UPPSALA
UNIVERSITET

UPTEC F 23039
Examensarbete 30 hp
Juni 2023

Application development of 3D LiDAR sensor for display computers

Ekstrand, Oskar

Civilingenjörsprogrammet i teknisk fysik

UPPSALA UNIVERSITET

Application development of 3D LiDAR sensor for display computers

Ekstrand, Oskar

Abstract

A highly accurate sensor for measuring distances, used for creating high-resolution 3D maps of the environment, utilize "Light Detection And Ranging" (LiDAR) technology. This degree project aims to investigate the implementation of 3D LiDAR sensors into off-highway vehicle display computers, called CCpilots. This involves a study of available low-cost 3D LiDAR sensors on the market and development of an application for visualizing real time data graphically, with room for optimization algorithms. The selected LiDAR sensor is "Livox Mid-360", a hybrid-solid technology and a field of view of 360° horizontally and 59° vertically.

The LiDAR application was developed using Livox SDK2 combined with a C++ back-end, in order to visualize data using Qt QML as the Graphical User Interface design tool. A filter was utilized from the Point Cloud Library (PCL), called a voxel grid filter, for optimization purpose. Real time 3D LiDAR sensor data was graphically visualized on the display computer CCpilot X900. The voxel grid filter had a few visual advantages, although it consumed more processor power compared to when no filter was used. Whether a filter was used or not, all points generated by the LiDAR sensor could be processed and visualized by the developed application without any latency.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Richard Hedlund Ämnesgranskare: Ping Wu
Examinator: Tomas Nyberg

Populärvetenskaplig sammanfattning

Teknik och fordon runt om oss blir smartare och interagerar mer och mer med våra liv. Vardagligen använder vi oss av mobiler, datorer och andra enheter som samlar in information från omgivningen. De senaste åren har bilindustrin gjort en storsatsning på självkörande fordon. Den autonoma körningen uppnås genom att använda många olika typer av sensorer runtom hela bilen för att garantera att bilen håller sig på vägen, inte kör mot röttljus och inte kolliderar med andra fordon. En av sensorerna som används för att mäta avstånd till objekt och andra trafikanter är en så kallad "LiDAR sensor". Sensorn använder en laser teknologi, där en osynlig laser skickas ut och reflekteras tillbaka från objektet till sensorn. Avståndet till föremålet bestäms genom att använda sig av den uppmätta tiden för laserstrålen att färdas från givaren, träffa objektet och återvända tillbaka till sensorn. En annan tillämpning av LiDAR sensorer är att skapa högupplösta 3D-kartor av omgivningen.

Företaget CrossControl designar och producerar robusta fordonsdatorer till maskiner som arbetar i tuffa miljöer. Många av deras kunder är inom gruv-, bygg- och jordbruksindustrin. LiDAR teknologin börjar även komma till dessa industrier och därmed är det i CrossControls intresse att utforska tekniken samt vara redo med lösningar för hantering av sensorn. I detta arbete skapas en applikation på en av fordonsdatorerna för att visa avståndsdata från sensorn. Det ingår även en undersökning av tillgängliga LiDAR sensorer på marknaden utifrån funktionalitet- och kostnadsperspektiv. Dessutom studeras möjlighet till att optimera sensordata med hjälp av ett databehandlingsfilter.

Applikationen utvecklades genom kodning i programmeringsspråken C++ och QML, i kombination med sensorns inbyggda ramverk och funktioner. Den slutgiltiga applikationen lyckades starta på en fordonsdator, vid namn "CCpilot X900". På fordonsdatorn visas realtidsdata från LiDAR sensorn "Livox Mid-360", som har en möjlighet att mäta punkter i hela rummet (360° vertikalt och 59° horisontellt). Ett databehandlingsfilter användes för att ta bort irrelevanta insamlade datapunkter från sensorn, därefter utvärderades filtret. En utvärdering av applikationen gjordes genom att mäta processorlast och om en fördröjning uppstod på fordonsdatorn. Resultatet visade att filtret använde mer processorkraft jämfört med när inget filter nyttjades. Däremot kunde sensordata upplevas som mer strukturerat när ett databehandlingsfilter användes. Oberoende om ett filter användes eller ej så kunde inte någon fördröjning uppmätas på fordonsdatorn.

Table of Contents

List of Abbreviations

The following list describes the significance of various abbreviations and acronyms used throughout the thesis.

Abbreviation	**Definition**
API	Application Programming Interface
CPU	Central Processing Unit
FOV	Field Of View
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
IMU	Internal Measurement Unit
LiDAR	Light Detection And Ranging
MEMS	Micro-Electro-Mechanical Systems
OS	Operating System
PCL	Point Cloud Library
QML	Qt Modeling Language
SDK	Software Development Kit
TFT	Thin-Film Transistor
ToF	Time of Flight
VM	Virtual Machine

1 Introduction

1.1 Background

As technology advances, more and more aspects of our lives are becoming digitized and automated. Machines and vehicles around us are getting smarter by collecting data throughout different measurements of the world. In the last decade, the car industry has made a great leap towards fully self driving cars by using a lot of data from various sensors around the vehicle. One of the highly accurate sensors that is commonly integrated with the car uses "Light Detection And Ranging" (LiDAR) technology [1]. The LiDAR sensor measures distances with great precision and can be used to create high-resolution 3D maps of the environment. One of its main uses is to utilize the distance measurements to prevent collisions.

Despite the advancements made by the car industry in this aspect, the closely related industry of off-highway vehicles, including vehicles used in mining, construction and agriculture, is to some degree behind in implemented automotive driving. In contrast, the complexity of vehicles in off-highway industry are usually higher and demands more steering. Although implementation of different sensor and cameras are common in off-highway units, such as temperature gauges, pressure sensors and reversing cameras. The demand of automotive vehicles is increasing in this industry and there are a lot of opportunities for automation, which could be accomplished by implementing a LiDAR senor. Moreover, the cost of LiDAR technology is continuously decreasing and low budget alternatives is brought to the market [1]. The highly accurate LiDAR sensors could also be used as an aid, for instance reversing a truck so that it is perfectly aligned and capable of picking up a container.

The company CrossControl, that design and manufactures display computers for challenging environments, is intrigued to learn more about the widely used LiDAR technology and its applications. CrossControl is aiming to be prepared with its display computer solutions as its off-highway vehicle business customers are exploring LiDAR sensors. Demonstrating a completed LiDAR application, running on the company's units, indicates that the display computers are performant enough.

1.2 Purpose

This master degree project aims to investigate the implementation of 3D LiDAR sensors into off-highway vehicle display computers, called CCpilots. That involves an investigation of the features and costs of the available 3D LiDAR sensors on the market, and based on the investigation select the most suitable. In addition, the purpose is to look into software framework in order to handle sensor data in the operating system CCLinux.

The objective is to develop an application for a chosen sensor that operates on one of the vehicle computers, where real time data is visualized graphically. An evaluation and implementation of algorithms to optimize the performance, based on latency and processor load, will be executed.

1.3 Tasks

In order to reach the desired outcome of the project, the following tasks are to be carried out.

(i) A literature study of available 3D LiDAR sensors on the market from a feature and cost perspective.

(ii) Examine the compatibility possibilities of the CCpilots with the sensors, including their interface, kernel drivers and Application Programming Interfaces (API).

(iii) Sensor selection based on the literature study and compatibility of the display computers.

(iv) Investigate software frameworks for handling sensor data in Linux and application development platforms.

(v) Develop an application for a chosen sensor that visualizes real time data graphically.

(vi) An evaluation of the performance of the application, based on latency and processor load.

(vii) Develop and explore algorithms to optimize the performance of the application, based on latency and processor load.

1.4 Scope

Various limitations have impacted the scope of the project. Firstly, the number of available LiDAR sensor is very large and it is not feasible to include all sensors in a 20 weeks project. The selection of the sensor is also affected of the limited budget for the project, thus the scope is low-cost LiDARs.

Moreover, there is a fixed time of 20 weeks to be carried out. This also affects the selection of LiDAR sensor, since the delivery time of a sensor could be many weeks, which is not feasible in order to achieve all the objectives of the project. Lastly, there are a lots of ways to evaluate the performance of the application. The scope of the application performance evaluation is determined to be latency and processor load, due to the limited time and priorities of the tasks in the project.

2 Theory

2.1 LiDAR

The LiDAR technology was originally developed by the American Navy in 1970s. The purpose was to measure elements of the sea or map land from an airplane [2]. Another early use case of LiDAR systems was during the Apollo 15 mission 1971, when a laser altimeter mapped the surface of the Moon [3]. The initial name was “Colidar”, an acronym for “coherent light detecting and ranging”, derived from RADAR, “radio detection and ranging” [4]. The Colidar systems are the predecessors of today’s laser rangefinders, laser altimeters, and LiDAR units.

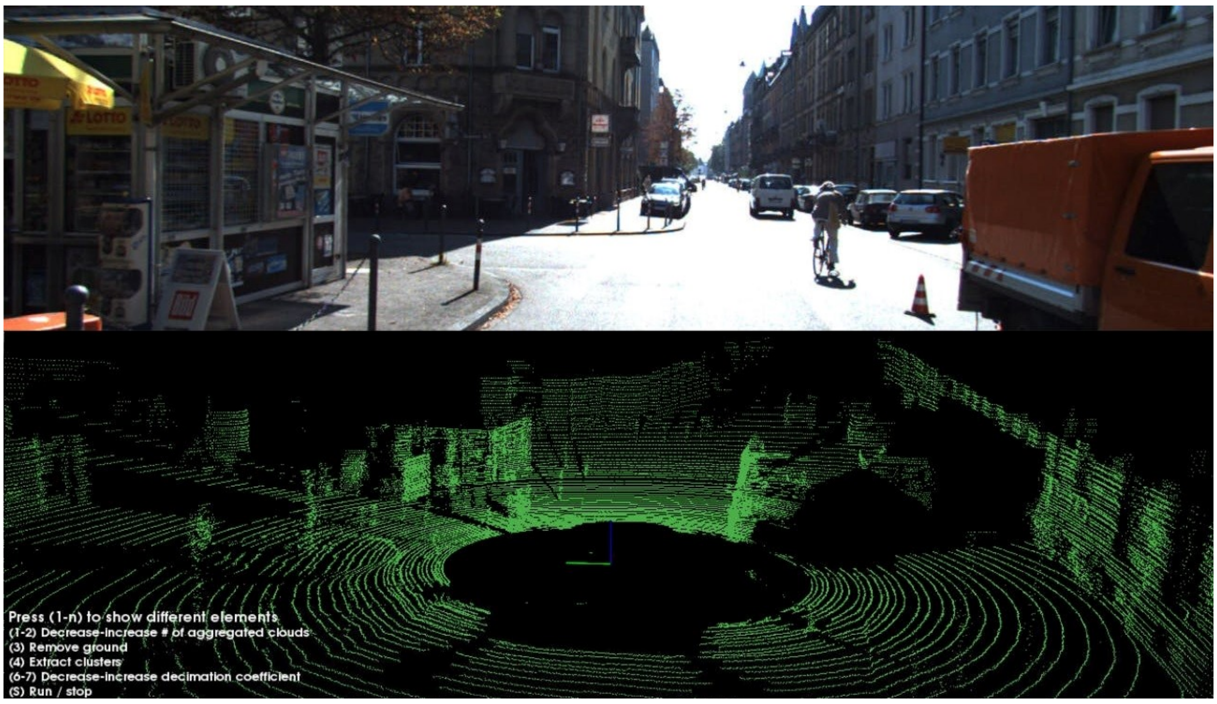

Press (1-n) to show different elements
(1-2) Decrease-increase # of aggregated clouds
(3) Remove ground
(4) Extract clusters
(6-7) Decrease-increase decimation coefficient
(S) Run / stop

Figure 2.1: A camera view at the top and the LiDAR view at the bottom [5].

2.1.1 LiDAR main components

The LiDAR sensor consists of three key components which are a transmitter (usually a laser), a receiver, together with a positioning and navigation unit. Two common systems that are used for positioning are an Internal Measurement Unit (IMU) or a Global Positioning System (GPS) [6].

The first component is the transmitter, which emits laser light pulses and have the capacity to emit up to 150,000 pulses/s [6]. The emissions are in the infrared range of the electromagnetic spectrum and are typically 905 nm or 1550 nm. These wavelengths are not visible for the human eye as can be seen in the spectrum presented below.

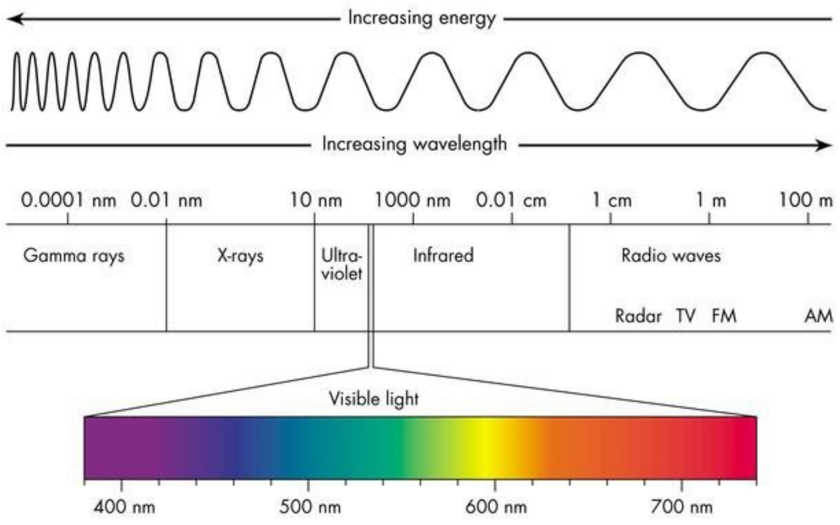


Figure 2.2: Electromagnetic spectrum [7].

Emissions at 1550 nm demand more energy compared to those emitted at 905 nm because the water in the atmosphere starts absorbing energy from 1400 nm [2]. However, from an eye safety perspective, the 905 nm wavelength could cause damage on the retina. In contrast, wavelength above 1400 nm can no longer pass through the eye to the retina, because the wavelength get stopped by the cornea and the lens [8]. Nevertheless, the scanning speed of the lasers are very high which results in the beam only reaching the retina for a small amount of time. Thus, the shorter wave length causes very little to no damage. There are also safe classifications of the lasers used in the industry which ensures eye safety while operating. Laser transmitters used for vehicles belong to Class 1, according to the International Electrotechnical Commission (ICE) Standard [2].

The next one is the receiver, that is measuring the time of flight (ToF) by the incoming light beam sent out from the transmitter. The receiver can also determine the object's reflection by measuring the light intensity bounced back from the object. In order to measure low-intensity light from a non-reflective area, such as a black surface, a highly sensitive receiver is needed. Silicon PIN detector, Silicon avalanche photodiode (APD), and Photomuliplier tube (PMT) are techniques that are used for low light detection [6]. In Section 2.1.4 is reflectivity described in greater details.

The last component is an IMU in the LiDAR sensor tracking the object's angle and positioning of the sensor. The IMU is usually a accelerometer combined with a gyroscope. The position of an object can also be determine by a GPS, which determines the object's longitude, latitude, and altitude on the surface of the Earth [6].

2.1.2 Working principle

The working principle of a LiDAR system relies on measuring the time of flight (ToF) of a pulsed light emitted from a laser diode till it has reached back to an receiver. Assuming that the medium is air, the distance between the LiDAR sensor and the surface can be determined by the following equation,

$$d = \frac{c \cdot t}{2} \tag{2.1}$$

where d is the distance [m], c is the speed of light ($3 \cdot 10^8$ m/s) and t is the ToF given in seconds [s] [2][6].

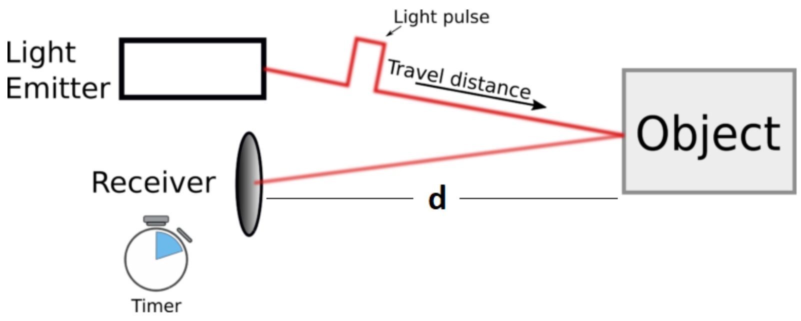


Figure 2.3: Illustrating the working principle of a LiDAR sensor [9].

2.1.3 Classification and characteristics

LiDAR sensors can be classified in different ways. One way is based on the type of information they gather from their surroundings in 2D or 3D. Another way is to classify according to their construction, called spinning or solid state [6]. Sections 2.1.5 and 2.1.6 describe spinning and solid state LiDARs in more detail. In general, 2D LiDARs have just one rotating laser beam perpendicular to the axis rotation. Hence, the laser sweeps in a plane, which is why they are called "2D LiDAR" [10]. In contrast, a 3D LiDAR uses a set of lasers mounted on a part that is spinning at high speed. 3D LiDARs provide a 3D map of the environment, where the accuracy is effected by the number of laser beams. The number of lasers emitted by the sensor is called "channels" or "lines", and are currently ranging from 4 to 128 lasers on today's market [2].

The field of view (FOV) is varying both horizontal and vertical from sensor to sensor. At present, the horizontal FOV is up to 360° and the vertical oscillates between 20-90° [2][11]. FOV converage is a standard metric when comparing LiDAR sensors performance. It is a type of measure of the ratio covered by the sensor's laser beams and total area in FOV, typically denoted in percentage. The FOV coverage (C) can be determined using Equation 2.2 [12],

$$C = \frac{\text{Total area illuminated by laser beams}}{\text{Total area in FOV}} \tag{2.2}$$

Moreover, the minimum and maximum range are two other characteristics of a LiDAR. The minimum range is the closet distance that the sensor can detect an object, typically just a few centimeters to one or two decimeters. On the other hand, the maximum range can vary a lot and is specified in a certain way. The reflectivity level of the measured object affects the maximum range, thus is the range stated at fixed reflectivity. A widely used format is to specify the maximum range at 10% and 80% reflectivity level [11]. A few manufactures uses an additional parameter, "probability of detection", which is the ratio between detected objects relative all possible detectable targets (given in percentage) [11]. However, short-range LiDARs could measure object at a distance of a few meters, while long-range LiDARs can reach multiple thousand of meters [13].

Finally, the accuracy of measuring distances to a object with a LiDAR is quite precise. This is often specified in a metric such as, "distance error" or "depth error". In general, is the error ±1 cm to ±5 cm for a 3D LiDAR [11]. It is common to state the distance error at a fixed distance and a fixed reflectivity level.

2.1.4 Reflectivity

The reflectivity from an object can be perceived in two distinct ways, either according to the "Lambertian reflection model" or to be considered as "retroreflection". A Lambertian surface reflects light uniformly in all directions, regardless of the angle of incidence of the light. Hence, the direction of the light has no effect of the reflected light. This type of surface does not exist in the real world, but is a useful tool for applications, computer graphics and optics. The surface is also called a ideal diffusion surface and in Figure 2.4 is an illustration. The "specular reflection" has the same angle of incidence relative the surface normal as the incoming light. Hence, a specular surface behaves like a mirror [14].

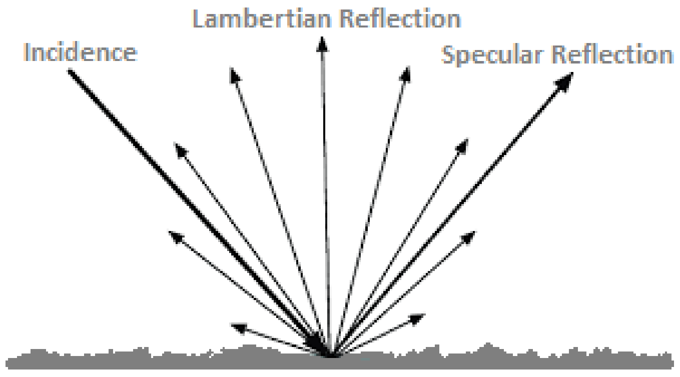


Figure 2.4: Lambertian reflection model.

Retroreflection occurs when an object reflects light back in the same direction as its source, without being significantly scattered or absorbed. In other words, a large amount of the emitted light reflects back and can be seen from a distance (shown in Figure 2.5). This property is often used for designing reflective material, for instance, in traffic signs and safe clothing [15].

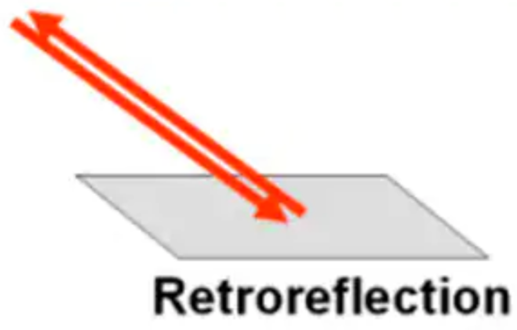
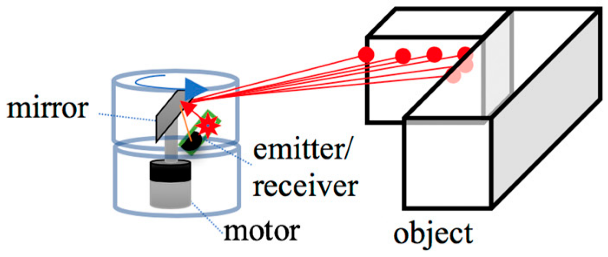


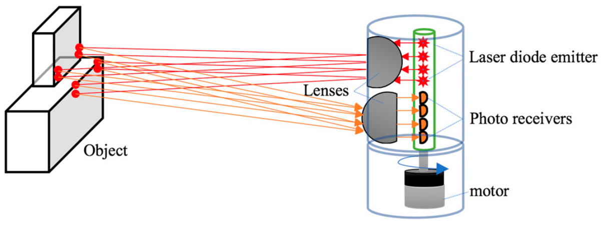
Figure 2.5: Retroreflection [16].

2.1.5 Spinning LiDAR

The defining characteristic of the spinning LiDAR is the motors, one for rotating the emitter and receiver in the plane and another for tilting the mirror [6]. This ensures that the emitted laser light is redirected to a new position, thus measures the distance in another direction. The spinning technique greatly impacts and enables a wide FOV both vertically and horizontally. Nevertheless, the draw back is that the sensor becomes more vulnerable to vibration because of the moving parts [6]. The principle of a 2D and a 3D spinning LiDAR, respectively, are illustrated in the Figure 2.6.



(a) Principle of 2D spinning LiDAR.



(b) Principle of 3D spinning LiDAR.

Figure 2.6: Two types of spinning LiDARs [2].

2.1.6 Solid state LiDAR

The solid state LiDAR technique typically utilize a single laser beam which is synchronized with a micro mirror Micro-Electro-Mechanical Systems (MEMS) circuit [2]. This enables the sensor to scan the horizontal FOV in multiple lines. To accomplish this, the laser beam is reflected over a diffuser lens by the micro mirror, forming a vertical line that reaches the objects. A lens captures the reflected light and the light is processed by a photodiode array in order to create the first row of a 3D matrix. The procedure is repeated until the desired output is generated [2]. As a result, the solid state LiDAR has a few to no moving parts. In other words, the sensors are more robust compared to spinning LiDARs and are also more cost-effective [8]. At present, spinning LiDARs have a wider FOV than solid state. The two techniques differ the most in horizontal FOV, where solid state has 270° compared with 360° [8]. In vertical FOV, the two operate with similar outcome of approximate 90°. A principle illustration of a solid state LiDAR is presented in Figure 2.7.

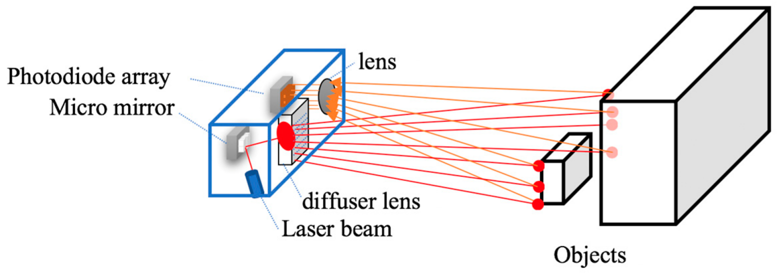


Figure 2.7: Principle of a solid state LiDAR.

Recently, various manufactures started to combine the spinning and solid state technique. The technology is called "hybrid solid state LiDAR" and is built upon a rotating mirror in a solid state manner. This improves the robustness, apart from expanding the FOV compared with solid state LiDARs [12].

2.1.7 LiDAR output

The output of a LiDAR is regularly referred to as a "point cloud". It is a collection of multiple-dimensional points in a data format, the structure of the data can vary from sensor to sensor. Within a 3D point cloud, the points usually corresponds to the x, y and z in a "Cartesian coordinate system", representing a sampled surface [17]. The Cartesian coordinate system is described in greater detail in Section 2.2. A widely used application is to generate high-resolution 3D maps of the surrounding, which will be done in this project.

2.2 Cartesian coordinate system

A fundamental way to represent a point in a three-dimensional space is to make use of the Cartesian coordinate system. The system utilizes 3 perpendicular axis, in the x, y and z direction. A point P (x, y, z) in the 3D space belongs to $\mathbb{R}^3$. The directions x, y and z can be seen as height, width and depth, which varies depending on the context. The intersection of the axis is called "Origo" (0, 0, 0), which is a common reference point for measuring coordinates [18]. The orientation of the axis follows the right-hand rule convention, where the positive z direction is upwards, positive y direction points to the right and positive x direction outward from the plane (seen in Figure 2.8). Hence, positive rotation occurs counterclockwise around an axis when looking in a positive direction of that axis [18].

The distance between two points $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$ in the 3D space can be calculated by using the following equation, which is the distance formula [18],

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{2.3}$$

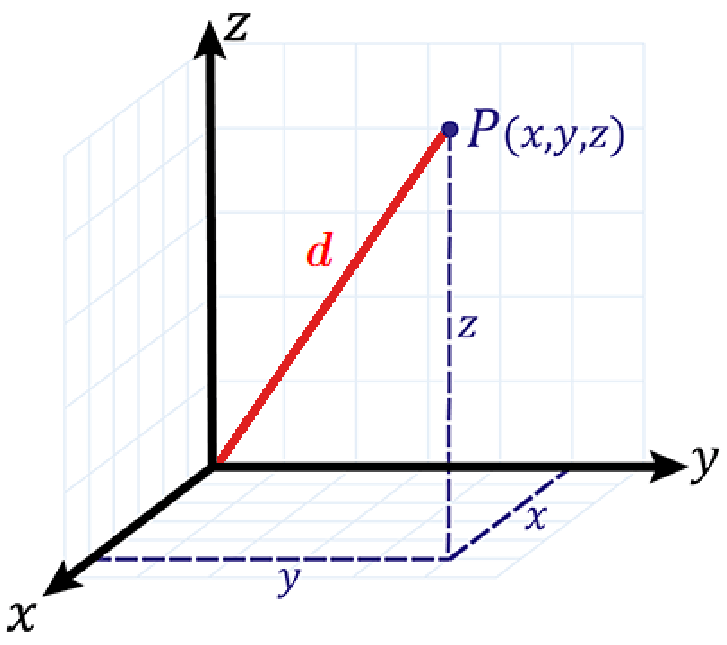


Figure 2.8: The 3-dimensional Cartesian coordinate system with the axis x, y and z. The red line represent the distance (d) from Origo to the point P [19].

2.3 Voxel grid filter

The term "voxel" is a combination of "volume" and "pixel", thus a volume element. In a voxel grid filter the point cloud data is divided into smaller 3D boxes (voxels) in order to apply a filter to each voxel. The applied filter performs a calculation to derive the centroid with respect to all points encompassed within each voxel [20]. The idea is to downsample the original data to less points, but still preserve the structure and geometry of the data, as shown in Figure 2.9.

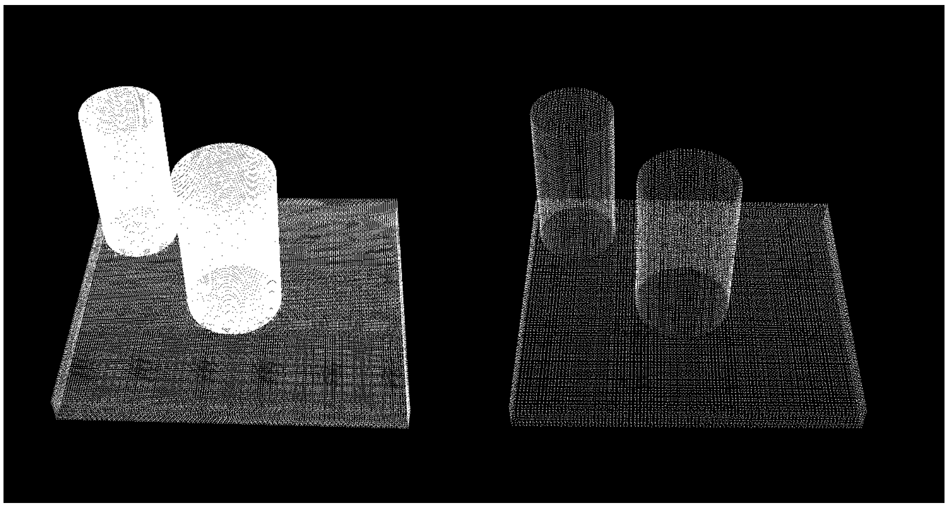


Figure 2.9: Voxel grid filter for downsampling, showcasing the original point cloud data on the left and the downsampled data on the right [21].

The process removes noisy data and insignificant small-scale variation in the data. Apart from that, the centroid represent the underlying surface accurately, it helps to identify regions where the data undergoes significant shifts in value or geometry [20]. Signs of high variability or significant changes in data could be identified if the values of centroid's local neighbourhood differs considerable from the value of the centroid itself.

The voxel grid filter executes the downsampling of a point cloud according to the following steps,

1. Define the Voxel grid filter parameters. Firstly, the size of each voxel, the Leaf size (L_i), is set for x, y, and z respectively. Secondly, the grid dimensions is set based on the minimum and maximum dimension of the input point cloud along each axis. An 3D grid space is initialized with the given parameters.

2. Assign the points of the cloud to each voxel. For each point $(x,\ y,\ z)$ the voxel index (v_i) is calculated, by using the following equations:

$$v_x = \frac{x}{L_x} \quad v_y = \frac{y}{L_y} \quad v_z = \frac{z}{L_z} \tag{2.4}$$

3. The centroid (C_i) of each voxel is calculated. Points assigned to the same voxel index will contribute to a single centroid point $(C_x,\ C_y,\ C_z)$, according to the following equations:

$$C_x = \frac{1}{N}\sum_{i=1}^{N} x_i \quad C_y = \frac{1}{N}\sum_{i=1}^{N} y_i \quad C_z = \frac{1}{N}\sum_{i=1}^{N} z_i \tag{2.5}$$

4. A new point cloud is generated by using the computed centroiod of each voxel.

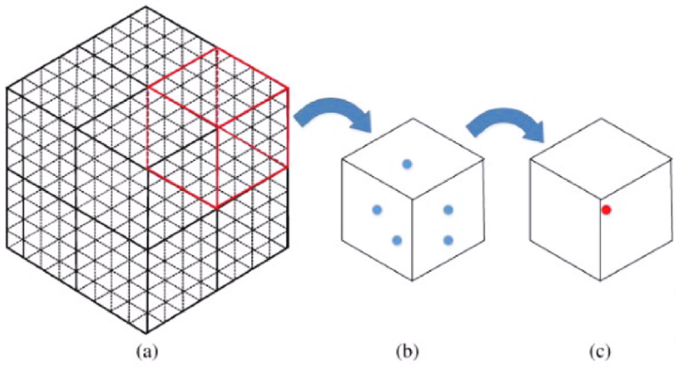


Figure 2.10: Illustration of the working principle of the Voxel grid filter. In (a) is a unfiltered voxel, in (b) is an unfiltered unit voxel, and (c) is the mass center of the unit voxel [22].

2.4 RGBA color model

There are a number of alternatives to represent a color in the color space, one way is to use the "RGBA color model". RGBA denotes red, green, blue and alpha, which is a extension of the RGB color model with an alpha parameter. The alpha parameter signifies the opaqueness and allows the generated color to be more transparent [23]. In the both models, the intensity of each primary color is based on a specified value. A small value implies a low intensity of the particular color, red, green, or blue, while a large value indicates high intensity. Thus, a wide spectrum of colors can be generated by utilizing various value combinations.

A typical application of the model is to encode pixels, small picture elements, in computer graphics. The 4 components of the RGBA is stored in a 32-bit integer, with 8 bits each. There are various orders the bits can be stored, in the big-endian format is the top 8 bits red follow by green, blue, and alpha. It is common for the values to be expressed in 8 hexadecimal digits, from 0 to F (0-15), which form the full 32-bit [23]. For instance, 'FFFF0080' represent 100 % red and green, 0 % blue and 50 % transparent, which results in a half transparent yellow color.

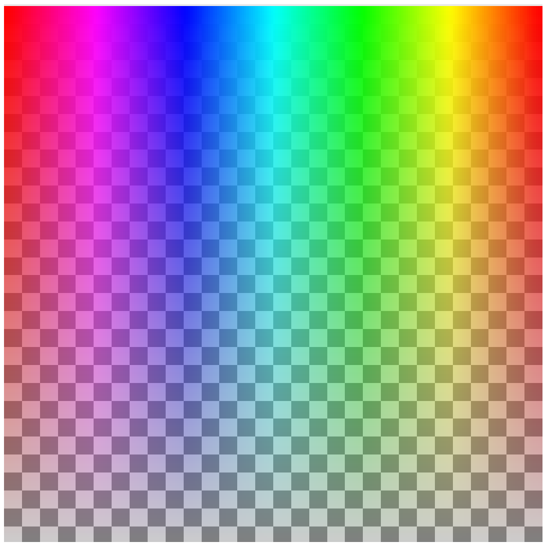


Figure 2.11: RGBA colorization over a checkerboard background [24].

3 Sensor selection

There are a lot of characteristics to take into account when selecting a 3D LiDAR. A few of the features are, for instance, range, accuracy, FOV, data points per second, Spinning LiDAR or Solid state LiDAR. The communication interface also varies for different devices where Ethernet and Universal Serial Bus (USB) are the most common. When selecting a LiDAR sensor for Simultaneous Localization and Mapping (SLAM) the characteristics of FOV, range and frame rate have the greatest impact on the performance [11]. For indoor use cases typically a wider FOV is required due to many objects being located at different heights near the sensor. On the other hand, for outdoor use cases other parameters could have greater significance, such as range and the probability of detection at what reflectivity level [11]. The range for outdoor applications should be in general at least 50 m, but when used in large warehouses a higher range is required. For robotic application, a frame rate of 10 Hz is sufficient and for vehicle-based applications is a higher frame rate recommended [11].

3.1 Key characteristics

In this project, the following characteristics and parameters were prioritized while selecting the 3D LiDAR, as shown in Figure 3.1.

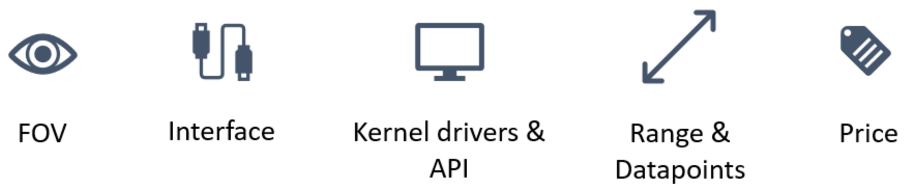


Figure 3.1: Characteristics and parameters for 3D LiDARs prioritized when selecting sensor for this project.

Firstly, the horizontal and vertical FOV are important in order to produce a high resolution point cloud of the surrounding. Therefore is it more beneficial to select a sensor with high FOV in both orientations. Secondly, the interface of the data communication must be compatible with the CrossControl's display computers. For the device "CCpilot X900", the interfaces provided are Ethernet, CAN bus and USB. Further specifications of the device can be seen in Section 4.2.2. Moreover, existing kernel drivers and Application Programming Interface (API) were important parameters for facilitating the integration. This enables the project focus to be more on the application development, rather than implementing drivers for the sensor.

Two other significant characteristics are the range of the sensor and number of data points collected per second. The minimum and maximum range of the sensor are both important, the minimum range can not be excessively high and the maximum range should not be too short. Lastly, the price tag of the sensor was also taking in consideration. The sensors on today's market can cost a couple of hundreds USD up to several hundreds of thousands of dollars. The target for this project was low-cost LiDARs, due to budget limitations. In addition, a low-cost LiDAR is also more affordable for CrossControl's costumers, hence a more interesting and attractive sensor alternative. The delivery time was also taken into consideration, too long delivery was not possible because of the set time of the project.

3.2 Sensor comparison

The investigation of 3D LiDAR sensors was done by searching on the Internet in order to gain knowledge of the available sensors according to the key characteristics. It is not feasible to include all of the studied sensors in this thesis, there are simply too many. Additionally, the budget also constrains the LiDAR selection. Thus, this thesis includes three of the studied sensors on today's market. In Table 3.1 is a comparison of the sensors "Velodyne Velarray M1600", "Livox Mid-40" and "Livox Mid-360".

Table 3.1: Comparison of the sensors Velodyne Velarray M1600, Livox Mid-40 and Livox Mid-360. The data is collected from the datasheet for respective sensor [12][25][26].

Characteristics	Velodyne Velarray M1600	Livox Mid-40	Livox Mid-360
Type of LiDAR	Solid state	Solid state	Hybrid solid
FOV Horizontal	120°	38.4°	360°
FOV Vertical	32°	38.4°	59°
Interface	Ethernet	Ethernet	Ethernet
Kernel drivers	Yes	Yes	Yes
API	C++	C++	C++
Max Range [m] (@10% reflectivity)	30	90	40
Data points [points/s]	Not available	100,000	200,000
Price* [USD]	1,500	599	749

*The price is taken from the company's recommended distributor respectively.

The 3 sensors all have in common that the interface is ethernet, kernel drivers exist and the API is written in C++, which is presented in Table 3.1. On the other hand, characteristics such as FOV, max range, data points per second, and price differs. Livox Mid-40 has the greatest max range which is 90 m at 10 % reflectivity compared to 30 m respectively 40 m. In addition, Mid-40 is slightly cheaper than Mid-360 and considerably cheaper than Velarray M1600. Nevertheless, Livox Mid-360 outperforms the other two in terms of FOV in both orientations and generates more data points per second.

3.3 Sensor choice

The 3D LiDAR sensor for this project was decided upon the Livox Mid-360 based on several strengths. Firstly, Mid-360 uses a interface that is compatible with CrossControl's display computers. Moreover, kernel drivers exist for the device and an API written in C++, both of which enables a smooth integration. The max range of the sensor is quite good with 40 m at 10% reflectivity and the price tag is reasonable. Compared with sensors in the same price range, Mid-360 performs extremely well in FOV, with 360° horizontally and 59° vertically. In the Figure 3.2 is a comparison of the FOV coverage between Livox Mid-360 and LiDAR sensors using spinning scanning technique with different number of lines (lasers) [12].

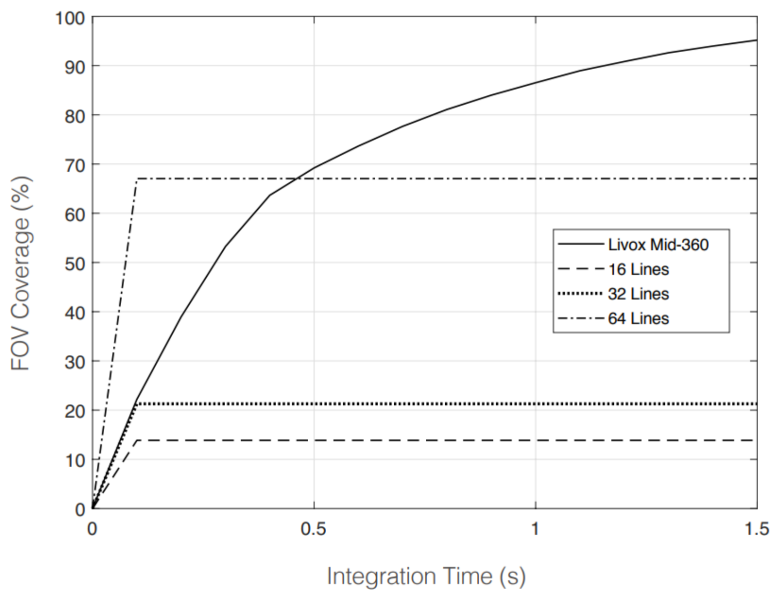


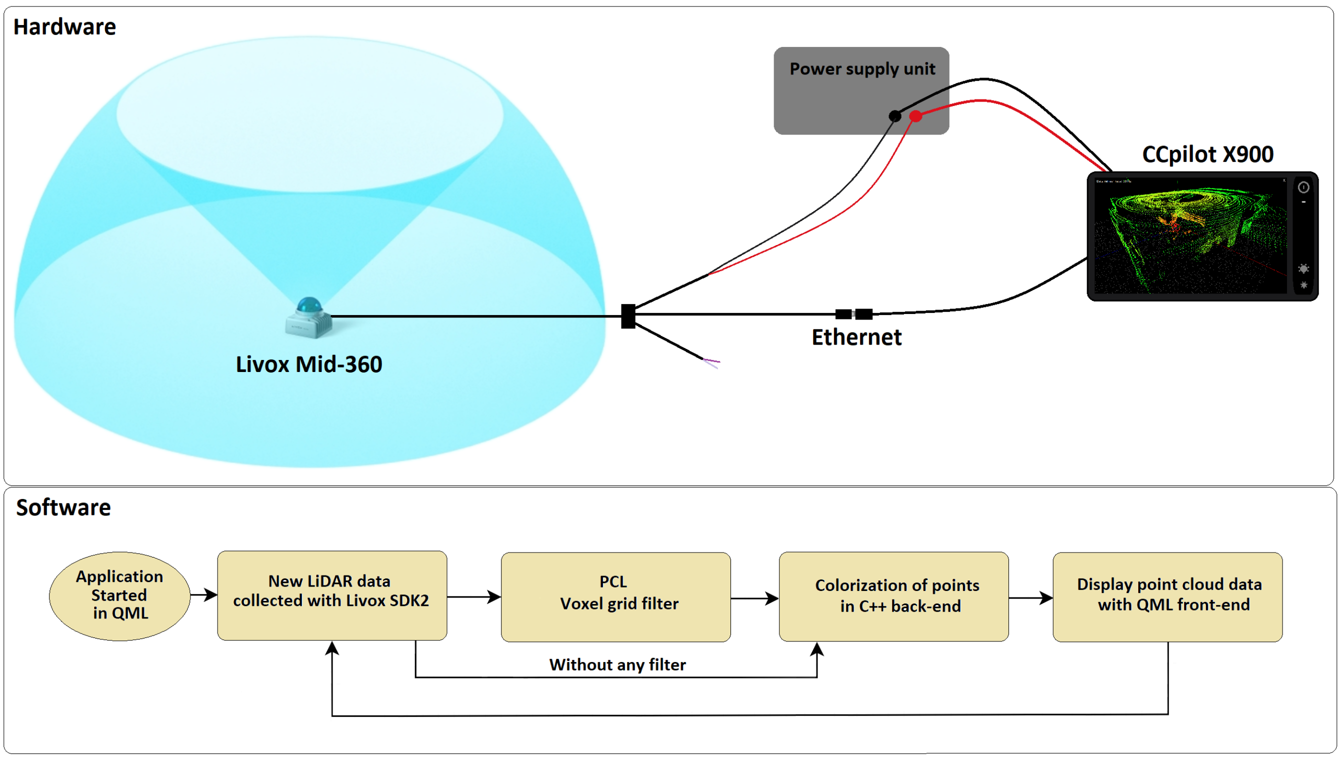
Figure 3.2: A comparison of FOV converage between Livox Mid-360 and 3 different LiDAR sensors using spinning scanning technique with 16, 32 respectively 64 lines. Collected from User Manual for Livox Mid-360 [12].

It is clear while studying the graph that Mid-360 has a greater FOV coverage compared with the spinning technique after 0.5 seconds. Furthermore, the collected data points per second can reach up to 200,000, which is critical in order to create a representative point cloud of the surrounding environment (combined with the FOV). The sensor also has a built-in IMU to keep track of the sensors position and angle relative to the environment [12]. To sum up, all of the above mentioned beneficial characteristics contributed to the sensor selection of Livox Mid-360.

4 Methods and implementation

4.1 Overview of the system

In the Figure below is an overview of the system, both hardware and software are represented. The hardware system overview is represented with the Livox Mid-360 LiDAR to the left connected, through Ethernet, to the display computer CCpilot X900 to the right. Both devices are plugged into a power supply unit. The software overview is represented with a simplified block scheme over the implemented code. Firstly, the application is started with Qt Modeling Language (QML) and LiDAR data is collected by utilization of the Livox SDK2. A voxel grid filter from Point Cloud Library (PCL) downsamples the point cloud data, or the program goes directly to the next block when no filter is used. Next, the colorization of the points is done in the C++ back-end and the LiDAR data is displayed with the QML front-end. This process is repeated when new point cloud data is collected by the LiDAR sensor.



Hardware
Power supply unit
CCpilot X900
Livox Mid-360
Ethernet
Software
Application Started in QML
New LiDAR data collected with Livox SDK2
PCL Voxel grid filter
Colorization of points in C++ back-end
Display point cloud data with QML front-end
Without any filter

Figure 4.1: Overview of the system.

4.2 Hardware and components

4.2.1 Livox Mid-360

The Livox Mid-360 using a hybrid solid state technology, which is built upon a rotating mirror in a solid state manner. The dimensions of the senor are 65×65×60 mm (width×depth×height). As mention in earlier Section 3.2, the sensor has a FOV 360° horizontally and 59° vertically, apart from a built-in IMU sensor. The max range of the sensor is 40 m at 10% reflectivity and 70 m at 80%

reflectivity. The number of points collected by the LiDAR is 200,000 points per second [12].

Moreover, the sensor is connected with a M12 A-Code aviation connector, which is spitted into 3 cables. One cable for power supply, another for Ethernet data transmission and a third for additional functionality. The operating power is 9 V to 27 V, with a recommended working voltage of 12 V. The power consummation is 6.5 W, but can reach up to 18 W in the start-up process. The sensor is controlled with a provided SDK, called “Livox SDK2” (described further in Section 4.3.4). The sensor is of laser safety Class 1 and the laser has the wavelength of 905 nm [12]. The full data sheet can be found in References list [12]. A illustration of the Livox Mid-360 is given in Figure 4.2.

Figure 4.2: LiDAR sensor Livox Mid-360 [12].

4.2.2 CCpilot X900

The CCpilot X900 is a 9" display computer that runs a Intel Atom processor, with a storage capacity of 8 GB and a RAM of 4 GB. The GPU, Graphics Processing Unit, is Intel’s HD Grapic (9th gen, Apollo Lake) integrated processor unit, with 400 MHz Base Frequency. The Thin-Filmed Tranisitor (TFT) display is covered with temepred clear glass and supported with Light Emitting Diodes (LED) backlight. The device is equipped with different connections including, Ethernet, CAN bus (Controller Area Network), and USB [27].

The CCpilot X900 supports advanced tools, such as multiple digital cameras streams, instrumental display, machine control through a Human-Machine Interface. The operating system (OS) supported are CrossControl’s customized Linux version “CCLinux” and Windows. Moreover, multiple program language are supported including, C++, C, QML, JavaScript, Python and HTML5 [27].

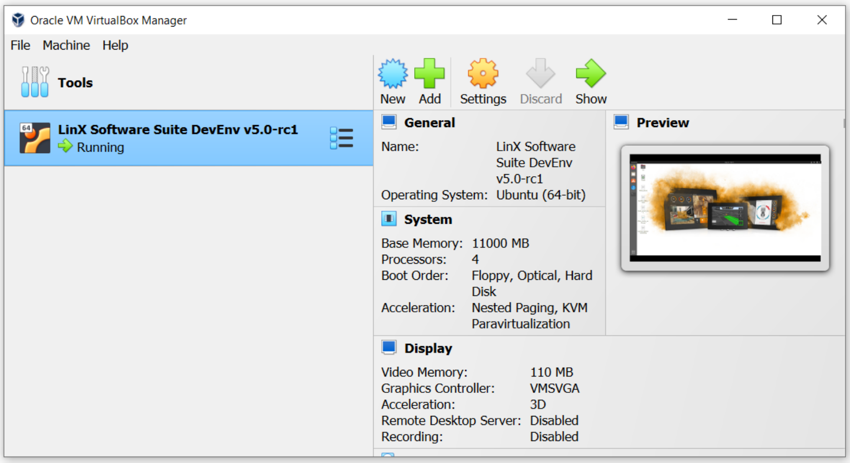
Figure 4.3: Display computer CCpilot X900 [27].

4.3 Software and development tools

4.3.1 VirtualBox and LinX Software Suite

A potent software development tool used in this project was the “Orcale VM VirtualBox” version 7.0, which enables cross-platform visualization of different OS on the same machine [28]. In other words, it extends the computer’s possibility to run other OS simultaneously in a virtual environment, often called “virtual machine” (VM). In the VirtualBox Manager one can handle the number of processors, the RAM memory and the storage space used by the virtual machine [28]. Moreover, shared folders with the real machine can be enabled and network configurations can be set in order to transmit data over Ethernet.

CrossControl’s pre-package application development kit, “LinX Software Suite”, was integrated with VirtualBox. LinX Software Suite includes tools for graphics, controls and fieldbus networking. The OS is Linux based and Qt Creator is preinstalled, further described in Section 4.3.2.



LinX Software Suite DevEnv v5.0-rc1
Running

General
Name: LinX Software Suite DevEnv v5.0-rc1
Operating System: Ubuntu (64-bit)

System
Base Memory: 11000 MB
Processors: 4
Boot Order: Floppy, Optical, Hard Disk
Acceleration: Nested Paging, KVM Paravirtualization

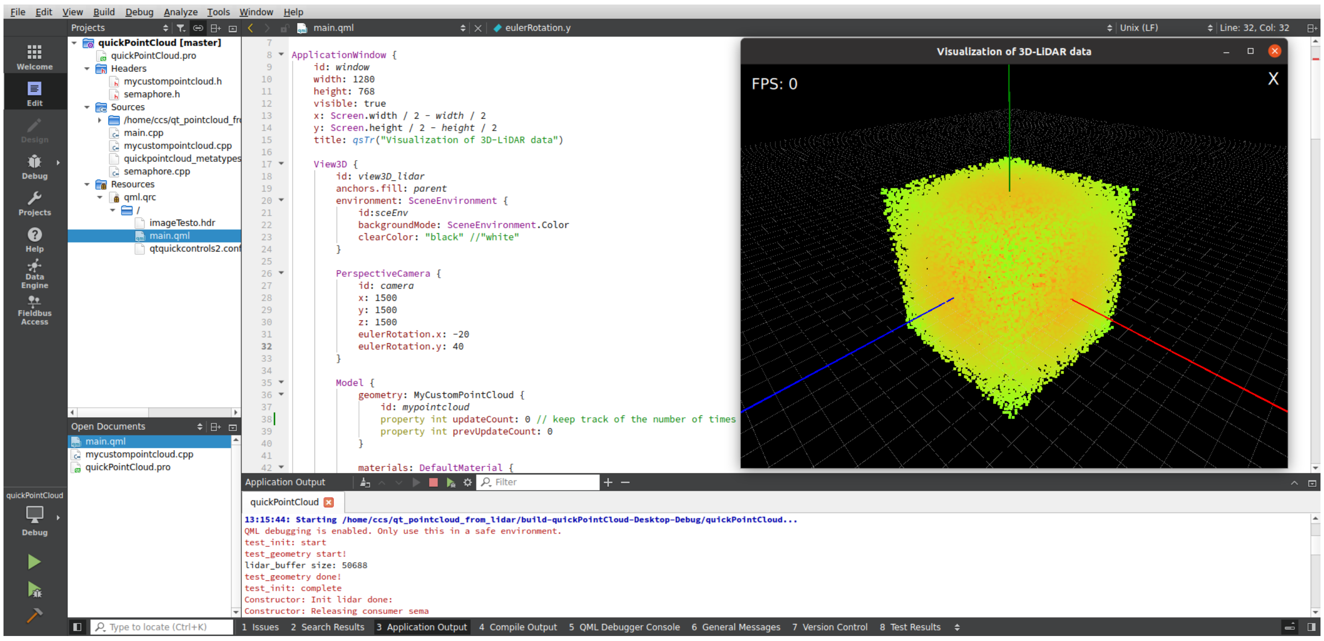
Display
Video Memory: 110 MB
Graphics Controller: VMSVGA
Acceleration: 3D
Remote Desktop Server: Disabled
Recording: Disabled

Figure 4.4: Oracle VM VirtualBox Manager with LinX Software Suite installed and running.

4.3.2 Qt Creator

In this project, Qt Creator was used as the Integrated Development Environment (IDE). The IDE serves as a valuable tool for CrossControl's users to create applications on their embedded display computers. Two build system tools for the development environment are QMake and CMake. QMake operates through a project file (".pro") to define the structure, build configuration, source files. On the other hand, CMake is a widely flexible build system used in different software projects, the project configuration is set in the file "CMakeLists.txt". Qt Creator supports programming languages such as C++ and Qt Modeling Language (QML), along with others [29]. The back-end of the application is usually C++ and QML is utilized for the creation of the Graphical User Interface (GUI). The QML provides built-in tools to develop the GUI design, besides writing code.

Furthermore, one of the most remarkable qualities of Qt Creator is the cross-platform development possibility [29]. Consequently, only one code needs to be developed for multiple instruction set architectures, for instance ARM and x86. This is highly convenient while working with CrossControl's display computers which have various CPU architectures. In Figure 4.5 the IDE is shown with the code editor in the background and an application window to the right with a test point cloud forming a cube.



```
ApplicationWindow {
    id: window
    width: 1280
    height: 768
    visible: true
    x: Screen.width / 2 - width / 2
    y: Screen.height / 2 - height / 2
    title: qsTr("Visualization of 3D-LiDAR data")

    View3D {
        id: view3D_lidar
        anchors.fill: parent
        environment: SceneEnvironment {
            id:sceEnv
            backgroundMode: SceneEnvironment.Color
            clearColor: "black" //"white"
        }

        PerspectiveCamera {
            id: camera
            x: 1500
            y: 1500
            z: 1500
            eulerRotation.x: -20
            eulerRotation.y: 40
        }

        Model {
            geometry: MyCustomPointCloud {
                id: mypointcloud
                property int updateCount: 0 // keep track of the number of times
                property int prevUpdateCount: 0
            }

            materials: DefaultMaterial {
```

```
13:15:44: Starting /home/ccs/qt_pointcloud_from_lidar/build-quickPointCloud-Desktop-Debug/quickPointCloud...
QML debugging is enabled. Only use this in a safe environment.
test_init: start
test_geometry start!
lidar_buffer size: 50688
test_geometry done!
test_init: complete
Constructor: Init lidar done:
Constructor: Releasing consumer sema
```

Figure 4.5: The Qt Creator IDE. All of the project is shown to the left, the code editor in the background and an application window to the right with a test point cloud forming a cube.

4.3.3 Point Cloud Library

A powerful open-source library for processing point cloud data and 3D geometries is the Point Cloud Library (PCL) [17]. The software framework contains several algorithms, such as filtering, surface reconstruction, and data segmentation. Some of the algorithms features is to filter outliers from noisy data, merge 3D point cloud together, extract keypoints and visualize surfaces based on a input point cloud [17]. Moreover, PCL is cross-platform, in the same manner as Qt Creator, and can be used on Linux, Windows, MacOS and Android [17]. The required third-party libraries for the PCL framework to work properly are “Boost”, “Eigen” and “FLANN”. The Boost library is utilized in order to improve memory usage by enable threading for shared pointers and Eigen is a mathematical library for essentially linear algebra operations. Lastly, FLANN performs fast calculations for the non-parametric method k-nearest neighbors in high multidimensional spaces, commonly used for classification of data clusters [17].

In this project, PCL is used to optimize the performance of the application by applying a voxel grid filter to the collected point cloud data. The filtered downsampled data preserves the structure and geometry of the point cloud with less points.

4.3.4 Livox SDK2

The software development kit “Livox SDK2” is designed to enable the utilization and integration of LiDARs developed by Livox. The C/C++ based SDK provides an API to control the LiDARs, by following the its communication protocol and transmit data over Ethernet, in order to receive a point cloud data output. Hence, the API brings C/C++ functions in order to control the LiDAR. The protocol types are a “Point cloud & IMU data protocol” and a “Control command protocol” [30]. In Figure 4.6 displays the protocol format of the Point cloud & IMU data protocol for Livox Mid-360, which is the data format sent from the LiDAR.

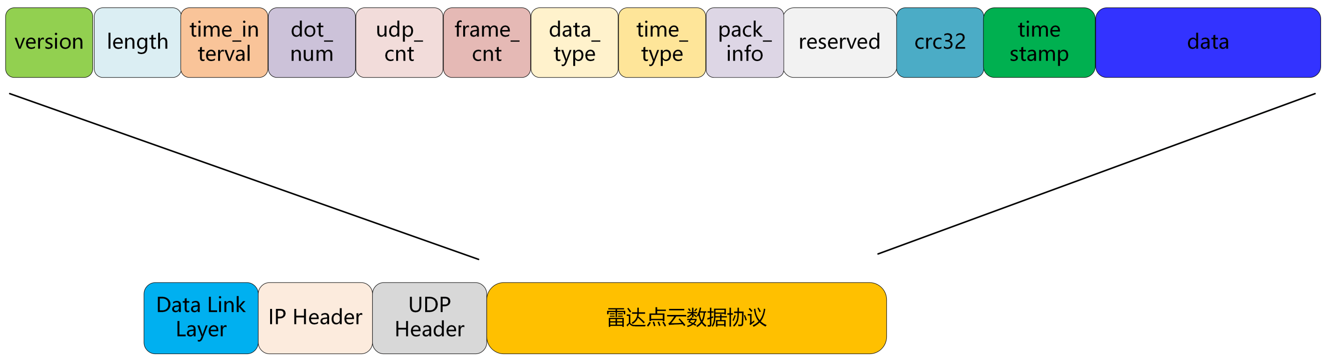


Figure 4.6: Point cloud & IMU data protocol of Livox Mid-360 [30].

The most noteworthy fields are `dot_num`, `data_type` and `data`. The field `dot_num`, with the size of 2 bytes, contains number of points in the packet. The size of `data_type` is 1 byte and determines data mode of either: IMU data, Cartesian coordinates (16 bit or 32 bit) or Spherical coordinates. The data information is stored in the field `data` and varies in size depending on the data type [30]. In this project, 32 bit Cartesian coordinates are used, which is further specified in Table 4.1. Each coordinate is of `int32_t` and gives the precision of the distance in mm.

Table 4.1: The data type Cartesian coordinates 32 bit from the Point cloud & IMU data protocol [30].

Field	Offset [bytes]	Data type	Description
x	0	int32_t	x-axis, unit: mm
y	4	int32_t	y-axis, unit: mm
z	8	int32_t	z-axis, unit: mm
Reflectivity	12	unit8_t	reflectivity value, 0-255
Tag	13	unit8_t	additional info about various measurement disturbances

On the other hand, the Control command protocol is similar to the one shown in Figure 4.6. This protocol is used to configure and query LiDAR parameters, push and query LiDAR status, reset the sensor and upgrade the sensor [30]. The interaction with the control commands are rarely used since the commands are abstracted through API utilization. Livox SDK2 can easily be accessed through Livox's Github [31].

4.4 LiDAR data access implementation

The LiDAR data access implementation was achieved in VirtualBox with CrossControl's LinX Software Suite installed.

4.4.1 First collection of LiDAR data

Firstly, the Livox SDK2 was installed on the VM with the Linux based OS. This was done by the command `git clone https://github.com/Livox-SDK/Livox-SDK2.git` from Livox's Github [31]. Secondly, the M12 cable was connected to the Livox Mid-360 for power supply apart from data transmission. However, to get the data transmission to work properly over Ethernet, a few setting was to be done in VirtualBox and in the settings of Linux OS. In VirtualBox settings, a network adapter was added, the option called "Briged Adapter". This allows VirtualBox to intercept data from the physical adapter and inject data into the VM. The data transmission occurs across a static IP address which is 192.168.1.50 with the subnet mask of 255.255.255.0. The IP address with subnet mask was set in the VMs network settings by enable manual mode.

The SDK provides a C++ test program for a quick start of the Mid-360, called "`livox_lidar_quick_start.cpp`", besides necessary configuration file, "config.json". The program utilizes built-in functions from the API, such as, `void PointCloudCallback(·)` in order to collect point cloud data from the device. The data generated, with default setting of 32 bit Cartesian coordinate data type, from the callback function is extracted to x, y and z through a for-loop. The x, y and z coordinates was printed in the terminal when executing the code. In other words, when the power cables of the M12 cable was connected to a power supply of 20-24 V, the code was executed.

An extension of the `livox_lidar_quick_start.cpp` was created to save the collected data to a `csv`-file. A new function was written, called `WritePointsToFile(·)`, which utilize the `std::ofstream` to open, write to a file and close it. The code for the function is provided below.

```
bool WritePointsToFile(std::string file_name, std::string x_point, std::string y_point, std::string z_point) {
  std::ofstream file;
  file.open(file_name, std::ios_base::app);
  file << x_point << "," << y_point << "," << z_point << std::endl;
  file.close();
  return true;
}
```

Thus, the function was called in the `void PointCloudCallback(·)`, with the command `bool writetofile = WritePointsToFile("test.csv", x_point, y_point, z_point);`, where the points were saved to `x_point`, `y_point` and `z_point`. The entire code is provided in the Appendix.

4.4.2 Voxel grid filter with LiDAR data

Since the data from the Livox Mid-360 has been collected to a `csv`-file (described in Section 4.4.1), the next step was to test the voxel grid filter supplied by PCL. The purpose of this is to determine a feasible size (Leaf size) of each voxel for the point cloud data. In PCL's documentation, there is a tutorial with a C++ voxel grid filter program, the name of the tutorial is "Downsampling a PointCloud using a VoxelGrid filter" [32]. The C++ program from the tutorial was used, where the key commands are `pcl::VoxelGrid<pcl::PCLPointCloud2> sor` to create a voxel grid object and `sor.setLeafSize (1f, 1f, 1f);` to set the size of the voxel. The voxel grid filter part of the code can be found below and the rest of the script can be found in the Appendix.

```
  // Create the filtering object
  pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
  sor.setInputCloud (cloud);
  sor.setLeafSize (1f, 1f, 1f);
  sor.filter (*cloud_filtered);
```

The test was performed by changing the parameters of `setLeafSize(1f, 1f, 1f)` with 50,000 points (46264 points after removing (0,0,0)) as input from the `csv`-file. The 3 input parameters to the function was uniformly varied between `0.8f` (0.08 cm) to `200f` (20 cm).

4.5 Application implementation

The entire application development was achieved in VirtualBox with CrossControl's LinX Software Suite installed. The application was developed with Qt Creator and built with QMake operating through a project file, called `quickPointCloud.pro`. In the `pro`-file the structure was defined, all include paths were set for the various libraries, and the Livox SDK2 was statically linked. The file `quickPointCloud.pro` can be found in the Appendix.

4.5.1 Class MyCustomPointCloud

The purpose of the application is to visualize real time data graphically, thus LiDAR data handling is needed to process and modify the data. This task is accomplished with a custom-made `class` called `MyCustomPointCloud`. The `class` is based on Qt's `QQuick3DGeometry`, which can specify a custom geometry for model in a Qt Quick 3D scene. Firstly, a header file `mycustompointcloud.h` includes all the libraries and defines all the functions in the `Class`. The Livox SDK2 is included in the header files `livox_lidar_def.h` and `livox_lidar_api.h`, where device definitions and the API for the LiDAR are defined, respectively. The functions utilized, when LiDAR data is displayed, are the following, `init_lidar()`, `cb(·)`, and `updateGeometry()`. The `class` also contains an option to display randomized test data, used when no LiDAR sensor is connected, with the functions `test_cb()`, `test_init()` and `test_geometry()`.

Furthermore, a variable of the data type `struct` is defined in order to temporary store LiDAR data, created to match generated raw data from the Mid-360 (as seen in Table 4.1). The `struct lidar_data` contains x, y, z, reflectivity, tag and pad (two extra bytes), seen below (entire code in Appendix).

```
// LiDAR data structure
typedef struct {
   int32_t x;
   int32_t y;
   int32_t z;
   uint8_t ref;
   uint8_t tag;
   uint16_t pad;
} lidar_data;
```

Moreover, the `Class` is set as a `QML_ELEMENT` allowing it to be accessed and manipulated directly within the QML code, which will be important when updating to new LiDAR data (described in more detail below). `MyCustomPointCloud` also defines configuration path to the file `lidar_config.json` that configures the static IP address and ports according to the Control command protocol, equally as in Section 4.4.1.

4.5.2 Semaphore

Before describing how the LiDAR data is updated and displayed, another key implementation of the project is the "`semaphore`" for handling data. A semaphore enables control access for synchronization of shared resources and data among multiple processes. In other words, it ensures that no data get missing when storing LiDAR data into a buffer. Two fundamental operations by the semaphore are defined in `semaphore.h` and `semaphore.cpp` (found in Appendix), which are `acquire()` and `release()`. The operation `acquire()` is utilized when a process wants access to the shared data. In the case when the value of the semaphore is greater than zero, the value will decrement by 1. If the value is zero, the process is blocked, and will wait until the semaphore value is greater than zero again. On the other hand, `release()` is used when the shared data is made available and increment the semaphore value by 1. This potentially wakes up any waiting process blocked by the semaphore. The size of the buffer is determined with the `BUF_SIZE` parameter in `mycustompointcloud.h`, and is often referred to input point cloud size in this thesis.

4.5.3 Collecting LiDAR data

The function for initializing the LiDAR, in order to collect real time data, is `init_lidar()` from the C++ program `mycustompointcloud.cpp` (entire code in Appendix). The function verifies that the configuration json-file for the sensor is included and creates a function pointer `cbfuncptr lidar_cb`. The function pointer is stated to be able to access the callback function initialization through `SetLivoxLidarPointCloudCallBack(lidar_cb, nullptr)`. The callback function `cb(·)` is then set, which is an extension of the `PointCloudCallback(·)`, described in Section 4.4.1. The data x, y, z, reflectivity, and tag is collected from the LiDAR and stored into an array called `lidar_buffer`, with the element type of `lidar_data` and the size of the array is the determined `BUF_SIZE`. When the `lidar_buffer` array is of the size `BUF_SIZE`, the production semaphore is released (`sem_prod.release()`) and waits for a consumer semaphore (`sem_cons.acquire()`). The collection of test data, by using `test_cb()`, works in a similar way but the data is randomly generated.

4.5.4 Filtering real time data

The most central member of the `class MyCustomPointCloud` is the function `updateGeometry()` in the C++ code `mycustompointcloud.cpp`. The function `updateGeometry()` handles, among other things, the voxel grid filter for real time LiDAR data. Firstly, the semaphore buffer is verified if it is accessible with the code `sem_prod.acquire()`, otherwise it will wait until it is accessible. Then a preparation for the filter is done by transforming the data from the `lidar_buffer` to a PCL point called `PointXYZRGB point`. The point cloud is then converted to a `PCLPointCloud2`, then filtered, and converted back to a `PointCloud<pcl::PointXYZRGB>`. The voxel grid filtering process is identical to the code provided in Section 4.4.2, except that the leaf size was set to 75.0f (7.5 cm). The filtered point cloud is stored into `cloud_rgb` PCL object.

4.5.5 Pre-processing and colorization of point cloud data

The function `updateGeometry()` also has the important task to pre-process the filtered point cloud data, with the purpose of transforming it to a `QQuick3DGeometry` object. Hence, the `QQuick3DGeometry` object can be integrated with the Qt Quick 3D scene in the QML code, which is described in more detail in Section 4.5.6. The built-in function `update()` is called in order to notify the underlying Qt system that the widget need to be repainted in the GUI. The pre-processing starts when a `QByteArray vertexData` is created and resized to the size of the LiDAR data and the color of the points. The bytes used by the coordinates is 3 (x, y, z) and for the colorization of the points is 4 (one bytes for each in RGBA), which sums up to 7. Therefore, the size of the `vertexData` becomes 4 times 7 times `BUF_SIZE` bytes, since the size of a float is 4 bytes. The following line `float *p = reinterpret_cast<float *>(vertexData.data())` creates a float pointer `p` and assigns it the memory address of the data stored in `vertexData`. The `reinterpret_cast` enables float type data to be added to the `vertexData`. The most essential parts of the pre-processing and the colorization of the points are given below:

```
void MyCustomPointCloud::updateGeometry()
{
    //Built-in function that signals that the graphics will update in QML:
    update();
```

```
//Waiting for production semaphore
sem_prod.acquire();

/*
Here is the voxel grid filter in the complete code
*/

QByteArray vertexData;
//Resizeing the QByteArray according to size of LiDAR data, 3 for coordinates and 4 for color
vertexData.resize(sizeof(float) * 7 * cloud_rgb->width);
//Create a pointer to access the data
float *p = reinterpret_cast<float *>(vertexData.data());

//Iterating for storing points and color in the pointer p:
for (int var = 0; var < cloud_rgb->width; ++var)
{
    //Access the point cloud data:
    pcl::PointXYZRGB point = cloud_rgb->at(var);
    //Add the points to the pointer:
    *p++ = point.x;
    *p++ = point.z; //"y" is set to z for the axis transformation between sensor and QML
    *p++ = -point.y; //"z" is set to -y for the axis transformation between sensor and QML

    float r, g, b, distance, lidar_ref;

    //Calculate the distance from origo:
    distance = std::sqrt(point.x * point.x + point.y * point.y + point.z * point.z);

    //Check if colorization by distance or reflection (from the LiDAR mode):
    #ifdef LIDAR_DIST
        //Set color to red when short distance and gradually gets more green further away from origo:
        r = 1-distance/3000+0.2;
        g = distance/3000-0.2;
        b = 0.0;
    #endif

    #ifdef LIDAR_REF
        lidar_ref = point.r;
        /*
        Code for setting the color according to the reflectivity
        */
    #endif

    //Add the color to the pointer
    *p++ = r;
    *p++ = g;
    *p++ = b;
    *p++ = 0.9f; //alpha value
}

//Set the data of the QQuick3DGeometry object:
setVertexData(vertexData);
//Set the point type to "points"
setPrimitiveType(QQuick3DGeometry::PrimitiveType::Points);
```

```
    //Set the size of each data point, 3 for coordinates and 4 for color
    setStride(7 * sizeof(float));
    //Add an attribute for the position (x,y,z), no offset, type float:
    addAttribute(QQuick3DGeometry::Attribute::PositionSemantic,
    0,
    QQuick3DGeometry::Attribute::F32Type);
    //Add an attribute for the color (r,g,b,a), 3 offset, type float:
    addAttribute(QQuick3DGeometry::Attribute::ColorSemantic,
    3 * sizeof(float),
    QQuick3DGeometry::Attribute::F32Type);

    //Releasing consumer semaphore
    sem_cons.release();
}
```

Next, a for-loop iterates over all points from the filtered point cloud data and is accessed through `pcl::PointXYZRGB point = cloud_rgb->at(var)`. The coordinates for x, y and z are added to the pointer. Notice, the y coordinate is set to z and the z coordinate is set to -y. The objective of doing this is to match the coordinate system from the LiDAR sensor to the Qt application. The axis transformation is illustrated in the Figure 4.7.

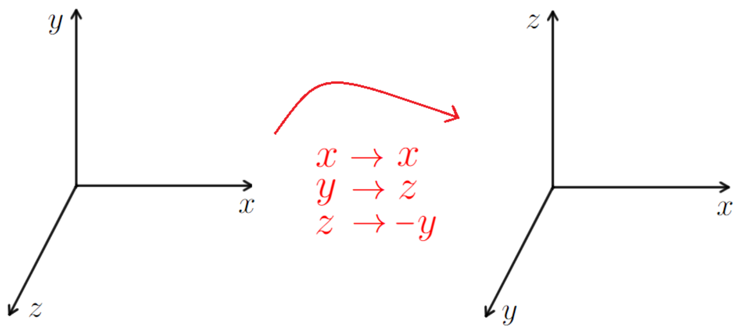


Figure 4.7: Axis transformation between Qt's standard axis to the coordinate system provided by Livox Mid-360. To the left is the standard coordinate system in Qt and to the right is the transformed system.

There are two options for colorizing the points, either based on the distance from the sensor or based on the reflectivity value generated by the Mid-360. These options are called `LIDAR_DIST` and `LIDAR_REF`, and the option choice is made at the beginning of `mycustompointcloud.cpp`. The distance colorization is set with the distance formula given in Equation 2.3, where red represents short range and gradually gets more green further away from the sensor. On the other hand, the reflectivity colorization is based on an integer reflection value from 0-255 given by the sensor. A lower reflectivity value is represented with blue and gradually increases to green, yellow, orange and red, in that specific order. The colors are, in both cases, set with the pointer `p` for the 4 RGBA values.

Lastly, the data of the `QQuick3DGeometry` is set with the line `setVertexData(vertexData)`. Moreover, the primitive type (shape) of the data is set to "Points" and the stride is set to 28 ($7 \cdot 4$), which is the byte distance between two consecutive data points in the `vertexData`. The attributes are added for the position and color, in order to specify the usage (point or color), the offset bytes and data type for `QQuick3DGeometry`. The last thing done by the function is to releasing the consumer semaphore, which unblock the semaphore and new data can be stored in the buffer (done in the `cb(·)` function).

4.5.6 Front-end of the application

The front-end of the application is developed in QML, in a file called `main.qml`. In the file `main.cpp`, the standard start up settings for the QML engine of the application are located, and `main.qml` is initialized there (`main.cpp` can be found in Appendix). In `main.qml`, the LiDAR point cloud data is sent through a custom geometry model, done with `QQuick3DGeometry`, to be able to show the data in a Qt Quick 3D scene. The modules imported in QML-file are `QtQuick` and `QtQuick3D`. Initially, the application window is set, with pixel width, pixel height, and positioning on the screen. The object `View3D` sets up the 3D environment of the application. The 3D environment is anchored to fill the entire application window and a black background is set with the following code:

```
    View3D {
        id: view3D_lidar
        anchors.fill: parent //Anchord to the entire application window
        environment: SceneEnvironment {
            id: sceEnv
            backgroundMode: SceneEnvironment.Color //The backgroud mode of the 3D view
            clearColor: "black" //Backgroud is set to black
        }
```

The object `PerspectiveCamera` controls the user's perspective of the 3D environment. A initial position and rotation of the camera is determined. The camera can be moved in the 3D space with WASD-controls and rotated with the mouse or finger, depending on the device used. Axis is also added to the 3D space, with object `AxisHelper`, where the red axis represent the x-axis, the y-axis is blue and the z-axis is green (after the axis transformation illustrated in Figure 4.7).

A `Model` object calls the customized geometry `MyCustomPointCloud` created in the C++ back-end, described in the Section 4.5.5. The default materials of the points are set to the color white, with no lightning and point size 4 in the code below. The color is set to white with no lightning to be able to set the color properly from the back-end code.

```
Model {
    geometry: MyCustomPointCloud {
        id: mypointcloud
        property int updateCount: 0 //Keep track of the number of times updateGeometry() is called
        property int prevUpdateCount: 0 //Previously number of updateGeometry() was called
        }
        //Pre-setting all point colors to white with no lightning
        materials: DefaultMaterial {
        lighting: DefaultMaterial.NoLighting
        diffuseColor: "white"
        pointSize: 4 //Size of the points
      }
    }
```

The most critical QML code is a `Timer`, called “`updateTimer`”, which is constantly running and triggers every 25 ms. When the `Timer` is triggered it sends a signal to the function `updateGeometry()` in the back-end code. This is made possible by utilizing the Qt’s “`Q_INVOKABLE`” for the `updateGeometry()` function, enabling the invocation of a C++ function from the QML code. Note that the update of the point cloud only happens if the function `updateGeometry()` can acquire the LiDAR data through the semaphore. The code for the `updateTimer` is provided below:

```
Timer {
      id: updateTimer
        running: true
        interval: 25
        repeat: true
        onTriggered: {
            mypointcloud.updateGeometry()
            mypointcloud.updateCount = mypointcloud.updateCount + 1
            }
    }
```

The command `mypointcloud.updateCount = mypointcloud.updateCount + 1` determines the data refresh rate parameter. Every time the point cloud is updated, the data refresh rate is incremented by 1 during 1000 ms. The previously update count is then subtracted from the update count, hence the variable `updateDiff` is displayed. The data refresh rate is displayed in the upper left corner with the following line, `dataRefreshRateText.text = "Data refresh rate:  " + updateDiff + " Hz"`. The entire code for `main.qml` is provided in the Appendix.

4.5.7 Flow chart for the code

Here is a flow chart (Figure 4.8) showing how the code works in brief.

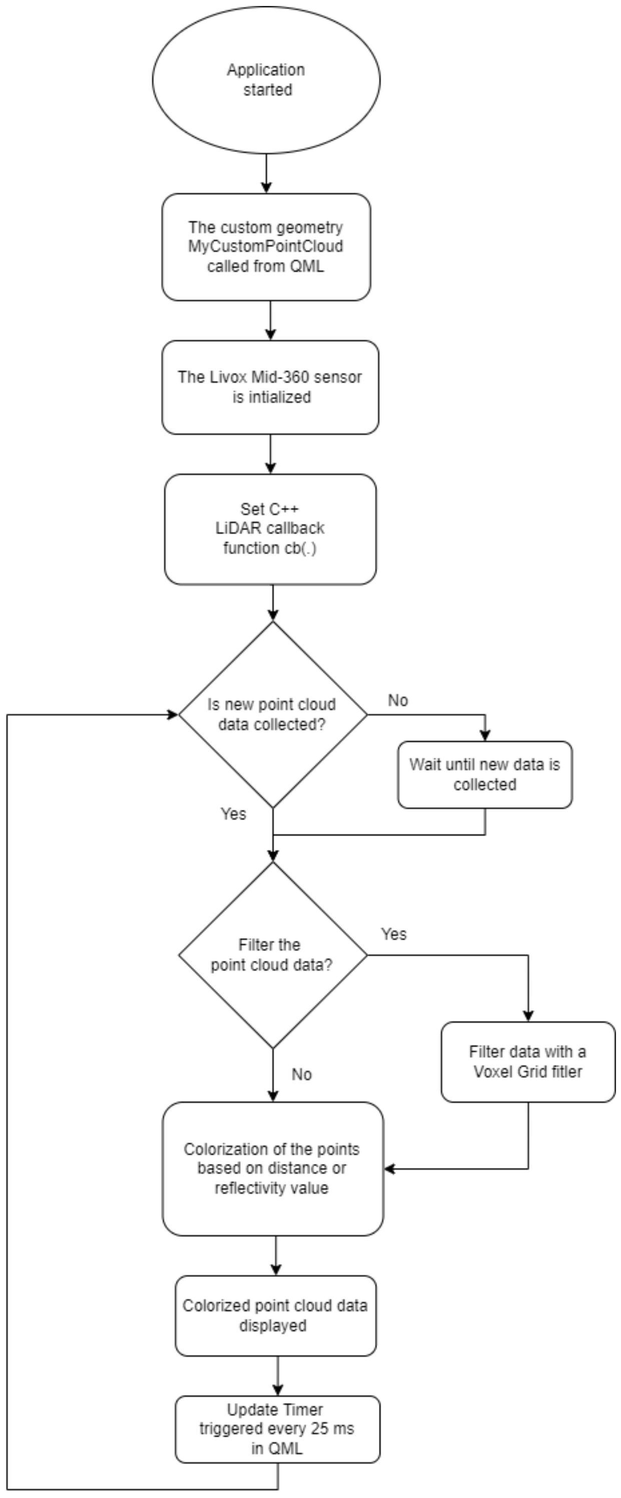

Application started
The custom geometry MyCustomPointCloud called from QML
The Livox Mid-360 sensor is intialized
Set C++ LiDAR callback function cb(.)
Is new point cloud data collected?
No
Wait until new data is collected
Yes
Filter the point cloud data?
Yes
Filter data with a Voxel Grid fitler
No
Colorization of the points based on distance or reflectivity value
Colorized point cloud data displayed
Update Timer triggered every 25 ms in QML

Figure 4.8: A Flow chart of the code. Created in "draw.io".

4.5.8 Cross compilation and setup

The display computer CCpilot X900 operates on a different OS than the VM, where the application is developed. Thus, a cross compilation is needed in order to launch the application on CCpilot X900. The cross compilation was performed with CrossControl's custom-made SDK for the device. A specific QMake was selected with the following command, `/opt/XM9/Qt-6.4.1/bin/qmake` and then the command `make` was executed. This resulted in a executable file, which was sent to the display computer with the Linux command `scp`. In the Figure 4.9 is the setup of the CCpilot X900, the Mid-360 LiDAR sensor, a power supply unit and various wires.

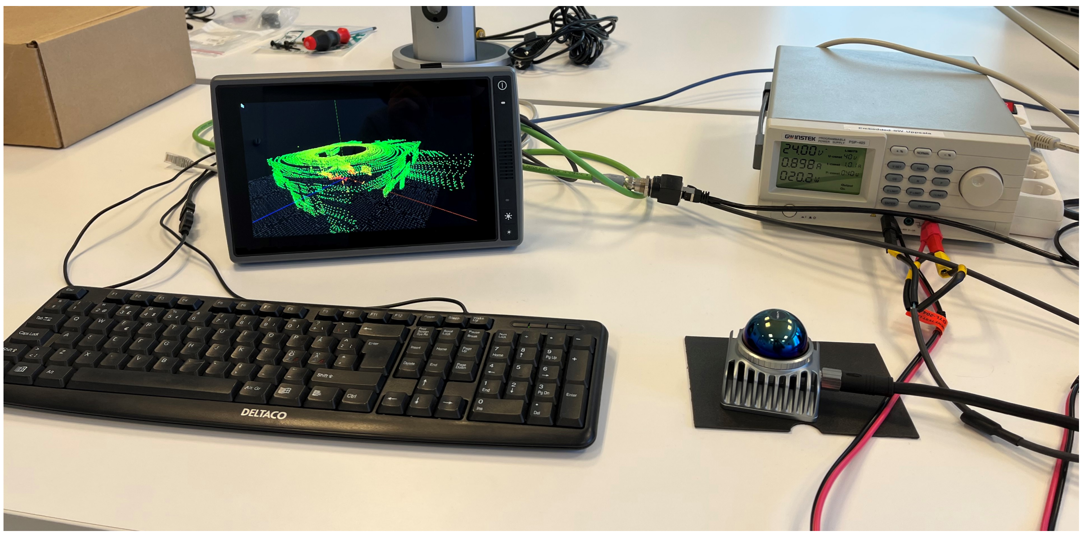


Figure 4.9: The CCpilot X900 display computer to left presenting point cloud data from the Mid-360 LiDAR to the right.

4.6 Performance evaluation

In order to investigate the performance of the application, measurements were needed while operating the application on the display computer. The processor load was measured with the Linux command `top` for different numbers of input point cloud sizes (`BUF_size` parameter was varied). The following command was utilized in order to save CPU load data to a txt-file:

```
top —b —n 300 —d 0.2 | grep ./quickPointCloud_text_top>/opt/quickPointCloud/log/test_log_XX.txt
```

The option `-b` disables the interactive interface of `top`. The CPU load data was captured during 300 iterations with 0.2 s delay, with the options `-n 300 -d 0.2`. Hence, the application was evaluated during 60 seconds. The logged data was processed and an average of the CPU load was calculated. Moreover, the minimum CPU load and maximum CPU load was derived from the data. The default setting of `top` on the CCpilot X900 is to setting "Irix mode" to off, which implies that the `%CPU` value is taken across all 4 cores. In other words, the cumulative CPU usage can reach 400 % if all cores are used to its fullest extent (unlikely to occur since another factor could be a bottleneck).

Another performance evaluation was carried out, by observing the data refresh rate for different numbers of input point cloud sizes (`BUF_size` parameter was varied). The objective was to find out if any latency occurred. Consequently, if the input point cloud size is set and data refresh rate is observed, then number of points per second can be calculated by simply multiplying the two variables. The result can be compared with the give number of points per second given in the Livox Mid-360 datasheet, which is 200,000 points per second. The application was launched on the display computer, for varied sizes of the `BUF_size` parameter, and the data refresh rate was registered. For a few cases the data refresh rate did fluctuate between two values. In that event, the registered value was observed to be ".5" between the two.

5 Results and Discussions

5.1 Voxel grid filter with LiDAR data

The test of the voxel grid filter in order to determine the leaf size (voxel size) is presented within this section. In the Figure 5.1 is a graph showing the number of downsampled points relative the leaf size after filtering. The number of points before filtering is 46264 points and the leaf size is varied between 0.08 cm to 20 cm.

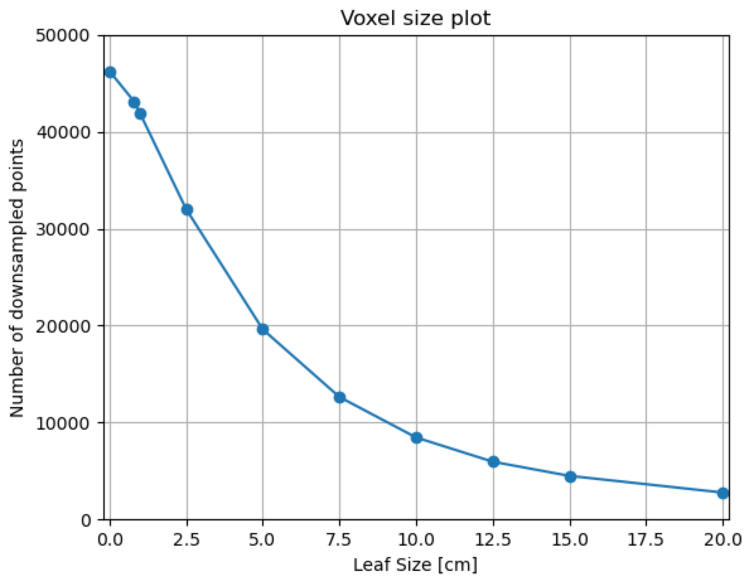


Figure 5.1: A graph showing the number of downsampled points relative the leaf size after filtering. The number of points before filtering was 46264 points and the leaf size is varied between 0.08 cm to 20 cm.

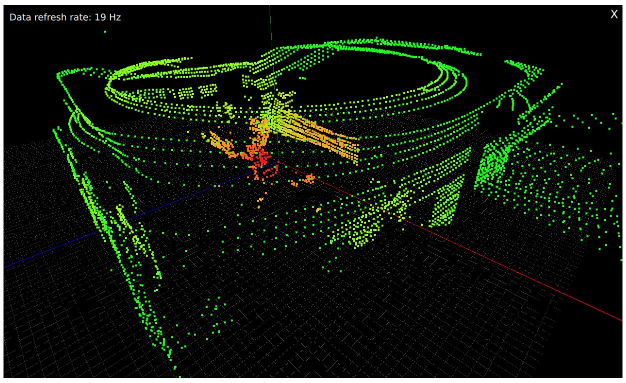
Based on the graph, it is clear that the number of points decreases with a greater leaf size. A low value of the leaf size does not have great impact of number of points, in other words, only a few points are removed by the filter. On the other hand, a large value does almost downsample all points. Therefore, the leaf size was set to 7.5 cm which is situated in the middle spectrum, resulting in 12626 downsampled points. In addition, the leaf size of 7.5 cm was confirmed to represent the surrounding accurately in the result presented in Section 5.5.

5.2 Application

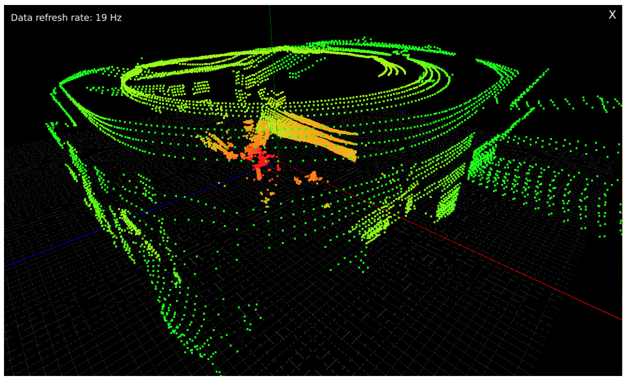
Two videos of the application, operating on the CCpilot X900, are provided in the caption for Figure 5.6 and Figure 5.7. The colorization of the points is relative the distance to the LiDAR sensor or based on the reflectivity value.

5.2.1 Comparison between voxel grid filter and without a filter

The graphical visualization is evaluated here between the voxel grid filter and no filter utilized. This is done by comparing 3 diverse point sizes of 10.6k, 21.1k, and 52.8k points, respectively. The Figures 5.2, 5.3 and 5.4 are showing one snapshot of the application with the voxel grid filter to the left and another snapshot to the right with no filter. The colorization of the points is relative the distance from the LiDAR.

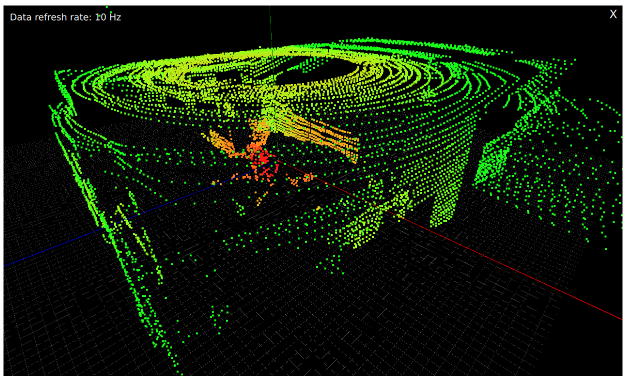


(a) Voxel grid filter, input 10.6k points.

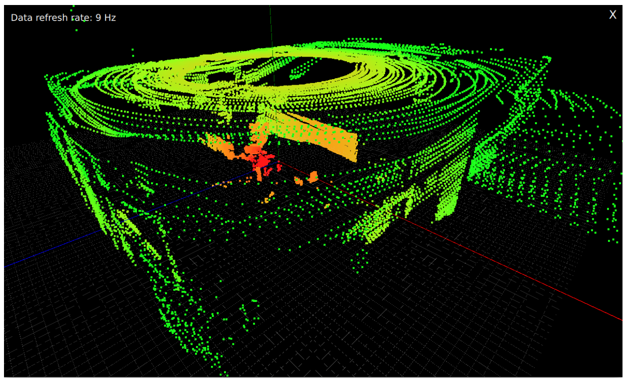


(b) No filter, input 10.6k points.

Figure 5.2: Two snapshots of the application, the voxel grid filter is utilized to the left and no filter is utilized to the right. The number of input points is 10.6k points and the points were colorized according to the distance from the sensor. Short distance is represented with red and gradually gets more green further away from the sensor.

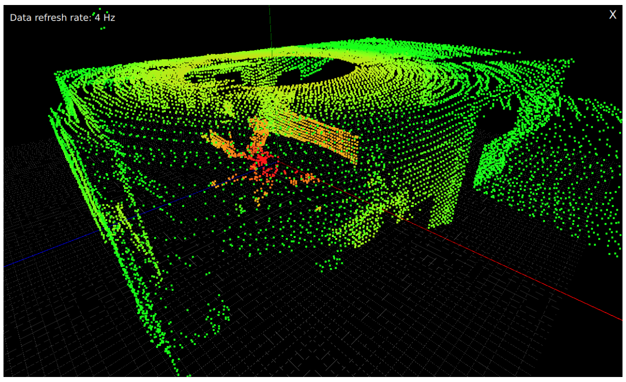


(a) Voxel grid filter, input 21.1k points.

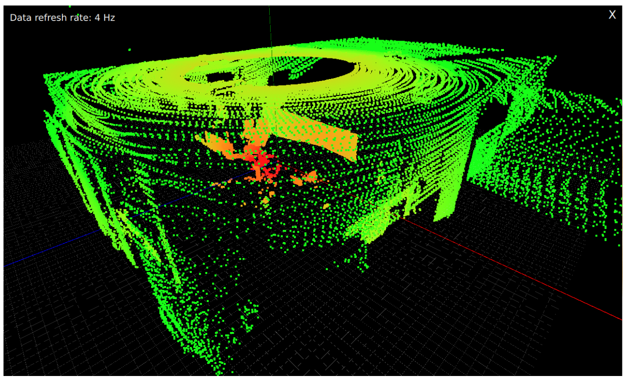


(b) No filter, input 21.1k points.

Figure 5.3: Two snapshots of the application, the voxel grid filter is utilized to the left and no filter is utilized to the right. The number of input points is 21.1k points and the points were colorized according to the distance from the sensor. Short distance is represented with red and gradually gets more green further away from the sensor.



(a) Voxel grid filter, input 52.8k points.



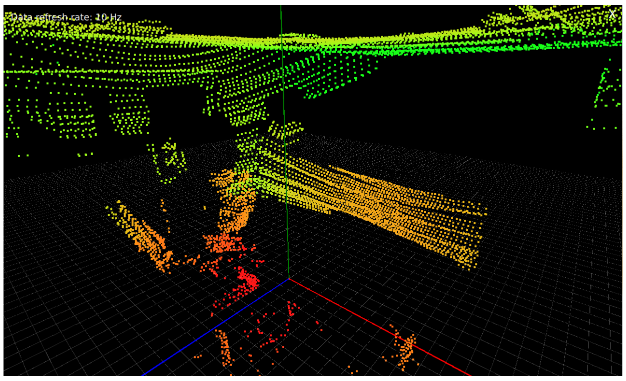
(b) No filter, input 52.8k points.

Figure 5.4: Two snapshots of the application, the voxel grid filter is utilized to the left and no filter is utilized to the right. The number of input points is 52.8k points and the points were colorized according to the distance from the sensor. Short distance is represented with red and gradually gets more green further away from the sensor.

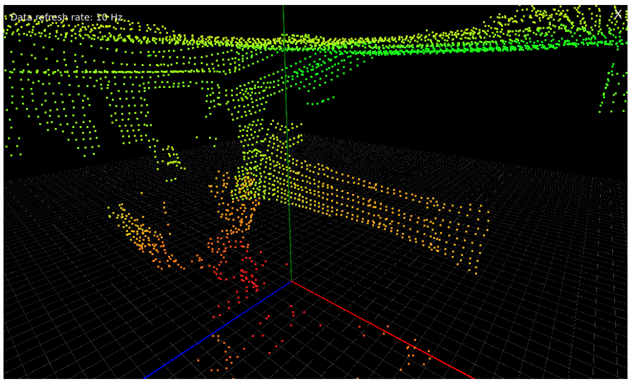
It is difficult to distinguish any major visual difference between the filtered point cloud and the non-filtered data. Despite that, more points intersect when no filter is used. In few cases, without the utilization of a filter, points further away from the camera is obscured of the high dense clusters of points. This occurs to a lesser extent with the voxel grid filter. As a result, the voxel grid filter does keep the structure and geometry of the input point cloud when utilizing less points.

5.2.2 Voxel grid filter comparison

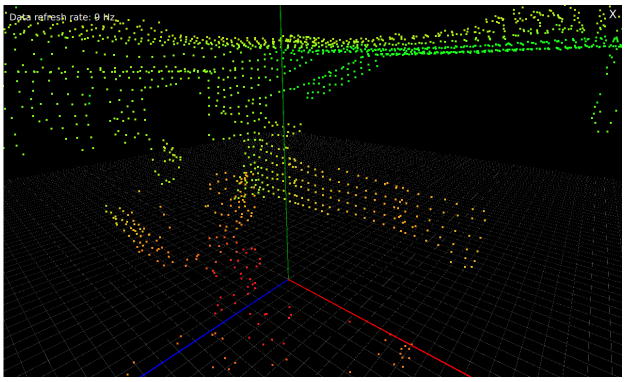
A visual comparison between different voxel grid filter leaf sizes of 2.5, 7.5 and 12.5 cm was made. A snapshot of the application, for each leaf size, was taken in the same camera angle and is shown in Figure 5.5.



(a) Leaf size of 2.5 cm.



(b) Leaf size of 7.5 cm.



(c) Leaf size of 12.5 cm.

Figure 5.5: Visual comparison between different voxel grid filter leaf sizes of 2.5, 7.5, and 12.5 cm, when filtering a input point cloud of 21.1k points. The points were colorized according to the distance from the sensor. Short distance is represented with red and gradually gets more green further away from the sensor.

The snapshot presented in Figure 5.5a (leaf size 2.5 cm) is showing a highly dense point cloud, where points intercept with each other. On the other hand, Figure 5.5c (leaf size 12.5 cm) loses a lot of information, thus the filter removes too many points and does not represent the surrounding accurately. The leaf size 7.5 cm, shown in Figure 5.5b, is a trade-off between the previously mentioned leaf sizes of 2.5 and 12.5 cm. Hence, points are not intercepting and the geometrical structure from the point cloud before filtering is preserved with the leaf size of 7.5 cm.

5.2.3 Final version of the application

The final application, operating on the CCpilot X900, is presented here with two ways of colorize the point cloud. The colorization of the points is relative the distance to the LiDAR in Figure 5.6. The reflectivity value from the sensor determines the point color in Figure 5.7. The LiDAR sensor is placed where the 3 perpendicular axis meet. The data refresh rate in Hz is displayed in the upper left corner and a shut down button can be found in the upper right corner (a white cross). A video of the application is created with “Screen2gif” and is provided in the caption of figure, respectively.

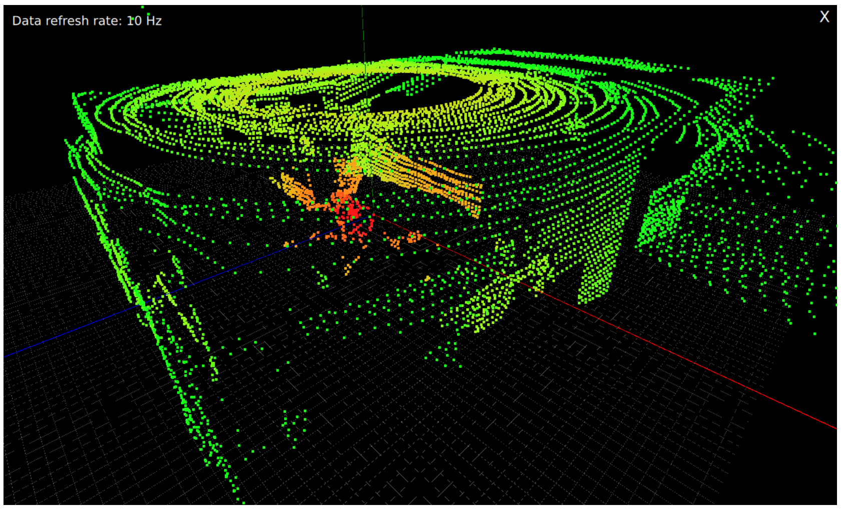


Figure 5.6: The final application where the colorization of the points are relative the distance to the LiDAR sensor. Short distance is represented with red and gradually gets more green further away from the sensor. The number of input points is 21.1k points. On the following link can a video of the application be found: Video link.

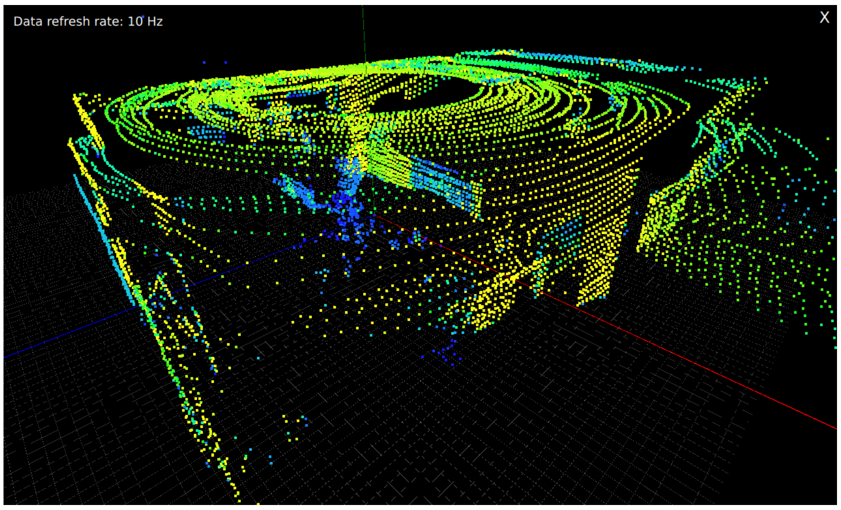


Figure 5.7: The final application where the colorization of the points are based on reflectivity value from the sensor. A lower reflectivity value is represented with blue and gradually increases to green, yellow, orange and red, in that specific order. The number of input points is 21.1k points. On the following link can a video of the application be found: Video link.

This result fulfills the objective of the degree project, which is to develop a application for a sensor that operates on one of the vehicle computers, where real time data is visualized graphically.

5.3 Performance evaluation

In this section, the performance evaluated of the application is presented, based on the processor (CPU) load and data refresh rate for different input point cloud sizes (size of `BUF_size`). All performance evaluation was executed on the CCpilot X900 device.

5.3.1 Processor load

The measurements of the processor load is divided into 3 parameters, CPU average, CPU min and CPU max. The average is taken during 60 seconds while the application is launched on the display computer. The minimum and maximum are derived under the same period. Two graphs, showing the processor load over input point cloud size, are presented in Figure 5.8 and Figure 5.9 for the voxel grid filter and the case when no filter is used, respectively.

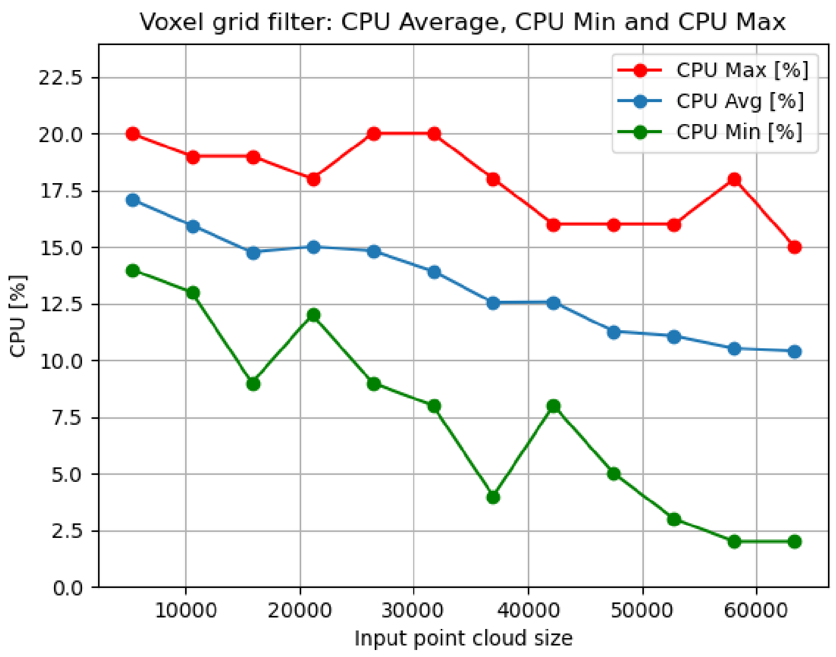


Figure 5.8: The processor load of the CCpilot X900 while operating the application, utilizing the voxel grid filter. The graph shows the correlation between CPU load and number of input point cloud size. The red graph is the maximal CPU usage, the blue graph represent the CPU average, and the green graph shows the minimal CPU usage.

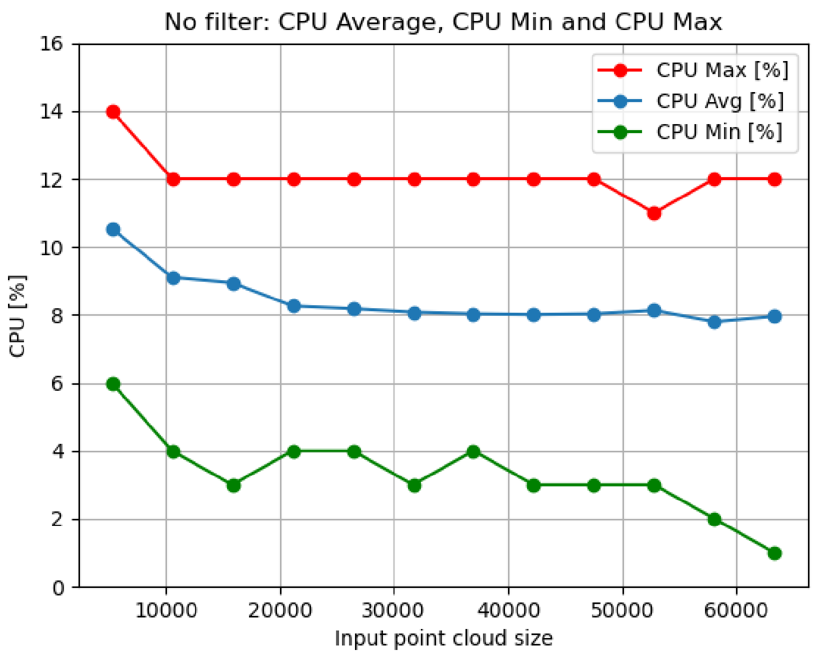


Figure 5.9: The processor load of the CCpilot X900 while operating the application, when no filter is used. The graph shows the correlation between CPU load and number of input point cloud size. The red graph is the maximal CPU usage, the blue graph represent the CPU average, and the green graph shows the minimal CPU usage.

The difference between CPU usage is clearly visible in the two graphs above. All three CPU load parameters are higher for the voxel grid filter compared to when no filter is used. The most representative parameter during the measurement period, the CPU average usage, differs significantly for smaller input point clouds sizes. When the input point cloud size increases, the CPU average usage difference does decrease to approximate 3%.

In general terms, the voxel grid filter is more depending on the size of the input point cloud, in contrast to when no filter is used. That indicates that executing the filter many instances per seconds require more CPU usage than filtering a larger number of points. In addition, temporary peaks of maximal CPU usage does not occur for larger number of points, as can be seen in Figure 5.8. Hence, the voxel grid filter does not require more CPU usage when handling larger point clouds, otherwise would the graph probably appeared differently.

5.3.2 Data refresh rate

The results of the data refresh rate evaluation are shown in Table 5.1. The data refresh rate was captured when utilizing the voxel grid filter and when no filter was used. If the value was fluctuating between two different values, the observed value was noted as ".5". The quantity "points per second" is calculated by multiplying the amount of points with the data refresh rate.

Table 5.1: The data refresh rate [Hz] is presented for when utilizing the voxel grid filter and when no filter was used. The quantity points per second is calculated by multiplying the amount of points with then data refresh rate.

Points	Data refresh rate [Hz] Voxel grid filter	Data refresh rate [Hz] No filter	Points per second Voxel grid filter	Points per second No filter
5,280	37.5	38	198,000	200,640
10,560	19	19	200,640	200,640
15,840	12.5	12.5	198,000	198,000
21,120	9.5	9.5	200,640	200,640
26,400	7.5	7.5	198,000	198,000
31,680	6.5	6.5	205,920	205,920
36,960	5.5	5.5	203,280	203,280
42,240	4.5	5	190,080	211,200
47,520	4	4	190,080	190,080
52,800	4	3.5	211,200	184,800
58,080	3.5	3.5	203,280	203,280
63,360	3	3	190,080	190,080

It is evident from Table 5.1 that the data refresh rate is almost identical when using a voxel grid filter compared to using no filter. Thus, the voxel grid filter does not create a delay when handling and filter point cloud data. The calculation of point per second is between 190.1k to 211.2k for all input point cloud sizes and for the both cases. It correlates strongly with the stated value in the data sheet for the Livox Mid-360, which is 200,000 points per second. Consequently, all points generated by the Mid-360 LiDAR can be processed in the application, either with or without a filter.

5.4 Challenges

Throughout the project, a number of various challenges were faced and needed to be managed. One of the greatest challenges was to collect LiDAR data and visualize it in real time graphically. At first, it was difficult to share the collected data between different sections and functions of the application. In addition, there was no way to ensure that every data point got processed, in other words, that no data loss occurred. The solution to this was to implement a semaphore, which enabled control access for synchronization of the shared resource buffer with LiDAR data. The semaphore is described in detail in Section 4.5.2.

Another challenge was the understanding of the Livox SDK2 and how to utilize the Livox SDK2 with the included API. The API for the Mid-360 brings various functions to control the unit and collect data in different data formats. It required a significant amount of time to get familiar with the software. The software connection between the back-end written in C++ and the front-end (QML) was an additional obstacle to overcome. There was an issue that the collected data in the back-end was not updated and displayed. The solution to this issue was to implement the `updateTimer` in QML and set the `updateGeometry()` function in the C++ back-end to a `Q_INVOKABLE`. Every 25 ms the `Timer` is triggered and the `Q_INVOKABLE` enabled the invocation of `updateGeometry()` from the QML front-end. This was a crucial communication link, in order to quickly display real time data graphically on the application.

6 Conclusion and further work

6.1 Conclusion

The outcome choice of the literature study of available 3D LiDAR sensors on the market was the Livox Mid-360. The sensor was selected due to the great compatibility possibilities with CrossControl's display computers, considering its interface and provided Livox SDK2 (including an API). Moreover, the Mid-360 performs extremely well in FOV and number of data points per second.

The investigation of a software framework for handling sensor data ended up being the Point Cloud Library (PCL), where a voxel grid filter was utilized. The LiDAR application was developed with the Livox SDK2 combined with a C++ back-end, in order to visualize data using QML as the GUI design tool. Real time LiDAR sensor data was graphically visualized on the display computer CCpilot X900.

The comparison between the voxel grid filter utilization and without using a filter has showed that the filter potentially has visual advantages. The voxel grid filter keeps the structure and geometry of the input point cloud, while utilizing less points. On the other hand, the average processor load was greater for the filter compared to when no filter was used. Whether a filter was used or not, all points generated by the Livox Mid-360 could be processed and visualized by the developed application without any latency.

6.2 Further work

Throughout the project, a number of improvement opportunities and recommendations for further work were observed. Firstly, the LiDAR sensor market is growing rapidly and the technique keeps improving, apart from declining prices. The number of choices for low-budget LiDAR sensors will increase and appealing characteristics will evolves further. Hence, exploring more of the market and investigating further into new upcoming LiDAR sensors will always be of great interest.

Regarding the developed application, further improvements and features can be made. An improvement is to add a button on the GUI where the user can switch between two colorization of the points. To accomplish that, further reading about the interaction between C++ back-end and the QML code is needed. Furthermore, another improvement is to add a color scale on GUI and make the set colors more distinguishable from each other. The idea with color scale is to represent a distance value connected to the metric system in meters. Thus, the user would get further information about the displayed point cloud.

Lastly, the PCL provides multiple features regarding point cloud handling. There are other filters supplied by the PCL, such as, "Extracting indices" filter which can remove points in a certain direction. In addition, the PCL has also implemented a way of representing a point cloud graphically, called "PCL Visualisation". A library, with the name "VTK", was required in order to work properly. Unfortunately, the VTK library was not supported in the current version of the operating system running on the CCpilot X900. However, if it would be able to install the library on the display computer in the future, PCL Visualisation is a appealing and powerful software alternative.

References

[1] N. Watanabe, and H. Ryugen, “Cheaper lidar sensors brighten the future of autonomous cars”, Nikkei Asia, May 2023. [Online] Available: https://asia.nikkei.com/Business/Automobiles/Cheaper-lidar-sensors-brighten-the-future-of-autonomous-cars [Accessed May 3, 2023]

[2] F. Rosique, P. J. Navarro, C.Fernández, and A. Padilla, “A Systematic Review of Perception System and Simulators for Autonomous Vehicles Research”, Polytechnic University of Cartagena, February 2019. [Online] Available: https://www.mdpi.com/1424-8220/19/3/648 [Accessed Feb. 2, 2023].

[3] National Aeronautics and Space Administration (NASA), “The Apollo 15 Lunar Laser Ranging RetroReflector”, April 2010. [Online] Available: https://www.livoxtech.com/mid-360/downloads [Accessed Apr. 4, 2023].

[4] M. L. Stitch, E. J.Woodburry, and J. H. Morse, “Optical ranging system uses laser transmitter”, April 1961. Electronics. 34: pp. 51–53.

[5] D. Khizbullin, “Autonomous Driving Lidar Perception Stack with PCL: An Algorithmic Implementation”, February 2022. [Online] https://hackernoon.com/autonomous-driving-lidar-perception-stack-with-pcl-an-algorithmic-implementation [Accessed May 4, 2023].

[6] S. Islam, S. Tanvir, R. Habib, T. Tashrif, A. Ahmed, T. Ferdous, R. Arefin, and S. Alam, “Autonomous Driving Vehicle System Using LiDAR Sensor”, ResearchGate, February 2022. [Online] https://www.researchgate.net/publication/358910737_Autonomous_Driving_Vehicle_System_Using_LiDAR_Sensor [Accessed Apr. 5, 2023].

[7] Cyberphysics UK, “EMSpectrumcolor”. [Online] https://www.cyberphysics.co.uk/topics/radioact/Radio/EMSpectrumcolor.jpg [Accessed May 3, 2023]

[8] A. Rodnitzky, “Sensors 201: Scanning and Solid State LiDAR”, Tangram vision, July 2021. [Online] https://www.tangramvision.com/blog/sensors-201-scanning-and-solid-state-lidar [Accessed Feb. 9, 2023].

[9] Paul, “LiDAR and ToF Cameras – Technologies explained”, ToF-Insights, November 2018. [Online] https://tof-insights.com/time-of-flight-lidar-and-scanners-technologies-explained/ [Accessed Jun. 5, 2023]

[10] Kudan, “3D lidar SLAM: The Basics”, Kudan, June 2022. [Online] https://www.kudan.io/blog/3d-lidar-slam-the-basics/ [Accessed Feb. 2, 2023].

[11] Kudan, “How to Select the Best 3D Lidar for SLAM”, Kudan, September 2022. [Online] https://www.kudan.io/blog/3d-lidar-slam-the-basics/ [Accessed Feb. 3, 2023].

[12] Livoxtech, “Livox Mid-360 User Manual”, January 2023. [Online] Available: https://www.livoxtech.com/mid-360/downloads [Accessed Feb. 10, 2023].

[13] Shenzhen Leishen Intelligent System Co., “LS Long Range LiDAR Scanner Products”, LSLIDAR. [Online] https://www.leishenlidar.com/wp-content/uploads/2022/03/LS30MVA.pdf [Accessed May 24, 2023].

[14] Y. Wu, “Radiometry, BRDF and Photometric Stereo”, Northwestern University Evanston. [Online] http://users.eecs.northwestern.edu/~yingwu/teaching/EECS432/Notes/lighting.pdf [Accessed Apr. 5, 2023].

[15] 3M, “What is retroreflectivity & why is it important?”, 3M. [Online] https://www.3m.com/3M/en_US/road-safety-us/resources/road-transportation-safety-center-blog/full-story/~/road-signs-retroreflectivity/?storyid=328c8880-941b-4adc-a9f9-46a1cd79e637[Accessed Apr. 5, 2023].

[16] 3M, “How 3M Scotchlite Reflective Material Works”, 3M. [Online] https://www.3mindia.in/3M/en_IN/scotchlite-reflective-material-in/industries-active-lifestyle/active-lifestyle/how-retroreflection-works/ [Accessed Apr. 5, 2023].

[17] R. B. Rusu, and S. Cousins, “3D is here: Point Cloud Library (PCL)”, IEEE International Conference on Robotics and Automation (ICRA), May 2011. [Online] https://pointclouds.org/assets/pdf/pcl_icra2011.pdf [Accessed Mar. 8, 2023].

[18] J. Lambers, “Three-Dimensional Coordinate Systems” ,University of Southern Missisipi, Sept 2009. [Online] https://www.math.usm.edu/lambers/mat169/fall09/lecture17.pdf [Accessed May 3, 2023].

[19] SkillsYouNeed, “An Introduction to Cartesian Coordinate Systems”, skillsyouneed.com. [Online] https://www.skillsyouneed.com/num/cartesian-coordinates.html [Accessed May 3, 2023].

[20] A. Hacinecipoglu, E. I. Konukseven, and A. B. Koku, “Pose Invariant People Detection in Point Clouds for Mobile Robots”, Middle East Technical University, Ankara, May 2020. [Online] https://www.researchgate.net/publication/346331400_Pose_Invariant_People_Detection_in_Point_Clouds_for_Mobile_Robots [Accessed May 5, 2023].

[21] Chinese Software Developer Network, “PCL uses VoxelGrid for point cloud sampling”, December 2020. [Online] https://blog.csdn.net/wdf666520/article/details/111873685 [Accessed May 4, 2023].

[22] L. Chen, S. Huang, and B. Hunag, “Precise 6DOF Localization of Robot End Effectors Using 3D Vision and Registration without Referencing Targets”, Vision Sensors - Recent Advances, October 2022. [Online] https://www.researchgate.net/publication/364615519_Precise_6DOF_Localization_of_Robot_End_Effectors_Using_3D_Vision_and_Registration_without_Referencing_Targets [Accessed May 5, 2023].

[23] Microsoft Learn, “Color Structure”, Microsoft, November 2011. [Online] https://learn.microsoft.com/en-us/previous-versions/windows/silverlight/dotnet-windows-silverlight/ms653055(v=vs.95)?redirectedfrom=MSDN [Accessed May 5, 2023].

[24] G. Pipis, “Iterate Over Image Pixels”, predictivehacks.com, August 2019. [Online] https://predictivehacks.com/iterate-over-image-pixels/ [Accessed May 5, 2023].

[25] Livoxtech, “Livox Mid Series User Manual v1.2”, May 2020. [Online] Available: https://www.livoxtech.com/mid-40-and-mid-100/downloads [Accessed Feb. 10, 2023].

[26] Velarray M1600, “Velarray M1600 PRECISE NEAR-FIELD VISION FOR AUTONOMY”, Velodyne Lidar. [Online] https://www.gpsolution.com/uploads/files/2021/1124/48kOVJmPqTiBmRCxTI0qO0hBpoMiQ9XLKfOKOLWC.pdf [Accessed Feb. 9, 2023].

[27] CrossControl, “CCpilot X900”, crosscontrol.com. [Online] https://crosscontrol.com/displays-computers/ccpilot-x900/ [Accessed May 4, 2023].

[28] Orcale, “Oracle VM VirtualBox Overview”, oracle.com, June 2021. [Online] https://www.oracle.com/assets/oracle-vm-virtualbox-overview-2981353.pdf [Accessed May 4, 2023].

[29] Qt Creator, “Qt Creator Manual”, Qt Documentation. [Online] https://doc.qt.io/qtcreator/ [Accessed Apr. 3, 2023].

[30] Livox Wiki, “Livox LiDAR Communication Protocol–Mid360”, Livox English documentation for Mid-360, Mars 2023. [Online] https://livox-wiki-en.readthedocs.io/en/latest/tutorials/new_product/mid360/livox_eth_protocol_mid360.html#point-cloud-imu-data-protocol [Accessed Mar. 16, 2023]

[31] Livox SDK2 Github, “Livox SDK2”, github.com. [Online] https://github.com/Livox-SDK/Livox-SDK2 [Accessed Mar. 16, 2023].

[32] Point Cloud Library tutorial of a VoxelGrid filter, “Downsampling a PointCloud using a VoxelGrid filter”, PCL Documentation. [Online] https://pcl.readthedocs.io/projects/tutorials/en/latest/voxel_grid.html#voxelgrid [Accessed Feb. 15, 2023].

Appendix

Code for first collection of LiDAR data (livox_lidar_quick_start.cpp)

```
#include "livox_lidar_def.h"
#include "livox_lidar_api.h"

#ifdef WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <thread>
#include <chrono>
#include <iostream>

#include <fstream>
#include <string>
#include <vector>


bool WritePointsToFile(std::string file_name, std::string x_point, std::string y_point, std::string z_point);

bool WritePointsToFile(std::string file_name, std::string x_point, std::string y_point, std::string z_point) {
  std::ofstream file;
  file.open(file_name, std::ios_base::app);
  file << x_point << "," << y_point << "," << z_point << std::endl;
  file.close();
  return true;
}

void PointCloudCallback(uint32_t handle, const uint8_t dev_type,
LivoxLidarEthernetPacket* data, void* client_data) {
  if (data == nullptr) {
    return;
  }

  std::string x_point;
  std::string y_point;
  std::string z_point;

  if (data->data_type == kLivoxLidarCartesianCoordinateHighData) {
    LivoxLidarCartesianHighRawPoint *p_point_data = (LivoxLidarCartesianHighRawPoint *)data->data;
    //std::cout << (LivoxLidarCartesianHighRawPoint *)data->data << std::endl;
    for (uint32_t i = 0; i < data->dot_num; i++) {
      x_point = std::to_string(p_point_data[i].x);
      y_point = std::to_string(p_point_data[i].y);
      z_point = std::to_string(p_point_data[i].z);

      std::cout << "reflectivity: " << (int)(p_point_data[i].reflectivity) << std::endl;
      std::cout << "tag:␣" << (int)(p_point_data[i].tag) << std::endl;
      bool writetofile = WritePointsToFile("test.csv", x_point, y_point, z_point);
    }
  }
  else if (data->data_type == kLivoxLidarCartesianCoordinateLowData) {
    LivoxLidarCartesianLowRawPoint *p_point_data = (LivoxLidarCartesianLowRawPoint *)data->data;
  } else if (data->data_type == kLivoxLidarSphericalCoordinateData) {
    LivoxLidarSpherPoint* p_point_data = (LivoxLidarSpherPoint *)data->data;
  }
}

int main(int argc, const char *argv[]) {
  if (argc != 2) {
    printf("Params␣Invalid,␣must␣input␣config␣path.\n");
    return -1;
  }
  const std::string path = argv[1];

  if (!LivoxLidarSdkInit(path.c_str())) {
    printf("Livox␣Init␣Failed\n");
    LivoxLidarSdkUninit();
    return -1;
  }
  WritePointsToFile("test.csv", "x", "y", "z");

  SetLivoxLidarPointCloudCallBack(PointCloudCallback, nullptr);

#ifdef WIN32
  Sleep(3000);
#else
  sleep(300);
#endif
  LivoxLidarSdkUninit();
  printf("Livox␣Quick␣Start␣Demo␣End!\n");
  return 0;
}
```

Code for voxel grid filter test (voxel_grid.cpp)

```cpp
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>

int main ()
{
   pcl::PCLPointCloud2::Ptr cloud (new pcl::PCLPointCloud2 ());
   pcl::PCLPointCloud2::Ptr cloud_filtered (new pcl::PCLPointCloud2 ());
   // Fill in the cloud data
   pcl::PCDReader reader;
   // Replace the path below with the path where you saved your file
   reader.read ("test_pcd_3.pcd", *cloud);

   std::cerr << "PointCloud before filtering: " << cloud->width * cloud->height
      << " data points (" << pcl::getFieldsList (*cloud) << ")." << std::endl;

   // Create the filtering object
   pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
   sor.setInputCloud (cloud);
   sor.setLeafSize (1f, 1f, 1f); //set the leaf size
   sor.filter (*cloud_filtered); //output from the filter in *cloud_filter

   std::cerr << "PointCloud after filtering: " << cloud_filtered->width * cloud_filtered->height
      << " data points (" << pcl::getFieldsList (*cloud_filtered) << ")." << std::endl;

   pcl::PCDWriter writer;
   writer.write ("test_pcd_3_downsampled_0.125.pcd", *cloud_filtered,
        Eigen::Vector4f::Zero (), Eigen::Quaternionf::Identity (), false);

   return (0);
}
```

Code for the back-end of the application

quickPointCloud.pro

```
QT += quick quick3d core
CONFIG += c++11 qmltypes debug
LIBS += -L/usr/local/lib/ -l:liblivox_lidar_sdk_static.a

QML_IMPORT_NAME = pointcloud
QML_IMPORT_MAJOR_VERSION = 1

INCLUDEPATH += /usr/include/pcl-1.10
INCLUDEPATH += /usr/include/eigen3
INCLUDEPATH += /usr/local/include/
LIBS += -lpcl_common
LIBS += -lpcl_filters

SOURCES += \
        main.cpp \
        mycustompointcloud.cpp \
        semaphore.cpp

RESOURCES += qml.qrc

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

HEADERS += \
    mycustompointcloud.h \
    semaphore.h
```

mycustompointcloud.h

```
//Here are all libirays for the class mycustompointcloud
#ifndef MYCUSTOMPOINTCLOUD_H
#define MYCUSTOMPOINTCLOUD_H

#include <QtQuick3D/QQuick3DGeometry>
#include <QVector3D>

#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>

#include "livox_lidar_def.h"
#include "livox_lidar_api.h"

#include "semaphore.h"
```

```
#include <array>
#include <chrono>
#include <thread>

#include <QDebug>

//Only for the test data:
#include <QRandomGenerator>
#define COORD_MIN -3000
#define COORD_MAX 3000

//Input point cloud size:
#define BUF_SIZE 1056*20

//LiDAR data structure:
typedef struct {
    int32_t x;
    int32_t y;
    int32_t z;
    uint8_t ref;
    uint8_t tag;
    uint16_t pad;
} lidar_data;

//Define the class:
class MyCustomPointCloud : public QQuick3DGeometry
{
    //Enables the coupling between C++ and QML:
    Q_OBJECT
    QML_ELEMENT

public:
    MyCustomPointCloud();
    //Make a invokable function which can be called from QML
    Q_INVOKABLE void updateGeometry();

private:
    //Methods used for lidar sensor:
    void init_lidar();
    static void cb(uint32_t handle, const uint8_t dev_type, LivoxLidarEthernetPacket* data, void* client_data);

    //Methods used for testing:
    static void test_cb();
    void test_init();
    void test_geometry();

    //The configuration file for the Livox Mid-360:
    const char* config_path = "../lidar_config.json";
};


#endif // MYCUSTOMPOINTCLOUD_H
```

mycustompointcloud.cpp

```
#include "mycustompointcloud.h"

//***Define LiDAR Mode***
#define LIDAR_DIST
//#define LIDAR_REF
//#define TEST_DATA

/*
Global LiDAR data buffer shared between consumer and producer
Consumer: updateGeometry
Producer: livox callback function handling sensor data
*/
std::array<lidar_data, BUF_SIZE> lidar_buffer = {};

/*
Semaphore as synchronization primitive for sharing the LiDAR data buffer
sem_cons: semaphore released by updateGeometry when graphics data has been rendered
sem_prod: semaphore released by lidar data callback new data is available for rendering
*/
Semaphore sem_cons, sem_prod;

MyCustomPointCloud::MyCustomPointCloud()
{
#ifdef TEST_DATA
    test_init();
#else
    init_lidar();
#endif
    qDebug() << "Init LiDAR done ";
    sem_cons.release();
}

void MyCustomPointCloud::init_lidar()
{
    if (!LivoxLidarSdkInit(config_path))
    {
        qDebug() << "Livox init failed ";
        LivoxLidarSdkUninit();
    }
    qDebug() << "Livox init success ";

    typedef void(* cbfuncptr)(uint32_t handle, const uint8_t dev_type, LivoxLidarEthernetPacket* data, void*
      client_data);
    cbfuncptr lidar_cb = MyCustomPointCloud::cb;
    SetLivoxLidarPointCloudCallBack(lidar_cb, nullptr);

    qDebug() << "init_lidar: Livox LiDAR callback started";
    MyCustomPointCloud::test_geometry();
}

void MyCustomPointCloud::test_init(){
    qDebug() << "test_init: start";
    //Launch thread
    std::thread t1(MyCustomPointCloud::test_cb);
    t1.detach();
    MyCustomPointCloud::test_geometry();
    qDebug() << "test_init: complete";
}

void MyCustomPointCloud::test_cb(){

    //Collect test data
    while(1){
    static int data_point_ctr = 0;
    const int dot_num = 96;

    //Every run generate 96 new points with random data

    //Wait for consumer to get done with the data
    for (uint32_t i = 0; i < dot_num; i++) {
        if (data_point_ctr > BUF_SIZE - 1)
        {
            qDebug("DEBUG: shared buffer full, should not get here");
            return;
        }
        //Setting random coordinates in x, y and z
        lidar_buffer[data_point_ctr].x = rand()%(COORD_MAX-(COORD_MIN) + 1) + COORD_MIN;
        lidar_buffer[data_point_ctr].y = rand()%(COORD_MAX-(COORD_MIN) + 1) + COORD_MIN;
        lidar_buffer[data_point_ctr].z = rand()%(COORD_MAX-(COORD_MIN) + 1) + COORD_MIN;

        //Setting ref and tag to 0
        lidar_buffer[data_point_ctr].ref =  0;
        lidar_buffer[data_point_ctr].tag =  0;
        ++data_point_ctr;
    }

    if(data_point_ctr == BUF_SIZE)
    {
        //Production done and the production semaphore is released:
        sem_prod.release();

        //Waiting for consumer semaphore
        sem_cons.acquire();

        //Setting data_point_ctr to 0
        data_point_ctr = 0;
    }
    }
}

void MyCustomPointCloud::cb(uint32_t handle, const uint8_t dev_type, LivoxLidarEthernetPacket* data, void*
      client_data)
{
    //Check if data is collected
    if (data == nullptr)
    {
        return;
    }
    //Setting data_point_ctr to 0
    static int data_point_ctr = 0;

    if (data->data_type == kLivoxLidarCartesianCoordinateHighData)
    {
        LivoxLidarCartesianHighRawPoint *p_point_data = (LivoxLidarCartesianHighRawPoint *)data->data;

        //Wait for consumer to get done with the data
        for (uint32_t i = 0; i < data->dot_num; i++) {
            if (data_point_ctr > BUF_SIZE - 1)
            {
                qDebug("DEBUG: shared buffer full, should not get here");
                return;
            }

            //Store x, y, z, reflectivity and tag in the buffer
            lidar_buffer[data_point_ctr].x =  p_point_data[i].x;
            lidar_buffer[data_point_ctr].y =  p_point_data[i].y;
            lidar_buffer[data_point_ctr].z =  p_point_data[i].z;
            lidar_buffer[data_point_ctr].ref =  p_point_data[i].reflectivity;
            lidar_buffer[data_point_ctr].tag =  p_point_data[i].tag;
            ++data_point_ctr;
        }
    }


    if(data_point_ctr == BUF_SIZE)
    {
        //Production done and the production semaphore is released:
        sem_prod.release();

        //Waiting for consumer semaphore
        sem_cons.acquire();

        //Setting data_point_ctr to 0 again
        data_point_ctr = 0;
    }
}

void MyCustomPointCloud::test_geometry(){

    qDebug() << "test_geometry start!";
    //Built-in function that signals that the graphics will update in QML:
    update();

    //Setting the vertexData and resizeing to the size of the according to the LiDAR data:
    QByteArray vertexData;
    vertexData.resize(sizeof(float) * 7 * lidar_buffer.size());
    float *p = reinterpret_cast<float *>(vertexData.data());

    //Iterating to set all test points and setting color
    for (int var = 0; var < lidar_buffer.size(); ++var)
    {
        //Add the points to the pointer
        *p++ = (float)lidar_buffer[var].x;
        *p++ = (float)lidar_buffer[var].y;
        *p++ = (float)lidar_buffer[var].z;

        //Set the color of the points to green
        float r=0, g=1, b=0;

        //Add the color to the pointer
        *p++ = r;
        *p++ = g;
        *p++ = b;
        *p++ = 0.9f;
    }
    //Set the data of the QQuick3DGeometry object:
    setVertexData(vertexData);
    //Set the point type to "points"
    setPrimitiveType(QQuick3DGeometry::PrimitiveType::Points);
    //Set the size of each data point, 3 for coordinates and 4 for color
    setStride(7 * sizeof(float));
    //Add an attribute for the position (x,y,z), no offset, type float:
    addAttribute(QQuick3DGeometry::Attribute::PositionSemantic,
    0,
    QQuick3DGeometry::Attribute::F32Type);
    //Add an attribute for the color (r,g,b,a), 3 offset, type float:
    addAttribute(QQuick3DGeometry::Attribute::ColorSemantic,
    3 * sizeof(float),
    QQuick3DGeometry::Attribute::F32Type);

}

void MyCustomPointCloud::updateGeometry()
{
    //Built-in function that signals that the graphics will update in QML:
    update();

    //Intilize filter pointers:
    pcl::PCLPointCloud2::Ptr cloud (new pcl::PCLPointCloud2 ());
    pcl::PCLPointCloud2::Ptr cloud_filtered (new pcl::PCLPointCloud2 ());
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_rgb(new pcl::PointCloud<pcl::PointXYZRGB>);
    pcl::PointXYZRGB point;

    //Waiting for production semaphore
    sem_prod.acquire();
    //Production semaphore acquired in updateGeometry

    //Prepare for filtering by storing data in a PCL PointXYZRGB
    for (int i = 0; i < lidar_buffer.size(); i++)  {
        point.x = (float)lidar_buffer[i].x;
        point.y = (float)lidar_buffer[i].y;
        point.z = (float)lidar_buffer[i].z;
        point.r = (float)lidar_buffer[i].ref;
        cloud_rgb->push_back(point);
    }

    // Convert to PCLPointCloud2
```

```
pcl::toPCLPointCloud2(*cloud_rgb, *cloud);

// Create the filtering object & inserting the input point cloud
pcl::VoxelGrid<pcl::PCLPointCloud2> voxel;
voxel.setInputCloud (cloud);
voxel.setLeafSize (75.0f, 75.0f, 75.0f); //box of 75 mm
voxel.filter (*cloud_filtered); //output in *cloud_filtered

// Convert back to pcl::PointCloud<pcl::PointXYZRGB>
pcl::fromPCLPointCloud2(*cloud_filtered, *cloud_rgb);

/*
***If no filter used***
sem_prod.acquire();
QByteArray vertexData;
vertexData.resize(sizeof(float) * 7 * lidar_buffer.size());
float *p = reinterpret_cast<float *>(vertexData.data());
*/

QByteArray vertexData;
//Resizeing the QByteArray according to size of LiDAR data, 3 for coordinates and 4 for color
vertexData.resize(sizeof(float) * 7 * cloud_rgb->width);
//Create a pointer to access the data
float *p = reinterpret_cast<float *>(vertexData.data());

//Iterating for storing points and color in the pointer p:
for (int var = 0; var < cloud_rgb->width; ++var) //change cloud_rgb->width to lidar_buffer.size() if no
  filter
{
    //Access the point cloud data:
    pcl::PointXYZRGB point = cloud_rgb->at(var);
    //Add the points to the pointer:
    *p++ = point.x;
    *p++ = point.z; //"y" is set to z for the axis transformation between sensor and QML
    *p++ = -point.y; //"z" is set to -y for the axis transformation between sensor and QML

    //Add the points to the pointer (if no filter is used):
    //*p++ = lidar_buffer[var].x;
    //*p++ = lidar_buffer[var].z;
    //*p++ = -lidar_buffer[var].y;

    float r, g, b, distance, lidar_ref;

    //Calculate the distance from origo:
    distance = std::sqrt(point.x * point.x + point.y * point.y + point.z * point.z);
    //Calculate the distance from origo (no filter):
    //distance = std::sqrt(lidar_buffer[var].x * lidar_buffer[var].x + lidar_buffer[var].y * lidar_buffer[var].y +
  lidar_buffer[var].z * lidar_buffer[var].z);

    //Check if colorization by distance or reflection (from the LiDAR mode):
    #ifdef LIDAR_DIST
        //Set color to red when short distance and gradually gets more green further away from origo:
        r = 1-distance/3000+0.2;
        g = distance/3000-0.2;
        b = 0.0;
    #endif

    #ifdef LIDAR_REF
        lidar_ref = point.r; //set to "lidar_buffer[var].ref" when no filter is used
        if (lidar_ref < 64) {
            //Gradually from blue to green
            r = 0;
            g = lidar_ref / 64.0f;
            b = 1 − lidar_ref / 64.0f;
            }
        else if (lidar_ref < 128) {
            //Gradually from green to yellow
            r = (lidar_ref − 64) / 64.0f;
            g = 1;
            b = 0;
            }
            else if (lidar_ref < 192) {
            //Gradually from yellow to orange
            r = 1;
            g = (255 − lidar_ref) / 64.0f;
            b = 0;
            }
        else {
            //Gradually from orange to red
            r = 1;
            g = 0;
            b = (lidar_ref − 192) / 63.0f;
            }
    #endif

    //Add the color to the pointer
    ∗p++ = r;
    ∗p++ = g;
    ∗p++ = b;
    ∗p++ = 0.9f; //alpha value

}

//Set the data of the QQuick3DGeometry object:
setVertexData(vertexData);
//Set the point type to "points"
setPrimitiveType(QQuick3DGeometry::PrimitiveType::Points);
//Set the size of each data point, 3 for coordinates and 4 for color
setStride(7 ∗ sizeof(float));
//Add an attribute for the position (x,y,z), no offset, type float:
addAttribute(QQuick3DGeometry::Attribute::PositionSemantic,
0,
QQuick3DGeometry::Attribute::F32Type);
//Add an attribute for the color (r,g,b,a), 3 offset, type float:
addAttribute(QQuick3DGeometry::Attribute::ColorSemantic,
3 ∗ sizeof(float),
QQuick3DGeometry::Attribute::F32Type);

//Releasing consumer semaphore
sem_cons.release();
```

```
}
```

semaphore.h

```
#include <mutex>
#include <condition_variable>

class Semaphore
{
    std::mutex mutex_;
    std::condition_variable condition_;
    unsigned long count_ = 0;

    public:
        void release();
        void acquire();
};
```

semaphore.cpp

```
#include <semaphore.h>

/*
 * count_ = 0 means locked
 * count_ = 1 means unlocked
 */
void Semaphore::release()
{
        std::lock_guard<decltype(mutex_)> lock(mutex_);
        ++count_;
        condition_.notify_one();
}

void Semaphore::acquire()
{
        std::unique_lock<decltype(mutex_)> lock(mutex_);
        while(!count_)
            condition_.wait(lock);
        --count_;
}
```

lidar_config.json

```
{
  "MID360": {
    "lidar_net_info" : {
      "cmd_data_port": 56100,
      "push_msg_port": 56200,
      "point_data_port": 56300,
      "imu_data_port": 56400,
      "log_data_port": 56500
    },
    "host_net_info" : {
      "cmd_data_ip" : "192.168.1.50",
      "cmd_data_port": 56101,
      "push_msg_ip": "192.168.1.50",
      "push_msg_port": 56201,
      "point_data_ip": "192.168.1.50",
      "point_data_port": 56301,
      "imu_data_ip" : "192.168.1.50",
      "imu_data_port": 56401,
      "log_data_ip" : "192.168.1.50",
      "log_data_port": 56501
    }
  }
}
```

main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <mycustompointcloud.h>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;

    //Launching application
    const QUrl url(QStringLiteral("qrc:/main.qml"));
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                     &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

Code for the front-end of the application (main.qml)

```
import QtQuick
import QtQuick.Controls
import QtQuick3D
import QtQuick3D.Helpers
import pointcloud 1

//Set up for application window
ApplicationWindow {
    id: window
    width: 1280 //the number of pixels in width
    height: 768 //the number of pixels in height
    visible: true //Application is visiable on the screen
    x: Screen.width / 2 − width / 2 //launching in middle of the screen
    y: Screen.height / 2 − height / 2 //launching in middle of the screen
    title: qsTr("Visualization␣of␣3D␣LiDAR␣data")

    //Setting up the 3D view
    View3D {
        id: view3D_lidar
        anchors.fill: parent //Anchored to the entire application window
        environment: SceneEnvironment {
            id:sceEnv
            backgroundMode: SceneEnvironment.Color //The backgroud mode of the 3D view
            clearColor: "black" //Backgroud is set to black
        }
        //Setting the intial postion and rotation of the camera
        PerspectiveCamera {
            id: camera
            x: 1500
            y: 1500
            z: 1500
            eulerRotation.x: −20
            eulerRotation.y: 40
        }
        //Importing the model from the Class MyCustomPointCloud
        Model {
            geometry: MyCustomPointCloud {
                id: mypointcloud
                property int updateCount: 0 //Keep track of the number of times updateGeometry() is called
                property int prevUpdateCount: 0 //Previously number of updateGeometry() was called
            }
            //Pre−setting all point colors to white with no lightning
            materials: DefaultMaterial {
                lighting: DefaultMaterial.NoLighting
                diffuseColor: "white"
                pointSize: 4 //Size of the points
            }
        }
        //Enable the axis
        AxisHelper{
        }

    }
```

```
//Enable the control of the camera with WASD
WasdController{
    controlledObject: camera
    speed: 1
    shiftSpeed : 10 //Faster when holding shift
}

//Add a button for closing the appication located in the upper right corner
Rectangle {
    width: 40
    height: 40
    color: "transparent"
    radius: width*0.5
    anchors.top: parent.top
    anchors.right: parent.right

    //Close button is represented with an "X"
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        font.pointSize: 18
        color: "white"
        text: qsTr("X")
    }
    //Printing in the termal that the application was closed with button
    MouseArea {
        anchors.fill: parent
        onPressed: console.log("Close button was pressed")
        onClicked: window.close()
    }
}
//The timer that signals updateGeometry() to update the point cloud
Timer {
  id: updateTimer
    running: true
    interval: 25 //How often the timer is updated, every 25 ms
    repeat: true //Repeatedly running

    onTriggered: {
        mypointcloud.updateGeometry() //Sends to updateGeometry to update the points
        //mypointcloud.test_geometry() (if test_geometry is used)
        mypointcloud.updateCount = mypointcloud.updateCount + 1 // increment the update count every time
  updateGeometry() is called
        }
}
//The Data Refresh Rate is determined here:
Text {
        id: dataRefreshRateText
        text: "Data refresh rate: "
        font.pointSize: 14
        color: "white"
        //Located in upper left corner:
        x: 15
        y: 15
```

```
            Timer {
                id: dataRefreshRateTimer
                interval: 1000 //Calculateds every second
                running: true
                repeat: true //Repeatedly running
                onTriggered: {
                    //Removes the previously update count from the update count:
                    var updateDiff = mypointcloud.updateCount - mypointcloud.prevUpdateCount
                    mypointcloud.prevUpdateCount = mypointcloud.updateCount //Sets the previously update count
      to the current update count
                    dataRefreshRateText.text = "Data refresh rate: " + updateDiff + " Hz" //The displayed text for
       the application

                }
            }
        }

}
```