UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Vitalii Zakharov

# Framework for extracting and solving game puzzles (Rubik's cube like)

Master's Thesis (30 ECTS)

Supervisor: Artjom Lind, PhD

Tartu 2017

# Contents

# Abstract

This thesis describes and investigates how computer vision algorithms, moreover stereo vision algorithms, may be applied to the problem of objects detection extrapolated on game puzzles detection, extraction and solving. The study examines different approaches from non related approaches applied to the given problem, as currently there is a little written on this subject. This is achieved through empirical research represented as a set of experiments in order to ensure whether approaches are suitable. To accomplish these goals huge amount of computer vision theory has been analyzed. The topic was chosen as the relevance of game puzzles is always the question because some new similar but different puzzles appear all the time and the relevance of real-time processing. Different real-time Structure from Motion algorithms (SLAM, PTAM) were successfully applied for navigation or augmented reality problems but none of them for objects tracking. This thesis examines how these different approaches can be applied for the given problem to help uninformed users be on easy touch with logical games. Moreover, it produces a side effect which is possibility to track objects movement (rotation, translation) that can be used for manipulating a rendered game puzzle and increase interactivity and user engagement.

# Acknowledgements

I would like to express my special thanks of gratitude to my supervisor Artjom Lind as well as the head of the Distributed Systems research group professor Eero Vainikko who have guided me for this two long years through the complex world of computer vision field of knowledge. This involved me in multiple researches on which basis the current project was created and I am really thankful to them.

# Abbreviations and Acronyms

**IoT** Internet of Things

**SfM** Structure from Motion

**OCR** Object Character Recognition

**PTAM** Parallel Tracking and Mapping

**SLAM** Simultaneous Localization and Mapping

**DTAM** Dense Tracking and Mapping in real-time

**API** Application Programming Interface

**ORB** Oriented FAST and Rotated BRIEF

**SURF** Speeded Up Robust Features

**SIFT** Scale-Invariant Feature Transform

**FAST** Features from Accelerated Segment Test

**BRIEF** Binary Robust Independent Elementary Features

**PnP** Perspective-n-Point

**kNN** k-Nearest Neighbors

**SVM** Support Vector Machine

**SVD** Singular Value Decomposition

**HSV** Hue, saturation and value

**RGB** Red, green and blue

**HPF** High-Pass Filters

**LPF** Low-Pass Filters

**FLANN** Fast Library for Approximate Nearest Neighbors

**CCA** Conected Components Analysis

**POI** Points of Interest

**LoG** Laplacian of Gaussian

**DoG** Difference of Gaussian

**Linq** Language-Integrated Query

**GPU** Graphics processing unit

**CPU** Central processing unit

**JSON** JavaScript Object Notation

**OpenCV** Open Computer Vision

**EmguCV** Emgu Computer Vision

**OpenGL** Open Graphics Library

**OpenTK** Open Toolkit library

**FPS** Frames Per Second

# 1 Introduction

Nowadays computer vision is a widely used technology. It has started developing since the late 1960s but a real leap into the future has been done in 1990s, when a lot of research topics got a second chance, for instance, projective 3D reconstruction, camera calibration, multi-view stereo techniques and etc. On the other side, hardware world was rapidly evolving which resulted to the current situation in computer vision field of research. Today the level of development (Internet of Things (IoT) devices, smart phones, tablets and etc.) allow to integrate computer vision tightly into humans life. There can be enormous amount of examples like parking systems when there is a camera mounted on a barrier at an entrance of a parking and a car number is recognized with this camera to decide whether the barrier will be opened or not; cameras mounted on the roads for catching speeding offenders; OCR technologies for analyzing text on images which allows to take a picture of a document and receive a typed version of it without typing by yourself; applications for finding pedestrians on a road; applications for face recognition, they help to find criminals in the streets, airports or any public place where cameras are mounted; this list can be almost infinite.

Human need for these technologies motivates researches to develop new techniques, improve algorithms that already exist and that what we have for the past decade. This decade changed the world, there have been created dozens of new concepts like social media, video platforms, 3d printing, cameras in mobile phones and etc. All of these have made a great influence on a human life. Peoples appetite is growing significantly and that is why new entertainment systems appear. Microsoft created Kinect, Sony has Move, Nintendo has Remote Plus and such kind of tools are based on computer vision but the technologies beneath them are not ideal, they need to gather a lot of data with a good quality, proceed complex computations and the result is not always as good as expected. All of that leads to the creation of new algorithms and attempts to get as much as possible of existing hardware.

## 1.1 Problem statement

Tools mentioned above allow to capture objects movements, recognize patterns and etc. With their help it is possible to analyze density of an image, create a 3d reconstruction, control game objects with our hands moving, shaking and many other interactive activity became real. But what if we look closer at Kinect and Move; it is clearly seen that they are just an expensive set of sensors for computer vision but in addition to multiple cameras they have infrared sensors, color sensors, depth sensors, microphones and so on; all these sets of sensors allow them to gather a huge amount of data. From this perspective, even the simplest algorithms may

provide quite a good result but what if we do not have such an expensive hardware; that is the case where algorithmic part takes the leading role.

There are dozens of different algorithms already invented, some of them are applicable to the nowadays situation and some of them need an improvement. Algorithms could have concrete purpose at the beginning but with a time they might be applied to other field and become a real breakthrough there. This thesis will focus on experiments with the algorithms that allow to get the same result as with Kinect or another expensive hardware while having a simple monocular camera. These experiments are needed for understanding whether it is possible to build an approach for successful tracking of objects and mapping them to their 3d representations on a computer. All existing algorithms have different purposes and can't be easily applied to the mentioned problem out of the box.

Towards this end, main research goals of this thesis are:

1. Analyze every existing algorithm that can be applied to this problem.
2. Take the parts of the theory beneath those algorithms and combine them together.
3. Design a new approach for a mentioned problem based on the main seeds of the already existing algorithms.

## 1.2 Contributions

Methodology of this thesis is so called empirical research and it targets to achieve next goals:

- Examine feature detection methods in computer vision.
- Examine feature classification methods in computer vision.
- Examine theory of stereovision (camera calibration, epipolar geometry, triangulation, pose estimation) in computer vision.
- Analyze theory beneath "Structure from Motion" approach.
- Analyze theory beneath "Tracking and Mapping" approaches.
- Build a framework for tracking and recognizing objects using monocular camera. Since Rubik's cube was picked as a simple shape object, using captured data it will be reconstructed and solved with an already implemented module of the framework.

## 1.3 Road map

The rest of the master thesis has the following chapters:

Chapter 2: Presents an overview of all related theory in computer vision: basic theory as feature detection/classification, basics of stereo vision as a camera calibration, "Structure from Motion" concept overview, "Tracking and Mapping" concept and its derivatives overview, Kalman filter explanations.

Chapter 3: Contains overview of all experiments with algorithms and their implementations from the Chapter 2. Includes descriptions of test cases, used approaches, used implementations or self written ones. Covers testing and validating results.

Chapter 4: Covers a solution to the mentioned problem based on the results of experiments. Self written PTAM based approach covered and reasoned. Results are validated, pros and cons are presented.

Chapter 5: Overviews the results of the thesis and possible future work and perspectives of designed approach.

# 2 Background and Related work

This chapter describes theoretical research that was done in order to gain enough knowledge for further applying it to the stated problem.

## 2.1 Computer vision

This subsection covers theoretical material related to the computer vision field of knowledge.

### 2.1.1 Theoretical

The following text is the research of computer vision algorithms theoretical low-down. This material covers theory from the simple manipulation with digital images, for instance, blurring, to the complex theory of stereo vision and Structure from Motion problem.

#### 2.1.1.1 Computer vision basics

**Digital images** are the first step in any computer vision process. They represent an electronic snapshots of something that we see, for instance, captured scene by photo camera or some scanned document and etc. It got used to sample all digital images as a set of pixels (picture elements or grid of dots, in other words). Any pixel is represented as a binary value (set of zeros and ones) so these values illustrates tonality of pixels (white, black, etc.). The binary digits or bits are interpreted and read by a computer to produce an analog version of the image in order to display or print it.

Images have a property called "Bit Depth", its value is constituted by number of bits used to represent a pixel. Image tonality is dependant on bits depth so the larger its value the larger variety of possible tones for the pixel. Digital images could be bi-tonal(black and white), gray-scale, or color. For example, pixels in bi-tonal images are depicted as one bit (zero or one, black and white); gray-scale images, on the contrary, can be made of multiple bits, usually from two to eight but can be more. Color digital images are another side of coin, they are typically represented as a set of bits in range from 8 to 24 or more. Taking the 24-bit images into the view, their values are usually divided into 3 group, first 8 bits for red color, next 8 for green, and last 8 for blue. Here is a list that shows how possible variety of tones is dependant on number of bits per pixel:

- 1 bit, $(2^1) = 2$ tones
- 2 bits, $(2^2) = 4$ tones

- 3 bits, $(2^3) = 8$ tones
- 4 bits, $(2^4) = 16$ tones
- 8 bits, $(2^8) = 256$ tones
- 16 bits, $(2^{16}) = 65{,}536$ tones
- 24 bits, $(2^{24}) = 16.7$ million tones

**Computer vision pipeline** can be divided into 3 main phases: low-level processing where image results into another image (blurring, sharpening, thresholding), mid-level processing where image results into set of features (Features from Accelerated Segment Test (FAST), Speeded Up Robust Features (SURF) feature detection algorithms), high-level processing where features are analyzed and some data is received (Conected Components Analysis (CCA)).

**Thresholding**. Nowadays computer vision is gray-scale and cannot work with colors. In order to remove noise and make gray-scale image more precise for computer vision it should be thresholded. This operation is one of the simplest segmentation operations. There are also some algorithms that can threshold color images but they still produce gray-scale images as a result. There are multiple thresholding algorithms and they produces very different result due to complexity of computations. There are two the most simplest thresholding algorithms: global and adaptive thresholding. The first one is very primitive, it is provided with a threshold value and compares each pixel intensity whether its larger or lower this value. The intensity of a pixel is calculated as $((R + G + B)/3)$. Based on the comparison result it replaces pixel intensity with either 0 or maximal intensity. There are multiple variants of this operation, it can be binary, inverted binary, truncated, truncated to zero, threshold to zero and inverted.

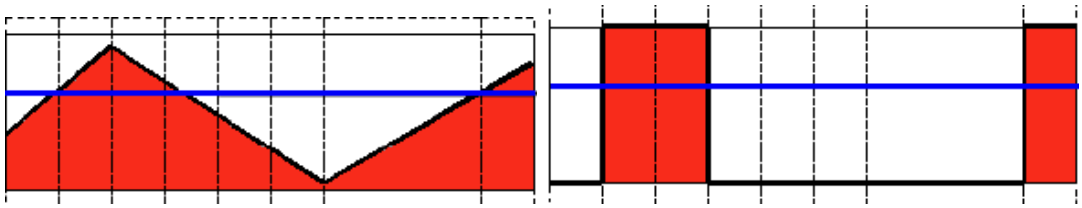

Figure 1: Global Threshold example

Truncated threshold, on the contrary to binary, replaces only values that are larger or lower the trash, depending on whether operation is inverted or not, not both at the same time.

$$\text{newIntensity} = \begin{cases} maxValue, & \text{if } currentIntensity > tresh \\ 0, & \text{if } currentIntensity < tresh \end{cases}$$

Figure 2: Formula of Global Threshold

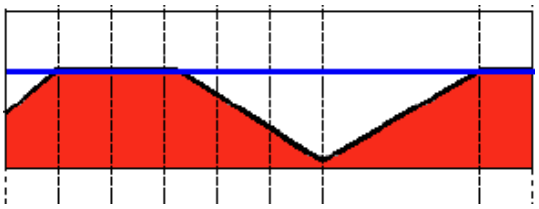

Figure 3: Global Truncated Threshold example

Threshold to zero is quite similar to the simple threshold but instead of putting maximal value it puts the real one.

Adaptive thresholding, from the other side, does not use fixed threshold value, it computes this value from surrounding pixels, for instance, it takes an eleven by eleven square where the needed pixel is in the center and computes threshold value from this block of pixels. This algorithm is much slower but it produces much better results and it is more appropriate for real world tasks, since it works better for images with varying illumination.

**Otsu's Binarization** uses different approach to calculate threshold value, it works with bi-modal images (histogram has 2 peaks) so the threshold value is taken approximately in the middle of these two peaks. The algorithm is accurate only for bi-modal images, it's better not to use it for others, results will be unsatisfying.

**Image Filtering**. In computer vision filtering is used to clear images from noise. There are multiple common types of it: "Salt and pepper" noise that contains multiple randomly located black and white pixels, "Impulse noise" that contains randomly located white pixels, and Gaussian noise where noise is Gaussian-distributed. One of the most important entities in the image filtering is convolutional matrix, also known as kernel or mask, which is a small matrix used for image filtering. It is related to a form of mathematical convolution. In addition, it should be mentioned that convolution is not traditional matrix multiplication, even though it is denoted by * (asterisk). It is the process of adding each element of the image to its local neighbors, weighted by the kernel.

There are many filters for various purposes developed, some of them are Low-Pass Filters (LPF) that are used for removing noise or blurring and High-Pass Filters (HPF) used to find edges, etc.

In order to blur an image it should be convolved with a low-pass filter mask.

$$\text{newIntensity} = \begin{cases} thresh, & \text{if } currentIntensity > tresh \\ currentIntensity, & \text{if } currentIntensity < tresh \end{cases}$$

Figure 4: Formula of Global Truncated Threshold

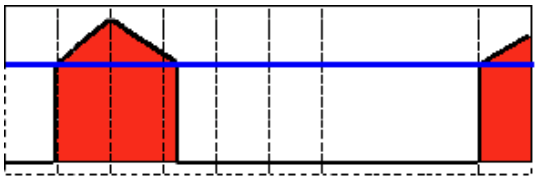

Figure 5: Global Threshold to Zero example

$$\text{newIntensity} = \begin{cases} currentIntensity, & \text{if } currentIntensity > tresh \\ 0, & \text{if } currentIntensity < tresh \end{cases}$$

Figure 6: Formula of Global Threshold to Zero

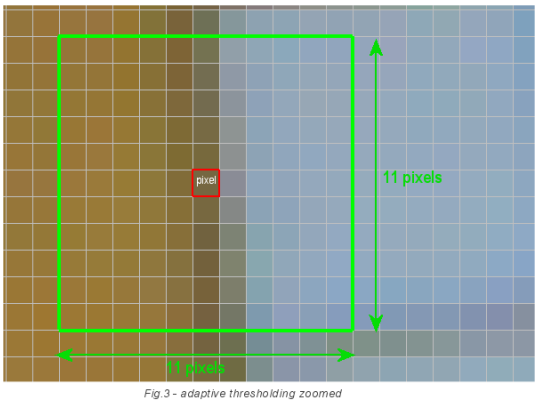


Figure 7: Adaptive thresholding logic demonstration

Basically, this operation removes high frequency content, such as noise or edges, for instance, it may result in edges being blurred when using this filter but not all blurring techniques do blur edges.

In image processing, a Gaussian blur is blurring of an image with a Gaussian function. This effect is often used in computer graphics software and as initial processing step in computer vision algorithms for enhancing image structures at various scale space representations and implementations. Basically, applying this filter to an image is the same as if the image is convolved with a Gaussian function

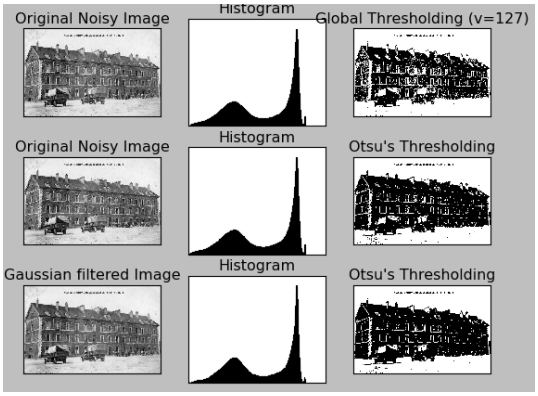


Figure 8: Otsu's Binarization demonstration

$$\left( \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right) \left[ \ 2, 2 \ \right] = (i * 1) + (h * 2) + (g * 3) + (f * 4) +$$

$$\text{(e*5)+ (d*6) + (c*7) + (b*8) + (a*9) (1)}$$

Figure 9: Example of convolution

and produces the image with reduced high frequency components effect. Mentioned action is known as a two dimensional Weierstrass transformation.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{2}$$

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

Figure 10: Gaussian functions for one and two dimensions

In the Gaussian function $x$ and $y$ represent the distances from the origin in the horizontal and vertical axis while $\sigma$ represents the standard deviation of the Gaussian distribution. In order to build a convolution matrix two dimensional function is used to get concentric circles with a Gaussian distribution. The values of this distribution from the center points are used in convolutional matrix.

Median Filtering is quite simple, it computes the median of all the pixels from the current pixel neighborhood, so called window, and the central pixel is replaced with this median value. The convolutional matrix size for this technique must be a positive odd integer. This type filtering is super effective for removing such kind of noise as so called "salt and pepper".

Average filtering or Mean filter is very similar to Median but the value for replacement is computed as the average value for the pixel neighborhood.

Bilateral filtering is significantly different from the others, this filtering technique removes noise without blurring edges, it preserves them. It makes much dipper analysis then Gaussian filter, checks pixel intensity or if some of them lie on the same edge and etc. This filter has a Gaussian filter under the hood for space domain and also uses a function of pixel intensity differences, that is also a multiplicative Gaussian filter component. This analysis helps to keep edges unblurred while noise is removed but operation is taking some time so its much slower than other competitors.

$$I^{filtered}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(||I(x_i) - I(x)||) g_s(||x_i - x||) \tag{4}$$

$$W_p = \sum_{x_i \in \Omega} f_r(||I(x_i) - I(x)||) g_s(||x_i - x||) \tag{5}$$

Figure 11: The bilateral filter definition and its normalization term

$I^{filtered}$ from the formula 4 is the resulting image; $I$ is the initial image, $x$ is the coordinate of the current pixel; $\Omega$ is the neighborhood, so called window, with $x$ in its center; $f_r$ and $g_s$ are two convolutional matrices for smoothing and they could be Gaussian functions.

**Morphological transformations** are widely used and simple operations based on the image shape, they are techniques for analysis and processing of geometrical structures. Initially they were designed for binary digital images but nowadays they can also be applied to gray-scale image and different spatial structures as graphs, solids, or meshes. The most common morphological operations are erosion, dilation, opening and closing.

A structuring element in morphological transformations is a shape mask. It can be of any shape or size that can be represented digitally.

Erosion operation moves the boundaries of foreground object away. The main idea of erosion transformation is similar soil erosion, basically, the window moves through the image, action is similar to 2-D convolution and check whether all items in the window for the current pixels are the same value as pixel value itself. If

the pixel is 1 and all kernel items are ones then its value is kept, otherwise eroded to zero. Definitely, this is related to binary images erosion that is widely used in computer vision. This morphological operation is useful for removing small white noise objects or for detaching two connected objects and etc.

$$A \ominus B = \bigcap_{b \in B} A_b \tag{6}$$

$$(f \ominus b)(x) = \operatorname*{infimum}_{y \in E}[f(y) - b(y - x)] \tag{7}$$

Figure 12: Definitions of erosion for binary and gray-scale images

Dilation, on the contrary, keeps pixel value "1" in case if at least one of the elements in the neighborhood is "1". This operation enlarges the white space in the image and usually used straight after erosion. The idea under this action is that erosion has removed noise and increasing white area won't bring it back while it may connect broken parts of objects and restore shrinked objects.

$$A \oplus B = \bigcup_{b \in B} A_b \tag{8}$$

$$(f \oplus b)(x) = \operatorname*{supermum}_{y \in E}[f(y) + b(y - x)] \tag{9}$$

Figure 13: Definitions of erosion for binary and gray-scale images

Opening is the name for two operations, erosion followed by dilation. As was mentioned above, it is useful for removing noise.

$$A \circ B = (A \ominus B) \oplus B \tag{10}$$

Figure 14: Definition of opening

Closing is the name of reversed Opening where dilation is followed by erosion. Most commonly used in computer vision for closing little holes in the foreground objects and etc.

Morphological gradient in computer vision is simply a difference between dilation and erosion of an input image. This operation is useful for edge detection and segmentation purposes because it results into image where each pixel value represents the contrast intensity among neighboring pixels.

$$A \bullet B = (A \oplus B) \ominus B \tag{11}$$

Figure 15: Definition of closing

$$G(f) = f \oplus b - f \ominus b \tag{12}$$

Figure 16: Definition of morphological gradient where f is a gray-scale image and b is structuring element.

**Feature detection and description**. In image processing there is no way to define image features as something exact and there is no universal algorithm to get them from an image. Usually the definition of features goes out from the problem that is trying to be solved but in general they can be characterized as Points of Interest (POI) for a given problem. Feature detection is an operation of finding and analyzing image for some interesting points; it often results in the form of points, lines, curves, or regions lists. Detection and analyzing operations could be really time consuming or require large computational power thus feature detection algorithms could become a part of more high level algorithms so that they are called only on some localized regions to improve speed and time complexity.

Some popular types of features have become very popular and are widely used, they are edges, corners (points of intersection), and blobs (regions of interesting points). Edges, in general, are defined as points on the edge (boundary) in between image blobs. Corners are referred as point-like features, basically, the name is just used by tradition. In fact those features are not corners at every scenario. They were developed from the edge detection when people tried to analyze rapid changes in direction, which is a corner. Blob features are good when images are too smooth so that corners can't be detected. These features contain additional information about image structures in terms of regions, different from what point like features do.

Feature extraction, on the other side, is the operation applied to the image with the information retrieved from the detection step, it proceed the extraction of image region around the feature. This action gives a feature descriptor as a result but may consume a large amount of computational power.

Nowadays, many different types of features have been defined, thus many different feature detection and extraction algorithms have been developed.

Canny edge detection is one of the most popular edge detection algorithms. It is named by its author, John Canny. This algorithm has multiple stages:

noise reduction, computing intensity gradient, non-maximum suppression, hysteresis thresholding. It uses 5 by 5 Gaussian filter to reduce noise because edge detection operation is very vulnerable to a noise. Image is filtered with Sobel matrix to get horizontal and vertical direction first derivatives which is done to find edge gradient and direction. After this, image is filtered to reduce pixels that might not be components of the edge. Filtering is done by checking local maximum of the pixel in the direction of gradient through the neighboring pixels. In short, if the result of this check gives a local maximum, pixel is kept for the next stage, otherwise put to zero. Binary image is given in the end of this stage. Last step is analysis of detected edges, they are filtered (thresholded) on the basis of two thresholding values, minimum and maximum. If the pixel intensity gradient is larger then maximum, it is marked as edge, if lower then minimum then discarded. The most interesting case is when the value lies in between, in this case edges are checked for connectivity with those who is already marked as real edge, if the connection exists they are also marked as edges, otherwise discarded.

$$gradient(G) = \sqrt{G_x^2 + G_y^2} \tag{13}$$

$$angle(\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{14}$$

Figure 17: Definition of gradient and direction where $G_x$ and $G_y$ are first derivatives in horizontal and vertical directions.

Harris corner detection algorithm is developed by Chris Harris and Mike Stephens. They were one of the earliest people who were trying to find corners, regions in the image where intensity value in all directions varies in the large range. The idea was to get the difference in intensity in all directions. For corner detection $E(u, v)$ function is maximized by applying Taylor Expansion. This was done to create a score function that will determine whether the corner is in the window or not. Eigenvalues $\lambda_1$ and $\lambda_2$ are retrieved from the matrix created in the process of maximizing $E(u, v)$ and they decide what region is corner. The region is determined to be an edge when its score is less then zero, that happens when $\lambda_1 >> \lambda_2$ or $\lambda_1 << \lambda_2$, which means that the difference between them should be significant. On the contrary, if the score is large, which happens when lambdas are large and approximately equal, region is determined as a corner.

From the other side, Jianbo Shi and Carlo Thomasi in 1994 wrote a paper called "Good features to track", where they suggested to take another scoring function which gave much better results in comparison with Harris Corner Detection. The idea was to compare results of scoring function to a threshold value. In their

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2 \tag{15}$$

Figure 18: Definition of intensity difference where $w(x,y)$ is a window function, $I(x+u, y+v)$ is a shifted intensity and $I(x,y)$ is the current intensity.

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{16}$$

Figure 19: Definition of the Harris scoring function.

version, the determined region as a corner only if both $\lambda_1$ and $\lambda_2$ are larger then the score, which is minimum from both lambda values.

$$R = min(\lambda_1, \lambda_2) \tag{17}$$

Figure 20: Definition of the Shi-Tomasi scoring function.

Scale-Invariant Feature Transform (SIFT) is an algorithm authored by David Lowe in 2004 and described in the article "Distinctive Image Features from Scale-Invariant Keypoint". The author has created it due to the scale problems for corner detection in already created algorithms, while corner remains a corner after rotation, scaling breaks the logic. Algorithm contains 5 stages: scale-space extrema detection, key point localization, orientation assignment, key point descriptor, key point matching. First of all, it uses different windows to detect key points with different scale and it uses scale-space filtering for this. Laplacian of Gaussian (LoG) is used as a blob detector with a scale parameter $\sigma$. Then it searches for the local maxima through scale-space and results into a list of tuples $(x, y, \sigma)$ that can be a potential key points. Since the LoG is a bit costly operation SIFT uses difference of Gaussian because it is less computationally expensive and, in fact, is LoG's approximation. It is done by getting difference of blurring image with two different values ($\sigma$). After Difference of Gaussian (DoG) is complete, scale-space is searched for a local extrema and if it is found, that is a potential key point. Paper gives some optimal values retrieved by empirical research, $\sigma$ value is 1.6, and another $\sigma$ is $1.6\sqrt{2}$. Potential key points are found but they are not accurate enough.To get more precise location of extrema algorithm uses Taylor series expansion of scale-space and compares intensity of current extrema with threshold value so that if it

is less, key point is rejected. It also uses similar to Harris Corner detection edge filtering since DoG is not good at that and discard those key points whose ration is greater then a threshold. Next, in order to be invariant to image rotation, each key point is assigned with orientation. To achieve this orientation histogram covering 360 degrees is created. The next step is creation of key point descriptor from the 16 by 16 neighborhood of the current key point. This region is divided into 16 sub-locks where for each an orientation histogram is created. It also performs some operations to be resistant to illumination changes, rotation changes and etc. Total of 128 bin values is represented as a vector to represent a key point descriptor. The last step is key points matching, that is done by identifying the nearest neighbour. Due to some reasons like noise or others the second nearest match may be very close to the first one and in this case ratio between them is taken and depending on its value they might be rejected or not.

In 2006 3 people, Herbert Bay, Tinne Tuytelaars, Luc Van Gool created a replacement for SIFT called SURF. Basically, this algorithm brings speed improvement at each step of SIFT. For the fist step, it goes much deeper and approximates LoG with box filter, which is a convolution filter. This type of operation has some advantage: easily calculated with a help of integral images, can be performed in parallel for different scales. During orientation assignment it uses wavelet responses in both directions. It also estimates the dominant orientation as a sum of all responses in the orientation window with angle of $60^\circ$. Since many tasks do not require rotation invarience, SURF has a concept called Upright-SURF or U-SURF where there is no orientation findings that speeds up the process by 15%. On the feature description step it uses wavelet responses again in both directions. The neighborhood of the key point is divided into 4 by 4 sub-regions and for each of them it forms a vector from wavelet responses.

$$v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|) \tag{18}$$

Figure 21: Definition of a vector formed from wavelet responses.

This results into the feature descriptor with 64 dimensions but SURF also has an extended 128 dimension version where $\sum d_x$ and $\sum |d_x|$ are computed separately for $d_y < 0$ and $d_y \geq 0$, the same is done for $\sum d_y$ and $\sum |d_y|$ depending on $d_x$ sign.

SURF also has a great improvement which is a use of a sign of Laplacian. It is already computed during detection, a trace of Hessian Matrix, thus it adds no cost to be computed. It allows to distinguish bright blobs on dark background and the opposite situation. During matching, it uses minimal information, compares features by having the same type of contrast, which allows to do matching faster

without reducing performance.

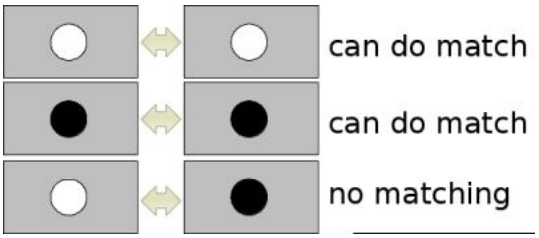


Figure 22: Surf matching example

All mentioned features detection algorithms are not fast enough to be used in real time tasks, one of examples is Simultaneous Localization and Mapping (SLAM). In 2006 Tom Drummond and Edward Rosten have written an article "Machine learning for high-speed corner detection" where they proposed new algorithm called FAST.

The idea is pretty simple, it takes a circle of 16 pixels around the current pixel and if at least $n$ of them are contiguous and their intensities are larger then the sum of current pixel intensity and threshold value or less then difference of them it implies that the pixel is corner. To improve the speed it also has a concept of a high-speed test to reject more non-corners then it already dose. It takes only 4 pixels, first it takes 1-st and 9-th and if they meet the condition next couple of 5-th and 13-th are checked. Only to those who passed high-speed test full check is applied. The method has its pros and cons so to fix some of cons machine learning non-maximal suppression is used.

Machine learning part of algorithm works as follows, it takes a set of images from the same domain and extract features as described before then forms vectors of 16 surrounding pixels for each feature point in all images from the set. Then it divides the vector into 3 sub vectors under 19 conditions. After that decision tree classifier is applied to these sub-vectors using the knowledge of whether feature point is a corner or not, let this value be $K_p$. It picks some value $x$ which yields the information measured by the entropy of $K_p$. This is applied to each sub-vector recursively until the entropy becomes zero. This decision tree is used afterwards for fast feature detection. Applying machine learning helps to get better performance in cases when number of contiguous pixels is less then 12 and to improve not optimal pixel's picking mechanism. There is also one additional option that improves feature detection and it is rejecting unnecessary feature points in locations where more then one feature were found way too close to each other. This is done by applying non-maximal suppression where depending on value of score function $V$, which is the sum of absolute differences between current feature point

and its 16 surrounding pixel values, features are discarded or not. FAST is at least twice faster then other algorithms but it is strongly influenced by the presence of noise in the image but can be adjusted by the threshold value.

$$S_{p\to x} = \begin{cases} darker, & \text{if } I_{p\to x} \leq I_p - thresh \\ similar, & \text{if } I_p - thresh < I_{p\to x} < I_p + thresh \\ brighter, & \text{if } I_{p\to x} \geq I_p + thresh \end{cases} \tag{19}$$

Figure 23: FAST feature state conditions.

Binary Robust Independent Elementary Features (BRIEF) was published by Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua in 2010. This algorithm brings a significant improvement to the way how features are matched. Basically, it is not a fully functional feature detection algorithm, it is only matching already found features thus have to be used in tandem with some feature detection algorithm. SIFT and SURF use vectors as feature descriptors but they take enormous amount of space, 256 bytes per feature which is not sufficient for embedded systems, for instance. These vectors can be compressed to binary string that can be used to match features with Hamiltonian distance which is a simple XOR on bit count operations. Unfortunately compression does not solve the problem of memory consumption because it still has to compute descriptors. Here comes BRIEF, it can find this binary strings without actually finding feature descriptors. In order to do this, it uses unique algorithm to get some set of location pairs from the smoothed image patch. Iterating over this set it compares elements of pairs and compares their intensity, depending on the result of comparison it creates a bit-string of zeroes and ones. Afterwards, Hamiltonian distance can be used to match the results.

In 2011, Vincent Rabaud, Ethan Rublee, Kurt Konolige and Gary Bradski from "OpenCV Labs" created an alternative to SIFT and SURF. They described it in the paper called "ORB: An efficient alternative to SIFT or SURF". The main attractiveness of Oriented FAST and Rotated BRIEF (ORB) is that it is free from patents while is good as SIFT and SURF. Basically it uses FAST for key points detection and by applying Harris corner finds $N$ best points. It also uses BRIEF as the feature descriptor. To fix the issues with rotations for both FAST and BRIEF it suggests some additional computations. As an improvement to FAST, to get the rotation algorithm computes the direction vector from the corner point to the intensity weighted centroid with this corner placed in the center. For BRIEF it controls the process according to key points orientation. For a set of binary tests at some location it creates a matrix, with a size twice larger then the set, with

coordinates of pixels. Afterwards using the orientation of key point it computes rotation matrix with the help of which it computes rotated version of matrix with points coordinates. In addition, it uses multi-probe local sensitivity hashing instead of original version of it for matching. The algorithm is good for inefficient devices with low computational power.

Matchers are algorithms for checking feature descriptors for similarity. Query descriptor can be compared with all other descriptors via Brute Force approach or use other algorithms with better heuristic like k-Nearest Neighbors (kNN) or Fast Library for Approximate Nearest Neighbors (FLANN). The main problem is that each matcher can't work with all feature descriptors. For instance simple FLANN can't work with bit strings as descriptors while FLANN with locality sensitive hashing index can do this.

The most popular combinations of feature detector, extractor and matcher are SURF with SURF and FLANN, SIFT with SIFT and FLANN, ORB with ORB and Brute force, ORB with BRIEF and Brute force. FAST can be used as a feature detector in all these combinations.

Brute-Force matcher simply takes the descriptor of some feature from the first image and compares it with all other features descriptors from the second image using distance calculation so the closest feature is its match.

FLANN is used for large data-sets because it provides much better performance then Brute force. It consists of a set of algorithms for nearest neighbour search so that those algorithms are optimized to be faster.

Homography, in computer vision, is a relation between two point's sets from the same planar surface which is true for the pinhole camera model. It can be computed with knowledge of camera parameters so that rotation and translation information can be extracted. This information is useful in many tasks; they are augmented reality, navigation and etc.

In computer vision, machine learning techniques are widely used. One of them is classification, analyzing new observed features for belonging to some defined categories. The process of creating those categories is also known as clustering.

In pattern recognition, the kNNis a non-parametric method used for classification and regression. In both cases, the input consists of the $k$ closest training examples in the feature space. The output depends on whether kNN is used for classification or regression.

kNN algorithm is one of the simplest algorithms in machine learning. It is used for classification and regression and belongs to a type of lazy learning algorithms, which means that the function is just approximated but all the computations are delayed until classification begins. The idea under beneath is quite simple, to classify the object it take into consideration the majority of votes from the object $k$ neighbours, while $k$ is user-defined. This means that object will be labeled

with that class which is the most repeated through out the neighborhood. During the training phase algorithm only stores the feature vectors and labels of classes retrieved from training data. Two commonly used distance metrics in the algorithm are Euclidean and Hamming depending on what should be classified, for example, Hamming is used for text classification.

Support Vector Machine (SVM) algorithm was developed by Vladimir Vapnik and Alexey Chervonenkis in 1963 but in 1992, Bernhard Boser, Isabelle Guyon and Vladimir Vapnik suggested a way to make SVM nonlinear classifier. SVM as it is known today, with soft margin, was published in 1995 by one of its initial authors, Vapnik, and Corinna Cortes. SVM algorithm is a set of controlled learning models used to analyze data for regression and classification analysis. With a help of training data this algorithm builds a model that can classify new samples to one of the two categories. Thus it is linear binary classifier but with a help of trick, called "kernel trick", that performs implicit mapping to a high-dimensional feature spaces, it can become non-linear classifier. In addition, it can work with unlabeled sample data with a help of support vector clustering algorithm. Basically, SVMs are very useful for working with text, especially handwritten, because they make pre-labeling stage unnecessary. They also show high search accuracy after multiple runs of relevance feedback.

Connected components analysis or labeling is a category of algorithms that are used to identify and analyze connected set of pixels. In general it is done by taking binary image and producing another image where each pixel is marked for belonging to some group, object on the image represented as a blob of connected pixels.

Figure 24: A binary image with 5 connected components.

There are many various algorithms for the connected component analysis. Some of them can keep whole image in memory, which is not suitable for low-performance computers, and process one component at a time while moving across the image. On the other hand, there are some algorithms that can process very large images while working only with some blob at a time, keeping some piece of

image in the memory but not the whole image.

| | | |
|---|---|---|
| | 1 | |
| 2 | * | 3 |
| | 4 | |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | * | 5 |
| 6 | 7 | 8 |

Figure 25: Scan-line order.

Recursive labeling algorithm works as follows, it take binary image and negates it to make all pixels with value 1 to be $-1$, this is done to distinguish unprocessed pixels from the labeled, whose label is 1. After that it takes pixel with $-1$ value and assign it a new label then it searches for a neighbour and recursively repeats the process. Pixel neighbours are returned according to the pattern specified on the figure 25, which is scan-line order.

Row by row labeling algorithm or Two-pass algorithm is based on connected components algorithm for graphs [6]. It performs two passes, on the first one it records stores equivalences and mark pixels with temporary labels. Before the second pass it analysis binary relations from the stored equivalences to generate equivalence classes of the relation. Afterwards, on the second pass it reassigns pixels with its equivalence class instead of temporary value.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 2 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 2 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 2 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| 3 | 3 | 3 | 3 | 0 | 4 | 0 | 2 |
| 0 | 0 | 0 | 3 | 0 | 4 | 0 | 2 |
| 5 | 5 | 0 | 3 | 0 | 0 | 0 | 2 |
| 5 | 5 | 0 | 3 | 0 | 2 | 2 | 2 |

Figure 26: Grid representations of binary image and it's labeled version.

#### 2.1.1.2 Stereovision basics

Stereo vision is a field of computer vision and its purpose is to provide information about 3D structures on digital images. This is achieved by comparing some feature points of scene taken from two views at the same time. In traditional way it is done by a stereo camera, which is, basically, two cameras placed horizontally within some distance, as human eyes. This technique allows to get relative distance from a camera to the object on the scene. This information is also known as relative depth information which is represented as disparity map.

There are few steps that precede finding of depth information because images taken by camera could contain some distortions and etc. The first and the most important step is to calibrate cameras with their inartistic parameters to make it function as a single system. This operation will provide extrinsic parameters of the camera, which is rotation and translation vectors of the camera. Calibration is achieved by showing calibration grid to the camera which looks like a chessboard. It is used because size of squares should be known and the grid pattern is simple for detection.

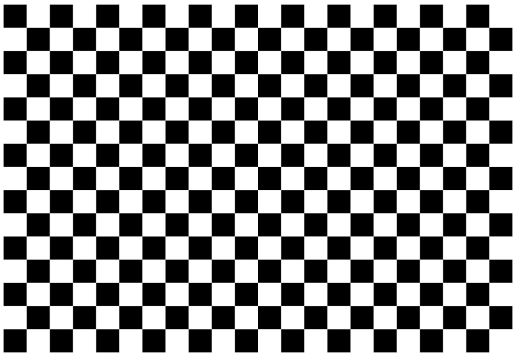

Figure 27: Calibration grid.

The results of calibration process are used to rectify images so that both images are changed in order to be projections to the same real world plane. Rectified images significantly reduce computations that are needed to get the 3D information from the scene. This refers to epipolar geometry, stereo vision geometry. It describes relations between projections and 3D points that comes from the assumption that camera corresponds to the pinhole camera model. This model describes already mentioned relations from the ideal pinhole camera perspective. Pinhole is a light proof box with a tiny hole from one side thus pinhole camera does not have any lens, as a result, it produce inverted image projected to the opposite to the hole side. From the nature of the camera, model does not take into consideration

any kind of distortions caused by lens and etc. To correct the lens distortion pixel to real 3D coordinates transformation can be used.

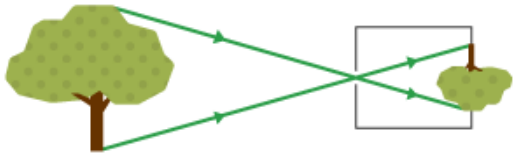

Figure 28: Pinhole camera model.

Epipolar constraints between two cameras are described by essential and fundamental matrices. Essential matrix can be thought as a pre-step before fundamental matrix. One limitation of the essential matrix is that it can be used only with calibrated cameras however if the cameras are calibrated it can be used to get relative orientation in the cameras space and to find 3D position of point's pairs. It is possible to extract rotation and translation vectors with this method using Singular Value Decomposition (SVD) decomposition. On the other hand, fundamental matrix deals with the uncalibrated cameras. In case having fundamental matrix and camera calibration information it is possible to get essential matrix for further processing.

$$E = K'^{T} F K \tag{20}$$

Figure 29: Equation of Essential matrix, K is the intrinsic camera parameters. [see 3, p257]

In a simple stereo-vision system depth of a point can be easily found with an equation 21 where $f$ is a focal length of the camera, $b$ is a distance between cameras, $d$ is a distance between corresponding points, also known as disparity.

$$Depth = f * b/d \tag{21}$$

Figure 30: Formula of the point depth.

To determine the 3D point coordinates by two or more its 2D projection so called triangulation is used. It is much easier to do if the images are rectified but even if they are not they can be transformed to match the requirements.

$$Z = f\frac{B}{D}, X = u\frac{Z}{f}, Y = v\frac{Z}{f} \tag{22}$$

Figure 31: Formulas of 3D points.

Basically, if the baseline is known, 3D coordinates could be found with the equation 22 where B is a baseline, d is a disparity, f is a focal length, u and v are column and row with the origin in center of the image.

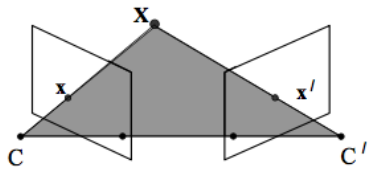


Figure 32: Triangulation illustration.

#### 2.1.1.3 Structure from motion

[7]

Structure from motion problem is, basically, the case of finding a set of 3D points with projection matrices and translation vectors for corresponding views from the set of images. This process is a 3D reconstruction from a sequence of images.

Sequential methods are usually used, they are iterative thus reconstruction is done partially, step by step. The process starts from the first image, when the new image is registered algorithm processes new portion of data, performs triangulation and add new 3D data to the reconstruction model. Initialization is usually achieved by finding fundamental matrix from the first two views and decomposing it.

These methods have some complications, first of all, they require huge amount of corresponding points per each image in a sequence. Usually seven or more correspondence must be present at three or more views. Large set of images require too much computational power in order to process these. Secondly, there are a number of structure and motion combinations that are not appropriate for the mentioned methods. These cases might be the camera rotation without any translation or planar scenes. Basically, it is impossible to avoid those cases without an expert planning on how to take pictures for the structure from motion sequence.

The most common strategy for registering images is epipolar constraints. It is achieved by using the correspondence of the image from the current view to the image of the previous view. Essential matrix is typically used but intrinsic camera parameters must be known. Its decomposition gives relative camera orientation and translation vector.

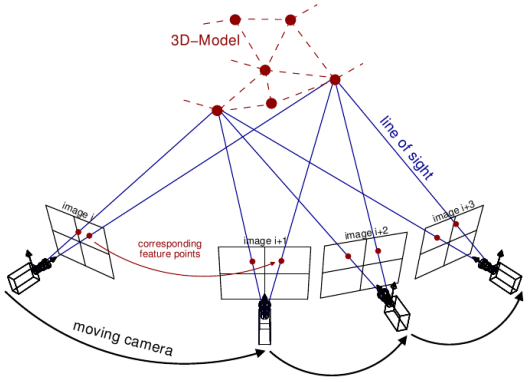


Figure 33: Demonstration of incremental structure from motion. (http://www.theia-sfm.org/sfm.html)

On the other hand, factorization methods do the job simultaneously. These

methods belong to the family of batch methods. SVD factorization based linear methods have been created for many affine camera models like orthographic, para-perspective or weak perspective and etc. These methods distribute reconstruction error among all measurements but, unfortunately, they are not applicable for the real world situations because camera lenses have too wide angle thus cannot be approximated as linear.

Lastly, after receiving initial estimations for 3D points and projection matrices it is needed to minimize function cost with the non-linear iterative optimization. This optimization is called bundled adjustment. It, basically, refines camera and structure parameters initial estimations to get those parameters that predict the locations of features among all images in the most efficient way.

$$\min_{a_j b_i} = \sum_{i=1}^{n} \sum_{j=1}^{m} v_{ij} d(Q(a_j, b_i), x_{ij})^2 \tag{23}$$

Figure 34: Bundle adjustment minimization function.

In the equation 23 $a$ is a vector that parametrizes camera and $b$ is a vector for 3D point so that $Q(a_j, b_i)$ is the prediction of $i$ point projection onto $j$ image. Euclidean distance is represented as $d(,)$ where its parameters are vectors.

#### 2.1.1.4 Tracking and Mapping

Tracking and Mapping is somewhat a logical group that unites multiple algorithms under the hood. For instance, real-time Structure from Motion is also referred to as monocular SLAM which is a group of tracking and mapping algorithms.

SLAM approximately solves the problem of building and updating the map of environment; at the same time it computes and keeps track of the object position in the environment, e.g. robot.

Maps are used to determine a position in space and for a graphic representation of a terrain plan or for navigation. They are used to assess the actual location by recording information obtained from the perception form and comparing it with the current set of representations. The contribution of maps to the assessment of the current location in space increases with a decrease in the accuracy and quality of space sensing sensors. Maps basically reflect the type of space fixed at the time of their construction. It is not at all necessary that the kind of space will be the same at the time of using the maps.

The complexity of the technical process of determining the current location and the construction of the map is due to the low precision of the instruments participating in the calculation of the current location. The method of simultaneous localization and mapping is a concept that connects two independent processes into a continuous series of sequential computations. In this case, the results of one process are involved in the computation of another process.

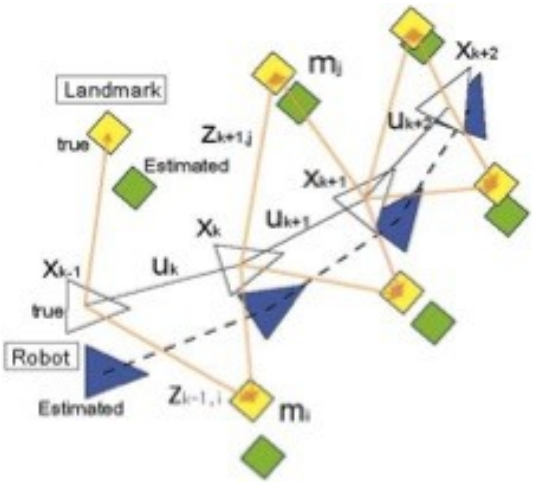


Figure 35: SLAM demonstration.

Monte Carlo methods and Extended Kalman Filter are often used in approximation process. These statistical approximation methods estimate object's pose

and map environment parameters probability functions. Set-membership methods generate a set where object's pose and map approximation are contained. These methods are based on interval constraint propagation which is used for propagating uncertainties if the error data is represented as intervals. Already known Bundle adjustment can also be used for a better map reconstruction based on simultaneous estimations of object poses and landmarks positions.

SLAM group of algorithms usually used in robotics, self-driven cars,and etc. Despite this fact, it is still an open field of research since current solutions are not ideal.

Methods for finding camera pose in an unknown environment were already developed by adapting monocular SLAM. Two the most known algorithms are EKF-SLAM and FastSLAM 2.0 but they are incremental which means that tracking and mapping operations are done one after another in a single pipeline. This leads to an update of every landmark and camera position at every frame. While this approach might be suitable for robotics it is not good enough for hand-held camera. This statement is argued by the fact that robot is able to move at a stable constant slow speed, for the hand it is not that easy to do, and robot usually uses more data then just camera to build map with SLAM. Errors generated during the work-flow of hand-held monocular SLAM can cause the corruption of generated maps in the mentioned iterative methods.

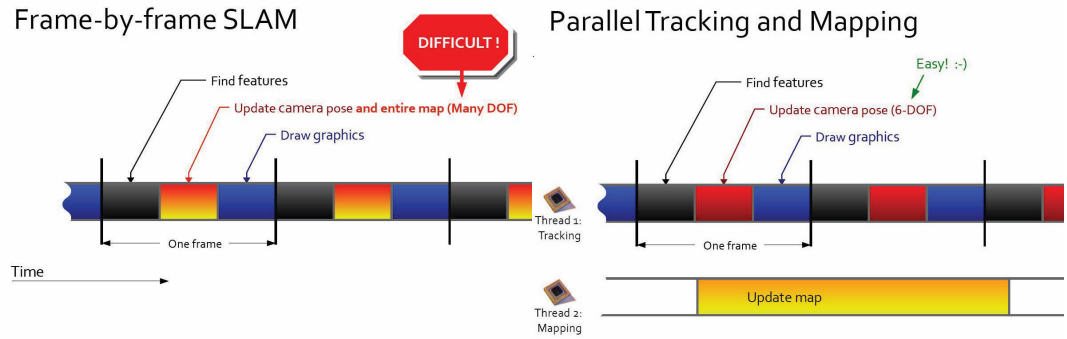


Figure 36: SLAM and PTAM work-flow comparison.

Since none of mentioned approaches was robust enough for hand-held case for the augmented reality usage, Georg Klein and David Murray developed PTAM. It offers the approach where tracking and mapping processes are not linked together so that they can be executed in parallel. It makes tracking probabilistically independent from mapping thus any tracking method can be used, for instance, authors used coarse-to-fine with robust estimator.

PTAM initialization should be performed before tracking begins. Initialization

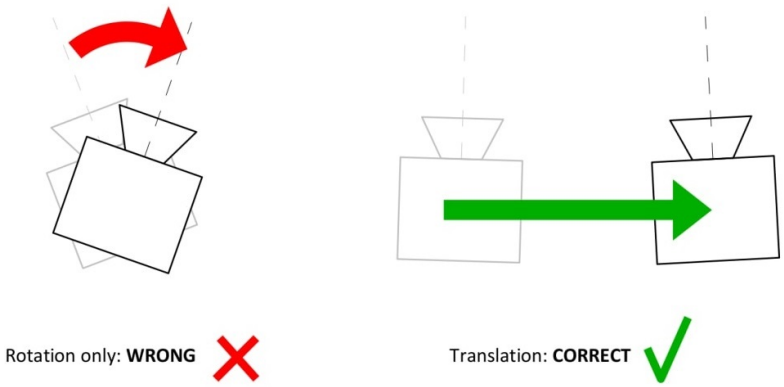


Figure 37: PTAM initialization.

movement of camera must include slight translation, just rotating camera will fail it. Decoupling of tracking and mapping allows algorithm not to process each frame since the algorithm is not iterative. Selective processing of frames is necessary for rejecting those with redundant information, for example, similar frames when camera is not moving. This selectivity helps to speed up the algorithm in comparison with iterative methods. Fact that key-frames are not following each other frame by frame means that processing can be done not within a strict real-time but should be finished by the time when new key-frame is added.

From the other side, bundle adjustment replaces incremental mapping. It was chosen because of its successful usage in real-time visual odometry and tracking same as it is well-proven instrument in Structure from Motion. The initial map is built with five-point algorithm. Tracking is achieved by applying local bundle adjustment to the $n$ most recent camera poses where $n$ is set manually to achieve real-time performance. Building long-term map makes this algorithm different in comparison with other solutions. This map is built so that feature points are visited over and over again thus it can perform full-map optimization. In addition, 2D features tracking was not efficient enough for initialization step thus authors decided to rely on epipolar feature search.

On the contrary, there is another algorithm aiming to solve the same problem as PTAM, its name is Dense Tracking and Mapping in real-time (DTAM). Authors claim that dense methods for tracking and mapping can provide more accurate and robust results in comparison with other world perception models based on features. In contrast with other real-time monocular SLAM algorithms it builds dense 3D model and uses it for camera pose tracking.

Basically, to track the camera motion at each frame, since it is iterative algo-

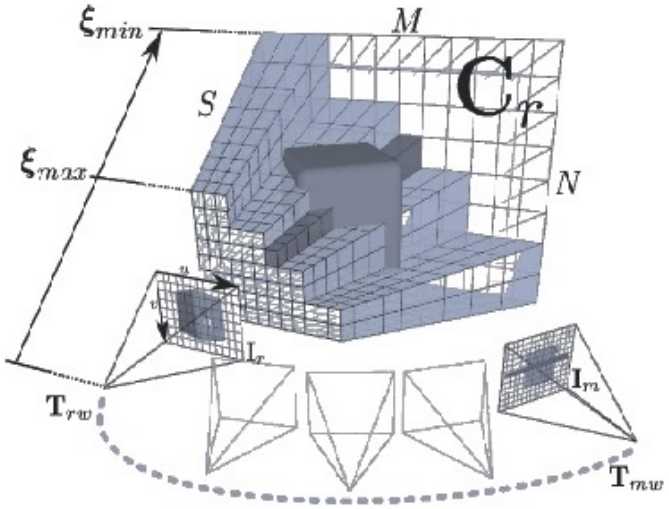


Figure 38: Estimating depth map from the bundle of images.

rithm, it uses dense frame alignment to compare to the already build scene dense model. At each new frame the dense model is expanded and modified with the help of the dense textured depth maps. The texture-mapped model is generated from depth maps, they are created with the help of stereo multi-view and dense reconstruction from a set of images.

In addition, photometric information is gathered sequentially in order to solve depths maps incrementally. They have done this with the help of non-convex optimization and accelerated exhaustive search in order not to loose small details.

The way of how DTAM tracks camera pose is more robust then its competitors and at least as good as feature-based ways of doing that. This happens because dense model is capable of handling occlusions and scale operations. Everything has its pros and cons so do dense models, they perform worse because of motion blur or unfocused camera. Algorithm performs very good for local illumination changes while it is not robust for global illumination changes. Normalized cross correlation measure was also used for better handling of local and global illumination changes.

After bootstrap is done system becomes self-supported and no tracking is required.

Authors believe that people do not understand the power of dense methods for tracking and reconstruction even though they bring a lot of workarounds for feature-based algorithm's problems. It was also claimed that dense methods are the most robust and accurate because of the ability of matching at every pixel.

### 2.1.2 Implementational

This section describes researched .NET libraries used for computer vision programming and mathematical operations.

#### 2.1.2.1 OpenCvSharp

OpenCvSharp is one of many C# OpenCV's wrappers. It has been created by the guy from Japan on GitHub called shimat. This project is opensource and licensed under BSD 3-Clause License. This wrapper is good for its similarity to the classical C++ Application Programming Interface (API) of Open Computer Vision (OpenCV) though it easy to translate C++ samples to C# without major changes. It was modelled close to the native C++ API as much as possible.

Almost all classes of this library implements *IDisposable* interface and that means there is no need to think of unmanaged resources as in C++, no memory leaks and etc. OpenCvSharp does not force users to stick to object-oriented work flow, if user wants to, it is possible to use functional style that is native for OpenCV. Library also includes transformation of OpenCV images to .NET bitmaps that simplifies image manipulations.

It can be used on both Windows and Linux but for Linux it is only possible via Mono Framework. In general, any platform that Mono supports can be used.

```
// Edge detection by Canny algorithm
using OpenCvSharp;

public class Program
{
    public static void Main()
    {
        Mat source = new Mat("image.png", ImreadModes.GrayScale);
        Mat target = new Mat();

        Cv2.Canny(source, target, 50, 200);
        using (new Window("Source image", source))
        using (new Window("Target image", target))
            Cv2.WaitKey();
    }
}
```

Figure 39: OpenCvSharp Canny edge detection sample.

As it can be seen from figures 39 and 40 API's are very similar but C++ requires more code to write for the same operation but it is quite easy to translate these samples in both ways due to their similarity.

```
#include <opencv2/core/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <iostream>
#include <string>

using namespace cv;
using namespace std;

int main( int argc, char** argv )
{
    string imageName = "image.png";

    Mat src, src_gray;
    Mat dst, detected_edges;

    src = imread(imageName.c_str(), IMREAD_COLOR);
    cvtColor(src, src_gray, COLOR_BGR2GRAY);

    Canny(src_gray, detected_edges, ...);
    dst = Scalar::all(0);

    src.copyTo(dst, detected_edges);

    namedWindow( "Display window", WINDOW_AUTOSIZE);
    imshow("Source image", image);
    imshow("Target image", dst);

    waitKey(0);
    return 0;
}
```

Figure 40: C++ OpenCV API Canny edge detection sample.

#### 2.1.2.2 EmguCV

Emgu Computer Vision (EmguCV) is one of the most mature and complete OpenCV wrappers for the .NET platform. For the open source projects it is free if the project source code will be contributed to the open source community with the right to share it with anyone. This is done under GNU GPL license v3. On the contrary, for the private use commercial licence have to be bought.

This framework is cross platform thus it can be used almost on every known platform such as Windows, Linux, Mac OS X, iOS, Android and Windows Phone. The whole implementation was written in pure C# thus all the header files were

ported.

In general, EmguCV is on of the best wrappers for the .NET but it has its pros and cons. If the advantages were already mentioned its disadvantages are still hidden. Basically, the main problem is that EmguCV has its own enums and some methods that do not match with those in original C++ API. This fact makes translating some code samples a hard task or sometimes even impossible. From the figure 41 that common syntax is made in the functional way thus it is easier to understand it to the C++ API people.

```
using System;
using Emgu.CV;
using Emgu.CV.CvEnum;
using Emgu.CV.Structure;

namespace HelloWorld
{
   public class Program
   {
      public static void Main(string[] args)
      {
         String win1 = "Test Window";
         CvInvoke.NamedWindow(win1);

         Mat img = new Mat(200, 400, DepthType.Cv8U, 3);
         img.SetTo(new Bgr(255, 0, 0).MCvScalar);

         //Draw "Hello, world." on the image using the specific font
         CvInvoke.PutText(
            img,
            "Hello, world",
            new System.Drawing.Point(10, 80),
            FontFace.HersheyComplex,
            1.0,
            new Bgr(0, 255, 0).MCvScalar);


         CvInvoke.Imshow(win1, img);
         CvInvoke.WaitKey(0);
         CvInvoke.DestroyWindow(win1);
      }
   }
}
```

Figure 41: EmguCV hello world application sample.

#### 2.1.2.3 Accord.NET

Accord NET is a framework for machine learning, math statistics, basic scientific computing and computer vision entirely made with C#. It includes classification algorithms (SVM, decision trees, neural networks, deep learning and etc.), regression algorithms (linear regression, polynomial regression, logarithmic regression, logistic regression and etc.), clustering algorithms (K-Means, Mean-Shift, Gaussian Mixture Models and etc.), algorithms for distribution analyses.

In general, Accord.NET was created to extend features of the AForge.NET created by Andrew Kirillov for the .NET Framework in 2006. However in 2012 Andrew announced the end of support for the AForge.NET thus Accord.NET absorbed most of the original AForge.NET code into its code-base and continued to develop and support the code under Accord.NET name.

```
double[,] A =
{
    {1, 2, 3},
    {6, 2, 0},
    {0, 0, 1}
};

// Singular value decomposition
var svd = new SingularValueDecomposition(A);
var U = svd.LeftSingularVectors;
var S = svd.Diagonal;
var V = svd.RightSingularVectors;

// Eigenvalue decomposition
var eig = new EigenvalueDecomposition(A);
var V = eig.Eigenvectors;
var D = eig.DiagonalMatrix;

// LU decomposition
var lu = new LuDecomposition(A);
var L = lu.LowerTriangularFactor;
var U = lu.UpperTriangularFactor;
```

Figure 42: Accord.NET decomposition samples.

As can be seen from the 42 Accord.NET includes a huge amount of different mathematical extensions for the standard .NET value types thus having a single .NET matrix a value decomposition can be done in 1 line of code. This library has such type of helpers for the enormous amount of mathematical operations.

## 2.2 Kalman filter

Kalman filter is one of the most known and used set of mathematical tools for sensors noise influence reduction. It was named after the author of "A New Approach to Linear Filtering and Prediction Problems" [4] R. E. Kalman. Basically, this filter is a set of mathematical equations designed to solve the predictor-corrector estimation problem and its huge advantage is that it minimizes estimated error covariance. Advantages in digital computing and simplicity made this filter very practical and provoked an extensive research in the field of navigation.

Generally, Kalman filter can be used in any dynamic system where some uncertain information exists. This filter is making educated guesses about the next step of the system based on mathematical model of the system. Even when noisy data interferes with the real measurements Kalman filter does a great job on guessing what is really going on in the system. Due to the nature of Kalman filter it is an ideal tool for dynamic, changing systems. The reason for that is it does not require huge amounts of memory or computational power since it only needs to store previous step state in order to compute the prediction and current measurements to adjust current system state. All of that is done by a set of mathematical equations thus it makes Kalman filter a great tool for real time or embedded systems.

2.2 Kalman filter vision

Kalman filter requires mathematical description of the system with its parameters and it assumes that these parameters are Gaussian distributed and random. Every parameter has the most likely state which is a mean of the random distribution and range of uncertainty values, variance. One more important thing about parameters is that they must be correlated which means that one parameter can be used to find the other one. Basically, Kalman filters aims to get as much information from uncertain measurements as possible. It describes correlation of parameters with so called covariance matrix. This matrix is symmetric and its values are degrees of correlation between two parameters states where these parameters are taken depending on the indices of the matrix value. Covariance matrix is often labelled as $\sum$.

As a next step, filter must somehow predict the next step depending on the previous one without knowing which state is real and which is not. This is done by prediction matrix F which should transform estimation of one state to the predicted one assuming that the original state was real. This manipulations leads to covariance matrix changes, if all values in the distribution are multiplied by prediction than new covariance matrix will be computed as multiplication of prediction matrix by covariance matrix and by transposed prediction matrix.

2.3 Extraneous known influence

$$\hat{x_k} = \begin{bmatrix} \text{position} \\ \text{velocity} \end{bmatrix} \tag{24}$$

$$P_k = \begin{bmatrix} \sum_{pp} & \sum_{pv} \\ \sum_{vp} & \sum_{vv} \end{bmatrix} \tag{25}$$

Figure 43: Example of covariance matrix with position and velocity parameters.

$$\hat{x_k} = F_k \hat{x}_{k-1} \tag{26}$$

$$P_k = F_k P_{k-1} F_k^T \tag{27}$$

Figure 44: Example of computation the next step state and covariance matrix equations.

In the real world it might happen that the system is affected by something not related to the systems inside world, for instance, hole in the road that will affect the way car goes or the wind blowing at some region. This knowledge of some external factors that may affect system can be described mathematically and added to the prediction calculation as a correction. They are expressed as control vector and matrix multiplied together. Control vector contains known outside parameters, for example, speed of the wind, and control matrix contains changing parameters, for example, time delta between two moments on the time-line.

$$\hat{x_k} = F_k \hat{x}_{k-1} + B_k \vec{u_k} \tag{28}$$

Figure 45: Example of expanded system state prediction equation.

2.4 Extraneous unknown influence

It also might happen that factors of influence are not known and they canâĂŹt be described as control matrix and vector. In this case it is possible to model some uncertainty around the predicted state. This influence is treated as noise with some covariance but because of multiple possible next step noisy predictions they are treated as a single Gaussian blob with different covariance but same mean. Thus, in order to get expanded system covariance matrix that takes into consideration possible noise it is needed to add noise blob covariance to the system covariance.

$$P_k = F_k P_{k-1} F_k^T + Q_k \tag{29}$$

Figure 46: Example of expanded system covariance matrix equation.

The new uncertainty is predicted with the previous one plus some correction on current uncertainty from the environment.

2.5 Prediction correction with measurements

System may contain several sensors that are giving some indirect information about its state. Data from sensors wonâĂŹt be the same as data tracked by Kalman filter. One fact about filter is that it is very good for handling sensor noise. Here comes sensor matrix that transforms the predicted state to a sensor reading prediction. The next step is computing the most likely state based on predicted state Gaussian distribution and sensor reading Gaussian distribution. To get the most likely state it needs to understand if two probabilities are both true. The first one is that sensor reading is estimated measurement and the second one is that previous estimate is the reading that should come from sensor. To get the most accurate estimate it multiplies those two Gaussian blobs and receives their intersection which is the estimate and the best guess. This new estimate is the Gaussian blob with its own mean and covariance. To get these mean and covariance matrix two formulas are used 30 and 31.

$$\vec{\mu'} = \vec{\mu_0} + K(\vec{\mu_1} - \vec{\mu_0}) \tag{30}$$

$$\sum{}' = \sum_0 - K \sum_0 \tag{31}$$

Figure 47: Example of the new mean and covariance matrix equations.

K is a Kalman gain that is computed from modeled sensor matrix $H_k$, system covariance matrix $P_k$, and the covariance of uncertainty (sensor nose) $R_k$.

$$K' = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} \tag{32}$$

Figure 48: Example of Kalman gain equation.

$$\hat{x_k}' = \hat{x}_k + K'(z'_k - H_k\hat{x}_k) \tag{33}$$

$$P'_k = P_k - K'H_kP_k \tag{34}$$

Figure 49: Example of the final estimation equations.

From the equation 33 $z_k$ is the mean of the estimated sensor distribution and it is equal to the reading from the sensor. All of these manipulations are the update stage of Kalman filter.

2.6 Conclusion

To implement linear Kalman filter it is basically needed to only implement predict and update equations. After each time and measurement update pair, the process is repeated with the previous a posteriori estimates used to project or predict the new a priori estimates. This recursive nature is one of the very appealing features of the Kalman filter, it makes practical implementations much more feasible than an implementation of a Wiener filter (Brown and Hwang 1996) which is designed to operate on all of the data directly for each estimate. The Kalman filter instead recursively conditions the current estimate on all of the past measurements. For nonlinear systems Extended Kalamn filter is used. It linearize measurements and predictions of Kalman filter about their mean.

# 3 Experiments

This chapter contains detailed description of multiple experiments using knowledge from the Chapter 2.

## 3.1 Sudoku experiment

The OpenCV research has been started from the Sudoku Solver project. It was created with Python as it is the tool with lowest entry threshold. This project requirements were perfect for starting, they allowed to understand computer vision basics deeper and showed how to map existing practises onto the problem of object detection. Furthermore, not only object the detection but also the object character recognition was covered in it.

### 3.1.1 Detection and Extraction

In order to extract a Sudoku grid it should be found first. For this purpose some assumptions had to be made and this assumption was that Sudoku grid should cover from 60 to 100 percentages of an observed image. As a starting point image was turned to black and white with adaptive thresholding as the most suitable approach. The next step was contours detection and iteration over them. Contours were filtered with some specified threshold value and only closed were interesting for the process. In order to make the process a bit automative the program were able to change upper and lower thresholding bounds if the needed contour was not found with the default values. To get the largest contour maximal area filtering was applied. To detect if the largest contour is appropriate polynomial approximation was applied. This manipulations allowed to get straight lines in the contour and if four of them were found their intersections were returned as corner points. These points were assumed to be the corners of the Sudoku grid.

In order to extract the Sudoku grid properly linear transformation was applied to the image using detected grid's corner points. This approach has some limitation related to the quality of an image. If the image quality is poor, Sudoku grid borders are broken into pieces or do not exist then it is not possible to detect grid corners and extract the Sudoku grid from the image. However if the image quality is good enough then this algorithm is very sufficient because it performs well and it is reliable enough.

### 3.1.2 Transformation

Having the square shape image perspective transformation was applied. Perspective matrix is constructed with the corner points found on the previous step and the new image dimensions computed on this stage.

In 50 equations there are 8 unknown variables but they could be found using construction of 8 equations system (four for each of two equations). Using this system, it is possible to map points from distorted image to a flat square image.

$$x = \frac{ax + by + c}{gx + hy + 1} \tag{35}$$

$$y = \frac{dx + ey + c}{gx + hy + 1} \tag{36}$$

Figure 50: Perspective transformation equations.

### 3.1.3 Flooding

Initially this stage was done as coloring all the small connected components into the background color of the image but after refactoring it was changed to OpenCV method $fastNlMeansDenoising$ that removes noise from the image. Parameters for it were found empirically to get the best results and performance. This method created partially blurred imaged thus it was necessary to threshold image in order to get binary image. Image in this stage was inverted, it means background was black and numbers with lines were white.

### 3.1.4 Segmentation

This part of the program is responsible for determining where the numbers on the image are. Usually segmentation is the hardest part of the OCR process but, in this case, software should process standard 9 by 9 Sudoku, which grid is square and its size is already known, that is why there is no need to do complex segmentation operations. But even in case of none-standard grid it was possible to do the segmentation with Hough-lines algorithm, this would make the process universal. But for time reasons it was decided to stick to the 9 by 9 grid. It was cut into 81 equal pieces. Then all the pieces are checked whether they contain a number or not. In order to do this there was area selected so that it was half of the size of the piece and was centered. If the sum of the pixels inside was equal to 0 it means that it was empty. Otherwise, piece of the image was processed in order to find all contours by picking the middle point. Algorithm works as follows, it tries to find the biggest and the closest to the center point contour. Found contour is supposed to be a number. Then it expands the size of the image until some bounding value. Segmented images are stored in the matrix of image arrays and "None"'s and sent to the OCR stage.

### 3.1.5 Optical Character Recognition

The purpose of the OCR is to take an image, process it and return characters that are presented on the image. Every recognition algorithms has steps:

1. Determine input image features
2. Train a classification algorithm with some training data.
3. Classify input image

Images may have different parameters (shape, color scheme, borders, etc.). Before classifying it is needed to bring all the images to the same standard (shape, binary color). For this purpose images were resized to some specified size and image matrix was vectorized because vector must represent an image in the classification algorithm. Optimal size of the image was chosen by experiment. For SVM classification pixel values were changes to zero for black and one for white. That was done because SVM classification algorithm requires that kind of representation. kNN classification algorithm was also used for experiments. Both algorithms gave almost the same results.

Classification of Sudoku numbers was done as follows, the matrix with segments matrices in positions where some numbers are supposed to be and 'NONE' objects where it is not is transposed to a vector and passed it to a classification algorithm. Classification algorithm, from its side, returns the class of the classified image, which is the needed number.

### 3.1.6 Conclusion

This project gave a good understanding of computer vision algorithms and their purpose on the real example. Since the exploration of computer vision was started from scratch such kind of a project was necessary to gain needed background.

Python implementation had shown itself as a good approach for the images with medium or high quality. For the bad quality images results are not that good but the success rate is still more than 30 percents.

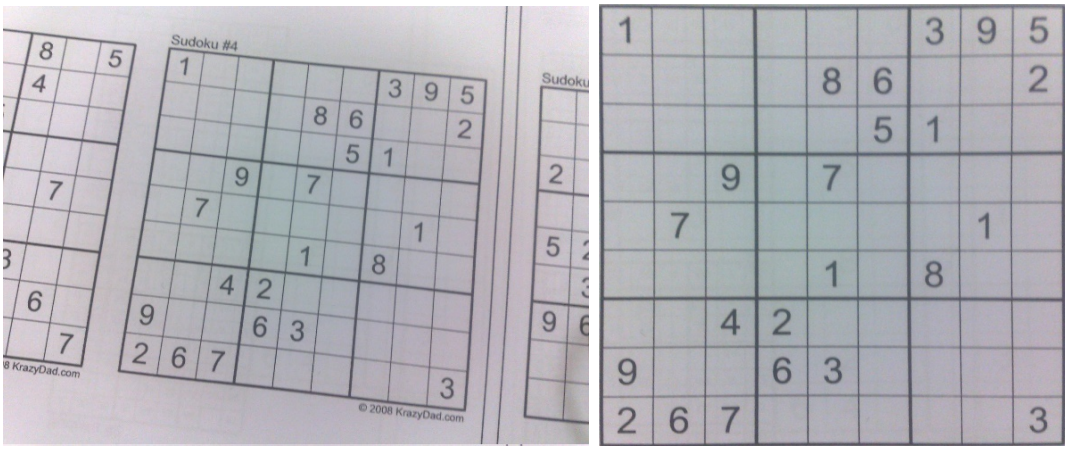


Figure 51: Extracting Sudoku's grid example.

Figure 52: OCR result displayed on the extracted grid.

## 3.2 Rubiks cube lines experiment

This project is a continuation of the previous experiment described in Section 3.1. Ideas developed for Sudoku were extrapolated on a Rubik's cube face detection problem as they are assumed to be similar in some sense.

### 3.2.1 Contour and corner detection

Contour detection was covered in the previous step but in this case there were some issues to be solved. From scratch, it was a problem that edges of the cube are not very clear so it was not possible to extract faces just by finding bounding boxes for faces contours. On the contrary, it was decided to use contours that were detected with better quality, contours of each piece of a Rubik's cube. So as the first step, the contours that are closed and whose approximation gives 4 corners were found, that means they might be our diffused square pieces of cube's face. These manipulations leaded to multiple duplicated for some the cube pieces contours. They occur sometimes if OpenCV sees a couple of different contours around the same object, so all the contours with almost the same center of mass were removed but one. At this point, corners of these distinct contours were computed with their diagonal to the horizon and if there are 9 of them with the same angle to the horizon then these are the face's pieces. The next step is quite simple, find the extremums or top-left, top-right, bottom-left, bottom-right points, depending on the angle of the face, these points will be the corners of the face.

### 3.2.2 Extraction

Since the detection of corners for each face is described in the previous section, at this point, there are 4 corners and they simply need to be passed to the next step where linear transformation will be applied. As a result, plane image of the face will be stored. This approach has the limitation for the input, it is not possible to detect face corners and extract the actual face if an image or video quality is quite poor, contours are broken or etc. However, if the source quality is good enough than this algorithm is quite sufficient because of its simplicity and performance.

### 3.2.3 Transforming Image

At this point the same approach as in the Sudoku experiment was used. The same set of equations was used to transform the image. With those system of equations it is possible to map points from distorted image to a flat square image. Basically, in the program, OpenCV's $GetPerspectiveTransform$ and $WarpPerspective$ methods were used in order to apply perspective transformation to the source.

### 3.2.4 Identifying unique faces

There are many possible options to check images for similarities like feature matching, color histograms, cross correlation, euclidean distance. Looking at a cube it is clear that there are no specific features on it, only colored pieces. The other case is the observation problem, cube can be shown in different orientations. From

the observations made, color histograms was chosen as a best choice to distinguish faces. It was decided to keep all uniquely detected faces in the list and compare each newly detected one with all from the list. In order to do that it is needed to convert images to Hue, saturation and value (HSV), then calculate histogram from HSV images and normalize them. In the end, just compare images with one of possible methods, in this case correlation was used, it worked fine. Basically, if the correlation is more than 0.6 it is assumed that faces are to similar and it could be the same face. This algorithm might not be the better choice but the empirical research showed that it works as expected.

### 3.2.5 Color detection

Color detection is nearly the easiest part of the program. In order to detect Red, green and blue (RGB) color representation for each piece of cube face it is simply needed to segment image into $n$ equal squares, since the 3 by 3 Rubik's cube was used, segmentation produced 9 pieces and this code does not work for larger cubes. After a piece of a face is cut out, the pixel in the center is taken with its intensity, that is all, RGB color detected. Collect them to some data structure and, basically, the process is finished. Extracted RGB colors were mapped to the specific data structure in order to convert it to the Rubik's cube 3D model for solving and rendering purposes.

### 3.2.6 Conclusion

This project was a continuation of the Sudoku ideas projected onto the Rubik's cube. At this stage single faces were extracted correctly but some faces extraction might fail due to illumination conditions. Basically, cube's surface may be different and it means that when the light falls on the surface it may be reprojected, create color distortions and brake edge detection. Under the good illumination conditions faces corners and pieces colors are extracted correctly.

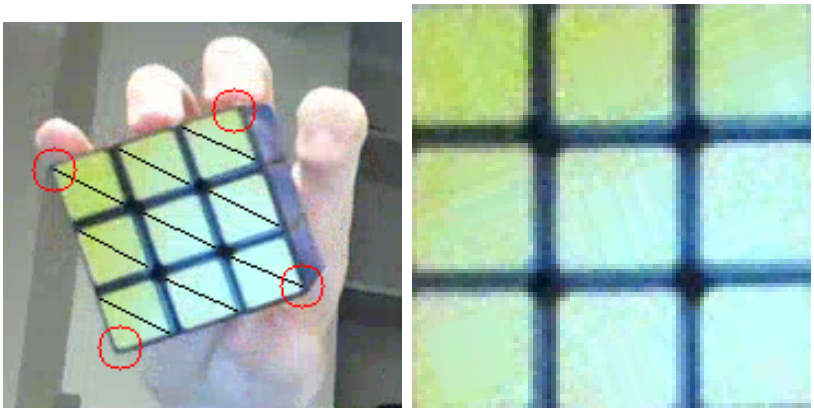

Figure 53: Detected and extracted face sample with distorted colors.

## 3.3 Rubiks cube plane intersection-experiment

At this point, there was an idea to use planes as an alternative to contours approach. PTAM was chosen for finding planes and in future it should become the mechanism for an object tracking due to its nature.

### 3.3.1 Bootstrapping

This step is the first one and the most simple. It just stores initial data, for instance, extracted key points of the first image, grayscale first image and sets the pipeline to move on. All the magic happens on next step.

### 3.3.2 Bootstrap tracking

This part of the algorithm is considered to be the most important. It should analyze images from a video stream or an image sequence by comparing it to the first image from the bootstrap step. As a result 3D point cloud is created, basically its simplified version. In addition, PTAM is the algorithm for augmented reality thus it is possible to project different objects onto the detected surface.

**Optical flow and homography filtering.**

In order to know where the key points of the previous image will appear on the next image this algorithm uses "Optical flow" algorithm. After that it checks if 80 percents of the points have survived, if not it stops bootstrapping and says that tracking failed. If this condition is satisfied and more than 4 points are tracked then algorithm applies homography, it takes bootstrap key points and tracked key points. Using the mask it checks how many points survived homography and

filters the lists of bootstrapped and tracked key points again so that these lists contain only the most precise key points. Then it analyzes camera motion using key points from the bootstrap step and currently tracked ones, it checks sufficient motion with OpenCV's function $estimateRigidTransform$. If all the conditions are satisfied it moves to the next step.

**Extracting essential matrix from fundamental.**

During this step it finds fundamental matrix using the bootstrapped and tracked key points. In computer vision, the fundamental matrix is a 3 by 3 matrix that relates corresponding points in stereo images. Then it filters key points again. The next step is computing essential matrix. In computer vision, the essential matrix is a 3 by 3 matrix, with some additional properties, which relates corresponding points in stereo images assuming that the cameras satisfy the pinhole camera model. The algorithm needs to have intrinsic parameters of the camera before it starts the work because it does not include calibration part. Intrinsic parameters are physical camera parameters like focal length and etc. To compute essential matrix it multiplies transposed intrinsic matrix by fundamental matrix and multiplies this by intrinsic matrix, not transposed. This matrix is needed to extract camera rotation and translation in relative perspective from the bootstrapped image to the current one. In order to do this it decomposes essential matrix with SVD. Decomposition results into 2 possible rotation matrices and 2 possible translation vectors. If the determinant of the first rotation matrix plus $1.0f$ is less than $1e-09$ than algorithm changes signs of all numbers of the essential matrix and computes decomposition again due to âĂIJShowing that it is validâĂİ chapter of essential matrix wiki page. Then it proceeds to the next step.

**Triangulation of points.**

During this step it assumes that the first image (camera) extrinsic parameters matrix and the second one are constructed from the rotation matrices and translation vectors. The resulting matrix is 3 by 4. Since there are 2 rotation matrices and 2 translation vectors, the algorithm creates 4 possible extrinsic matrices for the second camera and tries to triangulate until the correct one is found or fails if none was found. In order to triangulate, it normalizes bootstrapped and tracked key points coordinates. Then it proceeds triangulation and computes status by checking $z$ component of the points. It filters key points by this status array. After that it computes reprojection error. If the error is not in the acceptable range it runs triangulation again with different set of rotation and translation components. When the error is acceptable it goes to the next step where it finds the plane.

**Finding a plane using 3d point cloud.**

This step contains **PCA!** (**PCA!**). It is used to extract normal of plane and proceed key points filtering again. If more than 75 percents of points are on the same plane bootstrap tracking is considered to be finished.

### 3.3.3 Tracking

This step is the final for this algorithm. It calculates optical flow for the points that passed triangulation. Then it solves **Pnp!** (**Pnp!**) in order to calculate Model-View matrix for the rendering. That is it, simple augmented reality. The main difference of this algorithm in comparison with the real PTAM is that PTAM is tracking all captured plane points even if they disappear from the camera view when you put camera back it still sees those points. On the contrary, described approach simply cuts down all unseen key points until the minimal amount exists, when it is not satisfied it breaks. This simplification significantly reduces time for implementation and understanding of the algorithm work flow.

### 3.3.4 Planes for a Rubik's cube

The idea behind the PTAM usage was to find all contiguous planes where every 3 contiguous planes with 90 degrees angle between each other create a Rubik's cube angle. For this purpose, this triplets are used to find planes intersection (cube's corners). Using the triangulated data it is possible to find 3D coordinate for any 2D point in the specific view. These manipulations are useful in two senses, they allow to find a Rubik's cube corners and to track those corners in the 3D space, after some algorithm modification.

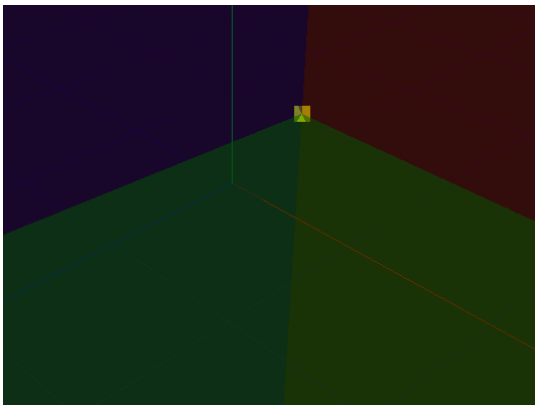

Figure 54: Example of planes intersection, cube's corner.

### 3.3.5 Conclusion

Algorithm performed very well for finding and tracking a single Rubik's cube face but it showed pretty bad results for the continuous running of the algorithm to find contiguous planes. The reason for that is illumination. The algorithm was able to find one, two or three planes in a row but it definitely will brake somewhere in the middle. This causes problems for triangulation stage, since the algorithm world space is relative to the first bootstrap thus if the algorithm fails the next bootstrap will become new first bootstrap, in simple worlds, algorithm is lost and it has to start from scratch. It is possible to modify this algorithm to handle such type of problems but that is the topic for one more research.

# 4 Contribution

This chapter contains detailed description of the final project road map; displaying how the ideas tried in the Chapter 3 were applied to develop "Framework for extracting and solving game puzzles (Rubik's cube like)".

## 4.1 Design

Design of the approach for a Rubik's cube detection and extraction was a tough thing. While it was easy to generate ideas meanwhile it was hard to prove that those ideas can survive experiments due to the quality of the image and large variety of algorithms that give good results in different situations. Basically, the design of this project contained multiple separate stages to complete the final goal, which was extract, map, solve and display the solution using 3D model. There was also tracking opportunity idea due to similarities of ideas for achieving extraction and tracking.

### Extraction

This is the hardest, from the research point of view, problem since it is related to the computer vision field of knowledge. There are large variety of approaches to deal with extraction. It could be done using feature key points or using contours and different line algorithms, blobs, histograms and etc. This work is an empirical research thus it was decided to try different approaches to see what fits better. The first try was the Sudoku solver project where all the basics of computer vision were tested, researched and analyzed. That project was a starting point and the place where contour detection for a square field detection was applied. It contained everything from detection to extraction and transforming. This idea was the first option to the assigned task. The next step was to find out how to do the same for a Rubik's cube face detection. The last idea was "Tracking and Mapping" group of algorithms. There are SLAM, PTAM, DTAM and many others included to this category. Those algorithms were picked for the research because they allow to analyze 3D structures and the cube is also a 3D structure. The thoughts behind this were to find and track cubes faces because they are planes. Having 6 perpendicular planes their intersections could be found which are 8 points, cube's corners. Knowing all corners and tracking the cube faces allows to find if 4 corners are presented at the current view thus it is possible to extract the rectangle formed with these points. There is also a simplified idea of using PTAM for face extraction. It can be supposed that the image mostly contains a cube thus any plane that is found is cube face. To extract it the blob that includes all tracked key points should be found. This approach is open for research because this blob can be

found in many possible ways.

### Tracking

"Tracking and Mapping" algorithms have tracking idea in their kernel. While for the extraction stage only their 3D analyses was necessary they created a possibility to make tracking of a cube through a video possible. There is also such a powerful tool for reducing measurement noise as a Kalman filter. Due to the predicting and updating nature this algorithm can be used for the tracking helper because when the face is not seen in a picture Kalman filter can still predict where it would go until it sees it again to update its model under the hood. Unfortunately, this topic is mostly derived from the previous one and does not affect any further steps thus it is not that critical.

### Mapping

This stage is a bridge between computer vision, rendering and solving parts of the project. There is some structure that represents the Rubik's cube. From the other side, there are 6 extracted faces and they should me mapped to that structure in order to do something with them. Mapping options vary depending on the picked extraction options. Using the contour way of extraction there is a list of 6 independent faces images while "Tracking and Mapping" allows to analyze corner positions and to know the relations between faces. That is why for the contour extraction way there should be some way to map faces correctly to the model. Since the pieces colors extraction is the next step for both ways it can be used for position analyses. The middle pieces colors have strict positions in relation to others thus it is easy to find where the face should go. But there is another problem, it is a face orientation. Mapping faces as they were extracted is a bad idea because with wrong orientation it will build wrong cube with impossible color combinations and it will not be possible to solve it while it is still possible to display this figure. The possible face orientation can be done in two ways also. There is an option to analyze possible cubies (some face pieces that form a cube piece) and to put new faces in correspondence with other already mapped faces. The second option is to map faces randomly and run some algorithm to detect whether this cube is correct, could be solved at all. If the result is "No" then rotate faces and try all possible combinations. This might be time consuming but will definitely give the result without any previous knowledge of possible relations between face pieces..

### Solving

This part of the project might be thought as an easy one but it is not true. There many different options to solve a cube and the implementation difficulties vary depending on the chosen algorithm. The first to think of might be brute-force algorithm but when for a 2 by 2 it is possible, for a 3 by 3 Rubik's cube it might take years to find the solution. Thus it is better to look at existing solutions. The algorithm for beginners was picked because it contains a small amount of formulas to know for solving a cube from scratch. This choice is argued by the fact that, for instance, Fridrich method contains 119 formulas and while it gives shorter and quicker solution it is hard to implement it without knowing all the formulas. To deal with this it is simply needed to run the formulas on the cube model depending on its state.

### Rendering

Rendering might be easy if you dealt with it before. This field of knowledge same as computer vision requires some mathematical background to be used. This stage also contains a couple of options how to be solved. The first one is to use OpenGL which requires some time to understand the technology and start using it. Another option is to implement simple software rendering engine that does not use Graphics processing unit (GPU) but rather uses Central processing unit (CPU) to render an image. This process is much more interesting while it still gives you some background for further usage of Open Graphics Library (OpenGL), for example.

## 4.2 Implementation

Implementation was done using .NET C# and its wrappers for OpenCV library. It could be easily reasoned by the fact that it is better to deal with the tough things using the tool that you know and can use to reach the aim. On the contrary, C# is a great tool with many convenient tools that help to make implementation quicker and less painful. Multiple OpenCV wrappers were used in the implementation on different stages of development process. EmguCV was applied later because open source wrapper OpenCvSharp that was good enough on the whole way before was not able to do specific operations for the PTAM code base thus it was replaced.

All various logic units were split into the separate static classes for exact purposes. The project was also split into some set of libraries projects in order to separate OpenCV, Rubik's cube and Rendering code bases. Since the project is called framework, every library can be easily substituted with any other. The project also contains 2 projects for camera calibration for different versions of EmguCV.

### Extraction

Extraction logic is contained in *RubiksCube.OpenCV* project and its *FaceDetector* and *FaceExtractor* classes. The workflow is as follows *FaceDetector* calls another utility class *ContourUtil* to find specific contours. These specific contours are those who passes specified in the configuration threshold values and whose approximation shows 4 corners and convex structure. Afterwards, found contours are distinguished by the center of mass, it means that contours with similar or close enough center of mass are rejected to keep only one of them. This is done with OpenCV method *Cv2.Moments* and some magical C# Language-Integrated Query (Linq) code. After that, filtered contours are checked for having the same angle to the horizon. With 9 distinct contours under the same angle it is supposed that 9 face pieces are found. They form a cube face thus it is needed to find 4 extremums of the group which are face corners. If some points appear to be the same or to close then additional filtering logic is applied. It is needed to extract the found face thus *FaceExtractor* utility class is called to extract OpenCV *Mat* class containing extracted and transformed in perspective face image. With the help of *FaceUniquenessDetector* utility class extracted *Mat* object is checked for its originality. It is done with OpenCV *Cv2.CalcHist* and *Cv2.CompareHist* methods, in short. If the correlation between the current face and any of the previous ones is too high, the current face is not unique. The last step is extracting colors from unique faces and forming some data structure to pass it to the next stage. *ColorsExtractor* utility class is used to extract colors. It has to segment a face into specified, in the configuration, number of pieces and take color probe from the center pixel of each piece and convert it to the .NET *Color* class. Later this colors are compared to six possible cube face colors and converted to the closest to them using *ClosestColorHue* of *ColorsExtractor*. That is it, extracted colors are sent to be mapped to the Rubik's cube operational and rendering models. PTAM was another implemented algorithm but since its code was initially written with C++ and some currently unsupported libraries it was decided to write another simplified version of PTAM. It does not allow to track key points when they are out of the point of view but while camera sees them everything is as designed, all main stages are included. PTAM like approach is implemented in *RubiksCube.OpenCV.TestCase PtamLileAlgorithm* class split into 3 main stages: bootstrapping, bootstrap tracking and tracking. The most interesting for extraction stage is bootstrap tracking logic. Firstly, it computes optical flow with *CvInvoke.CalcOpticalFlowPyrLK*, then it validates if enough key points survived optical flow. Homography validation follows next to check how many key points at the current step survived in comparison with the initial value, if this value fails threshold, the process is considered to be failed. Secondly, stereo vision algorithms used. It estimates the rigid transformation with

*CvInvoke.EstimateRigidTransform* to validate camera motion between bootstrapped image and the current one. If the motion is sufficient it performs points triangulation. It is very interesting process, implemented in *OpenCvUtilities* class. It finds fundamental matrix with *CvInvoke.FindFundamentalMat* and checks how many key points survived. After this, essential matrix is computed from the intrinsic camera parameters and fundamental matrix. Essential matrix is decomposed with SVD decomposition and produces two rotation matrices and two translation vectors. These results are mixed into four possible projection matrices. To check what projection matrix is the right one triangulation procedure is run until the matrix is found. This process is also very important, at this point 3D points are projected back to the image and re-projection error is found. It helps to understand how many points survived and successfully were triangulated thus can be used for tracking. Finding 3D points is important because the aim is to find information about face plane in 3D space. With the help of **PCA!** using *CvInvoke.PCACompute* it finds eigenvectors and mean of the known 3D point matrix. Afterwards, **PCA!** of *Accord* library is used because *EmguCV* does not return eigenvalues. Retrieved data is used to find planes normal and its inliers, 3D points that lie in the same plane. Described analyses is what was needed for extraction logic.

### Tracking

PTAM like approach makes tracking possible. In comparison with bootstrap tracking of a plane simple tracking is done much easier. It also uses optical flow with the same code base as in the previous step. The difference is that it does not need to triangulate points again, it uses already found 3D point and their 2D position on the current image.

### Mapping

Extracted colors should be mapped to the cube model in order to solve and render it. Data is mapped to *ScrarchEngine.Libraries.RubiksCube.Models.RubiksCubeModel* class that represents a Rubik's cube in the project. This model has multiple methods for rotating cube's layers, flipping it or checking for cubes consistency. The mapping also includes face orientation analyses implemented through the iterative face orientation checkup. Faces positions are simply determined on the basis of their central pixel relation.

### Solving

Beginner solving algorithm was implemented to solve any 3 by 3 Rubik's cube. *BeginnerSolver* class is inherited from *BaseSolver* thus new solver can be easily

created and applied. To simplify and replace solution hard-coding declarative JavaScript Object Notation (JSON) structure was created. Solution formulas now could be written in JSON where the name of the formula, its phase, state that should be to run the formula and moves written in specific globally known notation.

```
[
  {
      "Name": "Up Front cubie on right|front",
      "Phase": "FirstCross",
      "IsFinal": true,
      "StateFrom": {
        "Right": [
          [ null, null, null ],
          [ "Up", null, null ],
          [ null, null, null ]
        ],
        "Front": [
          [ null, null, null ],
          [ null, null, "Front" ],
          [ null, null, null ]
        ]
      },
      "Moves": "F"
    },
]
```

Figure 55: Example of json formula declaration.

As can be seen from the listing 55 state is declared as faces where at some positions should be specified face names whose color is on this face. For instance, from the above listing, Up-Front cubie is on the Right-Front sides thus it is written like this and if the cube front layer will be rotated clockwise the cubie will be placed where it should be. $IsFinal$ flag saying that this algorithm puts cubie in its place and no more formulas for it should be used. In other case, the solver will run formulas until the cubie is in place. On the other hand, this JSON notation is not yet perfect, since algorithms are for humans, cube must be flipped and rotated to run formulas when handling cube at some specific position. Currently, it is necessary to write some code within the solver class inherited from $BaseSolver$ for specific algorithm stages or flips, they cannot be reflected in JSON yet.

### Rendering

To render the cube self-written software rendering engine oriented on cube was used. $RenderingControl$ was created with inheritance from $UserControl$. This

class handles rendering and mouse handling split into partial classes. This class encapsulates rendering and updating loops with abstract $Update(Action < IEnumerable < T >> set)$ and $Render(Graphicsg, IEnumerable < T > frame)$ methods used to write, basically, the logic of the game. Rendering control allows to zoom and rotate displayed image. To render the cube $RubiksCubeModel$ is split into cubies, imitating real puzzle pieces and each of them is converted to the 3D cube while only visible pieces are colored, others are black. Every cube is rotated with respect to its position, it means layers containing it. These manipulations are performed in $GenerateCubes3D$ and the result is a list of 3D cubes with specific position and rotation in 3D. Afterwards, cubes are projected on the screen and polygons are returned. At each step of update loop described transformation code is performed to translate $RubiksCubeModel$ to the list of polygons represented as $Face3D$ in order to be rendered with render loop.

## 4.3 Validation

Design and implementation details were described in the previous two sections. Theoretical ideas and its implementation with specific technologies were covered. Now its time to describe results achieved through the long way of research.

### Extraction

Extraction was the most challenging stage of the whole process. Contours usage to find a face of a cube showed good results if the illumination condition are sufficient enough. Contours detection is very prone to the influence of illumination. If the light is falling on glossy Rubik's cube surface at some angles detected contours might be broken and this will fail face detection for this specific image. On the contrary, PTAM like approach tracks key points with optical flow and is also affected by illumination but is also influenced by the smoothness of motion. Both approaches can be used to succeed with extraction but it may take some time, maybe even too much for a regular user. Contours approach was chosen as more sufficient approach because of multiple reasons. The first one, it could be even applied to a limited amount of separate images. The reason for this is that contours detection does not require any motion but it needs the image with quality sufficient enough to find all needed contours. The second reason to choose it was detection speed since to extract a face algorithm needs to capture only one image with sufficient state.

From the other side, PTAM requires to capture multiple faces in order to get their intersections which are cube's corners. This has not given wanted results because PTAM runs bootstrap tracking to analyze motion and get plane information; this operation should be performed per each plane (e.g. face). To find cube's

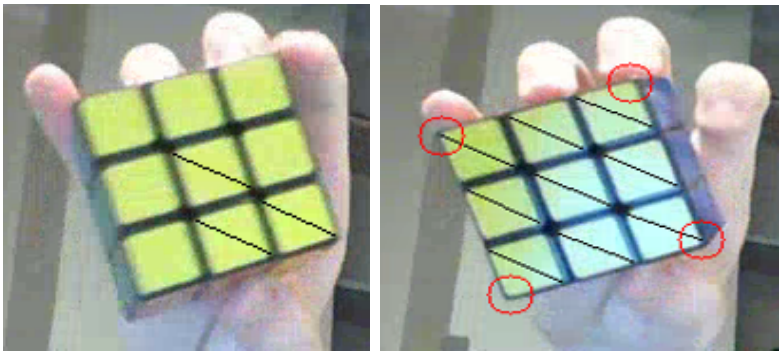

Figure 56: Contours approach face extraction process.

corners that will have position with respect to each other system should compute rotations and translation from the first successfully detected plane which is zero point for this relative system. If algorithm fails in the middle everything should be started from scratch. Even though, it is possible to store the current system state it have not given any results at the current stage.

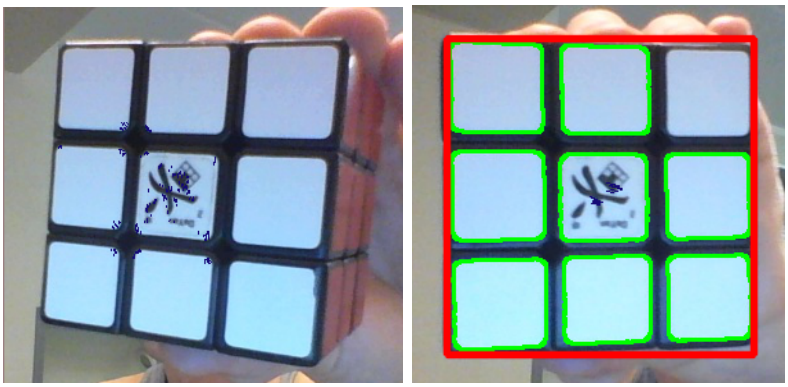

Figure 57: PTAM approach face extraction process.

In addition, extracted faces were checked if they were already seen before with the histogram comparison. If the face is approximately taking the same amount of space in an image then the correlation is pretty high and the code produces almost 100 percents success rate. Sometimes, illumination conditions distort face colors though two different colors might be considered to be the same. In this case one face will never be confirmed and the process will never stop because it is waiting for 6 unique face to be detected in order to go further. To conclude, from a video perspective, contours approach can be used with any video while all faces

detected, success rate is close to 70 percents. PTAM gives results close to zero, the only chance to get at least somewhat results is to use real time video when operator can control camera motion per each face but not randomly rotated cube in the video.

### Tracking

In comparison with extraction, tracking task is much more suitable for PTAM like approach. Since continuous faces detection has been failed a cube cannot be tracked with this approach. From the other point of view, cube rotation can be tracked only using single face that can be easily done with PTAM. Basically, PTAM like approach that has been implemented is not suitable for this because it cannot remember tracked key points when they disappear from a camera view. With the help of modified PTAM like approach and Kalman filter it can be possible to simulate cube rotation tracking by a static camera capturing smoothly rotating cube in front of it. Stereo vision practices used in PTAM will think the camera is rotating around the object thus rotation and translation data can be extracted and changed in order to be cubes rotation and translation representations. Kalman could be used to predict tracked key points position when they fade from a camera view and when the camera sees them again Kalman will update its state. This will allow to render 3D model rotation on a screen very smoothly and continuously. Described techniques are theoretically researched and proved to be possible.

### Rendering

Software rendering was successfully implemented with the described in 4.1 approach. Basically it showed pretty good performance for 3D Rubik's cube model rendering, full 30 Frames Per Second (FPS). Model can be zoomed and rotated without any performance drops. Unfortunately, software rendering cannot be used to render complex 3D models from thousands of polygons or to render a scene because it will not be possible to keep sufficient FPS. The reason is that GPU is not used, everything is done by CPU. Since the project is a framework, rendering library can be replaced with any other, for instance, using Open Toolkit library (OpenTK) which is the most popular .NET OpenGL wrapper.

### Mapping and Solving

Mapping and solving parts are tightly coupled, both of them performed well. Basically, mapping is a stage where extracted cubes faces data should be analyzed and mapped to the correspondent positions. On the other hand, mapping is also storing the data from one data structure to another with a challenge to design such a data structure that will be suitable for usage with both stages solving and

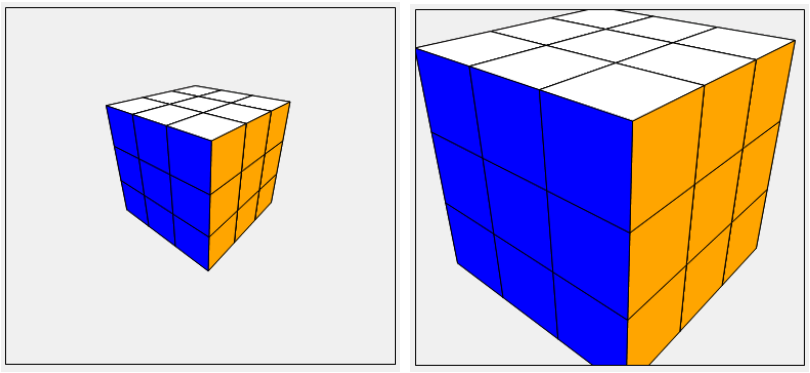

Figure 58: Rendered cube model.

rendering. This data should be able to allow cube's layers rotation and changing cube's orientation. All the goals were accomplished thus all basic Rubik's cube operations can be performed with the model. Solution adapters were designed to operate with the model from the mapping stage. Developed JSON declarative notation appeared to be a very convenient tool to notate large amount of formulas in text files and the code base become a dozen times smaller or even a hundred times. Unfortunately, Rubik's cube human orientated solution algorithms may contain such manipulations that cannot be easily described within the designed notation. Nevertheless, the current implementation showed 100 percents success rate.

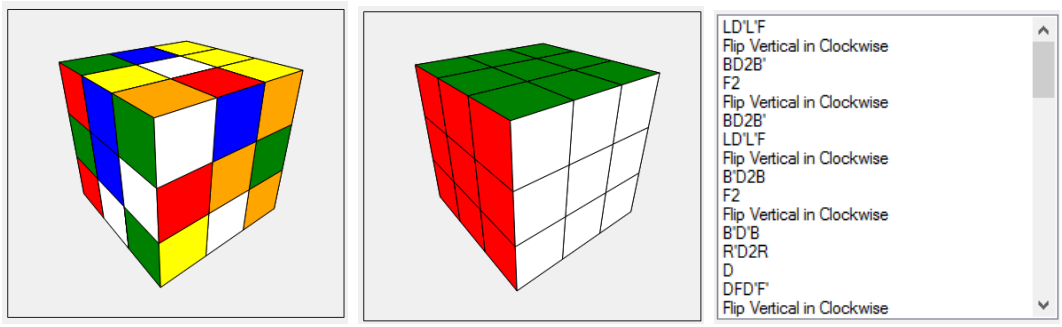


Figure 59: Solving random cube example.

# 5 Conclusion

The current project is representing a deep research in the field of computer vision with its complex algorithms and stereo vision concepts. Kalman filter was analyzed for suitability to the current problem. The program in C# was developed, it can extract data about a Rubik's cube, build cube's model, solve and render it. Even though the extraction part is not giving incredible results it works as expected.

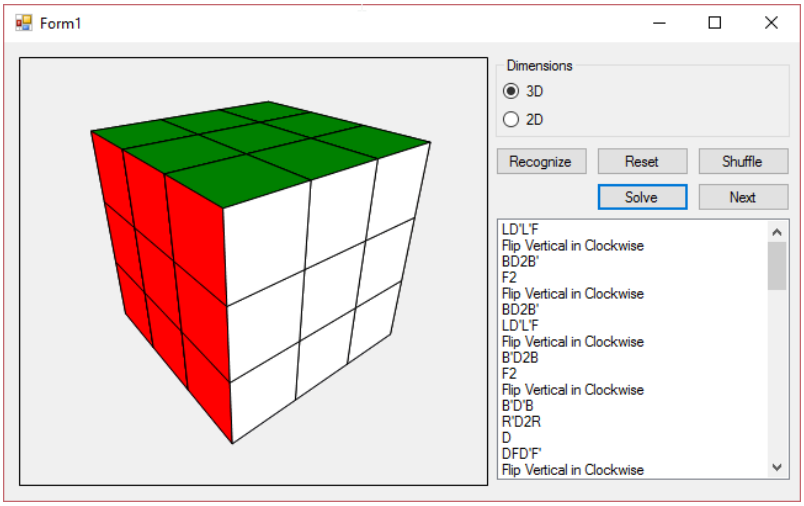


Figure 60: Window of the program.

As can be seen form the figure 60, the program has two buttons for solving. They are "Next" and "Solve", next button provide step by step solution displaying though user can solve the cube sitting in front of computer, solve button, on the contrary, tries to solve and display the result immediately, if the cube is correct and can be solved. There is a possibility to solve a cube using logged line by line solution formulas in the specific text area.

TEXT[2]
TEXT[see 2, p10]
TEXT[compare 5]
TEXT[e.g. 1, page 300]

# References


[1] Albert Einstein. "Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]". In: *Annalen der Physik* 322.10 (1905), pp. 891–921. DOI: http://dx.doi.org/10.1002/andp.19053221004.

[2] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Reading, Massachusetts: Addison-Wesley, 1993.

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

[4] Rudolph Emil Kalman. "A New Approach to Linear Filtering and Prediction Problems". In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D (1960), pp. 35–45.

[5] Donald Knuth. *Knuth: Computers and Typesetting*. 1984. URL: http://www-cs-faculty.stanford.edu/~uno/abcde.html.

[6] A. Rosenfeld and J. Pfaltz. "Sequential operations in digital picture processing." In: *Journal of the Association for Computing Machinery* (1966), pp. 471–494. DOI: http://pageperso.lif.univ-mrs.fr/~edouard.thiel/rech/1966-rosenfeld-pfaltz.pdf.

[7] Richard Szeliski. *Computer Vision: Algorithms and Applications*. 2010.