Freie Universität Berlin

Fachbereich Mathematik und Informatik

Masterarbeit

Three dimensional occlusion mapping using a LiDAR sensor

von: David Damm

Matrikelnummer: 4547660

Email: david.damm@fu-berlin.de

Betreuer: Prof. Dr. Daniel Göhring und Fritz Ulbrich

Berlin, 30.08.2018

Abstract

Decision-making is an important task in autonomous driving. Especially in dynamic environments, it is hard to be aware of every unpredictable event that would affect the decision. To face this issue, an autonomous car is equipped with different types of sensors. The LiDAR laser sensors, for example, can create a three dimensional 360 degree representation of the environment with precise depth information. But like humans, the laser sensors have their limitations. The field of view could be partly blocked by a truck and with that, the area behind the truck would be completely out of view. These occluded areas in the environment increase the uncertainty of the decision and ignoring this uncertainty would increase the risk of an accident. This thesis presents an approach to estimate such areas from the data of a LiDAR sensor. Therefore, different methods will be discussed and finally the preferred method evaluated.

Statement of Academic Integrity

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such. This paper has neither been previously submitted to another authority nor has it been published yet.

August 30, 2018

David Damm

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Fully autonomous driving is one of the biggest challenges for the car industry. Traditional manufacturers are no longer the only parties involved in the development of autonomous technologies. A growing number of companies from different industrial sectors are working on components for self driving cars (see [Muo17] and [Kap18]) and the amount of software used inside the car is increasing rapidly. The transition from manually driven cars to completely self driving cars is a long process and will take many years. SAE international has published the standard J3016, which divides this process into six levels [SAE14]. Figure 1.1 shows an overview of all levels.

The first three levels range from no automation up to automatic steering and acceleration. This means, they still require a driver for the monitoring of the environment to eventually intervene. In the last three levels, monitoring is part of the system and the responsibility of the driver decreases from *being ready to intervene* to none. To reach the last three levels, it will be necessary to develop a system that can compete with or exceed the human perception. It is necessary to collect and interpret the information that is automatically included in the decision making of the human driver. One example is the localization of occluded regions in the field of view of the driver. A human driver can implicitly estimate which regions are not visible to him and interprets this information. If parts of the region that is relevant for driving a maneuver are hidden, a human driver will drive more carefully than with a free view. One example is the right turn maneuver onto a main road, where the car has to give priority to the cars in the lane it is driving into. If big parts of this lane are hidden by a truck or a car, the human driver should move forward slowly until he gets a better view of the lane.

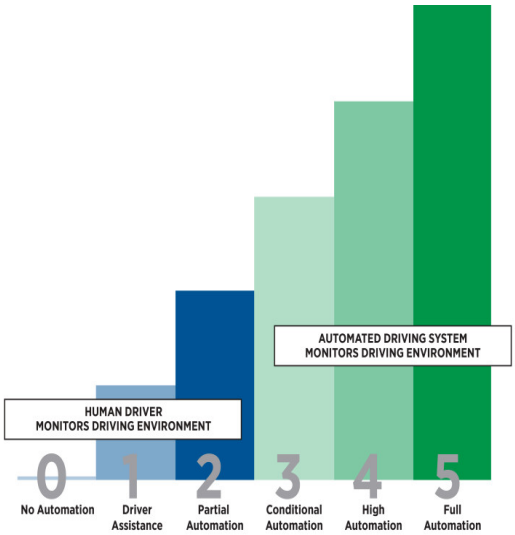


Figure 1.1: Level of autonomous driving as defined by SAE [SAE14]

In order to replace the human driver in the future, autonomous cars have to be aware of hidden regions. This is necessary to make correct decisions in street traffic. If the decision making process of the car just focuses on moving objects on the lane, it could get a false confidence and could decide that it is save to turn into the priority lane even though most of the lane is hidden by other obstacles.

1.2 Goals

The objective of this thesis is to develop a system that can provide information about hidden regions in street traffic to the planing components of a car. In order to reach this goal, the depth information of laser sensors will be processed to estimate the occlusions. The estimated occlusions will then be provided to the planing components. Different methods will be compared to each other and the preferred method will be evaluated regarding its computation time and its accuracy.

1.3 Thesis Structure

After providing a brief introduction into this thesis, the following chapter will focus on the fundamentals like the used software and methods. The third chapter starts with a comparison of different methods for the occlusion estimation and presents an implementation of the preferred method. At the end of chapter 3, further improvements will be presented and discussed. In chapter 4, the used method will separated into smaller tasks, which will then be evaluated. Chapter 5 gives a final conclusion and ends this thesis with an outlook on future work.

1.4 Related work

Object detection and decision making under uncertainty are two big topics in the field of autonomous driving. Both are affected by sensor occlusions. Different approaches have been presented in the last years to handle such uncertainties[Tho18, NB08, GOE15]. Object detection is affected by complete or partially occluded objects. For partially occluded objects, it is harder to classify them because some of the relevant features could be missing. Another issue is the motion tracking of the object. They could disappear behind an occluder or suddenly appear in the field of view.

The author of [Tho18] presents a way of controlling the vehicle speed in case of a pedestrian crosswalk that is concealed by a vehicle. As in many control and decision making algorithms, the author uses a planar projection of the hidden areas because most of the 3D information is superfluous for the used scenario. The planar projection presents the occlusions in a 2D map. In [MRN13], an occlusion map is used for a right turn driving situation in a left-hand traffic, where the driver needs to negotiate. Here, the author also uses a planar projection of the occlusions.

In both cases a specific region needs to be considered in their decision making process and leads to the conclusion that certain driving maneuvers need the observation of certain regions. Such regions are called *regions of interest*. If big parts of such regions are not visible, the velocity should be reduced accordingly.

They also show that, with the consideration of the hidden areas, the risk of an accident can be decreased. The car can reduce the velocity and brake in time if a pedestrian is crossing the crosswalk. As already mentioned, both papers use a planar occlusion estimation. This can be due to the orientation and the position of the sensor. They are often positioned in the bumper and have a view that is parallel to the ground plane. A 3D method would bring no significant advantages. But in case of a high positioned sensor with a sloping view of the street, the 3D information can be beneficial in some situations and give a more precise occlusion estimation. The sensor can, for example, overlook smaller obstacles, so that the hidden areas are in fact not as big as they seem in a planar projection.

In [OG13], the author presents different ways of detecting 3D occlusions in aerial photos. Due to the perspective view, they always contain distortions. These distortions render them unreliable for measurements and prevent a correct plane projection of the images. The geometrically corrected aerial photos, also called orthophotos, present buildings and objects with a top view. To correct such distortions, it is necessary to find the occlusions in aerial images. In this scenario, the occlusions are caused by tall buildings and other objects. The paper presents a method, among others, that computes the occlusions by comparing different depth information and creating a depth buffer. This method has its origin in computer graphics and is also the chosen method in this thesis.

2 Fundamentals

This chapter provides a general background and a introduction to the basic knowledge and techniques used in this thesis. It starts with the autonomous vehicle and the sensor used for acquiring the data. Following those, the software that is being used and the methods will be introduced.

2.1 AutoNOMOS Cars

Since 1998, Freie Universität Berlin has been building different types of autonomous robots from small (<18cm) to mid size(<50cm)[FF]. They participated in several robocup competitions. After visiting the Stanford University in 2006 and with help of its team, the AutoNOMOS team decided to take part in the field of autonomous driving [Autb].

The project started with the *Spirit of Berlin* (fig. 2.1), a modified Dodge Caravan for handicapped drivers. It was already equipped with a few components, such as a linear lever for braking and gas, which supports the autonomous controlling of the vehicle. After installing additional electronics and further development of the vehicle, it was ready to participate in the DARPA Urban challenge in 2007 [Dar]. The participating vehicles had to perform complex maneuvers such as merging, parking and negotiating intersections in a certain time frame.

The successor of the *Spirit of Berlin*, the *Made in Germany* (fig. 2.2), is a Volkswagen Passat Variant 3c, which was modified by the Volkswagen Research Garage to fit the needs of autonomous driving [Autc].

To access actuating elements, it is equipped with the Drive-by-Wire technology, which provides direct access via the CAN-BUS. This allows the controlling of the vehicle by software. In the front window are three cameras mounted for

Figure 2.1: Spirit of Berlin [Autd]

visual recognition, which can be used for line detection and other image based techniques. Additionally, the vehicle has 6 IBEO LUX laser sensors and one 360° Velodyne HDL-64E laser sensor on the roof. These components use laser light pulses to scan the environment.

For the localization, the car is equipped with a precise GPS-System, which can be combined with other sensors to provide an accurate position.

The third vehicle is called *e-Instein*, a modified Misubishi i-MiEV [Auta]. For accessing the actuating elements, it uses the Drive-by-Wire technology, as does the MIG. It has less sensors than the MIG and focuses on navigation and indoor localization. Therefore, it is also equipped with laser sensors and cameras.

2.2 Velodyne LiDAR sensor

The focus of this thesis lies on the depth information coming from a Velodyne HDL-64E sensor. It is a LiDAR sensor with a 360° field of view and a data rate of more than one million points per second.

LiDAR stands for Light Detection and Ranging and can be used for detecting surfaces. In the autonomous industry it is often used for object detection and other methods for environment perception. Apart from that, it is often

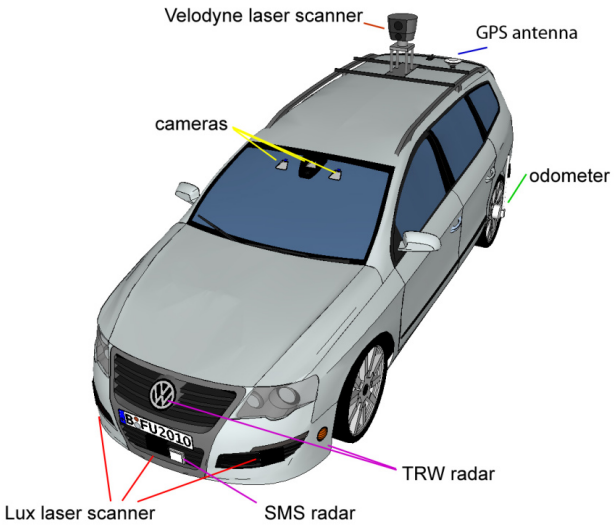


Figure 2.2: Made In Germany [Autc]

applied for geographical and archaeological research, where it is used for examining natural and manmade environments from the air [NOA]. The technology is based on the RADAR technology, which emits radio waves and uses the reflections for distance calculations [Dav]. But instead of emitting radio waves, a LiDAR sensor emits infrared light pulses (lasers) that are not visible to the human eye.

The time interval between sending and receiving the light pulse is measured and used for calculating the distance of the impact point with the formula:

$$Distance = \frac{SpeedOfLight * TimeOfFlight}{2} \tag{2.1}$$

The speed of light is ~300,000,000 $\frac{m}{s}$ and therefore the sensors must be extremely fast to measure the time difference between sending and receiving the light pulse. For example, the time difference for an object at a distance of 10 m is:

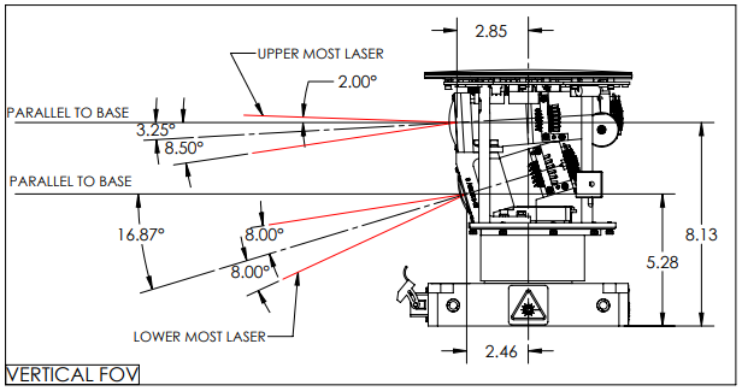


Figure 2.3: HDL-64E S2 outline drawing [Vel]

$$\begin{aligned} TimeOfFlight &= \frac{Distance * 2}{SpeedOfLight} \\ &= \frac{20m}{3 * 10^8 \frac{m}{s}} \\ &= 6.6666 * 10^{-8} s \end{aligned} \tag{2.2}$$

Even though the interval between sending and receiving is very short, the position of the sensor between sending and receiving the light pulse can differ and distort the measurement. Therefore, it is necessary for a moving LiDAR sensor to include the movement of the sensor in the calculation. The calculated impact points can then be stored as a point cloud containing the coordinates of the points.

The Velodyne HDL-64E sensor uses 64 lasers, which are grouped in one upper block and one lower block (fig. 2.3). Each block has two groups of 16 laser emitters on the left and the right side of the laser receiver[Velnd]. With a vertical range of ~26.8°, the vertical resolution is ~0.4°. But the angles are not evenly distributed. The upper block has a range of ~10.5° and therefore a resolution of ~0.328°. Whereas the lower block has a range of ~16° and a resolution of 0.5°. The Velodyne sensor is mounted on the roof of the car, as seen in figure 2.2. To create the 360° field of view, the sensor rotates with a velocity of 5-15 Hz and starts a scan at approximately every 0.09 degrees. The light pulse has a frequency of 20 kHz and creates up to 1.3 million measurements per sec-

ond. The following listing shows a quick overview of the Velodyne HDL-64E specifications.

Specs [Velnd]:

- 64 lasers/detectors
- 360 degree field of view (azimuth)
- 0.09 degree angular resolution (azimuth)
- 26.8 degree vertical field of view (elevation) +2° up to -24.8° down with 64 equally spaced angular subdivisions (~ 0.4°)
- <5 cm distance accuracy
- 5-15 Hz rotation rate update (user selectable)
- 50 m range for pavement (~ 0.10 reflectivity)
- 120 m range for cars and foliage (~ 0.80 reflectivity)
- >1M points per second
- <0.05 ms latency

In the best case, all of the 64 lasers lie on a vertical line and the intersection point is the origin of the sensor frame, as seen in figure 2.3. This optimal configuration could provide a precise measurement of the environment. But due to hardware limitations every Velodyne sensor has an unique alignment of its laser emitters and provides small distortions. Because of the need of highly precise measurements, each Velodyne sensor is shipped with an unique calibration file. It contains the following parameters:

rotCorrection rotational angle correction around the z-axis

vertCorretion vertical angle correction around the y-axis

distCorrection correction of the measured distance

vertoffsetCorrection height of each sensor, one fixed value for each laser block

horizOffsetCorrection horizontal offset of each laser as viewed from the back of the laser

The measurements of the Velodyne sensor are published successively as UDP Ethernet packets during the rotation. One packet can be either for the lower block or for the upper block of the sensor and contains the following information:

header info indicator for lower or upper block

rotational info gives the rotational position

32 laser returns each return contains the distance and intensity

The corrective parameter can be used to calculate the precise 3D impact points from the data packets. [SHN12] describes a way of self calibrating such 3D lasers by finding the optimal parameters. Therefore, they pose the quality of the 3D point cloud produced by the sensor as a maximization problem. The quality is therefore defined as a function and the parameter are then optimized to maximize the quality.

2.3 ROS

The Robot Operating System (ROS) [ROSa] is a Software framework for developing robot components and encourage collaborative robotics software development. It offers libraries, tools and conventions, which help developing robot applications. The libraries are released under the BSD license and are free for commercial and private usage. With its big community it also offers help and and a big variety of third party open source libraries. More than 3000 packages are publicly available.

The strength of this framework is the distributed and modular system. Any group that is interested can participate and create its module and repository [ROSb]. The core functionalities are the communications infrastructure, robotic-specific features and other tools like the visualization tool RViz [ROSc]. The communications infrastructure allows interprocess messages and request/response function calls. The processes can either be nodes or nodelets. The latter can run multiple algorithms with the benefit of zero copy transport [ROSd]. With that, the shared data does not have to be copied between the nodelets. The robotic-specific features are common functionalities that are used

in robotic systems, such as the navigation, localization, communication interfaces and a robot description language.

2.4 Coordinate systems and transformations

This section describes the transformations between different coordinate systems and between different representations of coordinates.

A robot system usually consists of several components. These components have their own local 3D coordinate system, called frames. The Velodyne sensor has two of these frames. One for the whole Velodyne component, which is called *velodyne_base_link* and one for the sensor itself, the *velodyne* frame. The local frame for the car is called *base_link*, which is the commonly used name for a mobile base frame.[ROSe]

The transformation of a point from one frame into another is done via multiple transformations, in which each transformation only transforms one point into a directly connected frame. The entirety of all transformations are called *transformation tree*. A transformation for a 3D space is defined by a 4x4 matrix, which consists of a rotational part and a translational part. For example, a point P_v of the *velodyne* frame is first transformed into the *velodyne_base_link* frame by multiplying it with the transformation matrix T_1. It is then further transformed into the *base_link* frame by multiplying the previous result with T_2.

$$P_b = T_2 * T_1 * P_v \tag{2.3}$$

By multiplying several transformations, they can also be expressed as only one, which can transform a point directly into the target frame.

$$T = T_2 * T_1 \tag{2.4}$$

$$P_b = T * P_v \tag{2.5}$$

If a transformation does not change over time, it is called a static transformation and its transformation matrix stays the same.

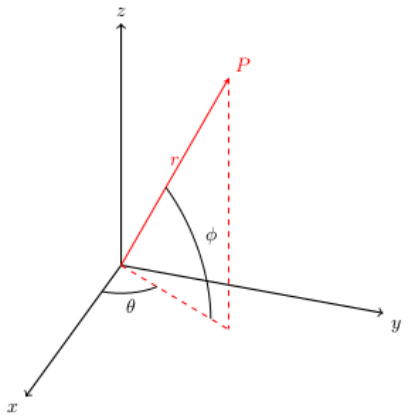


Figure 2.4: A spherical coordinate system, in which P is represented by (r, θ, ϕ).

Additionally to the frame transformation, the position of a point can also be expressed in different representations. The commonly used representation uses the Cartesian coordinate system. This system consists of perpendicular coordinate axis (X,Y,Z in 3D space). A point can be localized by its signed distance from each axis. Another representation uses a spherical coordinate system. In this system, a point can be localized by the distance to the origin and its angles around the coordinate axis(see fig. 2.4). Multiple different conventions exist for this representation[Wei]. Sometimes the angle symbols are switched or the colatitude($\phi = 90$ - γ, where γ is the latitude) is used instead of the latitude. Figure 2.4 shows the representation used in this thesis and the transformations between the representations are shown below.

$$\begin{pmatrix} r \\ \theta \\ \phi \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2 + z^2} \\ \tan^{-1}(\frac{y}{x}) \\ \sin^{-1}(\frac{z}{r}) \end{pmatrix} \tag{2.6}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r * \cos(\phi) * \cos(\theta) \\ r * \cos(\phi) * \sin(\theta) \\ r * \sin(\phi) \end{pmatrix} = r * \begin{pmatrix} \cos(\phi) * \cos(\theta) \\ \cos(\phi) * \sin(\theta) \\ \sin(\phi) \end{pmatrix} \tag{2.7}$$

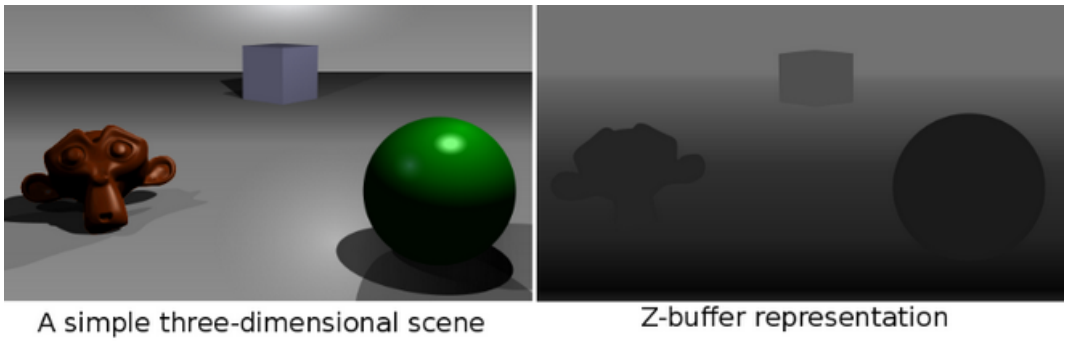


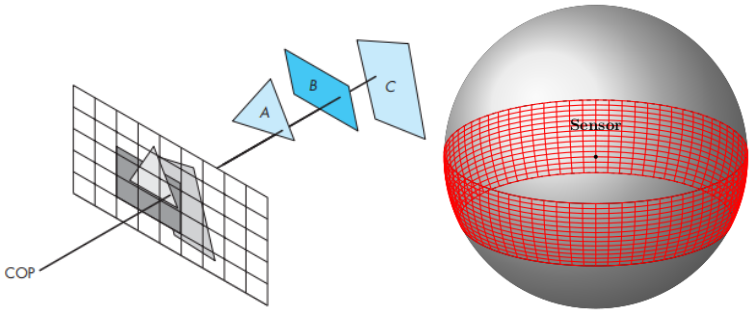
Figure 2.5: Visualized depth buffer [Wikc]. The closer the object is to the point of view, the darker it is

2.5 Depth buffer

A commonly used method in computer graphics is called *depth buffering* or *z-buffering*. This method manages depth information and helps to decide whether a part of an object is visible from a certain point of view (POV) by comparing depth information. It is first described by Wolfgang Straßer in 1974 in his PhD thesis [Str74] and is used in almost all computers nowadays. To improve the performance, graphic cards have the buffer directly implemented into the hardware, it is nonetheless often used in software.

The depth buffer is usually a 2D array, in which the cells contain the depth value of objects that have already been rendered. It is initialized with a maximum value or a background value. If a new object is being rendered, the depth values of the depth buffer are compared the new object. The object that is closer to the point of view is then being rendered. The depth buffer is then being updated with the new depth value. To visualize the buffer, the depth information can be mapped to a value between 0 and 255 to create a black and white image. This image is called a *depth image* and is able to visualize the distances of the objects, as it can be seen in figure 2.5. The closer the object is to the point of view, the darker it is in the image.

In the area of robotics, the depth buffer is commonly used to store depth information of various range sensors, such as the LiDAR sensors. In this context, the buffer is also called *range image* or *depth image*. For the Velodyne sensor, the buffer represents a spherical grid around the sensor to cover the whole 360° FOV (as shown in fig. 2.6). But still, it can be stored as a 2D array, in which



(a) Planar depth buffer (b) Spherical depth buffer

Figure 2.6: Comparison of planar[AS] and spherical depth buffer

each cell represents a laser of the sensor and stores the measured distance to the object surfaces. The height and the width of the buffer depend on the angular range and the resolution. Both can be calculated by the equation:

$$height/width = \frac{angularRange}{resolution} \tag{2.8}$$

The buffer can be created directly from the data packets of the sensor, described in 2.2. As a result, each scan creates a column of the depth buffer. This causes the spherical shape of the buffer, because the impact points of the lasers are implicitly represented by spherical coordinates. For each impact point, only the measured distance is stored. The horizontal and vertical angle can be derived from the index of the sensor and the rotational info in the data packet. Since the buffer only stores the depth information and not the different vertical angles, described in 2.2, it is important to consider the different vertical angles in further calculations. The visualization of the buffer uses the average of both angels and so the upper angles are stretched and the lower are shrinked.

The different shapes of the depth buffers have also an effect on the computation time for mapping a 3D point to the corresponding cell in the depth buffer. The planar depth buffer uses a planar projection plane, onto which all depth values are projected to. Since the mapping onto the planar projection plane requires less trigonometric functions, it is less complex as the mapping onto the sphere. The mapping of a point to the spherical depth buffer can be achieved by transforming the Cartesian coordinates into spherical coordinates, as described in section 2.4. As a result of this conversion each point is represented by its

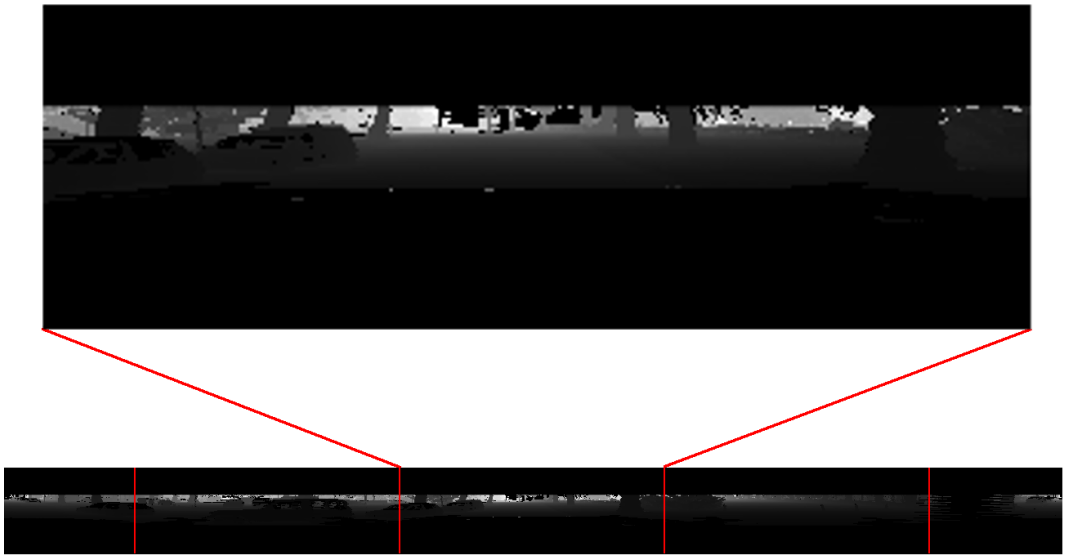


Figure 2.7: Depth image of the Velodyne sensor. The closer the object is to the point of view, the darker it is.

horizontal angle, its vertical angle and its distance to the origin point. The correct cell can be found by dividing both angles by the respective resolution.

As an alternative to the depth buffer, there is also the ray tracing algorithm. Here, the visibility of an object is decided by tracing the path from the point of view to the first obstacle. This method is further described in section **??**.

2.6 Point clouds and PCL

A point cloud is a collection of multi-dimensional points, which can store additional information, such as color or intensity. They can either be generated by sensors, such as LiDAR sensors, or by software without any underlying measurements. Figure 2.8 shows the point cloud of the HDL-64E Velodyne sensor as an example. The points are given by their Cartesian coordinates and the color represents the intensities of the light pulses.

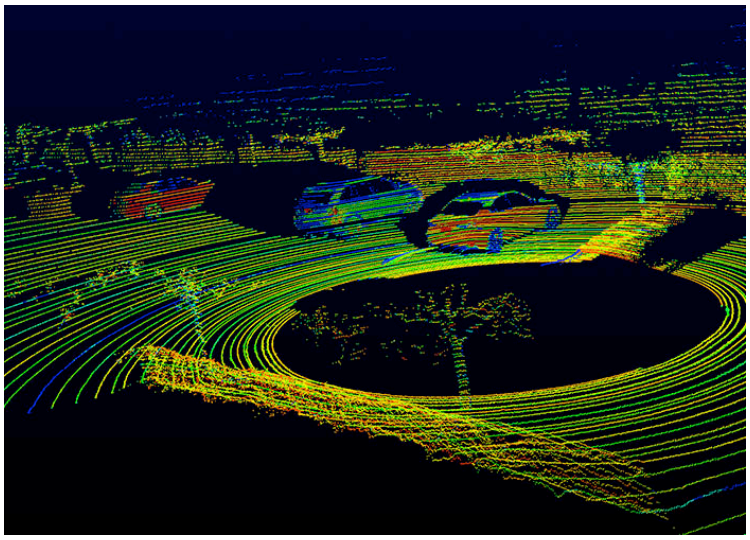


Figure 2.8: Point cloud of a Velodyne sensor. The color represents the intensity of the received light pulse.

In a 3D coordinate system, point clouds are commonly used to describe the shape of objects. If additional information is stored for each point, the point cloud becomes a four- or multi-dimensional point cloud. The clouds can be either organized or unorganized. Organized point clouds require less computation time with some algorithms, such as nearest neighbors operations [PCLa]. But, creating an organized point cloud from unorganized data can take additional time.

PCL is the commonly used library for dealing with point clouds since it provides different algorithms such as filtering, segmentation and surface reconstruction [PCLb]. These can be used, among others things, to filter out noise or to segment a point cloud in a ground plane and objects.

2.7 Octree

An octree is a hierarchical tree data structure, in which each node has exactly eight children. It is often used in 3D graphics for partitioning space because of the fast algorithms, such as collision detection and nearest neighbor search. Due to the structure, it is easy to traverse the tree for a specific node or get neighboring nodes. Each node of the octree represents a volume that can be recursively subdivided in smaller octants. It can be used for cubic volumes or

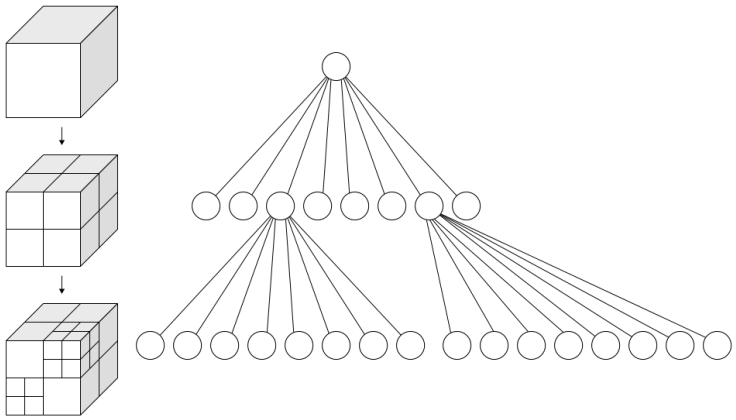


Figure 2.9: Schematic model of an octree used for cubic segmentation[Wika]

any other shape that can be subdivided in eight smaller partitions(see fig.2.9). The smallest octant is called a voxel and its size is defined by the maximum height of the octree and the size of the entire octree volume . If all eight octants of a node have the same value, the node can be collapsed to store the value of all its child nodes. This improves the performance of the octree algorithms and decreases the allocated memory. The root node of the octree contains all voxels in the immediate environment of the vehicle.

In this thesis, the octree will be used for effective search algorithms in the point cloud of occluded regions to evaluate the results.

3 Occlusion Mapping

This chapter focuses on the creation of the occlusion map and the idea behind the chosen methods. It will compare different methods and will show some advantages and disadvantages of them.

3.1 Why 3D?

There are two main aspects to be considered in this section. The first is the detection of the occlusions and the second is the data structure of the map. Both can be done in 2D and 3D and the map can be additionally created in 2.5D, which is called *elevation map*.

A commonly used method is the 2D detection of occlusions. It uses a planar field of view and is often used for occlusion handling [NB08, Tho18] because it is less complex and the computation is fast. The obstacles are represented as 2D objects. A region is evaluated as occluded if the planar line of sight to the point of view is not free. That means that any region behind an object will be classified as occluded. But when it comes to perspective, this method has disadvantages. Smaller obstacles, which can actually be overlooked, will have the same impact as higher ones for the occlusion estimation. In figure 3.1 for example, the object that is closer to the point of view could be a small obstacle which actually could be overlooked. Yet in the 2D method, the entire space behind the detected object would be interpreted as occluded.

Another disadvantage to the 2D method is the impact of closer obstacles to the size of the occluded area. The closer the obstacle is to the point of view, the bigger the area becomes. But in the case of the higher positioned sensor, smaller objects that can be overlooked will create a smaller area the closer they are. This behavior can be seen in figure 2.8, where the black areas behind

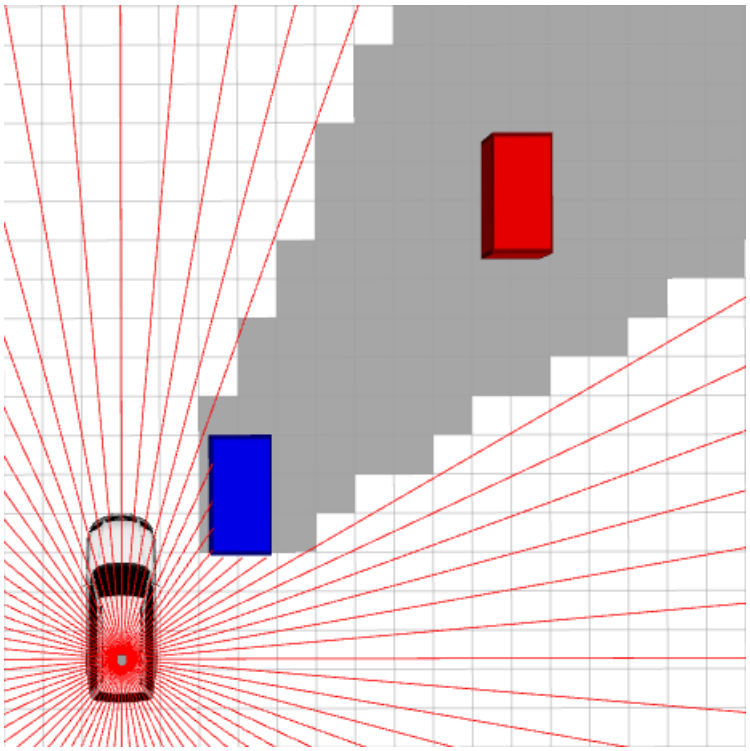


Figure 3.1: Planar occlusion estimation for a 2D map. A smaller obstacle near the sensor occludes the more distanced bigger obstacle.

the cars are the occluded areas. The farther away the obstacle is to the point of view, the more the region will be stretched. This can be resolved by a 3D detection, in which the vertical position of the sensor will also be considered for the estimation. This method will use the original line of sight in space and not the projected one. Due to this advantage, smaller objects can be overlooked and the estimations are more accurate. The calculation for 3D is therefore more complex and has a higher computation time. To improve the performance, the height of the region of interest can be reduced and the space above that height and under the ground plane can be ignored. Still, the the computation time will be higher than with the 2D method.

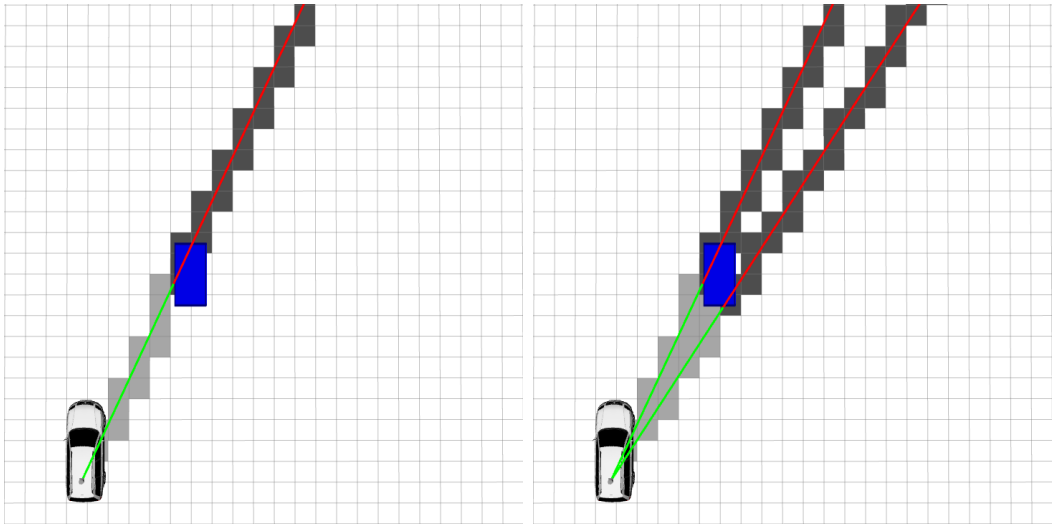
The different types of maps also have differences regarding the computation time because of their structure. The 2D map stores whether a cell on the planar grid is hidden or not. Using the elevation map (2.5D map) each cell in the grid map contains a height of the occlusion instead of a boolean value for occluded or not occluded. The advantage of the 2D and the elevation map is the lower memory usage compared to a 3D map. Depending on the height and the

resolution of the map, the 3D map provides $\frac{height}{resolution}$ times more cells, which needs to be stored. More cells also increase the time for further calculations because the search algorithms, such as nearest neighbors search, are more complex and take more computation time. On the other hand, 2D maps and elevation maps reduce the amount of information available. In case these would be important for further calculations, false results are possible. For example, if the vertical space above a cell is partially occluded. Small objects like a boom barrier or hedge can mark the whole vertical line as occluded and have the same impact as a concrete wall in the occlusion map.

Since the thesis focuses on the Velodyne sensor on the roof of the car, the 3D method is chosen for the occlusion estimation. The results of the estimation will stored in 3D point cloud and will be projected onto a 2D map, in which each cell contains the number of vertically occluded voxels.

3.2 Ray tracing vs. depth buffer

There are two common ways to compute occlusions, which are both working for 2D and 3D. One is the depth buffer method, which has already been described in section 2.5. The other method is *ray tracing*, a method that was first described in 1968 by Arthur Appel [App68] and is used to determine the visibility of objects from a specific point of view. Different methods have been developed to improve the performance, like [Gla84] and [AW15]. The basic idea of this technique is to follow rays from the point of view to different directions and find intersections with the scene. A scene can be any environment, like a computer generated world or the environment of a vehicle. Figure 3.2(a) shows an example of the ray tracing method. The first intersection with the scene is the visible point(impact point). Any other intersection of the same ray is hidden by the first one. The method follows each ray from the origin until an intersection has been found (green ray in example) or a maximum distance has been reached. Each cell that has been intersected to that point is free(gray cells in example). To mark the occlusions the ray can be extended to the maximum distance (red ray in example) and every cell that is intersected by the extended ray can be marked as occluded(black cells in example).



(a) Ray tracing for one ray

(b) Ray tracing for two rays with increasing space between rays

Figure 3.2: Ray tracing examples with one and two rays. The green ray is the visible part and the red ray is the extended part, which is occluded. Gray cells are visited by the ray tracing algorithm and are free. The Black cells are occluded

When using the ray tracing method with the Velodyne sensor, it is possible to generate the rays from the Velodyne point cloud. Each point in the point cloud corresponds to one impact point, and the origin is always the sensor origin. The point cloud of the Velodyne HDL-64E contains the measurements for one rotation, which produces ~142000 points. To consider all measurements for the occlusion map the algorithm would need to trace the same amount of rays. Therefore, this algorithm is often used for preprocessing scenes with high quality. The complexity can be shown by a quick estimation for the 2D case. The number of intersected cells for each ray depends on the orientation related to the grid. A ray intersects the fewest cells if it lies directly on the diagonals of the cells. In that case the ray uses then the maximum space of each cell. Using this simplification, the number of intersected cells is:

$$numberOfIntersectedCells = \frac{lengthOfRay}{\sqrt{MapResolution^2 + MapResolution^2}} \quad (3.1)$$

For a maximum range of 30 m and a resolution of 0.5 m, each ray intersects at least ~43 cells and at the most 60 cells. Using the ray tracing method for a sensor with 350 rays would require to iterate over 15050 cells. The number of

cells in a 60 m × 60 m 2D map is 14400. With an increasing number of rays, the number of cells that are visited multiple times also increases.

To improve the performance of the ray tracing algorithm, downsampling the rays is a suitable method. This can be achieved by filtering out close neighbored impact points. As a result, just one representant point of the neighboring points will be considered for further computations. However, this increases another drawback of this method. The less rays are being used for the ray tracing, the bigger the space between the rays will become(see figure 3.2(b)). This also affects the occlusion map, since the space between the rays is increasing in relation to the distance to the origin. Therefore, there is a need to interpolate this space.

Additionally, the computation time can be further improved, by ignoring the free space and just mark the occlusions. With that, the ray tracing starts at the first intersection which is exactly the impact point that is given by the sensor. It is sufficient to follow the extended part of the ray through the cells and mark the intersected cells as occluded.

Although even with these improvements, the complexity of the ray tracing method is affected by the number of rays. With an increasing number of rays, it will be more and more suitable to iterate over all cells of the map and check if the cell is occluded or not. This can be done with the depth buffer.

An alternative to the ray tracing method uses the depth buffer mentioned in 2.5. In the 2D case, the buffer can be seen as a ring around the origin point. The buffer can be stored as an one dimensional array. The ring is equally divided into cells, which represent the outgoing lasers by their depth measurements. To find out whether a point of the map is hidden or not, the algorithm compares the distance of the point to the depth value in the corresponding cell. The cell can be derived from the angular values of the spherical representation of the point. If the value of the depth buffer is smaller than the distance of the point, then the point is hidden by an obstacle. The advantage of this method is that if the depth buffer is already given, then the complexity is independent to the number of rays. It then only depends on the resolution of the map. The higher the resolution, the more points need to be compared to the depth buffer.

Analog to 2D, the depth buffer for 3D can be seen as a sphere around the sensor. It is rastered into cells, which contain the depth values of the laser measurements. The assignment of a cell in the depth buffer to a 3D point works similar to the 2D case. In addition to the horizontal angle, there is now a vertical one. This buffer can be stored in a 2D array, one dimension for each angle. The depth buffer itself is already an implicitly segmentation of the scene into free, unknown and occluded space. For example, in 2D each cell in the depth buffer represents one circular sector around the sensor origin. The angular width of each sector is given by the number of cells in the buffer and the angular range of the sensor. The circular sector is then further segmented into free and occluded space and the value in the cell gives the distance of the separation line to the origin.

A 3D voxel grid with a vertical height of 2 m and and resolution of 0.5 m has four horizontal layers of voxels. The number of voxels that need to be checked in a 60 m $\times$ 60 m voxel grid is 57600. As already mentioned, the Velodyne sensor creates ~142000 rays. Even by ignoring some of the rays, the ray tracing method will iterate over some voxels multiple times. With that, it would need to iterate over more than 57600 voxels to fill the entire voxel grid. But the iteration over all voxels is more expensive with the depth buffer method because of the trigonometric functions. Whereas the ray tracing method just uses neighborhood relations to find the corresponding voxels to each measurement of the sensor. With that, the number of visited voxels of the ray tracing algorithm cannot be directly compared to the visited voxels of the depth buffer algorithm because the costs per voxel are not equal. The trigonometric functions have also different costs, depending on the compiler and the system.

All in all the depth buffer algorithm is more suitable for the real time occlusion estimation, since every voxel in the region of interest needs to be checked only once. Instead of iterating multiple times over partially occluded voxels, as it is done by the ray tracing method, the depth buffer method uses the closest ray for each voxel.

3.3 Implementation

The first task of this occlusion mapping method is to find the corresponding cell in the depth buffer for each voxel in the region of interest. This assignment can be reused afterwards, if the transformation from the map frame to the sensor frame is not changing.

The corresponding cell is found by basic trigonometric calculations (see alg. 1) and some coordinate transformations. First, the coordinate of the voxel center is transformed into the coordinate system of the sensor. After this transformation, the Cartesian coordinate of the center point is transformed into the spherical representation, as described in 2.4. The position of the cell in the depth buffer can then be derived from the angular part of the representation. The functions to find the horizontal and vertical indices depend on the range

Algorithm 1 : Find corresponding cell in depth buffer for each voxel

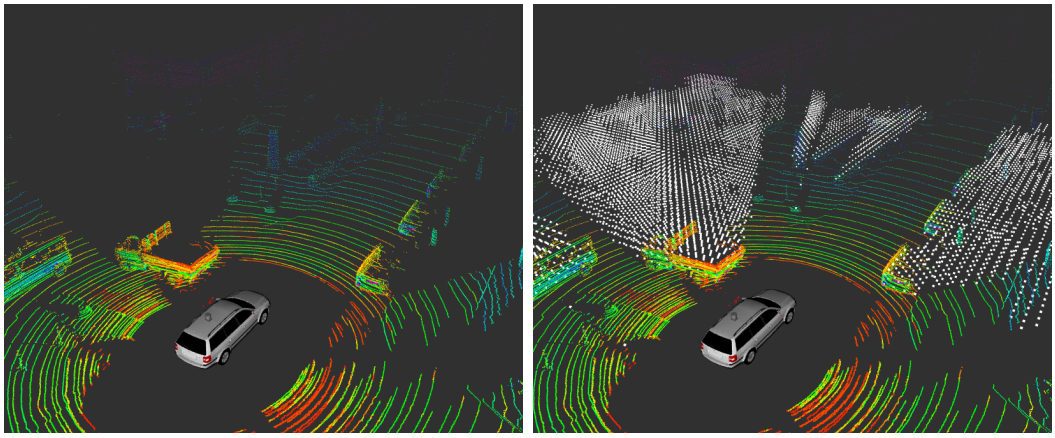
```
foreach Voxel v ∈ regionOfInterest do
    c = v.centerPoint();
    t := transformPointToSensorFrame(c);
    r := √(t.x² + t.y² + t.z²);
    θ := arctan(t.x/t.y);
    φ := arcsin(t.z/r);
    hIdx := depthBuffer.getHorizontalIdx(θ);
    vIdx := depthBuffer.getVerticalIdx(φ);
    if depthbuffer.indicesInRange(hIdx,vIdx) then
        saveAssignment(c,hIdx,vIdx,r);
```

of the sensor and the angular resolution. Since both are properties of the depth buffer, it can also provide the transformation of angles to the corresponding indices. The indices can be calculated with:

$$angleIdx = \frac{maxAngle - angle}{resolution} \tag{3.2}$$

In case of the Velodyne sensor there are two vertical resolutions. One for the upper 32 lasers and one for the lower 32 lasers.

The second task iterates over the assigned center points and compares the distances of the points to the origin with the values in the depth buffer cells



(a) Example point cloud

(b) Example point cloud and centers of occluded voxels

Figure 3.3: Result of occlusion estimation

(see alg. 2). If the distance of a point is bigger than the value in the depth buffer, the voxel is marked as occluded, else it is marked as free.

Algorithm 2 : Iterate over all assigned voxels and check for visibility

```
depthBuffer.insert(velodyneDepthImage);
foreach AssignedVoxel v ∈ MappedVoxelList do
    depthValue = depthBuffer.getValue(v.vIdx,v.hIdx);
    if depthValue!=0 and depthValue < v.r then
        markVoxelAsOccluded(v.c);
```

The result of this section is a map of all occluded voxels. This map can be a normal point cloud of all center points or a 2D projection. Different map formats will be discussed in section 3.5. In figure 3.3, the point cloud of the occlusions is shown additionally to the sensor point cloud. If needed, the algorithm can be extended to additionally mark all *free* or *unknown* voxels. Therefore, voxels with indices that are out of depth buffer range are marked as *unknown*. The same with voxels, whose corresponding depth buffer value is 0. If the depth buffer value is bigger than the voxel distance, it can be marked as *free*.

The presented method uses two loops, one for creating the assignment of the voxels to the depth buffer cells and one for the visibility checks. They can also be combined in one loop, so that the assigned voxels are checked directly for

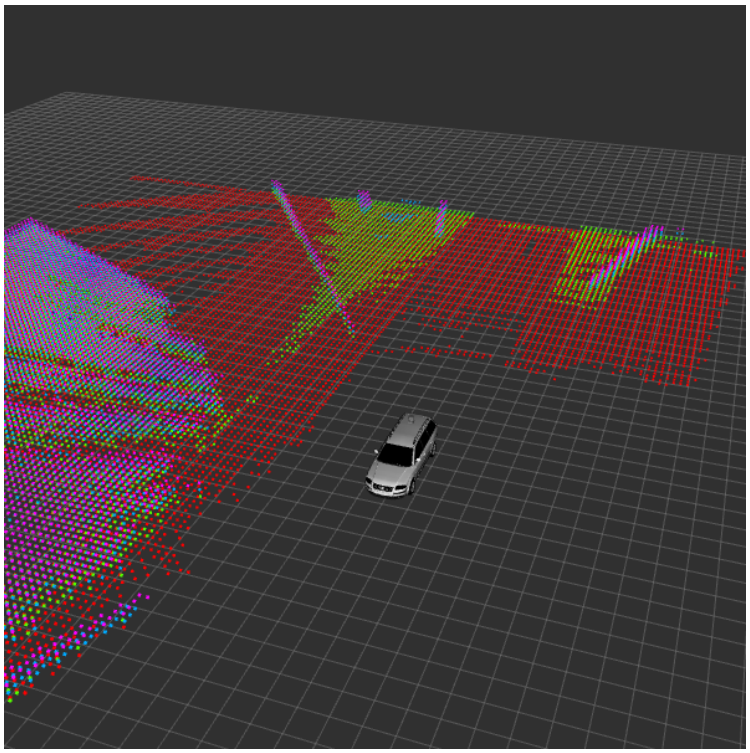


Figure 3.4: Occlusion caused by ground plane. The color represents the height of the voxel centers in relation to the base_link frame

visibility. But this will reduce the readability and for static transformations, the assignment needs to be calculated only once.

3.4 Ground filter

With the previously described depth buffer method, the occlusions can now be mapped in 3D. But there are some occlusions marked, which are superfluous(see fig. 3.4). Since the orientation of the ground plane is constantly changing in relation to the vehicle, it can happen that parts of the region of interest are under the ground plane. The algorithm will declare such voxels as occluded. To prevent these occlusions it is necessary to decide whether a point is hidden by the ground plane or not.

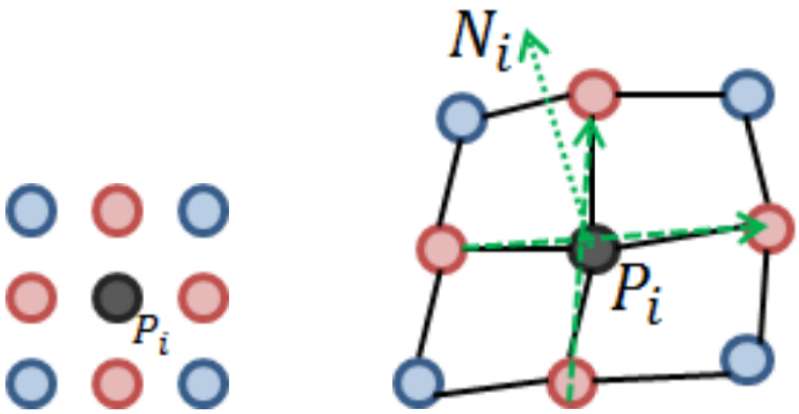
One approach uses the height of the voxel itself to decide whether it is under the ground plane or not. But often no precise equation of the ground plane is available and so the height in relation to the real ground cannot be

distinguished. As an alternative the coordinate frame that is attached to the mobile base of the vehicle (base_link) could be used and everything under the base_link ground plane is excluded. But the measurements of the Velodyne sensor are not exact and the ground is not perfectly planar. Therefore, it is necessary to define an threshold under which every voxel will be considered as a *ground voxel*. Increasing the threshold will reduce the number of occlusions that are caused by the real ground plane but will also cut of the lower parts of the normal occlusions. Instead of this method, the minimal height of the occlusion map can just be increased. The main disadvantage of this approach is that it still cannot distinguish between space that is hidden by an object or by the ground plane.

Another approach uses the values in the depth image to calculate the height of the impact points. First the spherical representation of the impact point needs to be calculated with the inverted equation 3.2. Afterwards, the point can be transformed into the Cartesian representation and then further into the base_link frame. The transformed coordinate contains the height of the impact point in relation to the base_link frame. With this information the algorithm can now give an estimation about whether a voxel is hidden by an object or by the ground. The improvement to the first approach is that the lower parts of object occlusions are still marked. But as with the first method inaccurate measurements and a pitched ground plane will also decrease the reliability of this method.

Especially with a changing ground plane orientation, both described approaches will not work correctly. Since the pitch of the ground plane relative to the vehicle changes constantly, a dynamic approach that fits the ground plane to the current measurements is desirable. In [YKP+13], the author describes a fast method for estimating the ground plane considering a dynamic orientation. In this method, the SVD (Singular Value Decomposition) is used in combination with a least squares method on prefiltered points to estimate the ground plane coefficients. The prefiltered points are collected from an ordered point cloud by the following method.

For each point P_i of the ordered point cloud, the algorithm estimates its normal vector N_i by using the surrounding points, as shown in figure 3.5. The surrounding points are called up, down, left and right. The cross product of



(a) Points in ordered point cloud (b) Points in 3D space

Figure 3.5: Computation of local normal vectors [YKP+13]

the vectors up -> down and left -> right is the normal vector of the the Point P_i. This normal vector is an estimation of the local plane orientation at a specific point P_i, which can be used for a first filtering. One characteristic of a ground plane point is that the surface normal is almost orthogonal to the base_link ground plane. For the first filtering step, the normal vector of each point can be compared to the normal vector N* of the X-Y plane of the base_link coordinate system (0.0,0.0,1.0). The normal vector of the ground plane should differ from the one of the X-Y plane only by a few degrees, since the vehicle moves parallel to the ground. The angle between two vectors can be calculated by the equation 3.3.

$$cos(\alpha) = \frac{\vec{a} * \vec{b}}{||\vec{a}|| * ||\vec{b}||} \tag{3.3}$$

$$cos(\alpha) = \hat{a} * \hat{b} \tag{3.4}$$

$$\hat{a} * \hat{b} <= \gamma \tag{3.5}$$

Using the unity vector of $\vec{a}$ and $\vec{b}$ and defining a threshold γ, by calculating the cosine of a specified angle, 3.5 can be used for filtering out local planes that have a high slope. The remaining points, with a lower local slope, can then

further be grouped so that each group represents one plane. The characteristics for each group are:

$$|\hat{N}_i * \hat{N}_j| > 1 - \epsilon_1 \tag{3.6}$$

$$|\frac{P_i - P_j}{||P_i - P_j||} * \hat{N}_i| < 0 - \epsilon_2 \tag{3.7}$$

Two points P_i and P_j are in one group if both equations are fulfilled. Again, both equations use the equation 3.5 to compare the orientation of the normal vectors. Equation 3.6 checks for similarity of the normals. If two normal vectors are equal, then the dot product is cos(0°), which is 1. Because of measuring errors and a non planar ground plane, the normal vectors of the points differ a few degrees, even if they lie on the same plane. The threshold ϵ_1 defines the allowed range of these differences. With a threshold of 15°, ϵ_1 would be $cos(0°) - cos(15°) \approx 0.034$. After using equation 3.6 to group the points regarding their normal vector, the points inside a group can still lie on parallel planes. Equation 3.7 checks whether the line between the points P_i and P_j is perpendicular to the normal vector of the group or not. If both points lie on the same plane, the vector from P_i to P_j should be perpendicular to the groups normal. As before, the equation 3.5 is used to check the similarity between the vectors. But this time the result for two perpendicular vectors should be $cos(90°) = 0$, with a threshold of ϵ_2.

To improve the computation time a set of uniformly distributed sample points is used, instead of the entire point cloud. After these steps, all points are grouped by their surface normals. The group that contains the most points is supposed to be the ground plane, for which the coefficients can be found by using the least squares method. This method finds the best fitting ground plane to a set of points by minimizing the sum of all orthogonal distances between the points and the estimated plane. The orthogonal distance δ of a point to a plane can be calculated with equation 3.8.

$$a * p_x + b * p_y + c * p_z + d = \delta \tag{3.8}$$

$$\begin{bmatrix} p_{1,x} & p_{1,y} & p_{1,z} & 1 \\ p_{2,x} & p_{2,y} & p_{2,z} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ p_{n,x} & p_{n,y} & p_{n,z} & 1 \end{bmatrix} * \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \Delta \tag{3.9}$$

The coefficients a, b, c are the components of the plane normal vector and d is the orthogonal distance of the plane to the origin point (0,0,0). Applying this equation on a point(x,y,z) results in the relative distance of the point to the plane. A value > 0 or < 0 indicates that the point lies above or under the plane relative to the origin. The value 0 indicates that a point is directly on the plane. To calculate the distance of every point to the plane, the points can be vertically aligned as a matrix and the ground plane coefficients as a vector (see 3.9). The distance of all points to the plane can be evaluated with the squared Euclidean distance:

$$||\Delta||_2^2 = \delta_1^2 + \delta_2^2 + ... + \delta_n^2 \tag{3.10}$$

This equation sums up all squared distances and can be used for the minimization problem. It can be written as 3.11, in which A is the point matrix and $\vec{x}$ is the vector containing the plane coefficients.

$$\min_{\vec{x}} ||A\vec{x}||_2^2 \tag{3.11}$$

But since $A\vec{x}$ is a set of homogeneous linear equations, $\vec{x} = \vec{0}$ would be a correct but useless solution for the minimization problem. Using the constraint that c=1.0 would result in an inhomogeneous linear equation 3.12. This would normally be a problem, because the z-component of the plane equation could be 0. But, in case of the almost vertical normal vectors of the filtered points, the z-component should never be 0. By using the constraint, the estimated plane normal will not be normalized, since the length of the normal vector will be > 1.0. This can be fixed afterwards by normalizing the vector. The matrix notation (eq. 3.13) and the minimization problem (eq. 3.14) have also changed accordingly. Both have now an inhomogenous part $\vec{z}$.

$$a * p_x + b * p_y * d + p_z = \delta \tag{3.12}$$

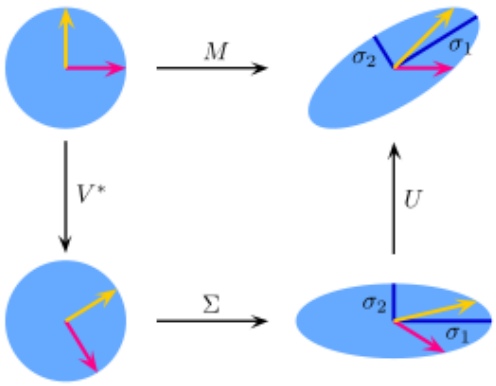


Figure 3.6: Schematic model of the singular value decomposition of a 2×2 matrix[Wikb]

$$\begin{bmatrix} p_{1,x} & p_{1,y} & 1 & \\ p_{2,x} & p_{2,y} & 1 & \\ \vdots & \vdots & \vdots & \vdots \\ p_{n,x} & p_{n,y} & 1 & \end{bmatrix} * \begin{bmatrix} a \\ b \\ d \end{bmatrix} + \begin{bmatrix} p_{1,z} \\ p_{2,z} \\ \vdots \\ p_{n,z} \end{bmatrix} = \Delta \tag{3.13}$$

$$\min_{\vec{x}} ||A\vec{x} + \vec{z}||_2^2 \tag{3.14}$$

With this constraint, $\vec{x} = \vec{0}$ is no longer a solution for the minimization problem and it can be solved with the singular value decomposition.

3.4.1 Singular value decomposition

This section gives a short overview of the singular value decomposition, which is used for the ground plane estimation. It is a commonly used algorithm for approximation, compression and data reduction of matrices. The main idea is to decompose a $m \times n$ matrix M into its constituent parts and simplify it. Figure 3.6 shows an intuitive example of the factorization of a 2×2 Matrix M into two unitary matrices U and V and the diagonal matrix Σ.

The columns of U are called the left singular vectors and they are the orthogonal eigenvectors of $M * M^T$. Σ is a $m \times n$ diagonal matrix and contains the so called singular values. It has the following form:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & & \dots & 0 \\ 0 & \sigma_2 & 0 & & \\ & & \ddots & & \\ & & \ddots \quad \sigma_r & \ddots & \vdots \\ \vdots & & & 0 & \\ & & & & \ddots \\ 0 & & \dots & & 0 \end{bmatrix} \tag{3.15}$$

with $\sigma_1 >= \sigma_2 >= ... >= \sigma_r >= 0$, where r is the rank of the matrix M and $\sigma_1, \sigma_2 ... \sigma_r$ are the square roots of the eigenvalues of $M^T * M$. The columns of V are called the right singular vectors and they are the orthogonal eigenvectors of $M^T * M$.

3.4.2 Using SVD for the least squares method

With the singular value decomposition, the minimization problem of 3.14 can be reduced and solved. First, by using the properties of an unitary matrix, it can be written as:

$$||A\vec{x} + \vec{z}||_2^2 = ||U^T(A\vec{x} + \vec{z})||_2^2 \tag{3.16}$$

The right part uses the fact that the Euclidean norm of a vector is not changing after it is multiplied by an unitary matrix. After replacing matrix A with the singular value decomposition, the minimization problem can be further simplified (3.17).

$$\begin{aligned} ||U^T(A\vec{x}+\vec{z})||_2^2 &= ||U^T(U\Sigma V^T\vec{x}+\vec{z})||_2^2 \\ &= ||\Sigma \underbrace{V^T\vec{x}}_{\text{substitue with } \vec{\mu}} + U^T\vec{z}||_2^2 \\ &= ||\Sigma\vec{\mu}+U^T\vec{z}||_2^2 \end{aligned} \tag{3.17}$$

Since Σ is a diagonal matrix, where only the first r diagonal cells are not 0 and therefore $\sigma_i * \vec{\mu}_i = 0$ for i>r :

$$||U^T(A\vec{x}+\vec{z})||_2^2 = \sum_{i=1}^{r}(\sigma_i * \vec{\mu}_i + u_i^T * \vec{z})^2 + \sum_{i=r+1}^{m}(u_i^T * \vec{z})^2 \tag{3.18}$$

To minimize the result depending on $\vec{x}$, $\vec{\mu}$ has the form:

$$\vec{\mu_i} = \begin{cases} -\frac{u_i^T * \vec{z}}{\sigma_i} & \text{if } i = 1...r \\ arbitrary & \text{if } i = r+1...n \end{cases}$$

and the minimized result is:

$$min_{\vec{x}}||U^T(A\vec{x}+\vec{z})||_2^2 = \sum_{i=r+1}^{m}(u_i^T * \vec{z})^2 \tag{3.19}$$

Since $\vec{\mu_i}$ can be arbitrary for $i = r+1..n$, a vector $\vec{\mu^*}$ can be defined as:

$$\vec{\mu_i^*} = \begin{cases} -\frac{u_i^T * \vec{z}}{\sigma_i} & \text{if } i = 1...r \\ 0 & \text{if } i = r+1...n \end{cases}$$

The solution for the minimization problem is:

$$\begin{aligned} \vec{x^*} &= V * \vec{\mu^*} \\ &= \sum_{i=1}^{r} v_i * \vec{\mu_i^*} \\ &= -\sum_{i=1}^{r} v_i * \frac{u_i^T * \vec{z}}{\sigma_i} \end{aligned} \tag{3.20}$$

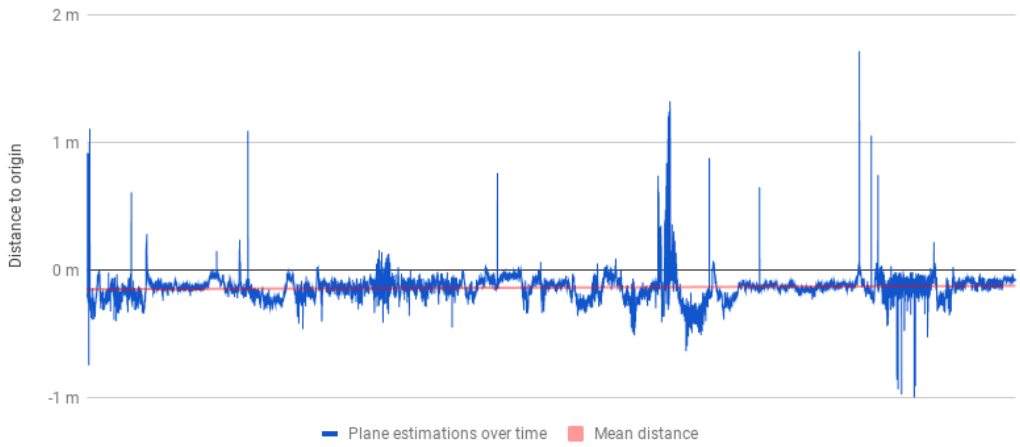


Figure 3.7: Orthogonal distances of estimated planes to origin (0,0,0). Average distance for ~11,000 iterations was -0,1477190043 m. 50 outliers were above an absolute distance of 0.5 m

The resulting vector $\vec{x^*}$ contains the optimized coefficients for the plane equation. The distance of a point to the ground plane can then be calculated by inserting its coordinates into the equation. This is used to determine whether an impact point of the laser sensor is close to ground plane or beneath it.

3.4.3 Evaluation of the ground filter

One of the values to quantify the accuracy of the ground plane estimation is the orthogonal distance of the origin point(0,0,0) to the estimated plane. The base_link coordinate system of the used vehicle is configured in the way that the origin point lies on the front axis. Taking the calibration offset and the movement of the car into account, the orthogonal distance of the estimated ground plane to the origin should be in the range of -0.1 to -0.3 m. Figure 3.7 shows the orthogonal distances of the plane estimations during a 18 min drive through Berlin. The average distance was -0,1477190043 m with a total of ~11,000 estimations and 50 estimations were above an absolute distance of 0.5 m. The outliers can occur in situations, in which multiple horizontal planes on different heights exist and most of the sample points lie on a different plane then the ground plane. To overcome such situations, the plane estimation can be repeatedly executed with the next smaller point group until a sufficient plane has been found or the remaining point groups fall below a specified size.

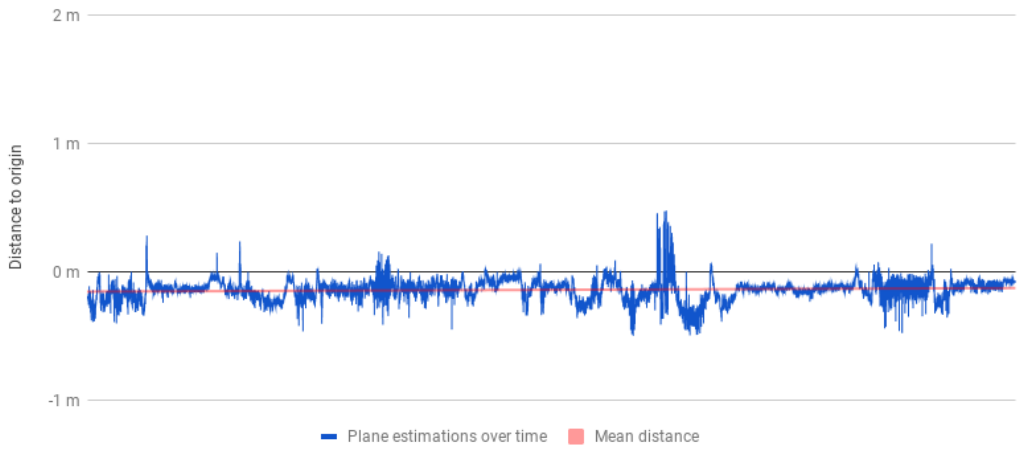
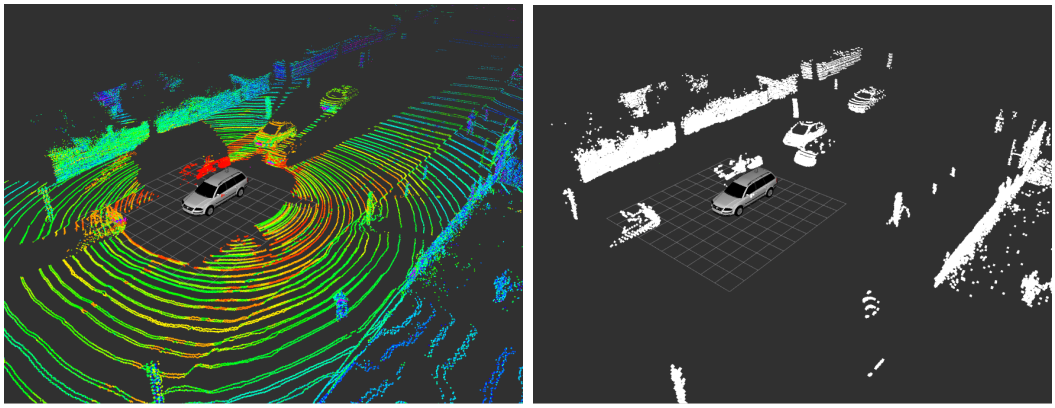


Figure 3.8: Orthogonal distances of filtered estimated planes to origin (0,0,0). Average distance for ~11,000 iterations is 0,1444376775 m. The estimation failed in 6 iterations

Figure 3.8 shows the results after removing the outliers. The average absolute distance decreases to 0,1444376775 m. With a minimal point group size of 50 points, the plane estimation failed to estimate a ground plane in 6 out of ~11,000 iterations. To decrease the number of fails, the minimal number of points in a point group can be further decreased but this will also increase the risk of choosing a wrong plane. The maximum absolute distance to the origin can also be decreased, to get better results, but since the real distance can vary due to external factors, the number of fails and the computation time can increase. In figure 3.9, the plane estimation is used to filter out every point of the point cloud whose orthogonal distance to the ground plane is less than 0.3 m.

Since the point samples for the plane estimation are uniformly distributed in the point set, but not in space, it is more likely that the algorithm adapts the closest ground plane orientation to the vehicle. The distribution in space increases with the distance to the sensor. Therefore, the points farther away are more likely to have different normal vectors. This decreases the probability of adapting the ground plane orientation of farther distanced points.

The figures in 3.10 show two different interesting scenarios. In Scenario 3.10(a) the vehicle is leaving the Motorway and the lane has split. Both lanes have different ground plane orientations. Some ground points at the edge of the



(a) Original velodyne point cloud (b) Filtered point cloud

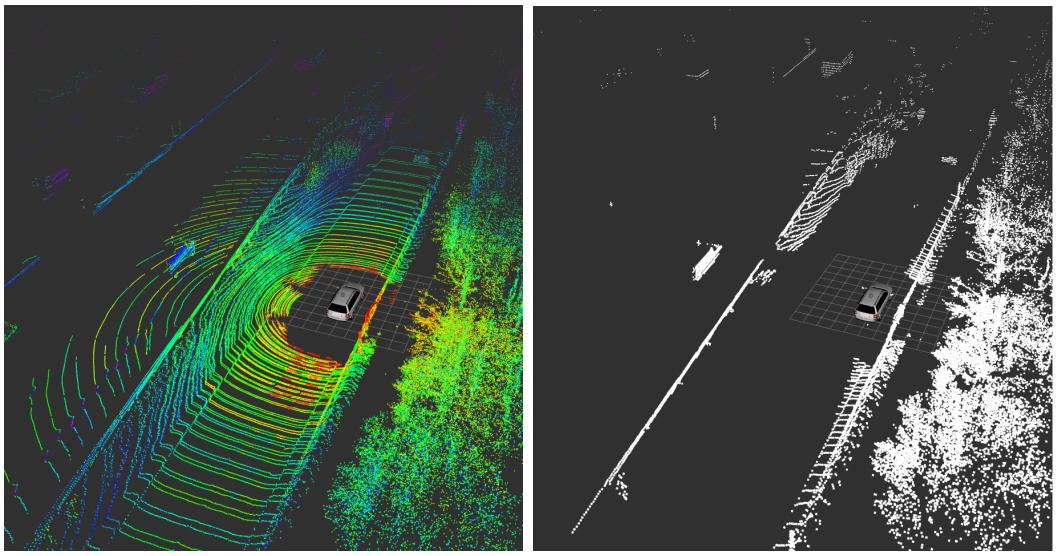
Figure 3.9: Comparison of the original Velodyne point cloud and the filtered point cloud.

lanes are still visible, but the most points of the lanes were removed. Scenario 3.10(b) shows the situation where the vehicle is surrounded by other vehicles and so the direct ground plane is mostly occluded. Still, with ~1800 sample points, the method found a group with 68 ground points to estimate the ground plane.

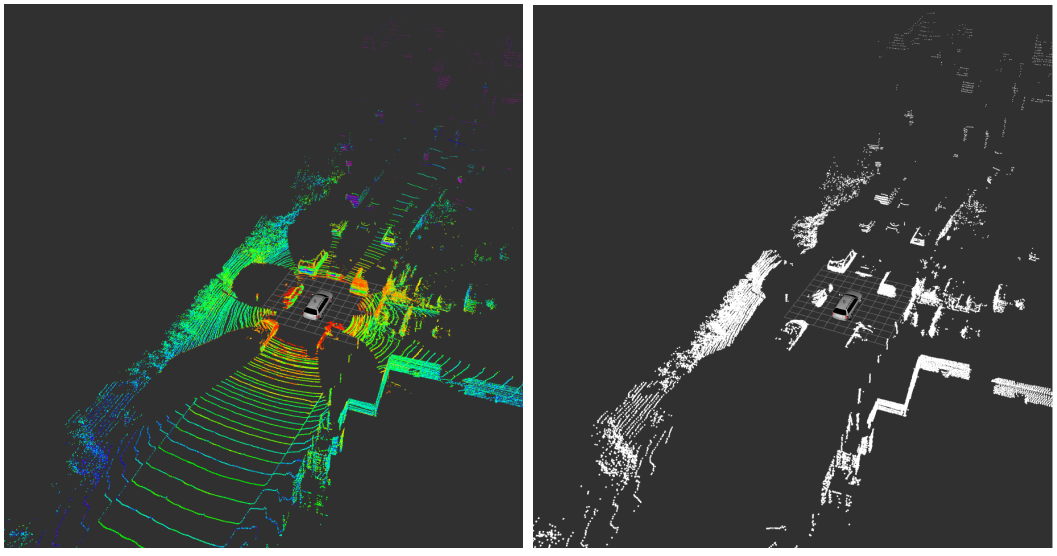
In figure 3.11, the ground plane filter is applied on the occlusions estimation. Impact points, which have a distance < 0.3 m to the estimated ground plane will be classified as ground points and their occlusions are ignored.

3.5 Map storage

This section describes the storage of the occluded areas. Three important aspects are the creation of the map, the serialization and the reusability. Each aspect needs to be considered when selecting the right map structure. A Fast creation and serialization, for example, are useless in a realtime system if it takes too much time to deserialize and use the map. The deserialization and the usage should also happen in realtime. So, each aspect needs to be considered in terms of their computation time. The best way is to create an already serialized map in realtime so it can be directly forwarded to all requesting components, which can deserialize it in realtime.

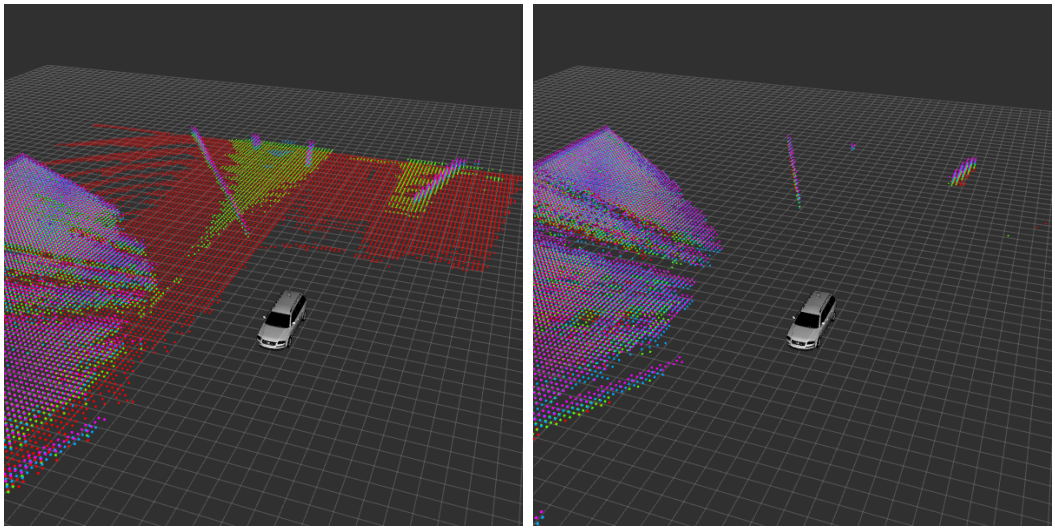


(a) Motorway exit with different ground plane orientations



(b) Big parts of the ground plane are occluded

Figure 3.10: Ground plane filter applied on different point clouds.



(a) Occlusions caused by the ground plane (b) Ground plane occlusions were removed

Figure 3.11: Before and after removing the ground plane occlusions. The color represents the height of the voxel centers in relation to the base_link frame.

One way of handling the occlusions is to store each center point of the occluded voxels as a point in a point cloud. The points can be added directly and require less computation time. No complex computations are necessary to forward the data, since a point cloud can easily be serialized as a list of points. Although the deserialization is not complex, using the data for subsequent computations like nearest neighbor search is not efficient. Other data structures, like octrees, would be more efficient in this case. But since the data is serialized as a point cloud, it can be converted into an octree. The points can also be saved directly into octrees during the creation of the occlusion map, but this requires a tree traversal for each inserted point, which is too expensive for a realtime algorithm. The octree would also have to be serialized and then deserialized afterwards for further computations.

Another way to represent the data is to project the occluded voxels onto a 2D grid map. Each cell of the grid map stores a value in relation to the number of hidden voxels in the vertical line. The creation of the map can be done during the iterations by increasing the value of the corresponding cell in the 2D grid map. The serialization and deserialization of the map is not expensive, since the map can be stored as a 2D array. And since it is a 2D map, the computation time of using search algorithms in this map is also lower than with a 3D map.

This map can also be stored as a binary map where each cell is either occluded or not, depending on how many of the voxels in the vertical line are hidden or not.

3.6 Regions of interest

A region of interest is the region that needs to be checked by the occlusion algorithm. The size of it influences the computation time. The bigger the region, the more sample points need to be checked for visibility. To decrease the number of computations, the region of interest can be reduced to temporary interesting regions. Such regions may be, for example, a pedestrian crosswalk or a driving lane. The pedestrian crosswalk can be represented with a bounding box that includes important areas, which need to be considered for a driving maneuver. The lanes, however, need a more complex representation.

3.6.1 Bounding boxes

The previous examples for the occlusion estimations in this thesis already used a bounding box, which included the environment of the car up to a specific range. This bounding box was placed in the local base_link frame and therefore, the transformation to the sensor frame was static. This is beneficial for creating an occlusion map of the entire environment. But bounding boxes can also be placed in different coordinate systems. A bounding box for a pedestrian crosswalk can be specified in the global map coordinate system. The global position of the bounding box will not change over time, but the coordinates related to the sensor will. This requires a new computation of the assignment of the voxels to the corresponding depth buffer cells. To prevent superfluous computation time, they can be added or removed as needed during runtime.

In figure 3.12 two bounding boxes were used for pedestrian crosswalks. In this experiment, the free voxels were also marked to visualize the bounding boxes. The results of the occlusion estimation can be forwarded to the planing

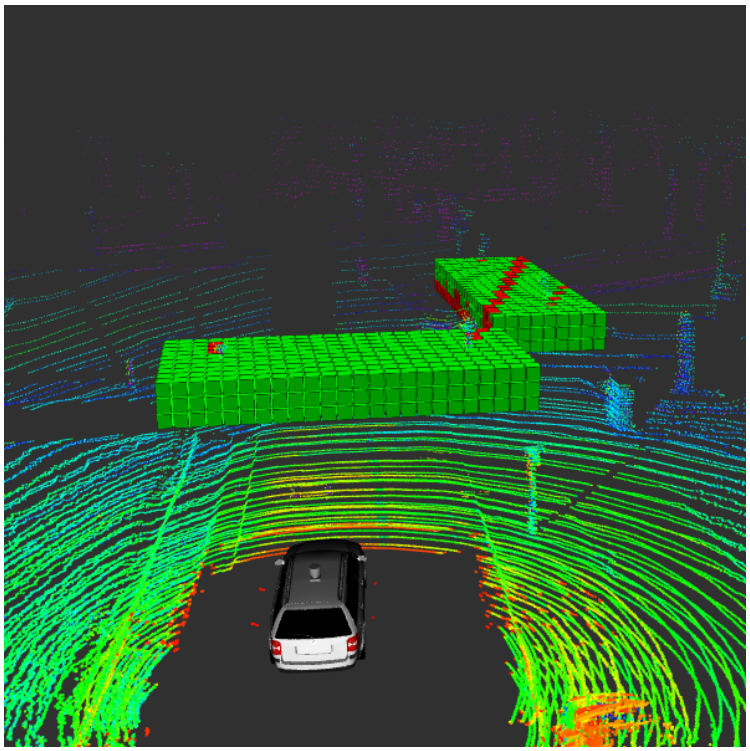
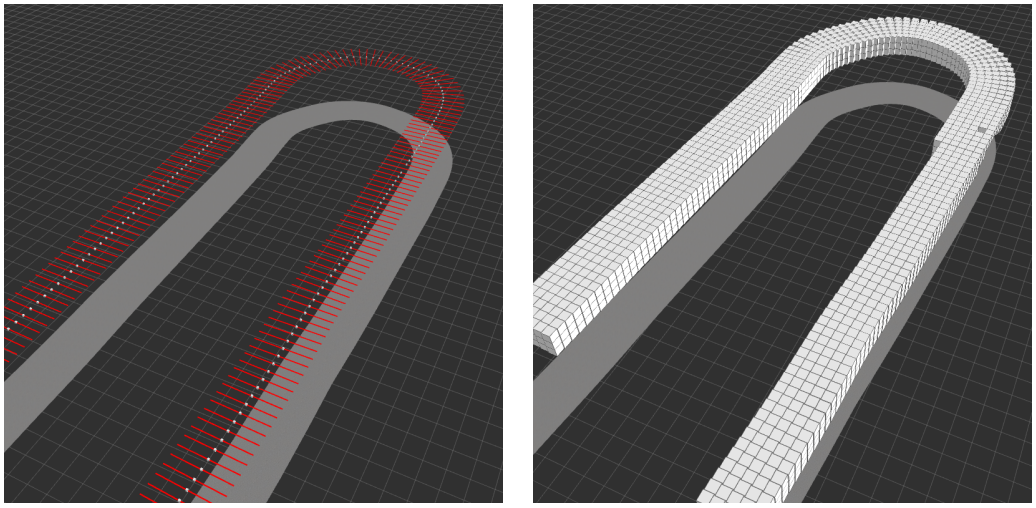


Figure 3.12: Bounding box example for smaller regions with a resolution of 0.5 m

components, as already described in 3.5, in a point cloud or projected onto a 2D grid.

3.6.2 Lanes

Additionally to the bounding boxes, driving maneuvers can also require to check driving lanes for occlusions. In a right turn maneuver for example, it is necessary to observe the pedestrian crosswalk, the bike lane and also all lanes that the vehicle crosses. To check driving lanes for occlusions, the region of interest can be limited to the lane space. Lanes are often represented as splines, which are functions that are piecewise defined by polynomials. To create a voxel grid inside the lane space, the voxels can be aligned along the splines, without considering the map frame orientation. Therefore, equally distanced way points are first created on the spline, where the distance is given by the voxel size. For each spline point, the center points of the voxels are created perpendicular to the spline. With that, the positions and orientations of the



(a) Spline normals on each spline point (b) Voxel grid along a spline

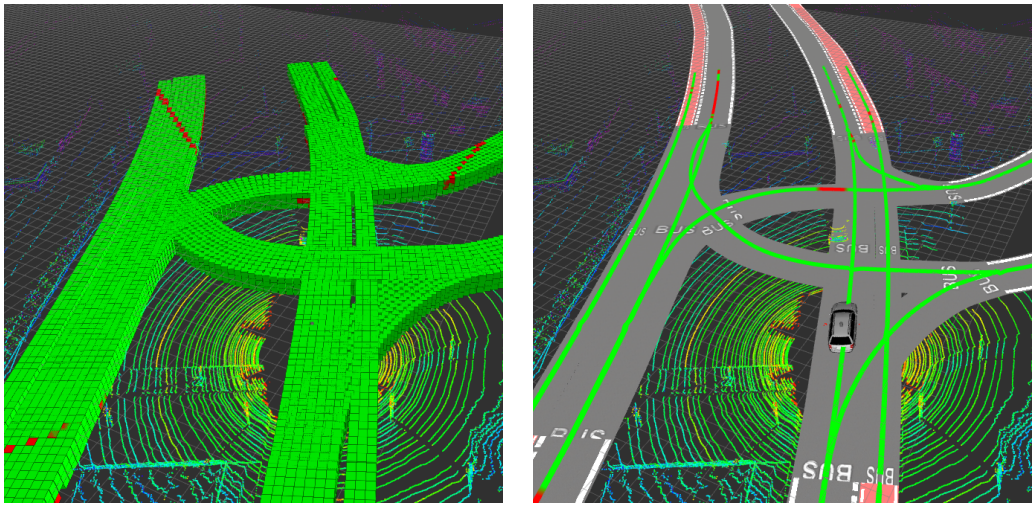
Figure 3.13: Voxel grid generation along a spline

voxels depend on the gradient of the corresponding spline point. Figure 3.13(a) shows the way points and the horizontal perpendicular lines to the spline. The number of horizontal voxels depend on the width of the lane and the number of vertical voxels needs to be specified. In Figure 3.13(b), a height of 1.5 m and a resolution of 0.5 m are used. The voxels can now be checked for occlusions with the same algorithm as before.

The result of the occlusion estimation for lanes on a junction with multiple lanes is shown in figure 3.14(a). To provide the estimations to the planing algorithms, the information about occlusions can be stored directly to each spline spline point, as it is shown in 3.14(b)). A threshold is used to decide whether a spline point is classified as occluded or not. It also shows the underlying map, which also contains bike lanes. Since they are handled as normal lanes, they are also checked for occlusions. Using this representation of the lane occlusions, a planing algorithm can check directly which part of a lane is classified as occluded.

3.6.3 Conclusion

The usage of smaller regions requires less computation time for checking the visibility of the voxels. But on the hand, the computation of the corresponding



(a) Voxel grid of multiple lanes (b) Lane occlusions

Figure 3.14: Lane occlusions for multiple lanes on a junction

cells in the depth buffer needs to be calculated in each iteration, since the position of the region of interest moves in relation to the sensor. It also often requires additional functions to find the interesting regions. The lane based method, for example, first needs to find all lane points in a given range and these points need to be updated permanently. The approach in this thesis consecutively extends or shortens the lanes in every iteration. New lanes are added based on their neighborhood relations and lanes without any lane points in the specified range are removed from the regions of interest.

The correctness of the occlusions strongly depends on a precise localization of the region of interest and the vehicle. A wrong localization can lead to false occlusions in regions that are actually not occluded and this can bring false decisions pf the planing algorithms.

4 Evaluation

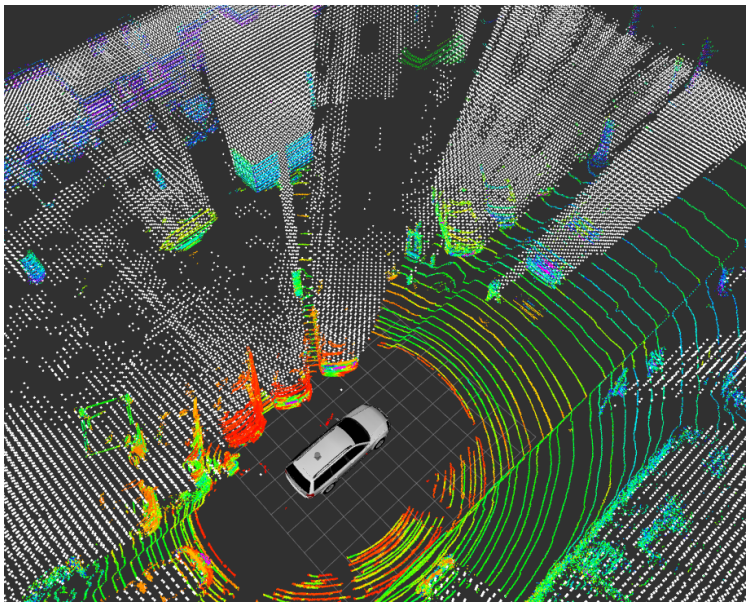
This chapter evaluates the results of the implemented depth buffer algorithm regarding to its computation time and accuracy. For the accuracy, the occlusions of the depth buffer are compared to the occlusions that would be generated by a more precise but therefore slower ray tracing algorithm, which uses the Velodyne point cloud. For the computation time, the algorithm is splitted into multiple parts, which are tested with different resolutions of the voxel grid.

4.1 Empty car windows

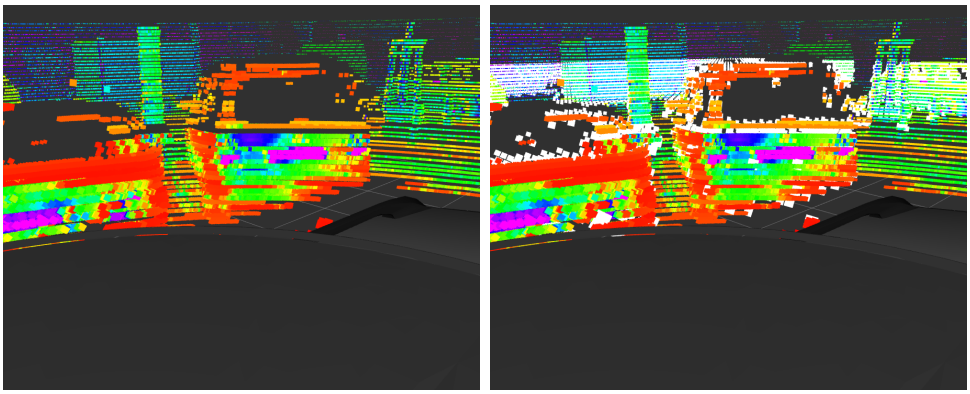
One issue that often effects the occlusion mapping are the missing sensor measurements in car windows. Figure 4.1 shows the resulting occlusions in such a case. The colored points are the measured impact points of the Velodyne lasers and the white points are the centers of the occluded voxels. Since no laser reflection has been received in this case, the sensor evaluates the ray as free. The occlusion estimation will map these empty pixels as free space. For example, if a car stands in front of a building, then some space of the building could be marked as free space because of the missing measurements in the car windows.

4.2 Ground plane estimation

The ground plane estimation is used to prevent superfluous occlusions caused by the ground plane. Since it is only an additional functionality to the occlusion estimation, this process should not cost much computation time. Therefore,



(a) Velodyne point cloud and occlusion point cloud



(b) Close up of Velodyne point cloud and occlusion point cloud

Figure 4.1: Missing measurements in Velodyne point clouds

a fast method was chosen, which only estimates one ground plane. Table 4.1 shows the results for the estimations with different numbers of sample points.

Table 4.1: Computation time for estimating the ground plane with the method described in section 3.4.

Number of sample points	Computation time	Mean distance of ground plane to origin	Average absolute deviation
986	0.90 ms	-0.138 m	0.0889 m
1420	1.19 ms	-0.166 m	0.0692 m
2218	3.24 ms	-0.207 m	0.0659 m
3944	4.99 ms	-0.216 m	0.0586 m
8875	12.66 ms	-0.197 m	0.0580 m

For each test, 100 successive ground plane estimations were executed on the same data. The data of a steady driving maneuver was used and the distance of the ground plane to the origin should not move too much during the tests. Since the origin of the base_link frame is on the front axis of the car, the orthogonal distance of the ground plane to the origin should be in a range from -0.1 m to -0.3 m. With a high number of sample points, the average absolute deviation from the mean distance is at ~0.06 m. And even with a lower number of sample points, the deviation is still at 0.07 m - 0.09 m.

4.3 Accuracy of the estimations

To measure the accuracy of the estimated occlusions, the results are compared to the more accurate ray tracing method mentioned in 3.2. This algorithm uses the point cloud of the Velodyne sensor as a reference.

The point cloud of estimated occlusions is first converted into an octree, which provides a faster ray tracing. Afterwards, a ray is created from the sensor origin to each impact point of the Velodyne point cloud. Each voxel that is crossed by the ray is compared to the estimated occlusions. With an infinite small size of the voxels, all voxels that are crossed by the ray should be free. But, depending on the voxel size and the angular resolution, the voxels will often be crossed by

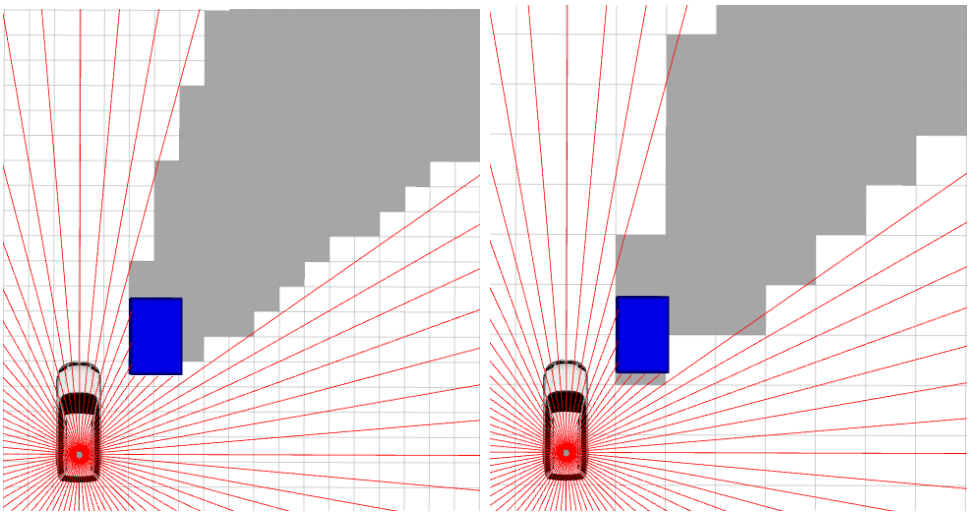


Figure 4.2: 2D example for partially occluded grid cells with different resolutions.

multiple rays. Figure 4.2 shows an example for 2D with different resolutions. The rays can also be extended and all voxels that are crossed by the extended rays should be occluded, in case of an infinite small size of the voxels. Voxels that are crossed by at least one normal ray and one extended ray are partially occluded voxels. These voxels can have different states, depending on the used algorithm. Some of the methods to classify such voxels are:

- Voxel is occluded if at least one extended ray has intersected it
- Each voxel stores a relative value of how many rays were extended
- The closest ray to the voxel center decides whether a voxel is occluded or not

The depth buffer uses implicitly the second method, since it assigns the closest ray to each voxel. With an increasing size of the voxels, it is more likely that a voxel is partially occluded. This also depends on the angular resolution of the outgoing rays. Smaller angles between the rays also increase the probability of partially occluded voxels.

In this experiment, each voxel that is crossed by the rays or the extended rays is compared to the estimated occlusions and partially occluded voxels are count. To make sure that each voxel has been counted only once, it is marked

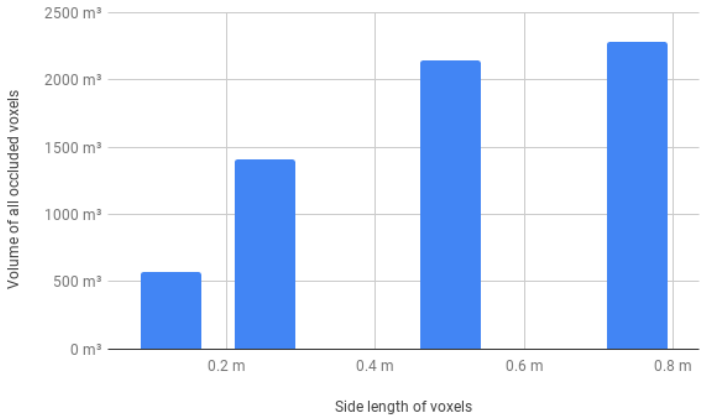


Figure 4.3: Volume of partially occluded voxels

as visited. The results are then normalized by the voxel size to make them more comparable with other resolutions.

Table 4.2 and figure 4.3 show the results. The tests used a bounding box with the size of size of $100m \times 100m \times 1.5m$. The volume of this bounding box was 15,000 m^3. To ignore the ground plane occlusions, the minimal height of the bounding box was 0.75 m and the maximal height was 2.25 m. Since the number of all voxels inside the bounding box is growing exponentially with a decreasing voxel size, the number of all partially occluded voxels is also growing exponentially. But in relation to the volume of the voxels and to the volume of the entire bounding box, the volume of the partially occluded voxels decreases.

Table 4.2: Accuracy of the occlusion estimations

Side length of voxels	**Number of partially occluded voxels**	**Volume of all partially occluded voxels**	**Relative value to the examined region**
0.125 m	291732	$569.79m^3$	0.038
0.25 m	90290	$1410.78m^3$	0.094
0.5 m	17166	$2145.75m^3$	0.143
0.75 m	5414	$2284.03m^3$	0.152

4.4 Computation time

While the previous experiment focused on the accuracy of the occlusion estimation, this experiment will target the computation time. The tests were performed on a Lenovo Ideapad E31-70 with the following specifications:

OS	Ubuntu 16.04 xenial 64 bit
CPU	Intel Core i3-5005U @ 2.00GHz
RAM	4 GB

As already mentioned, the computation time depends on the number of voxels inside the regions of interest. Therefore, each test in this section was executed with different numbers of voxels. To focus on the basic occlusion estimation, the first two experiments were executed on normal bounding boxes, which include the immediate environment of the car up to a certain range. The third experiment will focus on the lane occlusions, since this method is more complex.

Table 4.3: Computation time for creating all voxel to depth buffer assignments.

Number of voxels	**Map size**	**Voxel side length**	**Duration for assigning all voxels to depth buffer cells**
57,600	60x60x2 m	0.5 m	13.9 ms
90,000	60x60x1.6 m	0.4 m	22.8 ms
160,000	60x60x1.2 m	0.3 m	39.6 ms
360,000	60x60x0.8 m	0.2 m	86.2 ms
640,000	100x100x1m	0.25 m	142.6 ms

The basic occlusion estimation can be separated into two tasks. The first task assigns all center points of the voxels inside the bounding box to the cells of the depth buffer. Table 4.3 shows the results of the executed tests for this task. It shows that it is more efficient to reuse the assignment for a high number of voxels if the transformation is not changing. And if the assignment needs to be calculated in each iteration, the computation time can be decreased by reducing the resolution of the voxel grid. The example in section 3.6.1 used

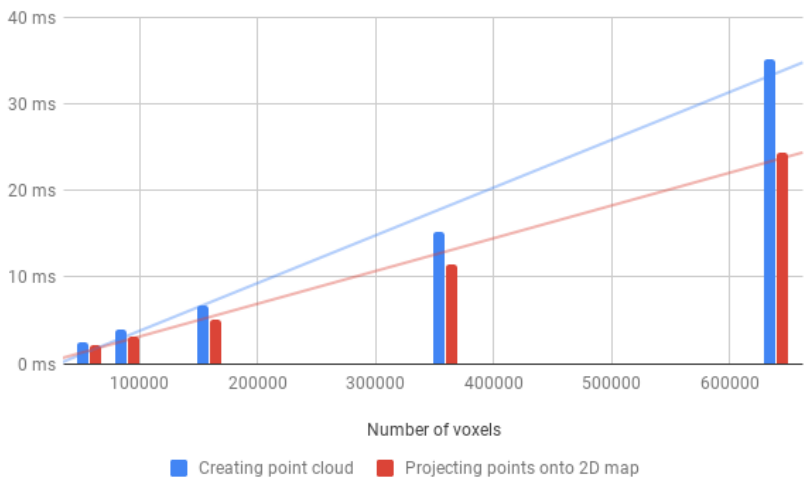


Figure 4.4: Computation time for checking all depth values and creating output.

a resolution of 0.5 m for two bounding boxes with each ~720 voxels. The assignment process for each bounding box took ~0.2 ms.

Table 4.4: Computation time for comparing all depth values and creating the output.

Number of voxels	**Map size**	**Voxel side length**	**Duration for creating point cloud**	**Duration for projecting occlusions onto 2D grid**
57,600	60x60x2 m	0.5 m	2.53 ms	2.09 ms
90,000	60x60x1.6 m	0.4 m	3.88 ms	3.15 ms
160,000	60x60x1.2 m	0.3 m	6.75 ms	5.12 ms
360,000	60x60x0.8 m	0.2 m	15.17 ms	11.48 ms
640,000	100x100x1m	0.25 m	35.09 ms	24.33 ms

The second task iterates over all voxels and compares their distances to the values in the depth buffer. Again, the computation time depends on the number of voxels in the region of interest. This can be seen in table 4.4. In this experiment, the assignment of the voxels to the depth buffer cells was already calculated and reused in each iteration. The table contains the results for cre-

ating the occluded voxels as a point cloud and as a projection onto a 2D grid. With an increasing number of voxels, the computation time has a higher growth in case of a point cloud creation than in case of the 2D projection.

In both experiments, the computation time of the depth buffer method is linear dependent on the number of voxels in the region of interest. This was one of decisive factors for the depth buffer method (section 3.2). The time for the ray tracing method additionally depends on the number of rays.

The estimation of the lane occlusions described in 3.6.2 is more complex and needs more computation time than using bounding boxes. Creating a voxel grid inside the lane space requires more calculations than subdividing a bounding box into voxels. To test this method, the occlusions were estimated for a junction with multiple lanes (as used in 3.6.2). All lanes in the range of 50 m to the vehicle were checked for occlusions. The results are shown in table 4.5 and figure 4.5. The result for each resolution is only a representative value of the test executions, since the number of voxels changes constantly during a drive. The time for iterating over all voxels is similar to the bounding box method. Even though the process of creating the voxel grid of the lane is more complex, the computation time for the first task is only slightly higher compared to the bounding box method.

Table 4.5: Computation time for estimating lane occlusions on a junction.

Number of voxels	**Voxel side length**	**Duration finding lane voxels**	**Duration for checking all voxels for occlusions**
21,683	0.5 m	6.24 ms	1.68 ms
32,890	0.4 m	9.26 ms	1.95 ms
58,810	0.3 m	18.12 ms	4.14 ms
135,339	0.2 m	35.61 ms	6.41 ms

All in all, the computation time of the lane occlusions is slightly higher than with the bounding boxes. Since a voxel grid is created independently of the global voxel grid for each lane, they can overlap. This causes additional computation time, since some regions are checked multiple times for occlusions.

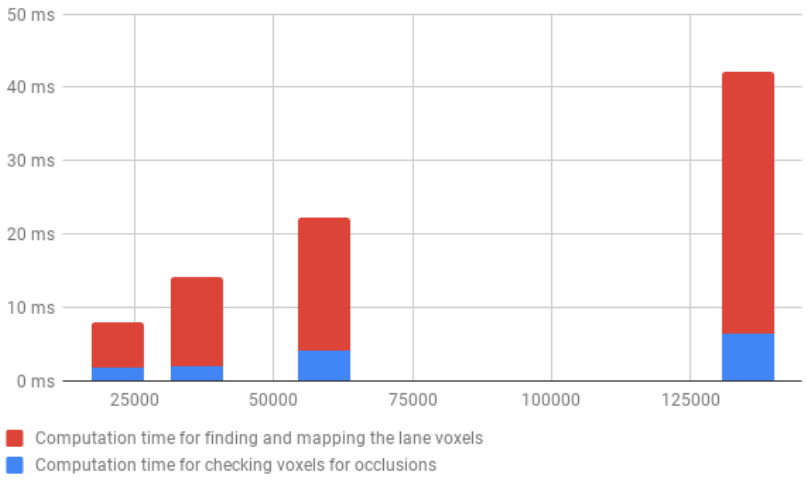


Figure 4.5: Computation time for estimating lane occlusions.

5 Conclusions and future work

5.1 Conclusions

The goal of this thesis was to find an efficient way to detect occlusions from LiDAR sensor data to get a better environment perception. Therefore, different methods have been considered for this task and compared to each other. To estimate the occlusions, the ray tracing method and the depth buffer method were compared to each other. Both are common used methods in computer graphics. Differences between a 2D and a 3D method were also discussed and the preferred 3D depth buffer method was then described.

Due to the changing ground plane orientation in relation to the vehicle, some of the examined regions were temporary under the ground plane. Therefore, a least squares method was presented to filter out such superfluous occlusions. Although this method is limited to planar ground planes the results during the recorded test drive through Berlin were satisfactory. Even with multiple ground plane orientations, the algorithm adapted the closest one.

To improve the performance and to provide an use case example for the occlusion estimations, the bounding boxes and the lanes were introduced. Both can be placed into the global map frame.

The results of the occlusion estimations have been examined and tested. One issue that effects the occlusion estimations are the missing sensor measurements in car windows. To evaluate the accuracy and the computation time, tests were executed for different sized bounding boxes. Regarding the accuracy, the results were compared to the more precise ray tracing algorithm. The test showed that the volume of all partially occluded voxels decreases in relation to the voxel size. In terms of the computation time, the method was divided into two parts. The first part computes the assignment of the center points of the

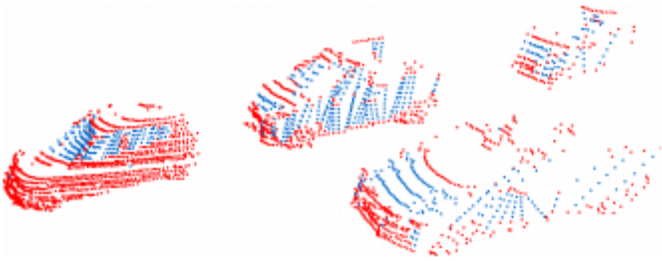


Figure 5.1: Artificial points generated in car windows. The blue points are the generated points and the red points are the measured impact points[BW18]

voxels to the corresponding cells in the depth buffer. The second part checks all center points for visibility. The computation time for both parts was examined. Afterwards, the computation time of the more complex method for the lane occlusion was examined and was only slightly higher than the bounding box method.

5.2 Future work

A few improvements of the used algorithm are still left for future work. These improvements concern the computation time, the accuracy and additional features of the occlusion estimation.

As mentioned in section 4.1, the issue with the missing measurements in vehicle windows needs further investigations since it decreases the reliability of the occlusion map. In [BW18], the authors presents a method to generate artificial points in such windows. His approach constructs a mesh graph by using the intrinsic pattern of the Velodyne sensor, which is similar to the range image used in this thesis. The vertices in this graph contain the information of the impact point and are connected to their horizontal and their vertical neighbors. The information contains also one of the values: free space, obstacle, noise, edge and invalid, which is found by a labeling algorithm. The invalid vertices can be further grouped and the range value can be approximated if the group fulfills predefined requirements. The result of this approach can be seen in figure 5.1.

Another issue that effects the reliability of the occlusion map is the ground plane estimation. The method that is used in this thesis estimates only one single plane. Although it worked well for the tested scenarios, it has its limits. In situations with a non planar ground plane, the method will probably adapt the ground plane orientation that is closer to the vehicle. Areas with a different ground plane orientation can cause false occlusions under the ground plane. The used ground plane estimation algorithm could be extended to calculate multiple ground planes at the same time. This can be achieved by separating the area into multiple smaller areas. The method used in this thesis can then estimate a ground plane in each area. The smaller and the more areas are used, the closer the estimation gets to a non planar ground plane. But on the other hand, the risk of using wrong points for the ground plane estimation also increases.

As described in 3.6, the method for finding the corresponding voxels to a lane is complex and need further improvements. The goal was to give an example use case for the occlusion mapping for smaller regions. The main issue that needs to be improved is the process of deciding which parts of the lanes should be checked for occlusions. This could be the whole lane or just the part inside a specific range. If the whole lane is going to be checked for occlusions then the computation time for a long lane can be very high. An approach with growing and shrinking lanes depending on the defined range is used in this thesis.

There are also a few things that can be done to improve the computation time. First, the assignment of the voxels to the corresponding cells in the depth buffer costs a lot of computation time due to the trigonometric functions. These functions can be implemented by a lookup table or an approximation as it is used in the NVIDIA CG Toolkit[NVIa]. NVIDIA also provides a programming model for computing on a graphical processing unit(GPU)[NVIb]. Since the method used in this thesis is based on graphical methods, the computation time can be improved by parallelizing and outsourcing parts to the graphical processing unit.

The occlusion estimation can also been used for other range based sensors. Therefore, the depth buffer needs to be changed in order to provide the correct assignments of the voxels to the correct cells. It can be extended to use other laser based sensors with a higher range and a smaller resolution. But it can also

be extended to use a planar projection of the depth information, as described in 2.5. These are used in stereo cameras. In order to use the depth buffer for point clouds, the depth buffer can be extended to project the distances of the points into its corresponding cells. The additional computation time depends on the size of the point cloud.

Bibliography

[App68] Appel, Arthur: Some techniques for shading machine rederings of solids. (1968) 3.2

[AS] Angel, Edward ; Shreiner, Dave: *Interactive Computer Graphics* (document), 2.6

[Auta] Autonomos: *e-Instein*. http://autonomos-labs.com/vehicles/e-instein/. – Online; accessed 06.04.18 2.1

[Autb] Autonomos: *History*. http://autonomos-labs.com/history/. – Online; accessed 06.04.18 2.1

[Autc] Autonomos: *Made In Germany*. http://autonomos-labs.com/vehicles/made-in-germany/. – Online; accessed 06.04.18 (document), 2.1, 2.2

[Autd] Autonomos: *Spirit of Berlin*. http://autonomos-labs.com/vehicles/spirit-of-berlin/. – Online; accessed 06.04.18 (document), 2.1

[AW15] Amanatides, John ; Woo, Andrew: A Fast Voxel Traversal Algorithm for Ray Tracing. (2015) 3.2

[BW18] Burger, Patrick ; Wuensche, Hans-Joachim: Fast Multi-Pass 3D Point Segmentation Based on a Structured Mesh Graph for Ground Vehicles. In: *IEEE Intelligent Vehicles Symposium* (2018) 5.1, 5.2

[Dar] Darpa: *DARPA Urban Challenge*. http://archive.darpa.mil/grandchallenge/. – Online; accessed 06.04.18 2.1

[Dav] Davies, Alex: *What is LiDAR, Why do self-driving cars need it, and can it see nerf bullets?* https://www.wired.com/story/lidar-self-driving-cars-luminar-video/. – Online; accessed 06.04.18 2.2

[FF] FU-Fighters: *Welcome to FU-Fighters!* http://robocup.mi.fu-berlin.de/pmwiki/Main/HomePage.html. – Online; accessed 06.04.18 2.1

[Gla84] Glassner, Andrew S.: Space Subdivision for Fast Ray Tracing. (1984) 3.2

[GOE15] Galceran, Enric ; Olson, Edwin ; Eustice, Ryan M.: Augmented Vehicle Tracking under Occlusions for Decision-Making in Autonomous Driving. (2015) 1.4

[Kap18] Kaplan, Jeremy: Heres every company developing self-driving car tech at CES 2018. In: *Digitaltrends* (2018). https://www.digitaltrends.com/cars/every-company-developing-self-driving-car-tech-ces-2018/. – Online; accessed 02.04.18 1.1

[MRN13] Matsumi, Ryosuke ; Raksincharoensak, Pongsathorn ; Nagai, Masao: Autonomous Braking Control System for Pedestrian Collision Avoidance by Using Potential Field. (2013) 1.4

[Muo17] Muoio, Danielle: The 18 companies most likely to get self-driving cars on the road first. In: *Business Insider* (2017). http://www.businessinsider.de/the-companies-most-likely-to-get-driverless-cars-on-the-road-first-2017-4?r=US&IR=T. – Online; accessed 02.04.18 1.1

[NB08] Nashashibi, Fawzi ; Bargeton, Alexandre: Laser-based vehicles tracking and classification using occlusion reasoning and confidence estimation. (2008) 1.4, 3.1

[NOA] NOAA: *What is LIDAR?* https://oceanservice.noaa.gov/facts/lidar.html. – Online; accessed 06.04.18 2.2

[NVIa] NVIDIA: *Cg 3.1 Toolkit Documentation*. http://developer.download.nvidia.com/cg/index_stdlib.html. – Online; accessed 08.08.18 5.2

[NVIb] NVIDIA: *CUDA Zone*. https://developer.nvidia.com/cuda-zone. – Online; accessed 29.08.18 5.2

[OG13] Oliveira, Henrique C. ; Galo, Mauricio: Occlusion detection by height gradient for true orthophoto generation, using LiDAR data. (2013) 1.4

[PCLa] PCL: *Getting Started / Basic Structures*. http://pointclouds.org/documentation/tutorials/basic_structures.php. – Online; accessed 06.04.18 2.6

[PCLb] PCL: *What is PCL?* http://pointclouds.org/about/. – Online; accessed 06.04.18 2.6

[ROSa] ROS: *About ROS*. http://www.ros.org/about-ros/. – Online; accessed 03.04.18 2.3

[ROSb] ROS: *Is ROS for Me ?* http://www.ros.org/is-ros-for-me/. – Online; accessed 04.04.18 2.3

[ROSc] ROS: *ROS Core Components*. http://www.ros.org/core-components/. – Online; accessed 04.04.18 2.3

[ROSd] ROS: *ROS Nodelets*. http://wiki.ros.org/nodelet. – Online; accessed 04.04.18 2.3

[ROSe] ROS: *Setting up your robot using tf*. http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF. – Online; accessed 28.05.18 2.4

[SAE14] SAE: *Automated driving - Levels of driving automation are defined in new SAE international standard J3016*. 2014 (document), 1.1, 1.1

[SHN12] Sheehan, Mark ; Harrison, Alastair ; Newma, Paul: Self-Calibration for a 3D Laser. (2012) 2.2

[Str74] StraSSer, Wolfgang: Schnelle Kurven- und Flächendarstellung auf grafischen Sichtgeräten. (1974) 2.5

[Tho18] Thornton, Sarah M.: Autonomous Vehicle Speed Control for Safe Navigation of Occluded Pedestrian Crosswalk. (2018) 1.4, 3.1

[Vel] Velodyne: *REV A OUTLINE DRAWING HDL-64E*. Velodyne Acoustics, Inc. 345 Digital Drive Morgan Hill, CA 95037 (document), 2.3

[Velnd] Velodyne: *HDL - 64E User's Manual*. 1. Velodyne Acoustics, Inc. 345 Digital Drive Morgan Hill, CA 95037, n.d. 2.2

[Wei] Weisstein, Eric W.: *Spherical coordinates*. MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/SphericalCoordinates.html. – Online; accessed 28.08.18 2.4

[Wika] Wikipedia: *Octree*. https://en.wikipedia.org/wiki/Octree. – Online; accessed 10.05.18 2.9

[Wikb] Wikipedia: *Singular-value decomposition*. https://en.wikipedia.org/wiki/Singular-value_decomposition. – Online; accessed 27.08.18 (document), 3.6

[Wikc] Wikipedia: *Z-buffering*. https://en.wikipedia.org/wiki/Z-buffering. – Online; accessed 19.05.18 2.5

[YKP+13] Yoo, Hyun W. ; Kim, Woo H. ; Park, Jeong W. ; Lee, Won H. ; Chung, Myung J.: Real-Time Plane Detection Based on Depth Map from Kinect. (2013) (document), 3.4, 3.5