DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Semester Thesis

Autonomous Driving Simulator and Benchmark on Neurorobotics Platform

Jun Meng

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Semester Thesis

Autonomous Driving Simulator and Benchmark on Neurorobotics Platform

Autonomes Fahren Simulator und Benchmark auf Neurorobotics Plattform

Author:	Jun Meng
Supervisor:	Prof. Dr.-Ing. habil. Alois C. Knoll
Advisor:	Liguo Zhou
Submission Date:	Dec. 15, 2022

I confirm that this semester thesis is my own work and I have documented all sources and material used.

Garching bei München, Dec. 15, 2022 Jun Meng

Abstract

Autonomous driving simulator development is as important as the development of autonomous driving pipeline itself. Basing on the high qualified simulation framework Neurorobotic Platform, the cross-platform game engine Unity and the well developed robotic communication framework Robot operation System (ROS), we plan to develop an autonomous driving simulator of high performance, which consists of AI brain engine, weather engine, robot engine, pedestrian engine, AI car engine and city+cars engine. To satisfy the requirements on real-time and capability, we develop the software basing on ROS2. Two of the basic applications are object detection and depth estimation, which are implemented with YOLOv5 and Semi-global Block Matching (SGBM) correspondingly. In terms of object detection, we trained the YOLO model for our own scenarios specially; In terms of depth estimation, we use the mature SGM algorithm to generate a global depth map, and then combine the results of object detection to obtain the depth value at the center point of the object bounding box as the result of object distance estimation.

Kurzfassung

Die Entwicklung von Simulatoren für autonomes Fahren ist genauso wichtig wie die Entwicklung der autonomen Fahrpipeline selbst. Basierend auf dem hochqualifizierten Simulationsframework Neurorobotic Platform, der plattformübergreifenden Game-Engine Unity und dem gut entwickelten Roboterkommunikations-Framework Robot Operating System (ROS) planen wir, einen autonomen Fahrsimulator mit hoher Leistung zu entwickeln, der aus KI-Gehirn-Engine, Wetter-Engine, Roboter-Engine, Fußgänger-Engine, AI-Auto-Engine und City+Cars-Engine besteht. Um den Anforderungen an Echtzeit und Leistungsfähigkeit gerecht zu werden, entwickeln wir die Software auf Basis von ROS2. Zwei der Basisanwendungen sind Objekterkennung und Tiefenschätzung, die mit YOLOv5 und Semi-global Block Matching (SGBM) entsprechend umgesetzt werden. In Bezug auf die Objekterkennung haben wir das YOLO-Modell speziell für unsere eigenen Szenarien trainiert; In Bezug auf die Tiefenschätzung verwenden wir den ausgereiften SGM-Algorithmus, um eine globale Tiefenkarte zu generieren, und kombinieren dann die Ergebnisse der Objekterkennung, um den Tiefenwert am Mittelpunkt des Objektbegrenzungsrahmens als Ergebnis der Objektentfernungsschätzung zu erhalten.

Contents

1 Introduction

1.1 Motivation

Developing autonomous driving platform is rife with challenges. Chief among them remains the estimated 8.8 billion miles of road testing that would be required to ensure fully autonomous cars are safe enough to hit the road. Besides, most of the errors occurring on real platforms and corner cases are hard to reproduce, and can result in severe traffic accidents.

The autonomous driving simulation was developed to reduce field testing times and to separate problems for testing, making debugging easier, and finally reducing damages of real vehicles. This is also convenient for student projects, where access to an expensive and complex real platform cannot always be guaranteed.

1.2 Contribution

In general, we attempt to achieve the following functionalities: send the collected images, point clouds from virtual sensors in Unity to ROS2 in the corresponding ROS message format; Implement object detection and stereo depth estimation algorithms in the corresponding ROS execution units (i.e. nodes), process the data published from Unity, and obtain the information we need: the detected objects in the image and their distances to our ego agent in the simulation environment; Finally publish and visualize them back in Unity.

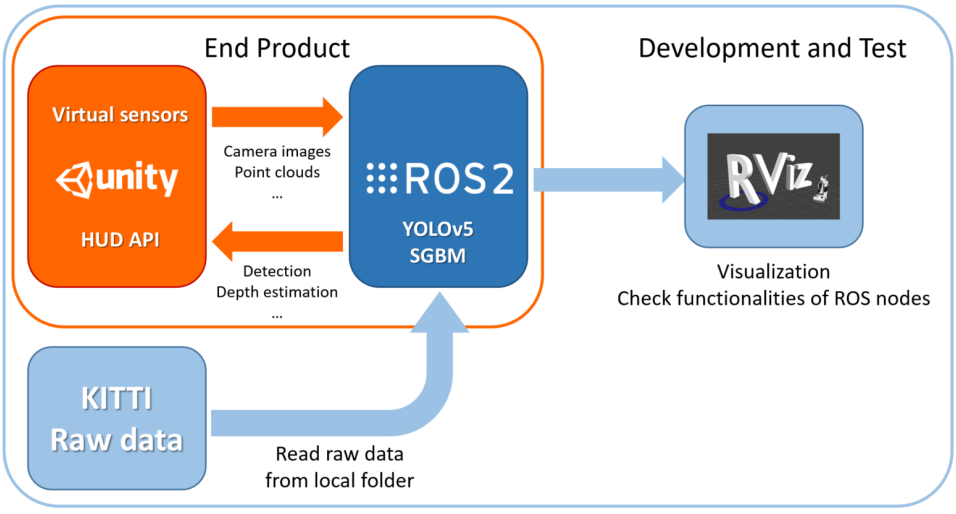


Figure 1.1: Development concept

My contribution to this project is ROS2-related software development, using KITTI dataset as the substitution of those that are supposed to be published from Unity. A summary of work needed to be done is as follows:

(1) Publish KITTI raw data in ROS2 and visualize them in Rviz. Prepare to test the algorithm implementations.

(2) Develop ROS2 application for object detection by implementing YOLOv5 algorithm.

(3) Develop ROS2 application for stereo depth estimation by implementing SGBM algorithm.

(4) Visualization and inspect estimated depths using point cloud ground truth.

1.3 Outline

The first chapter introduces an overview of this project. In chapter 2, the softwares like ROS and Unity and the algorithms like YOLO and SGBM for our development are introduced. Chapter 3 gives the detailed contents about algorithm implementation and parameter configuration. Finally in chapter 4 I draw the conclusion and discuss about some possible improvements of our development.

2 Background

This chapter introduces the tools, datasets and softwares we used for the development of our autonomous driving simulation platform.

2.1 Neurorobotics Platform (NRP)

The Neurorobotics Platform (NRP) [1] is an integrative simulation framework: it enables in silico experimentation and embodiment of brain models inside virtual agents interacting with realistic simulated environments.

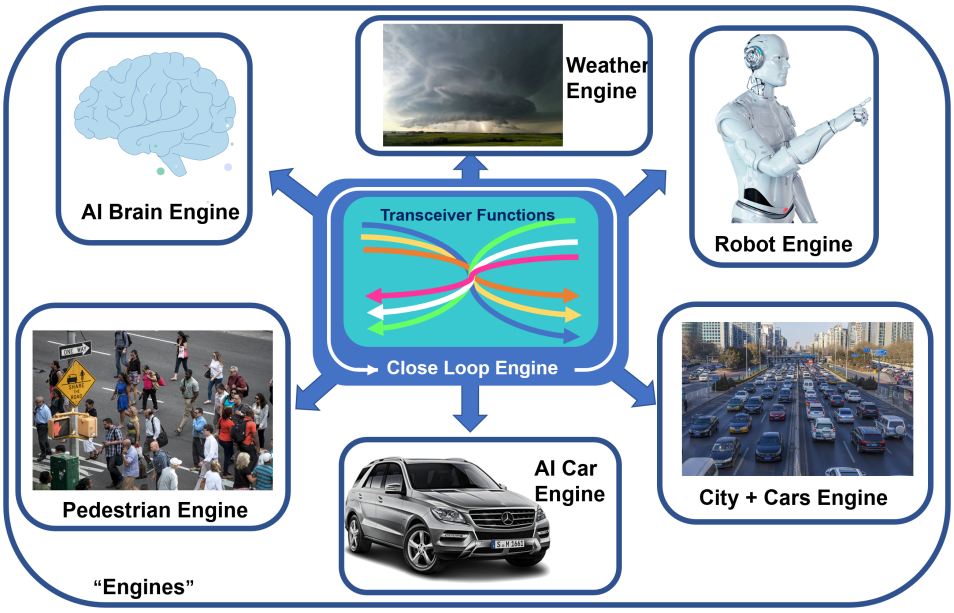


Figure 2.1: NRP basic model

Drawing upon the potential of both neurosciences and Artificial Intelligence in robotics, the NRP allows its users to observe, analyse and test the emergence of behavioural patterns in virtual agents controlled by state-of-the-art models of brain architecture and functions. The result is an unprecedented approach to simulation in which theoretical brain models can be checked against data-driven models thanks to simulations that take into account the dynamics of both the environment and of the agent itself.

NRP offers full feature set of a robot simulator and at the same time is arbitrarily scalable for massively parallel robotics experiments. In principle, self-driving cars are also a type of

robot. Based on NRP we develop an autonomous driving simulation system featuring high parallelism, minimalisim and photorealism.

2.2 Examples of autonomous driving simulation platform

Autonomous vehicles as cyber physical systems can be divided into sensing, computing, and actuation modules. Sensing devices, such as cameras, and laser scanners (LiDAR) are mostly used for Autonomous driving in urban areas. Computation is a foremost module of self-driving technology. Scene understanding, for instance, requires the location of ego vehicle, detection of static and moving obstacles around the ego vehicle, and prediction modules for predicting detected objects trajectories, whereas path planning is handled by motion based and mission-based modules. Actuation modules handle stroking and steering. These twisted control commands are generated by the path following module. Each module works with its own set of algorithms, uses deep learning networks, sensor data fusion etc.

Compared to other robots, self-driving cars have different requirements in terms of real-time behaviour and calculation speed. They have to master a wide range of manoeuvre from precise path-finding in parking situations to high-speed driving on highways [2]. There are already several well-developed autonomous driving simulators. Having insights through them, we can get some basic principles for our developing works.

2.2.1 CARLA

CARLA [3] has been developed from the ground up to support development, training, and validation of autonomous driving systems. In addition to open-source code and protocols, CARLA provides open digital assets, such as urban layouts, buildings, vehicles etc. The highlighted features of CARLA are:

(1) **Scalability via a server multi-client architecture:** multiple clients in the same or in different nodes can control different actors.

(2) **Flexible API:** CARLA exposes a powerful API that allows users to control all aspects related to the simulation, including traffic generation, pedestrian behaviors, weathers, sensors, and much more.

(3) **Autonomous Driving sensor suite:** users can configure diverse sensor suites including LIDARs, multiple cameras, depth sensors and GPS among others.

(4) **A wide range of environmental conditions:** The simulator supports two lighting conditions – midday and sunset – as well as nine weather conditions, differing in cloud cover, level of precipitation, and the presence of puddles in the streets.

(5) **Combination Unity and ROS2:** Substitute Unity as the simulation environment and accomplish Unity's ROS2 integration.

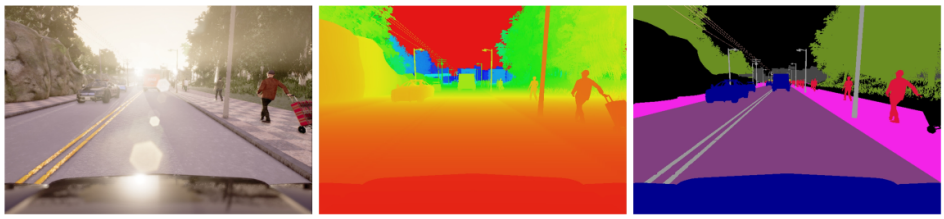


Figure 2.2: Three of the sensing modalities provided by CARLA. From left to right: normal vision camera, ground-truth depth, and ground-truth semantic segmentation

2.2.2 Autoware

Autoware [4] is an open-source software stack for self-driving vehicles, built on the Robot Operating System (ROS). It includes all of the necessary functions to drive an autonomous vehicles from localization and object detection to route planning and control, and was created with the aim of enabling as many individuals and organizations as possible to contribute to open innovations in autonomous driving technology.

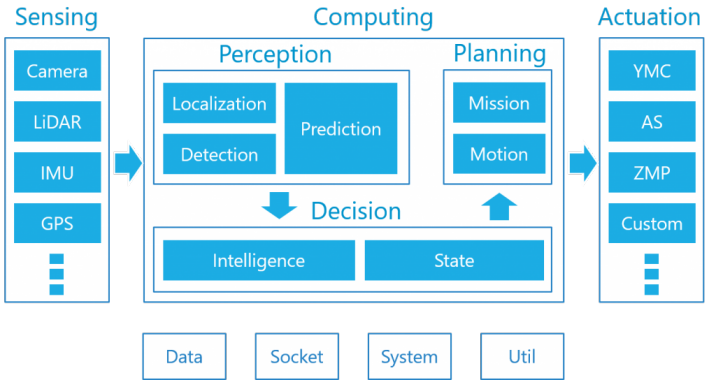


Figure 2.3: Autoware software architecture

2.3 Robot Operation System (ROS)

Robot Operating System (ROS or ros) is an open-source robotics middleware suite. It is not an operating system (OS) but a set of software frameworks for robot software development. Sometime before 2007, the first pieces of what eventually would become ROS began coalescing at Stanford University. Later on Willow Garage began developing the PR2 robot as a follow-up to the PR1, and ROS as the software to run it. Groups from more than twenty institutions made contributions to ROS, both the core software and the growing number of packages which worked with ROS to form a greater software ecosystem. ROS has provided the robot community with a relatively complete set of intermediate layers, tools, software and even common interfaces and standards. It can be said that with ROS, developers in the field of

robotics industry can quickly develop system prototypes and do testing and verification without reinventing wheels. Software in the ROS Ecosystem can be separated into three groups:

- language- and platform-independent tools used for building and distributing ROS-based software;
- ROS client library implementations such as roscpp, rospy, and roslisp;
- packages containing application-related code which uses one or more ROS client libraries.

ROS was designed to be open source, intending that users would be able to choose the configuration of tools and libraries which interacted with the core of ROS so that users could shift their software stacks to fit their robot and application area. As such, there is very little which is core to ROS, beyond the general structure within which programs must exist and communicate. In one sense, ROS is the underlying plumbing behind nodes and message passing. However, in reality, ROS is not only that plumbing, but a rich and mature set of tools, a wide-ranging set of robot-agnostic abilities provided by packages, and a greater ecosystem of additions to ROS (Figure 2.4). As one of the most popular projects in the robot-related open source community, there are already a large number of well-developed open source applications based on ROS, covering perception, planning, control, positioning, SLAM and mapping, visualization and almost all robot fields.

Figure 2.4: ROS equation: Plumbing + Tools + Capabilities + Ecosystem = ROS

2.3.1 Computation graph model of ROS

There are four major components behind ROS computation graph model:

- **Node:** A node represents one process running the ROS graph. Every node has a name, which it registers with the ROS master before it can take any other actions. Multiple nodes with different names can exist under different namespaces, or a node can be defined as anonymous, in which case it will randomly generate an additional identifier to add to its given name. Nodes are at the center of ROS programming, as most ROS client code is in the form of a ROS node which takes actions based on information received from other nodes, sends information to other nodes, or sends and receives requests for actions to and from other nodes.

- **Topic:** Topics are named buses over which nodes send and receive messages. Topic names must be unique within their namespace as well. To send messages to a topic, a node must publish to said topic, while to receive messages it must subscribe. The publish/subscribe model is anonymous: no node knows which nodes are sending or receiving on a topic, only that it is sending/receiving on that topic. The types of messages passed on a topic vary widely and can be user-defined. The content of these messages can be sensor data, motor control commands, state information, actuator commands, or anything else.
- **Service:** A node may also advertise services. A service represents an action that a node can take which will have a single result. As such, services are often used for actions which have a defined start and end, such as capturing a one-frame image, rather than processing velocity commands to a wheel motor or odometer data from a wheel encoder. Nodes advertise services and call services from one another.
- **Parameter server:** The parameter server is a database shared between nodes which allows for communal access to static or semi-static information. Data which does not change frequently and as such will be infrequently accessed, such as the distance between two fixed points in the environment, or the weight of the robot, are good candidates for storage in the parameter server.

ROS processes are represented as nodes in a graph structure, connected by edges called topics. ROS nodes can pass messages to one another through topics, make service calls to other nodes, provide a service for other nodes, or set or retrieve shared data from a communal database called the parameter server. A process called the ROS Master makes all of this possible by registering nodes to itself, setting up node-to-node communication for topics, and controlling parameter server updates. Messages and service calls do not pass through the master, rather the master sets up peer-to-peer communication between all node processes after they register themselves with the master. This decentralized architecture lends itself well to robots, which often consist of a subset of networked computer hardware, and may communicate with off-board computers for heavy computing or commands.

2.3.2 Improvements in ROS2

If we say ROS1 provides a good ecology for scientific research and prototype development, then ROS2 is the development architecture and corresponding toolchain for the deployment environment of actual products. Since ROS1 was initially developed for the research robot Willow Garage PR2, it is doomed that there were some shortcomings:

- No real-time characteristics;
- Not friendly to the implementations on embedded devices;
- High reliance on network, requires large bandwidth and stable connection;
- High flexibility brings non-standard programming patterns;

- Only single-agent application supported.

The application of ROS is not limited to academic researches any more. To satisfy the requirements of massive applications for certain performance (such as real-time, security, embedded porting, etc) in industrial fields, ROS2 adopts the following strategies to improve its applicability to production environments:

- Multiple-agent application supported;
- Implementations on embedded devices supported;
- Real-time system: real-time control supported, including real-time communication between processes and machines;
- Non-ideal network environments compliance: The system can still work in network environments such as low quality and high latency;
- Standardized programming model: to support build, develop, and deploy for large-scale ROS-based purposes.

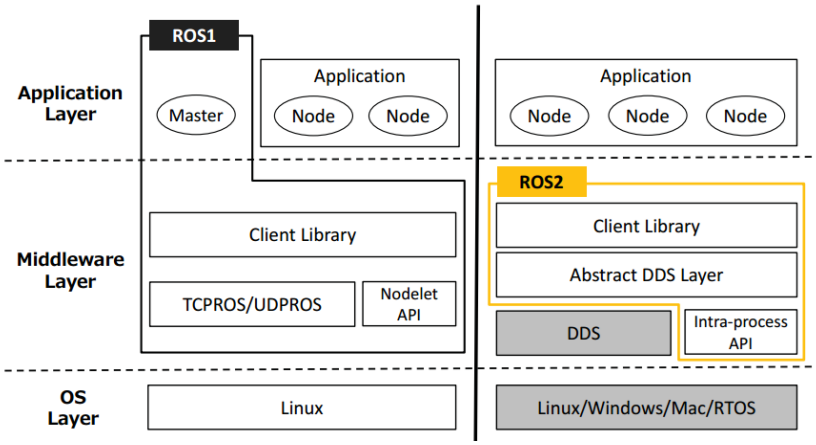


Figure 2.5: Comparison of ROS1 and ROS2

The highlights in ROS2 are Data-Distribution Service (DDS) and Quality of Service (QoS). In contrast to ROS1, whose core is an anonymous publish-subscribe communication intermediate layer based on the master central node, ROS2 uses DDS based on RTPS (Real-Time Publish-Subscribe) protocol (Figure 2.5). DDS is an industry standard for publish-subscribe communication of real-time and embedded systems. This point-to-point communication pattern is similar to the middle layer of ROS1, but DDS does not need to complete the communication between the two nodes through the master node like ROS1, which makes the system more fault-tolerant and flexible.

Quality of Service (QoS) is a collection of policies to configure communication between nodes. ROS1 only supports TCP-based communication, while in ROS2 users can achieve different communication behaviors by selecting corresponding QoS profiles. With this, ROS2 can demonstrate both the reliability of TCP and the high real-time performance of UDP.

2.4 Simulation environment modelling and rendering

2.4.1 Gazebo

Gazebo is an open-source 3D robotics simulator. It integrated the ODE physics engine, OpenGL rendering, and support code for sensor simulation and actuator control. Gazebo can use multiple high-performance physics engines, such as ODE, Bullet, etc. (the default is ODE). It provides realistic rendering of environments including high-quality lighting, shadows, and textures. It can model sensors that "see" the simulated environment, such as laser range finders, cameras (including wide-angle), Kinect style sensors, etc. For 3D rendering, Gazebo uses the OGRE engine.

As a stand-alone application, Gazebo can be used independently of ROS or ROS 2. The integration of Gazebo with either ROS version is done through a set of packages called "gazebo_ros_pkgs" (Fig.2.6). These packages provide a bridge between Gazebo's C++ API and transport system, and ROS 2 messages and services.

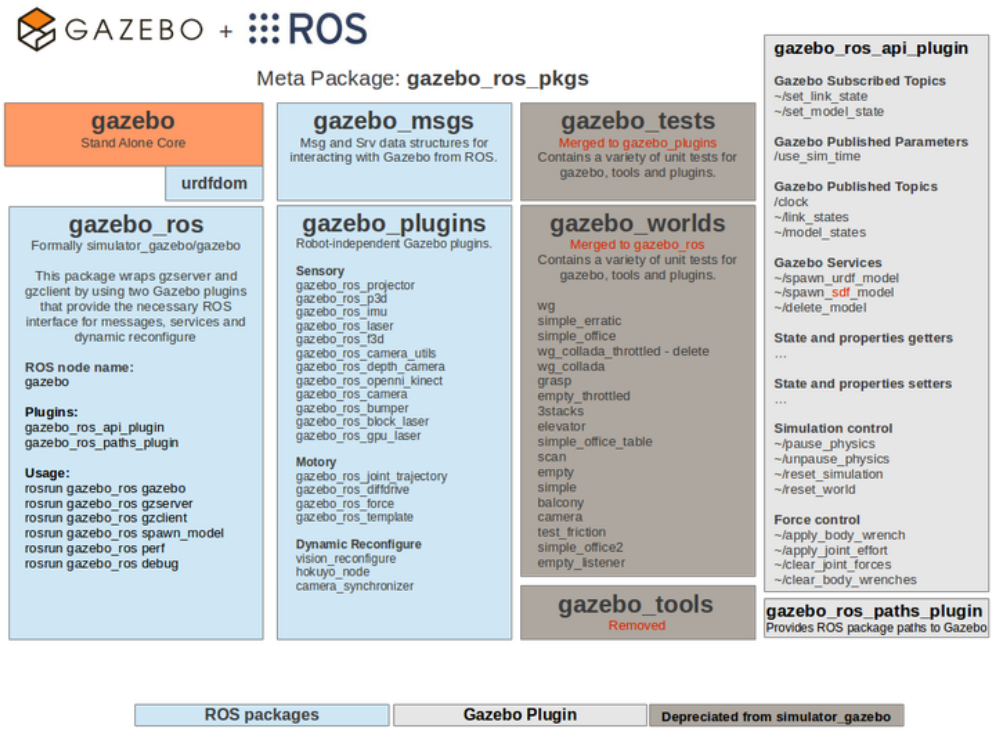


Figure 2.6: An overview of the gazebo_ros_pkgs interface.

2.4.2 RoadRunner

RoadRunner is an interactive editor that helps design 3D scenes for simulating and testing automated driving systems. The roadway scenes can be customized by creating region-specific road signs and markings. Besides, signs, signals, guardrails, and road damage, as well as foliage, buildings, and other 3D models can be inserted. RoadRunner provides tools for setting and configuring traffic signal timing, phases, and vehicle paths at intersections.

RoadRunner supports the visualization of lidar point cloud, aerial imagery, and GIS data. The road networks can be imported and exported using OpenDRIVE®. 3D scenes built with RoadRunner can be exported in FBX®, glTF™, OpenFlight, OpenSceneGraph, OBJ, and USD formats. The exported scenes can be used in automated driving simulators and game engines, including CARLA, Vires VTD, NVIDIA DRIVE Sim®, Baidu Apollo®, Cognata, Unity®, and Unreal® Engine.

Figure 2.7: The user interface of RoadRunner.

2.4.3 Unity

Unity is a cross-platform game engine developed by Unity Technologies. The engine can be used to create 3D and 2D games, as well as interactive simulations and other experiences. Unity offers a primary scripting API in C# using Mono, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality. Besides, creators can develop and sell user-generated assets to other game makers via the Unity Asset Store. This includes 3D and 2D assets and environments for developers to buy and sell. In the 2010s, Unity Technologies used its game engine to transition into other industries using the real-time 3D platform, including film and automotive.

2.5 The KITTI dataset

In this project, we use the KITTI dataset to develop ROS2-related software pipeline before our virtual environment setup in Unity is ready. KITTI dataset [5], developed by the team of Prof. Geiger from Karlsruhe Institute of Technology (KIT) and Toyota Technology Institute, is one of the most popular datasets for the use in mobile robotics and autonomous driving. KITTI dataset contains hours of traffic scenarios, such as the mid-size city of Karlsruhe, rural areas and highways, which was recorded with a standard station wagon equipped with variety of sensor modalities, including two pairs of high-resolution cameras, one RGB and the other grayscale, a 3D laser scanner as well as IMU and GPS modules.

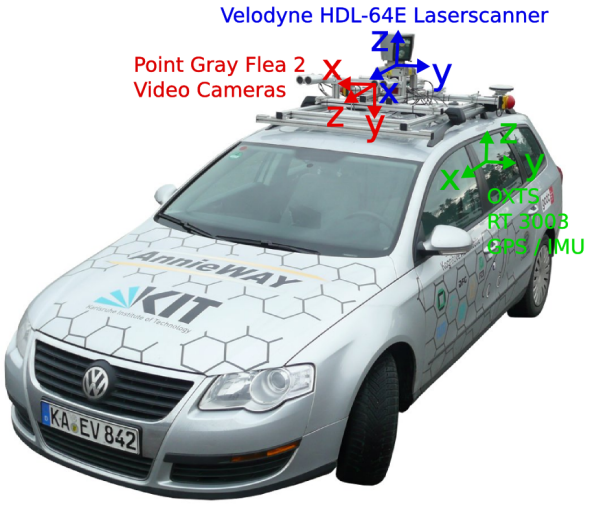


Figure 2.8: Fully equipped station wagon.

The recording platform is a Volkswagen Passat B6, which has been modified with actuators for both the gas and brake pedals and the steering wheel. The data is recorded using an eight-core i7 computer with RAID system, running Ubuntu Linux OS and a real-time database. Sensor setup is illustrated in the Figure 2.8.

2.6 YOLO: You Only Look Once

YOLO (You Only Look Once) [6] is a one-stage object detection algorithm introduced by Redmon et al. in the year 2016. The two-stage algorithms like R-CNN use region proposal methods to first generate potential bounding boxes in an image and then run a classifier on these proposed boxes, which causes high computational consumption and cannot satisfy the real-time capability. In comparison to this, YOLO reframes object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities.

YOLO uses only one convolutional neural network simultaneously predicts multiple bounding boxes and their class probabilities, trains on full images and directly optimizes

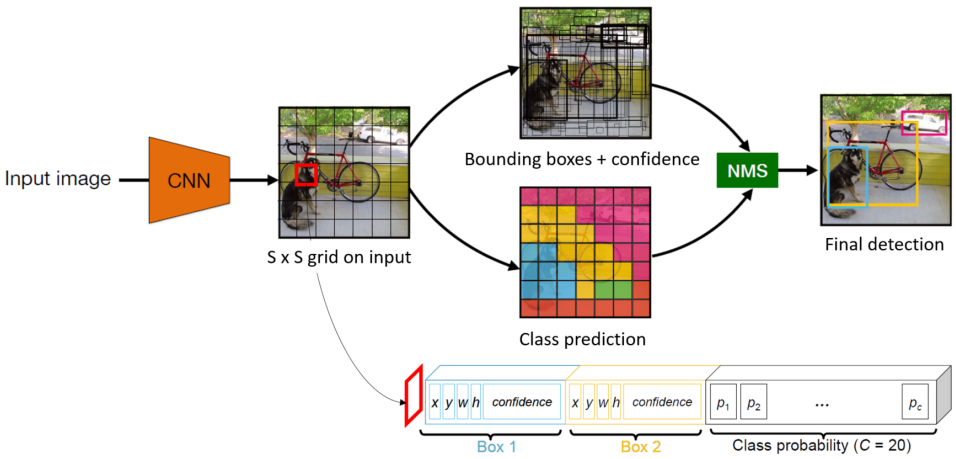


Figure 2.9: YOLO pipeline

detection performance. The unified model brings several benefits over traditional object detection algorithms:

- Fast and acceptable accurate detection: YOLO can process streaming video in real-time with less than 25 milliseconds of latency and achieves more than twice the mean average precision of other real-time systems;
- Reasons globally about the image: YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance;
- Learns generalizable representations of objects: YOLO performs well on different image styles. With the learned generative features it is less likely to break down when applied to new domains or unexpected inputs.

YOLO divides the input image into $S \times S$ grid and for each cell predicts B bounding boxes and confidence scores for those boxes, which indicates how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts. The confidence is defined formally as $\Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}}$, where IOU means intersection over union. Each bounding box consists 5 predictions: x, y, w, h as well as the confidence we just mentioned. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Each grid cell also predicts C conditional class probabilities $\Pr(\text{Class}_i|\text{Object})$, which are conditioned on the grid cell containing an object. For each grid cell only one set of class probabilities would be predicted, regardless of the number of boxes B. For the whole image all these predictions are encoded as a tensor of size $S \times S \times (B * 5 + C)$. At test time, the class-specific confidence score for each box would be obtained by multiplying the conditional probabilities and the individual box confidence predictions (Eq. 2.1). The final bounding box would be obtained via non maximum suppression (NMS), which recursively eliminates the candidate bounding

boxes with non maximal confidence scores but pretty high IOU respect to the box with the local maximal confidence.

$$\Pr(\text{Class}_i|\text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}} \tag{2.1}$$

2.7 Stereo depth estimation

With two cameras we can infer depth, by means of triangulation, if we are able to find corresponding (homologous) points in the two images [7]. Figure 2.10 shows a general overview of stereo vision system. The most important parts there are camera calibration and stereo correspondence matching. The intrinsic and extrinsic parameters of camera are obtained from offline camera calibration, which are required when rectifying the raw images and calculating depth from obtained disparity. And stereo correspondence matching aims to find the homologous pixel in target image of each pixel in reference image, and then to obtain the disparity map of the stereo pair.

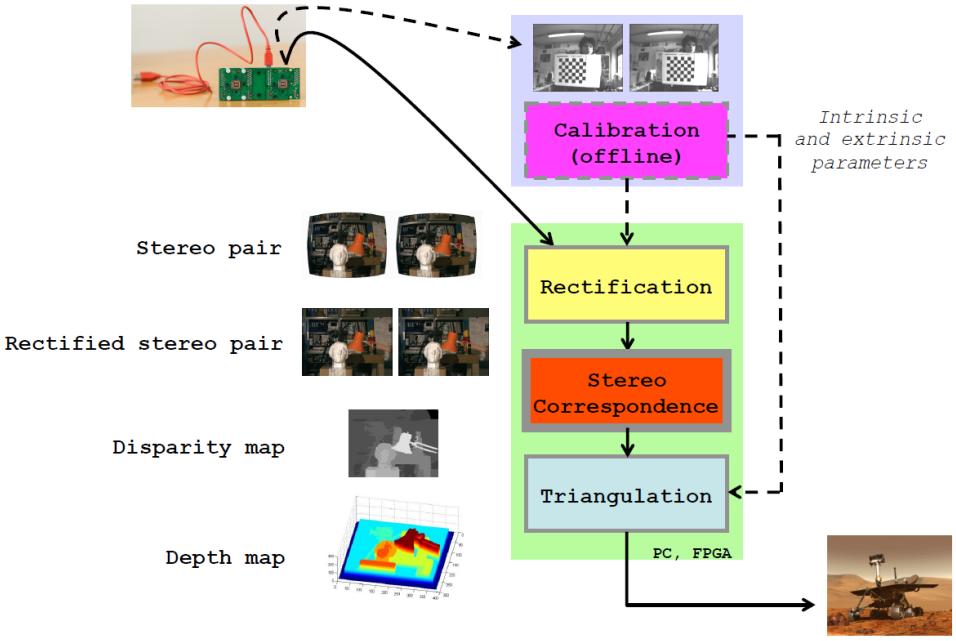


Figure 2.10: Overview of a stereo vision system.

Consider two points P and Q on the same line of sight of the reference image R, where both points project into the same image point $P \equiv Q$ on image plane π_R. The epipolar constraint states that the correspondence for a point belonging to the (red) line of sight lies on the green line on image plane π_T of target image T (Fig. 2.11). Once we know that the search space for corresponding points can be narrowed from 2D to 1D, we can put (virtually) the stereo rig in a more convenient configuration, i.e. the standard form (the light yellow image planes in Fig. 2.11), where corresponding points are constrained on the same image scanline. With

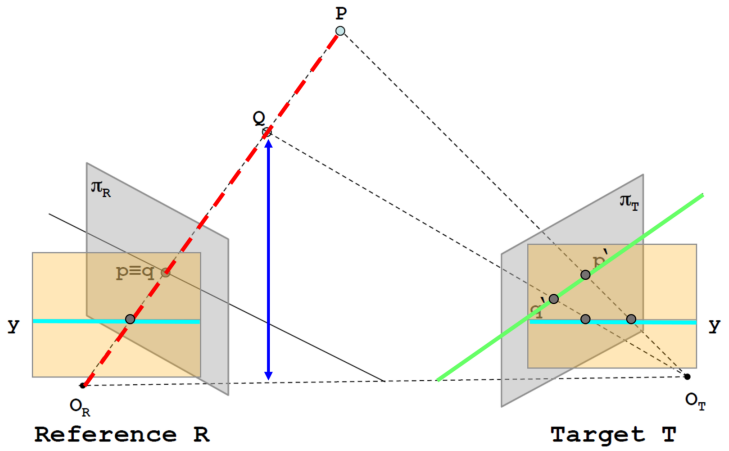
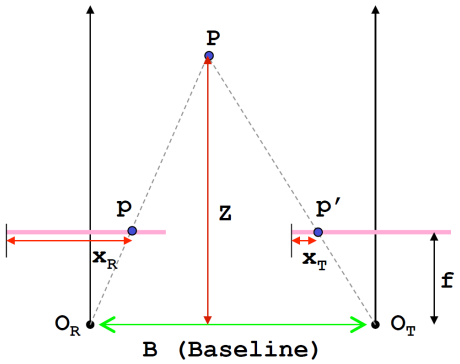


Figure 2.11: Epipolar constraint.

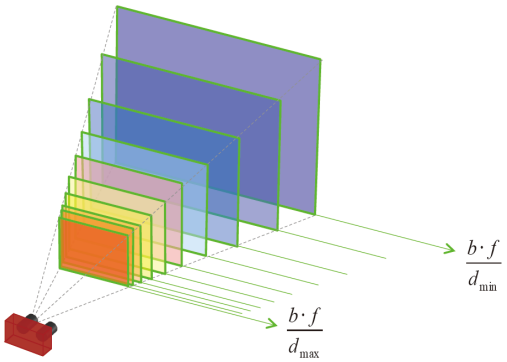
the stereo rig in standard form and by considering similar triangles (PO_RO_T and Ppp' in Fig 2.12a):

$$\frac{b}{Z} = \frac{(b + x_T) - x_R}{Z - f} \implies Z = \frac{b\dot{f}}{x_R - x_T} = \frac{b\dot{f}}{d} \tag{2.2}$$

where $d = x_R - x_T$ is the disparity and Z the depth. The disparity is the difference between the x coordinate of two corresponding points. The closer the point is to the camera, the higher is the disparity value. Therefore, given a stereo rig with baseline b and focal length f, the range field of the system is constrained by the disparity range $[d_{min}, d_{max}]$. As shown in Fig 2.12b, with discrete pixel level disparity values the depth measured by a stereo vision system is discretized into parallel planes (one for each disparity value), which shows the necessity of sub-pixel interpolation, which would be discussed later in subsection 3.3.4.



(a) The relationship between depth and disparity



(b) Discretized depth plans

Figure 2.12: Depth calculation.

3 Implementation and visualization

This chapter gives an overview about my contribution to this project. The Unity-related work is not counted in respect of this thesis. In order to develop and test algorithm integration in ROS2, the raw image and point cloud from KITTI dataset are taken as substitution of those that are supposed to be published from Unity. A summary of work needed to be done is as follows:

(1) Publish KITTI raw data in ROS2 and visualize them in Rviz. Prepare to test the algorithm implementations.

(2) Implement object detection algorithm (YOLOv5) on ROS2.

(3) Implement stereo depth estimation (SGBM algorithm) for the detected objects.

(4) Inspect estimated depths by calculating ground truth distances using labeled point cloud.

3.1 ROS node layout

As shown in Figure 3.1, the "/kitti_node" loads raw KITTI image and point cloud data from local, transfers them into legal ROS message format respectively and finally publishes them. It also publishes a "/kitti_frame" topic to broadcast the current frame number to other active nodes; The "/detect_node" instantiates an YOLOv5 detector, subscribes the image from left RGB camera, implements object detection and publishes the result label information as a custom ROS message YOLOlabels under the topic "/detect_labels"; The "/stereo_node" subscribes images from left and right grey cameras to compute disparity map using SGBM algorithm and then obtain depth map with the camera parameter. Besides, the "/stereo_node" also subscribes "/detect_labels" from "/detect_node" to get the center position of each detected object. Take the depth value at object center pixel as the object's depth. Finally, the "/stereo_node" publishes a colored disparity map and a labeled left RGB image, indicating object classes and distances (Fig.3.2). In addition, to validate the accuracy of estimated depth, the ground truth distances are calculated with the 3D bounding boxes from KITTI tracking data.

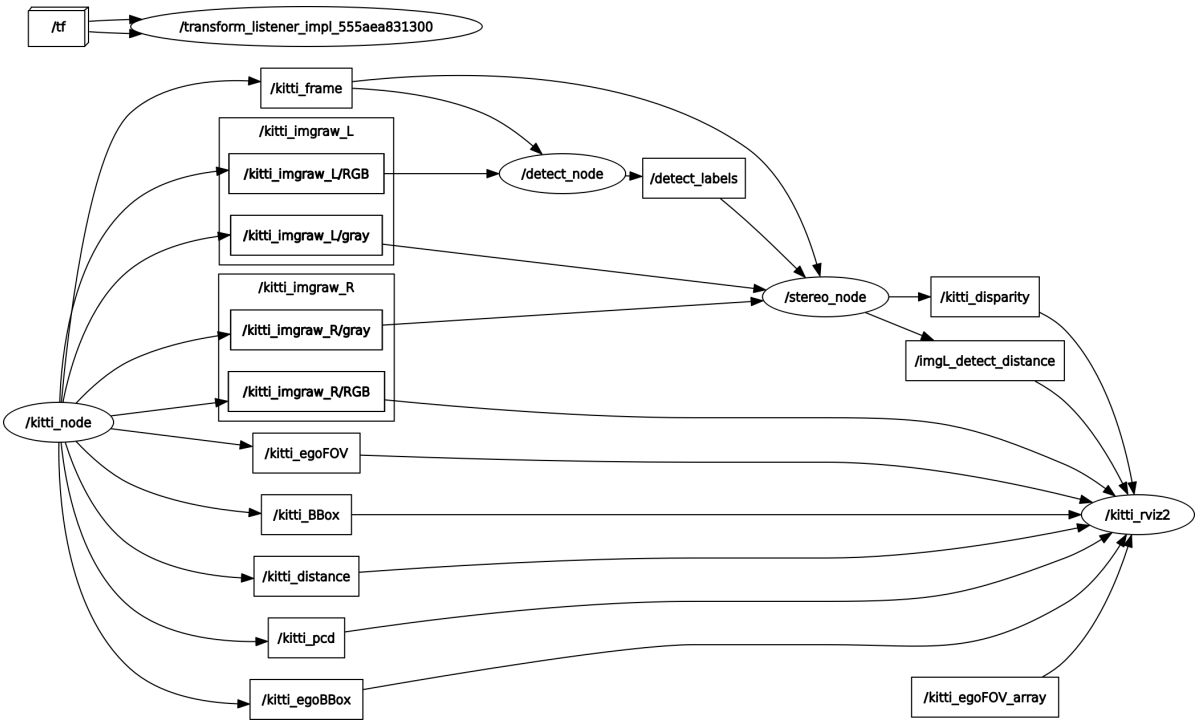


Figure 3.1: rqt graph

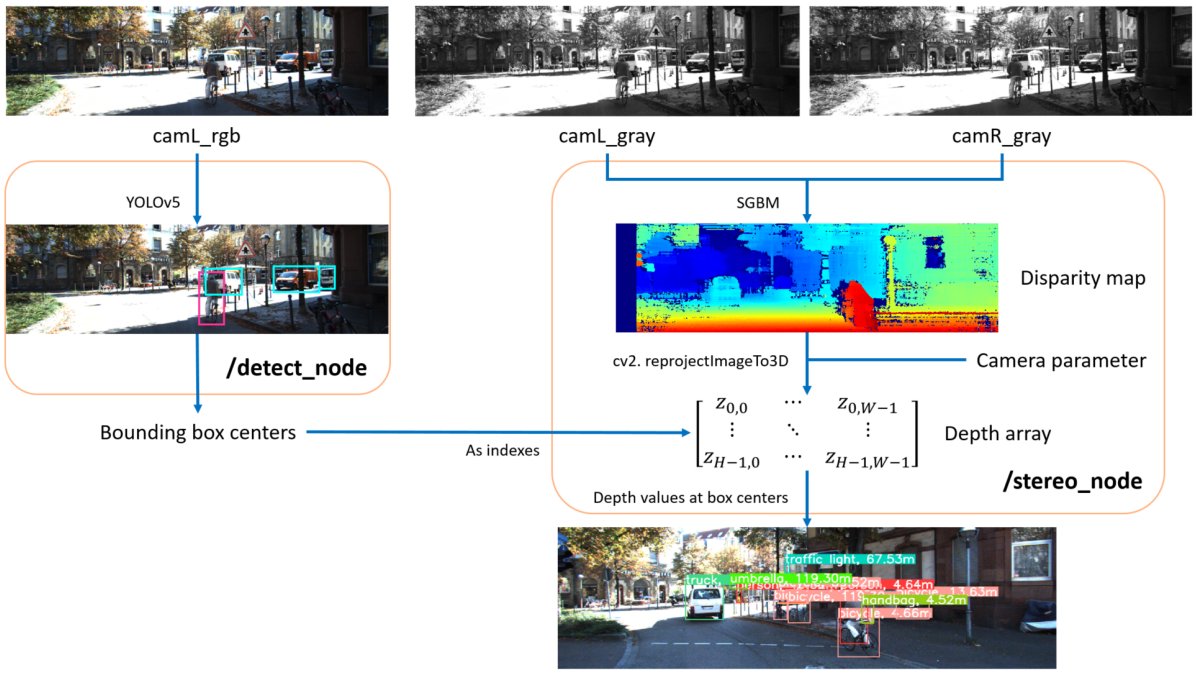


Figure 3.2: Schematic ROS program concept

3.2 Object detection: YOLOv5

YOLOv5 is a model in the YOLO family of object detection models. It shares similar architecture as YOLOv3 and YOLOv4 [8], which was based on Darknet framework. As shown in Fig.3.3, the YOLO network consists of three main procedures:

(1) Backbone: A convolutional neural network that aggregates and forms image features at different granularities;

(2) Neck: A series of layers to mix and combine image features to pass them forward to prediction;

(3) Head: Consumes features from the neck and takes box and class prediction steps.

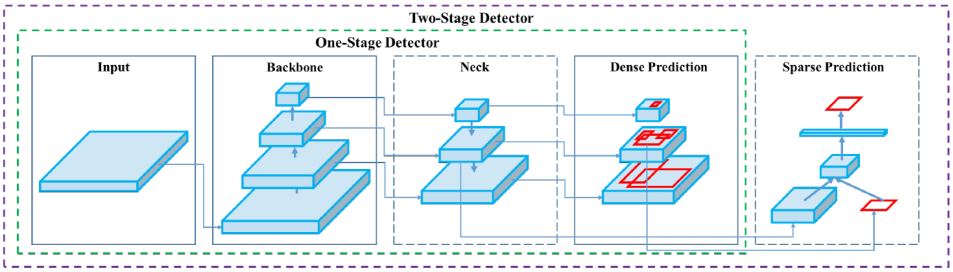


Figure 3.3: Object detector structure.

The largest contribution of YOLOv5 is to translate the Darknet research framework to the PyTorch framework. The Darknet framework is written primarily in C and offers fine grained control over the operations encoded into the network. In many ways the control of the lower level language is a boon to research, but it can make it slower to port in new research insights, as one writes custom gradient calculations with each new addition.

One of the novel specialties of YOLOv5 is mosaic data augmentation, which combines four images into four tiles of random ratio. Mosaic augmentation is especially useful for the popular COCO object detection benchmark, helping the model learn to address the well known "small object problem" - where small objects are not as accurately detected as larger objects.

Another specialty of YOLOv5 isauto learning bounding box anchors. The idea of learning anchor boxes based on the distribution of bounding boxes in the custom dataset with K-means and genetic learning algorithms was first introduced in YOLOv3. This is very important for custom tasks, because the distribution of bounding box sizes and locations may be dramatically different than the preset bounding box anchors in the COCO dataset. In order to make box predictions, the YOLOv5 network predicts bounding boxes as deviations from a list of anchor box dimensions. The most extreme difference in anchor boxes may occur if we are trying to detect something like giraffes that are very tall and skinny or manta rays that are very wide and flat. All YOLO anchor boxes are auto-learned in YOLOv5 with the given custom data.

3.2.1 Train a YOLOv5 object detector for KITTI dataset

YOLOv5 comes in four main versions: small (s), medium (m), large (l), and extra large (x), each offering progressively higher accuracy rates and taking a different amount of time to train (Fig.3.4). In this project we used YOLOv5s to test our pipeline. The model structure is configured in a yaml file.

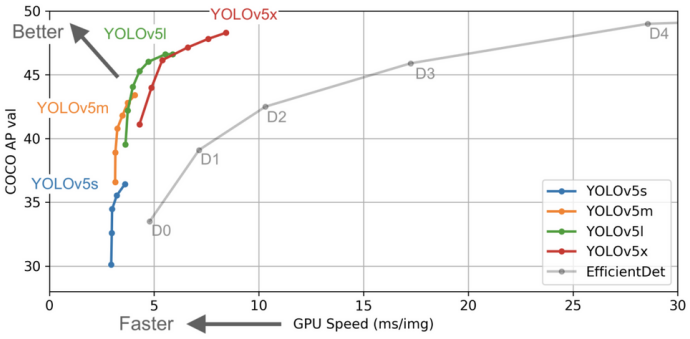


Figure 3.4: Performance comparison of YOLOv5 model vatiants.

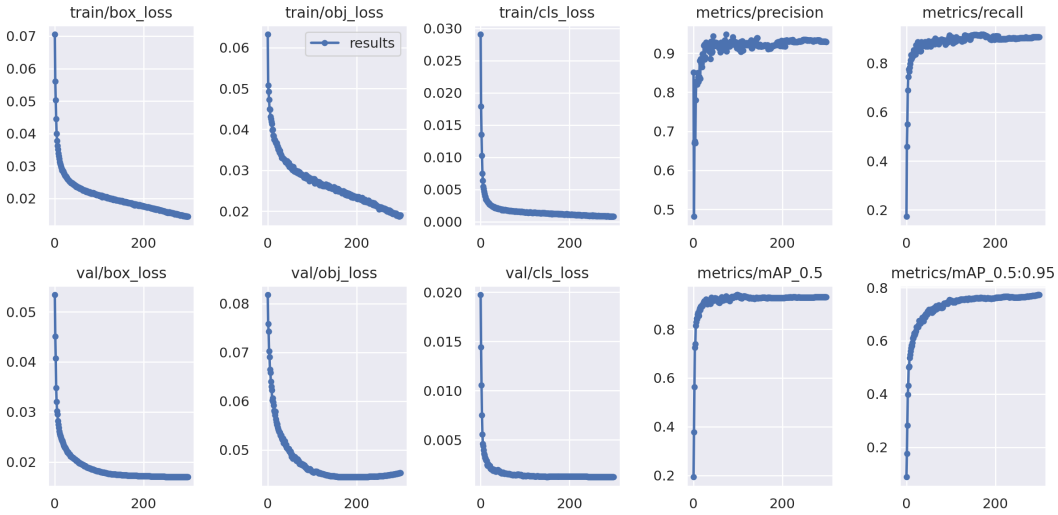
To match with the labeled ground truth provided by KITTI, the number of class is set as 8, including car, van, truck, pedestrian, person sitting, cyclist, tram and the classes we don't care as misc, which are merged into 4 classes: vehicle, pedestrian, cyclist and misc later when drawing labels. The train result is shown in Fig 3.5.

3.2.2 Custom ROS message YOLOlabels.msg

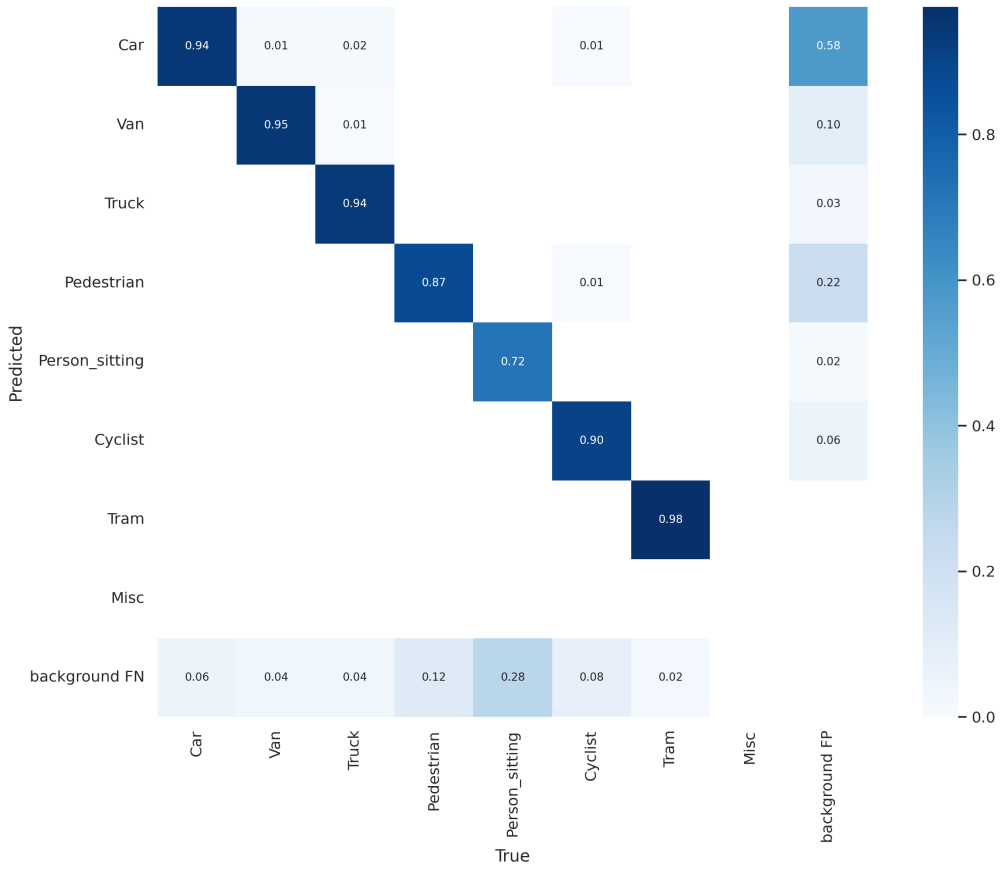
Object detection and depth estimation are implemented in two nodes separately. As mentioned before, we take the depth at bounding box center as the object's distance relative to image plane. To achieve this, the detected bounding boxes should be published from detection node to the stereo node via a custom ROS message. Message is one of the ROS interfaces, which act as a crucial component for the communication of ROS applications. ROS 2 uses a simplified description language, the interface definition language (IDL), to describe these interfaces. This description makes it easy for ROS tools to automatically generate source code for the interface type in several target languages. The ".msg" files describes the fields of a ROS message, declaring the data type and name of each attribute. The definition of YOLOlabels.msg is shown in Tab. 3.1. Each attribute is an array, since there are usually more than one detected objects in every single frame. The attributes of one object share the same index.

Table 3.1: Content of YOLOlabels.msg.

Name	x	y	w	h	conf	cls
Data type	int32[ ]	int32[ ]	int32[ ]	int32[ ]	float32[ ]	int32[ ]



(a) YOLOv5 train result



(b) YOLOv5 confusion matrix

Figure 3.5: YOLOv5 training result on KITTI dataset.

3.3 Stereo depth estimation: SGBM

Semi-Global Block Matching (SGBM), is OpenCV's implementation of Hirschmüller's original Semi-Global Matching (SGM) algorithm [9] [10], which calculates disparity map from a pair of rectified stereo images. The original SGM uses pixel-wise aggregation cost, while SGBM allows matching blocks. If the block size is set to 1, it's the same as working on pixels. So the "Block" in SGBM actually means calculating each cost value in blocks (domain summation operation) for disparity optimization. The SGBM requires rectified images, which are already privided in KITTI dataset. Generally, the semi-global matching algorithm consist of following 4 procedures:

(1) Matching cost computation;

(2) Cost aggregation;

(3) Disparity computation;

(4) Disparity refinement.

3.3.1 Matching cost computation

Matching cost is actually the pixel-based absolute difference between pixel intensities I. The correlation between the pixel to be matched and the candidate pixels can estimate their probability of being the homologous point.

The disparity search slope $D = D_{max} - D_{min}$ is defined before homologous point searching. During disparity searching, each single pixel would get a vector of size D, containing the matching costs at this pixel under every disparity value in slope D. Therefore, for the whole image of size $W \times H$ we will get a three dimensional matrix of size $W \times H \times D$, which is called disparity space image (DSI). Each element $C(x, y, d)$ of the DSI matrix represents the cost of the correspondence between the intensity at pixel (x, y) in reference and the intensity at pixel (x+d, y) in target, i.e. $I_R(x_R, y)$ and $I_T(x_R + d, y)$, which can indicate the likelihood and/or confidence of this correspondence.

3.3.2 Cost aggregation

Improve the accuracy that matching cost indicates pixel correlation. Since only considering local information, pixelwise cost calculation is generally ambiguous and wrong matches can easily have a lower cost than correct ones, due to noise or when the pixel is in a weak texture or repeated texture area.

Cost aggregation aims to establish the relationship between neighboring pixels with a certain criteria, such as neighboring pixels should have a continuous disparity value, to optimize the cost matrix. This optimization is generally global. The cost value of a single pixel under a certain disparity will be recalculated according to that of its neighboring pixels under the same or some nearby disparity value to obtain a new DSI matrix, **S**.

For SGM, the problem of stereo matching can be formulated as finding the disparity image D that minimizes the energy $E(D)$ (Eq.3.1), which indicates the pixelwise cost and the smoothness constraints.

$$E(D) = \sum_{\mathbf{p}} C(\mathbf{p}, D_{\mathbf{p}}) + \sum_{\mathbf{q} \in N_{\mathbf{p}}} P_1 \mathrm{T}[|D_{\mathbf{p}} - D_{\mathbf{q}}| = 1] + \sum_{\mathbf{q} \in N_{\mathbf{p}}} P_2 \mathrm{T}[|D_{\mathbf{p}} - D_{\mathbf{q}}| > 1] \tag{3.1}$$

The energy $E(D)$ supports smoothness by penalizing changes of neighboring disparities, where P1 and P2 are the penalty factors regarding to two different situations of neighboring disparity changes. The first term is the sum of all pixel matching costs for the disparities of D. The second term adds a constant penalty P_1 for all pixels $\mathbf{q}$ in the neighborhood $N_{\mathbf{p}}$ of $\mathbf{p}$, for which the disparity changes a little bit (i.e. 1 pixel). The third term adds a lager constant penalty P_2, for all lager disparity changes. Using a lower penalty for small changes permits an adaptation to slanted or curved surfaces. The constant penalty for all larger changes (i.e. independent of their size) preserves discontinuities.

3.3.3 Disparity computation

The optimal disparity value of each pixel is determined through the updated DSI $\mathbf{S}$ after cost aggregation, generally with the Winner-Takes-All (WTA) algorithm (Eq3.2). Among all the disparity values at each pixel, take the one that has the lowest cost as the optimal disparity value at this pixel. This step is very simple, where we totally trust the cost aggregation in the last step. That means the determination of P1 and P2 influences the accuracy of disparity computation directly.

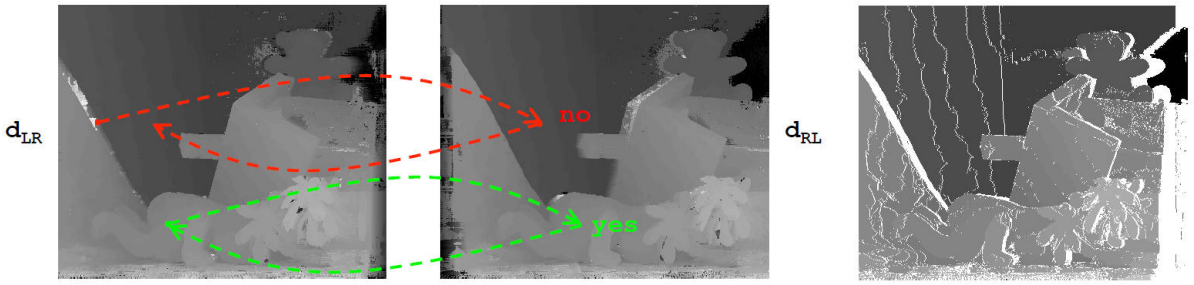
$$d(x, y) = \underset{d}{\operatorname{argmin}} |I_R(x, y) - I_T(x + d, y)| \tag{3.2}$$

3.3.4 Disparity optimization

The obtained disparity map can be optimized by eliminating false matches, removing small connected regions, appropriate smoothing and sub-pixel interpolation, etc. Intuitively, false match means the aggregated cost value of the true disparity value at some pixels are not the minimum. This can be caused by image noise, occlusion, weak or repetitive textures, and the limitations of the algorithm. In fact, there is no algorithm that can perfectly handle all of the above problems so far, so the elimination of false matches is necessary for all algorithms.

Considering there exists only one true disparity value at each pixel, i.e. the disparity uniqueness constraint, the Left-Right Check (or Bidirectional Matching) algorithm generates the disparity map twice: one assuming left image as reference getting d_{LR} and the other right as reference getting d_{RL} (Fig.3.6a). Only the disparity values that have acceptable difference at the homologous pixels in the two maps would be considered as consistent and be kept, while those that don't satisfy the expression 3.3 are considered as outliers, where the threshold T is typically set to 1.

$$|d_{LR}(x, y) - d_{RL}[x + d_{LR}(x, y), y]| < T \tag{3.3}$$



(a) Left-Right Check

(b) The detected outliers are encoded in white

Figure 3.6: The Left-Right Check algorithm.

Since the disparity maps are typically computed at discrete pixel level [11], SGM takes the quadratic interpolation method to obtain sub-pixel accuracy. After fitting the optimal disparity's and its two neighboring disparities' cost values with quadratic curve, take the disparity value at the minimum of fitting curve as new sub-pixel disparity (Fig.3.7).

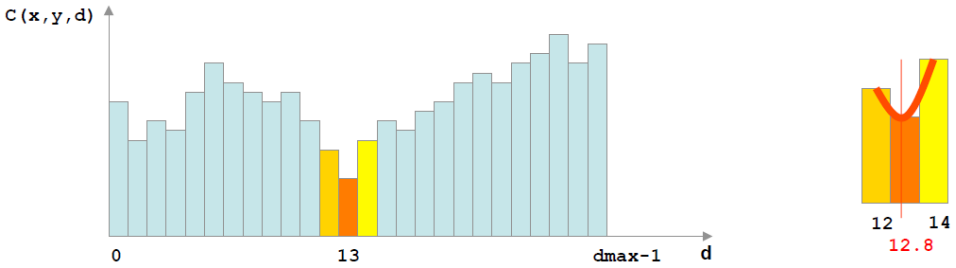


Figure 3.7: Quadratic sub-pixel interpolation

3.3.5 StereoSGBM parameter configuration

The StereoSGBM class in OpenCV contains the following parameters:

- **minDisparity**: Minimum possible disparity value. Normally, it is zero but sometimes rectification algorithms can shift images, so this parameter needs to be adjusted accordingly.
- **numDisparities**: Maximum disparity minus minimum disparity. The value is always greater than zero. In the current implementation, this parameter must be divisible by 16.
- **blockSize**: Matched block size. It must be an odd number >=1 . Normally, it should be somewhere in the 3...11 range.

- **P1** and **P2**: Two parameters that control the disparity smoothness. The larger the values are, the smoother the disparity is. P1 is the penalty on the disparity change by plus or minus 1 between neighbor pixels, considering the sloping or curved continuous surfaces. P2 is the penalty on the disparity change by more than 1, considering the edges of objects. The algorithm requires P2 > P1. Usually

$$P1 = 8 \times num_{Channels} \times blocksize \times blocksize \tag{3.4}$$

$$P2 = 32 \times num_{Channels} \times blocksize \times blocksize \tag{3.5}$$

- **disp12MaxDiff**: Maximum allowed difference (in integer pixel units) in the left-right disparity check. Set it to a non-positive value to disable the check.

- **preFilterCap**: Truncation value for the prefiltered image pixels. The algorithm first computes x-derivative at each pixel and clips its value by [-preFilterCap, preFilterCap] interval. The result values are passed to the Birchfield-Tomasi pixel cost function.

- **uniquenessRatio**: the uniqueness detection parameters. For the matching pixels on the left image, the lowest cost defined in the "numberOfDisparities" search interval is the minimum cost (mincost), and the next lowest cost is second minimum cost (secdmincost). If Eq.3.3.5 is satisfied, the difference between the lowest cost and the second cost is too small, that is, the matching degree is not distinguished enough, the current matching pixel is considered to be mismatched. Normally, a value within the 5...15 range is good enough.

$$\frac{secdmincost}{mincost} < \frac{100}{100 - uniquenessRatio} \tag{3.6}$$

- **speckleWindowSize**: Maximum size of smooth disparity regions to consider their noise speckles and invalidate. Set it to 0 to disable speckle filtering. Otherwise, set it somewhere in the 50...200 range.

- **speckleRange**: Maximum disparity variation within each connected component. This parameter can be regarded as the disparity connectivity condition. Calculating the connected region of a disparity point, when the absolute value of the disparity change of the next pixel is greater than speckleRange, the next and the current disparity pixels are considered as disconnected. If you do speckle filtering, set the parameter to a positive value, it will be implicitly multiplied by 16. Normally, 1 or 2 is good enough.

- **mode**: Set it to cv2.STEREO_SGBM_MODE_HH to run the full-scale two-pass dynamic programming algorithm. It will consume $\mathcal{O}(W \times H \times numDisparities)$ bytes, which is already large for 640 $\times$ 480 stereo and huge for HD-size pictures. By default, it is set to false.

The final StereoSGBM parameter configuration is shown in Table 3.2. Some heuristics about SGBM parameter tuning are referred from [12].

Table 3.2: SGBM parameter settings.

Parameter	Set value
minDisparity	5
numDisparities	4×16
blockSize	3
P1	$8 \times 3 \times 4 \times 4$
P2	$32 \times 3 \times 4 \times 4$
disp12MaxDiff	1
preFilterCap	-1
uniquenessRatio	10
speckleWindowSize	10
speckleRange	20
mode	cv2.STEREO_SGBM_MODE_SGBM_3WAY

3.3.6 Obtain depth map from disparity map

Now with the disparity map we can compute the depth map according to the triangular relationships shown in Fig.2.12a. Instead of calculating the depth values pixelwisely with Eq.2.2, which causes a huge memory consumption, OpenCV provides a function "reprojectImageTo3D" handling this task more efficiently via re-projecting the disparity map to 3D space (Fig.3.8).

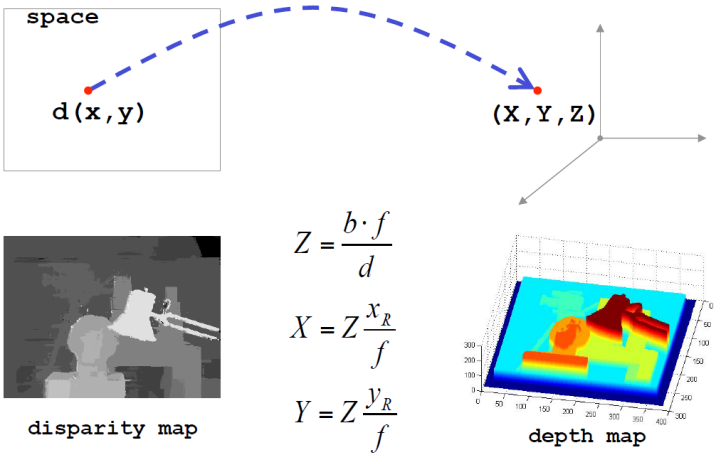


Figure 3.8: Reproject disparity map to 3D space.

In "reprojectImageTo3D", the camera parameters are represented as a perspective transformation matrix **Q**, which can be inferred from OpenCV function "stereoRectify" as Eq. 3.3.6 for horizontal stereo pairs.

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & -c \cdot x_1 \\ 0 & 1 & 0 & -c \cdot y \\ 0 & 0 & 0 & f \\ 0 & 0 & -\frac{1}{T_x} & \frac{c \cdot x_1 - c \cdot x_2}{T_x} \end{bmatrix} \tag{3.7}$$

The calibration file "calib_cam_to_cam.txt" in KITTI dataset contains the original camera parameters we need, which are processed to be the input variables for the OpenCV function "stereoRetify" according to KITTI setup layout shown in Fig.3.9. The input parameter for "stereoRectify" is shown in Table 3.3:

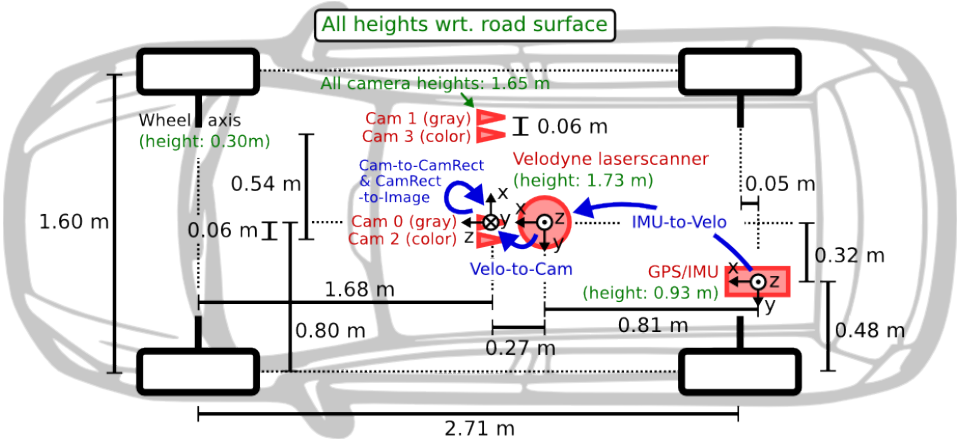


Figure 3.9: Schematic KITTI setup.

Table 3.3: Camera calibrations.

Parameter	Value
cameraMatrix1	$\begin{bmatrix} 9.037596e2 & 0 & 6.957519e2 \\ 0 & 9.019653e2 & 2.242509e2 \\ 0 & 0 & 1 \end{bmatrix}$ Right gray cam_01 matrix
distCoeffs1	$\begin{bmatrix} -3.639558e-1 \\ 1.788651e-1 \\ 6.029694e-4 \\ -3.922424e-4 \\ -5.382460e-2 \end{bmatrix}^T$ Right gray cam_01 distortion coefficients
cameraMatrix2	$\begin{bmatrix} 9.597910e2 & 0 & 6.960217e2 \\ 0 & 9.569251e2 & 2.241806e2 \\ 0 & 0 & 1 \end{bmatrix}$ Left gray cam_00 matrix
distCoeffs2	$\begin{bmatrix} -3.691481e-1 \\ 1.968681e-1 \\ 1.353473e-3 \\ 5.677587e-4 \\ -6.770705e-2 \end{bmatrix}^T$ Left gray cam_00 distortion coefficients
R	$\begin{bmatrix} 0.9996 & 0.0223 & -0.0198 \\ -0.0222 & 0.0998 & 0.0017 \\ 0.0198 & -0.0013 & 0.9998 \end{bmatrix}$ Rotation matrix of cam_03 relative to cam_02, i.e. SE_{23}
T	$\begin{bmatrix} -0.5327 & 0 & -0.0054 \end{bmatrix}$ Translation vector of cam_03 relative to cam_02

3.4 Ground truth distance computation

In this step we calculate the distance between the 3D bounding boxes of ego car to each labeled object frame by frame. For simplification, only the xy-plane projection of the 3D bounding boxes are considered, that means, we calculate the minimum distance between rectangles. The problem can be broken down into calculating the distances between points and line segments. There are three cases for projecting the point to a line segment, : the projection point is within the line segment (Fig.3.10a), exactly at one of the endpoints (Fig.3.10b) or outside of the segment (Fig.3.10c). For the first two cases, the distance is the length of the perpendicular segment, while for the third case, the distance is the length of the connection from the point to the nearer endpoint.

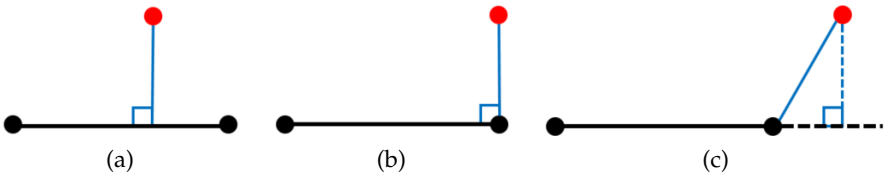


Figure 3.10: Calculate distance between point and line segment.

With this, we calculate the distance from each vertex of ego's rectangle to the four edges of the object's separately (Fig.3.11a), and then calculate the distance from each vertex of the object's rectangle to the four edges of the ego's in turn (Fig.3.11b). In these total 32 distances we take the minimum as the distance between two rectangles.

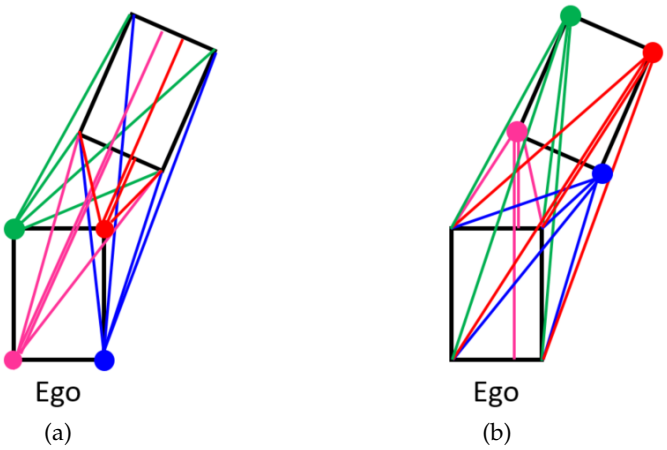


Figure 3.11: Calculate distance between rectangles.

3.5 Visualization in Rviz

The labeled images, disparity map, point cloud and 3D bounding boxes would be visualized in Rviz frame by frame. Rviz, short for "ROS visualization", is a 3D visualization software tool for robots, sensors, and algorithms. It can visualize the active ROS topic of the supported message types. The visualized ROS topics are listed in Table 3.4:

Table 3.4: Visualized ROS messages.

Topic	ROS message type	Content
/kitti_imgraw_L/RGB	sensor_msgs.Image	RGB image from cam 2
/kitti_imgraw_L/gray	sensor_msgs.Image	Gray image from cam 0
/kitti_imgraw_R/RGB	sensor_msgs.Image	RGB image from cam 3
/kitti_imgraw_R/gray	sensor_msgs.Image	Gray image from cam 1
/kitti_disparity	sensor_msgs.Image	Disparity map
/kitti_egoFOV	visualization_msgs.Marker	Ego car's Feld of View
/kitti_egoBBox	visualization_msgs.MarkerArray	Ego car's bounding box
/kitti_pcl	sensor_msgs.PointCloud2	Point cloud
/kitti_BBox	visualization_msgs.MarkerArray	Objects' bounding boxes
/kitti_distance	visualization_msgs.MarkerArray	Distance lines with text

The 3D bounding box labels of KITTI tracking data are given in form of height, width, length, center position (x, y, z) and rotation angle of y-axis, where x, y and z are horizontal, vertical and depth directions in respect of the image plane. With these data the coordinates of eight vertices of each 3D bounding box can be calculated. The order of vertex points and connection lines are shown in Fig.3.12, where the surface with cross lines indicates the object's front surface, which shows the orientations of the labeled objects. The result visualization is shown in Fig.3.13.

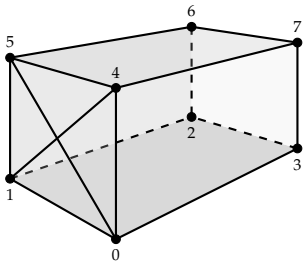


Figure 3.12: Point connections of KITTI 3D BBox labels.

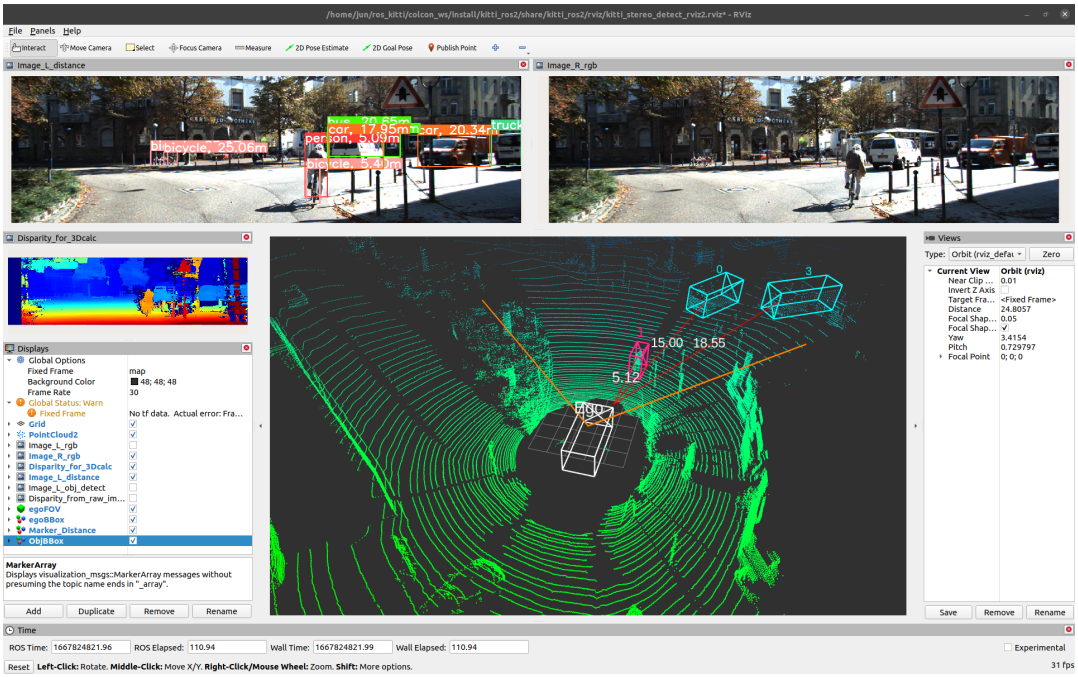


Figure 3.13: Visualization result in Rviz.

4 Conclusion and discussion

4.1 Conclusion

In this project I have successfully developed the ROS-based object detection and stereo depth estimation applications for our autonomous driving simulation platform as well as the visualization of 3D scenario presents including point cloud and 3D bounding box, which prepared us for the following joint development with Unity. The current frame-per-second (FPS) processing 1242 $\times$ 375 pixel KITTI images comes up to 30, which satisfies the real-time requirement. The error of depth estimation for objects are within 5% relative to the true distance.

4.2 Discussion

Due to time constraints there are some experiments and/or potential improvement which were not implemented:

- Apply GPU boost for SGBM algorithm to improve the performance: Currently the SGBM algorithm for depth estimation is operated on CPU, which cannot satisfy both high accuracy and real-time. Since "cv2.STEREO_SGBM_MODE_SGBM_3WAY" has less computation complexity and already accurate enough for our current application, we used this instead of the full-scale two-pass mode "cv2.STEREO_SGBM_MODE_HH", which has higher memory consumption. In our further development the HH mode might be necessary and thus GPU should solve its latency problem.
- Achieve stereo depth estimation using deep learning methods [13] [14] [15]: There are already some works that involve deep learning to stereo depth estimation, especially to the stage stereo matching, such as cycle GAN [16], HITNet [17], AnyNet [18], PSMNet [19], etc. Take cycle GAN as an example, similar to image style transformation task, stereo depth estimation also has a process of pixel matching. The correspondence field (i.e. the disparity map) between two image views in a calibrated stereo camera setting can be predicted by a deep generative network, which consists of two generative subnetworks jointly trained with adversarial learning for reconstructing the disparity map and organized in a cycle such as to provide mutual constraints and supervision to each other.
- Apply more YOLOv5 model variants to test the performance of object detection: In this project I only implemented the most light-weighted variant YOLOv5s to test our pipeline. The compatibility for more complicated models was not experimented.

List of Figures

List of Tables

Bibliography

[1] F. Roehrbein, M.-O. Gewaltig, C. Laschi, G. Klinker, P. Levi, and A. Knoll. "The Neurorobotic Platform: A simulation environment for brain-inspired robotics". In: *Proceedings of ISR 2016: 47st International Symposium on Robotics*. 2016, pp. 1–6.

[2] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis. "A Self-Driving Car Architecture in ROS2". In: *2020 International SAUPEC/RobMech/PRASA Conference*. 2020, pp. 1–6.

[3] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[4] V. M. Raju, V. Gupta, and S. Lomate. "Performance of open autonomous vehicle platforms: Autoware and Apollo". In: *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*. IEEE. 2019, pp. 1–5.

[5] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. "Vision meets robotics: The kitti dataset". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1231–1237.

[6] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[7] S. Degadwala, D. Vyas, and A. Mahajan. "Review on Stereo Vision Based Depth Estimation". In: *International Journal of Scientific Research in Science, Engineering and Technology* (Mar. 2020), pp. 665–671. DOI: 10.32628/IJSRSET207261.

[8] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. "Yolov4: Optimal speed and accuracy of object detection". In: *arXiv preprint arXiv:2004.10934* (2020).

[9] H. Hirschmuller. "Accurate and efficient stereo processing by semi-global matching and mutual information". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. 2005, 807–814 vol. 2.

[10] H. Hirschmuller. "Stereo processing by semiglobal matching and mutual information". In: *IEEE Transactions on pattern analysis and machine intelligence* 30.2 (2007), pp. 328–341.

[11] S. Birchfield and C. Tomasi. "Depth discontinuities by pixel-to-pixel stereo". In: *International Journal of Computer Vision* 35.3 (1999), pp. 269–293.

[12] P. H. Nguyen and C. W. Ahn. "Parameter selection framework for stereo correspondence". In: *Machine Vision and Applications* 31.4 (2020), pp. 1–15.

[13] H. Laga, L. V. Jospin, F. Boussaid, and M. Bennamoun. "A Survey on Deep Learning Techniques for Stereo-Based Depth Estimation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.4 (2022), pp. 1738–1764.

[14] J. Zbontar, Y. LeCun, et al. "Stereo matching by training a convolutional neural network to compare image patches." In: *J. Mach. Learn. Res.* 17.1 (2016), pp. 2287–2318.

[15] J. Choe, K. Joo, F. Rameau, and I. S. Kweon. "Stereo object matching network". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 12918–12924.

[16] A. Pilzer, D. Xu, M. Puscas, E. Ricci, and N. Sebe. "Unsupervised Adversarial Depth Estimation Using Cycled Generative Networks". In: *2018 International Conference on 3D Vision (3DV)*. 2018, pp. 587–595.

[17] V. Tankovich, C. Hane, Y. Zhang, A. Kowdle, S. Fanello, and S. Bouaziz. "Hitnet: Hierarchical iterative tile refinement network for real-time stereo matching". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 14362–14372.

[18] Y. Wang, Z. Lai, G. Huang, B. H. Wang, L. Van Der Maaten, M. Campbell, and K. Q. Weinberger. "Anytime stereo image depth estimation on mobile devices". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 5893–5900.

[19] J.-R. Chang and Y.-S. Chen. "Pyramid stereo matching network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 5410–5418.