Czech Technical University

Faculty of Electrotechnics

Bachelor Thesis

Localization system for a multi-robot platform

Author:

David Kolečkář

Program: Open Informatics

Specialization: Computer Systems

Prague, summer 2017

Advisor:

Ing. Jan Chudoba

Department of Cybernetics

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická
Katedra měření

Akademický rok **2016-17**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	**David Kolečkář**
Studijní program:	**Otevřená informatika**
Obor:	**Počítačové systémy**
Název tématu česky:	**Lokalizační systém pro multi-robotickou platformu**
Název tématu anglicky:	**Localization System for a Multi-robot System**

Pokyny pro vypracování:

Úkolem práce je návrh a implementace systému pro lokalizaci mobilních robotů, pohybujících se v omezené oblasti, metodami zpracování obrazu několika kamer snímajících pracovní oblast za předpokladu, že roboty budou vybaveny vizuálními vzory umožňujícími jejich identifikaci.

Seznamte se se základy 3D počítačového vidění a metodami pro robustní vyhledávání předem definovaných vizuálních vzorů v obraze. Po konzultaci s vedoucím práce zvolte vhodnou metodu detekce vizuálních vzorů. Implementujte systém pro současné zpracování obrazu z několika kamer, výpočet poloh rozpoznaných robotů a poskytování informací o poloze dalším aplikacím. Navrhněte proceduru kalibrace celého systému. Vytvořte prototyp systému s několika kamerami a vyhodnoťte dosažitelnou přesnost lokalizace.

Seznam odborné literatury:

[1] Šonka, M., Hlaváč, V.: Počítačové vidění. Grada, Praha 1992

[2] Olson, Edwin, Tag, April: A robust and flexible visual fiducial system, in Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), p. 3400-3407, IEEE, 2011

[3] Krajník, T., Nitsche, M., Faigl, Vaněk, Saska, Přeučil, Duckett, Mejail: A practical multirobot localization system, Journal of Intelligent and Robotic Systems, Heidelberg, Springer (2014).

Vedoucí bakalářské práce:	Ing. Jan Chudoba (K 13133)
Datum zadání bakalářské práce:	17. února 2017
Platnost zadání do[1]:	30. září 2018

L. S.

Prof. Ing. Jan Holub, Ph.D.
vedoucí katedry

Prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 17. 2. 2017

[1] Platnost zadání je omezena na dobu tří následujících semestrů.

Declaration

Thereby I declare that all work was done on my own and all sources used are cited.
Prague 22.5.2017

Acknowledgements

I would like to thank mr. Chudoba for his time and for introducing me to this interesting subject, I have never before dealt with. To my family for all the support and last not least to Liška for all the happiness and love.

Thesis Abstract

This work aims at learning the basics of 3D computer vision and object tracking. Models used for representing the geometry of the scene and theorems used for projecting the object to the image are described. This work also gives overview over popular camera calibration methods and over the popular robust tag tracking methods. The process of forming 2 dimension camera image from three dimensional object in front of the camera is described, with respect of different coordination systems used during the process. This work aims also to get familiar with the popular software tools, applications and libraries used in computer vision.

Abstrakt práce

Cílem této práce je seznámit se základy počítačového vidění , robustního vyhledávání vzorů z obrazu. Je popsána geometrie a základy teorie potřebné k pochopení vytvoření obrazu kamerou. Dále je popsán využitý software a nástroje při tvorbě aplikace. Na závěr jsou poznatky z teorie implementovany kódem.

Key words

computer vision, camera calibration, AprilTag, camera pose detection, OpenCV

Klíčová slova

strojové vidění, kalibrace kamery, AprilTag, určení pozice kamery, OpenCV

Contents

1 Introduction

First task was to get familiar with robust pattern recognition software and techniques that could be used for solving the task. During this part of research also learn the basics of computer vision, which comprises from several separate but related topics as projective geometry, camera and single view geometry and multiple view geometry. It is also necessary to be familiar with linear algebra, matrix operations and optimization methods. While I had no previous experience with computer vision as subject of study from before I had been expecting that this goal will be altogether difficult, as some basic solution could be found after grasping the basics but to get more reasonable and precise results the theoretical and mathematical machinery could be very complex.

Second goal was to do proper research and find out possible candidate for software that will be used, to choose one or two – for comparison – and install the software. As I have recalled the robotics and computer vision software and tools are becoming more widespread today and some are open-source, so I was expecting that there will be some base I could work with, even though I have never worked with such a software before. Next to read the documentation of the chosen software and to get familiar with the libraries that would be used, possibly to find different ports and to choose the most suitable one for the application and programming language. I have been expecting that this goal also could be a bit demanding, while getting hands on new software usually takes some time.

Third goal was - after getting the theoretical basics and having the overview of the problem and with found sufficient software – to try figuring out possible solutions for the final application structure, needed tools and equipment, also as researching how the similar problems are usually solved.

Final goal was implementing the solution with knowledge of previously learned theory and acquired computer vision and robust pattern recognition software tools. Building the real world model and experimenting with robust pattern recognition with data from multiple camera system.

2 Introduction to theory basis

In this chapter I try to introduce the necessary knowledge base in order to understand the building blocks, concepts and algortihms used for creating logic behind the camera platform. I start by geometry of camera and how image is made by it, all the theory mentioned is afterwards used in the software solution.

2.1 Introduction to Camera and Single View Geometry

During the process of forming the 2-dimensional images of the 3-dimensional world objects with camera, four different changes in coordinate system must be taken into count. 2D image point coordinates are in most literature denoted by lower-case bold letters $\boldsymbol{x}$ and 3D object point coordinates are written in upper-case bold letters $\boldsymbol{X}$. Coordinates could be expressed either in
inhomogeneous coordinates : $\boldsymbol{x} = [x,\ y]^T$, $\boldsymbol{X} = [X,\ Y,\ Z]^T$
or in homogeneous coordinates $\boldsymbol{x} = [x,\ y,\ 1]^T$, $\boldsymbol{X} = [X,\ Y,\ Z,\ 1]^T$.

2.2 Homogeneous coordinates

In computer vision tasks and projection geometry is often very convenient or even necessary to compute the solution - to express coordinates of points in so called homogeneous coordinates. In homogeneous coordinates $(x, y, 1)$ and $(3x, 3y, 3)$ and (kx, ky, k) for $k \neq 0$ represent the same point, or in other words the points that differ in common multiple represent the same equivalence class. Converting inhomogeneous coordinates of two dimensional point to homogeneous coordinates is done by $[u, v]^T \rightarrow [ku, kv, k]^T$ the inversion back to inhomogeneous from homogeneous coordinates is achieved by $u = u/k$ and $v = v/k$. When converting from inhomogeneous coordinates to homogenous the k is often choose as $k = 1$, because it could by seen like scaling. Thus scaling by 1 is the same. $[u, v]^T \rightarrow [u, v, 1]^T$. Some important reasons for usage of homogeneous coordinates are

- enables non-linear mapping such as perspective projection to be represented by linear matrix equations
- to express points in infinity more easily without using limits
- intersection of lines express as cross product of two vectors

The topic of geometrical entities in infity is more complex so for further information see [3] or [1]

Degrees of Freedom (DOF) It is also important to refer about degrees of freedom abr.DOF . $\boldsymbol{x} = [x, y]^T$ representing a 2D point in $\boldsymbol{R}^2$ has in homogeneous coordinates three entries but it has only two parameters that can vary independently x and y. So the degrees of freedom could be defined as the number of parameters that can vary independently. [1]

2.3 Four coordinate systems

World coordinate system First coordinate system is the 3-dimensional Euclidean coordinate system of the world. Any world 3D object could be expressed in arbitrary coordinate system. Consider a room with two pins lying on the ground. Observer could say that origin of this world coordinate system is exactly where the pinhead of the first pin is. So could the observer say that the origin will be at the head of the second pin. Or the origin could be choosen to be anywhere else. So long as coordinate system in arbitrary space is human abstract, up to this point – while referring about arbitrary world objects, it is entirely up to observer, where will be the origin placed and how will be the axes directed.

Camera coordinate system Second coordinate system is the 3-dimensional Euclidean coordinate system of the camera. This coordinate system expresses world objects related to the frame of the camera. Origin is located at the optical center O. The coordinate axis Z lies on the optical axis and its positive direction is pointing from the camera to the scene in front of the camera. Not always but usually it is said that Y axes points from up to down in its positive direction. The anatomy and geometry of camera is described later.

Image plane coordinate system Third coordinate system is the 2-dimensional Euclidean coordinate system of the image plane. This coordinate system has its axes aligned with the camera coordinate system, axis lying in the image plane corresponding to Camera coordinate system axis X will be denoted u_i and second image plane axis lying in this plane and corresponding to Camera coordinate system axis Y will be denoted v_i .

Pixel coordinate system also called image affine coordinate system (see paragraph Intrinsic camera parameters as 2D transformations). It represents the coordinate system of the pixels on the chip.

[1]

2.4 Camera Representation

Introduction The most simple Camera model we can consider - a camera which has no lens to focus light and has a small aperture that could be considered as a single point is called pinhole camera. This point is denoted as optical center O(in some literature also called center of projection). In the means of geometrical optics we could describe a ray of light by a single line. The light rays passing through the pinhole camera form behind the camera a 2 dimensional inverted image of the 3D world in front of the pinhole camera, this plane where the world scene is projected is called image plane. The axis perpendicular to the image plane and passing through the optical center is called optical axis.

Pinhole Camera Model Pinhole Camera Model describes geometry and the mathematical model of relations between a world – three dimensional - objects and their two dimensional projections onto a image plane using a pinhole camera. By virtual image plane is meant a virtual plane in front of the pinhole camera containing the upright image of the scene. Since pinhole camera is considered to have no lenses, this model does not include geometric distortions – aberrations - of image created by light passing through real lenses, while no lens is perfect. Using this model is considered suitable for most computer vision tasks .

We could further consider that our camera have a thin lens, which offers still good approximation to pinhole camera. More parameters to our model can be added, since the lenses have some inherent parameters. Rays parallel with optical axis after passing through the thin lens bends (in case of convex lenses - such as the lens in human eye- after passing through lens towards the optical axis) and intersect behind the lens with optical axis in point called focal point. Focal length is than the distance of the focal point from the optical center. Image plane is located at from the optical center in the focal length.

Aberations For this topic we could consider few common lens aberrations. Spherical aberration occurs when light beam coming into lens is too wide, making paraxial rays bend more than those on the edges of the lens.

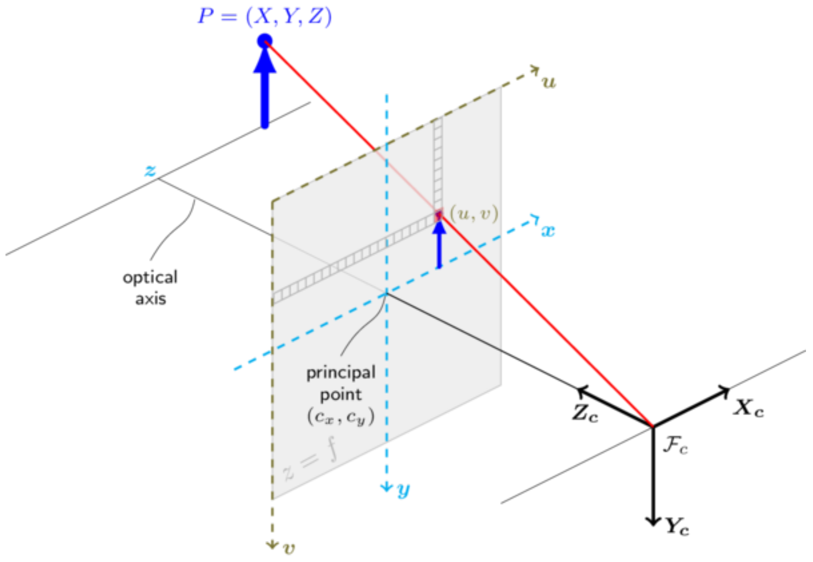

Figure 1: Simple pinhole camera geometry. F_c is the center of projection = optical center, f is the focal length. (Image courtesy of OpenCV documentation)

central projection Let be the optical center of the camera the center of projection of 3D scene to its 2D image in in the image plane. Rays passing through the centre of projection intersects with the image plane. This intersection could be seen as a image of the ray thus all the points lying on this ray. This geometricaly describes the idea of equivalence classes in homogeneous vectors. While by choosing different scales represents moving on the ray.

2.5 Transformations between coordinate systems

To project 3-dimensional world object in metres in front of the camera on the 2 dimensional image on the camera's chip in pixels, we have to make three transformations between these previous four coordinate system. In the end I will show that all these transformations can be described as one 3x4 homogeneous matrix P.

2.5.1 Extrinsic parameters

This section describes transformation between arbitrary world coordinate system and camera coordinate system. This transformation is composed of rotation and translation which relates those two coordinate systems. These parameters can be expressed as extrinsic matrix.

Extrinsic camera matrix It is rotation-translation matrix composed of a 3x3 rotation matrix R and by translation 3x1 vector $\boldsymbol{t}$. Columns of **R** expresses rotations along the X , Y and Z coordinate axes and the vector **t** represents translations in X, Y and Z axis directions. The translation could be geometricaly seen as aligning origins of these two coordinate systems and rotation represents aligning the axes. Both rotation R and translation $\boldsymbol{t}$ have the property of isometry - rigid transformation - with the property of preserving distances between every pair of points.

Expressing point $\boldsymbol{X_w}$ in world coordinates as point $\boldsymbol{X_c}$ in camera coordinate system represents following equation in inhomogeneous coordinates. Both points represent the same 3D point but in different coordinate systems.

$$\boldsymbol{X_c} = \boldsymbol{R}(\boldsymbol{X_w} - \boldsymbol{C}) \tag{1}$$

Where $\boldsymbol{C}$ is camera center in the world coordinate system and $\boldsymbol{R_c}$ describes orientation of the camera frame in it. [3] .
So to express point from world coordinate system in camera relative coordinate system you must perform this operation but $\boldsymbol{C}$ itself represents optical center(camera coordinate system) in world coordinate system. This often causes a confusion. To join **R** and **t** in one matrix it is necessary to express elements of this equation in homogeneous form (in order to match matrix dimension). So for this equation the extrinsic matrix would have form of

$$\boldsymbol{X_c} = \left[\begin{array}{c|c} \boldsymbol{R} & -\boldsymbol{RC} \\ \hline \boldsymbol{0} & 1 \end{array}\right] \boldsymbol{X} \tag{2}$$

It makes it also square, wich is useful for following formulations of excintric matrix and computations.

I would like to show here two differences that are not clear from many literature and can cause lot of confusion, since the notation is not always the same.

Substitution In the equation (1) we can substitute $-\boldsymbol{RC} = \boldsymbol{t}$ which represents the position of the world origin in camera coordinates. So now the equation is

$$\boldsymbol{X_c} = \boldsymbol{R}\boldsymbol{X_w} + \boldsymbol{t} \tag{3}$$

and the matrix can be written as

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4}$$

Or to see it other way around when we need to find out the camera center in world coordinates

$$\boldsymbol{C} = -\boldsymbol{R}^T\boldsymbol{t}$$

During these operation is used the fact that inverse of rotation matrix is equal to its transpose and the inverse of translation vector is its negation.

To derive previous equations is very important, while this could be on of the key parts in understanding relationships in task of finding 3D position of visual pattern. [3] , [2]

Rodrigues formula Rodrigues rotation formula computes the 3 dimensional rotation matrix corresponding to a rotation by an angle θ about a fixed axes specified by a unit vector $\boldsymbol{r} \in R^3$ [4]

Many OpenCV functions accepts rotation matrices represented by the rotation vector, since any rotation matrix has 3 degrees of freedom. OpenCV uses the Rodrigues rotation formula for the computation of rotation 3x3 matrix from the 3x1 vector .The rotation vector $\boldsymbol{r}$ could be transformed to rotation matrix $\boldsymbol{R}$ by OpenCV function cv::Rodrigues( ) which transforms r to R and vice versa by following (Rodrigues) formula. For input vector $\boldsymbol{v}$ Angle of rotataion θ equals to $\theta = |\boldsymbol{v}|_2 = \sqrt{v_1^2 + v_2^2 + v_3^2}$
Axis of rotation $\boldsymbol{r}$ is normalized input vector $\boldsymbol{v}$

$\boldsymbol{r} = \frac{\boldsymbol{v}}{\theta} = \frac{\boldsymbol{v}}{\sqrt{v_1^2+v_2^2+v_3^2}}$

$R = \boldsymbol{I} + \sin\theta\boldsymbol{w} + (1 - \cos\theta)\boldsymbol{w^2}$ where w is antisymmetric matrix $\begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$

See [5] for full documentation.

2.5.2 Intrinsic parameters

The intrinsic camera parameters could be seen as affine 2D transformation or as parameters of camera's internal geometry.

Object to image In this part the transformation between the camera frame coordinate system and the camera image coordinate system will be described. This is the part when comes to significant information loss as we express three dimensional objects in just two dimensions. Thus it is $R^3 \to R^2$ projection. The following expressions are derived from trigonometry, from triangle similarities.

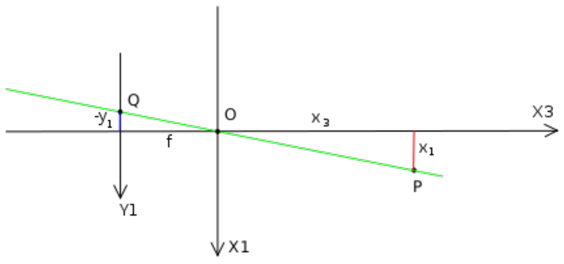

Figure 2: triangle similarities. (Image courtesy of OpenCV documentation)

Included image description: f is the focal length u could be one of the image plane coordinates and the X_1 represents the corresponding camera frame 3D coordinate. Z axis is denoted as X_3 . O is the optical center of camera and image plane is denoted here as Q . Image depicts lookin in up direction of Y coordinate of the camera coordinate system. From the triangle similarities comes the following equations

$$\frac{u_1}{f} = \frac{x_1}{z_1} \qquad \frac{v_1}{f} = \frac{y_1}{z_1} \tag{2}$$

The triangle similarity for axes v of image plane and Y of camera frame can be easily derived as previous equation just by lookin on the same picture but just from to X camera axis this time.

from these equations it is now possible to derive relation between camera and image coordinates systems. $u_i = \frac{X_c f}{Z_c}$ and $v_i = \frac{Y_c f}{Z_c}$

transformation of a point written in homogeneous notation

$$u_i = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} X_c \tag{3}$$

Offset of principial point The camera's "principal axis" is the line perpendicular to the image plane that passes through the pinhole. Its itersection with the image plane is referred to as the "principal point,". Nevertheles due to imperfections this principial point can be offset. It can be imagined as when increasing or decreasing these parameters shifts the pinhole to the left or right or up and down respectively.

Intrinsic Matrix

$$K = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Intrinsic matix parametrized in pixel coordinatex by [3]

Each of these parameters represent geometric property of the camera.

- Focal length f_x , f_y , is distance of the focal point from the optical center and represents the scaling along u and v image axes respectively.
- Principial point offset $u_0 \quad v_0$
- Axis skew s, in representation of pinhole camera there would be no skew, while it is the property of cameras where film, or photosensitive chip is placed off-perpendicular to optical axes. It is often omited.
- for parameters in pixels $\alpha_x = pixel\_size_x * f_x$, similarly for the rest of parameters

Intrinsic camera parameters as 2D transformations Intrinsic camera matrix can be view as a set of affine 2D transformations. Matrix $\boldsymbol{K}$ can now be decomposed as a sequence of scaling, shear and translation transformations. We could view Intrinsic camera transformations like as they occur "post-projection" from the world scene to image plane, while it is performed within the image plane.

$$\underbrace{\begin{bmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{K} = \underbrace{\begin{bmatrix} 1 & 0 & u_0 \\ 0 & 1 & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{2D Translation} \times \underbrace{\begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{2D scaling}$$

*shear is omitted.

2.5.3 Camera matrix

Now with both intrinsic and extrinsic parameters we can define the Camera matrix **P** which represents complete projection of 3D scene point **X** into its image **x**.

$$\boldsymbol{x} = \boldsymbol{PX} = \boldsymbol{KR}[I| - \boldsymbol{C}]\boldsymbol{X} \tag{6}$$

2.6 Planar Geometry

2.6.1 Homography

Also called *projective transformation* because it represents the transformation of plane projected (the central projection) by camera or *collineation* because lines are mapped to lines - which is one of the geometric properties that projective transformation preserves.

It is defined as mapping : $P^2 \rightarrow P^2$ and it is a homography only if there exists a non-singular 3x3 matrix H such that for any point x in P^2 $h(x) = Hx$. By stating that matrix H is non-singular is expressed that it has inverse H-1 thus the mapping is invertible. [3]

This is one of the most important concepts for solving this task , because it provides mathematical instrument to express relation between original plane and its image (planar AprilTag and its image projected by camera).

Finding H from x' = Hx is solved by point correspondences from world and image. Each such correspondence gives two equations, which are linear in elements of H. Thus 4 point correspondences are needed to solve H up to multiplicative factor. The points must be in general position = no three point are collinear.

$$\text{here } \boldsymbol{x'} = \boldsymbol{Hx} \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

2.7 Camera resectioning

(or in some literature also know as camera calibration)
The problem is the approximation of the camera projection matrix P from 3D object – 2D image correspondences. The simplest such correspondences are 3D points X and their 2D images x We want to find Psuch that

$$\boldsymbol{x} = \boldsymbol{PX} \tag{7}$$

where the $\boldsymbol{P}$ is 3x4 projection matrix, $\boldsymbol{x} = (x,\ y,\ 1)$ is the image homogeneous point and $\boldsymbol{X} = (X,\ Y,\ Z,\ 1)$ is the object homegeneous point. The basic solution is using the DLT algorithm similarly as for the homography in planar projection (where was x' = Hx). The difference here comes from different dimensions of points, H is 3x3 , P is 3x4. While P could be computed by DLT and intrinsic and extrinsic matrices can be decomposed by e.g. SVD and QR, more precise solutions exist.

3 Used Software

Research introduction From research on current methods and popular solutions for robust pattern tracking and recognition I have decided - after reading [6] - that choosing AprilTag could be the best software option for creating the most precise and solid solution. It is also very popular solution in the field of robotics and computre vision. Using AprilTag library toolkit with OpenCV seems to me like the best option, while OpenCV offers a lot of possibilities and tools for usage with other libraries and and has a very active developer-user base. OpenCV does not only mediates image acquisition from camera for the application software but also provides a whole collection of useful methods and algorithm implementation that could be also used for solving tag detection tasks and following manipulation with matrices and vectors.

Used Libraries This application uses standard C/C++ libraries, OpenCV , ApriTag C++ port by [7]

3.1 OpenCV

is a widely used library for computer vision image manipulation and processing. It is multi-platform – supporting Linux, Windows, iOS and Android - and free software developed under BSD license. The library offers multiple language interface, supporting C, C++, Python and Java. OpenCV is focusing on real-time applications, so it usage is also - besides many other fields of interest - popular in robotics. The library suits perfectly my task for its wide variety of functions designed for computational efficiency and multi-core processing. Besides OpenCV is very well documented and has a big active user-developer community.

3.1.1 OpenCV interface

For anyone who would wish to master her or his skills with using the OpenCV library I recommend visiting a OpenCV official sites and look for the tutorials, which I found very useful and helping. [8]. I would still like to present some of OpenCV structures and methods used in my application.

- cv::Mat A basic image continer in OpenCV - a pixel image matrix - is represented by this class. It is not just a matrix nevertheless. It has multiple specifications such as number of color channels. The operation with cv::Mat are very similar to operating with matrices in e.g. Octave or Matlab, but the unexperienced user could encounter some difficulties. While performing matices operations one must not forget, that cv::Mat object has more properties than just rows and columns, but also the number of color channels and specified data type as OpenCV constants `CV_64F` representing 64bit float value or `CV_64D` representing 64bit double value. In my case it have been often reason of various errors which were harder to identify in the code.

- cv::Point Another class I have widely used is, representing a point in chosen dimension and data format. For example creating three dimensional point in double values would be following: `cv::Point3d z(0, 0, 0);` , where '3' after Point represents 3 dimensions and the letter 'd', that data will be doubles. Elements of cv::Point could be accessed as following. Accessing x coordinate of point z is simple `double xCoordinate = z.x`

  It is often necessary to represent a cv::Point object as a cv::Mat object, in order to pass it as a argument to some OpenCV function. This could be achieved as demonstrated on following example. Create 4x1 cv::Mat of 3D points of doubles when having an array of 3D points of doubles

```
cv::Point3d points[4] = {cv::Point3d(-1, -1, 0),cv::Point3d(1, -1, 0),
                         cv::Point3d(1, 1, 0),cv::Point3d(-1, 1, 0) }
cv::Mat_<cv::Point3d> pointsMat(4, 1, points);
```

  Then I found important to mention how to access such matrix, while it caused me some trouble during developing the application. For example x coordinate of second point of matrix pointsMat could be accessed by two ways `pointsMat[1] ->x;` or by `tpointsMat.at<double>(1,0);`

- `cv::VideoCapture` class enables reading data from buffer in memory,
  by calling `open(dev.device\_number)` method on `cv::VideoCapture` object. In my case as argument is passed a camera device path.

- `cv::waitKey(x)` function does two things. 1. It handles any windowing events such as showing images with `cv::imshow()`. If this function wasn't called after `cv::imshow()` highgui is never given time to process the draw requests, thus nothing would diplay on the screen. 2.It waits for x milliseconds.

3.2 AprilTag

is called both software library used for detection of robust visual patterns – tags - and these tags themselves. This library is based on research described in papers [6]. The tags are similar to more known QR Codes - 2D bar codes – in sense of planar black and white images divided into squares with information coded in these patterns. AprilTags instead of QR carry much less information, just from 4- to 36 bits (as for the Tag36h11 family) making their detection easier and more robust and possible even from larger distances and narrower angles. From these characteristics arise multiple usages of AprilTag in robotics from camera calibration to determination of 3D position of the tag. The standard AprilTag library has implementations in both C and Java(now deprecated) . AprilTag software is also open-source and sufficiently well documented. Also algorithms used by this software are described in detail in AprilRoboticsLaboratory papers.

3.2.1 Tags description

By "tag" I further mean the AprilTag. The tags generations are called as tag families. The name of the family is made of number of bits encoded in the tag and the hamming distance. For example the last released family is Tag36h11 which encodes 36 bits with hamming distance of 11 between any two codewords (tags) .
The tags can be either downloaded from official AprilTag website or created with AprilTag library software on user's computer, in png format, which can be used in simulation software like Blender or code using OpenGL, or can be home-printed, it is preferable to stick printed tags to e.g. solid cardboard so they stay flat - planar. The software tool used for creating AprilTag png images is present in included software for this thesis.

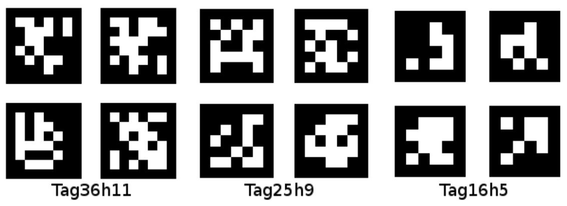

Figure 3: Example of 3 tag families. The different number of encoded bits in each family could be clearly seen.(Image courtesy of April Robotics Laboratory)

AprilTag detection algorithm As far as this work does not aim at further understanding how the image recognition works, I would still like to briefly describe how the AprilTag recognition algorithm works described in following 9 successive steps. It tries to introduce how algorithm works even to a reader with few or any knowledge from pattern recognition. For fully described algorithm read [6] for less complex description see [9] .

1. The pixel values of the image being processed are converted into floating point grayscale (pixel values from 0.0 to 1.0) and Gaussian blur is applied.
2. At every pixel is calculated local gradient – magnitude and direction.
3. Generate list of edges. First, neighbouring pixels with similar direction are grouped together. If the magnitude of the gradient of both pixels is significantly above zero, it is considered to be an edge.
4. From edges create clusters.
5. Loop over clusters, fitting lines called Segments.
6. For each Segment, find segments on its ends.
7. Search all connected segments. Find Quads = segment loops of length 4, representin the black border around a tag candidate.
8. Decode the quads. Inspect the pixels inside the border to see if they represent a valid tag code and from valid tags generate a list of TagDetections.
9. Search for overlapping TagDetections and take the best ones (lowest Hamming distance or largest perimeter); discard the rest.

These steps are described in [6] and [9].

I found important to briefly introduce this algorithm because even AprilTag still could be more or less used as a black box, the interface it provides – created by its objects and functions - includes for example the `TagDetector` class, representing the tags found in the image, with which I subsequently worke in the application.

Homography One of the parameters which `TagDetection` object contains is the homography. This parameter is computed during the recognition of the tag itself in the core of the software (TagDetector.cpp). For theoretical basis of planar projection represented by 3x3 matrix $\mathbf{H}$ see Planar Geometry . This fact is very useful for solving $\boldsymbol{x} = \boldsymbol{PX}$ - calibrating camera, where $\boldsymbol{x}$ is image point, $\boldsymbol{P}$ is camera and $\boldsymbol{X}$ is object point. $\boldsymbol{P} = \boldsymbol{K}[\boldsymbol{R}|\boldsymbol{t}]$ Whether approximating intrinsic parameters matrix $\mathbf{K}$ or extrinsic parameters $\mathbf{R}$ and $\mathbf{t}$(finding camera pose) it is often necessary to estimate $\mathbf{H}$ between object plane and image, which is often made by DLT, ML(Maximum Likelihood estimation) and other algorithms. AprilTag is allowing full 6 DOF localization of features from a single image.

3.2.2 AprilTag software

The other part of AprilTag library is the detection software itself. The library implementation in C language has no dependencies so there arises the need for image acquisition software. In my case – and based on [7] – OpenCV is used to acquire images from camera. The AprilTag Swatbotics C++ port library contains 10 source files, implementing the geometry and the tag detection of the system. One of the most important parts of this system is located in TagDetector.cpp (see included Software), implementing the tag detection algorithm from the image, about which talks previous section.

Various ports of AprilTag library I have found about three different versions of original AprilTag library which was created by prof. Edwin Olson. The one I have found most suitable for usage - as long as I decided to use C++ language in developing the application – was the AprilTag library C++ port by Swatbotics , created by author Matt Zucker [7]. In this implementation of AprilTag is used one dependency – OpenCV.

4 Implementing solution

4.1 Finding Camera Intrinsic Parameters

This part of the task solves the problem how to most accurately determine the intrinsic parameters of the camera. The calibration of the whole system while solving my task - in means of approximating both camera internal parameters and also its position towards the scene - could be divided into more successive steps. One of these steps is to determine the very parameters which every real camera has, thus finding the intrinsic camera matrix, see Camera and Single View chapter of this thesis. In the ideal camera the optical center would be right in the middle of the lens but in real cameras – and especially in low-cost cameras I have been using for this task – is not, rather it is slightly moved aside. The same goes for the focal length while focal length of x coordinate f_x slightly differs from f_y the y coordinate. E.g. from AprilTag Camera suite calibration application was the $f_x = 1107.77$ pixels and $f_y = 1113.75$ pixels. In ideal pinhole camera the $f_x = f_y$. This distortion could be caused by various reasons:

- Flaws in the digital camera sensor
- The camera's lens causes distortion
- Errors in camera calibration

As result the image pixels has non-square shape they are skewed.
[2]
In equations I have used is considered restricted camera matrix P where is assumed that pixels are square thus map from 3D to image is linear. [3] In some applications and simplified models I have been reading through is often used an approximation such: Optical center is approximated by the

center of the image, so for values of image frame (in pixels) 1280x720 would be the optical center $c_x = 640$ and $c_y = 360$, and the focal length by the width of the image so following previous example $f = 1280$ (in pixels) and radial distortion is not taken to count.
While this could be sufficient in most cases but I was trying to achieve maximal precision. In order to do so, I had to find out most plausible way to determine the camera intrinsic parameters. I have encountered more possible ways how to determine the intrinsic parameters. Such process of determining intrinsic parameters could be based on various camera models and various approximation and optimization algorithms, but the common characteristics for camera calibration is that often planar object - black and white pattern, such as chessboard, is used. There are some important reasons for using such thing as calibration tool.

- The black and white square pattern could be very good recognized during the image processing and can be thus more precise.
- The size of the pattern and its proportions are known, which is important for later geometry computation of e.g. point correspondences.
- Used for creating world-image point-point correspondences.

(either by DLT or with more complex algorithms) it is planar we could use homography .
I will denote in homogeneous coordinates image point $\boldsymbol{x} = (x, y, 1)^T$ and object point e.g. corner of chessboard/tag $\boldsymbol{X} = (X,\ Y,\ Z,\ 1)^T$.

$$[x\ y\ 1]^T = \boldsymbol{K}[\boldsymbol{r_1}\ \boldsymbol{r_2}\ \boldsymbol{r_3}\ \boldsymbol{t}][X\ Y\ Z\ 1]^T \tag{1}$$

Equation 1 represents projection of 3D point to image point. $\boldsymbol{r_1}\ \boldsymbol{r_2}\ \boldsymbol{r_3}$ are the columns of rotation matrix $\mathbf{R}$ and represent rotations along x y and z axes respectively, these columns are orthogonal. The vector $\boldsymbol{t}$ represents Further I will assume that tag is at $Z = 0$ in world coordinate system - coordinates for all points on tag are $Z = 0$, so $\boldsymbol{X} = (X,\ Y,\ 0,\ 1)^T$. So equation 1 is now

$$[x\ y\ 1]^T = \boldsymbol{K}[\boldsymbol{r_1}\ \boldsymbol{r_2}\ \boldsymbol{t}][X\ Y\ 1]^T \tag{2}$$

From this equation a constraints can be expressed and later used in analytical solution, which can be later optimized e.g. by Maximum Likelihood estimation. Full such algorithm can be found in [10].

It would be possible to implement such a camera system where intrinsic parameters would be calculated from one static tag detection. This would be however very not optimal because more correspondences are acquired the more precise is the result. In standard camera calibration process, such offers OpenCV with black and white chessboard, plays a great role how experienced is the person in such process, because to get good results is important know how to position a calibration object to create as many as possible different correspondences. After considering that finding most precise intrinsic camera parameters can be quite hard task, I have decided that best way is to determine these parameters before the main application starts.

April Camera Suite In order to minimize error caused by human factor during camera calibration, the April platform created a very useful tool for achieving so. April Camera Suite is a part of April Robotics Toolkit. It is interactive Java application using augmented reality - game-like - when user tries to position AprilTag, which is holding in front of the camera, to required position. The positions where user tries to move and rotate the tag are generated during the run of the application to ensure the best result. As [9] states this ensures the minimization of human error during calibration.

April Robotics Toolkit To use April Camera Suite for camera calibration, download the April Robotics Toolkit, because it is one of its parts. This toolkit is a Java library (now deprecated, but still very useful, with majority ported to C/C++) which was created in early development of ApriTag. I refer about this toolkit because it was quite demanding to install it on my computer. Before few years it would have been an easy task, but nowdays most of the dependencies are obsolete and are not located in any packages. It also uses Java 6 which is harder to download. First problem encountered was due to javac1.8 installed. After installing java7 with javac7 the build was successful. Before the build must be installed JOGL - OpenGL interface for Java.
April Robotic Toolkit Dependencies:

- Java 7
- OpenJDK 7
- Apache Ant 1.9
- JOGL

While it is very hard to find all dependencies for just one Linux distribution, I found it quite impossible to include this toolkit to included software for all the distributions and OSs. It is left up to user to decide how to obtain the intrinsic parameters. Using April Camera Suite or OpenCV calibration tools is recommended.

4.2 Aplication description

This application is using the [7] port of AprilTag library and is based on learning materials from OpenCV and Swatbotics port of AprilTag.

Brief description User places in sight of camera system the “calibration” AprilTag, of which middle represents the origin of world coordinate system. When application is started it firstly computes the approximation of world coordinate system given by the axes of the calibration AprilTag. While AprilTag detection is quite precise, it still has some variance of the detection values. In order to set a fixed origin of a world coordinates it is necessary to make appropriate approximation. While calculating average of measured values does not by far give the most satisfying result, this solution was implemented at first. By this step is determined a fixed world origin with some error. After this calibration step is completed the following tag detections are expressed in respect to this origin.

Run description When application is started two input configuration files are passed to application as command line arguments. These configuration files are then parsed by calling function `parseDeviceParametersFromXML(fs1)`; with input parameter `cv::FileStorage` object, which provides interface for handling xml files. The configuration parameters are then stored in `struct DeviceParameters`. This is done for both camera devices. Two instances of `std::thread` are started. Both threads execute the same function, which is processing images from the given camera device. First are set width and height of the image frame and objects representing matrices and other mathematical entities used in algorithm are created. For example the matrix of intrinsic parameters K as `cv::Mat` object and optical center as `cv::Point` object. The distortion coefficients are set to zero.
The calibration part follows, in means of determining the fixed world origin represented in scene by calibration tag. While loop is executed for given count, each loop represent processing of one single image represented by `cv::Mat` object called frame. First the frame is inspected for any tag detections and next is every detection further processed. The detections are processed by method `CameraUtil::homographyToPoseCV` from AprilTag library which computes the the tags pose in respect to camera. This method does following:

This method uses a OpenCV function `cv::solvePnP` which computes from 3D-2D point correspondences and camera's intrinsic matrix the rotation and translation vectors. The input for `cv::↩ solvePnP` are 3D and 2D points so the four points in general position are chosen – the corners of the tag – and are projected by inherently computed homography which every AprilTag detection contains. These are created by in `CameraUtil::homographyToPose` method by projecting 3D points representing tag corners ($[-1,-1,0],[1,-1,0],[1,1,0],[-1,1,0]$) and projecting them with detection homography to 2D corresponding points. By such action, the origin of world coordinate system is set to be the middle of the calibration tag. The projection can be represented as:

the vector $\boldsymbol{x'} = (x'_1, x'_2, x'_3)$ represents the homogeneous coordinates of point in the image and vector $\boldsymbol{x} = (x, y, 1)$ the coordinates of point in the world

$$x' = \frac{x_1'}{x_3'} = \frac{h_{11} * x_1 + h_{12} * + h_{13} * 1}{h_{31} * x + h_{32} * y + h_{33} * 1} \tag{8}$$

The result is in inhomogeneous coordinates

So output are two vectors $\boldsymbol{r}$, $\boldsymbol{t}$ which represent the rotation and translation of object -tag - in camera coordinate system. The Rotation matrix R is made by function Rodrigues and is inverted so camera origin in world coordinates could be computed such as $C = -R^T\ t$

After loop ends the extrinsic parameters are averaged and from this extrinsic matrix representing estimated world origin defined by calibration tag. After world origin is approximated the detection loop continues calculating positions of tag detections relative to chosen world origin.

note: If the duration of loops was higher than 1/25 which is minimal time interval which can human eye register as two different consecutive event, video stream of augmented scene would not be running smoothly I have calculated that duration of the loop is less then this number.

4.2.1 Build - CMake

CMake is open-source software tool used for software compilation process. It is platform and compiler independent and it uses quite simple format of configuration files with which you can generate native makefiles for Make (as this project was build on Ubuntu 16.04) . The configuration file for CMake must be named "`CMakeLists.txt`". In this file you specify how your project will be build. From choosing compiler and creating executables, creating and linking libraries and so on. Among many others, the OpenCV uses CMake and so does IDE Intelij CLion, which I was using during this project. I will describe the basic commands, so even user without any experience with CMake could understand the CMake configuration files. For complete documentation look at CMake documentation [11]. CMake has quite easy syntax, can use `if` statements, but some functions are not that intuitive. One of the CMakeFiles is presented.

```
add_library(AprilTags
  CameraUtil.cpp
  DebugImage.cpp
  Geometry.cpp
  GrayModel.cpp
  MathUtil.cpp
  Refine.cpp
  TagDetector.cpp
  TagFamily.cpp
  TagFamilies.cpp
  UnionFindSimple.cpp
)

set(AT_LIBS AprilTags ${OPENCV_LDFLAGS})

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -pthread")

add_executable(main main.cpp)
target_link_libraries(main ${AT_LIBS})

if (CAIRO_FOUND)
  add_executable(maketags maketags.cpp)
  target_link_libraries(maketags ${CAIRO_LIBRARIES} ${AT_LIBS} ${CAIRO_LIBS})
endif()
```

Figure 4: CMakeList.txt snippet

- `add_library([library_name] source_files(file1.cpp,..))` creates static library with chosen name and from given source files.
- `set(AT_LIBS AprilTags ${OPENCV_LDFLAGS})` now the varaible AT_LIBS refers to library AprilTags plus what was in variable `${OPENCV_LDFLAGS}`, which is variable defined in OpenCV cmake – internal OpenCV CMake variable. In the OpenCV CMake configuration file are

various variables used such as $OpenCV_INCLUDE_DIRS - the OpenCV include directories, which are than used in other CMake files

- `add_executable(main main.cpp)` declares that after cmake creates makefile and after you make, there will be executable named “main” build from source file named “main.cpp“
- `target_link_libraries(main ${AT_LIBS})` During the linking, target named “main” will be linked with libraries specified in CMake variable ${AT_LIBS}
- `set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -pthread")`
  CMake variable CMAKE_CXX_FLAGS containes flags for cpp compiler, you can pass parameters as shown, telling compiler to compile as c++11 standard

4.2.2 Input - Configuration file

I had also implemented a in the application input of various parameters from command line by using *getopt_long* function, but as the number of parameters grew and two cameras were added it became unmaintainable. Henceforth I decided using input from file. Finally only parameters passed by command line are paths to configuration file.

Configuration file "dev#.xml" Firstly I was considering how camera parameters could be passed to application. Passing through command line is very unsuitable, while there could be multiple cameras and tags. I decided that parameters will by passed by simply structured XML files, one for each camera. I could have either used lightweight .XML parser library or wrote my own one.The OpenCV framework offers '`cv::FileStorage`' class which handles input and output of XML and YAML files. Accessing parameters is then very straightforward. So as was said, the application needs two arguments - so far as we are using two camera.- paths to XML configuration files where user define camera parameters. For each camera there must be one XML file. These files must contain following parameters.

Code 1: Snippet of configuration file "dev.xml"

```xml

<?xml version="1.0"?>
<opencv_storage>
<dev_n>0</dev_n>
<tag_calib_id>0</tag_calib_id>
<tag_calib_size>0.128</tag_calib_size>
<tag_1_id>1</tag_1_id>
<tag_1_size>0.157</tag_1_size>
<height>720</height>
<width>1280</width>
<fx>1107.773799</fx>
<fy>1113.757253</fy>
<cx>639.402972</cx>
<cy>376.813597</cy>
<tag_family>Tag36h11</tag_family>
</opencv_storage>
```

1. **dev_n**, device number - that is number under your usb device is mounted, since usb-lowcost cameras were used. This could be find out by command (Ubuntu) `−ltr /dev/video*` .
2. **tag_#_id**, tag own unique id number in its AprilTag family
3. **tag_#_size**, length of tag side in metres
   # : **calib** for tag used for user defined origin and **1** for tracked target.
4. **height, width**, camera image frame in pixels eg. 1280x720
5. **fx, fy,cx, cy**, camera intrinsic parameters all in pixels: fx – focal length by x-axis , fy – focal length by y-axis, cx – x coordinate of optical center , cy – y coordinate of optical center.
6. **tag_family**, see AprilTag description

4.2.3 Encountered problems during the application implementation

Thread problems When I was creating the thread part of my application I have encounter strange behavior of threads not running as they should be. In each thread `cv::Mat frame` object representing the image that was get by camera by `cv::VideoCapture` object was processed. After the image was processed thread was put to sleep so the second thread could do the same. However each thread also displayed the processed frame by

```
cv::imshow(frame_name, frame);
cv::waitKey(milliseconds).
```

I have found that `cv::waitKey()` caused the wrong running of threads, while it waits for given milliseconds. In the end I found as the only solution to use this method instead of `std::this_thread_::sleep_for(std::chrono::milliseconds(s));`.

4.3 Building the model

Using simulations software for camera viewing scene is very useful, while it is precise in measurement object distances from virtual cameras, and could be much quicker than prototyping from scratch new real-world model suitable for solving the task. To make it possible to go from software simulations of cameras viewing scene just on my computer to building the real-world model of camera system for this task, there was need for a middle software between camera drivers and my application. Besides many other functions OpenCV provides image acquisition from camera devices. For building the camera acquisition system for this project I decided to use low-cost web-cameras. One reason was low – or rather no budget, the second reason was that many literature about calibrating cameras and robust visual pattern recognition states, that developed algorithms are suitable for the low-cost camera, performing well and precise, so I have decided to put it to a test. While I had no superior-quality camera I cannot make comparison from measured values on both low-cost and good-quality cameras, but the results could still be valuable.

I tried to build as sufficient model for cameras as I could in home conditions. In the beginning of building this camera model I was using just one camera. Firstly it was the built-in webcam in my laptop. This was quite useful in the beginning for the testing if image acquisition software was correctly installed and for basic experimentation with augmented reality – such one OpenCV offers - and tag detection software. However later when I wanted to build a scene to be observed by cameras in order to experiment with tag detection software it was necessary to start using an external camera. Firstly I had at hand an old web-camera, which wasn't serving well, so it was replaced by two – relatively new – web-cameras. In order to observe the scene from above and get necessary perspective I acquired a 5m USB cables , so I would be able to position the cameras in the distance from my laptop. While It would be more convenient to transmit data from cameras differently for example using a network connection and client-server model for gathering the image data from the cameras, this was the most cheapest way to gather some useful information. For creating a model with greater distances between the cameras using USB cables could be a problem while USB standard has defined maximal cable length.

Operating System Difficulties I was using Linux operating system – namely distribution Ubuntu 16.04 - for working on this task. I have undergone few difficulties I had to solve. While most of new webcams have drivers and support for various Linux distribution, even though when searching for official camera documentation , Linux is rarely found in supported OS, I had encountered a serious problems to run older web-cameras on my Ubuntu 16.04 because drivers became in 5,6 years obsolete and are no longer parts of packages. Before I have obtained two new cameras I was using 7 years old "Logitech QC 3000 for Business" , which was causing constant troubles. So advice for Linux users trying to run older web-cameras on their system, I would consider it not worth the trouble.

Possible improvements Using such averiging as is now used is very non-optimal. For getting better results the triangulation should be implemented , I have worked on such improvement but I didn't make to finish it. I have also made research on Kalman filters and other filters (such as median filter etc.) which could rasantly help with obtaining the images from camera while they can contain quite amount of noice etc.

experiments The [6] also states that AprilTag detection software is robust to lightning conditions. While AprilTag is considered as one of the best tag detection systems I have encountered some problems with tag recognition with various light conditions. This could be also due to usage of low-cost cameras. For example during experiments of detection of the tag with my built model with low-cost cameras in the room, when tag was placed on border of sunlight and shade, the recognition became poor or even sometimes not possible. I found out that tag detection by ApriTag with extreme light conditions were not beneficial either.

precision I have been measuring the distances in real world with distance laser measuring device with precision on unints of millimeters. While it is possible with such device with reasonable presision determine even the angles, due to lack of time and ill conditions the measurements were solely informational. The measured distances given by the application were from fraction of centimeters up to 10 centimeters, depending on distance from cameras, light conditions and angles of cameras. As I stated the angles of cameras during measurements were not computed with enough precision so I can only make assumptions of real precision.

5 Results

I consider the first task to be achieved quite well, while I was coming to this topic with very little knowledge from this fsubject. I made the research and learned the basics from computer vision and camera calibration. The first goal was maybe even harder than I have expected taking me quite considerable amount of little time I had. I also found that second goal which was to search for suitable software was quite successful. I have installed the suitable software and learned how to work with it. I also had to understand the code of AprilTag ports to be able to work with it and further improve it for my needs. Besides I got familiar with build software CMake and OpenCV platform which was very usefull. This task took also long time as it consisted of reading lot of documentation and - at the time – non familiar code. It was also sometime precarious to install and run necessary software tools, either they were not that user friendly or deprecated. Third goal to figure out possible solutions and designed the model was in my opinion quite successful too. I managed to build even in home conditions low-cost but quite worthy camera system and experiment with it in real time with installed software and printed tags. Even though the application gives quite reasonable result and implements usage of multiple cameras the finish wasn't that successful as I run out of time and cannot finish the triangulation and further improvements. Also more measurements should be done, for example on software measuremnt tools because as I found building real world model is quite demanding.

References

[1] M. Sonka, V. Hlavac, and R. Boyle, *Image processing, analysis, and machine vision.* Thomson, 2008.

[2] K. Simek, "Reading about topics on camera parameters." `http://ksimek.github.io/`.

[3] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision.* Cambridge University Press, ISBN: 0521540518, second ed., 2004.

[4] "Wolfram mathematical references." `http://mathworld.wolfram.com/RodriguesRotationFormula.html`.

[5] "Opencv official documention." `http://docs.opencv.org/3.2.0`.

[6] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3400–3407, IEEE, May 2011.

[7] M. Zucker, "Swatbotics apriltag library c++ port." `https://github.com/swatbotics/apriltags-cpp`.

[8] O. developers, "Opencv 3.2.0 official tutorials." `http://docs.opencv.org/trunk/d9/df8/tutorial_root.html`.

[9] A. Richardson, J. Strom, and E. Olson, "AprilCal: Assisted and repeatable camera calibration," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, November 2013.

[10] Z. Zhang, "A flexible new technique for camera calibration," December 2000.

[11] "Cmake documentation." `https://cmake.org/cmake/help/v3.9/`.