

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

Contents	i
List of Figures	ii
1 Introduction	1
2 Problem Description	1
3 Hardware	2
3.1 Hardware components	2
3.2 Visual setup	3
4 Software	3
4.1 Analysing different methods	3
4.1.1 Method 1: Thresholding	4
4.1.2 Method 2: Edge detection	4
4.1.3 Method 3: Background filter	4
4.2 Choice of method	5
4.2.1 Step 1: Converting to greyscale	5
4.2.2 Step 2: Filtering the image	6
4.2.3 Step 3: Edge detection	6
4.2.4 Step 4: Threshold	7
4.3 Analysis Depth Sensor	7
4.3.1 Step 1: Detection of the difference in height	7
4.3.2 Step 2: Filtering of the noise	8
5 Research	8
6 Further planning	10
7 Budget management	10
8 Course Integration	11
9 Conclusion	11
10 References	12
11 Appendices	13

List of Figures

1	Example of a possible result.	2
2	A comparison between the 3 different methods.	5
3	A comparison with the use of filters.	6
4	The original RGB image (left) and the image after the Sobel-Feldman operator (right) .	8
5	The original image after using a Sobel-Feldman operator and a threshold filter	8
6	Problem with stopping criteria Moore-Neighbor tracing algorithm	10

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 (Burge & Burger, 2008). 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 addressed.

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 preferred.

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 through 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.

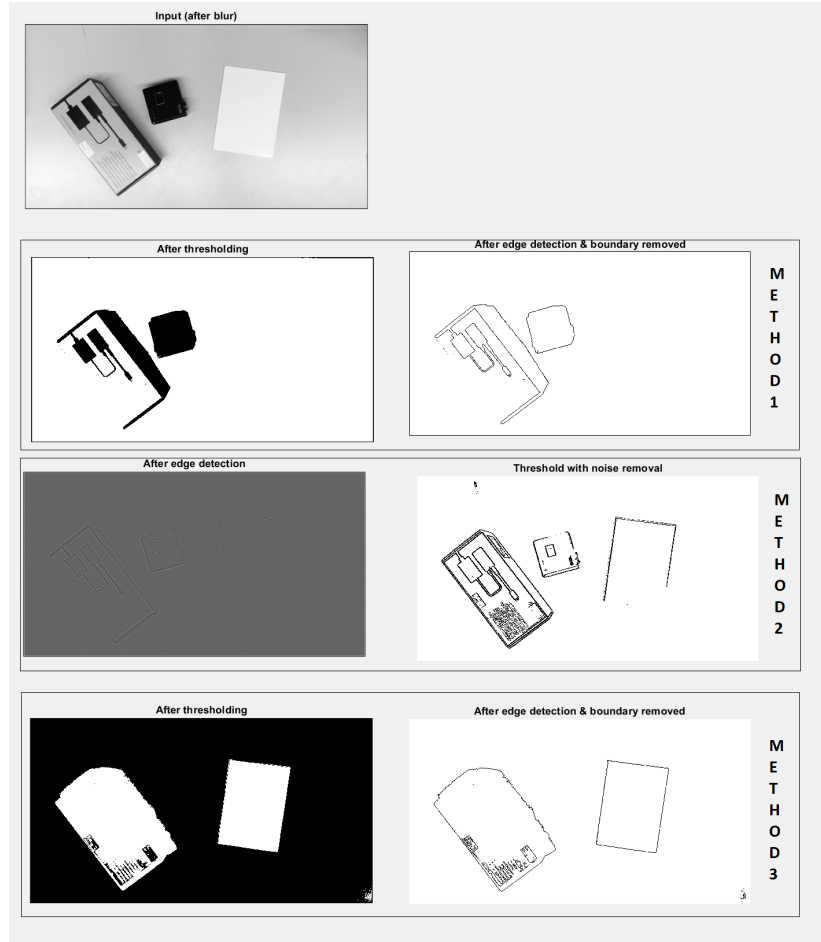


Figure 2: A comparison between the 3 different methods.

$$greyscale_image(row, col) = 0.2989 * RED + 0.5870 * GREEN + 0.1140 * BLUE \quad (1)$$

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.

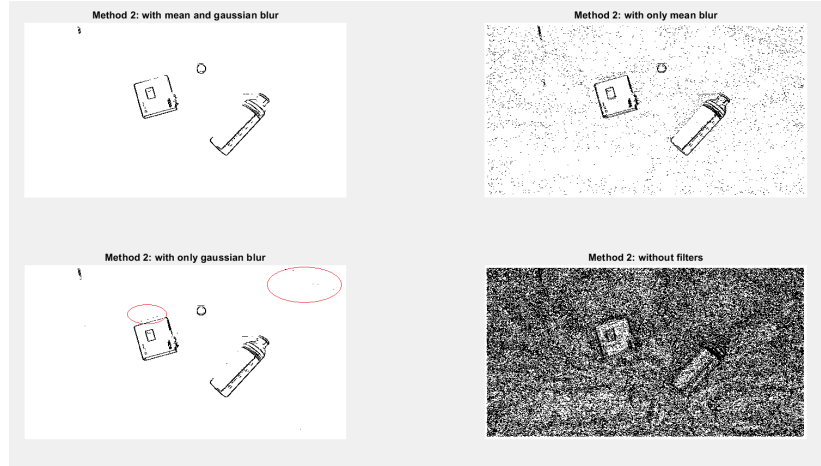


Figure 3: A comparison with the use of filters.

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

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.

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

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.

$$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad (4)$$

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 enriching 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.

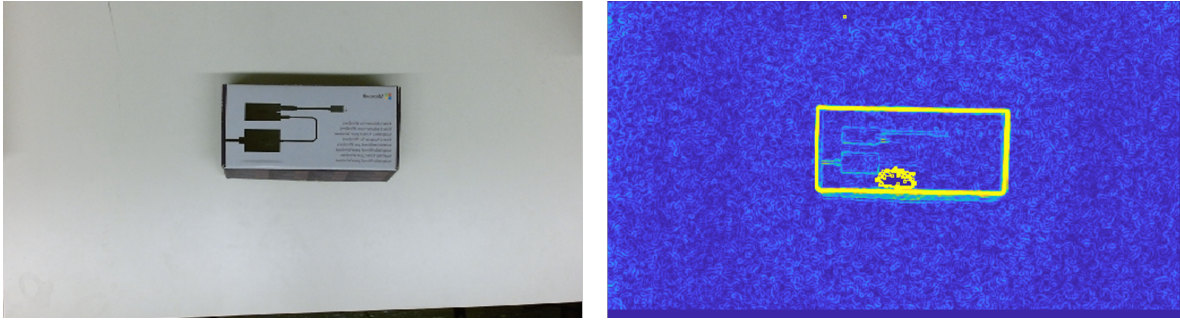


Figure 4: The original RGB image (left) and the image after the Sobel-Feldman operator (right)

4.3.2 Step 2: Filtering of the noise

After adding all 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.

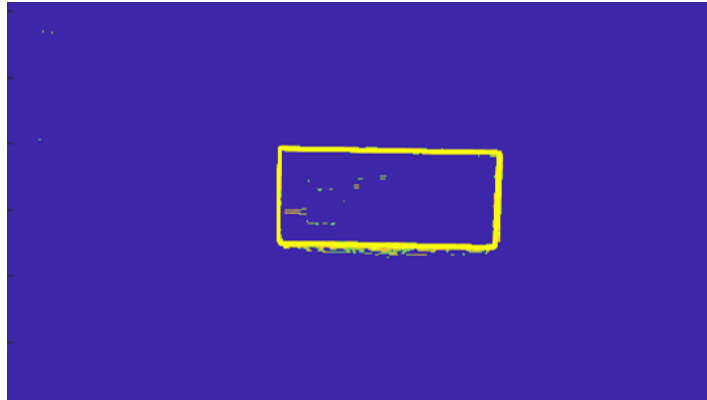


Figure 5: The original image after using a Sobel-Feldman operator and a threshold filter

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 build 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:

$$[L, num] = bwlabel(BW) \quad (5)$$

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 where 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

$$rectangle('Position', pos) \quad (6)$$

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:

$$B = bwboundaries(BW) \quad (7)$$

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).

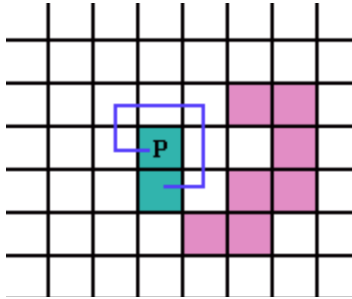


Figure 6: Problem with stopping criteria Moore-Neighbor tracing algorithm

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.

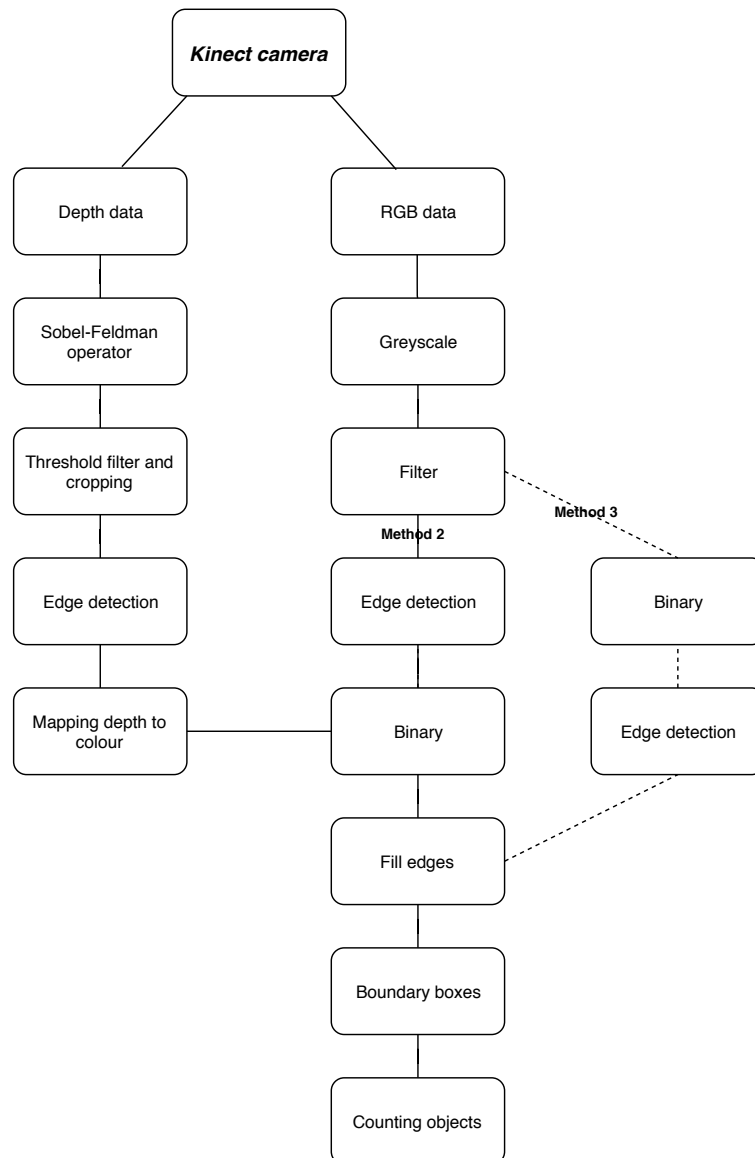
Next, the goal is to, after merging the processed data from the RGB and depth sensor, perfect the positioning of the edges and fill in possible blank spots. Subsequently, a step towards counting the objects will be made. If the amount of progress made per day will stay consistent, all the deadlines should 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_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))

11 Appendices

A: Diagram



B: Matlab Threshold Functions

```
1 %% Read image
2 I = imread('test.jpg');
3 imshow(I);
4
5 %% Rgb space
6 rmat= I(:, :,1);
7 gmat= I(:, :,2);
8 bmat= I(:, :,3);
9
10 subplot(2,2,1), imshow(rmat);
11 title('Red plane');
12 subplot(2,2,2), imshow(gmat);
13 title('Green plane');
14 subplot(2,2,3), imshow(bmat);
15 title('Blue plane');
16 subplot(2,2,4), imshow(I);
17 title('Original image');
18
19 %% Convert to black and white
20 rlevel = 0.5;
21 glevel = 0.5;
22 blevel = 0.5;
23
24 rthresh = imbinarize(rmat,'adaptive');
25 %rthresh = im2bw(rmat,rlevel);
26 gthresh = imbinarize(gmat,'adaptive');
27 %gthresh = im2bw(gmat,glevel);
28 bthresh = imbinarize(bmat,'adaptive');
29 %bthresh = im2bw(bmat,blevel);
30 Isum = (rthresh & gthresh & bthresh);
31
32 subplot(2,2,1), imshow(rthresh);
33 title('Red plane');
34 subplot(2,2,2), imshow(gthresh);
35 title('Green plane');
36 subplot(2,2,3), imshow(bthresh);
37 title('Blue plane');
38 subplot(2,2,4), imshow(Isum);
39 title('Sum');
40
41 %% Complement of image
42 Icomp = imcomplement(Isum);
43 imshow(Icomp);
44
45 %% Fill in holes
46 Ifilled = imfill(Isum,'holes');
47 imshow(Ifilled);
48
49 %% Erasing noise
50 se = strel('disk',10);
```

```

51 Iopenned = imopen(Ifilled , se);
52 imshow(Iopenned);
53
54 %% Extract features
55 [labeled , numObjects] = bwlabel(Iopenned , 4);
56
57 %% Use feature analysis to count objects
58 figure , imshow(I);
59 title([ 'There are ' , num2str(numObjects) , ' objects in the picture' ]);

```

C: Matlab Erasing Background

```
1 %% Read image
2 I = imread('shapes.jpg');
3 imshow(I);
4
5 %% Rgb space
6 rmat= I(:, :, 1);
7 gmat= I(:, :, 2);
8 bmat= I(:, :, 3);
9
10 subplot(2,2,1), imshow(rmat);
11 title('Red plane');
12 subplot(2,2,2), imshow(gmat);
13 title('Green plane');
14 subplot(2,2,3), imshow(bmat);
15 title('Blue plane');
16 subplot(2,2,4), imshow(I);
17 title('Original image');
18
19 %% Histogram
20 nb_bins = 2;
21 width_bins = 256/nb_bins;
22
23 image_size = size(rmat);
24 nb_elements = image_size(1)*image_size(2);
25
26 h1 = histogram(rmat,nb_bins);
27 hold on;
28 h2 = histogram(gmat,nb_bins);
29 h3 = histogram(bmat,nb_bins);
30 hold off;
31
32 biggest_bin_red = 0;
33 biggest_bin_green = 0;
34 biggest_bin_blue = 0;
35 nb_elems_red = h1.Values;
36 nb_elems_green = h2.Values;
37 nb_elems_blue = h3.Values;
38
39 for i = 1:nb_bins
40     if nb_elems_red(i) > biggest_bin_red
41         biggest_bin_red = nb_elems_red(i);
42         bin_red=i;
43     end
44     if nb_elems_green(i) > biggest_bin_green
45         biggest_bin_green = nb_elems_green(i);
46         bin_green=i;
47     end
48     if nb_elems_blue(i) > biggest_bin_blue
49         biggest_bin_blue = nb_elems_blue(i);
50         bin_blue=i;
```

```

51     end
52 end
53
54 colour_value_red_min = (bin_red-1)*width_bins;
55 colour_value_red_max = bin_red*width_bins;
56 colour_value_green_min = (bin_green-1)*width_bins;
57 colour_value_green_max = bin_green*width_bins;
58 colour_value_blue_min = (bin_blue-1)*width_bins;
59 colour_value_blue_max = bin_blue*width_bins;
60 disp(colour_value_red_min);
61 disp(colour_value_red_max);
62
63 %% Convert to black and white
64 %red
65 matrix_size = size(rmat);
66 MAXROW = matrix_size(1);
67 MAXCOLUMN = matrix_size(2);
68 rmat_bw = zeros(MAXROW,MAXCOLUMN);
69 for row=1:MAXROW
70     for col=1:MAXCOLUMN
71         if (colour_value_red_min < rmat(row,col)) && (rmat(row,col) <
            colour_value_red_max)
72             rmat_bw(row,col) = 255;
73         else
74             rmat_bw(row,col) = 0;
75         end
76     end
77 end
78
79 %green
80 matrix_size = size(gmat);
81 MAXROW = matrix_size(1);
82 MAXCOLUMN = matrix_size(2);
83 gmat_bw = zeros(MAXROW,MAXCOLUMN);
84 for row=1:MAXROW
85     for col=1:MAXCOLUMN
86         if (colour_value_green_min < gmat(row,col)) && (gmat(row,col) <
            colour_value_green_max)
87             gmat_bw(row,col) = 255;
88         else
89             gmat_bw(row,col) = 0;
90         end
91     end
92 end
93
94 %blue
95 matrix_size = size(bmat);
96 MAXROW = matrix_size(1);
97 MAXCOLUMN = matrix_size(2);
98 bmat_bw = zeros(MAXROW,MAXCOLUMN);
99 for row=1:MAXROW
100     for col=1:MAXCOLUMN

```

```

101         if (colour_value_blue_min < bmat(row,col)) && (bmat(row,col) <
            colour_value_blue_max)
102             bmat_bw(row,col) = 255;
103         else
104             bmat_bw(row,col) = 0;
105         end
106     end
107 end
108
109 Isum = (rmat_bw & bmat_bw & gmat_bw);
110
111 %plot
112 subplot(2,2,1), imshow(rmat_bw);
113 title('Red plane');
114 subplot(2,2,2), imshow(gmat_bw);
115 title('Green plane');
116 subplot(2,2,3), imshow(bmat_bw);
117 title('Blue plane');
118 subplot(2,2,4), imshow(Isum);
119 title('Sum');
120
121 %% Complement of image
122 Icomp = imcomplement(Isum);
123 imshow(Icomp);
124
125 %% Fill in holes
126 Ifilled = imfill(Icomp,'holes');
127 imshow(Ifilled);
128
129 %% Erasing noise
130 se = strel('disk',1);
131 Iopenned = imopen(Ifilled, se);
132 imshow(Iopenned);
133
134 %% Extract features
135 [labeled, numObjects] = bwlabel(Iopenned);
136 stats = regionprops(labeled,'Eccentricity','Area','BoundingBox');
137 eccentricities = [stats.Eccentricity];
138
139 %% Use feature analysis to count objects
140 idxOfObjects = find(eccentricities);
141
142 figure, imshow(I);
143 hold on;
144 for idx = 1 : length(idxOfObjects)
145     h = rectangle('Position', stats(idx).BoundingBox);
146     set(h,'EdgeColor',[.75 0 0]);
147     set(h,'LineWidth',2);
148 end
149
150 title(['There are ', num2str(numObjects), ' objects in the picture']);
151 hold off;

```

```
152
153 %% Draw boundaries
154 B = bwboundaries(Iopenned);
155 imshow(I);
156 hold on;
157 visboundaries(B);
158 hold off;
```

D: Matlab Code RGB Sensor

```
1 clearvars
2 img = imread('kinect/foto RGB 3.png'); % Load picture
3 A = greyscale(img); % Convert image to grayscale
4 A = symImgCrop(A, 50); % Crop image so it's the same size.
5 A = gaussian_blur(mean_blur(A)); % Filters
6
7 %% Method 1: First greyscale, then blur, then threshold, then edge
  detection.
8 B = threshold(A); % Threshold image
9 C = ~edge2_detect(B, 3);
10 D = remove_boundary(C, 25);
11 subplot(2,2,1), imshow(A, []);
12 title("Input (after blur)");
13 subplot(2,2,2), imshow(B, []);
14 title("After thresholding");
15 subplot(2,2,3), imshow(D, []);
16 title("After edge detection & boundary removed");
17
18 %% Method 2: First greyscale, then trreshold with background, than edge
  detection
19 bg = imread('kinect/foto RGB 1.png'); % Load background image
20 bg = greyscale(bg); % Convert image to grayscale
21 bg = symImgCrop(bg, 50); % CROP IMAGE SO IT's the same size.
22 bg = gaussian_blur(mean_blur(bg)); % Filters
23
24 B = threshold_ivm_background(A, bg); % Threshold with background
25 C = ~edge2_detect(B, 3); % Detect edges.
26 D = remove_boundary(C, 25); % Remove boundary around image.
27 subplot(2,2,1), imshow(A, []);
28 title("Input (after blur)");
29 subplot(2,2,2), imshow(B, []);
30 title("After thresholding");
31 subplot(2,2,3), imshow(D, []);
32 title("After edge detection & boundary removed");
33
34 %% Method 3: First greyscale, then blur, then edge detect then threshold
  and then noise removal
35 first_edge_detect = edge_detect(A); % Laplacian edge detection
36 without_noise_removal = threshold_edge(remove_boundary(first_edge_detect,
  15)); % Remove boundary around image & threshold the edges.
37 with_noise_removal = noise_deletion(without_noise_removal, 3); % Noise
  removal
38 subplot(2,2,1), imshow(A, []);
39 title("Input (after blur)");
40 subplot(2,2,2), imshow(first_edge_detect, []);
41 title("After edge detection");
42 subplot(2,2,3), imshow(without_noise_removal, []);
43 title("Threshold without noise removal");
44 subplot(2,2,4), imshow(with_noise_removal, []);
45 title("Method 2 with gaussian and mean blur");
```



```

46
47
48 function result = threshold_ivm_background(img, bg)
49     % DIMENSIONS MUST MATCH
50     % Compare pixel at img(row, col) with bg(row, col).
51     % if bg(row, col) - D <= img(row, col) <= bg(row, col) + D
52     %         The pixels are defined as background!! (= white)
53
54     D = 10;
55     WHITE = 1;
56     BLACK = 0;
57
58     matrix_size = size(img);
59     MAXROW = matrix_size(1);
60     MAXCOLUMN = matrix_size(2);
61
62     result = zeros(MAXROW,MAXCOLUMN,1);
63     for row=1:MAXROW
64         for col=1:MAXCOLUMN
65             if img(row, col) <= bg(row, col) + D && img(row, col) >= bg(
66                 row, col) - D
67                 % Classified as background
68                 result(row, col) = WHITE;
69             else
70                 % Not background
71                 result(row, col) = BLACK;
72             end
73         end
74     end
75 end
76
77 function nes = noise_deletion(img, window)
78     matrix_size = size(img);
79     MAXROW = matrix_size(1);
80     MAXCOLUMN = matrix_size(2);
81     side = floor(window/2);
82     nes = img;
83
84     for col=side+1:MAXCOLUMN-side
85         for row=side+1:MAXROW-side
86             list=zeros(window);
87             q=1;
88             for i=-side:side
89                 for j=-side:side
90                     list(q) = img(row+i, col+j);
91                     q = q+1;
92                 end
93             end
94             list=sort(list);
95             nes(row, col) = list(floor((window^2)/2)+1);
96         end
97     end

```

```

97     end
98 end
99
100 function result = remove_boundary(img, remove_size)
101     matrix_size = size(img);
102     MAXROW = matrix_size(1);
103     MAXCOLUMN = matrix_size(2);
104
105     result = zeros(MAXROW,MAXCOLUMN,1);
106     for row=1:MAXROW
107         for col=1:MAXCOLUMN
108             if row < remove_size || col < remove_size || row > (MAXROW -
                remove_size) || col > (MAXCOLUMN - remove_size)
109                 % Inside boundary ==> needs to be white (= 1)
110                 result(row, col) = 1;
111             else
112                 result(row, col) = img(row, col);
113             end
114         end
115     end
116 end
117 end
118
119 function thresholded_img = threshold_edge(img)
120     threshold_value = 2;
121     %most_occuring =mode(img) +100;
122     %threshold_value = most_occuring(1);
123
124     matrix_size = size(img);
125     MAXROW = matrix_size(1);
126     MAXCOLUMN = matrix_size(2);
127     THICKNESS = 3;
128
129     thresholded_img = zeros(MAXROW,MAXCOLUMN,1);
130     for row=1:MAXROW
131         for col=1:MAXCOLUMN
132             if img(row, col) > threshold_value
133                 value = 1;
134                 for i=1:THICKNESS
135                     % Create thicker edges (edges of THICKNESS pixels
                        thick)
136                     if (col - i) > 0
137                         thresholded_img(row, col-i) = 0;
138                     end
139                 end
140             else
141                 value = 0;
142             end
143             thresholded_img(row, col) = value;
144         end
145     end
146 end

```

```

147
148 function mean_blurred = mean_blur(img)
149     mean = (1/9) * [ 1 1 1; 1 1 1; 1 1 1];
150     mean_blurred = conv2(img, mean);
151 end
152
153 function gaussian_blurred = gaussian_blur(img)
154     gaussian = (1/159) * [2 4 5 4 2; 4 9 12 9 4; 5 12 15 12 5; 4 9 12 9
155         4; 2 4 5 4 2];
156     gaussian_blurred = conv2(img, gaussian);
157 end
158
159 function edge2 = edge2_detect(img,intolerance)
160     matrix_size = size(img);
161     MAXROW = matrix_size(1);
162     MAXCOLUMN = matrix_size(2);
163     edge2 = zeros(MAXROW,MAXCOLUMN,1);
164     THICKNESS = 2;
165
166     % Horizontaal laten checken voor edges.
167     previous_value = img(1,1);
168     for row=1:MAXROW % We gaan elke rij af
169         for col=1:MAXCOLUMN
170             i=1;
171             flag = 0;
172             if img(row, col) == 1 && previous_value == 0
173                 % DUS: Het begin van een object. (hele tijd wit, nu zwart
174                 ), flag voor intolerantie controle aanzetten.
175                 flag = 1;
176             elseif img(row, col) == 0 && previous_value == 1
177                 % DUS: Het einde van een object (hele tijd zwart, nu wit
178                 ), flag voor intolerantie controle aanzetten.
179                 flag = 1;
180             end
181             %Intolerantie controle
182             while i <= intolerance && flag && col+i <= MAXCOLUMN
183                 if img(row,col-1+i) ~= img(row,col+i)
184                     flag = 0;
185                 end
186                 i=i+1;
187             end
188             % Eertse maal edgematrix vullen
189             if flag
190                 edge2(row, col) = 1;
191
192                 for i=1:THICKNESS
193                     % Create thicker edges (edges of THICKNESS pixels
194                     thick)
195                     if (col - i) > 0
196                         edge2(row, col-i) = 1;

```

```

195         end
196     end
197     else
198         edge2(row, col) = 0;
199     end
200
201     previous_value = img(row, col);
202 end
203
204 previous_value = img(row,1);
205 end
206
207 % Verticaal controleren op edges.
208 previous_value = img(1,1);
209 for col=1:MAX_COLUMN % We gaan elke kolom af
210     for row=1:MAX_ROW
211         i=1;
212         flag = 0;
213         if img(row, col) == 1 && previous_value == 0
214             % DUS: Het begin van een object. (hele tijd wit, nu zwart
                ), flag voor intolerantie controle aanzetten.
215             %value = 1;
216             flag = 1;
217         elseif img(row, col) == 0 && previous_value == 1
218             %DUS: Het einde van een object (hele tijd zwart, nu wit)
                , flag voor intolerantie controle aanzetten.
219             %value = 1;
220             flag = 1;
221         end
222
223         % Intolerantie controle
224         while i <= intolerance && flag && row+i <= MAX_ROW
225             if img(row-1+i, col) ~= img(row+i, col)
226                 flag = 0;
227             end
228             i=i+1;
229         end
230
231         % Enkel nullen overriden
232         if flag
233             edge2(row, col) = 1;
234             for i=1:THICKNESS
235                 % Create thicker edges (edges of THICKNESS pixels
                    thick)
236                 if (row - i) > 0
237                     edge2(row - i, col) = 1;
238                 end
239             end
240         end
241
242         previous_value = img(row, col);
243     end

```

```

244
245     previous_value = img(1,col);
246     end
247
248 end
249
250 function edge = edge_detect(img)
251     klaplace=[0 -1 0; -1 4 -1; 0 -1 0];           % Laplacian filter
252     kernel                                     % convolve test img
253     edge=conv2(img, klaplace);
254
255 end
256
257 function thresholded_img = threshold(img)
258     threshold_value = 125;
259     %most_occurring =mode(img) +100;
260     %threshold_value = most_occurring(1);
261
262     matrix_size = size(img);
263     MAXROW = matrix_size(1);
264     MAXCOLUMN = matrix_size(2);
265
266     thresholded_img = zeros(MAXROW,MAXCOLUMN,1);
267     for row=1:MAXROW
268         for col=1:MAXCOLUMN
269             if img(row, col) > threshold_value
270                 value = 1;
271             else
272                 value = 0;
273             end
274             thresholded_img(row, col) = value;
275         end
276     end
277
278 end
279
280 function grey = greyscale(img)
281     matrix_size = size(img);
282     MAXROW = matrix_size(1);
283     MAXCOLUMN = matrix_size(2);
284
285     grey = zeros(MAXROW,MAXCOLUMN,1);
286     for row=1:MAXROW
287         for col=1:MAXCOLUMN
288             R = img(row, col, 1);
289             G = img(row, col, 2);
290             B = img(row, col, 3);
291             grey(row, col) = 0.2989 * R + 0.5870 * G + 0.1140 * B ;
292             %These are two methods for grayscaling.
293             %grey(row, col) = (R + G + B)/3;
294         end
295     end
296
297 end

```

```

294 end
295
296 %% Cropping and image rotation functions
297
298 function cropped_img = symImgCrop(img, cutted_edge_size)
299     original_img_size = size(img);
300     original_max_row = original_img_size(1);
301     original_max_column = original_img_size(2);
302
303     cropped_img = zeros(original_max_row - 2*cutted_edge_size ,
304         original_max_column - 2*cutted_edge_size ,1);
305
306     for row=cutted_edge_size:original_max_row - cutted_edge_size
307         for col=cutted_edge_size:original_max_column - cutted_edge_size
308             cropped_img(row - cutted_edge_size + 1,col - cutted_edge_size
309                 + 1) = img(row,col);
310         end
311     end
312 end
313
314 function img_rot = four_point_img_rotation_crop(img, fourp)
315     % A function which crops and rotates the given image so that the only
316     % thing shown will be the pixels inside the four points, in
317     % horizontal
318     % or vertical position.
319     fourp_base = [0 fourp(1,2)-fourp(1,1) fourp(1,3)-fourp(1,1) fourp
320         (1,4)-fourp(1,1); 0 fourp(2,2)-fourp(2,1) fourp(2,3)-fourp(2,1)
321         fourp(2,4)-fourp(2,1)];
322     if (fourp_base(4,1)-fourp_base(1,1)) > 4
323         sigma = atan((fourp_base(2,4)-fourp_base(2,1))/(fourp_base(1,4)-
324             fourp_base(1,1)));
325     else
326         sigma = pi/2;
327     end
328     if sigma <= pi/4
329         rot_mat = [cos(-sigma) -sin(-sigma); sin(-sigma) cos(-sigma)];
330     else
331         rot_mat = [cos(pi/2 - sigma) -sin(pi/2 - sigma); sin(pi/2 -
332             sigma) cos(pi/2 - sigma)];
333     end
334 end
335
336 function img_crop = generic_crop(img, fourp)
337     % A function which crops the given image so that the edges are
338     % defined by the four point given in fourp.
339     X_ARRAY = [fourp(1,1) fourp(1,2) fourp(1,3) fourp(1,4)];
340     Y_ARRAY = [fourp(2,1) fourp(2,2) fourp(2,3) fourp(2,4)];
341     MIN_X = min(X_ARRAY);
342     MAX_X = max(X_ARRAY);
343     MIN_Y = min(Y_ARRAY);

```

```

339     MAX_Y = max(Y_ARRAY);
340     img_crop = zeros(MAX_X-MIN_X,MAX_Y-MIN_Y);
341
342     for row = MIN_X:MAX_X
343         for col = MIN_Y:MAX_Y
344             img_crop(row - MIN_X + 1,col - MIN_Y + 1) = img(row,col);
345         end
346     end
347 end

```

E: Matlab Code Septh Sensor

```
1 %processing the image using the depthsensor
2
3
4 %treshhold values
5 min_tresh = 30;
6 max_tresh = 500;
7
8 % get image from depth sensor
9 depth = getsnapshot(depthVid);
10
11 %run the sobel operator
12 shapes = sobel_operator(depth);
13
14 %run the treshhold filter
15 shapes = treshhold(shapes, min_tresh, max_tresh);
16
17 %look at the result
18 image(depth);
19
20 function shapes = sobel_operator(img)
21
22     X = img;
23     Gx = [1 +2 +1; 0 0 0; -1 -2 -1]; Gy = Gx';
24     temp_x = conv2(X, Gx, 'same');
25     temp_y = conv2(X, Gy, 'same');
26     shapes = sqrt(temp_x.^2 + temp_y.^2);
27 end
28
29 function treshholded = treshhold(img, min_tresh, max_tresh)
30
31     matrix_size = size(img);
32
33     MAXROW = matrix_size(1);
34
35     MAXCOLUMN = matrix_size(2);
36
37     for row = 1 : MAXROW
38         for col = 1: MAXCOLUMN
39             if (img(row, col) > min_tresh) && (img(row, col) < max_tresh)
40                 img(row, col) = 1;
41             else
42                 img(row, col) = 0;
43             end
44         end
45     end
46     treshholded = img;
47 end
```


F: Matlab Library Functions

```
1 %% Extract features
2 [labeled, numObjects] = bwlabel(Image_black);
3 stats = regionprops(labeled, 'Eccentricity', 'Area', 'BoundingBox');
4 eccentricities = [stats.Eccentricity];
5
6 %% Use feature analysis to count objects
7 idxOfObjects = find(eccentricities);
8
9 figure, imshow(Image);
10 hold on;
11 for idx = 1 : length(idxOfObjects)
12     h = rectangle('Position', stats(idx).BoundingBox);
13     set(h, 'EdgeColor', [.75 0 0]);
14     set(h, 'LineWidth', 2);
15 end
16
17 title(['There are ', num2str(numObjects), ' objects in the picture']);
18 hold off;
19
20 %% Draw boundaries
21 B = bwboundaries(Image_black);
22 imshow(Image);
23 hold on;
24 visboundaries(B);
25 hold off;
```

G: Gantt chart

teamgantt
Created with Free Edition

