Navigation Assistance for a Semi-Autonomous Smart Wheelchair

Jakob Wyatt

Supervisors: Siavash Khaksar & Yifei Ren

B. Eng. Mechatronic Engineering Research Project
Department of Mechanical Engineering
Curtin University

Semester 2 2022

THIS PAGE REDACTED DUE TO SENSITIVE INFORMATION

ACKNOWLEDGEMENTS

I would like to thank Glide for collaborating with Curtin University on this thesis project and providing a powered wheelchair to the research team.

Further thanks go to Siavash Khaksar and Yifei Ren, for supervising this thesis and providing invaluable technical knowledge and support.

I am also grateful to the remaining smart wheelchair thesis project students; collaborating with likeminded engineers to research a semi-autonomous smart wheelchair was a highlight of this engineering thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Semi-autonomous smart wheelchair technology can enable visually impaired users to drive a wheelchair safely. In collaboration with wheelchair manufacturer Glide, a navigation assistance system for a smart wheelchair is implemented and evaluated. This smart wheelchair uses a CentroGlide wheelchair as a powered base and a ZED Mini RGB-D stereo camera to detect the surrounding environment. A semantic segmentation ML model was retrained to identify footpaths and other drivable areas around the wheelchair. Additionally, an algorithm was developed to identify obstacles and hazards using 3D point cloud data from the stereo camera. A birds-eye view occupancy map is generated to indicate the location of obstacles and drivable areas around the wheelchair. This occupancy map is used to determine a safe target direction for the wheelchair. To evaluate this navigation assistance system, an RGB-D wheelchair driving dataset was collected around Curtin University. It was found that the identification of drivable areas using semantic segmentation was effective in outdoor areas but less effective indoors. Obstacle detection was effective at identifying environmental obstacles such as walls and handrails. The assistive control algorithm successfully modifies the wheelchair's direction to avoid obstacles, however, can fail in some scenarios due to the low FOV of the stereo camera. Suggestions for future work are also discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

DECLARATION OF PUBLISHED WORKS

I acknowledge that the literature review of this thesis was previously submitted as part of the Progress Report assessment for MXEN4000. Additionally, I acknowledge that parts of the ‘Evaluation of ML models on preliminary dataset’ section of the methodology was previously submitted as part of the same progress report assessment. I also acknowledge that this thesis contains no material previously published by any other person except where due acknowledgment has been made.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

List of Figures

List of Tables

1 INTRODUCTION

1.1 Background and motivation

Powered wheelchairs have substantially benefited people with mobility challenges. The use of these wheelchair systems has been shown to improve the user's independence, quality of life and feeling of well-being [1]. However, this technology can be inaccessible or unsafe for people with vision impairment, multiple sclerosis, ALS, and cerebral palsy, due to difficulties with environmental perception or joystick use. A survey of 65 clinicians found that 10-40% of patients found it impossible or extremely difficult to use a powered wheelchair [2].

Smart wheelchairs add intelligent sensing and control to an existing powered wheelchair to avoid obstacles in the environment or to navigate the user. A smart wheelchair may include alternative input methods such as eye gaze [3], head tilt [4], and voice recognition [5]. Fully-autonomous smart wheelchairs move the user from a start pose to an end goal, and often require a map of the surrounding environment or markers such as QR codes [6]. Semi-autonomous smart wheelchairs blend input from the user with the control unit to avoid obstacles and improve safety. Using a smart wheelchair improves the user's safety and independence; Simpson et al. found that over 61% of current wheelchair users would benefit from some smart wheelchair features [7].

1.2 Aims

This research aims to develop a semi-autonomous smart wheelchair system. This research was done in collaboration with Glide, a WA wheelchair manufacturer, who has provided an existing powered wheelchair (CentroGlide) to use as a base for this functionality (fig. 1). By developing assistive technology for the wheelchair, the user is granted greater mobility, confidence, and independence.

The project team comprises multiple engineering project students, researchers, and interns. This team has worked on many smart wheelchair features, including motor controller design (Kosma Egan), input controller design (Brian Smith), doorway navigation (Avinash Sudhakaran), object detection (Krishnadas Suresh), and navigation to a vehicle (Nicolas Lee). The specific aim of this thesis is to implement navigation assistance, which involves both avoiding environmental obstacles such as walls and stairs, and guiding the user along suitable pathways. If a user were to unintentionally drive off a pathway, they could encounter a kerb or uneven terrain, which could lead to a fall.

1.3 Problem definition

An important requirement of this system is that the user still has control over their wheelchair and can override any autonomous functionality if required. If the smart wheelchair system mistakenly detects an obstacle, the user's mobility should not be compromised.

Another requirement of the system is that any sensors mounted to the wheelchair should not impede the user's comfort or the wheelchair's manoeuvrability. Many wheelchair users have specific requirements for wheelchair seat adjustments to avoid pressure sores and discomfort. Figure 1 shows the wheelchair configuration when fully reclined, demonstrating that some sensor mounting locations are infeasible.

The smart wheelchair system should also be commercially viable - high-cost components and sensors are infeasible. Internet connectivity should not be a requirement for the system to operate either - the round trip time required to communicate with a server could compromise the safety of a user. Because of this, all processing is performed locally on the wheelchair.

(a) Upright (b) Reclined

Figure 1: CentroGlide wheelchair

2 LITERATURE REVIEW

2.1 Sensors and hardware

2.1.1 Sensor types

Smart wheelchairs have used a varied range of sensor types to perceive the surrounding environment. RGB-D stereo cameras have been widely used in this field [8][9][10], alongside 2D Lidar [11] and ultrasonic sensors [12]. Self-driving cars built by companies such as Tesla and Waymo use cameras, mmWave Radar, and 3D Lidar to avoid traffic and pedestrians [13].

Selecting a sensor to use is not necessarily an either-or decision. Sensor fusion algorithms such as the Extended Kalman Filter (EKF) or Unscented Kalman Filter (UKF) [14] allow outputs from multiple sensors to be used together to improve their accuracy. Additionally, sensors can be used for different applications on the smart wheelchair - a stereo camera could be used to sense the surrounding environment while an inertial measurement unit (IMU) could be used for wheelchair odometry. Table 1 compares several available sensor types, taking into account factors such as resolution, cost, and accuracy.

Sensor	Advantages	Disadvantages
RGB-D Stereo Camera	Very high resolution	Low field of view (FOV)
mmWave Radar	High accuracy	Low resolution
3D Lidar	High resolution and accuracy	Very high cost
2D Lidar	High FOV and accuracy	Only detects obstacles within the same plane
Ultrasonic sensor	Low cost	One-dimensional

Table 1: Sensor comparisons

2.1.2 RGB-D cameras

One advantage RGB-D cameras have over alternative sensors is high RGB resolution, allowing them to utilise advances in machine learning and computer vision. Technologies such as Lidar may fail at path detection, as path markings cannot be detected.

When comparing these cameras, factors such as package size, field of view, and operating range are important to consider. Several commercial options are compared in Table 2 - all of the listed units come with an integrated IMU.

Name	Type	Cost (AUD)[1]	Dimensions (mm)
Stereolabs Zed Mini [15]	Passive	$595	$124.5 \times 30.5 \times 26.5$
Stereolabs Zed 2 [16]	Passive	$670	$175 \times 30 \times 33$
Intel RealSense D455 [17]	Active IR (Stereo)	$595	$124 \times 26 \times 29$
Microsoft Azure Kinect DK [18]	Active IR (Time of Flight)[2]	$595	$103 \times 39 \times 126$

Table 2: Stereo camera options

Name	FOV (Horizontal, Vertical, Depth)	Operating Range (m)
Stereolabs Zed Mini [15]	$90^\circ \times 60^\circ \times 100^\circ$	0.1-15
Stereolabs Zed 2 [16]	$110^\circ \times 70^\circ \times 120^\circ$	0.3-20
Intel RealSense D455 [17]	$90^\circ \times 65^\circ \times 87^\circ$	0.6-6
Microsoft Azure Kinect DK [18]	$75^\circ \times 65^\circ \times 75^\circ$	0.5-3.86

Table 2: Stereo camera options (continued)

Name	Cost (AUD)[1]	Release Year	Speed	Power	Notes
Nvidia Jetson Nano [19]	$150	Early 2019	0.5 TFLOPS (FP16)	10 W	-
Nvidia Jetson Xavier NX [20]	$595	Late 2019	21 TOPS (INT8)	20 W	-
Nvidia RTX 2080 [21]	$1040	2018	80.5 TFLOPS (FP16) 161.1 TOPS (INT8)	215 W	Doesn't include single-board computer
Google Coral Edge TPU [22]	$190	2020	4 TOPS (INT8)	2 W	Only supports TensorFlow Lite

Table 3: AI accelerator options

A caveat of the Stereolabs products is that they require a separate CUDA-enabled GPU (manufactured by Nvidia) to generate the point-cloud and RGB-D image. In contrast, the Kinect DK only requires a CPU for processing, while the Intel RealSense performs processing onboard and requires a USB-C 3.1 interface to communicate.

2.1.3 Compute element

A compute element inside a semi-autonomous driving system generally consists of several components - a microcontroller to process user inputs and send signals to the motors, a general-purpose computer to run pathfinding algorithms and log information, and an AI accelerator to improve the performance of on-board machine learning (ML) algorithms.

AI accelerators use specialised hardware to perform operations common in ML algorithms (such as matrix multiplication and convolution) more efficiently than a CPU can. GPUs have been used widely for this application; however, their high power usage is infeasible for some applications. Embedded AI accelerators aim to provide greater power efficiency at the cost of specialisation.

Machine learning models often use mixed-precision (FP16) datatypes to store weights while training. Although improving the model's accuracy, FP16 datatypes are slow to manipulate during inference. Model quantisation [23] is a process where this FP16 datatype is replaced with an INT8 datatype (using a scaling factor and bias) after training. Quantisation greatly improves speed while only losing a small amount of model accuracy. For this reason, modern AI accelerators focus on the performance of INT8 operations (TOPS, 10^9 operations per second), whereas earlier accelerators state the performance of FP16 operations (TFLOPS, 10^9 floating-point operations per second).

The Nvidia Jetson and Google Coral products are both popular options for embedded AI acceleration. These accelerators are compared in Table 3 alongside a gaming GPU.

[1] Costs are taken at RRP with an exchange rate of 1 AUD = 0.74 USD

[2] The Microsoft Azure Kinect DK has multiple operating modes that trade-off between FOV, operating range, and resolution. The `NFOV unbinned` mode was compared, which provides a good trade-off between operating range and resolution.

2.2 Scene understanding

Scene understanding is a broad field and involves using computer vision methods on visual or spatial data to gain better knowledge about the surrounding environment. Convolutional Neural Networks (CNNs) are commonly used for this application, as they can exploit the local nature of image features to reduce the number of required computations.

2.2.1 Image classification

Image classification is a core problem within this field and involves identifying the subject of an image (such as an animal or object). AlexNet [24], based on the earlier digit-recognition CNN LeNet-5 [25], was one of the first deep CNNs applied to this problem. AlexNet was trained on the large ImageNet dataset [26], which consists of 15M images and 22K categories, and achieved an error of only 15.3% on a 1000 class subset. The underlying architecture uses a series of 5 convolutional layers and 3 fully connected layers, which can be seen in fig. 2.

Neural network architectures have become deeper and more accurate over time, enabled by both growth in computational power and dataset size. VGG-16 [27] and GoogLeNet [28] are 16 and 22 layers deep respectively, and approached human performance on the ImageNet dataset. ResNet [29] is up to 156 layers deep, and exceeds human performance at image classification with an error of 3.57%. ResNet uses a 'skipping' architecture to improve network training, where the output of a layer relies directly on the input of a previous layer.

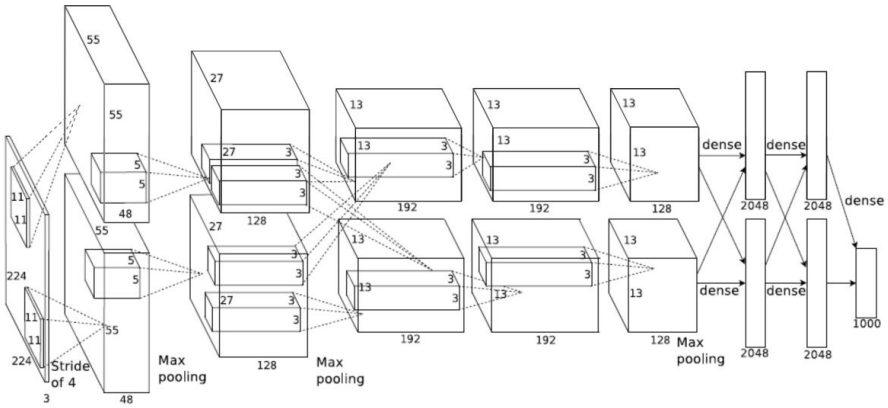


Figure 2: Architecture of the AlexNet image classification network. Reproduced from Krizhevsky et al. [24]

2.2.2 Object localisation

Object localisation is another core problem within this field, and involves identifying the location of objects within an image as well as classifying them. Object localisation can be used on a semi-autonomous wheelchair to identify a pedestrian or obstacle within the environment. R-CNN [30] was one of the first object classification models which utilised convolutional networks, by identifying potential bounding boxes and running an image classifier on these bounding boxes. Fast and Faster R-CNN [31][32] improved the speed of this model by running an image classifier backbone once on the entire image, and using a CNN to improve the identification of bounding boxes. Pascal VOC [33] and MS COCO [34] are datasets that are commonly used to evaluate object classification models.

YOLO (You Only Look Once) [35][36][37][38] is another object classification model which focuses on improving the speed of the model. In particular, YOLOv4 [38] reaches over 60 fps (frames per second) on the Tesla V100, which enables its use in real-time applications such as autonomous driving and security camera footage. YOLO divides an image into an $S \times S$ grid, and uses a single convolutional network to output both bounding box predictions and an image classification for each grid square. Low probability and overlapping bounding boxes are then removed before the final output.

2.2.3 Semantic segmentation

Semantic segmentation involves labelling each pixel of an object, rather than drawing a bounding box around the entire object. This technique is often used in medical applications, where different components of a scan need to be labelled. Another application semantic segmentation can be used for is drivable area detection, as a bounding box would not be able to cleanly identify a road or kerb. Figure 3 compares the output of semantic segmentation to image classification and object localisation.

Most semantic segmentation algorithms use an encoder-decoder architecture, where information about the image is encoded into a small feature space. This feature space is then decoded back to the size of the original image using deconvolutional layers to obtain the segmented output. Encoding is typically done using a pre-trained model backbone, such as ResNet, which reduces the computational power required to train the model. An issue with this architecture is that the resulting segmentation can be low quality, as the image encoding is low resolution. U-net [39] is a semantic segmentation network that helps to rectify this issue, by using higher-resolution features during deconvolution. This makes the segmented output sharper and more accurate.

Another semantic segmentation algorithm is DeepLab [40][41][42]. DeepLab uses atrous convolution (otherwise known as dilated convolution), which is a type of convolution that widens the FOV of a convolutional layer. It does this by leaving gaps in the convolutional layer, as illustrated in fig. 4. By widening the FOV of each convolution, less downscaling is required during encoding. This allows the image to be encoded in a much higher resolution, leading to a more accurate output. To obtain the segmented output, a technique called atrous spatial pyramid pooling (ASPP) is used. ASPP samples the feature space at different scales using atrous convolution to classify each pixel in the image. These techniques improve both the accuracy and speed of the network - DeepLabv3 obtained 86.9% accuracy on the PASCAL VOC 2012 test set.

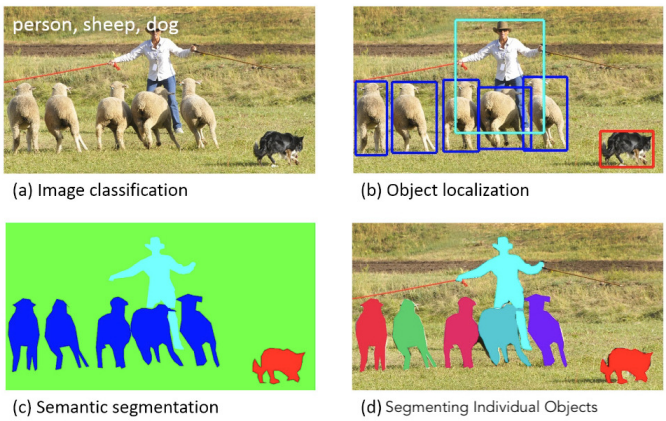


Figure 3: Types of classification in machine vision. Reproduced from Lin et al. [34]

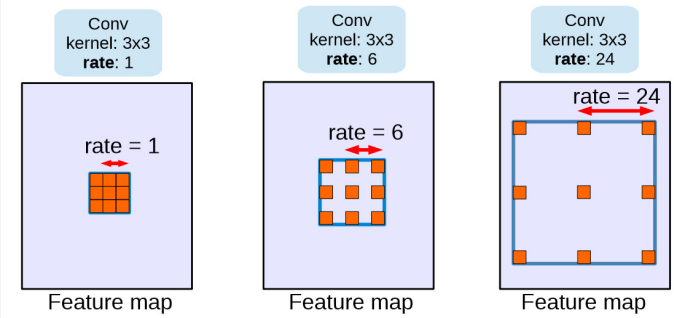


Figure 4: Atrous convolution with a 3x3 kernel, showing increasing FOV. Reproduced from Chen et al. [42]

2.2.4 Autonomous driving

Autonomous driving and autonomous wheelchair control involve similar challenges, including drivable area segmentation and object detection. As described previously, many machine learning models utilise an image classification backbone to extract features from an image, with a 'detection head' then used for the final prediction.

Running multiple classification backbones for different models can be inefficient, as the work of feature extraction is duplicated many times over. One way to improve the performance of the autonomous system is to share a classification backbone between models, and use different detection heads for various tasks.

An example of this architecture is HydraNets [13], a machine learning model used by Tesla to perform tasks such as traffic light detection, lane prediction, and object detection efficiently. Other machine learning models that utilise this architecture include YOLOP [43] and Hybridnet [44], which focus on lane segmentation, drivable area segmentation, and object detection. The model architecture of YOLOP is shown in fig. 5.

This approach is valuable in situations where compute hardware is limited. Real-time applications such as autonomous wheelchair control require fast inference times to react to obstacles in the surrounding environment.

To train these ML models, a model backbone (pre-trained on a dataset such as ImageNet [26]) is retrained on a driving dataset. This technique is known as transfer learning, and can vastly reduce the amount of time required to train a new model. Driving datasets such as Cityscapes [45] and Berkeley DeepDrive [46] can be used for retraining. Additionally, game-engine based driving simulators such as CARLA [47] can be used to generate a synthetic dataset for training.

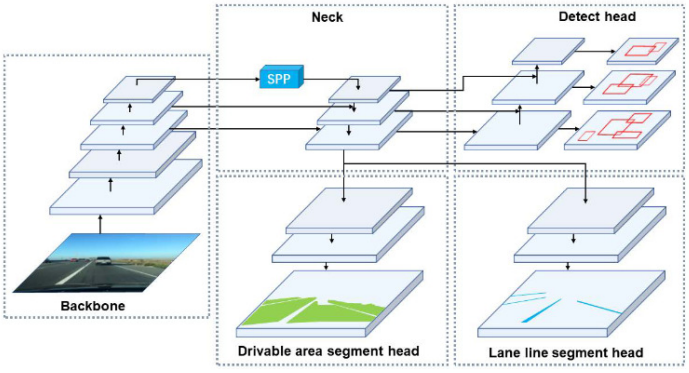


Figure 5: YOLOP model architecture. Reproduced from Wu et al. [43]

2.3 Assistive control

Once an understanding of the 3D scene has been built, the user can be navigated through the environment. The surrounding environment is generally represented as an occupancy grid [48], which is a top-down view of the area where each grid cell indicates the probability that it is occupied by an obstacle. It is possible to include more detailed information about paths and obstacles by adding more information to the occupancy grid.

In semi-autonomous control, the user decides the desired speed and direction of the wheelchair, with any intervention only occurring before a collision takes place. This is in contrast to full autonomy, where the user specifies the desired end goal and the wheelchair navigates to that goal [8].

2.3.1 Path-based algorithms

Path-based algorithms take an occupancy grid as an input and output a path between the start point and a goal point. A* is an example of this and uses a heuristic to efficiently find the shortest path between the start and end point. Other algorithms such as RRT* (rapidly-exploring random tree) [49] build a tree from randomly sampled points to find a path to the goal node. RRT* may not find the shortest path initially, but can find an efficient path with much less computation required.

One potential issue with these two algorithms is that they fail to take into account the smoothness of the resulting path. Although the path may be short, sharp changes in the trajectory could be uncomfortable for the user.

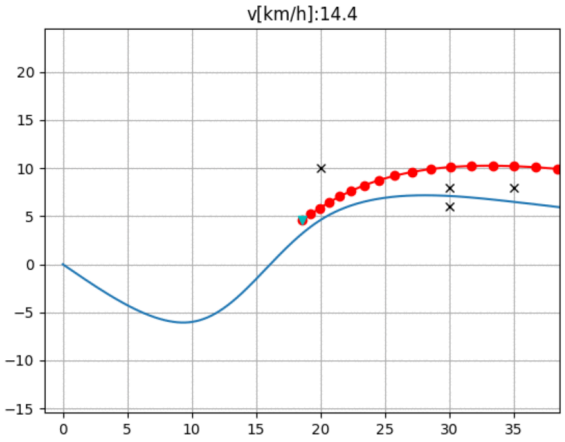


Figure 6: Frenét-Frame path planning, with reference path and local path illustrated. Reproduced from Sakai et al. [50]

Trajectory planning algorithms aim to solve this - one such algorithm is optimal-control in a Frenét-Frame [51], which can be used in autonomous vehicle control. This algorithm takes a reference path as an input and outputs a local path that avoids collisions and minimises jerk (rate of change of acceleration). This is done by generating sample trajectories (represented with quintic polynomials), removing those which cause collisions, and choosing the remaining trajectory with the lowest change in acceleration. Figure 6 illustrates the reference path, obstacles, and generated local path in an example scenario. It should be noted that this algorithm still requires a reference path, which could be generated with one of the pathfinding algorithms mentioned above.

2.3.2 Local algorithms

Local algorithms only consider obstacles currently in the proximity of the wheelchair, and use this information to set the current speed and direction of the wheelchair. VFH+ (vector field histogram) [52] is one example, which has been applied to wheelchair control algorithms in prior work [4]. VFH+ calculates a polar obstacle density histogram around the robot based on the occupancy grid. The histogram is then binarized, to classify sectors around the robot as either occupied or not occupied. Next, a masked polar histogram is generated, which excludes paths that are not possible given the robots turning radius and kinematics. Finally, a safe direction is chosen which is nearest to the user's desired direction. An example binary histogram is shown in fig. 7; the chosen direction avoids the obstacle in the desired direction.

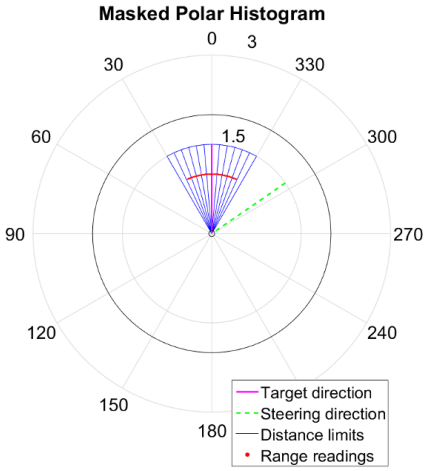


Figure 7: VFH+ binary histogram, representing the direction of obstacles. Reproduced from MathWorks [53]

An advantage to this algorithm is that it gives the user more fine-grained control over their speed and direction, rather than planning a path to their assumed goal. However an issue with VFH+ is that it does not control the wheelchair's speed, and instead only finds a safe direction. Ideally, the wheelchair should slow down if an obstacle is present.

Another approach to assistive control with local algorithms is haptic feedback. Rather than blending inputs from the autonomous software and the user, the joystick itself is actuated to make it more difficult to move in the direction of obstacles [54][55]. An advantage of this approach is that it gives the user total control over which direction of movement they choose, however, the additional force required to actuate the joystick may fatigue the user.

3 METHODOLOGY

3.1 Hardware

The smart wheelchair must sense the environment, process this information, and maneuver within the environment. Doing so requires hardware, including a sensor system, compute unit, and motor controller.

The literature review compares several types of sensors and sensor manufacturers. An RGB-D camera was selected as the primary wheelchair sensor, as it provides high-resolution images and depth information at a commercially viable price point. Capturing high-resolution image data builds flexibility into the system and enables the utilisation of popular machine learning algorithms.

This RGB-D camera must be mounted to the wheelchair at an appropriate point. The front of the joystick control unit was selected for several reasons:

1. A clear view of the environment in front of the wheelchair is provided.
2. The user does not obstruct the camera's view in any wheelchair configuration.
3. The camera does not obstruct the user's view or comfort in any wheelchair configuration.
4. When exiting the wheelchair, the user can move the joystick control unit and camera out of the way using the existing joystick control unit mount.

Some considerations must be addressed when using this mounting point.

1. Unstable video footage could be observed due to low rigidity in the joystick control unit mount.
2. Mount point is $790\,\text{mm}$ forward from the rear of the wheelchair, impacting the visibility of the rear and side of the wheelchair.
3. Doorway maneuverability is impacted if RGB-D camera width exceeds $150\,\text{mm}$.

This sensor mounting point is positioned on the right-hand side of the wheelchair, $720\,\text{mm}$ above the ground and $300\,\text{mm}$ behind the front of the wheelchair (measured from the footplate), and can be seen in fig. 9.

The RGB-D camera model selected was the Stereolabs ZED Mini, which uses passive stereo-vision to generate a depth map. Active IR RGB-D cameras were not viable for this application due to their poor outdoor performance and range. The width of the ZED Mini also fulfils the size requirements of the selected mounting point.

A 3D printed sensor mount was designed to fix the ZED Mini to the wheelchair mounting point. This sensor mount was based on a ZED Mini mount designed by Walter Lucetti at Stereolabs [56], with several major modifications made using Autodesk Inventor:

1. Width of the mount was greatly reduced to improve maneuverability.

2. A mounting plate was added, allowing the sensor mount to bolt onto the existing joystick control unit. Bolt holes are 8 mm in diameter.
3. Some sensor clips were modified to make sensor removal more convenient.
4. Sharp corners were rounded to reduce the risk of injury to a user.

Figure 8 shows a render of the ZED Mini camera and custom mount. Figure 9 shows the ZED Mini camera mounted to the joystick control unit on the wheelchair.

In addition to the RGB-D camera and Lidar sensor, an AI accelerator is required to process the sensors' output and run ML algorithms. The literature review compares several AI accelerators across categories such as computational speed, power draw, and price. An Nvidia Jetson Xavier NX was selected for this task, as the Stereolabs ZED Mini requires a CUDA-enabled accelerator to generate a depth map. This accelerator's low power draw and small form factor are suited for mobile robot applications such as smart wheelchairs. Due to budget constraints and the ongoing chip shortage, the project team could not procure this AI accelerator. Instead, a laptop with an RTX 3080 graphics card was used to record datasets and evaluate the speed of ML algorithms.

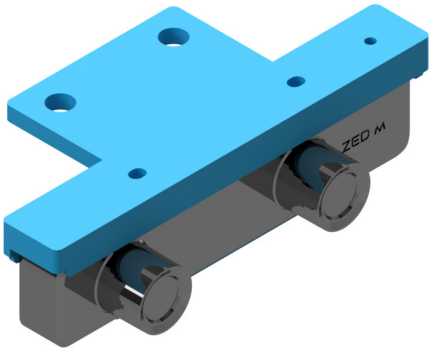


Figure 8: 3D render of ZED Mini camera and wheelchair mount

Figure 9: ZED Mini camera and mount fixed to joystick control unit

A CentroGlide powered wheelchair was used as a base for the smart wheelchair functionality. The CentroGlide is a mid-wheel drive wheelchair with two independently controlled powered wheels and four unpowered castor wheels. A joystick module communicates user commands to a power module, which controls each motor. An intelligent seating module (ISM) adjusts the seat tilt and recline [57]. This wheelchair is 1100 mm in length (including footplate), 620 mm in width, and 1030 mm in height. Communication between the joystick module, power module, and ISM is shown in fig. 10.

An input controller is being developed by project student Brian Smith to intercept commands from the joystick module and communicate with the smart wheelchair navigation system. The input controller enables the navigation system to receive user commands and choose a safe direction and speed for the wheelchair. The input controller will validate the navigation system's output, ensuring that the user can still safely maneuver the wheelchair in the case of a software failure. Additionally, a motor controller was developed by project student Kosma Egan, which receives commands from the input controller to drive and monitor the motors.

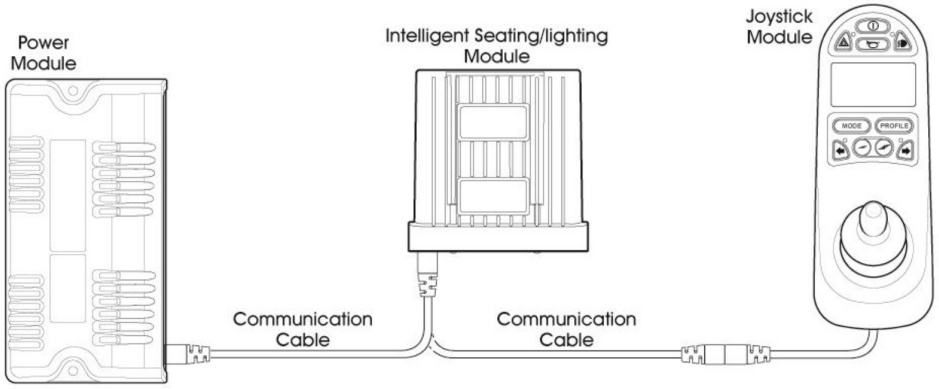


Figure 10: Communication between CentroGlide control modules. Reproduced from Curtiss-Wright. [58]

3.2 Data collection

An RGB-D wheelchair driving dataset was collected around Curtin University to test and evaluate the performance of the navigation assistance system. This dataset is 7.14 GB in size and 47 min in length and includes features such as indoor and outdoor navigation, doorways, pedestrians, elevator use, wheelchair access ramps, and car parks. A link to the dataset can be found in Appendix B.

This dataset was collected using the ZED Mini camera and is encoded in the proprietary Stereolabs SVO file format, which can be read using the ZED SDK. This format includes image data from left and right cameras, IMU data, and metadata such as timestamps. Depth map and point cloud data are not stored in the dataset and are instead generated when the file is read using the ZED SDK. Image data was recorded at 1080p, 30fps for the left and right cameras. A 6 min segment of the Curtin driving dataset was coarsely labelled with drivable areas; refer to appendix D for further information about the labelling process.

Four compression modes are available during data collection: Lossless (PNG), Lossy (JPG), H.264 (Video), and H.265. Both video compression modes require a CUDA-enabled device during data collection. A 10-second sample dataset was recorded for each compression mode to evaluate image quality and file size (results in table 4). H.264 compression was used when collecting the wheelchair driving dataset due to the higher compression ratio provided. Figure 11 shows an example image frame and depth map from the dataset.

Compression mode	File size	Relative increase	Image quality
Lossless (PNG)	1240 MB	41	Ok
Lossy (JPG)	640 MB	21	Some interlacing
H.264	30 MB	1	OK
H.265	30 MB	1	Some frame tearing

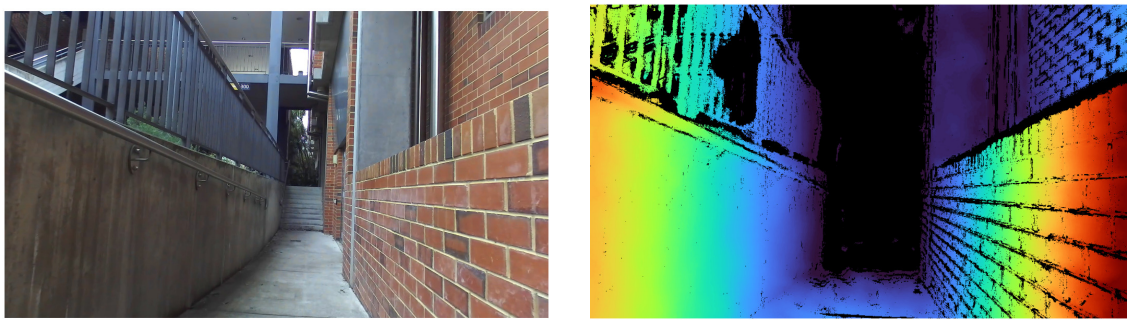
Table 4: Comparison between dataset compression modes

After some analysis of the raw point cloud data from the Curtin RGB-D driving dataset, it was found that the camera was slightly tilted upwards while recording the dataset. The tilt angle was consistent between data collection periods and was likely due to the ergonomics and tilt of the joystick control unit. An example of this tilt can be seen in fig. 12; note that the floor is angled downwards due to the tilt of the RGB-D camera.

Meshlab was used to determine the coordinates of points on the ground at a close distance and far distance, and hence calculate the sensor tilt angle. This was done for two different point clouds (one indoor and one outdoor) with the results averaged. Working is shown in eq. (1).

$$
\begin{aligned}
p_{1,1} &= (-0.53, -1.30, -3.81) \qquad p_{1,2} = (-0.54, -2.31, -11.31) \\
&\qquad\qquad \text{gradient}_1 = \frac{y_2 - y_1}{z_2 - z_1} = -0.135 \\
p_{2,1} &= (1.03, -1.29, -4.17) \qquad p_{2,2} = (0.93, -2.80, -14.03) \\
\text{gradient}_2 &= -0.153 \qquad \text{gradient}_{avg} = -0.144 \\
\theta_{tilt} &= -\tan^{-1}(-0.144) = 8.2^\circ
\end{aligned}
\tag{1}
$$

In addition to the RGB-D dataset, a preliminary wheelchair driving video-only dataset was collected around Curtin University. This dataset was 34 min in length and 7.08 GB in size (1920x1080 @ 24 fps) and collected using a GoPro Hero 4; the experimental setup used to collect this dataset can be seen in fig. 13. This dataset was used to evaluate machine learning scene recognition algorithms before the ZED Mini had been ordered.



(a) Image frame (b) Depth map

Figure 11: Sample data from the Curtin University RGB-D wheelchair driving dataset

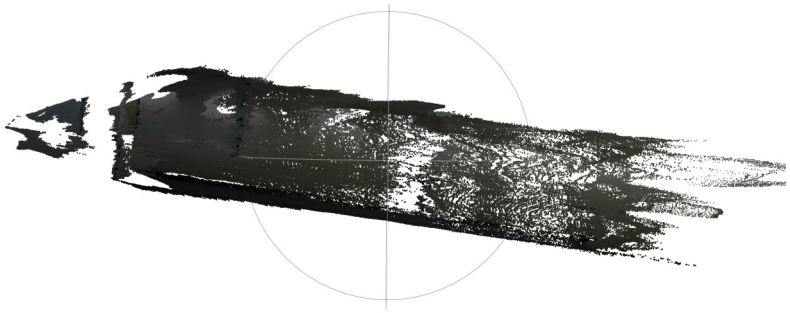


Figure 12: Point cloud information of indoor hallway - note the floor angle due to sensor tilt

Figure 13: Experimental setup for preliminary data collection

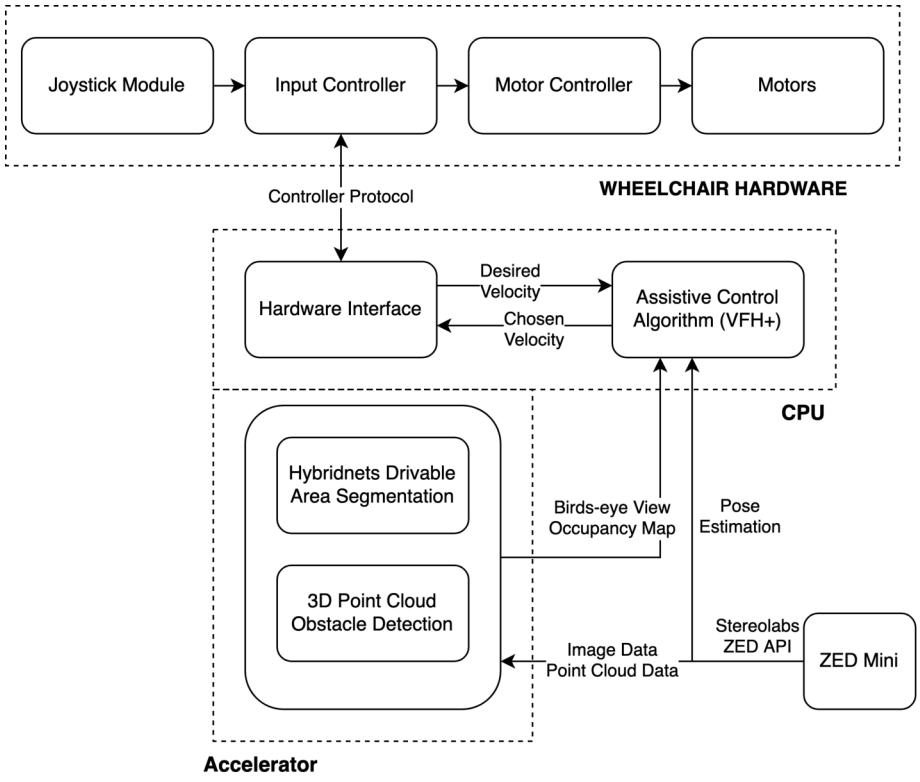


Figure 14: Block diagram of software system and interaction with hardware components

3.3 Software

The software system involves several scene recognition algorithms to update an internal map of the surrounding area, keeping track of suitable driving paths and static obstacles such as walls and stairs. Once this map is created, an algorithm blends the user's input with this information to determine a safe path forward. Appendix B contains a link to the source code, and appendix C contains instructions on how to run this source code so that the results can be reproduced. A block diagram of the software system and how it interacts with the hardware is shown in fig. 14.

3.3.1 Evaluation of ML models on preliminary dataset

The preliminary video-only driving dataset was used to evaluate several machine learning models before the ZED Mini had been ordered. Object detection models such as YOLOv5 [59] and image segmentation models such as DeepLabv3 [42] were evaluated for speed, accuracy, and efficacy. Hybridnets [44], a machine learning model which performs drivable area segmentation and object detection on the same backbone, was also evaluated on this dataset.

To evaluate these models, a video encoder and decoder needed to be created to feed image frames into the model. This was implemented using a Python generator so that frames in the video could be iterated over within a for loop and only decoded when required. OpenCV [60] was the underlying library used to decode each video frame into a BGR pixel array. Some scene processing algorithms do not run in real-time due to hardware limitations or performance issues. When this occurs, some frames must be dropped during model evaluation to keep the scene model up to date and minimise latency. The number of frames to drop can be calculated using eq. (2), where t_{frame} is the time taken to process the previous frame. A frame is dropped by simply discarding it once decoded.

$$n_{drop} = \lceil fps \times t_{frame} \rceil - 1 \tag{2}$$

To avoid training each model from scratch, a pre-trained model is loaded from PyTorch Hub. The DeepLabv3 and YOLOv5 models were both trained on the MS COCO [34] dataset, while the Hybridnets model was trained on the Berkeley DeepDrive dataset [46] (BDD100K).

As mentioned in the literature review, some machine learning algorithms can accept different image classifiers as a model backbone. MobileNetV3 [61] was used as a backbone for DeepLabv3 due to its fast performance. YOLOv5 has 5 model backbones to choose

between (nano, small, medium, large, extra-large), with smaller models sacrificing accuracy for speed. Due to the low latency requirements of a fast-moving wheelchair, YOLOv5 was evaluated using the small model size (known as YOLOv5s). Machine learning was done using PyTorch [62], due to good compatibility with existing machine learning models.

3.3.2 Hybridnets drivable area segmentation

To test the performance of the Hybridnets drivable area segmentation model, the model was evaluated on the BDD100K and Cityscapes [45] datasets. The model was also retrained using the Cityscapes dataset to improve its performance on non-uniform surfaces such as paved brick.

Machine learning was done within the Google Colaboratory environment, and a Colab Pro subscription was purchased to enable access to a faster Nvidia V100 GPU (with 32GB memory [63]). Datasets were uploaded onto Google Drive so that they could be mounted onto Colab and unzipped into the allocated machine's storage. Pytorch was used to train the model and `torchvision.datasets.Cityscapes` was used to load the Cityscapes dataset. To load the BDD100K dataset, a data loader within the Hybridnets source code was used.

The Hybridnets model contains an image classification backbone, an object detection head, and a segmentation head. The final layer of the segmentation head was modified so that it would output only drivable area predictions, rather than both lane predictions and drivable area predictions. The weights for the backbone and detection head were 'frozen', which means they were not changed during training. This is standard practice when training large models such as these, and reduces the time taken to train the model.

The mIoU (mean intersection over union) metric was used to evaluate the performance of the model. Equation (3) can be used to calculate the mIoU which relies on the true positive, false positive, and false negative pixel counts in the segmented image.

$$mIoU = \frac{1}{N} \sum_{i=1}^{N} \frac{tp_i}{tp_i + fp_i + fn_i} \tag{3}$$

The Cityscapes dataset was selected in order to improve the model's performance on non-uniform surfaces such as paved brick. This dataset categorises sidewalks as well as roads, which was hypothesised to provide a wider variety of surfaces during training. Both sidewalks and roads were classified as drivable areas during evaluation and training.

Recall (true positive rate, $\frac{tp}{tp+fn}$) was used as a metric to quantify the model's performance on sidewalks and roads separately. The AdamW optimizer [64][65] was used during training, with the model weights and other metrics saved to Google Drive each epoch. The model was trained for 20 epochs in total.

3.3.3 Birds-eye view occupancy map

The output of the Hybridnets drivable area segmentation model is a segmentation mask, an array that indicates which pixels are drivable and which are not. This output must be transformed into a 2D birds-eye view occupancy map, which indicates the drivable area around the wheelchair. The occupancy map extends 15 m in front of the wheelchair and 5 m on either side. This occupancy map is a simplified representation of the surrounding world and is used as an input to the wheelchair control algorithms.

The ZED Mini 3D point cloud data was used to transform the segmentation mask into an occupancy map. The ZED SDK uses the pinhole camera model, as seen in fig. 15, to describe the relationship between pixels on the image plane and the coordinates of corresponding objects. The point cloud data is represented as an array the same size as the original image, with each pixel containing XYZ data about the location of that object in 3D space.

To obtain the occupancy map, the point cloud is first filtered using the segmentation mask so that non-drivable points are removed. The ZED SDK is unable to resolve the depth of some pixels and instead represents their location as `nan`; these points are also removed.

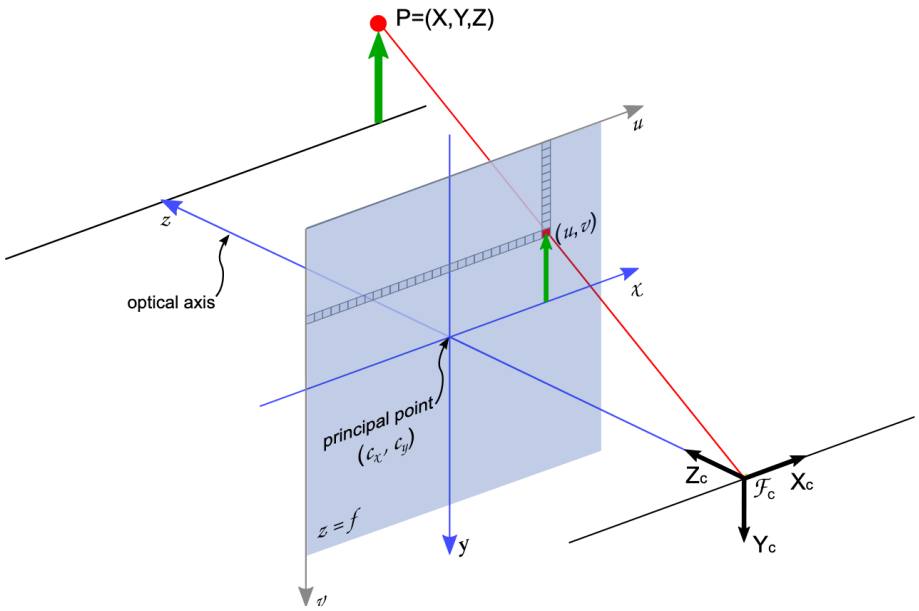


Figure 15: Pinhole camera model. Reproduced from Nvidia [66]

Next, the array is simplified by removing non-essential information such as colour and altitude data. What remains is an array of XZ points that represent drivable areas identified by the segmentation algorithm. These XZ points are referenced using a right-handed y-up coordinate frame, with the origin at the left lens of the ZED Mini; see fig. 16.

These points must be scaled to place them on the occupancy map. This occupancy map extends $15\,\text{m}$ in front of the wheelchair and $5\,\text{m}$ on either side, with each cell of the grid corresponding to a $25\,\text{mm} \times 25\,\text{mm}$ area. Equation (4) was used to scale these points. Note that p_x and p_z are represented in metres.

$$\begin{aligned} i_x &= min\left(max\left(\frac{p_x + 5.0}{0.025}, 1\right), \frac{10.0}{0.025}\right) - 1 \\ i_y &= min\left(max\left(\frac{p_z + 15.0}{0.025}, 1\right), \frac{15.0}{0.025}\right) - 1 \end{aligned} \tag{4}$$

An issue with the occupancy map at this stage is that it is rather sparse due to the one-to-one mapping between segmented pixels and grid cells. Morphological image processing techniques were used to improve the density of the occupancy map. The OpenCV [60] implementation of morphological dilation was used to join the space between drivable pixels to create a continuous driving surface. A 10×10 kernel was used to perform this dilation.

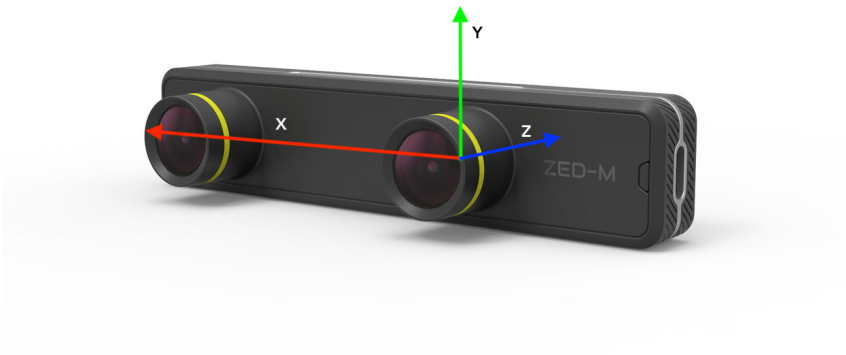


Figure 16: ZED Mini coordinate frame (right-handed y-up)

3.3.4 3D point cloud obstacle detection

Another method that was tested to identify environmental obstacles and drivable areas was the direct processing of the 3D point cloud data, which removes any use of RGB image data. One approach that was tested was the inbuilt ZED SDK function `find_floor_plane`. This function takes the current floor height and camera orientation as arguments and uses them to identify the floor plane. The `find_floor_plane` function returns a 3D polygon of the floor plane boundary, which was projected onto the 2D XZ plane and displayed using Pillow (a Python image processing library).

Another approach that was tested was a custom algorithm that processes the 3D point cloud data to identify the drivable area. The inequality in eq. (5) was initially used to filter and identify points as drivable, using a measured camera height $y_{camera} = 740\,\text{mm}$ and an initial error parameter $y_{error} = 150\,\text{mm}$. Equation (5) retains 3D points within a certain error margin of the floor plane.

$$-y_{error} \leq p_y + y_{camera} \leq y_{error} \tag{5}$$

An issue with this equation is that it does not take the tilt of the camera into account. Additionally, a wheelchair ramp may not be recognised as drivable, as the gradient of a wheelchair ramp may cause sections of the ramp to fall outside the error margin. Australian Standard 1428.1:2021 7.3a [67] specifies that the maximum gradient of a wheelchair ramp should not be greater than 1:14.

To rectify these issues, eq. (6) was also tested to identify the drivable area. This approach takes sensor tilt and wheelchair ramps into account, and assumes that any ramps will be encountered face-on (along the z-axis) and that sensor tilt remains constant. Note that in our coordinate system, p_z is negative in front of the wheelchair.

$$\begin{gathered} p_h = p_y - p_z \tan\theta_{tilt} + y_{camera} \\ p_z \times \frac{1}{14} - y_{error} \leq p_h \leq -p_z \times \frac{1}{14} + y_{error} \end{gathered} \tag{6}$$

Direct processing of the 3D point cloud data was also tested to detect static obstacles, such as stairs and walls. A basic implementation was tested using the existing sensor tilt compensation in eq. (6). Equation (7) shows the point filtering criteria for obstacles between $0.5\,\text{m}$ and $2.0\,\text{m}$ tall.

$$0.5 \leq p_h \leq 2.0 \tag{7}$$

Once the 3D points for drivable areas and environmental obstacles have been identified, they must be placed on the birds-eye view occupancy map. The approach used to do this is identical to the approach outlined in section 3.3.3; each point is scaled to fit on the map and morphology is used to improve the density of the map.

3.3.5 Assistive control algorithm

The VFH+ (vector field histogram) algorithm [52] was used to blend the user's desired direction with the occupancy map to determine a safe target direction. One limitation of the VFH+ algorithm is that it only adjusts the direction of the wheelchair and not the wheelchairs speed; as such, this algorithm was implemented as a proof of concept rather than a complete assistive control algorithm. The occupancy map generated by the 3D obstacle detection algorithm (detailed in section 3.3.4) was input into the VFH+ algorithm.

This algorithm was implemented in MATLAB using the Navigation toolbox and was encapsulated into a MATLAB function. The 'MATLAB Engine' Python API was used to call the assistive control algorithm from within Python code. Full implementation of a control algorithm would require the ability to log joystick commands during data collection, and to intercept joystick commands from the user. As the input controller had not been implemented at this time, the user's desired direction was assumed to be the current direction of the wheelchair.

As the VFH+ control algorithm had already been implemented within the Navigation toolbox, the MATLAB implementation was relatively simple. Equation (8) was used to adjust the pose from the RGB-D camera to the centre of the wheelchair (dimensions in metres).

$$\begin{aligned} p_{wheelchair,x} &= p_{camera,x} - \frac{0.62}{2} \\ p_{wheelchair,y} &= p_{camera,y} + \text{offset}_{\text{sensor}} - \frac{1.1}{2} \end{aligned} \tag{8}$$

3.3.6 Pose estimation

Pose estimation allows the smart wheelchair to update its position within the surrounding world. The ZED Sensors API and ZED Positional Tracking API were tested to estimate the pose of the wheelchair. These APIs were tested as a proof of concept for pose estimation and were not directly used for assistive control.

The Sensors API performs pose estimation using 6 DOF IMU sensor fusion and can also output raw accelerometer and gyroscope data. Although the IMU has a sampling rate of 800 Hz, this sensor information is only saved to the SVO file at the camera frame rate of 30 Hz.

The Positional Tracking API combines sensor data with image data to track the camera's movement. The camera's movement can be specified using two coordinate frames: CAMERA and WORLD. The WORLD frame outputs the position of the camera relative to the point where the ZED Mini started motion tracking (P_i), while the CAMERA frame outputs the difference between the current camera position and the camera position in the previous image (ΔP).

Another method to calculate ΔP would be to store the previous camera pose and use eq. (9) to calculate the difference between consecutive poses.

$$\begin{aligned} P_i \Delta P &= P_{i+1} \\ \Delta P &= P_i^{-1} P_{i+1} \end{aligned} \tag{9}$$

An advantage of this approach over using the CAMERA reference frame is that if a frame is skipped due to performance or latency issues, no movement data is lost. Performance should not be a concern, as the inversion of a pose matrix can be done very efficiently.

THIS PAGE INTENTIONALLY LEFT BLANK

4 RESULTS AND DISCUSSION

This thesis project has involved the implementation and evaluation of many different components required to create an end-to-end wheelchair navigation assistance system. The algorithms and methods discussed in the software methodology have been tested on the Curtin dataset, and their accuracy and speed have been evaluated. The strengths and weaknesses of each approach are discussed.

4.1 Evaluation of machine learning models on preliminary dataset

The YOLOv5, DeepLabv3, and Hybridnets models were tested on the preliminary video-only Curtin driving dataset. As this dataset is unlabelled, the models were evaluated in terms of speed and potential usage, rather than accuracy.

A frame of YOLOv5 evaluated on the preliminary dataset can be seen in fig. 17. This model identifies vehicles and pedestrians with high accuracy, with the confidence in a prediction decreasing as the object moves further away from the camera.

A frame of DeepLabv3 evaluated on the preliminary dataset can be seen in fig. 18. DeepLabv3 identifies pedestrians with a high level of accuracy. However, the vehicles in the image are not accurately segmented. This is likely due to the lower occurrence of vehicles in the MS COCO dataset, which was used to train the model. To repurpose this model for the task of drivable area segmentation, it would have to be retrained on a labelled driving dataset such as BDD100K or Cityscapes.

Figure 17: YOLOv5s evaluated on the Curtin dataset

Figure 18: DeepLabv3 evaluated on the Curtin dataset

An example of Hybridnets drivable area segmentation can be seen in fig. 19. This model accurately detects drivable areas outdoors when a path is uniform, however has more difficulty identifying a drivable path for non-uniform surfaces such as paved brick. Hybridnets can also struggle to identify drivable paths indoors in some circumstances. This is likely due to problems with domain adaptation, as the Hybridnets model was trained on the BDD100K dataset which primarily consists of bitumen roads.

The performance of each model (in frames per second) is given in table 5. This performance was evaluated using a laptop with an RTX 3080 graphics card and AMD Ryzen 9 6900HX processor. Note that Hybridnets is CPU limited, and could likely be optimized further by modifying postprocessing and video decoding methods. Additionally, Hybridnets runs both object detection and image segmentation using the same model. YOLOv5 runs in real-time on our 24fps dataset, and can likely reach much higher speeds.

Model Name	FPS	Notes
YOLOv5	Real-time	(Frame-rate of dataset was 24fps)
DeepLabv3	21	-
Hybridnets	10	CPU limited (can likely be optimized)

Table 5: Performance comparison of ML models

(a) Outdoors (b) Indoors

Figure 19: Hybridnet drivable area segmentation evaluated on the Curtin dataset

4.2 Hybridnets drivable area segmentation

The Hybridnets model was evaluated on the Cityscapes dataset before and after training. Testing the pretrained Hybridnets model on the BDD100K validation dataset gave a mIoU of 85.7%. The same model achieved a mIoU of 26.6% when evaluated on the Cityscapes dataset. This difference is due to the methodology behind the labelling of these datasets: Cityscapes labels all road surfaces as 'road', whereas BDD100K only labels the current lane as 'drivable'. This can be seen in fig. 20.

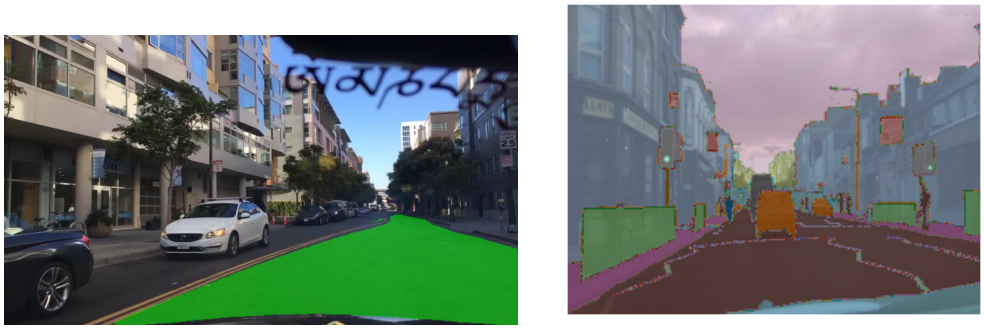
The recall for road detection on the Cityscapes dataset is 30.6% and the recall for sidewalk detection is much poorer at 1.52%. Table 6 shows the confusion matrix before retraining and demonstrates the low proportion of sidewalks compared to roads in this dataset.

	None	Road	Sidewalk
Non-drivable	81.0%	11.4%	2.33%
Drivable	0.197%	5.03%	0.036%

Table 6: Confusion matrix of Hybridnets model on Cityscapes dataset (before training)

The model generalises to the new dataset very quickly during training, as seen in fig. 21. The validation accuracy vastly increases after the first epoch, with not much change in mIoU after this point apart from a further drop in training loss (which can indicate overfitting).

After epoch 1, the mIoU is 87.5% (a 56.9% improvement), with a road detection recall of 96.2% and sidewalk detection recall of 68.2%. As seen in the training curve, the sidewalk detection recall varies epoch to epoch, with a variance of 4.2%. This higher variance is due to the lower occurrence of sidewalks in the dataset compared to roads.



(a) BDD100K (b) Cityscapes

Figure 20: BDD100K drivable area compared with Cityscapes road segmentation

Figure 22 shows an example of the retrained Hybridnets model compared to the pre-trained model when evaluated on the Curtin driving dataset. The new model identifies the paved drivable area further away from the wheelchair but struggles to identify the drivable area directly in front of the wheelchair. This is still an improvement compared to the pre-trained model, which does not identify any drivable area at all.

The retrained model (Cityscapes) produces a much noisier output than the pre-trained model (BDD100K). This could be due to the structure of the Cityscapes dataset - as pedestrian paths are often separated from the road by verges, the drivable area is more likely to be disconnected. This is another domain adaptation problem - although the model may identify pedestrian pathways easily from the viewpoint of a road, it can struggle with identifying these same pathways from a wheelchair user's perspective. This would also explain why the retrained model struggles to identify the paved drivable area directly in front of the wheelchair.

4.3 Efficiacy of birds-eye view occupancy map transform

Once the drivable area has been segmented, it is transformed into the XZ plane as a birds-eye view occupancy map. This is done by processing the 3D point cloud data from the ZED Mini camera. An example of the segmented output alongside the occupancy map can be seen in fig. 24, with the drivable area highlighted in green. Note that the occupancy map is 15 metres long and 10 metres wide. The location of the ZED Mini camera is depicted with an 'X'; the camera is mounted to the right-hand side of the wheelchair.

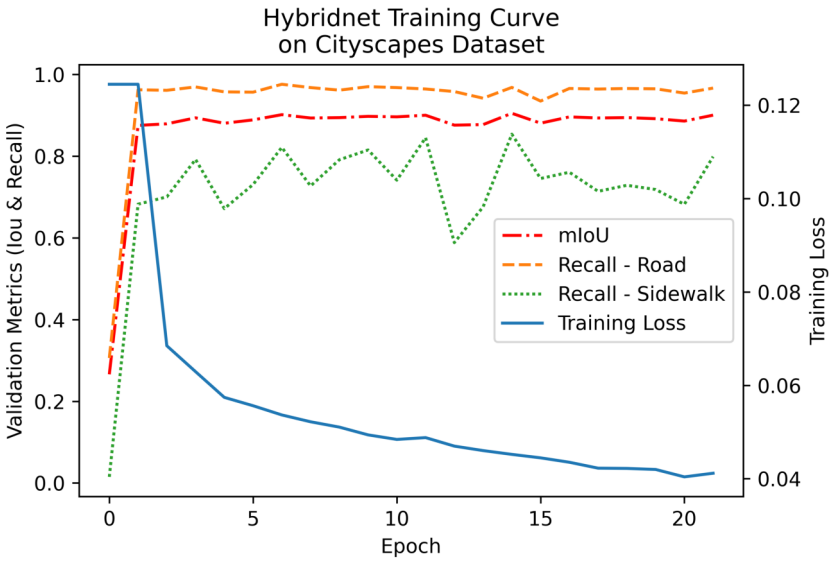
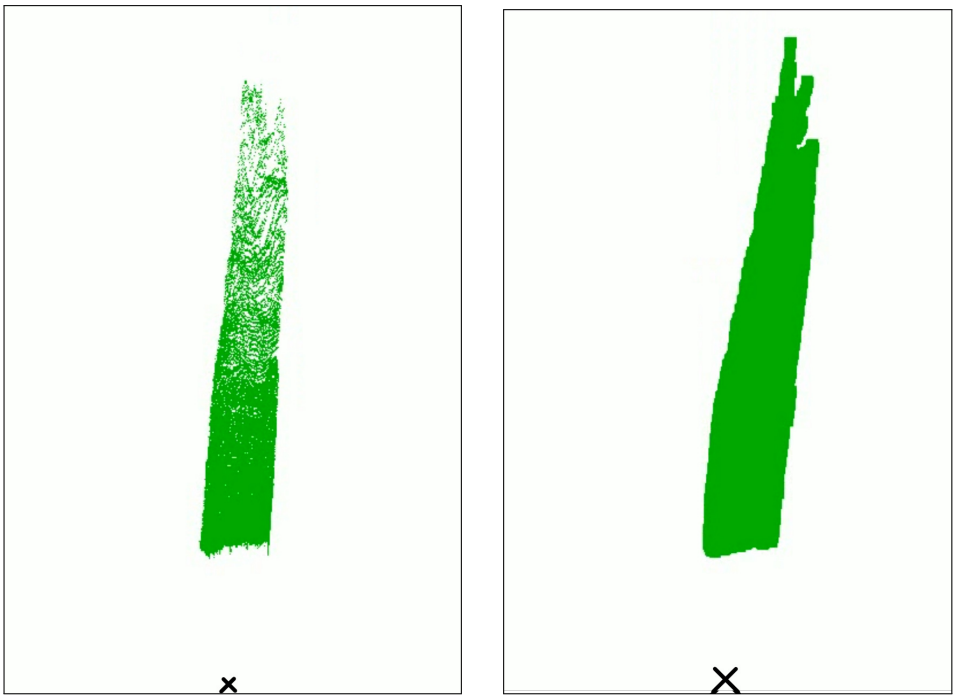


Figure 21: Hybridnets training curve

(a) Before retraining (b) After retraining

Figure 22: Hybridnets segmentation on the Curtin driving dataset



(a) Before processing (b) After processing

Figure 23: Comparison of occupancy map before and after morphological processing

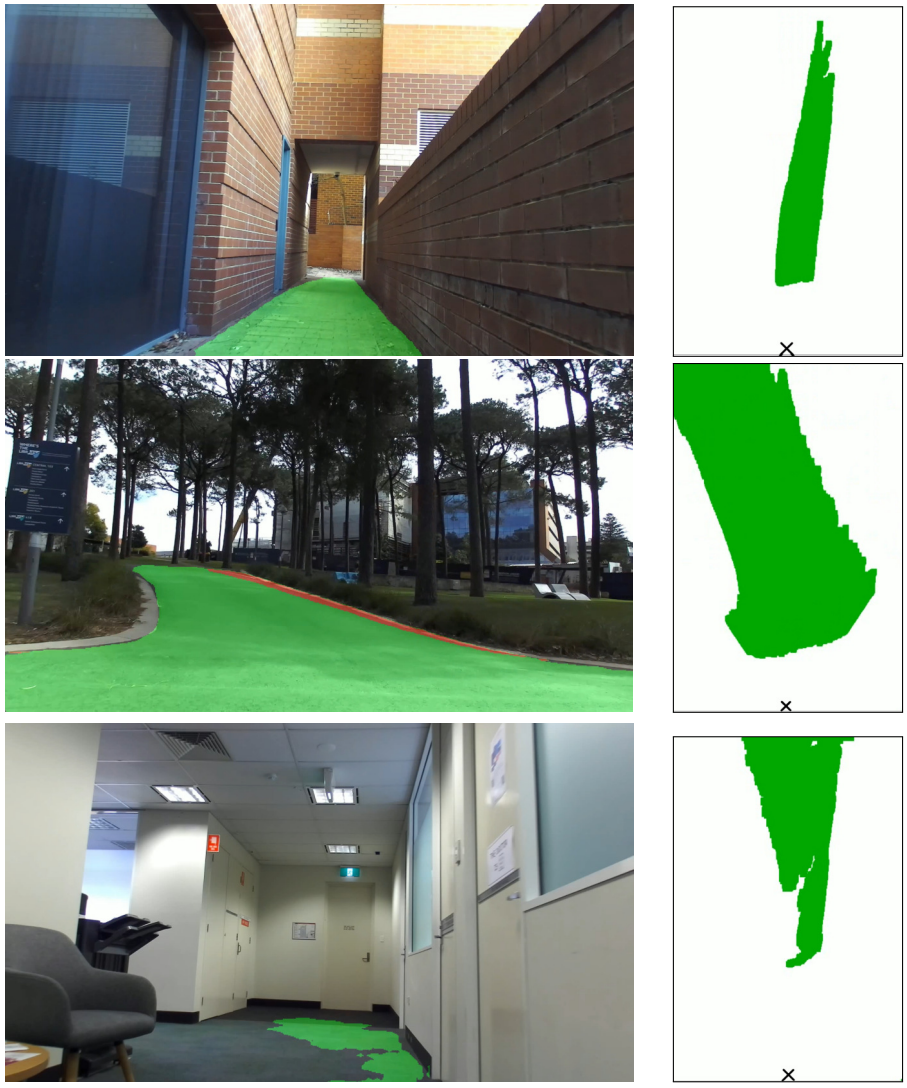


Figure 24: Segmented image output and corresponding occupancy map for three locations around Curtin University

The image to occupancy map transform works well in all observed cases, as long as the Hybridnets model identifies the drivable area accurately. One improvement that could be made to this implementation is its speed. The algorithm is bottlenecked by the occupancy map transform code, which takes between 150 ms and 500 ms to execute depending on the size of the drivable area.

Although this approach reliably maps drivable areas to an occupancy grid, the area directly in front of the wheelchair (approx. 2.6 m) and behind the wheelchair is unknown due to the FOV of the camera. Several approaches to rectify this are explored in the future work section of this thesis.

A comparison of the occupancy map before and after morphological processing is shown in fig. 23. Although the occupancy map has a higher resolution before this transformation, the low density of the map could make it more difficult to process. The time taken to perform this morphological processing is negligible when compared to other sections of this code.

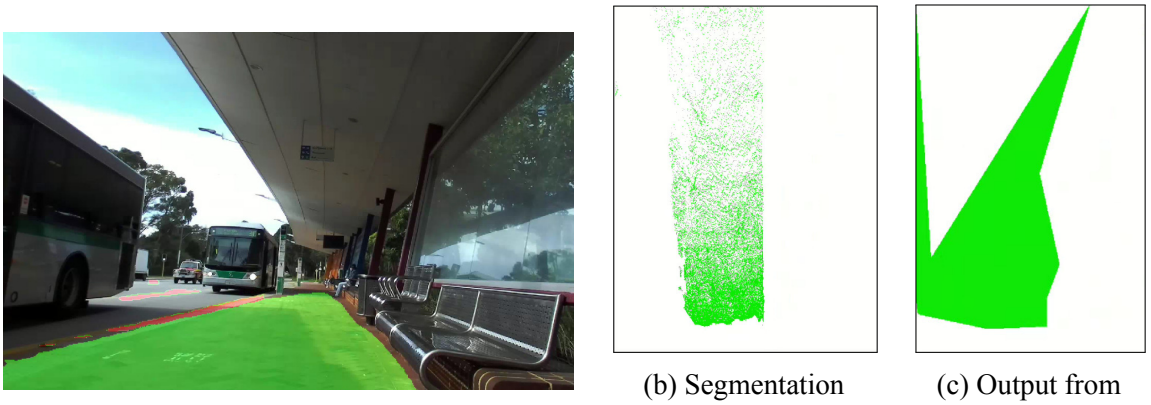
4.4 Evaluation of 3D point cloud obstacle detection algorithms

Direct processing of the 3D point cloud data was tested to identify environmental obstacles and drivable areas. One such approach involved using the inbuilt ZED SDK function `find_floor_plane` which outputs a polygon of the floor plane. This is shown in fig. 25 and compared alongside the segmentation occupancy map. This approach is very poor at identifying the floor plane accurately and has large frame-to-frame variation in its output. The only redeeming factor of this approach is its speed; as the function is GPU accelerated, this calculation takes approximately 60 ms.

Another approach that was tested was a custom algorithm to process the point cloud data, detailed in section 3.3.4. The results of this algorithm are shown in three scenarios: one indoor (fig. 26) and two outdoor (fig. 27 and fig. 28). Note that the drivable area is labelled in green and the environmental obstacles are labelled in red.

This algorithm works well in indoor settings (fig. 26) at identifying the drivable area as well as the position of walls and other static objects. Accurate occupancy maps are also generated for outdoor settings with well-defined boundaries, such as a wheelchair ramp with handrails (fig. 28). However, this algorithm can struggle with open outdoor settings as seen in fig. 27. Due to the uniformity of the pedestrian path in this image, the ZED Mini is unable to reliably generate a depth map and 3D point cloud for the path. This leads to the algorithm mistakenly identifying grassy and uneven terrain as drivable, but not identifying the path as drivable.

This algorithm is slower than the segmentation approach, with a mean latency of 550 ms. Due to this limitation, it is more suitable for identifying non-moving objects such as walls, and less suitable for identifying moving objects such as people and vehicles.

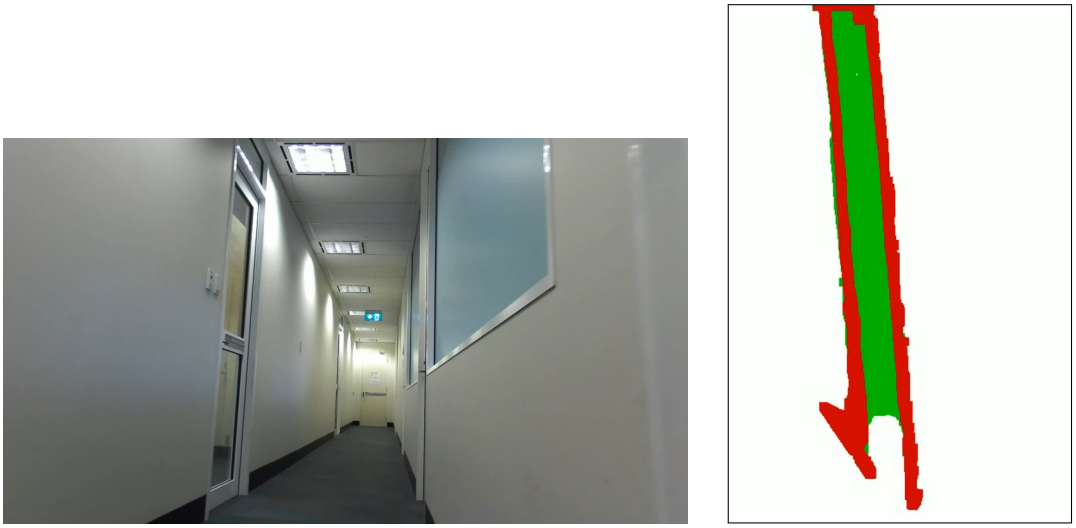


(a) Video footage

(b) Segmentation occupancy map

(c) Output from `find_floor_plane`

Figure 25: ZED SDK `find_floor_plane` function compared with segmentation occupancy map



(a) Video footage (b) Occupancy map

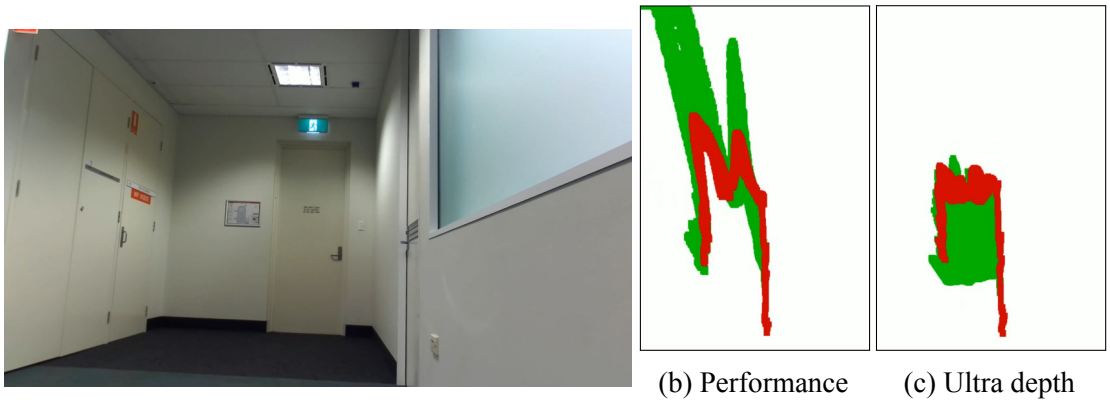
Figure 26: Point cloud processing result (Indoor)

(a) Video footage (b) Occupancy map

Figure 27: Point cloud processing result (Outdoor, uniform path)

(a) Video footage (b) Occupancy map

Figure 28: Point cloud processing result (Outdoor, wheelchair ramp)



(a) Video footage (b) Performance depth mode (c) Ultra depth mode

Figure 29: Comparison of ZED Mini depth modes

The ZED Mini has three depth modes: `ULTRA`, `QUALITY`, and `PERFORMANCE`. These depth modes trade performance for depth accuracy - a comparison between the occupancy maps generated by these modes can be seen in fig. 29. The `ULTRA` depth mode accurately identifies the back wall as flat. However, the `PERFORMANCE` depth mode does not identify the back wall as flat, and incorrectly classifies an area beyond the wall as drivable. Due to the high latency of the rest of this algorithm, the difference in speed between these two depth modes is negligible and hence the `ULTRA` depth mode was selected.

4.5 Evaluation of assistive control algorithm

VFH+ was implemented as a proof-of-concept assistive control algorithm. This algorithm blends the user's target direction with the occupancy map to determine a safe steering direction.

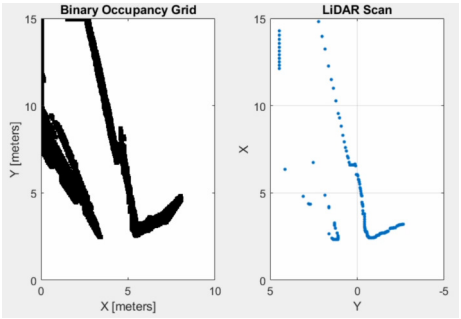
Figure 30 shows the VFH+ algorithm detecting an obstacle in front of the wheelchair and steering left to avoid this obstacle. This showcases the end-to-end navigation assistance pipeline, from environmental obstacle detection to avoidance of that obstacle using VFH+.

The VFH+ algorithm takes 240 ms to determine the steering direction, which increases to 430 ms when called from Python due to the overhead of the MATLAB Engine API. This latency could be a concern for a complete semi-autonomous wheelchair implementation; however, this algorithm is a proof of concept, and latency was not a concern during implementation. Additionally, VFH+ only adjusts the direction of the wheelchair and not the wheelchair's speed, making it unsuitable as a final assistive control algorithm.

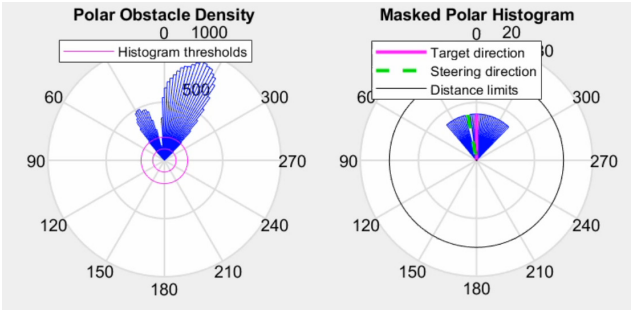
Due to the FOV of the camera, obstacles directly to the left or right of the wheelchair are not added to the occupancy map. Figure 31 demonstrates a scenario where this can become an issue. VFH+ mistakenly steers to the left to avoid the narrow hallway walls, which would cause a crash with the walls directly to the left of the wheelchair. This occurs because these walls are not in the occupancy map, and so the VFH+ algorithm assumes that there is free space in this area. Several approaches that could improve the occupancy map and rectify this issue are suggested in the future work section of this thesis.

(a) Video footage

(b) Occupancy map

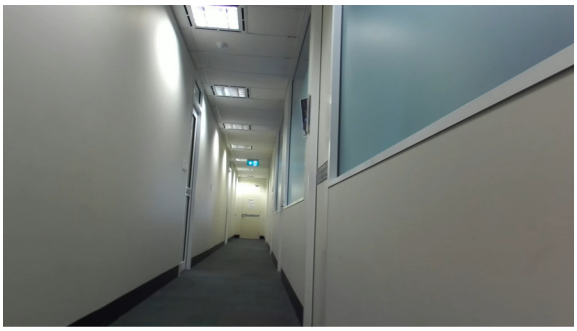


(c) Binary occupancy grid and LIDAR scan of obstacles

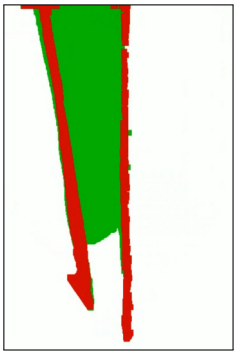


(d) Polar histogram of obstacles with target and steering directions

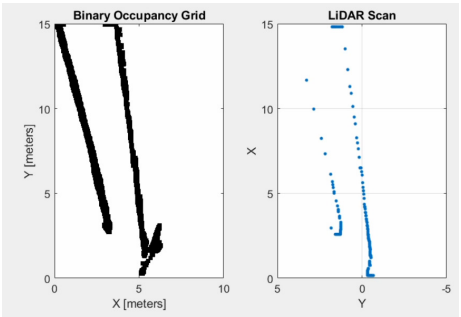
Figure 30: VFH+ changing direction to avoid an obstacle



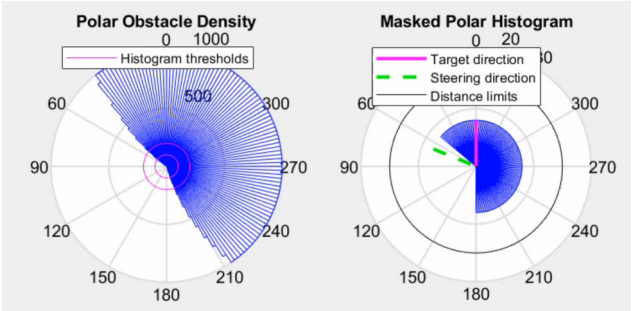
(a) Video footage



(b) Occupancy map



(c) Binary occupancy grid



(d) Polar histogram of obstacles

Figure 31: VFH+ mistakenly changing direction due to low sensor FOV

4.6 Evaluation of pose estimation APIs

The ZED Sensors API and ZED Positional Tracking API were tested to estimate the pose of the wheelchair.

The Sensors API was able to produce accurate estimates of the Euler angles of the ZED Mini. The camera has a slight tilt which was noted in the data collection methodology and measured as approximately 8.2° using point cloud data. The IMU measures this tilt (x-axis rotation) to be approximately 8.5° when travelling across flat ground, which agrees with the point cloud tilt measurement to within 0.3°.

The Sensors API performs much worse when measuring the overall distance travelled by the wheelchair. As part of the dataset, the wheelchair was driven along the length of the Curtin bus station in a straight line, for a distance of approximately 83 m (measured using Google Maps). The Sensors API is barely able to register any movement in the wheelchair and records a distance of 7.0 m in this time.

This error could be due to several factors. IMUs can have high position errors (drift) as the position is calculated by integrating the acceleration twice. Additionally, the sampling rate of the sensors is 800 Hz, however is recorded in the dataset at 30 Hz. The lower sampling rate could have helped these errors to propagate. In this scenario, the wheelchair was travelling at a relatively constant velocity, which would have caused the average acceleration to be close to zero throughout the trip. A small error in the initial velocity estimate would have led to a larger error in the resulting position estimate.

The Positional Tracking API, which combines sensor data with visual odometry, was also tested in this scenario. This API recorded a distance of 127 m, which is larger than the measured distance of 83 m by 53%. The wheelchair's movement is plotted in fig. 32 and scaled to show the actual path around the bus station; although the distance is not accurate, the estimated path taken by the wheelchair is relatively accurate. Note that the measured distance only consists of the path taken on the eastern side of the bus station.

A 'jump' can be seen in fig. 32; this can occur during visual odometry when the positional tracking algorithm attempts to correct the ZED Mini's location. The Positional Tracking API implements a parameter `enable_pose_smoothing` which is meant to smooth the impact of this jump over several frames. However, some testing indicates this feature is defective, as it produces completely unusable results when used alongside `enable_imu_fusion` in ZED SDK version 3.7.7.

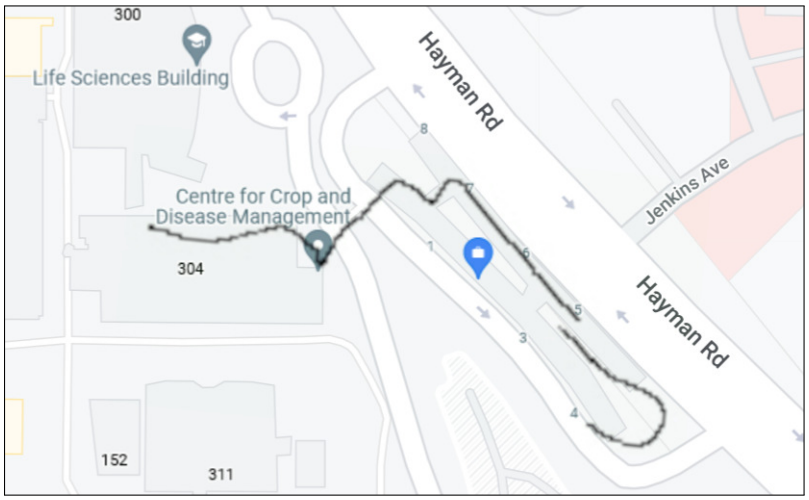


Figure 32: Movement of the wheelchair around the Curtin bus station, recorded using the Positional Tracking API. Map data: Google ©2022 [68]

5 CONCLUSION AND FUTURE WORK

5.1 Conclusion

The results show that the proposed navigation assistance system can successfully identify drivable areas and obstacles, place these drivable areas and obstacles on an occupancy map, and avoid obstacles using a semi-autonomous assistive control algorithm.

Retraining was able to improve the performance of the Hybridnets model on the Cityscapes dataset and enable it to correctly identify a wider range of non-uniform surfaces as drivable. Due to the content of the training dataset, Hybridnets has difficulty recognising drivable areas while indoors, making drivable area segmentation more suitable for outdoor use.

The custom 3D point cloud processing algorithm is effective when identifying static obstacles that are part of the surrounding environment, such as stairs, walls, and handrails. Due to the latency of this algorithm, it is not yet fast enough for use on moving obstacles such as pedestrians and vehicles. Machine learning approaches explored by Krishnadas Suresh (a project student within the smart wheelchair team) could be more suitable for moving objects such as pedestrians, as they can be cleanly identified using a bounding box.

The birds-eye view occupancy map was effective at combining information from several separate scene recognition algorithms into one map of the local environment. Morphological processing improved the density of the occupancy map, however, did impact the resolution of the map. Due to the FOV of the ZED Mini camera, the occupancy map did not contain obstacles behind, to the side of, or directly in front of the wheelchair.

VFH+ was used as a proof-of-concept assistive control algorithm and successfully steered the wheelchair to avoid obstacles identified in the occupancy map. As VFH+ does not change the speed of the wheelchair, it is not likely to be suitable for a final smart wheelchair implementation. However, this algorithm was effective at demonstrating the end-to-end navigation assistance pipeline from obstacle detection to avoidance.

Pose estimation is likely to form an important part of future mapping and control algorithms. The ZED Positional Tracking API was found to be much more effective than the ZED Sensors API at determining the absolute position of the sensor; although distance measurements were inaccurate, the path of the wheelchair was accurately estimated.

The Curtin University RGB-D wheelchair driving dataset was invaluable when testing various smart wheelchair algorithms, and a pre-existing dataset will aid future thesis project students in the development process. The navigation assistance technology developed and tested in this thesis project can be integrated with the work of other thesis students to create a semi-autonomous wheelchair system. This technology will grant safety and independence to wheelchair users.

5.2 Future work

Future work beyond this thesis will be carried out by the next group of thesis project students, in collaboration with Glide. The recommended future work involves improving the accuracy and speed of algorithms developed in this paper, developing new components required for a final smart wheelchair implementation, and integrating components developed by different project students into one cohesive system.

5.2.1 Improvements to navigation assistance system

Although retraining the Hybridnets model was effective at improving accuracy on pathways such as paved brick, this came at the cost of greater noise. The effect of this noise on model accuracy could not be quantitatively determined, as the full Curtin driving dataset was not labelled with drivable areas. Future work could involve labelling sections of the dataset to evaluate model performance; refer to appendix D for further comments on segmentation labelling. A drivable area labelled dataset could also be used to train the drivable area segmentation models, and would likely improve their performance indoors.

Several other models such as DeepLabv3 and YOLOP perform drivable area segmentation, however, could not be tested in this thesis due to time constraints. These models could be retrained and compared to Hybridnets to choose the most effective model for this task. Models such as YOLOP and Hybridnets can perform lane line segmentation as well as drivable area segmentation. This functionality could be repurposed during retraining to enable the segmentation and identification of kerbs, which can present a serious risk to a wheelchair user. Kerbs can be difficult to identify using 3D point cloud data as some kerbs only rise above the ground by a small amount.

The joystick controller of the wheelchair allows the user to choose between indoor and outdoor modes to control the speed of the wheelchair. This information could be intercepted using the input controller and passed to the software system, allowing different modes of operation for indoor and outdoor use. For example, it is likely that most indoor flooring will be drivable, making obstacle detection more important than drivable area detection in this setting. If drivable area detection was only enabled when in outdoor mode, the accuracy of the segmentation model in indoor settings would not be a concern.

The 3D point cloud obstacle detection algorithm was effective at identifying environmental obstacles such as walls and handrails. However, the latency of this algorithm can reduce its effectiveness in some scenarios. This could likely be addressed by rewriting the implementation in a faster language such as C++ or performing all computations on the GPU using a library such as PyTorch. The speed of the implementation was not a priority during

development, and as such, significant performance gains could be found by profiling the code. The Hybridnets drivable area algorithm is also bottlenecked by the occupancy map transformation code, and would see significant performance gains using the same techniques.

Another more general approach that could be used to improve the efficiency of a slow algorithm would be to cache results in the occupancy map. If the object is assumed to be unmoving, its position within the occupancy map can be updated using the wheelchair's change in position and heading (obtained using the positional tracking API). This allows different scene recognition algorithms to update the occupancy map at their own pace and not bottleneck one another.

The FOV of the camera reduces the number of obstacles that can be perceived near the wheelchair, which makes both assistive control and fully-autonomous control less effective. One approach to fix this would be to use hardware, either by adding a 2D LIDAR sensor to the rear of the wheelchair or adding several ultrasonic sensors to detect obstacles at the rear or side of the wheelchair. Another approach would be to cache objects in the occupancy map, as mentioned above, and use the positional tracking API to determine the wheelchair's movement and update their position.

A more robust solution would be to use an existing SLAM algorithm, such as ORB-SLAM2 [69], to update the occupancy map. This algorithm has the added benefit of performing loop closures, which would help to keep an accurate model of the surrounding environment. As the SLAM algorithm performs relocalization, the positional tracking API may not be necessary to determine the location of the wheelchair in this case.

5.2.2 Implementation of smart wheelchair system

Figure 33 shows the top-level block diagram for a future smart wheelchair system. The block diagram was planned alongside Krishnadas Suresh and Avinash Sudhakaran, and shows how the software components developed by thesis project students would work together in a complete smart wheelchair implementation. The default mode of operation of the smart wheelchair is semi-autonomous, in which path segmentation and point cloud analysis (explored in this thesis project) are used to generate a top-down occupancy map. Object detection, which was the focus of Krishnadas' thesis, would also be used to add moving objects such as pedestrians to the occupancy map. A SLAM algorithm could be used to update the occupancy map, as detailed above.

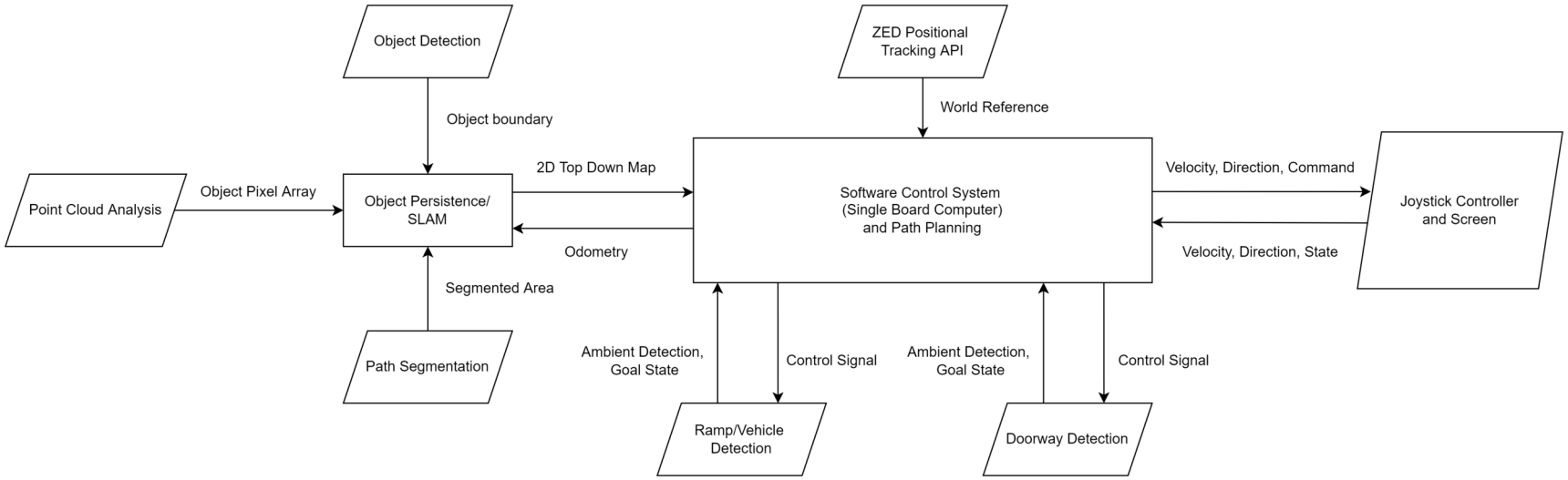


Figure 33: Block diagram for future smart wheelchair system (drawn by Avinash Sudhakaran)

Once this occupancy map is built, it is passed to the software control system and used to determine a safe speed and direction for the wheelchair. A proof-of-concept assistive control system was developed in this thesis using VFH+, but a more robust solution would likely be required for a final implementation. The wheelchair enters into fully autonomous control in two scenarios: ramp/vehicle navigation (Nicolas Lee), and doorway navigation (Avinash Sudhakaran). These fully autonomous navigation modes will require the user to press a button to begin autonomous navigation once a valid ramp or doorway is detected. These systems will send a goal pose to the software control system, which plans a safe path using the occupancy map and drives the wheelchair to this pose. Once this has been performed, semi-autonomous wheelchair control can resume.

To communicate between different software components, a publish/subscribe architecture is proposed. Using a framework such as ROS (robot operating system), each component of the smart wheelchair system would be defined as a node. These nodes can subscribe to other nodes to receive data, and publish data to a topic once processed. An advantage of this approach is that the interfaces between nodes are well-defined, which speeds up development when multiple project students are working on the same system.

Compute resources must be considered when designing a large system with many software components. If too many processes are using the CPU or GPU accelerator, the performance of the system could be impacted. In particular, an AI accelerator may not have enough memory or compute capability to run many complex models at once. One way to improve the performance of the AI accelerator would be to use a shared-backbone model architecture, which is used by Hybridnets and YOLOP. In this architecture, a complex model backbone is used to generate image features before several smaller ‘heads’ utilise these features for different purposes. This would allow several models (e.g. obstacle detection, drivable area detection, doorway detection) to run at once with only a small performance impact.

To communicate with the wheelchair hardware, an interface between the software control system and the input controller/joystick controller must be implemented. The simplest way to implement this interface would be to use a USB serial port to communicate between the microcontroller and SBC (single board computer). The input controller would send information such as joystick position, seat position, and operation mode (indoor or outdoor) to the SBC. The SBC would relay back the desired velocity and direction of the wheelchair. Additional low-bandwidth sensors, such as 1D ultrasonic sensors or wheel odometry sensors, could interface with the input controller which would relay this information back to the SBC. The input controller would validate the desired velocity and direction of the wheelchair

before sending this information to the motor controller, allowing it to override the SBC in the case of a software failure.

A secondary driving dataset could be collected once the input controller has been integrated into the software control system. This driving dataset would contain joystick data and other wheelchair data as well as the `svo` files used to record data from the ZED Mini. A simple method of storing joystick data would be to use a csv file; alternatively, a more complex data format such as HDF5 could be used to compress this data and store it in a binary format.

This thesis has involved implementation and testing of various components of smart wheelchair technology. The future work detailed in this section shows the steps needed to move from this initial research to a complete smart wheelchair system, which can improve the safety and independence of wheelchair users.

6 REFERENCES

[1] E. Trefler, S. G. Fitzgerald, D. A. Hobson, T. Bursick, and R. Joseph, "Outcomes of Wheelchair Systems Intervention With Residents of Long-Term Care Facilities," *Assistive technology*, vol. 16, no. 1, pp. 18–27, 2004, ISSN: 1040-0435. DOI: 10.1080/10400435.2004.10132071.

[2] L. Fehr, W. E. Langbein, and S. B. Skaar, "Adequacy of power wheelchair control interfaces for persons with severe disabilities: A clinical survey," *Journal of rehabilitation research and development*, vol. 37, no. 3, pp. 353–360, 2000, ISSN: 0748-7711.

[3] M. A. Eid, N. Giakoumidis, and A. El Saddik, "A Novel Eye-Gaze-Controlled Wheelchair System for Navigating Unknown Environments: Case Study With a Person With ALS," *IEEE access*, vol. 4, pp. 558–573, 2016, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2016.2520093.

[4] M. R. M. Tomari, Y. Kobayashi, and Y. Kuno, "Enhancing Wheelchair's Control Operation of a Severe Impairment User," in *The 8th International Conference on Robotic, Vision, Signal Processing & Power Applications*, H. A. Mat Sakim and M. T. Mustaffa, Eds., Singapore: Springer Singapore, 2014, pp. 65–72, ISBN: 978-981-4585-42-2. DOI: 10.1007/978-981-4585-42-2.

[5] M. Bakouri, M. Alsehaimi, H. F. Ismail, *et al.*, "Steering a Robotic Wheelchair Based on Voice Recognition System Using Convolutional Neural Networks," *Electronics (Basel)*, vol. 11, no. 1, p. 168, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11010168.

[6] L. Habha, U. Trivedi, R. Alqasemi, and R. Dubey, "Autonomous Wheelchair Indoor-Outdoor Navigation System through Accessible Routes," in *The 14th PErvasive Technologies Related to Assistive Environments Conference (PETRA 2021)*, Association for Computing Machinery, 2021, pp. 199–202, ISBN: 978-1-4503-8792-7. DOI: 10.1145/3453892.3453997.

[7] R. C. Simpson, E. F. LoPresti, and R. A. Cooper, "How many people would benefit from a smart wheelchair?" *Journal of rehabilitation research and development*, vol. 45, no. 1, pp. 53–72, 2008, ISSN: 0748-7711. DOI: 10.1682/JRRD.2007.01.0015.

[8] H. Wang, Y. Sun, R. Fan, and M. Liu, "S2P2: Self-Supervised Goal-Directed Path Planning Using RGB-D Data for Robotic Wheelchairs," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11 422–11 428, 2021. DOI: 10.1109/ICRA48506.2021.9561314. [Online]. Available: https://sites.google.com/view/s2p2.

[9] H. Wang, Y. Sun, and M. Liu, "Self-Supervised Drivable Area and Road Anomaly Segmentation Using RGB-D Data For Robotic Wheelchairs," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4386–4393, 2019, ISSN: 2377-3766. DOI: 10.1109/LRA.2019.2932874.

[10] S. Jain and B. Argall, "Automated perception of safe docking locations with alignment information for assistive wheelchairs," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2014, pp. 4997–5002. DOI: 10.1109/IROS.2014.6943272.

[11] M. Scudellari, "Self-driving wheelchairs debut in hospitals and airports [News]," *IEEE Spectrum*, vol. 54, no. 10, p. 14, 2017, ISSN: 0018-9235. DOI: 10.1109/MSPEC.2017.8048827.

[12] S. Levine, D. Bell, L. Jaros, R. Simpson, Y. Koren, and J. Borenstein, "The NavChair Assistive Wheelchair Navigation System," *IEEE Transactions on Rehabilitation Engineering*, vol. 7, no. 4, pp. 443–451, 1999, ISSN: 1063-6528. DOI: 10.1109/86.808948.

[13] A. Karpathy, director, *Tesla AI Day*, 2021. [Online]. Available: youtube.com/watch?v=j0z4FweCy4M.

[14] E. Wan and R. Van Der Merwe, "The unscented Kalman filter for nonlinear estimation," in *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium*, IEEE, 2000, pp. 153–158. DOI: 10.1109/ASSPCC.2000.882463.

[15] Stereolabs, *ZED Mini Camera and SDK Overview*, 2018. [Online]. Available: https://cdn.stereolabs.com/assets/datasheets/zed-mini-camera-datasheet.pdf.

[16] Stereolabs, *ZED 2 Camera and SDK Overview*, 2019. [Online]. Available: https://cdn.stereolabs.com/assets/datasheets/zed2-camera-datasheet.pdf.

[17] Intel, *Intel RealSense Product Family D400 Series Datasheet*, 2022. [Online]. Available: https://www.intelrealsense.com/wp-content/uploads/2022/03/Intel-RealSense-D400-Series-Datasheet-March-2022.pdf.

[18] Microsoft. "Azure Kinect DK Hardware Specifications." (2021), [Online]. Available: https://docs.microsoft.com/en-us/azure/Kinect-dk/hardware-specification.

[19] Nvidia, *Jetson Nano System-on-Module Data Sheet*, 2019. [Online]. Available: https://developer.nvidia.com/embedded/downloads.

[20] Nvidia, *Jetson Xavier NX Series System-on-Module Data Sheet*, 2019. [Online]. Available: https://developer.nvidia.com/embedded/downloads.

[21] Nvidia, *Turing GPU Architecture*, 2018. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[22] Google Coral, *Coral Dev Board Datasheet*, 2020. [Online]. Available: https://coral.ai/static/files/Coral-Dev-Board-datasheet.pdf.

[23] B. Jacob, S. Kligys, B. Chen, *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," 2017. DOI: 10.48550/ARXIV.1712.05877.

[24] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, pp. 1097–1105, 2012, ISSN: 0001-0782. DOI: 10.1145/3065386.

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, ISSN: 0018-9219. DOI: 10.1109/5.726791.

[26] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[27] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 2014. DOI: 10.48550/ARXIV.1409.1556.

[28] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going Deeper with Convolutions," 2014. DOI: 10.48550/ARXIV.1409.4842.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2016, IEEE, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[30] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2013.

[31] R. Girshick, "Fast R-CNN," in *2015 IEEE International Conference on Computer Vision*, vol. 2015, IEEE, 2015, pp. 1440–1448. DOI: 10.1109/ICCV.2015.169.

[32] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," 2015. DOI: 10.48550/ARXIV.1506.01497.

[33] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2009, ISSN: 0920-5691. DOI: 10.1007/s11263-009-0275-4.

[34] T.-Y. Lin, M. Maire, S. Belongie, *et al.*, "Microsoft COCO: Common Objects in Context," 2014. DOI: 10.48550/arXiv.1405.0312.

[35] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2015. DOI: 10.48550/arXiv.1506.02640.

[36] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," 2016. DOI: 10.48550/arXiv.1612.08242.

[37] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," 2018. DOI: 10.48550/arXiv.1804.02767.

[38] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," 2020. DOI: 10.48550/arXiv.2004.10934.

[39] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," 2015. DOI: 10.48550/ARXIV.1505.04597.

[40] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs," 2014. DOI: 10.48550/ARXIV.1412.7062.

[41] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs," 2016. DOI: 10.48550/ARXIV.1606.00915.

[42] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking Atrous Convolution for Semantic Image Segmentation," 2017. DOI: 10.48550/ARXIV.1706.05587.

[43] D. Wu, M. Liao, W. Zhang, *et al.*, "YOLOP: You Only Look Once for Panoptic Driving Perception," 2021. DOI: 10.48550/ARXIV.2108.11250.

[44] D. Vu, B. Ngo, and H. Phan, "HybridNets: End-to-End Perception Network," 2022. DOI: 10.48550/ARXIV.2203.09035.

[45] M. Cordts, M. Omran, S. Ramos, *et al.*, "The Cityscapes Dataset for Semantic Urban Scene Understanding," *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2016, pp. 3213–3223, 2016, ISSN: 1063-6919. DOI: 10.1109/CVPR.2016.350.

[46] F. Yu, H. Chen, X. Wang, *et al.*, "BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning," *arXiv*, 2018. DOI: 10.48550/ARXIV.1805.04687.

[47] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," *arXiv*, 2017. DOI: 10.48550/ARXIV.1711.03938.

[48] A. Elfes, "Using Occupancy Grids for Mobile Robot Perception and Navigation," *Computer (Long Beach, Calif.)*, vol. 22, no. 6, pp. 46–57, 1989, ISSN: 0018-9162. DOI: 10.1109/2.30720.

[49] S. Karaman and E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011, ISSN: 0278-3649. DOI: 10.1177/0278364911406761.

[50] A. Sakai, D. Ingram, J. Dinius, K. Chawla, A. Raffin, and A. Paques, "PythonRobotics: A Python code collection of robotics algorithms," 2018. DOI: 10.48550/ARXIV.1808.10703.

[51] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenét Frame," *IEEE International Conference on Robotics and Automation*, pp. 987–993, 2010, ISSN: 1050-4729. DOI: 10.1109/ROBOT.2010.5509799.

[52] I. Ulrich and J. Borenstein, "VFH+: Reliable Obstacle Avoidance for Fast Mobile Robots," *IEEE International Conference on Robotics and Automation*, vol. 2, pp. 1572–1577, 1998, ISSN: 1050-4729. DOI: 10.1109/ROBOT.1998.677362.

[53] MathWorks. "Vector Field Histogram." (2022), [Online]. Available: https://au.mathworks.com/help/nav/ug/vector-field-histograms.html.

[54] Y. Kondo, T. Miyoshi, K. Terashima, and H. Kitagawa, "Navigation Guidance Control Using Haptic Feedback for Obstacle Avoidance of Omni-directional Wheelchair," in *2008 Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, IEEE, 2008, pp. 437–444. DOI: 10.1109/HAPTICS.2008.4479990.

[55] E. B. Vander Poorten, E. Demeester, E. Reekmans, J. Philips, A. Huntemann, and J. De Schutter, "Powered wheelchair navigation assistance through kinematically correct environmental haptic feedback," *IEEE International Conference on Robotics and Automation*, pp. 3706–3712, 2012, ISSN: 1050-4729. DOI: 10.1109/ICRA.2012.6225349.

[56] W. Lucetti. "Stereolabs ZED Mini Holder." (2018), [Online]. Available: https://www.thingiverse.com/thing:3046034.

[57] Glide, *CentroGlide OWNER/USER MANUAL V 6.0*, 2022. [Online]. Available: https://static1.squarespace.com/static/5f58f6b0e418950766874381/t/624be15ab345a128a8a6a656/1649140064886/CentroGlide++2022+v6-0+Glide+Products.pdf.

[58] Curtiss-Wright, *PG DRIVES TECHNOLOGY R-NET TECHNICAL MANUAL*, 2016. [Online]. Available: http://sunrise.pgdrivestechnology.com/manuals/pgdt_rnet_manual_SK77981-14.pdf.

[59] Ultralytics, *YOLOv5*. [Online]. Available: https://github.com/ultralytics/yolov5.

[60] G. Bradski, *The OpenCV Library*, 2000. [Online]. Available: https://opencv.org/.

[61] A. Howard, M. Sandler, B. Chen, *et al.*, "Searching for MobileNetV3," *IEEE/CVF International Conference on Computer Vision*, vol. 2019, pp. 1314–1324, 2019, ISSN: 1550-5499. DOI: 10.1109/ICCV.2019.00140.

[62] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 8024–8035, 2019.

[63] NVIDIA, *Nvidia Tesla V100 GPU Accelerator*, Mar. 2018. [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf.

[64] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2014. DOI: 10.48550/ARXIV.1412.6980.

[65] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," 2017. DOI: 10.48550/ARXIV.1711.05101.

[66] Nvidia. "Vision Programming Interface: Pinhole Camera Model." (), [Online]. Available: https://docs.nvidia.com/vpi/appendix_pinhole_camera.html.

[67] Standards Australia, *AS 1428.1:2021 Design for access and mobility*, 2021.

[68] *Google Maps*, in collab. with Google, Curtin University, 2022. [Online]. Available: maps.google.com.

[69] R. Mur-Artal and J. D. Tardos, "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017, ISSN: 1552-3098. DOI: 10.1109/TRO.2017.2705103.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A - Thesis completion form

This form was signed by my academic supervisor rather than a Mechatronics engineering technical staff member, as all equipment used throughout the thesis was borrowed from the School of Electrical Engineering, Computing, and Mathematical Sciences (EECMS).

THIS PAGE REDACTED DUE TO SENSITIVE INFORMATION

APPENDIX B - Datasets and code

All supplementary material for this thesis has been submitted on a USB thumbdrive. Some supplementary material has also been uploaded to the 'Curtin X Glide (Smart Wheelchair)' Teams channel so that it can be used by future thesis students working on this project. Links to each of these files are provided.

- The preliminary Curtin University video-only driving dataset is hosted on Teams.
- The ZED Mini RGB-D Curtin University driving dataset is also hosted on Teams.
- The BDD100K and Cityscapes datasets used to train and evaluate the segmentation model have been uploaded to Teams.

Some material specific to this thesis has also been uploaded to the Teams channel. This material includes:

- **datasets/zed/labelled**: Drivable area annotations for a portion of the Curtin RGB-D driving dataset. Refer to appendix D for further information.
- **datasets/zed/video**: The Curtin RGB-D driving dataset converted into the avi video format, for use with devices that may not have the ZED SDK installed.
- **models**: Model weights for trained ML models, saved at each training epoch.
- **results**: Video results for segmentation, obstacle detection, and assistive control algorithms (tested on the Curtin RGB-D driving dataset).

All code used in this thesis can be found on Github. This repository contains:

- **code**: Python and MATLAB code used to implement various algorithms.
- **colab**: Colab notebooks used to train the segmentation model.
- **environments**: Conda environments used to manage dependencies.
- **models**: 3D models for the ZED mount (Inventor, Fusion, STL).
- **repositories**: Git submodules and sample code.

This repository also contains the latex source code used to format this document.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C - Manual for assistive control system

Appendix B details the location of the code and datasets used in this thesis project. This appendix outlines how to run the code and reproduce the results found in this thesis.

First, install the ZED SDK on your machine. A windows machine is recommended, and a CUDA-enabled GPU is necessary to install this SDK. CUDA GPU drivers will be installed automatically alongside the ZED SDK. Additionally, install Miniconda for python package management. Clone the GitHub repository containing the code and download submodules by using the commands in listing 1.

```
git clone <URL>
git submodule init
git submodule update
```

Listing 1: Update git submodules

All code is run using the `environments/environment-glide.yml` conda environment. Listing 2 shows the necessary commands used to create and activate this environment. The ZED SDK python API should also be installed into this environment so that it can be imported within the python scripts. Additionally, the `opencv-python-headless` package is uninstalled so that OpenCV can create windows and display videos.

```
conda env create -f environment-glide.yml
conda activate glide
cp 'C:\Program Files (x86)\ZED SDK\get_python_api.py'
↪  get_python_api.py
python get_python_api.py
pip uninstall opencv-python-headless
```

Listing 2: Create conda environment

Once the environment is set up, python scripts can be run to demonstrate different algorithms. Listing 3 describes the purpose of each script and how they can be invoked. Many of these scripts output the displayed video stream to a file in the current working directory after processing has finished.

```
# Tests segmentation and object detection algorithms on video
↪ dataset. --weights is optional and specifies which trained model
↪ weights should be used. --drop-frames is optional and used to
↪ run segmentation in real time.
python segmentation.py --source <dataset>.mp4 --model
↪ hybridnets|deeplab|yolo --weights hybridnet_epoch_1.pth
↪ --drop-frames

# Exports a proprietary .svo file to a .avi video file
python svo_export.py <input>.svo <output>.avi 0

# Connects to the ZED Mini camera and starts a new recording
python zedm_record.py <record>.svo

# Performs segmentation and generates occupancy map
python zedm_stream_seg.py --source <input>.svo

# Performs point cloud obstacle detection and generates occupancy
↪ map. Passes occupancy map to VFH+ controller and records
↪ results. Requires MATLAB R2022b, Navigation toolbox, and MATLAB
↪ Engine API for Python to be installed.
python zedm_stream_pointcloud.py --source <input>.svo

# Outputs ZED Mini translation and Euler angles using Sensors API
python zedm_sensors.py --source <input>.svo

# Plots ZED Mini movement on 2D map using Positional Tracking API
python zedm_positionaltracking.py --source <input>.svo
```

Listing 3: Command line arguments for python scripts

The ‘colab’ folder in the GitHub repository contains notebooks that can be uploaded to Google Colab to reproduce model training. Note that running these notebooks will require:

1. Mounting Google Drive to the Colab notebook.
2. Uploading the BDD100K and Cityscapes datasets to Google Drive. This may require Google One for additional storage.
3. Access to an Nvidia V100 GPU (or similar GPU with 32GB of memory). Colab Pro may be required for access to more powerful GPUs.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D - Dataset labelling results

A 6 min segment of the Curtin driving dataset was coarsely labelled with drivable areas using the V7 Labs Darwin labelling software and was uploaded to the Teams channel. A total of 348 images were extracted from the video (one per second) and each was labelled using the polygon tool. Polygon annotations are preferable to bounding box annotations for image segmentation as drivable areas cannot be cleanly identified with a rectangle. These annotations were exported in both MS COCO json format and image segmentation mask format - an example of the annotated dataset is provided in fig. 34.

One use of this annotated dataset could be to quantitatively evaluate the segmentation model on the Curtin driving dataset, which was not explored in this thesis due to time constraints. A larger amount of annotated data would enable retraining of the drivable area model on the Curtin dataset, which could potentially improve the performance of the model.

The V7 Labs Darwin labelling software provides an intuitive user interface and allows a maximum of three users per team for their education plan. It should be noted that this software automatically publishes training data and annotations under a Creative Commons license when using the education plan. Alternative labelling software that could be investigated includes CVAT, Roboflow, or Labelbox.

Figure 34: Curtin RGB-D dataset with drivable area annotation