Problem Solving and Engineering Design part 3

ESAT1A1

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

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

PRELIMINARY REPORT

Co-titular
Tinne Tuytelaars

Coaches
Xuanli Chen
José Oramas

ACADEMIC YEAR 2018-2019

Declaration of originality

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

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

Contents

List of Figures

1 Introduction

Digital image processing has been a crucial part of the current digitalisation movement. From industrial machinery to customer amusement, the vision of computer-aided systems has become a given for most users. While image alteration and manipulation remain a core part of this 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. 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 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° 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 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.

3.2.1 Analysis RGB Sensor

In the domain of the RGB sensor, no exception will be found.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 trough a thresholding algorithm with a pre-defined threshold value. The output is a binary matrix. Thus, this array only has 0's and 1's as elements, 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 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 that one from method one. The output still 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.

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 first step, as discussed 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. A linear combination is made.

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

The used coefficients 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 found below is used. This is

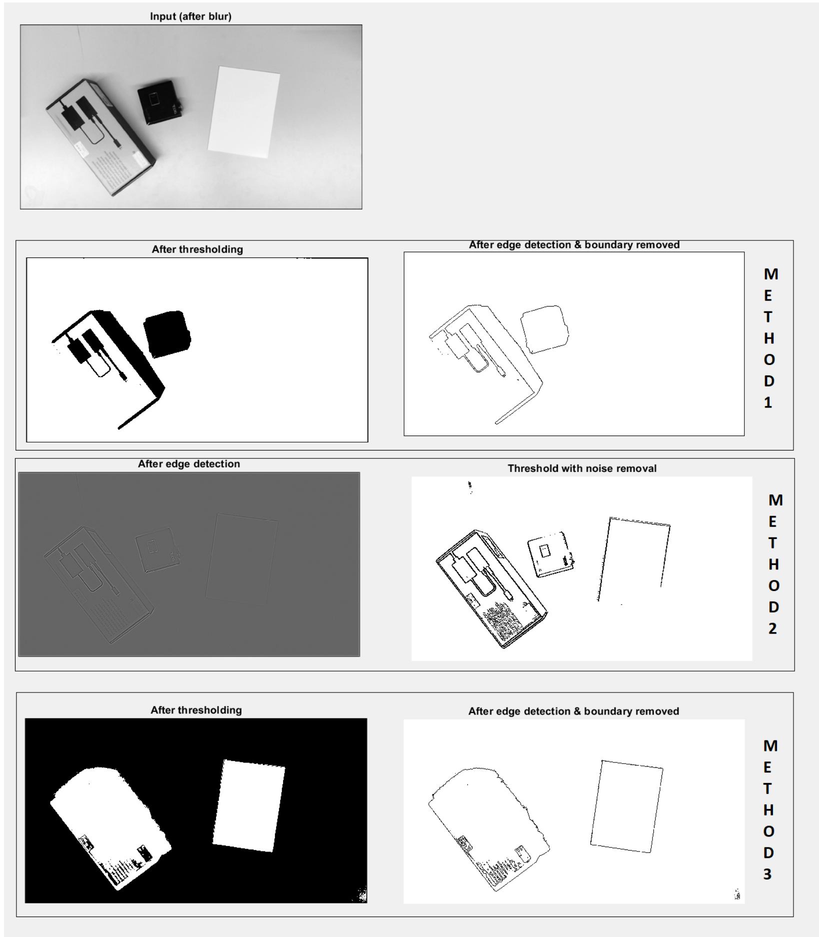


Figure 2: A comparison between the 3 different methods.

just a weighted average. This has as consequence that the elements centered around the main pixel have bigger weights than those at the edges.

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

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

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

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

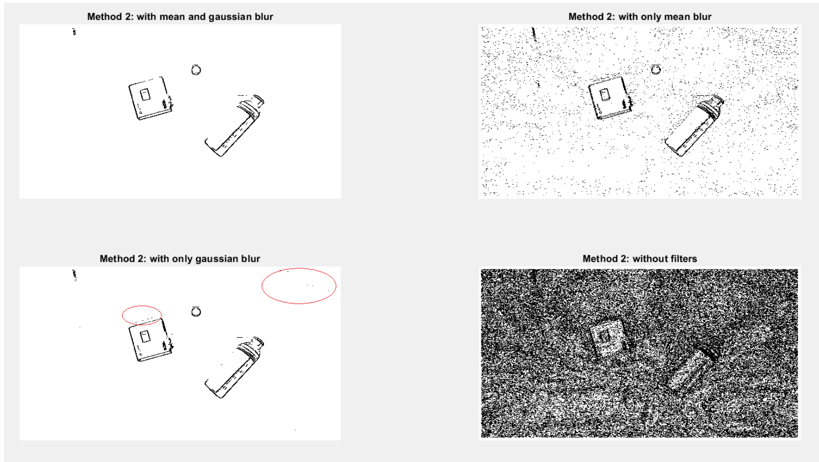


Figure 3: A comparison with the use of filters.

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 done by the matrix L given below. It calculates the *spatial derivative* or in simpler words, it highlights regions of rapid intensity change.

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

Note now how the kernel uses the pixels next to the evaluated pixel to see how much intensity changes. If the image wouldn't have been filtered before convolution with L, more 'edges' would have been drawn because more unregular 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. A different number type is used. 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. 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, but further research and programming has to be done to complete the whole program.

3.2.2 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 enrichening the object counting algorithm with the usage of a depth sensor is advised. Like featured in the section about the hardware, each element of the input data represents a distance in 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 (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.

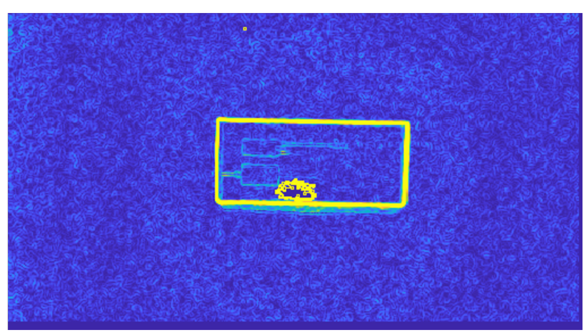


Figure 4: The original RGB image (left) and the image after the Sobel-Feldman operator (right)

Filtering of the noise

After adding al the different magnitudes of the gradients to an array, some anomalies still exist. There can be some impossible elements, 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 treshold: The maximum treshold 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 treshold can be decided by empirical research. Values lower than this value can be seen as noise and thus can be set to zero.

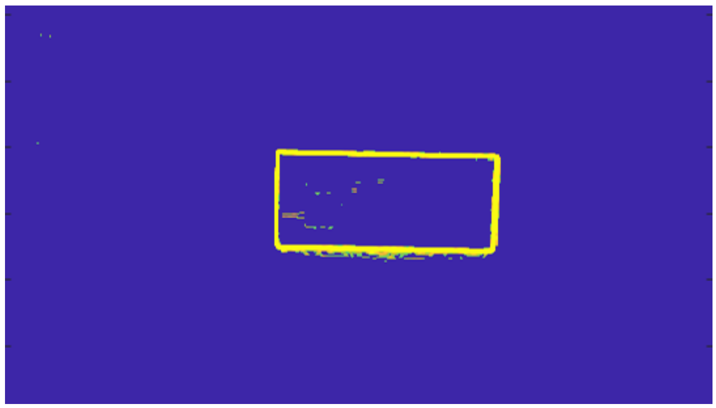


Figure 5: The original image after using a Sobel-Feldman operator and a threshold filter

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. 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(*Mathworks*, 2018), a few functions exist that are very usefull 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:

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

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

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

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:

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

where BW represents the input, this is a binary image which only consists out of black and white pixels; and B represents the output, which consist out of a cell array with N elements (number of image regions in the binary image), all these elements contain a list of the boundary pixels. Which in turn are fairly easy to draw. They can be inserted in the matrix of the image by replacing values, this is done by looping through the cell arrays. The advantage of this method is that the image can actually be printed. When they are drawn on top of the image with a function like visboundaries the actual values of the pixels stay unchanged, but it 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 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.

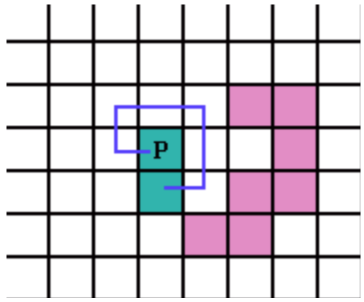


Figure 6: Problem with stopping criteria Moore-Neighbor tracing algorithm.

5 Further planning

Being halfway trough 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 at the institute of engineering science of the KUL are useful. These following courses are used to look in a more mathematical and structured way at the investigated data.
Viewing the fact that an image is represented by an array, Linear Algebra is very important. From basic operations like multiplying matrices up untill more difficult acts like convoluting

a matrix, has algebra an important role. Note that each picture taken from the Kinect v2 is viewed as a large matrix with roughly 350000 elements. A knowledge of Numerical Mathematics is advised to get an idea which influence measurement errors have on all of the calculations with this very large matrices. This course, together with Applied Informatics in Python, gives the tools to investigate the time complexity of the algorithm. When this project continues in a further stage, the course of Information Transmission and Processing could be used. Given this knowledge, the efficient use of memory during the algorithm can be monitored.

8 Conclusion

After four weeks of group gatherings, a lot 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. But perseverance and a critical point of view are needed to successfully finish this assignment.

9 References

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

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

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$_{f}ig1_{3}21$

Kernel (image processing). (n.d.). Retrieved from https://en.wikipedia.org/wiki/Kernel$_{(}image_{p}rocessing)$

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

10 Appendix

A: Matlab Code RGB sensor

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

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

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

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

%% Method 3: First greyscale, then blur, then edge detect then
    threshold and then noise removal
first_edge_detect = edge_detect(A); % Laplacian edge detection
without_noise_removal = threshold_edge(remove_boundary(
    first_edge_detect, 15)); % Remove boundary around image &
    threshold the edges.
with_noise_removal = noise_deletion(without_noise_removal, 3); %
```

```
    Noise removal
subplot(2,2,1), imshow(A, []);
title("Input (after blur)");
subplot(2,2,2), imshow(first_edge_detect, []);
title("After edge detection");
subplot(2,2,3), imshow(without_noise_removal, []);
title("Threshold without noise removal");
subplot(2,2,4), imshow(with_noise_removal, []);
title("Method 2 with gaussian and mean blur");


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

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

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

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

end

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

```

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

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

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

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

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

    result = zeros(MAX_ROW,MAX_COLUMN,1);
    for row=1:MAX_ROW
        for col=1:MAX_COLUMN
            if row < remove_size || col < remove_size || row > (MAX_ROW - remove_size) || col > (MAX_COLUMN - remove_size)
```

```
                % Inside boundary ==> needs to be white (= 1)
                result(row, col) = 1;
            else
                result(row, col) = img(row, col);
            end

        end
    end
end

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

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

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

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

function gaussian_blurred = gaussian_blur(img)
```

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

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

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

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

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

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

```
                        edge2(row, col-i) = 1;
                    end
                end
            else
                edge2(row, col) = 0;
            end

            previous_value = img(row, col);
        end

    previous_value = img(row,1);
    end

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

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

            % Enkel nullen overriden
            if flag
                edge2(row, col) = 1;
                for i=1:THICKNESS
                    % Create thicker edges (edges of THICKNESS
```

```
                    pixels thick)
                    if (row - i) > 0
                        edge2(row - i, col) = 1;
                    end
                end
            end

            previous_value = img(row, col);
        end

    previous_value = img(1,col);
    end

end

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

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

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

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

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

function grey = greyscale(img)
```

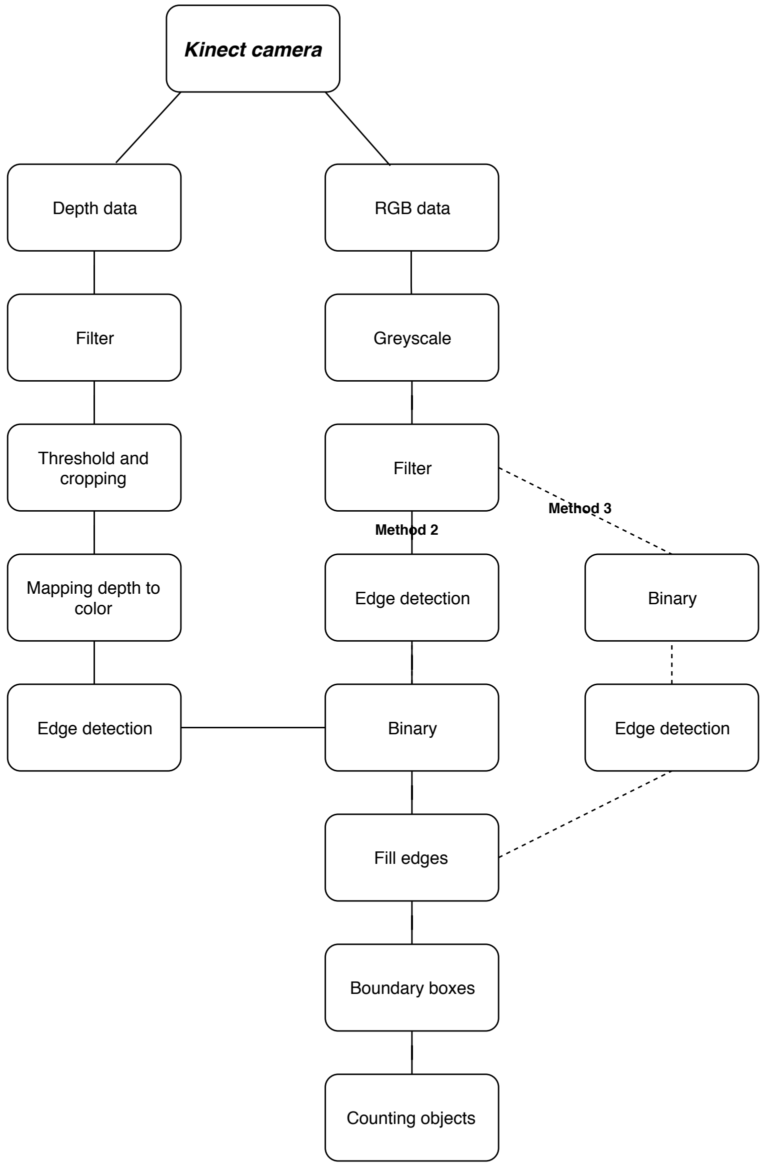
```
    matrix_size = size(img);
    MAX_ROW = matrix_size(1);
    MAX_COLUMN = matrix_size(2);

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

B: Matlab Code depth sensor

```
%processing the image using the depthsensor


%treshold values
min_tresh = 30;
max_tresh = 500;

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

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

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

%look at the result
image(depth);

function shapes = sobel_operator(img)

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

function tresholded = treshold(img, min_tresh, max_tresh)

    matrix_size = size(img);

    MAX_ROW = matrix_size(1);

    MAX_COLUMN = matrix_size(2);

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

```
            end
        end
        tresholded = img;
end
```

C: Diagram



Kinect camera
Depth data
RGB data
Filter
Greyscale
Threshold and cropping
Filter
Method 3
Method 2
Mapping depth to color
Edge detection
Binary
Edge detection
Binary
Edge detection
Fill edges
Boundary boxes
Counting objects

D: Gantt chart

teamgantt
Created with Free Edition