A brief introduction to OpenCV

Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, Mario Cifrek
Faculty of electrical engineering and computing, University of Zagreb, Zagreb, Croatia
ivan.culjak@fer.hr, david.abram@fer.hr, tomislav.pribanic@fer.hr, hrvoje.dzapo@fer.hr, mario.cifrek@fer.hr

Abstract - The purpose of this paper is to introduce and quickly make a reader familiar with OpenCV (Open Source Computer Vision) basics without having to go through the lengthy reference manuals and books. OpenCV is an open source library for image and video analysis, originally introduced more than decade ago by Intel. Since then, a number of programmers have contributed to the most recent library developments. The latest major change took place in 2009 (OpenCV 2) which includes main changes to the C++ interface. Nowadays the library has >2500 optimized algorithms. It is extensively used around the world, having >2.5M downloads and >40K people in the user group. Regardless of whether one is a novice C++ programmer or a professional software developer, unaware of OpenCV, the main library content should be interesting for the graduate students and researchers in image processing and computer vision areas. To master every library element it is necessary to consult many books available on the topic of OpenCV. However, reading such more comprehensive material should be easier after comprehending some basics about OpenCV from this paper.

I. INTRODUCTION

Computer Vision is the science of programming a computer to process and ultimately understand images and video, or simply saying making a computer see [1]. Solving even small parts of certain Computer Vision challenges, creates exciting new possibilities in technology, engineering and even entertainment. In order to advance vision research and disseminate vision knowledge, it is highly critical to have a library of programming functions with the optimized and portable code, and hopefully available for free. This was an original goal of Intel team back in 1999 when OpenCV (Open Source Computer Vision Library) was officially launched. Since then, a number of programmers have contributed to the most recent library developments. The latest major change took place in 2009 (OpenCV 2) which includes main changes to the C++ interface. The newest library release can be found on the OpenCV official website [2]. Nowadays the library has >2500 optimized algorithms. It is extensively used around the world, having >2.5M downloads and >40K people in the user group. OpenCV can be used in academic and commercial applications as well, under a BSD license [3]. To master every OpenCV library element it is necessary to consult many books available on the topic of OpenCV. Nevertheless, reading such more comprehensive material should be easier after comprehending a basic idea about OpenCV from this paper. In fact to make it even more convenient, the text presented here intentionally closely follows one of the most recent OpenCV sources [4].

II. BASIC LIBRARY STRUCTURE AND FEATURES

OpenCV library (since version 2.2) is divided into several modules, where each module can be understood, in general, as being dedicated to one group of computer vision problems. All the classes and functions are defined within the name space *cv*. Therefore to access them we can either precede the main function definition by the declaration *using namespace cv;* or prefix OpenCV class and function names by namespace specification *cv::*. The main object is of class *Mat*. As implicated by the class name it is essentially a matrix holding pixel values of some image and, in addition, a number of attributes about an image. In the simplest case an image can be created as *cv:: Mat image*;, creating an image of size 0 by 0. Perhaps the most important member variable of *image* object is *data* where *image.data* member is actually a pointer to the allocated memory block that contains the image data (in this trivial case it would be *image.data*=0). Alternatively during a creation of *Mat* object we can explicitly specify an initial size and the type of each matrix element. This type specifies, for example, signed 1-byte pixel image values (*CV_8U*), or three channels for a color image (*CV_8UC3*), or even 32-bit/64bit floating point numbers (*CV_32F*).

Once an object of class mat is defined, a nice feature about it (not present in the early versions of OpenCV) is that a memory allocation/deallocation takes place automatically. For instance, a memory automatically allocated during the image read out into some object, will be also automatically released once the corresponding object goes out of scope.

Another important thing is that *Mat* class implements the reference counting and shallow copy. Hence, when an image is assigned to another one, the image data itself is not copied and both images point to the same memory block (this also applies to images passed by/returned by value). However, since reference count is supported, a memory allocated for image (pixel) data itself will be released only when all of the references to the image are destructed.

In the versions prior to OpenCV 2, C like functions and structures were used (still can be though) and the main structure was *IplImage*. Although there is a convenient way to convert *IplImage* structure into *cv::Mat* object, it is highly recommended to avoid this deprecated data structure.

III. ACCESING PIXEL VALUES

The basic image content is made of picture elements known as pixels. Consequently, an efficient pixel

processing is of high importance since images can easily have tens of thousands of pixels. Since a matrix is a basic data structure in OpenCV, it seems reasonable that each element of the matrix represents one pixel. However for a gray level image, pixel values themselves are typically unsigned 8bit values, whereas for a color image three such 8bit unsigned values are used per pixel to describe three primary colors, i.e. channels (Red, Green and Blue; actually OpenCV uses BGR channel order). In more general sense, OpenCV allows to create matrix of various value types (as described earlier) and it is important to note that certain operations (pixel processing) can be applied only on the certain matrix types. Moreover, certain image processing could be more effective if appropriate color space is chosen [5].

In order to access each individual matrix element essentially we just need to specify row and column, but the way we actually do it can influence code mostly in terms of performance efficiency and clarity. One of the access method is to use method *at(int x, int y)*. However, in this case the type returned must be known at compile time and, consequently, *at* method is implemented as a template method. For example, accessing pixel on coordinates (j, i) of some *Mat* object *image* and where returned type is 8bit unsigned value, can be written as *image.at<uchar>(j, i)*. Note that *at* method does not perform any type conversion and it is a programmer's responsibility that specified type matches the actual one in the matrix. For color images a syntax is somewhat more complex since in that case we actually retrieve a vector of three 8bit values: *image.at<Vec3b>(j, i)*[channel]. *Channel* determines one of the three color channels.

Specifying a returned type as a template argument for each call may be cumbersome. In case an image type is *a priory* known we can use *Mat_class* which is a template subclass of *Mat* class. Now we specify matrix type only once at the object creation and later on we just use *operator()* to access individual pixel values, e.g. *Mat_<uchar> im2=image; im2(101,303)=0;*.

Accessing pixel elements with a pointer generally provides a more efficient code, frequently at cost of more cumbersome syntax. Fortunately, OpenCV offers *ptr* method which simplifies a pointer arithmetic by directly giving an address of an image row *j*, e.g. *uchar* data=image.ptr<uchar>(j);*

We can also access pixel simply using traditional pure low level pointer arithmetic. Considering that the image data is contained into memory block of unsigned chars, we can retrieve the address of the first element in this block, i.e. *uchar * data=image.data;*. Now obtaining the address of the pixel at row *j*, column *i* we should write *data=image.data+j*image.step+i*image.elemSize();* where *elemSize* returns number of bytes occupied by one pixel, *step* is the actual number of bytes per one row. Namely, for efficiency reasons the length of rows in bytes is often not necessarily equal to the theoretically excepted number equal to image width in pixels × number of bytes occupied by one pixel. Rather the rows are padded with few extra pixels in order to be multiples of 4 or 8. Certainly these extra pixels are neither displayed nor saved. On the other hand, in case the image is not padded with extra pixels at the end of each row, it is interesting to note that an image can be seen as one dimensional array of *Width×Height* pixels. We can take advantage of this continuity by scanning the image pixels within a single loop, which in turn can yield a more efficient code (particularly in cases when several smaller images are scanned simultaneously in the same loop). Whether or not image has been padded a convenient method *isContinous* can tell us.

Looping over data collection in object oriented programming is frequently done using iterators (a special class which hides how iteration over each element is specifically done for each data collection). Similarly as the Standard Template Library (STL) has an iterator class associated with each of its data collection classes, OpenCV provides *Mat* iterator class, which is compatible with standard iterators found in the C++ STL. Since iterators scan pixels of certain type, we need to declare iterator using template where we define a specific type to be known at compile time, e.g. *MatIterator_<Vec3b> it;* Nice thing about iterators is that no matter what collection is scanned it follows always the same pattern. We conclude that in principle, using iterators, similarly as *at* method, has the main objective to make the code less error-prone. However, working directly with pointer provides usually more time efficient code.

To combine images using arithmetic operators, OpenCV provides numerous functions which names by themselves reveal their purpose, e.g. *add, absdiff, multiply, divide, bitwise_and* etc. Note that in all cases function *saturate_cast* is used to assure the results within a defined pixel value domain. Optionally, we can include a mask in a function call where a particular operation will be executed only on pixels for which the mask value is not null. In addition, input images have to have the same size and type, only the output image will be re-allocated if it does not match the input size. Alternatively, we can use the usual C++ operators which are overloaded in OpenCV. Occasionally, it is necessary to work only on the particular image channel (plane). *split* function will copy the three channels of a color image into three distinct instances. A reverse action to combine a color image from three 1-channel images is done via *merge* function.

IV. MAIN ALGORITHMS AND METHODS

In the above we have briefly described some of the main engine features used for OpenCV implemented processing techniques and methods. Below we try to mention several OpenCV image processing techniques and algorithms which are known to be widely used too.

The concept of image histograms is well explored for image analysis. A histogram is a table showing a number of pixels having a given value in an image (occasionally a set of images). Histogram entries are called bins. OpenCV function which computes histogram is *calchist()*. To enhance image contrast perhaps the first idea would be to spread an original narrow range of intensities on a full available range (i.e. linear stretch). However, sometimes the real problem is that some

intensities are used more frequently than others, which is readily visible from the histogram intensities distribution. Based on the idea that a good quality image should make (almost) equal use of all available intensities, the concept of histogram equalization is brought up, which tries to make image histogram as flat as possible. This is a task of OpenCV *equalizeHist()* function. Besides, the intensities distribution can be understood as the probabilities that some pixel will have a particular value. This constitutes a base to detect specific image content using a histogram back projection. This back projection replaces each pixel value in an input image by its corresponding probability read in the histogram, eventually yielding a probability map. Furthermore, this probability map can be used to detect an object on the image utilizing a mean shift algorithm [6]. It is an iterative procedure which locates the local maxima of a probability function by finding the weighted centroid of the data point inside a predefined window. The algorithm then moves the window center to the centroid location and repeats the procedure until a window center converges to a stable point. The mean shift has been extensively used for visual tracking. In addition, a more advanced version where the size and the window orientation can change is implemented as well under the name CamShift algorithm [7].

Morphological operations are operations where the value of each pixel in the output image is based on a comparison of the corresponding pixel in the input image with its neighbors. By choosing the size and shape of the neighborhood (i.e. a structuring element), we can construct a morphological operation that is sensitive to specific shapes in the input image. A structuring element has an origin (usually at the center of the structuring element) which is aligned with a given pixel. The most fundamental morphological operators are erosion (the output pixel value is the minimum value of all the pixels in the input pixel's neighborhood) and dilation (the output pixel value is the maximum value of all the pixels in the input pixel's neighborhood). Morphological filters are usually applied on the binary images. Assuming that foreground (objects) is white and background is black, an erosion operation will reduce the object size whereas a dilatation will increase the object size. Combining dilation followed by erosion, with the same structuring element, defines a closing operation. The idea is to connect together the object components erroneously fragmented into smaller pieces (this would be a task of dilatation step, while the subsequent erosion step is to undo a previous dilatation effect for the very small objects which are most likely noise and thus meant not to be a part of any object). Similarly, combining erosion followed by dilation, with the same structuring element, defines an opening operation. Here, the idea is to remove (erosion part) a small objects (all of the ones too small to contain a structuring element) presumably introduced by the image noise. Since this erosion also shrunk the valid objects to undo this effect, opening operation also assumes a subsequent dilatation. In fact, closing and opening are usually applied in the sequence (e.g. during object detection). Alternatively, the opening can be applied before closing if priority is noise filtering, but possibly at the price of eliminating some (smaller) fragmented objects as well, before they have chance to be defragmented. It should be also noted that applying the same opening (closing) on an image several times has no additional effect on the result acquired after its first application. Morphological filters can be used to detect lines and corners also ([8], [9]). In the first case, this procedure essentially comes down to differencing dilated and eroded image. In the second case, in addition to the mentioned image differencing, an appropriate structuring element has to be chosen such to emphasize only the corner points and to leave straight edges unaffected.

Instead of observing distribution of image gray-level intensities, we can study the frequency of intensity variations. This point of view is referred as the frequency domain and allows a processing pallet aimed at image frequency filtering. A frequency domain analysis decomposes an image into its frequency content from the lowest to the highest frequencies. Such decomposition is often undertaken using Fourier or Cosine transform. Under the frequency analyses framework context a filter is an operation that amplifies/removes certain bands of frequency. OpenCV offers *blur()* function which smoothes the image by replacing each pixel by the average value computed over rectangle neighborhood. Hence, *blur()* function is a low-pass filter. A considered neighborhood in the form of mask is also called a kernel, and mathematically this operation is named convolution. More advanced function use allows filtering not only with a rectangle window, but with some other shapes e.g. Gaussian. Low-pass filtering is routinely used prior to image down sampling by some factor N. Without it, i.e. simply just discarding every N column and row would cause effect known as aliasing. Visually it manifests as jagged image distortions and it is caused by the fact that image is simply too small to represent fine textures i.e. high frequencies. This is why filtering out high frequencies is necessary prior to down sampling itself. Unfortunately, up sampling the down sampled image will not recover exactly the original image. In any case, this down/up sampling operations are often used to create image pyramids, a data structure built for certain efficient analyses, e.g. an object detection on smaller size image first, followed by the refined search on the higher resolution images. Median filter is another OpenCV implemented filter option (*medianblur()*) aimed at removing outlier noise (e.g. salt and paper type of noise) where pixel values are replaced by the median of all neighborhood considered pixels. We note that a median filter although very helpful in preserving the sharpness of edges, it also washes out the texture in uniform regions.

Sometimes the idea is to amplify high frequency instead of attenuating it. This is the case when using directional filters to perform edge detection where image intensity gradient (composed of first order derivates in two orthogonal directions) is computed. Alternatively, a high pass linear filter can be based on the (sum of) second order derivates, i.e. the Laplacian filter. Here, the edge is determined by zero crossing of the Laplacian function, were in principle we scan the Laplacian image and detect the change of sign for two neighborhood pixels. Thresholding the gradient magnitude and/or change of Laplacian around zero, we conveniently acquire the image edge map. However, choosing the right threshold

for edge detection is not a trivial task and some more advanced edge detectors actually use two thresholds, e.g. Canny [10]. The Canny operator is generally based on one of the gradient operators (e.g. Sobel). On such gradient image we first apply separately low and high threshold, producing low and high threshold edge map respectively. The idea is to combine those two edge maps in a way that from the low threshold edge map we keep only those points with a continuous path linked to some edge point on the high threshold edge map. Consequently, the Canny edge detector should yield a good compromise allowing good quality contours.

The Hough transform is another popular algorithm implemented in OpenCV, aimed at detecting lines, but extendable for some other image structures as well ([11], [12]). The Hough transform usually takes as input edge pixel map. The idea is to parameterize a particular image structures with a certain number of parameters. In the case of line $N=2$ parameters suffice. Next, N dimensional accumulator is constructed where each entry represents an image structure (line) with a particular set of parameters. Considering one point of input image we search all lines that pass through this point. The entries of the accumulator corresponding to the found line parameters are then incremented. Evidently when the same line passes through many points, a corresponding accumulator entry will be high and this line can be considered as significant.

Assuming an image has a distinguishing points, it is possible to detect them using interest point (also known as keypoints or feature points) detectors. Corners are one of the most common feature points. OpenCV offers an implementation of Harris corner detector ([13], [14]). A corner being the junction of two edges is a two-dimensional feature. Based on that fact, Harris detector looks at the rate of intensity change (characterized by a covariance matrix) around some point and obtains the maximal average intensity change for some direction. Next, it checks if average intensity change in the orthogonal direction is also high, then we have a corner.

A faster corner detector than Harris is offered in the form of FAST (Features from Accelerated Segment Test) algorithm [15]. It avoids a computation of image derivates by examining a circle of pixels centered around some candidate point. In more detail, if an arc of contiguous pixel points (all having intensities significantly different from the center point) of length greater than 3/4 of the circle perimeter is found, then a corner is declared.

Even more advanced keypoint detectors are designed to be scale invariant. This is particularly advantageous if we try to match the same feature from two images taken at a different distance from the object. Evidently, using a fixed size neighborhood for matching would be hard since the scale change will prevent intensity patterns from matching. OpenCV implemented the scale invariant SURF (Speeded Up Robust Features) keypoint detector [16]. SURF computes the image derivates on different scales (image resolutions). Given some image point, at one scale filter response will reach its maximum. If this maximum is at least a minimum value supplied as an algorithm input, a key point (i.e. a scale invariant feature) is declared. SURF actually uses a Hessian matrix to detect corners at image points with high local curvature, and at different scales. Thus, a scale invariant feature is declared when Hessian matrix determinant reaches its local maximum at certain scale and given pixel location. SURF approximates Gaussian kernels in order to speed up a computation. In addition, OpenCV implements a somewhat slower alternative, but considered more accurate, SIFT (Scale invariant Feature Transform) which uses Laplacian filer response instead of Hessian determinant [17].

Feature detectors are usually used for feature matching across different images and this normally requires definition of featured descriptors as N dimensional vectors that describe a feature point. The goal is to make descriptors invariant to change in lightening and to small perspective deformations. OpenCV provides function to compute SURF feature descriptors based on previously computed feature points. It allows additionally comparing feature descriptors from different images and finding eventually the best match for some feature. In turn, matches can be used to solve many task in 3D reconstruction, visual tracking, image registration etc.

A digital camera is a device that captures a 3D scene onto a 2D plane. Related to, it is a camera calibration procedure which aims to find function parameters describing a projection of 3D point onto 2D image plane [18]. OpenCV offers functions which based on the so called calibration images computes those parameters. In turn, two or more calibrated cameras allow a reconstruction of 3D point using only its correspondent image coordinates. The relation between two cameras is neatly captured through the concept of epipolar geometry and fundamental matrix. It simplifies corresponding image point search between two cameras, relating some image point of one camera with the corresponding line on the other camera, on which a corresponding image points is somewhere too. Therefore, OpenCV computes the fundamental matrix as well. Moreover, OpenCV supports a robust two view feature matching based on the fundamental matrix computation during a random sample consensus strategy. Finally, there is a relation relating two cameras views differing only by some rotation or equivalently if we consider projection of 3D planar points on cameras' images. In that case image points of two cameras are related by the so called homography matrix. This matrix OpenCV computes simply from a set of matched points between two cameras. Such homography is particularly useful in creating, for instance, panoramic images.

OpenCV offers advanced video analysis on video sequences too. Besides basic operations such as reading/viewing/writing video sequences, extracting individual frames etc. a typical two tasks are the foreground extraction and object tracking [19]. It is essential to note though that in order for OpenCV to open a specified video file, the corresponding codec must be installed on the computer. Besides, there is a possibility to read directly the video stream of a connected camera.

A point tracking in the current frame is based on the search around point's position in the previous frame. In general a point tracking algorithm is as follows. Firstly, a certain number of features are detected in the initial frame. It is assumed that an intensity of the feature point has not changed from one frame to the next one, which holds for some small displacements (u, v). Thus, it is possible to derive a feature intensity expression in the frame (t+1) as a function of an intensity in the current frame t and assumed displacement (u, v). This leads to famous Lukas-Kanade tracking algorithm [20]. It further assumes that displacement of all points in the neighborhood is the same which leads to over-determined set of equations to compute (u, v). Once OpenCV finds (*calcOpticalFlowPyrLK()*) output positions, normally we keep for further tracking only those points which have significantly moved by some amount of pixels. If by the time, the actual number of tracked points significantly decreases, it is recommended to add new feature points Note that we normally do that only in the initial frame and in the very next one we expect to find only points which actually moved and are therefore worth of tracking in the subsequent frames. When extracting a foreground from a background it is useful to compute a background dynamically (i.e. for each frame a background is updated) rather than to work with the same background image appearance throughout the process [21]. This can be accomplished by computing a so-called running average.

V. EXPERIMENTS

We demonstrate the library performance on the passive stereo matching. Stereo matching is the process of taking two or more images and estimating a 3D model of the scene by finding matching pixels in the images and converting their 2D positions into 3D depths. In more detail, for a given image location on the first (left) image (x_L, y_L), stereo matching algorithms try to find a disparity value d which will yield a correspondent image location on the second (right) image (x_R, y_R) [1]. If camera pair images are rectified [22], the following condition holds: $x_R=x_L$, $d=y_R-y_L$. As a rule there are two approaches available for the matching: area based and feature based [23]. Area based algorithms are typically further classified on the local and global optimization algorithms [24]. OpenCV has an implementation of both local and global approach, which theoretical details are beyond of this paper scope. However, we do compare OpenCV performance with the state of the art algorithms implementations using the well known Middlebury Stereo Evaluation test bed [25]. It is a globally recognized site which has an image data base utilized for the algorithm performance test in many, many published work. One of the most popular test images are perhaps Tsukuba, Venus, Cones and Teddy (see [25] for more details). The ground truth values for the pixels disparities are available also from [25]. Table I and Table II represents, for the mentioned images, a percentage of relative errors rates for the computed disparity where an estimated disparity is considered correct if it is within a ground truth disparity ±1. In terms of global methods (Table I), OpenCV demonstrates somewhat worse results. Still, we note two crucial things. Global methods are known to assume a number of threshold parameters and frequently require some experimenting before optimal ones are found for a specific image. This is what was basically done in referred work against which we compared OpenCV. Therefore, a parameter set for one image may perform quite badly for another. On the other hand, we have here used OpenCV implementation using its default parameters, i.e. the ones chosen automatically as best ones from the code itself. We feel this is a fair manner of testing since many practical users would like to use OpenCV functions with as few experiments and adjustments as possible. Thus, form this perspective we can say that OpenCV performs quite well. In terms of local methods (Table II) OpenCV implementation is side by side with the best known methods, again without any a priory manual parameter adjustment for some specific image.

Table I Performance in terms of accuracy for top ranking methods, as evaluated by Middlebury stereo site [25], which employ a significant additional processing in the form of (global) optimization and/or color segmentation.

Method	Tskuba	Venus	Teddy	Cones
ADCensus [26]	1.48	0.25	6.22	7.25
AdaptingBP [27]	1.37	0.21	7.06	7.92
CoopRegion [28]	1.16	0.21	8.31	7.18
DoubleBP [29]	1.29	0.45	8.30	8.78
OpenCV	6.79	5.01	14.84	9.57

Table II Performance in terms of accuracy for top ranking local methods, as reported in [30], which use local approach.

Method	Tskuba	Venus	Teddy	Cones
Segment support [31]	2.28	1.21	10.99	5.42
Adaptive weight [32]	4.66	4.61	12.70	5.50
VariableWindows [33]	4.10	10.66	13.93	7.24
Reliability [34]	5.14	3.86	16.96	13.52
ShiftableWindows [35]	6.53	6.60	16.16	9.55
OpenCV	9.64	5.92	13.41	7.45

VI. CONCLUSION

The purpose of this paper is to quickly make a reader familiar with OpenCV basics without having to go through lengthy reference manuals and books. Given the total number of OpenCV implemented algorithms (~thousands) and possible challenges in Computer Vision in general, it was normally beyond the scope of this paper to go in depth about every possible OpenCV detail. Actually, some more advanced topics, such as the use of GPU accelerated codes, were not even mentioned. However, the paper did present many basic and popular Computer Vision algorithms, along with many key references for an interested reader to pursue further details. The shown content should raise the interest and strengthen the awareness about OpenCV among the graduate students and researchers in image processing and computer vision areas as a whole, who may not to be

aware of it yet and/or its practical users. It is important to note that OpenCV is considered by many to be side by side with many commercial image processing packages, and yet it is an open source tool. Furthermore thanks to the fact that OpenCV keeps evolving is an additional guarantee that it will advance research in vision and promote the development of rich, vision-based CPU-intensive applications.

ACKNOWLEDGMENT

This work has been supported by the University of Zagreb Development Fund as a part of the project "Center for Computer Vision".

REFERENCES

[1] R. Szeliski. Computer Vision: Algorithms and Applications. Springer 2011.

[2] http://opencv.willowgarage.com/wiki/

[3] http://en.wikipedia.org/wiki/BSD_license

[4] R. Laganière. OpenCV 2 Computer Vision Application Programming Cookbook. Packt Publishing 2011.

[5] E. Dubois. The Structure and Properties of Color Spaces and the Representation of Color Images. Synthesis Lectures on Image, Video, and Multimedia Processing. Morgan & Claypool, 2010.

[6] C. Dorin; P. Meer. Mean Shift: A Robust Approach Toward Feature Space Analysis. IEEE Transactions on Pattern Analysis and Machine Intelligence vol. 24 (5), pp. 603–619, 2002.

[7] G.R. Bradski, Computer video face tracking for use in a perceptual user interface, 2nd Quarter, Intel Technology Journal, 1998.

[8] J.-F. Rivest, P. Soille, S. Beucher. Morphological gradients. Journal of Electronic Imaging, vol. 2 (4),pp. 326-336, 1993.

[9] F.Y. Shih, C.-F. Chuang, V. Gaddipati. A modified regulated morphological corner detector Pattern Recognition Letters, vol. 26(7), pp. 931-937, 2005.

[10] J. Canny. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Image Understanding, vol. 18 (6), pp. 679-698, 1986.

[11] C. Galambos, J. Kittler, J. Matas. Gradient-based Progressive Probabilistic Hough Transform. IEE Proc. of Vision, Image and Signal Processing, vol. 148 (3), pp. 158–165, 2001.

[12] H.K. Yuen, J. Princen, J. Illingworth, J Kittler. Comparative Study of Hough Transform Methods for Circle Finding. Image and Vision Computing, vol. 8 (1), pp. 71-77, 1990.

[13] C. Harris, M.J. Stephens. A combined corner and edge detector. Alvey Vision Conference, pp. 147–152, 1988.

[14] K. Mikolajczyk and C. Schmid. Scale and Affine invariant interest point detectors, International Journal of Computer Vision. Vol. 60(1), pp. 63–86, 2004.

[15] E. Rosten, T. Drummond. Machine learning for high-speed corner detection. European Conference on Computer Vision, pp. 430-443, 2006.

[16] H. Bay, A. Ess, T. Tuytelaars, L. Van Gool. SURF: Speeded Up Robust Features. Computer Vision and Image Understanding, vol. 110(3), pp. 346--359, 2008.

[17] D. Lowe. Distinctive Image Features from Scale Invariant Features. International Journal of Computer Vision, vol. 60(2), pp. 91-110, 2004.

[18] T. Pribanic, P. Sturm, M. Cifrek. Calibration of 3D kinematic systems using orthogonality constraints. Machine Vision and Applications. vol. 18 (6), pp. 367-381, 2007.

[19] J. Shi and C. Tomasi. Good Features to Track. IEEE Conference on Computer Vision and Pattern Recognition, pp. 593-600, 1994.

[20] B. Lucas, T. Kanade. An iterative image registration technique with an application to stereo vision. Int. Joint Conference in Artificial Intelligence, pp. 674-679, 1981.

[21] C. Stauffer, W.E.L. Grimson. Adaptive background mixture models for real-time tracking. Conf. on Computer Vision and Pattern Recognition, pp. 246-252, 1999.

[22] A. Fusiello, E. Trucco, A. Verri. A compact algorithm for rectification of stereo pairs. Machine Vision and Applications 12(1): 16–22, 2000.

[23] X. Hu and N. Ahuja. Matching point features with ordered geometric, rigidity, and disparity constraints. IEEE Transactions on Pattern Analysis and Machine Intelligence, 16(10):1041-1049, 2002.

[24] D. Scharstein and R. Szeliski. A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. International Journal of Computer Vision, 47(1/2/3):7-42, 2002.

[25] http://vision.middlebury.edu/stereo/

[26] X. Mei, X. Sun, M. Zhou, S. Jiao, H. Wang, and X. Zhang. On building an accurate stereo matching system on graphics hardware. GPUCV 2011 (In conjunction with ICCV 2011).

[27] A. Klaus, M. Sormann and K. Karner. Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure. In 18th International Conference on Pattern Recognition, pp. 15 – 18, 2006.

[28] Z. Wang and Z. Zheng. A region based stereo matching algorithm using cooperative optimization. In IEEE Conference on Computer Vision and Pattern Recognition, pp. 1-8, 2008.

[29] Q. Yang, L. Wang, R. Yang, H. Stewénius, and D. Nistér. Stereo matching with color-weighted correlation, hierarchical belief propagation and occlusion handling. IEEE Transactions on Pattern Analysis and Machine Intelligence 31(3): 492-504, 2009.

[30] F. Tombari, S. Mattoccia, L. Di Stefano, E. Addimanda, Classification and evaluation of cost aggregation methods for stereo correspondence. IEEE International Conference on Computer Vision and Pattern Recognition, pp. 24-26, 2008.

[31] F. Tombari, S. Mattoccia, and L. Di Stefano. Segmentation based adaptive support for accurate stereo correspondence. In *Proc. Pacific-Rim Symposium on Image and Video Technology*, 2007.

[32] K-J. Yoon and I.S. Kweon. Adaptive support-weight approach for correspondence search. IEEE Transactions on Pattern Analysis and Machine Intelligence, 28(4):650-656, 2006.

[33] O. Veksler. Fast variable window for stereo correspondence using integral images. In Proc. Conf. Computer Vision and Pattern Recognition, pp. 556–561, 2003.

[34] S. Kang, R. Szeliski, and J. Chai. Handling occlusions in dense multi-view stereo. In Proc. Conf. Computer Vision and Pattern Recognition, pp. 103–110, 2001

[35] A. F. Bobick and S. S. Intille. Large occlusion stereo. International Journal of Computer Vision 33(3):181–200, 1999.