

## 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.*

## Abstract

# Contents

<b>Contents</b>	<b>4</b>
<b>List of Tables</b>	<b>5</b>
<b>List of Figures</b>	<b>6</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Problem Description</b>	<b>1</b>
<b>3 Design</b>	<b>2</b>
3.1 Hardware . . . . .	2
3.2 Software . . . . .	3
3.2.1 Analysis Depth Sensor . . . . .	6
<b>4 Implementation</b>	<b>8</b>
<b>5 Further planning</b>	<b>10</b>
<b>6 Budget management</b>	<b>10</b>
<b>7 Course Integration</b>	<b>11</b>
<b>8 Conclusion</b>	<b>11</b>
<b>9 List of references</b>	<b>11</b>
<b>10 References</b>	<b>11</b>
<b>11 References</b>	<b>11</b>
<b>12 Appendix</b>	<b>11</b>

## List of Tables

## List of Figures

1	The example included in the assignment. . . . .	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) . . . . .	7
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 image processing, nowadays other image related problems are being solved by artificial intelligence. Most were considered to be an important part of digital image processing. Among these, the problem of this paper can be found: feature extraction. The ability to count objects in an image to be more exact. So why use 'traditional' methods to solve this problem? While being a great way for unravelling many problems, artificial intelligence mostly provides general solutions. However, 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.

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 !TITEL HARDWARE HERE!.

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.!REFERENTIE BOEK! These choices will be discussed in !TITEL SOFTWARE HERE!.

These methods, while being the core of the solution, are fairly simple to implement with the use of libraries or built-in functions. In this paper is opted to give a full implementation of these functions, limiting the usage of libraries to the minimum, in !TITEL IMPLEMENTATION HERE!. If the functions are deemed to be basic, only a simple explanation will be given.

## 2 Problem Description

The object counting system described in this report is capable of counting non-moving objects in a basket. These objects can vary in shape, size and colour. The colour of both the basket and its contents are free from restrictions as well as the shape of the objects.

In the primary stage of this paper, not all these variables are taken into account. The simplest objects, which the system is required to count, are 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: The example included in the assignment.

### 3 Design

#### 3.1 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 prior named necessities. Each of these hardware components is discussed in the following section.

Choosing the camera is a vital element 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 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  $84.1^\circ$  by  $53.8^\circ$ (REFERENTIE Smeenk). 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  $1080 \times 1920 \times 3$ . 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(REFERENCE researchgate). 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 half a meter and 4 meters. 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 via a virtual machine. The algorithm should run in an acceptable time frame on every machine.

To conclude this section 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 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 on FIG...XX... shows a couple of different methods. There is no 'right way' to count objects in an image. 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

These elements are also visible in the diagram(VERWIJZING NR DIAGRAM). In the next scope, three general methods are featured and briefly discussed. Each was investigated in prospect of this paper.

### Method 1

This method is the most simple and straightforward to implement. As input it requires a filtered greyscale image. This is passed through a thresholding algorithm with a pre-defined threshold value. The output is a binary matrix. This array only has 0's and 1's as elements, respectively representing the colours black and white. The key to solving the problem in this specific scheme is writing code that finds the threshold value based on environmental parameters. When in possession of a truly black and white image, a simple edge detection program is run which makes the edges visible.

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.

## Method 2

The second method tackles the colour analysis in the opposite order than the first method, as it starts with an edge detection algorithm. Since the input image is still very complex, this edge detection is way more comprehensive. The output is 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. Based on the fact that there is a lot of noise using this sequence of steps, it's recommended to include noise reduction code.

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.

## Method 3

The third way takes a different approach to solving the analysis of the colour image. When using this, a compromise in functionality is made. Since it needs a picture of the empty background without any objects, the user experience is worsened. After getting a background image, the picture of the situation with objects gets filtered and the algorithm converts it into a greyscale image. Using this less complex matrix, the code 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 on 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.

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. Note that the lighting conditions have to be unaffected in between taking the needed pictures for this algorithm.

## Implementation

After comparing these methods, the second method comes out as the better of the three. See Fig.2 for the comparison.

The first step, as seen above, is to convert the image to greyscale (*Greyscale*, n.d.). 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.

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

The weights used count up to 1 so the values in the greyscale image can vary from 0 to 255. All these values  $greyscale\_image(row, col)$  form the new 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 is applied. Most filters are a convolution of a kernel with the image. For a Gaussian blur the G kernel below is used. This is just a weighted average. The pixels centered around the main pixel have bigger weights than at the edges.



Figure 2: A comparison between the 3 different methods.

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

The blur can be applied by doing a convolution of the G matrix (the kernel) on the image matrix. The second blur is a mean blur (R. Fisher & Wolfart, n.d.-b). This is just the same, just another 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. // After both filters the image is ready to run through an edge detection algorithm(R. Fisher & Wolfart, n.d.-a). This algorithm is

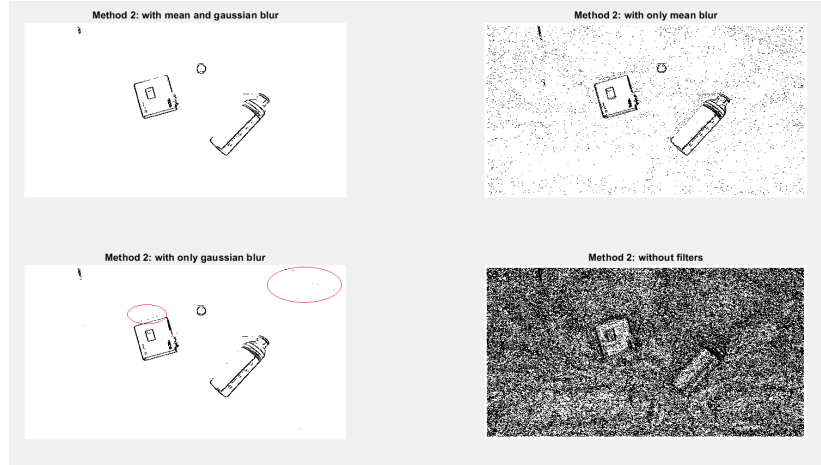


Figure 3: A comparison with the use of filters.

itself also a filter with kernel given by the matrix  $L$  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 filter before convolution with  $L$ , more 'edges' would have been drawn.

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.

The threshold algorithm, used in the following step, is based on this feature. This algorithm runs through to whole matrix and assigns each value with 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. After applying the algorithm, 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, but further research and programming has to be done to complete the whole program.

### 3.2.1 Analysis Depth Sensor

Using only the RGB image does have some shortcomings. It is rather difficult to distinguish an object from its shadow, a multicoloured object could be seen as multiple different objects and a lot 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 millimeters.

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 with a filter to get rid of the existing noise reduction. At last, 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 at page...**

### 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 this value will be different for different vertical positions of the kinect v2. Also, the image of the sensor contains some noise. For example: a picture of a big flat table will not be viewed as a equidistant surface. The elements of the matrix will be different. Another, and more preferred, method would be to use a Sobel-Feldman operator [**hier komt verwijzing naar boek in bronvermelding**]. 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:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

$$G = \sqrt{G_x^2 + G_y^2}$$

In the last equation,  $G$  is the magnitude of the total gradient as well as the value inserted in the new matrix.

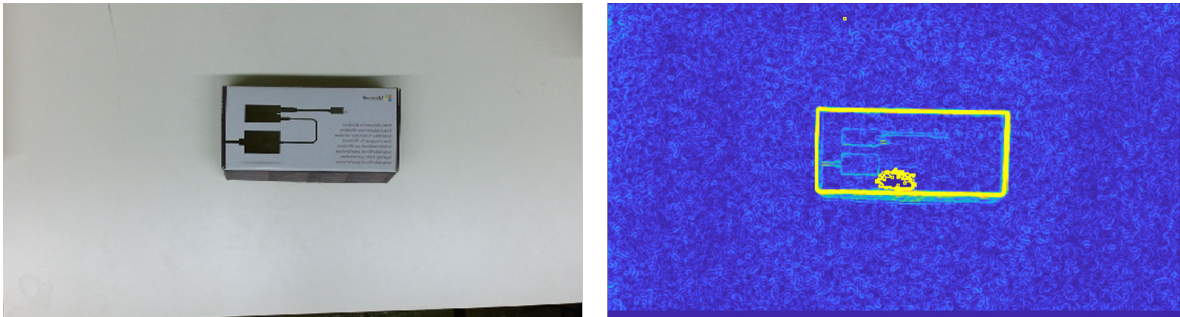


Figure 4: the original RGB image (left) and the image after the Sobel-Feldman operator (right)

### 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, like 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.

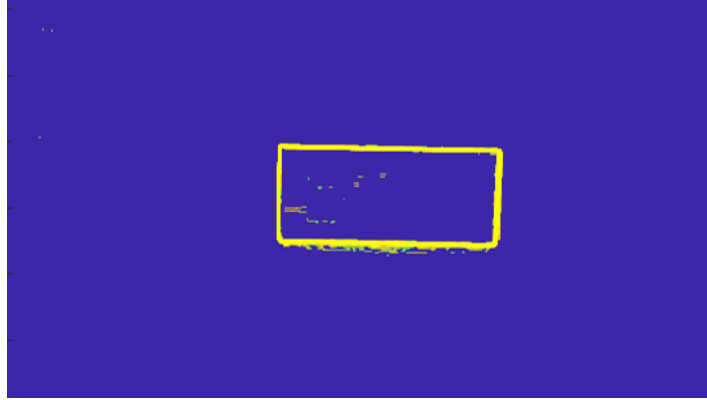


Figure 5: the original image after using a Sobel-Feldman operator and a threshold filter

## 4 Implementation

Throughout the project a lot of different approaches were tested and discarded. But in essence they all do the same thing they convert the original image to a binary image. In this binary image the objects are represented by one value and the background by another. Afterwards this binary image is analysed and a simple algorithm suffices to count the objects. In this fase of the program the same code is applicable. This code consists of a few important parts: the actual counting and the drawing of the boundary boxes.

Throughout the project a lot of different approaches were tested and discarded. But in essence ,they all do the same thing. They convert the original image to a binary image. Afterwards, this binary array is analysed and a simple algorithm suffices to count the objects. In this fase of the program the same code is applicable. This code consists of a few important parts: the actual counting and the drawing of the boundary boxes.

### 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 there exist a few functions that come in really handy for this kind of tasks. One of these functions `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 a count for how many objects there are.

### Boundary boxes

The image processing toolbox really simplifies the drawing of boundary boxes. Once a binary image is obtained the function `regionprops` 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 containing 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.

### Edge detection

There are a lot of ways to implement edge detection. Edge detection algorithms as described in PARAGRAPH exist for greyscale image. But if a binary image is available, this becomes much easier. For starters there exists a function in the image processing toolbox called `bwboundaries`. 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 become different figures. One with the image and another on top of it with the edges. The function `bwboundaries` implements the Moore-Neighbor tracing algorithm. 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 (See fig. 3).

This problem is resolved with the Jacob's stopping criterion. Which states that the algorithm can stop once the first start pixel is visited out of the same direction as it was initially

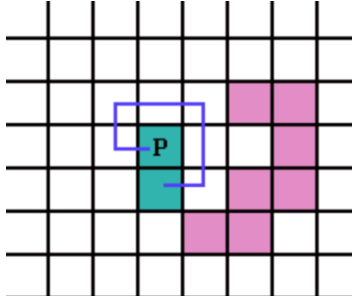


Figure 6: Problem with stopping criteria Moore-Neighbor tracing algorithm.

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.

## 5 Further planning

Being halfway through the project, a visual timeframe is created. In the appendices, a Gantt chart can be found.(APPENDIX) 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.

## 6 Budget management

As seen above, the system explained in this paper primarily consists of software which on its own doesn't cost anything. On the contrary, 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 EURO, this is feasible. Both the Kinect and the adapter have been ordered but as of writing this paper, 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.



## 7 Course Integration

For this project, some courses from the first three semesters are useful to be able to finish it. Linear algebra is used for working in the matrix the image represents. The programming in matlab is a lot easier after the course of computer science. Numerical Mathematics can be used for working with really large matrices. To make sure the system doesn't use too much memory, the course of information transmission and processing is used.

## 8 Conclusion

## 9 List of references

## 10 References

## 11 References

Dirac, P. A. M. (1981). *The principles of quantum mechanics*. Clarendon Press.

Einstein, A. (1905). Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]. *Annalen der Physik*, 322(10), 891–921. doi: <http://dx.doi.org/10.1002/andp.19053221004>

*Greyscale*. (n.d.). Retrieved from <https://en.wikipedia.org/wiki/Grayscale>

*Kernel (image processing)*. (n.d.). Retrieved from [https://en.wikipedia.org/wiki/Kernel\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel(image_processing))

Knuth, D. (1981). *Knuth: Computers and typesetting*. Retrieved from <http://www-cs-faculty.stanford.edu/~uno/abcde.html>

Knuth, D. E. (1973). Fundamental algorithms. In (chap. 1.2). Addison-Wesley.

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>

## 12 Appendix

### Matlab Code RGB sensor

```
1 <<<<<<<< HEAD
2 clearvars
3 img = imread('kinect/foto RGB 4.png');
4 %% Method 3: First greyscale, then threshold ivm background, then
   edge detection
```

```

5 A = greyscale(img); % Convert image to grayscale
6 A = symImgCrop(A, 50); % CROP IMAGE SO IT's the same size.
7 A = gaussian_blur(mean_blur(A)); % Filters
8
9 bg = imread('kinect/foto RGB 1.png');
10 bg = greyscale(bg); % Convert image to grayscale
11 bg = symImgCrop(bg, 50); % CROP IMAGE SO IT's the same size.
12 bg = gaussian_blur(mean_blur(bg)); % Filters
13
14 B = threshold_ivm_background(A, bg);
15 C = invertornot(B); % Check if threshold is OK or needs to be
    inverted
16 D = ~edge2_detect(C, 3);
17 E = remove_boundary(D, 25);
18 subplot(2,2,1), imshow(A, []);
19 title("Input (after blur)");
20 subplot(2,2,2), imshow(C, []);
21 title("After thresholding");
22 subplot(2,2,3), imshow(E, []);
23 title("After edge detection & boundary removed");
24 %% Method 1: First greyscale, then blur, then threshold, then edge
    detection.
25 A = greyscale(img); % Convert image to grayscale
26 A = symImgCrop(A, 50); % CROP IMAGE SO IT's the same size.
27 A = gaussian_blur(mean_blur(A)); % Filters
28 B = threshold(A); % Threshold image
29 C = invertornot(B); % Check if threshold is OK or needs to be
    inverted
30 D = ~edge2_detect(C, 3);
31 E = remove_boundary(D, 25);
32 subplot(2,2,1), imshow(A, []);
33 title("Input (after blur)");
34 subplot(2,2,2), imshow(C, []);
35 title("After thresholding");
36 subplot(2,2,3), imshow(E, []);
37 title("After edge detection & boundary removed");
38
39
40 %% Method 2: First greyscale, then blur, then edge detect then
    threshold and then noise removal
41 first_edge_detect = edge_detect(A);
42 without_noise_removal = threshold_edge(remove_boundary(
    first_edge_detect, 15));
43 with_noise_removal = noise_deletion(without_noise_removal, 3);
44 subplot(2,2,1), imshow(A, []);
45 title("Input (after blur)");

```

```

46 subplot(2,2,2), imshow(first_edge_detect , []);
47 title("After edge detection");
48 subplot(2,2,3), imshow(without_noise_removal , []);
49 title("Threshold without noise removal")
50 subplot(2,2,4), imshow(with_noise_removal , []);
51 title("Threshold with noise removal");
52
53 function result = threshold_ivm_background(img, bg)
54     % DIMENSIONS MUST MATCH
55     % Compare pixel at img(row, col) with bg(row, col).
56     % if bg(row, col) - D <= img(row, col) <= bg(row, col) + D
57     %     The pixels are defined as background!! (= white)
58
59     D = 10;
60     WHITE = 1;
61     BLACK = 0;
62
63     matrix_size = size(img);
64     MAXROW = matrix_size(1);
65     MAXCOLUMN = matrix_size(2);
66
67     result = zeros(MAXROW,MAXCOLUMN,1);
68     for row=1:MAXROW
69         for col=1:MAXCOLUMN
70             if img(row, col) <= bg(row, col) + D && img(row, col) >=
                bg(row, col) - D
71                 % Classified as background
72                 result(row, col) = WHITE;
73             else
74                 % Not background
75                 result(row, col) = BLACK;
76             end
77         end
78     end
79
80 end
81
82 function cropped_img = symImgCrop(img, cutted_edge_size)
83     original_img_size = size(img);
84     original_max_row = original_img_size(1);
85     original_max_column = original_img_size(2);
86
87     cropped_img = zeros(original_max_row - 2*cutted_edge_size ,
        original_max_column - 2*cutted_edge_size ,1);
88
89     for row=cutted_edge_size:original_max_row - cutted_edge_size

```

```

90         for col=cuttedge_size:original_max_column -
           cuttedge_size
91             cropped_img(row - cuttedge_size + 1,col -
              cuttedge_size + 1) = img(row,col);
92         end
93     end
94 end
95
96 function nes = noise_deletion(img,window)
97     matrix_size = size(img);
98     MAXROW = matrix_size(1);
99     MAXCOLUMN = matrix_size(2);
100     side = floor(window/2);
101     disp(floor((window^2)/2)+1);
102     nes = img;
103
104     for col=side+1:MAXCOLUMN-side
105         for row=side+1:MAXROW-side
106             list=zeros(window);
107             q=1;
108             for i=-side:side
109                 for j=-side:side
110                     list(q) = img(row+i , col+j);
111                     q = q+1;
112                 end
113             end
114             list=sort(list);
115             nes(row,col) = list(floor((window^2)/2)+1);
116         end
117     end
118 end
119
120 function result = remove_boundary(img, remove_size)
121     matrix_size = size(img);
122     MAXROW = matrix_size(1);
123     MAXCOLUMN = matrix_size(2);
124
125     result = zeros(MAXROW,MAXCOLUMN,1);
126     for row=1:MAXROW
127         for col=1:MAXCOLUMN
128             if row < remove_size || col < remove_size || row > (
              MAXROW - remove_size) || col > (MAXCOLUMN -
              remove_size)
129                 % Inside boundary ==> needs to be white (= 1)
130                 result(row, col) = 1;
131             else

```

```

132         result(row, col) = img(row, col);
133     end
134
135     end
136 end
137 end
138
139 function thresholded_img = threshold_edge(img)
140     threshold_value = 2;
141     %most_occuring = mode(img) +100;
142     %threshold_value = most_occuring(1);
143
144     matrix_size = size(img);
145     MAXROW = matrix_size(1);
146     MAXCOLUMN = matrix_size(2);
147     THICKNESS = 3;
148
149     thresholded_img = zeros(MAXROW,MAXCOLUMN,1);
150     for row=1:MAXROW
151         for col=1:MAXCOLUMN
152             if img(row, col) > threshold_value
153                 value = 0;
154                 for i=1:THICKNESS
155                     % Create thicker edges (edges of THICKNESS
156                     % pixels thick)
157                     if (col - i) > 0
158                         thresholded_img(row, col-i) = 0;
159                     end
160                 end
161             else
162                 value = 1;
163             end
164             thresholded_img(row, col) = value;
165         end
166     end
167
168 function threshold_value_calculated = determine_threshold_value(img
169 )
170     % By looking at the edge of the figure, determine background
171     % color.
172     % This color needs to be filtered out.
173     matrix_size = size(img);
174     MAXROW = matrix_size(1);
175     MAXCOLUMN = matrix_size(2);

```

```

175     number_of_edge_layers = 3; %3 rijen boven & onder en 3 kolommen
        links en rechts
176     values = 0;
177     counts = 0;
178     for row=1:MAXROW % We gaan elke rij af
179         for col=1:MAXCOLUMN
180             if col <= number_of_edge_layers || row <=
                number_of_edge_layers || col >= (MAXCOLUMN -
                number_of_edge_layers) || row >= (MAXROW -
                number_of_edge_layers)
181                 % Dit zijn de cellen tussen de rand, tel alle
                    waarden op en
182                 % neem gemiddelde.
183                 values = values + img(row, col);
184                 counts = counts + 1;
185             end
186         end
187     end
188
189     threshold_value_calculated = values / counts;
190     disp(threshold_value_calculated)
191 end
192
193 function mean_blurred = mean_blur(img)
194     mean = (1/9) * [ 1 1 1; 1 1 1; 1 1 1];
195     mean_blurred = conv2(img, mean);
196 end
197
198 function gaussian_blurred = gaussian_blur(img)
199     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];
200     gaussian_blurred = conv2(img, gaussian);
201 end
202
203 function edge2 = edge2_detect(img,intolerance)
204     matrix_size = size(img);
205     MAXROW = matrix_size(1);
206     MAXCOLUMN = matrix_size(2);
207     edge2 = zeros(MAXROW,MAXCOLUMN,1);
208     THICKNESS = 2;
209
210     % Horizontaal laten checken voor edges.
211     previous_value = img(1,1);
212     for row=1:MAXROW % We gaan elke rij af
213         for col=1:MAXCOLUMN
214             i=1;

```

```

215     flag = 0;
216     if img(row, col) == 1 && previous_value == 0
217         % DUS: Het begin van een object. (hele tijd wit, nu
            zwart), flag voor intolerantie controle
            aanzetten.
218         flag = 1;
219     elseif img(row, col) == 0 && previous_value == 1
220         % DUS: Het einde van een object (hele tijd zwart,
            nu wit), flag voor intolerantie controle
            aanzetten.
221         flag = 1;
222     end
223
224     %%Intolerantie controle
225     while i <= intolerance && flag && col+i <= MAXCOLUMN
226         if img(row,col-1+i) ~= img(row,col+i)
227             flag = 0;
228         end
229         i=i+1;
230     end
231
232     % Eertse maal edgematrix vullen
233     if flag
234         edge2(row, col) = 1;
235
236         for i=1:THICKNESS
237             % Create thicker edges (edges of THICKNESS
                pixels thick)
238             if (col - i) > 0
239                 edge2(row, col-i) = 1;
240             end
241         end
242     else
243         edge2(row, col) = 0;
244     end
245
246     previous_value = img(row, col);
247 end
248
249 previous_value = img(row,1);
250 end
251
252 % Verticaal controleren op edges.
253 previous_value = img(1,1);
254 for col=1:MAXCOLUMN % We gaan elke kolom af
255     for row=1:MAXROW

```

```

256         i=1;
257         flag = 0;
258         if img(row, col) == 1 && previous_value == 0
259             % DUS: Het begin van een object. (hele tijd wit, nu
                zwart), flag voor intolerantie controle
                aanzetten.
260             %value = 1;
261             flag = 1;
262         elseif img(row, col) == 0 && previous_value == 1
263             %DUS: Het einde van een object (hele tijd zwart,
                nu wit), flag voor intolerantie controle
                aanzetten.
264             %value = 1;
265             flag = 1;
266         end
267
268         % Intolerantie controle
269         while i <= intolerance && flag && row+i <= MAXROW
270             if img(row-1+i, col) ~= img(row+i, col)
271                 flag = 0;
272             end
273             i=i+1;
274         end
275
276         % Enkel nullen overriden
277         if flag
278             edge2(row, col) = 1;
279             for i=1:THICKNESS
280                 % Create thicker edges (edges of THICKNESS
                pixels thick)
281                 if (row - i) > 0
282                     edge2(row - i, col) = 1;
283                 end
284             end
285         end
286
287         previous_value = img(row, col);
288     end
289
290     previous_value = img(1, col);
291     end
292
293 end
294
295 function edge = edge_detect(img)
296     klaplace=[0 -1 0; -1 4 -1; 0 -1 0]; % Laplacian

```



```

        filter kernel
297     edge=conv2(img, klaplace);           % convolve
        test img with
298 end
299
300 function inverse = invertornot(img)
301     gem = mode(img);
302     disp(gem(1))
303     if gem(1) == 1
304         inverse = ~img;
305     else
306         inverse = img;
307     end
308 end
309
310 function thresholded_img2 = threshold2(img)
311     threshold_value = determine_threshold_value(img);
312     threshold_band = 220;
313
314     matrix_size = size(img);
315     MAXROW = matrix_size(1);
316     MAXCOLUMN = matrix_size(2);
317
318     thresholded_img2 = zeros(MAXROW,MAXCOLUMN,1);
319     for row=1:MAXROW
320         for col=1:MAXCOLUMN
321             if img(row, col) > (threshold_value - threshold_band)
322                 && img(row, col) < (threshold_value + threshold_band)
323                     value = 0;
324             else
325                 value = 1;
326             end
327             thresholded_img2(row, col) = value;
328         end
329     end
330
331 function grey = greyscale(img)
332     matrix_size = size(img);
333     MAXROW = matrix_size(1);
334     MAXCOLUMN = matrix_size(2);
335
336     grey = zeros(MAXROW,MAXCOLUMN,1);
337     for row=1:MAXROW
338         for col=1:MAXCOLUMN

```

```

339         R = img(row, col, 1);
340         G = img(row, col, 2);
341         B = img(row, col, 3);
342         grey(row, col) = 0.2989 * R + 0.5870 * G + 0.1140 * B ;
343         %These are two methods for grayscaling.
344         %grey(row, col) = (R + G + B)/3;
345     end
346 end
347 end
348
349 function thresholded_img = threshold(img)
350     threshold_value = 125;
351     %most_occurring = mode(img) + 100;
352     %threshold_value = most_occurring(1);
353
354     matrix_size = size(img);
355     MAXROW = matrix_size(1);
356     MAXCOLUMN = matrix_size(2);
357
358     thresholded_img = zeros(MAXROW, MAXCOLUMN, 1);
359     for row=1:MAXROW
360         for col=1:MAXCOLUMN
361             if img(row, col) > threshold_value
362                 value = 0;
363
364             else
365                 value = 1;
366             end
367             thresholded_img(row, col) = value;
368         end
369     end
370 end
371
372
373 =====
374 clearvars
375 img = imread('kinect/foto RGB 4.png');
376 %% Method 3: First greyscale, then trheshold ivm background, than
377     edge detection
378 A = greyscale(img); % Convert image to grayscale
379 A = symImgCrop(A, 50); % CROP IMAGE SO IT's the same size.
380 A = gaussian_blur(mean_blur(A)); % Filters
381
382 bg = imread('kinect/foto RGB 1.png');
383 bg = greyscale(bg); % Convert image to grayscale
384 bg = symImgCrop(bg, 50); % CROP IMAGE SO IT's the same size.

```

```

384 bg = gaussian_blur(mean_blur(bg)); % Filters
385
386 B = threshold_ivm_background(A, bg);
387 C = invertornot(B); % Check if threshold is OK or needs to be
    inverted
388 D = ~edge2_detect(C, 3);
389 E = remove_boundary(D, 25);
390 subplot(2,2,1), imshow(A, []);
391 title("Input (after blur)");
392 subplot(2,2,2), imshow(C, []);
393 title("After thresholding");
394 subplot(2,2,3), imshow(E, []);
395 title("After edge detection & boundary removed");
396 %% Method 1: First greyscale, then blur, then threshold, then edge
    detection.
397 A = greyscale(img); % Convert image to grayscale
398 A = symImgCrop(A, 50); % CROP IMAGE SO IT'S the same size.
399 A = gaussian_blur(mean_blur(A)); % Filters
400 B = threshold(A); % Threshold image
401 C = invertornot(B); % Check if threshold is OK or needs to be
    inverted
402 D = ~edge2_detect(C, 3);
403 E = remove_boundary(D, 25);
404 subplot(2,2,1), imshow(A, []);
405 title("Input (after blur)");
406 subplot(2,2,2), imshow(C, []);
407 title("After thresholding");
408 subplot(2,2,3), imshow(E, []);
409 title("After edge detection & boundary removed");
410
411
412 %% Method 2: First greyscale, then blur, then edge detect then
    threshold and then noise removal
413 first_edge_detect = edge_detect(A);
414 without_noise_removal = threshold_edge(remove_boundary(
    first_edge_detect, 15));
415 with_noise_removal = noise_deletion(without_noise_removal, 3);
416 subplot(2,2,1), imshow(A, []);
417 title("Input (after blur)");
418 subplot(2,2,2), imshow(first_edge_detect, []);
419 title("After edge detection");
420 subplot(2,2,3), imshow(without_noise_removal, []);
421 title("Threshold without noise removal")
422 subplot(2,2,4), imshow(with_noise_removal, []);
423 title("Threshold with noise removal");
424

```

```

425 function result = threshold_ivm_background(img, bg)
426     % DIMENSIONS MUST MATCH
427     % Compare pixel at img(row, col) with bg(row, col).
428     % if bg(row, col) - D <= img(row, col) <= bg(row, col) + D
429     %         The pixels are defined as background!! (= white)
430
431     D = 10;
432     WHITE = 1;
433     BLACK = 0;
434
435     matrix_size = size(img);
436     MAXROW = matrix_size(1);
437     MAXCOLUMN = matrix_size(2);
438
439     result = zeros(MAXROW,MAXCOLUMN,1);
440     for row=1:MAXROW
441         for col=1:MAXCOLUMN
442             if img(row, col) <= bg(row, col) + D && img(row, col) >=
443                 bg(row, col) - D
444                 % Classified as background
445                 result(row, col) = WHITE;
446             else
447                 % Not background
448                 result(row, col) = BLACK;
449             end
450         end
451     end
452 end
453
454 function cropped_img = symImgCrop(img, cutted_edge_size)
455     original_img_size = size(img);
456     original_max_row = original_img_size(1);
457     original_max_column = original_img_size(2);
458
459     cropped_img = zeros(original_max_row - 2*cutted_edge_size,
460         original_max_column - 2*cutted_edge_size, 1);
461
462     for row=cutted_edge_size:original_max_row - cutted_edge_size
463         for col=cutted_edge_size:original_max_column -
464             cutted_edge_size
465             cropped_img(row - cutted_edge_size + 1, col -
466                 cutted_edge_size + 1) = img(row, col);
467         end
468     end
469 end

```

```

467
468 function nes = noise_deletion(img,window)
469     matrix_size = size(img);
470     MAXROW = matrix_size(1);
471     MAXCOLUMN = matrix_size(2);
472     side = floor(window/2);
473     disp(floor((window^2)/2)+1);
474     nes = img;
475
476     for col=side+1:MAXCOLUMN-side
477         for row=side+1:MAXROW-side
478             list=zeros(window);
479             q=1;
480             for i=-side:side
481                 for j=-side:side
482                     list(q) = img(row+i , col+j);
483                     q = q+1;
484                 end
485             end
486             list=sort(list);
487             nes(row,col) = list(floor((window^2)/2)+1);
488         end
489     end
490 end
491
492 function result = remove_boundary(img, remove_size)
493     matrix_size = size(img);
494     MAXROW = matrix_size(1);
495     MAXCOLUMN = matrix_size(2);
496
497     result = zeros(MAXROW,MAXCOLUMN,1);
498     for row=1:MAXROW
499         for col=1:MAXCOLUMN
500             if row < remove_size || col < remove_size || row > (
                    MAXROW - remove_size) || col > (MAXCOLUMN -
                    remove_size)
501                 % Inside boundary ==> needs to be white (= 1)
502                 result(row, col) = 1;
503             else
504                 result(row, col) = img(row, col);
505             end
506         end
507     end
508 end
509 end
510

```

```

511 function thresholded_img = threshold_edge(img)
512     threshold_value = 2;
513     %most_occuring = mode(img) +100;
514     %threshold_value = most_occuring(1);
515
516     matrix_size = size(img);
517     MAXROW = matrix_size(1);
518     MAXCOLUMN = matrix_size(2);
519     THICKNESS = 3;
520
521     thresholded_img = zeros(MAXROW,MAXCOLUMN,1);
522     for row=1:MAXROW
523         for col=1:MAXCOLUMN
524             if img(row, col) > threshold_value
525                 value = 0;
526                 for i=1:THICKNESS
527                     % Create thicker edges (edges of THICKNESS
528                     % pixels thick)
529                     if (col - i) > 0
530                         thresholded_img(row, col-i) = 0;
531                     end
532                 end
533             else
534                 value = 1;
535             end
536             thresholded_img(row, col) = value;
537         end
538     end
539
540 function threshold_value_calculated = determine_threshold_value(img
541 )
542     % By looking at the edge of the figure, determine background
543     % color.
544     % This color needs to be filtered out.
545     matrix_size = size(img);
546     MAXROW = matrix_size(1);
547     MAXCOLUMN = matrix_size(2);
548
549     number_of_edge_layers = 3; %3 rijen boven & onder en 3 kolommen
550     % links en rechts
551     values = 0;
552     counts = 0;
553     for row=1:MAXROW % We gaan elke rij af
554         for col=1:MAXCOLUMN
555             if col <= number_of_edge_layers || row <=

```

```

        number_of_edge_layers || col >= (MAXCOLUMN -
        number_of_edge_layers) || row >= (MAXROW -
        number_of_edge_layers)
553     % Dit zijn de cellen tussen de rand, tel alle
        waarden op en
554     % neem gemiddelde.
555     values = values + img(row, col);
556     counts = counts + 1;
557     end
558     end
559     end
560
561     threshold_value_calculated = values / counts;
562     disp(threshold_value_calculated)
563 end
564
565 function mean_blurred = mean_blur(img)
566     mean = (1/9) * [ 1 1 1; 1 1 1; 1 1 1];
567     mean_blurred = conv2(img, mean);
568 end
569
570 function gaussian_blurred = gaussian_blur(img)
571     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;]
572     gaussian_blurred = conv2(img, gaussian);
573 end
574
575 function edge2 = edge2_detect(img,intolerance)
576     matrix_size = size(img);
577     MAXROW = matrix_size(1);
578     MAXCOLUMN = matrix_size(2);
579     edge2 = zeros(MAXROW,MAXCOLUMN,1);
580     THICKNESS = 2;
581
582     % Horizontaal laten checken voor edges.
583     previous_value = img(1,1);
584     for row=1:MAXROW % We gaan elke rij af
585         for col=1:MAXCOLUMN
586             i=1;
587             flag = 0;
588             if img(row, col) == 1 && previous_value == 0
589                 % DUS: Het begin van een object. (hele tijd wit, nu
                    zwart), flag voor intolerantie controle
                    aanzetten.
590                 flag = 1;
591             elseif img(row, col) == 0 && previous_value == 1

```

```

592         % DUS: Het einde van een object (hele tijd zwart,
           nu wit), flag voor intolerantie controle
           aanzetten.
593         flag = 1;
594     end
595
596     %%Intolerantie controle
597     while i <= intolerance && flag && col+i <= MAX.COLUMN
598         if img(row,col-1+i) ~= img(row,col+i)
599             flag = 0;
600         end
601         i=i+1;
602     end
603
604     % Eertse maal edgematrix vullen
605     if flag
606         edge2(row, col) = 1;
607
608         for i=1:THICKNESS
609             % Create thicker edges (edges of THICKNESS
               pixels thick)
610             if (col - i) > 0
611                 edge2(row, col-i) = 1;
612             end
613         end
614     else
615         edge2(row, col) = 0;
616     end
617
618     previous_value = img(row, col);
619 end
620
621 previous_value = img(row,1);
622 end
623
624 % Verticaal controleren op edges.
625 previous_value = img(1,1);
626 for col=1:MAX.COLUMN % We gaan elke kolom af
627     for row=1:MAX.ROW
628         i=1;
629         flag = 0;
630         if img(row, col) == 1 && previous_value == 0
631             % DUS: Het begin van een object. (hele tijd wit, nu
               zwart), flag voor intolerantie controle
               aanzetten.
632             %value = 1;

```



```

633         flag = 1;
634     elseif img(row, col) == 0 && previous_value == 1
635         %DUS: Het einde van een object (hele tijd zwart,
        nu wit), flag voor intolerantie controle
        aanzetten.
636         %value = 1;
637         flag = 1;
638     end
639
640     % Intolerantie controle
641     while i <= intolerance && flag && row+i <= MAXROW
642         if img(row-1+i, col) ~= img(row+i, col)
643             flag = 0;
644         end
645         i=i+1;
646     end
647
648     % Enkel nullen overriden
649     if flag
650         edge2(row, col) = 1;
651         for i=1:THICKNESS
652             % Create thicker edges (edges of THICKNESS
             pixels thick)
653             if (row - i) > 0
654                 edge2(row - i, col) = 1;
655             end
656         end
657     end
658
659     previous_value = img(row, col);
660 end
661
662 previous_value = img(1,col);
663 end
664
665 end
666
667 function edge = edge_detect(img)
668     klaplace=[0 -1 0; -1 4 -1; 0 -1 0];
669     filter kernel
670     edge=conv2(img,klaplace);
671     test img with
672
673     % Laplacian
674     % convolve
675
676 function inverse = invertornot(img)
677     gem = mode(img);

```

```

674     disp(gem(1))
675     if gem(1) == 1
676         inverse = ~img;
677     else
678         inverse = img;
679     end
680 end
681
682 function thresholded_img2 = threshold2(img)
683     threshold_value = determine_threshold_value(img);
684     threshold_band = 220;
685
686     matrix_size = size(img);
687     MAXROW = matrix_size(1);
688     MAXCOLUMN = matrix_size(2);
689
690     thresholded_img2 = zeros(MAXROW,MAXCOLUMN,1);
691     for row=1:MAXROW
692         for col=1:MAXCOLUMN
693             if img(row, col) > (threshold_value - threshold_band)
694                 && img(row, col) < (threshold_value + threshold_band)
695                     value = 0;
696             else
697                 value = 1;
698             end
699             thresholded_img2(row, col) = value;
700         end
701     end
702
703 function grey = greyscale(img)
704     matrix_size = size(img);
705     MAXROW = matrix_size(1);
706     MAXCOLUMN = matrix_size(2);
707
708     grey = zeros(MAXROW,MAXCOLUMN,1);
709     for row=1:MAXROW
710         for col=1:MAXCOLUMN
711             R = img(row, col, 1);
712             G = img(row, col, 2);
713             B = img(row, col, 3);
714             grey(row, col) = 0.2989 * R + 0.5870 * G + 0.1140 * B ;
715             %These are two methods for grayscaling.
716             %grey(row, col) = (R + G + B)/3;
717         end

```

```

718     end
719 end
720
721 function thresholded_img = threshold(img)
722     threshold_value = 125;
723     %most_occurring = mode(img) + 100;
724     %threshold_value = most_occurring(1);
725
726     matrix_size = size(img);
727     MAXROW = matrix_size(1);
728     MAXCOLUMN = matrix_size(2);
729
730     thresholded_img = zeros(MAXROW,MAXCOLUMN,1);
731     for row=1:MAXROW
732         for col=1:MAXCOLUMN
733             if img(row, col) > threshold_value
734                 value = 0;
735
736             else
737                 value = 1;
738             end
739             thresholded_img(row, col) = value;
740         end
741     end
742 end
743
744
745 >>>>>>> f57d7bae21cb23a358a76864756de25e897d8715

```

## Matlab Code depth 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 = treshold(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 tresholded = treshold(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     tresholded = img;
47 end

```