Feng Chia University
Outstanding Academic Paper by Students

Title：Camera distance estimation using LiDAR tool

Author(s): Le Minh Hai

Class: 1st year of PhD Program of Electrical and Communications Engineering

Student ID: P0970673

Course: Design and Analysis of Instrument Human Machine Interface

Instructor: Prof. Tong-Wen Wang

Department: Department of Electronic Engineering

Academic Year: Semester 2, 2020-2021

Abstract

Sensor combinations can enhance reliability and significantly reduce risk to autonomous vehicle systems. Traditional methods need a lot of time to adjust and the efficiency is not high. In this project, we build a tool that can assist in sensor matching by finding the relation matrix based on 4 known points in the LiDAR and image data. Our tools can perform more efficient, intuitive and easy to handle by using GUI on Python.

Keyword：Sensor fusion, relation matrix, python GUI.

Table of Content

List of Figures

Chapter 1 Introduction

There are a lot of companies that are creating a great resource for developing the self-driving car. It brings a big change in accident reduction and people protection. In most cases, the processing of information from sensors is especially important. The vehicles can know where they are going and immediately avoid objects, give right decision. The sensors are combined to reduce blind spots as well as enhance vehicle visibility. Cameras and radar are often used as primary sensors to collect information for autonomous vehicles. Recently, LiDAR is also integrated in self-driving cars to enhance system reliability. It is also considered the eye in vehicles which can provide enough information around the vehicle and fill the missing point of other sensors. The combination of multiple sensors has solved many autonomous vehicle problems such as 3D object recognition, lane detection and positioning.

Determining the distance between objects in the camera is increasingly interested by researchers. LiDAR is one of the new approaches to figure out the distance of objects in an image. In our project, we design a tool which can estimate the distance on an image base on LiDAR data. Our proposed tool can create the transmission matrix that can matching LiDAR point with pixel from camera's image. This is an important step in finding the calibration matrix between the camera and LiDAR. In some datasets, a confusion matrix is often given. Unfortunately, we have to spend a lot of time to find the correction matrix. Therefore, we recommend a set of tools to make calibration easier. We also test with the given data set (KITTI). The results from our toolkit are very effective. We hope our proposed tool can make applications from LiDAR more accessible.

Chapter 2 Literature Review

Sensor combine is applied in many different fields, especially in the field of automation, robotics and autonomous vehicles. In this study, we focus on the combination of LiDAR and camera. One of the key benefits is increased system reliability and being able to operate under different weather conditions in real-time processing. The camera delivers data with rich information with great resolution, while LiDAR provides accurate depth data and performs well in extreme weather conditions. The combination of two sensors is really necessary for autonomous

vehicle systems.

Calibration is an important step to combine 2 sensors. Sensor parameters have to clear such as the orientation and position of the sensors. Hence, finding the relation matrix is one of the most important in sensors fusion. In this research, we find the confusion matrix which can convert 3D LiDAR points cloud into 2D images. The common methods find the function to calculate the position of 2 sensors in 2D images. Two sensors placement on the top of the vehicle was shown in Figure 1[1]. There are a little researches based on the stereo vision camera because of its ability to perceive the environment in 3D [2]. However, the most limitation is the processing on real-time of images and measurement and estimation errors [3]. Morever, sensors fusion approaches were devided into three main classes based on the different levels of data used for fusion, namely low-level fusion, feature-level fusion, and high- level fusion [4,5]. In the low-level fusion, the sensors data was combined base on the sensor's charateristic and location between each sensor. In the high-level fusion, they used object detection or a tracking algorithm for each sensor. After that, they perform fusion method to combine sensors. Each method has different limitations and benefits. Low-level matching focuses on the physical method, distance and location of sensors, and sensor characteristics. They are very suitable for use in autonomous vehicles because of their short processing time, and can operate in real-time. In this project, we using low-level fusion for realtime processing which support an autonomous driving systems.

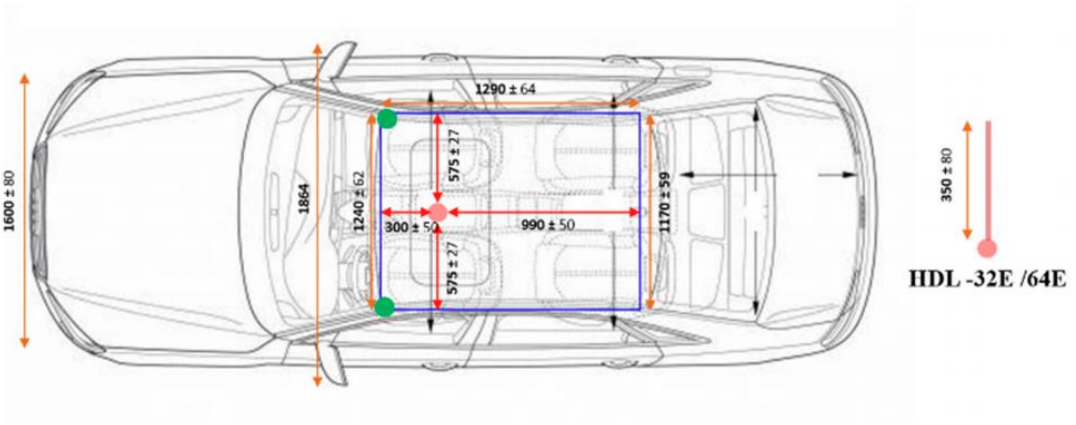

Figure 1 Sensor setup on the vehicle platform (top view).

Chapter 3 Propose System

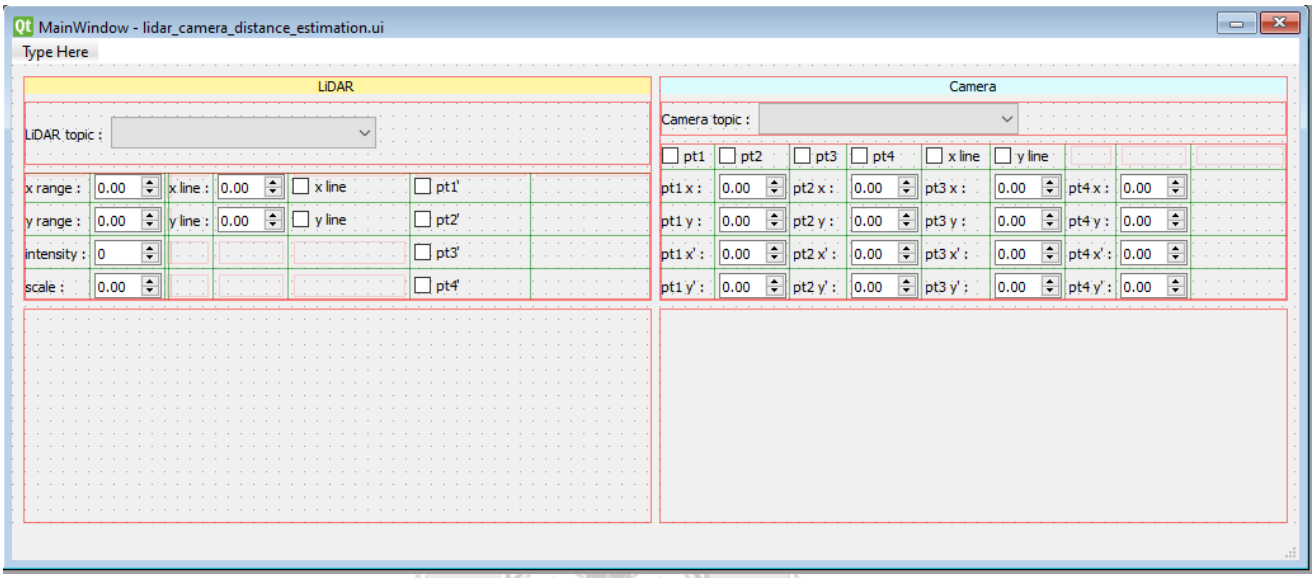

Figure 2 The User Interface of proposed system

Qt Designer was choosen to design the User Interface which was shown in Figure 2. In our tool, we devide window into 3 spaces. The left-bottom window was used to show the LiDAR information. Camera data was shown in right-bottom. The top of window shows the general information for create the confusion matrix between 2 point. Totally, this program can find the transmission matrix that matches the distance values from LiDAR with camera pixel values. This program run with pyqt and ROS (The Robot Operating System). ROS is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. Before running, we just need to match ROS message type. There are two message types on ROS.

LiDAR message type : sensor_msgs - PointCloud2

Camera message type : sensor_msgs – Image

In our proposed method, we also used 4 points to find the confusion matrix. Match the four points pt1, pt2, pt3, p4 of the image on the right with the four points pt1', pt2', pt3', pt4' corresponding to the distance of the left LiDAR data. After that, our UI will automatic create the matching matrix and show the results base on 2 lines (red line and green line) from points cloud and images.

Program language: Python

Before run Program:

Change KITTI dataset to “bag” file

Open terminal and type:

%open ROS environment

->roscore

% open kitti dataset

->rosbag play –l kitti_2011_09_26_drive_0002_syned.bag

%run python file

->python LiDAR_camera_distance_estimation.py

Chapter 4 Code

```
import sys
import os
import cv2
import numpy as np
import signal
from PIL import Image
import PySide2
from PySide2.QtUiTools import QUiLoader
from PySide2.QtWidgets import QWidget, QApplication, QPushButton, QLineEdit,
QLabel, QCheckBox, QComboBox, QTextBrowser, QDoubleSpinBox, QSpinBox
from PySide2.QtCore import QFile, QObject
import rospy
import roslib
import roslaunch
import rosbag
import ros_numpy
import pyrosbag as prb
import pcl_ros
import subprocess
```

```
import sensor_msgs.point_cloud2 as pc2
from sensor_msgs.msg import PointCloud2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

import PyQt5
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtWidgets import QApplication, QWidget, QInputDialog, QLineEdit,
QFileDialog
from PyQt5.QtWidgets import QFrame
from PyQt5.QtGui import QIcon, QImage, QPainter, QPalette, QPixmap
from PyQt5.QtCore import QTimer

import PySide2
from PySide2.QtUiTools import QUiLoader
from PySide2.QtWidgets import QWidget, QApplication, QPushButton, QLineEdit,
QLabel, QCheckBox, QComboBox, QTextBrowser, QDoubleSpinBox, QSpinBox
from PySide2.QtCore import QFile, QObject

class Form(QObject):
    def __init__(self, ui_file, parent=None):
        super(Form, self).__init__(parent)

        ui_file = QFile(ui_file)
        ui_file.open(QFile.ReadOnly)
        loader = QUiLoader()
        self.window = loader.load(ui_file)
        ui_file.close()

        # get objects from ui
        self.widget = self.window.findChild(QWidget, 'centralwidget')

        self.camera_frame = self.window.findChild(QLabel, 'label')
        self.LiDAR_frame = self.window.findChild(QLabel, 'label_2')
        self.LiDAR_SpinBox_1 = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox')
        self.LiDAR_SpinBox_2 = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_2')
```

```
        self.LiDAR_SpinBox_3 = self.window.findChild(QSpinBox, 'spinBox')
        self.LiDAR_SpinBox_4 = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_3')
        self.clibration_SpinBox_pt1_x = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_4')
        self.clibration_SpinBox_pt1_y = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_5')
        self.clibration_SpinBox_pt1_xp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_6')
        self.clibration_SpinBox_pt1_yp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_7')
        self.clibration_SpinBox_pt2_x = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_8')
        self.clibration_SpinBox_pt2_y = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_9')
        self.clibration_SpinBox_pt2_xp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_10')
        self.clibration_SpinBox_pt2_yp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_11')
        self.clibration_SpinBox_pt3_x = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_12')
        self.clibration_SpinBox_pt3_y = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_13')
        self.clibration_SpinBox_pt3_xp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_14')
        self.clibration_SpinBox_pt3_yp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_15')
        self.clibration_SpinBox_pt4_x = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_16')
        self.clibration_SpinBox_pt4_y = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_17')
        self.clibration_SpinBox_pt4_xp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_18')
        self.clibration_SpinBox_pt4_yp = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_19')
        self.draw_SpinBox_line_x = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_20')
```

```
        self.draw_SpinBox_line_y = self.window.findChild(QDoubleSpinBox,
'doubleSpinBox_21')
        self.LiDAR_x_line_CheckBox = self.window.findChild(QCheckBox,
'checkBox')
        self.LiDAR_y_line_CheckBox = self.window.findChild(QCheckBox,
'checkBox_2')
        self.pt1_CheckBox = self.window.findChild(QCheckBox, 'checkBox_3')
        self.pt1_p_CheckBox = self.window.findChild(QCheckBox, 'checkBox_4')
        self.pt2_CheckBox = self.window.findChild(QCheckBox, 'checkBox_5')
        self.pt2_p_CheckBox = self.window.findChild(QCheckBox, 'checkBox_6')
        self.pt3_CheckBox = self.window.findChild(QCheckBox, 'checkBox_7')
        self.pt3_p_CheckBox = self.window.findChild(QCheckBox, 'checkBox_8')
        self.pt4_CheckBox = self.window.findChild(QCheckBox, 'checkBox_9')
        self.pt4_p_CheckBox = self.window.findChild(QCheckBox, 'checkBox_10')
        self.camera_x_line_CheckBox = self.window.findChild(QCheckBox,
'checkBox_11')
        self.camera_y_line_CheckBox = self.window.findChild(QCheckBox,
'checkBox_12')

        self.window.comboBox.deleteLater()
        self.window.comboBox_2.deleteLater()
        self.LiDAR_combo_box = ComboBox()
        self.camera_combo_box = ComboBox()
        self.LiDAR_combo_box.setFixedSize(200, 25)
        self.camera_combo_box.setFixedSize(200, 25)
        self.window.horizontalLayout_2.addWidget(self.LiDAR_combo_box)
        self.window.horizontalLayout_3.addWidget(self.camera_combo_box)

        self.window.label_37.deleteLater()
        self.window.label_38.deleteLater()
        blank = PySide2.QtWidgets.QLabel()
        blank_2 = PySide2.QtWidgets.QLabel()
        self.window.horizontalLayout_2.addWidget(blank)
        self.window.horizontalLayout_3.addWidget(blank_2)

        # set variables
        self.sub_camera = None
        self.sub_LiDAR = None
```

```
self.camera_msg = None
self.LiDAR_msg = None
self.camera_map = None
self.LiDAR_map = None
self.bridge = CvBridge()

self.LiDAR_x = 30.0
self.LiDAR_y = 8.0
self.intensity = 30
self.scale = 20.0
self.h = int(self.LiDAR_x * self.scale)
self.w = int(2 * self.LiDAR_y * self.scale)

self.pt1_x = 444.0
self.pt1_y = 374.5
self.pt1_xp = 41.25
self.pt1_yp = 5.5
self.pt2_x = 265.0
self.pt2_y = 492.0
self.pt2_xp = 3.0
self.pt2_yp = 2.0
self.pt3_x = 562.5
self.pt3_y = 377.5
self.pt3_xp = 41.25
self.pt3_yp = -4.95
self.pt4_x = 700.0
self.pt4_y =    507.0
self.pt4_xp = 3.0
self.pt4_yp = -2.0

self.line_x = 20.0
self.line_y = 0.0

# set Timer
self.qTimer = QTimer()
self.qTimer.setInterval(1) # 1000 ms = 1 s
self.qTimer.timeout.connect(self.update)
self.qTimer.start()
```

```
        # initilization
        rospy.init_node('LiDAR_camera_calibration', anonymous=True)
        self.init_combo_box()
        self.init_SpinBox()
        self.init_CheckBox()

        # show
        self.window.show()

    def init_combo_box(self):

self.LiDAR_combo_box.popupAboutToBeShown.connect(self.LiDAR_combo_box_clicked)
        self.LiDAR_combo_box.activated.connect(self.LiDAR_topic_selected)

self.camera_combo_box.popupAboutToBeShown.connect(self.camera_combo_box_clicked)
        self.camera_combo_box.activated.connect(self.camera_topic_selected)

    def LiDAR_combo_box_clicked(self):
        try:
            # update rostopic list
            self.LiDAR_combo_box.clear()
            topic_list = []

            topic_list_dict = dict(rospy.get_published_topics())

            for key in topic_list_dict.keys():
                if topic_list_dict[key] == 'sensor_msgs/PointCloud2':
                    topic_list.append(key)

            topic_list.sort()
            self.LiDAR_combo_box.addItems(topic_list)
        except:
            pass

    def LiDAR_topic_selected(self, i):
```

```
            selected_topic = self.LiDAR_combo_box.currentText()

            if selected_topic != '':
                if self.sub_LiDAR is not None:
                    self.sub_LiDAR.unregister()
                self.sub_LiDAR = rospy.Subscriber(selected_topic, PointCloud2,
self.LiDAR_msg_callback)

        def camera_combo_box_clicked(self):
            try:
                # update rostopic list
                self.camera_combo_box.clear()
                topic_list = []

                topic_list_dict = dict(rospy.get_published_topics())

                for key in topic_list_dict.keys():
                    if topic_list_dict[key] == 'sensor_msgs/Image':
                        topic_list.append(key)

                topic_list.sort()
                self.camera_combo_box.addItems(topic_list)
            except:
                pass

        def camera_topic_selected(self, i):
            selected_topic = self.camera_combo_box.currentText()

            if selected_topic != '':
                if self.sub_camera is not None:
                    self.sub_camera.unregister()
                self.sub_camera = rospy.Subscriber(selected_topic, Image,
self.camera_msg_callback)

        def init_SpinBox(self):
            self.LiDAR_SpinBox_1.setRange(0.0, 100.0)
            self.LiDAR_SpinBox_2.setRange(0.0, 100.0)
            self.LiDAR_SpinBox_3.setRange(0, 255)
```

```
self.LiDAR_SpinBox_4.setRange(0.0, 100.0)
self.clibration_SpinBox_pt1_x.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt1_y.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt1_xp.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt1_yp.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt2_x.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt2_y.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt2_xp.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt2_yp.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt3_x.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt3_y.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt3_xp.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt3_yp.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt4_x.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt4_y.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt4_xp.setRange(-10000.0, 10000.0)
self.clibration_SpinBox_pt4_yp.setRange(-10000.0, 10000.0)
self.draw_SpinBox_line_x.setRange(0.0, 100.0)
self.draw_SpinBox_line_y.setRange(-100.0, 100.0)

self.LiDAR_SpinBox_1.setValue(30.0)
self.LiDAR_SpinBox_2.setValue(8.0)
self.LiDAR_SpinBox_3.setValue(30)
self.LiDAR_SpinBox_4.setValue(20.0)
self.clibration_SpinBox_pt1_x.setValue(450.0)
self.clibration_SpinBox_pt1_y.setValue(320.0)
self.clibration_SpinBox_pt1_xp.setValue(7.7)
self.clibration_SpinBox_pt1_yp.setValue(1.9)
self.clibration_SpinBox_pt2_x.setValue(390.0)
self.clibration_SpinBox_pt2_y.setValue(350.0)
self.clibration_SpinBox_pt2_xp.setValue(6.15)
self.clibration_SpinBox_pt2_yp.setValue(1.85)
self.clibration_SpinBox_pt3_x.setValue(875.0)
self.clibration_SpinBox_pt3_y.setValue(240.0)
self.clibration_SpinBox_pt3_xp.setValue(12.5)
self.clibration_SpinBox_pt3_yp.setValue(-4.20)
self.clibration_SpinBox_pt4_x.setValue(1000.0)
self.clibration_SpinBox_pt4_y.setValue(300.0)
```

```
            self.clibration_SpinBox_pt4_xp.setValue(9.0)
            self.clibration_SpinBox_pt4_yp.setValue(-4.7)
            self.draw_SpinBox_line_x.setValue(20.0)
            self.draw_SpinBox_line_y.setValue(0.0)

            self.LiDAR_SpinBox_1.valueChanged.connect(self.SpinBox_changed)
            self.LiDAR_SpinBox_2.valueChanged.connect(self.SpinBox_changed)
            self.LiDAR_SpinBox_3.valueChanged.connect(self.SpinBox_changed)
            self.LiDAR_SpinBox_4.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt1_x.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt1_y.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt1_xp.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt1_yp.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt2_x.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt2_y.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt2_xp.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt2_yp.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt3_x.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt3_y.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt3_xp.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt3_yp.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt4_x.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt4_y.valueChanged.connect(self.SpinBox_changed)
```

```
self.clibration_SpinBox_pt4_xp.valueChanged.connect(self.SpinBox_changed)

self.clibration_SpinBox_pt4_yp.valueChanged.connect(self.SpinBox_changed)
        self.draw_SpinBox_line_x.valueChanged.connect(self.SpinBox_changed)
        self.draw_SpinBox_line_y.valueChanged.connect(self.SpinBox_changed)

        self.clibration_SpinBox_pt1_x.setSingleStep(0.1)
        self.clibration_SpinBox_pt1_y.setSingleStep(0.1)
        self.clibration_SpinBox_pt1_xp.setSingleStep(0.1)
        self.clibration_SpinBox_pt1_yp.setSingleStep(0.1)
        self.clibration_SpinBox_pt2_x.setSingleStep(0.1)
        self.clibration_SpinBox_pt2_y.setSingleStep(0.1)
        self.clibration_SpinBox_pt2_xp.setSingleStep(0.1)
        self.clibration_SpinBox_pt2_yp.setSingleStep(0.1)
        self.clibration_SpinBox_pt3_x.setSingleStep(0.1)
        self.clibration_SpinBox_pt3_y.setSingleStep(0.1)
        self.clibration_SpinBox_pt3_xp.setSingleStep(0.1)
        self.clibration_SpinBox_pt3_yp.setSingleStep(0.1)
        self.clibration_SpinBox_pt4_x.setSingleStep(0.1)
        self.clibration_SpinBox_pt4_y.setSingleStep(0.1)
        self.clibration_SpinBox_pt4_xp.setSingleStep(0.1)
        self.clibration_SpinBox_pt4_yp.setSingleStep(0.1)
        self.draw_SpinBox_line_x.setSingleStep(0.1)
        self.draw_SpinBox_line_y.setSingleStep(0.1)

    def SpinBox_changed(self):
        self.LiDAR_x = self.LiDAR_SpinBox_1.value()
        self.LiDAR_y = self.LiDAR_SpinBox_2.value()
        self.intensity = self.LiDAR_SpinBox_3.value()
        self.scale = self.LiDAR_SpinBox_4.value()
        self.h = int(self.LiDAR_x * self.scale)
        self.w = int(2 * self.LiDAR_y * self.scale)
        self.line_x = self.draw_SpinBox_line_x.value()
        self.line_y = self.draw_SpinBox_line_y.value()
        self.pt1_x = self.clibration_SpinBox_pt1_x.value()
        self.pt1_y = self.clibration_SpinBox_pt1_y.value()
        self.pt1_xp = self.clibration_SpinBox_pt1_xp.value()
```

```
            self.pt1_yp = self.clibration_SpinBox_pt1_yp.value()
            self.pt2_x = self.clibration_SpinBox_pt2_x.value()
            self.pt2_y = self.clibration_SpinBox_pt2_y.value()
            self.pt2_xp = self.clibration_SpinBox_pt2_xp.value()
            self.pt2_yp = self.clibration_SpinBox_pt2_yp.value()
            self.pt3_x = self.clibration_SpinBox_pt3_x.value()
            self.pt3_y = self.clibration_SpinBox_pt3_y.value()
            self.pt3_xp = self.clibration_SpinBox_pt3_xp.value()
            self.pt3_yp = self.clibration_SpinBox_pt3_yp.value()
            self.pt4_x = self.clibration_SpinBox_pt4_x.value()
            self.pt4_y = self.clibration_SpinBox_pt4_y.value()
            self.pt4_xp = self.clibration_SpinBox_pt4_xp.value()
            self.pt4_yp = self.clibration_SpinBox_pt4_yp.value()

        def init_CheckBox(self):
            self.LiDAR_x_line_CheckBox.setChecked(True)
            self.LiDAR_y_line_CheckBox.setChecked(True)
            self.pt1_CheckBox.setChecked(True)
            self.pt1_p_CheckBox.setChecked(True)
            self.pt2_CheckBox.setChecked(True)
            self.pt2_p_CheckBox.setChecked(True)
            self.pt3_CheckBox.setChecked(True)
            self.pt3_p_CheckBox.setChecked(True)
            self.pt4_CheckBox.setChecked(True)
            self.pt4_p_CheckBox.setChecked(True)
            self.camera_x_line_CheckBox.setChecked(True)
            self.camera_y_line_CheckBox.setChecked(True)

        def camera_msg_callback(self, msg):
            self.camera_msg = msg

        def LiDAR_msg_callback(self, msg):
            self.LiDAR_msg = msg

        def draw_frames(self):
            if self.camera_msg is not None:
                cv_image = self.bridge.imgmsg_to_cv2(self.camera_msg,
desired_encoding="passthrough")
```

```
                h, w, _ = cv_image.shape
                self.camera_frame.resize(h, w)
                self.camera_map = cv_image
                self.camera_map = cv2.cvtColor(self.camera_map,
cv2.COLOR_BGR2RGB)

                if self.pt1_CheckBox.isChecked() == True:
                    cv2.circle(self.camera_map, (int(self.pt1_x), int(self.pt1_y)), 3, (0,
0, 255), -1)
                    cv2.putText(self.camera_map, 'pt1', (int(self.pt1_x) + 5,
int(self.pt1_y)), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1,
cv2.LINE_AA)

                if self.pt2_CheckBox.isChecked() == True:
                    cv2.circle(self.camera_map, (int(self.pt2_x), int(self.pt2_y)), 3, (0,
0, 255), -1)
                    cv2.putText(self.camera_map, 'pt2', (int(self.pt2_x) + 5,
int(self.pt2_y)), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1,
cv2.LINE_AA)

                if self.pt3_CheckBox.isChecked() == True:
                    cv2.circle(self.camera_map, (int(self.pt3_x), int(self.pt3_y)), 3, (0,
0, 255), -1)
                    cv2.putText(self.camera_map, 'pt3', (int(self.pt3_x) + 5,
int(self.pt3_y)), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1,
cv2.LINE_AA)

                if self.pt4_CheckBox.isChecked() == True:
                    cv2.circle(self.camera_map, (int(self.pt4_x), int(self.pt4_y)), 3, (0,
0, 255), -1)
                    cv2.putText(self.camera_map, 'pt4', (int(self.pt4_x) + 5,
int(self.pt4_y)), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1,
cv2.LINE_AA)

                if self.camera_x_line_CheckBox.isChecked() == True:
                    for i in np.arange(-abs(self.LiDAR_y), abs(self.LiDAR_y), 0.1):
                        x, y = self.distance_to_pixel(self.line_x, i)
                        if x < w and y < h:
```

```
                                self.camera_map = cv2.circle(self.camera_map, (int(x),
int(y)), 2, (255, 0, 0), -1)

                if self.camera_y_line_CheckBox.isChecked() == True:
                    for i in np.arange(-abs(self.LiDAR_x), abs(self.LiDAR_x), 0.1):
                        x, y = self.distance_to_pixel(i, self.line_y)
                        if x < w and y < h:
                            self.camera_map = cv2.circle(self.camera_map, (int(x),
int(y)), 2, (0, 255, 0), -1)

                height, width, channels = np.shape(self.camera_map)
                totalBytes = self.camera_map.nbytes
                bytesPerLine = int(totalBytes / height)
                qimg = PySide2.QtGui.QImage(self.camera_map.data,
self.camera_map.shape[1], self.camera_map.shape[0], bytesPerLine,
PySide2.QtGui.QImage.Format_RGB888)
                pixmap = PySide2.QtGui.QPixmap.fromImage(qimg)
                self.camera_frame.setPixmap(pixmap)
                self.camera_frame.show()

            if self.LiDAR_msg is not None:
                self.LiDAR_map = np.zeros((self.h, self.w, 3), dtype=np.uint8)

                for p in pc2.read_points(self.LiDAR_msg):
                    x, y, z, i, r = 0, 0, 0, 0, 0

                    if len(p) == 4:
                        x, y, z, i = p

                    elif len(p) == 5:
                        x, y, z, i, r = p

                    resized_x = self.scale * x
                    resized_y = self.scale * y

                    if self.h - resized_x > 0 and self.h - resized_x < self.h and self.w/2
- resized_y > 0 and self.w/2 - resized_y < self.w:
                        if i > self.intensity:
```

```
                                self.LiDAR_map[int(self.h - resized_x)][int(self.w/2 -
resized_y)][0] = 255
                                self.LiDAR_map[int(self.h - resized_x)][int(self.w/2 -
resized_y)][1] = 0
                                self.LiDAR_map[int(self.h - resized_x)][int(self.w/2 -
resized_y)][2] = 255
                            else:
                                self.LiDAR_map[int(self.h - resized_x)][int(self.w/2 -
resized_y)][0] = 255
                                self.LiDAR_map[int(self.h - resized_x)][int(self.w/2 -
resized_y)][1] = 255
                                self.LiDAR_map[int(self.h - resized_x)][int(self.w/2 -
resized_y)][2] = 255

                if self.LiDAR_x_line_CheckBox.isChecked() == True:
                    cv2.line(self.LiDAR_map, (0, int((self.LiDAR_x -
self.line_x)*self.scale)), (int(self.w), int((self.LiDAR_x - self.line_x)*self.scale)),
(255, 0, 0), 2)

                if self.LiDAR_y_line_CheckBox.isChecked() == True:
                    cv2.line(self.LiDAR_map, (int((self.LiDAR_y -
self.line_y)*self.scale), 0), (int((self.LiDAR_y - self.line_y)*self.scale), self.h), (0,
255, 0), 2)

                if self.pt1_p_CheckBox.isChecked() == True:
                    cv2.circle(self.LiDAR_map, (int(self.w/2 - self.pt1_yp*self.scale),
int(self.h - self.pt1_xp*self.scale)), 3, (0, 0, 255), -1)
                    cv2.putText(self.LiDAR_map, 'pt1\", (int(self.w/2 -
self.pt1_yp*self.scale) + 5, int(self.h - self.pt1_xp*self.scale)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)

                if self.pt2_p_CheckBox.isChecked() == True:
                    cv2.circle(self.LiDAR_map, (int(self.w/2 - self.pt2_yp*self.scale),
int(self.h - self.pt2_xp*self.scale)), 3, (0, 0, 255), -1)
                    cv2.putText(self.LiDAR_map, 'pt2\", (int(self.w/2 -
self.pt2_yp*self.scale) + 5, int(self.h - self.pt2_xp*self.scale)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)
```

```
            if self.pt3_p_CheckBox.isChecked() == True:
                cv2.circle(self.LiDAR_map, (int(self.w/2 - self.pt3_yp*self.scale),
int(self.h - self.pt3_xp*self.scale)), 3, (0, 0, 255), -1)
                cv2.putText(self.LiDAR_map, 'pt3\", (int(self.w/2 -
self.pt3_yp*self.scale) + 5, int(self.h - self.pt3_xp*self.scale)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)

            if self.pt4_p_CheckBox.isChecked() == True:
                cv2.circle(self.LiDAR_map, (int(self.w/2 - self.pt4_yp*self.scale),
int(self.h - self.pt4_xp*self.scale)), 3, (0, 0, 255), -1)
                cv2.putText(self.LiDAR_map, 'pt4\", (int(self.w/2 -
self.pt4_yp*self.scale) + 5, int(self.h - self.pt4_xp*self.scale)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)

            self.LiDAR_frame.resize(self.h, self.w)
            height, width, channels = np.shape(self.LiDAR_map)
            totalBytes = self.LiDAR_map.nbytes
            bytesPerLine = int(totalBytes / height)
            qimg = PySide2.QtGui.QImage(self.LiDAR_map.data,
self.LiDAR_map.shape[1], self.LiDAR_map.shape[0], bytesPerLine,
PySide2.QtGui.QImage.Format_RGB888)
            pixmap = PySide2.QtGui.QPixmap.fromImage(qimg)
            self.LiDAR_frame.setPixmap(pixmap)
            self.LiDAR_frame.show()

        self.init_frame()

    def init_frame(self):
        self.camera_map = None
        self.LiDAR_map = None
        self.calibration_map = None

    def update(self):
        try:
            # update frames
            self.draw_frames()
        except Exception as e:
            print e
```

```
                pass

    def distance_to_pixel(self, x, y):
            pixel = np.float32([[self.pt1_x, self.pt1_y], [self.pt2_x, self.pt2_y],
[self.pt3_x, self.pt3_y], [self.pt4_x, self.pt4_y]])
            distance = np.float32([[self.pt1_xp, self.pt1_yp], [self.pt2_xp, self.pt2_yp],
[self.pt3_xp, self.pt3_yp], [self.pt4_xp, self.pt4_yp]])

            IM = cv2.getPerspectiveTransform(distance, pixel)

            IMa = IM[0, 0]
            IMb = IM[0, 1]
            IMc = IM[0, 2]
            IMd = IM[1, 0]
            IMe = IM[1, 1]
            IMf = IM[1, 2]
            IMg = IM[2, 0]
            IMh = IM[2, 1]
            IMi = IM[2, 2]

            xa = (IMa * x + IMb * y + IMc) / (IMg * x + IMh * y + 1)
            ya = (IMd * x + IMe * y + IMf) / (IMg * x + IMh * y + 1)

            return xa, ya

class ComboBox(PySide2.QtWidgets.QComboBox):
    popupAboutToBeShown = PySide2.QtCore.Signal()

    def showPopup(self):
            self.popupAboutToBeShown.emit()
            super(ComboBox, self).showPopup()

if __name__ == '__main__':
    QtCore.QCoreApplication.setAttribute(QtCore.Qt.AA_ShareOpenGLContexts)
    #qt_app = QtWidgets.QApplication(sys.argv)
    app = QApplication(sys.argv)
    form = Form('LiDAR_camera_distance_estimation.ui')
    sys.exit(app.exec_())
```

Chapter 5 Results

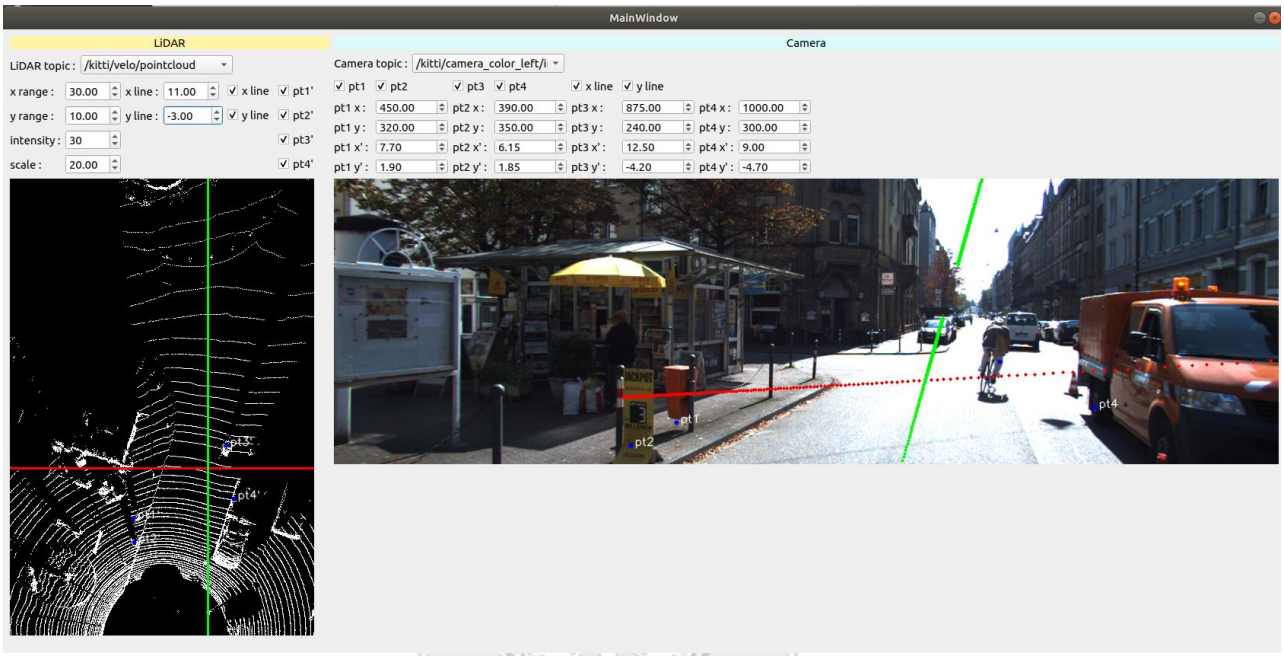

Figure 3 Final frame with KITTI dataset

In our project, we choose KITTI dataset for evaluate our method. The final result was shown in Figure 3. The detail of operation was demonstrated through video: https://youtu.be/gBV89bSepo4

Chapter 6 Conclusion

Our proposed UI can match 3D points cloud with 2D pixel on images base on 4 points. The proposed method saves a lot of time to find the correction matrix. The KITTI dataset was choosen for demonstrate our system. Two testing line can matching almost point and pixel. We hope we can easy to build up LiDAR on autonomous vehicle by our proposed tool in the future.

References

[1] G Ajay Kumar , J. H. Lee, J Hwang, J Park, S H Youn and S. Kwon. LiDAR and Camera Fusion Approach for Object Distance Estimation in Self-Driving Vehicles. Symmetry. Feb,2020.

[2] Maxime, D.; Aurelien, P.; Martial, S.; Guy Le, B. Moving Object Detection in Real-Time Using Stereo from a Mobile Platform. Unmanned Syst. 2015, 3, 253–266.

[3] Raphael, L.; Cyril, R.; Dominique, G.; Didier, A. Cooperative Fusion for Multi-Obstacles Detection with Use of Stereovision and Laser Scanner. Auton. Robot. 2005, 19, 117–140.

[4] Douillard, B.; Fox, D.; Ramos, F.; Durrant-Whyte, H. Classification and semantic mapping of urban environments. Int. J. Robot. Res. 2011, 30, 5–32.

[5] Li, Q.; Chen, L.; Li, M.; Shaw, S.L.; Nüchter, A. A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios. IEEE Trans. Veh. Technol. 2014, 63, 540–555.