 7-8	x4, y4	float	Fourth corner of the bounding box (normalized)

To enable the training process to locate the dataset, a `dataset.yaml` file, as shown in Listing 3.1, is required. This file is based on the COCO dataset format and contains the absolute path to the dataset's main folder, the relative path for the training set, and the relative path to the validation set. Additionally, the object classes (ids) and their respective names are specified.

Listing 3.1: dataset.yaml configuration file for the created KITTI BEV dataset

```
# KITTI BEV Dataset using the COCO dataset structure

path: KITTI Dataset/BEVdata-obb
train: training/images
val: validation/images

names:
```

```
KITTI Dataset/
├── dataset.yaml
├── training/
│   ├── images/
│   └── labels/
└── validation/
    ├── images/
    └── labels/
```

Figure 3.3: Directory structure of the created dataset

```
  0: Pedestrian
  1: Car
  2: Cyclist
```

3.3.2 Training

To train the YOLO11n-OBB model, the official YOLO Python library, Ultralytics [20], was used. Installing the Ultralytics library was sufficient, as the necessary dependencies are installed automatically. In this work, Ultralytics version 8.3.61 was used for training. A YOLO model can be trained either by loading a pre-trained model or an untrained model. In this work, training was performed from scratch. The most important parameters for training include `data`, `epochs`, `batch`, and `imgsz`. `data` specifies the dataset, and `imgsz` specifies the image size with which the model should be trained. `imgsz` should be set to the same size as the BEV maps. The parameter `epochs` defines the number of training iterations over the dataset. `batch` specifies how many images are processed at once. Additional parameters include `save_period`, which specifies after how many epochs checkpoints of the trained model should be saved. By default, only `best.pt` and `last.pt` are saved. `patience` is used to stop training early if no improvement is observed. For example, with a patience value of 50, training will stop after 50 epochs without improvement in the validation metrics. The parameter `device` specifies which GPU should be used for training. A custom project folder and name can be specified with `project` and `name`.

After each epoch, the model is validated using the validation set. If the validation metrics improve, the model from the most recent epoch is saved as `best.pt`. Once training is complete, the best-performing model, `best.pt`, is validated again. The validation metrics include `Class`, `Images`, `Instances`, and `Box(P, R, mAP50, mAP50-95)`. `Class` is the respective object class, `Images` is the number of validation images, and `Instances` indicates how many ground truth boxes are contained within the validation images. `Box` refers to the bounding box evaluation. `P` is the precision, indicating how many of the detected objects are actually correct. `R` is the recall, indicating how many of the correct objects were found. `mAP50` is the mean average precision with an Intersection over Union (IoU) of 50. The IoU is a value between 0 and 1 that indicates the percentage

of overlap between two boxes relative to their combined area. With an IoU threshold of 0.5, at least 50% of the detected box and the actual box must overlap to be considered correct. `mAP50-95` is a COCO standard metric. It provides the mean of the mAP with IoU values from 0.5 to 0.95 in 0.05 increments. `mAP50-95` is also used to determine whether an epoch counts as `best.pt`.

The precision is calculated as:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.3}$$

with TP being True Positives and FP being False Positives.

Recall is calculated as:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.4}$$

The IoU is calculated as:

$$\text{IoU} = \frac{|B_{\text{pred}} \cap B_{\text{gt}}|}{|B_{\text{pred}} \cup B_{\text{gt}}|} \tag{3.5}$$

with B_{pred} being the predicted bounding box and B_{gt} being the ground truth bounding box.

Several models were trained with BEV map resolutions of 320, 448, 512, 640, 768, 832, and 1024. Models with resolutions of 480, 608, and 800 were also trained. However, the resolutions were later found to be incompatible with the TensorFlow Lite GPU delegate. Training was performed with an epoch count of 600 and a patience value of 50. Each model was trained for approximately 300 to 400 epochs, with a training duration of between 4 and 6 hours. Listing 3.2 shows an example of the Python script used for training with a BEV map size of 640.

Listing 3.2: Example Python script for training of the YOLO11n-OBB model with a BEV map size of 640

```
from ultralytics import YOLO

if __name__ == '__main__':
    model = YOLO("yolo11n-obb.yaml")
    results = model.train(data="KITTI Dataset/dataset.yaml",
        epochs=600,
        imgsz=640,
        batch=16,
        save_period=1,
        patience=50,
        project="yolo11n-obb",
        name="BEV_size_640",
        device=[0]
    )
```

Ultralytics also offers several augmentation settings during training [21]. In addition to training with standard settings, the model with a BEV resolution of 640 was also trained with additional augmentations. `degrees` and `flipud` were used here. `degrees` rotates the training image randomly within the specified range, and `flipud` flips the training image randomly upside down. Listing 3.3 shows an example of the Python script used for training with a BEV map size of 640 and additional augmentations.

Listing 3.3: Example Python script for training of the YOLO11n-OBB model with a BEV map size of 640 and additional augmentation

```
from ultralytics import YOLO

if __name__ == '__main__':
    model = YOLO("yolo11n-obb.yaml")
    results = model.train(data="KITTI Dataset/dataset.yaml",
        degrees=45.0,
        flipud=0.5,
        epochs=600,
        imgsz=640,
        batch=16,
        save_period=1,
        patience=50,
        project="yolo11n-obb",
        name="BEV_size_640",
        device=[0]
    )
```

Exporting the trained YOLO model to the TensorFlow Lite (tflite) format also uses the Ultralytics library. For export, the newer version 8.4.14 of Ultralytics was used, due to improvements in the export functionality. By default, the model is exported as a Float32 variant and as a quantized Float16 variant. Setting `int8=True` exports an additionally quantized INT8 model. The dataset used for training is required for integer quantization. Additionally, `nms=True` can be specified to add non-maximum suppression (NMS) post-processing to the model. This filters out duplicate object detections using rotated-IoU and then returns only the best 300 detections. During export, the model input size used for training is automatically detected, but it should still be specified using `imgsz` for safety. Listing 3.4 shows an example of the Python script used for exporting the trained model to the TensorFlow Lite format.

Listing 3.4: Example Python script for exporting of the YOLO11n-OBB model with a BEV map size of 640

```
from ultralytics import YOLO

if __name__ == '__main__':
    model = YOLO("best.pt")
    #This will export the model, trained with a BEV map of the size
        640, without including NMS
    model.export(format="tflite", imgsz=640, nms=False)
```

```
7
8       #This will export the model, trained with a BEV map of the size
            640, with included NMS
9       model.export(format="tflite", imgsz=640, nms=True)
10
11      #This will export the model, trained with a BEV map of the size
            640, with integer quantization and without including NMS
12      #model.export(format="tflite", imgsz=640, nms=False, int8=True,
            data="KITTI Dataset/dataset.yaml")
13
14      #This will export the model, trained with a BEV map of the size
            640, with integer quantization and with included NMS
15      #model.export(format="tflite", imgsz=640, nms=True, int8=True,
            data="KITTI Dataset/dataset.yaml")
```

3.4 Implementation into the CycleSafely pipeline

To enable flexible replacement of object detectors in the future, and also to allow for the selection of multiple object detectors, an Object Detector Factory was added to the pipeline. Listing 3.5 shows the added enumeration and Object Detector Factory.

Listing 3.5: DetectorType Enumeration and Object Detector Factory enable the easy addition of new object detectors.

```
1
2  /// add detection models here
3  enum DetectorType {
4    yolo,
5    sfa3d,
6  }
7
8  import 'package:cyclesafelymobile/pipeline/modules/object_detector/
       detector_type.dart';
9  import 'package:cyclesafelymobile/pipeline/modules/object_detector/
       object_detector.dart';
10 import 'package:cyclesafelymobile/pipeline/modules/object_detector/
       object_detector_config.dart';
11
12 class ObjectDetectorFactory {
13   static (ObjectDetector, ObjectDetectorConfig) create(DetectorType
         type) {
14     switch (type) {
15       case DetectorType.yolo:
16         final config = ObjectDetectorConfigYolo();
17         return (ObjectDetectorYolo(config: config), config);
18
19       case DetectorType.sfa3d:
20         final config = ObjectDetectorConfigSfa3d();
```

```
21          return (ObjectDetectorSfa3d(config: config), config);
22
23        /// add detection models here
24      }
25    }
26  }
```

To enable the Object Detector Factory, an interface, as shown in Listing 3.6, was created for both the Object Detector and the Object Detector Configuration.

Listing 3.6: The interface for the Object Detector and Object Detector Config.

```
1  abstract class ObjectDetector {
2    void loadModelFromAddress(int interpreterAddress, {extended = false
       });
3    void loadParallelModelsFromAddresses(int address1, int address2, {
       extended = false});
4    Future<DetectionsDTO> infer(Float32List lidar, TransformationDTO
       ctwMatrix);
5    void setVisualizer(Visualizer? visualizer);
6  }
7
8  abstract class ObjectDetectorConfig {
9    double get discretization;
10 }
```

3.4.1 LiDAR data preprocessing

To integrate the YOLO11-OBB model into the pipeline, the ObjectDetectorYolo class was created. This class performs three main steps: a pre-processing step in which the LiDAR data is processed and a BEV map is created; an inference step in which the BEV map is passed to the YOLO model and inference is performed; and a post-processing step in which the detections made by the YOLO model are processed.

The LiDAR data is first transformed into a BEV map. Two BEV maps are created: one facing forward and the other facing backward. For this, a boundary must be defined in the object detector's configuration file. This boundary matches the one used during training. There are two boundaries in total. Both boundaries have a range of 25 m to the left and right. However, one boundary extends 50 m forward, and the other extends 50 m backward. Only points within the defined boundaries are considered for BEV map creation. To ensure efficient processing, the implementation uses `Float32List` instead of nested list structures such as List<List<List<double>>>. The BEV map consists of three channels: density, height, and intensity. The density channel stores the number of points assigned to each grid cell in the BEV map. The height channel contains the maximum normalized z-value per cell. The intensity channel stores the intensity value of the point with the highest z-value in each cell. The map is constructed in a single pass

over the point cloud without prior sorting or filtering to avoid additional computational overhead.

For each point, its x and y coordinates are discretized to determine the corresponding grid cell in the BEV map. The height value is calculated by normalizing the points z-value using the predefined range $z_{\min} = -2.73\,\text{m}$ and $z_{\max} = 1.27\,\text{m}$:

$$h = \frac{z_{\max}^{\text{cell}} - z_{\min}}{z_{\max} - z_{\min}} \tag{3.6}$$

If a point has a higher normalized z-value than the currently stored value in the cell, the height channel is updated accordingly, and the point's intensity value is stored. The density channel's counter is incremented for each point assigned to the cell. After processing all points, the density values are normalized using logarithmic scaling:

$$d = \min\left(1.0, \frac{\log(N+1)}{\log(64)}\right) \tag{3.7}$$

where N denotes the number of points in the respective cell. To improve performance, the required logarithmic values are pre-calculated using a lookup table. To avoid iterating over the point cloud twice, a combined boundary is used to generate both BEV maps in a single pass. The resulting BEV map is then split into a front BEV map and a back BEV map for use in YOLO inference.

3.4.2 Inference and post-processing

Inference is performed using the tflite_flutter package. Experiments showed that it is advantageous to pass a Float32List buffer instead of a nested list.

Inference is done in two steps, once for a front-facing BEV map and once for a back-facing BEV map.

In the post-processing step, the detections made by the YOLO model for both BEV maps are then processed and filtered separately. A distinction must be made between the detections of the model with non-maximum suppression (NMS) included and the model without NMS included.

YOLO without NMS

The YOLO model without NMS, trained with the three classes Pedestrian, Car, and Cyclist, has an output shape of $[b][8][k]$. In this case, b is the batch size, which is 1 by default for the exported TFLite model. k is the number of detections made. This number depends on the model's input size. With an input size of 640, 8400 detections are made. The following values are available for each detection: Table 3.3 shows the output format for a detection. Each detection includes the normalized center coordinates (x and y), the normalized width and height, the confidence score for each class, and the yaw angle.

Index	Parameter	Description
0	c_x	Normalized x-coordinate of the bounding box center
1	c_y	Normalized y-coordinate of the bounding box center
2	w	Normalized width of the bounding box
3	h	Normalized height of the bounding box
4	s_{ped}	Confidence score for the Pedestrian class
5	s_{car}	Confidence score for the Car class
6	s_{cyc}	Confidence score for the Cyclist class
7	ψ	Orientation angle (yaw) in radians within $[-\pi/4, 3\pi/4]$

Table 3.3: Detections from YOLO11n-OBB without NMS.

The detections are read from the Float32List returned by tflite_flutter and decoded into a separate Detections class. During decoding, the highest score is determined from the three classes to assign the detection to a class. Only detections with a class score above 0.25 are included; all others are discarded. Filtering by score already removes most detections. If more than 300 detections remain after filtering by score, only the top 300 are included. c_x, c_y, width, and height are multiplied by the BEV width and height to get pixel coordinates. Next, NMS is applied to remove duplicate bounding boxes for the same object. NMS compares two bounding boxes. Three different steps determine whether they represent the same object. If they are the same object, the box with the higher score is used. Since we want to reduce computationally intensive rotated IoU comparisons, two other comparisons are performed first. First, the center distance of both bounding boxes is calculated. If the distance is below a threshold, it is highly likely that they are the same object. This comparison allows us to find possible duplicate boxes that would not be detected by a simple IoU comparison. If the two boxes are not identified as the same using the center distance, they are compared using axis-aligned IoU. Axis-aligned IoU does not take the rotation of the boxes into account and is less computationally intensive than rotated IoU. An IoU value above 0.40 indicates that the two boxes are the same. In the final step, the boxes are compared using rotated IoU. For this, the Sutherland-Hodgman algorithm [19] is used, in which the boxes are converted into polygons, and the IoU is calculated using polygon clipping. If the value is above 0.45, the two boxes are then considered equal.

YOLO with NMS

If the YOLO model was exported with NMS, the output shape has the form $[b][k][7]$. b corresponds to the batch size and k to the number of detections. k is limited to the top 300 detections. Table 3.4 shows the output format for a detection after NMS was applied by the model. The following values are present for each detection: normalized center coordinates (x and y), the normalized width and height, the confidence score, the class ID, and the yaw angle.

Index	Parameter	Description
0	c_x	Normalized x-coordinate of the bounding box center
1	c_y	Normalized y-coordinate of the bounding box center
2	w	Normalized width of the bounding box
3	h	Normalized height of the bounding box
4	s	Confidence score of the predicted class
5	c	Class ID of the detected object
6	ψ	Orientation angle (yaw) in radians within $[-\pi/4, 3\pi/4]$

Table 3.4: Detections from YOLO11n-OBB with integrated NMS.

Since NMS filtering has already been handled by the model, the detections only need to be filtered by score. Only detections with a score above 0.25 are included.

Finally, the completed detections from the front BEV and back BEV are combined and converted into world coordinates. These are then passed on to the visualizer and the object tracker.

3.4.3 Optimizations and other adjustments

Several optimizations were made to the implementation. While the old implementation relied heavily on nested lists, especially in BEV map generation, Float32Lists were used instead. A Float32List has a fixed size. Instead of creating new Float32Lists, they are pre-allocated at the beginning and reused later.

In addition to GPU Delegate and XNNPACK, support for NNAPI on Android has now been added. Instead of manually selecting whether a neural network should run on the CPU or GPU, NNAPI allows the system to select the optimal hardware accelerator, like dedicated NPU hardware. However, NNAPI support depends on the device and Android version, as it relies on vendor-specific drivers and available hardware accelerators. On some devices, NNAPI may fall back to CPU execution or only provide minor speed improvements.

The CycleSafely app features a Parallel Detector mode that runs two independent TensorFlow Lite interpreter instances concurrently. Each interpreter performs inference on a different input, either the front BEV map or the back BEV map. Previously, this mode was limited to CPU execution without XNNPACK. The updated version enables the use of GPU Delegate and XNNPACK. One interpreter is executed using the GPU Delegate, falling back to CPU execution with XNNPACK when needed. The second interpreter runs on the CPU with XNNPACK acceleration. Although this is not a partitioning of a single model across CPU and GPU, it allows two separate inference tasks to be executed in parallel on different hardware back ends. This approach improves overall performance and can outperform a GPU-only configuration, as both the GPU and CPU are utilized simultaneously.

CHAPTER 4

Evaluation

This chapter evaluates the proposed approach in terms of inference speed and detection performance.

4.1 Evaluation hardware

For evaluation on a desktop computer, an AMD 7800x3D processor and an NVIDIA 2080ti graphics card were used.

For mobile evaluation, a Motorola Edge 50 Neo was used. The phone was released in August 2024 and uses a MediaTek Dimensity 7300 SoC with 12 GB RAM and is considered a mid-range device. This SoC has 8 CPU cores with 4 performance cores and 4 efficiency cores. The performance cores are ARM Cortex-A78 with 2.5 GHz, and the efficiency cores are ARM Cortex-A55 with 2.0 GHz. The GPU is an ARM Mali-G615 MC2.

4.2 Comparison of the runtimes of different models

To find a faster replacement for SFA3D, the inference speed of various object detection models was compared. This comparison was performed on a desktop computer using PyTorch, as a model version in TensorFlow Lite (tflite) format was not always available. Table 4.1 shows various models and their measured FPS. Among the LiDAR-based 3D detectors, SFA3D was the fastest. It should be noted that YOLOv11n-OBB operates on BEV images using 2D convolutions and is therefore not directly comparable to full 3D LiDAR-based detectors.

Table 4.1: Speed comparison of different models measured on a desktop computer. Times for SFA3D and YOLO include post-processing. YOLO speed taken from validation measurements. Types indicate the input representation and the type of detection (2D/3D). BEV-based models project the 3D point cloud into a bird's-eye view. Input resolution refers to the BEV map size; entries marked N/A have no BEV input.

Model	Type	Input resolution	FPS
SECOND [27]	LiDAR 3D (voxel-based)	N/A	21.4
PointPillars [12]	LiDAR 3D (pillar-based)	N/A	46.6
PV-RCNN [16]	LiDAR 3D (voxel + point-based)	N/A	9.1
Part-A2 [17]	LiDAR 3D (voxel + point-based)	N/A	8.7
SFA3D [5]	BEV-based 3D	608×608	85.49
Complex-YOLOv4 [18] [6]	BEV-based 3D	608×608	44.07
YOLOv11n-OBB	BEV-based 2D	640×640	250

4.3 Comparison of the detection performance of YOLO compared to SFA3D

The detection performance of the YOLO11n-OBB models, trained on various BEV map sizes, was evaluated using two approaches. First, using the Ultralytics library, which is also used during training. Ultralytics uses the COCO metric for its evaluation. Second, using the KITTI Python evaluation tool [26], which performs the official KITTI evaluation. The YOLO results are then compared with the evaluation results of SFA3D, which also uses the Python tool [14]. All evaluations of the detection performance were performed on a desktop computer using the full validation dataset of 1481 samples.

4.3.1 Evaluation using Ultralytics

Table 4.2 shows how the BEV map size affects the mAP. The mAP metric used has an IoU of 0.5. A BEV size of 320 is already sufficient for good car detection. With a higher BEV resolution, the mAP for cars hardly changes. For pedestrians and bicycles, the BEV resolution has a greater impact, likely due to their smaller object size and lower point density in the BEV representation. Here, the mAP increases significantly when the resolution is increased. The model with a BEV size of 640, which was trained with additional augmentation, shows lower mAP values. Although the values are lower, the model demonstrated improved temporal stability during object rotations in the KITTI Raw sequence used for testing.

Figure 4.1 visualizes the Ultralytics results. It shows that from a BEV resolution of 640 onwards, the mAP values for pedestrians and cyclists no longer increase as sharply.

Table 4.2: Ultralytics validation results (mAP@0.5) for different BEV input sizes.

BEV Size	Car	Pedestrian	Cyclist	All
320	0.987	0.505	0.730	0.741
448	0.989	0.681	0.767	0.812
512	0.989	0.745	0.836	0.857
640	0.989	0.831	0.891	0.903
640 (aug)	0.959	0.716	0.723	0.799
768	0.988	0.859	0.900	0.916
832	0.987	0.859	0.905	0.917
1024	0.987	0.898	0.931	0.939

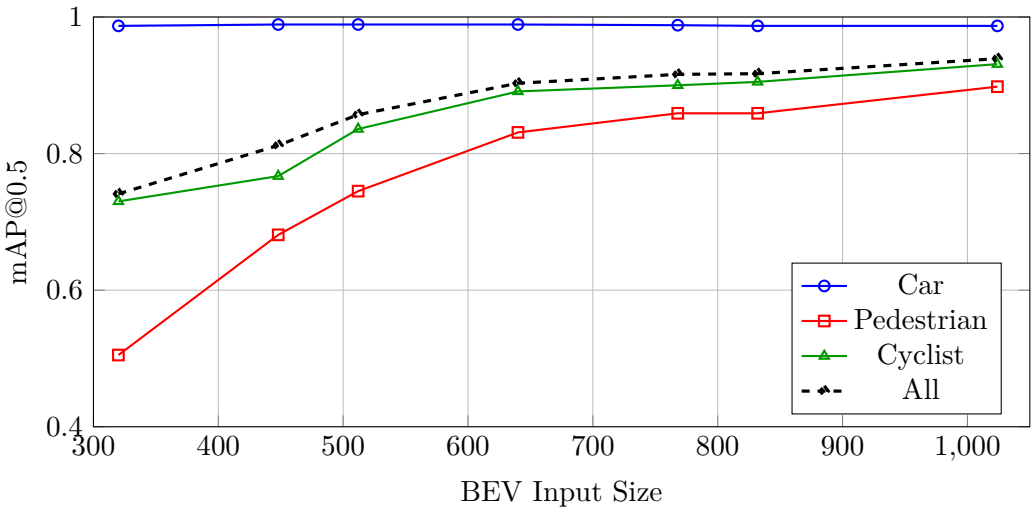


Figure 4.1: Ultralytics validation performance (mAP@0.5) for different BEV input resolutions (320, 448, 512, 640, 768, 832, 1024). The plot shows class-wise AP for Car, Pedestrian, and Cyclist, as well as the mean AP (All). The overall performance (All) increases consistently with higher BEV resolution. The model with augmentation has been excluded.

4.3.2 Evaluation using KITTI

The KITTI benchmark follows the PASCAL evaluation criteria for its evaluation. A bounding box overlap of 70% (IoU=0.7) is used for cars, and an overlap of 50% (IoU=0.5) for pedestrians and cyclists. KITTI uses three difficulty levels for its evaluation: Easy, Moderate, and Hard. Filtering for the validation bounding box labels is performed in the camera's 2D image plane. The following conditions apply to the difficulty levels. Only labels within the specified conditions will be used for evaluation [9][7]:

- Easy: Bounding boxes must have a minimum height of 40 px, and their maximum

Table 4.3: BEV Average Precision (%) on KITTI validation set for different BEV input sizes using YOLO11n-OBB.

BEV Size	Car AP@0.70			Pedestrian AP@0.50			Cyclist AP@0.50		
	Easy	Mod	Hard	Easy	Mod	Hard	Easy	Mod	Hard
320	90.78	90.37	89.97	31.62	30.88	23.93	49.52	49.37	41.93
448	90.87	90.59	90.33	42.09	34.07	33.85	48.11	49.43	49.16
512	90.82	90.52	90.21	48.69	48.36	41.21	69.97	61.22	60.96
640	90.66	90.56	90.40	76.11	68.15	60.16	77.52	70.33	69.64
640 (aug)	89.94	88.53	80.31	49.91	42.03	42.06	64.88	49.33	48.73
768	90.88	90.71	90.44	76.32	69.08	68.65	77.77	76.79	69.62
832	90.86	90.63	90.33	77.21	69.26	68.79	86.73	78.90	78.38
1024	90.78	90.61	90.42	79.14	78.52	70.58	87.64	86.62	79.14
SFA3D	97.52	89.62	89.78	68.14	69.36	65.41	82.11	75.41	75.64

occlusion level must be “fully visible”. Truncation is limited to a maximum of 15%.

- Moderate: Bounding boxes must have a minimum height of 25 px, and their maximum occlusion level must be “partly occluded”. Truncation is limited to a maximum of 30%.

- Hard: Bounding boxes must have a minimum height of 25 px, and their maximum occlusion level must be “difficult to see” (i.e., largely occluded). Truncation is limited to a maximum of 50%.

Table 4.3 shows that the YOLO models achieve a constant mAP across the three difficulty levels for cars. For pedestrians and cyclists, the mAP decreases for Moderate and Hard. This indicates that car detection is less sensitive to BEV resolution, presumably due to their larger size and more consistent point cloud representation. For comparison, the evaluation results from SFA3D are included. SFA3D shows a better value for Easy for cars. The remaining values are comparable or slightly better than YOLO with a BEV size of 640 and above. SFA3D itself uses a BEV size of 608.

Table 4.4 summarizes the KITTI evaluation results, taking only the moderate results. Additionally, the mean was calculated from the three classes.

Figure 4.2 shows the KITTI evaluation results for the YOLO models with different BEV sizes. The results show that the mAP values for pedestrians and cyclists increase sharply up to a BEV size of 640 and then only increase slowly.

Table 4.4: Mean Average Precision (mAP) across classes for different BEV input sizes on the KITTI validation set (Moderate difficulty).

BEV Size	Car AP	Pedestrian AP	Cyclist AP	Mean AP
320	90.37	30.88	49.37	56.87
448	90.59	34.07	49.43	58.03
512	90.52	48.36	61.22	66.70
640	90.56	68.15	70.33	76.35
640 (aug)	88.53	42.03	49.33	59.96
768	90.71	69.08	76.79	78.86
832	90.63	69.26	78.90	79.60
1024	90.61	78.52	86.62	**85.25**

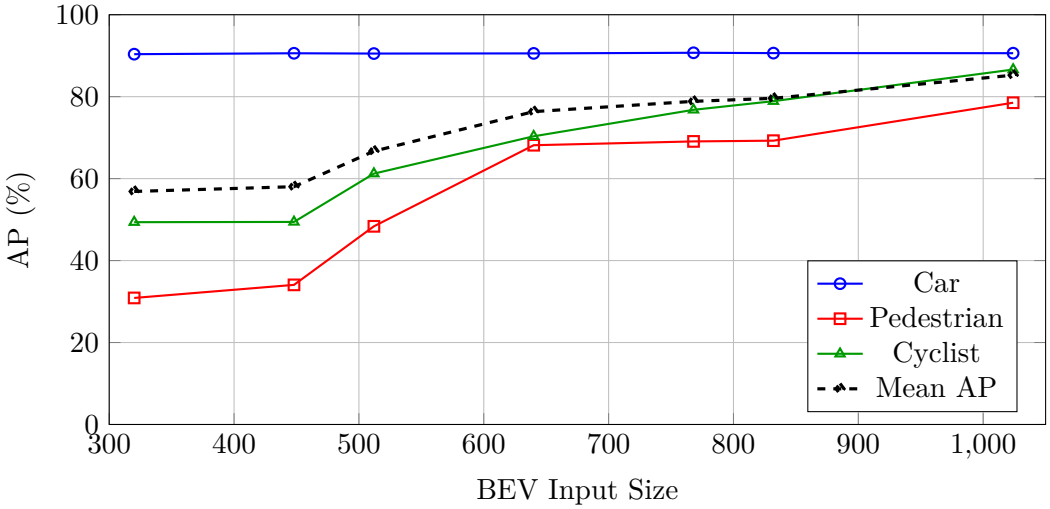


Figure 4.2: Detection performance on the KITTI validation set (Moderate difficulty) for different BEV input resolutions (320, 448, 512, 640, 768, 832, 1024). The plot shows class-wise AP for Car, Pedestrian, and Cyclist, as well as the mean AP. The mean AP across classes increases significantly with higher BEV resolution.

4.4 Runtime analysis of YOLO models with different BEV resolutions on mobile devices

This section presents runtime measurements of different YOLO models within the Cycle-Safely application. For the measurement, the sequence `2011_09_26_drive_0005_sync` from the city category of the KITTI Raw dataset [8] was used. The sequence comprises 154 frames, which were processed sequentially. Each model was measured with all 154 frames; however, the first two frames were excluded to account for model warm-up time. During the measurement, the test device showed no overheating or thermal throttling

of speed. In both the SFA3D and the YOLO implementation, inference is performed twice per frame, once for the front BEV map and once for the back BEV map. Each model was evaluated using four different configurations: a base measurement without enabling delegates, a measurement with XNNPACK enabled, a measurement with GPU delegate enabled, and a measurement for parallel detection. Four threads were selected for XNNPACK, as this configuration yielded the best performance. Parallel mode was used with both XNNPACK and GPU delegate enabled. All measurements were performed using the release build of the CycleSafely application, as it provides better performance than the debug build.

First, the old SFA3D implementation was measured. SFA3D offers three models to choose from: the standard model with Float32, a model with Float16 quantization, and a model with dynamic range quantization (DRQ). The Table 4.5 shows the measurement results. Without a delegate, the DRQ model, with a speed of 4 seconds per frame, was the fastest. In parallel mode, the standard model and the Float16 model achieved almost the same speed of 2.5 seconds. This corresponds to approximately 0.4 FPS. This confirms that SFA3D is not suitable for real-time on mobile.

Table 4.5: Inference performance of SFA3D models on Android (154 frames per configuration). FPS corresponds to the Parallel (GPU + XNNPACK) configuration.

Resolution	Base [ms]	XNNPACK [ms]	GPU [ms]	Parallel [ms]	FPS (Parallel)
SFA3D Standard	9744.63	3327.3	2882.51	2504.62	0.40
SFA3D Float16	9789.0	3318.28	2883.87	2522.64	0.40
SFA3D DRQ	4014.16	3960.38	2704.72	2768.1	0.36

The YOLO models were measured both with and without an integrated NMS. Table 4.6 shows the measurements of the YOLO models with different BEV sizes without integrated NMS. The results show that XNNPACK leads to a significant increase in speed compared to not using a delegate. The GPU delegate provides slightly better performance than XNNPACK. The best results were measured in parallel mode. The BEV size has a significant impact on speed. With a BEV size of 320, 23.8 FPS are achieved in parallel mode. This is approximately 60 times faster than SFA3D under comparable conditions. With a BEV size of 1024, only 2.3 FPS are achieved in parallel mode, although this is still approximately 6 times faster than SFA3D.

Figure 4.3 visualizes the results. The drop in FPS with increasing BEV size is immediately noticeable.

Table 4.7 shows the measurement results for the YOLO models with integrated NMS. The measurement results behave differently here than with YOLO without an integrated NMS. Measurements without delegates show that NMS introduces a constant overhead of approximately 230ms in the model, independent of the BEV resolution. XNNPACK speeds up execution, but the NMS overhead remains almost unchanged. The measurements for GPU Delegate show the greatest differences. With GPU Delegate, execution is almost

Table 4.6: Inference performance of YOLO11n-OBB models without NMS on different BEV resolutions (320, 448, 512, 640, 768, 832, 1024) using different configurations (base, XNNPACK, GPU, Parallel) on Android (154 frames per configuration). FPS corresponds to the Parallel (GPU + XNNPACK) configuration.

Resolution	Base [ms]	XNNPACK [ms]	GPU [ms]	Parallel [ms]	FPS (Parallel)
320	246.82	78.91	83.78	41.99	23.82
448	496.64	137.73	141.97	92.64	10.79
512	652.95	172.08	148.76	116.74	8.57
640	1046.88	267.18	216.30	167.97	5.95
768	1528.30	387.03	273.32	238.82	4.19
832	1809.74	453.35	330.51	280.09	3.57
1024	2807.71	721.58	465.03	440.29	2.27

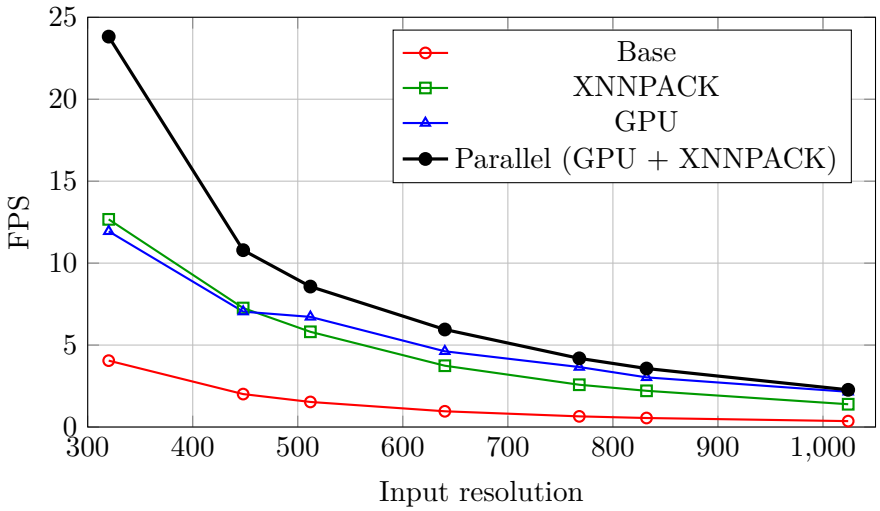


Figure 4.3: Inference speed of YOLO11n-OBB models without NMS for different BEV input resolutions (320, 448, 512, 640, 768, 832, 1024). The plot shows the average FPS for different configurations (base, XNNPACK, GPU, Parallel).

twice as slow as with XNNPACK and even slower than without Delegate. This indicates that the YOLO models with integrated NMS are not effectively accelerated by the GPU delegate. A possible explanation is that the NMS step involves specific operations that are not well supported by the GPU delegate, causing a fallback to CPU execution. The resulting synchronization and data transfer overhead between the GPU and CPU can outweigh the benefits of GPU acceleration, leading to an overall slower execution than if the entire operation were performed on the CPU. Parallel mode improves performance compared to GPU execution alone, but is not as fast as XNNPACK alone. With a BEV map size of 320, a speed of 4.16 FPS is achieved, which is approximately 6 times slower than without integrated NMS. With a BEV map size of 1024, it is 1.14 FPS, which is roughly twice as slow.

Table 4.7: Inference performance of YOLO11n-OBB models with integrated NMS on different BEV resolutions (320, 448, 512, 640, 768, 832, 1024) using different configurations (base, XNNPACK, GPU, Parallel) on Android (154 frames per configuration). FPS corresponds to the XNNPACK configuration.

Resolution	Base [ms]	XNNPACK [ms]	GPU [ms]	Parallel [ms]	FPS (XNNPACK)
320	476.99	243.57	498.74	267.24	4.16
448	734.89	304.62	759.27	328.93	3.28
512	873.56	339.86	926.53	363.98	2.94
640	1277.99	432.05	1300.32	459.68	2.31
768	1752.48	548.91	1787.72	583.01	1.82
832	2045.11	616.90	2079.42	656.84	1.62
1024	3033.66	874.84	3112.08	932.03	1.14

Figure 4.4 visualizes the measured results. The results show that the GPU delegate does not provide a performance improvement and is close to the base value.

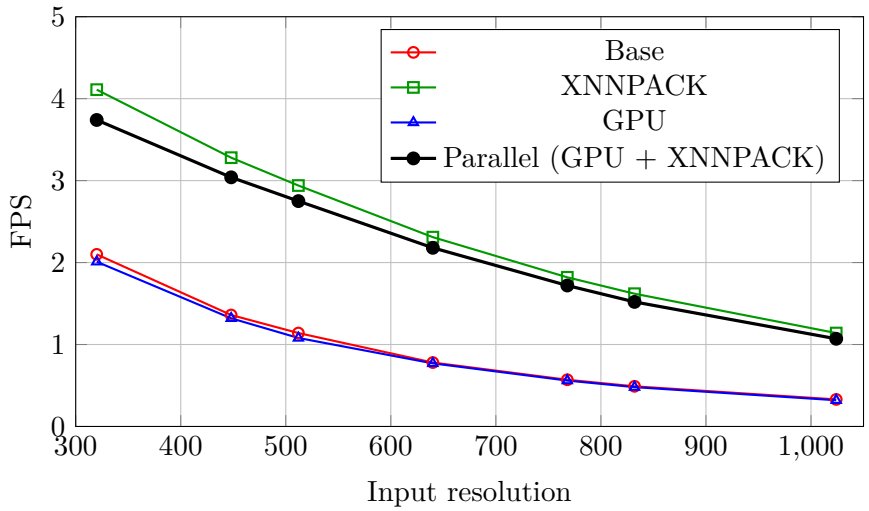


Figure 4.4: Inference speed of YOLO11n-OBB models with integrated NMS for different BEV input resolutions (320, 448, 512, 640, 768, 832, 1024). The plot shows the average FPS for different configurations (base, XNNPACK, GPU, Parallel).

The results show that the choice of BEV map resolution leads to a trade-off between detection accuracy and runtime performance. While higher BEV map resolutions help with object detection, especially smaller objects like pedestrians and cyclists, they also significantly reduce inference speed. A BEV map resolution of 640 provides a good balance between detection accuracy and speed, reaching around 6 FPS on a mid-range mobile device. Additionally, performing NMS outside the model significantly improved runtime performance, especially when using GPU acceleration.

CHAPTER 5

Future Work & Conclusion

This work showed that replacing SFA3D with YOLOv11n-OBB greatly improves runtime performance on mobile devices while maintaining good detection accuracy. With a BEV map resolution of 320, we achieved up to 24 FPS, while a resolution of 640 still reached approximately 6 FPS on a mid-range mobile device. This allows for near real-time performance on mobile hardware. However, the YOLO models still require further optimization. Experiments revealed occasional temporal instability, where objects are not detected consistently across consecutive frames. Additional training with more focus on data augmentation could improve temporal consistency and robustness.

YOLO supports input representations with more than three channels. One approach to improving YOLO's accuracy would be to work with BEV maps, which have more than three channels. Currently, YOLO is used with three channels: density, height, and intensity. Using more than three channels allows for more information, for example, by splitting the height map into multiple vertical slices. This additional information could lead to improved accuracy.

A limitation of YOLOv11-OBB is that it only predicts 2D oriented bounding boxes in BEV maps. This is sufficient for the current CycleSafely pipeline. Should 3D information be required in the future, it could be approximated from the BEV representation. The height map could be used for this purpose.

As with the SFA3D implementation, YOLO also performs inference twice. In addition to a slower runtime, this introduces a further limitation. If a car is located exactly in the middle where the two BEV maps meet, it may not be detected. A possible solution would be to use a single BEV map with an extended range, like 70 m to 80 m, and placing the LiDAR origin at the center.

Newer YOLO iterations are continuously being developed, promising further improvements in accuracy and runtime performance. For example, YOLO26 [23] achieves better accuracy performance than YOLO11 while improving runtime performance.

Overall, the results show that 2D BEV-based approaches with YOLO are a good alternative to traditional 3D LiDAR detectors for mobile applications. The approach achieves a reasonable trade-off between accuracy and runtime, making it a suitable alternative for real-time cyclist assistance systems.

Overview of Generative AI Tools Used

No AI tools were used to write this work.

List of Figures

List of Tables

Bibliography

[1] Waleed Ali, Sherif Abdelkarim, Mahmoud Zidan, Mohamed Zahran, and Ahmad El Sallab. YOLO3D: End-to-End Real-Time 3D Oriented Object Bounding Box Detection from LiDAR Point Cloud. In *Computer Vision – ECCV 2018 Workshops: Munich, Germany, September 8-14, 2018, Proceedings, Part III*, page 716–728, Berlin, Heidelberg, 2018. Springer-Verlag.

[2] Atanasko. Ultralytics – wod_obb_convert_dataset branch, 2024. `https://github.com/atanasko/ultralytics/blob/wod_obb_convert_dataset/`, GitHub repository, accessed: 2026-03-03.

[3] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuScenes: A Multimodal Dataset for Autonomous Driving . In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11618–11628, Los Alamitos, CA, USA, June 2020. IEEE Computer Society.

[4] R. Qi Charles, Hao Su, Mo Kaichun, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 77–85, July 2017.

[5] Nguyen Mau Dung. Super-Fast-Accurate-3D-Object-Detection-PyTorch, 2020. `https://github.com/maudzung/Super-Fast-Accurate-3D-Object-Detection`, GitHub repository, accessed: 2026-03-08.

[6] Nguyen Mau Dung. Complex YOLOv4, 2022. `https://github.com/maudzung/Complex-YOLOv4-Pytorch`, GitHub repository, accessed: 2026-03-13.

[7] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Bird's Eye View Evaluation 2017, 2017. `https://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=bev`, Online, accessed: 2026-03-14.

[8] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[9] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[10] Matyas Hoffer-Toth. CycleSafely on mobile. Bachelor's thesis, Research Unit of Computer Graphics, Institute of Visual Computing and Human-Centered Technology, Faculty of Informatics, TU Wien, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, August 2024.

[11] Glenn Jocher and Jing Qiu. Ultralytics YOLO11, 2024.

[12] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. PointPillars: Fast Encoders for Object Detection From Point Clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12689–12697, 2019.

[13] Jiageng Mao, Niu Minzhe, ChenHan Jiang, hanxue liang, Jingheng Chen, Xiaodan Liang, Yamin Li, Chaoqiang Ye, Wei Zhang, Zhenguo Li, Jie Yu, Chunjing XU, and Hang Xu. One Million Scenes for Autonomous Driving: ONCE Dataset. In J. Vanschoren and S. Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

[14] Mau Dung Nguyen and contributors. Pull Request #47 – SFA3D Repository, 2021. `https://github.com/maudzung/SFA3D/pull/47`,GitHub repository, accessed: 2026-03-08.

[15] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: deep hierarchical feature learning on point sets in a metric space. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 5105–5114, Red Hook, NY, USA, 2017. Curran Associates Inc.

[16] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection . In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10526–10535, Los Alamitos, CA, USA, June 2020. IEEE Computer Society.

[17] Shaoshuai Shi, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. From Points to Parts: 3D Object Detection From Point Cloud With Part-Aware and Part-Aggregation Network . *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 43(08):2647–2664, August 2021.

[18] Martin Simon, Stefan Milz, Karl Amende, and Horst-Michael Gross. Complex-YOLO: An Euler-Region-Proposal for Real-Time 3D Object Detection on Point Clouds. In *Computer Vision – ECCV 2018 Workshops: Munich, Germany, September 8-14, 2018, Proceedings, Part I*, page 197–209, Berlin, Heidelberg, 2018. Springer-Verlag.

[19] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42, January 1974.

[20] Ultralytics. Model Training with Ultralytics YOLO, 2023. `https://docs.ultralytics.com/modes/train/`, Updated 2026, accessed: 2026-03-13.

[21] Ultralytics. Model Training with Ultralytics YOLO: Augmentation Settings and Hyperparameters, 2023. `https://docs.ultralytics.com/modes/train/#augmentation-settings-and-hyperparameters`, Updated 2026, accessed: 2026-03-13.

[22] Ultralytics. Oriented Bounding Box (OBB) Datasets Overview, 2023. `https://docs.ultralytics.com/datasets/obb/`, Updated 2026, accessed: 2026-03-14.

[23] Ultralytics. YOLO26 vs YOLO11: A Generational Leap in Vision AI, 2026. `https://docs.ultralytics.com/compare/yolo26-vs-yolo11/`, Online, accessed: 2026-03-14.

[24] Waymo. Waymo Open Dataset: An autonomous driving dataset, 2019. `https://www.waymo.com/open`, accessed: 2024-11-15.

[25] Benjamin Wilson, William Qi, Tanmay Agarwal, John Lambert, Jagjeet Singh, Siddhesh Khandelwal, Bowen Pan, Ratnesh Kumar, Andrew Hartnett, Jhony Kaesemodel Pontes, Deva Ramanan, Peter Carr, and James Hays. Argoverse 2: Next Generation Datasets for Self-driving Perception and Forecasting. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS Datasets and Benchmarks 2021)*, 2021.

[26] Yan Yan. kitti-object-eval-python: Fast KITTI Object Detection Evaluation in Python, 2019. `https://github.com/traveller59/kitti-object-eval-python`, GitHub repository, accessed: 2026-03-08.

[27] Yan Yan, Yuxing Mao, and Bo Li. SECOND: Sparsely Embedded Convolutional Detection. *Sensors*, 18(10), 2018.