Problem Solving and Engineering Design part 3

ESAT7A1

Senne Bosmans
Polo De Bliek
Brecht De Cock
Annelies De Geeter
Robin Degezelle
Olivia Gevers

Automatic counting of objects in an image through image processing

PRELIMINARY REPORT

<u>Co-titular</u>
Gorjan Radevski

<u>Coach</u>(<u>es</u>)
Gorjan Radevski

ACADEMIC YEAR 2019 - 2020

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

The following paper describes a project conducted by students under the guidance of the **P**rocessing **S**peech and **I**mages lab at the KU Leuven. The goal of the project is to count the number of objects automatically in a controlled environment using a digital camera. With the help of the Microsoft Kinect$^{\text{TM}}$ designed for the Xbox One, a picture will be taken. Using digital image processing technology written in Python, a computer has to determine the number of objects as seen in the picture. This paper will detail all the different aspects of the software; from connecting the camera to displaying the end result, the actual hardware setup of the scanner, the underlying mathematics and science going on behind the project and our actual problems encountered during the duration of this project.

The first part of the algorithm makes it possible to fetch the data from the camera. The next step is processing the image to a simpler image which makes it possible to detect the objects. This is done by grayscaling and blurring the image followed by executing an edge detection algorithm called Sobel and sharpening the edges using a technique called hysteresis. Once the image is processed an object counting algorithm should be able to count the number of objects. In the final step the number of objects will be displayed.

Contents

List of Figures

1 Introduction

Digital image processing is used in a lot of applications, going from industrial machinery to the medical world. The conventional method for object counting is manual, time consuming and could affect the accuracy. When objects occur in a large number and overlap, counting the objects manually is tricky and leads to errors. That is why counting with computer vision is a better alternative. This way counting objects in a packaging line or cells in biological research will be a lot easier and reliable.

The goal of the project is to create a visual system to count objects. The object counting system described in the report is able to count non-moving objects in a rectangular basket. Based on color and depth images, the system will return the number of objects to the user. These articles have variable shapes, sizes and colors. The budget of the project is €250.

In this paper the focus lies on object counting based on images gained from a camera. The choice of hardware will be an important part that will be covered in section 2: Hardware. There, all the hardware components and the visual setup will be discussed.

The algorithm is the most important part of the system. Without the software, the items in the image can't be counted. The algorithm happens in a number of steps. First a picture is taken with a Kinect camera. Second, image processing will change the image in the following consecutive steps: grayscale, Gaussian blur, Sobel and hysteresis. The algorithm shall end with the object counting. These steps are described in section 3: Software.

For giving a demo it isn't exciting to just display the number of objects. To show all the steps of the image processing a graphic user interface is created which shows all the different steps. This interface is discussed in section 4: User interface.

The financial aspect of the project can be found in Section 5: Budget management.

Section 6: Course Integration, talks about all the subjects that were needed to complete the project. At last, section 7: Conclusion, describes the future planning of the project and gives a conclusion for every section.

2 Hardware

The setup of the object counting system will be quite simple. It consists of a Kinect camera, a computer and a cable which connects the camera to the computer. In order to get all the objects in the right position for the object counting to work, there are other parameters that need to be defined. All these essential elements are discussed in this section.

2.1 Hardware Components

To take a picture, a camera is a necessity. There are three options for a camera: an industrial camera, a web cam and a camera with depth sensors. To define which option is the best choice, the advantages and disadvantages from all three will be compared with its pros and cons. A web cam has a low cost but doesn't come with a good image quality. An industrial camera gives high quality images but a decent one is expensive. Because the budget is limited to €250 this is not a usable camera. The last option, a camera with depth sensors is in the price range and has an extra advantage. Besides an RGB image, it gives easily accessible data about the depth. Extra data gives more opportunities and possibilities to solve the problem.

Considering all of the above, the best option is a camera with depth sensors. More specifically, the system will be based on a Xbox One Kinect Sensor from Microsoft. The color lens of the Kinect V2 sensor is an RGB camera with a resolution of 1920×1080 pixels (Lachat, Macher, Landes, Grussenmeyer, 2015). This camera has a field of view of $84.1° \times 53.8°$. The resolution ensures a matrix representation of the real image where every element corresponds with a pixel of the image that varies between 0 and 255. The value given to a pixel corresponds to an intensity of the color per layer. Each color frame pulled from the Kinect is represented by an array structure of $1920 \times 1080 \times 3$, which can be separated into three different matrices. These are based on the three main colors: red, green and blue.

Alongside the color lens, the Kinect has a depth sensing lens based on an infrared (IR) camera. The infrared camera has a resolution of 512×424 pixels and a field of view of $70° \times 60°$. First, to retrieve a 3D depth scan, the data of a depth image is given in a matrix where every number represents the distance in millimetres from the depth sensor to the object. To get a correct image the object must be in the operative measuring range, 0.5 to 4.5m (Lachat, 2015).

Second, to get a visual image, a computer with Python version 3.7 is required. In a modern computer the processor should be good enough to run the algorithm in an acceptable time frame.

Third, a transfer cable is needed to connect the camera to the computer. Connecting the Kinect to a computer needs to be done with a Kinect Adapter for Windows. It consists of two main parts. One part transfers the data recorded by the camera to the computer and the other supplies power to the Kinect camera.

2.2 Visual setup

To keep the camera from moving and to get an image from the same angle every time the algorithm runs, a solid setup for the camera is needed. This will also help to take better pictures which leads to a more accurate result. A tripod will be stationed next to a table and should hold the Kinect camera in place. The camera should hang approximately one meter above the table and will be tightly fitted in a 3D-printed container. This container will make sure the ventilators and sensors aren't blocked and that the camera is attached to the tripod. Note that this setup hasn't been built yet and could be changed in the future.

To further help the recognition program, a clean white background (as bottom layer of the basket) will be used. The color white gives the highest contrast with the objects which will be placed in the basket and should help to find edges. To aid the image processing program even more, four lights will be placed in the upper corners of the basket. This is done to eliminate as much shadow as possible which is left behind by the objects. The usage of four lights will eliminate more shadow than e.g. two lights. This is shown in the figure below which simulates the effect of the light sources in Blender[1].

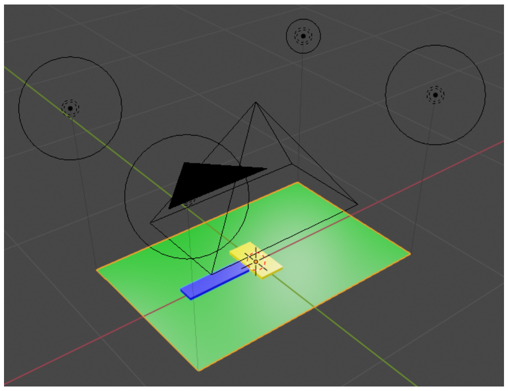

Figure 1: Setup with four lights and two objects

[1]Blender is an open-source, free 3D-imaging/-modelling software used to create several test-case images for the software to run on, it is used to control and change many variables (lighting, color, size of objects, etc.) quickly and efficiently

3 Software

Digital image processing is the most important part of this project. To fetch data, process the image and count the objects several algorithms need to be written. These algorithms have all been implemented using Python as programming language. Python was chosen because it has a wide range of open-source libraries which can be used in our advantage. However, the general approach as described in this paragraph can also be implemented in other programming languages e.g. `MATLAB`.

3.1 Taking a picture with the Kinect camera

An important part in the software is fetching the data. The data needed to count the objects are a color image and a depth frame. Using a cable as described in section 2: Hardware, the camera is connected to the computer. To take a color picture, the library 'Pykinect2' is used and the algorithm works with the function kinect_to_pc appendix 9 (lines 109-154). This function makes it possible to choose the width, height and dimension of the image as a variable. When running the function the Kinect will first take an image and afterwards stores the image in a folder of choice.

The data received is a row matrix with 2 073 600 elements. To make it easier to work with, the matrix is reshaped to a 1920×1080 matrix which is the size of the actual image. Because the Kinect sends an RGBA (Red, Green, Blue and Alpha) image, the next step converts it to the RGB image. The last step is flipping the image left to right because the image is mirrored. It's important to note that it's not actually needed to flip the image in order to do the object counting but this way it looks more natural to display the results. Taking a depth frame is quite similar. However, the data send from the Kinect is a row matrix with 217 088 elements. Once again, the matrix is reshaped to a 512×424 matrix.

3.2 Digital image processing

Before the object counting can be done, the image needs to be processed. The goal of the image processing is to convert the image to an image that is easier to use, and in turn which makes it possible to better count the objects. The final image will be an image displaying only the edges of the objects which is called a binary edge map. The result of each step used in this program are displayed in subsection 3.3.

3.2.1 Grayscale

The first step to prime the Kinect image is to apply a grayscale filter. Grayscale images are widely used in image processing and are utilized for measuring the intensity of light in images, using the following equation:

$$Y = 0.299R + 0.587G + 0.114B$$

in which Y represents the light intensity. The reason for using grayscale is more on the subtle side. Having info about colors increases the complexity of the model. So using an image processing tool using gray level images, as opposed to color, will reduce the computing time.

3.2.2 Gaussian blur

To discuss Gaussian blur and how it's performed, the process of image convolution with kernels must first be addressed. A kernel is a square matrix of an undetermined size while the image has also been transformed to a matrix with each element representing a pixel. In this case the image has been transformed to a grayscale as discussed in the previous subsection (3.2.1) so each element in the image matrix represents the light intensity of pixel in the image. Throughout this explanation, *matrix* will reference the image matrix with elements representing pixels, while a 3×3 unspecified kernel will be used for this example. In a color image there would be three or four separate matrices representing the Red, Green, Blue (and Alpha) intensities on which convolution would have to be applied. With convolution, for each pixel the same process is repeated. First, the selected pixel's surrounding elements are taken so that it forms a matrix of the same size as the kernel, with the selected pixel in the middle of the matrix. In the case of convoluting the pixel p_5 of the image matrix in I (highlighted in yellow) the 3×3 matrix in P is created. When a pixel at the edge needs to be convoluted the missing elements are filled by creating a symmetrical matrix with the non-missing items as can be seen in matrix R around pixel r_1.
Then, each element of the selected matrix is multiplied by the element in the same place in the kernel. So element p_1 in matrix P at location (1, 1) is multiplied by element k_1 in the kernel because it's also at location (1, 1). This produces the matrix C, of which each element is summed to give the end result c for pixel p_5.

$$I = \begin{bmatrix} r_1 & r_2 & \cdots & & & & \\ r_3 & r_4 & & & & & \\ \vdots & & \ddots & & \vdots & & \iddots \\ & & & p_1 & p_2 & p_3 & \\ & & & p_4 & p_5 & p_6 & \\ & & & p_7 & p_8 & p_9 & \\ & & & & \vdots & & \ddots \end{bmatrix}$$

$$P = \begin{bmatrix} p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \end{bmatrix} \qquad R = \begin{bmatrix} r_4 & r_3 & r_4 \\ r_2 & r_1 & r_2 \\ r_4 & r_3 & r_4 \end{bmatrix}$$

$$K = \begin{bmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{bmatrix} \quad C = \begin{bmatrix} k_1 \cdot p_1 & k_2 \cdot p_2 & k_3 \cdot p_3 \\ k_4 \cdot p_4 & k_5 \cdot p_5 & k_6 \cdot p_6 \\ k_7 \cdot p_7 & k_8 \cdot p_8 & k_9 \cdot p_9 \end{bmatrix}$$

$$c = k_1 \cdot p_1 + k_2 \cdot p_2 + k_3 \cdot p_3 + k_4 \cdot p_4 + k_5 \cdot p_5 + k_6 \cdot p_6 + k_7 \cdot p_7 + k_8 \cdot p_8 + k_9 \cdot p_9$$

This process of convolution is used for applying Gaussian blur. Gaussian blur is a filter with the following kernel:

$$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}$$

this is to remove noise coming from high ISO or sensor errors. During the course of the project, further experimentation with Gaussian blur will be researched in terms of how many times this blur should be performed or if larger kernels (such as 9×9) would be options for optimal results .

3.2.3 Sobel

The goal of the Sobel algorithm is to create an image emphasising the edges of the objects. This edge detection algorithm is chosen because it's well documented and relatively inexpensive in terms of calculations (Wikipedia, 2019, Sobel Operators).

Sobel uses two different 3×3 kernels. The following one is used for horizontal changes of colors:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

And the next one for vertical changes:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

These kernels are convolved separately over the blurred image to return an image of respectively the horizontal and vertical color changes.
The first matrix calculates the sum of the values of the pixels on the left hand side of the evaluated pixel and subtracts it from the sum of the values of the pixels on the right hand side of that pixel. In this calculation, color changes of the pixels on the same row of the evaluated pixel are more important and thus given a weight of (positive or negative) two, compared to the pixels located diagonally to the evaluated pixel which are given a weight of (positive or negative) one. The pixels on the same column of the current pixel don't contribute anything when it comes to detecting horizontal color changes and therefore have a weight of zero.
The second matrix is the transposed version of the first matrix. This is because it uses the same principle as the latter matrix, but for detecting vertical color changes.

The final step of the Sobel algorithm is combining the previously mentioned images of the horizontal and vertical color changes. As mentioned above, the values of the pixels in these images are an indication of the degree of the change of color. Therefore they can be viewed as an approximation of respectively the horizontal and vertical partial (color) derivatives which together form the gradient of the image. Combining these two derivatives is the same as calculating the length of the gradient, i.e. the square root of the sum of the squares of the pixel values. The result is a new image with bright and dull edges of the input image, as can be seen in figure 6.

3.2.4 Hysteresis

The Sobel algorithm returns an image in which all the edges have been marked. This includes small edges in the background for example. To count the objects, the only edges that need to be marked are those of the objects. To reach this goal, the next step in the image processing algorithm is a thresholding technique called hysteresis. The result of this algorithm will be a binary edge map in which the edges are drawn more precisely than after the Sobel algorithm (Medina-Carnicer, Madrid-Cuevas, Carmona-Poyato, Munoz-Salinas, 2008).

Before hysteresis can be applied, a high and a low threshold have to be determined. The best values for these thresholds still have to be experimentally decided. As illustrated in figure 2, hysteresis looks at two given thresholds (low and high) and the current value of a pixel. It will compare the current value of the pixel with the low threshold. If the value of the pixel is lower, it is marked as a '`0`'. Pixels with a greater value than the value of the high threshold are marked as '`1`'. For pixel values between the two thresholds, an extra step is needed. If a pixel is connected to at least one pixel marked as '`1`', it is given the value '`1`'. Otherwise it is marked as '`0`'.

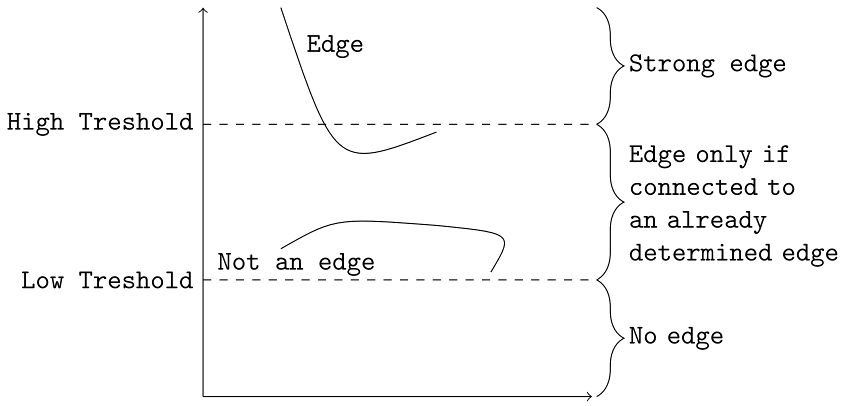

Figure 2: Hysteresis method

3.3 The results of the algorithm step by step

The following images show the different steps in the algorithm. Five variously sized test objects were used to test the correctness of the algorithm. Figure 3 shows the original image before performing the algorithm. Following, the image is grayscaled and blurred as shown in figures 4 and 5. After blurring the image, Sobel is performed and hysteresis is applied. This is displayed in figures 6 and 7.

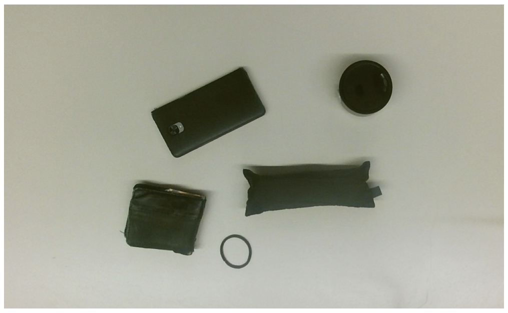
Figure 3: Original RGB image

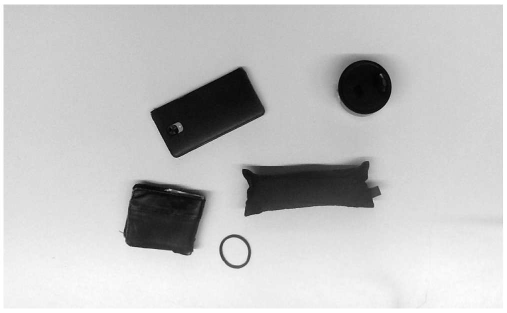
Figure 4: Grayscaled image

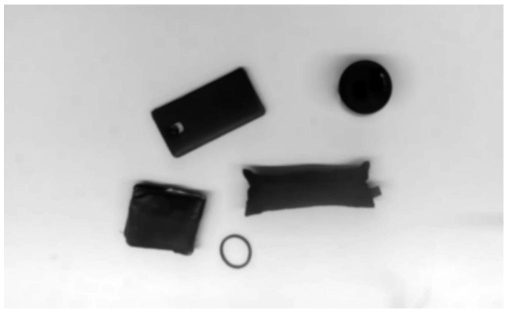
Figure 5: Blurred image

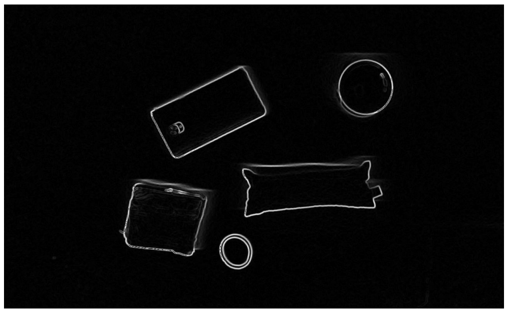
Figure 6: After performing Sobel

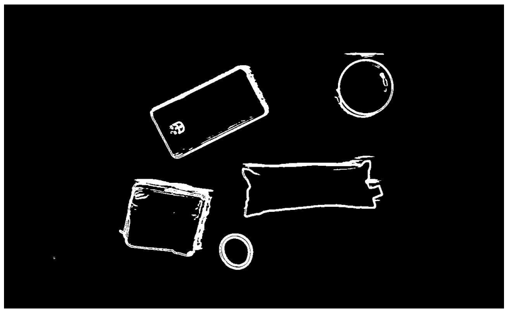
Figure 7: Applied hysteresis

3.4 Object counting

To count the objects in an image, it is important to make a difference between the actual objects and the noise. To filter out the points that are not part of any object, several clustering algorithms exist. The algorithm discussed here is DBSCAN (Density-Based Spatial Clustering of Applications with Noise). This is a density based clustering algorithm, meaning that given a dataset of points, it groups those who are close together. First, to obtain this dataset, a binary matrix is derived from the image through the processing techniques explained earlier. This binary image is a matrix that consists of 0's and 1's where '0' stands for *empty* and '1' for *possibly part of an object*. The dataset given to the clustering algorithm is then an array of the coordinates of all the 1's in this binary matrix. The algorithm requires two parameters: the maximum radius of the neighborhood (the ϵ -neighborhood) and the minimum number of points to form a dense region (minPts). Running through the algorithm, the points in the dataset can be classified as core points, density reachable points or noise.

- A point p is a core point if at least minPts are within a distance ϵ of p.
- A point p is density reachable from core point q if there is a chain of points p_1,...,p_n with p_1=q and p_n = p such that p_{i+1} is in the ϵ -neighborhood of p_i .
- All points that are not reachable from any core point are classified as noise.

A cluster consists of at least one core point and forms a cluster together with all points that are reachable from it. So a cluster starts with one core point and builds by recursively taking a core point and finding all its neighbors that are core points, finding all of their neighbors etc. The algorithm stops when all remaining points are visited.

Figure 8 is an example of the DBSCAN algorithm with four as parameter for the minimum points to form a dense region. The red points are core points, the yellow points are border points. These are still part of the cluster but don't have at least four points within an ϵ distance. The blue point is noise.

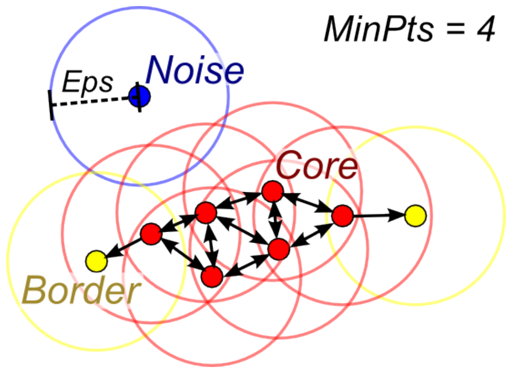

Figure 8: Illustration of DBscan

The algorithm starts at a random point, for which all the points within the ϵ neighbourhood are calculated. If there are at least as many points as the given minPts parameter, a new cluster is formed. Otherwise, this point is classified as noise. Mind that later on, this point can still be found in an epsilon distance from another point and hence be part of a cluster.

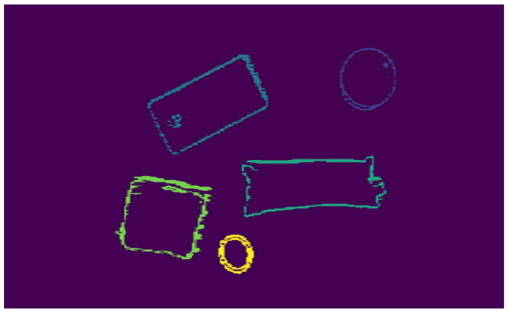

Figure 9: After DBSCAN algorithm

Figure 7 and 9 show the image before and after using the DBSCAN algorithm. A different color means a separate cluster. The number of clusters equals the number of objects. So in this image the number of objects is five.

The object at the top left corner has an inner set of points within the rectangle. To determine whether or not this inner and outer set of points are part of the same cluster, it is crucial to choose the parameters ϵ and minPts carefully. For this example ϵ equals 10 and minPoints equals 20. Unfortunately, these parameters don't work on every image. A goal later on in this project is to find parameters that can be generalized.

4 User interface

All algorithms, simple or complex, are only as good as their usability. Therefore, it is crucial to have an intuitive module to easily operate and interact with the algorithm. A GUI (**G**raphical **U**ser **I**nterface) provides a user friendly environment to run an algorithm and inspect its results, without interfering with or having to struggle through the actual code.

The GUI that is linked to the algorithm is made with the *tkinter* module and is shown in figure 10. It supplies the user with the option to take a new picture or select an already existing picture to run the algorithm on, while also giving them the option to either do or don't show the steps of the process and the final result. This way, testing the code is quick and easy, and at the end of the project it will provide an appealing and efficient way for the user to work with the algorithm.

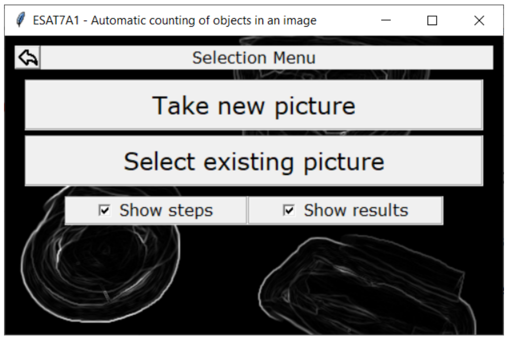

Figure 10: User interface

5 Budget management

The budget for the project is limited to €250. A large part of the project consists of software. The software is free of any costs. The hardware however does cost money. Luckily the Kinect Camera and Kinect Adapter for Windows are provided by the faculty. The estimated cost of these components are €180 for the Kinect V2 and €55 for the adapter. The other hardware elements for the setup of the camera, like the tripod and the 3D-printed case, still have to be bought. The price of these components will later be determined.

6 Course integration

This course is part of the Bachelor of Engineering curriculum of the KU Leuven. It is a course which uses concepts seen in others to solve problems. To design an object counting system, concepts of different courses which are given in the first and second year of the curriculum are used.

Knowledge of linear algebra is applied extensively. In the written software images are represented by matrices in which every element is a pixel. Simple concepts like matrix multiplication to more advanced ones e.g. matrix convolution form the core of the implemented algorithms. The programming language Python was used in 'Applied Informatics' and is used for the software. The general concepts of programming and the right way to tackle programming problems were learned during this course. Later on, concepts and techniques seen in the courses 'Mechanics' and 'Materials science' will be used to build the construction which holds the camera.

7 Conclusion

The camera which will take a picture of the objects is a Kinect V2. It can take depth and color frames. The construction which will hold the camera isn't build yet but the current idea is to place a tripod holding the Kinect above a table in a 3D-printed container. On the table the basket with the objects will be placed. The bottom layer from the basket will be white. In this setup the shape of the 3D-printed container and the size of the tripod are still variables. The variables will be determined and the construction build as soon as possible. This ensures there's enough time to test the counting in real-life situations and to solve occuring problems.

The global algorithm consists of two main parts. The first part is the processing of the image (taken by the Kinect) to a binary image in which the edges are indicated. This is done following these consecutive steps or algorithms: grayscaling, Gaussian blur, Sobel and hysteresis. Once this is done, the second important part can start. The object counting system makes use of the DBSCAN algorithm. It's a clustering algorithm meaning it groups the pixels that are close enough together. These groups are counted and the number of groups determine the number of objects in the basket. The processing of the image will be refined in the upcoming weeks. The idea is to make the edges smaller which makes the algorithm faster. Currently the total processing time is about three seconds but it varies with each image. The object counting algorithm has to be tested and the parameters defined in the algorithm need to be determined in order to make it work for every situation.

In the current algorithm the depth frames from the Kinect are not used. The idea is to implement an algorithm which counts the number of objects independently from the current algorithm using only the depth frames. The results from both images will be compared in a way the right number of objects are calculated.

8 References

[1] Lachat, E., Macher, H., Landes, T., Grussenmeyer, P.(2015). *scannerAssessment and Calibration of a RGB-D Camera (Kinect v2 Sensor) Towards a Potential Use for Close-Range 3D Modeling.* Remote sensing, 7, 13074.

[2] Medina-Carnicer, R., Madrid-Cuevas, F.J., Carmona-Poyato, A., Munoz-Salinas, R.(2008). *On candidates selection for hysteresis thresholds in edge detection.* Retrieved October 23, 2019 from `https://www.sciencedirect.com/science/article/abs/pii/S0031320308004664#!`

[3] Sachan,A.(2019).*Deep Learning based Edge Detection in OpenCV.* Retrieved October 12, 2019 from `https://cv-tricks.com/opencv-dnn/edge-detection-hed/`

[4] Sobel Operator. (2019). In Wikipedia. Retrieved October 23, 2019 from `https://en.wikipedia.org/wiki/Sobel_operator`

9 Appendices

1. Current image processing algorithm

```python
#import cv2
#from pykinect2 import PyKinectV2
#from pykinect2 import PyKinectRuntime

import numpy as np
from statistics import mean
from PIL import Image
import os
import matplotlib.pyplot as plt
from scipy import ndimage
import time

#Test Variables
timed                 = True
kinectFreeTesting     = True
printing              = False
show                  = False
gauss_repetitions     = 1
grayscale_save        = False
gauss_save            = False
sobel_save            = False
hysteresis_save       = False


if timed:
    time_gem_kleur   = 0
    time_flatten     = 0
    time_reconvert   = 0
    time_detection   = 0
    time_grayscale   = 0
    time_sobel       = 0
    time_Gaussian    = 0
    time_Hysteris    = 0
    time_processing = 0



###### HULPFUNCTIES #######
def gem_kleur_van_pixels(picture):
    if timed:
        t0 = time.time()
    image = np.array(picture).astype(np.uint8)

    gem_kleur_van_pixel = []
    for eachRow in image:
        for eachPix in eachRow:
            avgColor = mean(eachPix[:3])  # eerste 3 getallen vd array die de kleur geven
            gem_kleur_van_pixel.append(avgColor)
    if timed:
        t1 = time.time()
        global time_gem_kleur
        time_gem_kleur = t1 - t0
        print("IMPORTANT:", t1-t0)
    return gem_kleur_van_pixel


def flatten_matrix(matrix):
    if timed:
        t0 = time.time()
    image = np.array(matrix).astype(np.uint8)
    if timed:
        t1 = time.time()
        global time_flatten
        time_flatten = t1 - t0
    return image.flatten()


def reconvert_to_img(hyst_array, height, width, name_image):
    if timed:
        t0 = time.time()
    """
    :return: a saveable reconstructed image of the array that comes out of hysteresis
    """
    arary_Bool2Num = 255*hyst_array.astype(np.uint8)
    # reconstruct the (h,w,3)-matrix
    reconverted_array = np.reshape(arary_Bool2Num, (height, width))
    reconverted_image = reconverted_array.astype(np.uint8)  # the values need to be uint8 types

    # save and show the image
    # plt.figure()
    # plt.title('Processed image')
    # plt.imsave('../processed_images/' + name_image[:-4] + '_processed.png', reconverted_image, cmap='gray', format='png')
    # plt.imshow(reconverted_image, cmap='gray')
    #
    # plt.show()
    if timed:
        t1 = time.time()
        global time_reconvert
        time_reconvert = t1 - t0
    return reconverted_array


def make_detection_matrix(hyst_array, h, w):
    if timed:
        t0 = time.time()
    """
    :return: a matrix consisting of 0's and 1's for the object detection
    """
    array_Bool2Bin = hyst_array.astype(int)
    matrix = np.reshape(array_Bool2Bin, (h, w))
    if timed:
        t1 = time.time()
        global time_detection
        time_detection = t1 - t0
    return matrix


####### TAKING PICTURES #######
def kinect_to_pc(width, height, dimension):
    if timed:
        t0 = time.time()
    # https://github.com/daan/calibrating-with-python-opencv/blob/02c90e4291adfb2426072f8f0837033754fc3a55/kinect-v2/color.py
    if printing:
        print("connecting with kinect...")
    kinect = PyKinectRuntime.PyKinectRuntime(PyKinectV2.FrameSourceTypes_Color)
    if printing:
        print("connected to kinect")
    noPicture = True

    color_flipped = None  # give them a default value
    colorized_frame = None

    while noPicture:
        if kinect.has_new_color_frame():
            color_frame = kinect.get_last_color_frame()
            noPicture = False

            color_frame = color_frame.reshape((width,height,dimension))
            color_frame = cv2.cvtColor(color_frame, cv2.COLOR_BGRA2BGR)
            color_flipped = cv2.flip(color_frame, 1)

            cv2.imwrite("../input_images/KinectPicture.png", color_flipped)  # Save

    depth_image_size = (424, 512)

    kinect2 = PyKinectRuntime.PyKinectRuntime(PyKinectV2.FrameSourceTypes_Depth)
    noDepth = True

    while noDepth:
        if kinect2.has_new_depth_frame():
```

```python
            depth_frame = kinect2.get_last_depth_frame()
            noDepth = False

            depth_frame = depth_frame.reshape(depth_image_size)
            depth_frame = depth_frame * (256.0 / np.amax(depth_frame))
            colorized_frame = cv2.applyColorMap(np.uint8(depth_frame), cv2.COLORMAP_JET)
        #cv2.imshow('depth', colorized_frame)
            cv2.imwrite("../input_images/kinectDepthPicture.png", colorized_frame)  # Save
    if printing:
        print("pictures taken")
    if timed:
        t1 = time.time()
        print("kinect_to_pc", t1-t0)
    return color_flipped, colorized_frame


####### IMAGE PROCESSING #######
def grayscale(image):
    if timed:
        t0 = time.time()
    Image = (0.3 * image[:, :, 0] + 0.59 * image[:, :, 1] + 0.11 * image[:, :, 2]).astype(np.uint8)
    if timed:
        t1 = time.time()
        global time_grayscale
        time_grayscale = t1 - t0
    return Image


def gaussian_blur(image):
    if timed:
        t0 = time.time()
    h, w = image.shape
    GaussianKernel = np.array([[1 / 16, 1 / 8, 1 / 16], [1 / 8, 1 / 4, 1 / 8], [1 / 16, 1 / 8, 1 / 16]])
    newImg = ndimage.convolve(image, GaussianKernel)
    if timed:
        t1 = time.time()
        global time_Gaussian
        time_Gaussian = t1 - t0
    return newImg


def sobel(image):
    if timed:
        t0 = time.time()
    # get dimensions
    h, w = image.shape
    # define filters
    horizontal = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])  # s2
    vertical = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])  # s1

    # initialize new images
    newHorizontalImage = ndimage.convolve(image, horizontal)
    newVerticalImage = ndimage.convolve(image, vertical)
    newVerticalImage = np.square(newVerticalImage)
    newHorizontalImage = np.square(newHorizontalImage)
    newSum = newHorizontalImage + newVerticalImage
    newSum = np.sqrt(newSum)
    if timed:
        t1 = time.time()
        global time_sobel
        time_sobel = t1 - t0
    return newSum


def hysteresis(sobel_image, th_lo, th_hi, initial = False):
    if timed:
        t0 = time.time()
    """
    x : Numpy Array
        Series to apply hysteresis to.
    th_lo : float or int
        Below this threshold the value of hyst will be False (0).
    th_hi : float or int
        Above this threshold the value of hyst will be True (1).
    """

    # convert the image to an array
    # x = np.array(gem_kleur_van_pixels(sobel_image))  # enkel als ge het met een opgeslagen afbeelding moet doen
    x = np.array(flatten_matrix(sobel_image))  # sobel returns a 2D matrix now instead of an image

    if th_lo > th_hi: # If thresholds are reversed, x must be reversed as well
        x = x[::-1]
        th_lo, th_hi = th_hi, th_lo
        rev = True
    else:
        rev = False

    hi = x >= th_hi
    lo_or_hi = (x <= th_lo) | hi

    ind = np.nonzero(lo_or_hi)[0]  # Index f r alle darunter oder dar ber
    if not ind.size:  # prevent index error if ind is empty
        x_hyst = np.zeros_like(x, dtype=bool) | initial
    else:
        cnt = np.cumsum(lo_or_hi)  # from 0 to len(x)
        x_hyst = np.where(cnt, hi[ind[cnt-1]], initial)

    if rev:
        x_hyst = x_hyst[::-1]
    if timed:
        t1 = time.time()
        global time_Hysteris
        time_Hysteris = t1 - t0
    return x_hyst


# start the processing/filtering loop
def process_image(image):
    if timed:
        t0 = time.time()
    if printing:
        print("start image processing")
    # if the image doesn't come straight from the Kinect, but is a selected picture, open the selected picture
    if isinstance(image, str):
        name_image = image  # 'image' is a string
        image = np.array(Image.open("C:\\Users\\Polo\\Documents\\GitHub\\ESAT7A1\\testImages\\" + image))  # .astype(np.uint8)
    else:
        name_image = "KinectPicture"
    global gauss_repetitions, gauss_save, grayscale_save, sobel_save, hysteresis_save
    # constants
    low_threshold, high_threshold = 10, 27  # the thresholds for hysteresis
    h, w, d = image.shape  # the height, width and depth of the image
    if printing:
        print(h,w)

    # image processing/filtering process
    gray_image = grayscale(image)                      #153
    if grayscale_save:
        plt.imsave('C:\\Users\\Polo\\Documents\\GitHub\\ESAT7A1\\Gray.jpg', gray_image, cmap='gray', format='jpg')
    for i in range(gauss_repetitions):
        blurred_image = gaussian_blur(gray_image)      #163
    if gauss_save:
        plt.imsave('C:\\Users\\Polo\\Documents\\GitHub\\ESAT7A1\\Gauss.jpg', blurred_image, cmap='gray', format='jpg')
    sobel_image = sobel(blurred_image)                 #176
    if sobel_save:
```

```python
        plt.imsave('C:\\Users\\Polo\\Documents\\GitHub\\ESAT7A1\\Sobel.jpg', sobel_image, cmap='gray', format='jpg')
    hyst_image = hysteresis(sobel_image, low_threshold, high_threshold) #200
    reconvert_to_img(hyst_image, h, w, name_image) #050 , if you want to save the image (as e.g. a .png)
    if hysteresis_save:
        plt.imsave('C:\\Users\\Polo\\Documents\\GitHub\\ESAT7A1\\Hyst.jpg', hyst_image, cmap='RGB', format='jpg')
    tbReturned = make_detection_matrix(hyst_image, h, w) #081
    if timed:
        t1 = time.time()
        global time_processing
        time_processing = t1 - t0
    return tbReturned


####### MAIN #######
def main():
    t0 = time.time()
    # 1) take a picture
    if kinectFreeTesting:
        color_image = "HighRes.jpg"
    else:
        color_image, depth_image = kinect_to_pc(1080, 1920, 4)
    # 2) start the image processing
    matrix = process_image(color_image) #241

    t2 = time.time()
    if timed:
        print("PROCESSING:                ", time_processing)
        print("|-> Grayscale:             ", time_grayscale)
        print("|-> Gaussian :             ", time_Gaussian)
        print("|-> Sobel    :             ", time_sobel)
        print("|-> Hysteris :             ", time_Hysteris)
        print("    |-> Flatten Matrix:    ", time_flatten)
        print("|-> Reconvert:             ", time_reconvert)
        print("|-> Make Detection :       ", time_detection)
        print("==========================================")
        print("TOTAL:                     ", t2-t0)


if __name__ == '__main__':

    main()
```

2. Current object counting algorithm

```python
import numpy as np
import pickle
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.cluster import DBSCAN


matrix = pickle.load(open("kinectfoto_detection_matrix2.pkl", "rb"))
matrix = np.array(matrix)
image = Image.fromarray(matrix)
image = image.resize(size=(int(len(matrix) / 2), int(len(matrix[0]) / 2)))
matrix = np.array(image)
nb_columns = len(matrix[0])
nb_rows = len(matrix)
d = []

# DBSCAN needs a dataset of the coordinates of all the 1's in the matrix
def matrix_to_coordinates():
    for row in range(nb_rows):
        for column in range(nb_columns):
            if matrix[row][column] == 1:
                d.extend(np.array([[row, column]]))


def plot_image():
    for i in range(len(d)):
        row = d[i][0]
        column = d[i][1]
        matrix[row][column] = db.labels_[i]
    plt.imshow(matrix)
    plt.show()


matrix_to_coordinates()
db = DBSCAN(eps=3, min_samples=5).fit(d)
plot_image()
print("NUMBER OF OBJECTS:", max(db.labels_))
```