# FAST 3D CLUSTER-TRACKING FOR A MOBILE ROBOT USING 2D TECHNIQUES ON DEPTH IMAGES

ARNAUD RAMEY, MARÍA MALFAZ, AND MIGUEL A. SALICHS

Abstract. User simultaneous detection and tracking is an issue at the core of Human Robot Interaction (HRI). Several methods exist and give good results, many of them use image processing techniques on images provided by the camera. The increasing presence in mobile robots of range-imaging cameras (such as, structured light devices as Microsoft Kinects) allows us to develop image processing on depth maps. In this article a fast and lightweight algorithm is presented for the detection and tracking of 3D clusters thanks to classic 2D techniques such as, edge detection and connected components applied to the depth maps. The recognition of clusters is made using their 2D shape. An algorithm for the compression of depth maps has been specifically developed, allowing the distribution of the whole processing among several computers. The algorithm is then applied to a mobile robot for chasing an object selected by the user. The algorithm is coupled with a laser-based tracking to make up for the narrow field of view of the range-imaging camera. The workload created by the method is light enough to enable using it even with processors with limited capabilities. Extensive experimental results are given for verifying the usefulness of the proposed method.

## 1. INTRODUCTION

Human Robot Interaction (HRI) requires the robot to be aware of its environment. It needs to understand what lies in front of it, if users are there, and if so, if they want to interact with him. Furthermore, recognizing the objects that were previously seen is an important feature for obtaining a robot that learns from its experiences. We indeed expect it to recognize places, objects and people previously seen. Such a feature is of paramount importance in social robotics, where interaction with human users and daily objects is the keystone of the robot activity.

In this article, we aim at giving a social robot the possibility to detect and follow objects or people. The user indicates the robot what is the object to track, then, the robot plans its motion to stay at a given distance of this object-of-interest . The method is mainly based on the use of depth maps, which allows its use on a wide range of robots. For robust tracking while moving, a fallback method using a laser scanning range finder is used. Such a skill is a very useful feature for the social robot: making it approach to a given object or follow a given user is a frequent need.

2000 *Mathematics Subject Classification.* Primary 54C40, 14E20; Secondary 46E25, 20C20.

*Key words and phrases.* Depth map, Image segmentation, Multiuser detection, Human robot interaction,

The authors gratefully acknowledge the funds provided by the Spanish MICINN (Ministry of Science and Innovation) through the project "*Applications of Social Robots / Aplicaciones de los Robots Sociales*".

The article is structured as follows: in section 2, a comprehensive review of the existing solutions is presented. In section 3, we will define in further detail the goals and specifications of our research. In section 4, we will present the algorithmic details of our solution. Then, in section 5, we present the results of a comprehensive series of tests that show the validity of the proposed algorithm. Finally, its usefulness and the future directions of this work is discussed in section 6.

## 2. RELATED WORK

A depth map is an image that contains information about the distance of the surfaces which can be viewed from a viewpoint. As such, it directly provides a spatial comprehension of the observed scene. Devices that can provide depth maps have existed for many years, for instance via stereo vision. However, their presence on mobile robots was limited due to a relatively high price, a complex calibration, and an additional computational workload for the robot. This might explain why not many articles use information from a depth map.

Many techniques focus on mixing data from several devices, such as cameras, laser scanners, etc. Among them we can underline the works of Muñoz-Salinas ([15, 16]). However, the use of multi-modal fusion is often computationally costly and requires several expensive and bulky devices. Therefore, it is not perfectly suitable for small mobile robot bases.

Some articles, however, also provide solutions only using a stereo camera or a range imaging device. The release of the affordable and reliable Kinect device at the end of the year 2011 led to its soaring use in robotics. In [22], a simple motorized support for a Kinect device, called KATE (Kinect Active Tracking Equipment) is presented. It has two degrees of freedom, which enables the camera device to look at the object of interest. The latter can either be provided by face detection, or the center of the image. The segmentation of the depth images around this center is made by a modified version of the GrabCut method ([21]). However, the KATE platform is not mobile, making the segmentation much easier. The patented PrimeSense NITE middleware [2] allows detecting and tracking human shapes from depth maps. It has been shown to work at a very high frame-rate even with average CPUs. Although the technique employed and the source code are not available, it is likely that motion analysis and clustering techniques are at the core. Indeed, the human detection is activated by their motion. As such, detecting a still, motionless audience becomes challenging. This is especially an issue in HRI, where the user can stand still in front of the robot while speaking with him. However, the software is aimed at playing video games on the Microsoft XBox 360 [13]. As human players are bound to move their body to play, this context solves the detection problem.

Stereo-vision devices are also used on mobile platforms ([11], [12]). A complete pedestrian detection system only based on stereo-vision is presented in [11]. A polar-perspective map is built then segmented into regions of interest. The system is compatible with classical image processing techniques, such as appearance-based algorithms. Stereo-vision is, according to authors, a very liable solution for navigation and pedestrian detection. However, their system is designed for cameras mounted on outdoors vehicles, with ranges between 5 and 50 meters. It makes it difficult to use for indoor robotics platforms. In [12], a Point Grey stereo camera mounted on top of a mobile platform provides depth maps. Users detection is made through edge detection with a Canny filter ([4]). The tracking is made by an Extended Kalman Filter. However, the process requires a complex calibration process and restricted experimental results do not clearly state the robustness of the method. In a related field, the glasses for people with impaired vision presented in [14] perform obstacle detection based on segmentation using only depth images. The developed framework works with non-constrained camera position and orientation, and because it discards color data, it also works in the dark. However, its use is limited to the detection of objects coming closer and their notifications to the user. There is no recognition or tracking of the objects being detected.

The method proposed in this paper aims at correcting some of the limitations of the articles previously presented. It focuses on hardware requirements easy to meet and a light workload. The code is released under the LGPLv3 license and integrated into a very common software platform, allowing other teams to improve it further easily.

## 3. PROBLEM STATEMENT

The aim of our work is to provide a mobile robot a lightweight algorithm for clusters detection and tracking. These clusters can be human users, or objects. The only required input data is the stream of depth maps. The RGB data, i.e. color images, also supplied by the Kinect, is discarded in this article. This ensures the compatibility of the algorithm with devices such as PMDVision CamCubes or Asus Xtion PRO devices, that do not supply RGB images. It should be able to be either processed on-board or on a remote computer, but the goal is to have it done on-line, in real time. As such, the visual tracking of the object-of-interest is made simultaneously with the navigation of the robot.

More accurately, the tracking is made at two levels: on the first hand, at the sensor processing level, the robot is expected to keep track of the followed cluster. On the second hand, we want the former to physically move itself and keep close to the latter. The whole process results in the robot coming to the tracked cluster and following its trail if it is moving.

### 3.1. **Hardware specifications.** The target robot for this application is the social robot *MOPI* (non-definitive name). It is a home-brew robot of the RoboticsLab of University Carlos III of Madrid shaped as a mobile, car-like platform. It is most notably equipped with a Microsoft Kinect device tiled at 45°, and a (non tiltable) Hokuyo laser scanning range finder. The communication is made through to a WIFI connection.

Furthermore, we also make use of a touch screen computer. It is a TravelMate C110 computer equipped with a touch-sensitive screen, and a CPU of modest performance. It displays the Graphical User Interface, which enables the user to choose an object-of-interest .

### 3.2. **Software specifications.** The robot MOPI works according to the AD paradigm, as presented in [1]. This paradigm handles skills relying on primitives. Primitives are in direct communication with the physical devices of the robot, and send elementary orders to them. This includes the base motors, the laser sensor, the camera, etc. A skill is the ability of the robot to do a specific action. It relies on the data supplied by the primitives. The actions generated by a skill can be numerous: move the car to a given point, play games with the user, interact with electric appliances, etc.

Since it is an experimental platform, the robot MOPI has been used as a bridge between the traditional implementation of AD, as seen in [20], and a new one relying on the communication mechanisms of *ROS*, the Robot Operating System [17]. The version used for this application is ROS *Electric* running on top of Ubuntu 10.10. The use of ROS most notably gives the possibility to redistribute the computational workload. We can send easily via the wire or a wireless connection raw data from the sensors to remote computers for processing. The latter can be more powerful than the robot embodied PCs, also lightening the computation workload of the main computer. Then, the processed data is sent back to the robot. It also embeds the Stage simulator [9], which gives us the possibility of making first outlines of our algorithms, for instance the tracking one, before trying them on the real robot.

## 4. APPROACH: DETECTION AND TRACKING PROCESS

The whole processing pipeline is illustrated in Figure 1.

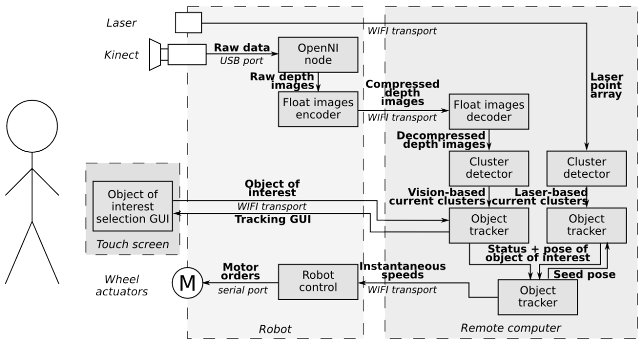


FIGURE 1. The flow chart of the whole sytem. It presents all the components needed for the vision-based algorithm and does not include the laser data fusion presented later in subsection 4.6.

It is detailed in the following order: the acquisition of depth images from the depth sensing device is detailed in subsection 4.1. As previously said, the RGB (color) images that can be provided by the Kinect sensor are not used. Thus, the algorithm can also be used with single depth imaging devices, such as CamCubes.

The algorithm here developed for converting float depth images into byte images is explained in subsection 4.2.

The compression of depth images for remote processing then comes in subsection 4.3.

Then, our tracking algorithm is articulated in three phases:

(1) **Cluster detection:** the clusters are found in the current depth map thanks to 2D image processing techniques. This is presented in subsection 4.4.
(2) **Cluster matching and tracking:** the clusters of the current depth map are matched to the objects found in the previous depth maps. If the user has selected an object to track, the 3D position of this object is estimated. More detail is given in subsection 4.5.
(3) **Robot control:** depending on the tracking results, the robot motion is controlled. For instance, if the object selected by the user has been found,

the robot will move and come closer. This is explained in subsection 4.6. Will also be presented how the results from our algorithm and from another one, based on the data of the laser, are mixed.

4.1. **Depth image acquisition.** The acquisition of the depth map is made thanks to the software provided by the ROS architecture. A so-called node is in charge of the communication through the USB port. The stream acquired via that port is transformed into proper depth maps. For each pixel $(x, y)$ in the depth map, the pixel value at $(x, y)$ corresponds to the distance of the closest object intersecting the 3D ray passing by this pixel, in meters.

As such, the values of the acquired depth map are float values, with a range depending on the device. For the Microsoft Kinect, they are typically within one to ten meters. Furthermore, the constructed light patterns projected by the device can be reflected for shiny surfaces for instance. The depth map then also contains some undefined values, represented by $NaN$ .

4.2. **Remapping of float images to `[0..255]` values.** Most image processing technique require the input image to be in the typical RGB space. In this space, grayscale values are represented as bytes values (while colors are tuples of bytes):

$$v_{byte} = g \in [0..255] \text{ for gray-scale images}$$

However, it was seen previously that the values depth maps supplied by the ROS node are decimal numbers representing the physical distance of the object. We then want to convert these images to the [0.255] range, while keeping track of the undefined $NaN$ points.

It is important not to lose the information about these undefined points. They indeed correspond to physical zones where we have no information. For instance, on the picture visible in Figure 2 (a), the black polygons in the ceiling could be for instance some reflecting surfaces or some nests in the roof, etc. In this example, they correspond to the neon lights. They reflect the light patterns emitted by the Kinect projector.

For each depth map, this remapping is made in several steps.

(1) Detection of the values range of the depth map: we find the minimum $m$ and the maximum $M$ values of the defined pixels in the current depth map.
(2) Computation of the affine transform: we determine $\alpha, \beta \in \mathbb{R}$ such as the transform $v_{byte} = \alpha \times v_{float} + \beta$ maps the float values to the byte values:

$$v_{float} \in [m..M] \mapsto v_{byte} \in [1, 255]$$

Note that the 0 value is discarded. It is used to represent the undefined $NaN$ values.

4.3. **Data compression for remote image processing.** Float images can come from various origins, for instance depth maps of a RGBD camera like Kinect. It can be desired, for CPUs with limited capabilities, to send out via the wire these images for processing them on a remote computer.

However, ROS did not support compression for float images. We hence developed a package for image transport providing this feature. Float images are remapped to byte images using an affine transform, as seen in subsection 4.2. They are then compressed using traditional image compression libraries.

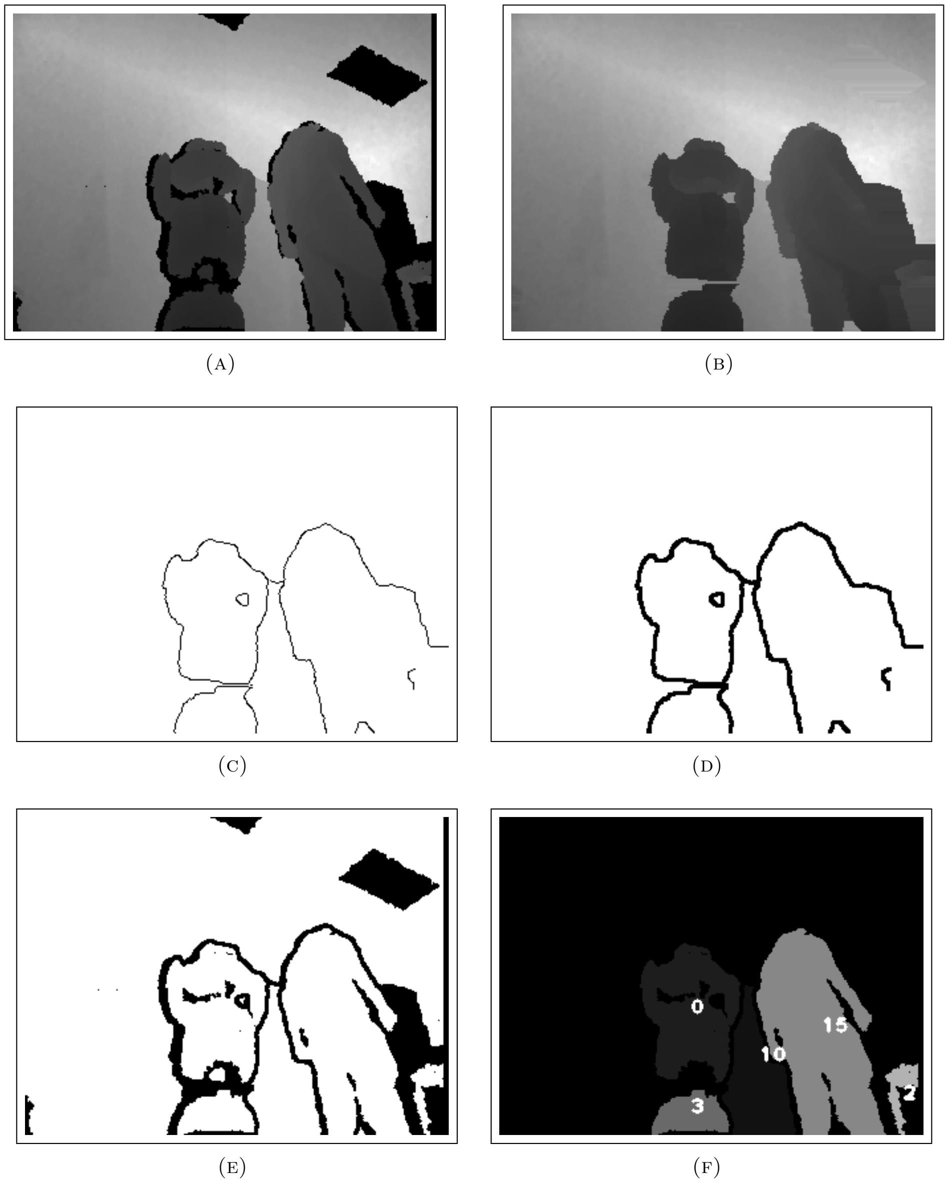


Figure 2. The detection flow chart.
(a): The depth map, as supplied by the range-imaging device (Kinect), and remapped to visible colors as seen in subsection 4.2.
(b): The "clean" depth map, after handling the *NaN* values. The technique used here is *directional propagation* (to the left).
(c): Canny edge detection on the clean depth map
(d): Morphological erosion of the Canny edges
(e): Recombination of the opened Canny edges and the *NaN* values
(f): The connected components found and their matches to objects. Each one is drawn with a color corresponding to the object it was matched to, and the index of this object is overlaid.

4.4. **Clusters detection.** In this section, the method used to find the 3D clusters in the current depth map is explained.

To be able to make use of conventional vision techniques, the float depth map is first converted into an usual images with values in the [0..255] span, by using the technique seen in subsection 4.2.

4.4.1. *Undefined values handling.* For an easier handling of the remapped depth map, *NaN* values are filtered. They correspond to a failure in the depth estimation at the given pixel. For instance, the Canny algorithm explained afterwards would fail with an image containing *NaN* values, as they are seen as a regular 0 value.

Three different and concurrent ways of solving the problem have been tried:

- **Average border**: on each side of the depth map, the average thickness, in pixels, of the *NaN* border is computed. Then the rectangular area corresponding to this average border out of the depth map is cropped out. For instance, if in average, the five first pixels at each line are undefined, the five first pixels of all lines are removed.
- **Inpainting:** the technique of inpainting, presented in [3], can be used for replacing undefined *NaN* values in the remapped depth. Inpainting was primarily designed for removing watermarks or damage to an image. It propagates the color values at the border of the damaged areas, and mixes them. Its use makes especially sense here as the *NaN* values correspond to unsuccessful depth measurements from the range-imaging device.
- **Directional value propagation:** Another way to fill the missing information is to propagate the values in a direction. It basically consists in setting the value of each undefined pixel to the one of its closest defined neighbor to the left. For undefined pixels at the left border, we set them to the closest defined pixel to the right. A sample is visible in Figure 2 (b).

A discussion about the advantages of each method is presented in subsection 5.5.

4.4.2. *Edge detection.* Now the depth images have been transformed into standard byte images and the undefined values have been handled. We can apply without problem "classical" image processing algorithms. As the goal is to find clusters in the depth map, we use a Canny filter on the remapped image [4]. The Canny filter helps us detecting edges in a grayscale image.

It requires two thresholds, a low and a high one. Two edges maps are obtained by passing first a Sobel operator on the grayscale image, then thresholding with the two thresholds. A Sobel operator is a linear filter based on a simple $3 \times 3$ kernel, and it approximates the gradient operator on the grayscale image.

The high threshold edge map contains broken, discontinuous edges, but they are likely to belong to the real contours of the objects. On the other hand, the low threshold edge map contains continuous edges, but with many edges that are not useful. The two maps are combined to create an "optimal" edge map: If a chain of the low threshold map enables to connect two pixels of the high threshold map otherwise disconnected, this chain is added to the final edge map. All the isolated chains of the low threshold map are then removed. A sample is visible in Figure 2 (c).

The two parameters of the Canny filter are defined for values of the RGB space, and their meaning depend of the values of $\alpha, \beta$. Hence, they are not consistent between frames. This is why instead of setting the thresholds, we set as constants the values of $\alpha \times low\_threshold, \alpha \times high\_threshold$ between frames. Parameter $\beta$ is not taken into account as it is a constant offset factor, it is neutralized by the gradient simulated by the Sobel operator.

4.4.3. *Morphological erosion.* We have explained in subsubsection 4.4.1 how undefined *NaN* values have been handled. However, it still can weaken local contrasts. To compensate this effect, we apply a morphological transformation that thickens the edges. This can close some edges that had been left opened by the Canny filter.

This result is obtained by passing a morphological erosion filter on the image. The erosion filter replaces each value in the image with the maximum of the values of the surrounding pixels. We use here a $3 \times 3$ pixels kernel. It could close gaps in the border that are up to 4 pixels wide. A sample is visible in Figure 2 (d).

Then, as we do not want to include the undefined *NaN* values in the objects, we need to restore the undefined values erased in the step of handling of undefined values. They are restored by: first, computing the minimum of the original remapped and the eroded edge map, second, thresholding this minimum with a binary threshold at value 0. A sample is visible in Figure 2 (e).

4.4.4. *Fast connected component detection.* First, we define *connected components* for depth maps: two pixels of a depth map belong to the same connected component if there is a chain from one to the other, such as there is no depth gap between two consecutive elements of the chain.

At this step of the processing pipeline, an edge map has been computed that also includes the undefined *NaN* values returned by the range-imaging device. A 3D object visible in the current depth map corresponds to a cluster without discontinuity in the inside, and with a discontinuity at the border with the neighbor pixels. It should then correspond to a connected component in our edge map.

In a previous work ([18]), the authors presented a lightweight and fast algorithm for connected components fetching in a monochrome image, such as our edge map. It is based on an efficient representation in memory of the connectivity between components thanks to a *disjoint-sets* forest. The disjoint-sets data structure was first presented in [8] in 1964. In [19], this algorithm was benchmarked against two popular ones for components labelling, flood-fill and Chang's [5]. It turned out to be 30% faster on the image collection used in the article. A sample is visible in Figure 2 (f).

The method also returns the bounding boxes of the components. We recall here that the bounding box of a set of 2D points is the smallest rectangle that fits all the points within its surface.

4.4.5. *Cluster filtering.* Some filters can be applied for removing some of the non-relevant clusters found in the current depth map. The actual version of the algorithm discards the too small clusters. This is obtained by checking if the size of the corresponding connected component is under a given threshold.

4.5. **Cluster matching and tracking.** In the previous section, the detection of the different clusters in the current depth map was explained. However, our aim is to give temporal coherency to this cluster information: we want to match these clusters to objects detected in previous images. For instance, if the robot is following a person, we want to spot what is the cluster corresponding to this person in the current image. Having this temporal coherency for objects gives the possibility of a meaningful chasing motion for the robot.

Let us define the concept of *object* in the scope of our detector: an object, is a set of connected components of the successive depth maps, each of which corresponds to the same physical entity. There is, at most, one connected component in each depth map corresponding to a given object. However, it might be that some frames do not contain any connected component corresponding to a given object, i.e., the object might be occluded or not successfully recognized.

For the clarity of the concept, an example issued from real data is presented in Figure 3. It shows the recognition state of an object labeled 3. This object has been recognized five times in the previous depth maps. For each frame, we have stored the connected component and the bounding box representing it. On the other hand, the object has not been recognized in the depth maps number $9, 10, 11, 14, 15$.

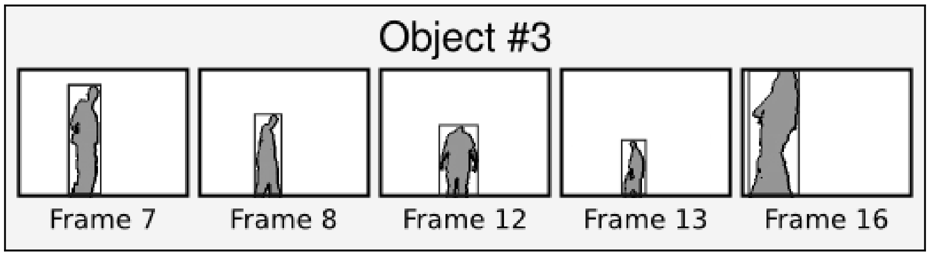


FIGURE 3. A example to clarify the concept of an object representation. The data is real and comes from the tracking sequence of a given user. The object connected component is the filled area and the bounding box is marked as a rectangle.

In order to maintain the computation time as low as possible, the recognition is made in two steps.

(1) A first rough matching using only the bounding boxes of the components gives us a first estimation of which clusters correspond to which objects. This is explained immediately after.
(2) Ambiguities are solved using an analytic distance with strong discrimination properties, the Hausdorff distance. This is explained in subsubsection 4.5.2.

The definitive matching is the one obtained at the end of this second phase.

4.5.1. *Rough estimator: bounding boxes correspondences.* Let us represent a given connected component $C$ of the current depth map. We want to have a fast estimation of what object is the most likely to be matched with $C$.

Let us now consider a given appearance at frame $i$ in the past of an object $O$ in the recognition history, that we call $O^i$. We compare the bounding box $bb_O^i$ of this appearance with the bounding box $bb_C$ of $C$.

Definition of the bounding box distance. Let us define a bounding box distance $d_{bbox}$ such as $\forall X, Y$ bounding box, $d_{bbox}(X, X) = 0$, and the more similar $X$ and $Y$, the smaller $d_{bbox}(X, Y)$.

Let us write $bb_X = \{TL_X, BR_X\}, bb_Y = \{TL_Y, BR_Y\}$ where $TL$ refers to the top-left corner of the bounding box, and $BR$ refers to the bottom-right corner.

Then, we define $d_{bbox}$ as defined in Equation 1. This corresponds to the sum of the distance between corresponding corners of both bounding boxes.

$$d_{bbox}(X,Y) = d\,(TL_X, TL_Y) + d\,(BR_X, BR_Y) \tag{1}$$

A distance function between 2D points, however, needs to be chosen. Three usual choices are possible, as written in Equation 2.

$$\begin{cases} d_{L_1}(a,b) = \mid a.x - b.x \mid + \mid a.y - b.y \mid \\ d_{L_2}(a,b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2} \\ d_{L_\infty}(a,b) = max(\mid a.x - b.x \mid, \mid a.y - b.y \mid) \end{cases} \tag{2}$$

Keeping in mind the speed of execution, the $L_1$ norm is here chosen. The (Euclidean) $L_2$ norm indeed needs expensive square root computations.
Application to the tracking. The result of $d_{bbox}(bb^i_O, bb_C)$ gives us an idea of the similarity of $O^i$, i.e. the appearance of the object $O$ at frame $i$ in the past, and $C$, i.e. the current connected component of the depth map. The smaller the value, the more likely it is that $C$ is an appearance of the object $O$ in the current frame. In addition, to take into account the age of this appearance of $O$, we weight the obtained $d_{bbox}(bb^i_O, bb_C)$ by the age (in seconds) of the depth map where $O^i$ belongs.

However, this estimation only takes in consideration the position of the objects in the frame, and not their shape. For our given connected component $C$, we compare it to all the appearances of all objects and store all the obtained marks in a list. We then sort this list by similarity, that is with the lowest $d_{bbox}$ distances firsts. Then, we use the precise estimator presented in the next section iteratively on each element of this sorted list. This gives us the final distance $d_{final}$ of $C$ with each object that appeared the previous frames. [1]

4.5.2. *Precise estimator: modified Hausdorff distance.* Using the rough matching with bounding boxes, matching ambiguities can occur. For instance, two connected components, similar in size and close to the last apparition of a given object, could both correspond to the same object then generate an ambiguous matching. For solving such ambiguities, we need a mathematical tool for properly comparing the shapes of components.
Modified Hausdorff Distance. In [6], a comparison is made between the different ways of computing a distance between two sets of points. According to these findings, we use the distance $d_{22}$, defined as in Equation 3.

$$\begin{gathered} \forall A, B \in \mathbb{R}^{2\mathbb{N}}, d_{22}(A,B) = max\left(d_6(A,B), d_6(B,A)\right) \\ \text{with } \begin{cases} d_6(A,B) = \dfrac{1}{\mid A \mid} \displaystyle\sum_{a \in A} d(a,B) \\ d(a,B) = \underset{b \in B}{min} \, \|a - b\| \end{cases} \end{gathered} \tag{3}$$

[1] We can thus avoid the computing of $d_{final}$ for the majority of the object appearances. If, for a given object appearance, its $d_{bbox}$ is superior to the smallest $d_{final}$ computed at that time, we can stop comparing all the following appearances in the list. They will indeed inevitably obtain a higher $d_{final}$ distance than this match.

For a given connected component $C$, $d_{22}(C, C) = 0$, and given two connected components $A$ and $B$, the lower the result returned by $d_{22}(A, B)$, the more similar they are.

$d_{22}$ requires the choice of a norm for estimating the distance between two points. Accurate component comparator. The Hausdorff distance helps solving ambiguities. For each frame, all the components are first scaled to a given size, say $32 \times 32$ pixels.

Then, the different candidates for the same object are compared using $d_{22}$ distance. In a similar way to how we weighted $d_{bbox}$, $d_{22}$ is weighted by the age of the object appearance. The final distance is then given by:

$$d_{final}(O^i, C) = d_{bbox}(bb^i_O, bb_C) + d_{22}(O^i, C)$$

The best object candidate for the recognition of $C$ is the one getting the lowest $d_{final}$. It is considered as a positive match if it gets a mark inferior to a given threshold. This mark is empirically determined and can be set through the graphical user interface . Its default value is 0.8.

4.5.3. *Object tracking: selection of the object-of-interest.* At this step, the connected components of the current depth map are matched to the objects already found. However, the tracking needs to know what object we aim at tracking. In other words, the user needs a way to select the object-of-interest . We will present in subsection 5.2 a Graphical User Interface we have developed.

The 3D pose of the object-of-interest is obtained thanks to the 3D reprojection of the barycenter of all points of the object-of-interest . The cluster matcher periodically republishes this 3D pose of the tracked object-of-interest .

## 4.6. **Robot motion and user tracking.**

4.6.1. *Multi-sensor fusion.* As all range-imaging devices, the Kinect is limited by a field of view. Horizontally, it can see objects that belong to an angular domain of 57 degrees, and of 43 vertically. Neither can it detect objects at a distance inferior to 1.2 meters according to the Kinect datasheet. Furthermore, for higher distances, the further the object, the lower the precision. On the other hand, the robot is also equipped with a Hokuyo laser scanning range finder, which can only detect objects on a horizontal plane, but with a field of view of 240 degrees. However, its range is limited to approximately four meters. Both fields of view are compared in Figure 4.

The poor visibility of the Kinect device results in objects moving laterally getting easily out of its view spectrum, and hence getting lost by the tracking algorithm, while they remain visible for the Hokuyo laser. On the other hand, distant object are out of range for the latter.

Thus, to enable a robust tracking even with challenging trajectories, including curves with hairpin turns, we chose to combine two different tracking algorithms: the vision-based method presented before and another one based on the laser data only. The latter consists in a simple 2D cluster-tracking algorithm. A cluster is defined by a continuous set of points where each point is separated from its neighbors by a distance inferior to a given threshold, say 35 cm. The tracking is initialized thanks to a 3D seed point supplied to the tracker.

These two tracking algorithms run simultaneously on the robot. They publish periodically their status, and the 3D pose of the object-of-interest obtained by the tracking (as defined previously). On top of them, a *dialog* node decides what algorithm has priority, and reinitialize the other thanks to the 3D pose it returns.

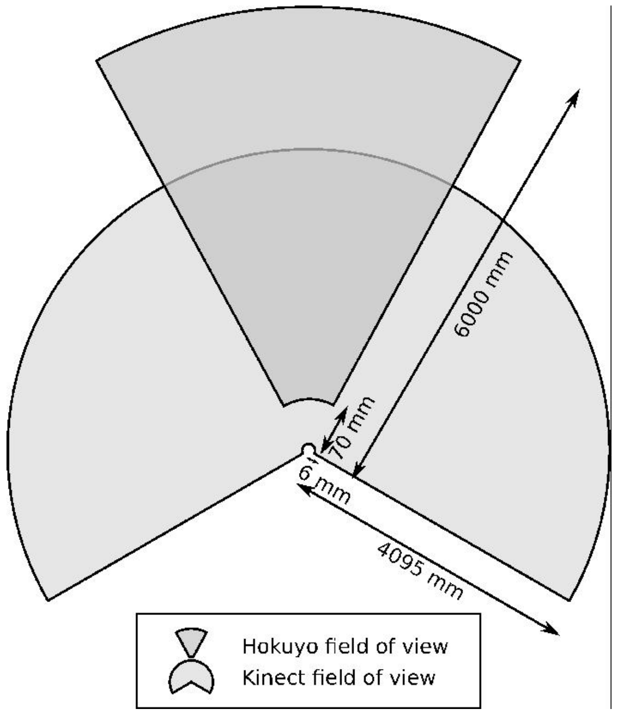


FIGURE 4. Field of view of both sensors mounted on the robot: the Kinect depth camera and the Hokuyo range finder.

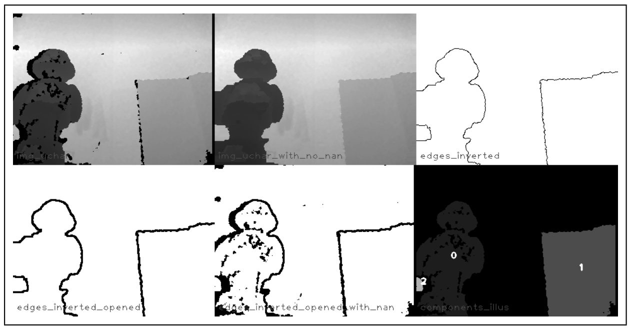
When it performs successfully, the vision-based algorithm presented in this paper always has priority on the laser-based one. As such, if the tracked object is located in the view frustum of the Kinect device, the former is used before the latter. The latter enables the tracking to go on successfully outside of the view frustum, and recovering from eventual failures of the vision-based tracking algorithm.

4.6.2. *Goal navigation.* The dialog node republishes the resulting 3D pose of the tracking algorithm that has priority. This is used as a *goal* for the motion planning system, that is the point to reach by the robot. This goal is moved after each iteration of the tracking algorithms. This corresponds to each acquisition of a depth map by the range-imaging device.

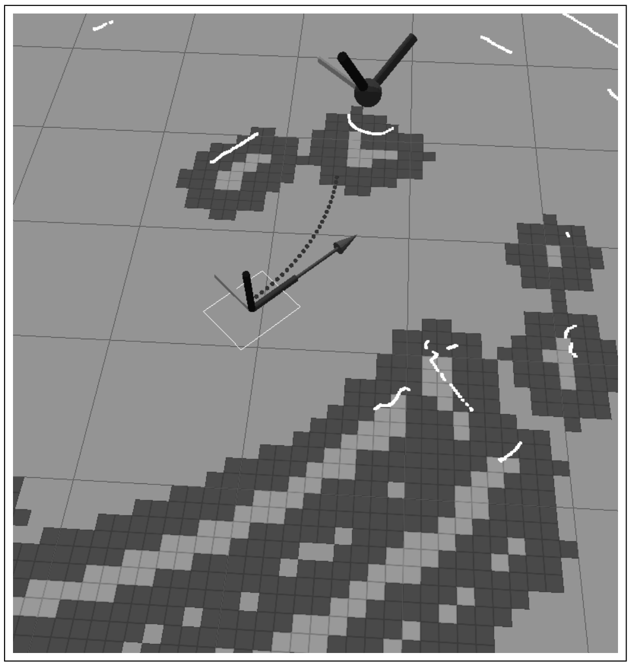
The motion control is made through a Dynamic Window Approach, as presented in [7].

The admissible velocities, that is, the ones where the robot is then able to stop without collision, are determined with the local costmap. This local costmap is obtained via the fusion of the laser scanning range finder and the reprojected Kinect point cloud.

When the robot is close enough to the goal, the robot keeps steady till the goal moves again. An example is visible in Figure 5.



(A)



(B)

FIGURE 5. Motion planning towards the goal at a given time.
(a): The detection status at a given time. The tracked object is the shape with a 0 index on the left of the bottom-right image.
(b): Visualization of the plan of the robot. The white spots correspond to the laser scan. The light gray squares represent the local costmap, and the dark ones to its inflated version (inflation by a radius corresponding to the one of the robot). The robot is indicated by its white footprint and its frame axes. The goal is the sphere with the tilted axes on top of the image. The direction indicated by the arrow corresponds to the orientation of the Kinect within the robot. The dotted curved line corresponds to the planned trajectory until the goal.

## 5. EXPERIMENTAL RESULTS

The method explained before has been implemented into the robot MOPI. The experiments have been made in several phases. First subsection 5.1 the performance of the depth map compression is evaluated. Then, the GUI developed to select the object-of-interest in subsection 5.2 is presented. The times needed for the algorithm to run in several hardware platforms in subsection 5.3 is presented and discussed. After that, how the workload can be distributed between several computers is explained in subsection 5.4. In subsection 5.5, a discussion about the advantages and limitations of both inpainting methods seen in subsubsection 4.4.1 is hold. And finally, in subsection 5.6, the accuracy of the whole tracking algorithm is measured. The success rate of the process is measured, while the user goes across a marked path with obstacles and occlusions.

5.1. **Depth image compression performance.** In subsection 4.3, the compression of depth maps for processing on a remote computer was explained. Some experimental results for the compression of depth maps are visible in Figure 6. They have been obtained during one of the experimental runs, when chasing the user.

The bandwidth required by the Kinect depth map topic typically drops from 9-10 MB/s to less than 300 kB/s. As seen on Figure 6, the ratio of the losslessy compressed image to the original one is under 10%, that is the image size is reduced more than ten times. The lossy algorithm requires the storage of the pixel indices with *NaN* value. As such, even if the image itself is smaller than with the lossless algorithm, the amount of data to transmit is most of the time bigger.

With a lossless algorithm, the compression generates an average relative error under 1% for values in a range of [1..10], which correspond to typical values of the Kinect depth map. The results in Figure 6 show an average error on that run under 0.5% for the lossless algorithm. The lossy algorithm generates two successive approximations for the depth map: first the remapping to [0..255] values, then the information loss generated by the `libjpeg` compression algorithm. As such, its error rate is higher.

The compression times presented in Figure 6 were obtained with an AMD Athlon 64 X2 Dual Core Processor 5200+. With the embodied computer of the robot, the compression of a 1-channel $320 \times 240$ image requires around 15 ms. This enables a 30Hz broadcasting (then needing a 30% CPU more or less).

Some images of input, output, error values are visible in Figure 7.

5.2. **Selection of the object-of-interest thanks to a GUI.** A GUI interface was developed for the touch screen. It enables the user to select the object-of-interest by clicking on it. A sample is visible in Figure 8.

5.3. **Time costs for clusters detection and tracking.** The tracking algorithm has been tried in several hardware architectures. It was first used on the on-board computer of the MOPI robot, which is an embedded computer with a CPU with limited capabilities (Intel Atom CPU Z530 @ 1.60Ghz ). In a second configuration, everything was processed on the touch-screen, which is eight years old (Intel Pentium M @ 1000Mhz). And finally, a desktop PC with a full-power processor was used (AMD Athlon 64 X2 Dual Core Processor 5200+ ).

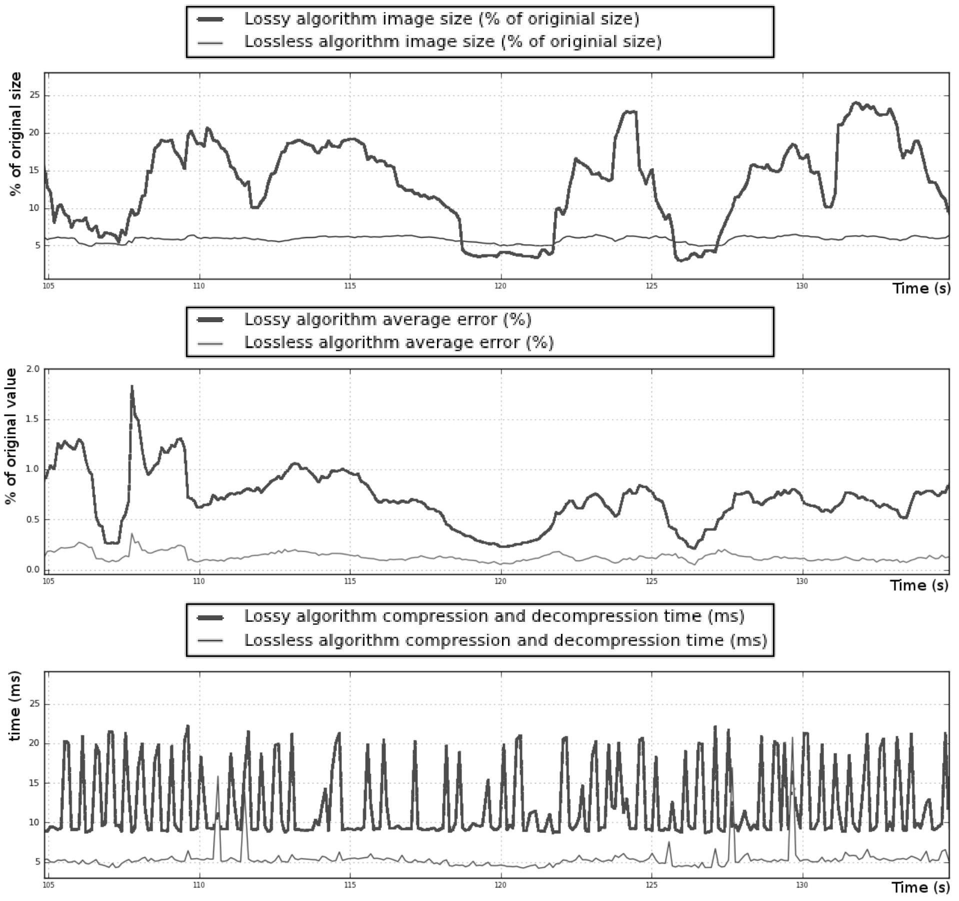
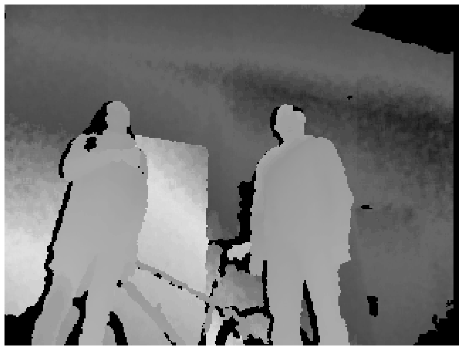


FIGURE 6. Compression ratio, average error per pixel and compression time for both lossless (`libpng`, compression factor of 6) and lossy (`libjpeg`, JPEG quality of 85) compressions. All the curves are here plotted against time during a typical tracking phase in an indoor environment.

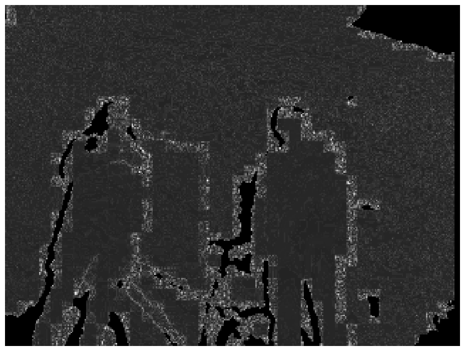
The time needed for running the algorithm is shown in Figure 9. The on-board computer needs around 50 milliseconds for a performing a cycle of the algorithm. As such, it can run up to to 20 Hz. However, it represents a workload for the CPU with limited capabilities. The use of a remote computer for processing solves this issue.

It would also be possible to run the algorithm in the touch screen, reducing the number of devices to two. However, it is preferable having a smooth GUI running at high frequency to a choppy display that might cause trouble to the user when selecting the object-of-interest of her choice. This would also reduce the battery autonomy of the touch screen.

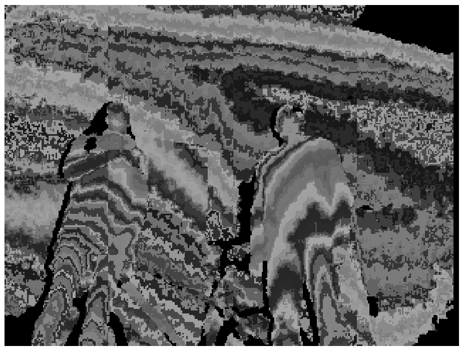
5.4. **Workload distribution among several computers.** It is possible to make use of the communication strengths of ROS: several computers can share the same



(A)



(B)



(C)

Figure 7. Error visualization for data compression. No visible difference is appreciable for the human eye between the original image and the ones after compression and decompression.
Each pixel corresponds to the error per pixel, in percents of the original value at that pixel. A black pixel corresponds to zero error, a white one to the scale indicated in caption. The average error (over the whole frame) is 0.11% for PNG compression, vs 0.65% for JPG. The maximum error, reached at one pixel, is 0.41% and 18.81% respectively.
(a): The original depth map without compression, as supplied by the Kinect device.
(b): Visualization of the compression loss per pixel for the lossy compression. A white pixel is a relative error of 18% to the original value.
(c): Visualization of the compression loss per pixel for the lossless compression. A white pixel is a relative error of 1.8% to the original value. Note the scale makes errors thirty times (30×) more visible than the other image using lossy compression.

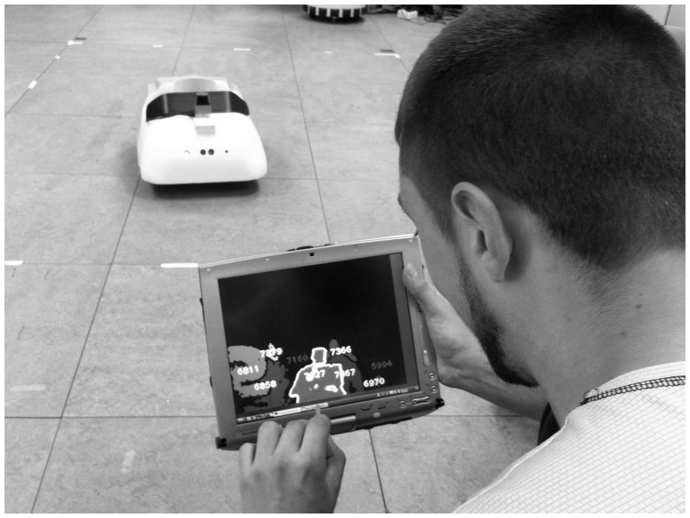


FIGURE 8. A sample picture of a user selecting the object-of-interest . It is indicated by the shape with a white border. The other objects are drawn with a different color.

data and the different subtasks can be distributed between them, without affecting the result.

As such, it is possible to send via a wireless connection the raw depth maps to a remote computer. It will run the algorithm presented previously and return the 3D pose of the object if found.

The whole architecture for processing is then structured as presented in Figure 10. The task demanding much CPU is the detection of the clusters. It has been moved to a remote computer with a faster processor. The results of the processing are sent back to the robot embedded computer.

5.5. **Comparison of the different methods for the undefined values handling.** Experimental results for methods presented in subsubsection 4.4.1 are presented in Figure 11.

- The average border removal presents the advantage of being extremely fast while removing most of the undefined values located near the images borders. However, it does not solve in any way the undefined zones in the middle of the depth maps, which can lead to objects cut in two.
- In the inpainting algorithm, the output colors in the undefined zones are obtained by propagation of the values from all along the damaged area perimeter. As such, the damaged area turns out to be a soft blending from these values. It especially smooths the strong contrasts that might occur from one side of the damaged area to the other.
- On the other hand, in the directional value propagation, the same value is propagated. Hence, the strong edges are maintained. Furthermore, it is computationally less expensive than normal inpainting.

These reasons justify the choice of the directional value propagation in the final algorithm.

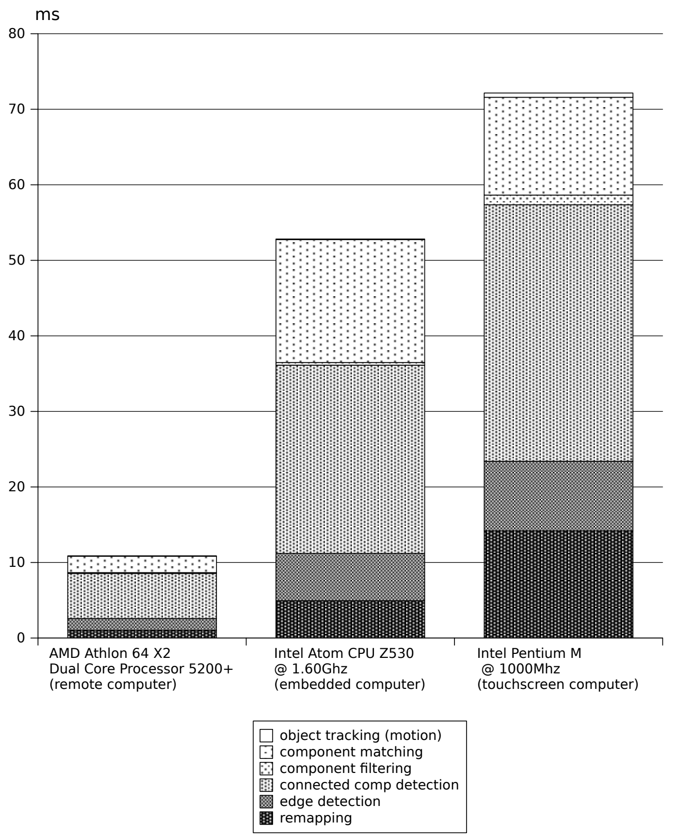


FIGURE 9. Time needs for running the algorithm on different hardware platforms. The time allocated for each step is also indicated.

5.6. **Tracking accuracy on a complicated path.** The multi-sensor fusion has been presented in subsection 4.6. It gives a high priority to the presented vision-based algorithm, and uses a laser-based fallback tracking to enable a successful tracking along a complicated path.

We measured the performance of the tracking system on a complicated path, here an unstructured lab environment. The path followed by the user is visible in Figure 12. The length of the whole path that the robot has to follow, from its start position to the finish circle, is around 33 meters. It includes straight lines, hairpin turns, and narrow passages.

A user initializes the tracking on him thanks to the GUI presented in subsection 5.2, then follows the path without giving more orders to the robot, until reaching the finish line. A run is declared as *successful* if the robot follows the user along the whole path, does not collide with any obstacle, and reaches the finish line. The experience is repeated for twenty runs.

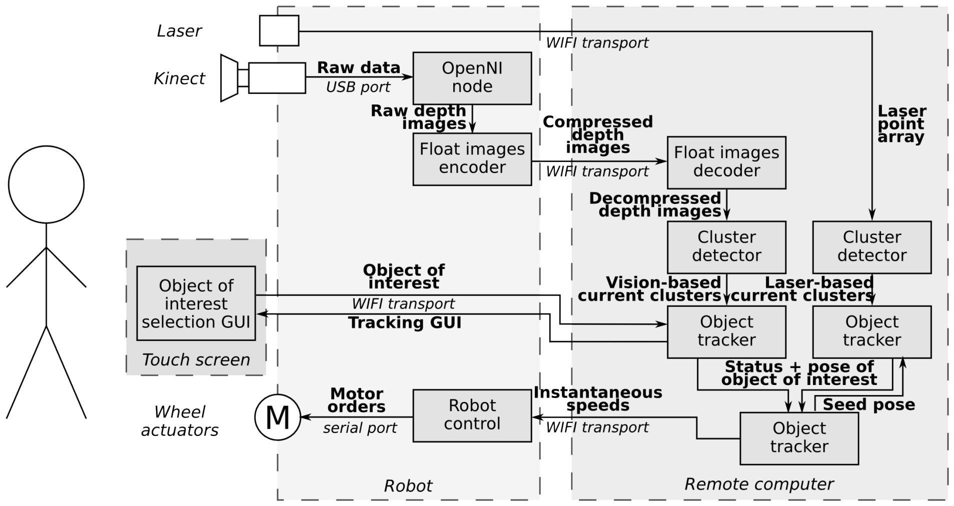


Figure 10. The flow chart of the whole sytem when distributed between several computers. It also includes the laser data fusion presented in subsection 4.6.

The results are presented in Table 1. Some screenshots of the GUI, obtained during a tracking sequence, are visible in Figure 13 and Figure 14. [2] 90 % of the runs have finished successfully, in an average time under two minutes. The cases of failure are most often due to a wrong match of the user's shape from one frame to the another. For instance, during one of the failed runs, the robot incorrectly recognized a passive observer as the tracked user, and started tracking him. A standard deviation of more than ten seconds can be noticed. The time needed to go along the whole path depends indeed on the speed of the user, which may vary from one run to another.

This high success rate experimentally validates the robustness and usability of the developed tracking system.

| | |
|---|---|
| Number of runs | 20 |
| Success rate | 90 % |
| Average time for successful runs | 114 s |
| Standard deviation of average time | 12.8 s |

Table 1. The result of the tracking runs along the complicated path.

## 6. CONCLUSIONS AND FUTURE WORKS

In this paper, we have presented a lightweight algorithm aimed at detecting and tracking 2D clusters using depth maps. This was applied to a mobile robot for the chase of a selected object-of-interest . A robust tracking system was built,

[2] The full output of the algorithm, is visible in an on-line video here. It lasts about two hours.

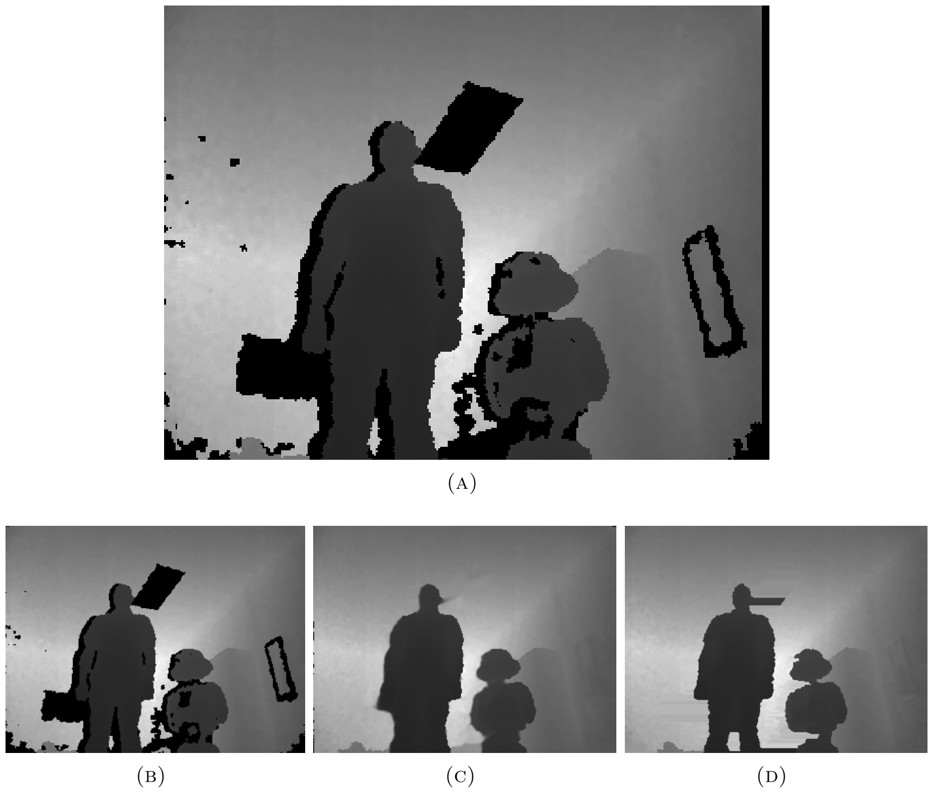


Figure 11. The different methods for undefined *NaN* values handling. The difference between methods is especially visible on the neon lamp, on the right of the user's head.
(a): The original image.
(b): *NaN* reduction thanks to average border removal. The image is identical except it is slightly more narrow. This is due to the removal of the right undefined border. Note the *NaN* values inside this border are not removed.
(c): *NaN* removal thanks to inpainting.
(d): *NaN* removal thanks to left-value propagation.

using a laser fallback when the selected object-of-interest remains out of the narrow field-of-view of the range-imaging camera.

In a near future, more extended measurement and user experiments is carried out. We especially want to measure the object-of-interest pose measurement error. This could be obtained by precisely measuring the position of the user thanks to a camera viewing from above, and matching it against the pose computed by the algorithm.

Furthermore, the RGB information given by the Kinect device was discarded here. We aim at experimenting some of the methods presented in the articles of the introduction and measure the improvement obtained by coupling both methods.

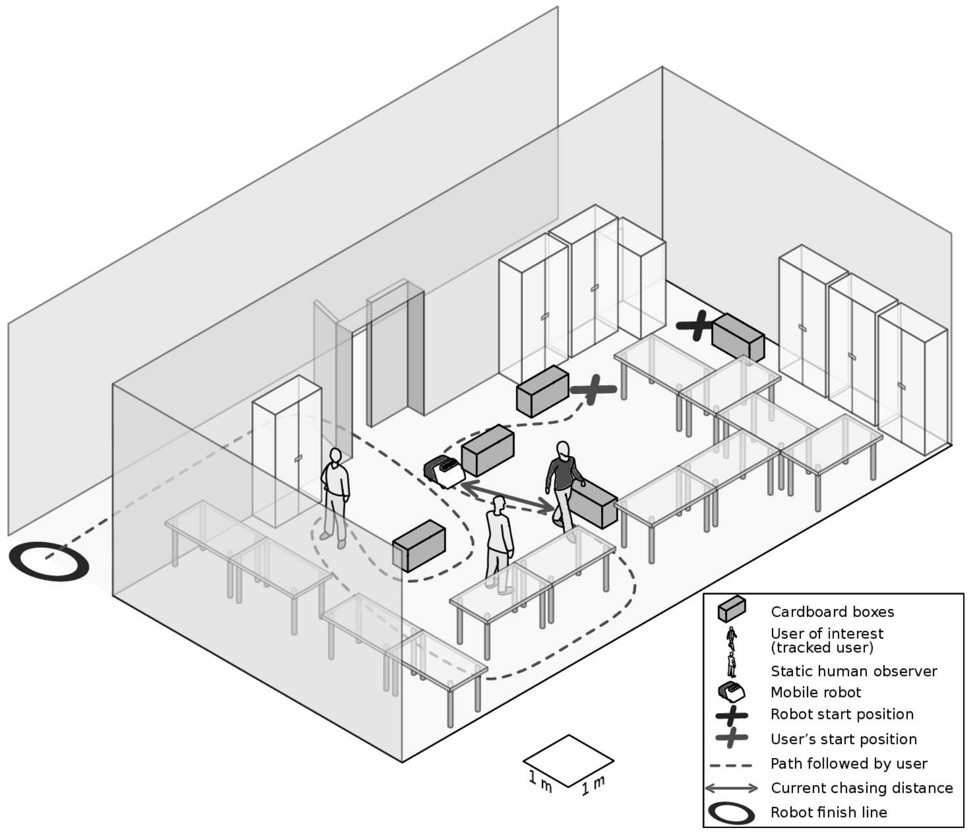


FIGURE 12. The path followed by the user to test the tracking accuracy. The gray boxes correspond to cardboard boxes aimed at making the path planning harder and preventing the robot from cutting the curves.
The robot intends to maintain the chasing distance as close as possible to the goal distance. When the robot enters the circle without losing the track of the user, the test is marked as successful.

## References

1. R Barber and MA Salichs, *A new human based architecture for intelligent autonomous robots*, Proceedings of The 4th IFAC Symposium on Intelligent Autonomous Vehicles, Elsevier, 2002, pp. 85–90.
2. T Berliner and Z Hendel, *Modeling Of Humanoid Forms From Depth Maps*, United States Patent Application 20100034457 (2007).
3. M Bertalmio and G Sapiro, *Image inpainting*, SIGGRAPH '00 Proceedings of the 27th annual conference on Computer graphics and interactive techniques (2000).
4. J Canny, *A computational approach to edge detection*, Pattern Analysis and Machine Intelligence, IEEE Transactions on (1986).
5. Seongju Chang, Sungil Ham, and Dongjun Suh, *ROHINI: A robotic flower system for intuitive smart home interface*, Control Automation and Systems (ICCAS), 2010 International Conference on, 2010, pp. 1773–1776.

6. M.-P. Dubuisson and A K Jain, *A modified Hausdorff distance for object matching*, Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision Image Processing., Proceedings of the 12th IAPR International Conference on, vol. 1, October 1994, pp. 566 –568 vol.1.
7. D. Fox, W. Burgard, and S. Thrun, *The dynamic window approach to collision avoidance*, Robotics & Automation Magazine, IEEE **4** (1997), no. 1, 23–33.
8. Bernard A Galler and Michael J Fisher, *An improved equivalence algorithm*, Commun. ACM **7** (1964), no. 5, 301–303.
9. B. Gerkey, R.T. Vaughan, and A. Howard, *The player/stage project: Tools for multi-robot and distributed sensor systems*, Proceedings of the 11th international conference on advanced robotics, Portugal, 2003, pp. 317–323.
10. G Grisettiyz, *Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling*, Robotics, IEEE Transactions on **23** (2005), no. 1, 34–46.
11. A Howard and LH Matthies, *Detecting pedestrians with stereo vision: safe operation of autonomous ground vehicles in dynamic environments*, Proceedings of the 13th Int. Symp. of Robotics Research (2007).
12. Songmin Jia, Liang Zhao, Xiuzhi Li, Wei Cui, and Jinbo Sheng, *Autonomous robot human detecting and tracking based on stereo vision*, 2011 IEEE International Conference on Mechatronics and Automation, IEEE, August 2011, pp. 640–645.
13. SG Latta and K Tsunoda, *Gesture keyboarding*, US Patent App. 12/391,145 (2009).
14. CH Lee, *An intelligent depth-based obstacle detection system for visually-impaired aid applications*, Image Analysis for Multimedia Interactive Services (WIAMIS), 2012 13th International Workshop on (2012).
15. R Muñoz Salinas, *People detection and tracking using stereo vision and color*, Image and Vision Computing **25** (2007), no. 6.
16. R Muñoz Salinas and E Aguirre, *Multi-agent system for people detection and tracking using stereo vision in mobile robots*, Robotica (2009).
17. M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, *ROS: an open-source Robot Operating System*, ICRA Workshop on Open Source Software, 2009.
18. Arnaud Ramey, *Playzones : A robust detector of game boards for playing visual games with robots*, Robot 2011 - III Workshop de Robótica : Robótica Experimental (Sevilla), 2011, p. 8.
19. ______, *Playzones: a robust detector of game boards and its application for games with robots*, Master thesis, University Carlos III of Madrid, 2012.
20. R Rivas, *Robot skill abstraction for ad architecture*, 6th IFAC Symposium on Intelligent Autonomous Vehicles **47** (2007), no. 4, 12–13.
21. C Rother, V Kolmogorov, and A Blake, *Grabcut: Interactive foreground extraction using iterated graph cuts*, ACM Transactions on Graphics (TOG) (2004).
22. Z Tomori, *Active Segmentation in 3D using Kinect Sensor*, 20th WSCG International Conference on Computer Graphics, Visualization and Computer Vision (2012).

Robotics Lab, Univ. Carlos III of Madrid, Leganés, Spain
*E-mail address*: arnaud.ramey@m4x.org

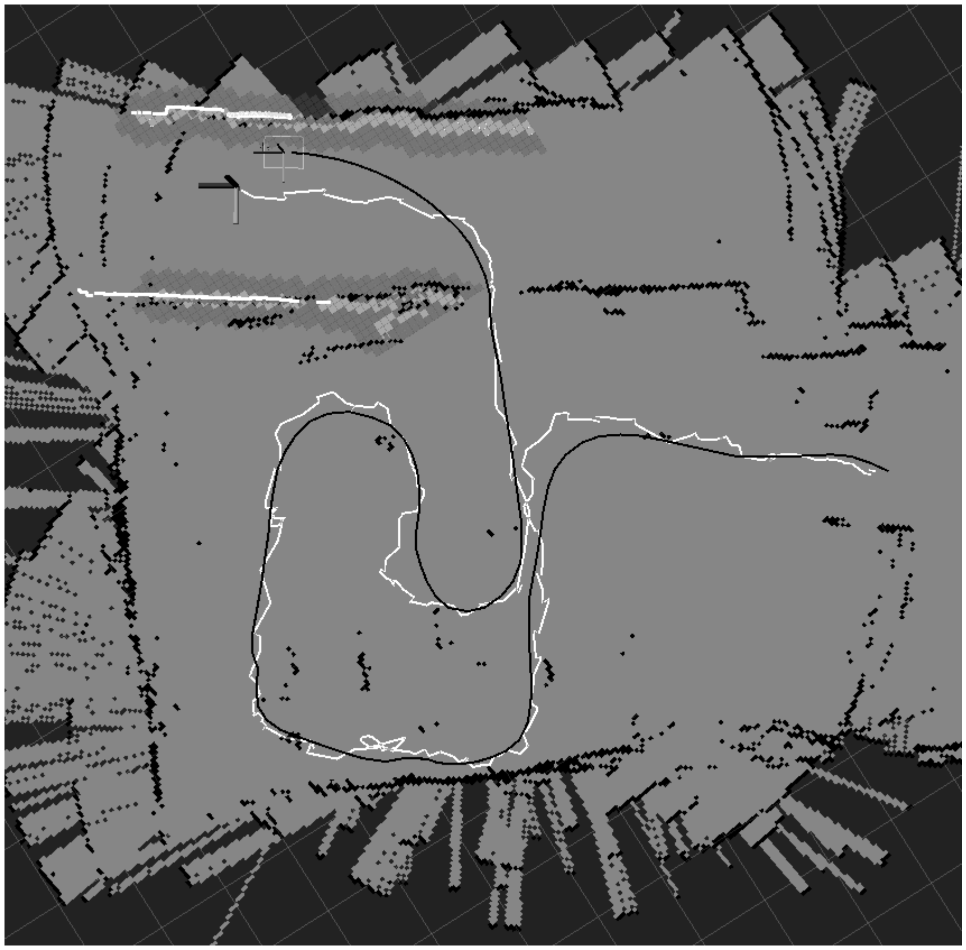


FIGURE 13. Map generated by a SLAM algorithm (GMapping [10]) overlaid with the paths generated during a run.
The light gray area corresponds to the free space of the map and the black edges its occupied space. The irregular line is the path of the user, as detected by the algorithm.
The smooth line ending into the rectangular shape corresponds to the path of the robot, as determined by its odometry.
The remaining symbols are identical to the ones used in Figure 5 (b).

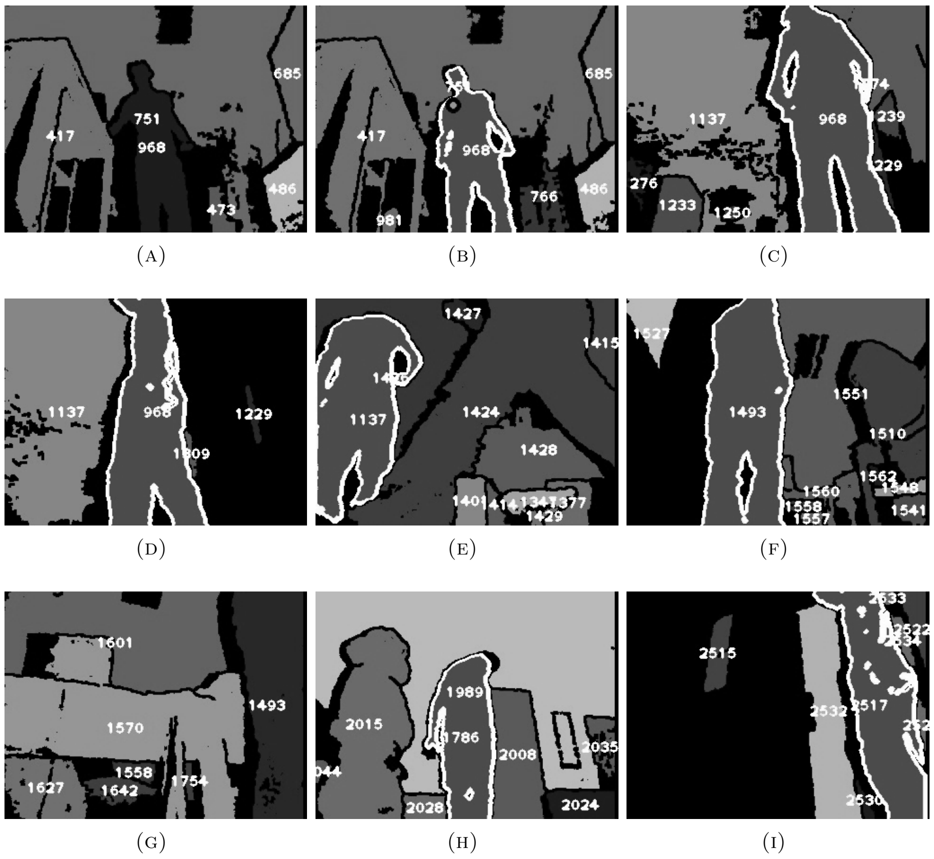


Figure 14. Different frames of the GUI during a tracking sequence on the test path. The numbers stand for the objects names. The object-of-interest has a white stroke.
(a): GUI just before the user initialization, and (b), just after. The selected user then is marked with a white stroke (cluster 968).
(c): Shortly after initialization, the robot has come next to the user (cluster 968), who already started walking along the path.
(d): The user starts walking in a transverse direction to the robot motion, which triggers a fast on-place rotation.
(e): Another user (cluster 1424) crosses the path of the robot. The tracking is not affected.
(f): In the narrow passage between the tables (cluster 1510) and the wall.
(g): The vision-based tracking loses track of the user (cluster 1493) after a sharp turn. The tracking goes on thanks to the laser-based tracking.
(h): After the sharp S-turns at the end, the final door is visible (cluster 2008).
(i): Approaching the final door (cluster 2532).