Fiducial Marker Detection Using FPGAs

Peter Samarin, Kenneth B. Kent, Rainer Herpers, and Timur Saitov

TR-13-227, September 18, 2013

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
fcs@unb.ca
http://www.cs.unb.ca

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Autonomous Systems

Copyright © 2013 by Peter Samarin

SUPERVISORS:
Rainer Herpers
Kenneth B. Kent
Timur Saitov

LOCATION:
Sankt Augustin

ABSTRACT

This work presents an approach for computing camera pose from images of fiducial markers. The processing is shared between an FPGA and a PC: the FPGA converts grayscale camera images into binary format and transfers them to the PC; and the PC detects fiducial markers in the received binary camera images and computes the position and orientation of the camera.

The approach is evaluated in terms of camera pose error and processing time by using a 3D simulator and subsequently tested in our Immersion Square environment. The 3D simulator has been developed in course of this work in order to enable a thorough theoretical evaluation. It is used to generate a large number of test images from different camera positions by simulating a subset of the Immersion Square. Synthetic images generated in this way undergo a set of transformations in order to make the evaluation closer to images obtained from a real camera by using Gaussian blur of different kernel sizes, and several types of noise.

The results of theoretical evaluation show that depending on the chosen parameters, a relatively high frame rate of approximately 60 fps can be achieved. Moreover, the parameters allow for a trade-off between speed, accuracy, and precision.

The insights gained from the theoretical evaluation are used to develop an optimized system that has been subsequently tested in the real Immersion Square. The tests show that the optimized system runs with the average of approximately 80 fps and is capable of achieving higher frame rates of 100 fps. It displays better accuracy, and higher precision than the unoptimized version. In addition, the real world tests show that the system produces very stable crosshairs, such that it can be used for interacting with the Immersion Square in applications where precise pointing is required.

CONTENTS

1 INTRODUCTION 2

2 FIDUCIAL MARKERS IN COMPUTER VISION 4
2.1 ARToolKit and ARToolKitPlus 5
2.2 ArTag 6
2.3 SVMS 7
2.4 TRIP 8
2.5 Fourier Tag 10
2.6 RUNE-Tag 11
2.7 Pi-Tag 12

3 METHODS 14
3.1 Marker Detection 14
3.2 Corner Refinement 16
3.3 Marker Analysis 17
3.4 Camera Pose Estimation 22

4 APPROACH 25
4.1 Working Approach 25
4.2 Initial Approach 27

5 SETUP 37

6 EVALUATION 38
6.1 Performance Variables 38
6.2 Evaluation Environment 39
6.3 Dataset Generation 41

7 RESULTS 52
7.1 General Trends 52
7.2 Evaluation of Methods 56
7.3 Optimized Implementation 57

8 CONCLUSIONS 61

BIBLIOGRAPHY 63

A APPENDIX 67
A.1 Blurry Dataset 67
A.2 Dataset with Gaussian Noise 70

1 INTRODUCTION

One of the tasks in the MI6 project is to precisely estimate position and orientation of a 6-DoF device that is used to interact with the Immersion Square. In previous work a unique pattern of infrared light spots was projected from the back of the Immersion Square [27, 24]. The 6-DoF device was equipped with a camera and an FPGA. The FPGA computed the coordinates of light spots' centers and transferred them to the PC, and the PC matched the received coordinates with the known pattern of light spots by using the Levenberg-Marquardt algorithm and estimated the pose of the camera.

However, experimental results have shown that using the light spots might not be the best way to accurately compute camera pose. There are several problems with using light spots. *First*, all light spots look the same, which requires computation-intensive pattern matching in order to find the ID of each light spot before the camera pose can be computed. *Second*, as discussed in [27], small errors in the calculated centers of light spots resulted in significant errors during camera pose estimation. *Third*, the Levenberg-Marquardt algorithm that was used to find the IDs of the light spots has a high demand on processing power, which limits the amount of devices that are able to operate in the Immersion Square at the same time.

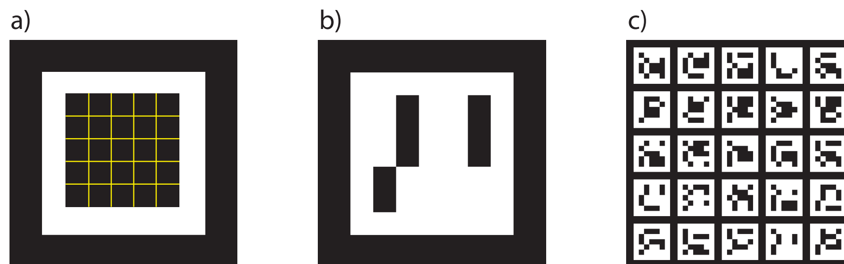


Figure 1.1. a) structure of a fiducial marker, b) an example of a fiducial marker with ID 647, c) a grid of 5x5 markers. The markers were generated by using the ArUco library [19].

To address the problems that result from using light spots, in this project the light spots are replaced by fiducial markers. A fiducial marker is represented by an image that provides enough information to calculate camera pose from only one marker. All fiducial markers have their unique ID numbers. The structure of a fiducial marker is shown in Fig. 1.1 a). The

marker is drawn on a black background and is comprised of a white edge, inside of which a 5x5 binary pattern encodes the ID of the marker. For example, the pattern of the marker in Fig. 1.1 b) corresponds to the number 641:

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Fiducial markers have several advantages over simple light spots. 1) Since each marker has a unique ID, seeing one marker is enough for camera pose computation. 2) The pattern inside of each marker is constructed in a way that allows error correction—even when the images are noisy, it is possible to reconstruct an ID that was not found in the data base. 3) Using a grid of markers introduces additional points that can improve the accuracy of camera pose computation. 4) Fiducial markers reduce the amount of processing required, because there is no need for computation-intensive pattern matching necessary to compute camera pose from light spots. 5) In previous work [24, 4], camera pose estimation was done on the PC, while only minor processing was done on the FPGA. This approach has been adapted in this work to perform fiducial marker detection.

This thesis is organized as follows. The next section reviews the current state of the art fiducial marker detection. It presents several marker systems and discusses their advantages and disadvantages. Section 3 explores the algorithms that are used for detecting fiducial markers whose shape is a square. Section 4 presents the approach that has been developed in the course of this project. Section 5 discusses the evaluation strategy that has been taken to measure the performance of the working system. Section 6 presents the result results of evaluation. The last section—section 7, concludes this thesis and gives directions for future work.

2 FIDUCIAL MARKERS IN COMPUTER VISION

Fiducial markers are artificial landmarks added to a scene to enable precise computation of position and orientation of some objects of interest. In computer vision, fiducial markers are used in situations when precise camera pose estimation is required, but no distinct features are present in the environment that would enable its unambiguous computation. Such situations often arise in human-made environments, where many similar-looking objects present. Fiducial markers allow robust computer vision by decreasing ambiguity that results from using natural features [11].

Fiducial markers are similar to barcodes, however, they are designed with the purpose of providing spacial information. Barcodes are primarily designed to provide information about the product. They do not provide enough information needed to calculate the exact position and orientation of the observer. Another problem of barcodes is the relatively dense amount of information that they contain, which can only be read by being relatively close to the marker. The planar markers solve a different problem to the fiducial marker systems—to provide information about the product at hand. However, several techniques from the design of non-fiducial markers, such as checksum and error correction are used in fiducial marker systems as well.

Many fiducial marker systems are available in the literature. The systems differ with each other in terms of the shape and the way how information is encoded in the data parts of the markers. Also, the marker systems differ in terms of the computer vision algorithms that they use. However, all systems are similar in the following:

1. Each marker uses a simple geometric shape as its distinguishing feature that allows the marker to be easily detected amidst all the other objects that might be present in the environment.

2. Each marker has a distinct id that is encoded somewhere in the marker region.

The most often used shapes are circular and rectangular shapes, however, other shapes are also used sometimes.

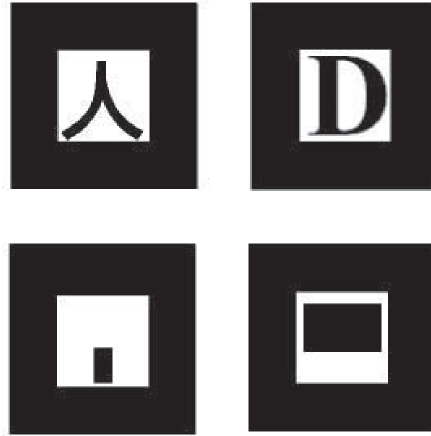


Figure 2.1. Examples of possible ARToolKit markers.

2.1 ARTOOLKIT AND ARTOOLKITPLUS

ARToolkit [15] is an open source marker system that uses a rectangular shape as its distinguishing feature and user-defined code that consists of an image. In order to decode it, correlation between a marker candidate and all markers in the library is computed. The marker is considered as detected if the correlation reaches a certain threshold. The marker code has been extended by [20] that makes it possible to use Fourier encoding [21].

As pointed out by [10], there is a problem of template markers that results from the fact that the information stored in the templates is highly redundant. Thus, each marker has a high chance of being recognized as another one. In addition, template matching is more computationally expensive than comparison of numbers and error correction.

ARToolKitPlus was developed by [33] to address some drawbacks of the ARToolKit marker system. It extends it by improving the detection approach with an addition of adaptive thresholding and vignetting. It also drops the use of correlation when checking the content of the marker. Instead, it assigns digital IDs to each marker. This improvement was inspired by the approach taken in the ARTag marker system [10]. The resulting marker system is very similar to ArTag. 4096 IDs are possible in the system. The internal field of the marker is coded using forward error correction (CRC).

ARToolKitPlus uses adaptive thresholding to detect marker boundaries. After one marker has been detected, the threshold is updated by taking the median of all extracted marker pixels and used as a threshold for detecting further markers. If no marker is detected, a random threshold is used.

2.2 ARTAG

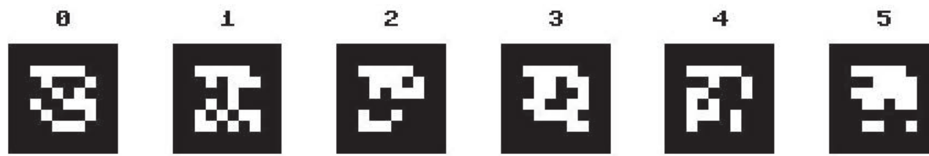


Figure 2.2. Examples of possible ArTag markers.

ARTag is a bitonal marker system that consists of a square border and a 6x6 interior code that stores the ID of the marker [10]. It was designed to overcome the shortcomings of ARToolkit. The interior code allocates 10 bits for the sub-ID, 16 bits for a checksum, and 10 bits for error correcting code, that allows correction of 2 falsely read bits. In this configuration, the library has 1001 unique markers, or 2002 markers if inverted markers are allowed—the 1001 unique markers can be drawn with white border on black background as well with black border on white background, which doubles the number of the markers in the library. The set of possible markers was reduced from 1024 to 1001 by removing 23 markers that greatly reduced the average Hamming distance of the marker set. Each marker is distinct from another. It also includes the rotational uniqueness, which means, that a rotated marker never has the same pattern as another marker from the library.

ARTag was inspired by the ARToolkit in its use of quads as shape, and by Datamatrix for its use of error correction [11]. First, the 10-bits code XOR-ed with a fixed 10-bits mask. This is done in order to prevent generation of all-black and all-white markers. In this way, the markers can be distinguished from random squares in the images. Next, a checksum is computed from the masked bit string. Subsequently, error-correcting bits are added to the bit string, altogether making a 36 bit code.

ARTag library comes with a description of an algorithm for robust marker detection. Marker detection stage directly influences the false negative rate of the markers. In case the algorithm is not robust, valid markers will be not recognized as such. The algorithm provided by ARTag is based on edge detection. It was explicitly decided against using a threshold, because the threshold increases the sensitivity of the algorithm in different lighting conditions and camera focal lengths. A global threshold might miss regions of images with local lighting irregularities.

The creators of ARTag suggest that finding quad edges is better done by finding connected line segments. The intersections of the lines are used to find corners that provide the four points necessary to compute perspec-

tive distortion. In this way, the marker can be rectified, which enables accurate reading of the 6x6 interior bit pattern. After the analysis of the ID of the marker, the rotation of the marker can be determined, and the homography of the marker can be updated with this information.

The library is constructed with the goal of providing robust markers that can be reliably used under different lighting conditions. Both stages—the stage of marker recognition and identification / error-correction—have well known properties. For example, the first 50 markers in the library have a hamming distance of 12, in order to reduce inter-marker confusion rate. The minimum marker size is 13 pixels. The false positive rate was estimated to be less than 0.0039% of the quadrilaterals found in the image [12]. The processing time is reported to be in the range of 10-50 ms depending on the number of markers in the image.

In [11], the ARTag marker system is compared to the ARToolkit Plus marker system. It turns out that the ARTag marker system is more robust in both—the marker detection stage and the ID verification stage. When detecting the marker, ARTag proposes to use edge detection algorithm to find potential quad positions. It is more robust under varying lighting conditions, especially when the light is spread over the image non uniformly. During the phase of marker verification, the ARTag marker system is more robust in terms of false-positive and inter marker confusion because it was designed with the goal of maximizing the Hamming distance between the markers. The average Hamming distance between the ARTag markers is higher than that of the ARToolkit Plus, which improves these two criteria in noisy images. The verification robustness of the two marker systems has been evaluated under varying noise conditions.

2.3 SVMS

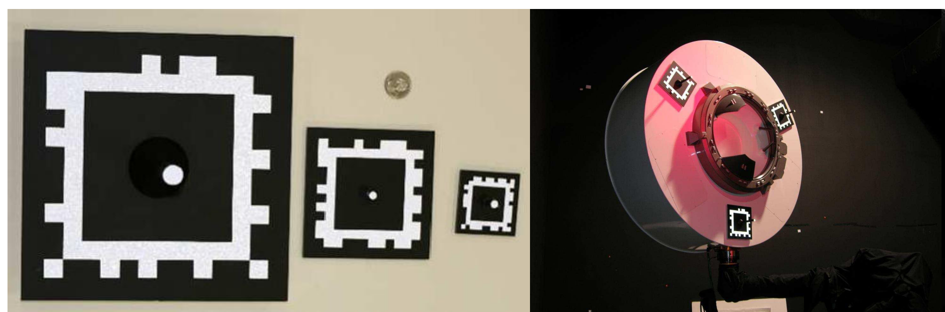


Figure 2.3. Examples of SVMS markers (left), and their application in space program (right).

SVMS is a fiducial marker system that was developed for aerospace operations, where precise pose calculation is crucial for fail-safe operation [5]. A typical application of the SVMS markers is during the docking of two space ships. The markers can also be used as a help for a robot performing repairs of a space ship.

SVMS markers are white squares with a black internal part that is also square. The ID of the marker is encoded in the outer part of the white square. Each side can hold 11 bits, which makes it 44 bits to represent the ID and provide some error correction capabilities. Error correction code is used to ensure low inter-marker confusion rate. There is a tradeoff between the total number of markers and marker redundancy. SVMS uses the BCH encoding for error correction. In the version described in the paper, 27 bits are used for error correction, which allows recovery from 5 wrong bits that can be caused by occlusion, noise, and bad illumination. This leaves 17 bits, and after removing symmetrical patterns, 15 bits for marker ID, which makes $2^{15} = 32768$ unique IDs.

Marker detection starts by performing edge detection and connecting the edge points into chains. The chains are segmented into straight line segments, which are then grouped into quadrilaterals using proximity of their end points and segment lengths. The intersections of the lines of a quadrilateral are used to compute a homography matrix that is used to rectify the image and read the ID of the marker. Those points are the corners of the marker. In the next step, the corners of the marker are used to compute perspective projection, which is used to rectify the marker. Subsequently, data bits are obtained by sampling the intensity of the marker. The data bits are decoded to compute the ID of the marker, or to reject the marker with an invalid ID. Initially, marker pose is computed using the four corners of the marker, its known size and the calibration matrix of the camera. The pose is refined using the point in the center. One of the drawbacks of the SVMS markers is that they are made from special materials that reflect the light back to its source.

2.4 TRIP

TRIP (target recognition using image processing) is a circular bitonal marker system that has a bull’s eye in its center [17]. In this marker system, the information is encoded in two concentric rings that are divided into 16 sectors. The first sector, which is called the synchronization sector, is used to simplify information extraction from the two data rings. This is the only sector that has two lines that go from marker’s center to the outer borders

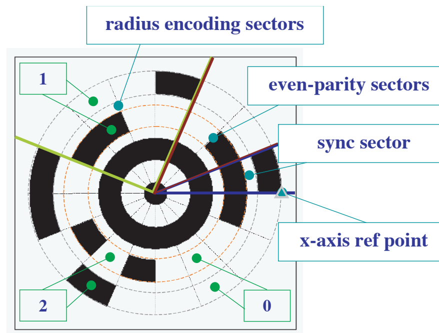


Figure 2.4. Examples of a TRIPTAG marker.

of the second ring. The point defined by marker's border and the first such line is also used in positional computation.

The next two sectors (going counter-clock wise) stores the parity that reduces the ratio of falsely-recognized markers. Subsequent four sectors encode the radius of the bull's eye in centimeters. The remaining 9 sectors are used to encode the ID of the marker. All data is encoded using ternary code—"0" corresponds to both rings of a sector being white, "1" corresponds to black sector of the inner ring, and white sector of the outer ring, "2" corresponds to a white sector of the inner ring, and a black sector of the outer ring. This configuration provides $3^9 - 1 = 19,683$ possible markers. However, other configurations, are also possible.

Adaptive thresholding is used to binarize the image. Adaptive thresholding is a robust way to do thresholding under various lighting conditions. The edges of the binary image are extracted by using simplified edge detection. Adjacent edge points are grouped together into chains. A simple heuristic is used to drop chains that are not likely to be ellipses. In the next step, least-squares ellipse fitting is performed in the remaining chains. Concentric ellipses, that is, ellipses with the same center are considered to belong to the same marker.

In the next step, the code of the marker is deciphered by sampling the pixels of the binary image based on the parameters of the ellipses. For each valid marker, the algorithm extracts the data stored in the marker—bull's eye radius and the ID of the marker. Bull's-eye alone is not enough to compute the pose of the marker. To compute the pose, the camera image is rotated along two axes until the ellipse becomes a circle. Two possible homographies result from the two rotations. In the next step, the outer point in the synchronization sector is used to disambiguate the actual camera pose.

2.5 FOURIER TAG

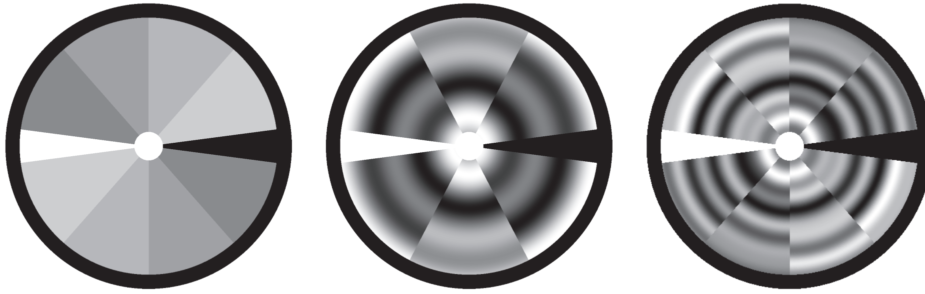


Figure 2.5. Layout and two examples of FourierTag markers.

This marker system was developed due to unsatisfactory robustness of ARTag under gloomy conditions that often arise underwater [25]. It has been improved in [34] by increasing the amount of data that each marker can hold, and by adding features that allow computation of camera pose from a single marker.

Fourier tag has a thin black outer ring, and a white spot in the center. A black and a white sector opposite of each other carry rotational information. They are the so-called *alignment sectors*. The digital data of the markers is encoded in the frequency domain by using sinusoidal patterns to encode bits of information. High frequencies are used for the least significant bits, and low frequencies for the most significant bits. In this way, the amount of information gracefully diminishes with increasing distance of the camera from the marker, because the information encoded in high-frequencies carries low importance.

The marker is broken down into several sections. Every two opposing sections encode the same data. During marker detection step, Sobel edge detection generates the gradient magnitudes and their directions in a gray scaled image. To detect the center of the marker, the gradients for each pixel are traced across the whole image. The lines are weighted by their corresponding gradient and collected in the "Hough accumulator map". The center of the marker is found by thresholding the obtained image and finding the largest cluster. The coordinates of the center are refined by searching the radial symmetry. The gradient image is binarized by using adaptive threshold. Edge elements whose gradients are aligned with the approximately estimated center point of the marker, are selected for the next stage. The border of the marker is obtained by searching for the edge elements starting from the hypothetical center point towards several angles by looking for the largest connected group. An ellipse fitting pro-

cedure is applied to the obtained edge elements. Fourier transform along the line brings back the encoded number.

Multiple rays from the marker's center are sent outwards at different angles, looking for white only and black only sectors along them. Upon finding the sectors, the intersection of the centers will be the actual center of the marker. The displacement between the actual center and the displaced center can be used to find the 3D position of the marker relative to the camera if the radius of the marker is known as well.

The digital number that is stored in the marker is obtained by first finding all the sectors of the marker. In the next step, each sector is sent processed by the fast Fourier transform. Amplitude and phase of specific frequencies are checked.

According to the results presented in the paper, the distance under which the marker can be successfully recognized is much larger for Fourier tag than for ARTag. However, it has worse performance when it comes to inter-marker confusion. It is easier to confuse the Fourier tag markers with each other than it is to confuse the ARTag markers.

2.6 RUNE-TAG

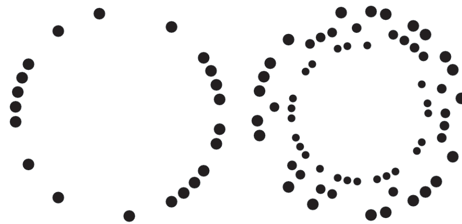


Figure 2.6. Examples of RUNE-43 (left) and RUNE-129 (right) markers.

The RUNE-Tag marker system takes advantage of some properties of projective transformation [2]. RUNE-Tag markers consists of a set of circular points arranged along rings that have the same center. Each marker can have several rings like that. 43 non-touching points can be drawn along each ring.

The authors present two different types of marker systems—RUNE-43 and RUNE-129. RUNE-43 uses a single ring of 43 points that provides 762 different markers with a minimum Hamming distance of 13 which allows error corrections of up to 6 bits. In RUNE-129, each marker consists of three concentric rings, each holding 43 bits of information. This marker system provides 19152 markers with a minimum Hamming distance of 30, which allows 14 errors in the marker to be corrected.

In order to recognize the markers, the proposed algorithm first detects all ellipses in the image. In the next step, the algorithm pairwise considers all ellipses and tries to transform them into circles. Under the same transformation, the circles on the same ring should have the same radius. Thus, ellipses with the same radius are considered to be on the same ring of a marker. Usually, two potential rings can be drawn through two points. The algorithm then searches for other ellipses along the two rings and eliminates the ring where no other ellipses are found. The algorithm iterates over the set of all ellipses and assigns them to their corresponding markers.

After identifying all points that belong to a marker and decoding the ID of the marker, the centers of the points are used to find the camera pose. The authors solve this problem by using OpenCV's `solvePnP()` procedure. The procedure takes a set of points in image coordinates, their corresponding points in the world coordinates and the camera matrix that holds camera focal length and produces the estimated camera pose. The algorithm uses Levenberg-Marquardt optimization that minimized reprojection error.

The RUNE-Tag marker system performs better than the ARToolkit and ARToolkit Plus marker systems in terms of accuracy. However, it needs more processing power, because the time to recognize 10 RUNE-129 markers is around 150 ms. The system performs badly when the camera angle with respect to the marker becomes far away from 90°, and also when markers are moderately far from the camera. This is due to the usage of relatively small circles. In addition, the system has not been evaluated in terms of inter-marker confusion rate.

2.7 PI-TAG

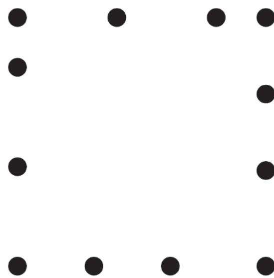


Figure 2.7. Example of a Pi-Tag marker.

Pi-Tag is a fiducial marker system that uses circles arranged along an edge of a square as markers [3]. It is based on four invariant properties of projective geometry. In the first step, the image is thresholded by us-

ing a locally adaptive threshold, as described in [26]. Adaptive threshold makes subsequent image processing more robust to varying lighting conditions. In the next step, all ellipses are found in the binary image by using OpenCV ellipse detector[1]. Ellipses that belong to the same marker are grouped together by using the invariant property of lines in projective geometry—lines are preserved after perspective distortion. Thus, ellipses on the same edge of the marker will be on the same line in images distorted by perspective projection. The algorithm searches for corner ellipses, which are ellipses on the same line that have exactly two other ellipses between them. The search is done by considering all found ellipses two at a time with each other. This requires $O(n^2)$ comparisons for n found ellipses.

After finding two corners, the search continues looking for the third corner in the remaining set of ellipses. If the third corner is found, the algorithm checks the cross ratio between the 4 collinear points of the two sites found. If the cross ratio of both sides is equal to each other, then the sides are identical. For two sides with different patterns, the cross-ratio is equal to a known constant σ.

The system has been shown to be more accurate than ARToolkit and its more recent version ARToolkit Plus. However, the time of processing the full computer vision pipeline from start to finish is reported to be lower than that of ARToolkit—between 10 ms (without false ellipses in the image) to 150 ms with several false ellipses. The authors do not test evaluate their system to find important properties of their system. For example, it would be interesting to know how the system behaves when there are many markers in the scene—how do slight inaccuracies in recognition of ellipse centers influence the inter-marker confusion rate? The authors do not mention how big is the library size of their fiducial marker system.

The authors mention that the system suffers from low detection rate when the markers are not close to the camera (no distance is given), which makes the ellipses very small, so that the proposed algorithm is unable to detect them. Another problem arises when the angle between the camera and the marker plane far away from 90°.

1 There are several off the shelf OpenCV procedures that can detect ellipses (e.g., `cv::HoughCircles`, `cv::fitEllipse`) , however, the authors do not tell which procedure they use.

3 METHODS

This section describes algorithms for detection of ArUco markers and subsequent camera pose estimation in a bottom-up manner. First, methods for local image analysis are introduced. These methods consider individual pixels. The subsequent batch of methods describes how the individual pixels can be grouped together to potential marker candidates. Each potential marker candidate is analyzed and the binary code of each proper marker is extracted. In the next step, the markers are considered altogether in order to estimate the camera pose. Several methods for controlling precision of camera pose estimation are introduced in form of corner refinement methods.

3.1 MARKER DETECTION

An ArUco board with markers has three distinct features: 1) the board itself is black, 2) each marker is a square, 3) each square has a white border. These features must be detected in order to extract fiducial information from an image of a board with markers. The first step marker detection is to binarize the image. Image binarization is an important step for identifying image regions that might contain potential markers. There are several ways to perform image binarization. Here we consider two of them—fixed thresholding and Canny edge detection.

3.1.1 *Image Binarization*

During fixed thresholding, the intensity value of every pixel in the image is compared to a fixed value—the threshold. Pixels that exceed that fixed value are considered to be foreground pixels, otherwise, the pixel is a background pixel. The resulting image contains zeroes and ones and needs only two bits to represent pixel intensities. A fixed threshold performs well under controlled environmental conditions, such as uniform lighting, no noise or blur in the camera image. Canny edge detection divides all pixels in an image into two classes—pixels that belong to an edge or pixels that do not belong to an edge [7]. Canny is more robust to the lighting conditions but is more computationally intensive.

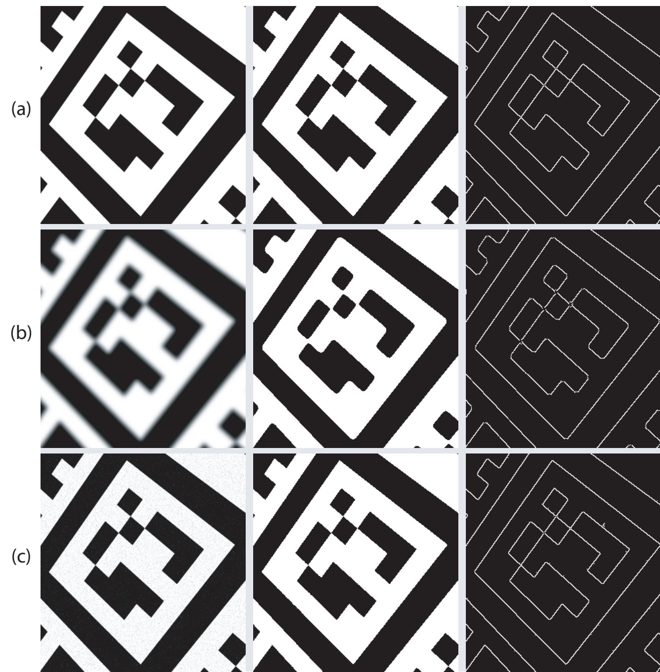


Figure 3.1. Binarization of a test image. Left column shows original images, central column shows binarized image with threshold of 100, right column shows application of Canny edge detection with first threshold equal to 100, and second threshold equal to 255. (a) Undistorted test image. (b) A test image blurred by a Gaussian kernel with a size of 19x19 pixels. (c) Test image with soft Gaussian noise with standard deviation of 15, and mean centered around each processed pixel.

3.1.2 *Blob Detection*

The foreground regions of the binary image are grouped together by finding contours [31]. A contour is a list of points that represents a curve [6, p. 234]. Each contour consists of a set of points that follow the blobs along the borders. In some situations, a hierarchy of contours can arise if there is a region of foreground pixels that inside of an already found contour that is decoupled from them by a region of background pixels. The exterior contour follows the edge of the blob, whereas the interior contour follows the *holes* along the interior border to the background pixels. White pixels are the foreground pixels, whereas black pixels are the background pixels.

After finding contours, it might happen that a polygon has only a few points. Such contours are unlikely to be markers and are deleted from the set of potential marker candidates. There are several situations where small contours can arise. Large numbers of small contours are usually detected in noisy images. Some markers are assembled in such a way that they contain small rectangles in their binary code. They will be detected as contours inside of a contour, but are not markers. To increase process-

ing time, small contours, and inner contours are excluded from further consideration.

The contours that are big enough are approximated to polygons with a small number of edges by following the algorithm described in [22, 9]. The algorithm iteratively approximates the smaller polygon as follows:

1. First, the two most distant points in the contour are found and connected by a straight line.
2. Another point that is the most distant from the line is added to the polygon.
3. The algorithm goes back to 2. but now checks the distance of the contour points to *all lines* in the approximated polygon.

The algorithm stops when the approximation is good enough, that is, when the distance between any point the contour and the approximated polygon is smaller than some fixed parameter that controls the precision of approximation.

Before a polygon can be considered as a rectangle, it needs to pass some conditions. One is that it should have exactly four points. If the contour originates from a rectangle, the polygon approximation algorithm will likely find only four points. But in some cases a marker is only partially present in the camera image, and the approximated polygon will not have exactly four points. In this case, the polygon is ignored. In addition, the polygon must be convex, otherwise it is not a rectangle. In the next step, the polygons are pairwise considered to find those that are too close to each other. Upon finding two polygons that are too close, the polygon with the smaller perimeter is discarded.

3.2 CORNER REFINEMENT

The four points computed by the contour approximation represent the corners of the rectangles as integer values. However, precise camera pose computation requires corner points in subpixel range. There are several ways to achieve subpixel precision for the corner points.

One method to refine corners is to fit lines into the contour and to find the intersections of the lines. At first, the four points from polygon approximation are used to break the contours into four sets of points. For each set of points a line equation is estimated that best fits the points, that is, a line that minimizes the squared distance between all points and the fitted line. This is done by using singular value decomposition that solves over-determined set of equations.

Fig. 3.2 shows a set of points sampled from a line equation $y = -x + 90$, shown by the solid black line, that has been distorted by Gaussian noise with standard deviation equal to 10. The line is fit by using linear least squares. Line fitting results in equation $y = -0.9566 \cdot x + 88.2597$, which is shown by the dashed gray line.

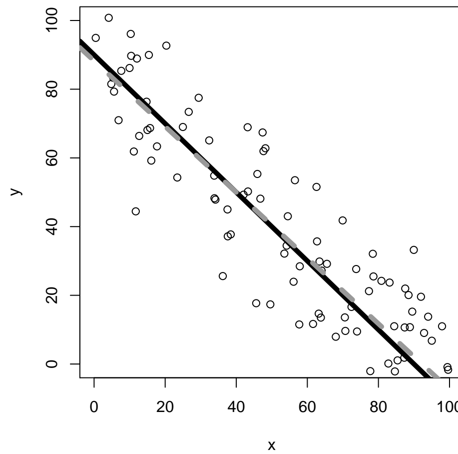


Figure 3.2. An example of using linear least squares to fit a line into a set of points.

To obtain corner coordinates, the intersections of adjacent lines are computed. Each intersection is a set of two linear equations with two unknowns. The contour cleaning steps described above ensure that the obtained equation can be solved exactly.

3.3 MARKER ANALYSIS

During the marker analysis step, the id of the marker is obtained by reading the binary code from the interior part of the marker. The next step decides whether the id represents a valid marker by searching through a database that contains all marker ids. Ids that are found in the database are kept for further processing, and the other IDs are discarded. This section describes the computational steps performed during marker analysis.

3.3.1 *Marker Rectification*

Each ArUco marker is a square that contains a white edge and a 5x5 bit binary code represented by black and white rectangles. A naive approach for reading the binary code could be to divide each line into seven equal parts and to connect points in the opposing lines, as shown in Fig. 3.3. This approach works well for images where the board with markers is

viewed from a distance and angles near to 90°, as illustrated on the left. However, in many situations the board is distorted by a perspective, such that the application of the naive approach will result in a large portion of ill-recognized markers.

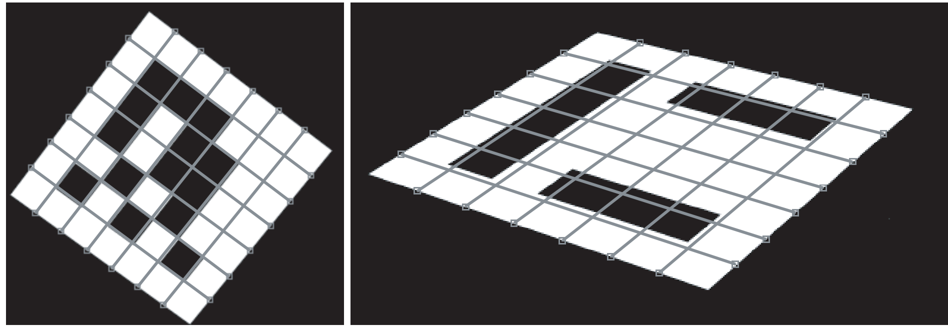


Figure 3.3. A naive approach to marker identification. Each side of the marker is divided into 7 equal parts. On the left, a marker is viewed from an angle close to 90° and a moderate distance. On the right, a marker is viewed from a steep angle and a short distance.

A more intelligent approach is to first undo the distortion caused by perspective and to read out the ids from rectified images. To rectify the image, the coordinates of the corners of the markers obtained in the previous step are used in combination with the knowledge that each marker is a square. It is not necessary to know which marker it is in order to rectify it. Given four corners of a marker candidate in pixels, we can map them to the origin of coordinates (0,0,1), (0,W-1,1), (W-1,W-1,1), and (W-1,0,1), where W is the *warp size*—the size of the square after rectification. The last "1" in the coordinates of each point is not the Z-coordinate, but instead an indicator that we are using homogeneous coordinates in a projective space [1]. Homogeneous coordinate system is a generalization of Euclidean space that allows representation of points and lines at infinity. In what follows, the material covered in [13, pp. 88–91] is explained.

Consider a set of points $\mathbf{x}'_i$ and the corresponding projections in the camera image plane $\mathbf{x}_i$. Perspective transformation (also called *homography*) from a 2D plane into another 2D plane is a nonlinear mapping $\mathbf{x}'_i \longleftrightarrow \mathbf{x}_i$, which is achieved by multiplying all points in one plane by a 3×3 matrix H:

1 From [13, p. 2], we know that in the projective space, each point in a 2D Euclidean space with coordinates (x, y) is extended to (x, y, 1). This new coordinate represents the same point in the projective space. The same point can be represented by (2x, 2y, 2) or, in general, by (kx, ky, k) for any non-zero k. Thus, the points are represented by *equivalence classes* of coordinate triples where two coordinates are equivalent if they differ by a common multiple. Such coordinates are called *homogeneous coordinates*. To obtain the original coordinates of the Euclidean space from homogeneous coordinates (xk,yk,k), we need to divide the homogeneous coordinates by the factor k.

$$\mathbf{x}_i' = H\mathbf{x}_i \tag{3.1}$$

with H equal to

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix}$$

Since $\mathbf{x}_i'$ and $\mathbf{x}_i$ are represented by homogeneous coordinates, we cannot bring the right hand side of the equation to the left hand side and set it equal to zero. Instead, we know that the points are on the same four line, and their cross-product is zero:

$$\mathbf{x}_i' \times H\mathbf{x}_i = \mathbf{0}$$

where $\mathbf{0}$ is a 3x1 vector of zeroes.

The next steps work towards making the equation of the form $A\mathbf{h} = \mathbf{0}$ where $\mathbf{h}$ is a 9x1 vector reshaped from the 3x3 homography matrix H. The first step is to rewrite the mapping $H\mathbf{x}_i$, assuming that $\mathbf{x}_i = (x_i, y_i, w_i)$:

$$H\mathbf{x}_i = \begin{pmatrix} h_1 x_i + h_2 y_i + h_3 w_i \\ h_4 x_i + h_5 y_i + h_6 w_i \\ h_7 x_i + h_8 y_i + h_9 w_i \end{pmatrix} = \begin{pmatrix} \mathbf{h}^{1T}\mathbf{x}_i \\ \mathbf{h}^{2T}\mathbf{x}_i \\ \mathbf{h}^{3T}\mathbf{x}_i \end{pmatrix} \tag{3.2}$$

where $\mathbf{h}^{jT}$ for j=1,...,3 are the rows of matrix H. Combining Eq. (3.1) and Eq. (3.2), we rewrite the cross product as:

$$\mathbf{x}_i' \times H\mathbf{x}_i = \begin{pmatrix} y'\mathbf{h}^{3T}\mathbf{x}_i - w'\mathbf{h}^{2T}\mathbf{x}_i \\ w'\mathbf{h}^{1T}\mathbf{x}_i - x'\mathbf{h}^{3T}\mathbf{x}_i \\ x'\mathbf{h}^{2T}\mathbf{x}_i - y'\mathbf{h}^{1T}\mathbf{x}_i \end{pmatrix} \tag{3.3}$$

where $\mathbf{x}'_i = (x'_i, y'_i, w'_i)^T$. This follows from the definition of cross product—consider two vectors $\mathbf{u} = (u_1, u_2, u_3)$ and $\mathbf{v} = (v_1, v_2, v_3)$. The crossproduct between $\mathbf{u}$ and $\mathbf{v}$ is:

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix} = (u_2v_3 - u_3v_2)\mathbf{i} - (u_1v_3 - u_3v_1)\mathbf{j} + (u_1v_2 - u_2v_1)\mathbf{k}$$
$$= \begin{pmatrix} u_2v_3 - u_3v_2 \\ u_1v_3 - u_3v_1 \\ u_1v_2 - u_2v_1 \end{pmatrix}$$

where $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are the standard basis vectors that are unit vectors directed towards the X, Y, and Z axes, respectively.

Since $\mathbf{h}^{j\,T}\mathbf{x}_i$ is the same as $\mathbf{x}_i^T\mathbf{h}^j$ for $j = 1, ..., 3$, the 3x1 vector from Eq. (3.3) can be extended with zeroes and converted into a 3x9 matrix by bringing all indices of H outside of the matrix:

$$\mathbf{x}'_i \times H\mathbf{x}_i = \begin{pmatrix} y'\mathbf{h}^{3T}\mathbf{x}_i - w'\mathbf{h}^{2T}\mathbf{x}_i \\ w'\mathbf{h}^{1T}\mathbf{x}_i - x'\mathbf{h}^{3T}\mathbf{x}_i \\ x'\mathbf{h}^{2T}\mathbf{x}_i - y'\mathbf{h}^{1T}\mathbf{x}_i \end{pmatrix} = \begin{pmatrix} y'\mathbf{x}_i^T\mathbf{h}^3 - w'\mathbf{x}_i^T\mathbf{h}^2 \\ w'\mathbf{x}_i^T\mathbf{h}^1 - x'\mathbf{x}_i^T\mathbf{h}^3 \\ x'\mathbf{x}_i^T\mathbf{h}^2 - y'\mathbf{x}_i^T\mathbf{h}^1 \end{pmatrix} =$$
$$= \begin{pmatrix} 0 - w'\mathbf{x}_i^T\mathbf{h}^2 + y'\mathbf{x}_i^T\mathbf{h}^3 \\ w'\mathbf{x}_i^T\mathbf{h}^1 + 0 - x'\mathbf{x}_i^T\mathbf{h}^3 \\ -y'\mathbf{x}_i^T\mathbf{h}^1 + x'\mathbf{x}_i^T\mathbf{h}^2 + 0 \end{pmatrix} =$$
$$= \begin{bmatrix} \mathbf{0}^T & -w'_i\mathbf{x}_i^T & y'_i\mathbf{x}_i^T \\ w'_i\mathbf{x}_i^T & \mathbf{0}^T & -x'_i\mathbf{x}_i^T \\ -y'_i\mathbf{x}_i^T & x'_i\mathbf{x}_i^T & \mathbf{0}^T \end{bmatrix} \begin{pmatrix} \mathbf{h}^1 \\ \mathbf{h}^2 \\ \mathbf{h}^3 \end{pmatrix}$$

where $\mathbf{0}^T$ is a 1x3 row vector containing zeroes and $\mathbf{h}$ is a 9x1 vector that contains the elements of H: $\mathbf{h} = (h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9)^T$. From Eq. (3.1), we know that the cross product of the vectors on the same line is equal to zero. Thus, the matrix above is equal to a 3x1 zero vector:

$$\begin{bmatrix} \mathbf{0}^T & -w'_i\mathbf{x}_i^T & y'_i\mathbf{x}_i^T \\ w'_i\mathbf{x}_i^T & \mathbf{0}^T & -x'_i\mathbf{x}_i^T \\ -y'_i\mathbf{x}_i^T & x'_i\mathbf{x}_i^T & \mathbf{0}^T \end{bmatrix} \begin{pmatrix} \mathbf{h}^1 \\ \mathbf{h}^2 \\ \mathbf{h}^3 \end{pmatrix} = \mathbf{0}$$

These equations have the desired form of $A_i\mathbf{h} = \mathbf{0}$, with A_i being a 9x3 matrix. The third equation is linearly dependent on the first two equa-

tions and will not contribute to the solution. Therefore, it is usually not considered, and the above equation is reduced to:

$$\begin{bmatrix} \mathbf{0}^\mathsf{T} & -w_i'\mathbf{x}_i^\mathsf{T} & y_i'\mathbf{x}_i^\mathsf{T} \\ w_i'\mathbf{x}_i^\mathsf{T} & \mathbf{0}^\mathsf{T} & -x_i'\mathbf{x}_i^\mathsf{T} \end{bmatrix} \begin{pmatrix} \mathbf{h}^1 \\ \mathbf{h}^2 \\ \mathbf{h}^3 \end{pmatrix} = \mathbf{0} \tag{3.4}$$

Thus, each pair of points produces two equations. In order to solve for all the parameters of H, eight equations are required.[2] To solve for H, a common approach is to use singular value decomposition. However, it is also possible to solve for H by using Gaussian elimination by assuming that one of the elements of H, e.g. h_9, is equal to 1. The drawback of this approach is that the results will be unstable when the true value of h_9 is close to zero or equal to zero. It is equal to zero when the point at infinity is mapped to the origin of coordinates [13, p. 91]. Fig. 3.4 shows some examples of marker rectification using the approach outlined above.

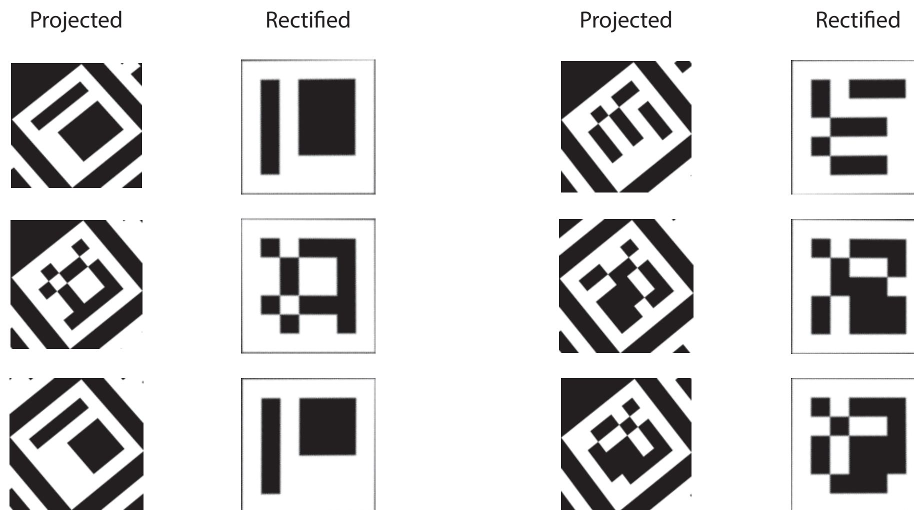


Figure 3.4. Marker rectification.

3.3.2 *Marker Code Extraction*

After the effects of perspective distortion are removed and the marker rectified, the id can be read out by following the naive approach described earlier. The rectified image is divided into 49 equal squares, and read out square by square. Each square represents one bit of data. A white square represents a one, and a black square a zero. It is possible to only sample

2 Even though the matrix has 9 elements, the matrix can be fully determined by finding 8 scale values between the 9 elements.

one pixel from the center of each square and obtain correct ids. But this approach is not very robust in images that were obtained from a noisy camera sensor.

To make the process of id readout more robust, the intensities of each square are added together and divided by the number of pixels in the square. In case that the rectified marker images are in binary format obtained from thresholding, counting the number of non-zero pixels is enough to determine the bit that the square represents. If half or more of the pixels are equal to 1, the square represents a one. Otherwise, the square represents a zero. Since each ArUco marker contains a white border, it should be ensured before reading out the binary code that the border is indeed white. By using this approach, each potential marker candidate is binarized into an id.

3.3.3 *Id Extraction*

To get back the id from the binary code, a series of operations is performed on the bitstream. Though it is possible to look up the id by using a hash table, the approach taken by the ArUco library is to decode each id every time anew.

3.4 CAMERA POSE ESTIMATION

The final step in the processing pipeline is to estimate camera pose given the ids and the corners of detected markers. The approach is similar to the approach for marker rectification. During the rectification, we were looking for a homography that mapped the corners of a marker into a square of fixed size at the origin of the coordinate frame. This time, however, the markers are mapped to their actual coordinates in the world frame, and not to the origin of the camera image coordinates. This section briefly covers how a pinhole camera can be modeled in the projective space. Subsequently, the computation of projective camera matrix and, finally, an approach to camera pose computation are presented.

The goal is to first, find the camera perspective matrix, and second, to extract the position and orientation of the camera from the camera matrix.

The starting point for computation of the camera projection matrix is Eq. (3.4). This time, however, a 3x4 camera projection matrix P is used

instead of the 3x3 homography matrix H. The equation can be derived in a similar manner [13, p. 179]:

$$\begin{bmatrix} \mathbf{0}^T & -w'_i\mathbf{x}_i^T & y'_i\mathbf{x}_i^T \\ w'_i\mathbf{x}_i^T & \mathbf{0}^T & -x'_i\mathbf{x}_i^T \end{bmatrix} \begin{pmatrix} \mathbf{p}^1 \\ \mathbf{p}^2 \\ \mathbf{p}^3 \end{pmatrix} = \mathbf{0}$$

where $\mathbf{x}'_i = (x'_i, y'_i, w'_i)^T$ are the homogeneous coordinates of points i in the camera image, $\mathbf{x}_i^T = (x_i, y_i, z_i, k_i)$ are the homogeneous coordinates of points i in the world coordinate frame, $\mathbf{0}^T$ and $\mathbf{0}$ are vectors containing four zeros, and $\mathbf{p}^j$ with j=1,..,3 are the transposed rows of the camera projection matrix P

$$P = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \tag{3.5}$$

The equation above represents a general case of perspective transformation from 3D points into the 2D camera plane. In our case, however, the transformation is between points of two planes. Without loss of generality, we can assume that the board with markers in the XY-plane, so that the Z-coordinates of all its points are equal to zero. Thus, we write

$$\begin{bmatrix} 0 & 0 & 0 & 0 & -w'_ix_i & -w'_iy_i & -w'_iz_i & -w'_ik_i & y'_ix_i & y'_iy_i & y'_iz_i & y'_ik_i \\ w'_ix_i & w'_iy_i & w'_iz_i & w'_ik_i & 0 & 0 & 0 & 0 & -x_ix_i & -x_iy_i & -x_iz_i & -x_ik_i \end{bmatrix} \begin{pmatrix} \mathbf{p}^1 \\ \mathbf{p}^2 \\ \mathbf{p}^3 \end{pmatrix} = \mathbf{0}$$

In the next step, we set all Z coordinates of the points in the board to zero. This affects 3rd, 7th, and 11th columns of the 2x12 matrix and the elements of projection matrix P with the same indices:

$$\begin{bmatrix} 0 & 0 & 0 & -w'_ix_i & -w'_iy_i & -w'_ik_i & y'_ix_i & y'_iy_i & y'_ik_i \\ w'_ix_i & w'_iy_i & w'_ik_i & 0 & 0 & 0 & -x_ix_i & -x_iy_i & -x_ik_i \end{bmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_4 \\ p_5 \\ p_6 \\ p_8 \\ p_9 \\ p_{10} \\ p_{12} \end{pmatrix} = \mathbf{0}$$

It follows that the mapping from points $\mathbf{x}_i$ in the plane and their image $\mathbf{x}'_i$ is a planar homography $\mathbf{x}' = H\mathbf{x}$ [13, p. 196]. As a consequence, the estimation of the homography can be carried out in the same manner as it is done for marker rectification. For each detected marker in the image there will be four point correspondences, which results in eight equations. If more than one marker is detected, the matrix is over-determined, and an approximate solution to the equation set will be found. To find the maximum likelihood estimate $\hat{H}$ that minimizes reprojection error by using an iterative error minimization algorithm, such as Levenberg-Marquardg [13, p. 114].

APPROACH

4

This section is divided into two parts: the first part presents a working approach that has been successfully implemented and evaluated; the second part describes the approach that has been initially considered, but has not been realized due to its complexity and the lack of time.

4.1 WORKING APPROACH

This section describes the approach that has been successfully realized and evaluated in the course of this project. The system design is illustrated in Fig. 4.1. On the FPGA, grayscale images captured by the on-board camera are converted into binary images by using a threshold. The resulting binary image is divided into a fixed number of packets, which are transferred to a PC one by one. The PC captures the data and converts it back into a binary image. In the next step, the camera pose estimation methods are applied onto the binary image, and the camera pose is estimated.

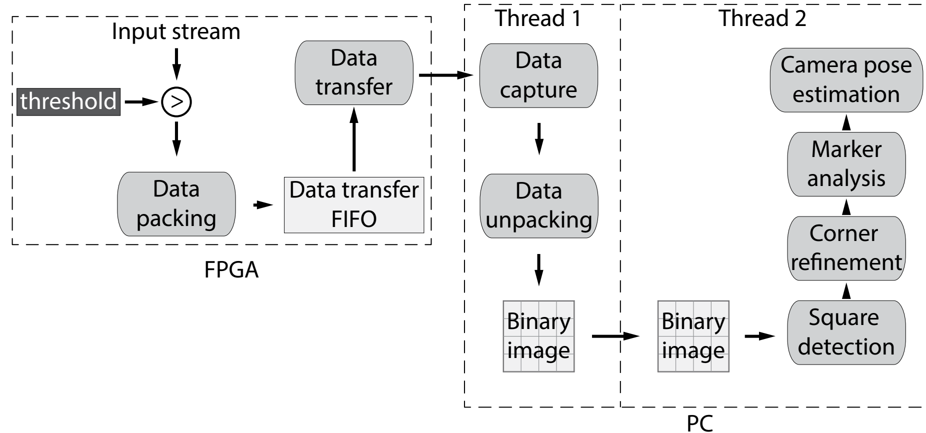


Figure 4.1. System design of the working approach.

4.1.1 *Thresholding on FPGA*

Thresholding is performed by comparing the intensities of all pixels in the camera image to a fixed threshold. Thresholding can be done on the fly without the need for a frame buffer.

4.1.2 *Data Packing*

The amount of data to be sent to the PC is known beforehand, so that it can be divided into a fixed number of parts. A 640x480 binary camera consists of $640 \cdot 480 = 307200$ bits and is divided into 75 equal intervals of 4096 bits each. The intervals are numbered from 0 to 74, and this number is sent together with the image bits to the PC. Numbering the packets in this way has the advantage that if the PC stops capturing data for a while, and the packets are not numbered, then there is no way to reconstruct the binary image once the capturing process has been started again. The number is converted into image coordinates, so that the place of every received bit is identified uniquely.

The bits of the binary image are grouped together into words of 16 bits and written into a FIFO that is used for data transfer by the Ethernet controller. Before packing 4096 bits of an interval together, the interval number is written into the data transfer FIFO. 16 bits are allocated for sending the interval number.

4.1.3 *Data Transfer*

Data is transferred from the FPGA to the PC by using the Ethernet protocol. First, the 6×8 bits MAC address of the sender is written, then the MAC address of the receiver. Subsequently, image data and the interval numbers are transferred. Thus each Ethernet packet consists of $2 \cdot 6 \cdot 8$ bits allocated for the MAC addresses, 16 bits for sending the interval number, and 4096 bits for sending the binary image data. Some additional data is added by the hardware, such as the 8 bytes preamble, and 4 bytes of checksum to detect errors in the Ethernet frame. Altogether, each packet is $2 \cdot 6 \cdot 8 + 16 + 4096 + 8 \cdot 8 + 4 \cdot 8 = 4304$ bits or 538 bytes. Assuming that the camera captures images at 100 frames per second, the expected traffic of this approach is $4304 \cdot 75 \cdot 100 = 32,280,000$ bits per second, or ≈ 32 Mbits/s.

4.1.4 *Data Capture on PC*

On the PC side, the images are captured by using the Berkeley packet filter, which allows low-level access to the network interface. To detect packets sent from the FPGA, the sender and receiver MAC addresses of all packets are inspected. Packets with the right addresses are considered for further processing. The data is unpacked by extracting the interval number and

the 4096 image data bits. Obtaining the address of the interval in image coordinates is done as follows:

```
1 int x = (interval * 4096) % 640;
2 int y = (internal * 4096) / 640;
```

x and y are used for saving the 4096 bits of image data in an image buffer.

4.1.5 *Camera Pose Estimation on PC*

To allow the utilization of multicore processors, data capture process and the camera pose estimation run in different POSIX threads [1]. To communicate between the threads, shared data guarded by a mutex is used. Upon reception of a complete binary image, a shared status variable is set to high, so that the camera pose estimation thread can copy the binary image into its own memory space and start the process of camera pose estimation. The image processing pipeline for detecting ArUco markers on the PC is provided by the ArUco library that is written in C++ on top of the OpenCV library.

4.2 INITIAL APPROACH

This section describes the approach that was developed initially, before falling back to a simpler solution due to the time constraints. Fig. 4.2 illustrates the processing pipeline of this approach. First, camera images are thresholded and at the same time, saved in a frame buffer. Most FPGAs do not have enough memory to hold an image of 640x480 grayscale values, therefore, an external memory module is required. In the next step, binarized images are run through a blob detecting circuit and a corner detecting circuit. Blob detection allows the individual pixels to be grouped into potential marker candidates, while corner detection provides information about the shape of the blobs. Blob labels and their corresponding corners are temporarily saved in a buffer.

The number of corners is reduced by a polygon approximation technique in order to find the squares in the image. If only four corners remain, the blob is divided into 7x7 approximately equal parts by following the naive approach described previously. The resulting blob regions are reduced to binary numbers, and the 5x5 internal code is obtained.

1 Though it would have been posible to use more portable and more generic multicore library, such as Intel's threading building blocks, or OpenMP library, the priority of this work was to show the proof of concept.

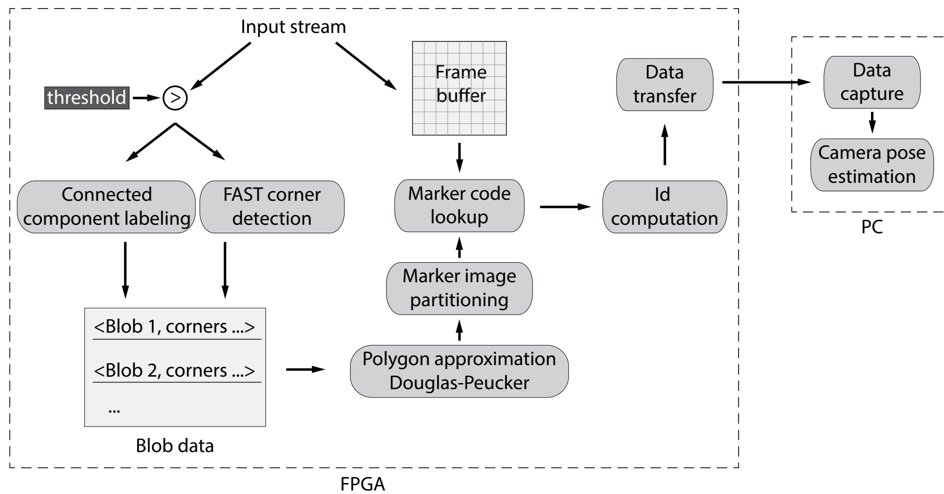


Figure 4.2. System design initially planned. The FPGA is used to extract corners and ids of the markers. The PC is used to compute the camera pose.

The code and its four rotated versions are decoded by computing their Hamming distance to a fixed 4x5 generator word. Hamming distance equal to zero means that a valid id has been detected. The id of the marker is computed in the next step. Subsequently, the id and the four corners are transferred to the PC. On the PC side, received ids and corners are used for camera pose computation by applying singular value decomposition and iterative reprojection error minimization achieved by the Levenberg-Marquardt minimization technique.

4.2.1 *Blob Detection*

Before detecting blobs, camera images are binarized by using a threshold. If the intensity of a pixel exceeds this value, it is reduced to a binary digit '1', otherwise it is a '0'. The results of binarization are used by a blob detecting circuit on the fly. The first approach mentioned in this work uses contours to accomplish blob detection. However, contour detection can result in a large number of points in images that are taken from a short distance to the ArUco board, which are considered sequentially during the polygon approximation phase. Either the number of points in the contour should be reduced before further processing, or another blob detection algorithm should be used. In this case, connected component labeling is used.

Connected component labeling (CCL) labels all pixels in an image based on their connectivity. If any two foreground pixels are neighbors (either by considering the 4-pixel neighborhood, or the 8-pixel neighborhood), they are assigned the same label. Similarly, if two background pixels are

neighbors, they will have the same label, and since we are usually only interested in the foreground pixels, all background pixels are given the label 0.

Connected component labeling is based on the union-find algorithm, that, in essence, unifies labels representing the same object. Three union-find algorithms are reviewed in what follows next[2]. Objects are represented by an array of labels, where the *i*-th object in the array points to the object stored at that index. Consider the following array where each object is only connected to itself:

i	0	1	2	3	4	5	6	7	8	9
label[i]	0	1	2	3	4	5	6	7	8	9

In the next example, there are three sets of objects: {0,1,2,3,4}, {5}, and {6,7,8,9}, represented by labels 0, 5, and 6, respectively:

i	0	1	2	3	4	5	6	7	8	9
label[i]	0	0	0	0	0	5	6	6	6	6

The algorithm has two basic procedures: `find` and `union`. `Find` returns the label of the object, given it's index. For example, the label of object `4` above is equal to `0`. `Union` of two labels unifies them, so that when a `find` is issued on either of them, the same label is returned. The two procedures can be implemented in several ways. We consider three of them to give an idea about the complexity of the algorithm in terms of the number of required memory accesses. In the first implementation, called the *quick-find*, `find` simply returns the label of the object by looking it up in the array. `Union` uses the find operation in order to obtain the labels of the requested objects. In case the found labels are equal, the objects are already connected. If they differ, all occurrences of the first label in the array are replaced by the second label. This implementation of the union-find algorithm has the worst case computation cost of $\sim N^2$. On the FPGA, the naive implementation requires a lot of memory for storing the pixel labels and a large number of memory accesses in the worst case.

Our second implementation of the union-find algorithm, called the *quick-union* algorithm, reduces the number of computations required to perform the `union` operation at the expense of the number of computations during `find`. In contrast to the previous version, this implementation interprets the content of the label array as a tree. Each entry in the array is consid-

2 More details about the union-find algorithm can be found in [28, pp. 216–234].

ered as a link from child to parent. Labels represent the same object if they have the same root node. Consider the array below:

i	0	1	2	3	4	5	6	7	8	9
label[i]	0	0	0	2	0	5	6	6	6	6

It represents three trees with root nodes 0, 5, and 6, as shown in Fig. 4.3. Now, in order to find whether a label is connected to another label, the

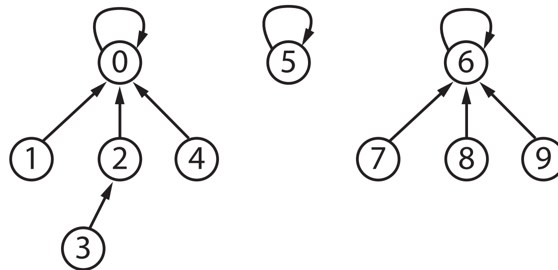


Figure 4.3. Tree interpretation of the quick-union approach. The nodes of a tree are all equivalent and represent the same object.

roots of the corresponding trees are searched. Thus, the new `find` algorithm is extended to traverse the tree recursively, as long as the parent node is not equal to itself. And the quick-union procedure for two labels searches for the root nodes of the queried labels. If they are equal, the labels are connected, otherwise, the root node of the first node is assigned to the root node of the second label. If we want to perform a union or a find operation N times, the quick-union algorithm requires $\sim N^2$ array accesses in the worst case.

The quick-union algorithm can be improved by adding a *weight* parameter to each node. The weight parameter stores the number of children of a node. The weights indicate which node is a better candidate to become the new root node when two unconnected labels are unified. This minor tweak changes the number of computations required in the worst case to $\sim \log N$. If the tree is kept flat at all times, however, the runtime requirements of the algorithm can be brought down to a constant time by using *path-compression*, which links all nodes visited during the `find` operation to the root node.

The union-find algorithm can be used to detect connected components in a binary image by labeling each pixel in the image. Assuming that the pixel arrive on the FPGA in a raster-scan order each pixel has four neighbors whose labels are known: up-left, up, up-right, and left. If the neighbors are all background pixels, the pixel will be labeled by a zero. If any neighbor has a non-zero label, a union operation will be performed on the pixel and the non-zero label.

Fig. 4.4 shows an example of how the union-find algorithm can be applied onto a binary image and extract connected components. The test image is shown at the top of the image. In this approach, each pixel starts as a root node pointing at itself. Thus, each pixel starts with a unique label. After processing the first row, pixels 2 and 4 are found to be the background pixels and are connected to the root node 0, which is the equivalence class of the background pixels. Furthermore, pixels 1, 3, and 5 were found to be foreground pixels. However, they are only connected to themselves, so that their labeling does not change. After the second row, pixels 6, 8, and 9 are added to the background. Pixel 1 is unified with 7, and subsequently, 7 is unified with 3. This is the case when a pixel is unified two times. The unification continues until all rows are processed. At the end, the algorithm finds one object that contains all foreground pixels, and all the background pixels are grouped together as well.

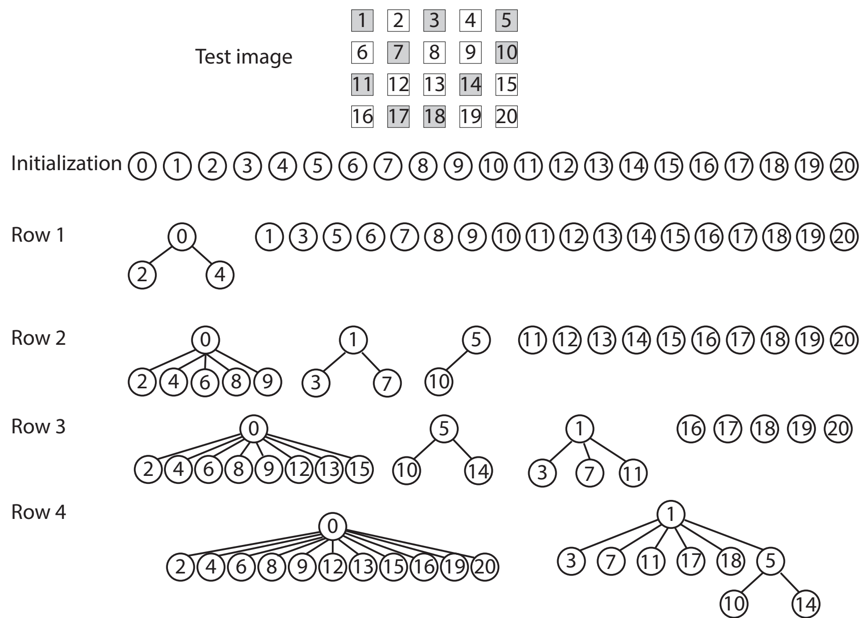


Figure 4.4. Application of weighted union-find algorithm on to a test image.

Even though the algorithm produces correct result, it requires a large amount of memory, because each pixel in the image has a unique label. For an image with 640x480 pixels, 307200 labels each represented by 19 bits. The overall memory requirement is $307200 \cdot 19\mathtt{bits}$. Thus, a direct application of the union-find algorithm to connected component labeling is inefficient. However, memory requirements can be drastically reduced by introducing new labels only when a foreground pixel is detected that is not yet connected to any other foreground pixel. We can make another observation by taking a look at the stricture of the images—it is not possible

to have more active pixels than the half-width of the image. If the width of the image is 640 pixels, only 320 blobs can be active in any line.

One issue that has been left out so far is the situation when a foreground pixel is connected to several already labeled pixels at the same time, in which case several updates to the tree are necessary. If the tree is stored in block RAM of the FPGA, there are several ways how the tree can be updated without loosing data: 1) the clock frequency of the block RAM can be run with a frequency twice as fast, 2) camera image can be stored in a sufficiently large FIFO, and 3) the tree can be updated during the horizontal and vertical blanking periods. The third approach is taken by [1, 18], however, its disadvantage is the large amount of exceptions that have to be considered during the implementation of the algorithm, and the overall high implementation complexity of the algorithm that involves a high degree of manual memory management. In this project, it was intended to develop a combination of 1) and 2) however, due to the timing constraints it has not been fully developed.

4.2.2 *Corner Detection*

In parallel to blob detection, the corners are extracted. In this work we use an approach for FAST (features from accelerated segment test) corner detection first presented in [23] and implemented on an FPGA by [14]. The basic idea of a FAST corner detector is to compute a corner score for every pixel in the image. The corner score is computed by comparing the intensity of a pixel I_p to the intensities of 16 pixels along the Bresentham circle, as shown in Fig. 4.5.

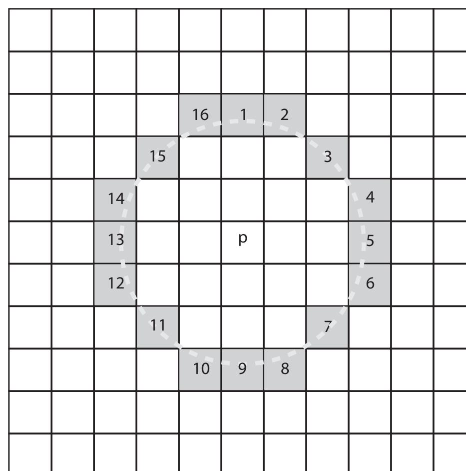


Figure 4.5. Bresentham circle used for computing the corner score of pixel p.

Each pixel along the Bresentham circle is classified as either a bright pixel or a dark pixel. Pixels whose intensities I_x are brighter than $I_p + t$, where I_p is the intensity of pixel p, are considered as bright pixels, and pixels with $I_x < I_p - t$ are considered as dark pixels. Formally, the corner score of a pixel is defined as:

$$V = max\left(\sum_{x \in S_{bright}} |I_x - I_p| - t, \sum_{s \in S_{dark}} |I_p - I_x| - t\right)$$

where S_{bright} are bright pixels, and S_{dark} are dark pixels. Only consecutive pixels are considered as candidates for corner score computation. For example, if there are exactly 9 consecutive bright pixels on the circle, their corner score is computed. The same holds for 9 contiguous dark pixels.

If such a combination is found, the result is stored in a buffer. Afterwards, a non-maximum suppression window of 5x5 pixels is applied onto that buffer. It means that if the corner score is not a part of local maximum, it will be discarded. In this way, only the corners with the highest local score are retained.

4.2.3 *Polygon Approximation*

Connected component labeling and corner detection result in a list of blobs with corners. During polygon approximation, corner points are considered sequentially. First, the two most distant points are connected by a line. Next, a point that is the most farthest from the line is found by searching through all remaining points. Euclidean distance between two 2D points is defined as:

$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

However, we are not interested in the actual distance between points. The Euclidean distance is only used to define an ordering of all the points. The square root function does not change the ordering or the underlying sum of squares because it is monotonic when applied to positive arguments. Thus, we can safely leave out the computation of square roots, so that only two multipliers are needed to implement this procedure.

4.2.4 *Naive Marker Partitioning*

The marker is split up into 49 parts by subdividing the outer borders into seven equal line segments and by connecting the segments of the op-

posing borders to each other. This approach has been criticized earlier in this thesis because a perspective projection is a nonlinear transformation that, depending on camera pose, can result in highly distorted marker images that might cause in a large number of false positives. However, the approach presented here is easier to implement and requires fewer resources.

Assume that we have two 2D points $P_a = (x_a, y_a)$ and $P_b = (x_b, y_b)$ that represent the line that we want to divide into seven equal line segments. The end point of the first line segment can be computed as follows:

$$P_1 = \left(x_a + \frac{x_b - x_b}{7}, y_b + \frac{y_b - y_a}{7}\right)$$

The second point:

$$P_2 = \left(x_a + \frac{(x_b - x_a) \cdot 2}{7}, y_a + \frac{(y_b - y_a) \cdot 2}{7}\right)$$

In general, each point with i=1,. . . ,6 can be computed as follows:

$$P_i = \left(x_a + \frac{(x_b - x_a) \cdot i}{7}, y_a + \frac{(y_b - y_a) \cdot i}{7}\right) \tag{4.1}$$

Computation of the six points for each line requires $6 \cdot 2 \cdot 4 = 48$ floating point divisions and $5 \cdot 2 \cdot 4$ floating point multiplications. The floating point division is the more expensive than floating point multiplication in terms of clock cycles. However, the number of divisions can be reduced by precomputing the division twice for every line—once for the x and y coordinates, respectively. Thus the number of floating point divisions is reduced to $2 \cdot 4 = 8$.

4.2.5 *Marker Code Extraction*

In order to extract the code from a marker, the partitioned grid is sampled in the middle of each region. Fig. 4.6 shows the basic idea of this approach.

Computing the central points of grid region is the same as dividing each size of the marker into 14 equal parts. Eq. (4.1) can be modified to compute starting points for marker code sampling as follows:

$$P_i = \left(x_a + \frac{(x_b - x_a) \cdot i}{14}, y_a + \frac{(y_b - y_a) \cdot i}{14}\right)$$

for $i = 1, 3, ..., 13$. In the next step, the marker is sampled by reading small regions around the sample points in order to obtain the 5x5 marker

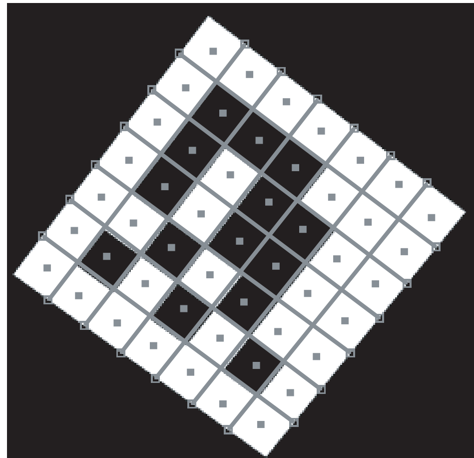


Figure 4.6. Marker sampling. Each grid region is sampled in its middle by using a window of a fixed size.

code. If the majority of pixels of a region are foreground pixels, the bit is assumed to be a 1, otherwise it will be set to zero.

4.2.6 *Id Analysis*

To determine whether the code belongs to a valid marker, the Hamming distance of the code is compared to the generator matrix of the ArUco marker in all four possible orientations. The generator matrix and its possible rotations are stored in the registers of the FPGA. To find the Hamming distance, the marker code and the generator matrix are compared to each other row-wise. The Hamming distance is the number of times when a bit of the generator matrix is not equal to the same bit in the code of the potential marker. However, since no error correction is performed, the id analysis phase can be reduced to an equality operation. If the numbers are not equal, the marker candidate is simply discarded.

4.2.7 *Id Computation*

To compute the marker id, only the second and the fourth rows of the marker code are required. The following excerpt from the ArUco library shows how the id of a marker can be computed from the marker code stored in the 5x5 array `bits`:

```
1  int id = 0;
2  for (int y = 0; y < 5; y++) {
3      id <<= 1;
4      if (bits.at<uchar>(y,1)) id |=1;
5      id <<= 1;
```

```
6       if (bits.at<uchar>(y,3)) id |=1;
7   }
```

Only shifting and inclusive OR operators are used during id computation.

5 SETUP

The approach developed in this project has been implemented by using VHDL on Altera's DE2-70 board developed by Terasic [32]. The core of the board is a Cyclone II 2C70 FPGA device with 68,416 logic elements and 250 M4K RAM blocks. The FPGA is connected to several interfaces that are used for communication with the PC, such as a serial port and a 100 Mbit/s Ethernet port.

A mvBlueCOUGAR-X 100 camera [16] is mounted on one of two general-purpose expansion headers (GPIO) of the board. The camera has a CMOS sensor that delivers 10- bit grayscale images with resolution of up to 752x480 pixels and allows frame rates of up to 117 frames per second. The camera has an on-board FPGA and can be configured from a PC over a 1 Gbit/s Ethernet connection.

The computer used during evaluation has a 2.4 GHz Intel Core i5 processor with two cores. The system has 8GB DDR3 RAM running at 1333 MHz. The operating system is Mac OSX, version 10.7.5.

6 EVALUATION

Several methods have been presented here. Depending on the utilized method, the accuracy and precision of marker detection will change. Evaluating the system under different conditions will help us to choose the proper method for each condition. However, which parameters can be used to assess the performance of the system? The ultimate goal of the marker detecting system is to provide an interface between the Immersion Square and the user. The user can *feel* whether the system is good or not by interacting with it. If the user points the interface onto a specific part of the canvas, but the system responds with pointing the virtual interface at another place, then the system will be perceived as not very accurate. If there is a long delay between user action and system reaction, then the system will be difficult to work with. These two variables—camera pose error and time—will be used to assess the performance of the system.

6.1 PERFORMANCE VARIABLES

The simplest way to compute the camera pose error is to measure the distance between the estimated camera pose and the ground truth. However, it is difficult to compare the errors with each other, because even a large error in camera pose might result in no "perceived" error by the user if the central lines of the ground truth camera and the estimated camera meet the canvas in the same point. The central lines of the two cameras with slightly different poses meet in the same point on the canvas.

To bring our error measurement closer to the perception of the user, another approach would be to compute the distances between the two central points of the ground truth camera and the computed camera. However, this method would unfairly penalize the performance when there is a large distance between camera and canvas, since the error in camera pose tends to be higher the farther away the camera is from the canvas. A user who is far away from the canvas might not even notice any difference in the position of the central point of the camera (crosshairs). Also if the camera is close to the canvas and looks at it under a steep angle, even a small error in camera pose might result in high errors that the user, however, does not notice.

Thus, a better way to measure the perceived error of the interface is to compute the angle between the estimated central line of the camera and the ground truth. Angular error has the advantage of being an absolute measure that is independent of the equipment used. The original camera pose error is meaningless without knowing the size of the canvas and its distance from the camera. The crosshairs position error is also perceived differently based on the distance to the canvas. However, an angular error of, e.g., 5° is perceived as the same by the user who is 1 m away from the canvas and the user who is only 30 cm away from the canvas. In this work we use all three error measures and show the difference between them.

One of the concerns for the hybrid approach presented in this work is that it might not be able process all the data fast enough. To ensure that the system does not drop frames, it is necessary that all processing is completed by the time a new frame is received by the PC. A delay in processing might result in dropped frames. For a camera running at 100 frames per second, the time window for marker detection is 10 milliseconds ($\frac{1000ms}{100} = 10ms$). Thus, another performance measure that will be used to evaluate the system is the computation time.

6.2 EVALUATION ENVIRONMENT

The approach presented in this thesis has been evaluated on a set of synthetic images generated in a simulation environment that has been written in the course of this project. Synthetic images have several advantages over real-world images. The ground truth data can be generated with nearly infinite precision, whereas in real systems, the ground truth is never error-free. A large amount of data can be generated and evaluated in a short amount of time. However, the disadvantages are that the system might behave differently under real conditions.

The evaluation environment consists of a board with a pattern of 16x16 ArUco markers and a camera that can be placed anywhere in the simulated space. The markers have been randomly placed onto the board in a way that their IDs are unique and do not repeat [1]. The board has the unit size of 1m x 1m and is placed at the origin of the coordinates at (0,0,0). The camera is modeled as a pinhole camera with the same parameters of the real MV camera. Fig. 6.1 depicts the simulation environment.

The board of N markers is drawn in a 3D environment using OpenGL. The center of the board is placed at $(0;0;0)$. Each marker is drawn by using black and white rectangles.

1 The board has been generated by Timur Saitov and burned into a special-purpose infrared projector.

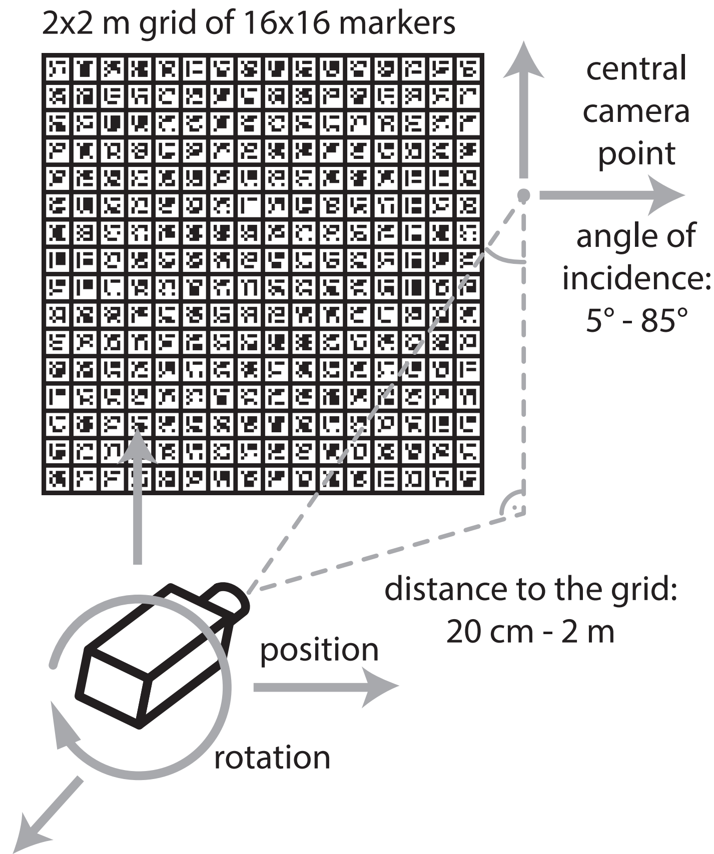


Figure 6.1. Generation of synthetic test images containing fiducial markers.

The environment has been implemented by using OpenGL. To model the camera, the specification of the real MatrixVision camera have been used to set the OpenGL perspective projection matrix. The pixel size of the MV camera is $6 \cdot 10^{-6}$ x $6 \cdot 10^{-6}$ meters and the focal distance is $6 \cdot 10^{-3}$ meters. The resolution of the camera is 752 x 480 pixels, however, during the evaluation a more common resolution of 640 x 480 pixels is used. To set the perspective projection matrix, the following OpenGL procedure has been used:

```
1 void gluPerspective(fovy, aspect, zNear, zFar)
```

`fovy` is the vertical field of view that can be computed by knowing the vertical resolution of the camera, its focal distance and the pixel size:

$$\mathtt{fovy} = 2 \cdot \operatorname{atan}\left(\frac{H \cdot pSize}{2f}\right) = 2 \cdot \operatorname{atan}\left(\frac{480 \cdot 6 \cdot 10^{-6}}{2 \cdot 6 \cdot 10^{-3}}\right) \approx 0.471 \text{ rad} = 26.99^{\circ}$$

`aspect` is the aspect ratio of the image, which can be computed by dividing the width of the image by its height; `zNear` and `zFar` are the distances between the camera and the clipping planes. Only the objects between

the clipping planes are visible to the camera. Fig. 6.2 shows the interplay between the arguments of the `gluPerspective` procedure.

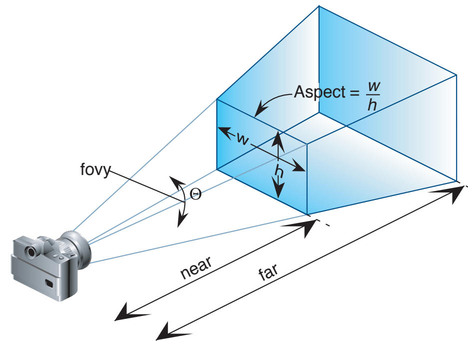


Figure 6.2. Perspective projection modeling in OpenGL.(Taken from [30]).

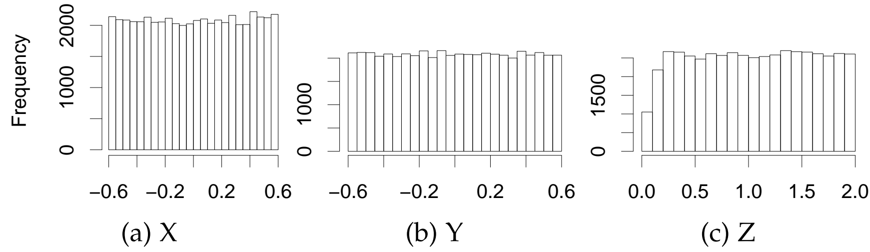
6.3 DATASET GENERATION

The evaluation of the system has been carried out in three steps. First, a set of test images has been generated by the simulation environment with a camera at different positions and varying angles. In the next step, the methods described in this thesis have been put to test under different parameters, such as blur, noise, and marker detecting method. In the last step, the performance measures—the pose errors, angular errors, and other important parameters—have been computed and analyzed. The following subsections describe each step in detail and also discuss some alternative ways to evaluate the system.

6.3.1 *Generation of Test Images*

The simulation environment has been used to generate 50,000 test images from different camera angles and positions. The x and y positions are uniformly distributed between -0.6 and 0.6. The z position is sampled from a normal distribution with $\mu = 1.0$ and $\sigma^2 = 0.7$. This is done because the user is more likely to be at a moderate distance from the board. Fig. 6.3 shows the histograms of x, y, and z.

After the position of the camera is fixed, the angles are randomly generated. *Roll* is sampled from a uniform distribution between $-90°$ and $90°$. *Pitch* and *heading* are sampled from a variable distribution that depends on the position of the camera—the distribution is set in such a way that the crosshairs point never leaves the board. This is done by computing the angle between the camera looking along the Z-axis and the camera looking towards the edges of the board to the left and to the right for *heading*,



(a) X (b) Y (c) Z

Figure 6.3. Histograms of individual coordinates of camera position.

and to the top and to the bottom edges of the board for *pitch*. Limiting the angle distribution ensures that the camera is always looking at the board. In the real world, a user that controls the Immersion Square by using an interaction device has no incentive to point outside of the canvases.

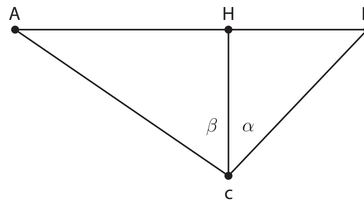


Figure 6.4. Restricting the heading angle.

Consider Fig. 6.4 where it is assumed that the camera is at (x, y, z). The angles to sample heading can be computed as follows:

$$HB = OB - OH; HA = OA - OH$$

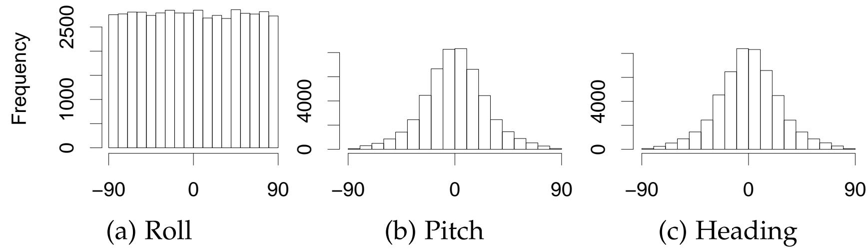
$$tan(\alpha) = \frac{HB}{CH}; tan(\beta) = \frac{HA}{CH}$$

$$\alpha = atan\left(\frac{OB - OH}{CH}\right) = atan\left(\frac{0.5 - x}{z}\right)$$

$$\beta = atan\left(\frac{OA - OH}{CH}\right) = atan\left(\frac{-0.5 - x}{z}\right)$$

AB is the size of the board and is equal to 1. OB is the half board size equal to 0.5. α and β are used to generate the heading by sampling from a uniform distribution between the two angles. Pitch can be computed in the same manner by substituting x with y. Fig. 6.5 shows the histograms of the angles in the dataset. Interestingly, the restriction applied to the camera position result in a Gaussian-like distribution for heading and pitch.

The camera is positioned in a way that at least one marker is always visible. This has the motivation to reduce the number of fully black images, as well as the images in which each visible marker is seen only partially. Each image that passes this condition is retained, and all other images



(a) Roll (b) Pitch (c) Heading

Figure 6.5. Histograms for roll, pitch, and heading.

discarded. The situations in which a marker is only partially visible arise when the camera is positioned at very close ranges to the board at angles around 90°. The marker detection algorithms are not able to recognize any markers in such images and are only skewing the evaluation statistics.

In order to compute the number of visible markers in an image, the corners of each marker on the board are projected onto the camera image plane. Markers whose corners are all in the image plane are considered as visible markers. Projection of coordinates from the world plane into the image plane can be done by multiplying the world coordinates by the camera projection matrix. The projection matrix can be constructed from the camera pose and the intrinsic camera parameters as follows [13, p. 56]:

$$P = K[R|t]$$

where K is the camera calibration matrix, R is a 3x3 rotation matrix that represents the rotation of the camera in the world coordinate frame, and t is equal to $-R\tilde{C}$ with $\tilde{C}$ being the camera position in the world coordinate frame. The camera calibration matrix can be computed as follows [13, p. 57]:

$$K = \begin{bmatrix} f \cdot m & & x_0 \\ & f \cdot m & y_0 \\ & & 1 \end{bmatrix}$$

where f is the camera's focal distance, m is the number of pixels per unit distance, which can be computed from the physical pixel size of the camera that is equal to $6 \cdot 10^{-6}m$, thus m is equal to $\frac{1}{6 \cdot 10^{-6}}$; x_0 and y_0 are the coordinates of the camera's principal point in pixels and are equal to 320 and 240, respectively.

The rotation matrix can be obtained by performing three rotations one after another. This can be accomplished by multiplying three rotation matrices. The order of multiplication matters and different results are obtained if the order differs. For example, rotating around the X axis first, followed by rotations around the Y, and the Z axes results in a different

rotation than doing it otherwise. This is illustrated by multiplying some example matrices. Let us assume that we want to rotate around the X-axis by $\alpha = 30°$, around the Y-axis by angle $\beta = 15°$, and around the Z-axis by $\gamma = 70°$. We can now construct three rotation matrices as follows:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.8666 & -0.5 \\ 0 & 0.5 & 0.8666 \end{pmatrix},$$

$$R_y(\beta) = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{pmatrix} = \begin{pmatrix} 0.9659 & 0 & 0.2588 \\ 0 & 1 & 0 \\ -0.2588 & 0 & 0.9659 \end{pmatrix},$$

$$R_z(\gamma) = \begin{pmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0.3420 & -0.9397 & 0 \\ 0.9397 & 0.3420 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Now we can show that applying rotation around the axes onto a test vector in different order produces different results. Rotating the vector $\mathbf{v} = (1\ 1\ 0)^T$ in the XYZ order (first X, then Y, then Z) results in:

$$\hat{\mathbf{v}}_{xyz} = R_Z \cdot R_Y \cdot R_X \cdot \mathbf{v} = \begin{pmatrix} 0.43082 \\ 1.86213 \\ 3.21666 \end{pmatrix}$$

Rotating in the ZYX order (first Z, then Y, then X) results in:

$$\hat{\mathbf{v}}_{zyx} = R_X \cdot R_Y \cdot R_Z \cdot \mathbf{v} = \begin{pmatrix} -0.70852 \\ -0.24164 \\ 3.66601 \end{pmatrix}$$

Rotating in the YXZ order results in:

$$\hat{\mathbf{v}}_{yxz} = R_Z \cdot R_X \cdot R_Y \cdot \mathbf{v} = \begin{pmatrix} 0.20824 \\ 1.77841 \\ 3.28541 \end{pmatrix}$$

In this work, we set the YXZ rotation order as the default. This rotation order is known in flight dynamics, as roll, pitch, and heading. First, the *heading* of the aircraft is specified—it is the rotation of the aircraft around the vertical axis, which is the Y-axis in OpenGL. The heading moves the X and the Z axes along with it. Second, the *pitch* of the aircraft is specified as its rotation around the axis spanned by the two wings, which is the rotated X-axis. The pitch moves the Z-axis again. Finally, the *heading* of the aircraft is specified by giving the angle around the Z-axis, which is the axis in which the aircraft is looking. We will use this convention to specify the three camera angles.

Here is an example of a projection matrix for a camera at P(0, 0, 1) with angles *roll* = 40°, *pitch* = 30°, and *heading* = 15°. The camera calibration matrix is:

$$K = \begin{pmatrix} -1000 & 0 & 320 \\ 0 & 1000 & 240 \\ 0 & 0 & 1 \end{pmatrix}$$

Camera rotation matrix computed in heading-pitch-roll order is equal to:

$$R = R_Z(40^\circ) \cdot R_X(30^\circ) \cdot R_Y(15^\circ) \approx \begin{pmatrix} 0.6568 & -0.5567 & 0.5087 \\ 0.7200 & 0.6634 & -0.2036 \\ -0.2241 & 0.5 & 0.8365 \end{pmatrix}$$

The rotated camera position vector $t = -R\tilde{C}$ is equal to

$$t = -\begin{pmatrix} 0.6568 & -0.5567 & 0.5087 \\ 0.7200 & 0.6634 & -0.2036 \\ -0.2241 & 0.5 & 0.8365 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.5087 \\ 0.2036 \\ -0.8365 \end{pmatrix}$$

The resulting camera projection matrix is computed as follows:

$$\begin{aligned} P &= K[R|t] \\ &= \begin{pmatrix} -1000 & 0 & 320 \\ 0 & 1000 & 240 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{bmatrix} 0.6568 & -0.5567 & 0.5087 & -0.5087 \\ 0.7200 & 0.6634 & -0.2036 & 0.2036 \\ -0.2241 & 0.5 & 0.8365 & -0.8365 \end{bmatrix} \\ &= \begin{pmatrix} -728.49 & 716.67 & -241.02 & 241.02 \\ 666.22 & 783.41 & -2.84 & 2.84 \\ -0.22 & 0.5 & 0.84 & -0.84 \end{pmatrix} \end{aligned}$$

Projecting the homogeneous coordinates of the corner points of the 1x1 canvas $C_0 = (0.5, 0.5, 0, 1)^T$, $C_1 = (0.5, -0.5, 0, 1)^T$, $C_2 = (-0.5, -0.5, 0, 1)^T$, $C_3 = (-0.5, 0.5, 0, 1)^T$ into the camera image results in following points:

$$P \cdot C_0 = \begin{pmatrix} 235.11 \\ 727.66 \\ -0.6986 \end{pmatrix}, P \cdot C_1 = \begin{pmatrix} -481.55 \\ -55.75 \\ -1.20 \end{pmatrix},$$
$$P \cdot C_2 = \begin{pmatrix} 246.93 \\ -721.98 \\ -0.97 \end{pmatrix}, P \cdot C_3 = \begin{pmatrix} 963.60 \\ 61.44 \\ -0.47 \end{pmatrix}$$

To obtain the coordinates of the projected corner points of the canvas in camera image coordinates, the homogeneous coordinates are normalized by their respective third values:

$$P \cdot C_0 = \begin{pmatrix} -336.56 \\ -1041.62 \\ 1.0 \end{pmatrix}, P \cdot C_1 = \begin{pmatrix} 401.77 \\ 46.52 \\ 1.0 \end{pmatrix},$$
$$P \cdot C_2 = \begin{pmatrix} -253.41 \\ 740.91 \\ 1.0 \end{pmatrix}, P \cdot C_3 = \begin{pmatrix} -2031.01 \\ -129.49 \\ 1.0 \end{pmatrix}$$

From the four corner points, only the lower-right corner C_1 is visible in the image. Fig. 6.6 shows the resulting projection of 1x1 board with ArUco markers.

By using this approach, the number of visible markers in each image is computed every time when a new camera pose is generated. This ensures the presence of at least one marker in each generated image. In addition, the number of markers in the image is a useful statistic that, combined with the number of markers recognized by our system, constitutes a useful performance that is helpful during evaluation of our system.

Alongside the test images, an accompanying dataset that contains useful information about each image is generated as well. The dataset is saved in a comma-separated values format and contains the following data: the names of the images; camera poses represented by roll, pitch, heading, x, y, and z; number of visible markers; and ids of visible markers.

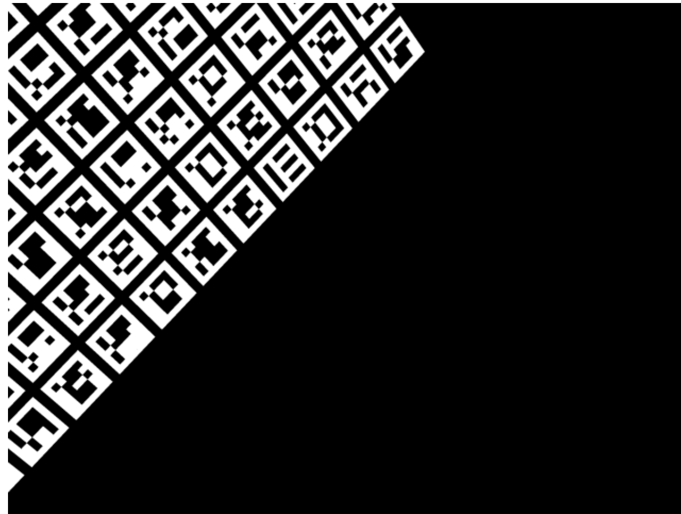


Figure 6.6. Projection of the marker board into the camera image. The lower-right corner of the board is projected to the point (401.77, 46.52) in pixel coordinates whose origin (0, 0) is at the upper-left corner of the image.

6.3.2 *Adding Blur and Noise*

To test the methods under different environmental conditions, artificial blur and noise are added to some images. In the real environment, the camera might be out of focus, which will result in blurry images. To model this situation, a set of test images will be artificially blurred by using Gaussian filter. The level of blur is controlled by the kernel size [29, p. 167]. Fig. 6.7 shows the result of artificial blur of a test image with different kernel sizes.

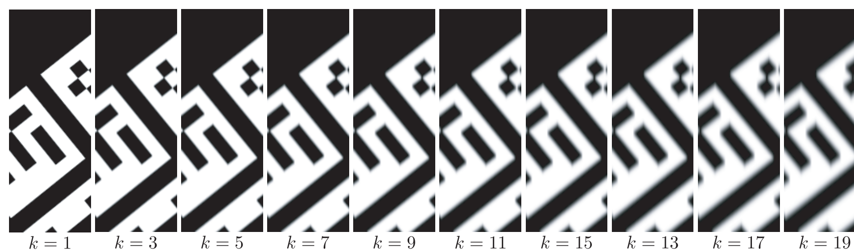


Figure 6.7. Gaussian blur with different kernel sizes.

To simulate systematic errors of the camera, random noise is added to some portion of the test images. Each pixel has some fixed probability of switching its intensity value to a random intensity between 0 and 256. The noise level for each image is sampled from a uniform distribution between 0 and 0.6. Fig. 6.8 shows some examples of images with different noise levels.

In addition to hard random noise, another portion of the test images is run through additive Gaussian noise. This type of noise is closer to

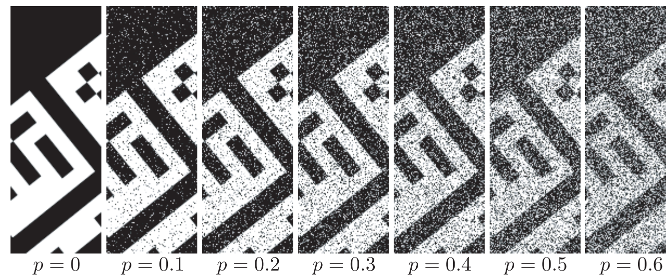


Figure 6.8. Noise with different probabilities for each pixel to switch to a random value.

the conditions of the real world. Usually, if a camera is susceptible to noise, the intensities of the pixels are very close to their actual values, but for some pixels, the value is distributed around the true intensity value altered by some variable number. It is likely that the noisy value is close to the real value, and less likely to be far away from it. Such distribution can be modeled by using Gaussian noise with the mean equal to the true intensity value of the pixel, and a variable variance. Fig. 6.9 shows some examples of images with different Gaussian distributions. The standard deviation for the noise is sampled randomly from a uniform distribution between [0; 255], inclusively.

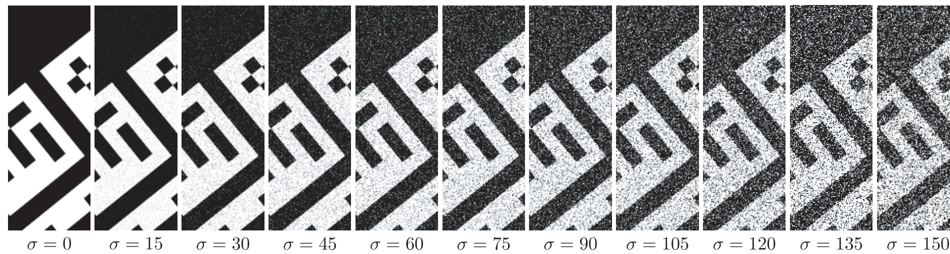


Figure 6.9. Additive Gaussian noise with different standard deviation. The mean of each distribution is different for each pixel and is equal to the pixel's intensity. Standard deviation σ controls the width of the Gaussian distribution.

Table 6.1 summarizes the image filters, their probability of occurrence, and ranges of the parameters. Each filter type (as well as the absence of filters) has equal probability of being applied on a test image.

Table 6.1. Parameters for image filters.

Filter	**Probability**	**Range**
None	0.25	—
Hard noise	0.25	$p \in [0; 0.6]$
Gaussian noise	0.25	$\sigma \in [0, 255]$
Gaussian blur	0.25	$k \in [1, 3, ..., 19]$

6.3.3 *ArUco Board Detection Methods*

After generating the test images, the next step is to perform board detection on every image. Two thresholding methods and four corner refinement methods are used during evaluation. In addition to the image processing methods, another parameter is varied during evaluation as well—the marker warp size. It defines the scaling of potential marker for marker analysis. A large marker warp size increases the robustness during marker identification phase. It helps in keeping down the number of false positives and false negatives. However, large marker warp sizes will increase the computation time. Including this parameter in evaluation might help us find optimal marker warp size that is appropriate in different situations. The marker warp size will be sampled randomly for each image from a discrete distribution from 7 to 70 by increments of 7. Table 6.2 summarizes the methods used during evaluation. From each column, one value is sampled randomly for each image.

Table 6.2. Summary of methods used in evaluation.

Corner ref. method	**Thresholding method**	**Marker warp size**
None	Fixed threshold	$1 \cdot 7$
Harris	Canny	$2 \cdot 7$
Subpixel		$\dots$
Lines		$10 \cdot 7$

6.3.4 *Final Dataset*

Board detection returns the estimated camera pose, the number of markers detected in the test image, and the coordinates of the corners of each detected marker. Some post-processing is applied on the results in order to compute the three errors described earlier and to determine the number of correctly detected markers in the image.

The first error measure introduced in the beginning of this section is the camera pose error. It is computed by taking the euclidean distance between the ground truth camera pose vector with 6 elements—x, y, z, roll, pitch, and heading. However, it is difficult to interpret the resulting number. For this purpose, the error is split into rotation error—the euclidean distance between the roll, pitch, heading angles—and position error that

is computed by taking the Euclidean distance between the estimated position and the actual position of the camera.

The second error is computed by finding the point where the principal camera axis goes through the board and by comparing it with the ground truth. The camera is at the point C(x,y,z). Initially, it is directed along the Z-axis towards the board and meets it in the point H(x,y,0), since the board is positioned at the origin of the coordinates along the XY-axis. After turning along the Y-axis by the angle *heading*, the camera's principal line meets the board in the point Y. After turning along the X-axis by the angle *pitch*, the camera's principal line meets the board in the point P, which is the crosshairs position that we are looking for. It can be computed step by step as follows:

$$HY = CH \cdot tan(heading) = z \cdot tan(heading)$$

$$CY = \frac{CH}{cos(heading)} = \frac{z}{cos(heading)}$$

$$PY = CY \cdot tan(pitch) = \frac{z \cdot tan(pitch)}{cos(heading)}$$

Thus, the coordinates of the crosshairs are:

$$(x + HY, y + PY, 0)$$

Crosshairs position error is determined by taking the Euclidean distance between the ground truth and the estimated crosshairs coordinates.

$$\sqrt{(\hat{P}_x - P_x)^2 + (\hat{P}_y - P_y)^2}$$

where P_x and P_y are the coordinates of ground truth crosshairs, and $\hat{P}_x$ and $\hat{P}_y$ are the estimated coordinates of the crosshairs computed during the board detection phase. The crosshairs rotation error is the difference between estimated roll angle and the ground truth.

The third error is the angular difference between the two crosshairs points computed in the previous step. In the triangle spanned by the vectors CP and $C\tilde{P}$, all the sides are known from the previous step. Thus, the angular error can be calculated by using the law of cosines:

$$P\tilde{P}^2 = CP^2 + C\tilde{P}^2 - 2 \cdot CP \cdot C\tilde{P} \cdot cos(\theta)$$

$$\theta = acos\left(\frac{CP^2 + C\tilde{P}^2 - P\tilde{P}^2}{2 \cdot CP \cdot C\tilde{P}}\right)$$

To find whether the marker is recognized correctly or not, it is not enough to confirm that the marker exists in the image by finding its id in the list of ids for that image. It is necessary to ensure that the marker is also in its right place, which can be done by computing its center and checking whether it is within the marker's bounding box. The center is computed by finding the intersection between the diagonals of the marker. Markers whose centers are not in the right place are considered false positives. Markers that are in the image, but are not recognized by the board detecting algorithm, are considered false negatives.

7 RESULTS

7.1 GENERAL TRENDS

This section discusses general dependencies between the five errors formulated in the beginning of evaluation section. The data points are summarized by using *lowess* [8, p. 94], which is a local iterative method that uses weighted least squares in order to fit a smooth curve to a set of data points. However, the data points themselves are also provided, since in some cases, the lowess curve is not meaningful by itself.

7.1.1 *Undistorted Data*

This part considers the data set that is free of noise and blur. The five errors—camera rotation error, camera position error, crosshairs rotation error, crosshairs position error and angular error—are plotted against the distance of the camera to the board, and the camera-board angle. Fig. 7.1 shows the trend of errors in camera translation. There is a strong dependency between camera-board distance and the translation error of the camera—the larger the distance from the camera, the higher is the error. However, it is interesting to see that the camera position error decreases the steeper the angle to the board. And when the angle is close to 90°—when the camera is directed along the z axis—the error is the largest. One of possible reasons might be that the corner features become easier to detect under steeper angles.

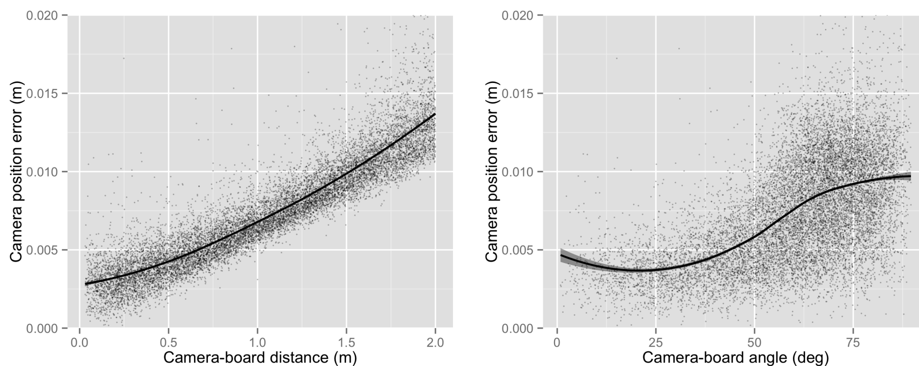


Figure 7.1. Camera position error.

Fig. 7.2 shows the behavior of the rotation error. Here we can see the opposite trend—the larger the distance from the board, the lower is the rotation error. Also, the more the camera-board angle approaches 90°, the lower is the camera rotation error. This can be explained with the fact that it is difficult to make an error in rotation when the camera-board distance is large, as long as at least one marker is recognized correctly. And steep angles between the camera and the board result in highly distorted images of rectangles to which the rotation is highly sensitive.

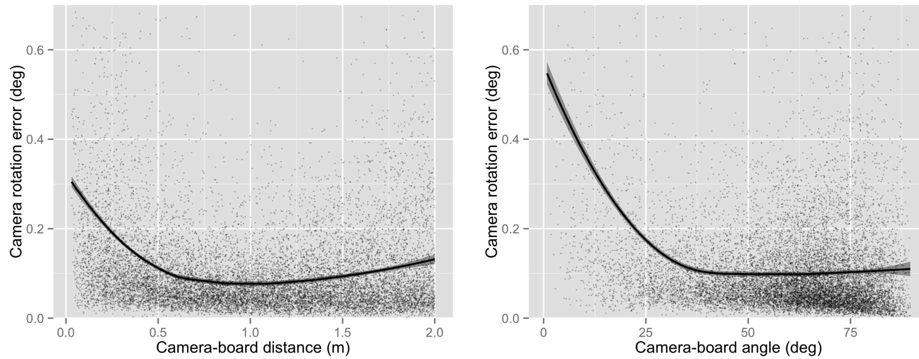


Figure 7.2. Camera rotation error.

In the next step, we consider the effects of distance and angle on the crosshairs position (CP) and crosshairs rotation (CR) error. CP error is the distance between the point the user is pointing to and the point that the system has computed. From Fig. 7.3, we can observe that the CP error gets higher as the camera-board distance increases, and lower as the camera-board angle approaches 90°.

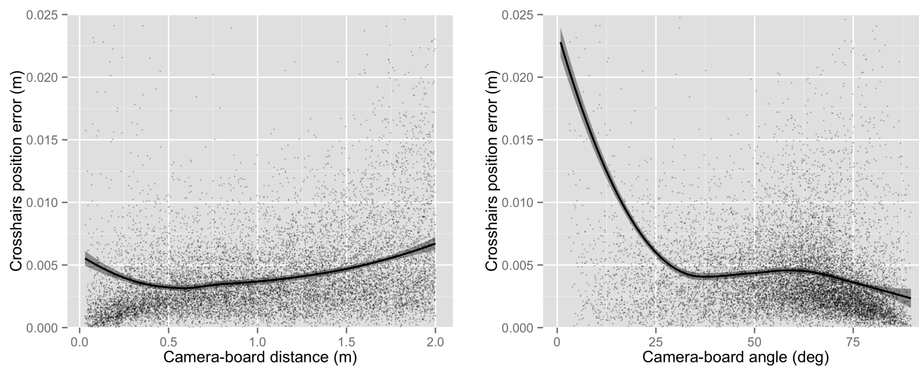


Figure 7.3. Crosshairs position error.

The CR error follows the opposite trend, as shown in Fig. 7.4. The longer the distance of the camera from the board, the lower is the rotation error. Therefore, if the user wants to perform some rotational command, it is better to do this from afar. Conversely, when the accuracy of pointing has

to be higher, the camera should be closer to the board. However, in both cases, the error is lower at angles close to 90 degrees.

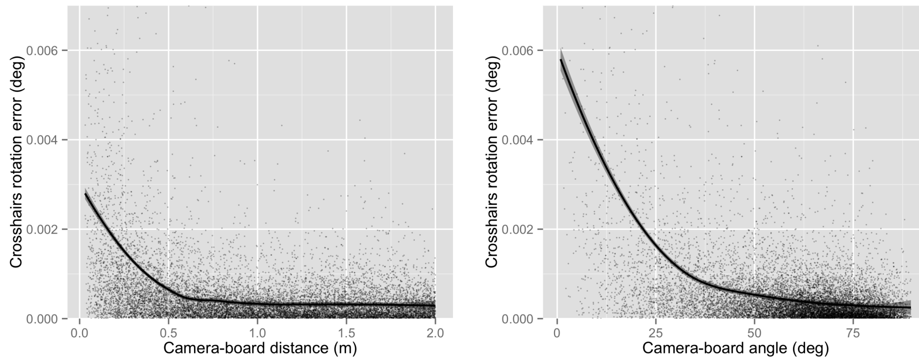


Figure 7.4. Crosshairs rotation error.

The angular error shown in Fig. 7.5 is the most interesting, as it does not depend on the camera-board distance. This is indicated by the uniform-like distribution of points.

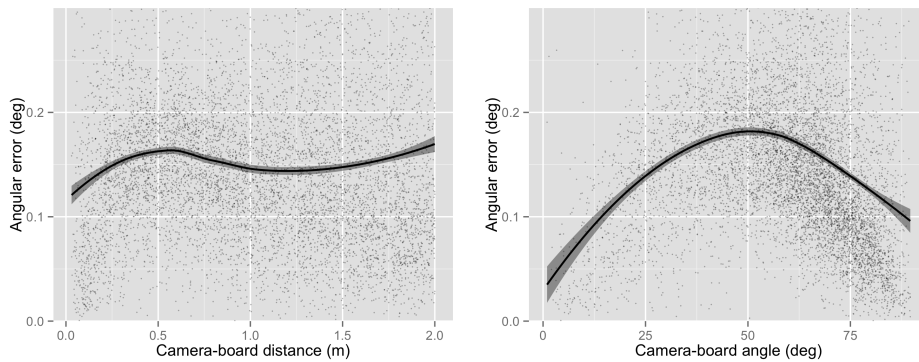


Figure 7.5. Angular error.

7.1.2 *Blurry and Noisy Data*

The trends for blurry data with respect to camera-board distance and the camera-board angle are similar to those of a clean dataset. However, the errors are higher on average. The corresponding figures can be found in the appendix. Here, we consider the effects of the size of the blur kernel. Fig. 7.6 shows how the camera pose error depends on the blur level. In general, all the errors increase, with increasing kernel size of the Gaussian blur. Similar effect is observed when the noise level is increased. The rest of the figures can be found in the appendix.

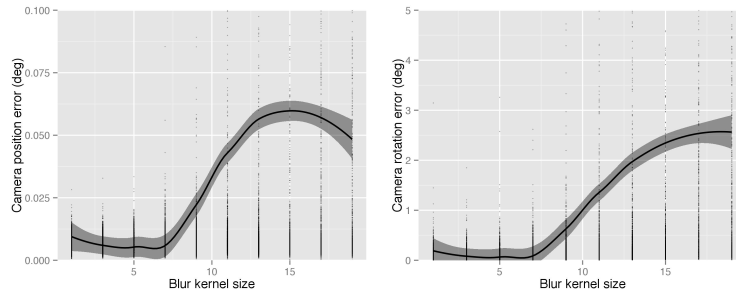


Figure 7.6. Camera position and rotation errors dependent on the size of the Gaussian blur kernel.

7.1.3 *Computation Time*

The computation time is an important criterion for our system, because it is used in real time. Computation time depends on the number of blobs in the image and on the warp size used during image rectification. Fig. 7.7 shows the computation time plotted against the number of markers in the image for clean images. And Fig. 7.8 depicts the computation time for blurry images.

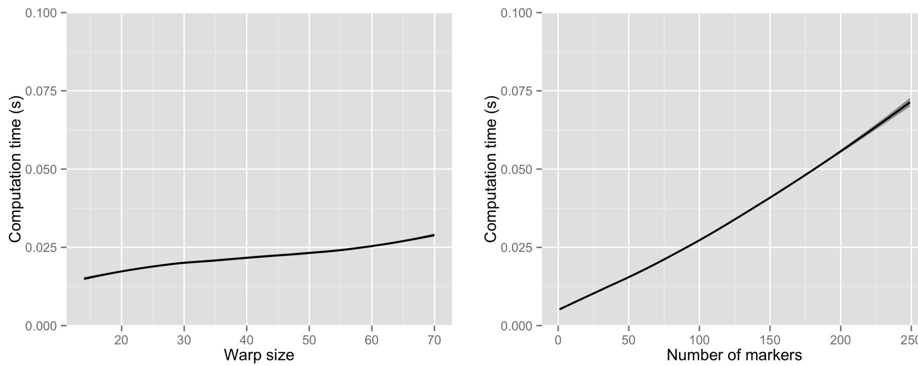


Figure 7.7. Computation time.

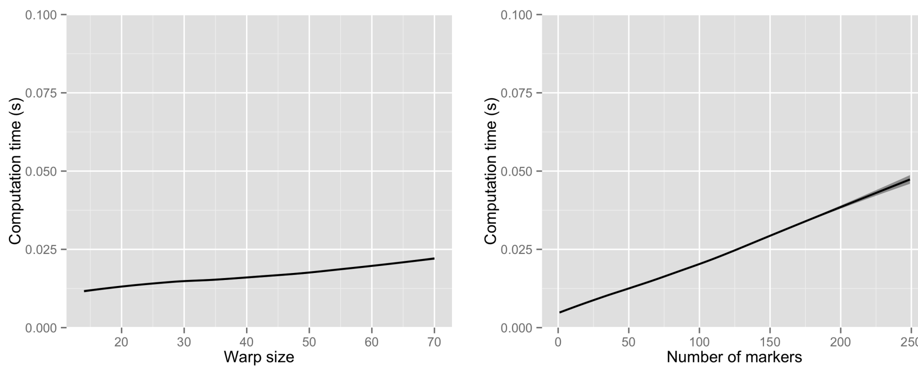


Figure 7.8. Computation time.

7.2 EVALUATION OF METHODS

This section shows the impact of using different corner refinement methods and thresholding methods on the error and computation time. Fig. 7.9 shows the crosshairs position error for each method. The combination of Canny edge detection and the lines method for corner computation has a lower error than all the other methods.

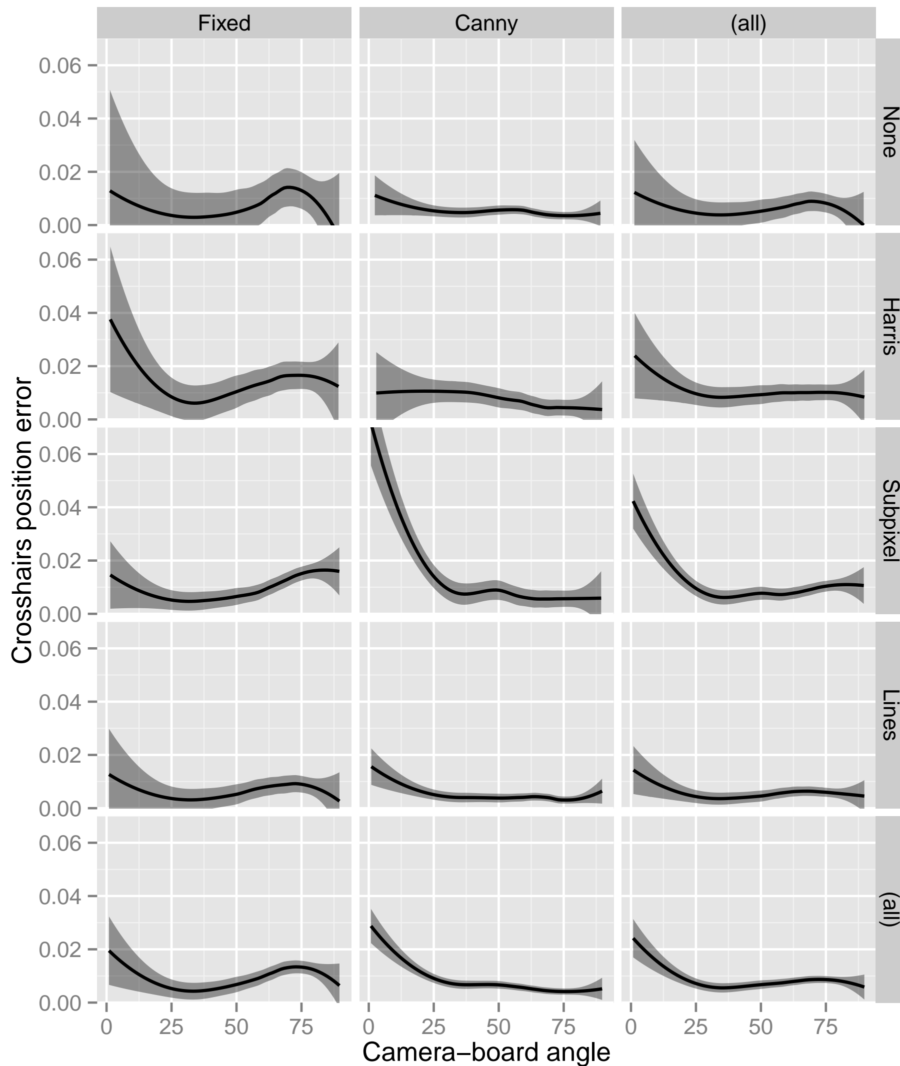


Figure 7.9. Crosshairs position error using fixed thresholding.

Fig. 7.10 shows the computation time based for each method dependent on the marker warp size. The usage of Canny edge detection requires more computation. No corner refinement requires slightly less time, however, all corner refinement methods use approximately the same amount of time.

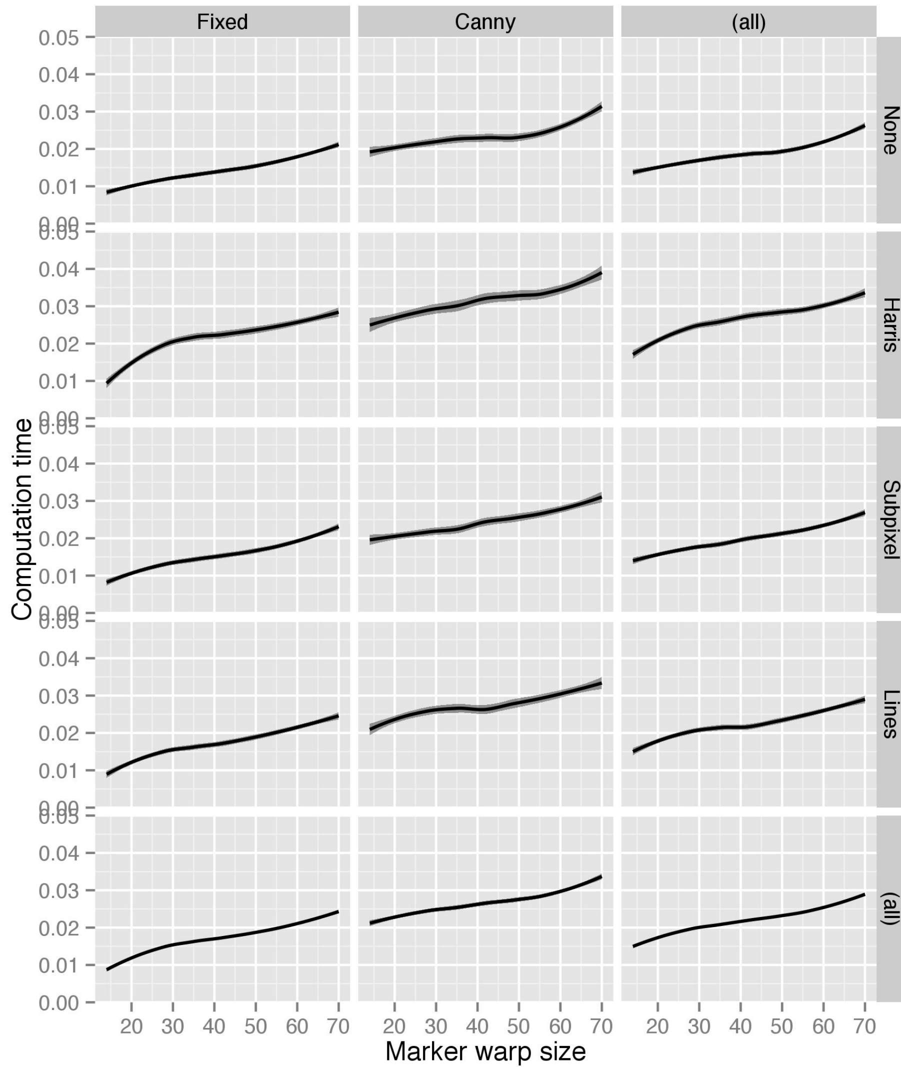


Figure 7.10. Computation time of different methods dependent on the marker warp size.

7.3 OPTIMIZED IMPLEMENTATION

The results of evaluation have been used to optimize the original implementation, to evaluate it using synthetic data, and to test the resulting system in the real world. The results show that Canny is better than the other evaluated methods for thresholding in terms of accuracy and precision, and the best method to detect the corners of the quads is the line approximation method. However, Canny also requires a much larger amount of time than all the other evaluated methods. Thus, the optimized implementation uses a fixed threshold and the lines approximation method as the basis.

In addition, a set of improvements over the original ArUco library have been incorporated in the optimized version:

1. All the marker candidates at the edges of the images are discarded because the markers are only partially visible, but still might be recognized as such. In this cases, the markers increase the evaluated errors by a large amount.
2. The results of evaluation have shown that the warp size decreases the camera pose errors by trading it off against a substantial increase of the processing time. However, at certain size, the decrease of camera pose error is not as large as the increase in the computation time, so that the marker warp size has been set to a fixed size of 49×49 pixels.
3. The error of polygon approximation method is set lower, so that the shape of the polygons has to more closely resemble a quad. The threshold in the original ArUco implementation is set to a high value so that even shapes that are not quads can be recognized as such. It has the intention of analyzing as many potential markers as possible. However, shapes that are not quads but recognized as such will always increase all errors by a large amount.
4. In addition to the improvements that are base on the insights gained during evaluation, the code of the ArUco implementation has been optimized and refactored.

7.3.1 *Evaluation Using Synthetic Data*

The optimized system shows several improvements compared to the original ArUco implementation, as shown in Table 7.1. Camera pose error and the crosshairs position error are almost halved when compared to the second best result of Canny and line approximation methods, whereas the rotation errors stay roughly the same. The computation time is substantially reduced.

Fig. 7.11 shows the average computation time of three different parts of the optimized implementation: quads detection, marker identification and camera pose estimation. The results of the tests for the undistorted dataset show that the largest portion of computation time is spent identifying markers, which involves the costly image warping operation. The computation time increases linearly with the tested numbers of markers. The number of markers is the number of markers in the image, and not the number of markers recognized by the system. This explains why the

Table 7.1. Comparison of original ArUco library with the optimized implementation by using the undistorted dataset. Bold font indicates the best value in each column. Gray color denotes the best value for each thresholding method.

Th	CA Method	Cam Pos (m)	Cam Rot (deg)	CH Pos (m)	CH Rot (deg)	Ang (deg)	Time (s)
Fixed	None	6.83e-3	1.03e-1	3.19e-3	6.81e-4	1.26e-1	1.52e-2
	Harris	8.16e-3	1.18e-1	4.86e-3	4.58e-4	1.80e-1	2.31e-2
	Subpixel	7.60e-3	1.48e-1	5.42e-3	8.32e-4	1.86e-1	1.65e-2
	Lines	6.19e-3	9.81e-2	2.97e-3	3.50e-4	1.18e-1	1.84e-2
Canny	None	7.59e-3	1.13e-1	3.75e-3	7.52e-4	1.50e-1	2.43e-2
	Harris	8.48e-3	1.13e-1	4.58e-3	4.17e-4	1.78e-1	3.25e-2
	Subpixel	8.13e-3	1.20e-1	6.42e-3	9.08e-4	1.90e-1	2.52e-2
	Lines	6.80e-3	**6.05e-2**	3.04e-3	2.79e-4	1.17e-1	2.82e-2
	Optimized	**1.25e-3**	7.24e-2	**1.50e-3**	**2.78e-4**	**5.42e-2**	**6.41e-3**

computation time to perform camera pose estimation does not increase as much as the time to perform marker identification.

In the dataset with soft Gaussian noise, the noise produces many potential marker candidates that have to be analyzed because they might be quads. Most of the time is devoted to detecting quads. In the dataset with Gaussian blur, many neighboring markers are merged by the blur kernel, which reduces the number of quads that the system is able to recognize. This, in turn, reduces the overall number of computations. If only a small number of markers is visible, which happens mostly because the camera is close to the canvas, the markers are more likely to be recognized—this explains why the system spends a larger amount of time to identify markers and to detect camera pose. In cases when many markers are visible to the camera, the blur is more likely to make the neighboring quads merge with each other.

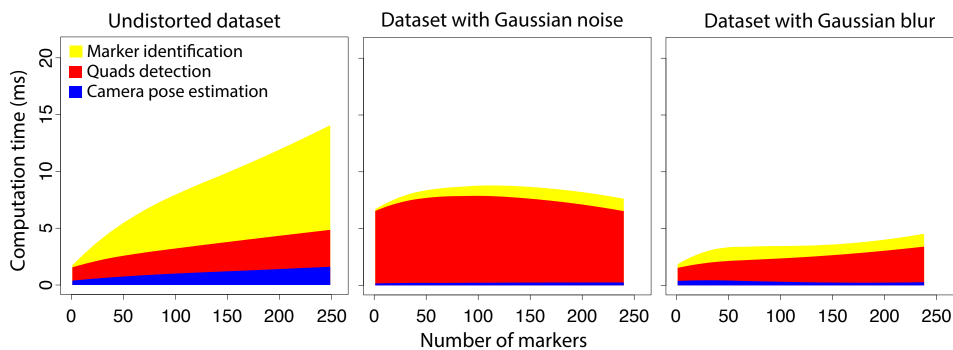


Figure 7.11. Dependency between the visible number of markers in test images and computation time of the optimized implementation for different datasets. Each color designates different phases of the marker detection procedure.

The time to process all markers in the image and to compute camera pose does not exceed 15 ms on average, which means that in the worst

case, the frame rate is 66 fps. However, in most cases, the required processing time is less than 15 ms, so that the system is able to process all images received from the camera that runs at 100 fps.

7.3.2 *Evaluation in the Immersion Square*

In order to evaluate the optimized implementation in the real world, it has been tested in the Immersion Square. The system runs on average with 80 frames per second, and is able to achieve 100 fps. In addition, the system was found to produce very stable crosshairs on the Immersion Square. One of the reasons why the crosshairs are so stable whereas the actual camera position is not is because of the way how camera pose is calculated. The Levenberg-Marquardt algorithm is used to minimize the backprojection error. If the backprojection error is high, than the corner points of the markers will highly deviate from their actual positions in the camera image. Minimization of backprojection error does not, however, minimize camera pose error. It only minimizes the error between actual data points and their backprojection.

Thus, small errors in the positions of marker corner points contribute to the camera pose error to a high degree, which makes the camera pose unstable. However, the same errors only change the position of crosshairs by a small amount, because the backprojection image is very close to the actual camera image. Since the crosshairs are always found in the center of the camera image, their position will be close to their actual position. Stable crosshairs are advantageous in a number of 3D applications where a precise camera pose is of a lesser importance.

8 CONCLUSIONS

This work has presented a system that detects fiducial markers by splitting the processing between FPGA and PC. The FPGA applies thresholding onto camera images received in raster scan order, packages the binary images into blocks, and sends them to the PC. The PC detects fiducial markers in the binary images and computes the camera pose.

The approach is evaluated by using synthetic images generated by a program that simulates a 3D space with a camera pointed at a virtual board of 16x16 markers. The test images are taken by a camera from different positions and angles. To make the evaluation process more realistic, a subset of test images is distorted by using Gaussian blur of different degrees and two types of noise.

Several approaches to marker detection are evaluated. The approaches differ with each other in terms of applied thresholding step, a corner refinement step and marker warp size. Four corner refinement methods and two thresholding methods have been evaluated. The system has been evaluated with respect to several error measurements: camera position and rotation errors; crosshairs position and rotation errors; angular error angle between two points.

The best corner refinement methods in terms of the camera pose errors is the lines method that uses the points of a contour to approximate the corners that can be found at the intersections of the lines. The best thresholding method in terms of the errors was found to be Canny. It produces results that are more precise and more accurate in terms of camera pose error. However, the fixed thresholding method and the absence of corner refinement needs the least amount of processing time.

The results of evaluation were used to implement an optimized system that achieves a good balance between precision and processing time. The optimized system is built upon a fixed threshold to binarize the images and uses the lines approximation method to detect corners of the markers. In addition, the optimized system has a marker warp size of 49, and introduces a set of improvements including code refactoring, which make the system more efficient.

The optimized system outperforms the unoptimized implementations by a large margin in terms of all errors measurements that were used to evaluate the system. The results of evaluation show that the system is

fast enough to process all camera images that it receives at 100 fps with visualization thread turned off. If the visualization thread is active, the system runs at an average of 80 fps.

Testing the system in the Immersion Square has produced an interesting finding—the position of crosshairs is very stable while the position and orientation of the camera is not. This can be explained the nature of the camera pose estimation algorithm that uses the backprojection error as a minimization criterion. Minimizing the backprojection error makes the backprojected camera image very similar to the actual camera image. Since the crosshairs are always in the center of the camera image, the crosshairs computed from the estimated camera pose will also be very close to the crosshairs in actual camera image.

The frame rate of the system can be improved by stopping the computation after a certain number of markers has been detected. This will decrease the number of equations for the SVD and Levenberg-Marquardt that are used to estimate the camera pose. The computation time can be tweaked by decreasing the marker warp size.

The main direction for future work is to implement ArUco marker detection completely on the FPGA. The approach that was pursued in the beginning of this project has turned out to be more difficult than initially estimated. The approach relies heavily on the assumption that the rectification of the potential marker candidates is not required. This assumption must be evaluated before implementing the approach completely. In case that it will result in diminished performance, image rectification will be necessary.

BIBLIOGRAPHY

[1] Bailey, D. and Johnston, C. (2007). Single pass connected components analysis. In *Proceedings of Image and Vision Computing New Zealand 2007*, pages 282–287. (cited on p. 32)

[2] Bergamasco, F., Albarelli, A., Rodola, E., and Torsello, A. (2011a). Rune-tag: A high accuracy fiducial marker with strong occlusion resilience. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 113–120. (cited on p. 11)

[3] Bergamasco, F., Albarelli, A., and Torsello, A. (2011b). Image-space marker detection and recognition using projective invariants. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2011 International Conference on*, pages 381–388. (cited on p. 12)

[4] Bochem, A., Herpers, R., and Kent, K. (2010). Hardware acceleration of blob detection for image processing. In *Third Int. Conf. on Advances in Circuits, Electronics and Micro-Electronics (CENICS)*, pages 28–33. (cited on p. 3)

[5] Bondy, M., Krishnasamy, R., Crymble, D., and Jasiobedzki, P. (2007). Space vision marker system (SVMS). In *AIAA SPACE 2007 Conference & Exposition*. American Institute of Aeronautics and Astronautics. (cited on p. 8)

[6] Bradski, G. and Kaehler, A. (2008). *Learning OpenCV*. O'Reilly Media, Inc. (cited on p. 15)

[7] Canny, J. (1986). A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698. (cited on p. 14)

[8] Chambers, J. M., Cleveland, W. S., Kleiner, B., and Tukey, P. A. (1983). *Graphical methods for data analysis*. Bell Laboratories. (cited on p. 52)

[9] Douglas, D. H. and Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2). (cited on p. 16)

[10] Fiala, M. (2005a). ARTag, a fiducial marker system using digital techniques. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005.*

IEEE Computer Society Conference on, volume 2, pages 590 – 596 vol. 2. (cited on pp. 5 and 6.)

[11] Fiala, M. (2005b). Comparing ARTag and ARToolkit Plus fiducial marker systems. In *Haptic Audio Visual Environments and their Applications, 2005. IEEE International Workshop on*, pages 128–153. (cited on pp. 4, 6, and 7.)

[12] Fiala, M. (2010). Designing highly reliable fiducial markers. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(7):1317–1324. (cited on p. 7)

[13] Hartley, R. I. and Zisserman, A. (2004). *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition. (cited on pp. 18, 21, 23, 24, and 43.)

[14] Kraft, M., Schmidt, A., and Kasinski, A. (2008). High-speed image feature detection using FPGA implementation of FAST algorithms. In *VISAPP 2008: Proceedings of the Third International Conference on Computer Vision Theory and Applications*, volume 1. INSTICC - Institute for Systems and Technologies of Information, Control and Communication. (cited on p. 32)

[15] Lamb, P. (2007). ARToolKit http://www.hitl.washington.edu/artoolkit/. (cited on p. 5)

[16] Lansche, U. (2012). *mvBlueCOUGAR-X documentation, V1.0b24*. (cited on p. 37)

[17] López de Ipiña, D., Mendonça, P. R. S., and Hopper, A. (2002). TRIP: A low-cost vision-based location system for ubiquitous computing. *Personal Ubiquitous Comput.*, 6(3):206–219. (cited on p. 8)

[18] Ma, N., Bailey, D., and Johnston, C. (2008). Optimised single pass connected components analysis. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 185 –192. (cited on p. 32)

[19] Munoz-Salinas, R. (2012). ArUco: a minimal library for Augmented Reality applications based on OpenCV. http://www.uco.es/investiga/grupos/ava/node/26. (cited on p. 2)

[20] Owen, C., Xiao, F., and Middlin, P. (2002). What is the best fiducial? In *Augmented Reality Toolkit, The First IEEE International Workshop*, page 8 pp. (cited on p. 5)

[21] Poupyrev, I., Kato, H., and Billinghurst., M. (2000). *ARToolKit version 2.33: A software library for Augmented Reality Applications*. (cited on p. 5)

[22] Ramer, U. (1972). An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244 – 256. (cited on p. 16)

[23] Rosten, E. and Drummond, T. (2005). Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pages 1508–1515 Vol. 2. (cited on p. 32)

[24] Samarin, P., Herpers, R., Kent, K. B., and Saitov, T. (2012). Evaluation of data transfer from FPGA to PC: Increasing frame rate by blob detection. Technical report, Bonn-Rhine-Sieg University and University of New Brunswick. (cited on pp. 2 and 3.)

[25] Sattar, J., Bourque, E., Giguere, P., and Dudek, G. (2007). Fourier tags: Smoothly degradable fiducial markers for use in human-robot interaction. In *Computer and Robot Vision, 2007. CRV '07. Fourth Canadian Conference on*, pages 165–174. (cited on p. 10)

[26] Sauvola, J. and Pietikäinen, M. (2000). Adaptive document image binarization. *Pattern Recognition*, 33(2):225 – 236. (cited on p. 13)

[27] Scherfgen, D., Saitov, T., Herpers, R., and Dayangac, E. (2011). An optical laser-based user interaction system for cave-type virtual reality environments. In *Proc. of the 4th Russian-German Workshop "Innovation Information Technologies: Theory and Practice"*. (cited on p. 2)

[28] Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Pearson Education, Inc., fourth edition. (cited on p. 29)

[29] Shapiro, L. and Stockman, G. (2001). *Computer Vision*. Prentice Hall. (cited on p. 47)

[30] Shreiner, D. (2010). *OpenGL programming guide : the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, Inc., 7th edition. (cited on p. 41)

[31] Suzuki, S. and Abe, K. (1985). Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32 – 46. (cited on p. 15)

[32] Terasic (2009). De2-70 development and education board. [Onlne]. Available: http://www.altera.com. (cited on p. 37)

[33] Wagner, D. and Schmalstieg, D. (2007). ARToolKitPlus for pose tracking on mobile devices. In *Computer Vision Winter Workshop 2007*. (cited on p. 5)

[34] Xu, A. and Dudek, G. (2011). Fourier tag: A smoothly degradable fiducial marker system with configurable payload capacity. In *Computer and Robot Vision (CRV), 2011 Canadian Conference on*, pages 40–47. (cited on p. 10)

APPENDIX

A

A.1 BLURRY DATASET

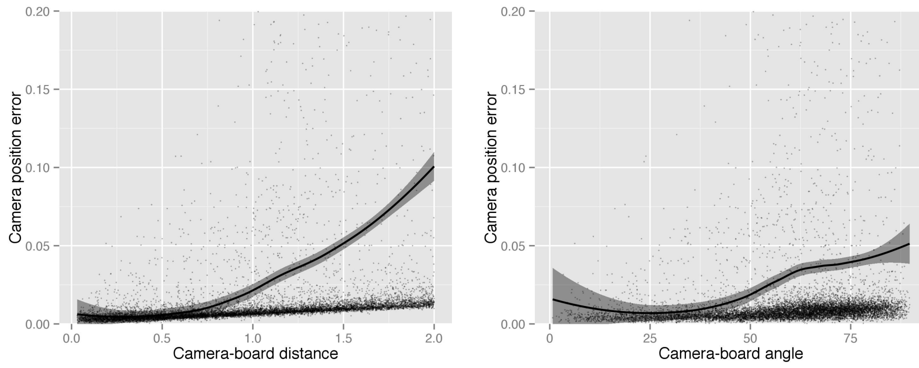


Figure A.1. Camera translation error.

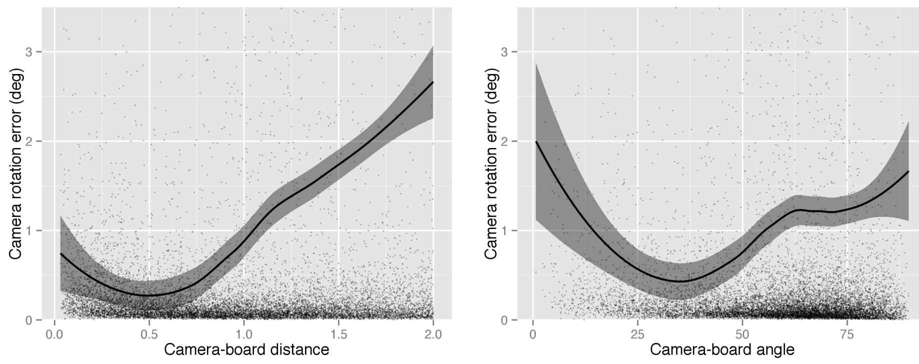


Figure A.2. Camera rotation error.

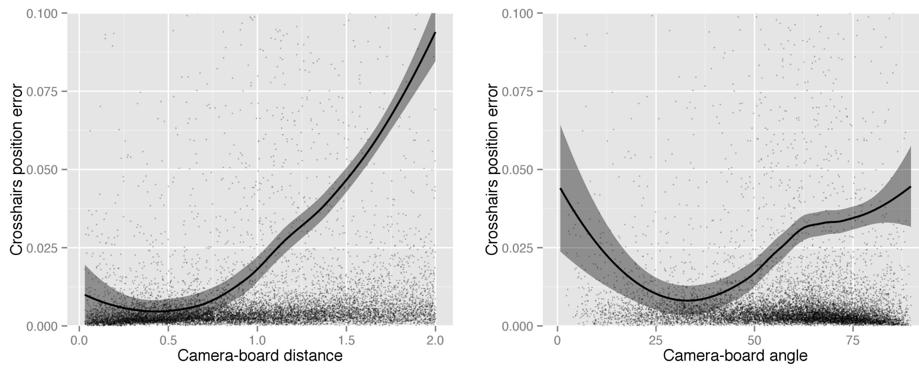


Figure A.3. Crosshairs position error.

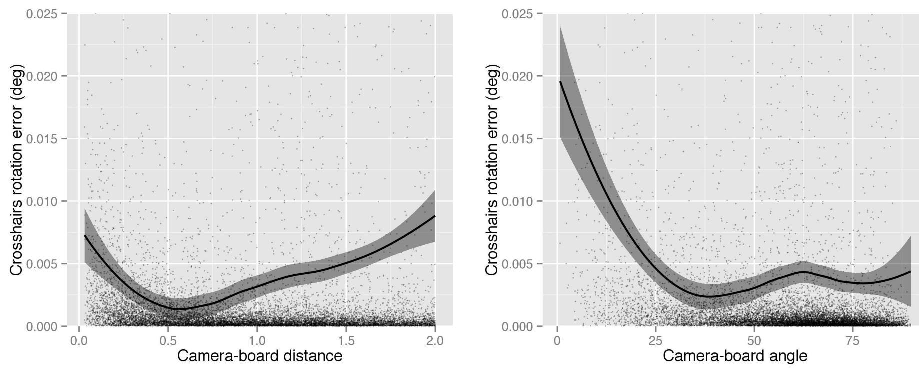


Figure A.4. Crosshairs rotation error.

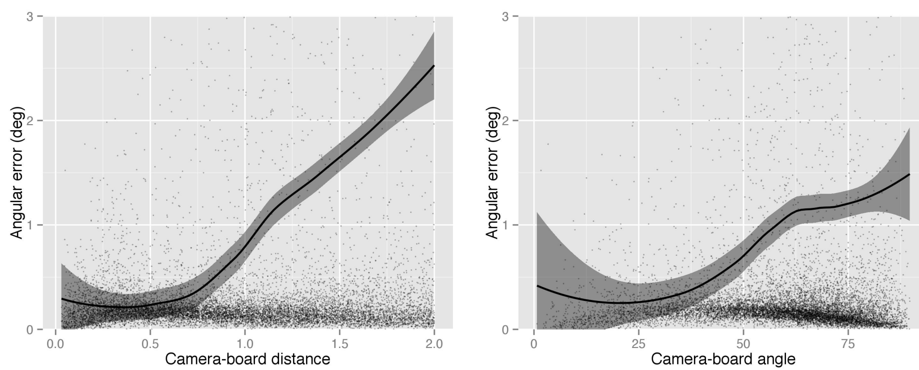


Figure A.5. Angular error.

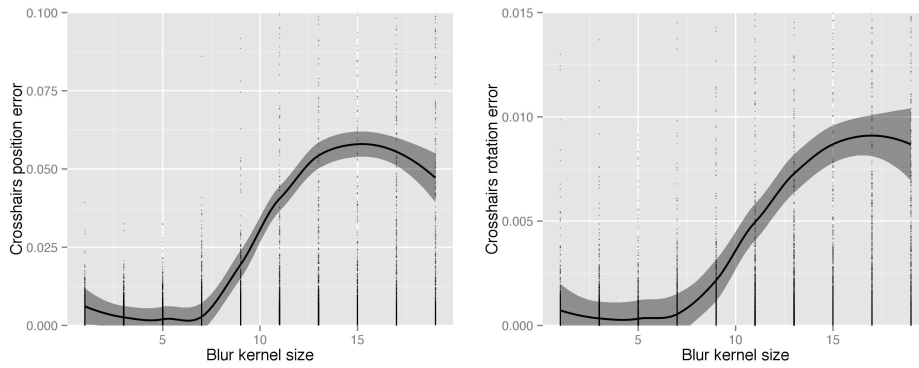


Figure A.6. Crosshairs position and rotation errors dependent on the size of the Gaussian blur kernel.

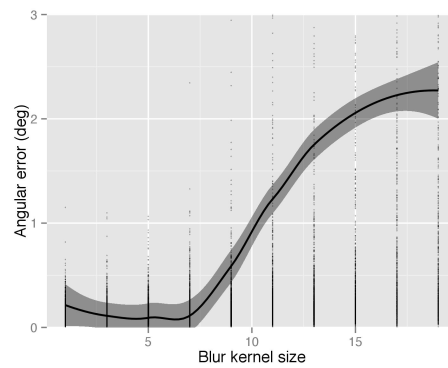


Figure A.7. Crosshairs position and rotation errors dependent on the size of the Gaussian blur kernel.

A.2 DATASET WITH GAUSSIAN NOISE

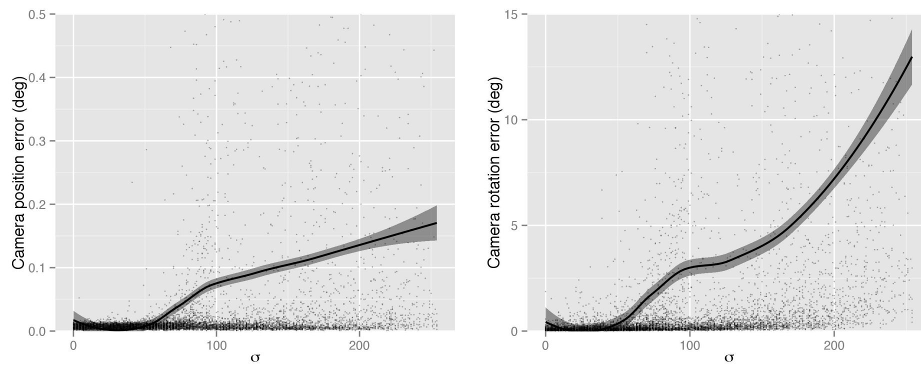


Figure A.8. Camera position and rotation errors dependent on the size of the Gaussian noise kernel.

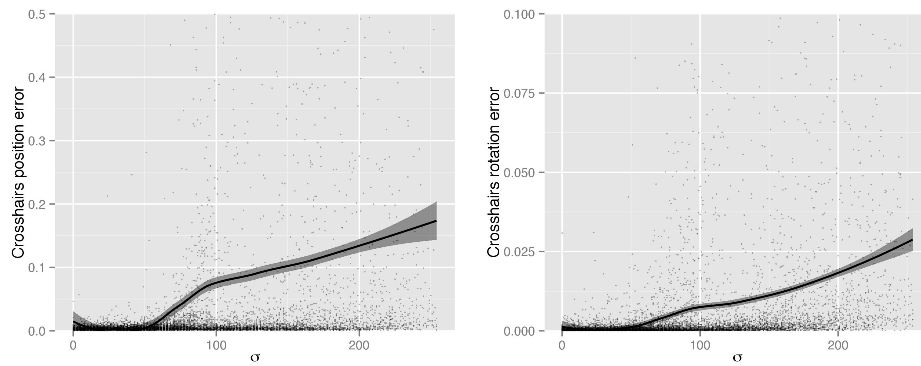


Figure A.9. Crosshairs position and rotation errors dependent on the size of the Gaussian noise kernel.

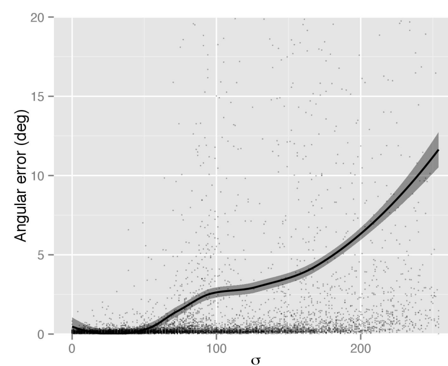


Figure A.10. Crosshairs position and rotation errors dependent on the size of the Gaussian noise kernel.