Pattern Recognition 60 (2016) 405–419

Contents lists available at ScienceDirect

Pattern Recognition

journal homepage: www.elsevier.com/locate/pr

ZebraRecognizer: Pedestrian crossing recognition for people with visual impairment or blindness

Sergio Mascetti [a,b,*], Dragan Ahmetovic [a], Andrea Gerino [a,b], Cristian Bernareggi [a,b]

[a] *Università degli Studi di Milano, Department of Computer Science, Via Comelico 39, 20135 Milan, Italy*
[b] *EveryWare Technologies, Via Comelico 39, 20135 Milan, Italy*

ARTICLE INFO

Article history:
Received 29 May 2015
Received in revised form
9 November 2015
Accepted 9 May 2016
Available online 22 May 2016

Keywords:
Visual impairment
Blindness
Pedestrian crossing
Mobile computing

ABSTRACT

Independent mobility is a challenge for people with visual impairment or blindness. Groundbreaking innovation comes from mobile devices (e.g., smartphones) that are convenient platforms to provide assistive technologies in the form of mobile applications.

This paper presents *ZebraRecognizer*, a software module that recognizes zebra crossings and that advances state-of-the-art along two directions. First, it removes projection distortion from the acquired image, hence improving the accuracy of the recognition and making it possible to compute the quantified relative position of the crossing with respect to the user, which is crucial to effectively guide the user. Second, *ZebraRecognizer* is efficient, as it adopts a customized version of the EDLines algorithm that is also implemented to run in parallel on the GPU. Experimental results show that *ZebraRecognizer* is accurate, efficient and it computes the crossings position precisely.

1. Introduction

In the past years mobile devices became accessible to people with visual impairment or blindness.[1] This makes it possible to develop mobile applications specifically designed for this class of users and indeed these apps are already available on online stores. For example, there are applications that support independent mobility in urban environments by reading aloud the current address and nearby points of interest. Other solutions that have been proposed in the scientific literature rely on more involved techniques to extract information from the environment and provide it to the user. In particular, in this paper we consider the problem of detecting pedestrian crossings from the images captured with the mobile device camera.

There are a number of challenges involved with the identification of pedestrian crossings. First, given the hazards inherently connected with road crossing, it is crucial to have no false positives, i.e., to erroneously recognize a crossing in an image that actually contains none. At the same time, in order to guarantee an effective solution, most pedestrian crossings should be properly identified. Second, it is necessary to precisely compute the relative position between the user and the pedestrian crossing.[2] Third, since the application should be responsive, the identification process should have a low execution time.

This paper describes a software module called *ZebraRecognizer* that adopts an original pattern matching technique to recognize zebra crossings, a very common type of pedestrian crossings. *ZebraRecognizer* is designed to be included in a mobile application called *ZebraX* that addresses the problem of guiding a person with visual impairment or blindness to cross along a pedestrian crossing. The extensive experimental evaluation highlights three major contributions of *ZebraRecognizer* with respect to the state-of-the-art.

- *ZebraRecognizer* rectifies selected features in the input image, hence removing projection distortion. This eases the analysis of the zebra crossing pattern, thus improving the quality of the recognition in terms of precision and recall. The result is that *ZebraRecognizer* incurs in no false positives and it correctly identifies 93% of zebra crossings.
- Since the stripes composing the zebra crossing are rectified, *ZebraRecognizer* computes the relative distance with *quantified* and *precise* measures. For example, in 96% of the cases the

* Corresponding author at: Università degli Studi di Milano, Department of Computer Science, Via Comelico 39, 20135 Milan, Italy.
E-mail addresses: sergio.mascetti@unimi.it (S. Mascetti), dragan.ahmetovic@unimi.it (D. Ahmetovic), andrea.gerino@unimi.it (A. Gerino), cristian.bernareggi@unimi.it (C. Bernareggi).

[1] In case the reader is unfamiliar with accessibility tools for people with visual impairment or blindness there is a short video introducing the main ideas: http://goo.gl/mEI6Uz.

[2] In this paper "relative position" refers to the relative position between the user and the pedestrian crossing.

http://dx.doi.org/10.1016/j.patcog.2016.05.002
0031-3203/

frontal distance is computed with an error smaller than 50 cm (approximately one step) and the rotation angle (i.e., heading) is always computed with an error smaller than 10°.
- *ZebraRecognizer* has been specifically engineered to be used on mobile devices and, in particular, the most expensive operations are computed in parallel on the smartphone GPU. As a result, on an iPhone 5S, *ZebraX* can process about 25 frames per second.

Note that, in addition to zebra crossing recognition, there are other challenges arising in the design and development of an application that guides people with visual impairment or blindness to cross over pedestrian crossings. Among others, there is the problem of computing a safe path to cross and the design of effective audio instructions. We are currently investigating these topics. However, they are out of the scope of this paper.

This paper extends a previous conference version [1] along a number of directions.

- This paper improves all main steps of the recognition algorithm:
  - ground plane rectification is now computed with a totally different and robust technique derived from the solution proposed by Lefler et al. [2] (see Section 3.1);
  - line segments merging is applied on rectified elements, rather than on the original image (see Section 3.3);
  - during line segments computation, orthogonal regression is adopted instead of least squares line fitting (see Section 3.3);
  - new version also adopts "vertical distance" criterion during line segments grouping (see Section 4.1);
- The new solution defines distance measurements that can be used by the other *ZebraX* modules to effectively guide the user. The computation of these measurements is not trivial and requires in-depth changes in the recognition process itself (see Section 4.3).
- The solution presented in this paper has been optimized and engineered and this positively impacts on accuracy and computational costs. In particular, this paper describes how to run the most expensive operations of the algorithm on the mobile device GPU (see Section 3.4).
- This paper presents new experimental results, including a detailed evaluation of the precision of the computed relative distance, showing that the proposed technique is not only accurate in terms of precision and recall, but also precise is computing the position of the zebra crossing (see Section 5).
- This paper extends the technical description of the solution, including two formal results with corresponding proofs (see Section 3, Appendix A and Appendix B).
- This paper improves presentation, including a description of *ZebraX* modules, extended discussion of related work, adds new examples and more than 20 figures.

The paper is organized as follows. Section 2 describes the background of *ZebraRecognizer*: related work, its role in *ZebraX* and the details of the pattern matching problem it addresses. The technical solution is described by showing how *ZebraRecognizer* extracts the features from the input image (Section 3) and processes them (Section 4). The results of the experimental evaluation are presented in Section 5, while Section 6 concludes the paper.

2. Background

2.1. Related work

To support independent mobility for people with visual impairment or blindness, some applications available on the market convey to the user information extracted from available data sources. For example, iMove[3] is an iOS application that informs the user about the current address and about nearby points of interest (e.g., shops, bus stops, etc.).

A more involved approach consists in extracting the contextual information from the analysis on images coming from the device camera, possibly combining this information with data obtained from other sensors, like accelerometer, for example. This section presents some of the contributions appearing in scientific literature that follow this approach to recognize pedestrian crossings.

The first solution proposed in literature [3] detects pedestrian crossings with the following approach: first, line segments and their vanishing points are detected through Hough line segment detector. Then, outliers are filtered out by a Random Sample Consensus algorithm. The result, expected to be a set of line segments belonging to the same zebra crossing, is then validated using cross ratio constraint. This solution was validated on some sample images, but no extensive evaluation on a large set of images was conducted. Moreover, no experimental evaluation with people with visual impairment or blindness was undertaken.

A different approach [4] first applies a bipolarity segmentation to detect areas of alternating black and white stripes and then validates the result through cross ratio invariant verification. This solution yields good results in terms of precision and recall, although the experimental evaluation has been conducted on a small data set (about 100 images), all with similar illumination conditions. Vice versa, our solution has been validated with a much larger set, captured with different illumination conditions including direct sunlight, night and cloudy weather.

Ivanchenko et al. propose two techniques for detecting pedestrian crossings. One solution detects zebra crossings but does not compute their relative position with respect to the user, which is a necessary step to guide the user towards the crossing [5]. The second solution detects the "two stripes" pedestrian crossings and adopts a rectification technique that, while not described in detail, appears to be similar to the one proposed in this contribution [6].

Murali et al. present an innovative approach to estimate the user's position in an intersection [7]. The idea is to acquire 360° image panoramas while turning in place on a sidewalk. The image panorama is then converted to an aerial (overhead) view of the nearby intersection, centered in the user's location. The goal is to match this aerial view with a template of the intersection obtained from a satellite image. The matching process allows crosswalk features to be detected and permits the estimation of the user's location in the intersection. The main difference is that the aim of our solution is to continuously detect the relative position of the crossing in order to guide the user. Vice versa the solution proposed by Murali et al. can identify the user's absolute position but it requires the acquisition of a complete panorama and hence it cannot be used while the user is crossing.

Our previous solution for the recognition of zebra crossings focuses on the computation of the instructions to guide the user (e.g., "shift left", "rotate right") [8]. This solution processes the projected image captured by the camera and this introduces non-negligible approximation in the evaluation of the relative position.

Finally, our recent contribution [9] adopts a solution similar to the one proposed in this work to recognize pedestrian crossings from geo-referenced online images (e.g., satellite images and "Google street view" images). The aim is to acquire information about the presence and position of zebra crossings that are not currently in the camera field of view.

[3] At the time of writing iMove is available for free download at https://itunes.apple.com/en/app/imove/id593874954.

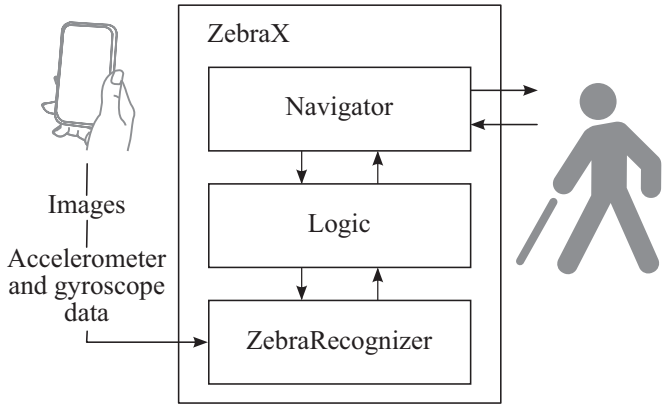


Fig. 1. Modules of the *ZebraX* application.

2.2. System modules

The *ZebraX* application is divided into three main modules, as depicted in Fig. 1.

The *Navigator* module is in charge of conveying audio instructions to guide the user towards and along the zebra crossing. The main challenge is that the user needs to be continuously informed about his/her position with respect to the zebra crossing. However a person with visual impairment or blindness should not be overwhelmed with too many audio messages, because they can divert the attention from the surrounding audio scenario, which is essential to acquire indispensable information (e.g., an approaching car, a person walking by, etc.). Ullman et al. remark that blind people run into difficulty while being guided by verbose speech messages [10].

The instructions that the *Navigator* module conveys to the user are computed by the *Logic* module. In this case the challenge is to generate a short and safe path in which the zebra crossing is always in the camera field of view.

To compute these guidance instructions the *Logic* module takes in input the relative position of the crossing.

This paper focuses on the procedure to recognize the zebra crossings and to compute their relative position. This computation is run by the *ZebraRecognizer* module which is internally divided into 6 steps, as shown in Fig. 2. The first three steps (i.e., rectification matrix computation, image pre-processing and line segments detection) are all aimed at extracting the line segments that represent the stripes (see Section 3). In the last three steps (i.e., line segments grouping, zebra crossing validation, final result computation) the line segments are processed and relative position is computed (see Section 4).

2.3. The pattern matching problem

The recognition technique described in this paper has been tuned for detecting zebra crossings as defined by Italian traffic regulations (see Fig. 3a), but it can be easily adapted to most definitions used worldwide. For example, given the similarity between Italian zebra crossings (Fig. 3b) and the US version (see Fig. 3c), it is possible to reconfigure *ZebraRecognizer* to recognize the US zebra crossings with very limited effort, by setting and retuning the detection parameters. Clearly, the solution does not directly apply to other types of pedestrian crossings, like the "two lines crossings" (see Fig. 3d). Still, the adopted methodology can be used to design similar solutions for other pedestrian crossings or other kinds of geometrically well-known horizontal traffic signs.

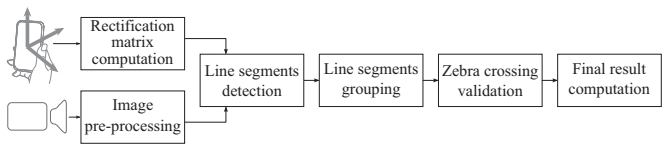


Fig. 2. *ZebraRecognizer* flowchart.

A zebra crossing is described as a horizontal traffic sign consisting of an alternating pattern of dark and light stripes (see Fig. 3a). It is composed of at least 2 light stripes and 1 dark stripe. The stripes are commonly rectangular and, less frequently, in case of diagonal crossings, parallelograms. They are 50 cm thick and have a width of at least 250 cm. The dark stripes are of the same color of the underlying road while the light stripes may be white or, in case of road works, yellow.

The recognition process is entirely computed locally on the mobile device because the responsiveness requirements of *ZebraX* makes it impractical to have a remote computation due to network latency. For the detection of zebra crossings, *ZebraRecognizer* relies on data sources available on off-the-shelf smartphones: video camera, accelerometer and gyroscope. The first captures image frames that can then be analyzed with computer vision techniques in order to detect zebra crossings, if present. Accelerometer and gyroscope, instead, can be used to extract the orientation of the device with respect to the ground plane and the detected crossings.

Technically, the input of *ZebraRecognizer* consists of user's height h_u, an image i with height i_h and width i_w and the gravity acceleration data represented as a three dimensional unit vector $a = \langle a_x, a_y, a_z \rangle$. Its elements a_x,a_y and a_z are measured in g=9.80665 m/s^2, take values in $[-1, 1]$ and represent, respectively, the portion of the gravity that is applied on the device x-, y- and z-axes (see Fig. 4a).

The output of the algorithm is the most suitable detected zebra crossing, if any. It is characterized by a list of stripes, each one defined by its top and bottom line segments and its color (i.e., black or white). We represent the position of each line segment both in the source image (e.g., Fig. 5a) and on the rectified ground plane (e.g., Fig. 5b). The result also includes four compact and easy-to-use distance measurements (see Fig. 5c): frontal distance, rotation angle, lateral distances from the left and right borders of the crossing.

3. Features extraction

The features that *ZebraRecognizer* uses to detect a zebra crossing in an image are the line segments representing the long edges of the stripes. They are recognized with a customized version of the EDLines algorithm, originally proposed by Akinlar et al. [11]. The modified version of this algorithm requires the knowledge of the horizon line in the image. Property 1 defines how to compute the horizon line equation (proof is in Appendix A).

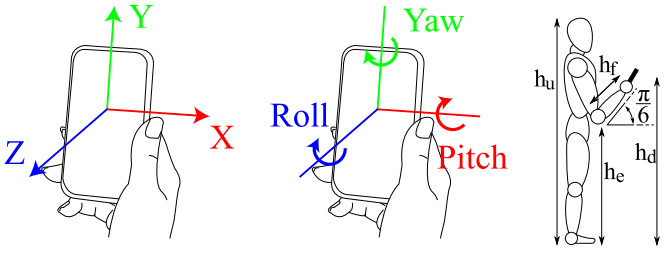
Property 1. *Let ρ and θ be the device pitch and roll angles respectively, $C = \langle C_x, C_y \rangle$ is the center of the image and f is the focal distance of the camera (in pixels). Then, the equation of the horizon line h inside the acquired image is*

$$\sin(\theta)x + \cos(\theta)y - \sin(\theta)(C_x + \tan(\rho)\sin(\theta)f) - \cos(\theta)(C_y + \tan(\rho)\cos(\theta)f) = 0 \quad (1)$$

The modified EDLines algorithm also takes in input the rectification matrix that is used to rectify the line segments being extracted (see Section 3.1). The image used by EDLines is pre-processed, as described in Section 3.2. The actual specialized version of EDLines is presented in Section 3.3 while Section 3.4 describes

(a) Zebra crossing pattern in Italy (b) Pedestrian crossing in Italy (c) Pedestrian crossing in USA (d) "Two lines crossing" in UK

Fig. 3. Examples of zebra crossings.



(a) Device axes (b) Rotations (c) Height

Fig. 4. Rotation and position of the mobile device while using *ZebraX*.

the algorithm implementation on the GPU.

3.1. Ground plane reconstruction

Planar rectification is a homography, represented by a 3×3 rectification matrix that removes the projective distortions from the image of a planar surface and returns a view of the same plane in which the camera's axis is perpendicular to the plane. For the ground plane, the rectified image is a view from directly above it, as seen in Fig. 6. Once the rectification matrix is known, it can be selectively applied to some elements (i.e., line segment end points) instead of the whole image, thus reducing the execution time. In our previous work [1] rectification matrix is obtained by using a well-known technique [12]. However, we experimentally observed that a more recent approach [2] yields better performance (mainly in terms of recall) and hence we decided to adopt it.

Note that the rectification matrix is computed for each new frame using the last available gravity data. Since on the system used for the experiments (iPhone 5S) gravity acceleration data is updated about 100 times per second, each time a new frame is received the rectification matrix is computed with values no older than 10 ms.

The application of the rectification matrix to the image yields a "rectified plane" in which the distances are proportional to those on the ground plane. More specifically, the distance between any two points on the ground plane is equal to the distance of the corresponding points on the rectified plane multiplied by a *zoom factor*. To compute the zoom factor it is necessary to know the distance between any two points in the rectified plane as well as the distance between the corresponding two points in the ground plane.

In our case, we consider two artificial points on the rectified plane that are crafted in such a way that we can derive the distance of the two corresponding points in the ground plane thanks to the knowledge of the camera position in space and camera parameters. Property 2 shows how to derive the zoom factor (proof in Appendix B).

Property 2. *Let ρ be the device pitch angle, h_d the device's height, C the center of the image and f the focal distance of the camera (in pixels). R is the rectification matrix computed previously while A_i and B_i are arbitrary points below the horizon and that lie on line vl that is perpendicular to the horizon and that passes through the image principal point C.*

Points $A_r = R \cdot A_i$ and $B_r = R \cdot B_i$ are rectified points corresponding to A_i and B_i respectively.

Then, the zoom factor z is:

$$z = \frac{h_d \cdot \left[\tan\left(\pi - \rho - a\tan\left(\frac{CB_i}{f}\right)\right) - \tan\left(\pi - \rho - a\tan\left(\frac{CA_i}{f}\right)\right)\right]}{A_r B_r} \quad (2)$$

Note that Property 2 assumes that the height h_d of the camera with respect to the ground is known. To estimate this value, *ZebraRecognizer* assumes that the user is holding the device in a position like the one depicted in Fig. 4c in which the elbow is close to the hip and the forearm has an inclination of about $\pi/6$ with respect to the ground plane. By considering the proportions of the human body [13], the device height can be derived from the user's height h_u (either estimated or asked to the user). Indeed, on average, the height at elbow is $0.615 \cdot h_u$ and the forearm length is $0.205 \cdot h_u$. Consequently, the device height from the ground is estimated as:

$$h_d = 0.615 \cdot h_u + \sin(\pi/6) \cdot 0.205 \cdot h_u \quad (3)$$

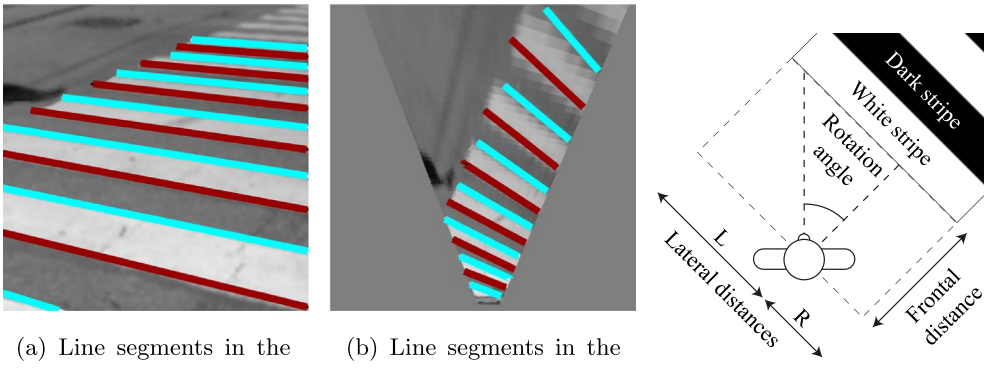
Clearly the above computation is subject to some approximation. However, the error does not practically affect navigation. For example, considering a 175 cm tall person, the technique estimates that the device is held at 125 cm from the ground. Even in the extreme case in which the device is actually kept at the height of the shoulders[4] (about 145 cm from ground), a zebra crossing at a distance of 2 m is computed as being 2.33 m from the user i.e., the error is less than an average step length.

3.2. Image pre-processing

As observed in Section 2.3, zebra crossings can be painted with different colors. Hence we are only interested in the light and dark components of the image. For this reason, we acquire grayscale images. Clearly, the use of single-channel images also helps improving the computation performance and reduces the memory footprint.

The acquired images contain many small details we are not

[4] This is an unnatural position that we never observed during experiments.



(a) Line segments in the source image.

(b) Line segments in the rectified ground plane.

(c) Relative distances.

Fig. 5. Zebra crossing identification and relative distances.

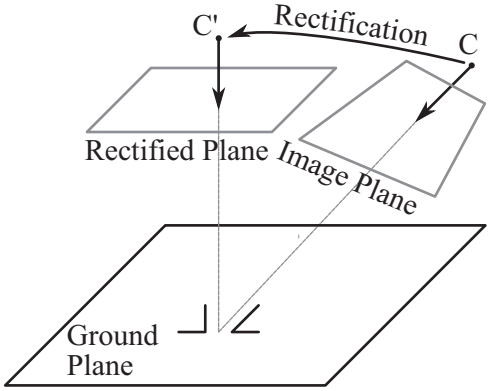


Fig. 6. Rectification homography.

interested in, such as cracks, paint imperfections, leaves and dirt. These imperfections may actually impair detection, hence we use resampling and blurring to filter them out. The first method rescales the image until small details become undetectable. Also, it reduces the image size and thus diminishes the execution time of per-pixel operations that follow. However, the size still has to be sufficient for a correct detection. As highlighted in our experiments (see Section 5), the best results of the recognition can be obtained with relatively low resolution (i.e., 180×320). *ZebraX* acquires images at this resolution. Vice versa, the images in the test-sets were recorded at the resolution of 1080×1920 and resized, with a linear interpolation filter, before running each test so that *ZebraRecognizer* can be evaluated with images at different resolutions.

Finally, a Gaussian blur filter is applied to the image. Similarly to the resampling, the aim is to filter out imperfections in the image and ease the line segments detection. Since this step reduces the number of recognized line segments, it also indirectly affects the computation performances because fewer line segments need to be processed in the following steps.

Fig. 7b shows an example of the pre-processing step applied to Fig. 7a (the portion of the image above the horizon is ignored). Henceforth with "image" we intend the result of the pre-processing step.

3.3. Line segments detection algorithm

The line segments detection step is a modified version of the EDLines algorithm [11]. The input is composed by the pre-processed image, the horizon line and the rectification matrix. The output is a set of detected segments in the rectified coordinate system. There are four main differences with respect to the original algorithm.

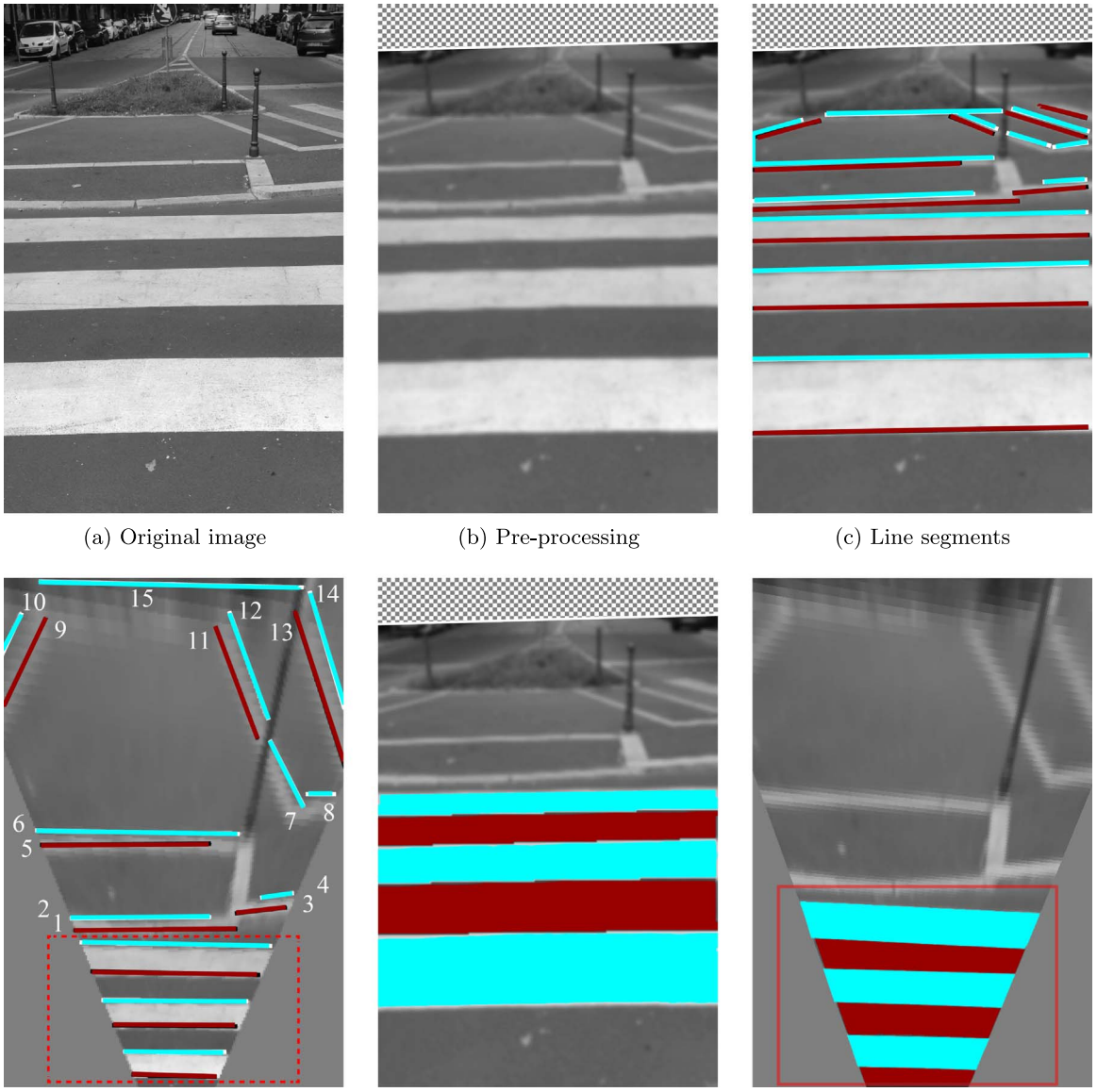
First, our technique ignores the portion of the image above the horizon since no zebra crossings will ever be found there. This approach significantly reduces the computation time for two different reasons: it speeds up the line segments detection process itself and it reduces the number of detected segments, hence reducing the computation time of successive processing steps. This solution also helps improving the recognition accuracy as it prevents false positives (i.e., a false crossing recognized above the horizon).

The second difference with respect to the original EDLines algorithm is that our solution computes additional information about the detected line segments. First, in addition to gradient direction, our solution also computes the gradient orientation of the detected segments, so, in practice, we compute the angle of the gradient in $[0, 2\pi)$ rather than in $[0, \pi)$. This information is useful in the following steps since the direction and the orientation of the gradient can differentiate between segments on the top and those on the bottom of each stripe. The second additional information computed by our version of EDLines is whether each end point of each line segment lies on the image boundary. This is useful, in the following computation, to distinguish between stripes that terminate in the end point position and those that, instead, can potentially continue but are not visible in the image.

The third difference is that our technique also merges close segments. Two segments having both slope distance and spatial distance lower than specified thresholds are merged. This is useful, for example, when two or more portions of a line segment have been recognized as different line segments due to minor imperfections in the image, noise, flawed coloration of the stripes or objects between the observer and stripes (Fig. 8 shows an example). The line segment s resulting from the merging of two line segments $s1$ and $s2$ is computed as follows: first, the lines $l1$ and $l2$ on which the two line segments lay are calculated. Then, a new line l (equation in general form: $ax + by + c = 0$) is computed with parameters a, b and c being weighted averages (based on the two segments' lengths) of the corresponding parameters of lines $l1$ and $l2$. Finally, the segment s is computed as the union of the two line segments' projections on l.

In our previous solution [1], this merging operation was computed using line segments in their representation on the image, hence subject to projection distortion. Vice versa, in our current solution, line segments are rectified before being merged.

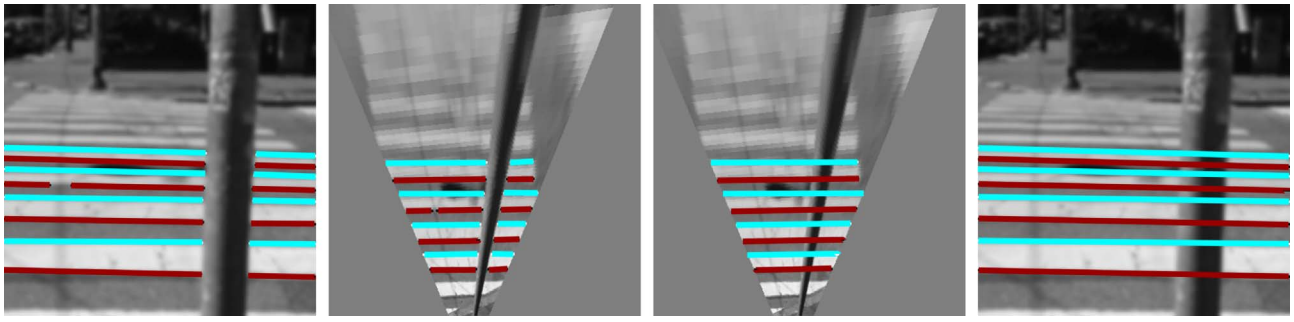
The fourth difference is that, during line segment computation, we use orthogonal regression instead of least squares line fitting for the purpose of determining the equation of the line on which each line segment lays. Orthogonal regression computes the orthogonal distance between each point and the candidate line,



(a) Original image (b) Pre-processing (c) Line segments

(d) Line segment groups (e) Crossing validation (f) Detected crossing

Fig. 7. Main steps of *ZebraRecognizer*.



(a) Split line segments (projected) (b) Split line segments (rectified) (c) Merged line segments (rectified) (d) Merged line segments (projected)

Fig. 8. Example of line segments merging.

differently from the line fitting algorithm that computes the vertical distance. Orthogonal regression is needed in our case since we are also interested in vertical line segments.

As a final step, after merging line segments, we prune the segments that are too short to possibly represent a stripe edge. Fig. 7c shows an example of application of our personalized version of EDLines.

3.4. GPU computation of line segments detection

While our implementation of EDLines has been highly optimized, it is still the most expensive operation of *ZebraRecognizer* and it takes about 45% of the entire computation time. The reason is that three operations required by EDLines have a time complexity linear in the number of pixels in the image. These three operations consist in the computation of gradient magnitude, gradient direction and anchors. Since the aim of these three operations is to extract the so-called "anchors", in this paper we globally refer to them as "anchors extraction".

To reduce the computation time of "anchors extraction", we implemented it through two fragment shaders, so that the computation can be run by the GPU highly parallel architecture. Indeed, while the general purpose GPU computation frameworks like CUDA and OPENCL are still not available on mobile devices, it is possible to use programmable fragment and vertex shaders that are actually available in mobile GPUs. The core idea behind a fragment shader is that it defines how to compute each pixel of an output image. To achieve a highly parallel computation, each pixel in the output image must be computed independently from all the others in the sense that it is not possible to use, in the computation of a pixel, the result of the computation of a different one.

The proposed solution adopts a single fragment shader to compute both gradient magnitude and direction. These two operations can be computed in a single fragment shader as both depend on the input image only. Vice versa, anchors computation depends on the result of the other two operations, hence it is implemented in a separate fragment shader. The result of each operation is stored in a different channel of an RGB image.

Our experimental results, run on an iPhone 5s with the methodology presented in Section 5, show that, on average, anchors extraction is more than 4 times faster when run on GPU. In absolute terms, the average time required to compute these operations on a single frame is about 8.5 ms when computed in CPU and less than 2 ms when computed in GPU.

4. Features processing

Starting from the line segments extracted from the image, *ZebraRecognizer* groups them into candidate crossings (Section 4.1) that are then validated (Section 4.2). Finally, *ZebraRecognizer* selects the most relevant crossing and computes the distance measurements (Section 4.3).

4.1. Line segments grouping

The aim of the line segments grouping phase is to partition the set of line segments into blocks, each one representing a different candidate crossing. Each candidate crossing is characterized by a set of stripes, that, in turn, are composed by a pair of line segments each. During line segments grouping, rectified line segments are processed, so that it is possible, for example, to straightforwardly check geometrical properties (e.g., parallelism) and to compute quantified measurements (e.g., the width of each stripe).

All line segments are first assumed to be part of a single set that is then partitioned according to three criteria: 'slope', 'horizontal overlapping' and 'vertical distance'. The idea behind the slope criterion is that the line segments in the same crossing are mutually parallel. For example, in Fig. 7d, line segment 7 is not grouped with the line segments in the dashed box due to the 'slope' criterion. The same holds for line segments 9–14.

In addition to being parallel, line segments composing a zebra crossing should also be reciprocally 'aligned'. Technically, consider the projections of the line segments on a line parallel to them; it should hold that the large part of each line segment projection overlaps with the projections of the other line segments. The evaluation of this criterion should take into account that in some cases a line segment can actually have a small overlap due to the fact that it is partially outside the field of view. It is possible to distinguish these cases because it is known, for each end-point of each line segment, if it lies on the image boundary (see Section 3.3). Consider the example of Fig. 7d. The line segments in the dashed box are all grouped together, even if the line segments closer to the user have a smaller overlapping: this is due to the fact that part of the line segment is outside the field of view. Vice versa, line segments 3, 4 and 8 are not grouped together with the line segments in the dashed box because their overlap with the other line segments is too small.

Finally, the vertical distance criterion guarantees that, in each group, two consecutive line segments must have opposite gradient directions and a distance of about 50 cm (this is specific for Italian regulation, see Section 2.3). For example, in Fig. 7d line segments 1 and 2 are too close to the line segments in the dashed box and hence are not grouped with these line segments. Analogously, line segments 5, 6 and 15 are too far away and, again, are not grouped together with the line segments in the dashed box.

Each grouping criterion is enforced by using an agglomerative hierarchical clustering technique with single linkage. The first criterion ("slope") is applied to the entire set of line segments (considered as a single set) and results in a set of blocks, each one used as input for the iterative application of the other two criteria.

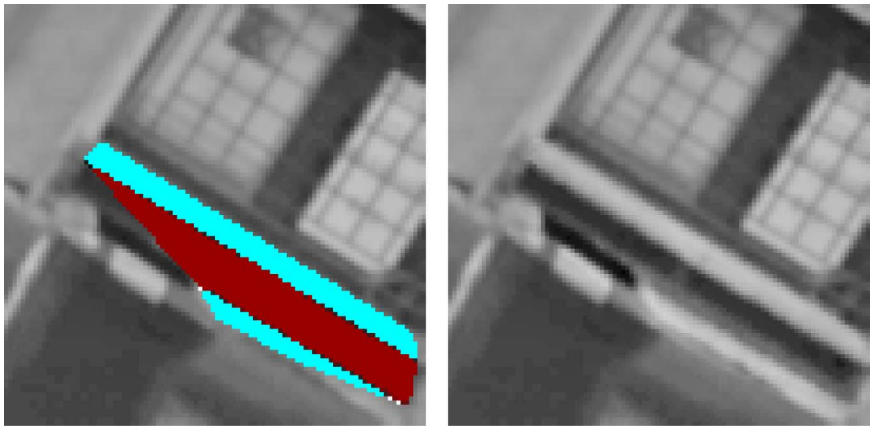
4.2. Zebra crossing validation

After the line segments grouping step, each resulting block is validated according to two criteria: 'grayscale consistency' and 'number of edges'.

Grayscale consistency criterion captures the fact that each light (or dark) stripe has a grayscale level that is lighter (darker, respectively) than the average grayscale level of the candidate crossing. Clearly the expected grayscale level (light or dark) of a stripe is known due to the fact that the gradient of its two edges is defined. The minimum required difference between the stripe grayscale level and the crossing average grayscale level is specified by the "grayscale consistency magnitude threshold" parameter. Thanks to this criterion, structures that are geometrically similar to stripes but without consistent dark/light alternating grayscale level are discarded. An example of application of the grayscale consistency criterion is shown in Fig. 9a and b. After the grouping phase, some line segments are grouped in a single block and hence are marked as a candidate crossing (Fig. 9a). However, as can be observed in Fig. 9b, there is a too small difference in the coloration of the identified stripes. By enforcing the grayscale consistency criterion the candidate crossing is discarded.

The second validation criterion, is "number of edges". It defines that a valid zebra crossing should be composed of a minimum number of edges. In most of our experiments, this value is set to 5, hence guaranteeing that each crossing contains at least two white stripes, as required by Italian regulation. Consequently, blocks that contain a smaller number of line segments are pruned.

In theory, the number of edges criterion could only be checked as the last step of the recognition procedure (i.e., after



(a) A false positive candidate crossing after grouping

(b) False positive is discarded by "grayscale consistency" criterion

Fig. 9. Application of the "grayscale consistency" criterion.

enforcement of grayscale consistency). However, checking the number of edges criterion requires a negligible time (i.e., it takes constant time in our implementation). For this reason this criterion is evaluated after each step of grouping and validation in order to reduce the number of line segments to process, hence improving the overall computation time of *ZebraRecognizer*.

A candidate crossing that meets the grayscale consistency and the number of edges criteria is marked as a 'validated crossing'.

4.3. Final result computation

In many cases either none or a single validated crossing is returned by the validation phase. However, it is possible that two or more crossings are returned. This happens, for example, at crossroads or when there are two consecutive zebra crossings separated by a traffic island. To decide which one is the "most relevant" crossing for the user, we adopted the following methodology. We identified, in a set of sample images (see Section 5) the cases in which two or more validated crossings are identified. By observing them, we empirically defined this procedure: the most relevant crossing is the closest to the user among those having roughly the same direction as the user. Consequently *ZebraRecognizer* first checks if any detected crossing has an orientation angle within a threshold from the user's orientation. If favorable crossings are available, all other crossings are discarded. Among the remaining ones, the closest one to the user is selected as the most relevant.

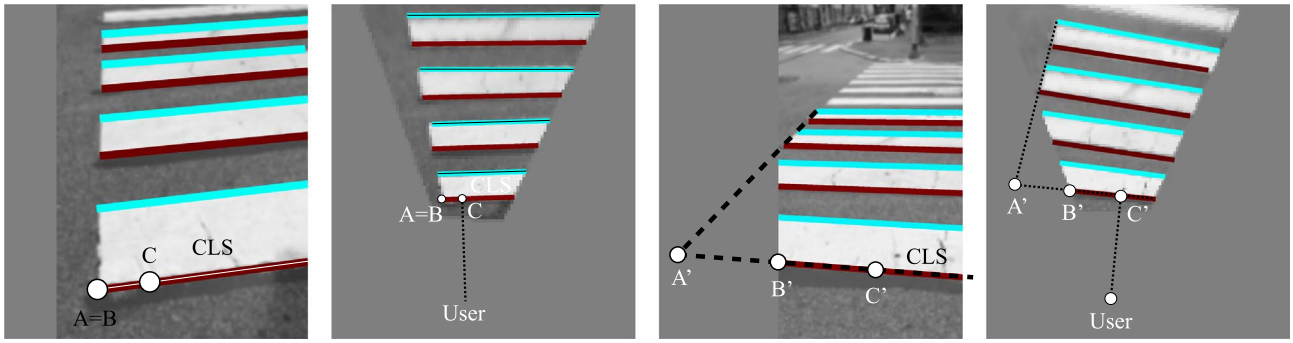
Once the most relevant crossing has been selected, its position with respect to the user is computed for the purpose of guiding the user during the crossing. In particular, the distance is computed as a set of four distance measurements, represented in Fig. 5c. "Frontal distance" is defined as the distance between the user and the closest line segment (called *CLS* in the following). "Rotation angle" is the (oriented) angular distance between the user's heading and the crossing. In the figures shown in this paper we represent the user pointing upwards, so the rotation angle corresponds to the stripes angle. In theory, since the line segments should be mutually parallel, the angle is the same for all line segments. However, in practice, there can be some approximation and hence the rotation angle is computed as the average angle of all line segments. The third and fourth distance measurements are "lateral distance left" and "lateral distance right". We will describe the former, the latter is analogous. "Lateral distance left" intuitively represents the distance between the user and the left border of the crossing measured on *CLS*. More formally, it is the (directed) distance between the left border of *CLS* and the projection of the user's position on *CLS*.

There is an issue arising in the computation of "lateral distance left" (the same holds for "lateral distance right"). Indeed, it is possible that the edge of the first detected stripe is not entirely contained in the image. In this case the left end-point of *CLS* does not necessarily represent the left border of the closest stripe. Let's consider two examples. In Fig. 10a the left end-point of *CLS* (point *B*) actually represents the left end of the stripe (point *A*). Fig. 10b shows the rectified view. Differently, in Fig. 10c and d the first stripe is not fully contained in the image and the left end-point of *CLS* (i.e., point B') is not the left end of the stripe (i.e., point A'). In the first case (Fig. 10a and b) it is clear that the user is close to the left border and hence he/she should be instructed to strife right before crossing. Should the same instruction be provided in the second case? The answer is negative. Indeed, by observing the stripes that are farther from the user, it is possible to infer that the first stripe extends on the left of the user hence, intuitively, it is safe to start crossing in the current position. To capture this intuitive reasoning, we take into account the left end-points that are marked as not-being on the image boundary (see Section 3.3). If there are too few of these points, the "lateral distance left" is marked as *not quantifiable*. Vice versa, we use an orthogonal regression algorithm to find the stripe "border" i.e., the line that passes through these points. We then compute the intersection A' of this line with the line where *CLS* lies. The "lateral distance left" is then computed as the length of $\overline{A'C'}$ distance.

5. Experimental evaluation

5.1. Experimental methodology

When *ZebraRecognizer* is run in *ZebraX*, the input data are taken directly from the device's camera and sensors, and this makes it impossible to run the recognition procedure twice with the same input. Clearly this is a problem with app debugging, parameters tuning and performance measuring. To overcome this issue we first collect images and then we process them off-line. We developed two applications. *zRecorder* is a mobile application that records the stream of images and motion sensors data (i.e., accelerometer and gyroscope). The other application, *zSimulator*, reads the data stored by *zRecorder* and uses it as an input to run *ZebraRecognizer* so that its performance can be measured. *zSimulator*



(a) Left edge is visible.

(b) Figure 10a rectified.

(c) Left edge is not visible.

(d) Figure 10c rectified.

Fig. 10. Computation of lateral distance.

can be run both on traditional devices (i.e., desktops and laptops) and on mobile ones. This approach significantly eases the debugging process and enables regression tests, parameters tuning and reproducible experimental tests.

We used *zRecorder* to create four sets of images (with corresponding motion data) at 1080×1920 resolution. All sets are publicly available.[5] The first set, called *Testset1*, consists of 40 videos and 4015 frames captured in different illumination conditions (sunny, cloudy and night). All frames have been manually annotated to distinguish those containing a zebra crossing (1877) from the remaining ones (2138). The second set, called *Testset2*, includes 6 videos with a total of 206 frames. In this case, for each frame we also annotated the relative position of the crossing. To minimize the approximation while collecting this information, we recorded the videos by using a tripod positioned at a given frontal and left/right distance from the crossing. Since the tripod is stationary, the frontal and lateral distances are fixed for each video, while the rotation angle varies. To measure the rotation, before starting the recording, we calibrate the device so that it is perfectly perpendicular with the stripes and then, for each frame, we measured the rotation angle by using gyroscopes information. We empirically observed that the error introduced by the gyroscopes is negligible, also considering that the duration of the recording is of few seconds and that the device is not subject to sudden movements (since it is on a tripod). The third set, called *Testset3*, is a subset of Testset1 that contains only heavily blurred images with zebra crossings (manually selected from Testset1). Testset3 contains 613 images, mostly taken in conditions of low ambient light. Finally, the fourth set, called *Testset4* contains 265 images of zebra crossings that are partially covered by external objects, for example a pole (like in Fig. 8).

We used a desktop pc for computationally intensive evaluations (e.g., parameters tuning) and an iPhone 5s smartphone for evaluating the execution time.

We take four indicators into consideration: precision, recall, execution time and positioning accuracy. Precision, calculated as the ratio between the correctly detected crossings and all the detected crossings, measures the amount of false positives. A precision score of 1.0 means that each detection corresponds to a crossing in the examined image, conversely a lower ratio implies that some crossings were detected where none was present. The recall metric is computed as the ratio between the detected crossings and all the correct crossings in the dataset. While a score of 1.0 means that all the crossings were correctly detected, lower values indicate that some of the crossings were not. Given the safety concerns for the navigation of users with visual impairment or blindness in a dangerous environment, we notice how anything less than a perfect precision score is unacceptable, while a high recall score, although important, is less critical. Henceforth, unless differently stated, we report our results in which the precision is always equal to one.

The execution time defines the average time needed to run *ZebraRecognizer*. It does not take into account the time required to load the image from the hard drive nor the time required to resize the input image. Indeed, when *ZebraRecognizer* is used in *ZebraX*, the input image is already acquired at the necessary resolution and no resizing is needed. Clearly, lower execution time allows higher frame rates, increasing the responsiveness of the detection with respect to the user's movements. Also, it means that the procedure is less computationally intensive, with a lower power consumption.

Finally, the positioning accuracy indicates the extent to which the relative position returned by *ZebraRecognizer* is precise. The positioning accuracy in a given frame is characterized by four values, one for each distance measurements. Each value is the difference between the distance computed by *ZebraRecognizer* and the expected (actual) value. Clearly, positioning accuracy can only be computed if the expected relative distance is known and hence only using *Testset2*.

5.2. Parameters tuning

This section reports the results of the study conducted for the tuning of five representative parameters that highly influence the recognition performances: “resolution”, “grouping angle”, “grayscale consistency”, “blur kernel size” and “blur standard deviation”.

The “resolution” parameter specifies the size of the image on which the detection is run. The “grouping angle” parameter defines the maximum angular distance between two line segments that are grouped together (see Section 4.1). The “grayscale consistency” parameter defines the minimum difference in grayscale level (value range between 0 and 255) between a stripe and the average grayscale level of the crossing (see Section 4.2). The last two parameters refer to the strength of the blur filter applied during image pre-processing (see Section 3.2). These parameters are listed in Table 1 together with their minimum and maximum values used during parameters' tuning, and their default chosen values.

Fig. 11a shows that with a very low resolution (below 90×160) recall diminishes drastically. This is due to the fact that in these cases the features are hard to detect. For high resolutions (above 180×320) there is also a reduction in recall due to the fact that

[5] http://webmind.di.unimi.it/ZebraRecognizerTestSet/

Table 1
Most influential parameters and their values.

Parameter	Min	Chosen	Max
Resolution	90×160	180×320	720×1280
Grouping angle	1.5	3	7.5
Grayscale consistency	1	5	9
Blur kernel size	3	9	13
Blur standard deviation	0.7	0.9	1.4

noise and imperfections are more visible and impair drastically the segment detection stage. While this behavior can be offset by using stronger blurring during the preprocessing step (see Section 3.2), higher resolutions do not improve the detection accuracy. Thus, the default resolution used for the detection is 180×320 pixels.

For the "grouping angle" parameter we observe (see Fig. 11b) that, for larger values of this parameter, recall is higher due to the fact that larger blocks of line segments are generated with the application of the "slope" criterion hence it is less likely that they are pruned by the "number of edges" criterion. However, for values larger than 3°, some false positives can be introduced and hence precision diminishes, although very slowly. For this reason, the default value is 3. The analysis for the "grayscale consistency" parameter is similar (see Fig. 12a): for smaller values of this parameter the "grayscale consistency" criterion is easier to satisfy, hence there is a higher recall. However, for values smaller than 5 precision is less than 1. Hence, we choose 5 as the default value.

Fig. 12b shows that, for what concerns the "blur standard deviation" parameter, there is a peak in both precision and recall for the value of 0.9. Thus, we chose this value as the parameter default. For the "blur kernel size" parameter, we can observe in Fig. 12c that, for values smaller than 7, there are some false positives (i.e., precision is less than 1). Vice versa, when this parameter is set to 7 or higher, precision is 1. For values larger than 7, both precision and recall are not influenced, but the computation costs are higher. Hence, we chose the value of 7 for this parameter as default.

5.3. Impact of GPU computation

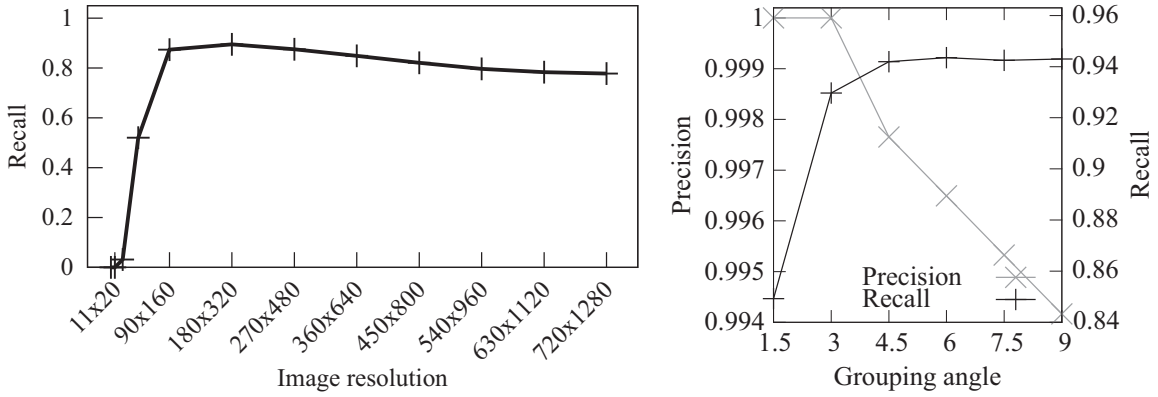
One set of experiments is aimed at assessing the improvements of the GPU implementation of anchors extraction (see Section 3.4). Fig. 13a shows the comparison between the CPU and the GPU implementations for different values of the "resolution" parameter. As expected, this parameter significantly influences the execution time of anchors computation as this is an operation with time complexity linear in the number of pixels. Indeed, the computation time of the CPU implementation is 2.5 ms for images with resolution 90×160, while it is almost exactly four times larger (i.e., 9.67 ms) for images with four times the number of pixels (i.e., 180×320). The same increase can be observed for images with resolution 360×640. Differently, with the GPU implementation, the total computation time is composed by a constant-time overhead (we estimate its cost to be about 1.5 ms) and the actual computation, whose cost is indeed linear in the number of pixels and about 10 times faster than with the CPU implementation. So, overall, while the computation on the GPU leads to an improvement of about 30% for 90×160 images, the improvement is much larger with 180×320 images (the default resolution value) where the GPU implementation is more than 4 times faster. In our experiments we also observed that for larger images the benefits are even higher (e.g., for 360×640 images the GPU implementation is about 8 times faster).

One question is how the GPU implementation of anchors extraction impacts on the overall computation time of *ZebraRecognizer*. Fig. 13b helps us provide an answer by showing, at the default resolution, how the entire computation time of *ZebraRecognizer* is divided between anchors extraction and all other operations. When anchors extraction is computed in CPU, it requires almost the same time as all the other operations (precisely, anchors extraction takes 44% of the entire computation time). Vice versa, with the GPU implementation, anchors extraction requires 15% of the entire computation time. Overall, since the GPU implementation is about 4 times faster, it improves the overall *ZebraRecognizer* computation time by about 30%.

5.4. Robustness

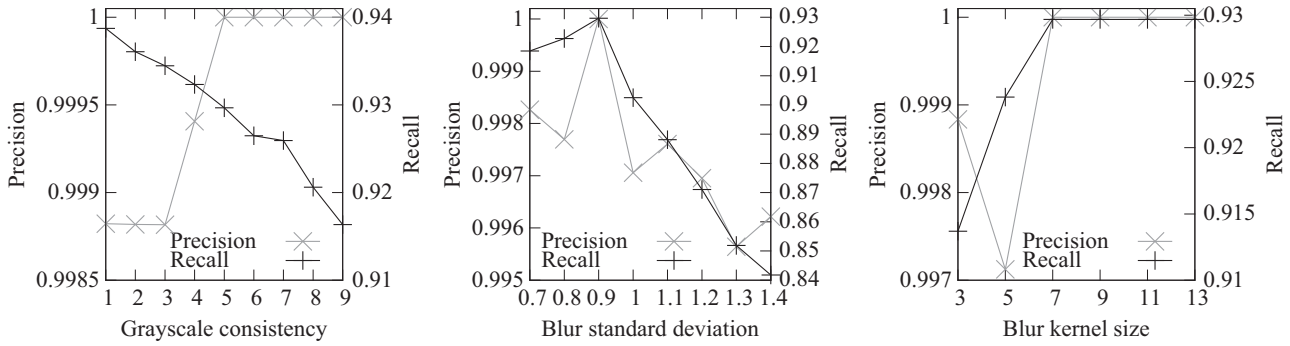
We used Testset3 and Testset4 to evaluate the robustness of the proposed solution when the zebra crossing is heavily blurred or partially covered. In this analysis, since the two testsets contain true positives only (all images contain a zebra crossing), we evaluated recall only. Note that *ZebraRecognizer* has not been specifically tuned for these two testsets of images: the values of all parameters are the same as defined in the tuning phase, conducted with Testset1 (see Section 5.2).

Fig. 14 shows a comparison of the recall obtained in Testset1, Testset3 and Testset4 when two different techniques are used for line segments detection. In this section we consider the default technique only (Edlines); we discuss the results with the other technique (LSD) in Section 5.5.



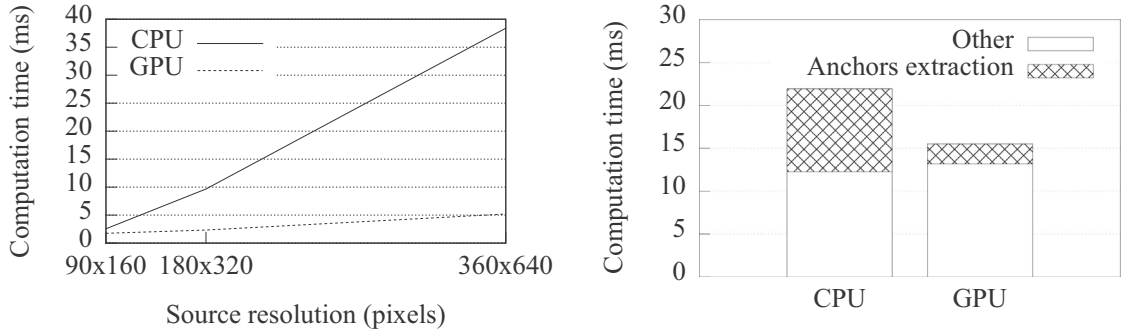
(a) Resolution parameter (b) Grouping angle

Fig. 11. Results of parameter tuning.



(a) Grayscale consistency (b) Blur standard deviation (c) Blur kernel size

Fig. 12. Results of parameter tuning.



(a) Computation time of anchors extraction (b) Impact of anchors computation time in *ZebraRecognizer*

Fig. 13. Computation time of *ZebraRecognizer*.

We can observe in Fig. 14 that, when *ZebraRecognizer* is run with heavily blurred images, recall slightly improves (from 0.93 with Testset1 to 0.96 with Testset3). This is due to the fact that images in Testset3 are mainly captured in conditions of low ambient light (when it is easier to have heavily blurred images). In this light condition, there is a higher contrast between light stripes and the dark background, which makes it easier to detect crossings.

For what concerns Testset4, we can observe that, considering only images in which the stripes are partially covered, the decrease in recall is very small: from 0.93 with Testset1 to 0.88 with Testset4. This supports the fact that, in the great majority of the cases, the line segment detection algorithm is able to reconstruct the entire stripe edge, even when it is partially occluded.

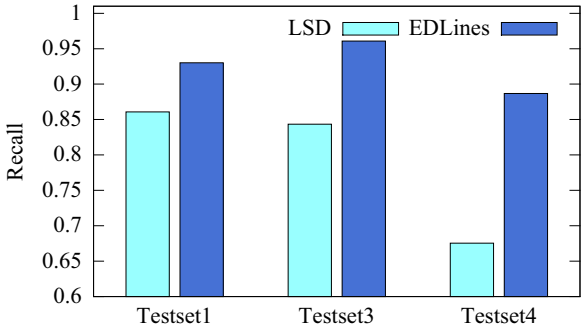


Fig. 14. Recall with Testset1, Testset3 and Testset4 using EDLines and LSD.

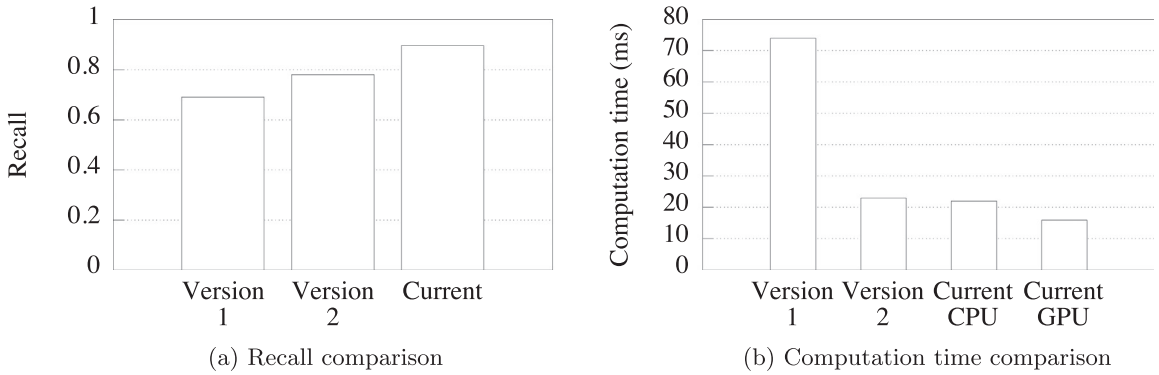
5.5. Comparison with previous solutions

In this section we first compare the impact on *ZebraRecognizer* of two different algorithms for line segment detection and then we compare the solution presented in this paper with our previous ones.

Since line segment detection is a crucial part of our technique, we investigated the impact of two different approaches: a customized version of EDLines (described in Section 3.3) and a customized version of Line Segment Detector (LSD), [14].

We implemented a version of *ZebraRecognizer* adopting LSD and we tuned it with the same methodology described in Section 5.2 for the "standard" *ZebraRecognizer* version that uses EDLines. In practice, we tuned the parameters to obtain no false positives (i.e., to have precision 1) and to have the highest possible recall. Fig. 14 shows that the algorithm performs consistently better, in terms of recall, when EDLines is adopted. Indeed, in Testset1, the recall is 0.93 and 0.86 for EDLines and LSD, respectively. A similar result is obtained for Testset3. For Testset4, the *ZebraRecognizer* version using EDlines yields a much higher recall score. This suggests that the solution based on LSD is less efficient in reconstructing the line segments if they are partially occluded. Also, when LSD is adopted, *ZebraRecognizer* has a computation time that is about 3 times higher than with EDLines. For the above reasons, we can conclude that EDLines outperforms LDS for this specific application.

We now compare the solution presented in this paper with our previous solutions [8,1], that are henceforth called "Version 1" and "Version 2", respectively. A direct comparison with other solutions is unfeasible because the implementation and the data used for their evaluation are not public. Vice versa, as we explain in Section 5.1, the data used for our tests is public, so that a direct comparison of future works with our solution is possible. The three



(a) Recall comparison (b) Computation time comparison

Fig. 15. Performance comparison (precision is 1 in all versions).

solutions are compared according to two metrics: recall and computation time.

For each version we use the corresponding default system parameters, which, as previously stated, are tuned to yield a precision equal to 1.

For what concerns recall, Fig. 15a shows that it improved from .69 in Version 1 to .78 in Version 2 up to .93 in the current version of *ZebraRecognizer*. The improvement from Version 1 to Version 2 is mainly due to the fact that in Version 2 the geometrical properties are checked on the rectified image. The improvements from Version 2 to the current version is due to the number of improvements described in Sections 3 and 4.

For what concerns the computation time, in Version 1 the average time to process each frame is 74 ms, while in Version 2 it is 23 ms. In the current version of *ZebraRecognizer* the average time is 22 ms with the CPU implementation of anchors extraction while it is 16 ms with the GPU implementation. Considering also the image acquisition time, *ZebraX* can process about 25 frames per second.

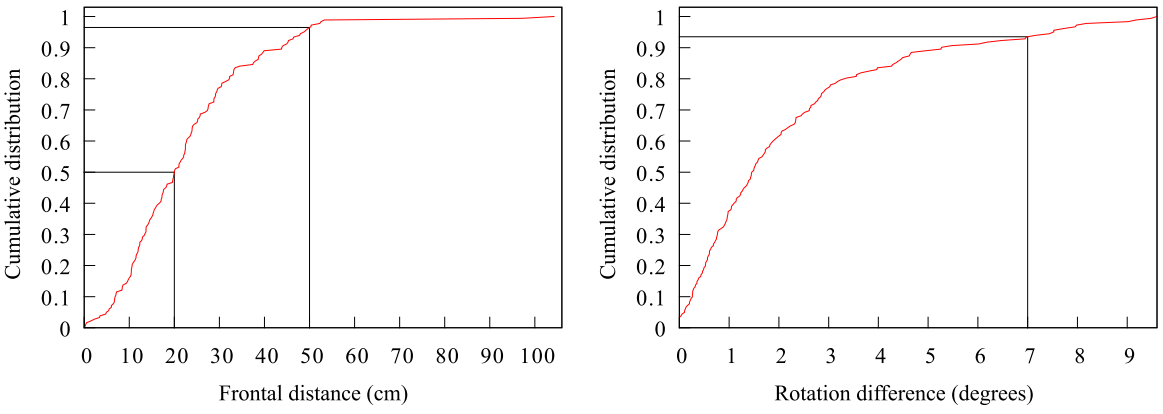
The small improvement between Version 2 and the current CPU implementation is due to two contrasting factors: on one side, we engineered and optimized the code, hence improving the computation time by about 20%. On the other side, we fixed a bug in the line segments merging algorithm (see Section 3.3). The effect of the bug was to erroneously terminate before merging was complete, hence resulting in a partially incorrect result but faster computation. After fixing this bug, all line segments are now correctly merged, but the improvement in computation time from Version 2 to the current version is negligible. Still, the GPU implementation of anchors extraction guarantees an improvement of about 30%.

5.6. Positioning accuracy

A set of experiments is aimed at asserting the approximation introduced when computing the four distance measurements (see Section 4.3). In the following we indicate as "error" the absolute value of the difference between the distance (frontal, angular or left/right shift) returned by *ZebraRecognizer* and the expected (correct) distance.

For what concerns the frontal distance, the average error is 0.22 m. Fig. 16a shows the cumulative distribution function (CDF) chart of the error occurring in the computation of frontal distance. It can be observed that in 50% of the cases the error is less than 20 cm, while in 96% of the cases the error is less than 50 cm, which corresponds to approximately one step.

In a few cases the error is about 1 m: this is due to the fact that the first white stripe is not recognized. Fortunately this problem occurs in few frames (less than 3%) that are generally non-consecutive (the longest sequence we measured is composed of two consecutive frames). This makes it possible to identify the occurrence of this problem in the *Logic* module by checking for sudden changes in the frontal distance. Indeed, since the temporal distance between two consecutive frames is less than 0.05 s



(a) Frontal distances cumulative distribution (b) Rotation distances cumulative distribution

Fig. 16. Accuracy of frontal and rotation distances.

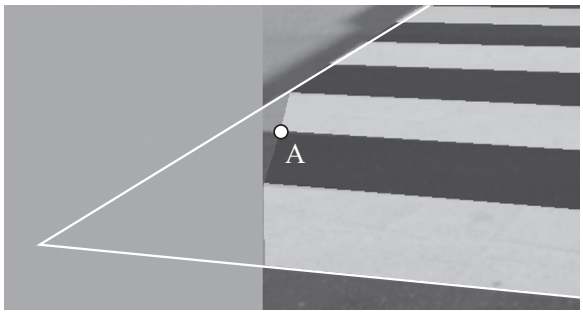


Fig. 17. Stripe edge not completely recognized.

(frequency is about 25 frames per second), a change in the frontal distance larger than 0.5 m clearly indicates that the closer stripe has not been recognized. The results of the frontal distance error also show that the mean error is larger when the observer is far from the crossing. For example, when the observer is 4 m far from the crossing, the mean error is 0.24 m, while for a distance of 2 m the mean error is less than 0.2 m.

For what concerns the rotation angle, the average error is about 2.2°. Fig. 16b shows the CDF chart: it can be observed that the error is up to 9.5° and that in 93% of the cases the error is less than 7°. In this case it is not possible to check for sudden changes, as a user can possibly rotate very quickly. Nevertheless values can be smoothed by using a moving average in the *Logic* module. For example, with a moving average of length 3, the average error in the rotation angle is 1.0° and the maximum error observed in the experiments is 3.0°.

During our experiments we observed that the computation of the lateral distance is subject to a non-negligible approximation caused by two factors: first, EDLines frequently does not recognize the entire stripe edge, but just a portion of it and consequently the computation of the border is not always precise. See Fig. 17 for an example. Second, the projection of the user's position on *CLS* (the line segment closest to the user) can be imprecise due to approximations in the computation of *CLS* angle. To address the former issue, our solution excludes, from the border computation, the points that introduce an error above a given threshold as, for example, points *A* in Fig. 17. This is useful, for example, when there are few line segments that are much shorter than the actual stripe edge. To address the latter issue, when computing the projection on *CLS*, instead of using the angle of *CLS*, we use the average angle computed among all the stripes. The resulting technique always correctly identifies a lateral distance as *not quantifiable* (i.e., the border is out of the field of view, see Section 4.3). In some rare cases it happens that, even if the border is visible, it is still identified as *not quantifiable*. This is often due to the fact that the border is only visible in the stripes that are far away from the user and these stripes are not recognized. In any case, in 87% of the cases, if a border is visible then it is identified by our technique and, in these cases the average error is 0.25 m.

6. Conclusions and future work

This paper presents *ZebraRecognizer*, a software module to recognize pedestrian crossings. The requirements of this module were derived from the experience in the development of *ZebraX*, an application that recognizes zebra crossings, computes the safe path to correctly align and provides audio feedback to guide the user with visual impairment or blindness. This paper shows that *ZebraRecognizer* can compute the quantified and accurate position of the zebra crossing without incurring into any false positive and with few false negatives. At the same time, *ZebraRecognizer* is efficient on mobile devices.

We are currently working on the other two modules composing *ZebraX*. The *Logic* module has two main objectives: first, to keep the information about the stripes position updated by using dead reckoning techniques and, second, to compute a safe path to guide the user towards the crossing. Also, this module could implement a form of spatio-temporal reasoning to track the already recognized zebra crossings between consecutive frames. The *Navigator* module interacts with the user and its main challenge is to provide effective audio feedback without distracting the user from the surrounding environment.

As a future work, we intend to integrate *ZebraX* with iMove, a commercial application we developed that supports independent mobility of people with visual impairment or blindness. As a parallel research direction we intend to extend the machine vision technique to recognize other elements of the urban environment relevant for a user with visual impairment or blindness, in particular traffic lights.

Appendix A. Proof of Property 1

The notation used in the proof refers to Fig. A1a and b.

We approximate the ground to an infinite plane. Thus, line l, which points from the device camera to the horizon, is parallel to the ground plane and angle $\widehat{FDP}$ is $\pi/2$.

We define the horizon line h in the image by using its angle θ and a point P where h passes. The equation of a line having slope m and passing through point $P = (P_x, P_y)$ is:

$$y - P_y = m(x - P_x)$$

The slope is computed from the line angle θ as $m = \tan(\theta) = \sin(\theta)/\cos(\theta)$.

Replacing m in the equation we obtain the horizon line h in its general form:

$$\sin(\theta)x + \cos(\theta)y - (\sin(\theta)P_x + \cos(\theta)P_y) = 0 \tag{A.1}$$

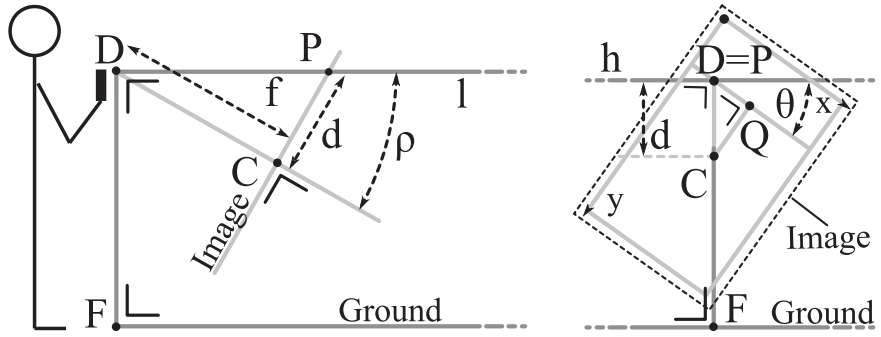
We now show how to compute θ and P.

Consider Fig. A1a. Let P be the point where the image plane intersects line l. Thus, point P lies on the horizon line h and P is inside the image. Also, since point D is the device, segment $\overline{DC}$ is perpendicular to $\overline{CP}$. Hence PCD is a right triangle. Since CD is the focal distance f and angle PDC is the device pitch angle ρ, the distance (in pixel) between the image center C and point P is $d = f \cdot \tan(\rho)$.

In the image plane, the device roll θ is the inclination of the device's x-axis with respect to the ground plane. Since the horizon line h is parallel to the ground plane, θ is also the inclination of the horizon in the image. Consider Fig. A1b. Let Q be the projection of C on the line parallel to the x-axis (in the device reference system) that passes through P. Since $\widehat{CPQ} + \theta = \pi/2$, it follows that $\widehat{PCQ} = \theta$. Since the distance d is known, then the distance of point P from point C along the x-axis is $d_x = PQ = d \cdot \sin(\theta)$. Analogously, the distance of point P from point C along the y-axis is $d_y = CQ = d \cdot \cos(\theta)$. Thus, the coordinates of point P are $P = < C_x - \sin(\theta)d, C_y - \cos(\theta)d >$.

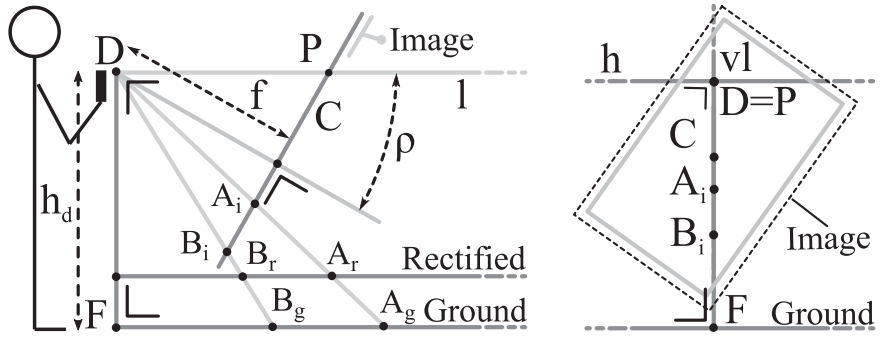
Finally, substituting d and P in Eq. A.1 we obtain:

$$\sin(\theta)x + \cos(\theta)y - \sin(\theta)(C_x + \tan(\rho)\sin(\theta)f) - \cos(\theta)(C_y + \tan(\rho)\cos(\theta)f) = 0 \tag{A.2}$$



(a) Lateral view. (b) Frontal view.

Fig. A1. Device orientation for the computation of the horizon line equation.



(a) Lateral view. (b) Frontal view.

Fig. B1. Zoom factor computation.

Appendix B. Proof of Property 2

To compute the zoom factor *ZebraRecognizer* takes into account two points A_i and B_i on the image plane and their projections A_r, B_r, A_g and B_g on the rectified plane and ground plane, respectively. See Fig. B1a.

A_i and B_i are arbitrary points below the horizon and lie on line *vl*, which is perpendicular to the horizon and passes through the image principal point *C*. The distance A_rB_r is computed by applying the rectification to A_i and B_i and by computing the Euclidean distance on the resulting points A_r and B_r. The distance A_gB_g on the ground plane is computed as $A_gB_g = FA_g - FB_g$.

To compute FA_g consider right triangle DFA_g of which we know the device's height h_d. The angle $\widehat{FDP}$ is right and it is also the sum of the device pitch ρ, angle $\widehat{FDA_g}$ and angle $\widehat{A_gDC}$. The angle $\widehat{A_gDC}$ can be computed from the device's focal distance f and the distance A_iC as $\widehat{A_gDC} = a\tan(A_iC/f)$. Thus, the angle $\widehat{FDA_g} = \pi - \rho - \widehat{A_gDC}$. Now, the distance FA_g is computed as $FA_g = \tan(\widehat{FDA_g})\cdot h_d$. An analogous approach can be used to compute the distance FB_g.

Finally, the zoom factor is computed as $z = \frac{A_gB_g}{A_rB_r}$ or, by substituting the known values:

$$z = \frac{h_d\cdot\left[\tan\left(\pi-\rho-a\tan\left(\frac{CB_i}{f}\right)\right)-\tan\left(\pi-\rho-a\tan\left(\frac{CA_i}{f}\right)\right)\right]}{A_rB_r} \quad \text{(B.1)}$$

References

[1] D. Ahmetovic, C. Bernareggi, A. Gerino, S. Mascetti, ZebraRecognizer: efficient and precise localization of pedestrian crossings, in: Proceedings of the 22nd International Conference on Pattern Recognition (ICPR), IEEE Computer Society, 2014.
[2] M. Lefler, H. Hel-Or, Y. Hel-Or, Metric plane rectification using symmetric vanishing points, in: Proceedings of the 20th International Conference on Image Processing, IEEE Computer Society, 2013.
[3] S. Se, Zebra-crossing detection for the partially sighted, in: Proceedings of the Conference on Computer Vision and Pattern Recognition, IEEE, 2000.
[4] M.S. Uddin, T. Shioyama, Detection of pedestrian crossing and measurement of crossing length – an image-based navigational aid for blind people, in: Transactions on Intelligent Transportation Systems, IEEE, 2005.
[5] V. Ivanchenko, J. Coughlan, H. Shen, Detecting and locating crosswalks using a camera phone, in: Computer Vision and Pattern Recognition Workshop, IEEE, 2008.
[6] V. Ivanchenko, J. Coughlan, H. Shen, Staying in the crosswalk: a system for guiding visually impaired pedestrians at traffic intersections, in: Assistive Technology Research Series, IOS, 2009.
[7] V. Murali, J.M. Coughlan, Smartphone-based crosswalk detection and localization for visually impaired pedestrians, in: Workshop on Multimodal and Alternative Perception for Visually Impaired People (MAP4VIP), International Conference on Multimedia and Expo, IEEE, 2013.
[8] D. Ahmetovic, C. Bernareggi, S. Mascetti, ZebraLocalizer: identification and localization of pedestrian crossings, in: Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services, ACM, 2011.
[9] D. Ahmetovic, R. Manduchi, J. Coughlan, S. Mascetti, Zebra crossing spotter: automatic population of spatial databases for increased safety of blind travelers, in: Proceedings of the 17th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2015), ACM, 2015.
[10] B.R. Ullman, N.D. Trout, Accommodating pedestrians with visual impairments in and around work zones, in: Journal of the Transportation Research Board, TRB, 2009, pp. 96–102.
[11] C. Akinlar, C. Topal, Edlines: a real-time line segment detector with a false detection control, Pattern Recognit. Lett. 32 (13) (2011) 1633–1642.
[12] D. Liebowitz, A. Zisserman, Metric rectification for perspective images of planes, in: Proceedings of Computer Vision and Pattern Recognition, IEEE, 1998.
[13] R. Huston, Principles of Biomechanics, CRC Press, 2008.
[14] R.G. Von Gioi, J. Jakubowicz, J.M. Morel, G. Randall, Lsd: a fast line segment detector with a false detection control, IEEE Trans. Pattern Anal. Mach. Intell. 32 (4) (2008) 722–732.

Sergio Mascetti is an Associate Professor at Università degli Studi di Milano, Department of Computer Science. His research interests include mobile data management, with focus on assistive technologies and privacy protection. He is a co-founder of EveryWare Technologies, a university spin-off developing assistive technologies on mobile devices.

Dragan Ahmetovic is a Ph.D. graduate in Computer Science at University of Milan with a thesis focusing on computer vision-based pedestrian crosswalk detection and crossing guidance for people with visual impairments. His research interests are computer vision, assistive technologies and sensor fusion-based localization.

Andrea Gerino is a Ph.D. student at the Computer Science Department of the University of Milan. His research interests include mobile data management and the development of mobile applications to support people with disabilities. He is a co-founder of EveryWare Technologies, a university spin-off developing assistive technologies on mobile devices.

Cristian Bernareggi is a scientific and technological collaborator at Università degli Studi di Milano, Computer Science Library. His research interests include human computer interaction with focus on assistive technologies for people with sight impairment. He is a co-founder of EveryWare Technologies, a university spin-off developing assistive technologies on mobile devices.