Thomas Pietsch, BSc.

Design and Implementation of an Electronic Travel Aid for Visually Impaired based on Google Tango

Master Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Dr.techn. Norbert Druml (Infineon Technologies Austria AG)

Graz, November 2017

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

In 2014, there were 285 million Visually Impaired (VI) people worldwide estimated by the World Health Organization (WHO), with 246 million having low vision and 39 million being blind. It is a challenging task for the VI to safely and quickly navigate through an unknown or unfamiliar environment. The most common and reliable travel aids are still the traditional white cane or a guide dog. With the traditional white cane, collisions can only be detected once they already occurred. While guide dogs are able to guide the user and avoid collisions, they are rather expensive, require massive amounts of training, and the VI might not want to or be able to care for an animal.

The recent improvement of Time-of-Flight (ToF) sensors made them much more viable for use in a discreet, yet highly functional, Electronic Travel Aid (ETA). With the power usage, as well as the physical dimensions going down, it is even possible for them to be included into modern smart phones. Current ETAs are rather large and have high power usage.

In the course of this thesis a proof-of-concept ETA on a Google Tango enabled device - the Lenovo Phab 2 Pro - was developed. The Lenovo Phab 2 Pro is an affordable smart phone with an integrated ToF sensor and OpenCL capabilities. Using some constraints about the ground floor allows to robustly detect it using a combination of v disparity and RANdom SAmple Consensus (RANSAC). Further, once the ground floor is detected, all remaining points are considered to be an obstacle. The obstacle point cloud has to be further processed to allow for a meaningful warning. Therefore, a so-called Conservative Polar Histogram was developed. With the help of the Conservative Polar Histogram the closest obstacle in each direction is detected and a tactile and/or acoustic warning is given to the user. Google Tango provides depth images at 5 Frames Per Second (FPS), while the implementation of our method is capable of handling about 7 FPS on the used hardware. However, these numbers can be severely sped up by further exploiting the parallel computing power of the Lenovo Phab 2 Pro.

Kurzfassung

Im Jahr 2014 waren laut Schätzungen der World Health Organization (WHO) weltweit 285 Millionen Menschen sehbehindert, 246 Millionen davon sehschwach und 39 Millionen blind. Für Sehbehinderte ist es sehr schwierig sich durch unbekannte bzw. ungewohnte Gegenden oder Räume zu bewegen. Derzeit sind der Blindenstock oder der Blindenhund immer noch die meist verwendeten Hilfsmittel. Allerdings werden Kollisionen mit dem Blindenstock erst erkannt, wenn sie bereits geschehen sind. Obwohl Blindenhunde zwar Kollisionen erkennen und verhindern können, sind sie sehr teuer, benötigen enorme Mengen an Training, und ein Sehbehinderter ist eventuell nicht in der Lage für ein Tier zu sorgen.

Neueste Verbesserungen im Bereich der Time-of-Flight (ToF) Sensoren machten es praktikabel diese als diskreten, jedoch dabei höchst funktionellen, Electronic Travel Aid (ETA) zu nutzen. Sowohl der Stromverbrauch, als auch die physikalischen Abmessungen der Sensoren sind so gering, dass sie sogar in Smartphones integriert werden können. Bisherige ETAs sind sehr groß, unhandlich und haben einen hohen Stromverbrauch.

In dieser Arbeit wurde ein proof-of-concept ETA auf einem Google Tango fähigem Smartphone - dem Lenovo Phab 2 Pro - entwickelt. Das Lenovo Phab 2 Pro ist ein erschwingliches Smartphone mit einem eingebauten ToF Sensor und OpenCL Fähigkeiten. Mit der Annahme von Restriktionen bezüglich des Bodens ist es möglich den Boden auf eine robuste Art und Weise zu detektieren, indem man eine Kombination von v disparity und RANdom SAmple Consensus (RANSAC) anwendet. Sobald die Punkte die zum Boden gehören selektiert wurden, werden alle weiteren Punkte als Hindernisse angesehen. Die Hindernis-Punktwolke wird dann weiterverarbeitet, um eine sinnvolle Warnung ausgeben zu können. Dazu wurde das Conservative Polar Histogram entwickelt. Mit dem Conservative Polar Histogram wird das nächste Objekt in jede Richtung erkannt und eine fühlbare und/oder akustische Warnung wird abgegeben. Google Tango liefert Punktwolken mit 5 Frames Per Second (FPS) und unsere Implementierung ist in der Lage die Punktwolke mit rund 7 FPS zu verarbeiten. Durch eine parallele Implementierung der notwendigen Transformationen könnte dies jedoch noch verbessert werden.

Contents

Abstract iii

Kurzfassung iv

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Outline . . . 2

2 Related Work 5
- 2.1 Google Tango . . . 5
  - 2.1.1 Motion Tracking . . . 5
  - 2.1.2 Area Learning . . . 6
  - 2.1.3 Depth Perception . . . 7
- 2.2 Depth Based Travel Aids for the Visually Impaired . . . 9
  - 2.2.1 RGB-D camera . . . 11
  - 2.2.2 Stereo based . . . 15
  - 2.2.3 Ultrasonic sensor based . . . 17
- 2.3 Computer Vision Technique Based Travel Aids for the Visually Impaired . . . 18
- 2.4 Other Aids for the Visually Impaired . . . 19
- 2.5 Warning Interfaces . . . 19
- 2.6 3-d Spatial Audio . . . 20
  - 2.6.1 Pan and Fade . . . 20
  - 2.6.2 Head-Related Transfer Function . . . 21
- 2.7 Algorithms Used or Tested . . . 21
  - 2.7.1 V Disparity . . . 21
  - 2.7.2 Random Sample Consensus (RANSAC) . . . 23
  - 2.7.3 Vector Field Histogram (VFH) . . . 23

3 Design 27
- 3.1 Requirements and Constraints . . . 27

3.2 Ground Plane Detection . . . 28
3.2.1 V Disparity Map . . . 29
3.2.2 RANSAC . . . 31
3.3 Obstacle Detection and Warning . . . 31
3.3.1 Obstacle Detection . . . 33
3.3.2 Obstacle Warning . . . 36
3.4 Software Design . . . 39
3.4.1 Point Cloud Provider . . . 39
3.4.2 Ground Detection Module . . . 42
3.4.3 Warning Module . . . 45

4 Implementation 47
4.1 Hardware Setup . . . 47
4.2 Development Environment . . . 49
4.2.1 Android Native Development Kit (NDK) . . . 49
4.2.2 Java Native Interface (JNI) . . . 52
4.2.3 Google Tango SDK . . . 53
4.2.4 Khronos OpenCL (on Android) . . . 58
4.2.5 Google Cardboard VR Audio . . . 63
4.2.6 Point Cloud Library (PCL) on Android . . . 64
4.2.7 OpenCV . . . 65
4.3 Software Components . . . 66
4.3.1 Java Components . . . 66
4.3.2 virtualWhiteCane Library . . . 67
4.3.3 groundPlaneDetection Library . . . 72
4.3.4 virtualWhiteCaneUtil Library . . . 75
4.3.5 warningModule Library . . . 78
4.4 Algorithm Implementation Details . . . 80
4.4.1 V Disparity in OpenCL . . . 80
4.4.2 Conservative Polar Histogram . . . 81

5 Results 89
5.1 Ground Floor Detection . . . 89
5.2 Conservative Polar Histogram . . . 97

6 Conclusion and Future Work 101
6.1 Conclusion . . . 101
6.2 Future Work . . . 102

Bibliography 105

List of Figures

1.1 Picture of the Lenovo Phab 2 Pro. 2

2.1 Drift Correction. 6
2.2 Depth from Stereo Principle. 8
2.3 Time of Flight Principle. 10
2.4 ToF Phase Difference Control Signals. 10
2.5 Example disparity map I_Δ and $I_{v\Delta}$. 22

3.1 Ground Detection Flowchart. 30
3.2 V disparity comparison, if device is not held parallel to the ground. 32
3.3 Horizontal lines on the checkerboard have the same depth. 33
3.4 Example Conservative Polar Histogram. 36
3.5 Obstacle Detection Flowchart. 37
3.6 Process architecture of the system. 40
3.7 Dataflow diagram of the application. 41
3.8 `onPointCloudAvailable` update routine. 43
3.9 Ground Detection Module routine. 44

4.1 Software stack of the Virtual White Cane Application. 49
4.2 The Android Stack. 50
4.3 The Native Development Kit (NDK) Build Process. 51
4.4 The Java Native Interface (JNI) interface pointer. 52
4.5 The Android application stack using JNI. 54
4.6 Google Tangos 'Right Hand Local Level' coordinate system. 55
4.7 Google Tangos 'Right Hand Android' coordinate system. 56
4.8 The OpenGL coordinate system. 58
4.9 The OpenCL Platform Model. 60
4.10 Example NDRange index space. 60
4.11 OpenCL Memory Model. 62
4.12 Typical interaction between Android and OpenCL. 63
4.13 Point Cloud Library (PCL) dependency graph 64
4.14 UML of the Java part of the application. 68

4.15 UML of the `virtualWhiteCane` library. 71
4.16 UML of the groundPlaneDetection library. 74
4.17 UML of the `virtualWhiteCaneUtil` library. 76
4.18 UML of the `warningModule` library. 79
4.19 V Disparity Kernel Layout. 83
4.20 Conservative Polar Histogram Kernel Layout. 86

5.1 Comparison between just v disparity, just RANSAC, and combination of both. 90
5.2 Cluttered scene comparison. 91
5.3 Large object in front of user. 91
5.4 Pitched device, floor segmentation comparison. 92
5.5 Detecting thin objects like a pole. 93
5.6 Downstairs scene, the stairs are considered as obstacles. 94
5.7 Our method on a downstairs scene while standing very close to the stairs. 94
5.8 v disparity map of a downstairs scene. 94
5.9 Upstairs scene, the stairs are considered as obstacles. 95
5.10 v disparity map of upstairs scene. 95
5.11 The Time-of-Flight (ToF) sensor has a larger Field of View (FOV) than the color camera. 98
5.12 Conservative Polar Histogram result. 99
5.13 Conservative Polar Histogram, multiple obstacles in same bin. . . . 99

List of Abbreviations

ABI	Application Binary Interface
adb	Android Debug Bridge
ADF	Area Description File
API	Application Programming Interface
BOVW	Bag-of-Visual-Words
EOA	Electronic Orientation Aid
ETA	Electronic Travel Aid
FOV	Field of View
FPS	Frames Per Second
GPGPU	General-purpose Computing on Graphics Processing Units
GPS	Global Positioning System
HRTF	Head-Related Transfer Function
IMU	Inertial Measurement Unit
JDK	Java Development Kit
JNI	Java Native Interface
NDK	Native Development Kit
OCR	Optical Character Recognition
PCL	Point Cloud Library
PLD	Position Locator Device
RANSAC	RANdom SAmple Consensus
RGB-D	Color and Depth sensing Camera

SURF	Speeded Up Robust Feature
SVM	Support Vector Machine
ToF	Time-of-Flight
TTS	Text-To-Speech
VCP	Vehicle Center Point
VFH	Vector Field Histogram
VI	Visually Impaired
VM	Virtual Machine
WHO	World Health Organization

Chapter 1

Introduction

1.1 Motivation

According to the World Health Organization (WHO) there were about 285 million Visually Impaired (VI) people in 2014, with 246 million of those having low vision and 39 million being blind [1]. Safely navigating in an unknown environment is a very challenging task for visually impaired people.

Although various different electronic travel aids have been developed [2], [3], [4], most of them never got past the prototype stages. Additionally, they are simply not small or cheap enough for everyday use, as they mostly use a powerful laptop somehow attached to the user or do not satisfy all needs of the visually impaired [5]. Therefore, the most common travel aids are still the traditional white cane or a guide dog. However, guide dogs are expensive and require massive amounts of training, and, additionally, caring for an animal might not even be possible or desired for the visually impaired owner. As the white cane is a cheap, reliable, and portable device it is not our intention to fully replace it, but to enhance the travelling experience by providing additional information.

With the recent release of Google Tango [6] enabled mobile phones [7], [8], reasonably priced devices with high computational power and range sensing capabilities at a very small and convenient form factor have become available. Figure 1.1 shows a picture of the Lenovo Phab 2 Pro used for this thesis.

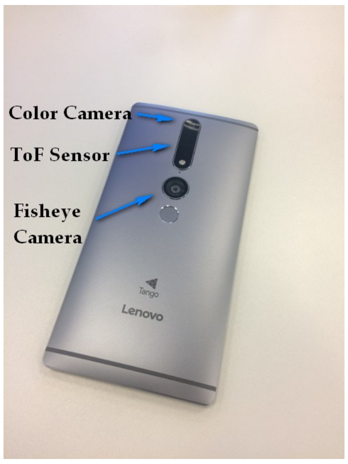


Figure 1.1: Picture of the backside of the Lenovo Phab 2 Pro. Sensors seen include a color camera, a ToF sensor, and a monochromatic fish eye camera.

1.2 Objectives

The focus of this thesis is on the design and implementation of a proof-of-concept "Virtual White Cane" application on a Google Tango enabled device. The extent of work can be divided into the following parts:

- Literature research on similar electronic travel aid projects
- Design and implementation of a real-time ground plane detection algorithm on a hand-held device
- Design and implementation of a warning module to provide visually impaired people with a mental image of their surroundings

The result of this thesis is an application showcasing what a future mobile electronic travel aid could look like. It is able to robustly detect the safe ground floor and obstacles in the path of the user. Different warnings, acoustic as well as tactile responses, are given to the user in case of a detected obstacle.

1.3 Outline

The rest of this thesis is structured as follows:

Chapter 2 provides information about other electronic travel aids and the algorithms they used. Further, it introduces the reader to the core Application

Programming Interfaces (APIs) and frameworks used in this thesis, as well as describing the algorithms, which are the basis of this work.

Chapter 3 elaborates how the system's components interact with each other. It gives details about the requirements and constraints for the Virtual White Cane. The purpose of each model is explained and how they are designed to be interchangeable, if the need arises.

The implementation Chapter 4 goes into further details and specifics about the software implementation on the Lenovo Phab 2 Pro. It highlights the hardware specifications and how such a system can be implemented on an Android device. The most important APIs and how they interact with Android are displayed. Afterwards, the software components and their interfaces are explained in detail. Lastly, it is shown how the algorithms are implemented in OpenCL to improve their performance and which problems arise developing GPU code for a mobile device.

The last chapters of this thesis, Chapter 5 and Chapter 6, summarize the results, as well as where the used algorithms excel at, likewise where their shortcomings are. It will also be mentioned how the system could be improved or extended.

Chapter 2

Related Work

This chapter provides an overview about other electronic aids for visually impaired users and how they detect obstacles. It will further provide information about the algorithms used in this thesis and explain the basics about Google Tango, which lies at the core of the application.

2.1 Google Tango

Google's Tango [6] gives mobile devices the ability to determine, understand, and track their position relative to the world around them. This is achieved using various computer vision concepts. To be able to use Google Tango, you need a device such as the Lenovo Phab 2 Pro [7], which has an array of sensors - for instance a wide-angle camera, a depth sensing camera, as well as accurate sensor timestamping. Google Tango builds on the following three main concepts:

- Motion Tracking
- Area Learning
- Depth Perception

2.1.1 Motion Tracking

Motion tracking allows a Tango-enabled device to track its motion. Using the device's Inertial Measurement Unit (IMU) paired with feature tracking from the wide-angle camera, Google Tango provides full 6-degrees-of-freedom (DOF) pose data in relation to a reference frame. The so-called start of service reference frame is always located at $(0, 0, 0)$, and it is initialized when the device is connected to the

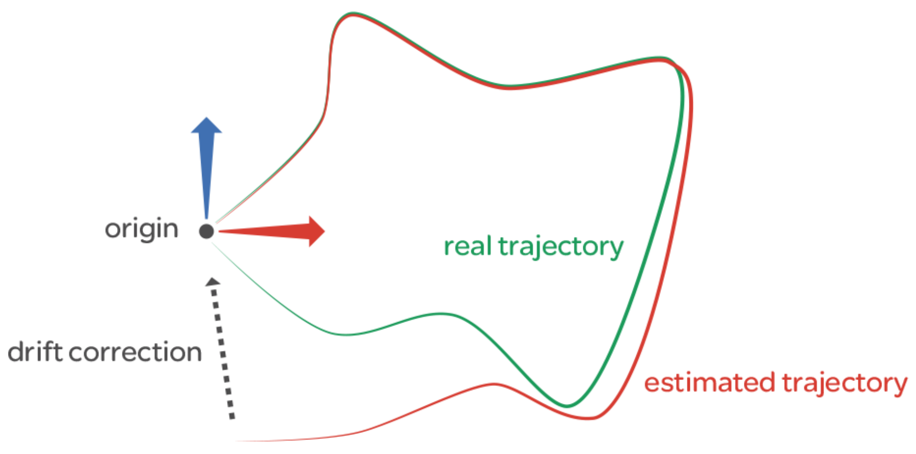


Figure 2.1: Measurements from Motion Tracking introduce a drift. To compensate for that error ADFs are used to detect and correct this drift as soon as an observed landmark is seen again. (Figure from [6])

Tango Service and it starts to track its motion. Pairing the IMU with the features from the fish eye lens camera improves the tracking. However, this technique still introduces inaccuracies. Over long periods of time and distances small errors in the measurements are accumulated and cause the measurements to "drift" and therefore produce a larger error in absolute position.

2.1.2 Area Learning

To help correct the error introduced by Motion Tracking, Area Learning can be used. When Area Learning is turned on, a so-called Area Description File, or ADF, is created. This file contains information of visual landmarks, like corners or other distinct features. Whenever the device "sees" a landmark it has already visited, a so-called loop-closure or drift correction is performed. Due to the fact that the landmark has already been seen, its position in the world is known. This can be used to correct the drift, which occurred in the motion tracking (see Figure 2.1).

Another important advantage of area learning is localization. Localization is the ability to recognize an area the device has already been in and position itself correctly within the area. This basically takes three steps:

1. Be in a learned area
2. Load the appropriate ADF
3. Localize

Localization enables you to remember where certain (virtual) objects are placed, or to label certain rooms/position such that it is possible to navigate the user to them. The quality of the ADF highly depends on the learning process, as only places the device has seen can be recognized. For example, with only one image of a room a very limited view is presented, while taking various pictures from different angles will provide a much better impression of what the room actually looks like. Likewise, the amount of features in a room influences the quality of the ADF. A long corridor with white walls will have less features than a room with lots of different (static) furniture. For localization purposes it is best that few pieces of furniture are moved to keep the same appearance. Lastly, the lighting conditions further influence the presentation of features. A feature might look different under the brightest and lowest lighting conditions. This problem can be solved by having different ADFs for different times of the day.

2.1.3 Depth Perception

Depth perception makes use of a 3-d depth sensor. This can be a stereo camera setup, or an infra-red light based technique using either Structured Light or Time-of-Flight. Depth perceptions allows the device to understand distances in the real world. This enables the device to interact with the real world, such as detecting a wall, or measuring the distance between two points. The 3-d depth sensor produces a point cloud, which can be used in conjunction with the color camera to produce a textured point cloud or mesh.

Depth from Stereo

Stereo vision systems extract depth information from digital color images. This is done by finding point correspondences in images from two different vantage points. A point correspondence is defined as two points x and x' in the image plane, which are both projections of the same 3-d point X. The easiest way to obtain these images is by having two cameras rigidly connected with a known distance between them, however, it does not matter whether the images are obtained by one camera moving or a rigid stereo rig. With the help of epipolar geometry these point correspondences can be found very efficiently [9]. A stereo rig is similar to

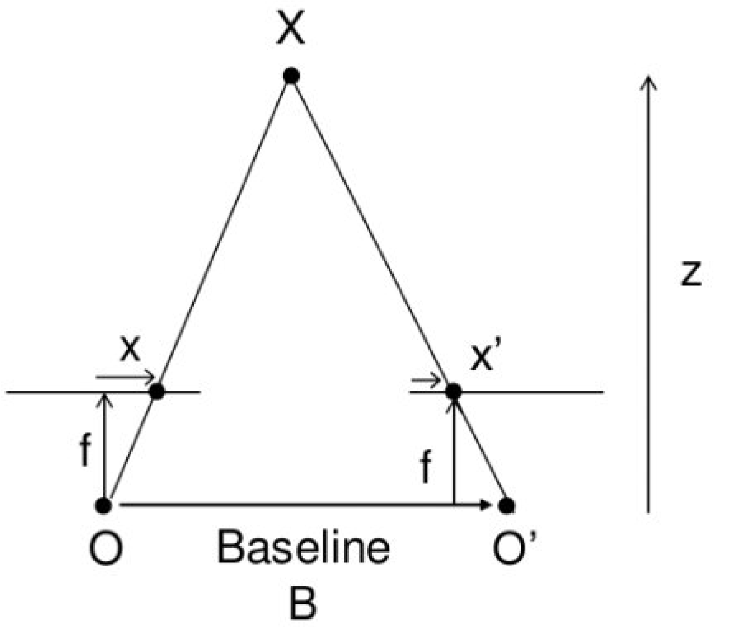


Figure 2.2: Using two images from the same scene from different vantage points it is possible to calculate the depth using Equation 2.1. (Figure from [10])

how human vision works. Obtaining the depth information of a scene is done via triangulation after a point correspondence is found.

Equation 2.1 shows that the depth Z of a point X in space is inversely proportional to the disparity of corresponding points in the image plane.

$$disparity = x - x' = \frac{Bf}{Z} \tag{2.1}$$

Where f is the focal length, B is the baseline - the distance between the camera centres, x and x' is the distance between the point X in the image plane and their camera centres O and O', respectively. Figure 2.2 shows why Equation 2.1 holds using equivalent triangles.

Depth from Structured Light

In principle, depth from structured light is similar to depth from stereo, but with one of the cameras replaced by a projector [11]. The projector projects a pattern [12], [13], [14] onto the scene, which alleviates the problem of finding point correspondences. To not interfere with other computer vision techniques infra-red patterns can be used.

Time-of-Flight Cameras

As the name suggests, Time-of-Flight (ToF) measures the travel time of an emitted light wave to produce range data. As the speed of light is known, the travel time directly corresponds to the distance between the sensor and the object. Measuring this time can be achieved by measuring the phase shift between a signal radiated towards an object and the reflection of it. This principle is shown in Figure 2.3. The emitted infra-red wave is shown in red, and the reflected and phase shifted infra-red wave is shown in blue. Calculation of the phase difference can be done by using four phase control signals C_1 to C_4. These control signals regulate the collection of electrons from the reflected signal and result in the electric charge values Q_1 to Q_4. Figure 2.4 shows how these electric charge values are obtained. Given these electric charges the phase difference t_d can be calculated as

$$t_d = arctan\frac{Q_3 - Q_4}{Q_1 - Q_2}. \tag{2.2}$$

The phase difference t_d can be used together with the speed of light c, and the signal frequency f to calculate the distance d as

$$d = \frac{c}{2f}\frac{t_d}{2\pi}. \tag{2.3}$$

2.2 Depth Based Travel Aids for the Visually Impaired

In [5] electronic devices for navigation are divided into three categories:

1. Vision Enhancement - Tries to help users with remaining residual vision by providing enhanced visual output.
2. Vision Replacement - Replaces human vision by providing output directly to the optic nerve or the human brain.
3. Vision Substitution - Uses other sensory means like auditory or tactile feedback to provide a mental image of the scene.

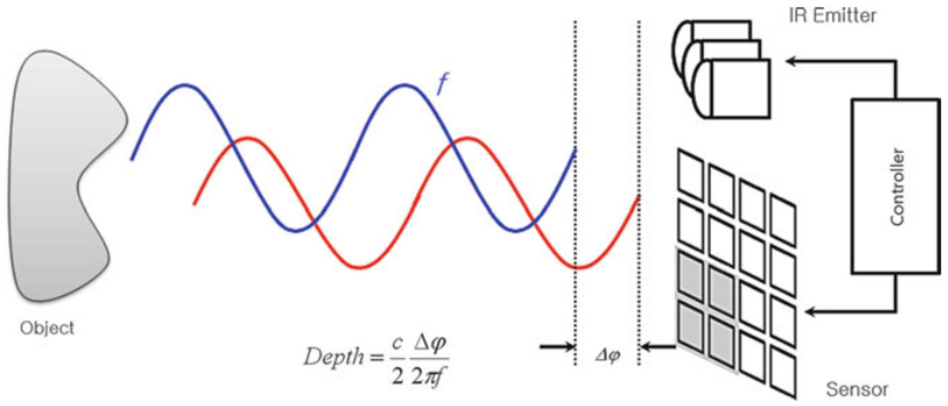


Figure 2.3: The principle of a time of flight camera. An infra-red wave is emitted (red) and the phase shift of its reflection (blue) is measured. (Figure from [15])

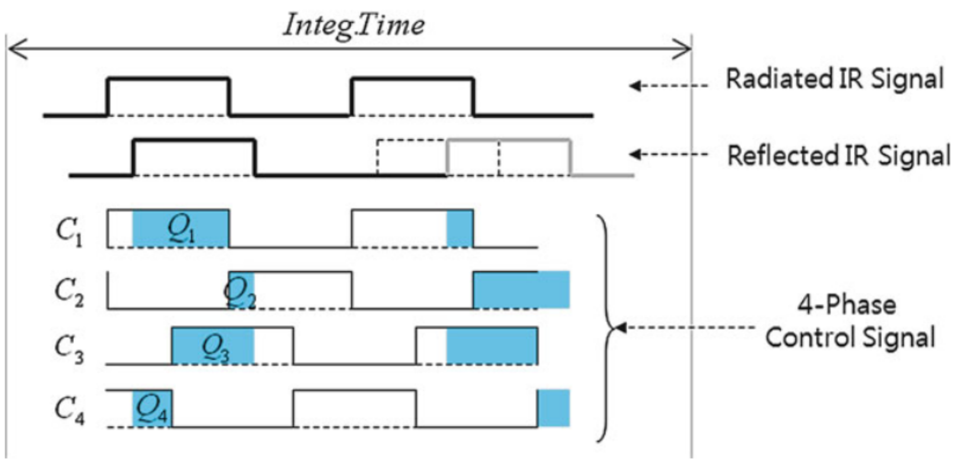


Figure 2.4: Four control signals C_1 to C_4 are used to gather the electronic charges Q_1 to Q_4 received by the reflected signal. These electronic charges can be used to estimate the time difference between the radiated and reflected signal. (Figure from [15])

Focusing only on vision substitution, it can be further divided into three main categories:

1. Electronic Travel Aids (ETAs) - provide information normally perceived visually by transforming them into a form, which can be communicated through other sensory modality.
2. Electronic Orientation Aids (EOAs) - provide orientation to the user.
3. Position Locator Devices (PLDs) - provide the user with positional information, for example by means of Global Positioning System (GPS).

This section will survey different ETAs and categorize them further based on the core range sensing technology used. However, all of them have in common that they are very large and obtrusive, as they require the user to wear not only a rather big depth sensing device like the Microsoft Kinect, but also a backpack with a laptop for computational power. Further, some systems described are not intended to be an ETA, but are navigational systems designed for robots. Still, algorithms intended to guide autonomous robots can be adapted and used to guide the visually impaired or provide them with a warning.

2.2.1 RGB-D camera

With the Microsoft Kinect a cheap Color and Depth sensing Camera (RGB-D) was introduced to the market. Due to the availability of the Microsoft Kinect and its SDK [16] it became popular among researchers.

Using RGB-D cameras such as the Kinect various different ETAs were introduced. They utilize distinct core methods from directly operating on the point cloud using RANSAC based approaches to more computer vision aided approaches, such as using region growing or marching cubes, or a mixture of both.

The system designed by Bernabei *et al.* [4] directly operates on the point cloud data and is meant to be time-critical. Therefore, each step in their algorithm is interruptible and should provide a meaningful, but conservative result. The algorithm has four main steps: "registration", "occupancy detector", "walk detector", and "analysis". The registration process detects the floor and transforms the data into a reference system centred at the user's feet with the help of the detected floor. The floor is detected as those points closer than a threshold to the plane fitted in the previous step. The initial step zero needs a scene without obstacles or motion to easily fit the ground plane. As the plane fitting depends on the amount of data used, to speed up the processing the data can be down-sampled if needed. This is done in a conservative manner, prioritizing the points nearest to the user. The occupancy detector creates a one-dimensional $Dx1$ depth map. The same conservative process as in the reduction of the data in the registration step is used and the nearest data points are prioritized. The resulting map gives a quantization of the space in front of the user, each bin representing a possible walking direction with the value of the nearest obstacle. This map is used to find the farthest point reachable considering the users width. To detect if a person is walking in front of the user, or the user is walking down a corridor a walk detector is needed. The accelerometer of the Microsoft Kinect is used to detect a walking motion and the walking direction. Only if the user walks towards an occupied space it is considered to be an obstacle. Further, if the distance to an obstacle stays approximately the same between readings, it is considered to be another person walking in front of the user.

One RANSAC based approach was introduced by Vlaminck *et al.* [17]. They use the point cloud obtained via a Microsoft Kinect sensor, which is transformed such that the xz-plane is parallel to the ground plane similar to [4]. Further, they down-sample it by using a $2x2$ sliding window averaging the values and filter it in x-direction such that no point is further than 75 cm away from the user. Afterwards, the most prominent planes with a surface normal perpendicular to the xz-plane are obtained using RANSAC with a fixed number of iterations. During each iteration the points within a fixed distance of the found plane are added to the set of inliers. If the cardinality of the set is larger than the cardinality of the previous iteration, it is considered to be the new ground plane. Walls are detected using RANdom SAmple Consensus (RANSAC) with the restriction of their normal to be parallel to the xz-plane. Local surface normals are used to speed up the detection of planes. Points which are not part of the dominant planes found in the previous step are clustered by their proximity. Their system is able to detect stairs by filtering the point cloud such that only points with a normal perpendicular to the xz-plane remain. These remaining points are sectioned into sets of points with a height equal to $h = H_n \pm H_{tol}$ with $H_N = H_{step}xN$. Each set is used to estimate a plane using RANSAC. An adaptive threshold for the amount of inliers is used to determine whether a step is detected or not. To detect doors, additionally the color information is used to further segment the "wall planes". These are further segmented into a set of planes using the surface normals and color information. A plane is fitted to these points using RANSAC and their convex hull is calculated. If it is too big or not big enough it is not considered to be a possible door. Lastly, the remaining points which are not considered to belong to any plane, but lie in a predefined door handle height are filtered. For each possible door it is checked if there are points in the predefined door handle zone. If both, a possible door and matching handle are found it is classified as a door.

Another approach utilizing RANSAC was introduced by Aladren *et al.* [3]. They use a consumer RGB-D camera and try to fuse RGB and depth data to extend the range of the depth sensor. To reduce the work load, a voxel-based filter to down-sample the point cloud is used. This greatly reduces the points considered in the point cloud (from 300000 to 7000) without losing main structural and representative information. Afterwards, a Manhattan world model (three main directions, which are orthogonal between each other [18]) is assumed and the RANSAC algorithm is used to find the most prominent planes. This is done by detecting a plane and removing its points, and repeating this process until most points are part of a plane. Planes are then identified by analyzing their surface normals and classified as floor or obstacle. Floor points are used as seeds for region growing to extent the range in the color image. Two different segmentation algorithms (polygonal floor segmentation and watershed segmentation) are combined to improve the result. A sound map is used to inform the user. They deploy audio instructions for high level commands, otherwise stereo beeps with a frequency depending on the distance to the obstacle are used.

A fast plane detection algorithm was introduced by Poppinga *et al.* [19]. They use a Swissranger SR3000 3-d range camera to obtain their point cloud and region growing to detect planes. Plane detection is done by first taking a random point r_1 and its nearest neighbor r_2, forming the initial set of points. This set is called region Π. The next step is to attempt and extend this region by trying to add a new point r' if, and only if, the distance between the region and r' is small enough and the mean square error to the optimal plane of the region with the new point is less than ϵ and the distance between the new point and the optimal plane is less than γ. The previous step is repeated until no new points can be added to the region Π. The size of this region is then checked and if it is more than θ it is added to a set of regions R, otherwise it is treated as unexplained and added to the set R'. Data points can be assigned to multiple regions as they could lie in intersections of two or more planes. The whole process is repeated until every point is either in R or R'.

Brock and Kristensson [20] use another approach to detect obstacles. Instead of trying to find structures in the point cloud directly, a depth map is used. As a depth sensing device the Microsoft Kinect is used. The depth map of the Kinect is down-sampled to a $20x15$ image giving the nearest depth value priority. To detect obstacles the marching squares algorithm is used. Afterwards, to inform the user of the obstacles their position is turned into a sound. With the pitch of the sound giving the height (y-axis) of the object, the volume representing the distance to the user, and the x position is encoded using stereo.

A more computer vision oriented approach is presented by Bhowmick *et al.* [21]. They use the Microsoft Kinect combined with Speeded Up Robust Feature (SURF) local descriptors, which are clustered using a Bag-of-Visual-Words (BOVW) model. The BOVW constructs feature vectors, which serve as input for a multi-class Support Vector Machine (SVM) to classify common office objects like laptops, chairs, stairs, etc. An audio feedback program is used to notify the user about obstacles ahead.

The work proposed by Jafri and Khan [22] employs the Google Tango Development Kit to develop an application to assist the visually impaired. This allows the system to be less obtrusive. The point cloud received from the depth sensors are converted into a depth image, which is pre-processed to remove noise. Then all edges are detected and removed using a canny edge detector and a region growing method finds all connected components. The largest component at the bottom is assumed to be the ground floor and removed. Navigational instructions are given after dividing the depth image into three sections. If the center section is free, no instructions are given, otherwise the right section is checked and if it is free a signal is given to bear right. If the right section is not free the left section is checked in the same manner. For users with low vision, the depth map is color coded with 4 colors to indicate how far away an obstacle is. In [23] the Google Tango Development Kit is used to help visually impaired users detect and recognize faces and objects. The depth information is used to tell the user where the face/object is in relation to the depth sensor. To recognize faces and objects they used a deep neural network (NVIDIA's CUDA® Deep Neural Network Library cuDNN). Another approach is proposed by Jafri *et al.* [24] utilizing the built-in Unity SDK of Google Tango. The Unity SDK is used to acquire a 3-D mesh of the environment and a character object will be used to perform distance calculations to obstacles.

Gao *et al.* [25] took the approach from [26] (see Section 2.7.1) and applied it to a 3-d range camera. The disparity is calculated assuming a baseline of a stereo-vision camera. To enhance the segmentation of obstacles on the road they utilize not only v disparity, but also u disparity. As this approach is used for assisted driving and no steps should occur, it can be assumed that no point has less depth than the ground surface on a given row/column. Furthermore, to be more robust a steerable filter is constructed and used on the disparity maps and the Hough transform is modified to improve efficiency and accuracy.

A different approach is applied by Peasley and Birchfield [27]. They make a ground plane assumption that the robot moves on the ground plane and all points above the ground plane are potential obstacles. To detect the ground plane, it is further assumed that initially the space directly in front of the robot is clear for calibration purposes. These points in the assumed free region are used to fit a plane by solving a least squares problem with a tolerance of 5 cm. For map building all points on the floor (± 5 cm) or higher than the robot are ignored. The remaining points are projected onto a 2-d ground plane by calculating $\mathbf{o}^{(i)} = \mathbf{p}^{(i)} - h^{(i)} * \mathbf{n}$, where $\mathbf{n}$ is the normal of the ground plane, $\mathbf{p}$ is the point to be projected, and h is the height. High reflective material produces invalid depth readings and to overcome this problem the 8 neighborhood of those invalid readings is observed. If the neighborhood contains a floor point, an "infinite pole" is synthesized. This represents an infinitely tall obstacle. Whenever there are more than 40% invalid depth readings, it is assumed that it is not safe to proceed any further. As this work is supposed to steer a robot an obstacle avoidance algorithm is implemented instead of a warning interface.

2.2.2 Stereo based

The advantage of stereo based depth sensing is their higher range. While a ToF sensor is mostly limited to a few meters, the stereo system does not suffer from this problem. However, there are also disadvantages in using a stereo vision system. Stereo vision depends on a feature rich (texture rich) environment and they are dependent on good lighting conditions. Further, to obtain a depth map from a stereo image pair, huge computational demands emerge. Still, modern processors have enough computational power to meet these demands.

A revolutionary algorithm to detect the ground plane in stereo images was introduced by Labayrade *et al.* [26]. Their motivation was to detect non-flat road surfaces for autonomous or assisted driving. They use the disparity map from a stereo camera to calculate a v disparity map. In the v disparity map a plane is mapped to a straight line. Therefore, Hough transform is used to detect the straight lines and filter the ground floor (street). For non flat-earth road geometry a succession of parts of planes is assumed. These are represented as a piecewise linear curve and the k (5 to 12) most prominent responses of the Hough transform are used. Obstacles are assumed to be vertical lines in the v disparity image. For a detailed explanation of the algorithm refer to Section 2.7.1.

Broggi *et al.* [28] took the v disparity and enhanced it slightly to be able to handle off-road scenarios. They use v disparity to find ground plane assumption. Static calibration data is used to compute the expected ground correlation line on a flat surface, as well as the ground correlation line for different pitch angles. A ground correlation line at a pitch angles oscillates parallel to itself. The v disparity along these candidate lines is accumulated and the line with the best correlation is chosen allowing to obtain a pitch estimation. To detect obstacles, a disparity space image is created.

Further enhancements to the v disparity algorithm were done by Soquet *et al.* [29] by combining v disparity for road detection from [26] with u disparity to detect obstacles. To cope with roll, yaw, pitch, or road slant Soquet *et al.* proposed a propagation algorithm. First, the global road profile is extracted, and then it is refined using a propagation phase. For every pixel on the initial global profile neighboring pixels are checked according to a pattern. If their intensity value is above a threshold, they are considered to be part of the global road profile and their neighbors are checked, too. To classify the free space, the u disparity map is used. The u disparity map creates a histogram similar to the v disparity, but instead of the rows the columns are accumulated. This allows to classify pixels with a disparity as follows: if the intensity of a pixel in the u disparity map is high it is classified as an obstacle pixel. If the pixel in the v disparity map belongs to the road profile it is classified as a road surface pixel. Otherwise, it is not classified and considered to be noise. This robustly classifies a few pixels, which are propagated to the remaining pixels. All unclassified pixels are influenced by the classified pixels. The contribution of each classified pixel depends on the distance to it. Once the free space is determined, a color segmentation is used to extract the road surface.

Another approach applying uv disparity maps was introduced by Bai *et al.* [30] using a two-part approach for obstacle detection - A detection-confirmation structure. On the one hand, the stereo image is captured and used to calculate a depth disparity map. This map is used to calculate a uv disparity map. Using the uv disparity maps allows the extraction of basic structural information of the scene. The original stereo images are smoothed using an anisotropic filter, which encourages intra-region smoothing whilst preserving edges. Given the basic geometric information and the smoothed image, a region of interest is selected supported by the geometric information. Further, approximately horizontal and vertical lines are extracted to detect obstacles. To confirm those obstacles, their location is determined and verified by depth discontinuities.

Musleh *et al.* [31] first apply a threshold to the u disparity to obtain an "obstacles map". With applying a threshold to the u disparity only obstacles with a certain height in pixels are kept. The v disparity is calculated on the "free map", which is every pixel that is not detected as an obstacle in the obstacle map. Removing the pixels which are part of obstacles increases the chance of the road surface to be the most prominent line in the v disparity map.

There are also RANSAC based solutions with stereo vision systems like the one introduced by Rodríguez *et al.* [32]. A stereo rig is mounted on the user's chest. It is assumed that the ground plane is the largest part in the bottom part of the image and RANSAC is used to calculate the plane coefficients. Further, to detect obstacles a cumulative polar grid histogram is used. Points, which are too high, are not counted as possible obstacles and a threshold of at least 500 points is used to determine whether a bin contains an obstacle or not. Warning the user is done using audio bone conducting technology, as this does not block the users ears.

Saez Martinez and Ruiz [33] designed their system to be able to warn the user about aerial obstacles at approximately chest height. This is done by using a stereo camera mounted to the user's chest. A map of a small vicinity around the user is kept and used for obstacle detection. The map is generated using a 6DOF visual odometry algorithm (SLAM). For obstacle detection the next step of the user is estimated according to the previous trajectory. Two virtual cubes are placed in the front, given by the moving direction, of the user. These cubes represent the user's body, more specifically where the body will be if the moving direction is kept. Therefore, the amount of points in these boxes is analyzed and obstacles are detected if the amount is above a threshold.

Another approach was introduced by Koester *et al.* [34]. They calculate the gradient of the disparity map given by a stereo vision setup. This gradient can be used to detect planar surfaces. The gradient is calculated using a discrete ∇ operator. Given the gradient in X and Y direction it is possible to calculate the surface region's orientation. With the assumption that the accessible section is connected to the bottom part of the image, the orientation of the surfaces is observed in a bottom up manner. As long as their orientation does not deviate too much from a perfect upright orientation they are considered accessible.

2.2.3 Ultrasonic sensor based

Borenstein and Ulrich [2] introduced the GuideCane. It consists of a long handle with a "sensor head". The sensor head is equipped with ultrasonic sensors. The sensor head sits on top of a steerable axle with wheels. They used the Vector Field Histogram (VFH) (see Section 2.7.3), which detects obstacles and provides the direction needed to avoid them. This direction is then used to guide the user by steering the wheels of the sensor head. The user intuitively follows the force resulting from the steering, like a trailer follows a car, thus avoiding a collision.

An ETA using similar techniques, which was introduced by Shoval *et al.* [35] is the NavBelt. The NavBelt is a different setup to the GuideCane, as the user wears a belt, which has ultrasonic sensors mounted to it. However, the algorithm used to detect and avoid obstacles is also based on VFH. There are three modes, which guide the user: the "Guidance Mode", "Directional Guidance Mode", and the "Image Mode". These modes offer varying control to the user over the global navigation, from steering the user to a known goal, over local obstacle avoidance, and lastly the "Image Mode" provides the user with an acoustic image of the surroundings.

2.3 Computer Vision Technique Based Travel Aids for the Visually Impaired

Peng *et al.* [36] use a mid range smart phone to provide a non-intrusive electronic travel aid. Their approach uses just the built in camera of the smart phone to determine whether the path in front of the user is safe or not. To reduce the computational cost, only a limited region of the camera image is used. It is assumed that the user holds the smart phone in the walking direction at a tilt angle of approximately $45°$. Therefore, a trapezoid can be calculated to estimate the region of interest. Further, it is always assumed that a small region in front of the user - the bottom of the image - is safe floor. Based on this assumption, a binary color histogram with 16^3 bins in RGB space is created. This binary color histogram does not concern pixel counts in each bin and just labels the corresponding bin, as well as its 9 neighboring bins, as true. Every pixel outside of the region of interest is labelled as "uninterested". Each pixel assumed to be floor is labelled as floor and all remaining pixels are labelled as "unvisited". To classify the unvisited pixels as floor or obstacle, every neighboring pixel of floor pixels is considered. First the current pixels histogram bin is checked, if it is true, it is determined as floor. Otherwise, a $3x3$ Laplacian edge detector is convolved with the given pixel in each RGB channel, respectively. If the convolved value in any channel is above a pre-defined threshold, it is determined as obstacle, if not it is determined as floor and the histogram is updated with the newly found pixel. The image is subdivided into three regions: left, center, and right. Once the closest obstacle in each sub-region is determined, its y position in the image can be converted into corresponding depths, given the focal length and tilt angle of the camera.

The work introduced by Shen *et al.* [37] uses the camera of a mobile phone to detect crosswalks and helps the user to align with the crosswalk. The user is supposed to take an image of the intersection, which is then processed to give feedback to the VI. Detection is done using a local grouping process based on figure-ground segmentation. To segment the pixels into figure and ground a series of compatibility relationships have to be met, which are checked by a graphical model. Once the crosswalk is found, the system helps to align the user's body and walking direction with the crosswalk. This is done by using the camera's line of sight direction corresponding to the center of the image as well as the vanishing point of the two lines "enclosing" the crosswalk.

2.4 Other Aids for the Visually Impaired

Other methods, such as the one described by Martinez-Sala *et al.* [38] need external preparation to function. This makes them unusable if the visited place is not prepared for this particular system. They use Ultra-Wideband (UWB) technology to deliver accurate indoor positioning. The user is equipped with an Ubisense UWB tag and directed via headphones. An advantage of systems needing to prepare a room beforehand is that even if the VI never was in this room, guidance to a desired destination can be given.

Further, there are aids, which intend to help the VI in other aspects of their everyday life. For example, Shaik *et al.* [39] use a mobile phone camera to help visually impaired persons. Using Optical Character Recognition (OCR) and Text-To-Speech (TTS) functionality of an Android phone, they enable visually impaired users access to reading any text. This could be anything from a street sign, to labels on groceries, as well as their expiration date.

2.5 Warning Interfaces

The ETAs described in the previous sections all employ different warning interfaces. The design of the warning interface is a very important part of an ETA, as it should be discreet, but it also needs to be highly functional.

Acoustic warning interfaces can range from high-level instructions like 'turn left', 'obstacle ahead', etc. to an alteration of pitch/volume of a sound depending on the position and size of the object. High-level commands are often very slow to process and might not deliver a reliable warning in time. A more advanced form of variation of pitch/volume is using 3-d spatial audio, as described in Section 2.6. However, hearing is an essential sense for VI people and its obstruction should be limited to a minimum.

A more complex device was developed by Hoang *et al.* [40]. They use an electrode matrix, which the user takes into the mouth, giving electrical impulses. Due to the sensitivity of the tongue, the user can perceive the electrical impulses accurately and translate them into directional impulses.

Amemiya [41], [42] uses a haptic hand-held wayfinder with pseudo-attraction force. Exploiting the non-linear relationship between perceived and physical acceleration they are able to generate a force sensation, which should guide the user. Another device, which relies on tactile responses was introduced by Mann *et al.* [43]. They use a vibrotactile helmet, which indicates direction through vibration. Using a helmet gives the advantage of a hands-free operation.

2.6 3-d Spatial Audio

Similar to stereo vision allowing for 3-d spatial perception, having two ears allows for the spatial localization of sounds. As visually impaired people heavily rely on their hearing to avoid dangerous situations, for example listen to the direction of traffic to cross the street without deviating from their path, obstructing their ears would be a major limitation. However, in [44] it was shown that bone conduction devices can be used for 3-d audio. Bone conduction devices use the bones of the head to propagate the waves to the ear drums and therefore, they do not need to obstruct the wearer's ears. This section will describe two ways of how synthesizing spatial audio works.

2.6.1 Pan and Fade

Due to the way ears are placed on the head, a sound source coming from the left would be louder in the left ear, as the head dampens the volume for the right ear. A simple way to provide spatial audio to the user is therefore to pan and fade the stereo signal.

The most basic variant would just alter the volume for each channel depending on the angle between the sound source and the head. To further improve the quality of the sound position the ear further away from the sound source has to hear it with a short delay of a few micro seconds. A drawback of this simple method is that it only provides basic sound localization abilities. For example, it would be impossible to distinguish whether the sound origin is in front or behind the user as it would yield the same intensity and delay in both cases.

2.6.2 Head-Related Transfer Function

In reality sound perception is much more complex than a simple pan and fade. Perception is altered by the shape of the head, ears, ear canal, density of the head, even the size and shape of nasal and oral cavities affect the sound perception.

In consequence of this complexity, a so-called Head-Related Transfer Function (HRTF) is used. This function describes how a sound wave is filtered given the position of the sound origin and the head. One way of obtaining the HRTF is to measure it using small microphones inside the ears of a person. This HRTF allows for a much better sound localization, as it incorporates the small differences between a sound source coming from the front or the back of the listener. Even though each person would have their own HRTF, the human brain is capable of learning a new HRTF and adapting to it [45]. Of course, the closer the used HRTF is to the real HRTF of the person, the easier and better the localization will be.

2.7 Algorithms Used or Tested

In this section the most important parts of the algorithms used or tested in this thesis will be explained.

2.7.1 V Disparity

V disparity was introduced by Labayrade *et al.* [26] to be able to detect obstacles on non-flat road surfaces. The idea is to create a histogram of the disparity values in an (u, v)-image environment and detect the road surface as a piecewise linear curve.

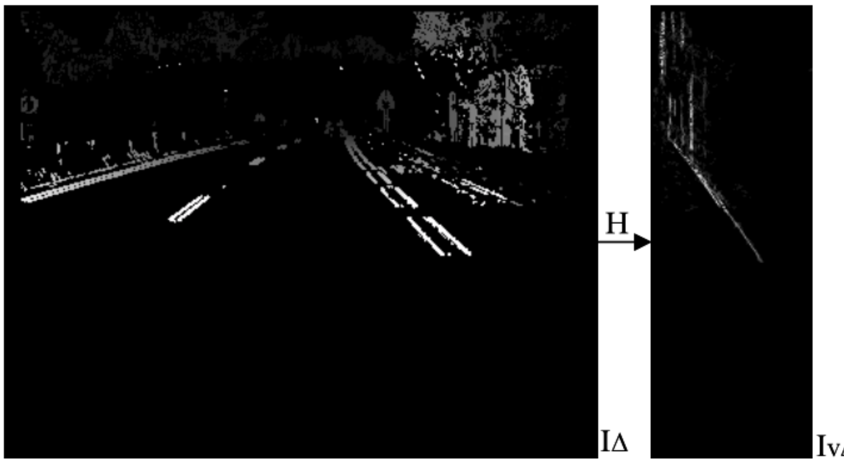


Figure 2.5: Example disparity map I_Δ and the corresponding v disparity map $I_{v\Delta}$. (Figure from [26])

Construction of the v disparity image is as follows: Suppose there is a disparity map I_Δ. Let H be a function of I_Δ, such that

$$H(I_\Delta) = I_{v\Delta}. \tag{2.4}$$

Denote $I_{v\Delta}$ as the v disparity image. Points with the same disparity along an image line i are accumulated by $H(I_\Delta)$. For each image line i, the abscissa u_M of a point M in $I_{v\Delta}$ corresponds to disparity Δ_M and its intensity value i_M corresponds to the number of points with that disparity Δ_M on line i. On each line i, i_M is calculated as follows:

$$\forall i : i_M = \sum_{P \in I_\Delta} \delta_{v_P,i}\delta_{\Delta_P,\Delta_M} \tag{2.5}$$

where $\delta_{i,j}$ is the Kronecker delta.

In other words, given disparity map I_Δ, the v disparity map $I_{v\Delta}$ is constructed by accumulating the points of same disparity in I_Δ along the $\vec{v}$ axis. Figure 2.5 shows an example disparity map I_Δ and the corresponding v disparity map $I_{v\Delta}$.

Once the v disparity map $I_{v\Delta}$ is constructed, robust line finding methods, such as Hough transformation can be used to find the appropriate lines, which represent the road surface. Given the disparity of the road surface, it is simple to remove it and only keep possible obstacles.

2.7.2 Random Sample Consensus (RANSAC)

RANdom SAmple Consensus (RANSAC) is a robust iterative algorithm. It can be used to estimate the coefficients of a mathematical model, such as a plane, that fits given data points. Due to the way RANSAC operates it is robust against outliers, which would have a severe impact on methods, such as least squares optimization [46]:

1. Randomly select minimum number of points needed to be able to fit the used model.
2. Check model hypothesis against the whole input data set.
3. Record how well the input data fits the model according to a loss function.
4. Repeat until sufficiently enough points consent with the hypothesis.

As the basic RANSAC algorithm weighs every point the same, it can lead to wrong results. For example, in case of a step, a slanted plane might have a better fit overall than the real geometry, therefore, it is possible to use surface normals to support the thesis. Only if the point's surface normal coincides with the surface normal of the hypothesis it is considered as an inlier. Various methods to estimate the surface normal of a point in a point cloud exist, like employed by the point cloud libraries surface normal estimation [47], which uses a kd-tree [48]. Depending on the layout of the data an organized neighbor tree or integral images [49] can be used to speed up the calculation. However, the surface normal estimation of a large but sparse point cloud is computationally expensive.

2.7.3 Vector Field Histogram (VFH)

The Vector Field Histogram (VFH) introduced by Borenstein and Koren [50] is an efficient wayfinding algorithm for robots. The Vector Field Histogram was developed as an improvement over their previous Virtual Force Field algorithm and eliminates its shortcomings. It employs a two-stage data-reduction technique resulting in three levels of data representation. As each new sensor reading influences the steering direction directly and continuously no abrupt changes in direction occur.

The two-stage data-reduction technique produces the following three levels of data representation:

- The highest level is a 2-d Cartesian histogram grid C, with each cell (i, j) holding a certainty (probability) value $c_{i,j}$. These certainty values respond to the confidence that there exists an obstacle within this cell.
- The intermediate level representation is a 1-d polar histogram H. It consists of n angular sectors of width α. The width α is freely choosable.
- The lowest level is the output of the VFH algorithm.

To perform the first data reduction step, an active region C^* of size $w_s \times w_s$ around the robot is defined. The robot is always in the middle of the active region, the so-called Vehicle Center Point (VCP). This active region C^* is mapped to H by treating the contents of each active cell $c^*_{i,j}$ as an obstacle vector. The direction of the vector is defined as follows:

$$\beta_{i,j} = \tan^{-1} \frac{y_j - y_0}{x_j - x_0} \tag{2.6}$$

and the magnitude of the vector is given by:

$$m_{i,j} = (c^*_{i,j})^2 (a - b d_{i,j}) \tag{2.7}$$

where

a, b	are positive constants
$c^*_{i,j}$	is the certainty value of active cell (i, j)
$d_{i,j}$	is the distance between active cell (i, j) and the VCP
$m_{i,j}$	is the magnitude of the obstacle vector
x_0, y_0	are the present coordinates of VCP
x_i, y_j	are the coordinates of active cell (i, j)
$\beta_{i,j}$	is the direction from active cell (i, j) to the VCP

Given the direction $\beta_{i,j}$ it is possible to calculate which sector the active cell $c^*_{i,j}$ belongs to:

$$k = \lfloor \frac{\beta_{i,j}}{\alpha} \rfloor . \tag{2.8}$$

Where α is the arbitrarily chosen resolution of the polar histogram H, for example $\alpha = 5^\circ, n = 72, k = 0, 1, 2, ..., n - 1$. Each sector k corresponds to a discrete angle $\rho = k\alpha$. Given this information the polar obstacle density (POD) h_k is calculated for each sector k by

$$h_k = \sum_{i,j} m_{i,j}. \tag{2.9}$$

Once the polar histogram H is constructed, the second data reduction is applied. This step determines the steering direction for the robot. A threshold is applied to the polar histogram H. The resulting valleys and ridges in the histogram are evaluated. The size of a valley is defined by the amount of consecutive sections below the threshold. To determine the steering direction, the biggest valley which best coincides with the target direction is chosen. The direction of the steering is then defined to be through the middle of the valley.

Chapter 3

Design

3.1 Requirements and Constraints

The following section describes the requirements and constraints of the Virtual White Cane. The overall goal of the system is to warn the user about possible obstacles in his path. To achieve this goal, it is necessary to detect and remove the ground floor, as it should not be considered to be an obstacle. All remaining data points are considered to be not traversable and thus are declared as an obstacle. The user should be warned about obstacles in their path. There are two different warning methods, which can be combined or used independently. An acoustic warning, providing an acoustic image of the obstacles by sequentially playing a sound originating from the object. This allows the user to create a mental image of the layout ahead of the device. The second warning method uses the vibrator of the device. If an obstacle is detected in front of the device and it is below a certain distance threshold, the device should vibrate and warn the user. This enables the user to actively scan their surroundings using a sweeping motion to find a free path.

The aforementioned functionality has to be provided using a soft real-time constraint. Currently, a Google Tango enabled device is restricted to five depth readings per second. Naturally, this provides a constraint on the timing and thus the computation of the obstacles has to finish before a new depth reading is provided. Further, due to the motion tracking of a Google Tango enabled device, it is possible to obtain a transformation such that the depth data stays at the correct global position, even if the device is moved between two depth readings. Thus, even though only five depth readings per second are provided, a faster computation is desired to be able to provide the user with a warning as soon and as accurate as possible.

Due to the design of the system, further constraints have to be met for optimal performance. The ground floor is considered to be between 0.8 m and 1.3 m below the device at any time. This coincides with the average height a mobile device is held at while the user is standing.

The tilt angle at which the device is held is largely irrelevant. As the angle of the line to be found in the v disparity map correlates with the tilt of the device, it can be adapted. With the device's internal gravity sensor it is easily possible to get the orientation of the device and therefore adapt to it. The angle of the line in the v disparity map changes in a linear fashion with the tilt angle of the device, therefore the angle can be calculated as follows:

$$angle = k * tilt + d \tag{3.1}$$

where k and d are the parameters of the line. To find the values of these parameters, the v disparity map of an empty room with no obstacles was generated. Given the v disparity map using Hough transform the angle of the line with the largest response was recorded as well as the tilt angle. Using linear regression on these recorded data points, the following values for k and d were found:

$$\begin{aligned} k &= 0.1478 \\ d &= 163.223 \end{aligned}$$

Using these parameters and the tilt angle of the device, a fairly tight bound of the line angle can be calculated. Thus, it is possible to only allow a deviation of 1°.

Obviously, for acoustic warning to be able to give positional sound, stereo speakers have to be available, for best results stereo headphones should be used.

3.2 Ground Plane Detection

Instead of detecting obstacles, the dual problem of finding the free space is used, which means the ground plane has to be detected and filtered and every remaining point is considered to be an obstacle. This section will describe how the ground floor is detected.

As the system will run on a hand-held device, only a broad assumption about the devices distance to the ground can be made and the distance might vary greatly between frames. Therefore, the user is restricted to holding the device at a certain height as described in Section 3.1.

To be able to robustly detect the ground plane, two algorithms are used sequentially. At first a v disparity map is created and using Hough transform a line representing the ground plane is found. The disparity values of this line are then filtered to separate the possible ground plane from the obstacles. To further refine the ground plane assumption the output of the v disparity map filtering is used as input for a RANSAC algorithm. The output coefficients of the RANSAC algorithm are then again used to crosscheck the whole point cloud. If the v disparity map fails to find a ground plane, it is assumed that no ground plane is visible and all points are considered as obstacles. The dataflow of the whole ground plane detection process is shown in Figure 3.1. The following sections will provide a detailed explanation of this process.

3.2.1 V Disparity Map

To be able to calculate a v disparity map, the point cloud has to be in an ordered (u, v) grid. This means that basically each X, Y world coordinate is given by the (u, v) coordinate and only the depth Z varies. Given an ordered point cloud, the v disparity can be calculated for each row as mentioned in Section 2.7.1. This v disparity map is then used to find a ground floor candidate. To do so a canny edge detector is applied, such that only the most prominent lines are preserved. Afterwards, a Hough transform is performed to detect these lines. The Hough transform can be limited to find a line with an angle θ between $[0, \pi/2]$, as the ground floor will never produce a line at an non-acute angle. Only lines which correspond to the ground plane assumption are used. The ground plane assumption is derived from the constraints defined in Section 3.1. Due to these constraints it is well defined where the lines have to cross the v disparity image borders, as well as their slope. It is necessary to follow these constraints, as otherwise it would not be possible to distinguish between a plane formed by, for example, a table or the floor. Especially if most of the floor is obstructed by obstacles like a large table, this would be the case as the most prominent Hough transform response would be the line representing the table. The resulting disparity values obtained by these lines are used to filter the point cloud into ground floor and obstacles.

Even though the tilt angle of the phone can be used to limit the slope of the line needed, the user still has to be careful how it is held. Not aligning the bottom edge of the phone to be roughly parallel to the floor will alter the v disparity map. Doing so violates the assumption that the floor can be modelled as a piecewise linear curve in the v disparity map, as shown in Figure 3.2 This will cause the

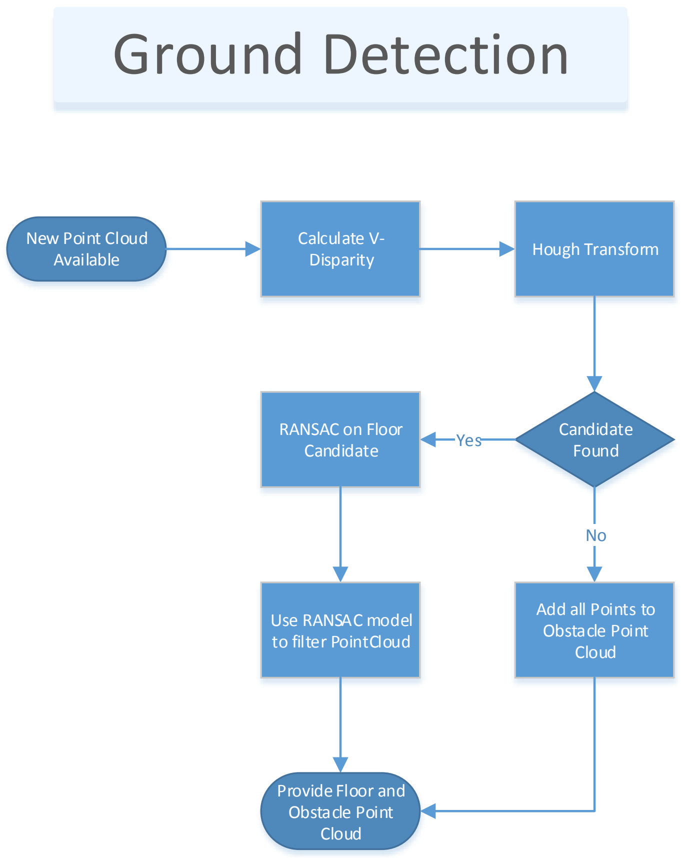

Ground Detection
New Point Cloud Available
Calculate V-Disparity
Hough Transform
Candidate Found
Yes
RANSAC on Floor Candidate
No
Use RANSAC model to filter PointCloud
Add all Points to Obstacle Point Cloud
Provide Floor and Obstacle Point Cloud

Figure 3.1: The point cloud is used to calculate the v disparity map. Using Hough transformation, the most relevant lines are detected. If no lines are detected all points are considered to be obstacles. Otherwise, the floor candidate is refined using RANSAC. Then all points matching the floor model are considered as floor and the remaining are labelled as obstacles.

line to oscillate and be much thicker than expected. Figure 3.3 illustrates why the v disparity starts to oscillate. The horizontal lines of the checkerboard have the same depth. As long as the device is parallel to the ground, the image lines are also parallel to the ground, therefore, the disparity is the same. However, once the device no longer is aligned, the depth readings also start to shift.

3.2.2 RANSAC

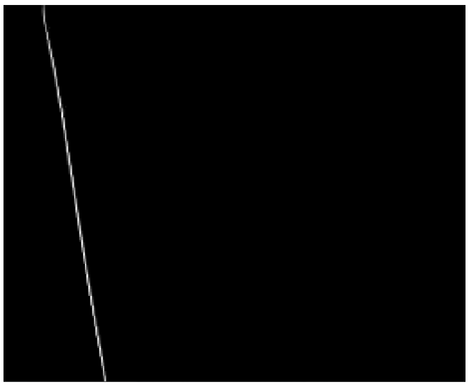
To further improve the quality of the ground plane candidate, the points obtained by the v disparity filtering are used and a ground plane is fitted using the RANSAC algorithm. To ensure the found plane has the correct orientation in space, the normal has to be verified. This also allows to compensate to a certain degree if the user is holding the lower edge of the device not parallel to the ground. The RANSAC algorithm is still capable of finding a good fit, even though only parts of the floor will be used as input for it. Finally, the coefficients found by the RANSAC algorithm are used to filter the whole point cloud and remove the ground plane. Points are part of the plane, if they are within 7 cm of the plane spanned by the coefficients. The threshold is set to accommodate for noise in the point cloud.

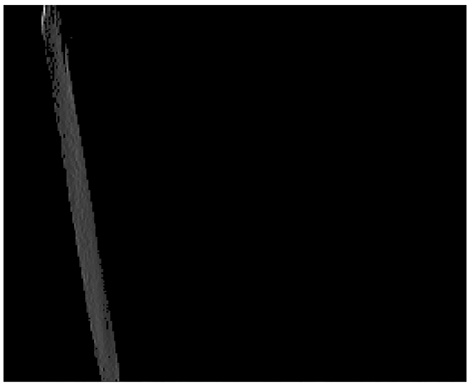
3.3 Obstacle Detection and Warning

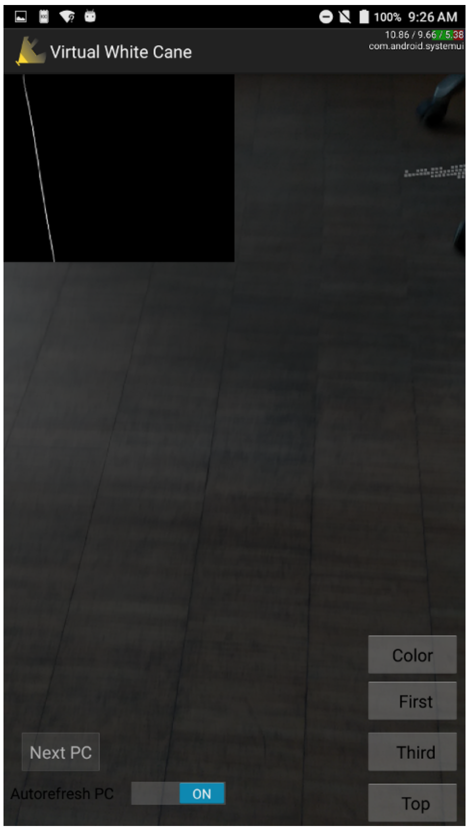
This section will describe how the obstacle detection and warning is performed. It consists of the Obstacle Detection algorithm and Obstacle Warning.

To be able to give a meaningful warning to the user, the obstacle point cloud needs to be further refined. For this purpose an obstacle detection algorithm called Conservative Polar Histogram was developed. This algorithm puts each point into a polar grid and creates a disparity histogram for each bin of the grid. A threshold gets applied to each disparity histogram to filter remaining outliers and the nearest remaining bin is considered to be an obstacle.

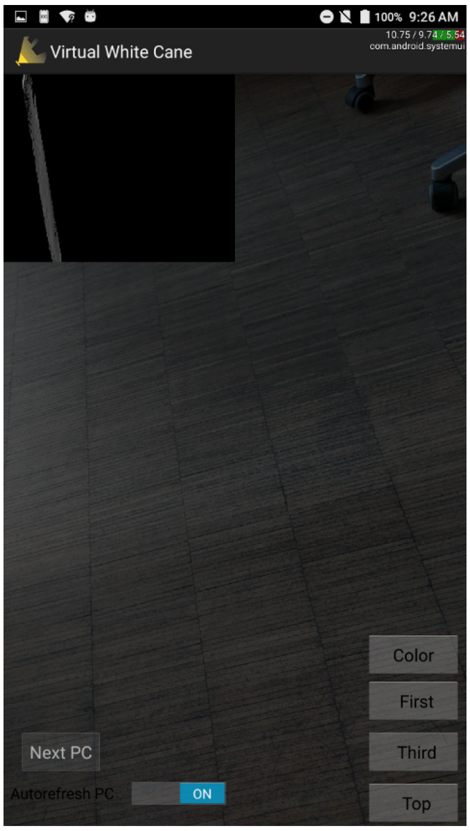
After the obstacles are detected, the user needs to be warned about them. Warning can be done using either tactile, or acoustic warning, or both methods at the same time. The obstacle detector provides the warning module with the nearest obstacle in each direction. Acoustic warning places a sound source for each object and plays them sequentially, while tactile warning is more limited in its capabilities. Neither the intensity of vibrations can be altered, nor is it possible to provide a


(a) Device held parallel.


(b) Device not parallel to ground.

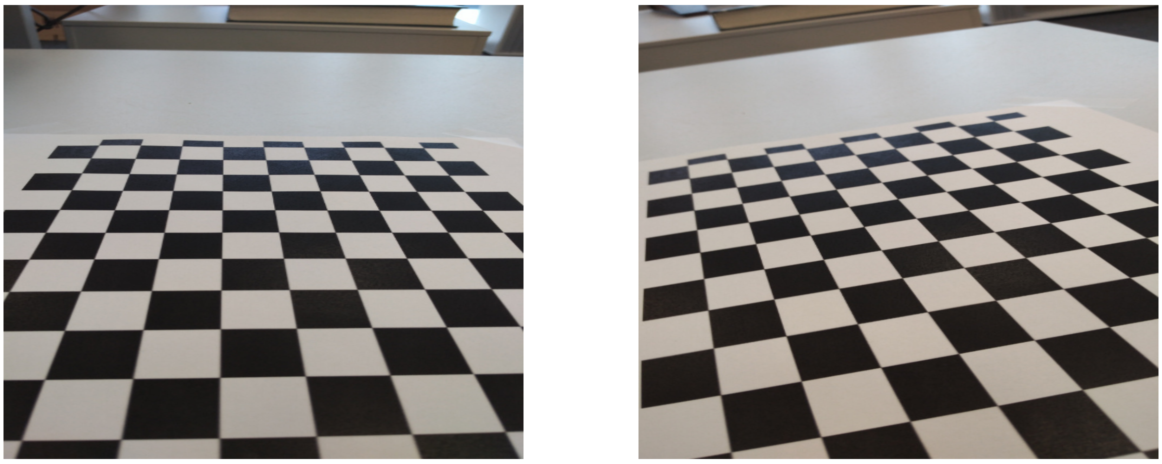


(c) Color image to the v disparity at Fig. 3.2a.



(d) Color image to the v disparity at Fig. 3.2b.

Figure 3.2: V disparity of the same scene, if device is not held parallel to the ground.



(a) Bottom edge parallel to the ground. (b) Bottom edge not parallel.

Figure 3.3: Horizontal lines on the checkerboard have the same depth.

sense of direction with the vibration unit of the device. Hence, only if an obstacle is immediately in front of the user, a warning is given.

Further details about the design of these two important core parts of the application will be given in the following sections. First, the obstacle detection is described followed by the two different ways of warning, which will be provided by the application.

3.3.1 Obstacle Detection

At the present time, widely used obstacle avoidance algorithms [51] use a polar histogram to detect likely obstacles and to steer a robot away from them. The VFH algorithm introduced by Ulrich and Borenstein [51] is not able to simply detect obstacles and let a person decide where they would want to go, based on the information given by the algorithm. It is not possible to distinguish between a small, but close, or a large, but far away obstacle. Both would produce a similar response. Due to the fact the robot is steered, this does not matter, as the response gets larger if there would be a collision with the obstacle and the robot would simply get the command to change direction towards a possibly free space. However, a human might not want to be "steered" and rather find his/her own way. Another problem with the VFH is the height and angle at which a person would hold a mobile device. It is possible to hold the device in such a way that for example a table edge in front of the user might produce the response of a small or far away obstacle, as there are only a few range readings on the table. Further, the traditional white cane would pass under the table and the table might not be detected as an obstacle and a collision might occur. To prevent this from happening, the user needs to be informed about each obstacle and the distance to it.

The point cloud provided by the Ground Plane Detection module described in Section 3.2 should now only contain obstacles. Obstacle detection is performed in a conservative manner similar to Bernabei *et al.* [4]. This means no obstacle should go undetected and give priority to the closest obstacle in the path. The points have to be transformed into the OpenGL world coordinate system and translated such that the camera is at the origin and then a polar histogram is created. With the points in the OpenGL coordinate system, a projection onto the ground plane can be easily achieved by omitting the y axis resulting in a 2-d point pt_2. Each point in the obstacle point cloud is associated with its respective bin k, according to their angle in polar coordinates calculated as in Eq. 2.6 and Eq. 2.8 with an $\alpha = 5°$. Additionally, for each 2-d point pt_2 the euclidean distance d from the origin - which is the camera - is calculated.

$$\begin{aligned} pt_2 &= (pt_{ogl}.x, pt_{ogl}.z) \\ d &= dot(pt_2, pt_2) \end{aligned} \tag{3.2}$$

The euclidean distance d is then inverted, scaled and discretized into a range from [0, 255]. As the exposure time of the ToF camera is fixed, a limit to the closest possible depth reading is inherent. Thus, a depth reading closer than 0.2 m is very unlikely to be valid and the numerator of d_{pseudo} can be set to 0.2. If closer depth readings can occur, this value should be changed. As the depth values are not obtained using stereo images, the values are interpreted as a 'pseudo disparity' d_{pseudo}.

$$d_{pseudo} = \lfloor \frac{0.2}{d} * 255 \rfloor \tag{3.3}$$

Given the pseudo disparity d_{pseudo} and the bin k, a disparity histogram is created for each bin by counting the number of occurrences for each disparity value d_{pseudo}. Assuming there are $K = 72$ bins in total, this results in an array of size 72×256. Listing 3.1 provides pseudo code of how the Conservative Polar Histogram is formed.

The disparity histogram allows to further filter outliers from the obstacle point cloud by applying a threshold and therefore removing bins containing only a

```
foreach point pt_ogl
  //''project'' to 2-d
  pt_2 = (pt_ogl.x, pt_ogl.z);

  // calculate angle beta and bin number e.g. ALPHA = 5 degrees
  beta = atan(pt_2.x / pt_2.z);
  k = beta/ALPHA;

  // calculate distance and pseudo disparity
  d = dot(pt_2, pt_2);
  d_pseudo = floor(0.2/distance * 255);

  // use k and d_pseudo to increment the histogram value
  disparityHistogram[k][d_pseudo]++;
```

Listing 3.1: Pseudo code of Conservative Polar Histogram creation.

few points. After a threshold was applied to the disparity histogram, the biggest remaining non-zero disparity is considered to be the nearest obstacle in that direction. Figure 3.4 shows an example of how the conservative polar histogram works. The blue objects are obstacles, and the area around the user is sectioned into bins. For each bin a disparity histogram is created, then a threshold is applied and evaluated. The red line indicates the distance to the objects. Figure 3.5 shows a flowchart of the performed steps during the obstacle detection algorithm.

Once the biggest disparity is found, it has to be converted back into 3-d X, Y, Z coordinates. The bin number k is easily converted back to an angle θ by multiplying it by α and adding $\alpha/2$. By adding $\alpha/2$, the angle will be in the middle of the bin instead of at the margin. Lastly, a conversion from degrees to radians is needed, as the trigonometric functions expect radians as input.

$$\theta = \frac{k * 5 + 2.5}{180} * \pi \tag{3.4}$$

Given the angle θ, the disparity value still needs to be converted back into the depth value d by simply inverting Eq. 3.3, giving:

$$d = \frac{0.2}{d_{pseudo}/255} \tag{3.5}$$

Angle θ and distance d are polar coordinates, therefore they can be easily converted into euclidean coordinates by:

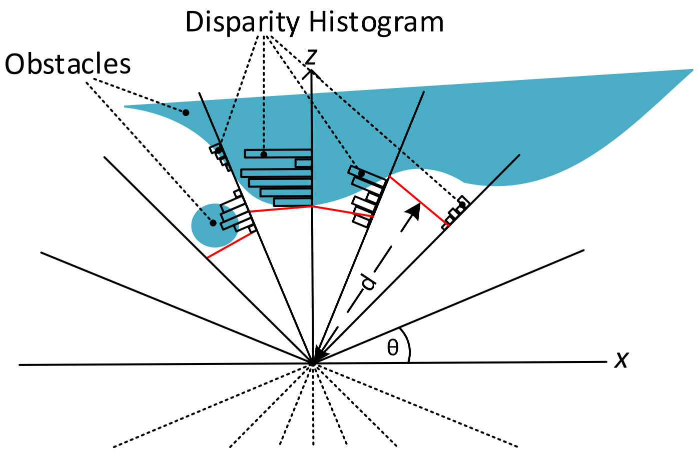


Figure 3.4: An exemplary Conservative Polar Histogram. The blue objects represent obstacles, while the red line indicates distance to the object. The points are transformed into their polar coordinates (θ, d) and categorized into bins. For each bin a disparity histogram is created and used to determine the closest obstacle.

$$\begin{aligned} X &= d * \cos(\theta) \\ Z &= d * \sin(\theta) \\ Y &= 1 \end{aligned} \tag{3.6}$$

As the Conservative Polar Histogram only works on a 2-d representation of the data, the third coordinate Y is lost. Hence, each obstacle is simply placed at a height of $Y = 1$ at the moment.

3.3.2 Obstacle Warning

The design of the obstacle warning is as central to the function of the application as finding obstacles in the first place. If the user cannot interpret the warning rapidly and easily, it might not provide any additional benefit to them. The worst case scenario would result in a collision, if the user continues to walk while trying to interpret the warning or simply misinterprets it.

Once all obstacles are detected, the user needs to get a warning about their position. This section will describe the two possible ways chosen to warn the user.

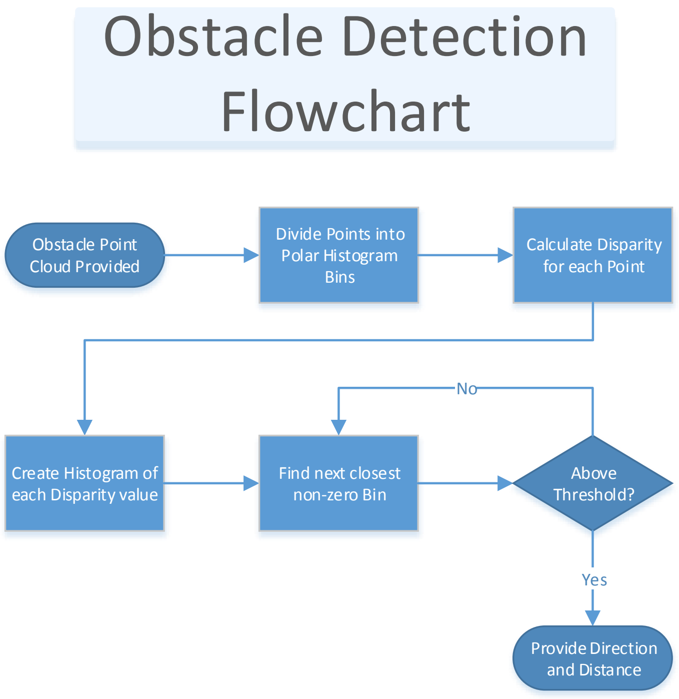

Obstacle Detection Flowchart
Obstacle Point Cloud Provided
Divide Points into Polar Histogram Bins
Calculate Disparity for each Point
Create Histogram of each Disparity value
Find next closest non-zero Bin
Above Threshold?
No
Yes
Provide Direction and Distance

Figure 3.5: Obstacle Detection Flowchart. Each point is put into a bin according to its position. Afterwards, the disparity is calculated and a histogram of the disparity is created. The nearest non-zero value above a threshold is considered to be the closest obstacle in the direction of the bin.

Vibrator Warning

One way to provide a warning is to use the vibrator to inform the user about obstacles in their path. Unfortunately, the intensity of the vibrations is not adjustable in Android. Therefore, it is only possible to provide a warning once an obstacle is within a certain distance, defined as 1 m, of the user. Further, it is not possible to communicate the exact position where an obstacle resides or provide the user with a full mental view of the scene in front of them without moving the device. Hence, only a warning if an obstacle is located directly in front of the mobile device is provided, which allows the visually impaired to use the device like a scanner. To achieve this, only the current front facing bins from the obstacle detection algorithm are used to warn the user. As the Conservative Polar Histogram resides in a world coordinate system with the device rotating at the origin, the front facing bins are calculated in relation to the device's orientation.

Acoustic Warning

Even though hearing is an essential sense for visually impaired people, which should not be hindered if possible, acoustic warning provides better possibilities to inform the user without the need of another device. Due to stereo sound and HRTF it is possible to position a source of sound, such that the user immediately gets a sense of where obstacles are in relation to the device.

A source of noise is positioned for each bin at the distance given by the Conservative Polar Histogram described in Section 3.3.1. These sounds are played sequentially like a radar. With this information the VI user can create a mental image of his surroundings.

Choosing the right sound file to be played is a difficult task. It should obviously be easily differentiated from any natural sound source, as this could lead to confusion otherwise. Further, the more complex the sound is, the easier it is to be located. It should contain various frequencies and not be a simple sine wave. However, the sound cannot be too long as it has to be played like a sonar and otherwise many sound sources would just overlap and confuse the user or it will take too long for each sound to be played and the user cannot react in time. As the sounds are played like a sonar in a round robin fashion, they seem to be animated (the sound moves along the obstacles) and they will repeat as the warning module is called repeatedly. Repeating and animating sounds helps the user to pinpoint them much faster.

3.4 Software Design

This section will describe how the software is designed and how each software component is supposed to communicate with each other. There are three main components:

- Point Cloud Provider
- Ground Detection Module
- Warning Module

The Point Cloud Provider follows a publish-subscribe pattern. Each time a new point cloud is available, the Point Cloud Provider publishes the new point cloud and notifies each subscriber. The Ground Detection Module is subscribed to the Point Cloud Provider and processes the point cloud as soon as it is available. After the first point cloud is received and processed by the Point Cloud Provider, it is possible to poll for the ground plane cloud or the obstacle cloud at any given time. The Warning Module polls the Ground Detection Module at set intervals and triggers a warning if needed. Figure 3.6 shows how the processes of each component interact with each other. Each process is supposed to be started once and run until the application is terminated. In Figure 3.7 the dataflow between the processes is shown. The `VirtualWhiteCaneService` connects to the `TangoService`, which acts as the Point Cloud Provider and provides the `TangoPointCloud` to the Ground Detection Module. The Ground Detection Module processes the point cloud using v disparity and RANSAC. The resulting obstacle cloud is polled by the Warning Module, which uses the `ObjectDetector` to create a `ConservativePolarHistogram`. This is used as input for the acoustic and tactile warning.

3.4.1 Point Cloud Provider

The main purpose of the Point Cloud Provider is to notify each subscriber whenever a new point cloud is available. As this is handled mostly by Google Tango, the design of this module is already given. Google Tango allows to connect a callback function, which is called whenever a new point cloud is available as shown in Listing 3.2.

However, it is important that this callback function does not do any processing on the data as no new data will be received until this callback returns. Google Tango further provides a struct `TangoSupportPointCloudManager`, which allows thread safe updates and access to the point cloud data. The data of this manager can

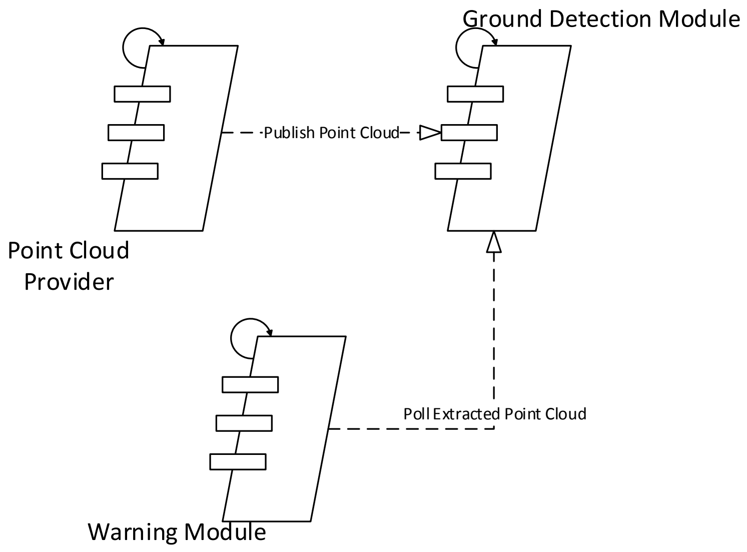


Figure 3.6: Process architecture of the system. The Point Cloud Provider continuously runs and provides new point cloud data to the Ground Detection Module. The Ground Detection Module processes this data and makes it available to the Warning Module, which periodically warns about the obstacles.

```
TangoErrorType TangoService_connectOnPointCloudAvailable(void(*)(void *
    context, const TangoPointCloud *cloud)
    TangoService_onPointCloudAvailable, ... )
```

Listing 3.2: Tango connect callback function on new point cloud.

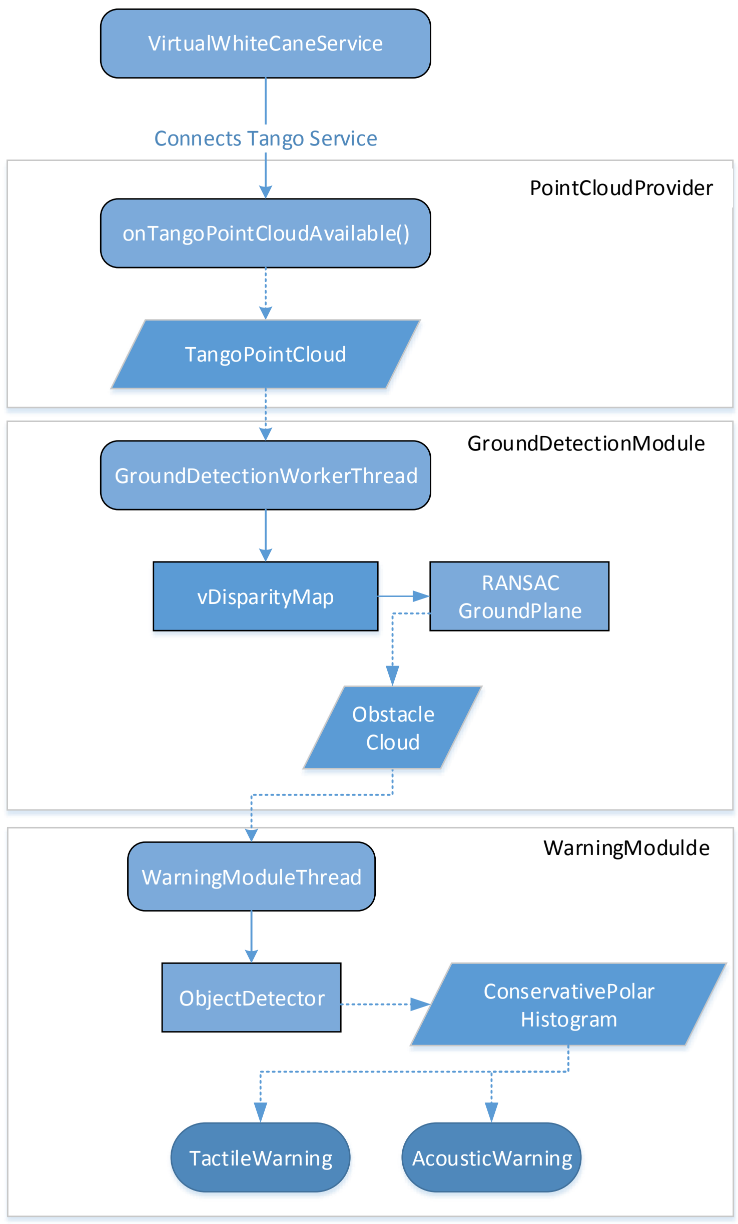


Figure 3.7: Dataflow diagram: The `VirtualWhiteCaneService` connects to the `TangoService`, which provides the point cloud to the Ground Detection Module. The Ground Detection Module generates a v disparity map and uses it as input for the RANSAC ground detection. The output of this is used as input for the Warning Module where it is used to create the `ConservativePolarHistogram` and create the warning.

```
std::vector<std::function<void()>*> subscriber_functions_pc_;
TangoSupportPointCloudManager *pointCloudManager_;

void TangoHandler::SubscribePointCloudUpdate(std::function<void()> *
    subscriber) {
      subscriber_functions_pc_.push_back(subscriber);
 }

void TangoHandler::onPointCloudAvailable(const TangoPointCloud *
    pointCloud) {
  //update pointCloudManager_ pointcloud data
  TangoSupport_updatePointCloud(pointCloudManager_, pointCloud);

  //call subscribed functions
  for (std::vector<std::function<void()> *>::iterator it =
    subscriber_functions_pc_.begin();
        it != subscriber_functions_pc_.end(); ++it) {
    (**it)();
  }
}
```

Listing 3.3: Subscribe callback function to be called as notification whenever a new point cloud is available.

therefore be updated with each `onPointCloudAvailable` callback. The registered callback further has to notify each subscriber about the new point cloud data. To subscribe, the subscriber has to register a callback function on their own, which gets called from within the `onPointCloudAvailable` callback (see Listing 3.3). Each subscriber is responsible to not do any processing on the data within its notification callback as this would block the `onPointCloudAvailable` callback.

Figure 3.8 shows this routine. As soon as new point cloud data is available, the `onPointCloudAvailable` callback is triggered, which in turn updates the `TangoSupportPointCloudManager` and notifies each subscriber.

3.4.2 Ground Detection Module

The Ground Detection Module is subscribed to the Point Cloud Provider described in the previous section. Subscription is done by registering a callback function to the Point Cloud Provider. The Ground Detection Module is implemented as a so-called functor, which allows to call an object like a function. To create a functor, the class has to overload `operator()()`, as demonstrated in Listing 3.4. Therefore,

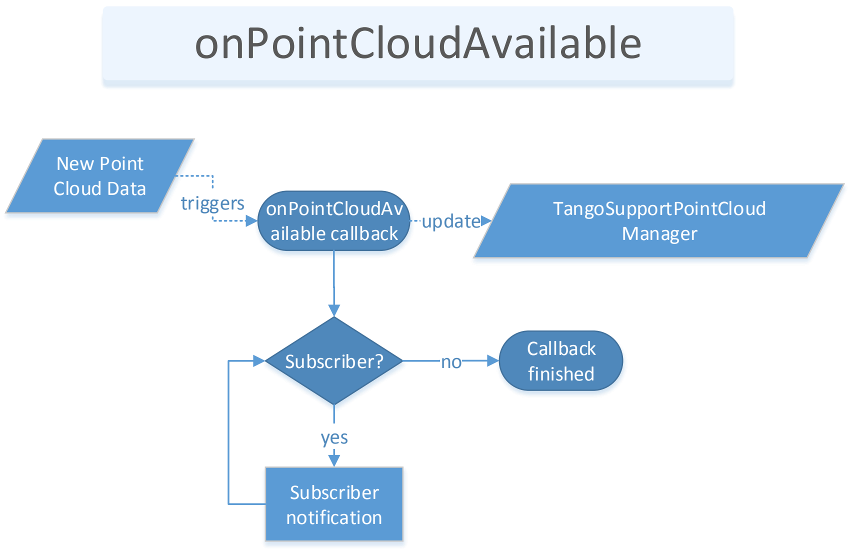


Figure 3.8: New point cloud data triggers the `onPointCloudAvailable` callback. Within this callback the `TangoSupportPointCloudManager` is updated with this new data. Further, each subscriber is notified.

it is possible to use the object itself as callback function for the subscription to the Point Cloud Provider. This module runs in its own thread to be able to process a new point cloud as soon as it is available without blocking the acquisition of new point cloud data or the Warning Module. Access to the raw point cloud data is done using the `TangoSupportPointCloudManager`.

```
TangoSupportPointCloudManager *pointCloudManager_;
std::condition_variable pointcloud_cv_;

void GroundDetectionWorkerThread::operator()() {
  //fetch latest point cloud manager
  pointCloudManager_ = tango_white_cane::util::tangoHandler.
    GetPointCloudManager();

  //wake up/notify thread loop to process new data
  pointcloud_cv_.notify_all();
}
```

Listing 3.4: To obtain a functor the class has to overload `operator()()`.

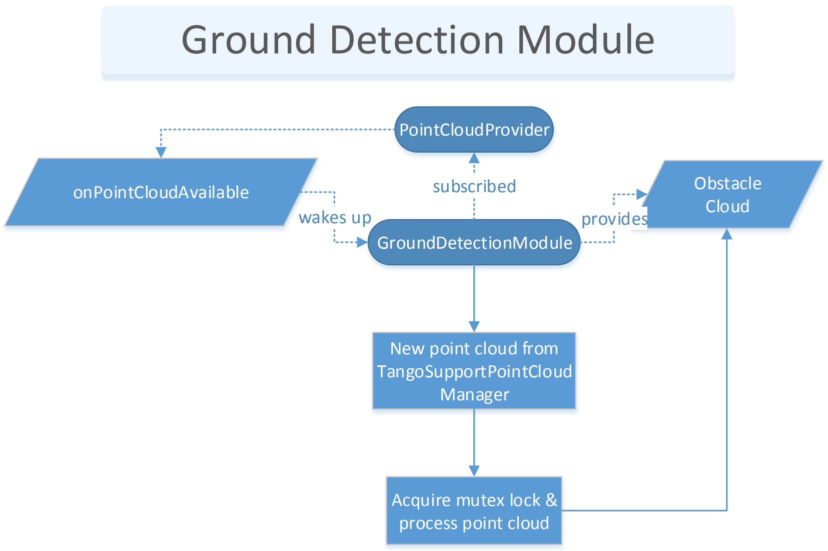


Figure 3.9: The Ground Detection Module is notified by the Point Cloud Provider as soon as a new point cloud is available. The Ground Detection Module then accesses the new point cloud data and acquires a mutex lock to safely process the point cloud.

As soon as new point cloud data is available from the Point Cloud Provider the Ground Detection Module thread wakes up and updates its obstacle point cloud. This is done in a two-step process, as described in Section 3.2.

It is crucial for the Warning Module to have access to the obstacle cloud. To ensure the validity of the point cloud data, a deep copy is required. Otherwise, it would be possible to update the data at the same time as it is read, leading to invalid data.

The dataflow of the Ground Detection Module is shown in Figure 3.9. The Point Cloud Provider notifies the Ground Detection Module which acquires the new point cloud using the `TangoSupportPointCloudManager`. Then a mutex lock is acquired to safely process the point cloud. Further, when the obstacle cloud is requested the lock is acquired and a deep copy of the obstacle cloud is returned to ensure it stays valid.

```
class ObjectDetector {
public:
  //Implementation has to ensure thread safety
  virtual std::vector<Eigen::Vector4f> GetObstacleCentroids() = 0;

  virtual void SetPointCloud(
          pcl::PointCloud<pcl::PointXYZ>::Ptr obstacleCloud) {
    std::lock_guard<std::mutex> lk(obstacle_cloud_mutex_);
    obstacleCloud_ = obstacleCloud;
  };

  virtual void SetMVP(float mvp[16]) {
    std::lock_guard<std::mutex> lk(obstacle_cloud_mutex_);
    std::copy(mvp, mvp + 16, mvp_);
  };

protected:
  std::mutex obstacle_cloud_mutex_;

  pcl::PointCloud<pcl::PointXYZ>::Ptr obstacleCloud_;
  float mvp_[16];
};
```

Listing 3.5: The `ObjectDetector` Interface.

3.4.3 Warning Module

The Warning Module runs within its own thread. This is crucial, as warning should not be blocked by any long running tasks like the Ground Plane Detection. The Warning Module has two main components: the object detector and the actual warning. This allows to easily change or enhance the way the warning is performed, as well as to change the method how obstacles are defined.

Object Detector Interface

It is required for the object detector to produce a list of 4-d vectors. The first three entries represent the 3-d coordinates around the device with fixed orientation (this is the OpenGL world coordinate system with the origin translated to the device). The fourth entry represents the bin number; this is not strictly necessary as this could be calculated from the world coordinates, but it saves some computational effort.

Even though this use of the 4-d vectors is tailored towards the implementation in this thesis, it is still possible to interchange the object detector rather easily. For it to work with the acoustic warning, only the first 3 coordinates are needed, while the tactile warning currently benefits slightly from the fourth entry.

The `ObjectDetector` interface (see Listing 3.5) provides a mutex lock, which ensures thread safety. The standard implementation of setting the point cloud to be processed and a possible transformation, which should be applied are already using this lock. However, to ensure the data does not change while it is processed within `GetObstacleCentroids` each implementation has to ensure thread safety by locking the `obstacle_cloud_mutex_`.

Warning Interface

The warning interface takes a list of obstacle coordinates as input. These coordinates are obtained by the object detector. It is up to the implementation of the warning module how these coordinates are interpreted.

Splitting the object detection and the warning interface into two separate classes allows to interchange the modules as needed. It further allows to reuse the output of an object detector for multiple warning modules. As already mentioned in the previous section, the tactile warning and the acoustic warning both can use the output of the same obstacle detector, however, a different module might interpret the given coordinates in a different way. For example, a different acoustic warning module might want to alter the sound depending on the size of the obstacle and therefore interpret two consecutive coordinates as a virtual rectangle bordering the obstacle.

```
class WarningModule {
public:
  virtual void warn(std::vector<Eigen::Vector4f> obstacleCoordinates)
   = 0;
};
```

Listing 3.6: The `WarningModule` interface.

Chapter 4

Implementation

This chapter describes implementation details of the Virtual White Cane application. It will first depict the hardware used, then the development environment and which frameworks and APIs were used. The last two sections will go into further detail about the software components and the algorithm design.

To be able to compile the software Android Studio 2.3 with the Android Native Development Kit (NDK), version 15b, is needed. Using a later release of the NDK might not work, as the used boost version does not work using unified headers and the old libc headers used by the NDK are considered deprecated and will be removed[1].

4.1 Hardware Setup

As of now (2017), there are currently only two Google Tango enabled devices available, the Lenovo Phab 2 Pro and the Asus Zenfone AR. At the start of this thesis the Asus Zenfone AR was only announced and not yet available for purchase. As only the Lenovo Phab 2 Pro was available and with Google Tango at the core of the application, the choice of hardware was therefore limited to one device.

The Lenovo Phab 2 Pro comes with a Qualcomm Snapdragon 652 mobile processor, which is an octa-core processor and comes with the Adreno 510 GPU. It has a 64-bit CPU architecture with the following specifications:

- $4x$ 1.8 GHz Cortex-A72
- $4x$ 1.4 GHz Cortex-A53

[1]https://android.googlesource.com/platform/ndk/+/ndk-r15-release/docs/UnifiedHeaders.md

- 64 GB ROM
- 4 GB RAM

Therefore, it has ample computing power and should be capable to cope with the demands of the application.

The GPU is also important, as it allows the use of Khronos OpenCL which can be used to speed up some computations dramatically. It has a clock of 600 MHz, 128 ALUs, a unified shader model, and unified memory. The unified memory is important, as this means the memory is shared with the CPU, therefore the GPU has access to the same data as the CPU without the need to copy it into another memory. This is very beneficial, as copying data between memories is often a bottleneck of General-purpose Computing on Graphics Processing Units (GPGPU) programs. The Adreno 510 has support for the following APIs:

- OpenCL 2.0 Full Profile
- OpenGL ES 3.2
- OpenGL 3.2
- Vulkan 1.0

While Vulkan is supported by the Adreno, it is not yet supported by the latest Android version (6.0 Marshmallow) available for the Lenovo Phab 2 Pro. However, it will be supported once Android 7.0 Nougat is available for the Phab 2 Pro. Even though there is no official support of OpenCL by Android, the Lenovo Phab 2 Pro comes with the Open CL 2.0 libraries, which allows developers to use it. Hence, the GPGPU programming language used was OpenCL.

It further comes equipped with a 224×172 pixel ToF depth camera, a 640×480 fish-eye lens camera used by Google Tango for motion tracking, and a high resolution full HD 1920×1080 pixel color camera. Further sensors included are:

- Compass / Magnetometer
- Proximity sensor
- Accelerometer
- Ambient light sensor
- Gyroscope
- Barometer

The Lenovo Phab 2 Pro is powered by a 4050 mAh Li-ion battery.

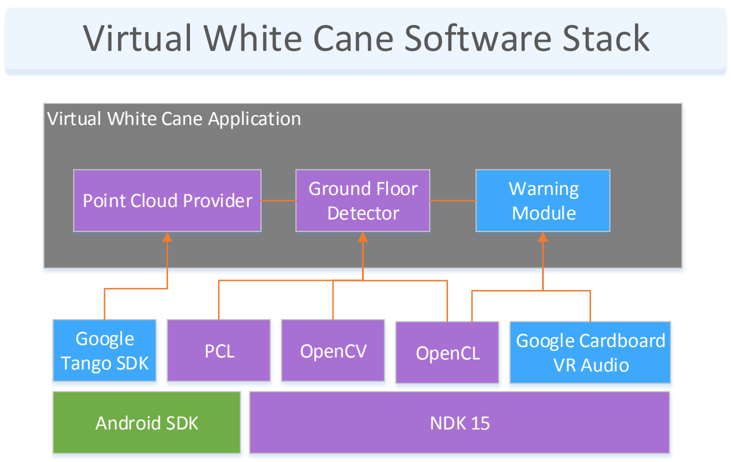


Figure 4.1: Software stack of the Virtual White Cane Application. Magenta colored boxes operate within the native layer, while the blue boxes operate in the native and the Java layer of Android.

4.2 Development Environment

The software was developed on Windows 7 using Android Studio 2.3 with NDK 15b. Figure 4.1 shows the different software components needed by the Virtual White Cane application. The magenta colored boxes operate entirely in the native layer, while the blue boxes use both, the native and the Java layer. This section describes these components in more detail.

4.2.1 Android Native Development Kit (NDK)

The Android Native Development Kit (NDK) helps with the development of Android applications using C/C++ code. It also provides some platform libraries, which allow to access physical device components like sensors. Using the NDK allows to reuse existing C/C++ libraries on Android and enables the developer to get some extra performance out of the device. Since Android Studio 2.2 it is possible to compile native code directly using Gradle. Communication between Java and native code is done using the Java Native Interface (JNI).

The NDK ships the toolchain to compile for the target platform, such as ARM, MIPS, or x86. Additionally, it allows to pack the compiled native libs into the apk

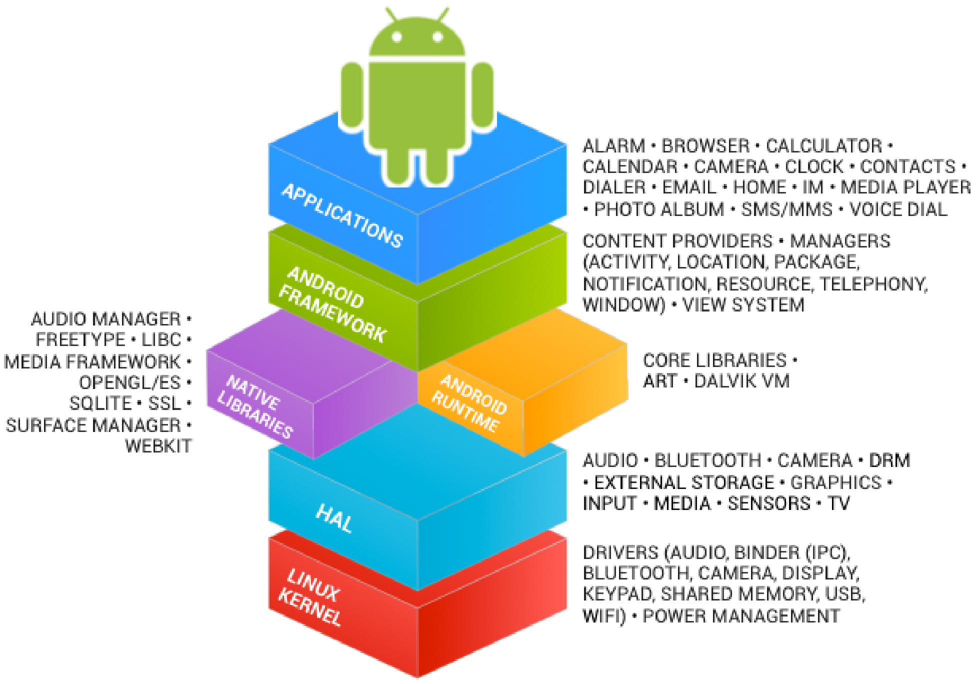

ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
APPLICATIONS
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
NATIVE LIBRARIES
ANDROID RUNTIME
CORE LIBRARIES • ART • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

Figure 4.2: Native libraries reside between the HAL and the Android Framework within the Android stack. (Figure from [52])

and therefore allows the developer to ship his native libraries with his application. Without this the native libs would have to reside in a specific folder on the device's internal memory (`'/sys/libs/'`), which is not accessible unless you are a OEM or work closely with one. However, using the NDK does not allow to overwrite the usual Android restrictions and every application still runs within its own sandbox. Figure 4.2 shows where native libraries reside within the Android stack.

Since Android Studio 2.2+ and the Android plugin for Gradle version 2.2.0+ it is possible to directly compile your native library with Gradle. This makes the build process straight forward, as everything is handled by Gradle. The Android plugin for Gradle allows to define an "externalNativeBuild" target where the targeted Application Binary Interfaces (ABIs) can be selected, as well as compiler flags or other arguments, for example CMake arguments. Despite the convenience given by the Android plugin for Gradle, a build file for the C/C++ source is still needed. There are two possible ways to configure the native build, namely ndk-build make files or CMake build files. Even though ndk-build make files are still supported for backwards compatibility reasons, it is recommended to start a new project using CMake build files. The configuration of the native library using CMake works just like any other C/C++ project using CMake though the paths for `find_*` calls

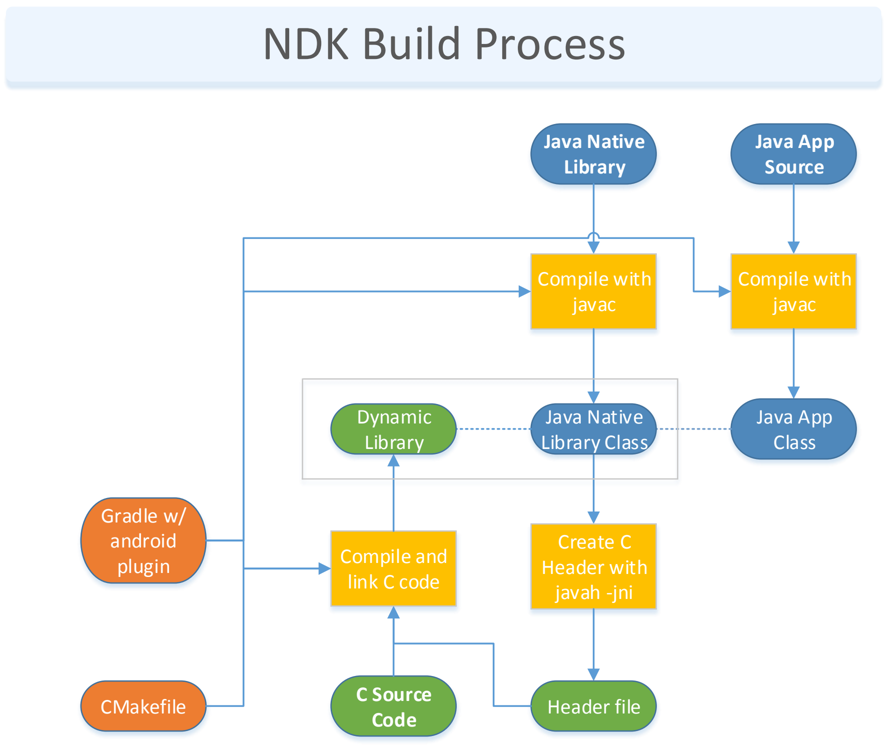


Figure 4.3: The 'Java App Source' and 'Java Native Library' (providing the JNI calls) are compiled by Gradle using javac. It is possible to generate the C Header using 'javah -jni' or write them by hand. The header and the C/C++ source code is then compiled according to the 'CMakefile'. (Figure adapted from[2])

are set to only search the sysroot path. Therefore, it prevents CMake to look for libraries on the host platform, which are probably compiled for the host platform. Consequently, one has to be aware of this, if adding other pre-compiled libraries is desired. Figure 4.3 shows how the components interact with each other during the NDK build process. The 'Java App Source', 'Java Native Library', 'C Source Code', and 'CMakefile' have to be written by the developer. The 'Java Native Library' represents the Java interface to the native library (see Java Native Interface (JNI)).

[2]M. Gargenta, *Learn about Android Internals and NDK - YouTube*. [Online]. Available: `https://www.youtube.com/watch?v=byFTAhXVF7k` (visited on 09/18/2017).

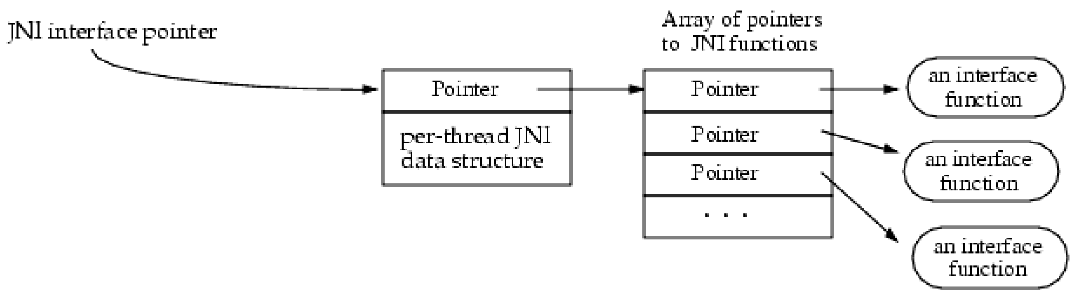


Figure 4.4: The JNI interface pointer. The interface pointer is a pointer to a pointer, which points to an array of pointers. Each pointer in the array points to an interface function. (Figure from [54])

4.2.2 Java Native Interface (JNI)

The Java Native Interface (JNI) is a standardized interface to call native code from Java. This allows to call for example C/C++ code directly from Java. It defines the naming and coding convention, such that the Java Virtual Machine (VM) knows how to find and call the native code. The JNI is built into the Java VM.

To indicate in Java which methods are supposed to be implemented in a native library, the method header needs to have the `native` keyword. Due to the `native` keyword the Java compiler assumes that the method is implemented elsewhere and accepts the unimplemented method. To load a native library, the `System.loadLibrary` method has to be called.

To access Java VM features, native code needs to call JNI functions. Those functions are available through an interface pointer which is structured like a C++ virtual function table (see Figure 4.4).

A native JNI function's first parameter is always a pointer to the `JNIEnv`, which is the interface pointer. This interface pointer is only valid within the current thread. The second parameter is always a `jobject`, representing the "this" pointer of the caller. All following parameters are the parameters defined by the function.

Data passed to a native function is always a local reference. This means it will be valid only until the end of the function. Normally, a developer does not have to worry about garbage collection of local references, as this will happen automatically. However, it is possible to manually delete these references once they are no longer needed. This might be particularly useful if large array elements are accessed and processed, but are no longer needed afterwards. To create a persistent reference, which will not succumb garbage collection at the end of the scope of the function, it has to be made global reference. A global reference will never be released automatically, therefore the developer has to delete those references once they are no longer needed.

As already mentioned earlier, a JNI interface pointer is only valid within a thread and it cannot be passed onto another thread. In case of the creation of a new thread in native code, it has to register to the Java VM by calling the `AttachCurrentThread` method. An attached thread has to make sure it detaches from the Java VM by calling the `DetachCurrentThread` method prior to termination, as the Java VM cannot unload as long as threads are attached to it. An attached thread is able to get its own JNI interface pointer by calling the `GetEnv` method from the Java VM.

JNI usually allows multiple Java VMs to exist. However, the Android implementation of JNI only allows a single Java VM. The Java VM provides the "invocation interface" functions. The best way to keep a reference to this single Java VM is during the `JNI_OnLoad` method.

Figure 4.5 shows how the JNI changes the way Android applications access native libraries. While usually Applications written entirely in Java would use the Android framework to access system libraries, using the JNI allows to directly access native libraries.

4.2.3 Google Tango SDK

The Google Tango SDK basically is the heart of the application. It provides not only the depth readings from the ToF camera, but also various transformations, such as from the depth camera coordinate system to the color camera. With the frames of reference defined by Google Tango it is possible to put the data into the same coordinate system.

Basically Google Tango distinguishes between the following two slightly different right-hand convention coordinate systems:

- Right Hand Local Level
- Right Hand Android (Identical to Android Sensor coordinate system)

The Right Hand Local Level coordinate system has the x axis horizontal and the positive direction pointing to the right, the z axis is vertical with positive direction

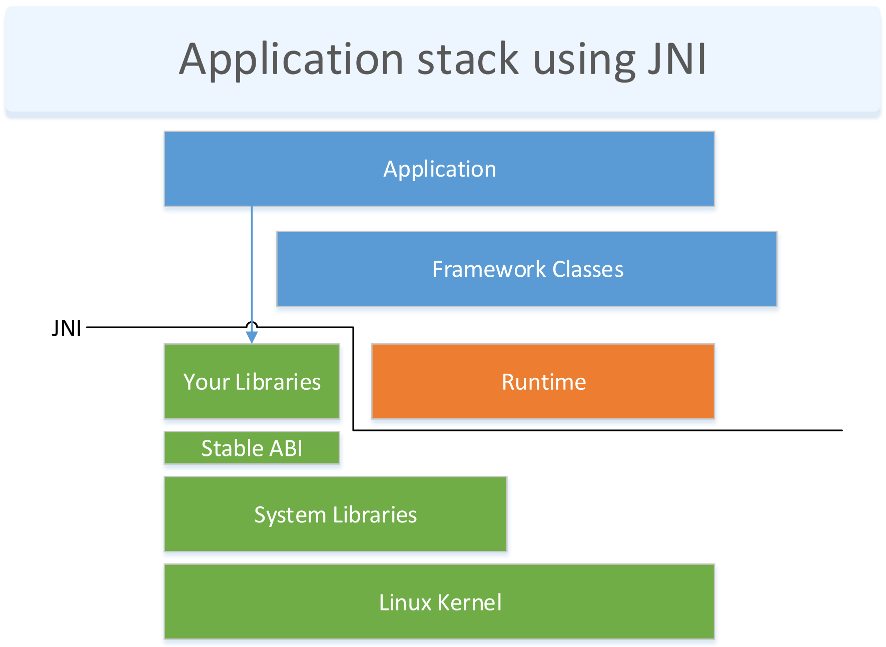


Figure 4.5: Normally, an Android application written entirely in Java would use the Android framework, which in turn uses JNI internally, to access underlying system libraries. Using the JNI enables the application code to bypass the Android framework and access your own native libraries.

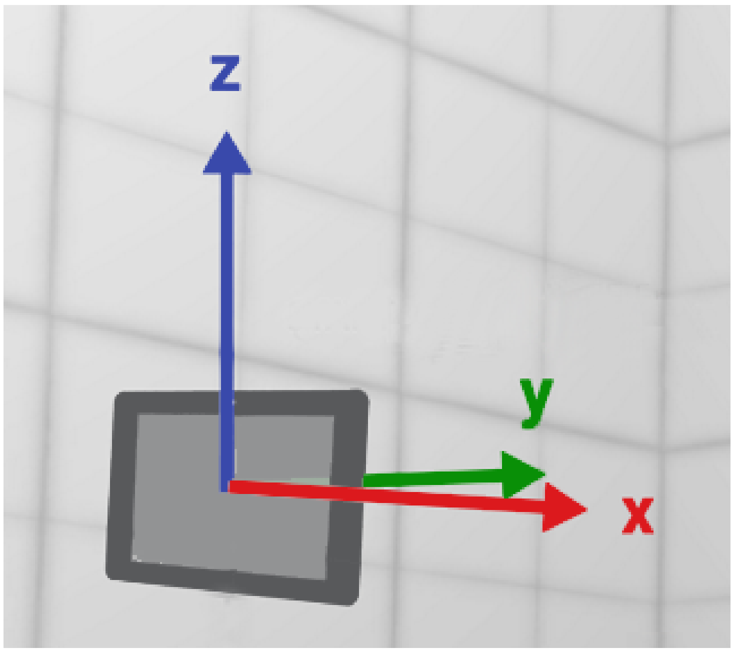


Figure 4.6: The right hand local level system. The x axis is horizontal and pointing to the right. The z axis is vertical and pointing up and the y axis is depth and pointing away from the user. (Figure from [6])

Base Frame	Coordinate System
COORDINATE_FRAME_START_OF_SERVICE	Right Hand Android
COORDINATE_FRAME_AREA_DESCRIPTION	Right Hand Local Level
COORDINATE_FRAME_DEVICE	Right Hand Android

Table 4.1: The Base Frame and its associated coordinate system.

pointing up and the y axis represents the depth pointing away from the user (see Figure 4.6).

The Right Hand Android coordinate system is identical to the Android Sensor coordinate system. If the device is held in the default orientation, the x axis is also horizontal and in positive direction pointing to the right. The y axis is vertical and positive direction pointing up and the z axis is pointing towards the user, representing depth (see Figure 4.7).

The different frames of reference used by Google Tango have different coordinate systems associated with them. See Table 4.1 for more information.

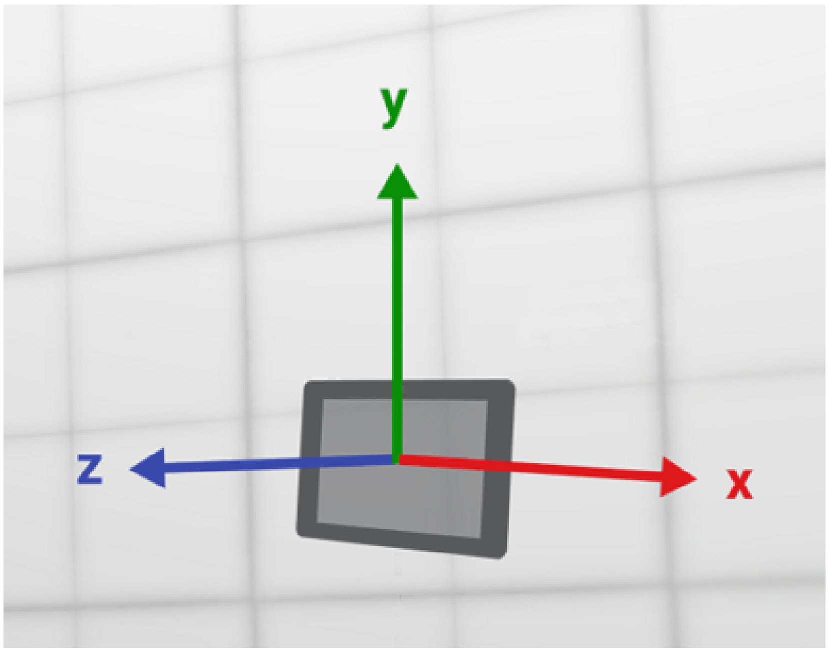


Figure 4.7: The right hand android system. The x axis is horizontal and pointing to the right, the y axis is vertical and pointing up, and the z axis is depth and pointing towards the user. (Figure from [6])

Each component such as the IMU, the depth camera, the color camera of a device resides in its own coordinate system with itself at the origin. Using intrinsic and extrinsic parameters, it is possible to calculate a transformation between them. An example extrinsic parameter would be the translational difference between the IMU and the color camera on the device. In other words, the extrinsic parameters account for the positional differences of the sensors. The intrinsic parameters of a camera are, for example, the focal length or the field of view. These intrinsic and extrinsic parameters are needed to transform, for example, the depth camera's readings into the color camera's image and can be obtained using the `TangoService_getCameraIntrinsics` method.

The Tango SDK provides the depth readings from the ToF camera as an unordered point cloud. However, given the camera intrinsics, it is easy to reproject those points into an ordered image.

Given the 3-d point (X, Y, Z) one can calculate the pixel coordinate by:

$$\begin{aligned} x &= \frac{X}{Z} * f_x + \frac{r_d}{r_u} + c_x, \\ y &= \frac{Y}{Z} * f_y + \frac{r_d}{r_u} + c_x \end{aligned} \tag{4.1}$$

with the normalized radial distance defined as

$$r_u = \sqrt{\frac{X^2 + Y^2}{Z^2}} \tag{4.2}$$

and the distorted radial distance as

$$r_d = r_u + k_1 + r_u^3 + k_2 * r_u^5 + k_3 * r_u^7. \tag{4.3}$$

The focal length f_x and f_y, the principal point c_x and c_y, as well as the distortion coefficients k_1, k_2, and k_3 are part of the intrinsic calibration. However, the distorted radial distance is depending on the camera model used. Equation 4.3 is only valid, if Brown's camera model [55] is used, as the calibration for the ToF camera does.

Another important system is the OpenGL coordinate system. While the axis follow the same notion as the Right Hand Android coordinate system, the OpenGL world frame defines a fixed point in space as origin. In the default orientation, an Android device can be used to control an OpenGL camera and they would have the same coordinate system. Figure 4.8 shows how the OpenGL world frame has a fixed point in space as origin, while the camera can move freely and will always have the camera as origin. In the default orientation, the OpenGL camera coordinate system is equal to the Right Hand Android system.

The Google Tango SDK provides a support function to easily get the transformation matrices between two key frames (see Listing 4.1). It is also possible to define the target and base coordinate systems (engine). The target frame and engine define which coordinate system the matrix converts from and the base frame and engine where the matrix converts to. The timestamp parameter defines the timestamp of the transformation matrix, while 0 will return the latest transformation matrix available. Display rotation is the index of the rotation between the device's default orientation and the current orientation. The transformation matrix is returned using a pointer as parameter in column-major order as doubles. A version of this function, which returns the matrix as floats does also exist. The return value of the

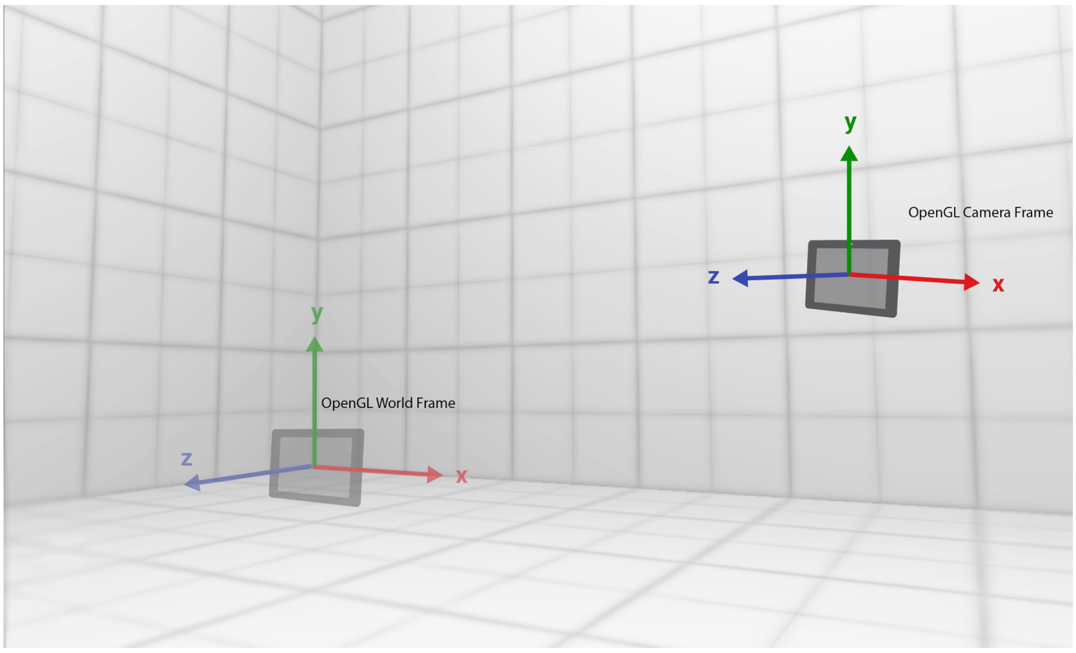


Figure 4.8: The OpenGL coordinate system. (Figure from [6])

function is either **TANGO_SUCCESS** on success, **TANGO_ERROR** on an error, or **TANGO_INVALID** if the input is invalid.

```
TangoErrorType TangoSupport_getDoubleMatrixTransformAtTime(
  double timestamp,
  TangoCoordinateFrameType base_frame,
  TangoCoordinateFrameType target_frame,
  TangoSupportEngineType base_engine,
  TangoSupportEngineType target_engine,
  TangoSupportRotation display_rotation,
  TangoDoubleMatrixTransformData *matrix_transform
)
```

Listing 4.1: Tango support function to obtain a transformation matrix.

4.2.4 Khronos OpenCL (on Android)

To improve the performance, some algorithms were designed and implemented to be run in parallel. This was achieved using Khronos OpenCL. Due to the fact that OpenCL is intended as a cross-platform standard, it is easy to run on any supported device. This section will give a brief overview over OpenCL in general and how it can be used with Android.

Khronos OpenCL (Open Computing Language)[3] is an open, royalty-free standard to perform cross-platform parallel programming for many different processors found in personal computers or even mobile devices. It consists of the following core ideas [57]:

- Platform Model
- Execution Model
- Memory Model
- Programming Model

The OpenCL Platform Model consists of one host and one to many compute devices. An example for a compute device might be a 4 core CPU, a powerful graphics card with hundreds of cores, or even a mobile phone. A compute device has one to many compute units, each of them having again one to many processing elements. In Figure 4.9 this abstraction how OpenCL views the hardware is shown graphically.

The Execution Model splits the program into two parts: the host program and the kernel. The host program is responsible for creating and managing the context in which the kernel will execute. This well-defined context includes information about the following:

• Devices	The devices exposed by the OpenCL platform.
• Kernel Objects	The OpenCL functions and their arguments that run on the OpenCL device.
• Program Objects	The program containing the kernel functions.
• Memory Objects	Variables used by the host and OpenCL devices. Instances of kernels operate on these.

The work of a Kernel is done on "work-items", which are grouped into "work-groups". The amount of work-items and work-groups to be launched is defined in an index space called "NDRange index space". This NDRange index space gives each work-item a unique global ID, as well as a local ID within its associated work-group. Each work-group is also assigned a unique group ID. An example of how a 2-d index space could look like is seen in Figure 4.10.

[3] *OpenCL - The open standard for parallel programming of heterogeneous systems.* [Online]. Available: `https://www.khronos.org/opencl/` (visited on 04/27/2017).

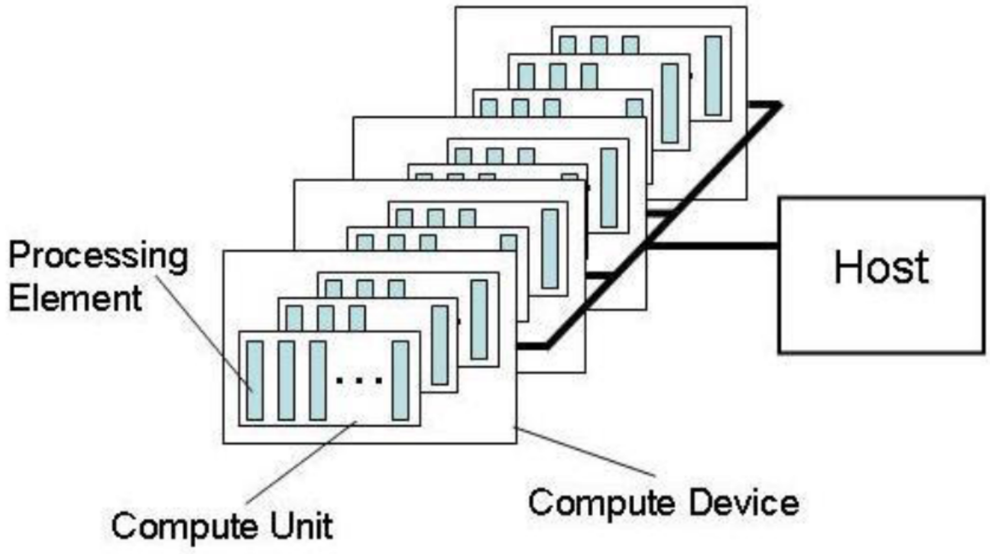


Figure 4.9: The OpenCL platform model. The host has one to many compute devices. Each compute device has one to many compute units, which again have one to many processing elements. (Figure from [57])

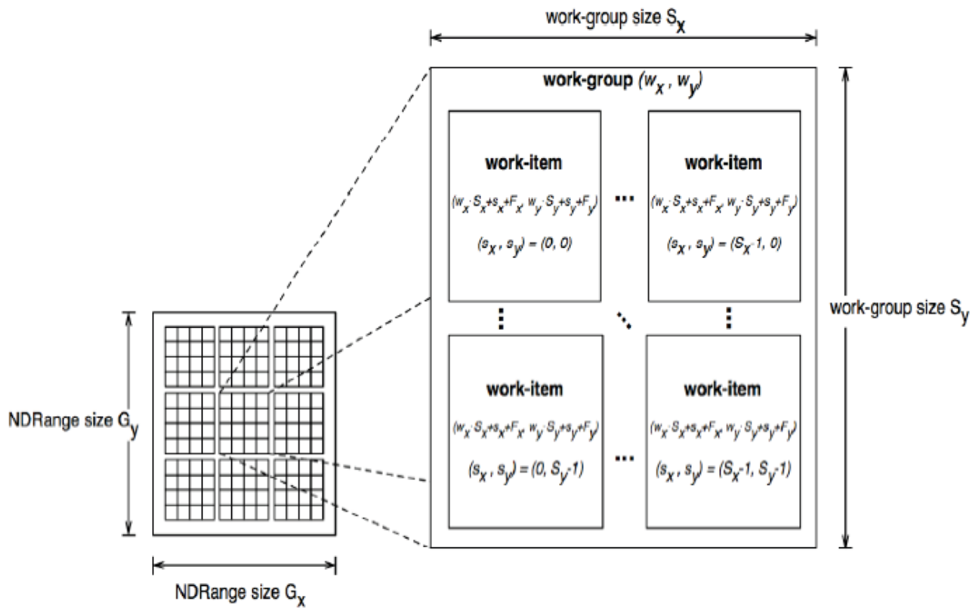

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x+F_x, w_y \cdot S_y+s_y+F_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 4.10: Example NDRange index space. Each work-item has its own global and local ID. Work-groups have their own unique IDs. (Figure from [57])

Interaction between the host and the device is handled through a command-queue. This command-queue has three command types:

• Kernel-enqueue commands	Kernel to be executed on a device.
• Memory commands	Transfer data between host and device memory objects.
• Synchronization commands	Explicit synchronization commands.

Commands submitted to a command-queue will only execute, if prerequisites are met. These prerequisites might be from two possible sources. Either a command-queue that constrains the order of launch of commands, or there exist dependencies between commands. These dependencies can be expressed through events.

The Memory Model logically divides the memory into host and device memory. The behaviour of the host memory is defined outside of OpenCL. Device memory is directly available to the kernel and it has 4 memory regions or address spaces, as seen in Figure 4.11.

• **Global Memory**	each work-item has read/write access
• **Constant Memory**	each work-item has read access
• **Local Memory**	restricted to work-items in a work-group
• **Private Memory**	only the work-item has access

There are no restrictions from OpenCL towards a physical difference between these different memory regions. Still, devices often provide dedicated memories for each region with different latencies, but there might be an overhead accessing these [58].

OpenCL is not officially supported by Android, however, many manufacturers of Android devices do provide OpenCL support. The prebuilt libraries are often found on the phone and can be obtained using the Android Debug Bridge (adb). Usually, Android applications are written using Java and native OpenCL host code is written using C/C++. Even though Java OpenCL bindings[4] exist, the host code was written in C/C++ utilizing the Android NDK and Java is only used for the user interface. OpenCL kernels are usually supplied as an asset in form of a plain text file. Communication between the Java code and the native C/C++ code is

[4]*jocl.org - Java bindings for OpenCL*. [Online]. Available: `http://www.jocl.org/` (visited on 07/05/2017).

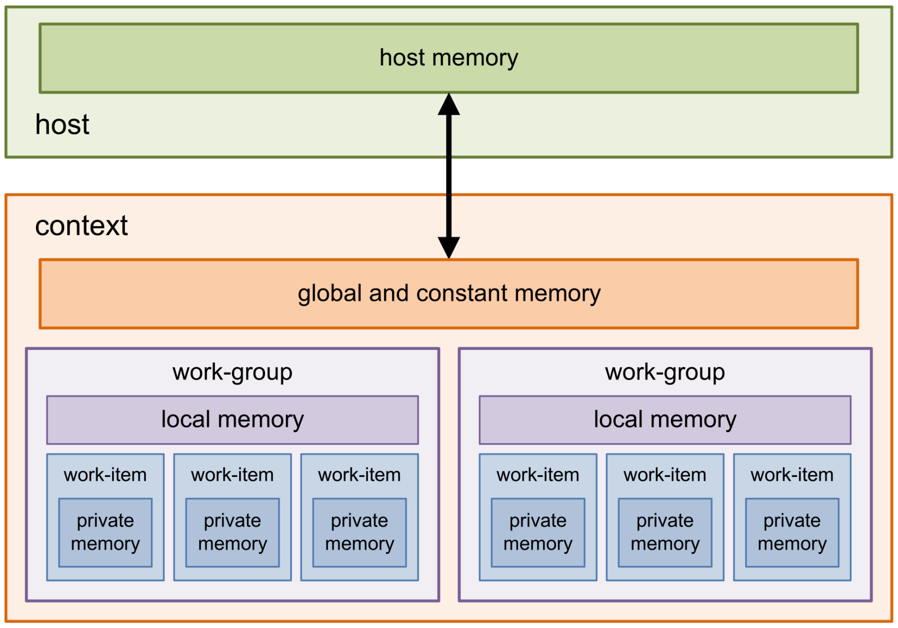


Figure 4.11: The OpenCL memory model consists of host memory and “context” memory. The context represents the device on which the code will be executed. It has several different memory regions with different access speeds, permissions, and sizes. (Figure from [59])

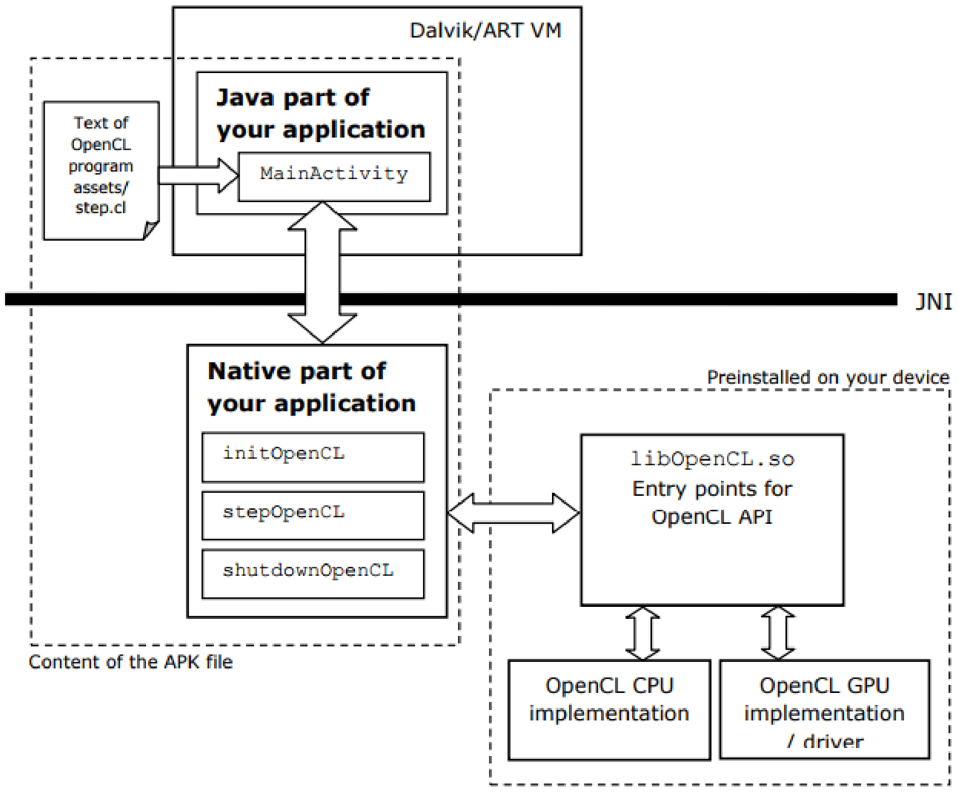

Dalvik/ART VM
Text of OpenCL program assets/ step.cl
Java part of your application
MainActivity
JNI
Native part of your application
initOpenCL
stepOpenCL
shutdownOpenCL
Content of the APK file
Preinstalled on your device
libOpenCL.so Entry points for OpenCL API
OpenCL CPU implementation
OpenCL GPU implementation / driver

Figure 4.12: Usually, the interaction between Android and OpenCL is using the Android NDK and the JNI. Java is used for the user interface, while native C/C++ code is used to interact with the libOpenCL.so library. Communication between the Java code and C/C++ code is done via the Java Native Interface and the OpenCL kernels are supplied as text file assets. (Figure from [61])

done via the Java Native Interface. A schematic view of this interaction between Java, C/C++, and OpenCL C is shown in Figure 4.12.late

4.2.5 Google Cardboard VR Audio

The acoustic warning is a crucial part of the application, therefore, it is important to be able to precisely position and locate a sound source. Android does allow developers to use OpenSL ES using the NDK, however, it does not implement any of its profiles fully [62]. While, according to its specification, OpenSL ES would implement a 3-d audio using a HRTF in the gaming profile [63], Android unfortunately only implements the pan and fade features. Therefore, the Google

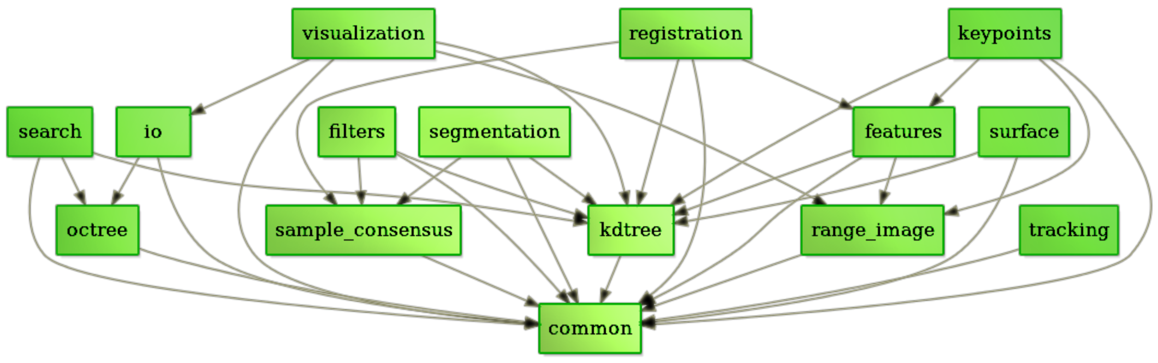


Figure 4.13: The dependency graph of the PCL modules. This modular approach allows to compile and link only the needed parts of PCL. (Figure from [5])

Cardboard VR Audio library is used, which implements a powerful audio engine supporting HRTF. As Google Cardboard VR Audio is made for mobile devices, integration into an existing Android project happens in a straight forward manner. The Google Cardboard VR Audio library allows to easily place a sound source in space and update the head pose. As the head pose, the device's pose is used. This allows the user to move the phone the same way one would move the head to increase sound localization.

Unfortunately, this introduces yet another coordinate system called the head space. In the head space, the head is at the origin and faces the negative z direction, this coincides with the Right Hand Android system described in Section 4.2.3. Sound sources are placed around the user in an OpenGL world coordinate system, translated such that the mobile phone is the origin.

4.2.6 Point Cloud Library (PCL) on Android

The Point Cloud Library (PCL)[5] from Rusu and Cousins [65] is a large scale, cross platform, open project to perform point cloud processing. It is released under the 3-clause BSD license and therefore free for commercial and research use.

To help reduce the size on platforms with computational or size constraints, PCL is build modular. Modules can be compiled and linked separately, which allows to only link the parts needed for the given application. Figure 4.13 gives an overview of PCLs dependencies.

[5] *PCL - Point Cloud Library (PCL)*. [Online]. Available: `http://www.pointclouds.org/` (visited on 05/02/2017).

As there is currently no pre-compiled version of the PCL available for Android, it is necessary to compile it yourself. This was done on a virtual machine running Ubuntu 16.04 LTS 64-bit[6], git version 2.7.4, CMake version 3.5.1, and the Android NDK r15b. Using a newer NDK version might not work, as there is a bug in the boost library with unified headers which replace the current libc headers supplied with the NDK. As PCL depends on boost[7], FLANN[8], and Eigen[9], it is suggested by the PCL documentation[10] to use a "superbuild". The problem with the superbuild provided by the PCL documentation is that it is outdated, but various forks of the original github project exist[11], allowing to build boost 1.60, FLANN 1.8.5, Eigen 3.2.8, and PCL 1.7.2 for Android.

Once the library is successfully built, it is necessary to force CMake to search not only the local, but also the system root path. The Android Toolchain is set to only search the local path to prevent CMake to link to libraries, which are not compiled for Android. Hence, forcing CMake to search the system root path has to be done carefully.

4.2.7 OpenCV

OpenCV is a software library written in optimized C/C++ to provide fast image processing capabilities. It is released under a BSD license. OpenCV provides a vast array of optimized algorithms, both classic and state-of-the-art computer vision and machine learning algorithms.

OpenCV is available for Android and the SDK provides the C++ library, to be used via JNI, as well as a Java wrapper to be used directly. The OpenCV Java API has a minimum platform requirement of Android 2.2. To be able to easily import the C++ library into an Android project, the SDK comes with ndk-build files, as well as a 'OpenCVConfig.cmake' file. Using these files, importing the library can be achieved conveniently.

As the whole application heavily relies on native C++, code the native OpenCV library is used in this thesis.

[6]https://www.ubuntu.com/
[7]http://www.boost.org/
[8]http://www.cs.ubc.ca/research/flann/
[9]http://eigen.tuxfamily.org/
[10]http://www.vtk.org/Wiki/VES/Point_Cloud_Library
[11]https://github.com/Sirokujira/pcl-superbuild

4.3 Software Components

This section will describe the implementation details of the software components and how they interact with each other. The Android application is written in Java and C++. Java is only used for the user interface and as a start up point for the application. Further, it handles the connection to the Google Tango service and other parts, which cannot be accessed easily using native C++ code, like activation of the vibrator or detecting screen rotation. The C++ code is the core of the application. Once the application is started and initialized, everything is handled directly in the native layer.

The C++ code is sectioned into multiple smaller libraries with different responsibilities:

virtualWhiteCane	The main application library. It makes use of the other library functionalities.
groundPlaneDetection	Provides the different functions to detect the ground plane, such as calculation of the v disparity map.
virtualWhiteCaneUtil	Is a collection of useful utility classes. This library is intended to ease the use of some JNI functionality and often used OpenCL calls. Those classes also provide a global "default implementation".
warningModule	holds the classes responsible for warning and object detection.

The following sections will describe these libraries in more detail.

4.3.1 Java Components

The main Java components of the application are the `VirtualWhiteCaneActivity`, the `VirtualWhiteCaneService`, and the `TangoJNINative` classes. The main activity of the application is the `VirtualWhiteCaneActivity`, thus, it extends the Android `Activity` class. Upon start up of the application, it launches the `VirtualWhiteCaneService`. Its main purpose is to provide debug visuals and controls, and is crucial to launch the service. However, depth readings of obstacles are projected onto them in red with the intensity inversely proportional to the distance (the closer the obstacle, the stronger the hue). This could help a partially sighted person avoid these directions. Additionally, once the activity is destroyed by the user it stops the `VirtualWhiteCaneService` to free the `TangoService`. It is important for the `VirtualWhiteCaneService` to keep the `TangoService` until it is destroyed by the user, as otherwise the warning might stop unexpectedly, giving the impression of a free path.

Figure 4.14 shows a UML diagram containing these classes. For the sake of clarity further classes used for the user interface are omitted, as well as functions, which are only used for debug purposes. The `TangoInitializationHelper` is placed on the outside of the package, as it is part of Google Tango.

The `VirtualWhiteCaneService` class is run as a background service. Therefore, it extends the Android `Service` class. This allows the warning to be active, even if the application is not in the focus. As soon as it is launched, it binds the Tango service using the `TangoInitializationHelper` provided by Google and launches the native part of the application using the `TangoJNINative` interface.

Lastly, the `TangoJNINative` interface holds all the calls to the native layer. Its main purpose is to tell Java which native functions are available in the native `virtualWhiteCane` library. Further, it is responsible for loading these shared native libraries providing the implementations of the function declarations:

libtango_client_api	The Tango shared library providing the Google Tango functionality.
virtualWhiteCane	The main library of the application developed using the NDK having the implementations of the methods declared in TangoJNINative.
gvr_audio	The Google VR audio library providing 3-d audio support using HRTF.

4.3.2 virtualWhiteCane Library

This library handles the interaction between the other libraries and is the core of the Virtual White Cane application. It has the C++ part of the JNI interface, which is the `jni_interface`. This interface implements all the previously defined JNI methods. The `VirtualWhiteCaneApp` class is called by the `VirtualWhiteCaneService` via JNI, where it starts two threads. The first thread is the `GroundDetectionWorkerThread`, which is subscribed to the `onPointcloudAvailable` callback and processes the point cloud to detect the ground floor. The second thread is the `WarningWorkerThread`.

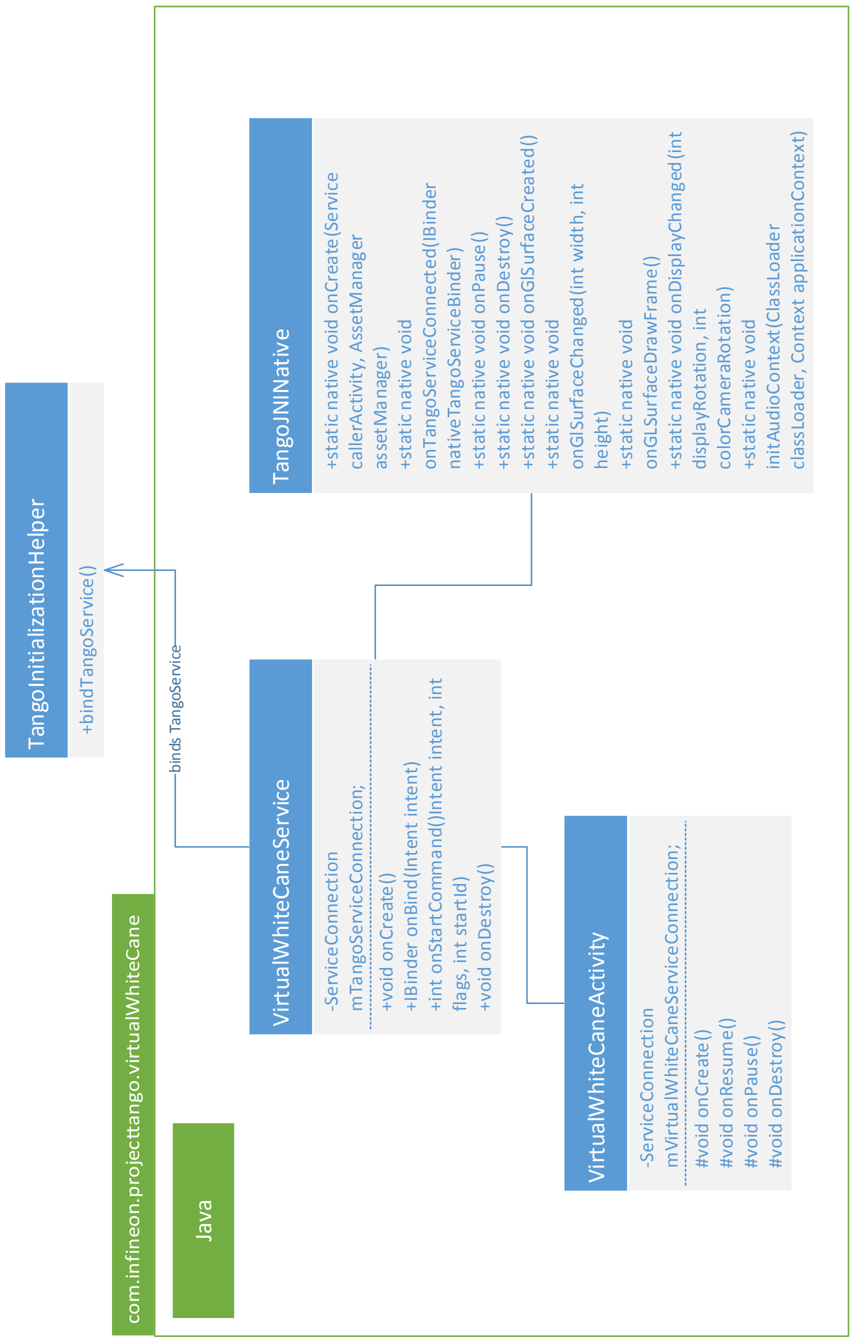

TangoInitializationHelper
+bindTangoService()
binds TangoService
com.infineon.projecttango.virtualWhiteCane
Java
VirtualWhiteCaneService
-ServiceConnection mTangoServiceConnection;
+void onCreate()
+IBinder onBind(Intent intent)
+int onStartCommand()Intent intent, int flags, int startId)
+void onDestroy()
VirtualWhiteCaneActivity
-ServiceConnection mVirtualWhiteCaneServiceConnection;
#void onCreate()
#void onResume()
#void onPause()
#void onDestroy()
TangoJNINative
+static native void onCreate(Service callerActivity, AssetManager assetManager)
+static native void onTangoServiceConnected(IBinder nativeTangoServiceBinder)
+static native void onPause()
+static native void onDestroy()
+static native void onGlSurfaceCreated()
+static native void onGlSurfaceChanged(int width, int height)
+static native void onGLSurfaceDrawFrame()
+static native void onDisplayChanged(int displayRotation, int colorCameraRotation)
+static native void initAudioContext(ClassLoader classLoader, Context applicationContext)

Figure 4.14: UML of the Java part. The `VirtualWhiteCaneActivity` starts the `VirtualWhiteCaneService`, which binds the `TangoService`. Communication with the native layer is done using the `TangoJNINative` interface.

This thread polls the `GroundDetectionWorkerThread` to get the obstacle point cloud and processes it into obstacles and warns about them.

The `TangoHandler` is another vital part of the application. It has a global default implementation and it constructs the Google Tango configuration, as well as defines its callbacks. This process is started once the `TangoService` is bound. The `GroundDetectionWorkerThread` subscribes to the `TangoHandler` to receive updates once new point clouds are available.

Figure 4.15 shows a UML diagram with the most important methods and members of the classes.

jni_interface

This interface is the connection between Java and C++. Each method defined as `native` in the `TangoJNIInterface` in Java is implemented here. It also initializes a static `VirtualWhiteCaneApp` object.

TangoHandler

The `TangoHandler` is responsible for binding the `TangoService` and configuring it properly. A default object is globally instantiated. The `jni_interface` calls the `OnTangoServiceConnected` method of the default object. Once the `TangoService` is bound, it can be configured and the callback methods are registered.

The `TangoHandler` is responsible for informing subscribers about new point cloud data. To do so, it is possible to subscribe a callback function using the method `SubscribePointCloudUpdate`. Within the `onPointCloudAvailable` callback these functions are called. The callback registered by the `GroundDetectionWorkerThread` wakes the thread up, so it can process the new point cloud.

VirtualWhiteCaneApp

This class represents the C++ part of the `VirtualWhiteCaneService` and the `VirtualWhiteCaneActivity`. It is responsible for resource acquisition upon creation, releasing resources once destroyed and only keeping resources needed to run in the background while the activity is paused.

Further, it handles the rendering of the OpenGL `GLSurfaceView`, which was created by the `VirtualWhiteCaneActivity`. With every `onDrawFrame` call, the color camera output is rendered to the display with the depth points on top of the obstacles. The depth value is converted to a hue of red, inversely depending on the distance of the object. The closer the object, the more saturated the red is.

Another vital responsibility of the `VirtualWhiteCaneApp` is to start two additional threads during `onCreate`: the `GroundDetectionWorkerThread` and the `WarningWorkerThread`. It also has to stop them, once the application is terminated. The threads are not paused during the `onPause` event because the warning should continue to run even if the application is not active on the screen.

GroundDetectionWorkerThread

Once the `GroundDetectionWorkerThread` is started by the `VirtualWhiteCaneApp` it registers itself to the `TangoHandler` to receive point cloud updates. To do so the `operator()()` is overwritten. This allows the thread object to be called like a function. Once the callback is called the thread wakes up and processes the new point cloud.

As soon as the thread is woken up, it tries to acquire a mutex lock to ensure the obstacle data cannot be read by any other thread. With the lock acquired, the new point cloud data can be processed. Whenever the outcome of the ground plane detection is accessed, a deep copy of the point cloud has to be returned to ensure thread safety.

WarningWorkerThread

The `WarningWorkerThread` periodically polls the `GroundDetectionWorkerThread` for the obstacle cloud. The obstacle cloud is then used as input for the Conservative Polar Histogram algorithm. This returns a list of obstacle centroids for each direction given by the binning of polar coordinates.

The `WarningWorkerThread` further has a list of `WarningModules`. Each `WarningModule` is called sequentially with the obstacle centroids as input. It is also the `WarningWorkerThread`'s responsibility to supply warning modules with additional resources if needed, like the `AcousticWarningModule`, which is supplied the Google VR audio API upon creation.

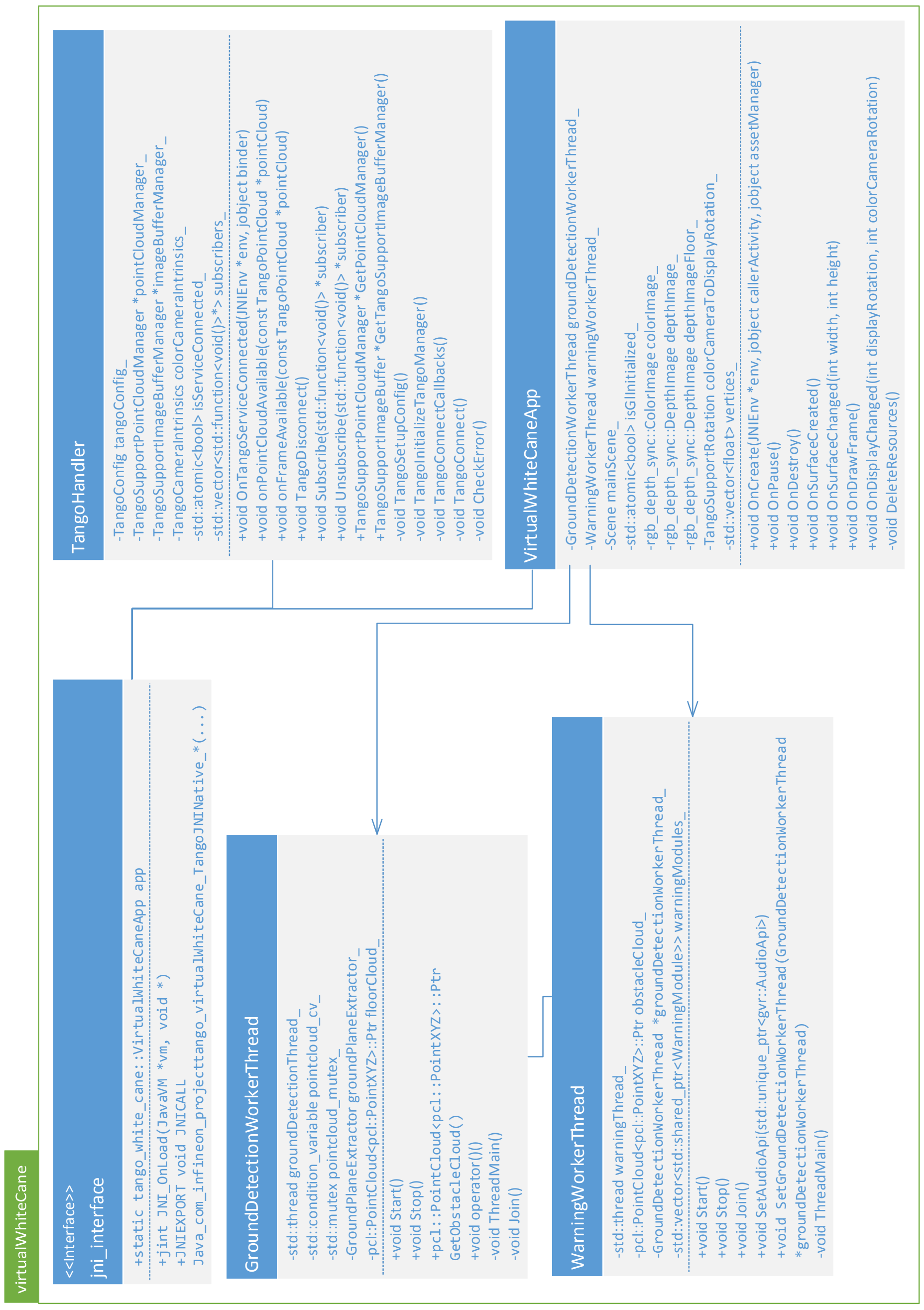

virtualWhiteCane
<<Interface>>
jni_interface
+static tango_white_cane::VirtualWhiteCaneApp app
+jint JNI_OnLoad(JavaVM *vm, void *)
+JNIEXPORT void JNICALL
Java_com_infineon_projecttango_virtualWhiteCane_TangoJNINative_*(...)
GroundDetectionWorkerThread
-std::thread groundDetectionThread_
-std::condition_variable pointcloud_cv_
-std::mutex pointcloud_mutex_
-GroundPlaneExtractor groundPlaneExtractor_
-pcl::PointCloud<pcl::PointXYZ>::Ptr floorCloud_
+void Start()
+void Stop()
+pcl::PointCloud<pcl::PointXYZ>::Ptr
GetObstacleCloud()
+void operator()()
-void ThreadMain()
-void Join()
WarningWorkerThread
-std::thread warningThread_
-pcl::PointCloud<pcl::PointXYZ>::Ptr obstacleCloud_
-GroundDetectionWorkerThread *groundDetectionWorkerThread_
-std::vector<std::shared_ptr<WarningModule>> warningModules_
+void Start()
+void Stop()
+void Join()
+void SetAudioApi(std::unique_ptr<gvr::AudioApi>)
+void SetGroundDetectionWorkerThread(GroundDetectionWorkerThread
*groundDetectionWorkerThread)
-void ThreadMain()
TangoHandler
-TangoConfig tangoConfig_
-TangoSupportPointCloudManager *pointCloudManager_
-TangoSupportImageBufferManager *imageBufferManager_
-TangoCameraIntrinsics colorCameraIntrinsics_
-std::atomic<bool> isServiceConnected_
-std::vector<std::function<void()>*> subscribers_
+void OnTangoServiceConnected(JNIEnv *env, jobject binder)
+void onPointCloudAvailable(const TangoPointCloud *pointCloud)
+void onFrameAvailable(const TangoPointCloud *pointCloud)
+void TangoDisconnect()
+void Subscribe(std::function<void()> *subscriber)
+void Unsubscribe(std::function<void()> *subscriber)
+TangoSupportPointCloudManager *GetPointCloudManager()
+TangoSupportImageBuffer *GetTangoSupportImageBufferManager()
-void TangoSetupConfig()
-void TangoInitializeTangoManager()
-void TangoConnectCallbacks()
-void TangoConnect()
-void CheckError()
VirtualWhiteCaneApp
-GroundDetectionWorkerThread groundDetectionWorkerThread_
-WarningWorkerThread warningWorkerThread_
-Scene mainScene_
-std::atomic<bool> isGlInitialized_
-rgb_depth_sync::ColorImage colorImage_
-rgb_depth_sync::DepthImage depthImage_
-rgb_depth_sync::DepthImage depthImageFloor_
-TangoSupportRotation colorCameraToDisplayRotation_
-std::vector<float> vertices_
+void OnCreate(JNIEnv *env, jobject callerActivity, jobject assetManager)
+void OnPause()
+void OnDestroy()
+void OnSurfaceCreated()
+void OnSurfaceChanged(int width, int height)
+void OnDrawFrame()
+void OnDisplayChanged(int displayRotation, int colorCameraRotation)
-void DeleteResources()

Figure 4.15: UML of the `virtualWhiteCane` library. The `virtualWhiteCaneApp` starts the `GroundDetectionWorkerThread` and the `WarningWorkerThread`. The `TangoHandler` handles getting and distributing the point cloud.

4.3.3 groundPlaneDetection Library

This library provides several useful methods employed for ground detection. The most important class is the `GroundPlaneExtractor` class. It provides methods to estimate a ground plane using RANSAC from the point cloud library, as well as calculating a v disparity map using OpenCL. The layout of this library can be seen in Figure 4.16 as an UML diagram.

Further, it wraps additional filter operations from the point cloud library in the `PointCloudFilter` class, which allows to easily produce a voxel grid or perform statistical outlier removal. However, as the voxel grid removes a lot of input points and the statistical outlier removal is too slow on the full input cloud, this functionality is not used.

The `SurfaceNormalEstimator` also wraps two methods of surface normal estimation for easier use. It allows to estimate the surface normals using a tree structure (for example KdTree, Octree, etc.) or, if the data is organized, it is possible to use integral images. Again, building a KdTree is very slow on a full point cloud and therefore surface normal estimation is not used.

GroundPlaneExtractor

The `GroundPlaneExtractor` implements two ways to detect a ground plane using either RANSAC or a v disparity map. The RANSAC implementation is provided by the PCL and the v disparity map algorithm is implemented in parallel using OpenCL.

The RANSAC implementation has some additional filters to ensure a better fit. First, it allows to filter the point cloud data points according to their Y value in space. This can eliminate planes, which are too high, like a table. Further, it allows to define the `distanceThreshold`, which defines whether a point is considered to be a inlier or not. Lastly, it returns the coefficients with the largest amount of inliers, if the resulting normal has a dominant y component, which means the y component has to be larger than the x and z component of the normal vector, and if they are between 0.8 m and 1.3 m below the phone. Checking for a dominant y component ensures that the plane coefficients coincide with the floor and not a wall. However, the plane has to contain at least 30% of all input points.

The method `CalcVDisparity` calculates the v disparity of an input depth map in parallel using OpenCL. For implementation details about the v disparity map calculation in OpenCL see Section 4.4.1. The `GroundPlaneExtractor` also provides a method to filter a depth map according to a disparity mask. Filtering is also implemented using OpenCL and details can also be found in the Section 4.4.1.

PointCloudFilter

The `PointCloudFilter` class provides easier usage of two filter methods implemented in the Point Cloud Library. Applying a voxel grid filter or statistical outlier removal to a point cloud can be done using a single call of the appropriate method.

The `VoxelGridFilter` takes a point cloud as an input parameter and applies voxel grid filtering. A voxel grid can be imagined as a set of 3-d boxes in space. All points contained in a voxel are approximated by a single new point. This point is located at the center of mass of the original points. Therefore, the point cloud is downsampled and greatly reduced in size.

Statistical outlier removal does a statistical analysis of the neighborhood of each point and removes them, if they can be considered to be noise. As the implementation internally creates a KdTree to define the neighborhood of each point; this process is rather slow on large point clouds and it can only be applied in real-time on a downsampled point cloud.

SurfaceNormalEstimator

For easier usage of surface normal estimation in the Virtual White Cane application, the `SurfaceNormalEstimator` class was created. It wraps two algorithms for surface normal estimation implemented by the Point Cloud Library into single templated function calls.

The `EstimateSurfaceNormalsUsingTree` method allows to easily interchange the used tree structure to estimate the surface normals. This allows to benefit from the `OrganizedNeighbor` structure from PCL, in case of organized point clouds. `EstimateSurfaceNormalsUsingIntegralImages` uses the algorithm described in [49].

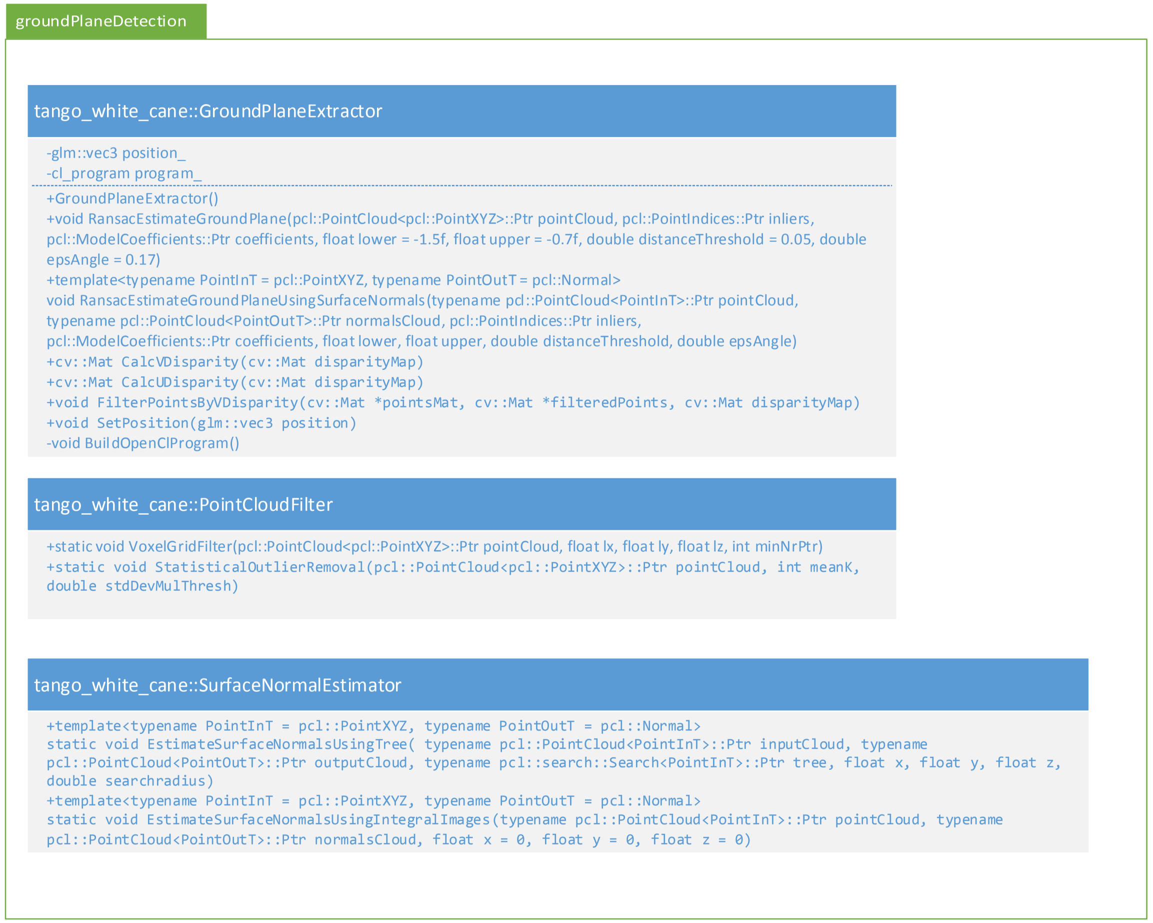

groundPlaneDetection
tango_white_cane::GroundPlaneExtractor
-glm::vec3 position_
-cl_program program_
+GroundPlaneExtractor()
+void RansacEstimateGroundPlane(pcl::PointCloud<pcl::PointXYZ>::Ptr pointCloud, pcl::PointIndices::Ptr inliers, pcl::ModelCoefficients::Ptr coefficients, float lower = -1.5f, float upper = -0.7f, double distanceThreshold = 0.05, double epsAngle = 0.17)
+template<typename PointInT = pcl::PointXYZ, typename PointOutT = pcl::Normal> void RansacEstimateGroundPlaneUsingSurfaceNormals(typename pcl::PointCloud<PointInT>::Ptr pointCloud, typename pcl::PointCloud<PointOutT>::Ptr normalsCloud, pcl::PointIndices::Ptr inliers, pcl::ModelCoefficients::Ptr coefficients, float lower, float upper, double distanceThreshold, double epsAngle)
+cv::Mat CalcVDisparity(cv::Mat disparityMap)
+cv::Mat CalcUDisparity(cv::Mat disparityMap)
+void FilterPointsByVDisparity(cv::Mat *pointsMat, cv::Mat *filteredPoints, cv::Mat disparityMap)
+void SetPosition(glm::vec3 position)
-void BuildOpenClProgram()
tango_white_cane::PointCloudFilter
+static void VoxelGridFilter(pcl::PointCloud<pcl::PointXYZ>::Ptr pointCloud, float lx, float ly, float lz, int minNrPtr)
+static void StatisticalOutlierRemoval(pcl::PointCloud<pcl::PointXYZ>::Ptr pointCloud, int meanK, double stdDevMulThresh)
tango_white_cane::SurfaceNormalEstimator
+template<typename PointInT = pcl::PointXYZ, typename PointOutT = pcl::Normal> static void EstimateSurfaceNormalsUsingTree(typename pcl::PointCloud<PointInT>::Ptr inputCloud, typename pcl::PointCloud<PointOutT>::Ptr outputCloud, typename pcl::search::Search<PointInT>::Ptr tree, float x, float y, float z, double searchradius)
+template<typename PointInT = pcl::PointXYZ, typename PointOutT = pcl::Normal> static void EstimateSurfaceNormalsUsingIntegralImages(typename pcl::PointCloud<PointInT>::Ptr pointCloud, typename pcl::PointCloud<PointOutT>::Ptr normalsCloud, float x = 0, float y = 0, float z = 0)

Figure 4.16: UML of the groundPlaneDetection library.

4.3.4 virtualWhiteCaneUtil Library

Some functionality is frequently needed across the whole application. These are implemented in the `virtualWhiteCaneUtil` library. The utility classes are composed within its own library such that they can be used in other projects as well. Figure 4.17 shows an UML diagram of the `virtualWhiteCaneUtil` library.

It is composed of two classes, which hold methods in need of further data and some static functions, which can be called directly. The `JNIHelper` is responsible for various JNI communications. For JNI functionality to work, members, such as the Java VM, are needed.

The second class `OpenClHelper` wraps OpenCL methods with frequently used parameters. It also handles globally identical members, such as the OpenCL context. Each method checks for an OpenCL error and reports it if one occurs.

JNIHelper

The Android implementation of JNI only supports a single Java VM, but it does not provide the static methods to retrieve it when needed in native code. Therefore, it is necessary to save the Java VM as a member. This is done during the `JNI_OnLoad` call, which sets the VM of the default `JNIHelper`.

The `JNIHelper` class initializes a default object, which can be accessed globally. This allows to create additional instances of the `JNIHelper`, if they are needed for some reason. For example, even though Android currently only supports a single Java VM, if this changes for some reason, a second `JNIHelper` object could be created, which uses the second VM.

Further, to allow access to Android assets the `JNIHelper` keeps a reference to the asset manager handle. This has to be done during start up and is provided via JNI from the Java layer.

The `JNIHelper` also allows to attach or detach a thread to the Java VM, which is crucial if a new thread wants to call a function in the Java layer from the C++ layer. Calling or registering functions, which return void, is also wrapped into a single call, respectively. To be able to call a function, it is necessary to register the caller activity and keep a global reference to it, which is kept in a map. A helper function, which takes the key to the caller activity, the method name, and signature, and if needed any arguments, which should be passed to the method, is implemented to allow easy access to methods in the Java layer.

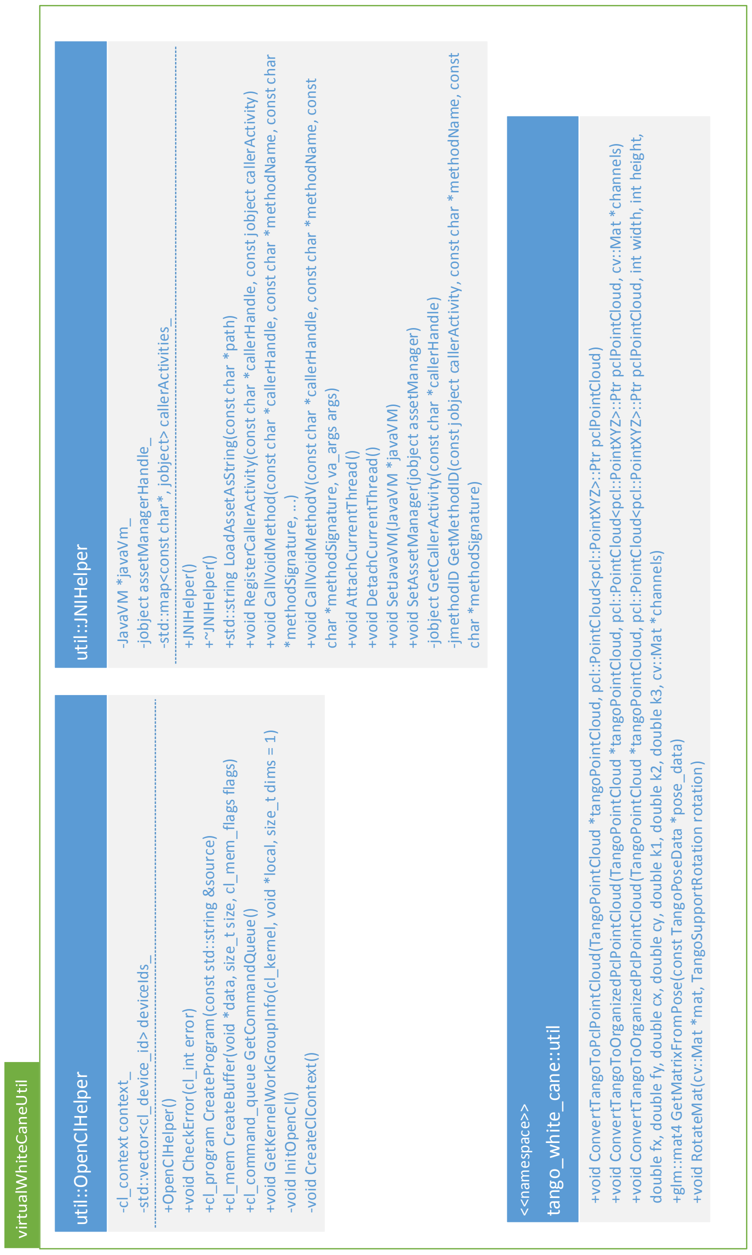

virtualWhiteCaneUtil
util::OpenClHelper
-cl_context context_
-std::vector<cl_device_id> deviceIds_
+OpenClHelper()
+void CheckError(cl_int error)
+cl_program CreateProgram(const std::string &source)
+cl_mem CreateBuffer(void *data, size_t size, cl_mem_flags flags)
+cl_command_queue GetCommandQueue()
+void GetKernelWorkGroupInfo(cl_kernel, void *local, size_t dims = 1)
-void InitOpenCl()
-void CreateClContext()
util::JNIHelper
-JavaVM *javaVm_
-jobject assetManagerHandle_
-std::map<const char*, jobject> callerActivities_
+JNIHelper()
+~JNIHelper()
+std::string LoadAssetAsString(const char *path)
+void RegisterCallerActivity(const char *callerHandle, const jobject callerActivity)
+void CallVoidMethod(const char *callerHandle, const char *methodName, const char *methodSignature, ...)
+void CallVoidMethodV(const char *callerHandle, const char *methodName, const char *methodSignature, va_args args)
+void AttachCurrentThread()
+void DetachCurrentThread()
+void SetJavaVM(JavaVM *javaVM)
+void SetAssetManager(jobject assetManager)
-jobject GetCallerActivity(const char *callerHandle)
-jmethodID GetMethodID(const jobject callerActivity, const char *methodName, const char *methodSignature)
<<namespace>>
tango_white_cane::util
+void ConvertTangoToPclPointCloud(TangoPointCloud *tangoPointCloud, pcl::PointCloud<pcl::PointXYZ>::Ptr pclPointCloud)
+void ConvertTangoToOrganizedPclPointCloud(TangoPointCloud *tangoPointCloud, pcl::PointCloud<pcl::PointXYZ>::Ptr pclPointCloud, cv::Mat *channels)
+void ConvertTangoToOrganizedPclPointCloud(TangoPointCloud *tangoPointCloud, pcl::PointCloud<pcl::PointXYZ>::Ptr pclPointCloud, int width, int height, double fx, double fy, double cx, double cy, double k1, double k2, double k3, cv::Mat *channels)
+glm::mat4 GetMatrixFromPose(const TangoPoseData *pose_data)
+void RotateMat(cv::Mat *mat, TangoSupportRotation rotation)

Figure 4.17: UML of the `virtualWhiteCaneUtil` library.

OpenClHelper

With the `OpenClHelper` class, common OpenCL procedures are combined into simple methods. The OpenCL context is created and configured during initialization of the object, as this often involves the same steps and parameters.

Creating and building a OpenCL program often involves the same steps and, therefore, a helper function exists. This helper function creates the program from the given source and builds it. It also checks if errors occurred and reports them, if necessary.

`CreateBuffer` is useful to create a buffer in OpenCL memory. A buffer is created for the context created during initialization. The helper function also checks and reports for errors occurring during buffer creation.

Lastly, a helper function, which returns a command queue exists. This is useful, as command queues are often created with the same parameters and the helper function also checks for errors.

util

Finally, the `VirtualWhiteCaneUtil` library has a namespace to provide some helper functions, which do not need to keep any references. Their range of tasks include converting Google Tango point cloud data into other formats and likewise creating transformation matrices from various input data formats.

To be able to use PCL functions, it is necessary to convert the point cloud data from Google Tango's data format to a PCL format. For this reason, three slightly different helper functions exist. `ConvertTangoToPclPointCloud` simply converts an unorganized point cloud into an unorganized point cloud in PCL. `ConvertTangoToOrganizedPclPointCloud` converts the point cloud into an organized point cloud. There are two different versions of this method, one takes additional parameters and one does not. The method without additional parameters receives them using the Google Tango SDK. The following parameters are needed:

width	the width of the organized cloud.
height	the height of the organized cloud.
f_x, f_y	the focal length of the camera, x and y axis, in pixels.

c_x, c_y	the central point, x and y axis, in pixels.
k_1, k_2, k_3	the distortion coefficients.
channels	this parameter can be a null pointer, otherwise it has to be an array of 4 `cv::Mat` matrices. Each component x, y, z of a 3-d point is stored separately in one channel. The fourth channel stores the confidence value given by Google's Tango.

Further, as with physical rotation of the device the orientation of the depth image no longer coincides with the OpenGL world, there is a helper function, which rotates a depth map according to the rotation of the device. `RotateMat` takes a `cv::Mat*` and `TangoSupportRotation` as input and rotates the matrix, such that the depth maps axis coincides with the the OpenGL world.

4.3.5 warningModule Library

This library handles the object detection and object warning. Figure 4.18 shows an UML diagram of the library. It defines two interfaces, which allow to change the underlying algorithm easily. The interfaces are described in more detail at Section 3.4.3.

The realization of the `ObjectDetector` interface is the `ConservativePolarHisto-gram`. It calculates a Conservative Polar Histogram and outputs a list of coordinates representing obstacles. Detailed information about the implementation in OpenCL can be found in Section 4.4.2.

For the `AcousticWarningModule` the Google VR audio API is used. It provides a powerful audio engine allowing to produce accurate 3-d sound. To be able to make use of the Google VR audio API, a pointer to the `gvr::AudioApi` has to be supplied. Access to the `gvr::AudioApi` allows to load sound files and place them in 3-d space. The Google VR audio API further implements a HRTF, which allows to create an authentic spatial sound. In addition to the audio cues, once an object is within 1 m and in front of the device, the vibrator is activated. To determine which bin corresponds to the front of the device, the pose of the smart phone needs to be determined. This is achieved by using the Google Tango SDK by getting the newest transformation matrix and eliminating the translational component. The vibrator is activated using a JNI call via the `JNIHelper`.

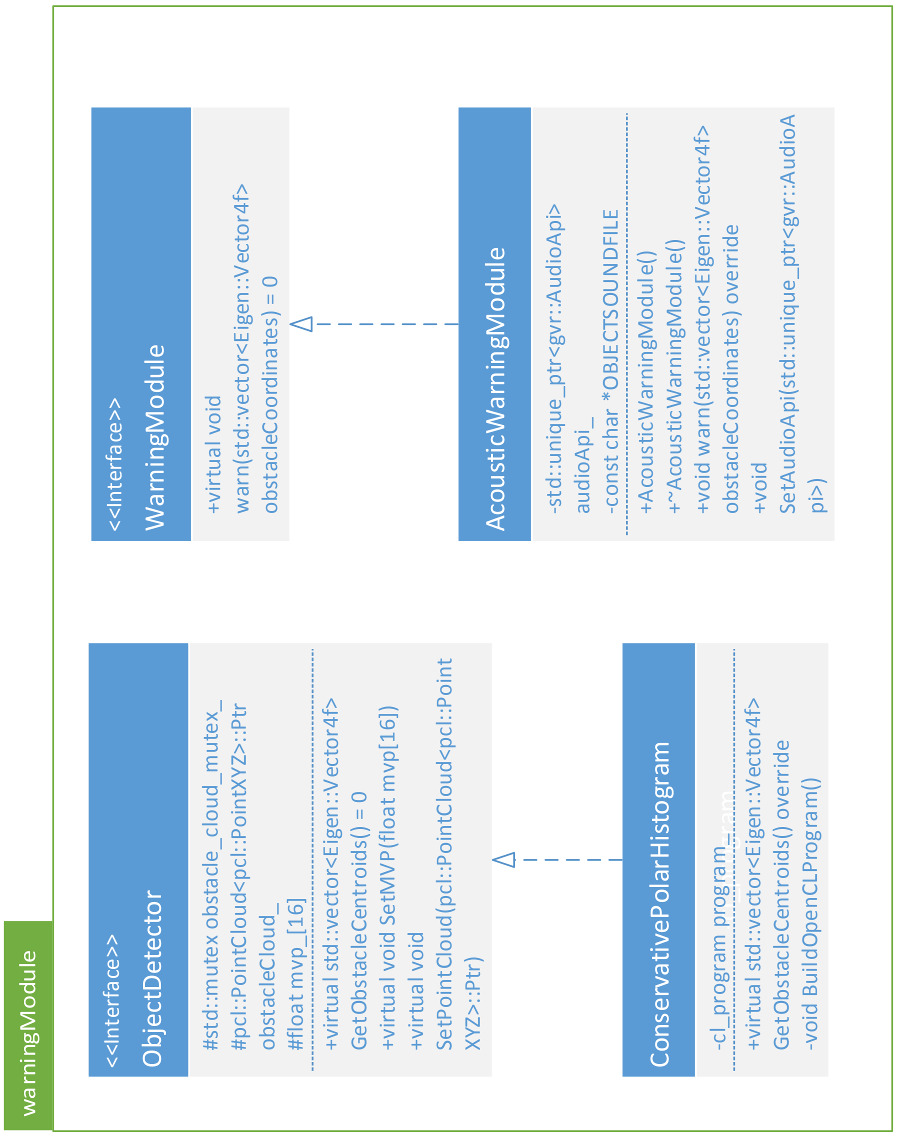

warningModule
<<Interface>>
ObjectDetector
#std::mutex obstacle_cloud_mutex_
#pcl::PointCloud<pcl::PointXYZ>::Ptr obstacleCloud_
#float mvp_[16]
+virtual std::vector<Eigen::Vector4f> GetObstacleCentroids() = 0
+virtual void SetMVP(float mvp[16])
+virtual void SetPointCloud(pcl::PointCloud<pcl::PointXYZ>::Ptr)
ConservativePolarHistogram
-cl_program program_
+virtual std::vector<Eigen::Vector4f> GetObstacleCentroids() override
-void BuildOpenCLProgram()
<<Interface>>
WarningModule
+virtual void warn(std::vector<Eigen::Vector4f> obstacleCoordinates) = 0
AcousticWarningModule
-std::unique_ptr<gvr::AudioApi> audioApi_
-const char *OBJECTSOUNDFILE
+AcousticWarningModule()
+~AcousticWarningModule()
+void warn(std::vector<Eigen::Vector4f> obstacleCoordinates) override
+void SetAudioApi(std::unique_ptr<gvr::AudioApi>)

Figure 4.18: UML of the `warningModule` library.

4.4 Algorithm Implementation Details

This section describes how the used algorithms were implemented and what had to be considered to be able to run them on the Lenovo Phab 2 Pro. It focuses on the problems arising with the very limited memory available for OpenCL implementations on mobile devices and how they were solved.

4.4.1 V Disparity in OpenCL

First, the point cloud has to be transformed according to Equation 4.1. This gives a 2-d image, where the pixel location (x, y) represents the 3-d X, Y coordinates and the intensity of the pixel represents the depth Z. As the ToF camera does not use disparity to calculate the depth of the pixel, a pseudo-disparity has to be introduced. As shown in [25], the disparity d can be calculated using

$$d = f * \frac{B}{z}. \tag{4.4}$$

With f being the focal length and B the assumed baseline. Due to the fact that Google Tango does not yet implement an auto exposure algorithm for the ToF camera and, therefore, objects, which are too close will result in overexposure and no valid depth readings, a minimum distance of 0.2 m was assumed. Hence, $f * B = 0.2$ was chosen, giving a disparity of $d = 1$ for $z = 0.2\,\text{m}$. Any object farther away will therefore have a smaller disparity. The disparity obtained in this way is discretized into 256 values.

The v disparity map is computed in parallel using OpenCL. For easier processing each work-group is supposed to handle a row of pixels. Therefore, *depthimage.rows* work-groups are needed and each work-group has *depthimage.cols* work-items. However, due to the fact that each work-item writes one value of the v disparity map as output, at least 256 work items are needed per work-group. If *depthimage.cols* happens to be lower than 256, the depth map has to be padded to 256 with *NaN* values. As the ToF camera used in the Lenovo Phab 2 Pro has a resolution of 172×224 pixels, the columns are padded to 256. Note that depending on the orientation of the phone, the image size might be transposed.

Accumulating the amount of pixels for each disparity value has to be synchronized, as otherwise race conditions would occur. Thus, atomic operations are needed to prevent these and ensure valid increments. Unfortunately, using global atomic operations is very costly and slow. As local atomic operations are much faster, each work-group has to operate on exactly one line of the depth map and use a local array to hold the respective v disparity line. Figure 4.19 shows how the OpenCL kernel operates. The input depth image is accessed as a 1-d array in the following way:

$$\begin{aligned} tid &= get\_local\_id(0), \\ group\_idx &= get\_group\_id(0) * 256, \\ idx(x,y) &= tid + group\_index. \end{aligned} \tag{4.5}$$

Listing 4.2 shows the implementation of the v disparity kernel. For this to work, there needs to be a local size of at least 256, which means each work-group holds 256 work-items and *depthimage.rows* work-groups are created.

Filtering a depth map by disparity values can be achieved in parallel as well. The kernel needs the depth map and a mask structured like the v disparity map. For each input point the disparity is calculated in the same way as for the v disparity map (see Equation 4.4). After the disparity value is discretized to a range from 0 to 255, this discretized value and the line number is used as an index for the disparity mask. If the value of the mask is 1, the point has to be filtered. Obviously, the input again has to be an ordered depth map, which tells the kernel which line of the v disparity mask to look at. As each work item only works and writes to one pixel location and they do not interfere with each other, a direct access pattern is given and no synchronization is needed.

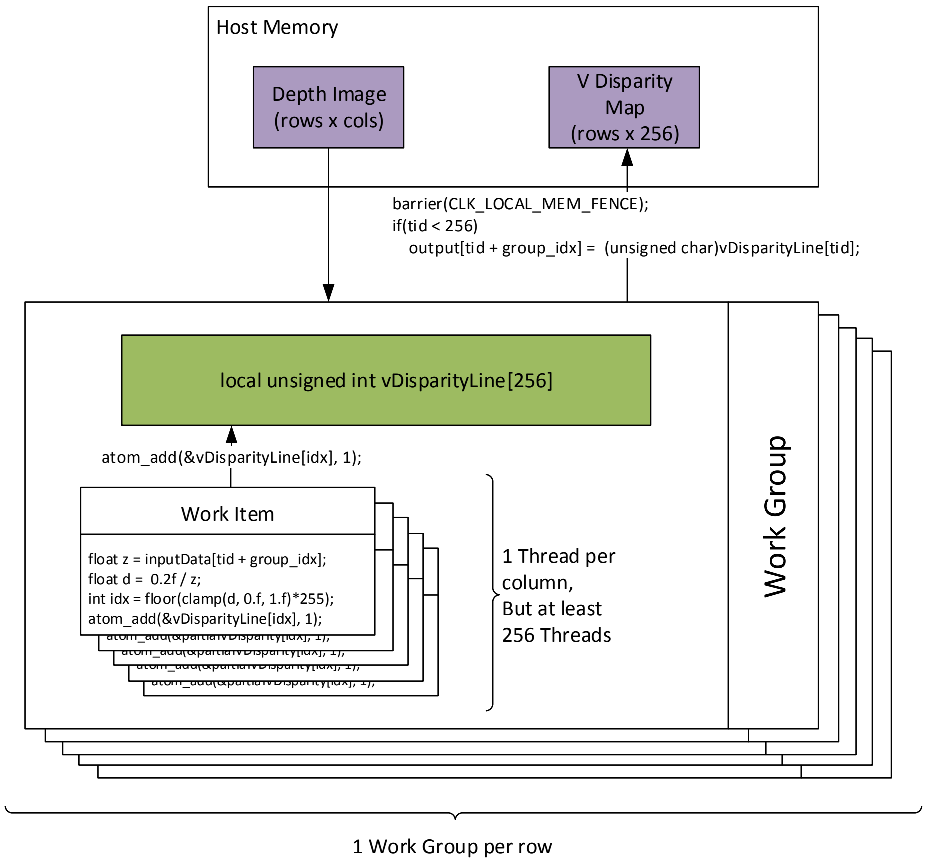
For this kernel to work properly, it has to be called with a certain configuration. Globally, there are $rows \times cols$ work-items created using a 2-d kernel. Therefore, the global IDs are also 2-d allowing direct access to the (u, v) coordinates of the depth map. Within each work-group, however, work-items are only accessed in a 1-d pattern and each work-group has *#cols* work-items. This results in *#rows* work-groups. In other words each work-group represents a row within the depth map allowing to easily access the correct row in the disparity mask. Listing 4.3 shows the implementation of this kernel.

4.4.2 Conservative Polar Histogram

In order to find the nearest obstacle within the point cloud once the floor points are removed, a Conservative Polar Histogram is developed. To calculate the Conservative Polar Histogram, the points are projected onto the x-z plane in the translated

```
__kernel void VDisparity(__global float*          inputData,
                                  int             width,
                                  int             height,
                         __global unsigned char*  output) {

    int tid = get_local_id(0);
    int group_idx = get_group_id(0) * 256;

    local unsigned int vDisparityLine[256];

    //initialize partialVDisparity
    if(tid < 256) {
        vDisparityLine[tid] = 0;
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    //only consider one row and restrict to depth readings >= 0.2m
    if(tid < width && !isnan(inputData[tid + group_idx]) && inputData[
   tid + group_idx] >= 0.2){

        //z should always positive because of depth camera coordinate
   system
        float z = inputData[tid + group_idx];

        //disparity = f * T / z
        float d = 0.2f / z;

        int idx = floor(clamp(d, 0.f, 1.f)*255);

        atom_add(&vDisparityLine[idx], 1);
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    if(tid < 256) {
        output[tid + group_idx] = (unsigned char)vDisparityLine[tid];
    }
}
```

Listing 4.2: Kernel to calculate the v disparity.


Host Memory
Depth Image (rows x cols)
V Disparity Map (rows x 256)
local unsigned int vDisparityLine[256]
Work Item
1 Thread per column, But at least 256 Threads
Work Group
1 Work Group per row

```
barrier(CLK_LOCAL_MEM_FENCE);
if(tid < 256)
   output[tid + group_idx] =  (unsigned char)vDisparityLine[tid];
```

```
atom_add(&vDisparityLine[idx], 1);
```

```
float z = inputData[tid + group_idx];
float d =  0.2f / z;
int idx = floor(clamp(d, 0.f, 1.f)*255);
atom_add(&vDisparityLine[idx], 1);
```

```
atom_add(&partialvDisparity[idx], 1);
atom_add(&partialvDisparity[idx], 1);
atom_add(&partialvDisparity[idx], 1);
atom_add(&partialvDisparity[idx], 1);
```

Figure 4.19: The v disparity kernel starts one thread per pixel. Each work group has at least 256 and at most *depthimage.cols* work items. Each work item calculates the disparity and adds it to a local array representing the v disparity line. As soon as all work items added their contribution to the local map, it is copied to the global v disparity map. The green block represents the local memory block and the lavender represents the host input and output memory.

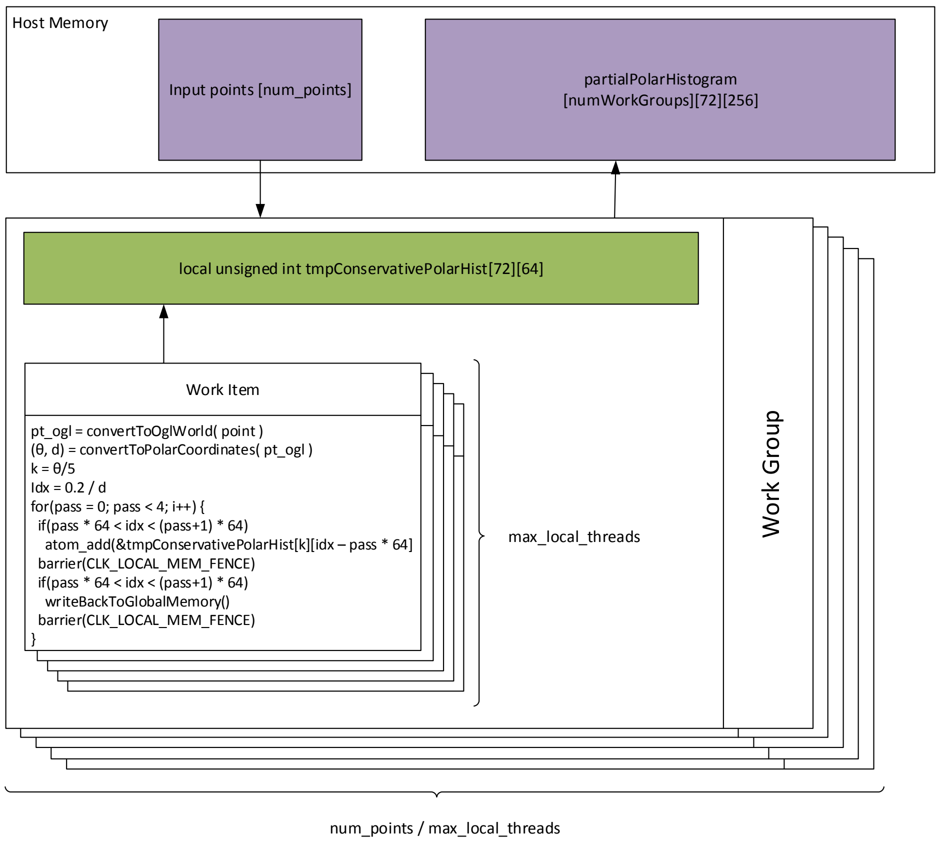
```
__kernel void FilterByDisparity(__constant float* inputData, __global float* filtered, __constant unsigned char* disparityMask, int width, int height) {
  //u,v coordinates of depth map/inputData
    int gid_x = get_global_id(0); // u
    int gid_y = get_global_id(1); // v
    int local_size = get_local_size(0) * get_local_size(1); //#cols * 1
    //group_idx to access correct row in disparityMask
    int group_idx = get_group_id(0) * 256;

    if(gid_x > width || gid_y > height) {
        return;
    }
  //each point has 3 coordinates (X,Y,Z)
    float depth = inputData[gid_y*local_size*3 + (3*gid_x) + 2];

    float d = 0.2f / depth;

    int idx = floor(clamp(d, 0.f, 1.f)*255);

  //if the disparityMask for the given index > 0, copy point to output
    if(disparityMask[gid_y*256 + idx]) {
        filtered[gid_y*local_size*3 + (3*gid_x) + 0] = inputData[gid_y*local_size*3 + (3*gid_x) + 0];
        filtered[gid_y*local_size*3 + (3*gid_x) + 1] = inputData[gid_y*local_size*3 + (3*gid_x) + 1];
        filtered[gid_y*local_size*3 + (3*gid_x) + 2] = inputData[gid_y*local_size*3 + (3*gid_x) + 2];
    }
}
```

Listing 4.3: Kernel to filter by v disparity.

OpenGL world frame and transformed into their polar coordinates. The projection onto the x-z plane is done by simply omitting the y coordinate of the point. The angular coordinates are sectioned into bins á 5 degrees, giving 72 bins in total. The radial coordinate is transformed into a discrete disparity value between 0 and 255. For each bin a disparity histogram is formed by accumulating the amount of points in that bin with the same disparity. These disparity histograms for each bin can now be used to easily find the closest obstacle by applying a threshold to the histogram.

The Conservative Polar Histogram algorithm is implemented using OpenCL. Figure 4.20 shows the kernel layout. First, the point has to be transformed into the OpenGL coordinate system by a simple matrix multiplication. Afterwards, the polar angle of the point is calculated, as well as the bin it belongs to. The distance is obtained by calculating the dot product of the projected point with itself. Once the distance and the polar angle are calculated, the histogram can be obtained.

As the Lenovo Phab 2 Pro has a local device memory limit of 32 kB and a full Conservative Polar Histogram with 72 bins would need $72 * 256 * sizeof(uint) =$ 72 kB, each work item has to do 4 passes. Thus, only a local array of size $72 * 64 * sizeof(uint) = 18kB$ is allocated. To still get a full Conservative Polar Histogram, each quarter of the disparity histogram is written sequentially. Again, as with the v disparity map, the access to the local memory has to be synchronized. Listing 4.4 is a shortened version of the kernel. Some code is shortened or omitted for brevity. First, the local memory is allocated, afterwards, the discrete disparity value idx and the polar coordinate bin k are determined. After each quarter pass, the memory is synchronized and written to the output array. As it is not possible for work groups to access another work group's local memory, each work group only has a partial Conservative Polar Histogram. Therefore, $\#workgroups$ Conservative Polar Histograms are created and need to be combined by another kernel. Listing 4.5 shows how the partial Conservative Polar Histograms are summed up to one Conservative Polar Histogram. Each thread locally sums up one value of the partial Conservative Polar Histogram.


Host Memory
Input points [num_points]
partialPolarHistogram [numWorkGroups][72][256]
local unsigned int tmpConservativePolarHist[72][64]
Work Item
max_local_threads
Work Group
num_points / max_local_threads

```
pt_ogl = convertToOglWorld( point )
(θ, d) = convertToPolarCoordinates( pt_ogl )
k = θ/5
Idx = 0.2 / d
for(pass = 0; pass < 4; i++) {
  if(pass * 64 < idx < (pass+1) * 64)
    atom_add(&tmpConservativePolarHist[k][idx – pass * 64]
  barrier(CLK_LOCAL_MEM_FENCE)
  if(pass * 64 < idx < (pass+1) * 64)
    writeBackToGlobalMemory()
  barrier(CLK_LOCAL_MEM_FENCE)
}
```

Figure 4.20: Conservative Polar Histogram Kernel Layout.

```
__kernel void PartialConservativePolarHistogramKernel(__constant float4
    *inputPoints, float16 MVP, long numPts, __global unsigned int *
    outputPartialPolarHistogram) {

  int gid = get_global_id(0);
  int tid = get_local_id(0);
  int group_idx = get_group_id(0);

  //72x256 is too much memory, do it in 4 passes
  __local unsigned int tmpConservativePolarHist[72][64];

  //transform to ogl. mvp is in column major order
  float4 pt_ogl =  transformPoint(MVP, inputPoints[gid]);

  // x/z gives the angle to the wrong axis -> z/x
  float beta = atan2pi(pt_ogl.z, pt_ogl.x) * 180.f;
  int k = floor(beta/5);
  float2 pt_ogl2 = (float2)(pt_ogl.x, pt_ogl.z);
  float distance = sqrt(dot(pt_ogl2, pt_ogl2));
  float disparity = 0.2f / distance;
  int idx = floor(clamp(disparity, 0.f, 1.f)*255);

  for(int c = 0; c < 4; ++c) {
    if((64*c) < idx < (64*(c+1))) {
      atom_add(&tmpConservativePolarHist[k][idx], 1);
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    if((64*(c)) <= tid && tid < (64*(c+1))) {
      #pragma unroll
      for(int i = 0; i < 72; ++i) {
        //set output
        outputPartialPolarHistogram[i * 256 + tid + group_idx * 72 *
    256] = tmpConservativePolarHist[i][tid - (64*c)];
        //reset local for next 1/4
        tmpConservativePolarHist[i][tid - (64*c)] = 0;
      }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

Listing 4.4: Kernel to calculate the partial ConservativePolarHistogram. Some code is omitted or shortened for brevity.

```
__kernel void SumPartialConservativePolarHistogram(__constant unsigned
    int *partialPolarHistogram, int numGroups, __global unsigned int *
    polarHistogram) {

//gid 0...72*256
  int gid = get_global_id(0);
  int tid = get_local_id(0);
  int local_size = get_local_size(0);
  int group_idx = get_group_id(0);

  unsigned int sum = 0;

  for(int i = 0; i < numGroups; ++i) {
    sum += partialPolarHistogram[i * 72 * 256 + gid];
  }

  polarHistogram[gid] = sum;

}
```

Listing 4.5: Kernel to sum the partial ConservativePolarHistogram.

Chapter 5

Results

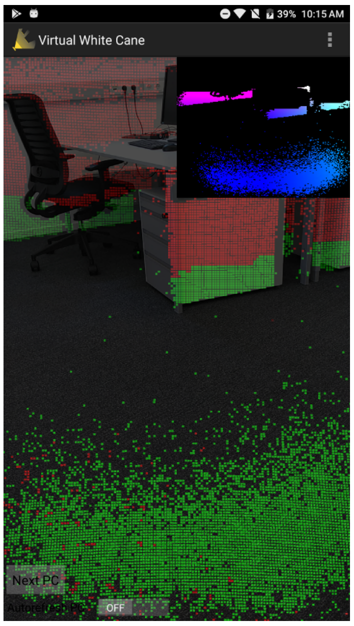
This chapter will present the results of this thesis. It will show examples where the used algorithms would fail on their own and how the combination of them improved the result.

First, the results of the ground floor detection algorithm are presented and discussed. We will show how the used algorithm improves the ground floor detection. Then the results of the Conservative Polar Histogram are shown.

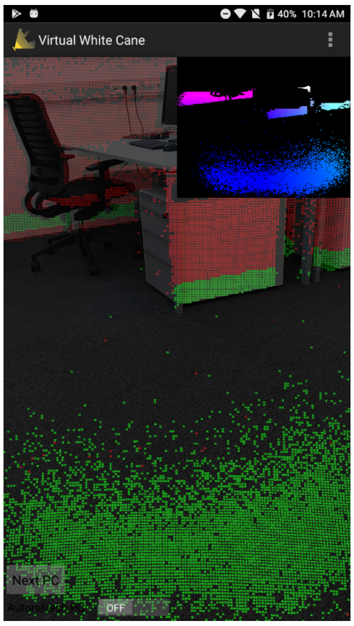
5.1 Ground Floor Detection

This section discusses how the algorithm improves the ground floor detection. The color coding of the pixels shows their classification after the ground detection process. Green pixels indicate floor pixels, while red pixels indicate obstacles. Pixels, which are not artificially colored, do not have a valid depth reading due to the ToF sensor's limitations and the lack of infra-red light reflected to the sensor.

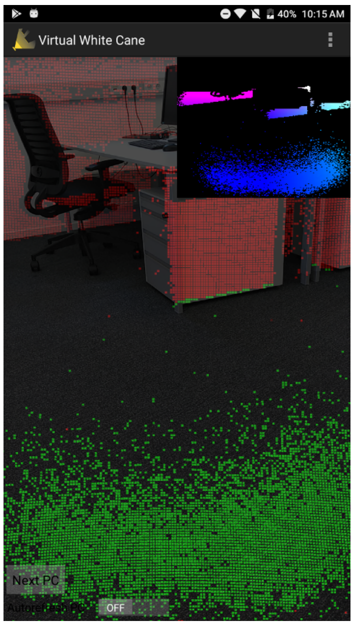
The combination of the v disparity and RANSAC algorithm helps to improve the accuracy of the RANSAC algorithm. Due to its random nature and the fitting process, the best fit might not always reflect the real geometry of the data. Figure 5.1b shows how the RANSAC algorithm found a better fit for a plane with a slight slope, because there are hardly any valid depth readings on the floor beyond a certain distance and the high noise ratio given by the carpeted floor. A similar problem occurs with the v disparity algorithm, as seen in Figure 5.1a. The line found by the Hough transformation also covers the pixels corresponding to the intersection of the obstacles and the floor. However, the combination of both provides a better fit, as seen in Figure 5.1c.



(a) v disparity.



(b) RANSAC.



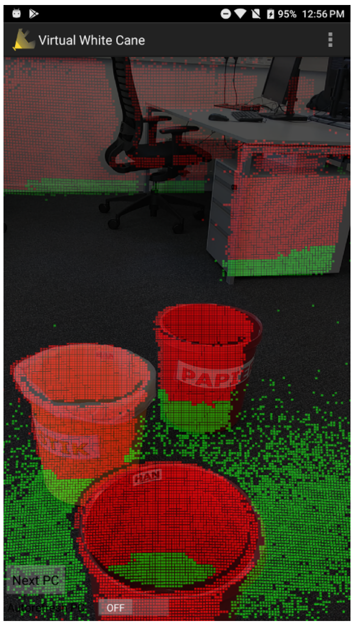
(c) Our method.

Figure 5.1: Image (a) shows how the floor is segmented just using v disparity, while (b) shows segmentation just using RANSAC, and (c) shows our method.

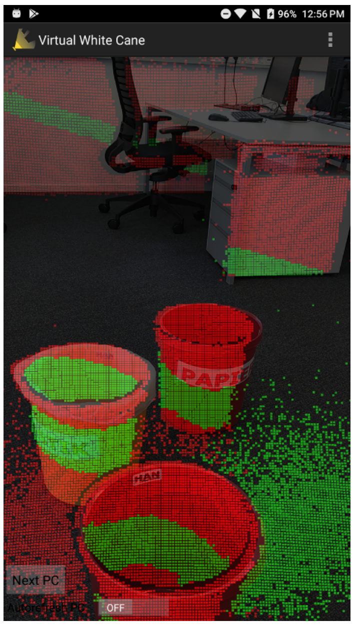
Our method also overcomes some problems in cluttered scenes. V disparity segmentation often classifies too many points as floor, while RANSAC is unable to find a properly fitting plane. However, due to the reduced number of input points, which already mostly belong to the floor, it performs well in our method. Figure 5.2 shows an example of a cluttered scene and how our method outperforms the algorithms on their own.

A similar situation arises with a large object covering most of the floor. Figure 5.3 shows how the RANSAC algorithm fails to classify the correct floor plane, while the v disparity algorithm already has a good classification. The combination of both again gives a slight improvement.

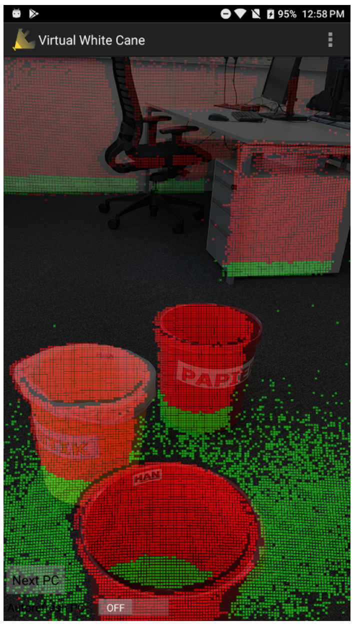
When the device is held at any angle, such that the bottom line of the phone is not parallel to the ground, the line representing the ground floor in the v disparity image oscillates. This results in a misclassification of the ground floor in the v disparity domain. This can be seen in Figure 5.4. RANSAC on its own is able to detect the ground plane fairly well. It also is capable of correcting the problems with the pure v disparity algorithm, as can be seen in Figure 5.4c.



(a) v disparity.

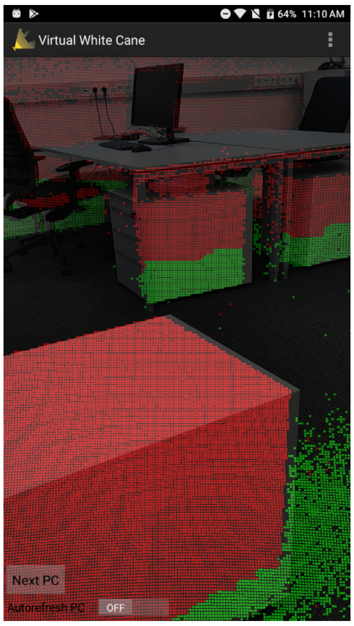


(b) RANSAC.

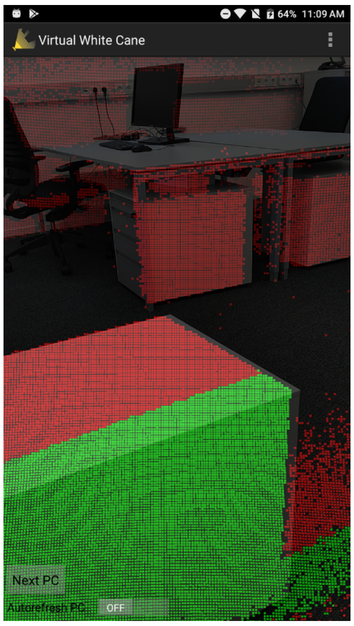


(c) Our method.

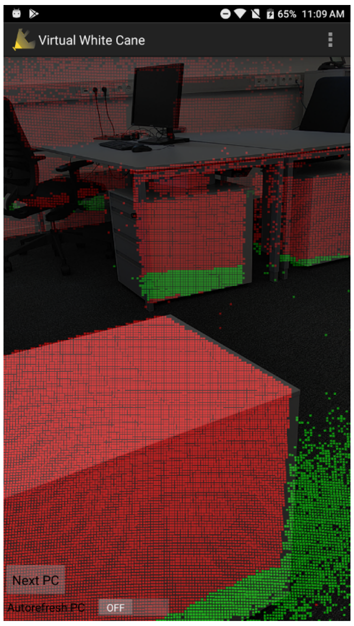
Figure 5.2: Our method outperforms in cluttered scenes.



(a) v disparity.

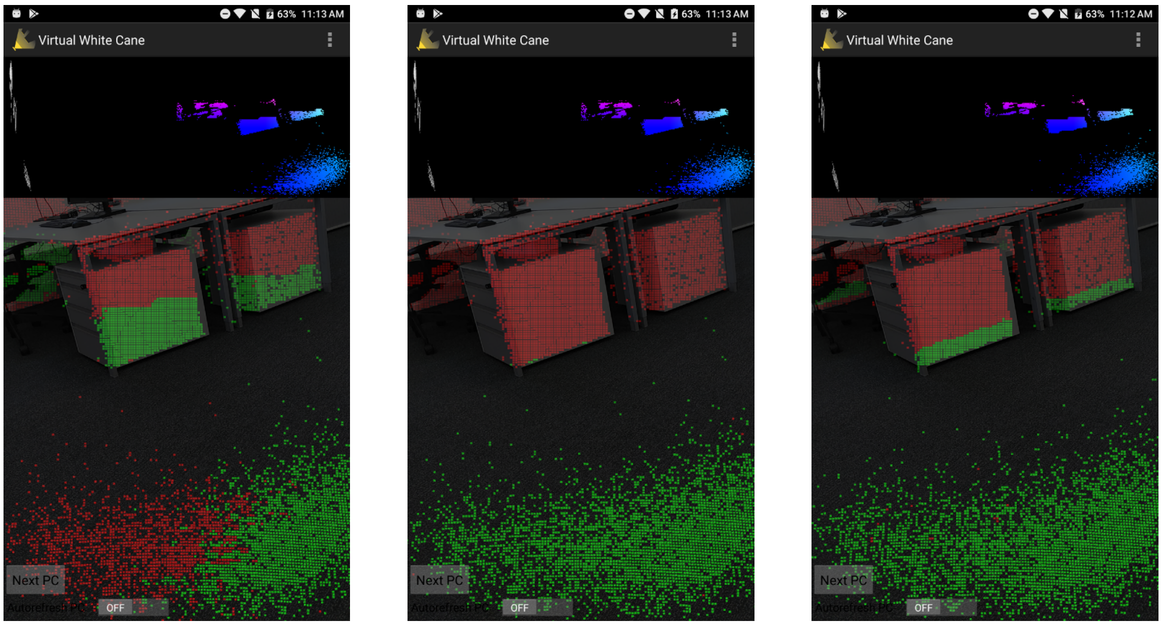


(b) RANSAC.



(c) Our method.

Figure 5.3: Again our method outperforms with a large object in front of the user.

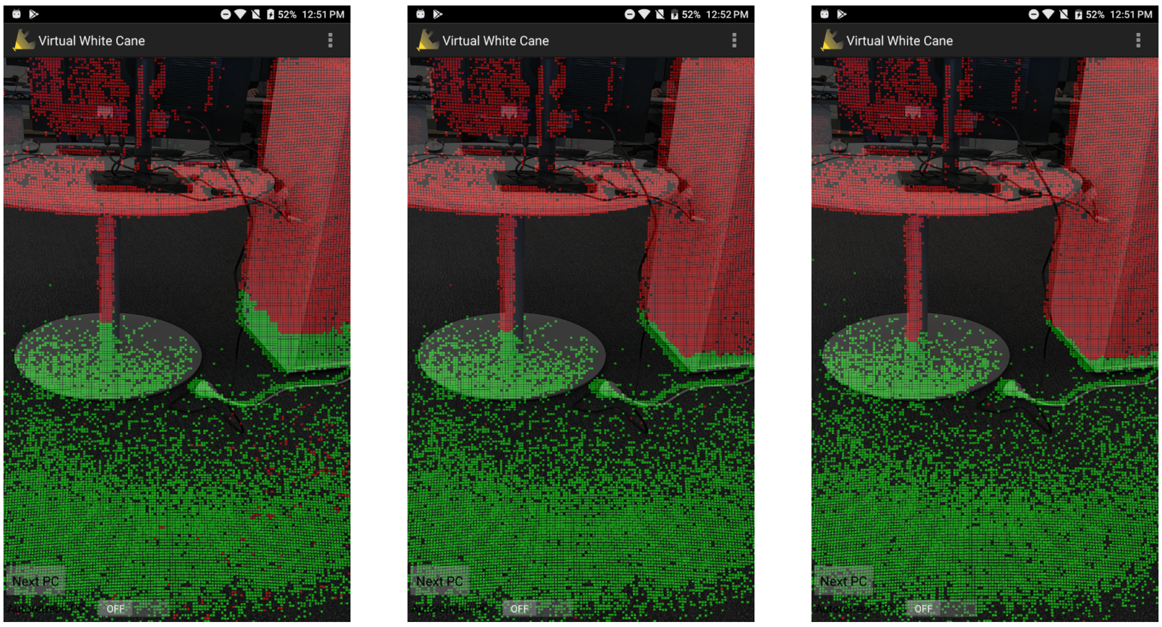


(a) v disparity. (b) RANSAC. (c) Our method.

Figure 5.4: Our implementation fixes the problems occurring with pure v disparity.

Detecting the floor in the presence of thin objects works almost equally well with each algorithm. Figure 5.5 shows how the v disparity classifies bits of the table leg as floor, while pure RANSAC and our method perform equally well.

Stairs are a very common sight in a man-made environment. Currently, downstairs are detected as an obstacle, however, they have to be visible to the sensor. This means, if no points are reflected from the stairs, the Virtual White Cane application is not aware of them. Figure 5.6 shows how the segmentation into ground plane and obstacles works. Every image shows points on the wall classified as floor, because the floor plane spans an infinite space. Still, the VI user would be warned about an obstacle in front of them because the remaining points are correctly classified as obstacles. Once the user moves closer to the stairs the first stair is considered to be the ground plane, as seen in Figure 5.7. This happens because there is a larger response from the Hough transform on the first step, due to the larger amount of points. The v disparity map of a downstairs scene shows multiple small responses for each step, as seen in Figure 5.8. The largest response at the bottom of the v disparity map represents the ground floor the user is standing on. Further, there is a discontinuity between the responses of each step. This is obvious due to the jump in depth. For stair detection, this distinctive pattern could turn out to be very helpful. Additionally, the steps further down provide a smaller



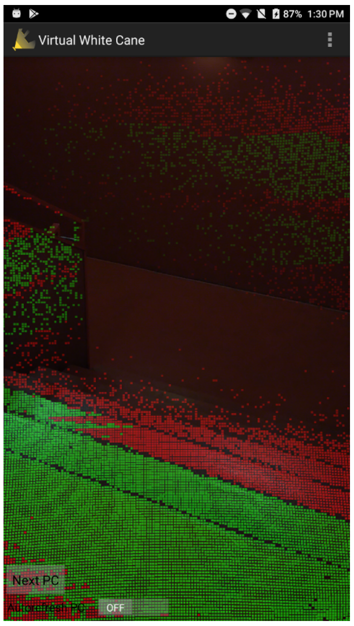
(a) v disparity. (b) RANSAC. (c) Our method.

Figure 5.5: Detecting thin objects like a pole.

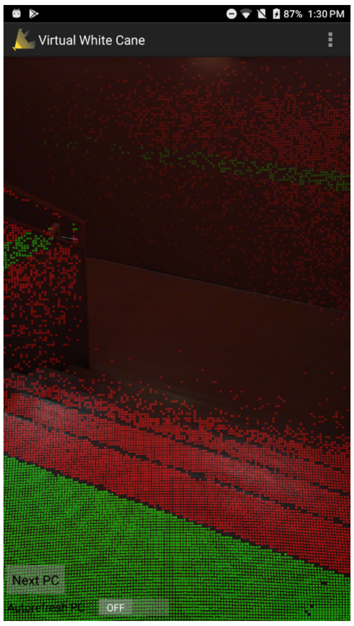
footprint.

Upstairs are also detected as obstacles, even though they would be passable. Figure 5.9 shows an upstairs scene with the floor segmented and the steps detected as obstacles. The v disparity map of an upstairs scene also shows a distinct pattern, as seen in Figure 5.10. Similar to the downstairs v disparity map, the upstairs v disparity map shows a distinctive pattern. However, the steps are connected by the vertical walls between them. These vertical walls are represented as vertical lines in the v disparity map, which connect the different lines representing the steps. This difference gets less and less pronounced the farther afield the steps are.

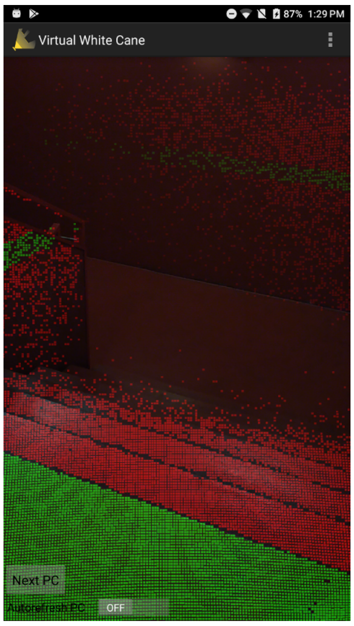
Table 5.1 shows the runtime of the different steps during ground plane segmentation. Steps like acquiring the `TangoPointcloud` or some necessary transformations are not explicitly measured, however, they are included in the "v disparity complete" and the "complete" rows. It can be seen that a frame rate of about 7 Frames Per Second (FPS) is obtained. While this is only marginally faster than the 5 FPS obtained by the depth sensor, the table helps to identify the bottlenecks in the performance. The biggest bottlenecks are the necessary transformations and the Hough transform. Especially the transformations can be easily sped up by using parallel computing. The measurements are averaged over 1000 calculations of the



(a) v disparity.



(b) RANSAC.



(c) Our method.

Figure 5.6: Downstairs scene, the stairs are considered as obstacles.

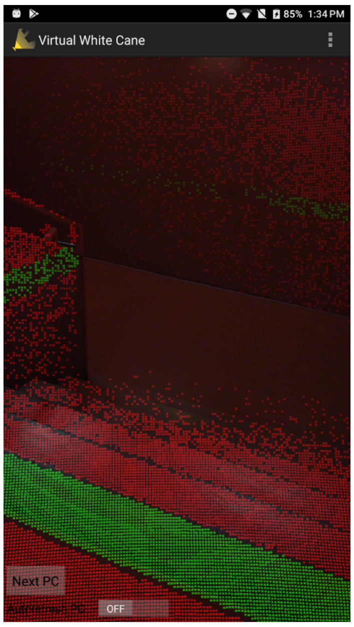


Figure 5.7: Our method on a downstairs scene. The first stair is more prominent than the actual floor the user is standing on.

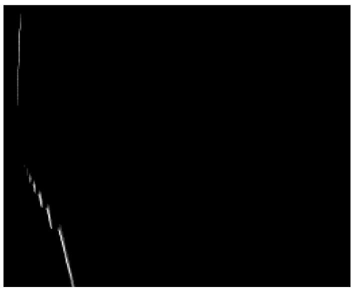
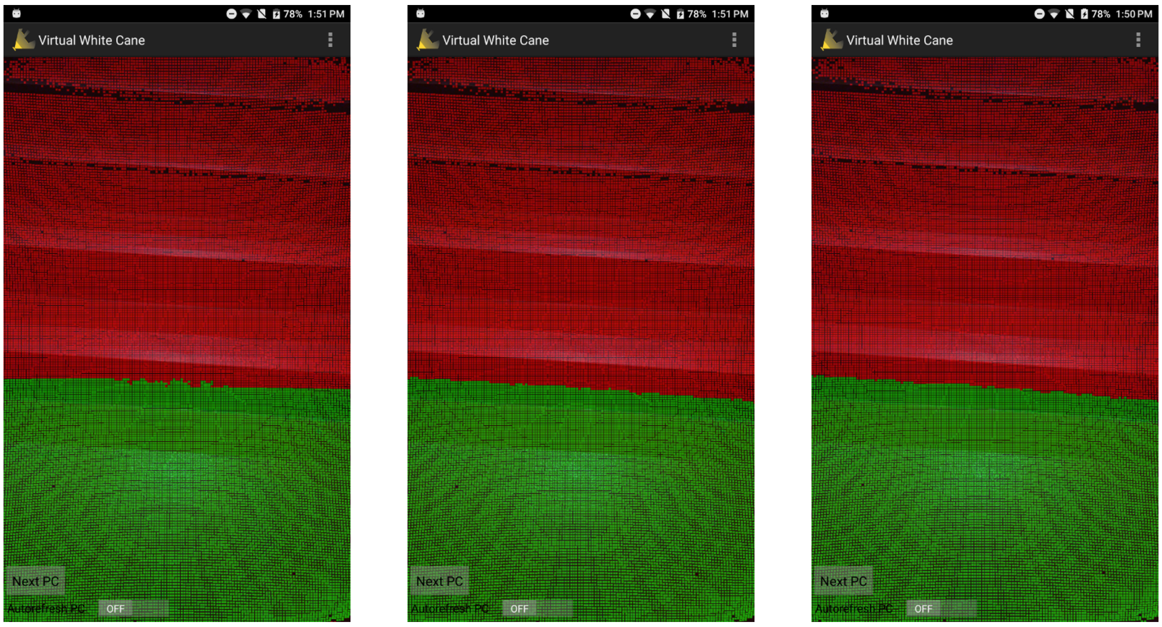


Figure 5.8: v disparity map of a downstairs scene.



(a) v disparity. (b) RANSAC. (c) Our method.

Figure 5.9: Upstairs scene, the stairs are considered as obstacles.

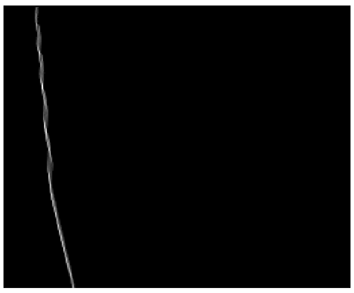


Figure 5.10: v disparity map of upstairs scene.

same static scene. The label of each row represents the following:

#Points	the average number of points obtained by the ToF sensor.
v disparity	the time it took to calculate the v disparity map with the depth map as input.
Hough	the Hough transformation and finding the best fitting line
filter	the time it takes to filter the disparity according to the given mask.
v disparity complete	the complete process to filter v disparity. Including necessary transformations and converting the Tango-Pointcloud.
RANSAC	how long it takes until the largest plane is found.
RANSAC+transformations	time RANSAC needs, as well as the necessary transformations.
complete	time it takes for the whole algorithm to find a ground plane.
FPS	time in FPS.

	Scene 1	**Scene 2**	**Scene 3**	**Scene 4**
#Points	13621	12869	12190	11909
v disparity	0.002899 s	0.002668 s	0.002636 s	0.002666 s
Hough	0.025121 s	0.024643 s	0.024968 s	0.024529 s
filter	0.004326 s	0.004420 s	0.004307 s	0.004248 s
v disparity complete	0.071596 s	0.070626 s	0.070847 s	0.069256 s
RANSAC	0.008736 s	0.009508 s	0.009607 s	0.009627 s
RANSAC+transformations	0.055555 s	0.057348 s	0.058670 s	0.059712 s
complete	0.141702 s	0.142561 s	0.143157 s	0.142376 s
FPS	7.057	7.014	6.985	7.023

Table 5.1: Runtime of ground detection.

5.2 Conservative Polar Histogram

This section shows the results of the Conservative Polar Histogram. Naturally, given by the binning in polar coordinates, the closer the object, the higher the resolution. An object close to the device will cover more bins as the same object farther afield. This allows the accurate placement of sound sources if objects are close to the user and fewer sound sources for objects far away, as they are not that important for immediate collision avoidance. It has to be noted that the ToF sensor has a larger Field of View (FOV) than the color camera (see Figure 5.11). Therefore, the leftmost and rightmost bins might deviate from the image of the scene and seem out of place.

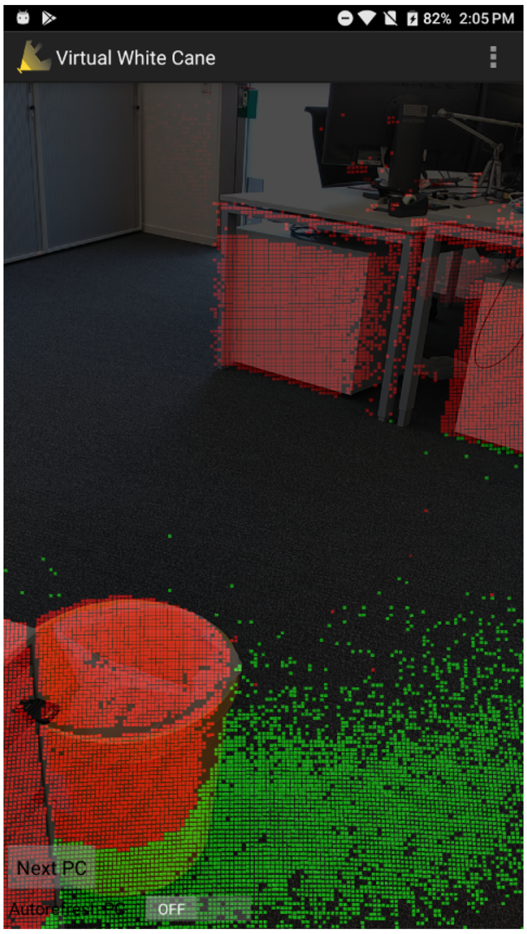
The Conservative Polar Histogram is represented as follows:

- Assign a random color to each bin, such that they can be easily distinguished.
- Draw a line for each bin representing two different values
  - The length of the line corresponds to the percentage of the total distance covered by obstacles to the distance covered by the bin.
  - The depth of the obstacle is encoded into the height on the image with a maximum of 7 m.

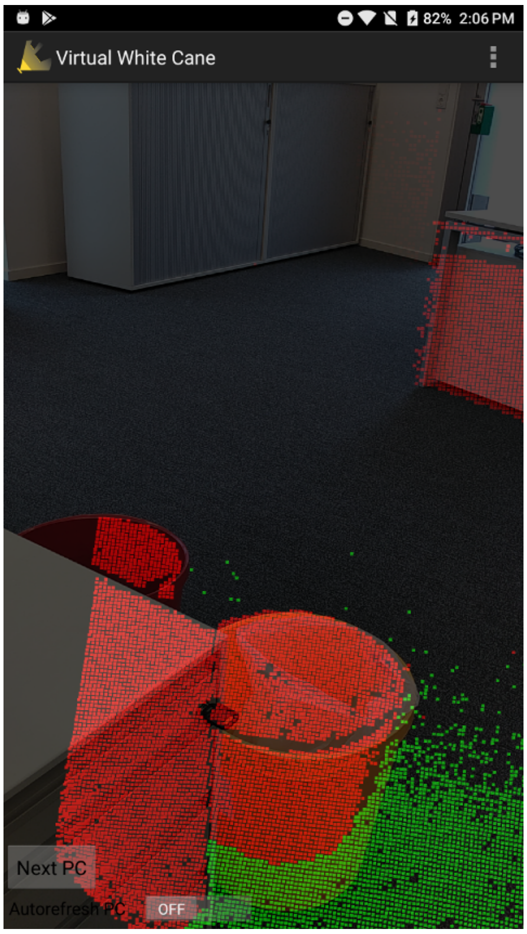
Figure 5.12 shows a scene with the corresponding Conservative Polar Histogram in the bottom right. The previously classified ground floor points are hidden and only the relevant obstacle points are projected onto the color image. It can be seen that the garbage can on the bottom left of the scene is represented by as many polar histogram bins as the table a few meters away. The bin on the far right is farther away, as it represents the object not visible in the scene due to the FOV discrepancy.

The Conservative Polar Histogram only warns about the closest obstacle. This is done to lower the cognitive load and still warn about the most important obstacles. Figure 5.13 shows how the garbage bin in front of the desk is detected and the desk in the background is only visible in "unobstructed" bins. The long magenta colored line represents the wall behind the desk.

As Table 5.2 shows the Conservative Polar Histogram runs at 50 FPS, even if almost 50% of the possible points (224×172) are considered to be an obstacle. In a less crowded scene even higher frame rates were obtained.



(a) Scene as seen by the color camera while the point cloud is recorded.



(b) Same point cloud with a slight rotation of the device.

Figure 5.11: The acquired point cloud by the ToF sensor has a larger FOV than the color camera captures.

#Points	**time**	**FPS**
6366	0.012183 s	82.082
4531	0.010480 s	95.419
16377	0.018828 s	53.112
2702	0.009527 s	104.961

Table 5.2: Timing of Conservative Polar Histogram.

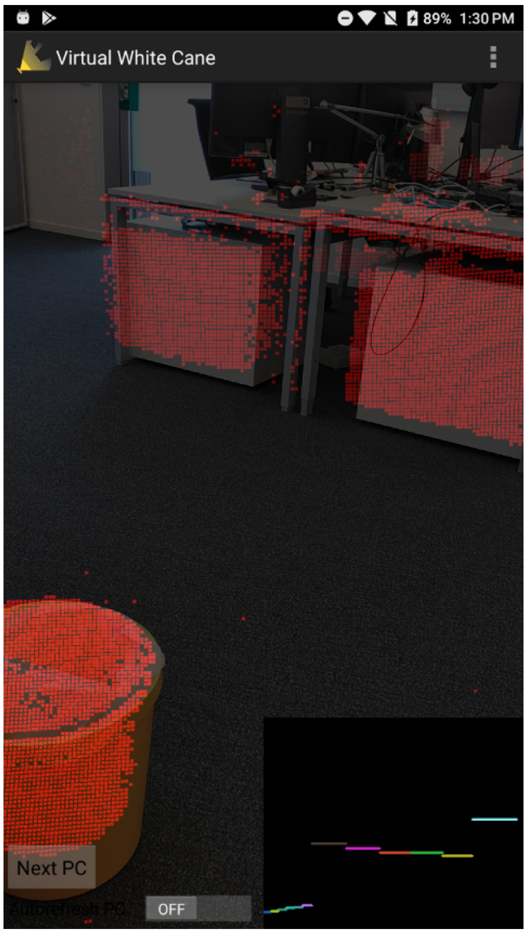


Figure 5.12: Conservative Polar Histogram.

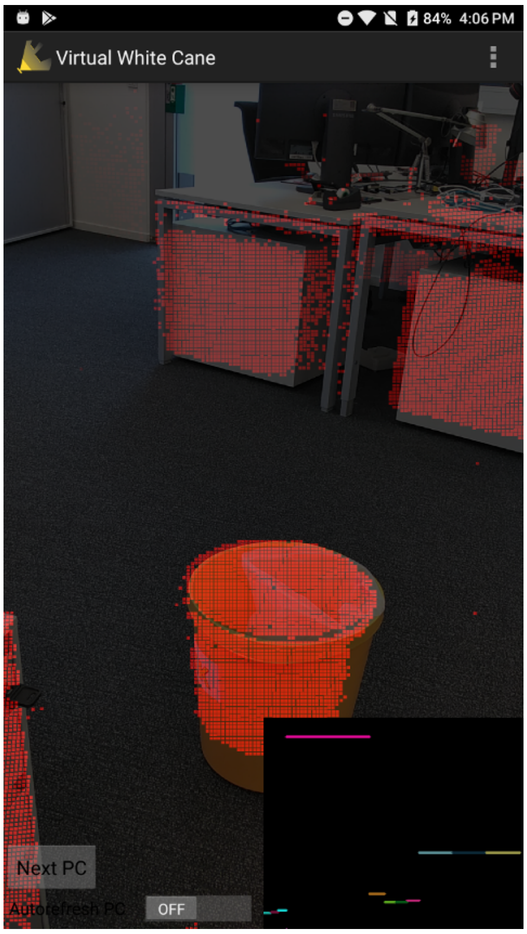


Figure 5.13: Conservative Polar Histogram result with multiple obstacles in the same bin, but only the closest is indicated.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

With the increased capabilities of smart phones due to the addition of ToF sensors, their high computational power, and the IMU, an ETA for the VI is more and more feasible. Due to the high acceptance of smart phones in every day life, they provide the ideal platform for an ETA, which is not intrusive and mobile per design.

It was shown that a modern smart phone has ample computational power to properly classify the floor in a point cloud in real-time and process the remaining points as obstacles to warn about them. A multi-threaded architecture was chosen, such that the detection of new obstacles and the warning about already known obstacles can run in parallel. To detect obstacles, the dual problem of detecting the free walkable ground plane was chosen. A combination of v disparity and RANSAC was used to overcome the weaknesses of each algorithm on their own. To speed up the algorithm, the v disparity calculation takes advantage of the parallel computing standard OpenCL, as well as the unified memory model of the Snapdragon 652. This resulted in a robust detection of the ground floor, largely independent of the device's orientation in space. Due to the IMU of the smart phone, the orientation of the device is known and can mostly be accounted for. However, the position of the device in space can only be measured in relation to the start of the service, which puts limitations on the user. It is assumed that the device is held at between 0.8 m and 1.3 m above ground.

Once the ground plane is removed, a Conservative Polar Histogram was developed to further detect the distance to obstacles in each direction and eliminate remaining outliers. The calculation of the Conservative Polar Histogram was also sped up using OpenCL's parallel computing capabilities. This allows to find the closest distance to the next obstacle in a given direction and warn the user about it.

Currently, two different warning modules exist, employing tactile and acoustic warning interfaces. The tactile warning module is limited to a response if there is an obstacle directly in front of the device, while the acoustic warning interface gives the user an mental image of the surrounding obstacles using 3-d sound provided by HRTF.

6.2 Future Work

This thesis presented a proof-of-concept ETA on a Google Tango enabled device. It provides the ideal platform to further explore the capabilities of a small and powerful ETA. This section provides various ideas and pointers for future improvements to the developed platform.

Ground floor detection: To further improve upon the robustness of the ground floor detection, the u disparity could be utilized as in [31]. Additionally, exploiting the temporal coherency between frames would allow to further refine the ground plane assumption. Further, the current system is limited to detecting a floor, whose surface normal is approximately parallel to the y-axis. Even though a natural slope would still be safely walkable, it is not properly detected.

Even though the current timing restrictions of 5 FPS are met, a higher frame rate would be beneficial, as the ToF sensor is only limited by Google Tango to 5 FPS. Improving the frame rate can be achieved by further utilizing parallel computing. The Hough transform [66], as well as the RANSAC algorithm [67] can both be greatly sped up by using a parallel implementation. Another bottleneck of the current implementation is the need to transform point clouds. Again, these transformations can be greatly sped up by employing parallel computing.

Warning Module: To improve the usability of the tactile warning module, different vibration patterns could be used to give a better indication of the distance to the next obstacle. An improvement for the acoustic warning module could be to cluster neighboring bins with approximately the same depth and placing one sound source for all of them. Indication of the object size can then be achieved by variation of the pitch, for example the lower the pitch, the larger the object.

Additional functionality: Another very important functionality would be to detect stairs. Stairs are a very common occurrence in man-made architecture, yet they are extremely dangerous and it is crucial to detect them properly. Various approaches exist that aim to detect stairs using RGB-D data [68], [69].

Detecting a crosswalk and aligning oneself towards it is another dangerous challenge faced by VI people. Hence, enhancing the functionality of the Virtual White Cane to help the VI safely cross an intersection seems like an ideal addition to the developed application. Additionally, detection of stairs and crosswalks is a similar topic, as these share the same features, which are parallel lines in RGB data, while the depth data helps to differentiate them [70].

Further, the current system does not detect closed doors as passable terrain and classifies them as obstacles, like walls. Therefore, exploiting RGB-D data to detect closed doors and inform the user about the door greatly enhances the maneuverability of the user.

Text-To-Speech functionality with OCR is another useful addition to the application. This allows the VI to read signs on the street, labels during grocery shopping, or help them identify the value of a banknote [39], [71]. Being able to read signs further helps the VI navigate independently and be able to find their destination.

Bibliography

[1] WHO, *WHO — Visual impairment and blindness*, 2014. [Online]. Available: http://www.who.int/mediacentre/factsheets/fs282/en/ (cit. on p. 1).

[2] J. Borenstein and I. Ulrich, "The GuideCane-a computerized travel aid for the active guidance of blind pedestrians," *Proceedings of International Conference on Robotics and Automation*, vol. 2, pp. 1283–1288, 1997. DOI: 10.1109/ROBOT.1997.614314 (cit. on pp. 1, 17).

[3] A. Aladren, G. Lopez-Nicolas, L. Puig, and J. J. Guerrero, "Navigation Assistance for the Visually Impaired Using RGB-D Sensor with Range Expansion," *IEEE Systems Journal*, vol. 10, no. 3, pp. 922–932, Sep. 2016. DOI: 10.1109/JSYST.2014.2320639 (cit. on pp. 1, 13).

[4] D. Bernabei, F. Ganovelli, M. D. Benedetto, M. Dellepiane, and R. Scopigno, "A Low-Cost Time-Critical Obstacle Avoidance System for the Visually Impaired," *International Conference On Indoor Positioning And Indoor Navigation*, no. September, pp. 21–23, 2011 (cit. on pp. 1, 11, 12, 34).

[5] D. Dakopoulos and N. G. Bourbakis, "Wearable Obstacle Avoidance Electronic Travel Aids for Blind: A Survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 1, pp. 25–35, Jan. 2010. DOI: 10.1109/TSMCC.2009.2021255 (cit. on pp. 1, 9).

[6] Google, *Tango — Google Developers*, 2014. [Online]. Available: https://developers.google.com/tango/ (cit. on pp. 1, 5, 6, 55, 56, 58).

[7] *Lenovo Phab 2 Pro — The World's First Google Tango-Enabled Smartphone — Lenovo® US*. [Online]. Available: http://shop.lenovo.com/us/en/tango/index.html (visited on 04/20/2017) (cit. on pp. 1, 5).

[8] *ZenFone AR (ZS571KL) — Phone — ASUS Global*. [Online]. Available: https://www.asus.com/Phone/ZenFone-AR-ZS571KL/ (visited on 05/29/2017) (cit. on p. 1).

[9] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. 2004, pp. 1–646. DOI: 10.1016/S0143-8166(01)00145-2 (cit. on p. 7).

[10] *OpenCV: Depth Map from Stereo Images*. [Online]. Available: http://docs.opencv.org/trunk/dd/d53/tutorial_py_depthmap.html (visited on 05/30/2017) (cit. on p. 8).

[11] P. J. Besl, "Active, Optical Range Imaging Sensors," *Machine Vision and Applications*, vol. 1, pp. 127–152, 1988. DOI: 10.1007/BF01212277 (cit. on p. 8).

[12] J. Salvi, S. Fernandez, T. Pribanic, and X. Llado, "A state of the art in structured light patterns for surface profilometry," *Pattern Recognition*, vol. 43, no. 8, pp. 2666–2680, 2010. DOI: 10.1016/j.patcog.2010.03.004 (cit. on p. 8).

[13] S. Zhang, "High-resolution, real-time 3-D shape measurement," PhD thesis, Stony Brook University, 2005. DOI: 10.1.1.138.3318 (cit. on p. 8).

[14] S. Zhang, M. Spie, and P. S. Huang, "From the SelectedWorks of Song Zhang High-resolution, real-time three-dimensional shape measurement High-resolution, real-time three-dimensional shape measurement," 2006. DOI: 10.1117/1.2402128 (cit. on p. 8).

[15] M. Hansard, S. Lee, O. Choi, and R. Horaud, *Time of Flight Cameras : Principles , Methods , and Applications*, ser. SpringerBriefs in Computer Science. London: Springer London, 2012, p. 95. DOI: 10.1007/978-1-4471-4658-2 (cit. on p. 10).

[16] *Kinect - Windows app development*. [Online]. Available: https://developer.microsoft.com/en-us/windows/kinect (visited on 08/28/2017) (cit. on p. 11).

[17] M. Vlaminck, L. Jovanov, P. Van Hese, B. Goossens, W. Philips, and A. Pizurica, "Obstacle detection for pedestrians with a visual impairment based on 3D imaging," in *2013 International Conference on 3D Imaging*, IEEE, Dec. 2013, pp. 1–7. DOI: 10.1109/IC3D.2013.6732091 (cit. on p. 12).

[18] J. M. Coughlan and A. Yuille, "Manhattan World: compass direction from a single image by Bayesian inference," *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, pp. 941–947, 1999. DOI: 10.1109/ICCV.1999.790349 (cit. on p. 13).

[19] J. Poppinga, N. Vaskevicius, A. Birk, and K. Pathak, "Fast plane detection and polygonalization in noisy 3D range images," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS*, IEEE, Sep. 2008, pp. 3378–3383. DOI: 10.1109/IROS.2008.4650729 (cit. on p. 13).

[20] M. Brock and P. O. Kristensson, "Supporting blind navigation using depth sensing and sonification," *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication - UbiComp '13 Adjunct*, p. 255, 2013. DOI: 10.1145/2494091.2494173 (cit. on p. 13).

[21] A. Bhowmick, S. Prakash, R. Bhagat, V. Prasad, and S. M. Hazarika, "IntelliNavi: Navigation for Blind Based on Kinect and Machine Learning," *Multi-disciplinary Trends in Artificial Intelligence*, pp. 172–183, 2014. DOI: 10.1007/978-3-319-13365-2_16 (cit. on p. 14).

[22] R. Jafri and M. M. Khan, "Obstacle detection and avoidance for the visually impaired in indoors environments using Google's project tango device," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9759, pp. 179–185, 2016. DOI: 10.1007/978-3-319-41267-2_24 (cit. on p. 14).

[23] R. Jafri, *A GPU-accelerated real-time contextual awareness application for the visually impaired on Google's project Tango device*, Oct. 2016. DOI: 10.1007/s11227-016-1891-8 (cit. on p. 14).

[24] R. Jafri, R. L. Campos, S. A. Ali, and H. R. Arabnia, "Utilizing the Google Project Tango Tablet Development Kit and the Unity Engine for Image and Infrared Data-Based Obstacle Detection for the Visually Impaired," in *Proceedings of the 2016 International Conference on Health Informatics and Medical Systems (HIMS'15)*, Las Vegas, Nevada, USA, 2016, pp. 163–164 (cit. on p. 14).

[25] Y. Gao, X. Ai, J. Rarity, and N. Dahnoun, "Obstacle detection with 3D camera using U-V-Disparity," *International Workshop on Systems, Signal Processing and their Applications, WOSSPA*, pp. 239–242, May 2011. DOI: 10.1109/WOSSPA.2011.5931462 (cit. on pp. 14, 80).

[26] R. Labayrade, D. Aubert, and J.-P. Tarel, "Real time obstacle detection in stereovision on non flat road geometry through "v-disparity"representation," *Intelligent Vehicle Symposium, 2002. IEEE*, vol. 2, no. January, pp. 646–651, 2002. DOI: 10.1109/IVS.2002.1188024 (cit. on pp. 14–16, 21, 22).

[27] B. Peasley and S. T. Birchfield, "Real-time obstacle detection and avoidance in the presence of specular surfaces using an active 3D sensor," in *2013 IEEE Workshop on Robot Vision, WORV 2013*, IEEE, Jan. 2013, pp. 197–202. DOI: 10.1109/WORV.2013.6521938 (cit. on p. 14).

[28] A. Broggi, C. Caraffi, R. I. Fedriga, and P. Grisleri, "Obstacle Detection with Stereo Vision for Off-Road Vehicle Navigation," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)-Workshops*, 2005, pp. 65–65. DOI: 10.1109/CVPR.2005.503 (cit. on p. 15).

[29] N. Soquet, D. Aubert, and N. Hautiere, "Road Segmentation Supervised by an Extended V-Disparity Algorithm for Autonomous Navigation," *Ivs*, pp. 160–165, 2007. DOI: `10.1109/IVS.2007.4290108` (cit. on p. 16).

[30] M. Bai, Y. Zhuang, and W. Wang, "Stereovision based obstacle detection approach for mobile robot navigation," *2010 International Conference on Intelligent Control and Information Processing*, pp. 328–333, 2010. DOI: `10.1109/ICICIP.2010.5565220` (cit. on p. 16).

[31] B. Musleh, A. De La Escalera, and J. M. Armingol, "U-V disparity analysis in urban environments," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6928 LNCS, 2012, pp. 426–432. DOI: `10.1007/978-3-642-27579-1_55` (cit. on pp. 16, 102).

[32] A. Rodríguez, J. J. Yebes, P. F. Alcantarilla, L. M. Bergasa, J. Almazán, and A. Cela, "Assisting the visually impaired: obstacle detection and warning system by acoustic feedback.," *Sensors (Basel, Switzerland)*, vol. 12, no. 12, pp. 17 476–96, Dec. 2012. DOI: `10.3390/s121217476` (cit. on p. 17).

[33] J. M. Saez Martinez and F. E. Ruiz, "Stereo-based Aerial Obstacle Detection for the Visually Impaired," in *Workshop on Computer Vision Applications for the Visually Impaired*, 2008 (cit. on p. 17).

[34] D. Koester, B. Schauerte, and R. Stiefelhagen, "Accessible section detection for visual guidance," in *Electronic Proceedings of the 2013 IEEE International Conference on Multimedia and Expo Workshops, ICMEW 2013*, 2013. DOI: `10.1109/ICMEW.2013.6618351` (cit. on p. 17).

[35] S. Shoval, I. Ulrich, and J. Borenstein, "NavBelt and the GuideCane," *IEEE Robotics and Automation Magazine*, vol. 10, no. 1, pp. 9–20, 2003. DOI: `10.1109/MRA.2003.1191706` (cit. on p. 18).

[36] E. Peng, P. Peursum, L. Li, and S. Venkatesh, "A Smartphone-Based Obstacle Sensor for the Visually Impaired," in, Springer Berlin Heidelberg, 2010, pp. 590–604. DOI: `10.1007/978-3-642-16355-5_45` (cit. on p. 18).

[37] H. Shen, K.-Y. Chan, J. M. Coughlan, and J. Brabyn, "A Mobile Phone System to Find Crosswalks for Visually Impaired Pedestrians," *Technology & Disability*, vol. 20, no. 3, pp. 217–224, 2008. DOI: `10.1016/j.surg.2006.10.010.Use` (cit. on p. 19).

[38] A. S. Martinez-Sala, F. Losilla, J. C. Sánchez-Aarnoutse, and J. García-Haro, "Design, Implementation and Evaluation of an Indoor Navigation System for Visually Impaired People.," *Sensors (Basel, Switzerland)*, vol. 15, no. 12, pp. 32 168–87, Dec. 2015. DOI: `10.3390/s151229912` (cit. on p. 19).

[39] A. S. Shaik, G. Hossain, and M. Yeasin, "Design, development and performance evaluation of reconfigured mobile Android phone for people who are blind or visually impaired," *Proceedings of the 28th ACM International Conference on Design of Communication - SIGDOC '10*, p. 159, 2010. DOI: 10.1145/1878450.1878478 (cit. on pp. 19, 103).

[40] V. N. Hoang, T. H. Nguyen, T.-L. Le, T. T. H. Tran, T. P. Vuong, and N. Vuillerme, "Obstacle detection and warning for visually impaired people based on electrode matrix and mobile Kinect," *Proceedings of 2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science, NICS 2015*, pp. 54–59, 2015. DOI: 10.1109/NICS.2015.7302222 (cit. on p. 20).

[41] T. Amemiya, "Haptic Direction Indicator for Visually Impaired People Based on Pseudo-Attraction," *Methods*, vol. I, no. 5, 2009 (cit. on p. 20).

[42] T. Amemiya and H. Sugiyama, "Haptic Handheld Wayfinder with Pseudo-attraction Force for Pedestrians with Visual Impairments," *Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility*, pp. 107–114, 2009. DOI: 10.1145/1639642.1639662 (cit. on p. 20).

[43] S. Mann, J. Huang, R. Janzen, R. Lo, V. Rampersad, A. Chen, and T. Doha, "Blind navigation with a wearable range camera and vibrotactile helmet," in *Proceedings of the 19th ACM international conference on Multimedia - MM '11*, New York, New York, USA: ACM Press, 2011, p. 1325. DOI: 10.1145/2072298.2072005 (cit. on p. 20).

[44] J. A. MacDonald, P. P. Henry, and T. R. Letowski, "Spatial audio through a bone conduction interface," *International Journal of Audiology*, vol. 45, no. 10, pp. 595–599, Jan. 2006. DOI: 10.1080/14992020600876519 (cit. on p. 20).

[45] P. M. Hofman, J. G. Van Riswick, and A. J. V. Opstal, "Relearning sound localization with new ears," *Nature Neuroscience*, vol. 1, no. 5, pp. 417–421, Sep. 1998. DOI: 10.1038/1633 (cit. on p. 21).

[46] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981. DOI: 10.1145/358669.358692 (cit. on p. 23).

[47] R. B. Rusu, "Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments," *KI - Kunstliche Intelligenz*, vol. 24, pp. 345–348, 2010. DOI: 10.1007/s13218-010-0059-6 (cit. on p. 23).

[48] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. DOI: 10.1145/361002.361007 (cit. on p. 23).

[49] S. Holzer, R. B. Rusu, M. Dixon, S. Gedikli, and N. Navab, "Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images," in *IEEE International Conference on Intelligent Robots and Systems*, IEEE, Oct. 2012, pp. 2684–2689. DOI: 10.1109/IROS.2012.6385999 (cit. on pp. 23, 73).

[50] J. Borenstein and Y. Koren, "The Vector Field Histogram - Fast obstacle avoidance for mobile robots," *IEEE Journal of Robotics and Automation*, vol. 7, no. 3, pp. 278–288, Jun. 1991. DOI: 10.1109/70.88137 (cit. on p. 23).

[51] I. Ulrich and J. Borenstein, "VFH*: Local Obstacle Avoidance with Look-Ahead Verification," *Robotics*, pp. 2505–2511, 2000. DOI: 10.1109/ROBOT.2000.846405 (cit. on p. 33).

[52] *The Android Source Code — Android Open Source Project*. [Online]. Available: https://source.android.com/source/ (visited on 07/10/2017) (cit. on p. 50).

[53] M. Gargenta, *Learn about Android Internals and NDK - YouTube*. [Online]. Available: https://www.youtube.com/watch?v=byFTAhXVF7k (visited on 09/18/2017) (cit. on p. 51).

[54] *JNI Design Overview*. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html#wp1253 (visited on 07/12/2017) (cit. on p. 52).

[55] D. Brown, "Decentering Distortion of Lenses," *Photometric Engineering*, vol. 32, no. 3, pp. 444–462, 1966 (cit. on p. 57).

[56] *OpenCL - The open standard for parallel programming of heterogeneous systems*. [Online]. Available: https://www.khronos.org/opencl/ (visited on 04/27/2017) (cit. on p. 59).

[57] L. Howes and A. Munshi, "The OpenCL Specification," [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf (cit. on pp. 59, 60).

[58] *Adreno GPU SDK - Tools - Qualcomm Developer Network*. [Online]. Available: https://developer.qualcomm.com/software/adreno-gpu-sdk/tools (visited on 04/28/2017) (cit. on p. 61).

[59] B. König, *Wikipedia - OpenCL Memory model*. [Online]. Available: https://de.wikipedia.org/wiki/Datei:OpenCL_Memory_model.svg (visited on 04/27/2017) (cit. on p. 62).

[60] *jocl.org - Java bindings for OpenCL*. [Online]. Available: http://www.jocl.org/ (visited on 07/05/2017) (cit. on p. 61).

[61] A. O. Tutorial, "OpenCL^{TM} Basic Tutorial for Android* OS User's Guide Compute Code Builder -Samples," [Online]. Available: https://software.intel.com/sites/default/files/managed/d3/18/AndroidBasicOpenCL.pdf (cit. on p. 63).

[62] *OpenSL ES — Android Developers*. [Online]. Available: https://developer.android.com/ndk/guides/audio/opensl/index.html (visited on 05/03/2017) (cit. on p. 63).

[63] Y. Ben-Tsvi, N. Charles, T. Longo, and S. Chao, "OpenSL ES 1.0.1 Specification," 2009. [Online]. Available: https://www.khronos.org/registry/OpenSL-ES/specs/OpenSL_ES_Specification_1.0.1.pdf (cit. on p. 63).

[64] *PCL - Point Cloud Library (PCL)*. [Online]. Available: http://www.pointclouds.org/ (visited on 05/02/2017) (cit. on p. 64).

[65] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *Proceedings - IEEE International Conference on Robotics and Automation*, IEEE, May 2011, pp. 1–4. DOI: 10.1109/ICRA.2011.5980567 (cit. on p. 64).

[66] R. Yam-Uicab, J. L. Lopez-Martinez, J. A. Trejo-Sanchez, H. Hidalgo-Silva, and S. Gonzalez-Segura, "A fast Hough Transform algorithm for straight lines detection in an image using GPU parallel computing with CUDA-C," *The Journal of Supercomputing*, pp. 1–20, Apr. 2017. DOI: 10.1007/s11227-017-2051-5 (cit. on p. 102).

[67] A. Hidalgo-Paniagua, M. A. Vega-Rodríguez, N. Pavón, J. Ferruz, A. Hidalgo-Paniagua, M. A. Vega-Rodríguez, N. Pavón, and J. Ferruz, "A Comparative Study of Parallel RANSAC Implementations in 3D Space," *International Journal of Parallel Programming*, vol. 43, no. 43, 2015. DOI: 10.1007/s10766-014-0316-7 (cit. on p. 102).

[68] C. Stahlschmidt, A. Gavriilidis, and A. Kummert, "Classification of ascending steps and stairs using Time-of-Flight sensor data," in *2015 IEEE 9th International Workshop on Multidimensional (nD) Systems, nDS 2015*, IEEE, Sep. 2015, pp. 1–6. DOI: 10.1109/NDS.2015.7332643 (cit. on p. 102).

[69] A. Perez-Yus, D. Gutierrez-Gomez, G. Lopez-Nicolas, and J. J. Guerrero, *Stairs detection with odometry-aided traversal from a wearable RGB-D camera*, 2016. DOI: 10.1016/j.cviu.2016.04.007 (cit. on p. 102).

[70] S. Wang and Y. Tian, "Detecting stairs and pedestrian crosswalks for the blind by RGBD camera," in *Proceedings - 2012 IEEE International Conference on Bioinformatics and Biomedicine Workshops, BIBMW 2012*, IEEE, Oct. 2012, pp. 732–739. DOI: `10.1109/BIBMW.2012.6470227` (cit. on p. 103).

[71] F. M. Hasanuzzaman, X. Yang, and Y. Tian, "Robust and effective component-based banknote recognition for the blind," *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 42, no. 6, pp. 1021–1030, Nov. 2012. DOI: `10.1109/TSMCC.2011.2178120` (cit. on p. 103).