Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Vehicle Visual and Distance Sensor Fusion and Real-Time Processing

Author:
Belén Barbed Martín

Supervisor:
Dr. Soteris Demetriou

Second Marker:
Dr. Anandha Gopalan

June 19, 2019

Abstract

In the wake of vehicle automation, there are still millions of cars in circulation with no sensing or smart capabilities. Given their expected longevity in many areas of the world, this report develops and analyses a specific system to elevate the autonomy of those cars with additional hardware and software that are of common availability nowadays. A car mount, designed to be fitted to the roof of a car or van, was developed, where a VLP-16 Velodyne LiDAR and four smartphones were responsible for gathering 3D, 360° information of its surroundings. A central compute node in the car runs ROS to manage the sensor network, from which the observations from all inputs are fused together to determine where other on-road vehicles are located, extrapolating their distance. Once all of this has been computed, it is visualised in real-time on the computer's screen. The system was successfully evaluated in an indoors environment using facial recognition instead of car detection, achieving accurate results overall. This paper focuses on the collection and real-time analysis of sensor data, but the software could be easily expanded to incorporate other additional capabilities such as path planning or lane following.

Acknowledgements

There are many people to thank for helping, advising, and generally being there for me for the duration of this project. Without them, this paper would not exist (deciding whether that is a good or bad thing is left as an exercise to the reader).

- To **Soteris**, my project supervisor, for proposing and guiding me throughout the entirety of this project, making it an amazing ending of my degree.
- To **Anandha and Tom Clarke**, for aiding and advising me throughout the project selection and development, and for having their door always open.
- To **Kai and Flora**, the Master's students who will take on this project after me, for their continuous support and pointing out the obvious when I needed it most.
- To all the **developers** who worked on the open-source platforms and projects used in this paper: ROS Melodic, OpenCV, Python, and all the packages that stemmed from papers hereby cited.
- To the **Robotics Society and all its members: Cheryl, Nick, Churk, Tomasz, Sergey, Gavin, Elsa, Sam...** for unknowingly reassuring me that I am leaving ICRS in the best of hands.
- To **Tom**, for listening to my "ROS isn't working!" rants at 11pm in ICRS.
- To **Mwana**, my housemate and fellow EIE girl, for not bringing up the topic of FYPs or reports at home.
- To my **family**, especially my **parents** and my boyfriend **Ben**, for their relentless love and support.

Thank you.

Contents

Chapter 1

Introduction

The UK's Department for Transport reported that there were 31.5 million licensed cars in circulation last year [1]. The average age of these vehicles was around 8 years, which is long before some manufacturers began to design and sell cars with sensor-rich capabilities, and even some level of autonomy in simpler tasks such as parking [2]. If the future of autonomous cars is upon us, what will happen to all these manual, sensor-less cars?

Two scenarios potentially arise: they may be forcefully removed from circulation through legislation, otherwise legacy and autonomous vehicles would have to coexist for an extended period of time whilst these older cars are slowly replaced over the span of almost decades. While autonomous and manual cars can indeed share the space, the latter would still pose the very same safety issues that we are trying to mitigate with smarter vehicles, such as distracted drivers, speeding, driving under the influence, etc. [3].

What if we turned the problem on its head and retroactively fitted legacy cars with the hardware and software needed in an attempt to match a new car's capabilities?

With this method, car manufacturers could future-proof their older cars with a hardware add-on which, much like a software update, would ensure they are up to the proper safety standards newer vehicles are held to. This could also lengthen the time before said cars become obsolete, saving their owners money by being on the road for longer with the appropriate capabilities.

Another interesting application is within the field of research. If an individual, team, or institution wanted to work with an autonomous or semi-autonomous vehicle, they would have to wait years for them to be available to consumers, to then spend considerable budget in obtaining a brand new state-of-the-art car. By allowing them to use an older model with retro-fitted sensors, they would be able to conduct the very necessary research around these kind of vehicles before they flood the market. It is for this particular demographic that the system should remain brand and model-agnostic.

1.1 Objectives

The overarching goal of the project is to emulate the sensory functionality of a modern or autonomous vehicle using a legacy car and external hardware.

More specifically, the system should achieve real-time visual and distance information of the vehicles surrounding the user's car: their car type, colour, angle and distance from the mount. In order to achieve this, a rig to hold the necessary sensors on the vehicle must be built, and the software to extract and process the data gathered by them be developed. The merging of sensor data into a single stream to achieve these results is commonly referred to as "sensor fusion", the terminology that shall be used throughout this report.

The measures used to evaluate the system will be latency (how close to "real-time" it actually is), framerate (of the output), and accuracy (how precisely surroundings cars are detected). Other secondary goals include having an easily replicable hardware set-up and making the system fully open-source.

To summarise, the project entails the following requirements:

- Building a mount that can be attached to the roof of a vehicle.
- Fitting said mount with the necessary visual and distance sensors.
- Calibrating the sensors to allow for sensor fusion later on.
- Connecting all the sensors to a ROS system running on a laptop.
- Collecting real-time data from these sensors.
- Detecting cars and their features on the camera feeds.
- Performing sensor fusion with the data to extrapolate the distance and angle of the cars discovered.
- Visualising all the resulting real-time 360° information in an appropriate manner.

1.2 Challenges

The main challenge this project posed was the implementation of all the functionality in real-time. Such a constraint has had to be kept in mind during every step of the software development cycle, from design to implementation and testing.

An associated restriction at the hardware level is related to the computational power, or rather the lack of it. The project has been developed on a run-of-the-mill laptop with an integrated graphics card and an average processor. Consequently, the code had to be as lightweight as possible while maintaining sensible levels of accuracy, and minimal latency.

Another relevant challenge, unassociated with performance, was connecting all the sensors together. Finding an adequate manner to access the phones' camera streams through ROS is one of the pillars of this research, and no single perfect solution exists.

1.3 Contributions

The author has made the following individual contributions:

- As part of the project, several ROS packages were written, which are detailed in Section 3.2.1 of the implementation Chapter.
- The resulting ROS system was integrated with the app that published the phones' sensor data into ROS topics, so that all nodes would have access to the readings.
- General performance and functionality improvements were also made to the external libraries employed in the project.
- As a task at the mount design phase, a prototype was made, which was used during the initial steps of implementation.
- Finally, a visualisation GUI was designed to showcase the real-time processed data.

Chapter 2

Related Works

This project was inspired by and spun out from a previous undertaking of its supervisor, Dr. Demetriou, titled "CoDrive: Improving Automobile Positioning via Collaborative Driving" [4]. In Dr. Demetriou's paper, a collaborative method between sensor-rich and older cars to improve location accuracy for both kinds of vehicles is proposed. A roadblock emerged while investigating and testing different mechanisms: a lack of accessibility to a programmable sensor-rich car. To circumvent this, the authors created an add-on that would make a legacy car behave like a modern one: the very same mount featured in this project. Using a Velodyne LiDAR and four smartphones, they were able to sufficiently replicate the behaviour and capabilities of a smart car within the context of their study.

The smartphones were used as cameras for vehicle detection, and the LiDAR, a laser-based sensor capable of measuring distances to objects in a 3D environment, was used to determine how far the surrounding vehicles were to the mount.

Although the final objective of their project differed from this one, their original premise has remained the same: how to best exploit the collaboration of modern and legacy cars, while also adding sensors and more intelligence to those old vehicles. Ultimately, the work developed could form the basis for a system that transforms a legacy car into an autonomous vehicle. The most relevant aspects of this paper to take inspiration from are the car mount design itself (shown in Figure 2.1), and the general ROS architecture that processes the sensors' inputs. The ROS design shown in the paper uses three publishers for the three inputs of GPS, camera feeds, and LiDAR datapoints, and a subscriber that is responsible for merging those values.

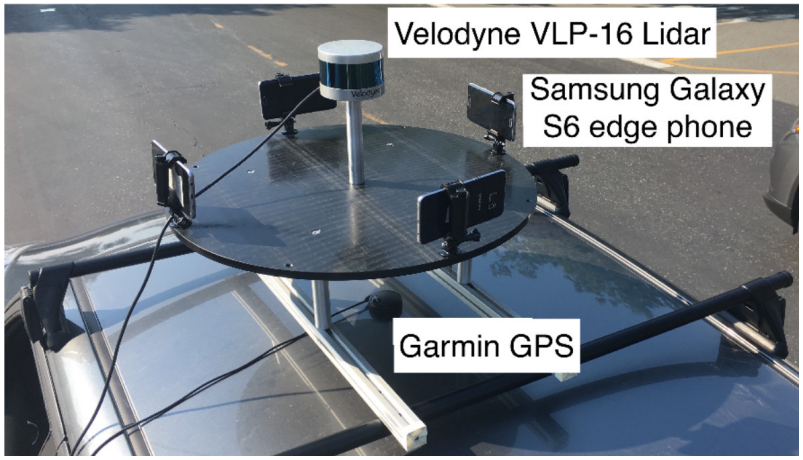


Figure 2.1: Car mount developed for Dr. Demetriou's CoDrive paper [4].

2.1 Hardware

As the purpose of this project is to, in short, make legacy cars closer to modern, sensor-rich cars, an upfront analysis of the offering of state-of-the-art vehicles forms the starting point of this report.

To first settle the relevant nomenclature standards when discussing autonomous cars, the SAE's (Society of Automotive Engineers) five "Levels of Driving Automation", as described in their widely-accepted standard [5], must be employed. The standard defines levels that go from 0 to 5, where level 0 represents no driving automation or assistance, and level 5 a fully autonomous vehicle with no user behind the wheel (even in the case of an emergency). Shortly summarising each category [6]:

- Level 0 (No automation): driver performs all duties at all times.
- Level 1 (Driver assistance): Driving Automation System (DAS) may perform either longitudinal or lateral vehicle movement (brake, steer, accelerate, etc.). Examples are lane following, cruise control, and emergency brakes.
- Level 2 (Partial automation): DAS may perform both longitudinal and lateral vehicle movements simultaneously for a short while. The smartest vehicles in the market (the likes of Tesla) are at this level.
- Level 3 (Conditional automation): DAS can drive with total autonomy under all conditions, but may relinquish control to the driver if deemed dangerous. Such autonomy is observed in the autonomous vehicles currently being tested by Google and Uber on public roads.
- Level 4 (High automation): DAS is designed to operate under all conditions and with no driver intervention at all, able to safely stop circulating in case of an emergency.
- Level 5 (Full automation): Similar to level 4, but the car itself is not designed to be driven by humans, and may therefore lack some or all controls for them.

According to said standard's report, as of 2018, only levels 1 and 2 have been realised in purchasable models thus far, with level 3 on the horizon. Considering this state of progress, it was determined that the system hereby developed should approximately correspond to a level 2 vehicle, with the intention of elevating a car from level 0 to level 2.

Consequently, some level 2 vehicles will be further analysed, starting with the aforementioned Tesla brand, specifically their most recent Model 3. When people think of electric, autonomous vehicles their minds tend focus on Tesla vehicles first. Their fame is not unfounded, as the manufacturer allegedly produces the best-selling electric cars in the world, with over 245,000 units sold globally in 2018 alone [7]. The Model 3 in question, which accounted for over half of the sales figure previously cited, boasts four cameras (front, rear, and sides), twelve ultrasonic sensors, and one long-range front-facing radar with a visibility of up to 160m ahead [2]. While recreating such an array of sensors would incur a budget unbecoming of this project, this reference model provides a good idea as to what sensors are appropriate for the application at hand. Firstly, a mixture of sensors should be contemplated, as each one of them serves its own purpose; for example, the long-range front-facing radar detects obstacles ahead well in advance, while the shorter-range ultrasonic sensors around the vehicle are used in more delicate manoeuvres such as parking, where accuracy at close quarters is imperative.

Another important aspect of this sensor network is being dual-purpose: this consideration allows the car to have autonomous behaviour whilst also aiding the driver in their decision-making by offering more information than what they could observe otherwise. This is common amongst vehicles of this calibre, with many featuring high-definition screens on their dashboards to display the plethora of information gathered by the aforementioned sensors.

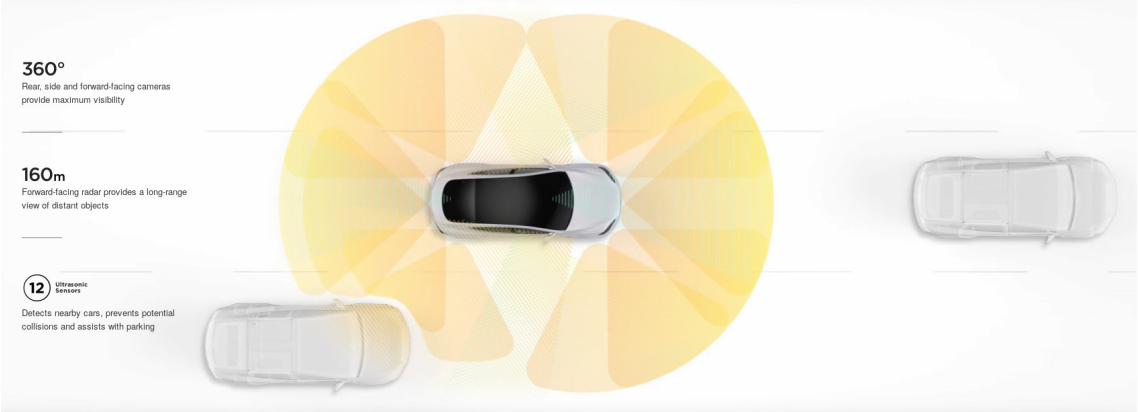


Figure 2.2: Graphic depicting the 360° sensing capabilities of the Tesla Model 3, as shown on the manufacturer's website [2].

Other big names in the race for marketable vehicle autonomy are Audi, BMW, Mercedes, and Cadillac, amongst several others [6].

A spotlight recently shone on the manufacturer Audi when it was forced to suppress some autonomous features in their latest A8 model due to UK legislation. The technology "Traffic Jam Pilot" is able to take full control of the car's manoeuvres at speeds below 60km/h. The reason this feature stands out is because Audi themselves cite it as a level 3 system, such that the driver does not have to be paying constant attention to the road while the pilot is in operation [8]. By definition, level 3 capabilities are not yet supported in the UK's law, as Article 104 of The Road Vehicles (Construction and Use) Regulations of 1986 directly forbids any mechanism that would enable the driver to relinquish control of the car [9]. It is most probable that such a regulation will be replaced soon in order for the technology to flourish, but the legal landscape hasn't changed just yet. It is also worth mentioning that the A8's dashboard includes a driver-facing camera that causes the system to fire a warning if the driver becomes distracted or tired, as well as a feature to safely stop the car if it cannot give control back to the driver when traffic exceeds 60km/h once again [10].

Diving into the A8's sensor network, capable of delivering level 3 autonomy, the car features twelve ultrasonic sensors, four 360° cameras in the front, rear, and side mirrors, and one long-range radar at the front of the vehicle. These sensors are, word for word, exactly the same as those featured in the Tesla Model 3. However, as befits a level 3 car, Audi's A8 includes several more: two extra front cameras (one up on the windshield and an infrared for night-time vision), another four mid-range radars on the four corners, and a laser scanner on the front (a sensor no other competitor has, as they claim) [10]. As seen in Figure 2.3, this results in a highly rich and redundant sensor network, an invaluable building block for autonomous driving.

Once again, similar conclusions may be drawn from this vehicle as those applicable to Tesla's Model 3: there exist multiple sensor types, and there is great redundancy between them, especially at the front of the vehicle, where it matters most.

While replicating a sensor network the likes of Audi's A8 or Tesla's Model 3 would be more than ideal, an extra bottleneck, besides budget, becomes evident: computational power. In a nutshell, it's no use having tens of sensors feeding information into the system if a processing unit capable of dealing with such a load is not in place. Computer vision, sensor fusion, and general safety-keeping features can be complex undertakings, even requiring dedicated GPUs and multi-core CPUs. For example, the A8 includes a Cylcone V FPGA and a NVIDIA Tegra K1 [10]. These are not ordinary components, and are integral to delivering the level 2 and 3 features the car has to offer. Moreover, they are more complex to program, as this would require at least handling three disciplines: classic CPU programming, parallel optimisation using GPUs, and circuit-level design on an FPGA. To put it simply, it would be an ambitious undertaking for a single individual.

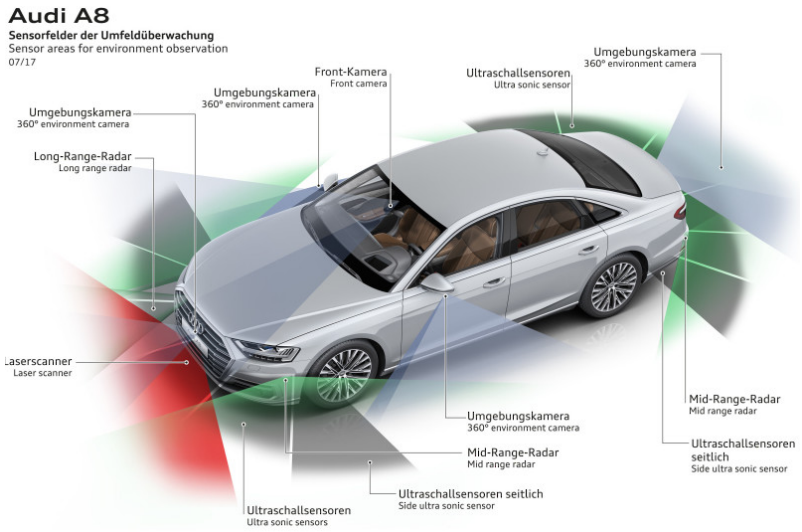


Figure 2.3: Audi A8's rich sensor network [10].

Bearing this in mind, the decision was made to build a simplified system with four cameras (front, rear, and both sides) and a 360° LiDAR. Once that had been determined, the issue of how to arrange the sensors subsequently arose. Certain previous papers have, in an attempt to imitate modern cars as much as possible, taken the vehicle apart to fully integrate the sensors into the infrastructure and driving system [11]. While it produces a realistic imitation, this approach requires the exclusive use of a car and undertaking the somewhat time-consuming process of taking it apart. Unfortunately this project cannot afford to incur either of these costs.

Alternatively, instead of distributing the sensors around the vehicle like a modern car would, the five elements could be placed on top of a removable mount held onto the car's roof rack. The main drawback of such a design, also featured in Dr. Demetriou's preceding paper [4], is the sensor's field of view. By having them placed so high up, they might miss readings from very close to the vehicle due to how far downwards they are able to perceive. For example, the Velodyne VLP-16 LiDAR has a maximum of 15° viewing angle above and below the xy plane [12], outside of which it cannot perceive objects. Because of these hardware limitations, placing the rig too high could vastly decrease the sensors' road visibility.

However, the mount design has two major advantages: mobility and universality. By placing all the hardware on a mount that can be easily bolted on top of a car, it can be tested and used on a wide variety of vehicles with minimal installation cost. It would also make it possible to conduct in-lab experiments instead of having to go out on the road every time an assessment of a new feature or change needs to be made.

The most pressing concern of using only four cameras as opposed to many overlapping sensors is the possibility of incurring blind spots, which purchasable cars cannot afford. To solve this issue, building a modular infrastructure to which adding more cameras or other sensors facilitating a feasible extension becomes imperative.

2.2 Sensor Calibration

Once it was determined that the approach would involve the mixture of visual and distance sensors, the matter of sensor fusion arose. Sensor fusion involves combining the information from two or more different sensors to obtain a more detailed view of the environment [13]. Often these sensors are observing the same world but analysing different spectra: visual, radar, infrared, GPS, etc. which is why merging their results can so greatly enrich our knowledge of the environment. Sensor fusion will be discussed in more detail in the following Section 2.3.

Having decided that real-time sensor fusion would be performed, the prerequisite of sensor calibration was investigated. By design, two sensors viewing the same world will have slightly different perspectives. From position, angle, and field of view, to depth of field and error tolerances at very close or very far distances, there are many factors that determine to what extent these sensors' focuses differ.

In order to correctly mesh together some sensor inputs, it becomes necessary to determine what each of them is capturing and how to translate from one to the other; in other words: if one sensor sees an object, where is that object in the others' field of view?

The transformation matrix, in computer graphics terms, is the solution to this application level problem [14]. Each sensor is operating in its own coordinate system in the world, and the software needs to easily translate points from one system to the other in order to figure out how two or more sensors are viewing the same object. Because the two coordinate worlds are fixed, once their relationship has been quantified and the transformation matrix has been calculated, applying the transformation to every point becomes an $O(n)$ problem.

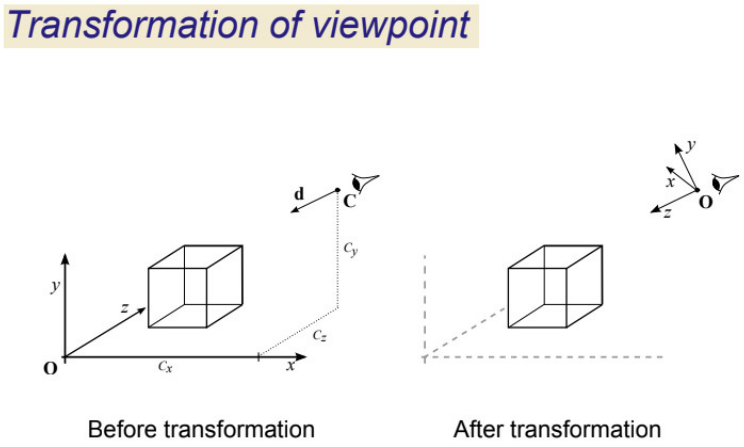


Figure 2.4: Application of the transformation matrix in the context of Computer Graphics. Taken from Imperial College London Department of Computing's CO317 Graphics course taught by Prof. Rueckert and Dr. Kainz [14].

In computer graphics, a transformation matrix is often used to change the camera's viewpoint, as shown in Figure 2.4, and is comprised of a translation (moving the origin from $(0, 0, 0)$ to the viewpoint) and a rotation around that viewpoint. For this project, the same principle applies: by using the transformation matrix, the viewpoint can be shifted from one sensor to the other. The mathematical derivation of such matrix taught in the CO317 Graphics [14] course is shown in Figure 2.5.

In literature, two main methods of sensor calibration can be found: dynamic and static.

Static methods revolve around finding the transformation matrix beforehand, such that while the application is running on the robot or vehicle, the same matrix is used for sensor fusion. Assuming that the sensors don't shift in space with respect to each other over time, the transformation matrix should remain constant. In this way, incurring a small amount of initial computational cost to determine the matrix means almost instant sensor fusion calculations in real-time.

$$\mathcal{A} = \begin{pmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \ \mathcal{B} = \begin{pmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \ \mathcal{C} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\phi & -sin\phi & 0 \\ 0 & sin\phi & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{T} = \mathcal{ABC}$$

where:

$\mathcal{A}$ is the translation to the viewpoint at (C_x, C_y, C_z).
$\mathcal{B}$ is the subsequent rotation about the y axis.
$\mathcal{C}$ is the subsequent rotation about the x axis.
and $\mathcal{T}$ is the final transformation matrix.

Figure 2.5: Derivation of the transformation matrix in the context of Computer Graphics. Taken from Imperial College London Department of Computing's CO317 Graphics course taught by Prof. Rueckert and Dr. Kainz [14].

A simplistic approach for static calibration is developed in the paper "Sensor Fusion for Obstacle Detection and Its application to an Unmanned Ground Vehicle" [15]. In this paper, the authors derive the transformation matrix from the vehicle's structure: they know the relative displacements of the visual and radar sensors in the x (forward) and z (height) axes. Having the relative positions of the sensors in the world, the matrix or equation that transforms one viewpoint into the other can be determined. While this is a sensible and straightforward method, it does not account for the gradual shift of sensors over time due to natural wear and tear, as well as being very hardware-dependent: any changes made to the structure of the vehicle or any of its sensors will warrant a new calibration iteration. It is for these reasons that this project did not employ a similar technique, and other static calibration methods were sought out.

The other branch of static calibration involves using the sensors in the robot or vehicle and placing some marker in their joint field of view. By analysing how each of them registers the common marker, their spatial relationship in the world can be inferred, and therefore the transformation matrix can be obtained. The main advantage of such an approach is that it may be performed any time the sensors are up and running to re-calibrate. If either the rig design, sensors, or sensor placement change, the calibration protocol may be performed again without having to manually measure the characteristics and locations of the new hardware.

The paper "Radar and Vision Sensor Fusion for Object Detection in Autonomous Vehicle Surroundings" involved calibrating one camera and one radar with the purpose of detecting and avoiding obstacles in a future autonomous vehicle [16]. They used seven corner-reflectors as markers, that can be detected by both the camera (visually) and the radar (by reflecting the wave back to the sensor). A derivation similar to that shown in Figure 2.5 was carried out with these seven pairs of variables (the locations of the markers in each of the 2 coordinate systems) and the transformation matrix was found. According to their evaluation, this technique resulted in an accuracy of 92% to 99%, and is thus deemed reliable. Learning about this method served the purpose of demonstrating how consistently accurate static calibration could be, but their mechanism could not be used in the project at hand, as we are working with a LiDAR distance sensor rather than a radar.

Involving camera and LiDAR calibration, a promising avenue was suggested by Dr. Demetriou: "Calibration of RGB Camera With Velodyne LiDAR" [17]. This method, like the one above, uses markers both sensors can detect in order to determine the so-called "6 degrees of freedom (6DoF)", another nomenclature for the transformation matrix. The technique developed uses a single marker: a large sheet of paper with four circles cut out of it, placed in front of a plain wall of a different colour. The camera is able to distinguish the marker due to the four evenly-spaced circles in a different colour than their surroundings, while the radar identifies four circular-shaped abrupt changes in distance, registering the wall where the cut-outs are.

These visualisations are shown in Figure 2.6, as taken from their paper. In order to confirm that the circles both sensors have detected are the ones from the marker, the calibration module takes in their radii and distance from each other. When it has been determined that they are both looking at the same correct marker, once again a transformation matrix is derived using the coordinates of both sensors.

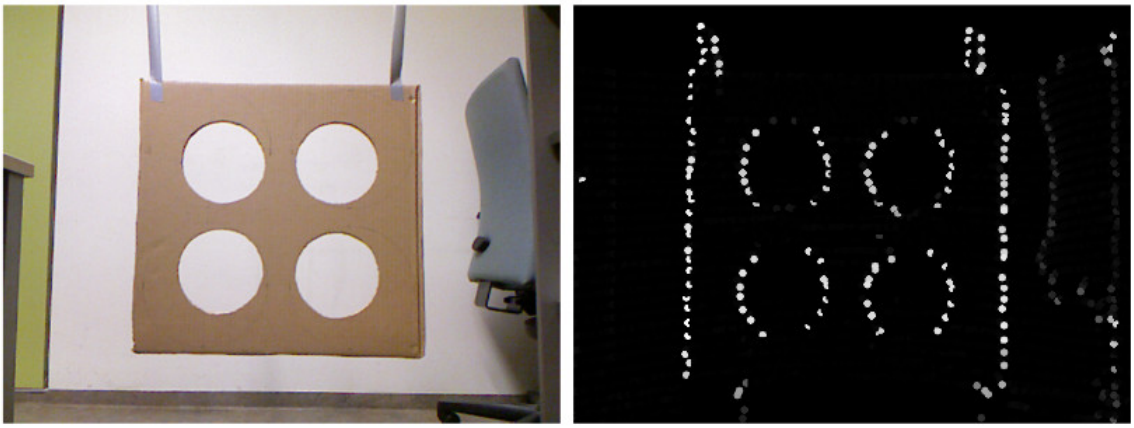


Figure 2.6: How the camera sees the marker (left), vs. how the LiDAR sees the marker (right) [17].

Aside from the theoretical derivation, the group developed a ROS package which implements exactly what was developed on their paper, which is available on GitHub [18]. Given that it was written by the same engineers (save a few external pull requests), it may be deemed a trustworthy and correct realisation of the theory, and thus it was chosen to be a dependency of the project. This approach fits our objective like a glove, as even the same brand of LiDAR is used: Velodyne. There were, however, some minor differences that led to the slight altering of their library, such as using four cameras instead of one, and the granularity of the LiDAR - the paper was using a VLP-32 and this project a VLP-16. The 32 channel version has double the number of laser emitter and receiver pairs (channels) than the 16 channel LiDAR, resulting in more granular readings of the environment. These changes made to the library will be discussed in more detail in section 3.2.1.

Another static calibration technique worth mentioning is developed in the paper "Accurate Calibration of LiDAR-Camera Systems Using Ordinary Boxes" [19]. Accessibility and set-up speed are some of its strong suits, as well as dealing with multiple camera feeds right off the bat. The final application is to calibrate a system of three LiDARs (two VLP-16 and one VLP-64) and two cameras placed on a car.

In order to identify the boxes in both the camera and LiDAR inputs, this method uses plane detection, but it does require the user to roughly delimit where the boxes are in the frames. It then iterates over the found planes and rotates them until the planes in different sensors converge. After inverting the resulting operations, the translation matrix is obtained. This technique was particularly interesting in that it used an iterative algorithm to derive its results.

Using iterative methods to derive the 6DoF required for sensor fusion has been used in earlier papers as well, such as in "Automatic camera and range sensor calibration using a single shot" [20]. This earlier method combines plane detection and printed chequerboards as markers.

So far, only static calibration techniques have been discussed. Dynamic methods are harder to come across due to their higher computational requirements and post-processing of sensor input. Dynamic calibration involves detecting objects individually in each of the sensor streams and then calculating whether two objects detected by two different sensors are indeed the same object in the real world. In "Sensor and Navigation Data Fusion for an Autonomous Vehicle" [21], a Kalman information filter is used to merge these found objects where it determines that the readings refer to the same object. Such a method effectively joins sensor calibration and sensor fusion in the same

computational step, meaning there is no initial calibration overhead, but the sensor fusion system is exponentially more complex due to the on-the-fly calibration of the sensors. It also results in not being able to freely translate from one sensor coordinate system to the other, relying on object detection to substitute such calculations.

Given the necessity for real-time operation and the sheer amount of information in the proposed sensor network (four HD cameras and a LiDAR), dynamic calibration may not be something this project can afford to do on a run-of-the-mill laptop; it is nevertheless an interesting approach deserving of mention.

2.3 Sensor Fusion

After having used a sensor calibration technique of choice, one can move on to sensor fusion. As outlined before, this involves merging the data from several sensors to enrich the system's knowledge of the environment around the vehicle.

Within this project, sensor fusion will have a fairly simple goal: once a car (or other suitable object) has been detected, the algorithm should take in the position of said car according to the relevant camera stream and extrapolate which distance reading that corresponds to. As such, it would only be necessary to use the transformation matrix on a single point per car. This point-to-point translation using a matrix was also covered in the CO317 Graphics course [14]. The matrix multiplication involved is computationally cheap to undertake, and is shown in Figure 2.7.

$$\mathbf{P}_t = \mathcal{T}\,\mathbf{P}$$

where:

$\mathbf{P}$ is the point in the original coordinate system.
$\mathbf{P}_t$ is the point in the new coordinate system.
and $\mathcal{T}$ is the transformation matrix.

Figure 2.7: Using the transformation matrix to move a point from one coordinate system to another. Taken from Imperial College London Department of Computing's CO317 Graphics course taught by Prof. Rueckert and Dr. Kainz [14].

The numerous papers explored in this section used this simplistic method to conduct sensor fusion under real-time constraints [13][15][16][17], most of which also discussed sensor calibration techniques.

The reason for this step appearing so straightforward is due to sensor calibration. By incurring that initial time and computational cost, the on-the-fly calculations are rendered trivial. If it was required to conduct more complex sensor fusion than inferring how far away a detected object is, a different approach may be more suitable. As mentioned before, there are more complex sensor fusion techniques, including ones that relate information between sensors with no regards to where in space they are located in relation to each other [21].

2.4 Object Recognition: Faces and Cars

Object recognition using Machine Learning (ML) methods in Computer Vision (CV) is an enormous field in its own right, and as such this project will mainly focus on its application potential rather than the theory behind the technology. Moreover, this project will be handed to two Master's students to continue working on, and one of them (Flora Qin) was tasked with developing car detection features using a trained neural network. As such, this paper will not concentrate on that section of the problem statement. It is worth noting that this separation of responsibilities requires the system to be modular so that the object recognition subsystem may be easily improved upon in the near future.

Vehicle detection in images and camera feeds is a well-developed field, and there are many solutions available involving different levels of complexity: from designing and training a neural network from scratch to using a library as a black box with no concern over how the detection is being performed. For reasons previously stated, this paper will take the latter approach, as a placeholder for the more advanced solution that will be implemented by Flora after the completion of this stage of the project.

OpenCV, a tool used often in real-time CV, allows for object detection using cascade classification, a lightweight image classifier characterised by having multiple stages of simple feature detection. It operates like a funnel: if an image passes the first classifier, it "cascades" into the next one, and so on; the image is discarded otherwise. If an image passes all the stages, then it is concluded that it contains the object the classifier was looking for [22]. The algorithm was developed with speed and low computational cost in mind, so it should be a suitable approach for a real-time system such as this one.

While originally developed to detect faces, cascade classification is a generic method that can be applied to any object detection needs. It unfortunately does require some previous training in order to be used, but there are numerous trained models online that can be used freely. For example, OpenCV can perform such a classification when given an `.xml` file containing the trained classifier.

Some Python libraries not only implement the algorithm, but also include the trained classifier within so the user can use it straight out of the box with no additional configuration or training; such is the case of Geitgey's `face_recognition` Python package [23].

It is important to mention that the above approach may not be an ideal long-term solution, and that training a new network from scratch with a different training set would most certainly be beneficial, as this particular library has been used by the author of this report in previous university projects with some undesirable results. In a past experiment, the algorithm repetitively and systematically was unable to distinguish between a group of men of south-east Asian origin. This finding was not in any way surprising, as facial recognition, especially when used for criminal and hiring purposes, has been heavily criticised for racial and/or gender bias [24].

2.5 ROS, IoT, and Vehicles

As a project requirement, Dr. Demetriou declared that this project's software infrastructure should be built around the Robot Operating System, ROS [25]. This was completely understandable, considering that there are numerous previous undertakings that use ROS as the framework to control and/or record readings from a vehicle [11][26]. After all, what is a smart car if not one large robot?

In most cases, these papers refer to work done on small vehicles, more commonly referred to as "rovers", where there is no room for a driver. These systems are excellent for prototyping sensor systems and easily evaluating them in a controlled environment that is not just a public road [26]. On the other hand, there exist more ambitious projects such as the aforementioned work or Jose et al. [11], where the ROS system directly interacts with a real car that has been rewired to allow for autonomous driving and integration with original and extra added sensors. Both of these approaches can attest to the suitability of ROS for our particular application.

2.6 Safety and Security Concerns

Even though it was not given as a requirement, it is imperative to discuss the safety of the infrastructure to be developed. Cars operate in a public setting, and, as such, interact with hundreds of other vehicles and pedestrians daily. This issue is not to be taken lightly, as driving-related incidents took the lives of almost 2,000 people in the UK during 2017 [27]. Given the social relevance of this matter, we shall delve into some of the safety and security concerns regarding the system.

Autonomous and semi-autonomous vehicles rely on their sensors as much as a human driver does on their senses; malicious attacks exploiting sensor weaknesses can be devastating and hard to avoid [28]. For example, tampering with a camera is as simple as blinding it with an infrared laser costing less than £1 [29]. It is equally feasible to inject fake data into the readings of a LiDAR by creating fake echoes that the sensor interprets as genuine [28]. While this kind of attack is time-sensitive and requires more hardware (around £40 worth), it is notably hard to account for without sacrificing latency and accuracy in the readings, as well as considerable computational power used for the necessary post-processing. These compromises for the sake of security and attack prevention are also incurred on cameras and other visual sensors, in addition to hardware related countermeasures such as sensor redundancy and lens changes.

Most papers describing possible attack patterns on sensors and autonomous vehicles tend to include suggested prevention mechanisms for such offensives, sometimes with great success at detecting and compensating for the damaged or injected data [30]. However, the most reliable ones tend to incur an extra computational time of 100ms, a harsh price to pay in a real-time application.

Just as the sensors are a point of failure, it is also possible to perform attacks on the system infrastructure itself, especially if there are exploitable wireless connections within the vehicle or bridging the on-board computer to a centralised server or command centre. In the case of the former, it would be possible to tamper with safety messages, traffic information, or even cause a Denial of Service (DoS) scenario with devastating consequences [29]. Cooperative driving frameworks are particularly vulnerable to such attacks due to their reliance on external information from other vehicles and/or centralised intelligence.

The former notwithstanding, it is still important to consider that this project is more of a proof-of-concept than a final product, so security will need to be compromised for the sake of rapid prototyping and ease of use.

Aside from deliberate attacks on a car's sensors or autonomous systems, there is an extra consideration to be taken into account, if only from a philosophical point of view: is autonomous driving as beneficial as it is made out to be? There is unfortunately not enough data to answer this question either way, but it would be hoped that eliminating human error from the equation can only improve safety on the road. It goes without saying that foregoing human error is only possible by incurring some amount of machine error, a compromise which will be seen at play in the years to come. Moreover, it is interesting to discover how human drivers react to partial or total vehicle automation, such as Terai's et al. study concerning driver behaviour in manual versus semi-automatic driving scenarios [31]. These researchers discovered that users showed less sensitivity to risk when using assisted driving, a conclusion that may not come off as a surprise after Uber's fatal accident involving one of their tester autonomous vehicles where it was found the driver had been watching Hulu on her phone for over 40 minutes at the time of the crash [32].

As with any state-of-the-art technologies, only time will tell its true effect on the roads in the years to come.

Chapter 3

Vehicle Visual and Distance Sensor Fusion and Real-Time Processing

3.1 Hardware

The project at hand consists of two interdependent components: hardware and software. The sensors and car mount comprise the hardware, and their design and development will be discussed in the following sections.

3.1.1 Sensors

The selection of sensors used in this project was predetermined by its supervisor Dr. Demetriou prior to project commencement.

For the LiDAR, he procured a Velodyne VLP-16, which has enviable specs: 16 IR laser/detector pairs that fire with a 18kHz frequency, resulting in 300,000 360° distance datapoints per second [12].

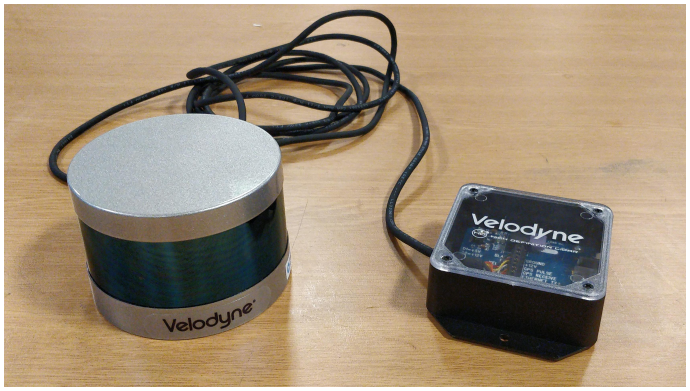


Figure 3.1: The Velodyne VLP-16 used in the project.

As for the four cameras, he determined that using smartphones would be most valuable for the project. Nowadays, phone cameras have incredible quality that can easily match handheld cameras, especially if extensive zoom capabilities and high quality close-up shots are not required [33]. A considerable advantage of the smartphones is all their extra functionality, mainly concerning connectivity; this has given phones the upper hand in the digital camera market to the point of almost eliminating compact camera sales [34]. If we are to integrate a camera into a real-time

system, having Bluetooth, Internet, and USB connectivity will be invaluable. How this connectivity is utilised in the project will be discussed in Section 3.3.

As for the phones, we will be using four Google Pixel 2 XL's. Its camera specifications leave nothing to be desired, with a rear camera resolution of 12.2MP, capable of recording 4K video at 30 fps [35]. With a 3520mAh battery and fast charging enabled, the phones should last a long time running on the car mount. Firmware-wise, they run the latest Android 9, which gives them access to the myriad of Play Store apps, as well as one being able to develop their own apps if the project so requires.

While able to record high definition video, the real-time constraint of the system meant that such capabilities could not be used, as the central compute node would need to process four camera streams simultaneously. Such a stream of information would also require high bandwidth to transfer from the phones to the on-board computer. Taking this into consideration, the camera resolution should be dialled down to ease the computational strain on the system.

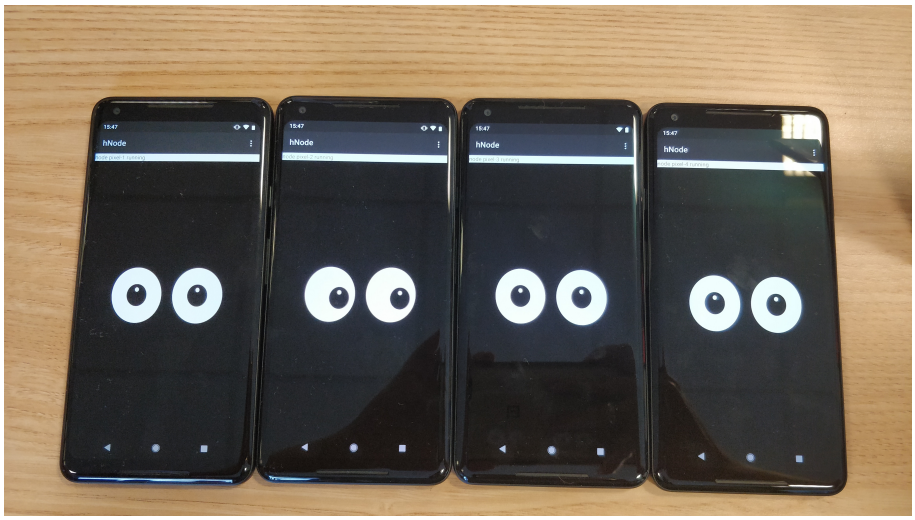


Figure 3.2: The four Pixel 2 XL phones used in the project.

3.1.2 Car Mount

The rig in use for this project was derived from Dr. Demetriou's CoDrive paper [4]. It features a symmetrical design: there is a disk that holds the four phones, evenly distributed along the edges, and the LiDAR in the centre, positioned slightly above the phones so they don't appear in its field of view. This disk is then held up by metal standoffs on top of some aluminium extrusions that are then secured onto the vehicle in question.

To facilitate the implementation, the order and location of each of the phones on the mount is fixed, so when one of them is addressed it is clear which direction it is facing, as seen in Figure 3.3.

In the beginning, only a single phone was used for initial prototyping, and a simple MDF disk was made to begin developing the system. The phone was held in place using a generic holder that could be screwed into the wood. As for the LiDAR, it was screwed onto a miniature tripod and placed on the centre of the disk. This was by no means safe to move around, but it was enough to start generating the relevant software before the rest of the sensors were acquired. This prototype is photographed in Figure 3.4.

After several attempts at contacting manufacturers to build the frame for the project, with quotations starting at £700, Dr. Demetriou resorted to building his own prototype, similar to the mount used in his CoDrive paper. The main difference between this prototype and the one developed by the author was the size: the supervisor's platform is approximately 50% larger. No stress analyses were conducted to determine whether a smaller or bigger size of mount would be better adapted to the use case, so working with a wider mount was not considered an issue by any means. Throughout the entirety of the project this rig has performed as expected. This final mount is shown in Figure 3.5.

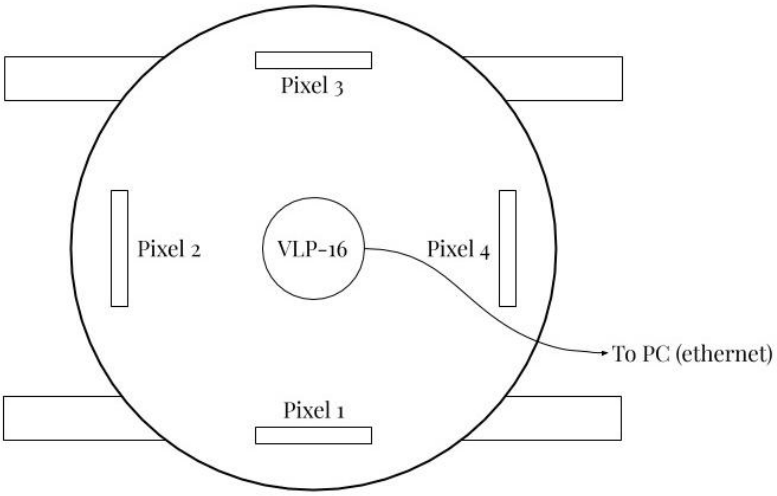


Figure 3.3: Top-down simplified drawing of the car mount.

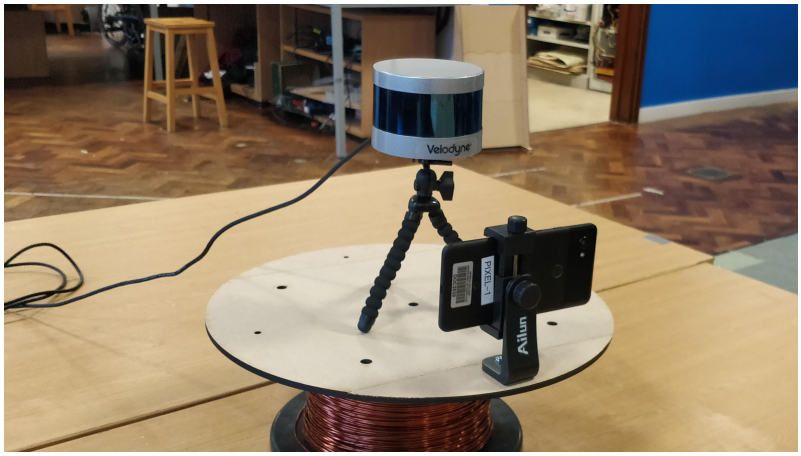


Figure 3.4: Initial prototype of the car mount.

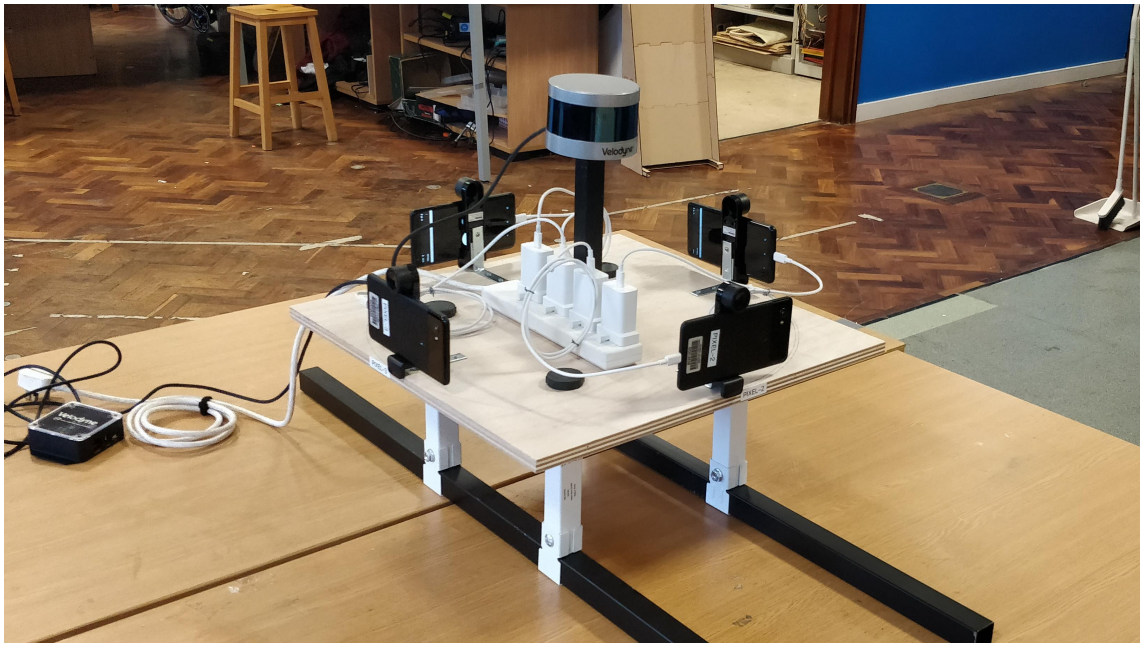


Figure 3.5: The final car mount built for the project by Dr. Demetriou.

3.2 ROS Infrastructure

To manage the sensor network and software infrastructure, the Robot Operating System (ROS) was chosen.

ROS is an open-source publisher/subscriber framework that allows for numerous decentralised modules to run in parallel and communicate with each other effectively [25]. Although it was conceived and developed for robotics applications, ROS is invaluable when working with a complex hardware system to interface with multiple sensors and have several programs running in parallel.

The essential unit of a ROS system is the "node", which acts as an individual program running on the machine. Nodes are compiled into packages, which can, in turn, depend on other packages. In order for nodes to communicate with each other, they use "topics". These are message buffers that any node may publish or subscribe to. Topics have a specific message type, and being subscribed to a topic means that a node can access all the messages being published to it, with the possibility of linking a callback function to execute upon reading a message. When nodes are running, some are publishing to topics, others are subscribed to topics, and some even do both.

ROS as an infrastructure brings numerous benefits. Among them are the following: being fully open-source; having widespread use resulting in plenty of useful libraries and packages being developed for it; and supporting multi-language systems, as nodes may be written in different languages (most commonly C++ and Python), but when running they may still communicate through topics without any issues. Furthermore, having to organise code in nodes which run in parallel results in systems which are extremely modular and decentralised, to which adding extra functionality requires no major restructuring or infrastructure changes at the design level.

ROS' most notable downsides are only being available on Linux (Ubuntu LTS 18.04 is used here), as well as incurring some amount of overhead when running and messaging through the topics. This is a meagre price to pay for the ease of implementation that ROS provides.

In Figure 3.6, a small section of the ROS system is depicted. In particular, the flow of data when running face detection, sensor fusion, and visualisation (in that order) is shown.

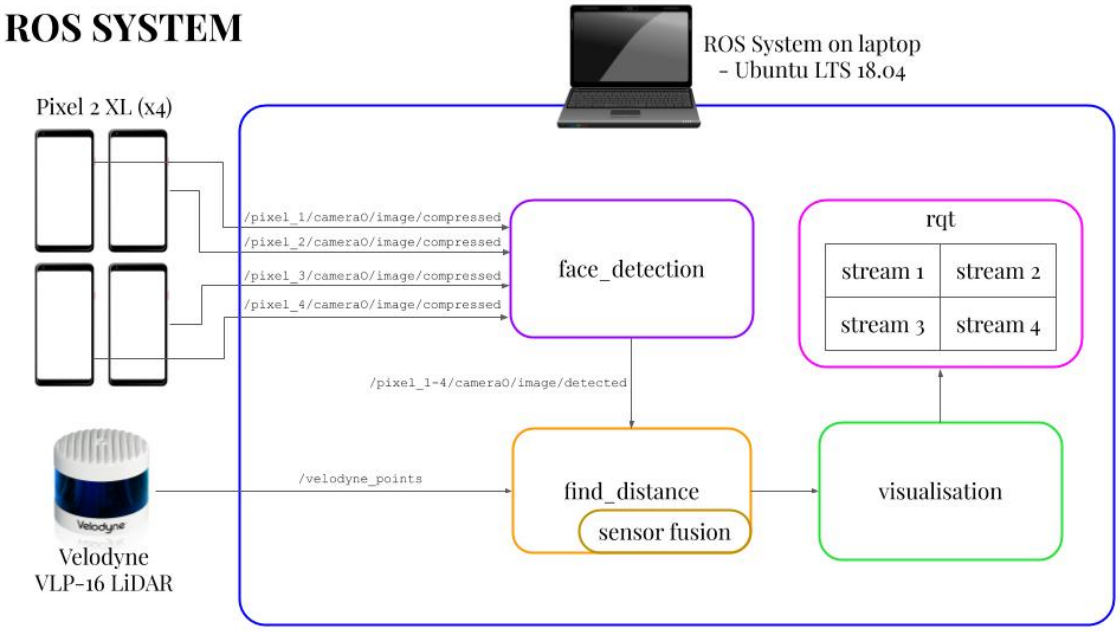


Figure 3.6: ROS system architecture for this project.

3.2.1 ROS Packages

The ROS code is contained within a `catkin` (ROS' build system) workspace directory, which manages the building of the packages. The `src` directory within the workspace holds all the folders (packages) of the system, both internal and external.

The full breakdown of the packages in the workspace folder is as follows, with external folders marked with "EXT":

`but_calibration_camera_velodyne` Package (EXT)

This is the external package containing the LiDAR calibration implementation developed in "Calibration of RGB Camera With Velodyne LiDAR" by Velas et al. [17]. It implements two types of calibration (coarse and fine) as well as a method to visualise the results of calibration through `rviz` (ROS visualization, a 3D sensor data and state visualiser).

As mentioned in the Related Works chapter, it was necessary to make some minor changes to this external library to adapt it to our particular needs. The changes include:

- Increasing the threshold for plane detection on the LiDAR pointcloud from `0.1` to `0.5` in the file `but_calibration_camera_velodyne/src/Calibration3DMarker.cpp`, due to too many planes being detected and obscuring the relevant one.
- Increasing the attempts at detecting circles on the LiDAR pointcloud from `64` to `2048` in the file `but_calibration_camera_velodyne/src/Calibration3DMarker.cpp`. This slowed the process considerably but the change intended to improve its reliability, and it resulted in less calibration failures overall.
- Increasing the tolerance for detecting circles on the LiDAR pointcloud from `0.03` (3cm) to `0.05` (5cm) in the file `but_calibration_camera_velodyne/src/Calibration3DMarker.cpp`, as calibration was failing too often at this stage.
- Decreasing the threshold for pointcloud refinement from `0.05` to `0.01` in the file `but_calibration_camera_velodyne/include/but_calibration_camera_velodyne/Calibration.h`.
- Adding the capability of calibrating for each of the individual phones by rotating the pointcloud to align its orientation with the phone in question. These changes were made in the file `but_calibration_camera_velodyne/src/calibration-node.cpp` and in all the `launch` files.
- Adding numerous debugging statements to identify where the calibration processes were failing whenever they did.

Coarse Calibration This is the faster, less accurate implementation of the calibration algorithm. If calibration fails, the application waits for 5s to try again, repeating this process until it succeeds or is terminated manually.

This node can be run with:

```
$ roslaunch but_calibration_camera_velodyne calibration_coarse.launch pixel:="PIXEL"
```

Where `PIXEL` is the phone (1-4) to perform calibration for.

Fine Calibration To provide a more refined calibration outcome, this node performs the coarse calibration firstly and subsequently, upon succeeding, executes a refinement procedure on the calculated 6DoF.

This node can be run with:

```
$ roslaunch but_calibration_camera_velodyne calibration_fine.launch pixel:="PIXEL"
```

Where `PIXEL` is the phone (1-4) to perform calibration for.

Colouring Visualisation In order to test the calibration results, one can run this node. It takes the LiDAR pointcloud and colours each of the points in the shade of the object the cameras are seeing in that location after performing sensor fusion with the given 6DoF (as obtained using one of the above calibration mechanisms).

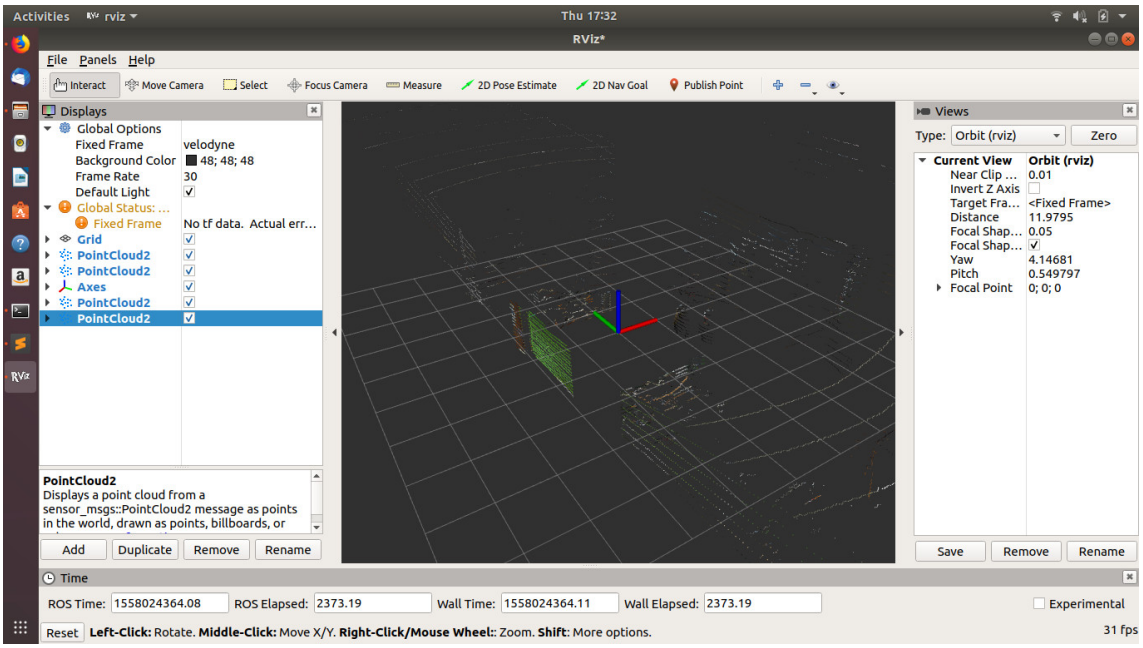


Figure 3.7: The visual output of running the colouring visualisation implemented in the `but_calibration_camera_velodyne` [18] package.

This node can be run with:

```
$ roslaunch but_calibration_camera_velodyne coloring.launch
```

`face_detection` Package

This package contains the nodes that perform facial detection on the four camera streams. After computing where the faces are located on each frame, the original frame and these face bounding boxes are published to another topic for the visualisation node to handle.

The reasons why this project included a face detection package will be discussed in Section 3.6.

Using Python The Python implementation of face detection uses Geitgey's `face_detection` library [36], downloaded using the Python 3 package manager `pip3`. The library allows for facial recognition on an individual basis, but in this implementation it is solely used to detect any faces in the field of view.

To avoid high latency, all the incoming images are reduced in size by half before processing. Even with the reduced resolution, facial detection is still successful overall and the system is able to cope with the four camera streams simultaneously.

```
$ rosrun face_detection face_detection_py.py
```

`car_detection` Package

As above, this package contains the nodes that perform car detection on the four camera streams. After computing where the cars are located on each frame, the original frame and the computed bounding boxes are published to another topic for the visualisation node to handle.

Using Python This Python implementation uses OpenCV's cascade classifier, giving it a trained classifier for vehicle detection in `.xml` format.

Much like in the `face_detection` Package, all incoming images are reduced by half before processing to speed up the recognition times.

```
$ rosrun car_detection car_detection_py.py
```

`phone_streams` Package

This package was created to source the nodes that pre-process the frames coming from the phones before they are used in the main application.

Camera Streams Converter This converter consists of a node that extracts the compressed image generated by the phones and re-publishes it in raw format. It is also responsible for broadcasting the camera information required for calibration against the LiDAR.

```
$ rosrun phone_streams phone_streams_publisher.py
```

Frames Counter In order to diagnose the fluctuating framerates from the camera feeds discussed in Section 3.3, a frames counter node was written. With this counter, all frames arriving from each of the four phones during the span of one second are counted, and the results returned on the terminal window. It runs indefinitely until it is terminated manually.

The node is executed with the command:

```
$ rosrun phone_streams phone_frames_counter.py
```

`velodyne` Package (EXT)

This external package contains the drivers for the VLP-16 Velodyne LiDAR, as provided by the manufacturer themselves.

In order to publish the pointcloud data on the topic `/velodyne_points`, the following command should be run:

```
$ roslaunch velodyne_pointcloud VLP16_points.launch
```

`vision_opencv` Package (EXT)

This OpenCV package manages both image manipulation and visualisation. No nodes are executed directly from the package, but its functions are called elsewhere in the project's code.

`visualization` package

This package contains the nodes that are responsible for sensor fusion and visualisation of its results.

Multi-feed Visualisation Containing the sensor fusion code, this node incorporates the four camera streams (after face or car detection has been performed on them) and extrapolates the positions of the detected objects to extract their distance information from the LiDAR pointclouds. These findings are visualised using ROS' graphic user interface (GUI) framework `rqt`.

```
$ rosrun visualization video_visualization
```

Pointcloud Merger The results of the four phones, when using the colouring node of the `but_calibration_camera_velodyne` package, are computed and published to topics separately. In order to visualise them all at the same time with `rviz`, a pointcloud merger node was written.

```
$ rosrun visualization join_pointclouds
```

3.3 Sensor Network

Up until now, the overall ROS framework developed has been explained, so the next step is to address how the system is able to interact with the five sensors to obtain their readings.

Provided that the mount with the sensors will be placed atop a moving vehicle, it would be best to avoid wiring as much as possible. Due to the real-time nature of the project, it was necessary to establish some low-latency wireless communication between the sensors and the on-board compute node. In the case of the LiDAR, such a requirement was not deemed feasible, as the sensor came with an Ethernet connection, and installing a router on the mount was not considered ideal. The cabling that comes with the LiDAR is long and weather-proof, so it would not be too difficult to route the wire through a slightly opened car window in order to connect it to the laptop within.

Having connected the LiDAR with the Ethernet cable, using the given Velodyne ROS driver results in the 360° pointcloud data being published in the ROS topic `/velodyne_points`, so that it is accessible to the other ROS packages and nodes running on the computer.

The phones were, however, a different issue. Having four of them, any wiring required for the functionality of a single phone is quadrupled for the overall system, increasing set-up complexity and general maintenance.

When attempting to connect two or more internet-enabled devices, an Internet of Things (IoT) approach seems most appropriate. This can have several drawbacks, however, namely the necessary continuous internet connection for all the devices. If the car is to be on the road, relying on wireless internet being readily available could become a burden, especially outside of cities. Alternatively, using a technology such as Bluetooth might also bring issues due to the devices being able to hold just one active connection at a time.

Because this is not a newly-discovered issue, particularly in the world of robotics, there exists a ready-made solution for this case that is solved by *Husarnet*, an IoT-turned-P2P network manager developed by the Polish robotics company *Husarion*. The company's engineers were concerned about the security and latency issues of established IoT providers which led them to devise a solution: rather than following the conventional communication flow of device1-server-device2,

they implemented direct peer-to-peer data transfers. The need to bypass IoT servers came from the aforementioned security concern (the server provider could peruse one's sent data at will), as well as the irony of requiring a connection to a remote server in order to establish communications between devices that were centimetres apart [37]. In short, what *Husarnet* enables is for two devices to connect directly to each other as if they were on the same network, even when they are not. Accordingly, the devices can talk to each other using their hostnames instead of the usual IP address.

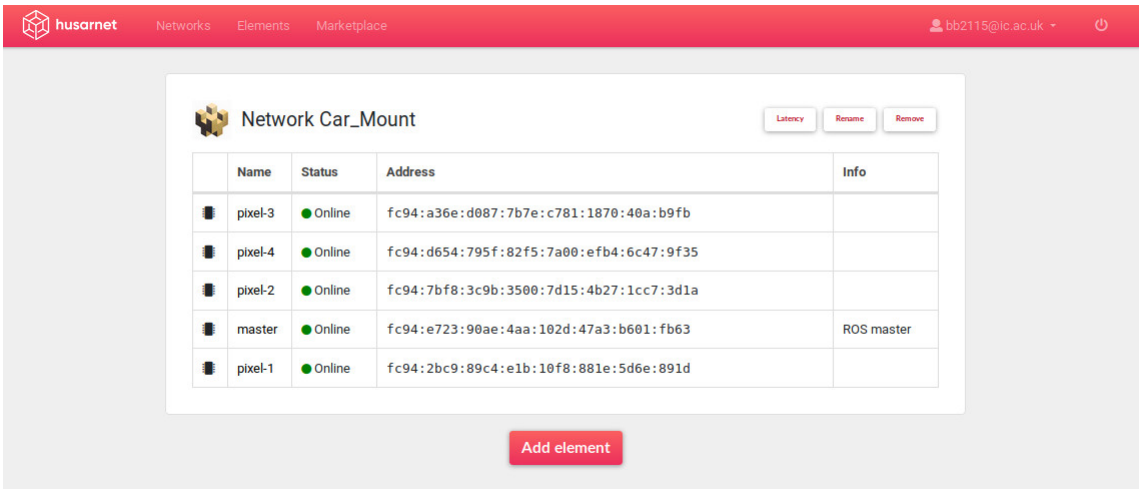


Figure 3.8: The four phones (hostnames: pixel-1 to pixel-4) and laptop (hostname: master) are registered on the same Husarnet network so that they may communicate.

While *Husarnet* was designed to connect different sensors to a central unit, the engineers at *Husarion* broadened the definition of a "sensor" by releasing the Android app named *hNode* [38]. This program allows the phone to connect to a central node in peer-to-peer mode so that such a central node captures the readings of the phone's sensors: accelerometer, gyroscope, light sensor, pressure sensor, and, most importantly, the cameras. To top it all off, *hNode* essentially behaves as a ROS node, publishing this extensive sensor information on ROS topics, eliminating the need for writing an Android app from scratch. Because the computer and all sensors are on the same network thanks to *Husarnet*, the central compute node has access to all those topics directly and wirelessly, which fully fulfils the connection needs.

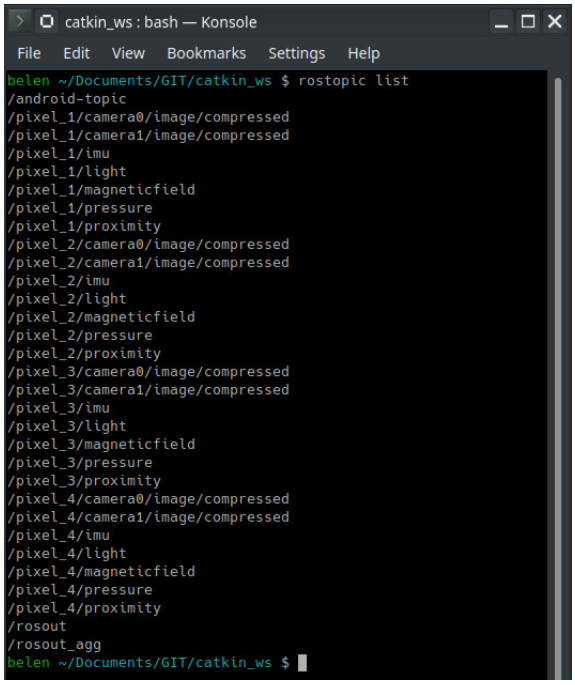


Figure 3.9: The phones' sensor data being published into ROS topics by the hNode app.

By using these tools, accessing the cameras on all four phones became trivial, without the need to write an Android app or set up a wireless Local Area Network (LAN) with dedicated IPs. As the camera feed was being published to ROS topics, any ROS node on the computer could listen to and use the video frames for post-processing whenever necessary. For example, the object detection nodes (face or car) would subscribe to the four camera streams - `/pixel_x/camera0/image/compressed`, where `x` is the phone to be addressed (1-4).

Unfortunately, the ease of use comes at a price: there is ultimately no control over the framerate or resolution of the incoming images from the phones. The images are published as a `sensor_msgs/CompressedImage` of resolution 640x480 pixels, which is an acceptable downscaling for this application; processing four streams of 4K or 1080p quality in real-time is not affordable to perform on a standard laptop.

The framerate issue, however, is not so favourable: having such an erratic flow of images greatly decreased sensor fusion accuracy. This was due to the camera and LiDAR inputs not aligning in a consistent manner, so sensor fusion was being performed on two inputs that had been received at slightly different times. There was no time to investigate or mitigate this issue, but it would be an interesting undertaking in the future.

A minor problem of less importance was the high power consumption of the app, with the phones lasting for 6-8 hours but rarely more. Taking into consideration that the camera was continuously running for the entirety of the application, this shortened lifespan was no surprise, and it should just be kept in mind when taking the system out on the road for testing.

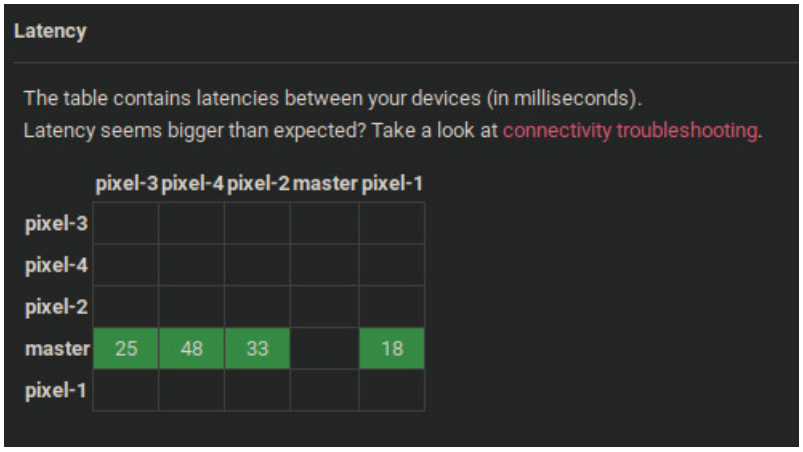


Figure 3.10: The low latency between the phones and the laptops reported by the Husarnet network.

```
[INFO] [1560871520.055518]: Framerates for second 1560871521:
[INFO] [1560871520.059975]:   pixel_1: 1 fps
[INFO] [1560871520.063116]:   pixel_2: 2 fps
[INFO] [1560871520.066885]:   pixel_3: 1 fps
[INFO] [1560871520.072671]:   pixel_4: 1 fps
[INFO] [1560871520.076433]:
[INFO] [1560871521.488202]:
[INFO] [1560871521.491679]: Framerates for second 1560871522:
[INFO] [1560871521.495184]:   pixel_1: 21 fps
[INFO] [1560871521.498316]:   pixel_2: 4 fps
[INFO] [1560871521.505341]:   pixel_3: 0 fps
[INFO] [1560871521.510494]:   pixel_4: 25 fps
[INFO] [1560871521.513591]:
[INFO] [1560871522.517093]:
[INFO] [1560871522.520281]: Framerates for second 1560871523:
[INFO] [1560871522.523865]:   pixel_1: 15 fps
[INFO] [1560871522.527458]:   pixel_2: 3 fps
[INFO] [1560871522.533441]:   pixel_3: 4 fps
[INFO] [1560871522.538033]:   pixel_4: 14 fps
[INFO] [1560871522.542618]:
[INFO] [1560871523.066452]:
[INFO] [1560871523.067805]: Framerates for second 1560871524:
[INFO] [1560871523.069011]:   pixel_1: 9 fps
[INFO] [1560871523.070240]:   pixel_2: 4 fps
[INFO] [1560871523.071603]:   pixel_3: 6 fps
[INFO] [1560871523.072686]:   pixel_4: 0 fps
[INFO] [1560871523.074030]:
```

Figure 3.11: Output from the developed frame counter node.

3.4 Sensor Calibration

In the Related Works Section 2.2, it was determined that the project would use Velas' et al. calibration method [17]. Using their own ROS package [18], and implementing the changes explained in Section 3.2.1, performing sensor fusion became a simpler ordeal. In order to do so, an A1 sized card was cut to feature the four squares necessary for the algorithm, and this card was then suspended in front of the phone that was to be calibrated (Figure 3.12). When moving on to calibrate the next phone, it was necessary to rotate the mount so it could, in turn, face the marker, but no extra code changes were required.

Once the calibration on all four phones was performed, the four resulting `6DoF` were hard-coded into the visualisation to be used for sensor fusion.

A necessary prerequisite to camera-to-LiDAR calibration was calibrating the camera itself. This process entails calculating a camera's projection matrix, of the form shown in Figure 3.14. This matrix represents the translation of the 3D points in the environment into the 2D image that a pinhole camera generates.

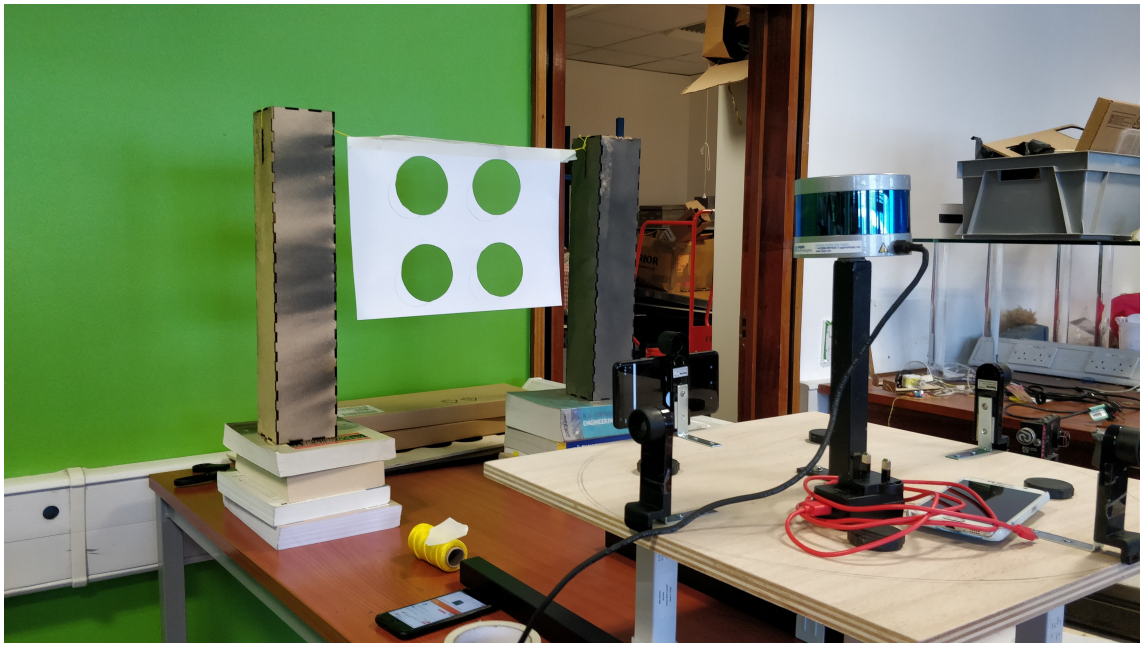
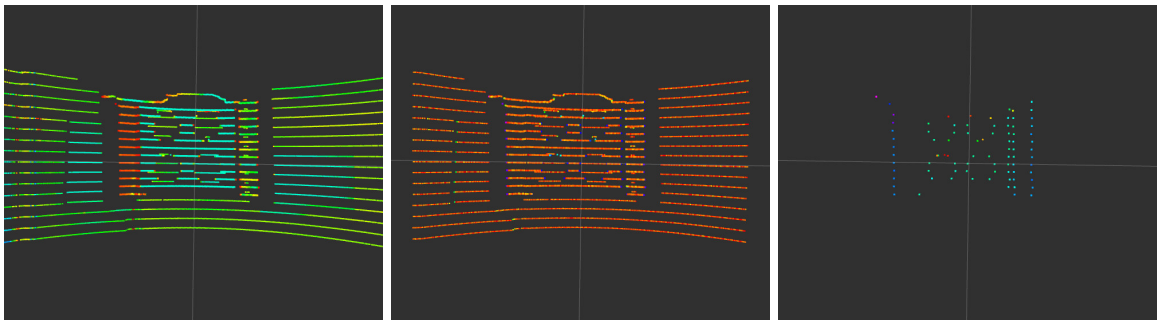


Figure 3.12: Performing camera-to-LiDAR calibration using the Velas et al. method [17].



(a) The original LiDAR pointcloud. (b) After plane detection. (c) After circle detection.

Figure 3.13: The calibration process in action.

To calibrate the phone cameras, the `MATLAB` function `cameraCalibrator` was employed [39], which used a 8x6 chequerboard with 24mm-sized squares, photographed 40 times from different angles and perspectives. This result returns the focal length and the principal points of the matrix (f and o_x and o_y in the projection matrix of Figure 3.14).

$$\mathcal{P} = \begin{pmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.14: A pinhole camera's projection matrix.

3.5 Sensor Fusion

As explained in the Related Works Section 2.3, the decision to incur a set-up cost for sensor calibration in order to achieve low latency sensor fusion at runtime was considered a valid approach. As a result, the implementation of sensor fusion in the project code takes at most two lines of code.

The first is transforming the LiDAR pointcloud to match the view of the camera in question with:

```
Velodyne::Velodyne transformed = pointcloud.transform(6DoF);
```

Where `pointcloud` is the original 360° distance data, and `6DoF` is the transformation matrix found during calibration for that particular camera.

After the pointcloud has been transformed, it is projected onto the frame in question in order to relate which pixels in the image correspond to which distance readings in the pointcloud.

```
transformed.project(&frame, camera_projection_matrix, &result);
```

The theory behind these transformations was discussed in detail in the Related Works Section 2.3.

Given the implementation is so concise, it was decided it would be best to include it within the `visualization` Package.

3.6 Face Detection

By this point in the report, the question of why a face detection node has been included in a car recognition system may have arisen. The answer is purely practical: for testing and demonstration purposes. Placing the car mount on a car to then take to the streets to test the system is a time consuming endeavour, which would also require the involvement of third parties, namely somebody who has a readily available car and holds a driver's license. The author of this report possesses neither of these.

Instead, if car detection was replaced with face tracking, the code may be tested in the lab where it was being developed by using one's own face. This approach would be sufficient to evaluate the sensor calibration and fusion performed, as measuring the distance of a face to the LiDAR and comparing that to the system's reported distance is a feasible and repeatable experiment. This is, in fact, the method devised for the project evaluation discussed in Chapter 4.

As pointed out in the Related Works Section 2.4, using Geitgey's `face_recognition` Python package delivered the speed needed in this real-time system: detecting faces on a single frame took just 0.055s on average. For the total of the four phones, this still only amounts to 0.22s, not taking into account any possible parallelisation. Bearing in mind this is being performed on a standard laptop, being able to run the complete system at around 4-5fps should be satisfactory as a proof-of-concept.

These fast recognition speeds were not only due to the inherent design of cascade classifiers, some extra measures were taken in the code to improve performance. The most effective was to reduce the resolution of all incoming frames by half before processing, but still use the original image only when visualising the results. No apparent reduction in accuracy was experienced when implementing such a change. The code also includes the option to only perform detection every x number of frames, but this corner-cutting was more visually evident in the results and is therefore not recommended.

3.7 Car Detection

The face and car recognition systems on this project were constructed in a similar manner: using external libraries as a black box, with the purpose of acting as placeholders for a better, in-house implementation that will be developed in the future.

Having experienced the performance benefits of cascade classifiers in `face_detection`, written prior, it was believed that following the same approach for car detection would return favourable results. As such, the car detection implementation of this project is done using OpenCV's cascade classifier code, which take as an input a trained classifier in `.xml` format. A suitable model was found online that had been extensively trained by its maker [40].

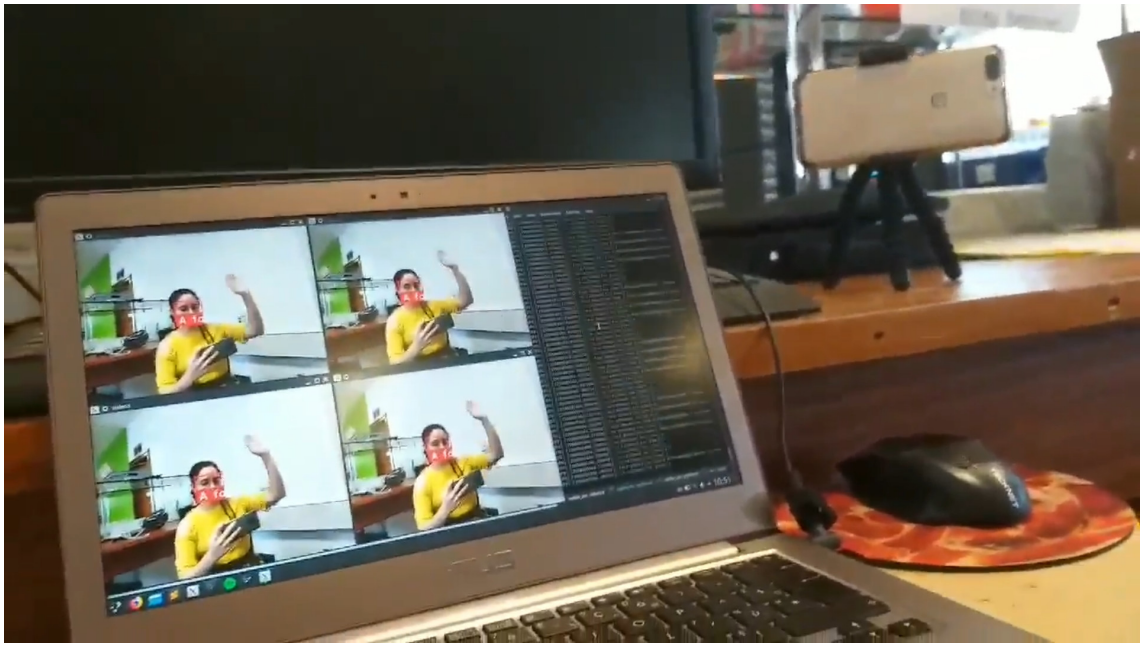


Figure 3.15: Implementing the facial recognition node on a single camera stream, quadrupled so as to test latency.

3.8 Visualisation

A crucial component of the project was visualising the computed results in real-time. In order to do so, two approaches were tested: using native ROS tools, and using OpenCV. Both methods are language-agnostic and could be used interchangeably.

When using OpenCV, it is possible to create a visual window to display images. By continuously overwriting the image in the window with the next frame, it appears as if a video is being played. The main advantages of using such a mechanism is that one may move the window around the screen at will, as well as change their size if needed.

On the other hand, ROS has some internal tools for data visualisation. For example, `rviz` has been used extensively during the development of this project to view the 3D 360° data the LiDAR was providing on the ROS topic `/velodyne_points`. This tool may be run by using:

```
$ rosrun rviz rviz -f velodyne
```

For a more generic visualisation and monitoring of ROS topics, `rqt` is available. The tool `rqt` is a Qt-based framework for GUI development for ROS. It allows the user to view any ROS topic within its window: video, images, text, etc. While it requires more set-up time than using OpenCV, `rqt` is extremely powerful, and it even allows for controls to be added so the user may send inputs to the ROS system at run-time. This level of versatility made the tool a more ideal candidate for the application at hand.

This tool can be run using:

```
$ rqt
```

At the time of writing, most of the visualisation is using OpenCV, as not enough time was allocated to designing a GUI in `rqt` in advance. This will become one of the author's main focuses before the demonstration of the project in a week's time.

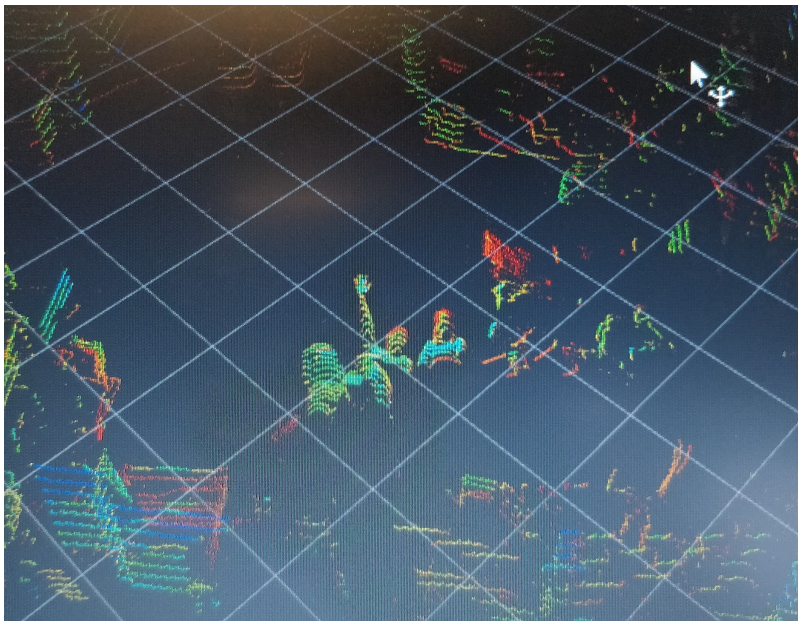


Figure 3.16: Viewing ourselves on the pointcloud data through ROS' `rviz`.

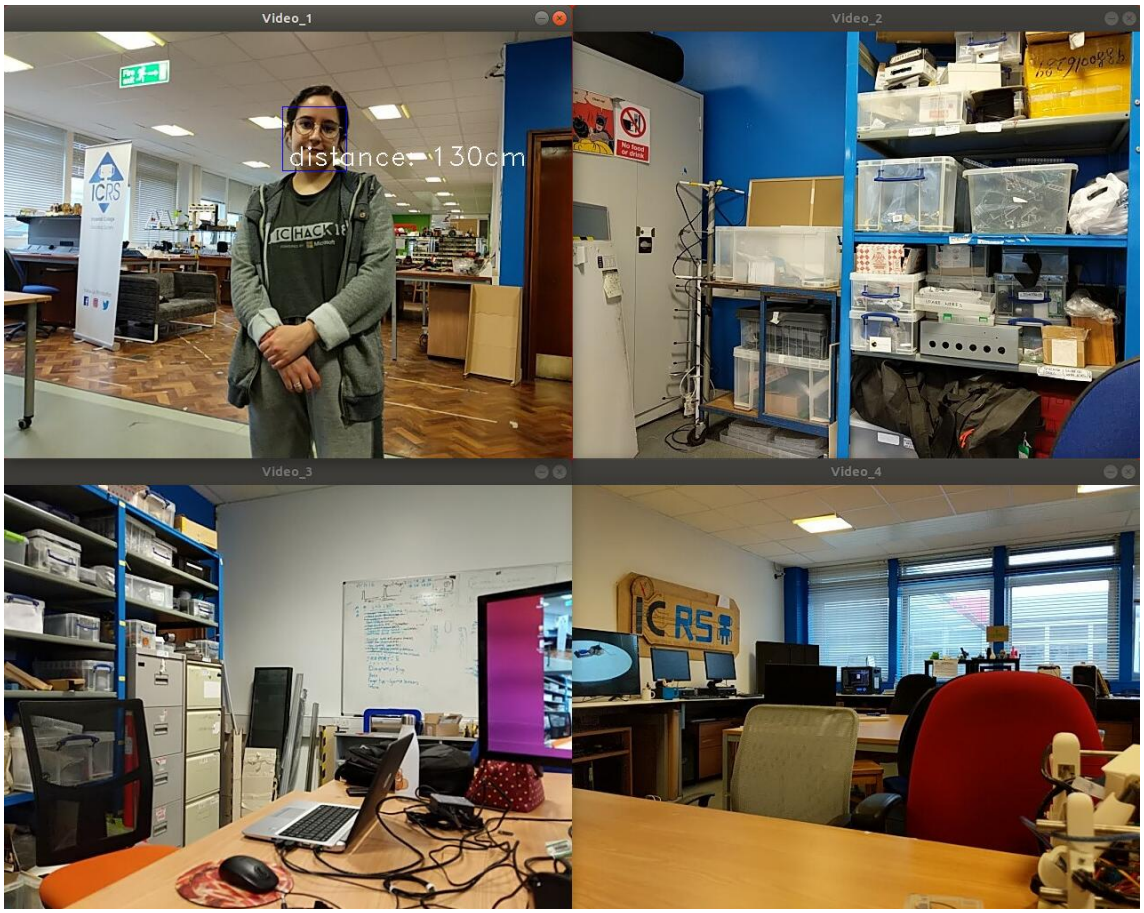


Figure 3.17: The results of visualising all four camera streams using OpenCV.

Chapter 4

Evaluation

As indicated in Chapter 3, the project is comprised of hardware and software components. It is appropriate to study these components separately using different metrics and methods.

4.1 Hardware Evaluation

Before the software implementation is reviewed, the suitability and performance of the sensors and mount used for the project shall be evaluated.

The Velodyne VLP-16 LiDAR performs wonderfully. It provides a constant influx of distance information that requires no preprocessing to be used in the code, and the packages provided to operate it with ROS present no issues whatsoever. The quality of its operation is definitely up to a piece of equipment worth £4,000.

While upgrading to the more accurate VLP-32 would clearly improve the robustness of the system, it is not believed that this sensor version is the current bottleneck. In fact, making the system handle twice the amount of incoming datapoints from the LiDAR may increase latency overall. In summary, the sensor used is considered ideal for the project.

With regards to the smartphones, they unfortunately raised several issues during the developmentstage; the most remarkable one being the erratic framerates of the incoming camera streams. This is a problem that could not be investigated further due to time constraints. It is believed that the most likely cause of such anomalous behaviour lies on the app used to send the images from the phones to the ROS system, hNode, and not on the phones' specifications themselves. Nevertheless, this problem did not ever cause the ROS nodes to crash or behave erratically, and at worst the visualisation occasionally stuttered. Unfortunately this is not a mistake a driving vehicle can afford to make.

As hypothesised in the Related Works Section 2.1, relying on just four visual sensors greatly impeded having any redundancy in the field of view. None of the cameras overlapped, resulting in a considerable blindspot between them, which is measured and depicted in Figure 4.1.

Having an approximate 70°-wide camera field of view, it becomes mathematically impossible to cover the full 360° space placing only four phones. This limitation was predicted at the design stage, and the infrastructure was desgined to be simple to upgrade and expand with more sensors. This weakness was thus well noted but no immediate actions were taken to minimise it.

The car mount where all the sensors are held to is the last hardware component left to be reviewed. Unfortunately, there is insufficient data to determine whether the structure is sturdy and weatherproof enough to work well atop a moving car, as the time constraints of the project impeded the undertaking of on-the-road tests.

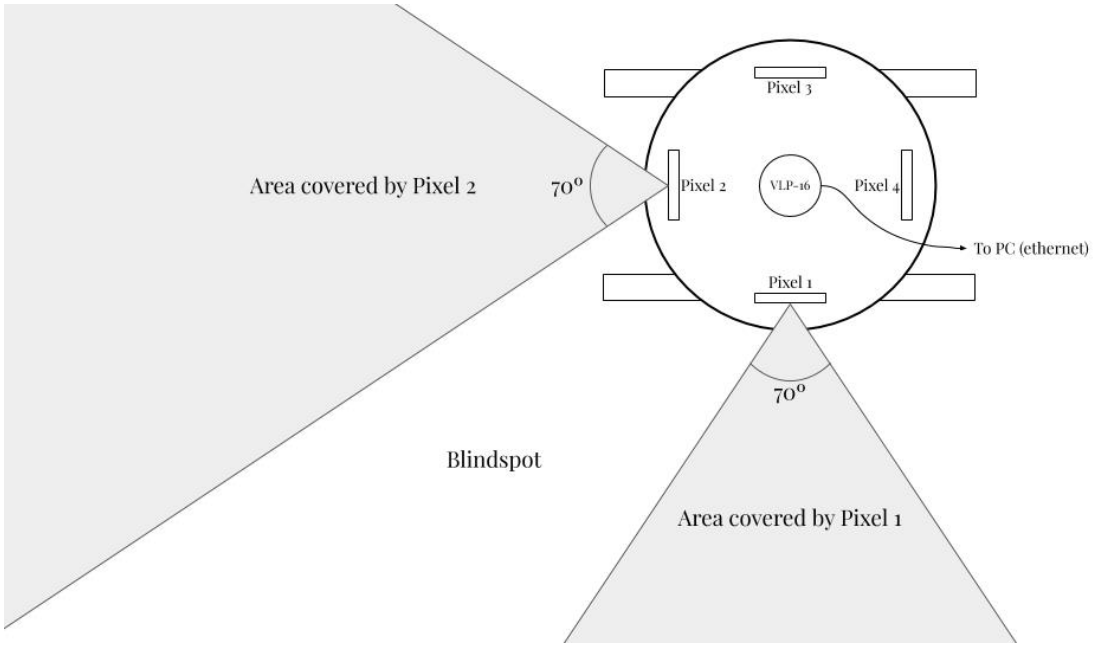


Figure 4.1: Diagram representing the mount's blindspot between each pair of phones.

4.2 Software Evaluation

Having no possibility to test the system on a real vehicle being driven around also had a consequence on the software methods employed. It became crucial to find an alternative method to car detection for testing accuracy, and the focus was then placed on face detection.

The supervisor of the project and the author came to an agreement that evaluating the system by using facial recognition instead of car detection made no difference; the only performance-affecting disparities between both methods would be the respective accuracy of the detection libraries. Given that the recognition metrics are outside this project's control (they are external sources), they may be used interchangeably.

Evaluating the system using facial detection was considerably simpler than having to monitor results on a moving car, and could be trivially performed indoors.

The tests were conducted with a person sitting on a wheeled chair and being placed within the field of view of one of the four cameras. The visualisation output reported that the face was at a certain distance away from the mount, and this reported distance was compared to the real distance, as recorded with a measuring tape extended from the subject's face to the LiDAR. The results of these experiments are compiled in Figure 4.3.

An immediate analysis of these tests shows rather positive results, with most datapoints landing very close to, if not directly on top of, the line representing `y = x`. However, as the detected face moved further away from the mount, some anomalies were observed: the reported distance value was magnitudes larger than the real one. An explanation to this disparity was quickly identified: the higher distance readings corresponded to the wall behind the subject. These outliers caused the system acouracy to be 79.77%.

In those cases, it was concluded that, while the face was being detected in the correct place, the LiDAR distance reading chosen to depict that position was incorrect. In other words, the sensor calibration's transformation matrix was slightly off. Considering a human face is approximately 20x20cm, the error meant that the sensor fusion was inaccurate by approximately ±15cm at larger distances.

(a) The subject posing normally.

(b) The subject employing the card method to mitigate the data anomalies.

Figure 4.2: Some tests being performed on the system using face detection.

To test this hypothesis, the subject was asked to hold an A1 sized piece of card behind the head, as seen in Figure 4.2. By artificially increasing the area of the head, the calibration error was offset, and the anomalous results were eliminated, as demonstrated in Figure 4.4. After making this improvement, accuracy rose to over 95.35%, a notable gain.

A proper solution for this problem would consist of improving the calibration method so that the projection matrix provides accurate results even at large distances. It is worth noting, however, that a vehicle is of a considerable larger size than a human head, implying that the error is less likely to affect the system when car is used instead of facial detection.

A final issue stemming from the use of facial detection was the maximum distance at which the algorithm stopped detecting faces. In this research, such distance was found to be 320cm, even in the case of processing the frames at full resolution instead of halving it to improve latency. This limitation was attributed to the external library and not to the system here developed.

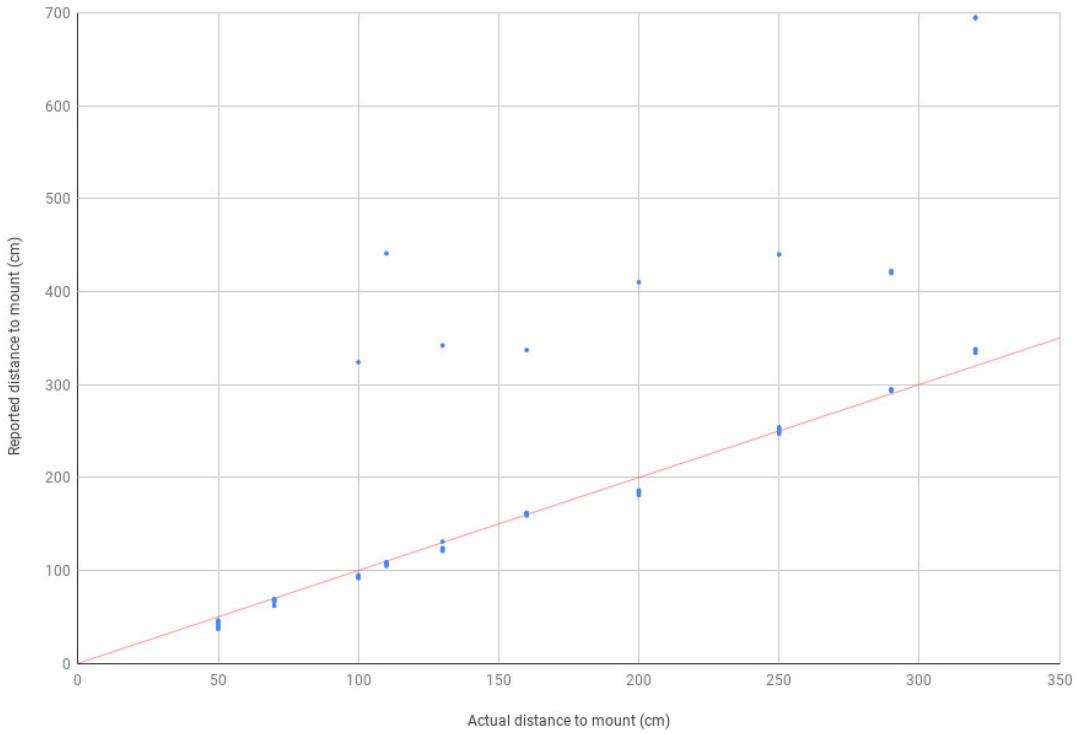


Figure 4.3: The reported distance of the faces detected plotted against their real distance to the mount. The red line shows where `y = x`.

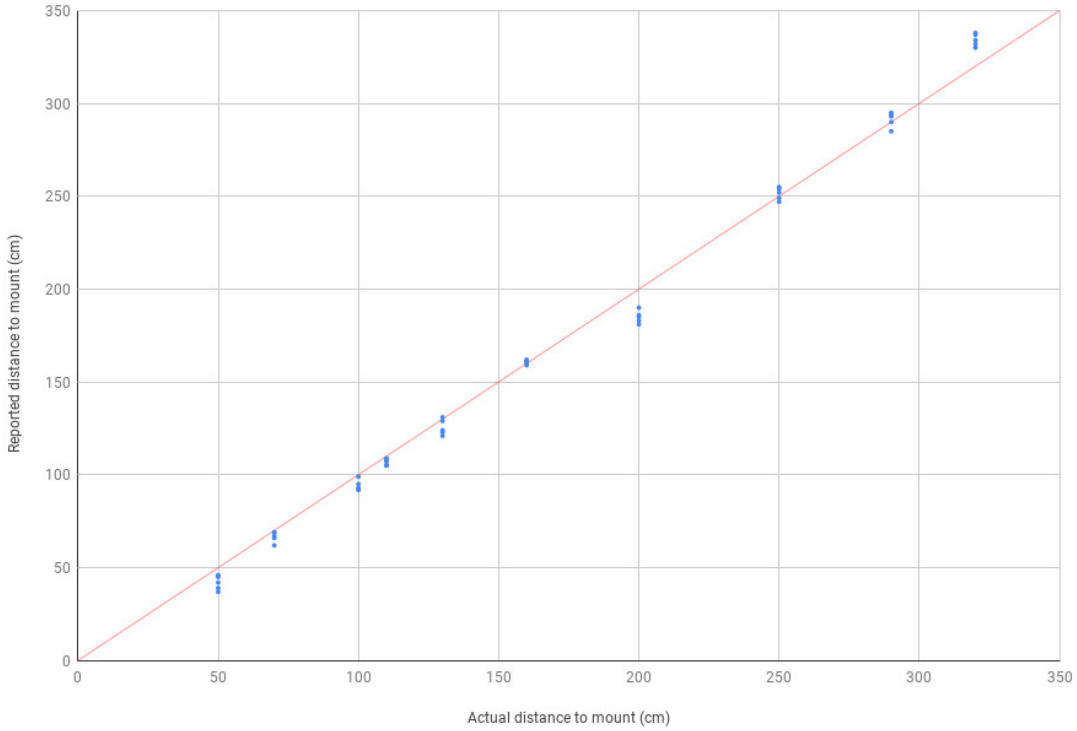


Figure 4.4: The improved results after using the card method.

Chapter 5

Conclusion

When comparing the initial objectives to what has been achieved, it would be reasonable to judge that the project has met its requirements. The results obtained, however, correspond to a proof-of-concept implementation; the work hereby undertaken is by no means a robust, polished product. This does not mean that the results obtained should be ignored; after all, the system achieved an accuracy of 90% on average during testing, a reliable signifier that it was a step in the right direction.

As evidenced by the Related Works Chapter, this project follows in the footsteps of many others centred around autonomous vehicles, their future and their potential. It is an increasingly necessary area of research, as the advancements made by private enterprises show no sign of stopping, and the author of this report is thankful for the opportunity to contribute to the discourse.

One of the most valuable lessons learnt when working with real-time systems was that doing the initial offline work pays off in the long term. This precept was most clearly seen in the implementation of sensor calibration and fusion. By spending time and resources to calibrate the sensors in advance, sensor fusion became a three-line ordeal at runtime. A similar effect was experienced with regards to visualisation: taking the time to design a GUI with `rqt` resulted in OpenCV monopolising less resources at runtime. These optimisations were particularly important considering the entire system was running on an ordinary laptop with no impressive hardware.

Perhaps what was most unexpected in the development of the system was the degree to which facial recognition was used over car detection. While this change was only intended to aid in the initial progress and evaluation, it ended up becoming a fundamental component at all stages of the implementation. Thanks to the ROS framework, changing the type of object recognition in use is uncomplicated, and it may be one of the most direct ways of improving the results provided.

While it would have been enriching to dive deeper into the world of object detection and classifiers, particularly in a real-time scenario, the project scope had to be limited somewhere. Knowing in advance that another student would be focusing on that area of the implementation allowed the rest of the work to flourish more in return.

5.1 Future Work

A considerable amount of possible improvements has been suggested throughout the report, and they will be further expanded upon in this section. These enhancements could be undertaken by the Master's students who are taking over the project after this progress report.

Starting with the sensors, a relevant upgrade would consist of solving the fluctuating framerates issue. As an example, the usage of Husarnet could be replaced by having all the phones on the same wireless LAN, hotspotted by the computer or an on-board router. This could require the building of an Android app similar to hNode in order to have the sensor data published into

ROS topics, but it would provide the user with considerably more control over the transmission of images (resolution and rate), as well as less reliance on third party software that, unlike most of this project, is not open-source. Designing such an app is an undertaking in its own right, but it is deemed to be a worthwhile investment.

The car mount, designed and developed by Dr. Demetriou, has not been tested on the road thus far, which makes it unclear at this point finding out how to improve the design to adapt it to such a use case. Considering that the main platform is made of thick plywood, some amount of weatherproofing could prevent future mishaps and facilitate ongoing maintenance. Changing the material to acrylic or some other type of hard plastic may help in this regard.

While deciding to use phones as cameras posed numerous advantages that have been already discussed in Section 3.1.1, it would be worthwhile investigating whether the employment of dedicated hardware such as a Pixy2 [41] or similar robot vision solutions could benefit the system's performance overall.

As often mentioned in this report, the most visually-evident modification that could be made to the codebase is the design and training of some CV system (such as a neural network or a cascade classifier) from scratch to more accurately detect the objects in question, be it cars or faces. Ideally, the training data for this algorithm should come from images taken with the mount being used on a circulating vehicle. Such an operation would prepare the network to detect cars in a similar environment as would be encountered during real world operation.

Another area deserving deeper attention is the investigation of other CV methods for object detection, such as YOLO (You Only Look Once), a real-time object detection package built specifically for ROS [42]. Unfortunately, this method is computationally expensive at run-time, therefore requiring a significant laptop upgrade in order to cope with four or more camera streams simultaneously. YOLO can be parallelised with a GPU, so running the system on a computer that is equipped with one could propel the results to another level of speed, framerate, and accuracy.

In summary, working further on the robustness of the project, by improving its hardware and software and conducting more extensive testing, will continue to progress in ensuring that this pilot implementation becomes a solidly packaged open-source solution in the extensive world of smart vehicles.

Bibliography

[1] T. Parry and D. Stillwell, "Vehicle Licensing Statistics: Annual 2018," tech. rep., HM Department for Transport, 2019. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/800502/vehicle-licensing-statistics-2018. Last accessed on 31/05/2019.

[2] Tesla, "Tesla Model 3." https://www.tesla.com/model3. Last accessed on 07/06/2019.

[3] The Automobile Association PLC, "Driven to collision: common causes of road accidents," *The AA*, May 2018. https://www.theaa.com/car-insurance/advice/road-accident-car-damage. Last accessed on 14/06/2019.

[4] S. Demetriou, P. Jain, and K. Kim, "Codrive: Improving automobile positioning via collaborative driving," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pp. 72–80, 2018. https://ieeexplore.ieee.org/document/8486281.

[5] SAE International, "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles," tech. rep., SAE International, 2018. https://saemobilus.sae.org/content/j3016_201806. Last accessed on 07/06/2019.

[6] J. M. Vincent, "Cars that are almost self-driving," *U.S. News*, Oct 2018. https://cars.usnews.com/cars-trucks/cars-that-are-almost-self-driving. Last accessed on 07/06/2019.

[7] Tesla, "Tesla fourth quarter & full year 2018 update," tech. rep., Jan 2019. https://ir.tesla.com/static-files/0b913415-467d-4c0d-be4c-9225c2cb0ae0. Last accessed on 07/06/2019.

[8] "Audi A8 - Audi AI traffic jam pilot," *Audi Technology Portal*, Jul 2017. https://www.audi-technology-portal.de/en/electrics-electronics/driver-assistant-systems/audi-a8-audi-ai-traffic-jam-pilot. Last accessed on 09/06/2019.

[9] Secretary of State for Transport UK, "The road vehicles (construction and use) regulations 1986," Jun 1986. http://www.legislation.gov.uk/uksi/1986/1078/regulation/104/made. Last accessed on 09/06/2019.

[10] "New Audi A8 debuting level 3 autonomous AI traffic jam pilot; parking and remote garage pilots; zFAS controller," *Green Car Congress*, Jul 2017. https://www.greencarcongress.com/2017/07/20170712-a8.html. Last accessed on 09/06/2019.

[11] S. Jose, V. V. S. Variyar, and K. P. Soman, "Effective utilization and analysis of ROS on embedded platform for implementing autonomous car vision and navigation modules," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 877–882, Sept 2017. https://ieeexplore.ieee.org/document/8125952.

[12] Velodyne LiDAR, *VLP-16 User Manual*, Dec 2017. https://usermanual.wiki/Pdf/VLP16Manual.1719942037.pdf. Last accessed on 09/06/2019.

[13] W. Elmenreich and S. Pitzek, "Using sensor fusion in a time-triggered network," in *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No.37243)*, vol. 1,

pp. 369–374 vol.1, Nov 2001. https://mobile.aau.at/~welmenre/papers/elmenreich_Dissertation_sensorFusionInTimeTriggeredSystems.pdf.

[14] P. D. Rueckert and D. B. Kainz, "CO317 Graphics," January 2018. Imperial College London - Department of Computing. Notes from the 2017-2018 academic year.

[15] S. Yang, H. Lho, and B. Song, "Sensor fusion for obstacle detection and its application to an unmanned ground vehicle," in *2009 ICCAS-SICE*, pp. 1365–1369, 2009. https://ieeexplore.ieee.org/document/5335263.

[16] J. Kim, D. S. Han, and B. Senouci, "Radar and vision sensor fusion for object detection in autonomous vehicle surroundings," in *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*, July 2018. https://ieeexplore.ieee.org/document/8436959.

[17] M. Velas, M. Spanel, Z. Materna, and A. Herout, "Calibration of RGB camera with Velodyne LiDAR," 2014. https://pdfs.semanticscholar.org/ed15/5d1a146e0cba6be98fd7128461439f88732a.pdf.

[18] M. Velas, M. Spanel, Z. Materna, and A. Herout, "ROS packages for Velodyne 3D LIDARs provided by RoboFIT group.," 2017. https://github.com/robofit/but_velodyne/tree/master/but_calibration_camera_velodyne. Commit used: 97b6943564920cbcfda0b143463e957c066c867a.

[19] Z. Pusztai and L. Hajder, "Accurate calibration of LiDAR-camera systems using ordinary boxes," in *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*, pp. 394–402, Oct 2017. https://ieeexplore.ieee.org/document/8265264.

[20] A. Geiger, F. Moosmann, Ö. Car, and B. Schuster, "Automatic camera and range sensor calibration using a single shot," in *2012 IEEE International Conference on Robotics and Automation*, pp. 3936–3943, May 2012.

[21] J. C. Becker and A. Simon, "Sensor and navigation data fusion for an autonomous vehicle," in *Proceedings of the IEEE Intelligent Vehicles Symposium 2000 (Cat. No.00TH8511)*, pp. 156–161, Oct 2000. https://ieeexplore.ieee.org/document/898335.

[22] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, pp. I–I, Dec 2001.

[23] A. Geitgey, "Machine Learning is fun! part 4: Modern Face Recognition with Deep Learning," *Medium*, Jul 2016. https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-face-recognition-with-deep-learning-c3cffc121d78. Last accessed on 20/05/2019.

[24] Q. Wong, "Why facial recognition's racial bias problem is so hard to crack," *cnet.com*, Mar 2019. https://www.cnet.com/news/why-facial-recognitions-racial-bias-problem-is-so-hard-to-crack/. Last accessed on 05/06/2019.

[25] Open Source Robotics Foundation, "ROS - About ROS," Aug 2018. https://www.ros.org/about-ros/. Last accessed on 16/06/2019.

[26] X. Wang, T. Zheng, and Z. Zhang, "Design and implementation of intelligent vehicle control system based on ROS," in *2018 Chinese Automation Congress (CAC)*, pp. 1661–1665, Nov 2018. https://ieeexplore.ieee.org/document/8623325.

[27] D. Robineau, "Reported road casualties in Great Britain: 2017 annual report," tech. rep., HM Department for Transport, 2018. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/744077/reported-road-casualties-annual-report-2017.pdf. Last accessed on 16/06/2019.

[28] J. Petit, B. Stottelaar, and M. Feiri, "Remote attacks on automated vehicles sensors: Experiments on camera and LiDAR," 2015. https://pdfs.semanticscholar.org/e06f/ef73f5bad0489bb033f490d41a046f61878a.pdf.

[29] J. Petit and S. E. Shladover, "Potential cyberattacks on automated vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, pp. 546–556, April 2015. https://ieeexplore.ieee.org/document/6899663.

[30] S. Lagraa, M. Cailac, S. Rivera, F. Beck, and R. State, "Real-time attack detection on robot cameras: A self-driving car application," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pp. 102–109, Feb 2019. https://ieeexplore.ieee.org/document/8675588.

[31] H. Terai, H. Okuda, K. Hitomi, T. Bando, C. Miyajima, T. Hirayama, Y. Shinohara, M. Egawa, and K. Takeda, "An experimental study on the difference in drivers' decision-making behavior during manual and supported driving," *6th International Conference on Applied Human Factors and Ergonomics (AHFE 2015) and the Affiliated Conferences, AHFE 2015*, vol. 3, pp. 3136 – 3141, 2015. http://www.sciencedirect.com/science/article/pii/S2351978915009750.

[32] C. Gartenberg, "Safety driver of fatal self-driving Uber crash was reportedly watching Hulu at time of accident," *The Verge*, Jun 2018. https://www.theverge.com/2018/6/22/17492320/safety-driver-self-driving-uber-crash-hulu-police-report. Last accessed on 16/06/2019.

[33] H. Bouckley, "Has the smartphone killed the compact camera? Is it worth buying a compact camera?," *BT*, Sept 2018. http://home.bt.com/tech-gadgets/photography/smartphone-cameras-vs-compacts-is-it-still-worth-buying-a-compact-camera-11363960940202. Last accessed on 14/06/2019.

[34] M. Zhang, "This latest camera sales chart shows the compact camera near death," *PetaPixel*, Mar 2017. https://petapixel.com/2017/03/03/latest-camera-sales-chart-reveals-death-compact-camera/. Last accessed on 14/06/2019.

[35] D. Bader, "Google Pixel 2 and Pixel 2 XL specs," *androidcentral.com*, Oct 2017. https://www.androidcentral.com/google-pixel-2-specs. Last accessed on 14/06/2019.

[36] A. Geitgey, "face_recognition: The world's simplest facial recognition API for Python and the command line," 2019. https://github.com/ageitgey/face_recognition. Commit used: 2f267d8537787ae5684e29f668e04fcdd3723006.

[37] D. Nowak, "Introducing Husarnet - a global LAN network," *Medium*, Oct 2018. https://medium.com/husarion-blog/introducing-husarnet-a-global-lan-network-5795222d2214. Last accessed on 13/06/2019.

[38] D. Nowak, "Don't buy expensive sensors for your robot - use your smartphone," *Medium*, Mar 2018. https://medium.com/husarion-blog/dont-buy-expensive-sensors-for-your-robot-use-your-smartphone-24380eab521. Last accessed on 23/04/2019.

[39] MathWorks, "cameraMatrix package documentation," 2019. https://uk.mathworks.com/help/vision/ref/cameramatrix.html. Last accessed on 30/05/2019.

[40] A. Sobral, "Vehicle detection by Haar Cascades with OpenCV," 2016. https://github.com/andrewssobral/vehicle_detection_haarcascades. Commit used: 4fe96fac22c268082dd761e7a39c7979e561448c.

[41] PixyCam, "Introducing Pixy2." https://pixycam.com/pixy2/. Last accessed on 16/06/2019.

[42] M. Bjelonic, "YOLO ROS: Real-time object detection for ROS," 2019. https://github.com/leggedrobotics/darknet_ros.

[43] B. Barbed, "FYP-VehicleSensorFusion: Final Year Project for Masters in Electronic Engineering," June 2019. https://github.com/belenbarbed/FYP-VehicleSensorFusion. Commit used: 7f457b38bb267e8c52698318bee758ac14219966.

Appendix A

User Guide

This section is a rewording of the project's `README.md` as seen in its GitHub repository [43].

A.1 First time setup

A.1.1 Phones

The phones are running the app *hNode*, https://play.google.com/store/apps/details?id=com.husarion.node&hl=en_GB, which establishes a network between them and the PC [38]. After being registered on the same Husarnet network together with the PC, running `rostopic list` shows the topics the phones are publishing to with their sensor data.

A.1.2 PC (master node)

Add the PC to the Husarnet network using:

```
$ curl https://install.husarnet.com/install.sh | sudo bash
$ sudo husarnet websetup
```

Follow the link returned to configure your device.

Make sure the `~/.bashrc` or `~/.zshrc` on the PC has the following lines to ensure ROS compiles and that the Husarion networks connects correctly:

```
source /opt/ros/melodic/setup.bash
source <ROS workspace dir>/devel/setup.sh

export ROS_IPV6=on
export ROS_MASTER_URI=http://master:11311
export ROS_HOSTNAME=master
```

After pulling this repository into the `src` of a catkin workspace, remember to also pull the git submodules present, `velodyne` and `vision_opencv`, with:

```
$ git submodule update --init --recursive
```

It is recommended to remove the `build/` and `devel/` folders and build the workspace from scratch at the beginning, using `catkin clean` and `catkin build`.

A.1.3 Velodyne

Follow the ROS integration instructions at http://wiki.ros.org/velodyne/Tutorials/Getting%20Started%20with%20the%20Velodyne%20VLP16. For IP address configuration, check out the slides available at https://velodynelidar.com/docs/manuals/63-9266%20REV%20A%20WEBSERVER%20USER%20GUIDE,HDL-32E%20&%20VLP-16.pdf.

A.2 Calibration

To calibrate the phone cameras, we used the Matlab function *cameraCalibrator*, https://uk.mathworks.com/help/vision/ref/cameramatrix.html, which uses a 8x6 chequerboard with 24mm-sized squares, photographed 40 times from different angles and perspectives. This result returns the focal length and the principal points of the matrix (f and o_x and o_y in the projection matrix of Figure A.1).

$$\mathcal{P} = \begin{pmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure A.1: A pinhole camera's projection matrix.

The four phones can be calibrated against the LiDAR separately. To do so, run:

```
$ roslaunch but_calibration_camera_velodyne calibration_fine.launch pixel:="PIXEL"
```

with `PIXEL` being the phone one wants to calibrate (1-4). Please ensure the relevant phone is facing the marker (a card with four equidistant circles cut from it) before beginning the calibration process.

The results from this calibration are printed on the terminal output and should be saved in `but_calibration_camera_velodyne/conf/coloring.yaml` in order to be used for the colouring node in the same package, executable with:

```
$ roslaunch but_calibration_camera_velodyne coloring.launch
```

For further camera-to-LiDAR calibration instructions, it is recommended to review the paper that explains the theory [17], as well as the original package that implemented it [18].

A.3 Usage

On the PC, several commands should be run.

On the first terminal tab, run `roscore`.

Then, to make the Velodyne LiDAR start posting pointcloud data, execute:

```
$ roslaunch velodyne_pointcloud VLP16_points.launch
```

After that, one can run:

```
$ rosrun face_detection face_detection_py.py
```

to detect faces, or:

```
$ rosrun car_detection car_detection_py.py
```

to detect cars in all four camera streams, and display them all using:

```
$ rosrun visualization video_visualization
```

A.4 Troubleshooting

If things are suddenly not working and one wants to start the build afresh, run:

```
$ cd <ROS workspace dir>
$ catkin clean
$ catkin build
```

Then remember to source in **ALL** open terminal tabs:

```
$ source devel/setup.sh
```

If the phones, PC, and LiDAR don't sync properly, check their exact system time. If it's not the same, down to the second, one may have sync issues. For reference, check out this atomic clock website, time.is, to match their system times.

Other useful troubleshooting guides may be found in the repository's `README.md`.