**fml** –Lehrstuhl für Fördertechnik Materialfluss Logistik
Prof. Dr.-Ing. Dipl.-Wi.-Ing. W. A. Günthner
Technische Universität München

Interdisciplinary Project

# Efficient marker-based localization of ground conveyors with a camera-enabled Raspberry Pi

Grzegorz Wedler

Matr.-Nr.: 3646062

19.05.2016

Betreuer Lehrstuhl **fml**:

Dipl.-Inf. Matthias Jung

# Contents

# 1 Introduction

Intralogistics is a backbone of modern Supply Chain in field of logistics [Gün-2015]. Importance of intralogistics grows steadily as Supply Chain management must develop more efficient processes to increase competitiveness. Production of custom based products results in more complex production processes. Thus automatic systems to assist surveillance of control processes in real time are necessity [Gün-2015]. Forklift truck is being operated mainly by an employee and this results in high proportion of manual work. In modern logistics this is not cost efficient, because human errors can cause among others delays, collisions, wrong goods pickup. The “Staplerauge“ is a project developed at the faculty of Fördertechnik materialfluss Logistik (fml) der Fakultät Machinenwesen in TUM. Aim of the projects is to develop a set of sensor functions, which eliminate previous mentioned errors to some extent by:

- Identification of loaded goods
- Identification of load state
- Identification of forklift truck position in space at loading and unloading

This project focuses on forklift truck location processing. Sensor process must ensure that forklift truck arrives at correct location to load or unload goods. Not only position of vehicle must be detected, but also its correct orientation in relation to warehouse alignment [Gün-2015]. As a result of this computation, errors of picking wrong goods or unloading in wrong location can be eradicated.

## 1.1 Identification of the problem

The use of fast computers and industrial cameras is a huge hurdle in the production environment. Two most important issues are price of this components and their power consumption which should be kept as low as possible. Implementation of this project on embedded platform like Raspberry Pi almost eliminates above mentioned disadvantages. Raspberry Pi can be also integrated into infrastructure of existing fork-lift truck easily. The overall performance of this embedded solution is disadvantageous on the other hand. This project seeks and evaluates a possible solution to this problem.

## 1.2 Baseline solution

Identification of markers in the image is being done with the use of “ArUco“ library, which is a “minimal C++ library for detection of Augmented Reality markers based on OpenCV (open source compute vision) exclusively“ [Aru-2015]. There is a set of unique coded markers attached to the ceiling of fork-lift truck operation area. The library detects borders of the markers and analyzes inner region of those contours for a valid code. Main advantages for using ArUco library are [Gün-2015]:

- compatibility with OpenCV
- high detection rate
- high detection security
- sufficient documentation and open source

There is a set of procedures which need to be executed for every frame in order to find suitable marker regions. Two the most important ones, which are being executed on a CPU of raspberry Pi with the use of OpenCV library are: edge detection (Canny filter) and a procedure, which finds all contours in result of previous pass. After measuring execution time of this approach it has become clear that these two steps in the program are the most important bottlenecks in the pipeline.

## 1.3 State of the art

The concept of marker extraction consists of several steps. First of all image is being captured by camera attached to the embedded system.

In next steps image must be processed. Depending on project needs one might need to convert image to black and white color space or enhance colors in image by applying white balancing. Image is then ready to extract edges. There is a number of available algorithms like Canny detection or adaptive threshold method. Once edges are detected image

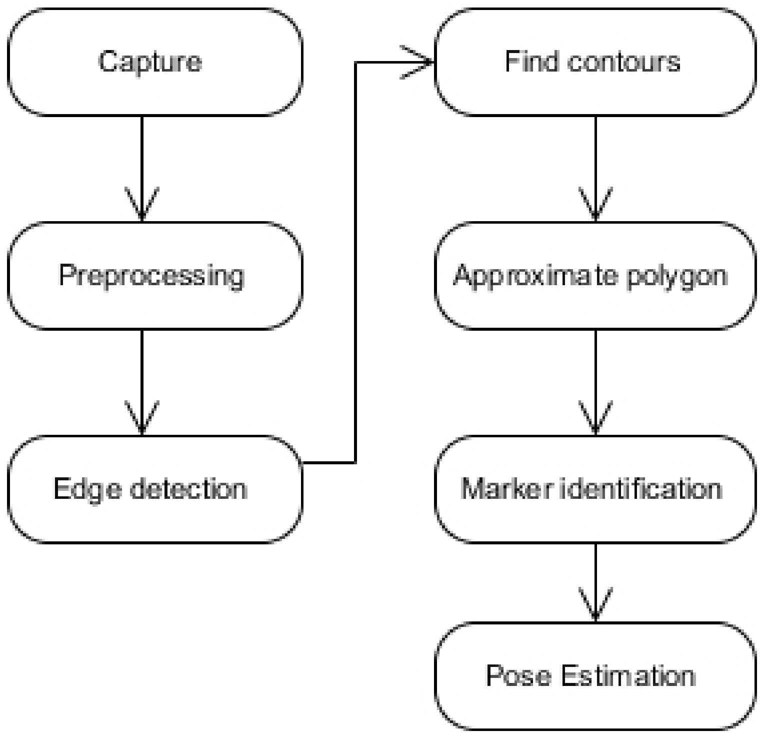


*Figure 1-1: Marker detection pipeline*

contains reduced number of information, which improve further processing performance. In final steps one need to approximate polygons where marker candidates may occur and identify markers within those candidates. Once marker is identified pipeline estimates pose in relation to the environment where device operates.

Shah in [Sha-2014] implements marker detection on variety of embedded platforms for further use in aviation industry, namely unmanned aerial systems (UAS). One of the important functionality aspects of such drone is to turn on autopilot in case of emergency and find safe spot to land. In the project a marker on the ground is used to determine safe landing zone. Author uses two platforms to test his solution, BeagleBone Black and Odroid XU. The BeagleBone platform is similar to the low cost Raspberry Pi solution, nevertheless it equipped with superior CPU than our platform. It has not only faster CPU but also contains NEON floating-point accelerator, which is being later utilized by the author. Processing pipeline is similar to ArUco library pipeline, it starts with few filters like Median Blur and Canny edge detection. Then the software finds contours in the image and approximates convex squares. All these steps are being realized with the use of OpenCV library. The library has been compiled with special set of additional compilation flags, which enable “advanced SIMD instruction set“ (NEON) [Sha-2014]. Author deleted one unnecessary step from his pipeline and used NEON where possible. Shah reports performance of about 5.8 frames per second, which is two times improvement over baseline implementation. Performance on Odroid XU is far more superior (30 fps) because of 8 core CPU.

Markus Konrad in [Kon-2015] uses OpenGL ES 2.0 library to use graphics processing units in mobile phones as general purpose processing units. Author determines that image processing takes most of the execution time on the hardware. This part of the process consists of scaling of image and black and white conversion. Other part of the pipeline, which Konrad implements on the GPU is identification of the marker. Nevertheless important in this project OpenCV function “findContorus“ is being executed on the CPU. Author notices up to 3.2 times performance speedup.

### 1.3.1 Possible speedup approaches

Plenty of general purpose GPU programming APIs are available on desktop platforms. Proprietary API from Nvidia “CUDA“ or “DirectCompute“ are not accessible on mobile devices [Kon-2015]. The most promising GPGPU technology for mobile devices or embedded solutions is Open Computing Language (OpenCL), because it is a generic framework with support of wide variety of devices. Nevertheless Raspberry Pi lacks implementation of OpenCL [Ras-2016]. Choice of selecting suitable GPGPU programming API is narrowed to use of its graphics pipeline by means of OpenGL [Ras-2016].

## 1.4 Parallel computing on Raspberry Pi

Michael J. Flynn created its own taxonomy which describes computer architectures based on their instruction and data streams [Raj-2012]. One of architectures is single instruction, multiple data stream (SIMD) in which computer execute simultaneously same instruction on different data. Most of the computer chips contain some logic which enables computation in SIMD fashion. Furthermore all graphics processing units can also be characterized by SIMD, because they execute same instruction on many elements in the GPU memory. Raspberry Pi has a single-core ARM1176JF CPU running with a clock speed at 700 MHz [Ras-2016]. This particular version of ARM chip does not contain the NEON instruction set, which confirms that there is no SIMD unit on the chip one can use for general purpose. As a consequence of this, the code running on the CPU cannot be paralleled in any way.

Other feature of Raspberry Pi is its graphics card, the Broadcom VideoCore IV. This chip is compatible with OpenGL ES 2.0 library and this creates some room for parallelism on the Raspberry Pi. With the use of programmable pipeline in OpenGL ES 2.0 one can execute some general purpose computing on raspberry Pi graphics unit.

Concept of computations in OpenGL ES is realized within so called “shaders“ [Mun-2009]. Shaders are small programs executed along graphics pipeline written in OpenGL Shading Language.

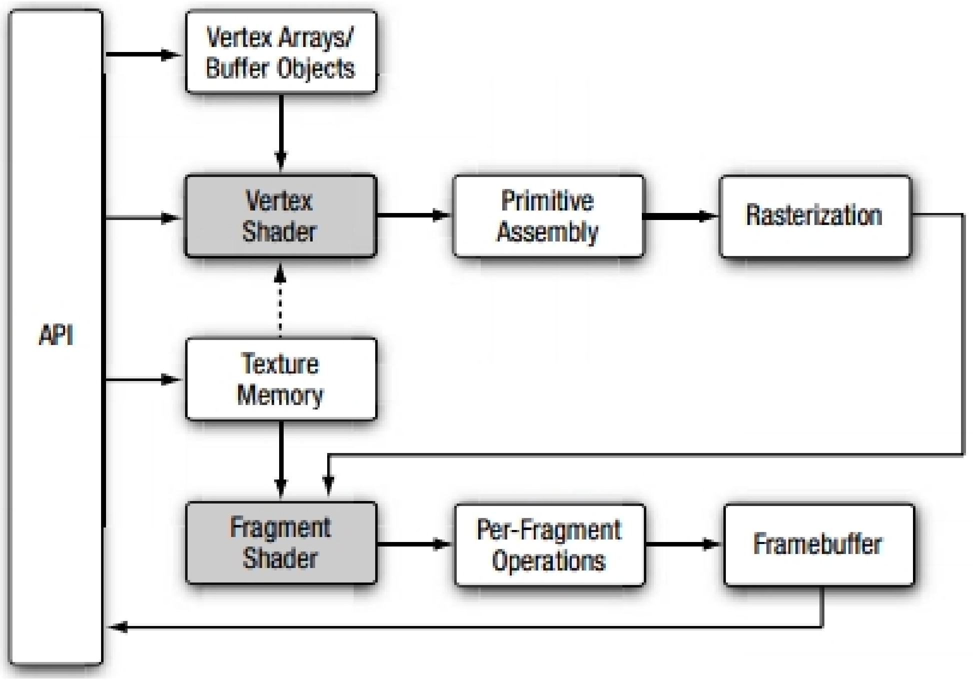


*Figure 1-2: OpenGL ES 2.0 render pipeline [Kon-2015]*

Two shaded regions in the OpenGL ES render pipeline are programmable, namely Vertex Shader and Fragment Shader. Vertex Shader stage is being used to compute vertex-based operations (operations on points in 3D space) such as translation of position, computation of lightning equation, transformation of texture coordinates [Mun-2009]. Fragment Shader is the method to compute fragments. Fragments are defined by individual pixels in the graphics pipeline texture object. Within fragment state one usually compute color value of the pixel. In general the pipeline executes fragment shader for each point (pixel) in the texture. It is not always true in case of vertex shader. In order to copy data to OpenGL pipeline one must use textures. Function “glTexImage2D“ defined in the OpenGL API stores data in GPU memory. To read the data back from the device one calls “glReadPixels“ function.

OpenGL ES is a subset of desktop OpenGL interface and there are some limitations as a consequence of this. The true GPGPU computing capabilities are not implemented within OpenGL ES 2.0 like “compute shaders“ [Kon-2014]. Vertex and Fragment shaders can be used to implement some relatively “lightweight“ functions. As a result of this one can apply fine grain parallelism, execute as small programs as possible for each data point in data-set. Passing data comes also with some limitation. Framebuffer in OpenGL ES 2.0 implementation on Raspberry Pi does not support “GL_LUMINANCE“ data format which contains single channel value. As a result of this one have to pass three channel points into the memory, thus memory transfer from GPU is decreased as function call handles more data. Another drawback is that texture data type is an 8 bit (unsigned char) type which encodes 256 values.

Raspberry Pi CPU has only one core, thus there is no possibility to speed up the software by the means of multithreading. OpenGL ES 2.0 implementation on the embedded device creates some possibilities to speed up the execution of program. Nevertheless, this model is highly limited and can be optimally used for computations based on pixels.

### 1.4.1 Parallelism of Marker detection

With regard to Marker detection pipeline, there exist three steps within the process which are suitable for GPGPU implementation on Raspberry Pi. Depending on algorithm used in preprocessing stage, it is advisable to execute this part in graphics unit. Reason for this is that most of the algorithms perform some computation on pixels of the input image. Edge detection is a good candidate for GPU implementation because of above mentioned argument. What is more, edge detection is often very expensive to compute. It is reasonable to implement contours detection on GPU as well, because of limited CPU performance on Raspberry Pi.

Rest of pipeline steps in the CPU section either not possible to improve (via GPGPU) like image capture, or makes no sense to implement on GPU whatsoever. As an example, identification of markers is a relatively straightforward routine, which operates on small subset of pixels within image [Aru-2015]. OpenGL overhead is too big in this instance.

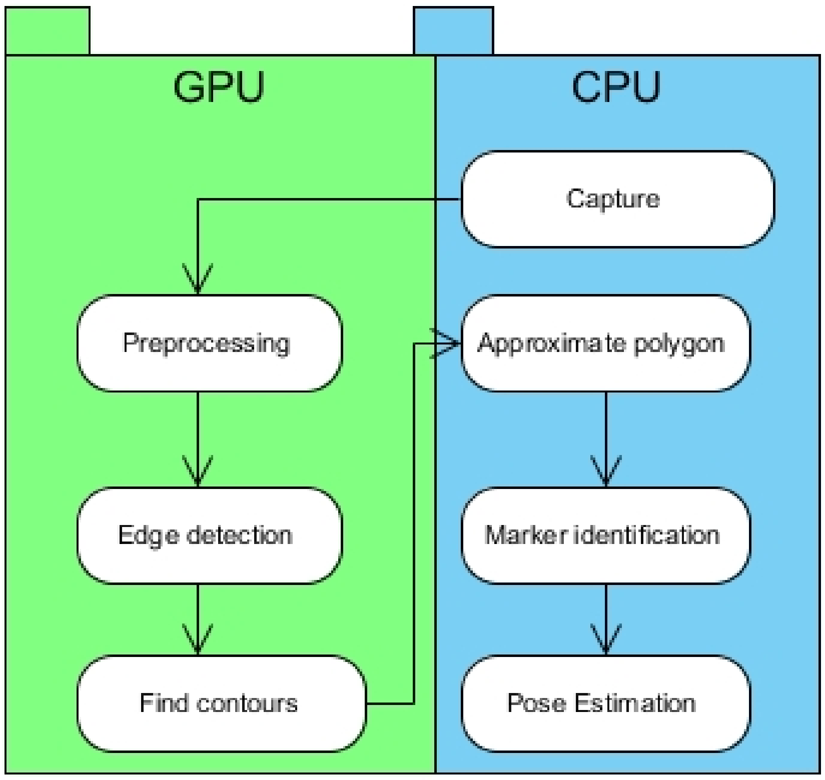


*Figure 1-3: Modified Marker detection pipeline*

## 1.5 Possible GPU based solutions

The following two algorithms are essential in the ArUco library pipeline. Canny edge detection has proven to be reliable and robust [Can-1986]. What is more, the algorithm can be implemented on the GPU in SIMD fashion. My tests revealed, that Raspberry Pi CPU needs relatively high amount of time (about 300ms) to execute OpenCV implementation of Canny detection. Motivation to implement edge detection on GPU is justified by this slow process. Following algorithm is Hough transform, which is being used instead of OpenCV function “findContours“. This step within the pipeline is another big bottleneck and there exists possibility to implement this procedure in GPU as well.

### 1.5.1 Edge detection

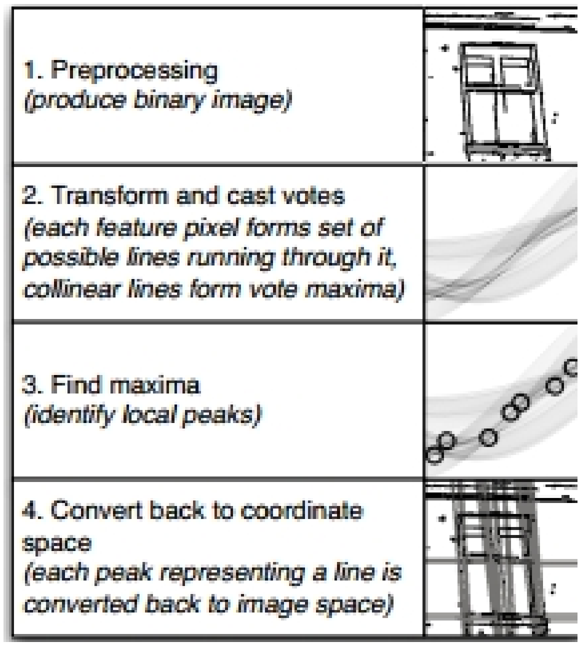
Algorithms to detect edges in image are very important in many computer vision systems [Can-1986]. By the means of edge detection one can simplify image for further processing, by deletion of unnecessary data. Edge detector is essential step in marker detection pipeline for augmented reality and approach proposed by John Canny can be implemented on GPU in SIMD fashion. Canny algorithm is decomposed into four computational steps [Kun-2016]. Firstly one applies a Gaussian blur in order to obtain slightly blurred version of original image. To do this a 3x3 or 5x5 mask is being executed over the input image. Each pixel in image equals the sum of pixels in its neighborhood times the corresponding Gaussian weight. This value is then divided by weight of the whole mask [Kun-2016]. In step two algorithm uses Sobel operators to get edge gradient strength and its direction for each pixel. This is applied in both x and y direction. Sum of Sobel mask in x direction times corresponding pixel is set as Gx. Gy value is a similar result computed in y direction. Edge strength is defined by square root of Gx squared plus Gy squared. Edge direction equals inverse tangent of Gx divided over Gy. In next step algorithm iterates over all pixels in image and compares current pixel gradient strength with upper and lower threshold parameter in the algorithm. If gradient strength of that pixel is within a range defined by those parameters, pixel is set to white. Last step of the Canny edge detector is being employed to find strong edges and make them weaker.

### 1.5.2 Hough Transform

The Hough Transform is a method to processes image data from Cartesian coordinate space into a polar parameter space [Zho-2014]. As a result of this, parameter space is used as accumulator array [Zho-2014]. Purpose of this mapping is to find geometric lines and other shapes in input image. Image with detected edges from previous step in the pipeline is an input for Hough Transform.

$$p = x * cos(\theta) + y * sin(\theta), \tag{1}$$

For each pixel in the image which belongs to edge algorithm maps x-y Cartesian coordinate to polar space with the use of equation 1. Map of polar coordinates serves as sort of accumulator where for range of angles t (from 0 to 180 degrees) each possible “line“ stores a vote. Ensemble of “sinusoids“ in plane ($\theta$p) intersect at one point and the highest value can be taken as line candidate in Cartesian space. Mapping from ($\theta$p) space back to original space is called Hough Transformation.



1. Preprocessing
*(produce binary image)*

2. Transform and cast votes
*(each feature pixel forms set of possible lines running through it, collinear lines form vote maxima)*

3. Find maxima
*(identify local peaks)*

4. Convert back to coordinate space
*(each peak representing a line is converted back to image space)*

*Figure 1-4: Hough Transform pipeline [Kon-2014]*

# 2 Implementation

I this section I describe technical approach to how implement the concept of Marker detection on Raspberry Pi. The solution consists of custom framework, which takes care of necessary OpenGL calls and prepares all mandatory objects. It is a backbone of the GPGPU solution. Furthermore approach to implementation of edge detection and Hough Transform is explained.

## 2.1 RaspiCam library for Raspberry Pi camera module

Raspicam is a C++ library for Raspberry Pi Camera module [Ras-2015]. The main advantages of choosing this library is that it is compatible with OpenCV image interfaces. The library provides full control of the camera in embedded device. It is possible to alter some properties of camera (controlled automatically) like camera gain or exposure. My tests have shown, that letting the camera driver set some properties automatically in dynamically changing light conditions is not always the most optimal way to proceed.

## 2.2 Raspi_gpgpu engine. Dynamic OpenGL programs

RaspiGPGPU is a small library created for this project, which creates OpenGL programs on the fly and executes them in pipeline. Each GPU program needs to execute some OpenGL API calls and set some properties in order to execute properly. The main structure is as follows:

Class “ShaderFactory“ manages Vertex and Shader programs. It creates, compiles the programs. In the next step the compiled software is being assigned into main OpenGL program. Class “MemoryTransfer“ maintains all memory transfers from and to memory in graphics processing unit. Each GPU program has its input and output “TextureID“. First shader in pipeline reads OpenCV image as input. Following shaders in pipeline read data set as output by previous program. When last shader finishes its execution, data is being read back from GPU memory.

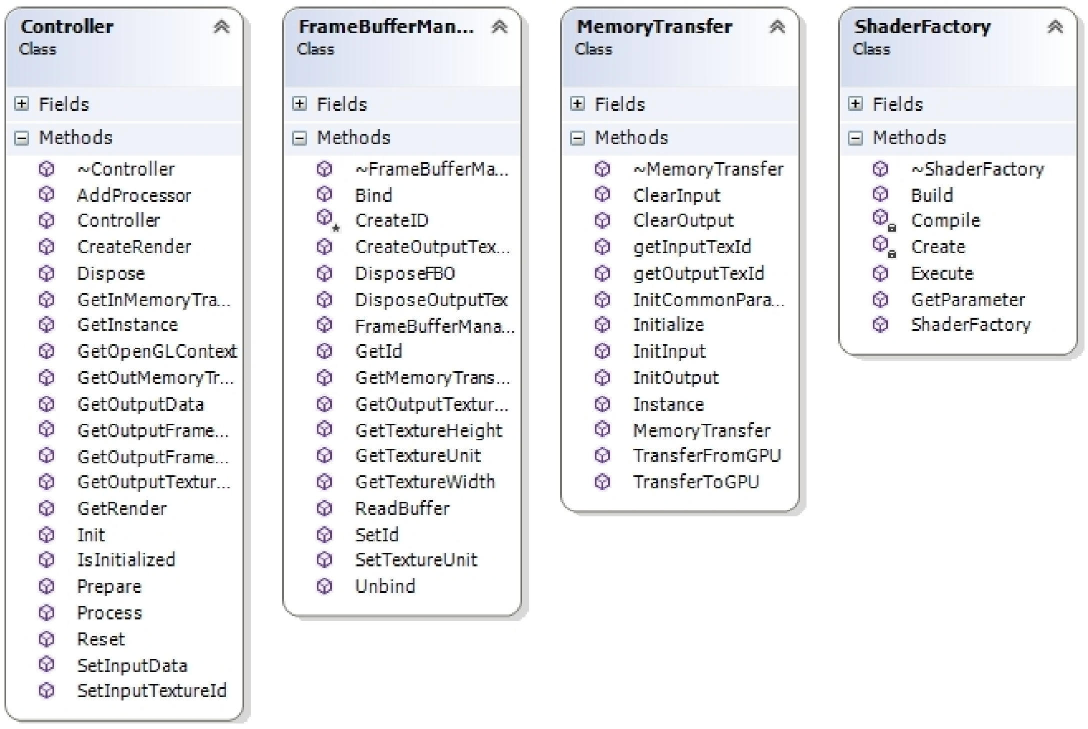


*Figure 2-1: OpenGL framework*

Frame buffer object (FBO) is a “2D image buffer allocated by the application“ [Mun-2009]. Since the project tries to use OpenGL as GPGPU unit, one have to draw/compute pixels to FBO instead of to the screen. Class “FrameBufferManager“ manages FBO in OpenGL context. Additionally it sets some important texture parameters. Class “Controller“ manages programs all in the pipeline. It adds GPU programs into a queue for further execution. Among others it passes result data to other modules.

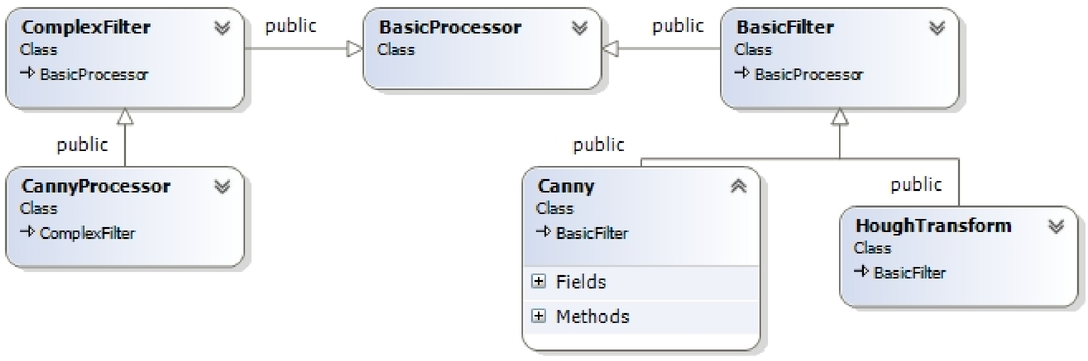


*Figure 2-2: OpenGL Processor framework*

Base class “BasicProcessor“ is a core interface in raspi_gpu framework. It contains handles to main framework objects like “ShaderFactory“ or “FrameBufferManager“. This software sets some basic properties like width or height of image as well as texture coordinates needed in vertex shaders. “BasicFilter“ and “ComplexFilter“ inherit “BasicProcessor“

class and are used as higher level wrappers for OpenGL programs. “HoughTransform“ and “Canny“ classes are higher level generalization of BasicProcessor and contain source code used in fragment shaders.

## 2.3 Canny Edge on GPU

Implementation of Canny Edge in library “GPUImage“ has been used to find contours in the image [Lar-2016]. There are some differences in this implementation in compare with original idea of Canny. Gaussian smoothing is carried out by 3x3 convolution kernel. As blurring the image in x and y direction is finished, gradient vector is calculated from nine neighboring pixels by Sobel X and Y operators. A mix of sign and step functions is used to determine primary direction of the gradient vector [Ens-2011]. Next pass executes non-maximal suppression by getting the length of the gradient vector and length of the gradient vector of two pixels in directions ($-\Delta x, -\Delta y$), ($\Delta x, \Delta y$). Length at the pixel is either 1 or 0 as a result of step function. Furthermore this length is then compared with maximum of two neighboring lengths. Two step threshold is realized by function “smoothstep“. Final shader in Canny pipeline removes weak edges from image. Idea of weak pixel, if eight neighboring pixels are strong has some disadvantages, because it requires multiple “if“ statements in code. In order to optimize the execution of this shader, the pixel gets nine neighboring pixel values. Then it “takes a linear combination of the edge strength measurement at the pixel with a step function that accepts a weak pixel if the sum of the nine edge strength measurements is at least 2.0“ [Ens-2011].

## 2.4 Hough Transform on GPU

Following figure demonstrates practical implementation of Hough Transform.

Input image in this example has size of n x n. Any line can be determined by perpendicular to it line, characterized by parameter p and angle $\theta$ coming out from the center of the image. Thus coordinates in both directions are in range from $[-n/2 + 1, n/2]$. Each pixel, which belongs to strong edge is mapped to $\theta$p space. This mapping is realized in Vertex shader in “basic_filter.cpp“. For each pixel, intensity threshold is evaluated. If pixel is “bright“ enough, its coordinates are mapped into $\theta$p space. Resulting coordinates in space are then mapped by OpenGL framework to corresponding pixel in Pixel Shader. Pixel shader increments pixel

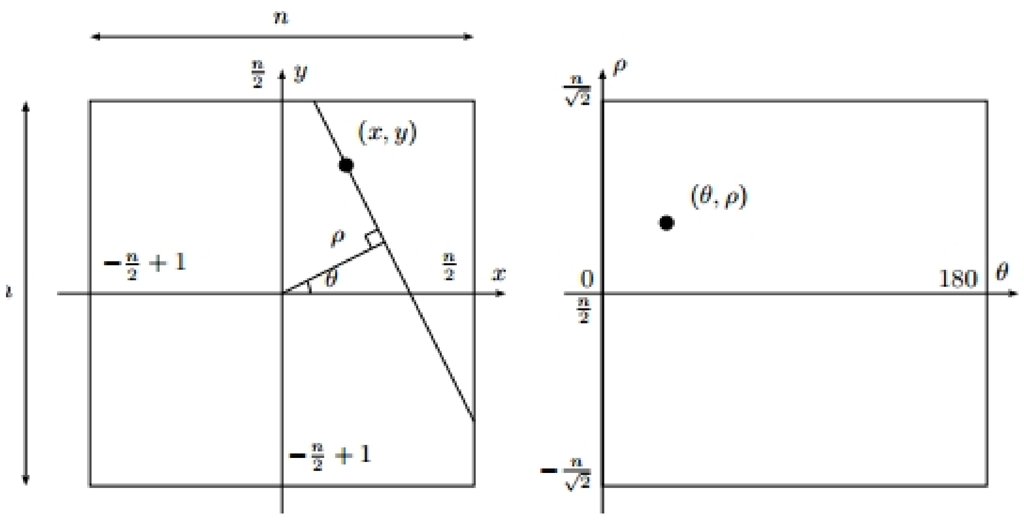


*Figure 2-3: Hough Transform [Zho-2014]*

brightness by $1/256$. This means that 256 votes for a pixel overflows the accumulator at a point. For marker detection this limit is not a huge issue, because edges of the markers rarely exceed value of 100.

### 2.4.1 Post processing

When HoughTransform OpenGL program finishes its execution the result texture is copied back from GPU memory.

My approach to interpret the results of Hough space is to apply a threshold. An upper threshold $t1 \in N$ and lower threshold $t2 \in N$. Line is detected if pixel value lies in the range set by two thresholds. If threshold is set to small, program detects too many lines. Consequently if threshold is set to high results in detecting too few lines or even none. Additionally a clustering is used to find local maximum in result texture. For each pixel a region around it is traversed to find more suitable line candidate. If a line candidate within a region has higher value than the current line candidate, current value is discarded.

Furthermore to find relevant lines which may belong to the markers one can apply a set of heuristics. Parallel lines in parameter space ($\theta$p) have same value of $\theta$. Each edge belonging to a marker is $\Delta$p away from its parallel neighbor in Hough space. Correct value of distance difference $\Delta$p is similar for common marker sizes and can be found by trial and error based on the test data. Post processing steps are executed on the CPU.

# 3 Evaluation

This section compares three different approaches to detect marker candidates in input images. First approach is a “Baseline OpenCV“ approach and marker detection logic is realized with the use of OpenCV framework only. Second approach is called “Hybrid“ and in this setup edges are detected on the GPU with Canny filter. Texture with edges is being read from GPU memory and contours detection is computed with the use of OpenCV framework. Third approach is “Raspi_gpu“ and in this approach detection of edges and contour lines is realized by the functions running on the GPU.

## 3.1 Setup

In the experiment a set of 52 images in resolution of 640 x 480 are being used to evaluate marker detection. Raspberry Pi was mounted to the upper part of the forklift and from there images have been captured. In each of the images a different amount of visible markers and other noticeable edges can be found. Forklift truck moved around a warehouse while images have been captured. During evaluation Raspberry Pi is overclocked to preset “Medium“, 900MHz CPU, 450MHz SDRAM, clock of the GPU unchanged.

## 3.2 Interpretation of lines

Interpretation of Hough space result is computed on the CPU. As described in section 2.4.1, lines are filtered by thresholds. Furthermore line candidates are compared against each other in their neighborhood. Figure 3-1 depicts an example where not optimal set of threshold and clustering parameters result in detection of high amount of lines. Amount of lines being detected has a direct influence in execution time of post processing phase.

On the other hand, as threshold range gets thinner, some lines, which belong to valid markers are filtered out. This situation is depicted by figure 3-2 and has a direct influence on accuracy of marker detection.

Finding the most optimal set of parameters to achieve best performance and accuracy is very difficult. The tighter threshold parameters are the shorter program execution is at the cost of marker detection accuracy. Although Hough Transform is one of the most robust line detection algorithms, thresholding is not advisable if a defined number of edges have to be found. Smaller lines are difficult to detect in presence of longer edges. Canny filter has proven to extract correct all edges belonging to the markers in all input images.

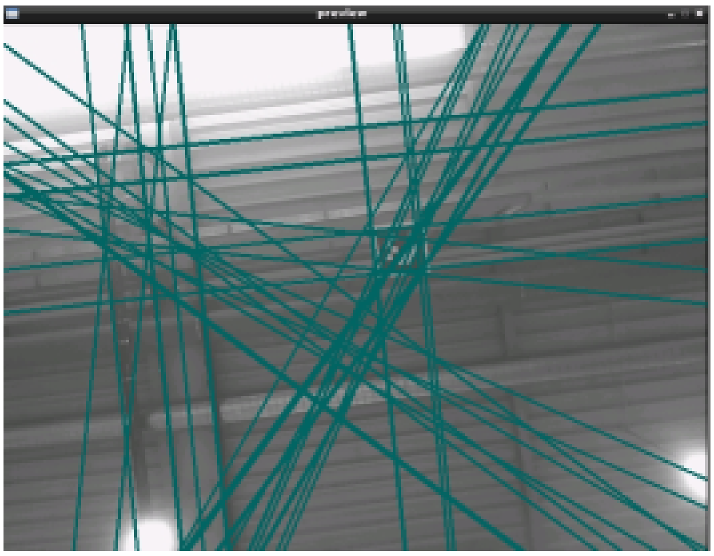


*Figure 3-1: Noisy lines set*

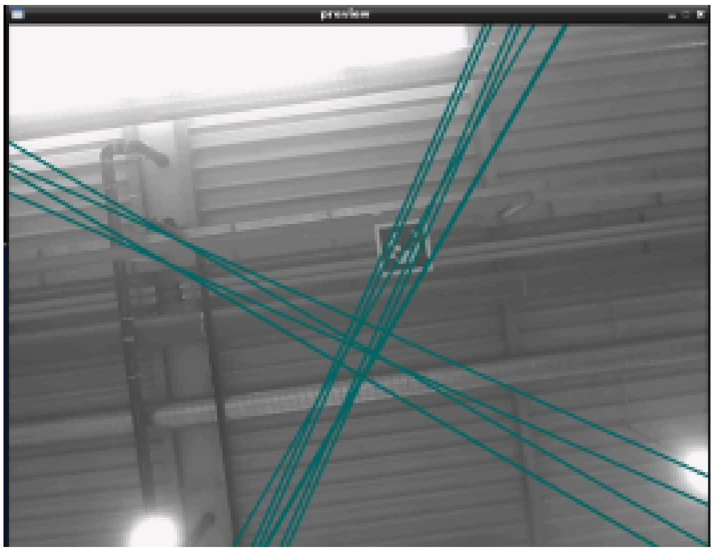


*Figure 3-2: Noisy lines set*

## 3.3 Execution time

Table 3-1 depicts run-time execution time in milliseconds of three above mentioned approaches. Baseline OpenCV approach needs about 118 milliseconds on average to compute. Difference between fastest and slowest run-time is about 40 ms and one can conclude that this approach is stable.

Hybrid approach delivers a speedup of about 31.5% on average in comparison to reference solution. Despite the fact that a result of Canny edge detection executed on the GPU has to be copied back from GPU memory. It is also the most stable solution in terms of time of execution. Difference between fastest and slowest image run-time was about 30 ms.

Raspi_gpu setup is slower than the reference solution run-time in case of average execution time. Discrepancy of the slowest and the fastest execution time equals about 460 ms and as a result of this the approach is less stable than other programs. At one frame max run-time equals 530 ms. In that particular frame algorithm detects 295 lines. Increase in amount of detected lines increases computational workload required to find marker candidates in the image.

*Table 3-1: Runtime in milliseconds*

| | Baseline OpenCV | Hybrid | Raspi_gpu |
|---|---|---|---|
| Average | 118,46 | 80,96 | 126,79 |
| Min | 100 | 70 | 70 |
| Max | 140 | 100 | 530 |

## 3.4 Accuracy

In the set of input images, markers are located in various parts of image. All three approaches compute location of polygons which are candidates of a marker. If the algorithm finds a polygon which encapsulates whole marker (and marker only), it is a valid marker candidate. All images in the data-set contain at least one marker, some contain as much as three. Figure 3-3 shows proportion of images where at least one valid marker candidate is detected. CPU based solution has proven to be quite robust in the test. It detects valid marker candidates in about 58% all images. Slightly behind Baseline approach is the Hybrid solution with accuracy of about 56%. Slightly better results regarding the accuracy is noted in case of the Raspi_gpu approach, where in about 62% of images a valid marker candidate can be found.

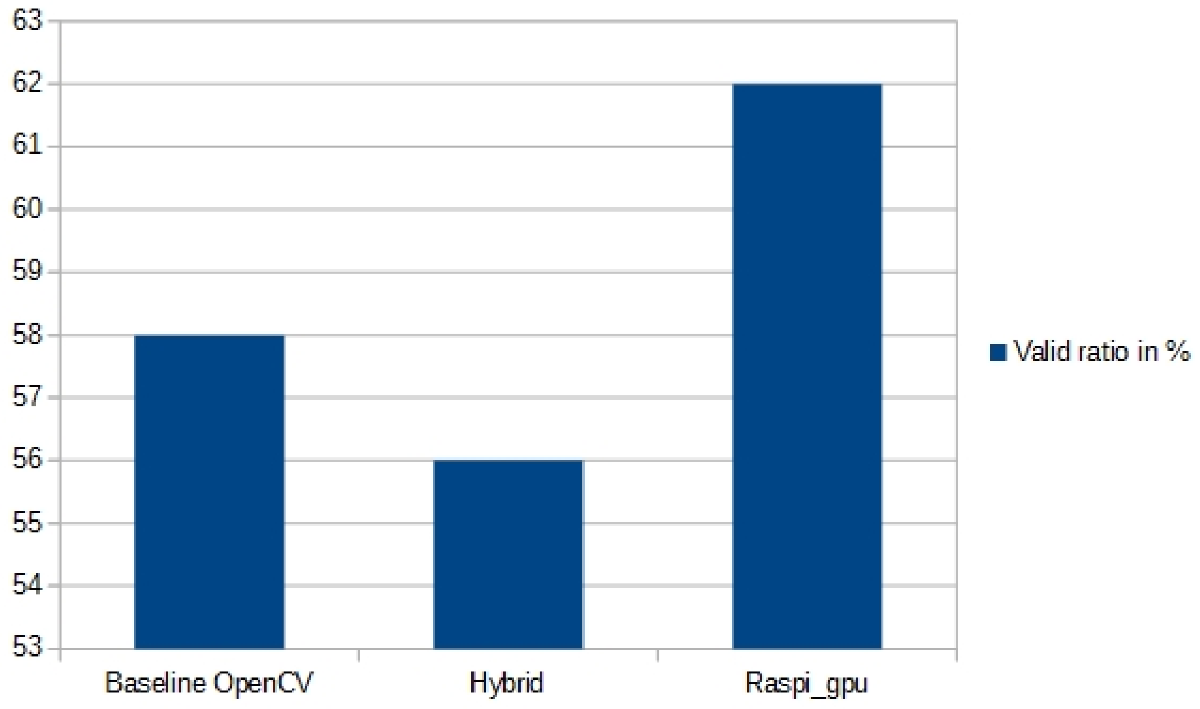


*Figure 3-3: Detection of valid marker candidates in relation to all images*

## 3.5 Conclusion

In this work a possible solution to marker detection on the embedded platform Raspberry Pi has been proposed. Performance of the solution may be characterized as acceptable, but is not as fast as initially expected. Overall performance and robustness of the Raspi_gpu solution is comparable to the performance and accuracy of reference solution. The proposed solution can be implemented on a wide variety of embedded solutions like Raspberry Pi which contribute to decrease of system costs in the production system.

# Bibliography

[Gün-2015] Günthner, W.A.;Hohenstein, F.;Jung, M.: Das Staplerauge, Forschungsbericht. 2015

[Aru-2015] ArUco: Augmented Reality library from the University of Cordoba. https://sourceforge.net/projects/aruco/files/1.3.0/README, Aufruf am 07.07.2015

[Sha-2014] Shah S.: Real-time Image Processing on Low cost Embedded Computers. 2014

[Kon-2015] Konrad, M.: Beschleunigung der Markererkennung einer Augmented Reality Anwendung auf mobilen Geräten mittels GPGPU. In: Knaut, M.: 16. Nachwuchswissenschaftlerkonferenz, Berlin, 2015, S. 362 - 369

[Raj-2012] Rajski, W.;Diede ,A.;Burri, D.;Oscar, N.: An Examination of the Current State of the Art in SIMD Processor Extensions for CS570 Winter 2012.

[Ras-2016] The Raspberry Pi Foundation. https://www.raspberrypi.org. 2016.

[Mun-2009] Munshi, A.;Ginsburg, D.;Shreiner, D.: OpenGL ES 2.0 Programming Guide. Addison-Wesley, 2009.

[Kon-2014] Konrad, M.: Parallel computing for digital Signal Processing on Mobile Device GPUs. Master's Thesis. Faculty of Business Sciences II, HTW Berlin, Berlin, 2014.

[Can-1986] Canny, J.: A computational approach to Edge Detection. 1986

[Kun-2016] Kuntz, N.: Canny Tutorial. http://www.pages.drexel.edu/ nk752/cannyTut2.html, Aufruf am 20.02.2016.

[Ens-2011] Ensor, A.; Hall, S.: GPU-based image analysis on mobile devices. 2011

[Zho-2014] Zhou, X.;Tomahou, N.;Ito, Y.;Nakano, K.: Implementations of the Hough Transform on the Embedded Multicore Processors. 2014

[Ras-2015] Raspicam. https://sourceforge.net/projects/raspicam/files/?, Aufruf am 20.11.2015.

[Lar-2016] Larson, B.: GPUImage. https://github.com/BradLarson/GPUImage , Aufruf am 15.01.2016.

# List of Figures

# List of Tables

## Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Arbeit selbständig angefertigt habe.

Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Diese Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

Garching, 19.05.2016

Grzegorz Wedler