Streamlining collection of training samples for object detection and classification in video

A. Bulović, D. Bučar, P. Palašek, B. Popović, A. Trbojević,
L. Zadrija, I. Kusalić, K. Brkić, Z. Kalafatić, S. Šegvić
Faculty of Electrical Engineering and Computing
Unska 3, 10000 Zagreb
Email: name.surname@fer.hr

***Abstract*—This paper is concerned with object recognition and detection in computer vision. Many promising approaches in the field exploit the knowledge contained in a collection of manually annotated training samples. In the resulting paradigm, the recognition algorithm is automatically constructed by some machine learning technique. It has been shown that the quantity and quality of positive and negative training samples is critical for good performance of such approaches. However, collecting the samples requires tedious manual effort which is expensive in time and prone to error. In this paper we present design and implementation of a software system which addresses these problems. The system supports an iterative approach whereby the current state-of-the-art detection and recognition algorithms are used to streamline the collection of additional training samples. The presented experiments have been performed in the frame of a research project aiming at automatic detection and recognition of traffic signs in video.**

I. Introduction

Machine learning based techniques for object detection and classification [1], [2] are trained on large collections of training samples which are usually manually collected [3]. Effectiveness of the resulting classification algorithms largely depends on quantity and quality of both positive and negative samples [4]. The process of annotating desired objects by hand in many images is a challenging and time consuming task [5]. Some approaches even require *dense* annotations, whereby the objects need to be manually located in each frame of the video sequence [6], which obviously makes the task even harder.

We propose an iterative approach to simplify and streamline this laborious job, by taking advantage of previously collected data. Each iteration of the approach is made up of two parts. Firstly, the available training collection is used to train the current generation of detection and recognition algorithms. Secondly, the trained algorithms are employed to perform *semi-automated* detection and classification. The processing results are verified by a human operator who needs to accept, reject or correct the proposed classification results. The annotated objects are consequently employed in the next iteration of the procedure, which is expected to result in a continuous performance improvement [4].

The desired properties of the envisioned software application would be as follows. The program should provide a user friendly interface with a visual toolset for object annotation. Multiple media formats should be supported on input, while the annotated material should be stored in a standardized manner. Since we are chiefly interested in annotating video, the program should have a full-fledged media player with browsing and playlist features. In order to support semi-automated operation, the program needs to seamlessly integrate production components which implement classification, detection and tracking algorithms. The detected and recognized objects need to be automatically tracked throughout the video sequence in order to achieve dense annotation. The user needs to be able to interactively validate the processing results, and correct them if needed. The training part is usually time consuming, so it does not need to be integrated with the main application, but can instead be implemented as a separate command-line program.

In this paper, we present the most interesting architectural and implementation details of the new version of our software system Marker, which conforms to the above requirements. A beta version of the software has been exhibited at the Zagreb University Fair in January 2010, and will soon be released for download from the web pages of our research project[1] [7]. Earlier Marker version is stable and can be freely downloaded and employed for non-commercial activities[2]. It differs from the development version because it supports only manual annotation as illustrated in Fig. 1.

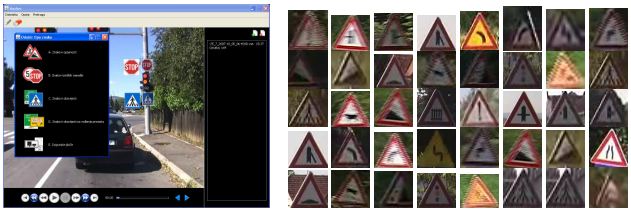

Fig. 1. A typical annotation session using the earlier stable version of the software (left) and a small subset of annotated signs (right).

The paper is structured as follows. Section II presents related software applications for collecting the training samples. The employed computer vision techniques are reviewed in Section III. Section IV describes the architecture of the proposed software system. Some experimental results in the field of training the object detector are presented in Section V. Section VI provides some implementation details about the detection component, while the paper is concluded in Section VII.

This research has been jointly funded by Croatian National Foundation for Science, Higher Education and Technological Development, and Institute of Traffic and Communications, under programme Partnership in Basic research, project number #04/20.

[1]The project's web site is at http://www.zemris.fer.hr/~ssegvic/mastif.

[2]The earlier stable version (Marker v1.0) can be downloaded from: http://www.zemris.fer.hr/~ssegvic/mastif/marker/marker.zip.

II. Related software

Similar projects in the field of annotating objects in images are the Improved Object Marker[3]. and the LabelMe[4] service from MIT [8]. To the best of our knowledge, none of the previous research has addressed the problem of streamlining the collection of dense annotations in video sequences.

The Improved Object Marker is a command-line application written in C++ which enables annotating objects in single images. It features a cropping tool and a mechanism for saving the annotated object positions in a format conforming to the tools from OpenCV [9]. Unlike our system, it can't operate on video, and lacks a graphical user interface.

The LabelMe project [8] is also concerned with object annotation, however its purpose significantly differs from ours. LabelMe is basically a web service which provides a huge collection of annotated images. The goal of the project is to train detectors for finding multiple classes of objects in complex scenes. The LabelMe annotation tool enables users to contribute additional annotations to the collection. The tool has a well designed user interface for manual object labeling. However, the tool can not automatically propose annotations, and is not capable to work on video sequences.

III. The employed computer vision algorithms

This section briefly reviews the three employed computer vision algorithms: detection by a boosted Haar cascade [10], [11], [3], recognition by support vector machines (SVM) [3] and differential tracking [12], [13], [14].

A. Detection by a boosted Haar cascade

One of the most prominent detection algorithms today is based on a compound binary classifier constructed as a boosted cascade of Haar classifiers [10], [11], [3]. The compound classifier need to be trained on a large collection of training samples. The algorithm employs a sliding detection window, i.e. applies the classifier over a comprehensive range of locations and scales in the input image. The detections are reported at positions where the binary classifier returns true. Typically, very good recalls can be obtained, however the precision tends to be problematic [5]. For each valid detection we typically obtain at least one false positive, as illlustrated in Fig. 2.

Another problem of this approach is that for each object in the input image one typically obtains many detections in the vicinity of the true detection. This problem is resolved by an ad hoc algorithm for grouping the adjacent detections, which often works good as illustrated in Fig. 3. However, at times the grouping algorithm produces responses which deviate from the true position for several pixels. Due to simple heuristic nature of the grouping algorithm, the deviations occur whenever the spurious surrounding responses are asymmetrically distributed around

[3]Improved object marker can be downloaded from: http://www.cse.unsw.edu.au/~gherman/Improved_Object_Marker.pdf

[4]Web site of the LabelMe project is at http://labelme.csail.mit.edu/

Fig. 2. Typical detection results obtained by a boosted Haar Cascade.

(a) (b) (c) (d)

Fig. 3. Results of the heuristics for grouping multiple detections: correct grouping (a,b), and biased grouping due to image border (c,d).

the true response. As illustrated in 3(c,d), this usually happens when the object is close to the image border, resulting in the bias towards the opposite side. These deviations need to be taken into account by subsequent recognition algorithms in order to avoid a performance loss [3].

Our annotation software supports detection of triangular signs triggered by the operator. The user typically quickly skips the road parts with no signs, initiates detection when a sign becomes visible, and finally validates the produced results. Our implementation of the detector has been inspired by the reference implementation from OpenCV[5] [9].

B. SVM classification

For purposes of classification, we use support vector machines (SVM) [2]. SVM is a supervised machine learning technique used for binary classification. Given a set of n-dimensional vectors each of which belongs to one of the two classes, the training algorithm generates a maximum margin hyperplane dividing these objects into classes[6]. After the training, the hyperplane is used for classification of newly acquired unclassified objects, acting as a classifier. The distance between a vector and the hyperplane can be used for measuring the probability of correct classification, with greater distance meaning higher certainty of correctness.

The SVM classifier is by definition unable to distinguish between more than two classes of objects at once. Real world problems dictate a need for classifying multiple object types, so a special class "ALL" is mandatory.

[5]OpenCV can be downloaded from http://sourceforge.net/projects/opencvlibrary/.

[6]Note that if the classes are not linearly separable, one first needs to transform the input vectors by a non-linear kernel function.

It is basically an array of all other object classes. For example, if only five types of traffic signs existed, A, B, C, D and E, when training the A vs "ALL" SVM classifier, the "ALL" class would consist of sign types B, C, D, E and could also include other objects that aren't the type A sign, i.e. trees, asphalt or the sky. Using this type of classifier, we can predict whether the object in question belongs to a certain class or not. For every sign type X from the set of all sign types, a SVM classifier X vs "ALL" is trained, stored and later used for multiple class classification.

The multiple class classification process is as follows: an object is classified using all of the trained X vs "ALL" SVM classifiers, each of them returning X or "ALL" with the probability of correct classification as a result. In a best case scenario, only one classifier will confirm that the object belongs to its class, and all other will return the "ALL" class as a result. If there have been multiple confirmations, they are then sorted in order of greatest probability of correct classification.

Our annotation software makes a classification request whenever a new sign is detected or manually annotated. The selected image window is first scaled to match the size of training samples and then passed on to all X vs "ALL" SVM classifiers. The number of positive answers that are returned to Marker is limited to four. Marker prompts the user to select the correct classification between the most probable results, as can be seen in Figure 4.

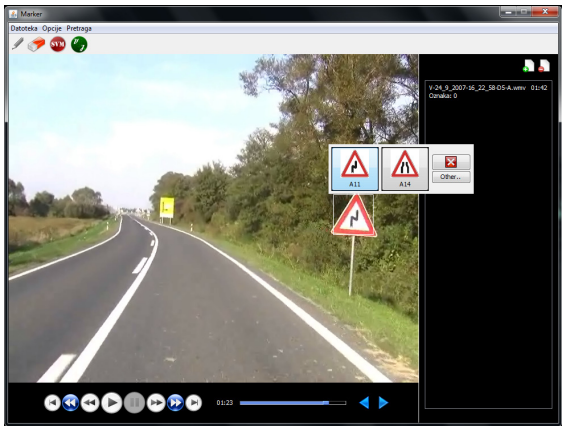

Fig. 4. The results of object classification ordered by probability of correct classification

The SVM learning algorithm uses feature vectors that are calculated from each image. The features we used were the RGB values of each pixel. All images were beforehand scaled to 24x24 pixels, which means that we have 1728 (24*24*3) features. Because of the size of the training sets (which was considerably smaller than the number of features), no kernel functions were used.

C. Differential tracking

Differential tracking [12], [13], [14] is often employed for locating corresponding rectangular patches in neighbouring frames of a video sequence. The correspondence is established by a suitable gradient descent optimization technique, usually starting from the previous patch location. The algorithm tends to be sensitive to large displacements between the initial approximation and the solution, however this can be alleviated by a resolution pyramid and suitable heuristics for improving the initial approximation. In the implementation, a modified version of a known public library has been used[7]

Our annotation software tracks each annotated object which could have been classified either automatically or manually. If the tracked object is lost, the operator can manually locate it and restart the tracking. The tracking stops either when the object leaves the field of view, or when the object considerably changes its appearance.

IV. The Proposed Software Architecture

The proposed software architecture is illustrated in Fig. 5. On the top of the hierarchy we have the main GUI application entitled Marker. Marker is written in Java in order to simplify the development of a complex GUI. Unfortunately, Java is not appropriate for all domains of software engineering. Thus, the main application delegates several pieces of functionality to external plugins written in C++. Since Java can not access arbitrary dynamic libraries, the plugins need to conform to the JNI specification [15], which makes them unsuitable for use in other projects involving different languages. Consequently, the plugins with reuse potential have been implemented as library pairs including i) a thin JNI wrapper (modules libmastif_jni and libvs_jni), and ii) a native reusable library (modules libmastif and libvs). The reusable libraries are implemented in terms of production components written in C++, which can be conveniently tested and debugged from the separate program cvsh.

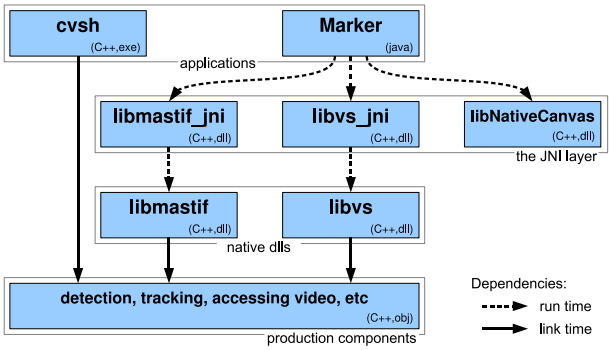

Fig. 5. The proposed architecture of the annotation system

A. The GUI application Marker

Automating the annotating process with use of the before mentioned algorithms is a formidable task. To do so, one requires a user friendly application to put all of the parts together. The main goal of Marker is to make the process of annotating objects by hand as easy as possible. By using Java as a primary coding language, the application gained an aspect of portability[8]. From the implementation viewpoint, Marker's main strength

[7]The KLT tracker maintained by Stan Birchfield can be accessed at http://www.ces.clemson.edu/~stb/klt/.

[8]Not needing to be installed and being capable of running on any chosen platform

is modular and extensible design based on the Model-View-Controller (MVC) design pattern [16]. The program implementation had been extended with interfaces for different component classes (such as Tracker, Classifier and Detector), so it could later be upgraded or changed with as least work as possible. We are experimenting with options to achieve a semi-automatic work mode, which would activate adequate algorithms instead of the user when certain criteria are met.

B. The bridge - Java Native Interface

Java Native interface (JNI) [15] is a feature of the Java platform which allows applications written in Java to incorporate native code written in programming languages such as C and C++. JNI is a two-way interface that allows Java applications to invoke native code and vice versa.

Modules libmastif_jni and libvs_jni are dynamic link libraries which provide a thin JNI wrapper around native libraries libmastif and libvs. Such layered design allows Marker to reuse functionality from independently developed native libraries without code duplication. The main functionality of the JNI counterparts is to convert the parameters from JNI types to native C++ types and vice versa.

The module libNativeCanvas provides the functionality of native image drawing. This module was required since it turned out that drawing images in Java at video rate and faster is considerably slower than what can be obtained with native APIs.

C. The native libraries

The module libmastif provides a flexible interface for invoking computer vision algorithms over a DLL interface. This is especially beneficial for algorithms such as detection or tracking, for which Java is not suitable due to extreme computational demands. At the implementation side, the module transparently employs low-level production components which have been developed and tested within our C++ development framework entitled cvsh (cf. IV-D). Thus the module promotes interoperability between the C++ production code and clients written in different programming languages such as Java or Visual Basic.

The module libvs allows its clients to interact with native libraries for reading proprietary video formats such as WMV or AVI. Unfortunately, it appears that there are no public Java libraries with such functionality, so that use of native libraries is mandatory.

D. The C++ development framework

The computer vision shell cvsh is our C++ framework for experimenting with computer vision and image processing procedures. The framework can be employed both on Windows and Unix operating systems such as Linux or Mac OS. It provides command-line and point-and-click user interfaces, handles image acquisition and presentation of results, and offers powerful registry services which allow the application logic to be independent from the incorporated computer vision algorithms. Thus, users can *add* custom algorithms without having to *change* any other component in the program.

V. Experiments with detection performance

In this section we report some experiences with training the boosted Haar cascade for detecting triangular traffic signs. Currently we employ the training tool from OpenCV [17], [9], which requires i) a large set of positive samples (cropped images of desired objects), and ii) a collection of backgrounds (images in which no objects are present). The negative samples are automatically extracted from the background images. In order to obtain proper evaluation results, the training and evaluation sets are extracted from different video materials. For the purpose of this study, 1100 positive evaluation samples have been annotated using Marker on a recently acquired video.

A. Impact of the training set size

There are many parameters which affect the performance of the trained detector, however here we focus on the impact of the training set size. For each training configuration we report the algorithm which obtained best evaluation results. The results are summarized in Table I. The columns of the table correspond to number

TABLE I
Impact of the training set to the detection performance

N_{pos}	N_{bg}	recall	precision
352	110	68%	46%
898	230	80%	64%
2154	711	96%	54%

of positive samples, number of background images, the obtained detection rate, and the obtained precision. The first row of the table is taken from [7], while the second two rows correspond to recent experiments. The results clearly show that the performance increases with increasing sizes of the training collections. The low precision is problematic, and therefore represents an important subject for future work.

B. Impact of scale discretization

As explained in III-A, the detection window is applied over the whole image on many scales, starting at 24×24 and finishing when the detection window becomes greater than the input image. The factor between two neighbouring scales is a major parameter of the detection algorithm. A typical distribution of results is shown in table II. The

TABLE II
Impact of scale discretization to detection performance

sf	recall	precision
1,35	97.2%	13%
1,31	98.1%	11%
1,3	98.1%	13%
1,28	97.6%	11%
1,2	97.6%	24%
1,1	94.4%	6%
1,05	91.0%	3%

columns of the table correspond to the scale discretization factor (the ratio between successive scales during the detection), and the obtained recall and precision for a given classifier. Although it was expected that smaller scale factors would always result in larger recall, this does not happen in practice due to poor performance of the

grouping algorithm (cf. III-A) when there are too many detections.

C. Accuracy of the grouping algorithm

We examined the accuracy of the grouping algorithm as the distance from the true response center to the center of the reported position of the group. The distance is normalized with respect to the size of the true response. The distribution of the obtained relative distances in situations when the object is far from the image border is shown in Fig. 6.

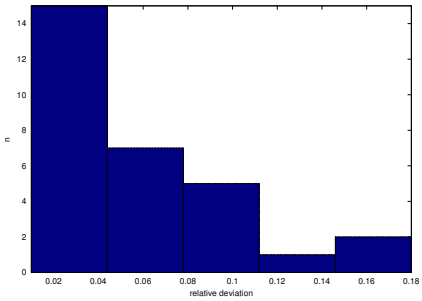

Fig. 6. The distribution of the relative deviation of the grouping algorithm.

We can see that, although most grouped responses are quite close to the true position, many grouped responses deviate considerably. The mean relative deviation is about 6%, which corresponds to about 3-4 pixels when the object is 60 pixels wide.

D. Discussion

The best algorithm detects most signs in at least two out of four images, regardless of the detector variant. Most of missed detections occur when the sign is farthest from the camera. When the sign is near the border the frame tends to moved away from the border, as illustrated in 3. False positives usually appear on the roadway, or on the roofs. False positives occasionally appear even in the clear sky or in the tree tops. This happens because the detection algorithm performs local photometric equalization before applying the classifier to the detection window.

VI. Impact of various optimizations to the detection speed

A straightforward implementation of detection based on a boosted Haar cascade is too slow for practical use. Processing of single 720x576 image approximately took 2.25 seconds on a modern test machine, with full compiler optimization. In this section we review the impact of the following optimization techniques to the detection speed:

- parallel programming
- replacing C++ source code with machine code
- modifications of manipulation with data structures

A. Parallel programming

Most modern processors have multiple cores. However, conventional languages such as C++ or Java can not exploit this processing power without delving into synchronization intricacies between processes and threads. This problem has been addressed by a new technology called OpenMP OpenMP is an application programming interface which supports implicit shared memory multiprocessing[9].

The implementation of boosted Haar cascade detector has six nested for loops, over x, y, scale, cascade level, Haar classifiers and individual rectangles. In the straightforward implementation this segment of source code was taking 96% of time execution. By employing OpenMP in second nested loop, the performance was improved significantly on multi-core processors. If a block contains six nested loops than total number of iterations n_{total} is:

$$n_{\text{total}} = n_1 \times n_2 \times n_3 \times n_4 \times n_5 \times n_6 \qquad (1)$$

where n_i is the number of iterations of the i-th loop. On a parallel machine with n_{threads} threads we can hope to decrease the total count of iterations to:

$$n_{\text{total}} = \frac{n_1 \times n_2 \times n_3 \times n_4 \times n_5 \times n_6}{n_{\text{threads}}} \qquad (2)$$

It should be pointed out that parallelizing the 6th or the 2nd loop is not the same thing. When master thread encounters a parallel region it forks to a team of threads; after all threads complete their jobs, they join the master thread. Forking and joining uses valuable processor time, so it is better to do as little forks/joins as possible. For example, if the 6th loop is parallelized then the total number of forks/joins is $\frac{1}{n_{\text{threads}}} \prod_{i=1}^{6} n_i$, while if the 2nd loop is parallelized, the number of forks and joins is $\frac{1}{n_{\text{threads}}} \prod_{i=1}^{2} n_i$. Thus we see that it is better to optimize the 2nd loop than the 6th. Also, some loops can not be parallelized because of the loop carried dependencies. In our case, this holds for the first loop.

After applying this optimization as sketched above, the performance on a dual core system increased for 87%.

B. Replacing C++ source code with machine code

Modern compilers still can not exploit all processing power offered by modern hardware. In particular, many x86 compilers refrain from generating instructions from the multimedia subsets such as Streaming SIMD Extensions (SSE). Thus, the conversion from the type double to the type `int` which often needs to be performed in the implementation of the detection is by default performed by x87 instructions, and not by the considerably faster `cvtsd2si` instruction from SSE. Replacing the C++ truncation code with this instruction improved the performance for 7%.

C. Modifications of data structures

The detection algorithm has to use complex data structures which require complex calculations of memory addresses. This problem can be alleviated by precalculating references to the data structures as soon as possible, in order to release the burden on inner loops. The implemented solution improved the performance for 77%

[9]More information about OpenMP is available from http://www.openmp.org.

D. Overall impact of optimizations

The contribution of all optimizations is summarized in Table III. Experiments have been performed on processor Intel Core 2 Duo 2.4 GHz resulting in the current performance of 0.66 seconds per frame.

TABLE III
IMPACT OF INDIVIDUAL OPTIMIZATIONS

optimization	processing time	throughput
initial state	2.25 s	0.4 fps
OpenMP	1.2 s	0,8 fps
machine code	2.18 s	0,5 fps
modified C++ code	1.26 s	0,8 fps
all optimizations	0.66 s	1,5 fps

VII. CONCLUSION

We have presented architectural and implementation details of our software system for streamlining the collection of training samples for machine learning based detection and recognition algorithms. The presented architecture allows a flexible cooperation between Java GUI and independently developed and tested production components in C++. We have also presented some experiments dealing with the training, performance and optimization of the detector based on a boosted Haar cascade. The developed software shall be exploited in our future work, where we shall research ways to obtain better detection and recognition results as well as even more streamlined collection of the training data.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the contribution of colleagues Toni Benussi, Petra Bosilj, Mateja Čuljak, Darko Jurić, Bojan Miklenić, Martin Morava, Josip Hucaljuk, Bruno Kovačić and Alan Sambol.

REFERENCES

[1] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. New York: Wiley, 2. ed., 2001.

[2] C. M. Bishop, *Pattern Recognition and Machine learning*. Information Science and Statistics, Springer, 2006.

[3] M. Enzweiler and D. M. Gavrila, "Monocular pedestrian detection: Survey and experiments," *IEEE Trans. PAMI*, vol. 31, no. 12, pp. 2179–2195, 2009.

[4] S. Munder and D. Gavrila, "An experimental study on pedestrian classification," *IEEE Trans. PAMI*, vol. 28, no. 11, pp. 1863–1868, 2006.

[5] P. M. Roth, *On-line Conservative learning*. PhD thesis, Graz university of Technology, Institute for Computer Vision and Graphics, 2008.

[6] G. J. Brostow, J. Shotton, J. Fauqueur, and R. Cipolla, "Segmentation and recognition using structure from motion point clouds," in *Proc. of ECCV*, (Marseille, France), pp. 44–57, Oct. 2008.

[7] S. Šegvić, K. Brkić, Z. Kalafatić, V. Stanisavljević, D. Budimir, and I. Dadić, "Towards automatic assessment and mapping of traffic infrastructure by adding vision capabilities to a geoinformation inventory," in *Proc. of MIPRO*, (Opatija, Croatia), May 2009.

[8] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman, "Labelme: A database and web-based tool for image annotation," *Int. J. Comput. Vis.*, vol. 77, no. 1-3, pp. 157–173, 2008.

[9] G. R. Bradski and A. Kaehler, *Learning OpenCV*. O'Reilly Media, Inc., 2008.

[10] P. Viola and M. J. Jones, "Robust real-time face detection," *Int. J. Comput. Vis.*, vol. 57, no. 2, pp. 137–154, 2004.

[11] K. Brkić, A. Pinz, and S. Šegvić, "Traffic sign detection as a component of an automated traffic infrastructure inventory system," in *Proc. of AAPR/ÖAGM*, (Stainz, Austria), May 2009.

[12] J. Shi and C. Tomasi, "Good features to track," in *Proc. of CVPR*, (Seattle, Washington), pp. 593–600, June 1994.

[13] H. Jin, P. Favaro, and S. Soatto, "Real-time feature tracking and outlier rejection with changes in illumination," in *Proc. of ICCV*, vol. 1, (Vancouver, Canada), pp. 684–689, 2001.

[14] S. Segvic, A. Remazeilles, A. Diosi, and F. Chaumette, "A scalable mapping and localization framework for robust appearance-based navigation," *Comput. Vis. Image Underst.*, vol. 113, pp. 172–187, Feb. 2009.

[15] S. Liang, *Java Native Interface: Programmer's Guide and Reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[16] E. Freeman, E. Freeman, K. Sierra, and B. Bates, *Head First Design Patterns*. O'Reilly Media, Inc., 2004.

[17] R. Lienhart, A. Kuranov, and V. Pisarevsky, "Empirical analysis of detection cascades of boosted classifiers for rapid object detection," in *Proc. of DAGM*, (Magdeburg, Germany), pp. 297–304, 2003.