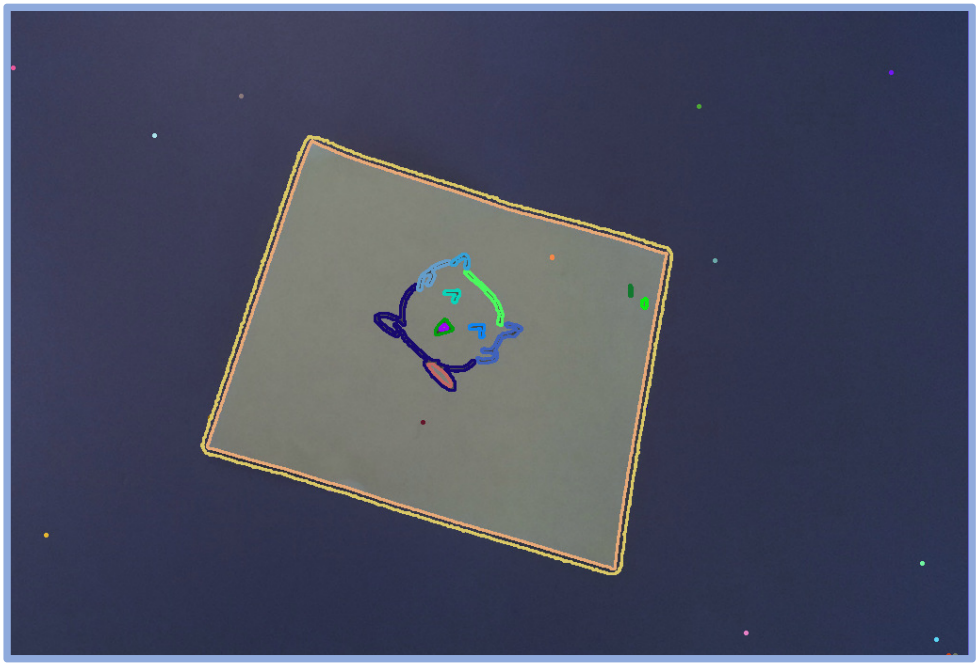
CHALMERS



Object detection and analysis using computer vision

- An open source development, implemented using Raspberry Pi

Bachelor thesis in Mechatronic Engineering

Victor Friedmann Sandin
Anna Thomsen

Department of Computer Science and Communication
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Bachelor thesis

Object detection and analysis using computer vision

- An open source development, implemented using Raspberry Pi

VICTOR FRIEDMANN SANDIN
ANNA THOMSEN

Examiner : Peter Lundin, Department of Computer Science and Engineering

Department of Computer Science and Communication
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Object detection and analysis using computer vision
An open source development, implemented using Raspberry Pi

VICTOR FRIEDMANN SANDIN, ANNA THOMSEN

Department of Computer Science and Communication
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Preface

This bachelor thesis marks the end of the three year mechatronics programme at Chalmers University in Gothenburg, Sweden. The Work was carried out for the department of Computer Science and Engineering in cooperation with the company Consat Engineering AB at Lindholmen.

Many people has contributed in the process of making this bachelor thesis. We are greatly thankful for the support and guidance from all involved colleagues at Consat Engineering AB during our time at the office. Also Chalmers Technical University for the educational support. Last but not least our friends and family for continuous cheers and advice.

Special thanks goes to:

Chalmers University
Peter Lundin - Examiner
Uno Holmer - Handler
Göran Hult - Man with contacts

Consat Engineering AB
Ragnar Hallgren - Moral support
Jonas Williamsson - Proofreading and moral support
Johan Rubensson - Handler
Martin Lundh - Technical vision expertise

Object detection and analysis using computer vision
An open source development implemented using Raspberry Pi
VICTOR FRIEDMANN SANDIN and ANNA THOMSEN
Department of Computer Science and Communication
Chalmers University of Technology
University of Gothenburg

Abstract

The project was performed at Consat Engineering AB's facilities at Lindholmen, Gothenburg, in the spring of 2017. Consat Engineering AB produced a prototype that included a vision system, to analyze objects. This thesis exists to develop a proof of concept in form of a vision program, lowering the cost of the vision system, in order to be able to implement it in a commercial product. A pre-study comparing different hardware platforms and software libraries was conducted in order to best meet the requested specifications. The open source vision library OpenCV was chosen as the software library. A Raspberry Pi, running a Linux based OS called Raspbian, equipped with a camera was chosen as the hardware platform. Both of the choices included minimal expense towards the final cost of a future product that could be developed from this concept. After a finalized concept an evaluation test concluded that the program developed during this project fulfilled the requirements and proved that an open source solution significantly can lower the cost of the current vision system the prototype utilizes.

Keywords: Computer Vision, OpenCV, Raspberry Pi, open source software.

Object detection and analysis using computer vision
An open source development implemented using Raspberry Pi
VICTOR FRIEDMANN SANDIN and ANNA THOMSEN
Department of Computer Science and Communication
Chalmers University of Technology
University of Gothenburg

Sammanfattning

Examensarbetet utförs i Consat Engineering ABs lokaler på Lindholmen, Göteborg under våren 2017. Consat Engineering AB har utvecklat en prototyp som, bland annat, innehåller ett vision system, för att identifiera objekt. Detta examensarbete går ut på att utveckla ett koncept som visar att kostnaden av vision systemet kan sänkas tillräckligt för att vara lönsamt för att i framtiden implementeras i en kommersiell produkt. En förstudie har utförts för att undersöka den mest lämpliga kombinationen av mjukvara och hårdvara för att ta fram det billigaste alternativet som uppfyller kraven på slutprodukten. Förstudien resulterade i en Raspberry Pi 3 med ett Linux baserat OS vid namn Raspbian och Open source vision biblioteket OpenCV. Efter utvecklingen av visionprogrammet utfördes ett utvärderingstest för att säkerställa funktionalitet och precision av programmet. Med positiva körresultat kan det konstateras att det framtagna konceptet uppnår examensarbetets uppställda krav och visar god potential för vidareutveckling.

Nyckelord: Computer Vision, OpenCV, Raspberry Pi, Open source

Table of content

Innehåll

Abbreviations

VVP	Vector Vector Pointer
function_name()	Indication of a library function.
UI	User Interface
OS	Operating System
API	Application Programming Interface
RAM	Read And write Memory
CPU	Central Processing Unit
GPU	Graphical Processing Unit
IDE	Integrated Development Environment
BSD	Berkeley Software Distribution
ROI	Region of interest

1. Introduction

This chapter will introduce the basis of which the thesis stands upon, which will be broken down into categories such as: the background of the project, the purpose of the thesis along with measurable goals and limitations.

1.1 Background

Consat Engineering AB developed a prototype that among other things included a vision system, to analyze objects. Since the prototype is expensive and of an industrial caliber, it will be less cost-effective to produce on a larger scale intended for commercial distribution. There is a need to downscale the prototype to make it viable for a larger market. Therefore a “proof of concept” project to lower the cost of the device was started. A proof of concept being a preliminary research to investigate the possibilities for further development [1].

1.2 Purpose

The main purpose with this project is to research if a potentially cheaper version of the vision system, which can perform at the same or even higher level than the original prototype, can be developed. A preliminary study will be conducted concerning different tools and solutions that can be utilized to reach an adequate result, as well as widening the perspective into systems and solutions of the same type used in the industry today.

1.3 Goals

The main focus is developing an program that specializes in continuously analyzing images of different singular objects within the visual field of the camera. The goal of the program is to pass or fail the individual object within set criterias of size and colour. The object will only be approved if a dominant colour can be determined and the size fits into a predetermined range of size. The following evaluation parameters will be used to validate the result of the thesis:

- Is there a way to make the program run continuously for a trial of 54 tests without failing?
- Can a size test be developed to give the correct output in 90% of the cases during a test of 54 objects?
- Can a colour test be developed that can differentiate between four colours and determine the dominant one in 90% of the cases during a test of 54 objects?

1.4 Limitations

Below is a list of limitations set for this project:

- **Decision box -** The pass or fail signal will be passed on to a different part of the finished product that collect the result from the program and perform an action depending on the outcome. This different part will henceforth be called the decision box. This report will only cover the development of the program of the vision system.
- **Time -** The project will be a full time endeavour during ten weeks with a preliminary end in the middle of June 2017.
- **Geographic -** All of the development conducted during this project will be performed on location in Consat Engineering AB's office at Lindholmen, Gothenburg, Sweden. The main reason being that the hardware is located on site and cannot be transported.
- **Economical -** There is no set budget for the project but all the necessary tools and materials that are required to finish the project will be provided by Consat Engineering AB.
- **Hardware -** Due to time limitation the project will not pay any extended attention to the hardware. A commercially available platform will therefore be used to perform tests. The project will not include any hardware development.
- **Software -** The program that will be developed is to be built as modular as possible to facilitate easy adaptation and recycling into other projects.

2. Method

This chapter will discuss the methods chosen to reach a conclusion in this thesis. It is broken down into three paragraphs - pre-study, development and evaluation - where each part of the process will be described in order to elevate the comprehensibility.

2.1 Pre-study

The first step will be a pre-study in order to determine the specifics of the program and how it should be able to perform once completed. The pre-study will include an analysis of three software libraries and two hardware platforms. The advantages and disadvantages will be compared to each other, where one of each from the two categories will be chosen as the best option for developing the program. The software libraries will be compared looking at licensing fees, language support and hardware compatibility. The hardware will be compared focusing on performance and speed of implementation. This is done in order to develop an program which can run on the chosen hardware, during ideal conditions, providing proof of concept. In a case of computer vision an ideal condition would be where there are no variables that could lower the success rate of a demonstration.

2.2 Development

Once the pre-study is complete the development of the program will begin. This part of the project will include a large portion of searching for and learning from existing program and program examples. The gathering of information from the chosen software library will be ongoing throughout the development of the program. This is done in order to find the best and most efficient way to reach a final result. The development will be a process of trying to utilize new tools in order to present an adequate proof of concept.

2.3 Evaluation

A runtime evaluation will be conducted on the program’s continuity to secure a memory efficient concept and analyze if the specifications can be met. The framework of the evaluation will be based on the evaluation parameters put forth previously in the thesis (see *1.3 Goals).*

2.3.1 Evaluation test

The runtime test will be conducted running the program on the chosen hardware while documenting the result from each object put in the field of view of the camera. The test will consist of a variety of 18 objects. The objects will be selected by size and colour listed below:

- Six different colours, whereas four are in the approved range:
  - Pink - Not approved
  - White - Not approved
  - Blue - Approved
  - Yellow - Approved
  - Green - Approved
  - Orange - Approved
- Three different sizes within each colour, whereas one is approved:
  - Small - Not approved
  - Medium - Approved
  - Large - Not approved

Each separate object will be tested three times to ensure the same result is reached each time, adding up to 54 tests. If the result deviates, testing the same object it will be considered a failed iteration. If the program for some reason should terminate before all tests are finished, the entire test will be restarted. The desired result of the test is to achieve the previously mentioned evaluation parameters (see *1.3 Goals*).

3. Pre-study

Included in this study is a comparison of three different software libraries available for developing and analyzing visual content and two popular hardware platforms often used in projects of this type. This study will be an inspection to see which software library and hardware platform is best suited to reach the requirements of this project (see *1.3 Goals*).

3.1 Hardware platform

Since the main focus of this project will be developing the program, the fastest way to get started is to find a commercially available platform on which to develop and run the program. The two most frequently used hardware platforms for developing concepts and testing smaller projects are Raspberry Pi and Arduino [2-4]. They are both compact, affordable and have great support for peripheral equipment. Criterias to be evaluated are: clocking speed, memory, storage, peripheral support and programmability. These criterias have been chosen due to the impact they will have on the development.

3.1.1 Arduino Due

There are a multitude of different Arduino microcontroller models with different areas of intended use [5]. The *Due* was chosen as Arduino's option in this comparison, since the Arduino model *Due* has the highest clocking speed among the different models with 84 Mhz. Thus, making it the model most likely to be able to handle the processing of images that this project entails [6]. This model has support for connecting special Arduino cameras and even a standard USB webcam [7]. The onboard flash memory is at 512 kB which is supposed to hold the program and potential images to be captured. This means small sized images or adding extended memory to increase storage potential. Even if the image is small enough to fit the built in flash memory the SRAM at 96 kB provides limited room to perform larger functions for instance compare two images. The *Due* has 54 I/O pins that can supply a total current output of 800 mA, although each pin is limited to 130 mA [6].

The Arduino program is written and compiled on a separate computer and then uploaded to the board via USB port [6]. Since the Arduino is using a C/C++ language to write code in, there is a possibility to use an external Integrated Development Environment (IDE) not affiliated with Arduino to write and compile code, as long as the Arduino core library is linked properly to the compiler [8]. The easiest option will be to use the free *ARDUINO* software which has a compiler and upload functions built in along with serial monitoring and other useful debug tools [9].

3.1.2 Raspberry Pi

The microprocessor Raspberry Pi comes in different models separated by performance and price [10]. It is now on its third generation with the first generation being released in 2012. The main series is named *model B* and a cheaper version with the same capabilities but slightly lower performance is named *model A*. The alternative for consideration from Raspberry Pi will be the *Raspberry Pi 3 model B* since it is the latest and has the highest performance. The microprocessor is equipped with a quad core processor, each core with a clock speed of 1.2 GHz, and a 1 GB of SDRAM which is shared with the Graphical

Processing Unit (GPU). However, no onboard memory storage is included on the board but there is a micro SD card module which is supposed to hold the primary storage [10,11]. There are 17 General Purpose I/O (GPIO) pins that can, without risk of damage to the board, supply 50 mA from the 3.3V rail and around 300 mA from the 5V rail. The 5V rail is not set, it is calculated from the current supplied by the USB power supply minus the current used by the hardware itself [12,13].

The Raspberry Pi is using its own Operating System (OS), has audio/video outputs and USB inputs for keyboard and mouse. It is therefore more accurate to call it a computer instead of a microcontroller like the Arduino. These features add to the simplicity of working with the Raspberry Pi. The OS used is a Linux based version called Raspbian and functions like any Linux based OS. This allows the user to write code, compile and run it without having to upload it to an external controller (which would have slowed down the speed of implementation). Since it is using an OS the Raspberry Pi is fully configurable, meaning any programming language is available with just a few commands in the control terminal. Making it even more adaptable to personal preference or requirements, due to libraries being written in specific languages [14,15].

3.1.3 Resulting Hardware platform

Raspberry Pi differs from the Arduino in that it is a microprocessor and not a microcontroller. The Arduino focuses on fast processing of analog and digital I/O-signals and a small amount on arithmetic processing of data. The latter being where the Raspberry Pi excels.

The strength of the Arduino Due is the amount of current it can supply making it ideal for use as a controller when using servo’s and other small electric motors. The max output of current from the Arduino Due, before damage is an issue, is around 800 mA compared to where the Raspberry Pi 3 model B can supply 50 mA on the 3.3V rail and around 300mA on the 5V rail. Seeing as high currents is not required in this project the Arduino exceeds requirements.

The main focus of this project is to develop a program that processes images at a fast pace, continuously. The memory and clocking speed of the Arduino Due would be on the lower end of performance needed to develop a finished product. It is therefore discarded as an alternative. The Raspberry Pi 3 Model B on the other hand excels in these areas with its high clocking speed, around 14 times faster, and a memory about 10 000 times larger than the Arduino Due’s.

Due to the above mentioned specifications the Raspberry Pi 3 Model B will be the hardware platform the vision program will be developed upon along with the camera Raspberry Pi Camera V2. The camera is selected due to the high resolution (8 MP) it captures pictures in and the wide colour spectrum it can detect (infrared to violet). As an added bonus it has pre-installed drivers and communicates with the Raspberry Pi well [16].

3.2 Software Library

Each of the three software libraries about to be compared in this study are powerful enough to produce an adequate program that will meet the specification of this project (see *1.3 Goals*).

3.2.1 Adaptive Vision

In this software library the building of applications are made simple by adding filter functions from well categorized lists in the graphical IDE: Adaptive Vision Studio. The drag and drop approach makes the IDE easy to use and will open up computer vision for non-programming engineers to create and manage advanced vision systems [17]. The company supplies a video tutorial series to get users started with the software adding to the achievable low threshold this software library presents [18]. A variety of cameras and frame grabbers are compatible with Adaptive Vision.

Adaptive Vision Studio allows partial code to be exported into larger C/C++ and .NET projects, making it a good addition for optimizing code that is written using other libraries [18]. The IDE also allows third party libraries to be added into the graphical interface, making it an even stronger and more versatile software [19].

Adding all these benefits together with the support the company supplies, continuous updates and performance tests ensures a powerful, easy to use all-in-one solution to industrial computer vision. A drawback, in the scope of this project, to Adaptive Vision is the fact that it has licensing fees [20].

3.2.2 Halcon

Halcon from MVtec software GmbH is a professional vision library developed for industrial and commercial applications. It is optimised for performance and precision [21]. Halcon applications are built using the IDE HDevelop or by writing code using its library of functions and algorithms. The IDE is a user friendly User Interface (UI) that allows non-programmers to create applications by selecting filters and functions from different menus. The results are then displayed instantly and calibrations updates the result for every new alteration. When the building of an application is finished and tested in the IDE, it is exported as a source file in the language that best fits the rest of the project structure [22]. Thus, the user has a simple way of integrating the vision programs, created in the IDE, into large projects. Currently Halcon supports C, C++, C#, Visual Basic, .NET and Delphi [21]. A variety of cameras and frame grabbers are compatible with Halcon [23].

Halcon is a product with technical support along with the library of optimized functions. Nevertheless, similar to Adaptive Vision Studio, Halcons is a product with licensing fees which would not be desirable in this project [24].

3.2.3 OpenCV

OpenCV (Open Source Computer Vision) is a community driven open source software library, making it a good place to start working with computer vision. An obvious drawback with using this library is the fact that there is no official IDE to use when building applications [25]. To build an application the user is required to use one of the programming languages mentioned in the paragraph below, making this the tool with the steepest learning curve. Due to the immense user base there are however, many forums in which problems and new ideas can be discussed, making the lack of an IDE less of a problem [26].

Written in C++ the library is accessible using C, C++, Python and Java adding to the availability and possibilities when using this library. Furthermore, OpenCV supports most cameras and frame grabbers. The problem that could arise concerning these is: driver compatibility with the computer's operating system. This is however, a possible problem which concerns all three of the libraries in question [26].

The software library is registered as a BSD licence meaning the modification and copying of the source code is allowed without having to share the results with the community. Therefore, it is free to use in commercial, industrial and private projects [27].

3.2.4 Resulting Software Library

As mentioned previously, the three vision software libraries are all separately powerful enough to be able to reach the required specifications of the program (see *1.3 Goals*). This is based on the fact that each of them are frequently used in industrial vision systems [26,28,29], much like the one to be developed in this project.

The following evaluation will focus on:

- Prerequisite skill required
- Technical support
- Hardware compatibility
- Licence fees
- Programming language

Since the ultimate goal of this project is to minimize the cost of the product the program will have to be built using open source material. This means that all the development will be focused around a Linux based environment utilizing free to use software. This limitation excludes Halcon and Adaptive vision since they both require a licence to use. This leaves OpenCV as the best (and only) option to begin developing the program.

4. Technical background

This chapter will give a basic introduction to how computer vision represent colours and which different colour types that are relevant to this project. It will also provide explanations and information regarding utilized functions from the software library.

4.1 An introduction to computer vision

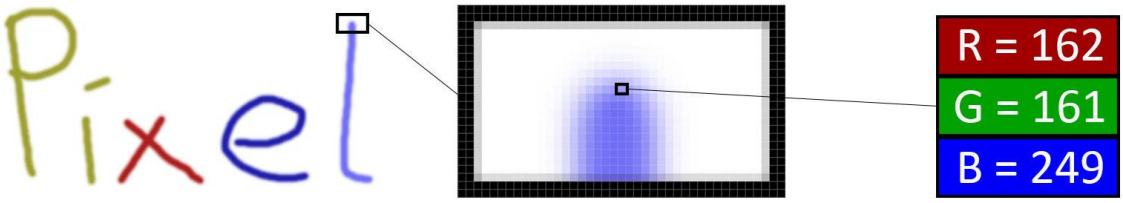


Figure 4.1 *A visual explanation of computer vision and colour representation using red green and Blue (RGB) cells.*

Computer vision is based on a multidimensional matrix to represent images and video (stream of images) [30]. An element in the image matrix is more commonly known as a pixel, which is an abbreviation of the phrase “picture element” [31]. As depicted in figure 4.1 every pixel in an image matrix is divided into cells, referred to as channels in computer vision [32]. Each channel holds the specific intensity level that correspond to the amount of colour needed, when combining the channels, to result in the colour hue the pixel represents. If the matrix depicted in figure 4.2 is interpreted in RGB-mode, those four pixels would all be white since the intensity value in every channel is maxed out. The fact that the channels max out at 255 is explained by computer vision using unsigned char containers as default to store the integer values in, ranging it 0-255 [33]. It is by manipulating and analyzing these element’s channels in various ways the image can be altered.

255,255,255	255,255,255
255,255,255	255,255,255

Figure 4.2 *2x2 matrix with 3 channels (cells) per pixel (element).*

4.1.1 RGB - Red, Green, Blue

A liquid crystal display (LCD) monitor RGB pixel consists of three cells, or channels, containing the intensity value of each individual colour. Each pixel is basically a square containing three areas of the colours red, green and blue, as seen in figure 4.3. Irrelevant of the shape, RGB-pixels use the same colours: red, green and blue, to produce the resulting colour [34]. Considering that many monitors use RGB-pixel technology, it is advantageous to use RGB matrices when displaying an output image. Even though each of the different colours are displayed in separate areas every pixel on a monitor, the individual cells are too small to make out with the human eye. This leads to the separate colours blending together and being perceived as the actual colour the monitor is trying to display [33].

RGB colour theory also follows a logic that most people can relate to, the mixing of colours to create a new combined colour. In OpenCV the channels are reversed using the order of BGR to represent colours [35].

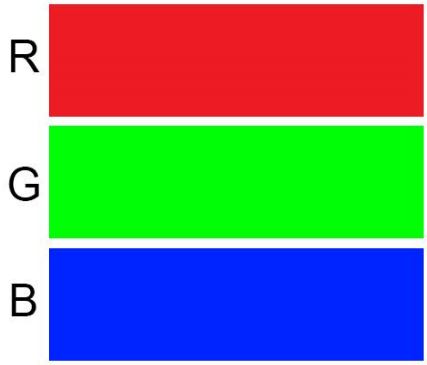


Figure 4.3 *Composition of a RGB-Pixel, this particular colour blend [255,255,255] would result in a white pixel if small enough.*

4.1.2 HSV - Hue, Saturation, Value

The first channel in HSV is *hue* and it defines the pure colour. The second channel *saturation* determines the amount of white that will be incorporated in the mix. And finally choosing the resulting shade of the colour with the third channel *value* which defines the amount of black that will be included [36]. A visual explanation of the three different channels in HSV can be seen in figure 4.4.

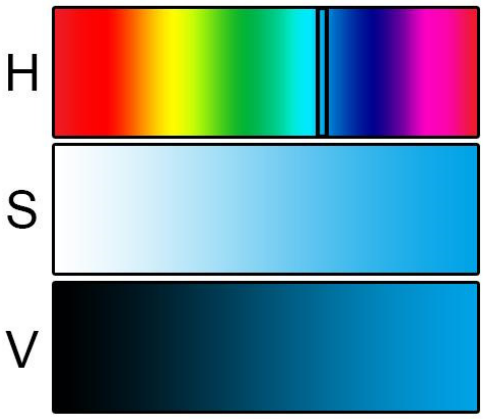


Figure 4.4 *HSV colour composition*

This colour composition has many advantages in computer vision, since the first channel alone defines the colour, apart from the RGB channels which has to be summarized to get the final colour. This allows algorithms to find a specific colour hue, without having to weigh all channels to find the resulting colour, and it is therefore often used in object-tracking software [37].

A HSV matrix cannot be displayed in a correct manner on a RGB monitor if not converted into RGB first. Since the HSV elements use the same amount of channels as RGB (three), the matrix can be interpreted as RGB but all the colours would come out incorrect. Forcing a RGB monitor to display a HSV matrix would result in the right image in figure 4.5.

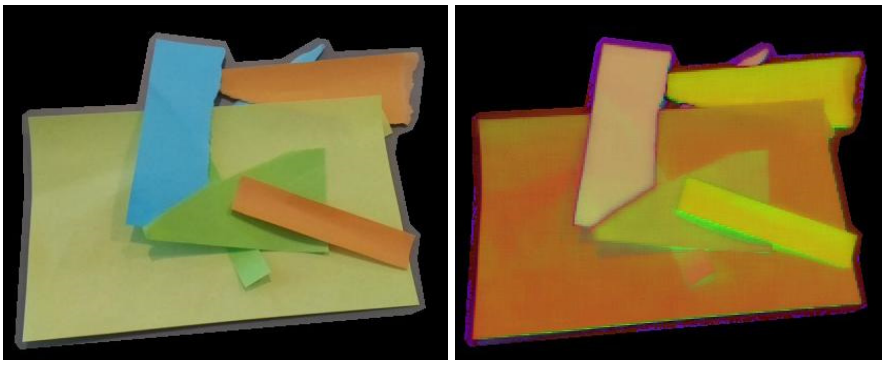


Figure 4.5 *From left to right: RGB matrix displayed on RGB-monitor, HSV matrix displayed on RGB-monitor.*

4.1.3 Grayscale

By standard definition grayscale is defined to be outside the spectrum of perceivable colours and is composed of light and lack thereof. For simplicity grayscale will hereby be referred to as colours mixed from quantities of black and white. White is academically composed out of all the colours in the visible spectrum while black is the absence of any colour [38]. In computer vision the computer interprets black areas as the lowest intensity and vice versa white is the highest intensity. All shades of gray can be observed in between the minimum and maximum values, depicted in figure 4.6.

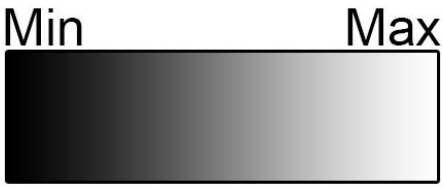


Figure 4.6 *Grayscale spectrum, Min and Max representing high and low.*

Grayscale matrices is a useful tool in computer vision, since it is represented using only a single channel. Using only two intensity levels from the spectrum this type can be interpreted as a boolean *High* and *Low*, binary value [39].

4.1.4 Thresholding

In computer vision and image processing there are many advantages with using binary images. Binary in this case meaning boolean *high* or *low,* in the grayscale spectrum. *Low* represented with a zero intensity (black) and *high* is set by the user, however by default set to max intensity (white) [39]. Binary images can be utilized in tracking certain colours or finding patterns in images otherwise obscured by varying intensity [40]. There are different methods to convert an image into a binary image. The common factor of these methods is the boundary that separate *high* from *low,* i.e threshold. There are ways to create dynamic boundaries which will move the threshold, using a larger area to calculate the best threshold for the center pixel located in that area. The basic way to threshold an image is to have a global static boundary to compare each pixel to and set it accordingly, *high* or *low,* depending on individual intensity [41]. An example of a global threshold can be seen in figure 4.7.

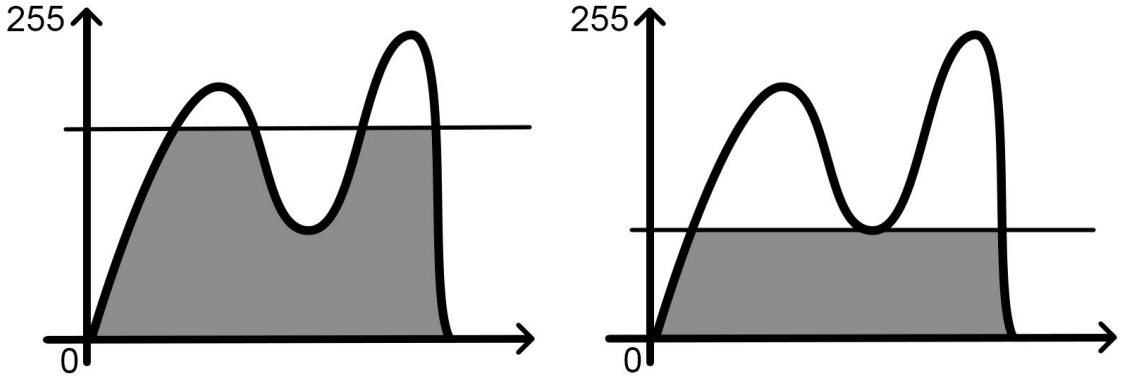


Figure 4.7 *Threshold set at different levels yields different results.*

The result of setting the global threshold too high can be seen in the left image of figure 4.8, where some of the information is lost due to shadows or very small intensity variations. In the right image of figure 4.8 the whole object can be seen as a result of the global threshold being set at the right level for this particular image.

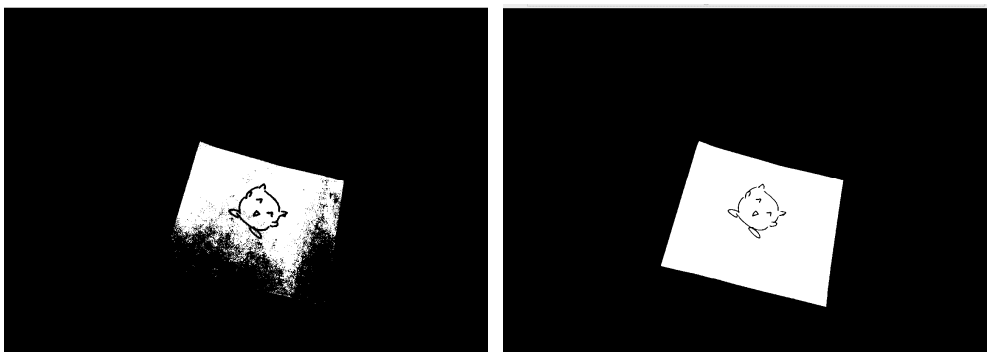


Figure 4.8 *Resulting image after threshold in* Figure 4.7, *from left to right: Too high threshold, moderate threshold.*

4.2 Vectors in C++

A convenient way of storing information in C/C++ is the array container. It makes replacing, pulling or erasing data a simple task by selecting an index in the container of interest and assigning the new data or reading the existing data. The drawback to arrays, in the scope of this project, is that an array must be assigned a size/length, limiting the amount of data points that can be stored while the program is running. This is where the vector class in C++ make its entrance [42].

A vector is similar to an array since information can be accessed from the vector by the index of the element holding the information. The way that vectors differs from arrays is the functionality of vectors that their size/length is dynamically altered when elements are added or removed [43]. This functionality makes it a perfect candidate for conserving memory since an array would have to be allocated in its entirety regardless if it is filled or not. When using vectors it is important to erase or empty the vector when it is used in any kind of continuous loop, since it will keep adding to the end of the vector creating a memory leak.

4.3 Utilized features in OpenCV

The following paragraphs is a fraction of the over 500 functions included in the OpenCV software library [26]. The following functions are the biggest building blocks in the resulting program of this project.

Mat - Image/matrix container
The Mat container is the container in which OpenCV stores an image, the matrix of pixels. Mat is a C++ class consisting of two parts: the header and the pointer. The header holds the information about the matrix such as: matrix size, memory location, type and so on. The pointer refers to the pixel information and is the dynamic part of the class, meaning that this is the part that varies in size while the header always remain the same. The pointer tracks all individual pixel values and channels, both depending on the type of matrix (see *4.1 An introduction to computer vision*) . A Mat can be passed as reference to functions making it possible to minimize memory usage by not creating large matrices for temporary use [44,45].

clone()
If a complete copy of a matrix is needed the function *clone()* will allocate new space and copy all the information to the new matrix [46].

copyTo()
The function *copyTo()* will not create a new matrix like *clone()* does. It will, however, copy all the information from one matrix to a pre-existing matrix. If the destination matrix is of another size the function will re-allocate the space needed. The function also allows for a mask matrix to be overlaid on the destination matrix creating a cropped matrix with what is called a region of interest (ROI)[46].

cvtColor()
There are many types that can explain how matrices is supposed to be interpreted by different functions. Some types of matrices stores colour values in their channels while some are single channeled matrices holding only the intensity. The type also indicates the resolution of each channel. *cvtColour()* is a function that converts a matrix from one type to another. It calculates the corresponding value of the new type and places it in the right channel of the destination matrix.

The default matrix type of OpenCV is *CV_8UC3*. The method to decode type names is by breaking it down into specific parts, listed and explained below:

- “CV_” is the indication that it is an inherent OpenCV type.
- “8U” indicates that the size of the individual cells is built of 8-bit unsigned chars, giving it values between 0-255.
- “C3” indicates the amounts of channels or dimensions of each matrix element (pixel).

The function *cvtColor()* is used to change the size of the matrix and recalculate the colours represented by the matrix. For example:
Converting from BGR-*CV_8U3C* to Grayscale-*CV_8U1C* will mean the height and width of the matrix will stay the same but the amount of channels will be reduced from three to one.

This effectively limits the possible outcome of each pixel from 256^3 = 16 777 216 to 256^1 = 256. Along with the reduction of channels the values inside the remaining channel will be calculated to the grayscale equivalent of the previous resulting colour of each pixel.

The two types in the example above both use 8-bit unsigned char containers in their matrices. However this is not the only size of container a cell can have. There are types that, instead of using 8-bit cells, use 16, 32 and 64-bit size. This dramatically increase the resolution of the chosen colour spectrum. If we compare the RGB colour resolution in the 8-bit to the 64-bit we have 256^3 against 9,223,372,036,854,775,807^3 different colours, Although not discernable with human eyes, this high resolution can be used in computer vision due to everything being interpreted as numerical values. Using the method from earlier to decode the identifier, the type in the resolution comparison above would be *CV_64FC3,* where U for uchar is replaced with F for float, although the channels, C3, remain the same since RGB still has three channels [47].

erode(), dilate()

The functions *erode()* and *dilate()* are each other's inverse functions, the pair can be used to eliminate or enhance light and dark areas in any matrix. They are often used in succession of each other and is an effective way to eliminate noise and shadows from a matrix. The effect can be seen in figure 4.9 where it is demonstrated that *dilate()* expands light areas contra *erode()* that diminishes the same. Depending on the composition of the matrix they are applied to, they can be utilized to many different effects [48,49].

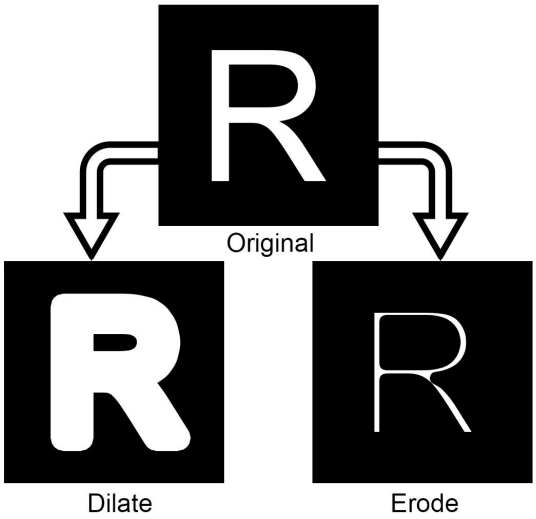


Figure 4.9 *Visualizing the effect of erode/dilate functions*

Threshold()

The function *Threshold()* represent the most basic method to threshold an image. Using a static global intensity value that every pixel is compared to and then set *high* or *low* depending on the outcome of that comparison. The function requires a single channel matrix as input, in most cases a grayscale matrix is used. Different types of thresholding exist that will affect the output of the function [39,50]. The types relevant in this project is listed below:

- THRESH_BINARY - pixel intensity above the threshold is set to *high,* other to *low.*
- THRESH_BINARY_INV - pixel intensity below the threshold is set to *high,* other to *low.*

There are three other types that can be used in this function, however, they will not produce a binary image and are therefore not mentioned [39].

adaptiveThreshold()

In the event that the previous function *Threshold()* does not produce a desired outcome, the more advanced *adaptiveThreshold()* can be utilized to great effect. Instead of evaluating each pixel individually, a small matrix, default size 3x3, is established. That matrix is then used to examine the surrounding pixels intensity to determine where the threshold should be set, for the pixel currently in the center element of the matrix. This matrix is then moved so as the center element of the matrix is placed over the pixel that is yet to be analyzed. This method of thresholding will not suffer from the same shortcomings, over and underexposure, as *Threshold()* does [41,51]. As seen in figure 4.10 the adaptive method of thresholding will produce an image with minimal loss of information.

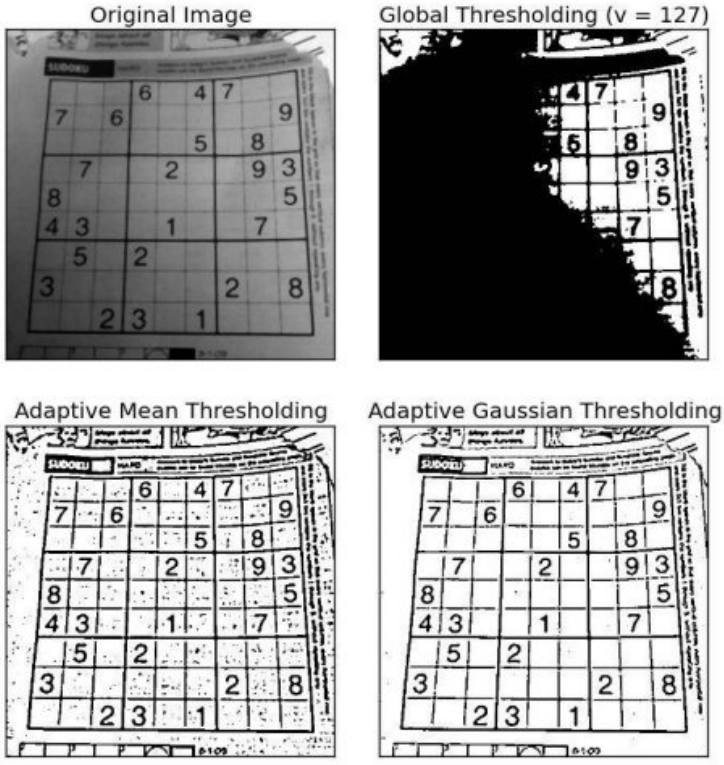


Figure 4.10 *Image series displaying the different results that can come from the different threshold functions [41].*

There are two methods to evaluate the neighborhood of pixels:

- ADAPTIVE_THRESH_MEAN_C
- ADAPTIVE_THRESH_GAUSSIAN_C

Both methods use a straightforward formula for calculating the mean value of the neighborhood of pixels before subtracting a constant C. The formula can be seen below [52]:

$$Threshold = \frac{\Sigma(neighbour * multiplier)}{size\ of\ neighborhood} - C$$

The difference between the *Mean* and the *Gaussian* method is that the neighborhood matrix used in the *Mean*-matrix is filled with ones as a multiplier, to create the average of the neighborhood. The neighborhood matrix used in the *Gaussian* method is filled with multipliers mimicking the gaussian bell curve or normal distribution curve [53]. This will add more weight to the pixel neighbours closest to the center pixel when calculating the threshold. An example of a 7x7 multiplier neighborhood matrix can be seen in figure 4.11.

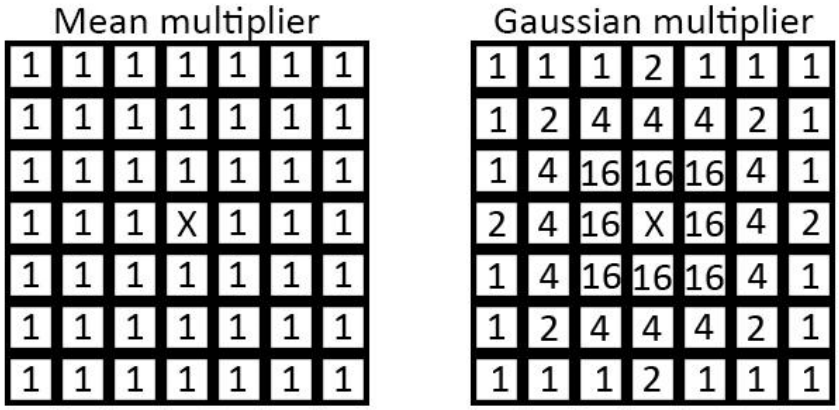


Figure 4.11 *From left to right: Examples of Mean multiplier neighborhood and Gaussian multiplier neighborhood.*

Canny()

Canny works differently from the other methods briefed above, the most noticeable difference being that it uses two threshold values to resemble a type of hysteresis interval. This allows the function to detect strong contrasts as the beginning of edges and follow the edge down to the lowest threshold, making it very proficient in drawing closed edges that envelopes an entire object.

Canny() identifies the center pixel of the edge gradient and draws the edge a single pixel wide, giving it a more precise representation in the output matrix [54,55], an example of this can be seen in figure 4.12. This is where *Canny()* differs the most from the earlier binary conversion functions, which follow their set parameters and set each pixel either *high* or *low* (white or black) based on the intensity of each pixel (*Threshold()*) or based on the neighborhood of pixels (*adaptiveTreshold()*).

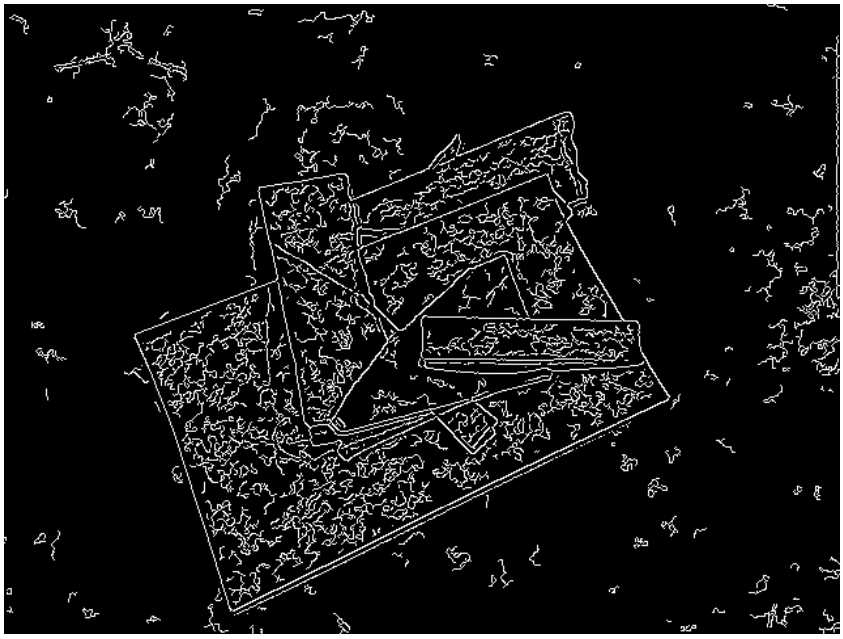


Figure 4.12 *Canny edge detection results*

findContours()

The function *findContours()*, as the name suggest, finds contours, exclusively in binary matrices as the function encircles areas of pixels with *high* values. All pixels next to each other that encircle a single area of *High*-intensity pixels is grouped together and stored in a vector vector pointer (VVP). What we call a contour the computer sees as a vector of pointers that point to pixels that edges the transition from *high* to *low*-intensity and vice versa. Due to the fact that the contour is stored as a vector it is not connected to a matrix but is an independent container of data ready to be used in other functions [56].

Hierarchy

To distinguish one contour from another, in a VVP, a hierarchy is set when the contour is found to give it a place in a sequence. Hierarchy level is defined using four identifiers: [next, previous, first_child, parent]. With these it is possible to locate a certain contour within another by looking at the identifiers *parent* and *previous.* If an identifier is “-1” it does not have a contour that fits that parameter. As in figure 4.13 the first *parent* is the “0” contour but since there is no other contour in the same level of hierarchy as “0” it will have no *previous*, *next* or *parent*. The children of “0” are “1”, “2” and “3” but only the *first child* is set as an identifier. The identifier for the “0” would be [-1,-1,1,-1]. As seen in the figure the *first child* of the “0” contour the “1” is also a *parent* to the “4” contour inside of it and the identifier for that hierarchy level would look like: [-1, 2, 4, 0] [Ref:57,58].

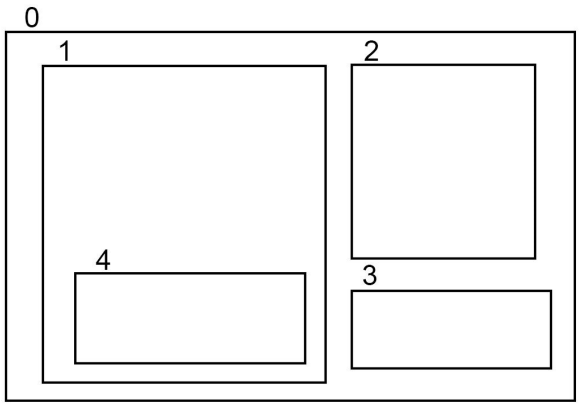


Figure 4.13 *Descriptive figure clarifying how hierarchy is used in OpenCV.*

This functionality can be used to pinpoint certain attributes of the contours by locating its parents or else by looking at contours which has children. The latter would signify that there is a contour inside another contour which could mean a hole or deviation in the object's surface.

approxPolyDP()
This function is used to approximate the contours found using *findContours()*. The function will approximate the new contour by minimizing convex and concave areas, and in doing so smoothing out the contour [59-61].

contourArea()
The function *contourArea()* can be used to calculate the amount of pixels that are encircled by a contour. It will also indicate the orientation of the contour, positive or negative [62].

arcLength()
This function is another way to determine the approximate size of an area encircled by a contour. This is achieved by measuring the arclength of the contour and use the results to compare contours to each other. The longer contours will theoretically encircle a larger area [62].

drawContours()
The function *drawContours()* can draw a found contour or fill the contour with any specified colour in any given destination matrix [62].

countNonZero()
Similar to *contourArea()* the function *countNonZero()* can be used to calculate area. Instead of calculating the amount of pixels encircled by a contour like *contourArea(), countNonZero()* will summarize the pixels that are not equal to zero, which is all pixels that are not black elements. This function can be applied to both colour or Grayscale matrices and is particularly useful along with the function *absdiff()* [63].

absdiff()
By calculating the absolute difference between two matrices of the same height and width, the function *absdiff()* will produce a grayscale matrix, the difference of each pixel marked with the corresponding intensity in the grayscale matrix. This will create a matrix that will highlight the difference between the two source matrices. This can be utilized in motion tracking and background reduction among other things [64].

inRange()
The *inRange()* function is another way to threshold into a binary matrix. The difference and great benefit with using this function, contra *Threshold(),* is that the input matrix does not need to be a single channel type. As a combination of *cvtColour()* and *Threshold()*, *inRange()* will convert the type of the matrix to output a grayscale matrix with only binary *High* and *Low* pixel values. The power of this function lies in the range that it compares every pixels against. The upper and lower bounds are set for each channel in order to get the desired result. Whether it is a HSV or BGR matrix, *inRange()* is a good way to find the pixels that fit the predefined range. Using the binary matrix output to create regions of interest or pair it with the *countNonZero()* function to get the amount of pixels that are in range, are a few different ways the function can be used [65].

5. Theoretical draft

By reading a tutorial from OpenCV's tutorial documentation [45] a theoretical draft was formed as a first approach to build an program that could perform according to specification in *1.3 Goals*. The draft in form of a flowchart below in figure 5.1. displays an overview of the preliminary flow of the program.

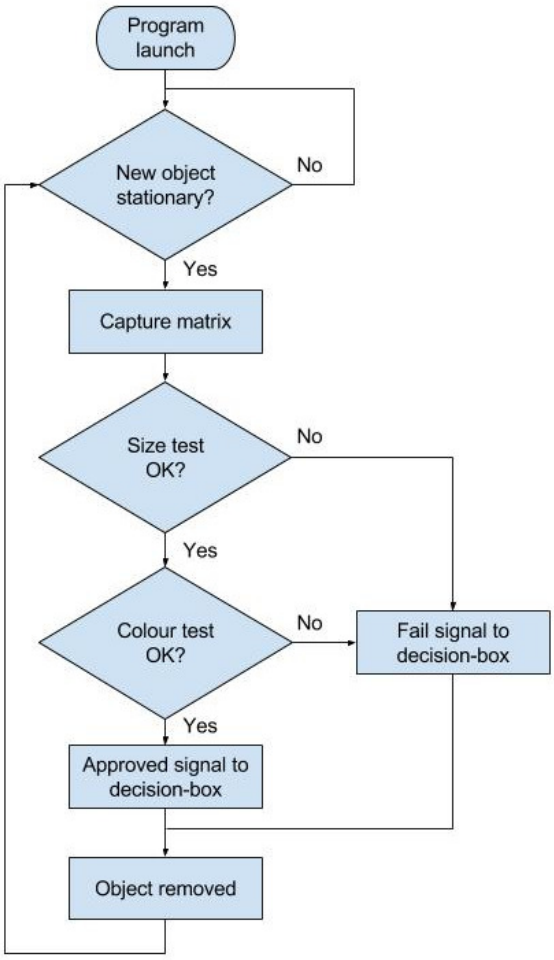


Figure 5.1 *Theoretical draft of the program.*

After the program has declared functions, defined variables and connected various libraries, the program will enter its main loop. When the object arrive in the camera's field of view it will still be in motion and therefore a function is required to verify when the object is stationary. In order to capture a detailed image to analyze, the object has to be stationary. Once this condition is satisfied, the analysis will be initialized.

The first attribute to be analyzed is the object's size. Depending on the outcome of the size analysis the program can skip the upcoming colour analysis by sending a "fail signal" to the decision box to remove the object since both tests has to be fulfilled in order to send an "approved signal". After the object has been removed, irrespective of the outcome of the analysis, the program will jump to the beginning. The main loop, as seen in figure 5.1, will continue until manually terminated, this allows for continuous analysis of objects.

6. Implementation

This chapter starts with an overview of the program's workflow to roughly explain how the program works. The core revolves around the size test and colour test, which each requires preparations to produce the tests' input.

6.1 Program overview

As seen in figure 6.1 the flow of the final program has more steps compared to figure 5.1 from chapter *5. Theoretical draft*. Those steps are added to improve the accuracy of each analysis and to ensure the continuity of the program. The added steps to the flow of the final program are:

- Detection of a new object in the camera's field of view by using motion detection.
- Confirming object removal with motion detection.
- Reset the program in conjunction with previous step.

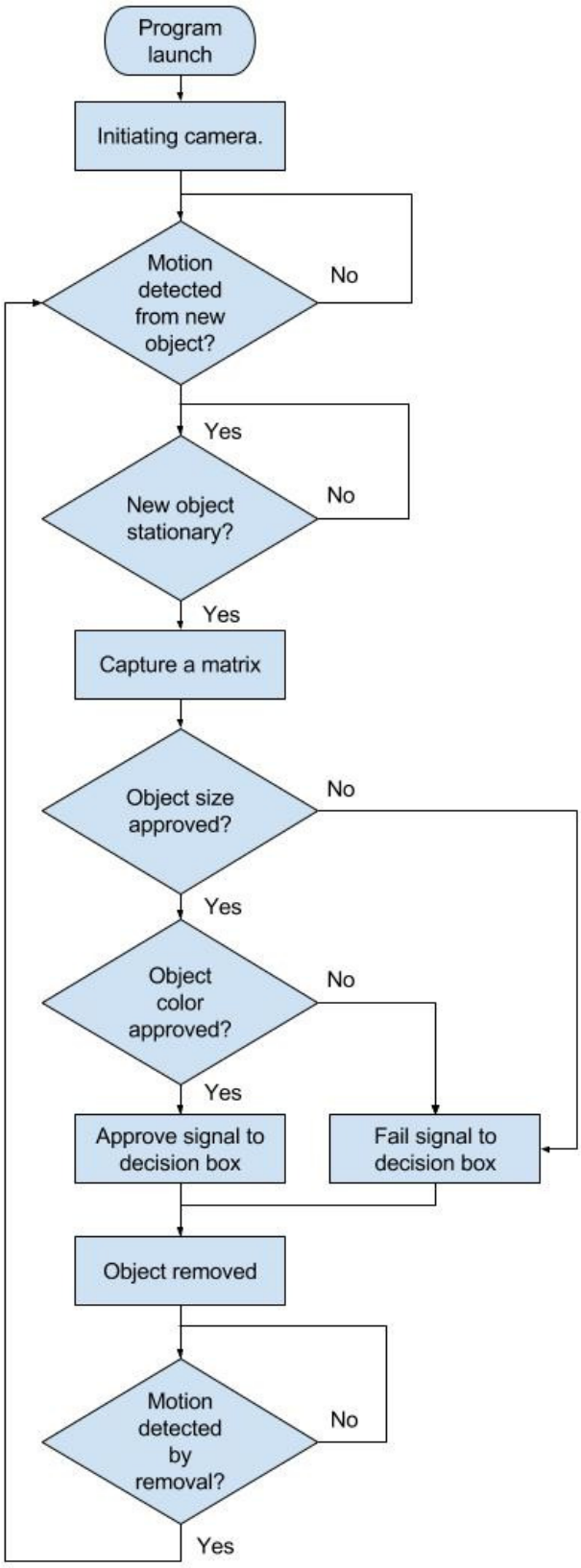


Figure 6.1 *The final flowchart of the programs workflow.*

6.2 Startup and new object detection

Before the object is ready to be analyzed the program is required to perform a couple of start up necessities. After the start up is done two loops follows to determine if the object's state lacks motion and thus is ready to be analyzed. This in order to ensure that sharp images can be captured.

6.2.1 Initiation

Upon executing the program, the required setup is performed as listed below:

- Connecting all relevant libraries.
- Function declarations.
- Input and output devices are defined.
- Containers are allocated and initiated.
- Debug windows are created.

In addition to the standard C++ libraries there are several other libraries that must be linked with the program to function. The most important are:

- Library containing functions to operate the Raspberry Pi camera.
- Library containing functions to control GPIO-pins of the Raspberry Pi.
- OpenCV vision library.

6.2.2 Wait sequence for new object

The solution to secure that the object have attained a stationary state is divided into two parts. The first part is a loop to detect motion and the second part is another loop that detects the lack of motion. In other words the first loop detects if a new object has arrived in the camera's field of view, and the second loop detects if if the new object has stopped moving.

Before entering the loop to detect motion a matrix is captured as reference and stored as a file. Once the program has entered the motion detection loop another matrix is captured and also stored as a file. The motion detection function compares the two latest captured matrices using the function *absdiff()*, which calculates the difference between two pixels and displays the result in a grayscale matrix. A change has occurred if the result is separated and is displayed as a grayscaled pixel with its intensity varying by the difference. If no change has occurred the pixel is displayed by a *low value* (white). The greater the amount of low value pixels, the less motion has been detected. The generated grayscale matrix is used as an input to the function *countNonZero()*, which calculates the amount of low value pixels. The amount of low value pixels are then divided with the total amount of pixels contained within the whole matrix. This gives the portion of pixels that has not encountered any change. If more than 10% change has been detected in the two last captured matrices the loop will be terminated, otherwise the detect motion function will start over.

The second loop is almost identical to the first loop explained above, but instead the loop terminates if less than 5% motion has been detected. Upon the second loop terminating the program will recognize the object as ready to be analyzed.

6.3 Creating the matrix

The program now recognize the object to be still and ready to be captured by the camera. In order for the program to accurately analyze the matrix's attributes, and later use them in the size and colour test, the matrix must first undergo a few procedures. The required modifications are:

- A grayscale conversion.
- Removal of white and black clutter.
- A binary conversion.
- Reinforcement of brittle edges.

The goal is to achieve a matrix that displays the background by *low values* (black) and the object by *high values* (white).

6.3.1 Capturing a matrix

At this stage the object is motionless and the camera can capture a sharp matrix. Per default the information from the camera is only stored temporarily by the chosen hardware platform. To be able to use the captured matrix in later operations the program stores the matrix as a file and also as an element in a vector of mats. It is this file that will be continuously edited throughout the program, and in next program cycle replaced with a new matrix. The vector of mats has twenty elements and stores matrices captured and post significant modifications. The functionality of the program does however only require two stored matrices for comparison functions but the additional history can sometimes come in handy. The sequence of matrices can be displayed on a monitor to ease debugging.

6.3.2 Conversion to a grayscale matrix

The program has a matrix to work with and the modifications of the matrix can begin. The grayscale conversion is performed by the function *cvtColor()*, see chapter *4.1.3 Grayscale*. The function's input is a RGB coloured matrix and it calculates a single value (one channel) out of the three channels. The calculated value ranges from 0-255 and is displayed as a intensity going from black to white. As seen below in figure 6.2 the input image on the left has been transformed into a colourless matrix on the right.

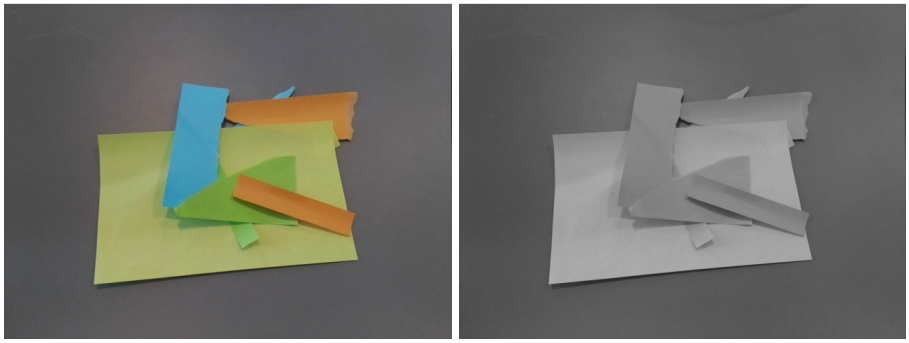


Figure 6.2 *From left to right: The input matrix to the function cvtColor() and the function's output to the right.*

6.3.3 Removal of noise

The matrix is now grayscaled but contains white and black clutter from the conversion, these needs to be removed in order to get a black background, as mentioned in *6.3 Creating the matrix*. The clutter is random data produced by the camera in the moment of when the matrix is captured, this phenomenon is referred to as noise [66]. The human eye may not be disturbed by the noise when trying to decipher the matrix, but the program do not know which pixels are flawed and not intended to be taken in account for. The program interprets the flawed pixels just as valid as any other pixel and the noise will therefore be of a problem in later stages if not addressed appropriately. To cut down the amount of noise two functions called *dilate()* and *erode()* are used, see chapter *4.3 Utilized features in OpenCV*. In combination they smooth out contrasts and eliminate a significant amount of noise.

First *dilate()* is used ten times to expand light areas and overgrow nearby clutter, then *erode()* ten times to decrease the light areas to their original size. Below in figure 6.3 the image to the left shows the matrix before the use of *dilate()* and *erode()* and to the right the result thereof. Note that the right image have smoother intensity transitions but a contrast between the foreground and background is still notably present.

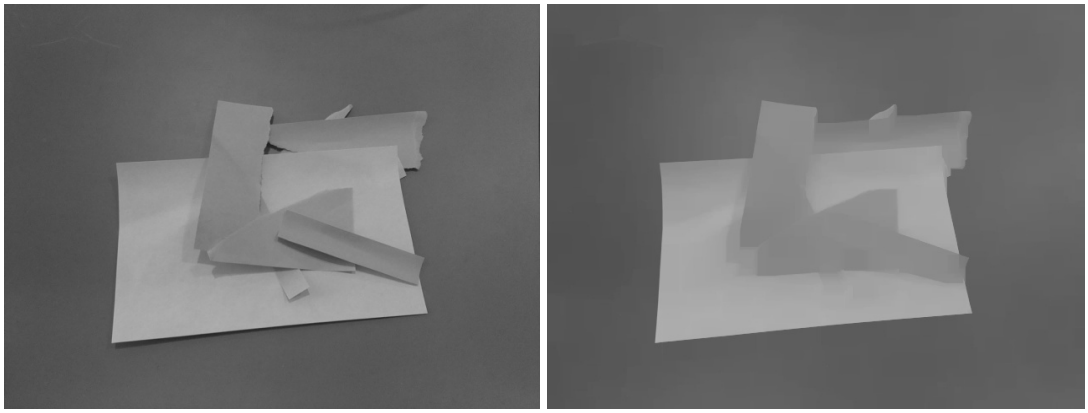


Figure 6.3 *From left to right: Grayscale image and the result from the use of erode and dilate functions.*

6.3.4 Conversion to a binary matrix

Most of the noise in the grayscaled matrix has been blended into the matrix and the contrast between the foreground and background has been made more notable by the earlier grayscale conversion. To gain maximum contrast in the matrix it is converted into a binary matrix where the object will be displayed by high values and the background by low values. Unfortunately some noise might be leftover and also be displayed by high values. Four methods are considered for the conversion and those functions are:

- *threshold()*.
- *adaptiveThreshold()*, mean method.
- *adaptiveThreshold()*, gaussian method.
- *canny()*.

Below is a briefing of all four methods and an examination of their results. The methods are arbitrarily judged by their visual outcome with the most important aspect being how accurately represents the object's shape. In an ideal case the whole object is displayed by high values and the background by low values. The chapter ends with a conclusion of which method is chosen to be best suited for the task.

All four methods convert a single pixel to either a *high* or low value but each method has its own way of determining the value. See chapter *4.3 Utilized features in OpenCV* for a deeper explanation of each method.

Function: *Threshold*
The function *threshold()* is based on a single pixel's value and therefore found to be the most straightforward binary conversion. The user inputs a value of 0-255 into the function and the function determines if a pixel belongs to the upper or lower range of that value. If the pixel belongs to the upper range it results as a low value pixel and vice versa. This option is very basic and sometimes leaves the object unrecognizable because it does not take in account for variances by light.

Function: *adaptiveThreshold*, mean method
The function *adaptiveThreshold()* converts a pixel depending on its own value and also the pixel's neighborhood. The width of the neighborhood can be configured to adjust the smoothness of the light transitions. There are two methods to *adaptiveThreshold()*, the first method is by the neighborhoods *mean* value and the second is a weighted value by a *gaussian* curve, the latter method will be addressed later. The *adaptiveThreshold()* function is useful for when the light varies across the image, as the additional information make the pixel better reflect its surroundings. This means the function can better preserve edges.

Function: *adaptiveThreshold*, gaussian method
The second method of *adaptiveThreshold()* is by the *gaussian* curve. In addition to the width of the neighborhood also the pixel's output value can be configured by two parameters: the impact of the gaussian curve and a constant subtracted post the gaussian calculation. A higher value of the gaussian curve parameter means the pixel's own value weigh more than the neighborhood and its original value is better preserved. The calculated value is then subtracted with a constant. By the *gaussian* method the function produced a similar result as with the *mean* method. An advantage to the *gaussian* method is its ability to smooth out larger noise that has been left over from the previous noise removal functions in chapter *6.3.3 Removal of noise*. This was achieved by the *gaussian* curve minimizing the influence of the noise and using a large neighborhood.

Function: *Canny()*
The function *canny()* does not work as previously explained methods. It displays the image's contrasts as thin lines, not as an area as the other three methods. As a head feature it can pick up many details in an image but an important issue is that the lines do not necessarily connect. For the output matrix to be adaptable with later functions the lines needs to be connected. In order to use the function *Canny()* additional image processing is required on the output matrix.

Resulting binary conversion method
The rugged result produced by the function *Threshold()* caused it to be disqualified at an early stage. Because *adaptiveThreshold()* produced acceptable results without further image processing needed to attain good results, the function *Canny()* was scrapped. This left the function *adaptiveThreshold()* with *mean* and *gaussian* method as the most viable options. Between the two methods the *gaussian* proved better by its bonus ability to also eliminate larger noise by itself. The final parameters used was a square neighborhood with the length and height of 19 elements. The subtracted constant was set to 3.

6.3.5 Enhancement of thin lines

Empiric studies has proven that the matrix sometimes ends up with the object's sharp corners chopped off, for example a squared object's corners was excluded. This issue can be solved by performing multiple iterations of *dilate()* on the matrix to extend the area of the object. The iteration is performed two times.

6.4 Creating region of interest and size test

The original matrix has in the been altered to only contain *high* and low values, which separate the object and the background from each other. Some scattered noise still remains and are also represented by high values, in the same way as the object. To determine which of the high value shape that is correct and represents the object the program needs a way to interpret the shapes individually and to assign them. Furthermore the program also need a sorting mechanism to single out the correct shape. Once the object's shape is obtained the program can analyze it in the size test.

6.4.1 Finding contours in the binary matrix

Currently the matrix contain a mix of high and low values and the program cannot distinguish one shape from another or address them in some way. The solutions to this revolves around a key function called *findContours()* which have the ability to find all high value contrasts within a binary matrix and create a way to address them. The function detects areas with high values that are encircled by high contrast and formulate a range of points that describes the areas' boundaries, called a contour. The advantage to use contours in the selected software is that there are many functions to use that can calculate heavy tasks with basic input parameters.

The information found by the function *findContours()* is stored by using vectors. A single contour's range of points are stored as elements within in a single vector, a so called a *vector of points*. To store multiple contours all the vectors of points are stored in an outer vector, creating a *vector vector of points*. This vector vector pointer works like a library of contours that describes the contours pixel positions in the matrix. Each contour can be reached by indexing the outermost vector, if you know which one to look for.

A disadvantage of the library is that the contours are not arranged in a systematic order. When *findContours()* is performed on a matrix an additional vector vector pointer is created that stores information about each contour's hierarchy attributes.

The hierarchy tells how the contours are set in relationship to each other: if they are inside or outside one another and in which order they are in if they share the same level. See chapter *4.3 Utilized features in openCV* for further explanation about the hierarchy system. To be able to use a specific contour in another function its index must be known to reach the contour in the library of contours. The key to get the index of a specific contour is to look for its hierarchic attributes from the beginning of the hierarchy's vector vector pointer til the end and along the way let an integer numerically keep track of how many vector of pointers has been analyzed. When the a contour is hit the integers value can be used as index in vector vector pointer of the library of contours, because it shares the same sequence of contours as the hierarchy. When the index is known a single contour can be selected, from the library of contours, to for example send the contour as an input to another function to calculate its area, or in another function to be manipulated.

6.4.2 Finding the object's outline

The remaining noise will be detected by *findContours()* and those contours needs to differentiated from the correct contour and sorted out. The undesired contours originates from noise and random structures found in the background and the elimination of the them can be done in a few different ways. Three of these methods are considered below and utilizes a contours' attributes such as their enclosed area, contour length and contour hierarchy level. All methods outputs a single contour that should depict the object.

The methods are visually observed by the use of *findContours()* sibling called *drawContours()*, see chapter *4.3 Utilized features in OpenCV*. The function can draw an input contour in any preferred colour onto an input matrix. This makes it possible in this case to see where in the matrix contours are found, how many of them and how precise they *findContours()* has detected them. With the help of *drawContours()* the three methods below has been visually examined by their reliability to pinpoint the correct contour and how well the object's boundary is depicted. The region within the correct contour will be referred to as *region of interest* (ROI).

Hierarchy
As mentioned earlier in chapter *6.4.1 Finding formations in the binary matrix*, hierarchy can be utilized to find contours by their relation to other contours. In this case, all the contours that does not have a parent are subjected to possibly be the object's contour. In general several contours fits this description and a further sorting mechanism is needed. The remaining contours are sent as input to the function *contourArea()* to calculate each contour's enclosed area. The function's output is compared and the largest area is assumed to represent the object and the rest are discarded. This method was controlled manually and proved reliable.

Length
A method considered was to sort out the correct contour by measuring the contours lengths using the function *arcLength()*. The trouble with measuring the contours length is that a contour's points can be approximated quite roughly in the earlier stage by *findContour()*. For example, a contour describing a circle's circumference could take on the shape of a zig-zag line instead of a smooth line.

The function *arcLength()* would then calculate the zig-zag line as of greater length, than if the line would have been more smooth and followed the circle better. Another function called *approxPolyDP()* can help to refine the contour's sometimes roughly allocated points, see chapter *4.3 Utilized features in OpenCV*. The function relocates deviating points and puts them along an approximated calculated line based of its neighboring points. By the looks of it, this function makes the contour considerably smoother but does not help *arcLength()* enough to give an accurate depiction of the object. This method was discarded.

Area
This method has already been mentioned above in the hierarchy method. The function *contourArea()* with a basic area comparison can alone be used as a method to determine the contour with the greatest area.

Resulting contour sorting method
The area method is the most straightforward and has proven in empirical studies to be sufficient enough on its own. A drawback is that it only works in a process where the program can assume only a single object is observed. In this case the project is only focused on a single object but to maintain the possibility to further develop the program for other purposes it is relevant to preserve the ability to evaluate multiple objects. In the case of multiple objects the hierarchy will ensure only the outermost contours are selected. It can prove quite difficult to achieve the same by depending on the *contourArea()* as the key function. In the case of two objects with different size the function *findContour()* might find an irrelevant contour within the bigger object's outmost contour. That contour could be larger than the second object's outmost contour, and thus faulty determine the irrelevant contour to be an object's contour. The conclusion is that the method utilizing hierarchy is the better choice in the long run.

6.4.3 Size test

Only one contour remain and the matrix is ready to be analyzed in the size test. It is difficult to approximate the physical size of an object from a sole matrix since the matrix it only have two dimensions while the object have three. The camera is however in a fixed position and the object is located almost at the same spot every time. This opens up the possibility to approximate a *relative size* as a portion of the matrix's area. The amount of pixels the object occupies have already been calculated by the function *contourArea()* in chapter *6.4.2 Finding the objects formation*. The total amount of pixels contained in a matrix is calculated as well by *contourArea()* by using a matrix filled with only high values as input. The occupied space by the object is set in portion to the whole matrix to give a relative size.

Finally the portion is tested if it is within in the range of 5-20% of the matrix total size. If the test is approved the program will continue the process towards the colour test and activate an I/O-pin on the hardware to light a diode. The diode neatly display the size test result without the need to look at the code. If the portion on the other hand is failed the program will try to reset itself by waiting for the object to be disposed of and for a new object to arrive. The disposal of the object will be explained in a later chapter.

6.5 Calculating colours in region of interest and colour test

After the size test has been cleared the program's next step is to prepare the matrix for the colour test and execute it. To solely calculate the object's amount of colours they need to be isolated from the object's environment. The result strived for is a black matrix with only the object left in colours. Once the isolation is done, the program will analyze each pixel's value if it fits into a few preset colour intervals to determine how much of the object consist of each colour. At last the calculated colour portions will weigh in at the colour test.

6.5.1 Isolating the object's colours by its contour

The process to produce a matrix with the object in colour and the surroundings portrayed in only low values is done in a few steps. In order to isolate the object a masked matrix is first created by filling a matrix with low values and then draw the object's contour in high values by the function the familiar function *drawContour()*. A input parameter to *drawContours()* is set to enable the ability to fill the contour with high values as well. This creates a masked matrix, see left image in figure 6.5. The binary matrix is then applied to the original matrix, see middle image in figure 6.5, by the Mat function *copyTo()*. The function layer one input matrix ontop of a second input matrix, and ignores the high values found in the top layer. This allows the values from the underlying matrix to emerge and by that the cropped matrix is finished, see the right image in figure 6.5.

Figure 6.5 *From left to right: Masked matrix, original matrix, cropped matrix.*

6.5.2 Finding the object's amounts of colour

Only the pixels representing the object remain in colour and the program can start to prepare for the colour test to obtain the quantity of a four predetermined colours. There colours are: green, blue, orange and yellow. Other colours can be added as well but only four was chosen to prove the concept. The four colours are predefined by intervals, describing the channel values required to define a pixel as a certain colour. So far the matrix values has been described by the standard BGR, which describe a hue by three channels. However, a simpler way to go about describing a colour when the intensity is the most interesting value is by the standard HSV (see chapter *4.1.2 HSV - Hue, Saturation, Value*), because the hue is only described by one channel. The earlier used function *cvtColor()* is used to convert the matrix from BGR to HSV. The initial idea to only describe a colour with one channel did not work entirely as intended and the result by using only the H channel came out flawed. A more accurate colour description was achieved by using all three channels in HSV. Even though this nullified the intended practical advantage of converting the matrix from BGR to HSV, the colour mixing is more comprehensible with HSV.

A short separate program was written to find each colour's interval values. The program captures and displays a binary matrix and six track bars. Each channel has a lower and upper track bar that can be drawn to change its value, the included colours are displayed by high values and the rest by low values. The binary presentation makes it easy to see when the whole shape of the object is included. With the six track bars the program can immediately display how a change of an interval value would affect the output. Practically the colours was found by capturing a matrix of a monochrome object of each colour and fiddle with the track bars until only the object's shape was visible. This eased the empiric studies to find appropriate channel intervals significantly.

With the intervals at hand the analysis can finally begin. The program can only analyze one predefined colour at the time and makes use of the function *inRange()* to examine which of the matrix's pixels fits the colour description. The output is a binary matrix with the current analyzed colour displayed in high values and the rest in low values. The amount of high valued pixels are calculated by the function *countNonZero()*. The quantity of high value pixels are set in portion to the pixel quantity contained in the whole matrix. The procedure is repeated as many times as there are predefined colours. In the end the greatest weighted portion is claimed as the dominant colour of the cropped matrix and is stored away for later use in the colour test. As can be seen below in *figure 6.6* four procedures exerted on the cropped image generate four correlating binary matrixes as in figure 6.7.

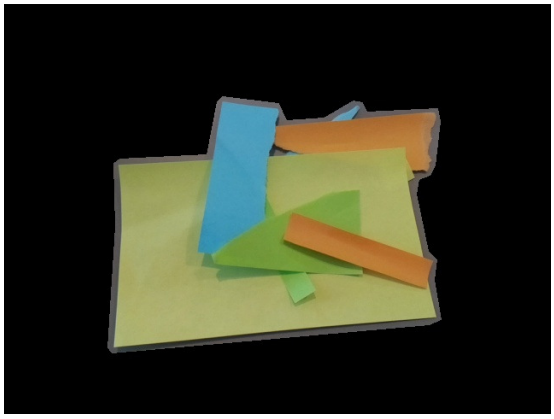


Figure 6.6 The cropped matrix.

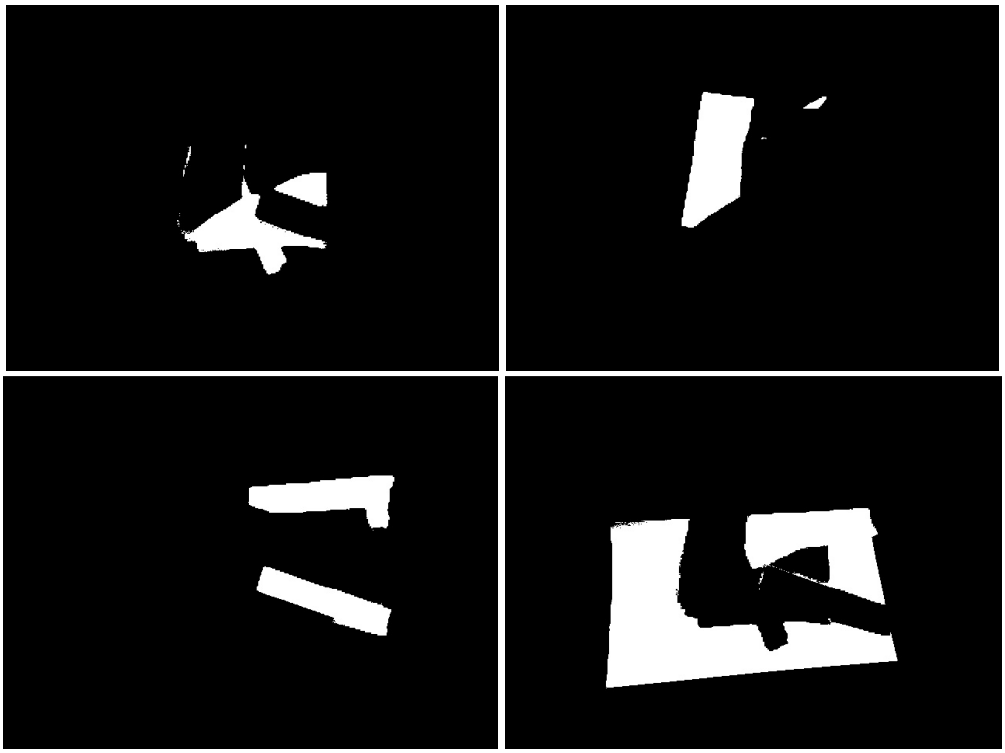


Figure 6.7 *From top left to bottom right: Binary matrices representing the areas of green, blue, orange and yellow.*

6.5.3 Colour test

The dominant portion of colour determined in the previous chapter can now be used as input to the colour test. The object will be approved in the colour test if the dominant colour's quantity exceeds 10% of the matrix's pixels. The error portion of 10% has been taken in consideration to expel small quantities of unintentional colours contained in the matrix that accidently match any of the predefined colours. If the colour is approved an I/O-pin is activated by the program to display the that the object has passed the colour test and as well a third diode to show that the object has passed both test and is OK. In theory an approved signal would have been sent to the decision box. If the object instead is failed the program will try to reset itself.

6.6 Reset of the program

To reach the reset point of the program the object has either cleared both the size and colour test or been failed in the size test. The main purpose of the program is done and the object shall be removed to allow the next object to enter the camera's field of view. At this point the program has theoretically transmitted an approve or fail signal to the decision box which next would remove the object on scene. The decision box could then tell the program that a removal has been made but since this is not the practical case the signal need a replacement. The signal instead comes from another run of the motion detection loop used in *6.2.2 Wait sequence for new object*. The object is removed by hand and when the motion detection loop notices a difference of 10% the program starts over.

7. Result

In this chapter the resulting program of the implementation will be presented by discussing the results from the evaluation test, on said program. The test was performed in order to ensure the program's functionality and whether the requirements of this thesis were met (see *1.3 Goals*).

7.1 Results from evaluation test

The resulting program, developed during this project, performs well and has fulfilled all requirements stated in *1.3 Goals.* The documentation from the evaluation test can be viewed in its entirety in *Appendix A.* The summarized results can be seen in figure 7.1.

Categories	Required accuracy	Measured accuracy
Size	90%	100%
Colour	90%	94.4%
Continuity	Yes	Yes

Figure 7.1 *Summarized results of the evaluation test.*

The only deviation that occurred during the test (see "Measured accuracy - Colour" in figure 7.1) can be explained by variables in the hardware settings, which were not taken into account. The camera hardware used as the visual sensor in this project (see *3.1.3 Resulting Hardware platform*) is programmed, by default, to adjust the amount of light permitted into the optical sensor. This is a built in feature that is present in most modern cameras to avoid over and underexposure in images, commonly referred to as *Auto White Balance* [67]. The deviation in the evaluation test occurred due to the the yellow, large test object reflecting too much light back into the camera which automatically lowered the amount of light permitted in order to balance the exposure. This automatic adjustment of light led to a shift in the perceived colour, causing the object to fall outside the "allowed" range of colours, hence failing it. As can be seen in figure 7.2 the small and medium sized, yellow test objects produced the expected result while the larger ones failed. The approved ranges are set using medium sized objects of the correct colour, making this deviation a previously not discovered occurrence. This is a risk connected to bright colours and having set ranges, in which colours are approved or not, due to the camera having auto white balance enabled. A solution would be to turn off this feature on the camera and supply constant light conditions at all times and recalibrate the ranges accordingly.

Test case	Size	Hue	Expected size result	Actual size result	Expected colour result	Actual colour result	Pass (green) Fail (red)
28	S	Y	F	F	A	A	
29	S	Y	F	F	A	A	
30	S	Y	F	F	A	A	
31	M	Y	A	A	A	A	
32	M	Y	A	A	A	A	
33	M	Y	A	A	A	A	
34	L	Y	F	F	A	F	
35	L	Y	F	F	A	F	
36	L	Y	F	F	A	F	

Figure 7.2 *Yellow test results deviating from expected result during the evaluation test (see* 2.3.1 Evaluation test*).*

7.2 Displaying the results

When running the program, with a monitor connected to the hardware, the resulting image matrix is displayed in different stages of processing. The status of the program and the relevant results are printed in the terminal window and displayed on the monitor. The different stages of processing and colour portions are mainly displayed for debugging purposes during the development of the program. The final cropped image, in BGR colours, are shown in figure 7.3 alongside the printout in the terminal. The final outcome of the object is communicated using the three LED indicators, see figure 7.4, lighting up if the object is approved in the tests of size and colour.

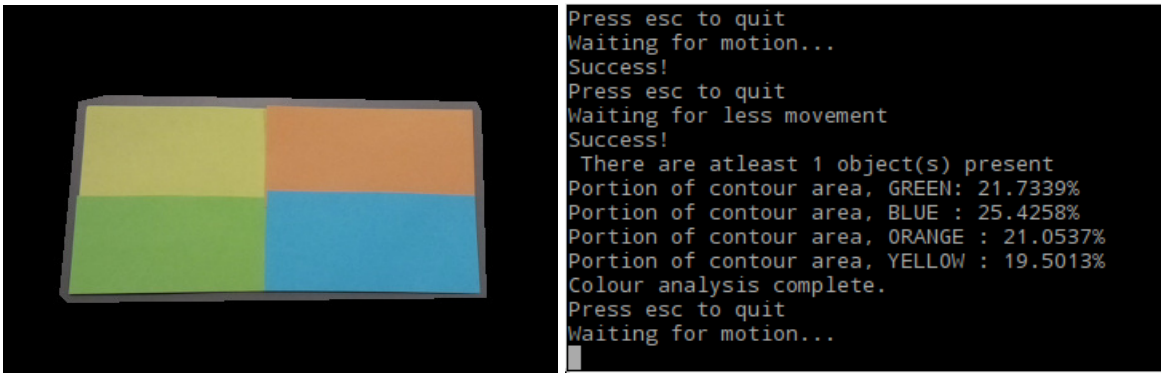


Figure 7.3 *The results of the colour analysis performed on the image to the left is displayed in the terminal window to the right.*

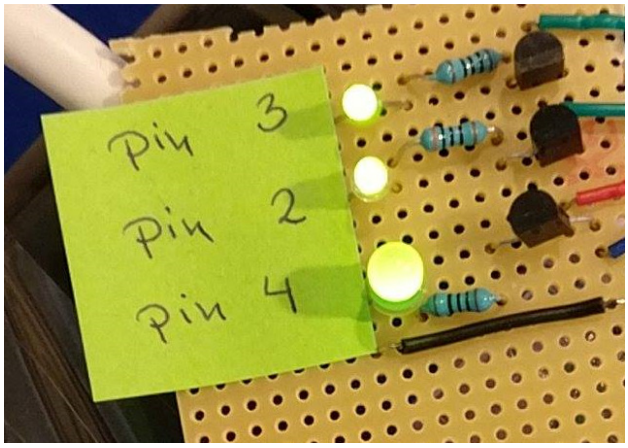


Figure 7.4 *LED used to indicate if test results are approved. "Pin 3" indicating size result, "Pin 2" indicating colour result, "Pin 4" indicate the final verdict approving the object's attributes.*

7.3 Discussion

The development of the program has constantly presented itself with hard decisions, leading to alternative solutions left unexplored. Furthermore, the size of the OpenCV library is both a great benefit and a source of eternal strife. The fact that the library could solve anything requested of it, within reason, was really a good feature. On the other hand, the library often had many different solutions to each problem which led to frustration when not all pros and cons were obvious during implementation. Drawbacks to a particular function could show itself much later in the progression of the program, which created many instances of rewriting functions that were considered sound. However, this is a common problem when building applications and programs in a new environment and would have been a problem whichever software library used while developing the program. Learning was always a large part of the project, and with learning comes mistakes to learn from, i.e. experience.

A constant issue during the development was light consistency in the environment around the workstation. The fact that the equipment is located on a desk close to an east facing window with bright buildings reflecting different amounts of light depending on time of day, created issues. The windows automatic shades did not improve the situation, going up and down seemingly at will. The testing of functions became difficult since the result differed depending on the time of day and present weather conditions outside. A permanent solution to this problem would have been to build some sort of shielding to limit the amount of light permitted into the camera's field of vision. The conditions are briefly mentioned in *2.1 Pre-study* when stating that the proof of concept should be able to meet the required specifications "*during ideal conditions*". A final solution was never really found for this particular problem. A result of this issue was that all testing was performed after calibrating the ranges and values of the program to fit the situational conditions in the environment, a problem that has to be solved in order to have a fully automated process.

The fact that OpenCV does not have any official technical support raises the question whether "free" really is the cheapest alternative in the long run. The problem lies in that the community that is keeping OpenCV alive with updates, tutorials and examples could simply disappear.

There are no rules or guidelines to when or if a question should be answered in a user based forum and web pages with information can be taken down without notice since most of them are run by organisations not driven by monetary gain. This is where a licence comes in handy, since when a contract is struck, the buyer/user knows exactly how many years the support will be available through service agreements.

7.3.1 Ethics and sustainability

Ethically this project does not conflict with anything new, since it is not new technology. Therefore the same can be said about this project as with all other technology that aim to automate a process and diminish human error/involvement. There is a possibility that some work opportunities are lost due to this automatisation, but it could on the other hand create jobs in terms of service and updates.

An argument can be made that this kind of technology can be weaponized and used for destructive purposes. However, this fact is true for almost all technology developed today.

Concerning the sustainability of a product that can be produced using this research, it provides opportunities to automate processes, and in that regard save resources in other cases spent on transporting workers and vehicles. Diverting these resources into larger problems that exist in society would be a possible outcome from making use of a product like this. A centralized control facility would be able to better manage resources and minimize production problems by monitoring huge amount of locations, that human employees would not be able to reach physically.

8. Conclusion

With the gathered results from the final program, it is clear that the prototype (see *1.1 Background*) can be replaced with a cheaper alternative. The program developed during this project is well suited for continuous analysis of objects placed in the field of view of the camera, which clearly meets the specifications listed below:

The stated specifications stated in *1.3 Goals* are repeated below:

- Is there a way to make the program run continuously for a trial of 54 tests without failing?
- Can a size test be developed to give the correct output in 90% of the cases during a test of 54 objects?
- Can a colour test be developed that can differentiate between four colours and determine the dominant one in 90% of the cases during a test of 54 objects?

The results from the evaluation test (see *2.3.1 Evaluation test*) can be seen in *Appendix A*. The conclusion to be drawn from the results is that with a 94.4% colour accuracy and a 100% size accuracy, while never terminating the program, is enough to provide the proof of concept this project was started to achieve (see *1.1 Background*).

OpenCV as the vision library of choice, is more than large and powerful enough to build an program of this type. The documentation supporting the library is well written and easy to understand. The wealth of examples and explanations in the OpenCV community have been able to answer any and all questions that arose during the development process, and has been invaluable as a source of knowledge.

The program has been developed, exclusively, using open source software. This points to the fact that further development would most likely lead to a stable and cheap solution that could be incorporated into a commercial product, with no attached licensing fees. The argument can also be made, concerning the hardware, that the Raspberry Pi, while being a good platform during development, is too costly as the ultimate hardware in production. The hardware performance can definitely be lower, than what the Raspberry Pi supplies, and still perform at a satisfying speed. Thus, cutting the cost of a final product even further.

8.1 Further development

The current program can only analyze one object at the time by its two dimensional size and only determine the object to be one of four colours. The range of colours is simple to expand and the approved size interval easily changed in the code. A few ideas that could further develop the functionality of the program are:

- Document the amount of detected, passed and failed items for statistics by the use of the size test and colour test.
- Analyze an object of multiple colours by a more complexed colour portion program.
- Analyze multiple objects in camera's field of view.
- Expand the reset of the program to notice when an object has not been removed as anticipated.

Applications that demands more time to develop:

- Implement shape and pattern recognition, for example to find defect items in a production line or for scanning barcodes.
- An algorithm that can learn a colour by analyzing several objects/images with the right colour.

References

All figures and images are created by the authors with the sole exception of figure 4.10 which is referenced in the list at **[41]**.

[1] Investorwords. *Proof of concept*. Available at: www.investorwords.com. Last read: 2017-06-04.
[2] Di Justo, P. *Raspberry Pi or Arduino Uno? One Simple Rule to Choose the Right Board.* Available at: www.makezine.com. Last read: 2017-06-04.
[3] Bourque, B. *Arduino vs. Raspberry Pi: Mortal enemies, or Best Friends?* Available at: www.digitaltrends.com. Last read: 2017-06-04.
[4] Suehle, R. *Should I get an Arduino or a Raspberry Pi?* Available at: https://opensource.com/. Last read: 2017-06-04.
[5] Arduino. *Arduino Products.* Available at: www.arduino.cc. Last read: 2017-06-04.
[6] Arduino. *Arduino Due.* Available at: www.arduino.cc. Last read: 2017-06-04.
[7] Arducam. *ArduCAM USB Camera Shield Released*. Available: www.arducam.com. Last read: 2017-06-04.
[8] Arduino. *Language Reference.* Available at: www.arduino.cc. Last read: 2017-06-04.
[9] Arduino. *Access the Online IDE.* Available at: www.arduino.cc. Last read: 2017-06-04.
[10] Raspberry Pi. *FAQS*. Available at: www.raspberrypi.org. Last read: 2017-06-04.
[11] Wikipedia (ENG). *Raspberry Pi.* Available at: https://en.wikipedia.org. Last read: 2017-06-04.
[12] Elinux. *RPi Low-level peripherals - Power pins*. Available at: http://elinux.org. Last read: 2017-06-04.
[13] Suehle, R & Callaway, T. *Raspberry Pi Hacks - Tips & Tools for Making Things with the Inexpensive Linux Computer.* O'Reilly: Sebastopol, 2014. Page 26.
[14] Long, S. *Jessie is here.* Available at: www.raspberrypi.org. Last read: 2017-06-04.
[15] Wikipedia (ENG). *Linux*. Available at: https://en.wikipedia.org. Last read: 2017-06-04.
[16] Raspberry Pi. *Camera Module*. Available at: www.raspberrypi.org. Last read: 2017-06-04.
[17] Adaptive Vision. *Software for Machine Vision Engineers*. Available at: www.adaptive-vision.com. Last read: 2017-06-04.
[18] Adaptive Vision. *Videos*. Available at: www.adaptive-vision.com. Last read: 2017-06-04.
[19] OpenCV. *Adaptive Vision Studio 4.3 Lite*. Available at: http://opencv.org. Last read: 2017-06-04.
[20] Adaptive Vision. *Licensing*. Available at: www.adaptive-vision.com. Last read: 2017-06-04.
[21] MVTec Software GmbH. *Product Information*. Available at: www.mvtec.com. Last read: 2017-06-04.
[22] MVTec Software GmbH. *Integrated Development Environment for Machine Vision*. Available at: www.mvtec.com. Last read: 2017-06-04.
[23] MVTec Software GmbH. *HALCON – The power of machine vision.* Available at: www.mvtec.com. Last read: 2017-06-04.
[24] Halcon. *Halcon the Power of Machine Vision - Installation Guide 7.1.* MVTec Software GmbH: München, 2008. Page 2.

[25] Brahmbhatt, S. *Practical OpenCV - Hands on Projects for Computer Vision on the Windows, Linux and Raspberry Pi Platforms.* Springer Science+Business Media New York: New York, 2013. Page 12.
[26] OpenCV. *About*. Available at: http://opencv.org. Last read: 2017-06-04.
[27] Wikipedia (ENG). *Berkeley Software Distribution*. Available at: https://en.wikipedia.org. Last read: 2017-06-04.
[28] Adaptive Vision. *About Adaptive Vision*. Available at: www.adaptive-vision.com. Last read: 2017-06-04.
[29] MVTec Software GmbH. *Services & Solutions*. Available at: www.mvtec.com. Last read: 2017-06-04.
[30] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 3.
[31] Encyclopædia Britannica. *Pixel*. Available at: www.britannica.com. Last read: 2017-06-04.
[32] Brahmbhatt, S. *Practical OpenCV - Hands on Projects for Computer Vision on the Windows, Linux and Raspberry Pi Platforms.* Springer Science+Business Media New York, New York: 2013. Page 24.
[33] Minnick, J. *Web Design with HTML5 & CSS3 - Comprehensive Eighth Edition.* Cengage Learning, Boston: 2017. Page 148.
[34] NXP Philips. *Introduction to graphics and LCD technologies.* Available at: http://www.nxp.com/wcm_documents/techzones/microcontrollers-techzone/Presentations/graphics.lcd.technologies.pdf. Last read: 2017-06-04. Page 7.
[35] Pajankar, A et.al. *Raspberry Pi: Marking Amazing Projects Right from Scratch!* Packt Publishing, Birgimhamn: 2016. Page 128.
[36] Rhyne, T-M. *Applying Color Theory to Digital Media and Visualization.* CRC Press, Durham: 2017. Page 59f.
[37] OpenCV. *Changing Colorspaces*. Available at: http://docs.opencv.org/3.1.0. Last read: 2017-06-04.
[38] Opara, E & Cantwell, J. *Best Practices for Graphic Designers, Color Works - An Essenatial Guide to Understanding and Applying Color Design Principles.* Rockport Publishers, Beverly: 2014. Page 189.
[39] OpenCV. *Basic Thresholding Operations*. Available at: http://docs.opencv.org/2.4. Last read: 2017-06-04.
[40] Mathworks. *Multiple object tracking in binary images*. Available at: https://se.mathworks.com. Last read: 2017-06-04.
[41] OpenCV. *Image Thresholding*. Available at: http://docs.opencv.org. Last read: 2017-06-04.
[42] CPlusPlus. *Arrays*. Available at: www.cplusplus.com. Last read: 2017-06-04.
[43] CPlusPlus. *Vector*. Available at: www.cplusplus.com. Last read: 2017-06-04.
[44] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 33
[45] OpenCV. *Mat - The Basic Image Container.* Available at: http://docs.opencv.org/2.4. Last read: 2017-06-04.
[46] OpenCV. *Basic Structures.* Available at: http://docs.opencv.org/2.4. Last read: 2017-06-04.
[47] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 58f.

[48] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 117f.
[49] OpenCV. *Eroding and Dilating.* Available at: http://docs.opencv.org/2.4. Last read: 2017-06-04.
[50] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 135f.
[51] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 138f.
[52] OpenCV. *Miscellaneous Image Transformations.* Available at: http://docs.opencv.org/2.4. Last read: 2017-06-04.
[53] Wikipedia. *Normal distribution.* Available at: https://en.wikipedia.org. Last read: 2017-06-04.
[54] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 151f.
[55] Brahmbhatt, S. *Practical OpenCV - Hands on Projects for Computer Vision on the Windows, Linux and Raspberry Pi Platforms.* Springer Science+Business Media New York: New York, 2013. Page 56.
[56] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 234f.
[57] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 236f.
[58] OpenCV. *Contours Hierarchy*. Available at: http://docs.opencv.org/trunk. Last read: 2017-06-04.
[59] Brahmbhatt, S. *Practical OpenCV - Hands on Projects for Computer Vision on the Windows, Linux and Raspberry Pi Platforms.* Springer Science+Business Media New York, New York: 2013. Page 70.
[60] OpenCV. *Structural Analysis and Shape Descriptors*. Available at: http://docs.opencv.org/2.4. Last read: 2017-06-04.
[61] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 245f.
[62] OpenCV. *Structural Analysis and Shape Descriptors*. Available at: http://docs.opencv.org/2.4 Last read: 2017-06-04.
[63] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 57.
[64] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 49f.
[65] Bradski, G & Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Sebastopol: 2008. Page 65.
[66] Wikipedia (ENG). *Image noise*. Available at https://en.wikipedia.org. Last read: 2017-07-20.
[67] Raspberry Pi. *Camera Module*. Available at: www.raspberrypi.org. Last read: 2017-06-12.

Appendix A – Table of evaluation test results

Test case	Size	Hue	Expected size result	Actual size result	Expected colour result	Actual colour result
1	S	P	F	F	F	F
2	S	P	F	F	F	F
3	S	P	F	F	F	F
4	M	P	A	A	F	F
5	M	P	A	A	F	F
6	M	P	A	A	F	F
7	L	P	F	F	F	F
8	L	P	F	F	F	F
9	L	P	F	F	F	F
10	S	G	F	F	A	A
11	S	G	F	F	A	A
12	S	G	F	F	A	A
13	M	G	A	A	A	A
14	M	G	A	A	A	A
15	M	G	A	A	A	A
16	L	G	F	F	A	A
17	L	G	F	F	A	A
18	L	G	F	F	A	A
19	S	B	F	F	A	A
20	S	B	F	F	A	A
21	S	B	F	F	A	A
22	M	B	A	A	A	A
23	M	B	A	A	A	A
24	M	B	A	A	A	A
25	L	B	F	F	A	A
26	L	B	F	F	A	A
27	L	B	F	F	A	A

Test case	Size	Hue	Expected size result	Actual size result	Expected colour result	Actual colour result
28	S	Y	F	F	A	A
29	S	Y	F	F	A	A
30	S	Y	F	F	A	A
31	M	Y	A	A	A	A
32	M	Y	A	A	A	A
33	M	Y	A	A	A	A
34	L	Y	F	F	A	F
35	L	Y	F	F	A	F
36	L	Y	F	F	A	F
37	S	W	F	F	F	F
38	S	W	F	F	F	F
39	S	W	F	F	F	F
40	M	W	A	A	F	F
41	M	W	A	A	F	F
42	M	W	A	A	F	F
43	L	W	F	F	F	F
44	L	W	F	F	F	F
45	L	W	F	F	F	F
46	S	O	F	F	A	A
47	S	O	F	F	A	A
48	S	O	F	F	A	A
49	M	O	A	A	A	A
50	M	O	A	A	A	A
51	M	O	A	A	A	A
52	L	O	F	F	A	A
53	L	O	F	F	A	A
54	L	O	F	F	A	A

S = Small, M = Medium, L = Large
P = Pink, G = Green, B = Blue
Y = Yellow, W = White, O = Orange
A = Approved, F = Failed