ECMI Modeling Week 2023
University of Szeged, Faculty of Science and Informatics

STEREO DEPTH CAMERA CALIBRATION AND CONTROL SUPPORT SYSTEM

Szabolcs Szalánczi (B+N Referencia Zrt.)

Leonardo Rafael Amado Brito (Instituto Superior Técnico)
Luca Ratki (University of Szeged)

August 30, 2023

Contents

1 Introduction

In the realm of robotics, precision is paramount when it comes to tracking the position and movement of devices. While many off-the-shelf solutions exist for this purpose, they often fall short in terms of the desired accuracy or the seamless synchronization of various sensors. A significant challenge lies in accurately projecting depth data onto predefined planes and minimizing the measurement errors inherent in sensor readings. Additionally, leveraging the capabilities of an inertial measurement unit (IMU) for tracking device movement can offer an extra layer of insight to complement image processing techniques.

At the heart of this endeavor is the Intel RealSense Depth Camera D435i, a sensor equipped with both depth and IMU capabilities. This project revolves around harnessing the potential of this advanced hardware using the pyrealsense2 module, a Python-based interface to interact with the camera's functionalities, but also exploring more capabilities of the camera.

Our primary objective was to calibrate, measure, track, and recognize. We started to construct a comprehensive framework that not only manages and rectifies real-time projection of both BGR and depth camera data but also extends its capabilities to integrate data from the built-in IMU. Through the lens of computer vision, projective geometry, statistics, and proficient Python programming, we aimed to address the challenges of accurate device positioning in the robotics domain.

This report delves into the methodologies, strategies, and outcomes of our endeavor, emphasizing the critical role of fine-tuning the Intel RealSense Depth Camera D435i's capabilities for precise and reliable robotic position tracking. We explore the complexities of depth projection, sensor error reduction, and optional IMU integration. Additionally, we highlight the significance of transfer learning by fine-tuning pre-trained models and the insights derived from this process.

Keywords/Skills: computer vision, projective geometry, statistics, Python programming.

2 Analyse the camera

2.1 Setup

To start off we wanted to check how accurate the camera's information about the depth values are, and if there is a difference when the camera is further away or closer to whatever the camera is aimed at. We set up an environment where all the pixels are supposed to be at the same depth (we chose a wall). We set up the camera in front of the wall with expected depth values of 660 mm, 1160 mm, 1660 mm, 2160 mm, and 3160 mm. Then from each setup, we saved every 100th frame ranging from 100 to 1200 frames.

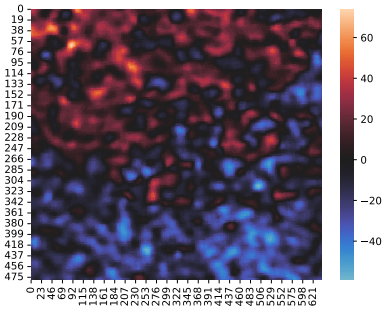

Figure 1: This heatmap represents the difference between the measured and the expected depth values for every pixel. The expected depth in this case is 1160 mm.

The expected depth pictures are the same matrices, where every element is 660 (or in other cases 1160, 1660, 2160, and 3160).

We evaluated the errors by subtracting the expected values from each matrix. These errors can be represented as an error matrix, which is shown in 1. These heatmaps are centered around 0 as they represent errors.

2.2 Heatmaps

We created heatmaps using the error matrices, and as can be interpreted, we have some frames with pixels that are extremely "off", making the heatmap hard to understand and untrustable 2.

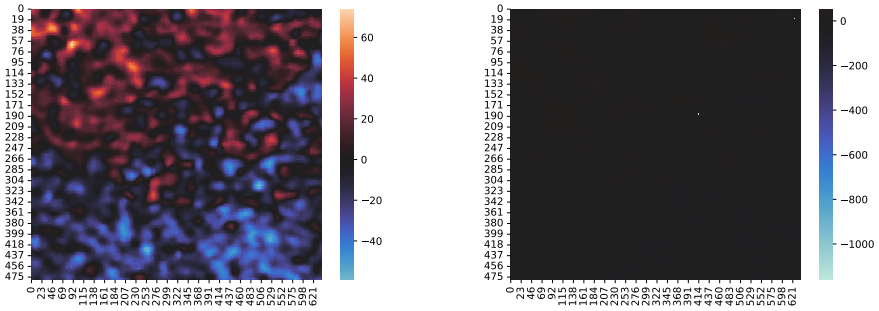

Figure 2: In this pictures, we see two different frames, from the same picture, with the same expected depth matrix. We see a great noisy error matrix on the first one, but in the second one, some pixels are very "off", making the image hard to analyse.

To solve this problem, we mixed the matrices from different frames (but same expected depth) into one matrix in the following way:

$$E'_{i,j} = \min_{f \in frames} e^{(f)}_{i,j},$$

where $e^{(f)}_{i,j} \in E^{(f)}$ is the element of the error matrix corresponding to the frame f with the index (i, j).

This way, we have a corrected new matrix, which contains minimal errors in each element. This method is much better than smoothing the matrices with calculating the mean, because of the outliers.

Here we have the heatmaps of the smoothed errors, with 0 being the center.

As we can see from these pictures, there are some special cases. Upon moving the camera backward, the left wall and a table on the right side started disrupting the picture. To avoid any excess noise in the pictures, we cropped them to a fixed size to not include the wall and table. We kept the pixels from 200 to 400 in each axis, so we got squares (The original pictures were 420×640 pixels large).

In Figure 3 we present the error heatmaps of different expected depth images corrected in the above-mentioned way.

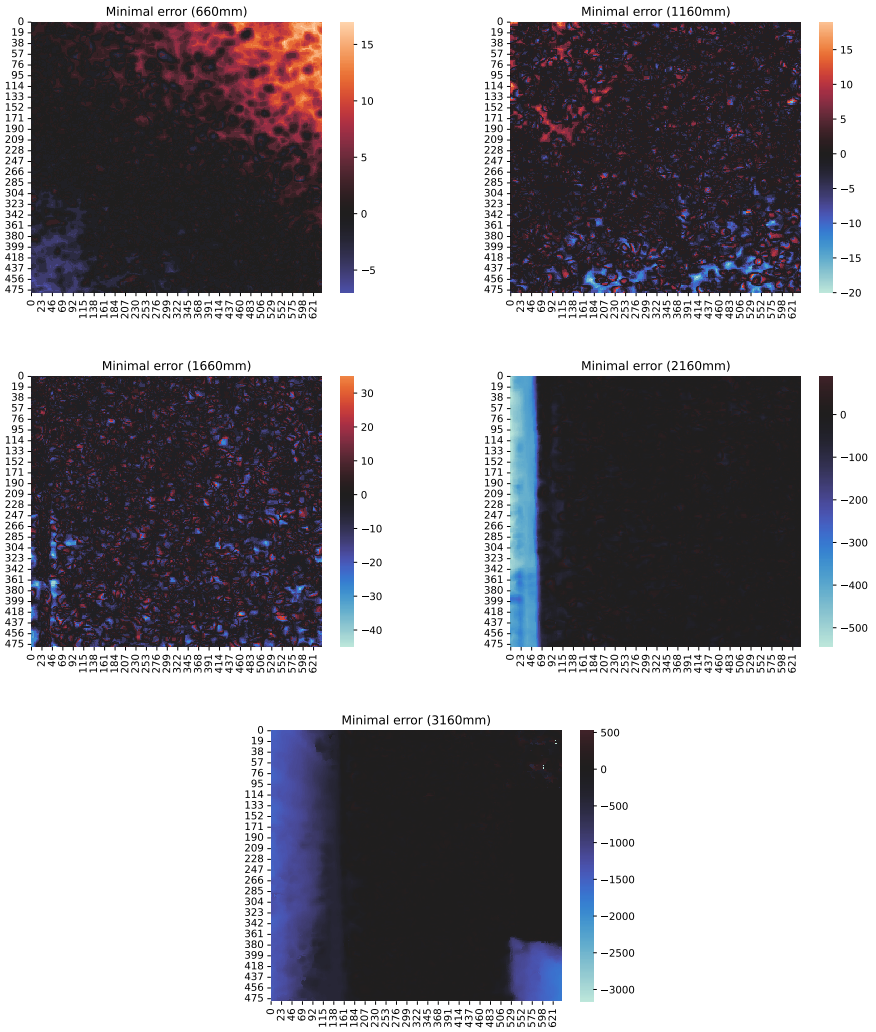

Figure 3: On these images, we can compare the errors by the depth. As expected, the closer images have smaller errors. However, we can see some consistent error as well, which is caused by the setup.

The first image is a little off in a consistent way, by which we assumed, that the angel of the camera was a little off when we got the picture.

The matrices with higher expected depth are very off but in a consistent way. It is because the other wall of the room and a table are in the picture. To be able to analyze these setups, we need to crop the images. The cropped heatmaps are shown in Figure 4

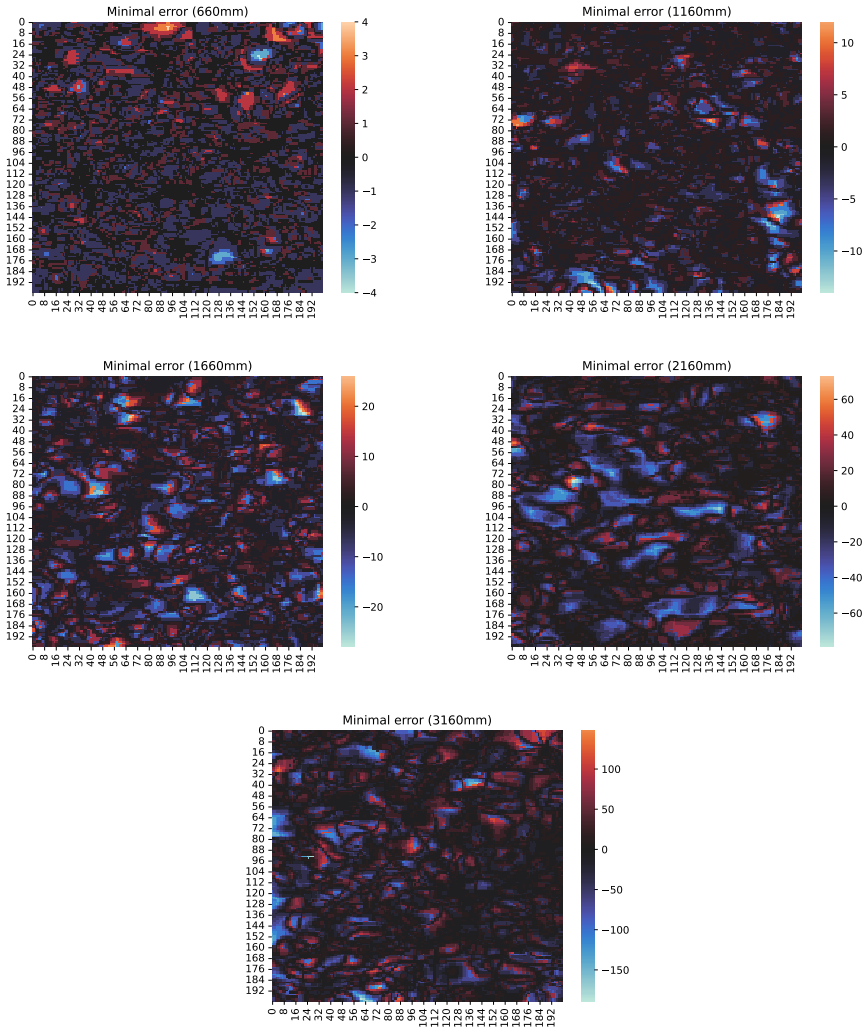

Figure 4: On these images, we can compare the errors by the depth, but on the same cropped square. As expected, the closer images have smaller errors.

One other way, in which we experimented with the camera is to create different environments to test it and compare the images. We compared two images in Figure 5. The first was taken with natural light and with the lights on, while when we did the other one we closed the curtains and switched the lights off. We did not move the camera or anything in the picture between the two pictures. The expected depth was 1160 mm for both pictures. Both images used 10 frames and were corrected with our correction method. It is clear that the light has a heavy impact on the performance of the camera.

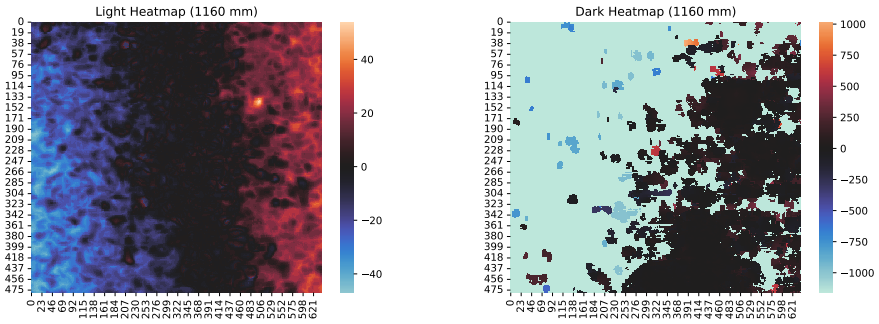

Figure 5: Here we compare two different setups, with the same expected depth. The first picture represents the depth error of the picture captured in full light, while the other was done with as little light as it was possible.

2.3 Statistical Analysis

Then we did a statistical analysis. A statistical analysis is an essential part of camera calibration to assess the quality, accuracy, and reliability of the calibration process. It helps in making informed decisions, improving the calibration framework, and ensuring that the camera can be used effectively in various applications where precise camera geometry is crucial.

Here we present some statistics about the different depth images with the same setup. We present tables with some basic statistics, such as mean or median. We analyze the full picture and the cropped one separately, due to our mistake in the environment setup. We also present some boxplots, histogram and a scatterplot. All values are in mm-s.

Table 1: Full Table

	mean	median	std	min	max
660 mm	0.0	0.0	1.291	-7	17
1160 mm	-1.0	-1.0	0.519	-20	19
1660 mm	-2.0	-2.0	1.226	-45	35
2160 mm	-5.0	-5.0	9.876	-545	90
3160 mm	-22.0	-22.0	188.486	-3160	527

Table 2: Cropped Table

	mean	median	std	min	max
660 mm	0.0	0.0	0.109	-3	4
1160 mm	-1.0	-1.0	0.410	-14	12
1660 mm	-2.0	-2.0	0.848	-28	26
2160 mm	-5.0	-5.0	2.443	-71	57
3160 mm	-6.0	-6.0	5.920	-189	148

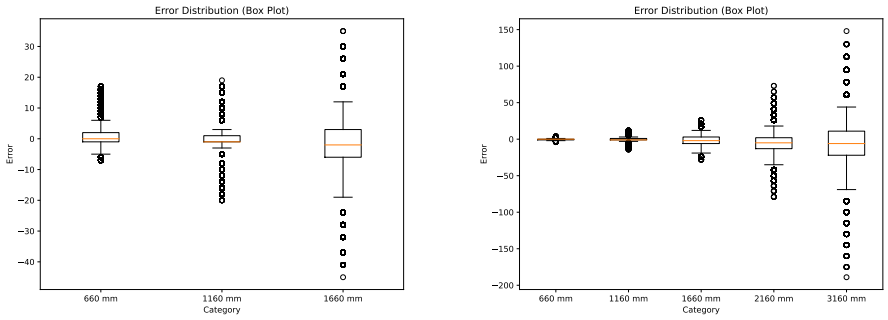

Figure 6: Boxplots of the different expected depth images. We used the corrected errors. The first picture shows the full images of the first three setups, the second one shows the cropped images

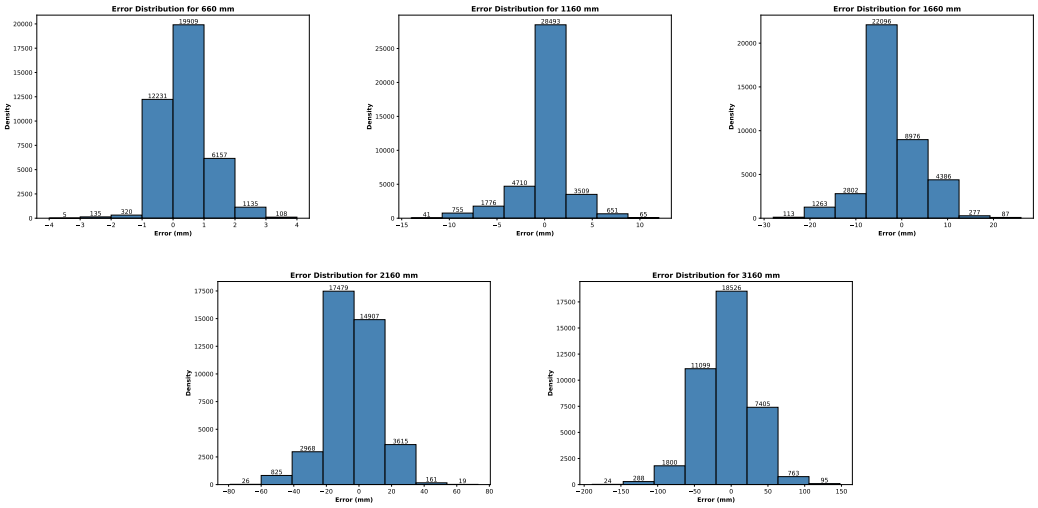

Figure 7: Histogram of the different images using the corrected cropped error matrices.

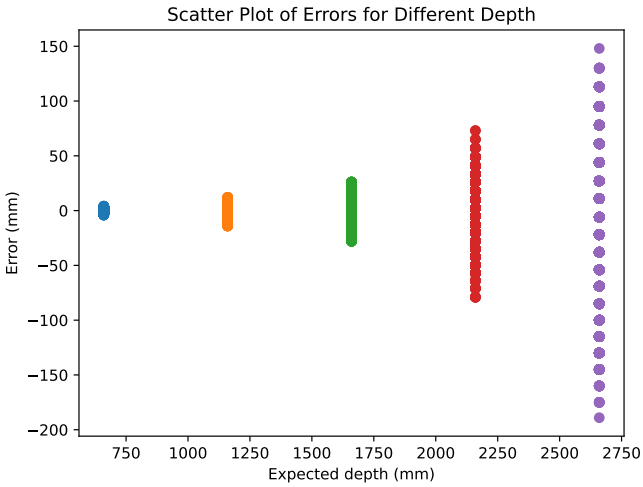

Figure 8: Scatterplot

We understood that the error does not increase linearly with the distance, increases a lot more.

3 Camera calibration

We use a target with some precise dimensions and patterns of shapes and colors to fix the measurements of the camera. The Intel RealSense Dynamic Calibration Print Target with Fixed Width (10 mm) Bars [2] is a crucial tool in the calibration process of RealSense depth cameras, including the camera that we were using, the Intel RealSense Depth Camera D435i. This target is designed to facilitate accurate and consistent calibration, which is vital for achieving precise depth measurements in various applications.
Accurate depth measurement is a fundamental requirement in many applications such as robotics, augmented reality, and 3D scanning. The RealSense depth cameras utilize various technologies, including structured light and time-of-flight, to capture depth information [3]. However, factors like environmental changes, temperature fluctuations, and physical camera adjustments can impact the accuracy of depth measurements over time. The Dynamic Calibration Print Target serves the purpose of maintaining consistent accuracy by allowing the camera's internal calibration algorithms to adjust for these external factors.

To calibrate, we did the following steps [4]:

- Print the Target: Begin by printing the Dynamic Calibration Print Target. This target consists of a series of horizontal bars with a fixed width of 10 mm. Each bar has a distinct pattern that the depth camera can use for calibration. We ensured that the measurements were corrected using a square to confirm after the printing.
- Mounting: Place the printed target in the camera's field of view. It's important to ensure that the target is flat and parallel to the camera's sensor plane. We used a tripod to stabilize the camera and cones to fix the target.
- Calibration Process: There is already a camera calibration software that facilitates the process, the Intel RealSense Viewer. The camera's software performs a calibration process by analyzing the patterns on the printed target. This process involves recognizing the known pattern and adjusting internal calibration parameters based on any deviations. The camera uses these parameters to correct depth measurements in real-world scenes.
- Dynamic Calibration: The term "dynamic" in the target's name emphasizes that this process can be performed at any time to ensure ongoing accuracy. Environmental changes, physical adjustments, and temperature variations can all impact calibration. Regularly recalibrating with the target helps the camera maintain consistent accuracy.

Benefits of calibration:

- Accuracy: Ensures accurate depth measurements by compensating for environmental changes and camera adjustments.

- Versatility: Applicable to a wide range of RealSense™ depth cameras, maintaining consistency across different camera models.
- Ease of Use: The target's fixed bar width simplifies the calibration process, making it accessible to users with varying technical expertise.
- Real-Time Calibration: Dynamic calibration allows for adjustments whenever necessary, ensuring accuracy over time.

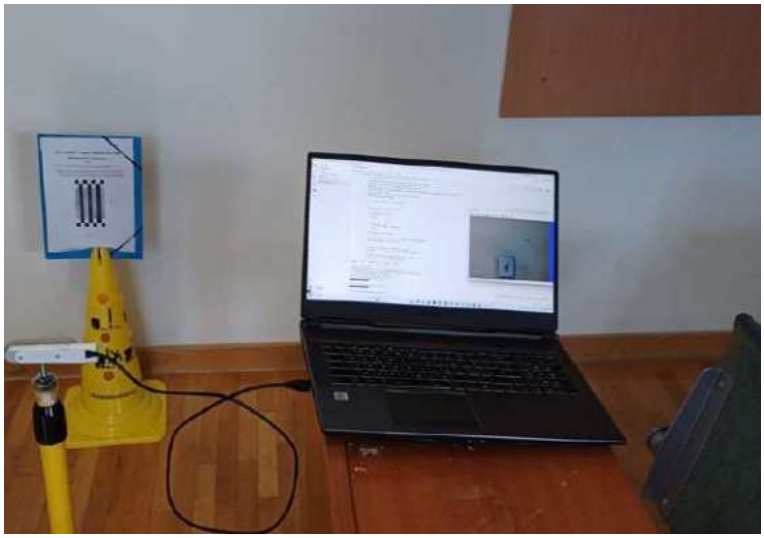

Figure 9: Setup used for the calibration process

3.1 Depth and length detection

How the depth calculation works in the D435i:

- Stereo Vision: The D435i is equipped with two infrared cameras that capture images from slightly different perspectives, simulating the human binocular vision. These images are used to create a stereo pair, where each image corresponds to the view from one camera.
- Infrared Projector: The camera also has an infrared projector that emits a pattern of structured light onto the scene. This pattern consists of a series of dots or lines that help the camera understand the geometry of the objects in the environment.
- Image Analysis: The stereo images captured by the two infrared cameras are compared pixel by pixel. By analyzing the displacement (parallax) of corresponding points in the two images, the camera can determine how far each point is from the camera. This disparity information is used to calculate depth.
- Triangulation: The depth calculation is based on the principle of triangulation. For each pixel, the camera projects a ray from the left camera's center to the corresponding point in the right image. The point where these rays intersect in 3D space represents the depth value of the pixel.

- Calibration and Algorithms: To ensure accurate depth measurements, the camera undergoes a calibration process during manufacturing. This calibration includes parameters like camera intrinsics (lens distortion, focal length) and extrinsics (position and orientation of the cameras relative to each other).
- Depth Map Generation: The calculated depth values are combined into a depth map, where each pixel corresponds to a distance value. This depth map can be used in various applications such as 3D reconstruction, object tracking, and gesture recognition.

The D435i also features an inertial measurement unit (IMU) that provides information about the camera's orientation and acceleration. This data can be fused with depth information to improve the accuracy of depth measurements, especially when the camera is in motion.

To ensure depth measures, we used a measuring tape to verify the depth values between the camera and the target.

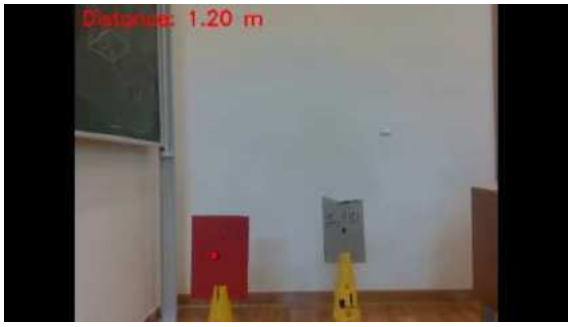

Figure 10: First target.

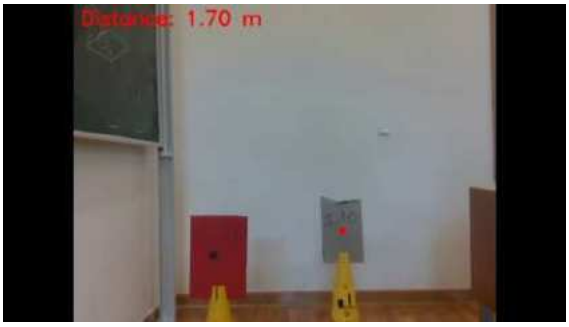

Figure 11: Second target.

Even with calibration, the values were changing constantly, although were not too much. Usually, the difference was just a couple of centimeters. To mitigate this problem we applied two approaches:

- Skipping the first frames: allows the sensor to settle and provide accurate readings. This avoids initial inaccuracies that might affect the quality of the smoothed values, ensuring better accuracy and reliability in the long run.
- Smoothing camera: Instead of relying on a single depth measurement, a series of recent depth values are collected over time. These values are stored in a list. The collected list of depth values is used to calculate the mean, which is the average value of all the depths. This mean value represents a smoothed estimate of the actual depth, as it takes into account the variations observed in the recent measurements. As new depth values are obtained, they are added to the list of recent measurements. To prevent the list from growing indefinitely and consuming excessive memory, a limit is set on the number of values in the list - we used fifty. When the list exceeds this limit, the oldest value is removed to make space for the new one. Smoothing the depth values in this way helps

to mitigate the impact of temporary fluctuations or inaccuracies in individual measurements. It provides a more stable representation of the true depth, which can be crucial for applications where consistent and reliable depth information is needed.

3.2 Callbacks

Callbacks are a mechanism used in programming to specify a function that should be executed in response to a particular event. They are commonly used in graphical user interfaces (GUIs) and interactive applications to define how the program should respond when a specific event, such as a mouse click, occurs.

In this particular case, callbacks are being used in a computer vision application, likely for the following purposes:

Mouse Click Event Handling: The code is setting up a callback function to handle mouse click events within a graphical window named 'dual_win.' When the user clicks the mouse within this window, the registered callback function is triggered.

Displaying Depth Information: Inside the callback function, it appears that the code is designed to retrieve depth information from an aligned depth frame at the coordinates (x, y) where the user clicked the mouse. The depth value is then displayed as "Distance" on the screen.

Delaying Response: There's a delay implemented using time.sleep(1) in the callback function. This delay might be used to give the user a moment to view the depth information on the screen before it changes or to control the rate at which depth information is updated.

Overall, the callbacks in our code [1] are used to respond to mouse clicks by displaying depth information at the clicked location on the screen. This could be part of a larger application, such as one that allows users to interact with depth data from a camera or other depth-sensing device.

3.3 Feature to save data

This feature was implemented to enable data storage from the camera for people whose computers had malfunctioning USB ports, preventing them from directly accessing the camera. Additionally, this data storage capability was utilized for conducting statistical analysis on the captured data.

In this part of the code, a series of actions are performed in response to user input (key presses) within a real-time image display window. Here's an explanation:

In this system, there's a continuous loop running that captures and processes frames,

likely originating from a camera source. This loop keeps track of the number of frames processed, which serves various purposes like naming saved files or monitoring the progress of frame processing.

Each frame goes through a process to ensure that color and depth information are correctly aligned. This alignment is crucial for accurate depth visualization. Once aligned, we obtain both the color and depth images from these frames. The depth image is typically represented with various colors denoting different depth values. These aligned frames, combining color and depth information, are then displayed in a window.

Additionally, a marker (crosshair) is drawn at a specified position on the color image. The depth value at this marker's location is calculated from the aligned depth frame and displayed as text on the color image.

To provide a comprehensive view of the data, the color image with the depth information colormap is arranged side by side in a dual-window display. This setup allows users to simultaneously view the color image and its corresponding depth information.

The application continuously listens for key presses within the image display window. Two keypresses are monitored:

The 'q' key is used to exit the application, effectively terminating the frame processing loop and closing the window. The 's' key is used to trigger specific actions, such as saving images or executing other user-defined tasks. It also serves as an exit command from the processing loop. When the 's' key is pressed, a function called "save_data" is executed. This function handles the task of saving relevant data, which might include information or images, to the computer. This data could be stored for later analysis or other purposes beyond real-time processing.

Overall, this part of the code allows users to view real-time color and depth information, display depth values at specific locations, and trigger actions (such as saving data) by pressing keys.

4 Tracking Keypoints

In the process, we applied contour analysis to the imagery to identify and delineate the shapes of interest. This involved capturing the boundaries that encapsulate the distinct objects in the scene. By analyzing the variations in color and intensity, we defined the perimeters of the objects, outlining their contours. This contour extraction process allowed us to differentiate between various objects, effectively segmenting them from the background. Subsequently, we utilized mathematical techniques to assess the geometric properties of these contours, such as area and centroid. These properties, derived from the contours, played a pivotal role in identifying and characterizing the objects of interest within the visual input. This way we could identify, for example, circles and rectangles.

4.1 Eucledian distance

The Euclidean distance is a measure of the straight-line distance between two points in space. In the context of this code, it's used to calculate the physical distance between two points in the real world, represented by the camera's depth data.

Imagine you have two points in a 3D space. To calculate their Euclidean distance, you first find the difference in each dimension (X, Y, and Z) between these points. These differences are squared, summed, and then the square root is taken. Mathematically, it looks like this for two points A(x1, y1, z1) and B(x2, y2, z2):

$$\text{Distance } = \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$$

In the code, the point1 represents the 3D coordinates of the center of the first rectangle, and point2 represents the center of the second rectangle. The Euclidean distance between these two points is calculated using this formula.

Figure 12: Keypoint detection and distance between shapes.

4.2 Calculation of areas between points

The area of a rectangle is a measure of the region enclosed by its boundaries. In this code, we are calculating the area between two rectangles. The key idea here is that the rectangles are assumed to be flat and parallel to a plane (e.g., a wall), so their areas can be calculated in a simple manner.

For a rectangle, you typically need the length of its two sides, often referred to as the width (W) and height (H). The area (A) of a rectangle is calculated by multiplying these two sides:

$$\mathrm{A} = \mathrm{W} \times \mathrm{H}$$

In the code, the rectangles' areas are calculated based on the lengths of their sides. The "distance_12", "distance_23", and "distance_31" represent the distances between the centers of the rectangles, which are used as the widths and heights of the rectangles for area calculation.

The semi-perimeter (s) is calculated as half of the sum of these distances:

$$s = \frac{\text{distance\_12} + \text{distance\_23} + \text{distance\_31}}{2}$$

Then, the area (A) of the triangle formed by these rectangles is calculated using Heron's formula, which is a general formula for calculating the area of a triangle when you know the lengths of all three sides. It is given by:

$$A = \sqrt{s \cdot (s - \text{ distance\_12}) \cdot (s - \text{ distance\_23}) \cdot (s - \text{ distance\_31})}$$

This area calculation gives you the area enclosed by the three rectangles, which is displayed as "Area between points" in the code's output.

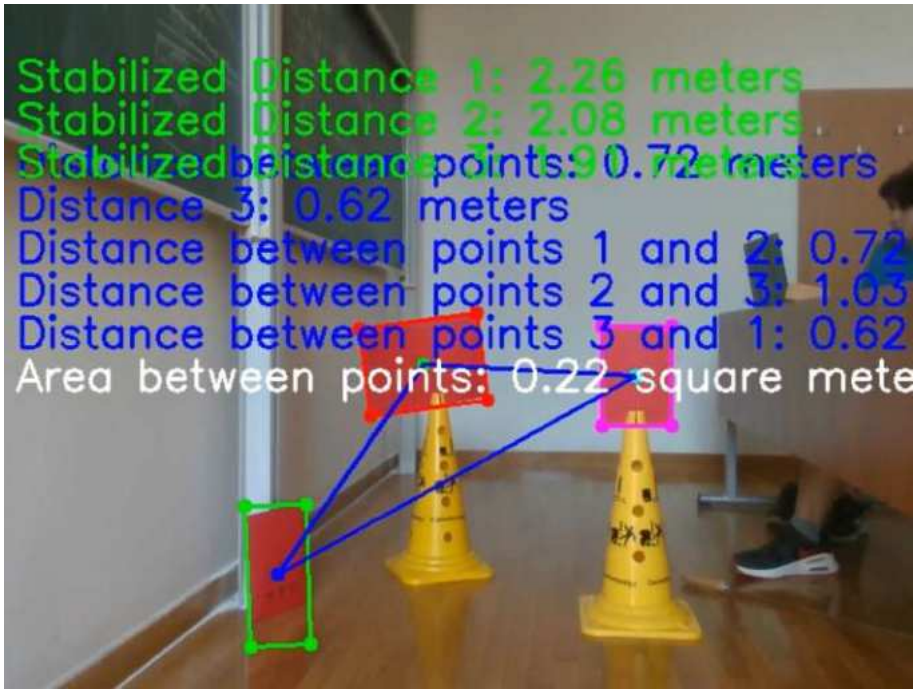

Figure 13: Area of 3-point detection.

4.3 Tic-Tac-Toe

The goal was to create an interactive and visually engaging experience where users could play the game by physically placing objects on a grid. Here, we outline the key components and functionalities of this implementation.

- Game Initialization: The Tic-Tac-Toe board was represented as a 3x3 grid. The visual grid was overlaid onto the camera feed, dividing it into nine equally sized cells (3x3), setting the stage for gameplay.
- Image Processing: The color camera feed was initially converted to grayscale to simplify subsequent processing steps. To reduce noise, a Gaussian blur was applied to the grayscale image. The Canny edge detection algorithm was then utilized to identify edges in the blurred image. This step was crucial in detecting the contours of the shapes placed on the grid.

- Game Logic: The implementation dynamically updated the game state based on the shapes detected in each frame. The coordinates of the detected shapes were used to determine the corresponding cell on the Tic-Tac-Toe grid. If a circle was detected in a cell, it was marked as "O," and if a triangle was detected, it was marked as "X". The update_game_state function was invoked with the detected shape and its center coordinates to update the game board accordingly.

- Win Condition: Following each player move, the implementation checked for a win condition. If any player (X or O) achieved victory, the message "WINNER!" was prominently displayed on the camera feed.

- Display and User Interaction: The camera feed, showcasing the Tic-Tac-Toe board and detected shapes, was continuously displayed using OpenCV's imshow function. The game proceeded until the user chose to exit by pressing the "q" key.

- Cleanup: Upon conclusion of the game, the code ensured the proper shutdown of the camera feed, the release of any output file resources, and the closure of all OpenCV display windows. This implementation provided an interactive and visually immersive experience, where users could engage in a game of Tic-Tac-Toe by physically placing objects on the grid. The computer vision techniques employed for shape detection, combined with the game logic, allowed for real-time gameplay and win detection.

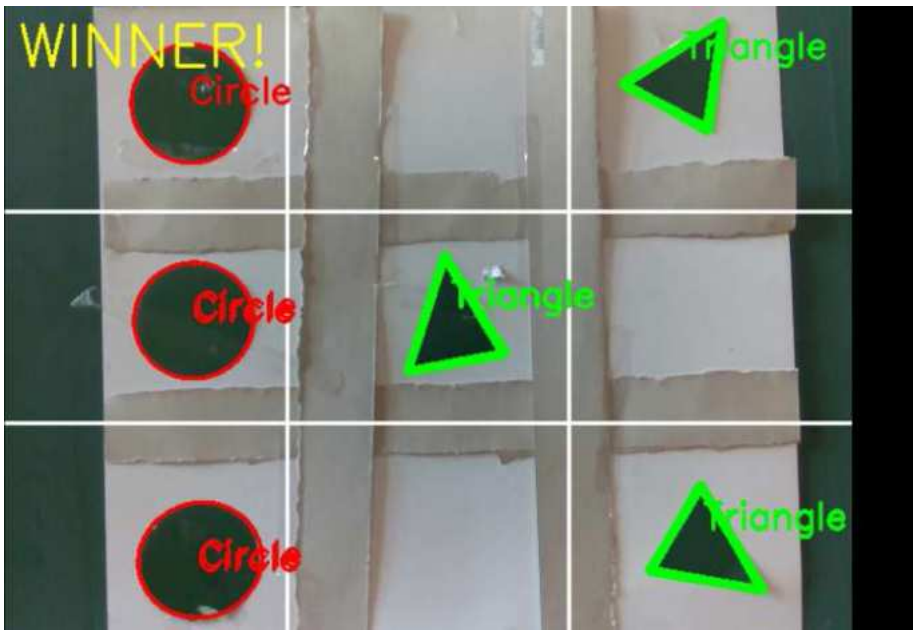

Figure 14: Tic-tac-toe.

5 Object detection

5.1 Fine-tunning

In the process of refining our model, we opted for fine-tuning while keeping all layers frozen. This strategic decision was made to capitalize on the wealth of knowledge embedded within the pre-trained model, YOLOv3-spp [5]. By maintaining the immutability of all layers, our goal was to preserve the model's comprehensive understanding of object features and patterns, which it had acquired during its initial training on extensive datasets like COCO.

The rationale behind freezing all layers stemmed from our recognition that the lower layers of a deep neural network tend to capture general features that transcend specific tasks or datasets. Safeguarding these foundational features was crucial to prevent their dilution during fine-tuning, especially given the limited size of our own dataset.

Although unfreezing layers to fine-tune them might offer certain advantages, such as capturing dataset-specific nuances, we were wary of the risk of overfitting, given the scarcity of our dataset. Thus, to strike a balance between harnessing the pre-trained knowledge of YOLOv3-spp and adapting it to our unique dataset, we opted for the strategy of freezing all layers.

5.2 The models used

The YOLO (You Only Look Once) model is a state-of-the-art object detection system that has become widely popular due to its high speed and accuracy. Unlike traditional two-step models, which first select regions and then classify them, YOLO performs both tasks in a single pass, making it significantly faster. The provided code snippet is a Python implementation using OpenCV's Deep Neural Network (DNN) module to load pre-trained YOLO weights and configuration files for object detection in a video stream.

The code uses YOLOv3, a specific variant of YOLO. It relies on a pre-trained model with weights (yolov3.weights) and a configuration file (yolov3.cfg) that describes the neural network architecture. The network consists of several layers, and the code identifies the output layers that are relevant for object detection.

Each frame is converted to a "blob" using OpenCV's cv2.dnn.blobFromImage function, which performs image normalization, resizing, and channel reordering. The blob serves as the input to the neural network.

The neural network performs forward propagation on the blob to predict object classes and their bounding boxes. The code uses a confidence threshold of 0.5 to filter out low-confidence predictions. Non-Maximum Suppression (NMS) with a threshold of 0.4 is applied to remove overlapping boxes.

Bounding boxes are drawn around the detected objects, and labels are added to indicate the object class and confidence score. The frame is displayed using OpenCV's imshow function.

The model used was the YOLO version 3. The mean average precision is 60.6, for the model used, and the floating point operations per second of 141.45, it's a common metric used to evaluate the accuracy of object detection models, like YOLOv3-spp. mAP quantifies how well the model can detect and locate objects in an image, considering both precision (how many detections are correct) and recall (how many correct objects are detected). It's a single value that represents the overall performance of the model on a given dataset. We also tried Tiny YOLO, which is basically a smaller version of YOLO with a mean average precision of 23.7 with a floating point operations per second of 5.41. Also, the bigger model was able to identify 80 different objects, instead of the other that identified just persons.

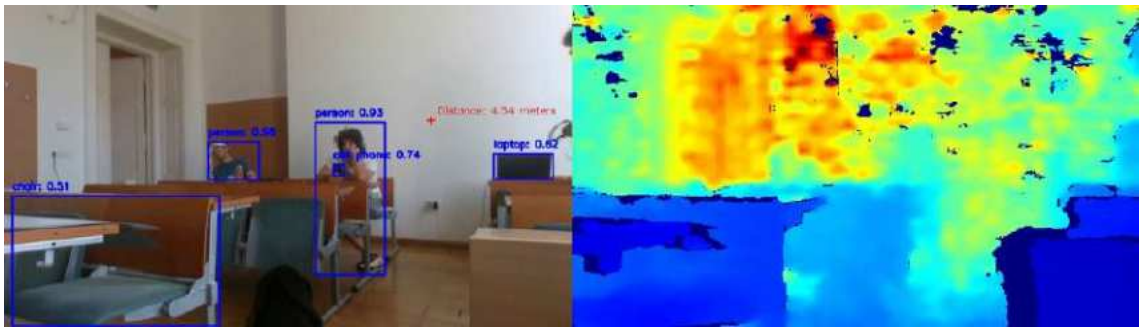

Figure 15: Object detection with version 3 of YOLO model.

Some examples of recognition, are possibly useful for future projects involving the tracking of the camera's movement in the 3D space.

6 Discussion and conclusions

In conclusion, our project ventured into the intricate realm of robotic device tracking, utilizing the Intel RealSense Depth Camera D435i as a cornerstone of our efforts. We set out to develop a comprehensive framework capable of handling real-time projection and correction of both BGR and depth camera data while offering the possibility of expanding functionality through the integrated IMU. Our approach was grounded in computer vision, projective geometry, statistics, and proficient Python programming, and we sought to address the inherent challenges in achieving precise device positioning.

Throughout the course of our project, we achieved several noteworthy milestones. Firstly, we successfully established a robust setup for interfacing with the Intel RealSense Depth Camera D435i, enabling seamless data acquisition. This initial success laid the foundation for subsequent phases of our work.

One of the significant accomplishments was the development of a depth projection system that mitigated sensor measurement errors. By integrating image processing techniques, we were able to enhance the accuracy of depth data projection onto predefined planes. Furthermore, the incorporation of IMU data allowed us to track device movement, providing valuable insights for applications in robotics and beyond.

Our foray into object tracking presented promising results, particularly in calculating Euclidean distances and areas between points. These capabilities are crucial for tasks such as spatial awareness and gesture recognition in robotic systems.

However, it's essential to acknowledge the limitations we encountered during this project. The primary constraint was the inadequacy of the setup in specific scenarios. Notably, individuals without functioning USB ports on their computers faced challenges in utilizing the Intel RealSense Depth Camera D435i, limiting the accessibility of our solution.

Moreover, with extended development time, we could expand the scope of our framework to accommodate a broader range of sensors and devices. This inclusivity would not only increase the versatility of our solution but also make it adaptable to an extensive array of robotic systems. Imagine the possibilities of seamlessly integrating various sensors, from lidar to infrared, to create a comprehensive perception system.

Delving deeper into object detection techniques is another promising avenue. Beyond our current capabilities, we could explore advanced object detection algorithms, including those based on deep learning models like convolutional neural networks (CNNs). These advanced models have the potential to significantly enhance our object-tracking capabilities, enabling us to identify and track more complex objects with intricate shapes and patterns.

Given more time, we could venture into more complex applications of our framework. For instance, we could explore the realm of autonomous robotics, where our enhanced perception and tracking capabilities would be invaluable. This might involve creating

robots capable of navigating dynamic environments, avoiding obstacles, and interacting with humans in a more sophisticated manner.

Extending our project's timeline would also enable us to delve deeper into real-time decision-making algorithms. With these, robots could not only perceive their surroundings but also make intelligent decisions based on the data they gather. This opens doors to applications in fields like autonomous vehicles, industrial automation, and healthcare robotics.

In summary, while our project has achieved significant milestones in the realms of depth projection, sensor integration, and object tracking, there remain exciting opportunities for refinement and expansion. The journey of enhancing precision in robotic device tracking continues, and we look forward to contributing to its evolution.

7 Group work dynamics

Our team approached the project by carefully planning our work. We initiated the project with a thorough analysis phase, during which we delved into the specifications of the Intel RealSense Depth Camera D435i and the pyrealsense2 module. This initial phase aimed to ensure a complete understanding of their capabilities and functionalities, allowing us to establish clear project objectives, including camera calibration and image and object recognition.

To distribute the workload effectively among team members, we strategically assigned specific tasks based on individual skills and expertise. Some team members focused on the critical task of calibrating the camera. They conducted precise measurements and collected data essential for generating accurate statistics. Meanwhile, others were responsible for developing the necessary code for image and object recognition. Their goal was to ensure that the software could efficiently utilize the calibrated data.

Regular team meetings were scheduled to facilitate updates on the status of various project components and to share essential information. Additionally, we provided valuable resources such as video tutorials on how to use Python modules and detailed guides on camera calibration. These materials assisted all team members in becoming proficient with the required tools and processes.

One of the most significant benefits of our teamwork was the diverse skills and perspectives brought by each team member. This diversity enriched our collective understanding and enabled us to address complex challenges more effectively.

However, managing time and meeting deadlines posed a notable challenge. Given our ambitious project goals, meticulous planning and strict adherence to schedules were essential. In this part, Luca managed to cover the statistics part and did some data collection and Leonardo kept the tasks organized, leading the team, covering the majority of the code, collecting data, writing and doing the final presentation, and the final vídeo. The other team members assisted with the code, data collection, and writing the final powerpoint.

In conclusion, our collaborative teamwork played a crucial role in the success of the project, which involved calibrating the Intel RealSense Depth Camera D435i and implementing image and object recognition. Through effective communication and the utilization of diverse skills, we overcame challenges and efficiently achieved our project objectives.

8 Instructor's assessment

In summary, the students successfully fulfilled the conditions of the project. In the field of robotics, you often have to work with sensors, and the first and main part of the project was learning how the Intel Realsense Camera works as a sensor.The three main tasks were to calibrate the camera, project the camera image into the plane, and then track the camera's movement. The first of these tasks was solved, while the students solved other tasks instead of the second and third tasks. The main reason for the change was that, unfortunately, the students did not have the necessary creativity to solve the task. On the other hand, when the problem was already defined, the solution was successfully completed. The substitute tasks are of course interesting and also provided some challenge. Object detection and tracking are also common tasks in the field of robotics, and they provided useful experience for the students.

In my opinion, it was a good experience for the students to gain insight into solving real industrial problems. The students successfully utilized their mathematical knowledge in the above-mentioned cases and were introduced to further learning and development opportunities.

References

[1] Code repository, https://github.com/leobt23/ecmi_modellingweek_sensorcalibration.

[2] Dynamic calibration print target.

[3] Self-calibration for depth cameras.

[4] Intel. Intel realsense depth d435i imu calibration, 2019.

[5] Joseph Redmon. Yolo: Real-time object detection, 2016.