Abstract

This group project is about researching methods of simultaneous localization and mapping for mobile robot. The goal of each team member was to implement a method that would allow to determine, partially or fully, the placement of a robot in space and construct a map of its surroundings. This report is about the implementation of one of the related algorithms, connected to the depth recognition system. Besides the description of the algorithm itself, the relevant theory was presented, along with a review of all instruments (applications and items) used. The outcome of the project is a code, which represents the required algorithm and can be launched such that it is possible to observe the depth image from the camera. This result was successfully obtained, with several further improvements.

The following is a list of keywords that are frequently used in the project. For simplicity and to avoid repetition, their definitions will be presented here once.

- **SLAM** – acronym for **Simultaneous Localization and Mapping**; a problem of creating a map of an unknown environment and keeping track of a target object's location simultaneously within this environment using external sensors.
- **IMU** – acronym for **Inertial Measurement Unit**; a device used to measure the acceleration and rotation (orientation in space) of the object it is installed in. This is possible due to the built-in accelerometer and gyroscope.

Table of Contents

1. Introduction

The world around us is changing every day, and the things that were considered as luxuries some decades ago, for instance, mobile phones, are now perceived as necessities, without which it is difficult to live a normal life. And, while this idea is applicable to a great variety of technologies present nowadays, the main subject area discussed in this project is robotics.

Starting in the 1980s, the commercial distribution of robots covered the whole world. As years passed, nearly every business in developed countries decided to exchange human workers for machines on routine jobs, as this way the work was done faster, with greater efficiency, and at a lower cost. However, the most interesting examples of robots began to appear starting in the 2000s. Not only were they able to do something on their own, but they also imitated humans' actions and feelings. The multiplicity of algorithms connecting the perception of the outer world and machine logic for decision-making about actions allowed to start integrating robots directly into the lives of humans and assigning them much more complicated tasks than before. One of the algorithms, known as SLAM, is what this group project is about.

1.1 Purpose and Relevance

The general purpose of the project was to develop and implement several algorithms that would allow a mobile robot to localize in an unknown environment and construct a map of its surroundings simultaneously. The methods like this are known as SLAMs and are widely used nowadays in autonomous robots and self-driving cars. By building a self-acting robot, which can make decisions without the direct mentorship of a human, we open the doors to a multitude of research and application opportunities. As a simple example, the planet exploration rover can benefit astronomers, and the fact that it is autonomous allows people not to be constantly occupied by its guidance. This way, the scientists' time is saved greatly, while the results can be more precise due to the elimination of the human factor. Such robots can also be used for human assistance on a daily basis, like the already introduced self-driving taxis or delivery rovers. There is a plethora of different applications available for autonomous robots, and our team decided to do deeper research into SLAM algorithms and try to implement some of them for further use in the robot car prototype. Each member of the team was assigned a separate task dedicated to solving a problem of SLAM. Personally, my main task was to perform the integration of a depth camera into the operating system used within the robot. It is a necessary step since depth (in more common words, distance from the object) is a main indicator of the location of a robot. As

a result, I should have received an image from the camera transmitted from one part of the system to the other, imitating the work of it on the real robot.

2. Review of Chosen Instruments

To start with, it is important to mention that this project was done within the self-driving robot car project, which is currently being developed in the robotics laboratory. Due to specific requirements and constraints for its best performance, I did not have a chance to search for analogues in terms of the instruments used. These were chosen beforehand and given to me for the work itself. However, this does not mean that I would not inspect the positive and negative sides and reasons for the choice of the alternatives.

The main part of my work was related to ROS. It is an acronym for Robotics Operating System, which is a set of open-source libraries and tools that help to build robot applications. It provides developers with a universal architecture that combines a variety of ready-to-use algorithms applicable for a multitude of purposes, from teaching and use in student projects to utilization in the industrial sector and even for space research[1]. In addition, it allows one to abstract from setting up hardware and controllers, since this is done automatically by drivers, and to fully immerse the developer into programming of the robot's actions. Overall, this is a truly unique system, which currently has no equivalents in the world. Right now, ROS exists in 2 versions, 1st released in 2007, and 2nd working from 2017. In this project, ROS 2 is being used due to its better stability, real-time robot control ability, improved supported languages (Python 3.5 and C++ 17) and, finally, an ability to control the node status[2], all in comparison to ROS 1. Hence, all mentions of ROS will be related to ROS 2. The recent version brings greater reliability and minimal external dependencies, so that is why it was chosen to be the basis for the robot and this project consequently. Also, the most modern distribution of ROS 2 at the moment when the robot car project started (November 2021) was Galactic Geochelone, so it was taken for this project as well.

Next, consider the chosen depth camera, the Intel RealSense D435i. First of all, Intel RealSense Technology is a product range where every device consists of high-quality cameras, sensors, and graphical processor[3]. For developers, RealSense Technology also presented an open-source cross-platform SDK – a package with a collection of tools for managing devices' settings. The devices include the D400 family of depth cameras, which are of the main interest for this project

[1] Gerkey, Brian. "Ros Running on ISS." *Ros.org*, 1 Sept. 2014, https://www.ros.org/news/2014/09/ros-running-on-iss.html. [Last accessed 28/05/2022]
[2] Zoldaten. "Ros2 Vs ROS1.Ustanovka Ros2 Na Ubuntu 18.04." *Habr*, Habr, 14 Mar. 2020, https://habr.com/ru/post/492058/. [Last accessed 28/05/2022]
[3] Yaroslav. "Obzor Tekhnologii Intel® RealSense™ i Noutbuka s 3D Kameroj." *Intel RealSense*, 2014, https://special.habrahabr.ru/intel/realsense/review/. [Last accessed 28/05/2022]

due to their depth characteristics. The general algorithm they use for acquiring depth, called Depth from Stereo, is the most suitable for car robot due to its indoor/outdoor working abilities and good range, while also having the best cost-to-performance in comparison to cameras based on other methods of determining depth. The camera chosen, the D435i, is relatively cheap compared to similar devices of other brands[4]. It is also modifiable in terms of algorithms through the RealSense SDK and shows good performance for depth frames even in poorly lit rooms. Compared to other RealSense depth cameras, D415, D435 and D455, the first one is narrow in its field of view compared to D435i, though it is said to present higher accuracy due to higher density of pixels in an image. Considering that a robot (at least in the initial stages) does not require very high accuracy of depth image but needs a wider field of view for better navigation, D435i was taken. Comparing to D435, D435i is different because of an additional element – a built-in IMU, which is a must-have for a moving robot since it allows tracking data about pose and movement. Additionally, this unit is ROS-compatible, which is important for this specific project[5]. Looking at D455, it is obviously more expensive since it is an improved version of D435i, with most of its good characteristics saved and the bad ones improved. It has an enhanced range of view due to its pair of IR stereo cameras set at a wider distance from each other on the camera body. For this project, D435i seems a suitable choice, while for use with the real robot, D455 will obviously be better.

The work was supposed to be done on a compact computing board so as to imitate the processes to be done later on the robot car (which has a similar board acting as a main processing unit). There are plenty of embedded platforms present on the market. Among them, Nvidia Jetson Nano was chosen as the most suitable for the project, being relatively low-cost and best by price/performance ratio[6]. In general, Nvidia has quite a big community of developers, and much information is present about its effective-in-use GPU. Since there is no huge computational power required for this project from an embedded system, this seems to be the best choice. The base image for Jetson Nano also includes Linux Ubuntu 18.04, allowing for enhanced performance due to possible additional adjustments through the terminal and direct access to code links.

[4] Phygitalism. “Kamery Glubiny. Obzor Ustrojstv.” *Medium*, PHYGITALISM, 3 Nov. 2020, https://medium.com/phygitalism/rgb-d-sensors-devices-430e8782feeb.

[5] Gurylev, Viktor. “Intel RealSense d435i: Nebol'shoe Obnovlenie i Nebol'shoj Istoricheskij Ekskurs.” *Habr*, Habr, 29 Nov. 2018, https://habr.com/ru/company/intel/blog/430720/. [Last accessed 28/05/2022]

[6] Mal’cev, Anton. “Ul'timativnoe Sravnenie Embedded Platform Dlya Ai.” *Habr*, Habr, 26 Oct. 2020, https://habr.com/ru/company/recognitor/blog/468421/. [Last accessed 28/05/2022]

Though it was possible to work directly on a Jetson machine, there were too many dependencies and packages required for the project, especially if taking into account that every person in this team was completing their own task, but the goal was to connect all of this on the same robot, so it would work cohesively. For this purpose, the Docker application was used, which allows for the creation of a cloud-based container. Inside the container, which can be utilized by multiple users at the same time independently, the whole stack of needed packages was installed (including Intel RealSense SDK 2.0, ROS2, etc.).

All the coding was done with C++ 17. In comparison to Python, which can also be used for ROS programming, it presents higher speed and works more effectively with memory.

For in-code work with images, OpenCV was used. It is an open-source, cross-platform computer vision library that presents a plethora of tools for image processing. Here, it allowed to convert RealSense image data to a special OpenCV matrix so that it is possible later to convert the matrix to the special Image message type of ROS using cv_bridge. No other library except OpenCV presents such an ability to transfer images through ROS.

The images brought from ROS were visualized in Foxglove Studio. It is an open-source app for robotics, which provides configurable tools and supports ROS, so it is possible to understand the data state and get depth images in real-time[7]. The app also works with the rosbridge_suite package, which allows it to connect to a running ROS system through the WebSockets computer communication protocol.

Finally, a Tmux terminal multiplexor was used since the processes of rosbridge connection with Foxglove Studio and ROS publishing node should run simultaneously.

2.1 Functional and Non-Functional Requirements

After analyzing analogues and setting the main instruments for use, it is possible to formulate the functional and non-functional requirements for the project part of mine.

2.1.1 Functional

- ROS node[8] accepts a depth image as an input
- ROS node publishes messages with depth images to the specified topic

[7] "FoxgloveStudio." *Ros.org*, http://wiki.ros.org/FoxgloveStudio. [Last accessed 28/05/2022]
[8] The simplest process that performs computation. See more in 3.1 ROS section.

- Along with the images themselves, the general data about format, publish time, image size, frame ID is published in the same message
- Depth images are colorized according to the depth data for better perception by the human eye
- Depth images are viewed in Foxglove Studio, which subscribes to the mentioned topics
- The data is published in topics only if they have a subscriber

2.1.2 Non-Functional

- Work with ROS node is intended to be done through Docker image on Linux Ubuntu 18.04
- The user is expected to provide the image data from a specified camera type, Intel RealSense D435i
- Depth data is accepted as grayscale 16-bit image through the usage of specific RealSense depth image format Z16
- The recommended image size is 848 x 420 pixels
- The output image format is BGR8 (3-channel 8-bit image)
- The colorization is performed with the usage of Hue color scheme
- Inside the node, images data is converted from RealSense format to ROS messages via OpenCV and cv_bridge
- Foxglove Studio should subscribe to the mentioned topics through rosbridge

3. Theory

3.1 ROS

The ROS architecture is not complicated in general. Inside it, there are independent **nodes**, each referring to a single specific task that they are required to perform. There are also **topics**, which serve as connecting units between nodes. This way, the information from **publisher** nodes is sent to the topic in the form of a message, while **subscriber** nodes connected to the same topic will receive the information in that message. The visual representation of this architecture is presented in *Image 1*. There, you can also see elements as **service**, **server**, **client**, **request,** and **response**, but they will not be considered in this report due to their immateriality to the project.

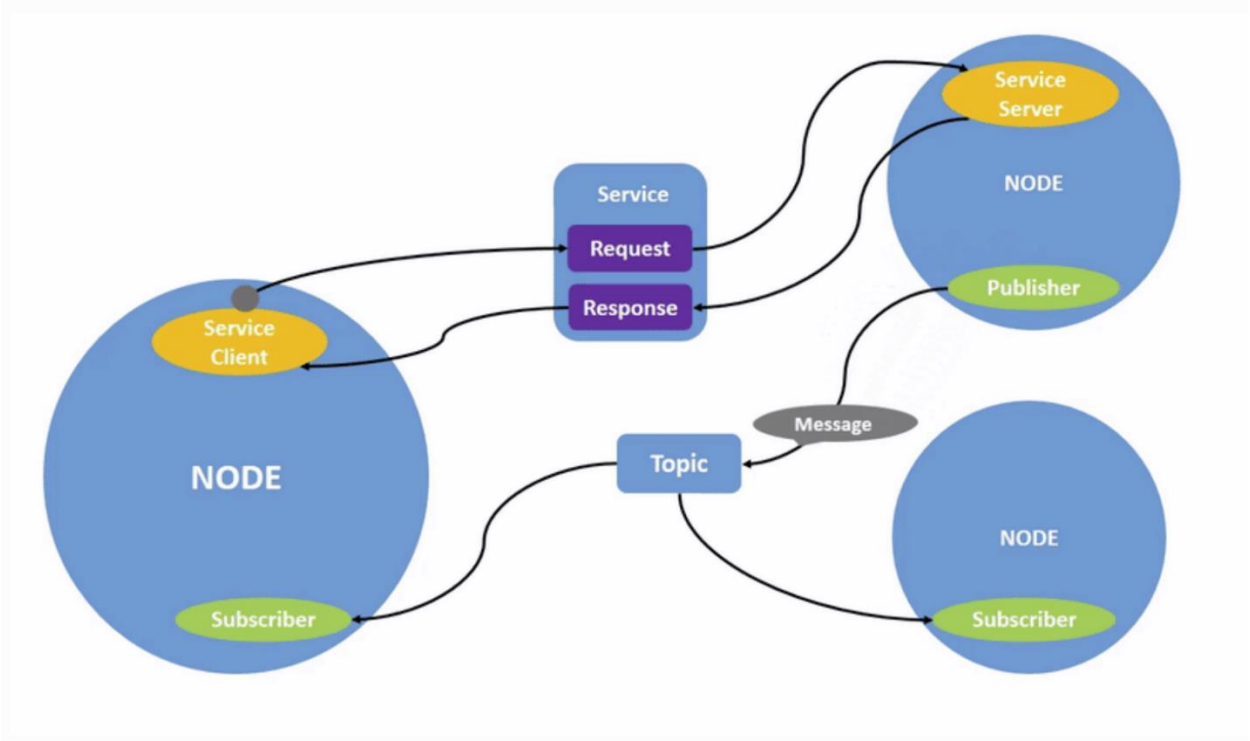

Image 1. ROS architecture.

Source: https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Nodes.html

While the described ROS structure seems not to be complex, the actual content of nodes is what allows for the flight of the programmer's imagination. Inside them, basically, any action within the ROS API can be done. This is an important part of ROS nodes, which was largely utilized in the implementation of the product.

ROS supports a wide range of message types, which allow the transfer of information between nodes. Depending on the type, different information can be sent, including simple text and arrays of data. A message type specifies the data structure that can be used with it, while all the information about the data types which can be used within a specific message type is written in

.msg files[9]. For example, **std_msgs/String** is a message type that can transfer only raw messages of type string[10], while **sensor_msgs/Image** type (which is of the most interest for this project) consists of a long list of simple data that should be provided, starting from the header datatype, which includes timestamp and frame_id, and finishing with actual data about the image – its integer size and matrix data, represented as an 8-bit integer array[11].

3.2 Depth Camera

Depth cameras are different from usual cameras in quite an understandable manner – they are able to detect the distance from their body to an object. In more technical terms, every pixel of an image stores not a color, like in a normal camera, but a distance value from this point to some objects in their range. There is a list of different types of such cameras, all varying in the method of depth calculation. The camera type used in this project utilizes stereo images to create a depth map, so this algorithm is going to be discussed here.

The principle of work of Depth from Stereo cameras is as follows. The IR Projector sends the signal to the object, which is, at the same time, collected by the pair of parallel and similarly directed Depth Sensors (or IR Stereo Cameras). The difference between the key points on the images collected by these sensors, called disparity, is what helps to find depth. *Image 2* shows the location of all the described elements on the RealSense D435i camera body.

In some sense, the Depth from Stereo algorithm is similar to animal binocular vision[12], which allows different species, including humans, to create a 3D image of objects in front of them. Consider the methods to calculate disparity and the disparity map.

3.2.1 Calculating Single Disparity

The disparity can be calculated quite intuitively using the formula (3.1).

$$disparity = x - x' = \frac{Bf}{Z} \tag{3.1}$$

Looking at *Image 3*, this formula tells us that disparity, which is equal to the difference between two points, is received by multiplying baseline, which is the difference between cameras'

[9] "common_msgs." *Ros.org*, https://wiki.ros.org/common_msgs. [Last accessed 28/05/2022]
[10] "Std_msgs/String Message." *std_msgs/String Documentation*, http://docs.ros.org/en/noetic/api/std_msgs/html/msg/String.html. [Last accessed 28/05/2022]
[11] "Sensor_msgs/Image Message." *sensor_msgs/Image Documentation*, http://docs.ros.org/en/api/sensor_msgs/html/msg/Image.html. [Last accessed 28/05/2022]
[12] Dorodnicov, Sergey. "Depth from Stereo." *GitHub*, 12 June 2018, https://github.com/Usphera/MedVirtua/blob/master/doc/depth-from-stereo.md. [Last accessed 28/05/2022]

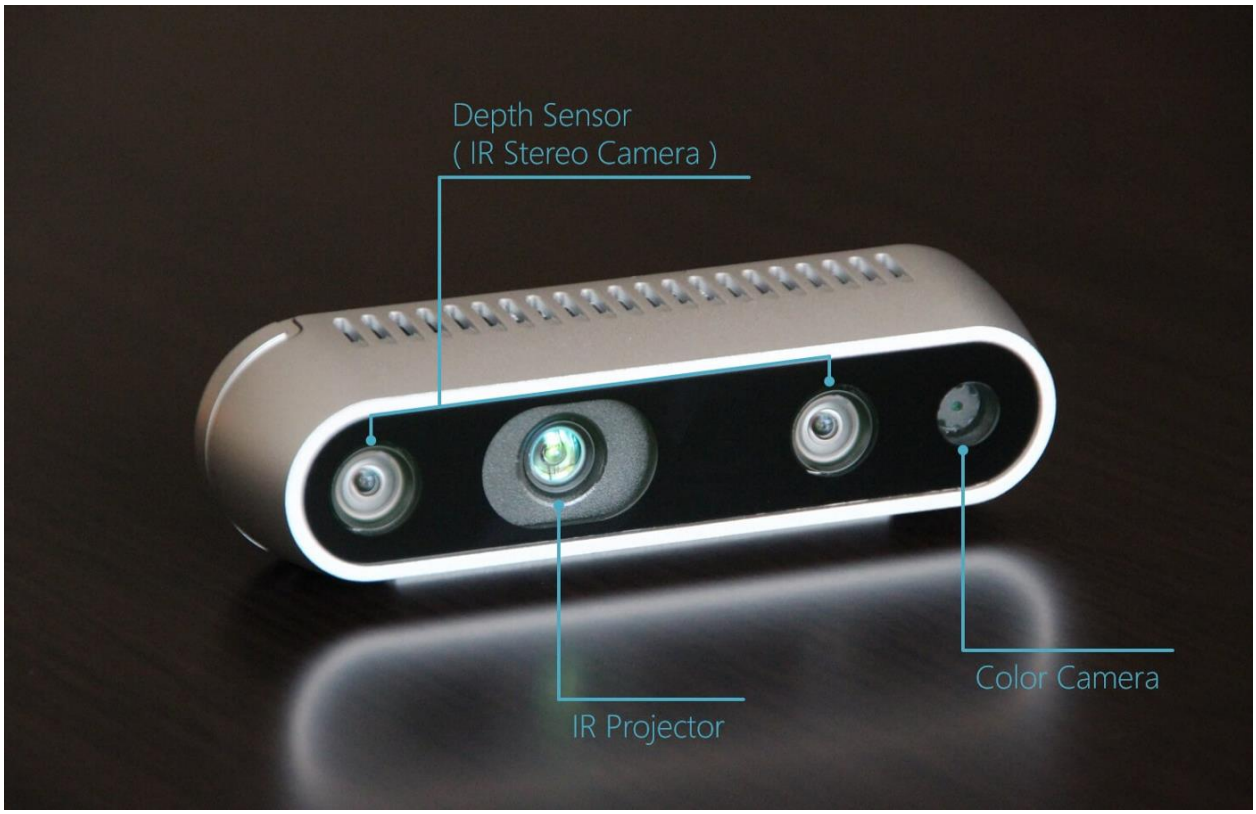

Image 2. Intel RealSense D435i camera body and location of depth-related elements on it.
Source: https://medium.com/phygitalism/rgb-d-sensors-devices-430e8782feeb

locations, by focal length, which is the distance from cameras to the image plane, and dividing this by distance to the object. Hence, the corresponding match between the two images is found. Assuming that depth is what we are required to find for the point X on *Image 3*, we can reform the formula (3.1) to get formula (3.2).

$$Z = \frac{Bf}{x - x'} \tag{3.2}$$

Therefore, it becomes clear that disparity is inversely proportional to depth, so disparity can be converted to depth through triangulation easily.

The focal length of the camera and the distance between image points on the plane are generally known from the intrinsic camera matrix from formula (3.5), which, in its turn, is obtained from the full camera matrix from formula (3.3).

$$P = [M| - MC] \tag{3.3}$$

In formula (3.3), M is an invertible 3x3 matrix, and C is a column vector representing the camera's position in world coordinates. The full camera matrix, as already mentioned, can be represented as a product of two matrices: extrinsic and intrinsic. This is shown in formula (3.4).

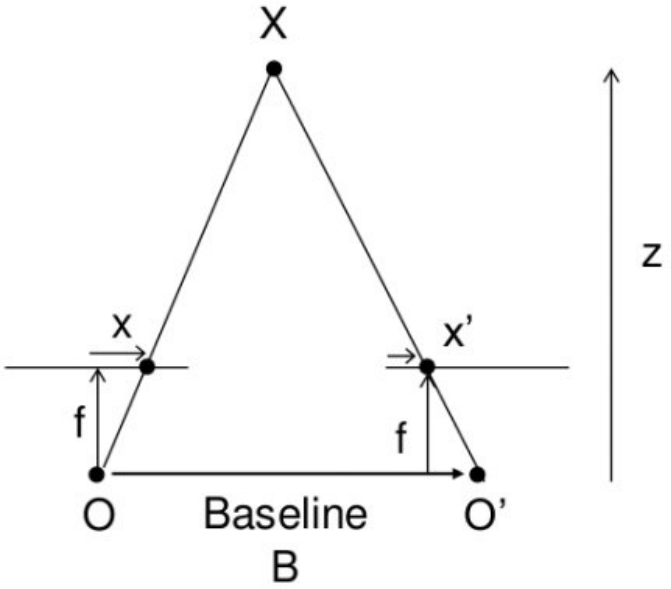

Image 3. Diagram telling how to acquire depth information about scene from two images of this scene. O and O' - location of pair of cameras, B – distance between cameras, f – focal length of camera, x and x' - points in image plane, Z – distance to the object X.
Source: https://docs.opencv.org/3.1.0/dd/d53/tutorial_py_depthmap.html

$$P = K[R|-RC] = K[R|t] \tag{3.4}$$

The extrinsic part in formula (3.4) is represented by the block-form matrix $[R|-RC]$, where R is the 3x3 rotation matrix whose columns show the directions of the world axes in the camera's frame of reference, and C is already mentioned column vector. Combination of R and C with minus sign on the right side of the block-form matrix gives the position of world origin in camera coordinates and is represented by t. K is an intrinsic camera matrix, which is shown in formula (3.5). The extrinsic camera matrix will not be considered in detail.

$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.5}$$

f_x and f_y presented in matrix in formula (3.5) are focal lengths to the corresponding points in the image plane. These are considered equal in most cases, so it is assumed that camera is calibrate and no unintentional distortion occurred while processing the image. x_0 and y_0 are principal point offsets, showing the distance of the principal point from the origin point by x and y coordinates on the image plane. See *Image 4* for the illustration of the principal point offset. The last parameter in the matrix in formula (3.5), s, is an axis skew, which shows shear distortion. It is set to be zero since we assume there is no distortion in our case. Therefore, we obtain a new formula (3.6) for disparity calculation in our case. Note that for each camera out of two. we have a different intrinsic matrix, where the focal length will be the same, but principal point offsets will vary.

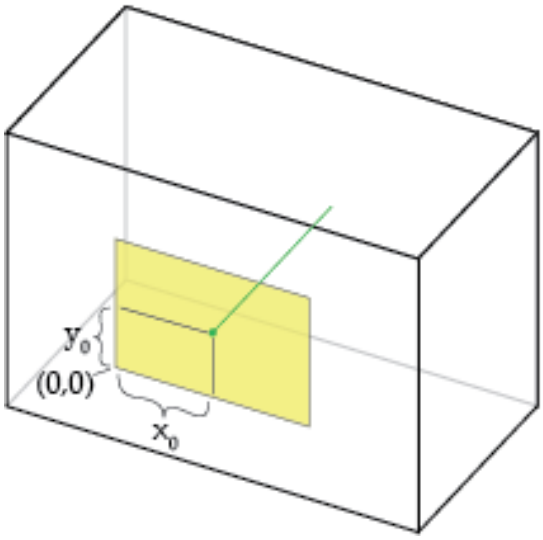

Image 4. Illustration of principal point offset concept. (0,0) is the origin, x_0 is an offset by x axis, y_0 is an offset by y axis.

Source: http://ksimek.github.io/2013/08/13/intrinsic/

$$K = \begin{pmatrix} f & 0 & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

It has now become clear how to calculate a single disparity for one point and how to derive the distance to that point. On the camera side, however, the calculations of disparity are more complex due to the need for its calculation for every single image pixel.

3.2.2 Calculating Disparity Map

The stereo matching done in the camera proceeds a few steps until the ready disparity map, or depth image, is output. First, the pair of images is collected on a calibrated binocular camera. Then, the rectification is performed, so that the corresponding points in two images lie on the same horizontal line. The process of image rectification can be observed in *Image 5*. After that, the correspondence problem is addressed, which is about determining the pair of pixels that are projections to the same point in space. The most common approach for this is the Block Matching algorithm, which is based on comparing a small "window" around the considered point in the first image with a few similar "windows" along the same horizontal line in the second image. Afterwards, the loss function is computed either using the Sum of Squared Differences (SSD) or using the Sum of Absolute Differences (SAD). These are shown in formulas (3.7) and (3.8) respectively.

$$SSD(W^L, W^R) = \sum_{i=1}^{N} \sum_{j=1}^{M} \left| W_{ij}^L - W_{ij}^R \right| \tag{3.7}$$

$$SAD(W^L, W^R) = \sum_{i=1}^{N} \sum_{j=1}^{M} \left(W_{ij}^L - W_{ij}^R \right)^2 \tag{3.8}$$

In both formulas (3.7) and (3.8) W^L and W^R denote the left and right "windows" (or blocks of points) respectively, while M x N is the blocks' size. Between these functions, SAD is preferable in general due to the lower noise and smaller number of outliers it presents. After calculating loss values for every considered block of pixels, the block with the lowest value is the desired match. Hence, the disparity value can be found by formula (3.9).

$$disparity = d(x, y) = x' - x \tag{3.9}$$

In formula (3.9), and x and x' represent x-coordinates in the first image and in the second image accordingly, y stays constant since we move by horizontal line.

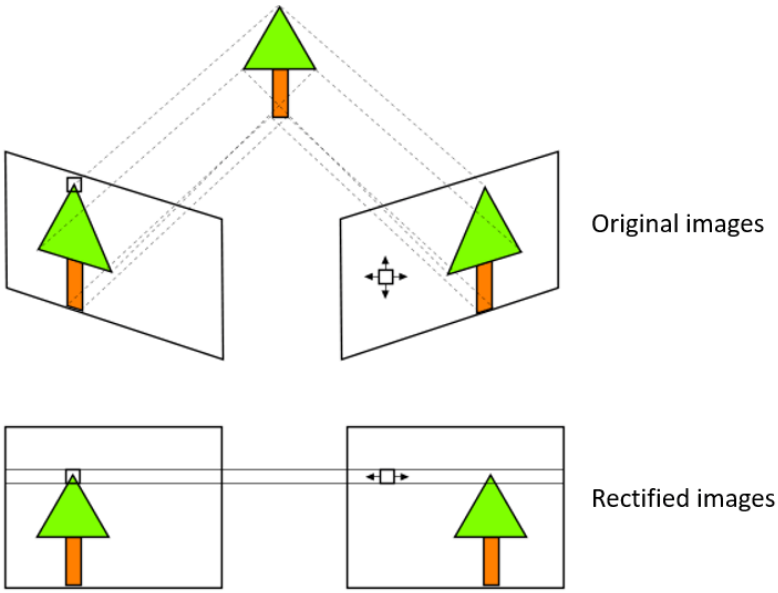

Image 5. The process of image rectification.
Source: https://www.baeldung.com/cs/disparity-map-stereo-vision

Performing all the described steps and applying the Block Matching algorithm to every pixel will lead to obtaining the disparity map (which is easily turned into a depth image, as already discussed in the 3.1.1 Calculating Single Disparity section). An example of it is shown in *Image 6*.

Applying the discussed depth image theory to the Intel RealSense D435i camera, it actually performs all the computational processes itself, so that the depth image is easily received by an average user. The functions, which allow one to acquire depth data and configure camera parameters, are defined in the Intel RealSense SDK.

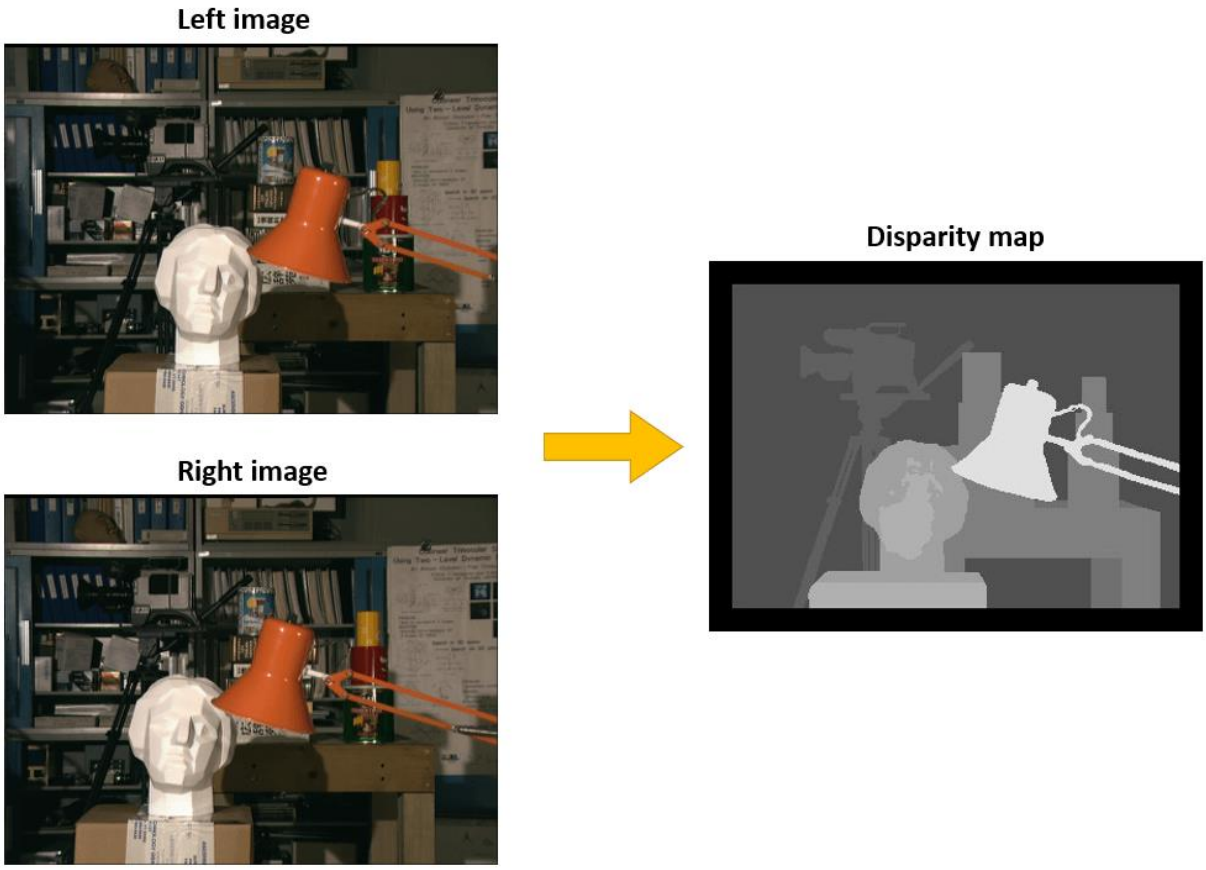

Image 6. Example of how disparity map is obtained. The farther objects are from the camera, the darker they look.

Source: https://www.baeldung.com/cs/disparity-map-stereo-vision

4. Implementation

4.1 Main Algorithm

As already mentioned in the 1.1 Purpose and Relevance section, the main goal of my part of the project was to integrate the Intel RealSense D435i depth camera into ROS such that depth images are successfully sent to the ROS topic, wherefrom they can be acquired by other devices subscribing to it. Since ROS and Intel RealSense SDK are not completely compatible and a RealSense depth type image cannot be sent as a message directly to the specified topic, the C++ code was written to connect these two. Its algorithm can be split into the following steps:

1. Create the ROS node and launch topics
2. Acquire a depth image from the RealSense D435i camera
3. Obtain the OpenCV matrix from depth data
4. Convert an OpenCV matrix to a ROS message type using cv_bridge
5. Publish the message to the appropriate topic

Consider these steps one by one.

The first step includes initialization of the roscpp package and setting the ROS node to spin (e.g., work infinitely until ROS invokes a shutdown) in the main() function. Next follows the creation of the ImagePublisher class, which is inherited from the Node class of the roscpp library. The class is divided into public and private sections. In the public section, the general constructor names the node image_publisher and the topic for the image to be published to is defined altogether with a timer, which allows the node to publish messages every second. Moving to the private section, the timer_callback() void function is the main function to publish messages on a repetitive basis. Inside, the following processes of image convert.

The second step begins in the node class's public section. The depth stream is enabled for the camera, with the required size and type set. Then, the parameters for the colorizer are also chosen: histogram equalization, which is a method of improving contrast in the image for better color distribution, and Hue color scheme. The rs2::colorizer class allows to paint depth images, which are initially greyscale, so that they are better perceived by the human eye. The pipeline is then started to process image data from the camera. These were the functions that were applied once to configure the RealSense system. Now, moving to the actual depth data processing. In private function timer_callback() the frames are received from the pipeline, and the depth data is collected, being colorized simultaneously.

Next goes the step of converting depth data into OpenCV matrix. This is done through creating a new object of cv::Mat type and initializing it with image size, OpenCV 3-channel 8-bit image encoding and given depth data.

After the OpenCV matrix is obtained, we can transform it to the specific ROS message format sensor_msgs/Image, which holds all the general information about the image. cv_bridge is a ROS library that allows to do that, while the only things user needs to specify explicitly are encoding and the message header. The header collects such data as a timestamp and frame_id for the required message object. After setting these, it is possible to create a complete instance of message with sensor_msgs/Image format using cv_bridge function .toImageMsg().

The final part of the algorithm is the publish function, which allows one to send the chosen message with all the data inside it to the chosen topic. The publisher function is put inside the if statement, which checks for the existence of node subscribers, whereas the number of subscribers is received through rclcpp function get_subscriptions_info_by_topic. Therefore, if no subscribers are present for the topic, the message is not published, and the other way round.

After discussing the main steps of the image capture and publishing algorithm, we can move to the launching procedure.

4.2 Program Launch and Testing

To launch the program, the C++ code file is not enough. ROS requires several additional files to build the package and run it. First, it is the CMakeLists.txt file. Inside it, the basic system requirements are set, the main libraries are found, and dependencies are clarified. Hence, the build tool knows where to find information for the specific function or class within the code. Second, package.xml is also required. This is a file with general meta information about the package. The template was taken from the ROS website[13], and some fields were changed to the relevant ones. This file also states the build tool and library dependencies in a similar way to CMakeLists.txt. With these documents being ready, it is possible to proceed to the next step – building.

[13] "Creating Your First Ros 2 Package." *Creating Your First ROS 2 Package - ROS 2 Documentation: Galactic Documentation*, https://docs.ros.org/en/galactic/Tutorials/Creating-Your-First-ROS2-Package.html. [Last accessed 28/05/2022]

The colcon is a unified build tool that combines several ROS build tools in one[14]. Using it, we can build the package, which is basically the name of the directory the three files described above are saved in. Using the command (4.1) and inserting the package name instead of curly brackets, the package will be built.

colcon build --packages-select {package name} (4.1)

As soon as the build is finished successfully, it is possible to run the publisher node. However, the result will not be observed until we are connected to the Foxglove Studio.

Start Tmux with 2 separate running sessions. In one, we run rosbridge_suite for ROS2 through calling a command (4.2).

ros2 launch rosbridge_server rosbridge_websocket_launch.xml (4.2)

In other session, we first source the setup files by executing command (4.3), and then run the node by using command (4.4). In (4.4), instead of the first curly brackets, we put the package name, and instead of the second curly brackets the executable name, which is specified in CMakeLists.txt.

. install/setup.bash (4.3)

ros2 run {package name} {executable name} (4.4)

So, we arrive at two simultaneously running processes: the rosbridge connection and the publisher node. After setting the rosbridge connection on the other side, at Foxglove Studio, the image itself and the data describing it are successfully captured and visualized. This is presented in *Image 7*.

4.3 Additional Feature: Image Compression

After implementing the general algorithm, additional actions were taken to optimize its work slightly. The decision was made to perform compression of images within the node before publishing to topic is performed. This could decrease the load of the CPU to some point, which is especially important for self-driving robot car since it has plenty of processes running on the CPU at the same moment. The most common compression method was chosen – JPEG. Thanks to the cv_bridge library, the function .toCompressedImage() exists, which can easily compress images to JPEG format, at the same time receiving relevant ROS message type

[14] "Using Colcon to Build Packages." *Using Colcon to Build Packages - ROS 2 Documentation: Galactic Documentation*, https://docs.ros.org/en/galactic/Tutorials/Colcon-Tutorial.html. [Last accessed 28/05/2022]

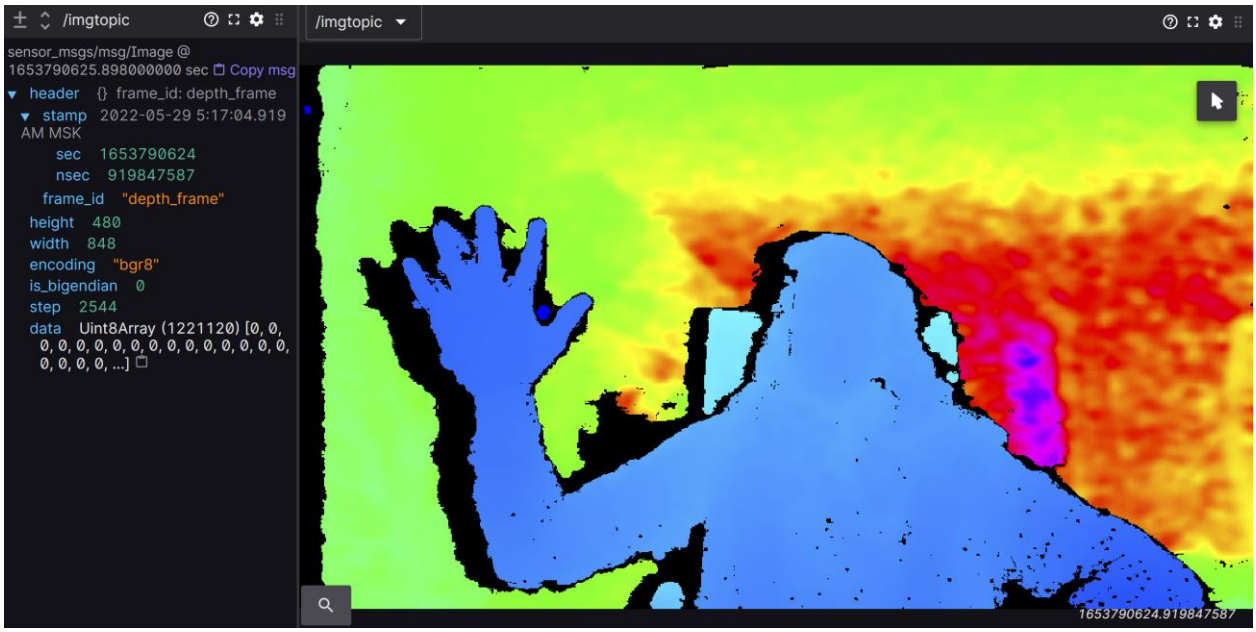

Image 7. The result of algorithm: depth image and information about it viewed from Foxglove Studio. On the right: image represents a person with raised hand sitting on a chair in a big room. Closer objects are colored in blue, and change according to the color circle, finishing at dark purple at the farthest point. On the left: main information about the image includes frame_id, timestamp, size (height, width) and encoding.

sensor_msgs/CompressedImage.

A minimal number of lines are added in the code for compressed image publishing. The separate topic was created with a relevant name, then separate header file with corresponding data was created, and, finally, the function .toCompressedImage() transforms the same depth data from the OpenCV matrix to a compressed image in JPEG format. The separate publisher function is created to publish different message types to different topics. The same if structure is used for on-demand publishing.

The rest of the procedure is the same as for the usual image. The results, therefore, are observed at Foxglove Studio, see *Image 8*.

4.4 Analysis and Evaluation of Results

The results of the completed work are pretty much satisfying. The program successfully collects depth image data from the Intel RealSense D435i camera, converts it to am instance of ROS message, and sends it to the topic via publisher node. As seen from *Image 7*, the quality is good enough to distinguish one object from another, as well as to understand which object is closer to the camera and which is farther. The relevant information is written in the raw messages box on the left of *Image 7*, indicating that the correct size and formats were used. In general, it is seen

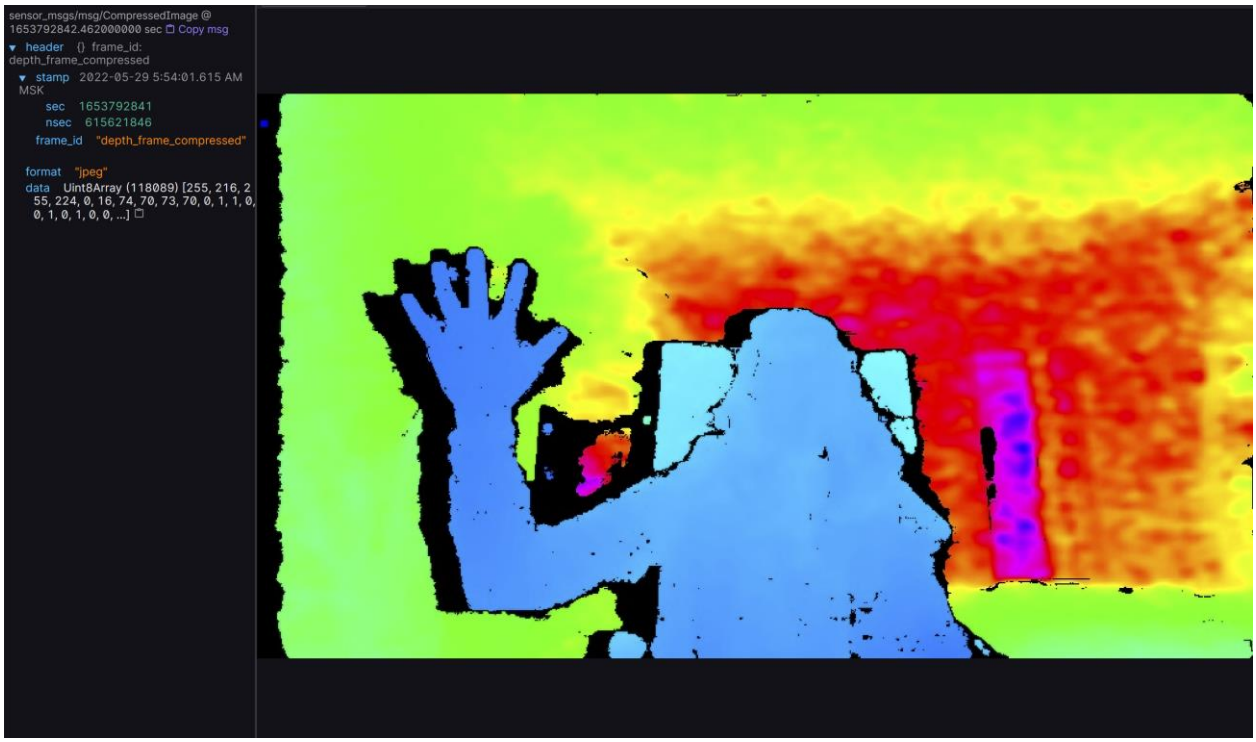

Image 8. The result of algorithm: compressed depth image and information about it viewed from Foxglove Studio. On the right: compressed image with characteristics similar to previously shown usual image. On the left: main information about the compressed image includes frame_id, timestamp, and compression format.

that the requirements stated in 2.1 Functional and Non-Functional Requirements were completely fulfilled.

Considering the compressed image, the results appeared to be much better than expected. Since JPEG is said to be a lossy approach, where large part of image quality is lost, it was assumed that the picture would have depth data about objects shown in a partly incorrect manner. However, as observed in *Image 8*, the received depth image was highly similar to the usual image, no visual differences were found.

5. Conclusion

5.1 Main Result

The general result of the work done is pleasing in its completeness. The initially stated goal was achieved as required, though hard work of combining different instruments was done for it. Besides that, the additional task of compressing an image was set afterwards and successfully implemented. This led to additional practice for me and advantages for car robot project. Overall, the fact that this project is a part of something bigger motivated me a lot. At every step of work, I told myself that the program must not fail, because part of the robot system relies on the outcome of my part of the project.

Apart from the physical benefit, I also received plenty of new knowledge related to different aspects that I would unlikely have received under other conditions. The camera structure algorithms of depth recognition are not the topics that I was somehow interested in before I started reading project-related books and Internet sources, and now they are the topics I know a lot about. The understanding of ROS concepts will certainly benefit me in my further life, if not by direct use, then at least because of their universal logic. Additional C++ code writing and documentation exploration never makes things worse. I also got the unique experience of working within a big team of professionals. All from different spheres, but they collaborated for the sake of creating something beautiful. Personally, I have always wanted to work with robots, and this project allowed me to try.

5.1 Prospects for Further Work

The main future prospects for the work of mine are quite obvious – the written algorithm should be optimized as highly as possible to work with robot car, so that it is then used on it for SLAM. First of all, the image format both in the OpenCV matrix and the ROS message should be changed to CV_16U and mono16, respectively. These formats represent 1-channel, 16-bit monochrome images, which are more suitable for robot due to easier depth recognition and data interaction. Colorizer is, basically, removed due to uselessness. The next possible task could be to perform image compression through nvJPEG. This is a Nvidia library that allows to work with JPEG images in a more optimal way through accelerating their encoding and decoding with the help of GPU resources. As stated on their website, applications that rely on nvJPEG deliver

higher throughput and lower latency compared to CPU-only decoding[15]. The move could be made to publish video frames instead of images, processing these effectively by applying GPU-based compression to them. Finally, if no other improvements are thought up, the task will move to the implementation of SLAM algorithms.

[15] "NvJPEG." *NVIDIA Developer*, 18 Feb. 2022, https://developer.nvidia.com/nvjpeg. [Last accessed 29/05/2022]

References

1. Kron, Andrej. “Roboty v Chelovecheskom Obshchestve.” *Habr*, Habr, 16 Sept. 2017, https://habr.com/ru/company/unet/blog/337902/.
2. Kumar, Akshay. “An Introduction to Simultaneous Localization and Mapping (SLAM) for Robots - Technical Articles.” *Control*, 1 May 2020, https://control.com/technical-articles/an-introduction-to-simultaneous-localization-and-mapping-slam-for-robots/.
3. “Why Ros?” *ROS*, https://www.ros.org/blog/why-ros/.
4. “Understanding ROS 2 Nodes.” *Understanding ROS 2 Nodes - ROS 2 Documentation: Galactic Documentation*, https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Nodes.html.
5. Merzlyakov, Alexey. “Ros: Stan' Kontrib'yutorom Samogo Bol'shogo Open Source Proekta v Robototekhnike.” *Habr*, Habr, 6 Aug. 2021, https://habr.com/ru/company/samsung/blog/571302/.
6. “Wiki.” *Ros.org*, https://wiki.ros.org/common_msgs.
7. “Wiki.” *Ros.org*, http://wiki.ros.org/Messages.
8. S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” Science Robotics vol. 7, May 2022.
9. “Which Intel RealSense Device Is Right for You? (Updated June 2020).” *Intel® RealSense™ Depth and Tracking Cameras*, 21 Mar. 2019, https://www.intelrealsense.com/which-device-is-right-for-you/.
10. Gurylev, Viktor. “Intel RealSense d435i: Nebol'shoe Obnovlenie i Nebol'shoj Istoricheskij Ekskurs.” *Habr*, Habr, 29 Nov. 2018, https://habr.com/ru/company/intel/blog/430720/.
11. Gurylev, Viktor. “Intel RealSense Depth Camera d455 - Четвертая Из Серии.” *Habr*, Habr, 27 July 2020, https://habr.com/ru/company/intel/blog/511848/.
12. Dorodnicov, Sergey. “Depth from Stereo.” *GitHub*, 12 June 2018, https://github.com/Usphera/MedVirtua/blob/master/doc/depth-from-stereo.md.
13. “Intel® RealSense™ - Obzor Kamer.” *Internet Magazin Mikrokomp'yuterov i Aksessuarov*, 12 Aug. 2020, https://evo.net.ua/intel-realsense-obzor-kamer/.
14. “Depth Map from Stereo Images.” *OpenCV*, 18 Dec. 2015, https://docs.opencv.org/3.1.0/dd/d53/tutorial_py_depthmap.html.

15. Hartley, Richard, and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed., Cambridge University Press, 2004.
16. Simek, Kyle. “Dissecting the Camera Matrix, Part 1: Extrinsic/Intrinsic Decomposition.” *A Computer Vision Blog*, 14 Aug. 2012, https://ksimek.github.io/2012/08/14/decompose/.
17. Simek, Kyle. “Dissecting the Camera Matrix, Part 3: The Intrinsic Matrix.” *A Computer Vision Blog*, 13 Aug. 2013, https://ksimek.github.io/2013/08/13/intrinsic/.
18. “Calculate the Disparity Map.” *Python Computer Vision*, https://programmersought.com/article/50045374193/#23__126.
19. Baeldung, Eugen. “Disparity Map in Stereo Vision.” *Baeldung on Computer Science*, 26 May 2022, https://www.baeldung.com/cs/disparity-map-stereo-vision.
20. McCormick, Chris. “Stereo Vision Tutorial - Part I.” *Stereo Vision Tutorial*, 10 Jan. 2014, https://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/.
21. “Writing a Simple Publisher and Subscriber (C++).” *Writing a Simple Publisher and Subscriber (C++) - ROS 2 Documentation: Galactic Documentation*, https://docs.ros.org/en/galactic/Tutorials/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html.