Problem Solving and Engineering Design part 3

ESAT1A1

Max Beerten
Brent De Bleser
Wouter Devos
Ben Fidlers
Simon Gulix
Tom Kerkhofs

Counting and recognizing non-moving objects by means of image processing

PRELIMINARY REPORT

Co-titular
Tinne Tuytelaars

Coaches
Xuanli Chen
José Oramas

ACADEMIC YEAR 2018-2019

Declaration of originality

We hereby declare that this submitted draft is entirely our own, subject to feedback and support given us by the didactic team, and subject to lawful cooperation which was agreed with the same didactic team.
Regarding this draft, we also declare that:

1. *Note has been taken of the text on academic integrity (https://eng.kuleuven.be/studeren/masterproef-en-papers/documenten/20161221-academischeintegriteit-okt2016.pdf).*
2. *No plagiarism has been committed as described on https://eng.kuleuven.be/studeren/masterproef-en-papers/plagiaat.*
3. *All experiments, tests, measurements, …, have been performed as described in this draft, and no data or measurement results have been manipulated.*
4. *All sources employed in this draft – including internet sources – have been correctly referenced.*

Contents

List of Figures

1 Introduction

Digital image processing has been a crucial part of the current digitalisation movement. From industrial machinery to customer amusement, the vision of computer-aided systems has become a given for most users. While image alteration and manipulation remain a core part of this field of study, nowadays other image related problems are being solved by artificial intelligence (AI). Most were considered to be an important part of digital image processing. Feature extraction, especially, has seen a surge in AI based methods. While being a great way for unravelling many problems, AI mostly provides general solutions. Certain cases are solved more efficiently by specific schemes. Such is the case with object counting: while deep learning algorithms need a big data set as training material, standard image processing only requires the image itself.

This paper's aim is to design a system which can count objects, without the use of AI, in an industrial environment. Such a system might be used on a production or packaging line. The specific explanation of this problem is given in section 2: Problem Description.
Regardless which way a method processes images, it needs a visual source. In this paper the focus is on live object counting, which is only possible with a camera. Evidently, the choice of hardware greatly impacts the methods that can be used. This choice will be covered in section 3.1: Hardware.
By far the most important part of this task is the algorithm by which the items in the picture will be numbered. Classically, object counting algorithms have a standard group of steps: filtering, converting to an intensity matrix, edge detection, converting to a binary matrix, boundary boxing and the counting itself. These segments don't have a fixed order and can occur multiple times in the final method. Most of these steps can also be approached in different ways. A wide range of possible filters, kernels, edge detection methods, etc. exist, which all have their benefits and drawbacks (Dirac, 1981). These choices will be discussed in section 3.2: Software.
These methods, while being the core of the solution, are fairly simple to implement with the use of libraries or built-in functions. Creating the system in this simpler way has the advantage that the algorithms themselves don't need to be understood. Which gives a clearer picture of the idea behind the process. Such an implementation is thus included in the report in section 4: Research. Since the self-implemented system isn't fully functional, section 4 can give an idea of the way the project's software side will continue to be developed.
From here on out the report contains a clearer and more practical picture of the continued work in section 5: Further planning. The financial side of the project is discussed in more detail in section 6: Budget management. In section 7: Course integration a list of courses will be given, which were used to create the current version of the system. Lastly the report will be summarized in section 8: Conclusion, which will run down the whole process, and point out some useful applications.

2 Problem Description

The object counting system described in this report is capable of counting non-moving objects in a basket. The project has to be based on a depth sensor and/or a camera. These objects can vary in shape, size and colour. Thus, the colour of both the basket and its contents are free from restrictions. In the primary stage of this paper, not all these variables are taken into account. The system is required to count the simplest objects, being rectangles, cylinders and circles, all with a uniform colour. If possible, the circumference of these objects can be outlined and measured as shown in Fig. 1.
All of this is done in real-time and with a budget of €250.

Figure 1: Example of a possible result.

3 Hardware

The hardware to create a system as described above, is not complicated. In essence, it consists of a computer, a camera and a cable to transfer the data between the latter two. However, for a working recognition system, more parameters in prospect to the visual setup need to be defined. Each of these elements is discussed in the following section.

3.1 Hardware components

Choosing the camera is a vital primary component in this project. If chosen poorly, it can fiercely limit the outcome of the final algorithm. There are three main options for visual input: an ordinary webcam, an industrial camera or a camera with built-in depth sensors. Each with its pros and cons. A webcam is cheap and readily available but does not assure good image quality and easy access to its data. A camera for industrial usage is rather expensive, especially with a budget of €250. Industrial grade options which are cheap enough do exist, but these models deliver their images in greyscale. This greatly limits the possible methods which can be used. Thirdly, the depth sensing cameras are available in a reasonable price range and deliver, overall, good quality data. Moreover these models have the added benefit of depth sensor which, in contrast to the previous option, adds more possible ways to solve the problem.

Having considered all of the above, the best option is the latter one. More specifically, the system described in this report is based on a Kinect V2 from Microsoft. This camera has a color lens with a resolution of 1920 by 1080 pixels and a corresponding field of view of 84.1° by 53.8°(Smeenk, 11 Mar 2014). The high resolution ensures an accurate matrix representation of the real image. Each color frame pulled from the Kinect V2 is represented by an array structure of 1080x1920x3. Every element corresponds with a pixel of the image and varies between 0 and 255. Obviously it can be separated into three different matrices each belonging to $\mathbb{R}^2$ and based on a different colour: red, green or blue. Next to the colour camera, the Kinect also possesses a depth sensor. An infrared projector and camera make this possible(Jiao, Yuan, Tang, & Wu, Nov 2017). It provides a 424x512 array making the depth image one of roughly 200000 pixels. The field of view of this function is 70.6° by 60°. Note that the depth camera provides data about parts of the environment that the color camera does not see, and vice versa. When the computer reads the depth data, every number in the matrix represents a distance in millimetres. Obviously there are some restrictions. This technology only provides correct information if the object is at a distance located in between 0.5m to 4m. This has to be taken into account for further implementation of this paper.

As second element of hardware the computer has a less important role. Preferably, OSX isn't used as operating system for this application, because the Kinect drivers do not exist for Macintosh computers. If the reader has a Mac, problems can be avoided by running either Windows or Ubuntu on a virtual machine. The algorithm should run in an acceptable time frame on every machine.
Thirdly, a brief word on the necessary transfer cable. Since a depth sensing camera is used, two types of data (depth and color) need to be transferred. The Microsoft OEM Kinect Adapter makes this possible. The special adapter is the only available option and consists of two general parts. One part for delivering current to the camera and the other to transfer both types of data to the connected computer.

3.2 Visual setup

To conclude this section, a brief elaboration on the visual setup. A few variables need to be declared and some restrictions of the system need to be adressed.
The option to only use one camera is made, this limiting the increasing difficulty of the algorithm. When using more cameras, thus obtaining pictures from different angles of the chosen viewing field, the data of all needs to be integrated and a coherent output has to be developed. This requires more intense matrix merging and in the scope of this project, one camera suffices. When using only one camera it is important to have an idea where this camera is located in comparison to the objects. Utilizing a tripod with an extension beam, the camera is placed in the middle of the inspected viewing field. This way, distortion is avoided the most and the depth sensor will return the most useful data. A different vital parameter is the angle of the camera in relation to the surface containing the objects. Due to the ease of processing the data, a parallel stance of the Kinect in respect to the surface is prefered.

4 Software

There are a lot of options when it comes to software and a wide range of different algorithms for image processing exist. The diagram in Appendix A shows a couple of different methods. But keep in mind that there is no 'right way' to count the number of objects in an image

4.1 Analysing different methods

Analysing RGB images is a creative process. Different approaches have their own advantages and disadvantages. The only general ideas that are common throughout most algorithms are:

- Converting the RGB image to greyscale
- Run filters over the image to remove noise
- Transforming the image to a binary image

In the next scope, three general methods are featured and briefly discussed. Each was investigated in prospect of this paper.

4.1.1 Method 1: Thresholding

The image first passes through an threshold algorithm. After that there is an algorithm to find the edges of an object.

This method is the most simple and straightforward to implement. the first step. Either the image can be converted to a greyscale one which is analysed or the image can be analysed in its three different spectra (red, green and blue) which are combined afterwards. In the next step the array is passed trough a thresholding algorithm with a pre-defined threshold value. The output is a binary matrix. Thus, this array only has 0's and 1's as elements, representing the colours black and white, respectively. When in possession of a truly black and white image, a simple edge detection program is run which makes the edges visible.

The key to solving the problem in this specific scheme is writing code that finds the threshold value based on environmental parameters. An example of such an algorithm is the imbinarize function from the processing toolbox (*Mathworks*, 2018). This function is based on the Otsu's method. An example of a program that uses this function can be found in Appendix B (?, ?).

Advantages: It's an easy and fast algorithm.

Disadvantages: With a pre-defined threshold value it just classifies pixels based on colour. A dynamic value is required.

4.1.2 Method 2: Edge detection

This method is the opposite of method 1. It starts with finding edges and than a threshold algorithm at last

The second method tackles the colour analysis in the opposite order than the first method, as it commences with an edge detection algorithm. Since the input image is still very complex, it is first converted to a filtered greyscale image. Still, this edge detection is more comprehensive than the corresponding code from method one. The output is still a greyscale image, contrary to the binary array the reader might expect. This is followed by some thresholding code with a pre-defined threshold value. The current image is now represented by a matrix where the edges are outlined using binary elements.

Advantages: It detects all kinds of objects, not based on colour or shape.

Disadvantages: The boundary between different objects needs to be clear for this to work.

4.1.3 Method 3: Background filter

This method compares the image to an image of the background and uses that to threshold the image and finding the outline of objects.

The third way takes a different approach to solving the analysis of the colour image. Instead of looking at the objects, the algorithm looks for the background. This is possible in two different ways and both make a compromise in functionality. The first way needs a picture of the empty background without any objects. The second way looks for a pre-defined range which contains more pixels than other ranges. The compromise of the second named perspective is the fact that there can't be too much objects in the image. After applying filters the program loops through the image pixel by pixel. This necessary but time consuming loop checks if the pixel on the image is more or less the same as the corresponding pixel of the background image. If located within a pre-determined range: that element of the array gets classified as background. The consequence is that the output is a binary image with clear-cut objects. An example of this implementation can be found in Appendix C.

Advantages: It is very good in detecting objects, not being based on colour or shape.

Disadvantages: There needs to be an image of the empty background or the image need to be a big part of your image and has to be consistent. On top of that the lighting conditions and shadow play a big part.

4.2 Choice of method

After comparing these different schemes, the second method comes out as the better of the three. See Fig. 2 for the comparison.
The actual code of this function can be found in Appendix D.

4.2.1 Step 1: Converting to greyscale

The first step, as discussed above, is to convert the image to greyscale (Unknown, n.d.-a). This is easily done by calculating a weighted average of the values of all three red, green and blue matrices as shown in the following equation.

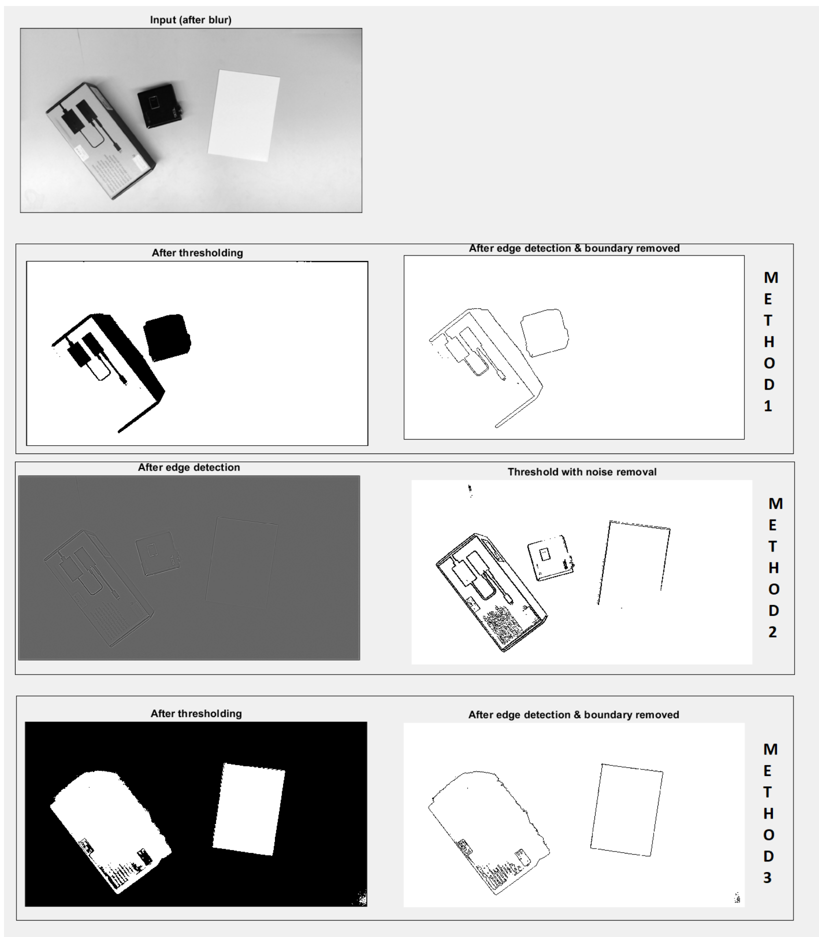


Figure 2: A comparison between the 3 different methods.

$$greyscale\_image(row, col) = 0.2989 * RED + 0.5870 * GREEN + 0.1140 * BLUE \tag{1}$$

The used coefficients add up to 1. This has to be the case, if else some values could exceed the 0 to 255 range. All these values $greyscale\_image(row, col)$ form the new image.

4.2.2 Step 2: Filtering the image

Before running the image through an edge detection algorithm, two filters are applied. Both blur the image to an extent such that noise after edge detection is considerably reduced. This effect is visualised in Fig. 3. Firstly a Gaussian blur(Unknown, n.d.-b) is applied. Most filters are a convolution of a kernel with the image. For a Gaussian blur the kernel G, found below, is used. This is just a weighted average. This has as consequence that the elements centred around the main pixel have bigger weights than those at the edges.

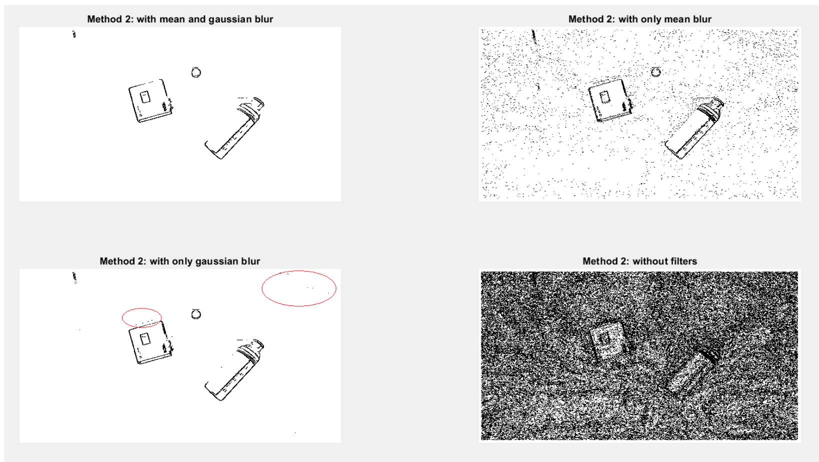


Figure 3: A comparison with the use of filters.

$$G = (1/159) * \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \tag{2}$$

The second blur is a mean blur (R. Fisher & Wolfart, n.d.-b). This is almost the same, the actions are just done with a different kernel. This kernel calculates the average of the values around the pixel.

$$M = (1/9) * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3}$$

Note that both G and M have a norm of 1. If this wasn't the case, pixel values of the filtered image could exceed the boundary values of 0 to 255 for uint8 numbers.

4.2.3 Step 3: Edge detection

After both filters are applied, the image is ready to be run through an edge detection algorithm(R. Fisher & Wolfart, n.d.-a). This algorithm is on itself also a filter with kernel given by the matrix L shown below. It calculates the *spatial derivative* or in simpler words, it highlights regions of rapid intensity change.

$$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{4}$$

Note now how the kernel uses the pixels next to the evaluated pixel to see how much intensity changes. If the image wouldn't have been filtered before convolution with L, more 'edges' would have been

drawn because more irregular areas exist.
Note also how this convolution returns a new image which can have negative values for its pixels. The more negative the value, the darker the image. To make this possible, a different number type is used.

4.2.4 Step 4: Threshold

The threshold algorithm, used in the following step, is based on this feature. This algorithm runs through the whole matrix and assigns each value to either a 0 or a 1. It decides this by assessing if the current value is either smaller than or bigger than a threshold value, respectively. After conducting multiple experiments and testing, a threshold value of 2 seems to do the trick. A more dynamic way of determining this value may be developed in the next weeks. After applying this edge detection, the matrix becomes a binary image with only the edges in white. Based on these edges, it is possible to outline the objects and count them. More on these functionalities can be found in section 4: Research.

4.3 Analysis Depth Sensor

Using only the RGB image does have some shortcomings. For example, it is rather difficult to distinguish an object from its shadow, a multicoloured object could be seen as multiple different objects and a decent amount of reflection could make an object undetectable. These are some of the reasons why enrichening the object counting algorithm with the usage of a depth sensor is advised. Like featured in the section about the hardware, each element of the input data represents a distance in millimetres. Firstly the code should be able to provide a clear difference in height between the objects and the background using the depth data. This is followed by a filter to get rid of the existing noise. Lastly, the filtered matrix will be used to detect the edges of the objects and thus detect the items themselves. The code that accompanies this description, can be found in appendix E.

4.3.1 Step 1: Detection of the difference in height

The goal is to see a clear difference between the objects and the background. This can be achieved in different ways: it is possible to use a threshold and label everything closer than this predetermined distance as an object. A disadvantage of this method is that the value will be different for various vertical positions of the Kinect v2. In addition, the image of the sensor contains some noise. For example: a picture of a large flat table will not be viewed as an equidistant surface. The elements of the matrix will be different. Another, and more preferred, method would be to use a Sobel-Feldman operator (Sobel, 2014). This operation approximates the gradient in each of the points of the matrix and gives an idea where there is a sudden difference in height (thus where there might be an object). It works by convolving 2 kernels with the image matrix A to become G_x and G_y: respectively one for the horizontal and one for the vertical change in height:

$$
\begin{aligned}
G_x &= \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A \\
G_y &= \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \\
G &= \sqrt{G_x^2 + G_y^2}
\end{aligned}
$$

In the last equation, G is the magnitude of the total gradient as well as the value inserted in the new matrix. The result can be seen in Fig. 4.

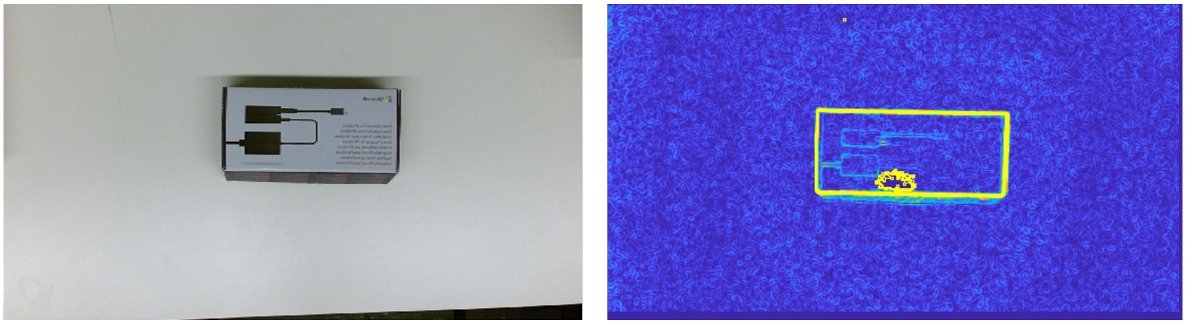


Figure 4: The original RGB image (left) and the image after the Sobel-Feldman operator (right)

4.3.2 Step 2: Filtering of the noise

After adding al the different magnitudes of the gradients to an array, some anomalies still exist. There can be some impossible elements, including points that seem to be further away than the basket or fluctuations in areas that are supposed to be flat (noise). The simplest way to solve this problem would be to use a maximum and minimum threshold: The maximum threshold can be a value that is further away than the basket. These values are impossible and the corresponding values in the matrix can be set to zero. The minimum threshold can be decided by empirical research. Values lower than this value can be seen as noise and thus can be set to zero. The final result can be seen in Fig. 5.

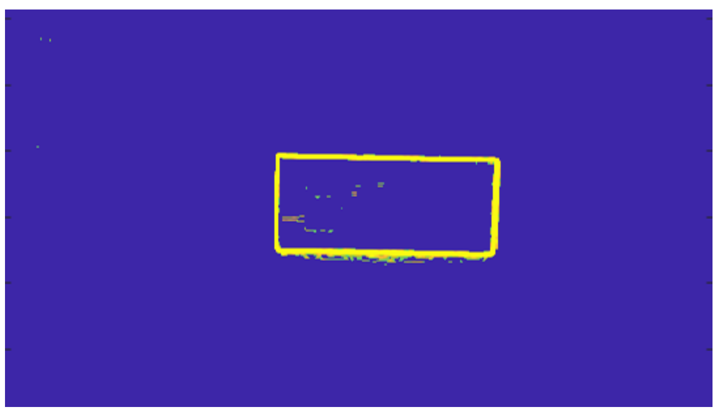


Figure 5: The original image after using a Sobel-Feldman operator and a threshold filter

5 Research

An important part of a project is research. Without reading through previous experiences or discoveries within the domain of the project, it will be very hard to figure out a solution. Let alone being comfortable enough in the domain to try something drastically new and (hopefully) develop new ideas in the specific branch of the subject. Although image processing is a recent research domain, it is quite popular and already has a lot of foundations to built from. Even Matlab has some predefined functions and libraries to assist the development of image processing software. However, the goal of this project is to get insight in the programming and working of image processing. To achieve this goal as good as possible, in the end result, there should be little use of predefined functions or libraries. They will only be used if they are completely comprehensible or if they make the code more elegant by doing ordinary operations. The next section goes over a few functions which could be used to build on the already existing program described above. All these functions are explained so they can be implemented manually at a later time. It starts by using functions to detect and count the objects, and ends with surrounding the objects with a rectangle, as well as highlighting the edges. An example of these functions can be found in Appendix F.

5.0.0.1 Counting of the objects

The central objective of this paper is counting the amount of objects in a specific rectangular field of view. The general approach to this problem is converting the image to a binary image where black pixels represent the background and white pixels represent the objects. By counting the groups of pixels, it is possible to know how many objects the original image contains. In the image processing toolbox for matlab(*Mathworks*, 2018), a few functions exist that are very useful for this kind of tasks. One of these functions called bwlabel actually counts group of pixels of at least 8 that are connected. The syntax of this function goes as follows:

$$[L, num] = bwlabel(BW) \tag{5}$$

where BW represents the binary (or black and white) image; num represents the number of objects in the BW image and where L represents a matrix were the first group of pixels are numbered 1, the second group 2 etc. That way it's easier to get an overview of how many objects there are.

5.0.0.2 Boundary boxes

The image processing toolbox really simplifies the drawing of boundary boxes. Once a binary image is obtained, the function regionprops (*Mathworks*, 2018) can extract properties about image regions. Where image regions are defined as 8-connected components in an binary image. This means that each image region contains at least 8 interconnected white pixels, since the black pixels are registered as background. The property that's interesting for this part of the project is called 'boundingbox'. This property returns for every image region the smallest rectangle that contains this region. In two dimensions this is a vector with 4 values, the x-coordinate of the upper left corner, the y-coordinate of that corner, the width and the height. The function

$$rectangle('Position', pos) \tag{6}$$

where 'Position' declares the input and where pos is the input obtained from regionprops, can easily display this boundingbox.

5.0.0.3 Edge detection

There are a lot of ways to implement edge detection. Edge detection algorithms as described in section 3.2.1 paragraph 4, exist for greyscale images. But if a binary image is available, this becomes much easier. To start, a function called bwboundaries exists in the image processing toolbox (*Mathworks*, 2018). The syntax of that function goes as follows:

$$B = bwboundaries(BW) \tag{7}$$

where BW represents the input. This is a binary image which only consists out of black and white pixels; and B represents the output, which consist out of a cell array with N elements (number of image regions in the binary image), all these elements contain a list of the boundary pixels. Which in turn are fairly easy to draw. They can be inserted in the matrix of the image by replacing values, this is done by looping through the cell arrays. The advantage of this method is that the image can actually be printed. When they are drawn on top of the image with a function like visboundaries, the actual values of the pixels stay unchanged, but it different figures arise. One with the image and another on top of it with the edges. The function bwboundaries implements the Moore-Neighbor tracing algorithm (Ghuneim, n.d.). The algorithm loops through the entire matrix until it finds a white pixel (a pixel that belongs to an image region). This pixel is defined as the start pixel. Once it finds a start pixel it searches for the next connected white pixel. This means another white pixel in one of the eight regions around the start. The algorithm does this by examining the pixels in a clockwise direction. Once it finds a new white pixel, this pixel is added to the sequence B and becomes our new start pixel. This process keeps on running until the algorithm visits the first start pixel for a second time. The only problem with this algorithm is that sometimes the first start pixel is visited for a second time before all of the outline is visited. This is illustrated in Fig. 6).

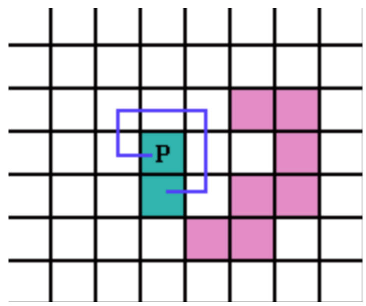


Figure 6: Problem with stopping criteria Moore-Neighbor tracing algorithm.(Ghuneim, n.d.)

This problem is resolved with the Jacob's stopping criterion. Which states that the algorithm can stop once the first start pixel is visited from the same direction as it initially was entered. This leaves four possibilities that need to be checked, from below, from the left, from above or from the right. With this additional criteria, every pixel at the edge of a connected region is visited. To find the edges of all the interconnected image regions this process is repeated until every pixel of the image matrix has been checked.

6 Further planning

Being halfway trough the project, a visual timeframe is created. The Gantt chart followed in the project can be found in Appendix G. This visualizes the current state of the solution for the described problem, as well as the schedule for the next few weeks. This project contains five milestones, two of them are already achieved. These have a minor value in prospect to the total paper though. The most important occupancy of the next weeks is implementing key elements of the final algorithm. Key components like filling the edges, creating the boundary boxes and eventually counting the number of objects still need the necessary attention. All of this while the given deadlines need to be respected. As it is possible to view in the chart, the decision to start rather early on the folder is made. A professional representation of the findings takes time. So planning it like this, ensures enough time to perfect the folder.
In general, the project is on schedule. From a critical point of view, too much time writing the report during the team sessions was wasted. For the final paper, more individual work is recommended and will happen.

7 Budget management

As seen above, the system explained in this paper primarily consists of software which on its own doesn't cost anything. The necessary hardware is rather costly. The current set-up consists of a tripod and the electronics. The tripod is lend for free by the faculty thus the only remaining costs are the Kinect v2 and its adapter to connect with a personal computer.
With a budget of €250, this is feasible. Both the Kinect and the adapter have been ordered but as of writing, a fixed price isn't known. At the current market prices, the estimated cost is €200. The remaining €50 are a safe backup for other small costs.

8 Course Integration

Solving the problem explained in this report is the main objective of the course P&O 3 (B-KUL-H01D4B). This course is part of the bachelor in engineering science curriculum of the KU Leuven. It is a course which uses concepts seen in others to solve problems. By now it should be apparent that, indeed, many courses were used to tackle the given problem.
Knowledge of linear algebra is applied extensively. Simple concepts like matrix multiplication to more advanced ones like matrix convolution form the core of the written algorithms. Numerical Mathematics is useful to get an idea what influence measurement errors have on calculations with large matrices. And this course, together with Applied Informatics, give the tools to investigate the time complexity of the algorithm. The latter course has also helped with grasping the Matlab programming language and environment since general programming concepts are a big part of this project.

9 Conclusion

After four weeks of group gatherings, a decent amount of of research has been done and a decent amount of progress was made. The project will be executed by using a Microsoft Kinect version 2. The combination of a depth and RGB sensor at a reasonable cost are some of the decisive arguments over a standard webcam or an industrial camera.
When obtaining the image from the RGB sensor, the three dimensional matrix is turned into a one dimensional matrix and thus into a greyscale image. A Gaussian blur and an edge detection algorithm are used to further reduce this matrix to a binary array where only the edges are highlighted. Meanwhile, using a Sobel-Feldman operator and a threshold filter, the image obtained from the depth sensor will become clean with a clearly visible outline around the differences in height.
Next, the goal is to, after merging the processed data from the RGB and depth sensor, perfect the positioning of the edges and fill in possible blank spots. Subsequently, a step towards counting the objects will be made. If the amount of progress made per day will stay consistent, all the deadlines should be accomplished in the given timeframe.However, perseverance and a critical point of view are needed to successfully finish this assignment.

10 References

Burge, M. J., & Burger, W. (2008). *Digital image processing* (1st ed.). Springer.

Ghuneim, A. G. (n.d.). *Moore-neighbor tracing.* Retrieved (2018-10-26), from http://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/moore.html

Jiao, J., Yuan, L., Tang, W., & Wu, Q. (Nov 2017). *Kinect v1 and kinect v2 fields of view compared.* Retrieved from https://www.researchgate.net/figure/Kinect-v2-sensor-working\-on-a-photographic-tripod_fig1_321048476/

Mathworks. (2018). Retrieved from https://nl.mathworks.com/help/

R. Fisher, A. W., S. Perkins, & Wolfart, E. (n.d.-a). *Laplacian of gaussian.* Retrieved from https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.html

R. Fisher, A. W., S. Perkins, & Wolfart, E. (n.d.-b). *Mean filter.* Retrieved from https://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.html

Smeenk, R. (11 Mar 2014). *Kinect v1 and kinect v2 fields of view compared.* Retrieved from http://smeenk.com/kinect-field-of-view-comparison/

Sobel, I. (2014, 02). An isotropic 3x3 image gradient operator.

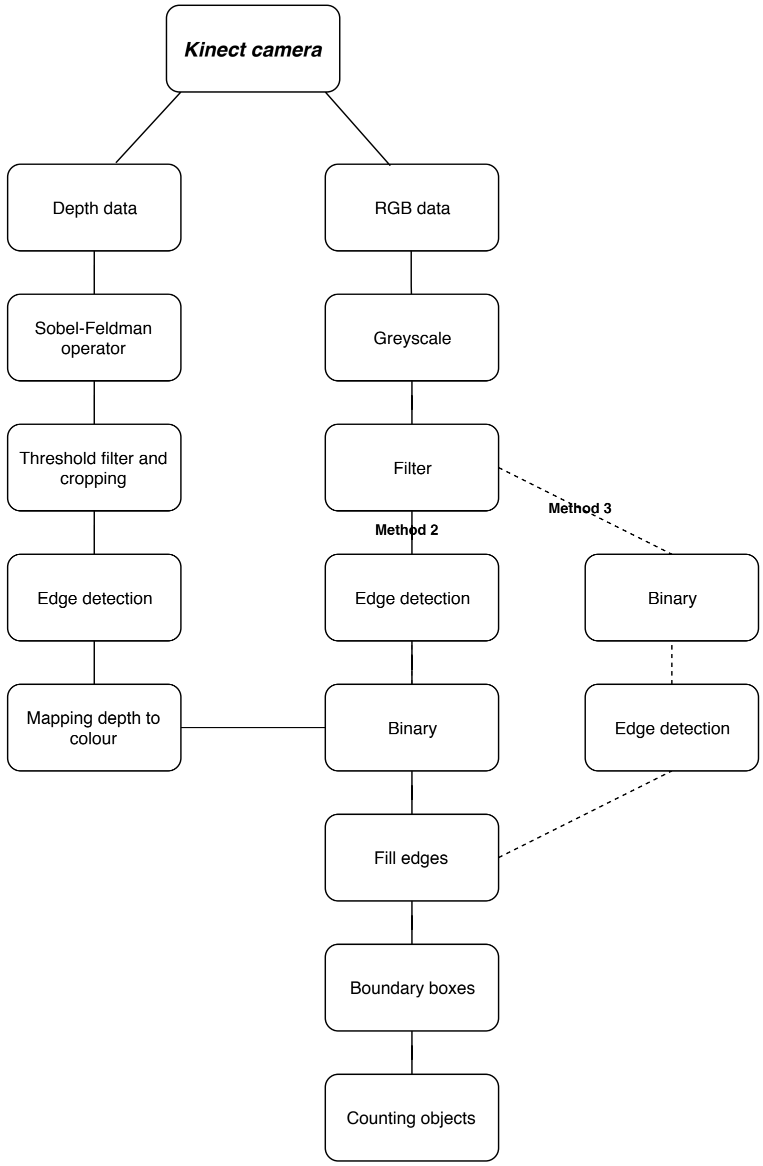
Stackoverflow. (2018). Retrieved from https://stackoverflow.com

Unknown. (n.d.-a). *Greyscale.* Retrieved (2018-10-1), from https://en.wikipedia.org/wiki/Grayscale

Unknown. (n.d.-b). *Kernel (image processing).* Retrieved (2018-10-1), from https://en.wikipedia.org/wiki/Kernel_(image_processing)

11 Appendices

A: Diagram



B: Matlab Threshold Functions

```
%% Read image
I = imread('test.jpg');
imshow(I);

%% Rgb space
rmat= I(:,:,1);
gmat= I(:,:,2);
bmat= I(:,:,3);

subplot(2,2,1), imshow(rmat);
title('Red plane');
subplot(2,2,2), imshow(gmat);
title('Green plane');
subplot(2,2,3), imshow(bmat);
title('Blue plane');
subplot(2,2,4), imshow(I);
title('Original image');

%% Convert to black and white
rlevel = 0.5;
glevel = 0.5;
blevel = 0.5;

rthresh = imbinarize(rmat,'adaptive');
%rthresh = im2bw(rmat,rlevel);
gthresh = imbinarize(gmat,'adaptive');
%gthresh = im2bw(gmat,glevel);
bthresh = imbinarize(bmat,'adaptive');
%bthresh = im2bw(bmat,blevel);
Isum = (rthresh & gthresh & bthresh);

subplot(2,2,1), imshow(rthresh);
title('Red plane');
subplot(2,2,2), imshow(gthresh);
title('Green plane');
subplot(2,2,3), imshow(bthresh);
title('Blue plane');
subplot(2,2,4), imshow(Isum);
title('Sum');

%% Complement of image
Icomp = imcomplement(Isum);
imshow(Icomp);

%% Fill in holes
Ifilled = imfill(Isum,'holes');
imshow(Ifilled);

%% Erasing noise
se = strel('disk',10);
```

```
Iopenned = imopen(Ifilled, se);
imshow(Iopenned);

%% Extract features
[labeled, numObjects] = bwlabel(Iopenned, 4);

%% Use feature analysis to count objects
figure, imshow(I);
title(['There are ', num2str(numObjects), ' objects in the picture']);
```

C: Matlab Erasing Background

```
%% Read image
I = imread('shapes.jpg');
imshow(I);

%% Rgb space
rmat= I(:,:,1);
gmat= I(:,:,2);
bmat= I(:,:,3);

subplot(2,2,1), imshow(rmat);
title('Red plane');
subplot(2,2,2), imshow(gmat);
title('Green plane');
subplot(2,2,3), imshow(bmat);
title('Blue plane');
subplot(2,2,4), imshow(I);
title('Original image');

%% Histogram
nb_bins = 2;
width_bins = 256/nb_bins;

image_size = size(rmat);
nb_elements = image_size(1)*image_size(2);

h1 = histogram(rmat,nb_bins);
hold on;
h2 = histogram(gmat,nb_bins);
h3 = histogram(bmat,nb_bins);
hold off;

biggest_bin_red = 0;
biggest_bin_green = 0;
biggest_bin_blue = 0;
nb_elems_red = h1.Values;
nb_elems_green = h2.Values;
nb_elems_blue = h3.Values;

for i = 1:nb_bins
    if nb_elems_red(i) > biggest_bin_red
        biggest_bin_red = nb_elems_red(i);
        bin_red=i;
    end
    if nb_elems_green(i) > biggest_bin_green
        biggest_bin_green = nb_elems_green(i);
        bin_green=i;
    end
    if nb_elems_blue(i) > biggest_bin_blue
        biggest_bin_blue = nb_elems_blue(i);
        bin_blue=i;
```

```
    end
end

colour_value_red_min = (bin_red -1)*width_bins;
colour_value_red_max = bin_red*width_bins;
colour_value_green_min = (bin_green -1)*width_bins;
colour_value_green_max = bin_green*width_bins;
colour_value_blue_min = (bin_blue -1)*width_bins;
colour_value_blue_max = bin_blue*width_bins;
disp(colour_value_red_min);
disp(colour_value_red_max);

%% Convert to black and white
%red
matrix_size = size(rmat);
MAX_ROW = matrix_size(1);
MAX_COLUMN = matrix_size(2);
rmat_bw = zeros(MAX_ROW,MAX_COLUMN);
for row=1:MAX_ROW
    for col=1:MAX_COLUMN
        if (colour_value_red_min < rmat(row,col)) && (rmat(row,col) <
            colour_value_red_max)
             rmat_bw(row,col) = 255;
        else
             rmat_bw(row,col) = 0;
        end
    end
end

%green
matrix_size = size(gmat);
MAX_ROW = matrix_size(1);
MAX_COLUMN = matrix_size(2);
gmat_bw = zeros(MAX_ROW,MAX_COLUMN);
for row=1:MAX_ROW
    for col=1:MAX_COLUMN
        if (colour_value_green_min < gmat(row,col)) && (gmat(row,col) <
            colour_value_green_max)
             gmat_bw(row,col) = 255;
        else
             gmat_bw(row,col) = 0;
        end
    end
end

%blue
matrix_size = size(bmat);
MAX_ROW = matrix_size(1);
MAX_COLUMN = matrix_size(2);
bmat_bw = zeros(MAX_ROW,MAX_COLUMN);
for row=1:MAX_ROW
    for col=1:MAX_COLUMN
```

```
        if (colour_value_blue_min < bmat(row,col)) && (bmat(row,col) < colour_value_blue_max)
            bmat_bw(row,col) = 255;
        else
            bmat_bw(row,col) = 0;
        end
    end
end

Isum = (rmat_bw & bmat_bw & gmat_bw);

%plot
subplot(2,2,1), imshow(rmat_bw);
title('Red plane');
subplot(2,2,2), imshow(gmat_bw);
title('Green plane');
subplot(2,2,3), imshow(bmat_bw);
title('Blue plane');
subplot(2,2,4), imshow(Isum);
title('Sum');

%% Complement of image
Icomp = imcomplement(Isum);
imshow(Icomp);

%% Fill in holes
Ifilled = imfill(Icomp,'holes');
imshow(Ifilled);

%% Erasing noise
se = strel('disk',1);
Iopenned = imopen(Ifilled , se);
imshow(Iopenned);

%% Extract features
[labeled , numObjects] = bwlabel(Iopenned);
stats = regionprops(labeled,'Eccentricity', 'Area', 'BoundingBox');
eccentricities = [stats.Eccentricity];

%% Use feature analysis to count objects
idxOfObjects = find(eccentricities);

figure, imshow(I);
hold on;
for idx = 1 : length(idxOfObjects)
    h = rectangle ('Position', stats(idx).BoundingBox);
    set(h,'EdgeColor',[.75 0 0]);
    set(h,'LineWidth',2);
end

title(['There are ', num2str(numObjects), ' objects in the picture']);
hold off;
```

```
%% Draw boundaries
B = bwboundaries(Iopenned);
imshow(I);
hold on;
visboundaries(B);
hold off;
```

D: Matlab Code RGB Sensor

```matlab
clearvars
img = imread('kinect/foto RGB 3.png'); % Load picture
A = greyscale(img); % Convert image to grayscale
A = symImgCrop(A, 50); % Crop image so it's the same size.
A = gaussian_blur(mean_blur(A)); % Filters

%% Method 1: First greyscale, then blur, then threshold, then edge detection.
B = threshold(A); % Threshold image
C = ~edge2_detect(B, 3);
D = remove_boundary(C, 25);
subplot(2,2,1), imshow(A, []);
title("Input (after blur)");
subplot(2,2,2), imshow(B, []);
title("After thresholding");
subplot(2,2,3), imshow(D, []);
title("After edge detection & boundary removed");

%% Method 2: First greyscale, then trheshold with background, than edge detection
bg = imread('kinect/foto RGB 1.png'); % Load background image
bg = greyscale(bg); % Convert image to grayscale
bg = symImgCrop(bg, 50); % CROP IMAGE SO IT's the same size.
bg = gaussian_blur(mean_blur(bg)); % Filters

B = threshold_ivm_background(A, bg); % Threshold with background
C = ~edge2_detect(B, 3); % Detect edges.
D = remove_boundary(C, 25); % Remove boundary around image.
subplot(2,2,1), imshow(A, []);
title("Input (after blur)");
subplot(2,2,2), imshow(B, []);
title("After thresholding");
subplot(2,2,3), imshow(D, []);
title("After edge detection & boundary removed");

%% Method 3: First greyscale, then blur, then edge detect then threshold and then noise removal
first_edge_detect = edge_detect(A); % Laplacian edge detection
without_noise_removal = threshold_edge(remove_boundary(first_edge_detect, 15)); % Remove boundary around image & threshold the edges.
with_noise_removal = noise_deletion(without_noise_removal, 3); % Noise removal
subplot(2,2,1), imshow(A, []);
title("Input (after blur)");
subplot(2,2,2), imshow(first_edge_detect, []);
title("After edge detection");
subplot(2,2,3), imshow(without_noise_removal, []);
title("Threshold without noise removal");
subplot(2,2,4), imshow(with_noise_removal, []);
title("Method 2 with gaussian and mean blur");
```

```


function result = threshold_ivm_background(img, bg)
    % DIMENSIONS MUST MATCH
    % Compare pixel at img(row, col) with bg(row, col).
    % if bg(row, col) - D <= img(row, col) <= bg(row, col) + D
    %       The pixels are defined as background!! (= white)

    D = 10;
    WHITE = 1;
    BLACK = 0;

    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

    result = zeros(MAX_ROW,MAX_COLUMN,1);
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            if img(row, col) <= bg(row, col) + D && img(row, col) >= bg(row, col) - D
                % Classified as background
                result(row, col) = WHITE;
            else
                % Not background
                result(row, col) = BLACK;
            end
        end
    end

end

function nes = noise_deletion(img,window)
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);
    side = floor(window/2);
    nes = img;

    for col=side+1:MAX_COLUMN-side
        for row=side+1:MAX_ROW-side
            list=zeros(window);
            q=1;
            for i=-side:side
                for j=-side:side
                    list(q) = img(row+i,col+j);
                    q = q+1;
                end
            end
            list=sort(list);
            nes(row,col) = list(floor((window^2)/2)+1);
        end
```

```
    end
end

function result = remove_boundary(img, remove_size)
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

    result = zeros(MAX_ROW,MAX_COLUMN,1);
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            if row < remove_size || col < remove_size || row > (MAX_ROW - remove_size) || col > (MAX_COLUMN - remove_size)
                % Inside boundary ==> needs to be white (= 1)
                result(row, col) = 1;
            else
                result(row, col) = img(row, col);
            end

        end
    end
end

function thresholded_img = threshold_edge(img)
    threshold_value = 2;
    %most_occuring =mode(img) +100;
    %threshold_value = most_occuring(1);

    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);
    THICKNESS = 3;

    thresholded_img = zeros(MAX_ROW,MAX_COLUMN,1);
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            if img(row, col) > threshold_value
                value = 1;
                for i=1:THICKNESS
                    % Create thicker edges (edges of THICKNESS pixels thick)
                    if (col - i) > 0
                        thresholded_img(row, col-i) = 0;
                    end
                end
            else
                value = 0;
            end
            thresholded_img(row, col) = value;
        end
    end
end
```

```
function mean_blurred = mean_blur(img)
    mean = (1/9) * [ 1 1 1; 1 1 1; 1 1 1];
    mean_blurred = conv2(img, mean);
end

function gaussian_blurred = gaussian_blur(img)
    gaussian = (1/159) * [2 4 5 4 2; 4 9 12 9 4; 5 12 15 12 5; 4 9 12 9 4; 2 4 5 4 2;];
    gaussian_blurred = conv2(img, gaussian);
end

function edge2 = edge2_detect(img,intolerance)
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);
    edge2 = zeros(MAX_ROW,MAX_COLUMN,1);
    THICKNESS = 2;

    % Horizontaal laten checken voor edges.
    previous_value = img(1,1);
    for row=1:MAX_ROW % We gaan elke rij af
        for col=1:MAX_COLUMN
            i=1;
            flag = 0;
            if img(row, col) == 1 && previous_value == 0
                % DUS: Het begin van een object. (hele tijd wit, nu zwart), flag voor intolerantie controle aanzetten.
                flag = 1;
            elseif img(row, col) == 0 && previous_value == 1
                % DUS: Het einde van een object (hele tijd zwart, nu wit), flag voor intolerantie controle aanzetten.
                flag = 1;
            end

            %%Intolerantie controle
            while i <= intolerance && flag && col+i <= MAX_COLUMN
                if img(row,col-1+i) ~= img(row,col+i)
                    flag = 0;
                end
                i=i+1;
            end

            % Eertse maal edgematrix vullen
            if flag
                edge2(row, col) = 1;

                for i=1:THICKNESS
                    % Create thicker edges (edges of THICKNESS pixels thick)
                    if (col - i) > 0
                        edge2(row, col-i) = 1;
```

```
                    end
                end
            else
                edge2(row, col) = 0;
            end

            previous_value = img(row, col);
        end

    previous_value = img(row,1);
    end

    % Verticaal controleren op edges.
    previous_value = img(1,1);
    for col=1:MAX_COLUMN  % We gaan elke kolom af
        for row=1:MAX_ROW
            i=1;
            flag = 0;
            if img(row, col) == 1 && previous_value == 0
                % DUS: Het begin van een object. (hele tijd wit, nu zwart), flag voor intolerantie controle aanzetten.
                %value = 1;
                flag = 1;
            elseif img(row, col) == 0 && previous_value == 1
                %DUS: Het einde van een object (hele tijd zwart, nu wit), flag voor intolerantie controle aanzetten.
                %value = 1;
                flag = 1;
            end

            % Intolerantie controle
            while i <= intolerance && flag && row+i <= MAX_ROW
                if img(row-1+i,col) ~= img(row+i,col)
                    flag = 0;
                end
                i=i+1;
            end

            % Enkel nullen overriden
            if flag
                edge2(row, col) = 1;
                for i=1:THICKNESS
                    % Create thicker edges (edges of THICKNESS pixels thick)
                    if (row - i) > 0
                        edge2(row - i, col) = 1;
                    end
                end
            end

            previous_value = img(row, col);
        end
```

```
    previous_value = img(1,col);
    end

end

function edge = edge_detect(img)
    klaplace=[0 -1 0; -1 4 -1;  0 -1 0];              % Laplacian filter
        kernel
    edge=conv2(img,klaplace);                          % convolve test img
        with
end

function thresholded_img = threshold(img)
    threshold_value = 125;
    %most_occuring =mode(img) +100;
    %threshold_value = most_occuring(1);

    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

    thresholded_img = zeros(MAX_ROW,MAX_COLUMN,1);
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            if img(row, col) > threshold_value
                value = 1;

            else
                value = 0;
            end
            thresholded_img(row, col) = value;
        end
    end
end

function grey = greyscale(img)
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

    grey = zeros(MAX_ROW,MAX_COLUMN,1);
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            R = img(row, col, 1);
            G = img(row, col, 2);
            B = img(row, col, 3);
            grey(row, col) = 0.2989 * R + 0.5870 * G + 0.1140 * B ;
            %These are two methods for grayscaling.
            %grey(row, col) = (R + G + B)/3;
        end
    end
```

```
end

%% Cropping and image rotation functions

function cropped_img = symImgCrop(img,cutted_edge_size)
    original_img_size = size(img);
    original_max_row = original_img_size(1);
    original_max_column = original_img_size(2);

    cropped_img = zeros(original_max_row - 2*cutted_edge_size, original_max_column - 2*cutted_edge_size,1);

    for row=cutted_edge_size:original_max_row - cutted_edge_size
        for col=cutted_edge_size:original_max_column - cutted_edge_size
            cropped_img(row - cutted_edge_size + 1,col - cutted_edge_size + 1) = img(row,col);
        end
    end
end

function img_rot = four_point_img_rotation_crop(img, fourp)
    % A function which crops and rotates the given image so that the only
    % thing shown will be the pixels inside the four points, in horizontal
    % or vertical position.
     fourp_base = [0 fourp(1,2)-fourp(1,1) fourp(1,3)-fourp(1,1) fourp(1,4)-fourp(1,1); 0 fourp(2,2)-fourp(2,1) fourp(2,3)-fourp(2,1) fourp(2,4)-fourp(2,1)];
     if (fourp_base(4,1)-fourp_base(1,1)) > 4
         sigma = atan((fourp_base(2,4)-fourp_base(2,1))/(fourp_base(1,4)-fourp_base(1,1)));
     else
         sigma = pi/2;
     end
     if sigma <= pi/4
         rot_mat = [cos(-sigma) -sin(-sigma); sin(-sigma) cos(-sigma)];
     else
         rot_mat = [cos(pi/2 - sigma) -sin(pi/2 - sigma); sin(pi/2 - sigma) cos(pi/2 - sigma)];
     end


end

function img_crop = generic_crop(img, fourp)
    % A function which crops the given image so that the edges are
    % definened by the four point given in fourp.
    X_ARRAY = [fourp(1,1) fourp(1,2) fourp(1,3) fourp(1,4)];
    Y_ARRAY = [fourp(2,1) fourp(2,2) fourp(2,3) fourp(2,4)];
    MIN_X = min(X_ARRAY);
    MAX_X = max(X_ARRAY);
    MIN_Y = min(Y_ARRAY);
```

```
    MAX_Y = max(Y_ARRAY);
    img_crop = zeros(MAX_X-MIN_X,MAX_Y-MIN_Y);

    for row = MIN_X:MAX_X
        for col = MIN_Y:MAX_Y
            img_crop(row - MIN_X + 1,col - MIN_Y + 1) = img(row,col);
        end
    end
end
```

E: Matlab Code Septh Sensor

```
%processing the image using the depthsensor


%treshold values
min_tresh = 30;
max_tresh = 500;

% get image from depth sensor
depth = getsnapshot(depthVid);

%run the sobel operator
shapes = sobel_operator(depth);

%run the treshold filter
shapes = treshold(shapes, min_tresh, max_tresh);

%look at the result
image(depth);

function shapes = sobel_operator(img)

    X = img;
    Gx = [1 +2 +1; 0 0 0; -1 -2 -1]; Gy = Gx';
    temp_x = conv2(X, Gx, 'same');
    temp_y = conv2(X, Gy, 'same');
    shapes = sqrt(temp_x.^2 + temp_y.^2);
end

function tresholded = treshold(img, min_tresh, max_tresh)

    matrix_size = size(img);

    MAX_ROW = matrix_size(1);

    MAX_COLUMN = matrix_size(2);

    for row = 1 : MAX_ROW
        for col = 1: MAX_COLUMN
            if (img(row, col) > min_tresh) && (img(row, col)< max_tresh)
                img(row, col) = 1;
            else
                img(row, col) = 0;
            end
        end
    end
    tresholded = img;
end
```

F: Matlab Library Functions

```matlab
%% Extract features
[labeled, numObjects] = bwlabel(Image_black);
stats = regionprops(labeled,'Eccentricity', 'Area', 'BoundingBox');
eccentricities = [stats.Eccentricity];

%% Use feature analysis to count objects
idxOfObjects = find(eccentricities);

figure, imshow(Image);
hold on;
for idx = 1 : length(idxOfObjects)
    h = rectangle ('Position', stats(idx).BoundingBox);
    set(h,'EdgeColor',[.75 0 0]);
    set(h,'LineWidth',2);
end

title(['There are ', num2str(numObjects), ' objects in the picture']);
hold off;

%% Draw boundaries
B = bwboundaries(Image_black);
imshow(Image);
hold on;
visboundaries(B);
hold off;
```

G: Gantt chart

teamgantt
Created with Free Edition