Università degli studi di Trento

DIPARTIMENTO DI INGEGNERIA MECCATRONICA
Laurea in Ingegneria Meccatronica

Design and implementation of the project FOLLOW ME

Course of Robotic Perception and Action

Studente:
Marco Dei Rossi 214649

Anno Accademico 2019-2020

Indice

1 Main aim

The main aim of this project is to build an algorithm that can follow a specific person inside the environment, possibly avoiding obstacles and resolving occlusion problems when possible.
A practical application can be the manufacturing of a trolley able to follow the owner, guided thanks to a set of cameras included in the handle. The motion is given by electric motors connected to at least two wheels (rhombic-like model) powered by a battery inside the trolley.
First, a 2D and a 3D image of the environment are needed. These outputs are provided by a camera that provides these output and an algorithm for people detection. The easiest way is to elaborate the algorithm in 2D and then transpose the solution in the 3D image obtaining also the depth of the image. Lastly, the path planning and the smoothing of the trolley's path have to be implemented.
This project can be used in a lot of fields such as industrial warehouses or nursing homes or supermarkets. The algorithm remains the same except for the connection to the device or the size of the motors.

2 Introduction

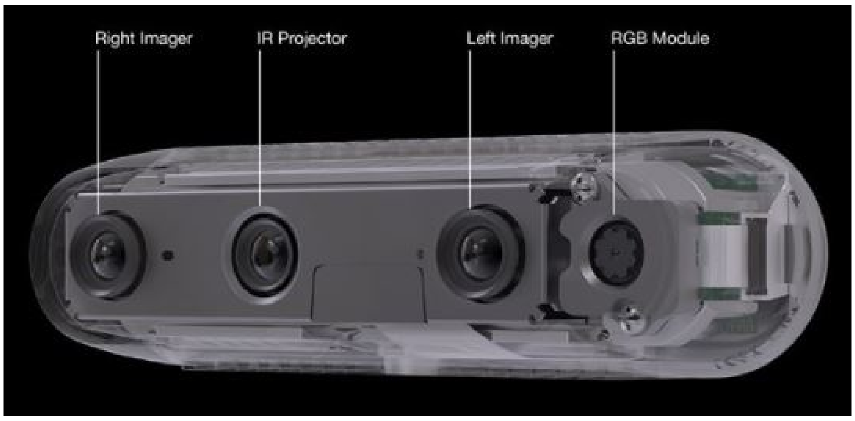

Figura 1:

The starting idea is to use a 3D camera that can scan the environment and detect a large range of object types, such as people or obstacles. The person that is the closest to the device when it is powered on should be the person to be followed.
The bounding box that delimits that person can be reduced to a single point projected on the ground that the trolley, or whatever machine is provided, should follow. Starting from two consecutive points, path planning can be implemented; for example a linear

path planning, or lines connected with an arch of circle, or even better heuristic method such as superposition of sinusoidal or polynomial components.
The main issues to be solved are:

- Choice of the suitable camera (maximization of the field of view, usage of a depth camera,ecc.);
- Find an algorithm that detects people and creates the bounding box around them ;
- Maintain a single target to be followed;
- Try to consider occlusion and obstacle avoidance;
- Find out the right transformation from 2D images to 3D images;
- Find the optimal path planning algorithm.

A suitable camera to be used is the Intel® RealSense^{TM} Depth Camera D435. It is a camera that fuses more different sensors. The chosen model is a depth camera; it can supply, in addition to the simple RGB image, also a 3D image and a point cloud. The D435 is the model with the widest range of vision along with a global shutter on the depth sensor that is ideal for fast-moving applications. Its dimensions are quite small: this allows us to mount it on a trolley handle. The camera is composed of four lenses. Two sensors are needed for stereo vision (right and left imager), moreover an infrared projector -that allows working also in condition of few illuminations- and an RGB module is provided.

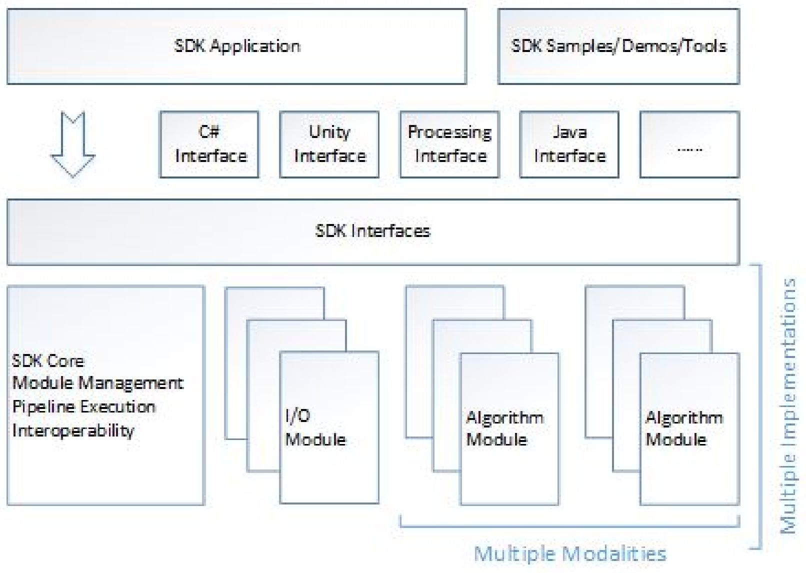

Figura 2: simplified sdk architecture

The stereo vision is something we encounter every day since it reflects the working principle of our sight.

The camera offers a depth image that is useful in mobile robotics or virtual/augmented reality. The accuracy is maintained at a quite long distance (declared 10m from the brochure) related to the dimensions of the camera.
Another big advantage of this camera is the complete integration with Intel's software packages that should allow easy programming.
Inter real sense has developed a quite complete software development kit. It consists of a set of software development tools that can be used to simplify programming an algorithm just recalling predefined functions. These are used to interface to a particular programming language or maybe complex hardware-specific tools.

2.1 SDK Architecture

The SDK library architecture, as illustrated in Figure 3, consists of several layers of components. The essence of the SDK functionalities lays in the input/output (I/O) modules and the algorithm modules. The I/O modules retrieve input from an input device or send output to an output device. The algorithm modules include various pattern detection and recognition algorithms that are critical ingredients of innovative human-computer experience, such as face recognition, gesture recognition, speech recognition, and text to speech.

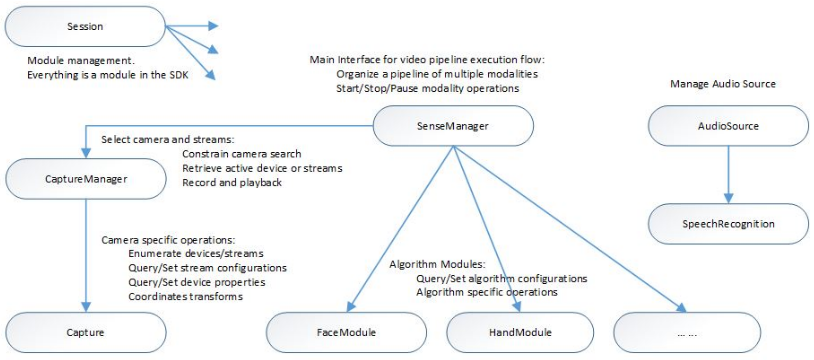
Session
Module management.
Everything is a module in the SDK
Main Interface for video pipeline execution flow:
Organize a pipeline of multiple modalities
Start/Stop/Pause modality operations
Manage Audio Source
SenseManager
AudioSource
Select camera and streams:
Constrain camera search
Retrieve active device or streams
Record and playback
CaptureManager
SpeechRecognition
Camera specific operations:
Enumerate devices/streams
Query/Set stream configurations
Query/Set device properties
Coordinates transforms
Algorithm Modules:
Query/Set algorithm configurations
Algorithm specific operations
Capture
FaceModule
HandModule
... ...

Figura 3: Illustrate the interface hierarchy.

The SDK standardizes the interfaces of the I/O and the algorithm modules so that the applications can access the functionalities without being concerned with the underlying implementations. Furthermore, multiple implementations of SDK interfaces may coexist.

The Session interface manages the following modules: I/O modules, algorithm modules, and any other SDK interface implementations. First, we need to create an instance

of the Session interface (through the CreateInstance function) in the application, then create other module instances from the Session instance.
For predefined usages such as hand tracking and face tracking, we can use the SenseManager interface. This interface organizes a multi-modal pipeline (that contains an I/O device and multiple algorithm modules) and controls the execution of the pipeline such as starting, stopping, pausing, and resuming the pipeline. To create the SenseManager instance, use the CreateInstance function.
Internally, the SenseManager uses the CaptureManager interface to select the I/O device and color/depth/audio streams. Retrieve the CaptureManager instance (from the SenseManager interface) to constrain the device search and/or to set file recording and playback during the pipeline initialization. We can subsequently retrieve the Capture interface for physical camera operations, such as enumerating devices/streams and querying stream configurations and device properties. See I/O Device Operations for more details.
During the pipeline execution, when some samples are ready from the I/O device, we can access the captured samples through the Image interfaces, which abstracts the image buffers. When an algorithm module in the pipeline is ready with some processing results, we need to access the specific interfaces of the algorithm, such as HandModule for hand tracking and FaceModule for face tracking. These interfaces provide algorithm specific functions to set algorithm configurations and algorithm data.
The audio path is a bit different: the application manages the audio source through the AudioSource interface, and specific voice features directly in the module interface, for example, SpeechRecognition.

2.2 Technical specification

Depth technology is based on an active IR stereo. The infrared image is a 2D plot representing the environment using infrared light. The spectrum used is defined as near-infrared, with a wavelength waring from 700nm to about 900nm. Moreover, we use the infrared image to build a stereo image combining it with at least one more similar image.

The field of view of the depth camera is: $87° \pm 3° X 58° \pm 1° X 95° \pm 3°$. The minimum measurable distance is 0.105 m and the maximum one is 10 m. The resolution of the depth image is output at a maximum resolution of 1280 X 720. The maximum frame rate is up to 90 fps. The RGB image instead has a resolution of 1920 X 1080, a frame rate of 30 fps and also a smaller field of view: $69.4° \pm 3° X 42.5° \pm 3° X 77° \pm 3°$.

This disparity of resolution could be a problem in the moment of the change of coordinate of a point from the 2D image to the depth version of it. We then should find a function that easily lets the transformation from one resolution to the other.

3 Bounding box in 2D and 3D

3.1 Pass from generic 2D point to corresponding 3D point

We can connect two different images taken from similar position thanks to the camera matrix.
We represent a point (pixel) in a 2D image as a vector of 2 elements.

$$P = \{x(t), y(t)\}$$

Often it is convenient to use homogeneous coordinates

$$P = \{x(t), y(t)\} = \{s * x(t), s * y(t), s\}$$

s is the scaling factor and usually is fixed at the value 1.

$$P = \{x(t), y(t)\} = \{x(t), y(t), 1\}.$$

In order to simplify calculations, we can omit transposition of axes, but we have to remember it.
We can define the affine transformation as a spatial transformation represented only by multiplications of matrix and homogeneous point. We can so define 3 basic types of transformation:

- Scaling
- Rotation
- Translation

The combination of these three matrices gives all the possible transformations.

We can define every single matrix:
Scaling

Linear transformation applied to all points. It can be uniform or non-uniform.

$$\begin{bmatrix} x' \\ \\ y' \end{bmatrix} = \begin{bmatrix} C_x & 0 \\ \\ 0 & C_y \end{bmatrix} * \begin{bmatrix} x \\ \\ y \end{bmatrix}$$

Rotation

Rotate all points with the same angle of rotational θ

$$\begin{bmatrix} x' \\ \\ y' \end{bmatrix} = \begin{bmatrix} Cos(\theta) & -Sin(\theta) \\ \\ Sin(\theta) & Cos(\theta) \end{bmatrix} * \begin{bmatrix} x \\ \\ y \end{bmatrix} = \begin{bmatrix} Cos(\theta)x - Sin(\theta)y \\ \\ Sin(\theta)x + Cos(\theta)y \end{bmatrix}$$

Translation

A uniform shift of the reference frame and corresponding object points. Equivalent to changing the origin position. We can apply a displacement D on the origin of a vector (x,y)

$$D([x, y]) = \{x + x0, y + y0\}.$$

Using homogeneous coordinates we obtain:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_0 \\ y + y_0 \\ 1 \end{bmatrix}$$

Combination of rotation, scaling, translation with all parameters (rotation angle, scaling factor, translation vector) permits to show the real object in an image I(r,c) on a planar surface W(x,y). These because we need to map the real world in an image and also form the image that came back to the real world. If we want to see how a point changes attitude into the plane, we can compute all the transformations:

$$^{W}P_j = D_{(x,y)} * S_s * R_\theta * \, ^{i}Pj$$

The j point in the world reference frame is a transformation of the j point in the image subjected to a displacement vector D(x,y), a scaling S_s, and a rotation R_θ. In order to solve the system, we need to determine these four parameters, so we need four equations. We need two points that we call "control points" or "matching points" (for each point we need two equations in x and in y) To obtain the generic transformation, we need to:

- Take two points, called control points, which are clearly visible;
- Determine the position vector in the world reference frame and in the image plane;
- Solve the problem for the initial point $(x_i, y_i, 1)$ with all kind of transformations that can arrive in final point $(x_w, y_w, 1)$

Now solving the system first for P1 and then for P2 we manage to find out all the unknowns.

$$\begin{bmatrix} x_W \\ y_W \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} Cos(\theta) & -Sin(\theta) & 0 \\ Sin(\theta) & Cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

We can solve the system moving from the first system to the second by computing the matrix, obtaining the wanted values. More specifically we don't need to know which partial transformation occurs at first, because the combination of these matrices is not sensitive to the order. Our priority assumption was that we were studying a rigid transformation. It is a big simplification indeed it led to the solution of a trivial linear problem:

- Distance between points after transformations is the same as before the transformation;
- Composing of rotation and translation;
- Scaling doesn't properly fit the problem.

Despite what we have just said, we can also have a non-rigid transformation. It is a more difficult problem because all previous properties don't hold anymore. Objects deform and the geometrical relationship between points are not known a priori.

3.1.1 General affine transformation

The "general 2D affine transformation matrix" handles parameters in a 3 by 3 matrix with six parameters easily found with a rigid transformation assumption using only three matching pairs. Errors may occur due to the non-infinite precision of the detection of the pixel in the image. The higher the number of points, the smaller the average of the error. If we take more points, errors can be reduced. In this way, we average the error that we may make taking measures. The most common method to evaluate the error occurrence in the calculation is the least-square value using the subsequent formula:

$$\epsilon(a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}) = \sum_{j=1}^{n}((a_{11}x_j + a_{12}y_j - u_j)^2 + (a_{21}x_j + a_{22}y_j - v_j)^2)$$

To determine the matrix coefficients we only need to take partial derivatives of the error function and set them equal to zero. The results are the following equations:

$$\begin{bmatrix} \sum x_j^2 & \sum x_j y_j & \sum x_j & 0 & 0 & 0 \\ \sum x_j y_j & \sum y_j^2 & \sum y_j & 0 & 0 & 0 \\ \sum x_j & \sum y_j & \sum 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sum x_j^2 & \sum x_j y_j & \sum x_j \\ 0 & 0 & 0 & \sum x_j y_j & \sum y_j^2 & \sum y_j \\ 0 & 0 & 0 & \sum x_j & \sum y_j & \sum 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{bmatrix} = \begin{bmatrix} \sum u_j x_j \\ \sum u_j y_j \\ \sum u_j \\ \sum v_j x_j \\ \sum v_j y_j \\ \sum v_j \end{bmatrix}$$

So we can finally define the 2D *camera matrix* with the best values possible. We can so represent all unknowns within a single set of coefficients.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The least-square method was developed to solve overdetermined equation systems (system of equations that has more equations than unknowns). The goal is to choose the equations in a way that minimizes the sum of the squares of the errors. The error is the distance between the observed value (u and v in our case) and the value obtained by applying the transformation (x and y after transformation). Going in the 3D case one view, in some cases, is not sufficient. Depth of a point couldn't be perceived with only one frame. In this case, we need at least two 2D different projections. The needing of acquiring similar images from different perspectives leads to the construction of:

- 3D mesh reconstruction
- Point cloud acquisition
- Position estimation
- Structure form motion
- Mosaicking

The 3D analysis handles the structure of objects and their real motion in space. It is way more robust than 2D estimation, but it is also more difficult to handle. Calibration is needed and we use both intrinsic and extrinsic parameter.
Intrinsic one could be the focal length, pixel size, distortion of the lens and all other stuff

that belong to the camera system.
Extrinsic parameters are those who relate to the camera system to the translational vector and rotation matrix of the reference frame of the camera in a fixed world reference frame.
The 3D affine transformation is similar to the 2D one, but with more unknowns. With homogeneous coordinates, the pixel position goes from [Px, Py, Pz] to [sPx, sPy, sPz, s].
3D Matrices:

Translation

$$\begin{bmatrix} {}^2P_x \\ {}^2P_y \\ {}^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1P_x \\ {}^1P_y \\ {}^1P_z \\ 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} {}^2P_x \\ {}^2P_y \\ {}^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1P_x \\ {}^1P_y \\ {}^1P_z \\ 1 \end{bmatrix}$$

Rotation

$$\begin{bmatrix} {}^2P_x \\ {}^2P_y \\ {}^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cos\theta & -Sin\theta & 0 \\ 0 & Sin\theta & Cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1P_x \\ {}^1P_y \\ {}^1P_z \\ 1 \end{bmatrix}$$

OR

$$\begin{bmatrix} Cos\theta & 0 & Sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -Sin\theta & 0 & Cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OR

$$\begin{bmatrix} Cos\theta & Sin\theta & 0 & 0 \\ -Sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The generic 3D affine transformation matrix can be expressed through a 4 by 4 matrix with all components of rotation and translation rescaled if needed.

$$\begin{bmatrix} {}^2P_x \\ {}^2P_y \\ {}^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1P_x \\ {}^1P_y \\ {}^1P_z \\ 1 \end{bmatrix}$$

As seen before we need at least 4 points to find the solution, but we have to use a bigger number to minimize the error.

As seen until this point, the camera model consists in the transformation that maps the 3D point (WP) into the image plane (IP), so:

$${}^IP ={}^I_W C *^W P$$

This is very useful to solve our problem because we could find also the position in the world reference frame (depth information) given a planar image.
Using a 3X4 camera matrix we can handle rotation, translation, and scaling. It is nice, but camera coordinates differ from world coordinates. In this case, we need to apply a roto-translation transformation to go from world coordinates (WP) to camera coordinates (CP) with a different matrix from ${}^I_WC$. Furthermore (CP) is about the camera coordinates and not the image coordinates one (FP).

$$\begin{bmatrix} {}^CP_x \\ {}^CP_y \\ {}^CP_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^WP_x \\ {}^WP_y \\ {}^WP_z \\ 1 \end{bmatrix}$$

$${}^{C}P = {}^{C}_{W}TR(\alpha, \beta, \gamma, t_x, t_y, t_z){}^{W}P$$

In the end, if we want to pass from the world reference frame to image passing through the camera we simply have to combine the two matrices below.

$${}^{F}P = \prod_{C}^{F}(f){}^{C}P$$

$${}^{F}P = \prod_{C}^{F}(f){}^{C}_{W}TR(\alpha, \beta, \gamma, t_x, t_y, t_z){}^{W}P$$

$$\begin{bmatrix} s^{F}P_r \\ s^{F}P_c \\ s \end{bmatrix} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & 1 \end{bmatrix} \begin{bmatrix} {}^{W}P_x \\ {}^{W}P_y \\ {}^{W}P_z \\ 1 \end{bmatrix}$$

To pass from mm to pixel it is matter of a scaling factor that is related to the real size of the pixel. We have to assume dx the horizontal size and dy the vertical size, but we also have to consider that the world reference frame origin usually is bottom left; instead in the image is top left.

$${}^{I}P = {}^{I}_{F}S{}^{F}P$$

$${}^{I}_{F}S = \begin{bmatrix} 0 & -1/d_y & 0 \\ 1/d_x & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In this way the final expression is:

$$[p_r, p_c]^T = {}^{I}P = {}^{I}_{F}S\prod_{C}^{F}(f){}^{C}_{W}TR(\alpha, \beta, \gamma, t_x, t_y, t_z){}^{W}P$$

$$\begin{bmatrix} s^{I}P_r \\ s^{I}P_c \\ s \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & 1 \end{bmatrix} \begin{bmatrix} {}^{W}P_x \\ {}^{W}P_y \\ {}^{W}P_z \\ 1 \end{bmatrix}$$

To find all the 11 coefficients of the camera matrix we have to compute the calibration. It starts with the usage of real 3D measurements of the environment. The process is iterative and consists of 3 steps:

- Use of an object with a well-known size;
- A set of points in the image/ world are taken, it least only 6 points (pairs)(overdetermined system 11 unknowns and 12+ equations), but a higher number are preferable;
- Form the camera matrix, given a 3D point and a 2D projection the calibration process is:

$$\begin{bmatrix} x_j & y_j & z_j & 1 & 0 & 0 & 0 & 0 & -x_ju_j & -y_ju_j & -z_ju_j \\ 0 & 0 & 0 & 0 & x_j & y_j & z_j & 1 & -x_jv_j & -y_jv_j & -z_jv_j \end{bmatrix} c^T = \begin{bmatrix} u_j \\ v_j \end{bmatrix}$$

  with c defined as the vector of coefficients of matrix ${}^I_W C$

$$c = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{21} & c_{22} & c_{23} & c_{24} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

To clarify this concept, we have two cameras, each one throwing a line from the projected point to the real 3D point. The point can be found at the intersection of the two lines. In reality, probably the two lines intersect in a point that is not the real one. If we know the real point, we can find out the distance of the two lines from that point and find the error.
It could happen that if we take the four equations all together, they would be inconsistent, because the four solutions we can find out are not equal. This is caused by approximation in image points, approximation in the camera model, and in practice, the ray does not intersect where they should.

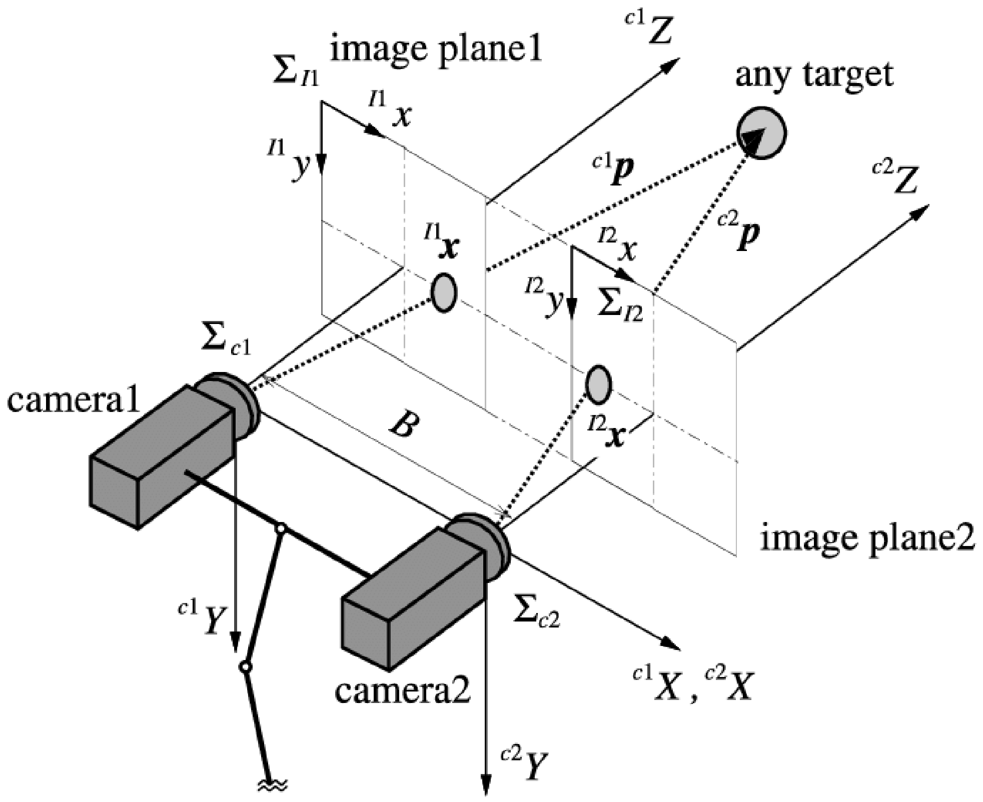

Figura 4: Model of the stereo image.

The easiest scenario that helps us to understand the real world thanks to two cameras is the binocular stereo. It is the same that allows human sight. Two images grab a scene from different positions. This method identifies two main problems:

- Compute correspondences (we need to find points that match across the two views);
- Reconstruction (that we can get the 3D coordinates).

Using two co-planar cameras is the easiest solution that fits the binocular stereo model. We have 2 cameras with the same orientation, parallel and aligned. Both the cameras have a "range of vision" where they can properly see and is defined as a cone. Only in the points reached/seen by both cameras, the depth information can be extracted.

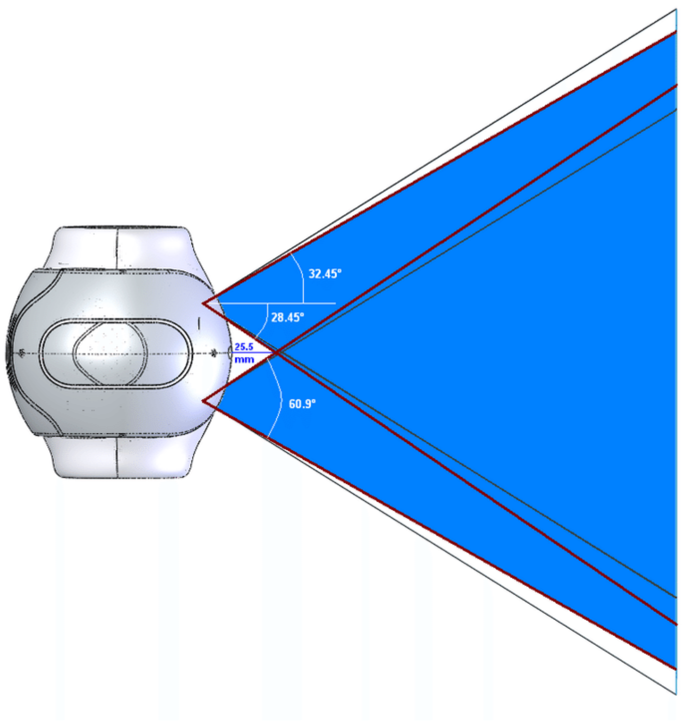

Figura 5: Field of view of the stereo image model.

To discover the depth information, we should compute correspondences in the two different images. Both virtual points are the projection of the same real object (plus or minus small evaluation errors). The image coupling is possible because the two images do not differ much between each other. False correspondence may be present, due to uniform colors or other similarities. We do have to introduce other constraints such as epipolar constraint. This consists of correspondences that can be met along a horizontal line called the epipolar line. Once the points in the 2 images are matched, reconstruction is possible. Before computing correspondences, it is necessary to calibrate the cameras to determine extrinsic and intrinsic parameters. The stereo vision is based on the so-called epipolar geometry. The estimation on the depth of a point is based on a technique that uses the horizontal disparity of the projections. The problem consists in computing both the correspondences of intensity points (correlation) and the correspondences of features (edges and contours).

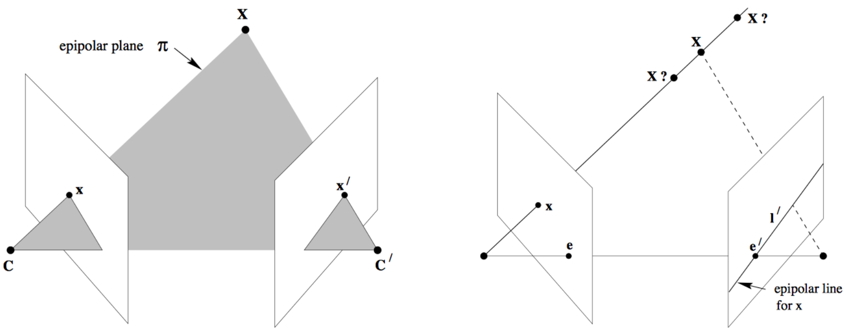

Figura 6: Epipolar structure with correspondences of features.

3.2 Projection and de-projection in realsense

In this part of the document, we explain the same concept before, applied directly to our sensor using his libraries. The projection of a point from the 2D image to the corresponding point in the 3D image is a nontrivial task. The main problem is that the two images we are going to compare have different fields of view, different resolution, and a different relative reference frame.
So the only way to accomplish this task is to create a function or a map that relates each pixel in the 2D image to the corresponding same point in the 3D image. We can do the same from the 3D to the 2D to check the process.
The libraries of or sensor luckily provide us the functions that achieve or goal.
To better comprehend this process, avoiding the mere application of the formulas, we are now going to present how they work and demonstrate how we found them out.
First, we start from the pixel coordinates in the 2D image. Each stream of images provided by our SDK is associated with a separated 2D coordinate space, specified in pixels, with the coordinate [0,0] referring to the center of the top-left pixel in the image, and [w-1,h-1] referring to the center of the bottom right pixel in an image containing exactly w columns and h rows. That is, from the perspective of the camera, the x-axis points to the right and the y-axis points down. Coordinates within this space are referred as "pixel coordinates", and are used to index into images to find the content of particular pixels.
The point coordinates instead are provided by this SDK in a 3D vector and are also associated with a separate 3D coordinate space, specified in meters, with the coordinate [0,0,0] referring to the center of the physical image. Within this space, the positive x-axis points to the right, the positive y-axis points down, and the positive z-axis points forward (positive enters in the photo). Coordinates within this space are referred as "points" and are functional to describe locations within 3D space that might be visible within a particular image.
Basically, we are in the worst possible situation, or rather all three parameters that describe the same point in the 2D image, 3D image, and 3D word are all different because

refereed to a different reference frames. To connect all these coordinates we should know the position of the camera and his intrinsic parameters. The position (pose and orientation) should be fixed in the word reference frame and can be easily measured. The intrinsic camera parameters are also fixed for our camera and can also be easily found. The relationship between a stream's 2D and 3D coordinate systems is described by its intrinsic camera parameters, contained in the *rs2_intrinsics* struct. Each model of RealSense device is somewhat different, and the *rs2_intrinsics* struct must be capable of describing the images produced by all of them. The basic set of assumptions is described below:

- Images may be of arbitrary size.

  The *width* and *height* fields describe the number of rows and columns in the image, respectively.

- The field of view of an image may vary.

  The *fx* and *fy* fields describe the focal length of the image, as a multiple of pixel width and height.

- The pixels of an image are not necessarily square.

  The *fx* and *fy* fields are allowed to be different (though they are commonly close).

- The center of projection is not necessarily the center of the image.

  The *ppx* and *ppy* fields describe the pixel coordinates of the principal point (center of projection).

- The image may contain distortion.

  The *model* field describes which of several supported distortion models was used to calibrate the image, and the *coeffs* field provides an array of up to five coefficients describing the distortion model.

Knowing the intrinsic camera parameters of images allows us to carry out the two fundamental mapping operations: Projection and Deprojection. As explained before projection takes a point from a stream's 3D coordinate space, and maps it to a 2D pixel location on that stream's images. It is provided by the header-only function *rs2_project_point_to_pixel(...)*. The deprojection instead takes a 2D pixel location on a stream's images, as well as a depth, specified in meters, and maps it to a 3D point location within the stream's associated 3D coordinate space. It is provided by the header-only function *rs2_deproject_pixel_to_point(...)*. Intrinsic parameters can be retrieved from any *rs2::video_stream_profile* object via a call to *get_intrinsics()*.

Distortion Models

Based on the design of each model of RealSense device, the different streams may be exposed via different distortion models.

- None

An image has no distortion, as though produced by an idealized pinhole camera. This is typically the result of some hardware or software algorithm undistorting an image produced by a physical imager, but may simply indicate that the image was derived from some other image or images which were already undistorted. Images with no distortion have closed-form formulas for both projection and deprojection, and can be used with both *rs2_project_point_to_pixel(...)* and *rs2_deproject_pixel_to_point(...)*.

- Modified Brown-Conrady Distortion

An image is distorted and has been calibrated according to a variation of the Brown-Conrady Distortion model. This model provides a closed-form formula to map from undistorted points to distorted points, while mapping in the other direction requires iteration or lookup tables. Therefore, images with Modified Brown-Conrady Distortion are being undistorted when calling *rs2_project_point_to_pixel(...)*. This model is used by the RealSense D415's color image stream.

- Inverse Brown-Conrady Distortion

An image is distorted and has been calibrated according to the inverse of the Brown-Conrady Distortion model. This model provides a closed-form formula to map from distorted points to undistorted points, while mapping in the other direction requires iteration or lookup tables. Therefore, images with Inverse Brown-Conrady Distortion are being undistorted when calling *rs2_deproject_pixel_to_point(...)*. This model is used by the RealSense SR300's depth and infrared image streams.

Extrinsic Camera Parameters

The 3D coordinate systems of each stream may, in general, be distinct. For instance, it is common for depth to be generated from one or more infrared imagers, while the color stream is provided by a separate color imager. The relationship between the separate 3D coordinate systems of separate streams is described by their extrinsic parameters, contained in the *rs2_extrinsics* struct. The basic set of assumptions is described below: Imagers may be in separate locations but are rigidly mounted on the same physical device The translation field contains the 3D translation between the imager's physical positions, specified in meters Imagers may be oriented differently, but are rigidly mounted on the same physical device The rotation field contains a 3x3 orthonormal rotation matrix between the imager's physical orientations All 3D coordinate systems are specified in meters There is no need for any sort of scaling in the transformation between two coordinate systems All coordinate systems are right-handed and have an orthogonal basis; There is no need for any sort of mirroring/skewing in the transformation between two coordinate systems Knowing the extrinsic parameters between two streams allows you to transform points from one coordinate space to another, which can be done by calling *rs2_ transform_point_ to_ point(...)*. This operation is defined as a standard affine transformation using a 3x3 rotation matrix and a 3-component translation vector.

Extrinsic parameters can be retrieved via a call to *rs2_ get_ extrinsics(...)* between any two streams which are supported by the device, or using a *rs2::stream_ profile* object via *get_ extrinsics_ to(...)*. One does not need to enable any streams beforehand, the device extrinsic are assumed to be independent of the content of the streams' images and constant for a given device for the lifetime of the program.

Depth Image Formats

As mentioned above, mapping from 2D pixel coordinates to 3D point coordinates via the *rs2_ intrinsics* structure and the *rs2_ deproject_pixel_ to_point(...)* function requires knowledge of the depth of that pixel in meters. Certain pixel formats, exposed by this SDK, contain per-pixel depth information, and can be immediately used with this function. Other images do not contain per-pixel depth information and thus would typically be projected into instead of deprojected from. *RS2_ FORMAT_ Z16* (Under *rs2_format*) Depth is stored as one unsigned 16-bit integer per pixel, mapped linearly to depth in camera-specific units. The distance, in meters, corresponding to one integer increment in depth values can be queried via *rs2_ get_ depth_ scale(...)* or using a *rs2::depth_ sensor* via *get_ depth_ scale()*. The following lines shows how to retrieve the depth of a pixel in meters:

Using C++ API:
rs2::depth_frame dpt_frame = frame.as<rs2::depth_frame>();
float pixel_ distance_ in_ meters = dpt_frame.get_ distance(x,y);

If a device fails to determine the depth of a given image pixel, a value of zero will be stored in the depth image. This is a reasonable sentinel for "no depth" because all pixels with a depth of zero would correspond to the same physical location, the location of the imager itself. The default scale of an SR300 device is 1/32th of a millimeter, allowing for a maximum expressive range of two meters. However, the scale is encoded into the camera's calibration information, potentially allowing for long-range models to use a different scaling factor.

The default scale of a D400 device is one millimeter, allowing for a maximum 'expressive range' of 65 meters. The sensing range is 10 m. The depth scale can be modified by calling *rs2_set_option(...)* with *RS2_OPTION_DEPTH_UNITS*, which specifies the number of meters per one increment of depth. 0.001 would indicate the millimeter scale, while 0.01 would indicate the centimeter scale.

4 Path planning

Path planning is the theory that lets us built the line that our trolley should follow in the environment. Thanks to the RealSense camera and to our code we manage to find out the bounding box over a person in a 2D image.

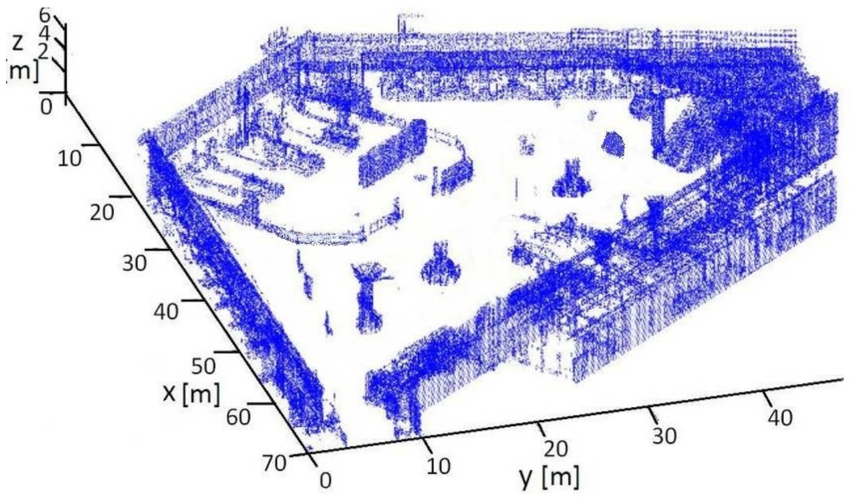

Figura 7: Environment of Helsinki airport. The world reference frame of the image is different from the one of the camera

```
...
float centrop[2];
centrop[0]=center.x;
centrop[1]=center.y;
float centrov[3];
float centrov1[3];
float centerdepth = depth_frame.get_distance(centrop[0], centrop[1]);

auto depth_stream = config.get_stream(RS2_STREAM_DEPTH)
.as<rs2::video_stream_profile>();
auto resolution = std::make_pair(depth_stream.width(), depth_stream.height());
auto i = depth_stream.get_intrinsics();
auto principal_point = std::make_pair(i.ppx, i.ppy);
auto focal_length = std::make_pair(i.fx, i.fy);
//auto matrix_parameters = std::make_tuple(i.coeffs[5]);
float c1 = 0.090554739064;
float c2 = -0.263085401898;
float c3 = 0;
float c4 = 0;
float c5 = 0;
auto matrix_parameters = std::make_tuple(c1,c2,c3,c4,c5);
rs2_distortion model = i.model;
...
float x = (centrop[0]-principal_point.first)/focal_length.first;
float y = (centrop[1]-principal_point.second)/focal_length.second;
float r2 = x*x + y*y;
float f = 1 + std::get<0>(matrix_parameters)*r2 + std::get<1>(matrix_parameters)*r2*r2 +
std::get<4>(matrix_parameters)*r2*r2*r2;
float ux = x*f + 2*std::get<2>(matrix_parameters)*x*y +
std::get<3>(matrix_parameters)*(r2 + 2*x*x);
float uy = y*f + 2*std::get<3>(matrix_parameters)*x*y +
std::get<2>(matrix_parameters)*(r2 + 2*y*y);
x = ux;
y = uy;
centrov[0] = centerdepth * x;
centrov[1] = centerdepth * y;
centrov[2] = centerdepth;
centrov1[0] = m[0] * x;        //NON NECESSARY POINT
centrov1[1] = m[0] * y;        //IT IS ONLY NEEDED TO CHECK DIFFERENCE FROM THE OTHER POINT
centrov1[2] = m[0];
...
```

This final part of the code let us transpose the 2D image coordinates into 3D world coordinates. We calculated both the 3D coordinates of the central point and the mean 3D coordinates inside the bounding box. They are respectively defined as *centrov* and *centrov1*. For path planning, we need a point on the ground so we don't care the y coordinate. The most important thing is that we are persistent in the reference frame we use. We decided to use as ground plane, the sensors one so that we don't need to make unnecessary changes of reference frames: one from the camera to the actual ground and another one to ground back to the camera. In the end, we saved all points to be interpolated in a single ordered struct *save* composed by i elements of three parts x,y,z. The camera can produce 30 frames per second.

We need to have some control over the relative position between consecutive points. Otherwise, our program could decide to follow different people that occur in the field of view of our camera. So we decided to apply two different control on the position that should be saved. The first is control over the distance between the previous position saved. We can assume that a person in a second can walk for $1.6m$. We can have a closer detection, so we control the position in the actual frame with the position 15 "steps" before, so half a second earlier.
In this way we can assume that the position at time i distance $0.8m$ at maximum from the position at *i-15*.
As said before this control can be made both on x and on z direction. On y we don't need it because it is fixed on value 0.

```
if(fabs(save[i][0]) < fabs(save[i-15][0]+ dist) ||
   fabs(save[i][0]) > fabs(save[i-15][0]- dist) )
```

```
        {
                if(fabs(save[i][2]) < fabs((save[i-15][2]+ dist)) ||
                   fabs(save[i][2]) > fabs(save[i-15][2]- dist) )
                {
                        save[i][0] = centrov1[0];
                        save[i][1] = 0;
                        save[i][2] = centrov1[2];
                }
        }
```

Another, more specific control, can be made on the histogram of the bounding box. We know that each image owns his histogram that let a unique representation at each instant of time. The histogram can vary dependently on the pose and the attitude of the person inside the image, on the light, and on many other factors that we can't control. Despite all the unknowns of this problem, we can easily assume that the histogram doesn't vary too much in a short amount of time. The histogram can be seen as a chart of the distribution of every single color inside the image. It is more specifically defined as a graphical representation of the tonal distribution in a digital image. It plots the number of pixels for each tonal value.
Both methods are not effective at 100% and for this reason we need this double-check. In this way, we characterize the person both with his position and his histogram. Both controls are updated in time, because all unknowns change little in a short time, but can vary a lot if time grow.

If we want a stricter and more efficient recognition of our target we can split up the histogram in many parts. For example, we can divide the bounding box into three parts:

- head
- chest
- legs

These divisions can be made on a known ratio of the human body. We can assume that the head is in the first 1/8 part of the total height from the top of the bounding box. In a similar way, we can also say that chest and leg have, more or less, similar dimension (3.5/8 each).
This type of constraint can be easily implemented with similar code to the one above.

```
Mat src, dst;// creiamo le matrici necessarie
src = color_mat;
// carichiamo l'immagine da confrontare come object vedi def sopra come rettangolo
//bounding box così però mi carica tutto il frame

Mat src1(src, Rect(xLeftBottom, yLeftBottom, xRightTop, yRightTop) );

vector<Mat> bgr_planes;
```

```
split( src1, bgr_planes );

int histSize = 256;
float range[] = { 0, 256 } ;
const float* histRange = { range };

bool uniform = true; bool accumulate = false;

Mat b_hist, g_hist, r_hist;

// Compute the histograms:
calcHist( &bgr_planes[0],1,0,Mat(),b_hist,1, &histSize, &histRange, uniform, accumulate );
calcHist( &bgr_planes[1],1,0,Mat(),g_hist,1, &histSize, &histRange, uniform, accumulate );
calcHist( &bgr_planes[2],1,0,Mat(),r_hist,1, &histSize, &histRange, uniform, accumulate );


// Draw the histograms for B, G and R
int hist_w = 512; int hist_h = 400;
int bin_w = cvRound( (double) hist_w/histSize );

Mat histImage( hist_h, hist_w, CV_8UC3, Scalar( 0,0,0) );

// Normalize the result to [ 0, histImage.rows ]
normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );

histogram1[i]= b_hist;
histogram2[i]= g_hist;
histogram3[i]= r_hist;

// Draw for each channel
for( int hs = 1; hs < histSize; hs++ )
{
line( histImage, Point( bin_w*(hs-1), hist_h - cvRound(b_hist.at<float>(hs-1)) ) ,
Point( bin_w*(hs), hist_h - cvRound(b_hist.at<float>(hs)) ),
Scalar( 255, 0, 0), 2, 8, 0  );
line( histImage, Point( bin_w*(hs-1), hist_h - cvRound(g_hist.at<float>(hs-1)) ) ,
Point( bin_w*(hs), hist_h - cvRound(g_hist.at<float>(hs)) ),
Scalar( 0, 255, 0), 2, 8, 0  );
line( histImage, Point( bin_w*(hs-1), hist_h - cvRound(r_hist.at<float>(hs-1)) ) ,
Point( bin_w*(hs), hist_h - cvRound(r_hist.at<float>(hs)) ),
Scalar( 0, 0, 255), 2, 8, 0  );
}
/// Display
namedWindow("calcHist␣Demo", CV_WINDOW_AUTOSIZE );
imshow("calcHist␣Demo", histImage );
// tutta la parte dalla definizione di histogram 1,2,3 serve solo per l'immagine


float t = 0.1;
//threshold da dare al cambiamento dell'istogramma bisogna capire
//come visualizzare come sigolo valore l'istogramma
// per questo valore ci vorrebbero dei try and attempt, può essere
//tranquillamente più grosso ma è un rischio che ritengo inutile
// praticamente vuol dire che tra 5 histogram consecutivi c'è
//una corrispondenza di almeno il 10 % su tutti i colori.
```

```

if(i>15)
{
if(abs(compareHist( histogram1[i], histogram1[i-5], 0 ))>t)
// ovviamente se l'istogramma è dato da tre calori bisogna
fare tre if su ogni valore dell'istogramma
{
if(abs(compareHist(␣histogram2[i],␣histogram2[i-5],␣0␣))>t)
//␣ovviamente␣se␣l'istogramma è dato da tre calori bisogna
//fare tre if su ogni valore dell'istogramma
{
if(abs(compareHist( histogram3[i], histogram3[i-5], 0 ))>t)
// ovviamente se l'istogramma è dato da tre calori bisogna
//fare tre if su ogni valore dell'istogramma
{
if(fabs(save[i][0]) < fabs(save[i-15][0]+ dist) ||
 fabs(save[i][0]) > fabs(save[i-15][0]- dist) )
{
if(fabs(save[i][2]) < fabs((save[i-15][2]+ dist)) ||
 fabs(save[i][2]) > fabs(save[i-15][2]- dist) )
{
save[i][0] = centrov1[0];
save[i][1] = 0;
save[i][2] = centrov1[2];
}
}
}
}
}
}
```

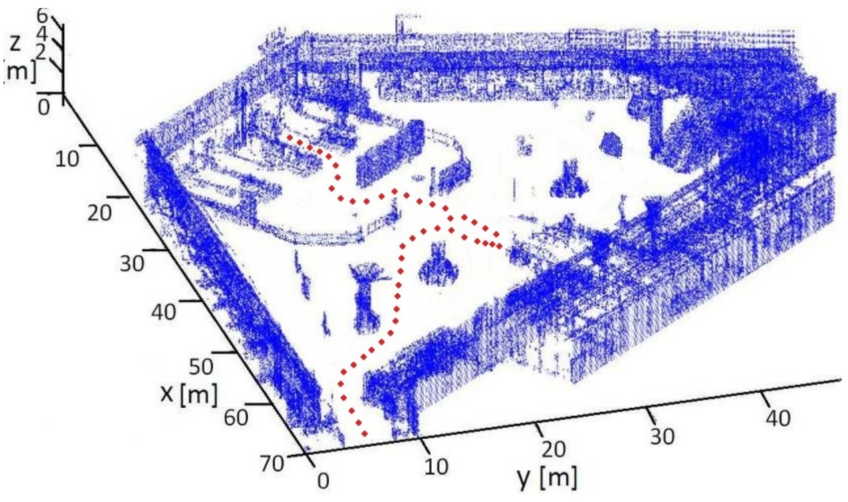

Figura 8: Points found from the camera on the moving person. The wold reference frame of the image is different from the one of the camera.

We can finally work on a series of points that characterize the person that we want to follow. One of the fundamental problems of autonomous navigation is, therefore, the planning of motion, on two levels: a high level of global strategy which is about achieving

the final goal and, functional to it, a lower level of local planning. In this paper, we are only facing a higher level to study which is the most feasible path. We don't look at obstacle avoidance because we assume that our trolley is following the person with a close distance. It should be calibrated avoiding both to hit the person and don't lose the image of the "driver". In this way, it seems difficult to us to occur in obstacles because we are more or less following the same path of the person that autonomously avoid them. It is a feature that can be easily implemented, but we believe that more code lines can only make the code heavier and more difficult to implement in real time. We preferred to have more samples than other control. We can instead think to apply a feedback control that is more useful to avoid the loss of the reference frame.
We should think to work in an unstructured environment. It means that there is not enough information to comprehensively plan the movements. The robot must be able to locate itself in real-time concerning the environment and plan the motion in progress to complete the task.
In this research, we have completely left out the part that treats the argument of the kinematical model of our autonomous vehicle. It is a part more related to the hardware of the motor applied to the wheels. Although we can suggest some type of kinematic models such us:

- two-wheel drive with differential guide model
- two-wheel drive with car-like model
- rhombic like model
- four steering and driving wheels

The space of a mobile robot configuration is defined by the minimum number of generalized coordinates that allow locating the entire system in its environment. We can so define the admissible trajectory, given the initial and final conditions of the configuration, as a solution of the system of differential equations that constitute the kinematic model of the robot.
A further definition of trajectory is the route to which the temporal law with which the path is followed by the robot is associated. This meaning is used to distinguish between three classes of problems based on the task assigned to the robot:

- Point-to-point motion
- Route tracking (path following)
- Trajectory tracking (trajectory tracking)

In point-to-point motion, the robot must reach a final position starting from a given initial position. In general, it must reach a final configuration given an initial configuration in the space of generalized coordinates. The problem, from the control point of view, is to stabilize the robot in a point of equilibrium in the configuration variables space. Any feedback control uses as an error to be minimized the difference between the current configuration and the desired one.
In path tracking the robot must reach and follow a path in Cartesian space starting from a given initial configuration inside or outside the path. The control algorithm is based on the geometrical description of the Cartesian trajectory, usually parameterized according to the curvilinear coordinate s. The control temporal law is not specified since the main goal is to approach as much as possible the vehicle to the trajectory reducing the distance from it. Therefore, the evolution of the parameter s over time can be chosen arbitrarily and the two inputs can be deducted with respect to time without changing the path. A choice may be to maintain the speed input of the traction constant or to vary it according to any temporal law, and to carry out the tracking control by acting only on the angle of steering.
In this case, the robot has to reach and follow a trajectory in Cartesian space starting from a given initial configuration in or out of the path, or a path with an imposed temporal law. One may think that the robot has to chase and reach a virtual robot that moves along the desired path with variable speed according to the desired law. From the control point of view, the objective is to minimize the Cartesian error e(t) between the actual position of the robot and the one provided for each moment of time t.
The problem, upstream, is to plan a route compatible with both non-holonomic constraints and with the presence of obstacles potentially interposed between the starting and landing point. This problem which applies to autonomous vehicles is called of steering.
If we are in a structured environment the route and motion planning can be calculated beforehand. Many methods can connect all these points.
The most basic one forms a path of segments that interpolate all points perfectly. This path is formally perfect because fit all point perfectly and minimize the distance between two consecutive points, although it will not be the one with the shortest path. The simplest method does never fit to the solution. In fact, it shows out many problems. It carries on many assumptions that theoretically can be done, but doesn't fit into practice. One of them is the possibility to set the steering on a fixed position in an infinitesimal amount of time.
For a point-to-point motion, the problem can be solved with more scientific rigor, directly only by calculating a law of motion that allows guiding the robot from the starting confi-

guration to the arrival one. In this way, the path appears to be an output of the planning algorithm. There are no physical obstacles as constraints but however virtual obstacles must be taken into account, such as the limitation on the steering angle.
From the mathematical point of view, to obtain a linear or piecewise linear solution, we need to solve a chained form system with constant input u. It can be shown that the chain systems of the form (2,n) are completely controllable. It is possible to use different types of inputs (u(t)):

- sinusoidal inputs
- piecewise constant inputs
- polynomial inputs

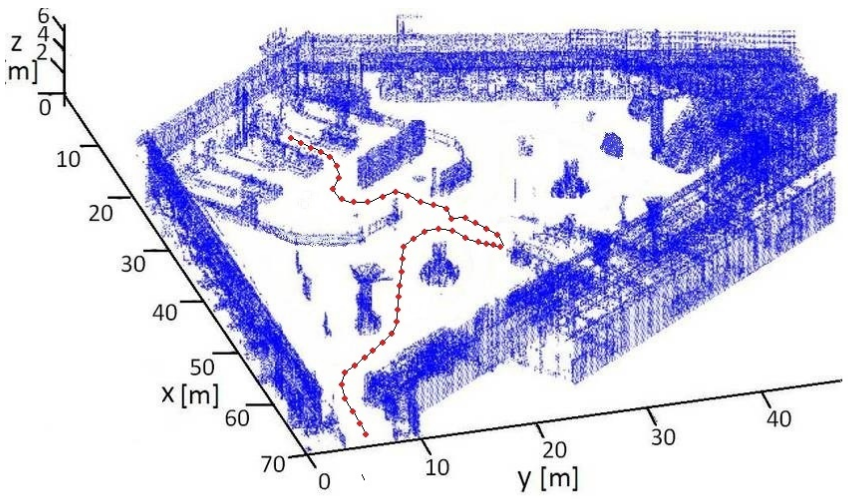

Figura 9: Image created with a linear path planning.

Intuitively we can immediately face that once we reach the first point we have to turn all the vehicles and make it align to the new segment. It is impossible if the vehicle is ongoing, we should stop the vehicle and turn the steering in the right position, align the vehicle to the new final point, stop again to put the steering wheel straight, and finally reach the end. With this iterative algorithm, we discover that this method isn't anymore the fastest because of the stops.
Moreover, so far seen methods allow the calculation of an admissible path and the relevant controls to bring a vehicle from a starting configuration to a landing one (point to point motion). But we cannot predict exactly what the planned route will be, which makes even more complex the task of moving the robot in an environment full of obstacles. It is so possible to calculate the traction and steering speed inputs (input trajectory) as a function of the desired cartesian path. We obtain equations that command the inputs, that should

be respected and so controlled. They let an optimal open-loop control starting in the exact initial position. Inputs depend only on the desired output Cartesian trajectory and its derivatives up to the third order. Therefore it is necessary, in addition to the above condition on the initial configuration, that the trajectory is differentiable up to three times concerning time. Here we can demonstrate that the segment path is not suitable anymore because it is not of class C3.
To be sure that the above conditions are respected, we can introduce the so defined Continuous curvature paths.
The first one that we can study is clothoid. It is well known, that the optimal paths in terms of length, for a simplified car model are made up of segments and circular arcs. Once more the most significant disadvantage of this class of paths is the presence of discontinuities of the curvature. As before these discontinuities occur in every transition among segments and arcs as the curvature jumps from zero to a non-null value. It is preferable, therefore, that the path is characterized by the continuity of curvature. To apply such a method we have to limit both curvature and his derivative.
The problem of planning continues curvatures paths is part of the class of problems called "path smoothing", in which it is attempted to "smooth out" the discontinuity starting from a nominal path, such as a polygonal line, or by a sequence of path configurations to obtain a smooth curve.
The curves used can be divided into two categories:

- Closed form expression

  B-spline

  Fifth degree polynomials

  Polar spline

- Parametric curves

  Clothoids

  Cubic spirals

  Spline G^2

Here we are going to present the most used one, the clothoid. They are obtained as a class of paths in which the curvature varies linearly with the length of the arc. Mathematically the law that describes the curvature is:

$$k = \frac{d\delta}{ds} = \lambda s + k(0)$$

λ is a constant value and describes how the curvature varies in function of the curvilinear coordinate. The solutions can not be solved in closed form but can be approximated with numerical methods. All parameters can be easily defined and approximated thanks to *Fresnel integrals* and their power series.

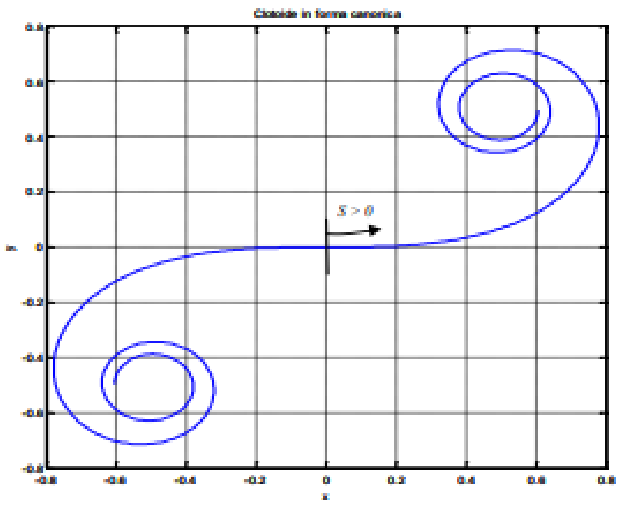

Figura 10: Chart of the clothoid in canonical form.

The clothoid allows us to manage directly the curvature of the Cartesian trajectory and is suitable to be united with geometric continuity to circular arcs and straight-line segments. In a similar way, we can connect the two basics part of the path with polynomial curvature

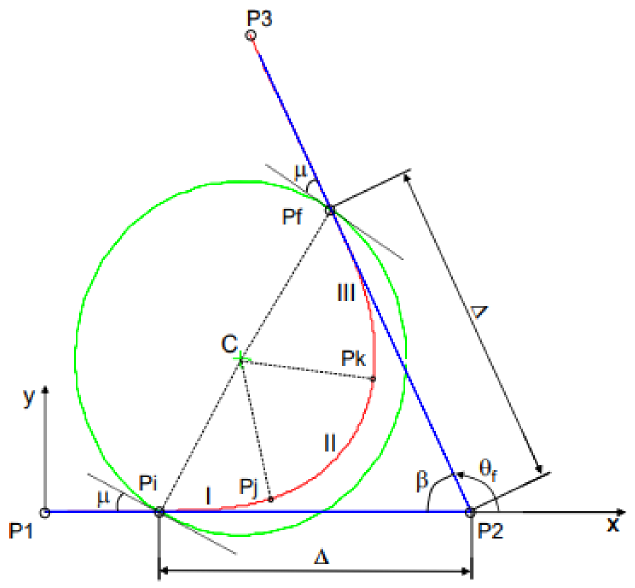

Figura 11: Building elements of continous curvature trajectory. In red the curve, in green the reference circumference of the curve. Stretches of curve I and III are the traits of clothoid, instead II is an arc of a circle.

paths. The curvature became a simple function of the third order. $k(s) = a+bs+cs^2+ds^3$ Similarly, it is for sure at C3 class (al least the third derivative is continuous). Again we can reconstruct all the functions that output the desired values and equations.
For the tracking of path or trajectory, it is possible to use heuristic control algorithms. Studying the initial case it is evident that the broken line that joins the points of passage is not an admissible path. We can implement a simple algorithm or rule to built an admissible path similar to the segmented one. We define the point on the plane with one more coordinate R_i.

$$P_i = \{x_i, y_i, R_i\}$$

R is defined as the distance from the point below which the reference passes to the next point.
If the vehicle is monitored to track the segment $P_{k-1}P_k$ and the distance from the point falls below the corresponding value of R_k control may pass to the next segment P_kP_{k+1}. This short description can let us choose the Heuristic method as the favorite for our solution because it is one of the fastest and at the same time one of the simplest. It can be suitable also because all our points are close to each other, so also more complicated methods produce similar paths, but take more time and more power.

5 Full code

In this section we present the full code for the program

```
// This example is derived from the ssd_mobilenet_object_detection opencv demo
// and adapted to be used with Intel RealSense Cameras
// Please see https://github.com/opencv/opencv/blob/master/LICENSE

#include <QCoreApplication>
#include <opencv2/dnn.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/imgproc/imgproc.hpp"
#include <librealsense2/rs.hpp>
#include <librealsense2-gl/rs_processing_gl.hpp>
#include <exception>
#include <iostream>
#include <stdio.h>
//#include </home/user/Software/librealsense/examples/measure/rs-measure.cpp>
//#include </home/user/librealsenseNew/librealsense/examples/sensor-control/api_how_to.h>
#include </home/user/librealsenseNew/librealsense/include/librealsense2/rsutil.h>
#include </home/user/librealsenseNew/librealsense/include/librealsense2/rs.hpp>
#include <cv-helpers.hpp>
#include <pcl/io/pcd_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/point_cloud.h>
#include <pcl/console/parse.h>
#include <pcl/common/transforms.h>
#include <pcl/visualization/pcl_visualizer.h>

//define the size of the detected window of 300 X 300

const size_t inWidth      = 300;
//define the size of the detected window of 300 X 300
const size_t inHeight     = 300;
const float WHRatio       = inWidth / (float)inHeight;
//define the ratio width/height
const float inScaleFactor = 0.007843f;
//define scale factor
const float meanVal       = 127.5;
//define men value
const char* classNames[]  = {"background",
        "aeroplane", "bicycle", "bird", "boat",
        "bottle", "bus", "car", "cat", "chair",
        "cow", "diningtable", "dog", "horse",
        "motorbike", "person", "pottedplant",
        "sheep", "sofa", "train", "tvmonitor"};
        //define the possible calles that the dnn can detect

int main(int argc, char** argv) try
//start of the itetation end of the declaration part and start of the real process
{
using namespace std;
using namespace cv;
//library to be used untill line 57
using namespace cv::dnn;
using namespace rs2;

Net net = readNetFromCaffe("/home/user/librealsenseNew/librealsense/wrappers/opencv
/dnn/build/MobileNetSSD_deploy.prototxt",
"/home/user/librealsenseNew/librealsense/wrappers/opencv
/dnn/build/MobileNetSSD_deploy.caffemodel");

//start streaming from Intel RealSense Camera
pipeline pipe;
//define pipeline to start the d435
auto config = pipe.start();
//auto configuration to get started
auto profile = config.get_stream(RS2_STREAM_COLOR)
.as<video_stream_profile>();
rs2::align align_to(RS2_STREAM_COLOR);

Size cropSize;
if (profile.width() / (float)profile.height() > WHRatio)
//start if actual ratio is bigger than the WHratio
{
cropSize = Size(static_cast<int>(profile.height() * WHRatio),profile.height());
//rescale the height
}
else
{
cropSize = Size(profile.width(),static_cast<int>(profile.width() / WHRatio));
//rescale the width
}

Rect crop(Point((profile.width() - cropSize.width) / 2,
(profile.height() - cropSize.height) / 2),
cropSize);

const auto window_name = "Display␣Image";
//define windows_name
namedWindow(window_name, WINDOW_AUTOSIZE);

while (getWindowProperty(window_name, WND_PROP_AUTOSIZE) >= 0)
{
// Wait for the next set of frames
auto data = pipe.wait_for_frames();
// Make sure the frames are spatially aligned
data = align_to.process(data);

auto color_frame = data.get_color_frame();
auto depth_frame = data.get_depth_frame();

// If we only received new depth frame, but the color did not update, continue
static int last_frame_number = 0;
if (color_frame.get_frame_number() == last_frame_number) continue;
last_frame_number = color_frame.get_frame_number();

// Convert RealSense frame to OpenCV matrix:
auto color_mat = frame_to_mat(color_frame);
auto depth_mat = depth_frame_to_meters(pipe, depth_frame);
//define depth_math as function of depth_frame_to_meters

```

```
Mat inputBlob = blobFromImage(color_mat, inScaleFactor,
Size(inWidth, inHeight), meanVal, false);
//Convert Mat to batch of images
net.setInput(inputBlob, "data");
//set the network input
Mat detection = net.forward("detection_out");
//compute output

Mat detectionMat(detection.size[2], detection.size[3], CV_32F, detection.ptr<float>());

// Crop both color and depth frames
color_mat = color_mat(crop);
depth_mat = depth_mat(crop);

float confidenceThreshold = 0.8f;
//define the condifence threshold / minimum accuracy required at 80%
for(int i = 0; i < detectionMat.rows; i++)
//for cycle
{
float save[i][3];
uchar histogram[i][3];

float confidence = detectionMat.at<float>(i,2);
// define the actual real confidence

if(confidence > confidenceThreshold)
//if cycle if we are sure that the detection is accurate
at least at 80% as defined before
{
size_t objectClass = (size_t)(detectionMat.at<float>(i, 1));

if(classNames[objectClass] == "person")
{
//define the two opposit points in right top and bottom left to define the rectangle
int xLeftBottom = static_cast<int>(detectionMat.at<float>(i, 3) * color_mat.cols);
int yLeftBottom = static_cast<int>(detectionMat.at<float>(i, 4) * color_mat.rows);
int xRightTop = static_cast<int>(detectionMat.at<float>(i, 5) * color_mat.cols);
int yRightTop = static_cast<int>(detectionMat.at<float>(i, 6) * color_mat.rows);

//define the rectangle as object from one point LB (xlb,ylb) with the lenght of 2 lines
//on x and on y (xrt-xlt) or (xrt-xlb)
Rect object((int)xLeftBottom, (int)yLeftBottom,
(int)(xRightTop - xLeftBottom),
//try also with (xrt-xlt) and also similarly in the following line
(int)(yRightTop - yLeftBottom));

object = object  & Rect(0, 0, depth_mat.cols, depth_mat.rows);

// Calculate mean depth inside the detection region
// This is a very naive way to estimate objects depth
// but it is intended to demonstrate how one might
// use depht data in general

Scalar m mean(depth_mat(object));
//define the depth as mean of the depth of each pixel contained in the rectangle

int xcenter = (xRightTop - (xLeftBottom / 2));
int ycenter = (yRightTop + (yLeftBottom / 2));
//we can use depth_mat in the center point  (SEARCH THE LIBRARY / DOCUMENTATION ON
//depth_frame_to_meters)
std::ostringstream ss;
ss << classNames[objectClass] << "␣";
//insert class name
ss << std::setprecision(2) << m[0] << "␣meters␣away";
//output the mean distance fixed at mean distance meter away
ss << ",␣centred␣in␣pixels␣";
//inseted new line to say center point
ss << xcenter ;
ss << "␣␣";
ss <<  ycenter;

String conf(ss.str());

rectangle(color_mat, object, Scalar(0, 255, 0));
//define rectangle color as green (0,255,0)
int baseLine = 0;
Size labelSize = getTextSize(ss.str(), FONT_HERSHEY_SIMPLEX, 0.5, 1, &baseLine);
//the size of the white box is slightly bigger than the dimention of the text

auto center = (object.br() + object.tl())*0.5;
center.x = center.x - labelSize.width / 2;


//define white rectangle for written part
rectangle(color_mat, Rect(Point(center.x, center.y - labelSize.height),
//it is centered in the center of the bounding box
Size(labelSize.width, labelSize.height + baseLine)),
//size of the label written
Scalar(255, 255, 255), FILLED);
//white color
putText(color_mat, ss.str(), center,
//the text should be centerd inside the box
FONT_HERSHEY_SIMPLEX, 0.5, Scalar(0,0,0));
//the text is black

float centrop[2];
//define the centerp in 2D as vector of 2 element
centrop[0]=center.x;
//assing the first value the center x
centrop[1]=center.y;
//assing the first value the center y
float centrov[3];
//define the centerv in 3D as vector of 3 element
float centrov1[3];
//define the centerv1 in 3D as vector of 3 element
float centerdepth = depth_frame.get_distance(centrop[0], centrop[1]);
//find out the distance of the center as get_distance of centerp


//   ad ora abbiamo le cordinate di profondità data come media
//   il centro xcenter e ycenter come misure algebriche dagli angoli
//   centrop centrov con centerdepth del centro dato da loro


```

```
//TO FIND OUT THE INTRISTIC PARAMETER

auto depth_stream = config.get_stream(RS2_STREAM_DEPTH)
.as<rs2::video_stream_profile>();
auto resolution = std::make_pair(depth_stream.width(), depth_stream.height());
auto in = depth_stream.get_intrinsics();
auto principal_point = std::make_pair(in.ppx, in.ppy);
auto focal_length = std::make_pair(in.fx, in.fy);
//auto matrix_parameters = std::make_tuple(i.coeffs[5]);
float c1 = 0.090554739064;
float c2 = -0.263085401898;
float c3 = 0;
float c4 = 0;
float c5 = 0;
auto matrix_parameters = std::make_tuple(c1,c2,c3,c4,c5);
rs2_distortion model = in.model;
//print the parameters
// cout << "Resolution: " << resolution.first << "," << resolution.second << endl;
// cout << "Principle point: " << principal_point.first << ","
// << principal_point.second << endl;
// cout << "Focal length: " << focal_length.first << "," << focal_length.second << endl;
// cout << "Matrix parameters are: " << std::get<0>(matrix_parameters)<< endl;
// cout << "Matrix parameters are: " << std::get<1>(matrix_parameters)<< endl;
// cout << "Matrix parameters are: " << std::get<2>(matrix_parameters)<< endl;
// cout << "Matrix parameters are: " << std::get<3>(matrix_parameters)<< endl;
// cout << "Matrix parameters are: " << std::get<4>(matrix_parameters)<< endl;
//return 0;

// abbiamo centrop con i valori sull'immagine del centro del bounding box
// e due vettori in 3d dove poter inserire i valori reali
// facciamo la conversione con leggi geometriche
// vedi funzione definita da realsense che ho copiato  rs2_deproject_pixel_to_point


float x = (centrop[0] - principal_point.first) / focal_length.first;
float y = (centrop[1] - principal_point.second) / focal_length.second;
float r2  = x*x + y*y;
float f = 1 + std::get<0>(matrix_parameters)*r2 + std::get<1>
          (matrix_parameters)*r2*r2 + std::get<4>(matrix_parameters)*r2*r2*r2;
float ux = x*f + 2*std::get<2>(matrix_parameters)*x*y +
           std::get<3>(matrix_parameters)*(r2 + 2*x*x);
float uy = y*f + 2*std::get<3>(matrix_parameters)*x*y +
           std::get<2>(matrix_parameters)*(r2 + 2*y*y);
x = ux;
y = uy;

// il punto in 3D ha valori dati da depth*x, depth*y, depth
// m[0] è il valore medio della profondità

centrov[0] = centerdepth * x;
centrov[1] = centerdepth * y;
centrov[2] = centerdepth;

float d;
d = m[0];
centrov1[0] = d * x;
//NON NECESSARY POINT
```

```
centrov1[1] = d * y;
//IT IS ONLY NEEDED TO CHECK DIFFERENCE FROM THE OTHER POINT
centrov1[2] = d;

cout << "THE␣PERSON␣IS␣CENTERED␣IN␣POINT:␣(" << centrop[0] << "," << centrop[1] << ")."
<< endl;
cout << "THE␣MEASURED␣DEPTH␣measured␣with␣get␣depth␣IS:␣" << centerdepth << "." << endl;
cout << "THE␣MEAN␣DEPTH␣(the␣one␣of␣the␣image)␣IS:␣" << m[0] << "." << endl;
cout << "WE␣CAN␣SEE␣THAT␣THE␣TWO␣DEPTH␣VALUE␣DON'T␣DIFFERE␣TOO␣MUCH␣SO␣WE␣ASSUME␣IT
IS␣RIGHT."<< endl;
cout << "COORDINATES␣OF␣THE␣POINT␣IN␣THE␣WORD␣REFERENE␣FRAME␣ARE:␣(" << centrov[0] <<
 "," << centrov[1] << "," << centrov[2] << ")." <<endl;
cout << "COORDINATES␣OF␣THE␣POINT␣IN␣THE␣WORD␣REFERENE␣FRAME␣ARE:␣(" << centrov1[0] <<
"," << centrov1[1] << "," << centrov1[2] << ")." <<endl;
cout << "WE␣REMIND␣THAT␣THE␣POSITIVE␣AXES␣POINT␣ON␣RIGHT,␣DOWN,␣INSIDE␣THE␣SCREEN."<<
endl;
cout << "WE␣ARE␣LOOKING␣FOR␣THE␣POINT␣ON␣THE␣GROUND␣SO␣WE␣SHOULD␣ASSUME␣THE␣SECOND
COORINATE␣EQUAL␣TO␣0:␣(" << centrov[0] << "," << 0 << "," << centrov[2] << ")." << endl;
cout << "We␣should␣now␣save␣all␣points␣to␣make␣a␣path␣planning." << endl;

// il frame rate è di 30 fps quindi uno ogni 1 ogni 0.033 sec. una persona in media
//cammina a 6 km/h quindi 1.4m/sec per essere cautelativi 1.6m/sec
// ogni 0.033 sec si fanno 0.053m

//ho scelto mezzo secondo quindi mezzo metro di spostamento perchè la precisione delle
//misure non è elevata, ma comunque dell'ordine di grandezza di meno del mezzo metro

float dist = 0.8;

if(i < 15)
{
save[i][0] = centrov1[0];
save[i][1] = 0;
save[i][2] = centrov1[2];
}

Mat histogram1[i];
Mat histogram2[i];
Mat histogram3[i];
// definiamo la variabile istogramma come array di valori
Mat src, dst;
// creiamo le matrici necessarie
src = color_mat;
// carichiamo l'immagine da confrontare come object vedi def //sopra come rettangolo
//bounding box così però mi carica tutto il frame

Mat src1(src, Rect(xLeftBottom, yLeftBottom, xRightTop, yRightTop) );

vector<Mat> bgr_planes;
split( src1, bgr_planes );
int histSize = 256;
float range[] = { 0, 256 } ;
const float* histRange = { range };
bool uniform = true; bool accumulate = false;

Mat b_hist, g_hist, r_hist;

```

```
// Compute the histograms:
calcHist( &bgr_planes[0], 1, 0, Mat(), b_hist, 1, &histSize, &histRange, uniform,
accumulate );
calcHist( &bgr_planes[1], 1, 0, Mat(), g_hist, 1, &histSize, &histRange, uniform,
accumulate );
calcHist( &bgr_planes[2], 1, 0, Mat(), r_hist, 1, &histSize, &histRange, uniform,
accumulate );

// Draw the histograms for B, G and R
int hist_w = 512; int hist_h = 400;
int bin_w = cvRound( (double) hist_w/histSize );

Mat histImage( hist_h, hist_w, CV_8UC3, Scalar( 0,0,0) );

// Normalize the result to [ 0, histImage.rows ]
normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() );
histogram1[i]= b_hist;
histogram2[i]= g_hist;
histogram3[i]= r_hist;

// Draw for each channel
for( int hs = 1; hs < histSize; hs++ )
{
line( histImage, Point( bin_w*(hs-1), hist_h - cvRound(b_hist.at<float>(hs-1)) ) ,
Point( bin_w*(hs), hist_h - cvRound(b_hist.at<float>(hs)) ),
Scalar( 255, 0, 0), 2, 8, 0  );
line( histImage, Point( bin_w*(hs-1), hist_h - cvRound(g_hist.at<float>(hs-1)) ) ,
Point( bin_w*(hs), hist_h - cvRound(g_hist.at<float>(hs)) ),
Scalar( 0, 255, 0), 2, 8, 0  );
line( histImage, Point( bin_w*(hs-1), hist_h - cvRound(r_hist.at<float>(hs-1)) ) ,
Point( bin_w*(hs), hist_h - cvRound(r_hist.at<float>(hs)) ),
Scalar( 0, 0, 255), 2, 8, 0  );
}
/// Display
namedWindow("calcHist␣Demo", CV_WINDOW_AUTOSIZE );
imshow("calcHist␣Demo", histImage );
// tutta la parte dalla definizione di histogram 1,2,3 serve solo per l'immagine

float t = 0.1;
//threshold da dare al cambiamento dell'istogramma bisogna capire come visualizzare come
//sigolo valore l'istogramma
// per questo valore ci vorrebbero dei try and attempt, può essere tranquillamente più
//grosso ma è un rischio che ritengo inutile
// praticamente vuol dire che tra 5 histogram consecutivi c'è una corrispondenza di
// almeno il 10 % su tutti i colori.

if(i>15)
{
if(abs(compareHist( histogram1[i], histogram1[i-5], 0 ))>t)
// ovviamente l'istogramma è dato da tre calori bisogna fare tre if su ogni valore
//dell'istogramma
{
if(abs(compareHist( histogram2[i], histogram2[i-5], 0 ))>t)
{
if(abs(compareHist( histogram3[i], histogram3[i-5], 0 ))>t)
{
if(fabs(save[i][0]) < fabs(save[i-15][0]+ dist) ||
fabs(save[i][0]) > fabs(save[i-15][0]- dist) )
{
if(fabs(save[i][2]) < fabs((save[i-15][2]+ dist)) ||
fabs(save[i][2]) > fabs(save[i-15][2]- dist) )
{
save[i][0] = centrov1[0];
save[i][1] = 0;
save[i][2] = centrov1[2];
}
}
}
}
}
}

// il ciclo for e gli if servono come controlli sulla distanza e sugli istogrammi in modo
//che non si prendano bounding box sbagliati

// N.B.:
// ho usato centrov1 perchè ho valutare quale sia il valore più realistico
// centrov1 è con la media e sembra il più veritiero
// logica vorrebbe che il valore della profondità del punto medio (CENTROV) sia più
// realistico, ma sembra essere meno affidabile
// LASCIO TUTTE LE RIGHE STAMPATE SOPRA COME CHECK DI QUALE PUNTO SIA IL MIGLIORE.
// IN SEGUITO POSSONO ESSERE TOLTE

cout << "Array␣con␣i␣punti␣salvati␣ogni␣volta␣dovrebbe␣essere␣più␣lungo␣di␣un␣elemento" <<
save << "." << endl;
}
}
imshow(window_name, color_mat);
//show the rectangle with that image and color
if (waitKey(1) >= 0) break;
//close function
}

return EXIT_SUCCESS;
}
}
catch (const rs2::error & e)
{
std::cerr << "RealSense␣error␣calling␣" << e.get_failed_function() << "(" <<
e.get_failed_args() << "):\n␣␣␣␣" << e.what() << std::endl;
return EXIT_FAILURE;
}
catch (const std::exception& e)
{
std::cerr << e.what() << std::endl;
return EXIT_FAILURE;
}
```

6 Possible future implementation

We could think to better implement the algorithm by creating some stricter control on the position, thanks to a more specific histogram or with a connection via bluetooth or gps of the phone that the suitcase should follow. The app can also permit to the person to set some basic parameters of the system and to specifically choose the person to follow. There could be some notification or haptic feedback it the distance become too big or if occur some problems.

We could also think to implement a Kalman filter in order to predict the motion model and therefore the next position of the person in the case in which we lose the bounding box of the person. There could be included also mobile obstacles prediction and real time obstacles avoidance. Other common prediction methods includes also learning algorithm, biometric approach and polynomial fitting method. Kalman filter anticipates the future path based on the past information. In this way, also a better path planning could be implemented using more complex, but more robust methods such as A* in real time. Thanks to Kalman filter we could create a Predictive RA*. In this way the suitcase can behave in an anticipatory manner making proactive decision reducing collision and lost of the signal risks. Kalman filter represent prediction as the probability of the point to move in a certain way. Gaussian probability distribution are used. The mean and the variance as output of the KF can be used for gauge the risks at each position of the path. It can also be used for a simpler prediction of the next position of the bounding box. It is trivial that, in this way, a simple A* can be implemented creating a more robust algorithm.

7 Bibliography

Appunti del corso di ROBOTIC PERCEPTION AND ACTION, *Mariolino De Cecco*
Autonomous Vehicles Navigation, Trajectory Planning and Control, *Luca Baglivo*
Appunti del corso di COMPUTER VISION, *Conci Nicola*
Intel RealSense^{TM} Depth Camera D435i IMU Calibration January 2019
Intel RealSense^{TM} Depth Module
D400 Series Custom Calibration January 2019
Predictive Path planning Algorithm Using Kalman Filters and MTL Robustness*S. Alqahtani, S. Taylor, I. Riley, R. Gamble, R. Mailler*
https://github.com/IntelRealSense/librealsense/wiki/Projection-in-RealSense-SDK-2.0

$$Bashar, Manijeh\&Haneda, Katsuyuki\&Burr, Alister\&Cumanan, Kanapathippillai.$$

$$(2018).AStudyofDynamicMultipathClustersat60GHzinaLargeIndoorEnvironment.$$