Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor Thesis

Robot Reading

presented by

Maik Philipp Simke

Student ID No. 7199787
B.Sc. Software-System-Engineering

Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics

submitted on 19.05.2023

First Reviewer: Dr. Norman Hendrich
Second Reviewer: Dr. Christian Wilms

Task

Despite the many advancements and steadily increasing accuracy of object detection systems and frameworks, led by the further development of neural networks and an increasing availability of OCR engines for text recognition, there still doesn't exist a software framework that allows a robot to detect and read text in an everyday environment.

This bachelor's thesis seeks out a generalized approach for detecting everyday objects via a simple camera image and transforming them in a way that allows for the recognition of larger text on flat surfaces, like the title of a book or the label and name of packaging and boxes, with the intention that the findings can be later applied in a real-life context using a robot platform, either as a stand-alone reading task or as part of a larger robot system taking advantage of text recognition.

Abstract

Over the last few years, text recognition gained an increasingly bigger influence in the fields of computer vision and especially robotics. While object detection and text recognition are two widely researched and explored fields, there currently does not exist a widespread framework that would allow robots to make use of it to accomplish complex reading tasks in a scene recognition environment.

In this bachelor's thesis, we introduce an approach for a generalized end-to-end processing pipeline for object and text recognition that only takes a simple RGB image from a camera as its input. The object is detected and transformed into an optimized representation for text recognition and identified using multiple open-source OCR programs.

The approach developed as part of this thesis was evaluated on test images captured by a PR2 robot. Results show that while the text recognition is not severely improved by the detection and subsequent transformation of the object in question, it might still be a necessary step for the text recognition to produce any useful output, as text in an unprocessed image might not be able to be recognized. However, evaluation has also shown that text recognition on an unprocessed image generally produces results robust enough to be used in a robot context.

Contents

1 Introduction **1**

2 Fundamentals **2**
2.1 Basics of Computer Vision . . . 2
2.1.1 OpenCV . . . 2
2.1.2 Grayscale and Color Image . . . 2
2.1.3 Binarization . . . 3
2.1.4 Brightness and Contrast . . . 4
2.1.5 Brightness and Contrast Correction . . . 4
2.1.6 Convolution and Kernels . . . 4
2.1.7 Blur . . . 5
2.1.8 Matrix Transformation . . . 5
2.2 Object Detection . . . 6
2.2.1 Morphological Operations . . . 6
2.2.2 Edge Detection . . . 7
2.2.3 Contour Detection . . . 8
2.2.4 Intersection over Union . . . 9
2.2.5 Object Detection utilizing Neural Networks . . . 9
2.3 Optical Character and Text Recognition . . . 10
2.3.1 OCR Programs . . . 10
2.3.2 String Matching and Text Similarity . . . 11
2.4 ROS and PR2 . . . 12

3 Related Work **13**
3.1 Trixi the Librarian . . . 13
3.1.1 Task, Approaches and Results . . . 13
3.1.2 Possible Improvements . . . 14
3.2 State of the Art . . . 15

4 Implementation **16**
4.1 Environment and Setup . . . 16
4.2 Overview of the Image Processing Pipeline . . . 16
4.3 Preprocessing, Object Detection and Perspective Transformation . . . 17
4.3.1 Automatic Brightness and Contrast Correction . . . 17
4.3.2 Morphological Closing . . . 19
4.3.3 Edge and Contour Detection . . . 20
4.3.4 Perspective Transformation . . . 22
4.4 Text Recognition . . . 23
4.4.1 Binarization . . . 23
4.4.2 Rotations . . . 24
4.4.3 Text Recognition using OCR . . . 24

5 Evaluation 25
5.1 Evaluating Object Detection . . . 25
5.2 Quantifying and Evaluating Text Similarity . . . 26
5.2.1 String Matching . . . 26
5.2.2 Determining Text Similarity . . . 27
5.3 Testing Environment . . . 28
5.4 Evaluation of the Pipeline . . . 30
5.4.1 Evaluating Individual Steps of the Pipeline . . . 30
5.4.2 Examples of Common Errors . . . 32
5.5 Evaluation of the Text Recognition and Results . . . 37
5.5.1 Evaluation of the Entire Pipeline . . . 37
5.5.2 Evaluation of Text Recognition without Perspective Transformation . . 38
5.5.3 Evaluation of Text Similarity with and without Perspective Transformation 39

6 Summary and Outlook 41

Appendices 42

A Excerpt of Test Images 43

List of Figures, Tables and Listings 49

7 References 51

1 Introduction

With the advent of more and more robots entering spaces formerly only inhabited by humans [34], it becomes more critical for robots to understand their surroundings and working environment, especially if it is not predetermined and can change over time. One such aspect, that would help the robot navigate its surroundings, is the ability to understand and read written text from either signs or objects in their working context. Examples of such cases are autonomous cars or service robots, that would need to detect and understand text to accomplish their tasks.

One such robot system is Trixi the Librarian [38], which is able to detect and sort books along a shelf. However, instead of reading the content of the book spine to identify a book, it compares the HSV histogram of the book spine with an image stored in a separate database. This might lead to mix-ups between similar looking books, which could otherwise be mitigated by the inclusion of a text recognition step that would either read the book title for identification in the initial perception step or be utilized later as confirmation for validating the initial action.

This bachelor's thesis explores a generalized approach to object detection and text recognition by first detecting a rectangular area of an object via an object detection pipeline utilizing classical computer vision operations. Subsequently, the object will be transformed into a view more appropriate for text recognition by inverting the detected perspective to create a top-down view of the detected area of the text. Lastly, text recognition is applied to detect the final output of the object.

In this thesis, we evaluate three different open-source OCR programs, namely Tesseract, Easy-OCR and Keras-OCR, and evaluate how accurate they are on the transformed images, the unprocessed images and how the transformation affects the results of the text recognition step. We also evaluate whether current text recognition is robust enough to be applied in a robotics context and how the presented approach can be further optimized to increase accuracy and usability. The test images, on which this processing pipeline was applied were captured by a camera mounted on the head of a PR2 robot and should be an accurate representation for other robots of its kind.

The implementation of the presented approach is available on GitHub[1].

1. https://github.com/HansiMcKlaus/Robot-Reading

2 Fundamentals

This chapter introduces the necessary fundamentals and concepts needed to understand the methodology used and developed in chapter 4.

We start out by introducing common computer vision basics and operations that are necessary building blocks to understand the approach and relevant steps for object detection. Furthermore, we provide an overview of optical character recognition (OCR), including the three dominant implementations for Python, as well as ways to measure the similarity of the recognized and actual text.

2.1 Basics of Computer Vision

This section introduces some of the basics of computer vision and image manipulation, such as the typical structure of an image representation, color spaces and common computer vision operations.

2.1.1 OpenCV

OpenCV [22] is a computer vision library implementing many computer vision algorithms, operations and more. While OpenCV is primarily written in C/C++, there exist bindings for other programming languages such as Python, Java or JavaScript, though the latter only supports a selected subset of the OpenCV functions. There are also packages for other frameworks, namely ROS, which provide an interface for the integration of OpenCV outside of the usual environments, such as Linux or Windows systems.

2.1.2 Grayscale and Color Image

An image is typically represented by a two-dimensional array, with each entry corresponding to a pixel and their color being determined by their value. For grayscale images, a pixel is usually given a value between 0 and 255, possibly scaled down to a range of 0 to 1. This value indicates its intensity, in most cases the brightness, where black corresponds to 0 and white to 255 or 1 respectively. In an RGB color image, a single pixel consists of three such values, making up the red, green and blue color components respectively, creating a large number of possible combinations of hue, saturation and lightness, also known as HSL, which is an alternative representation of the RGB model. [9]

Hue describes the color of a pixel along a color wheel going from red (0°) over green (120°) to blue (240°) and wrapping back to red. Saturation describes how saturated a color looks, with a saturated color appearing more vivid and a low saturation color appearing closer to gray. Lastly, lightness describes its overall brightness and luminance, with a less light color appearing more

dark than a lighter color and a color with either maximum or minimum lightness being white or black. [9]

It is possible to convert an RGB image into a grayscale image by combining the red, green and blue value of a pixel into a single intensity value. The simplest approach would be calculating the arithmetic mean value of the RGB pixel, by forming the sum of the R-, G- and B-value and dividing it by three. However, depending on the context, the color components can also be weighted differently, to create a slightly altered result. For example, the gray value can be computed as described by the following equation 2.1, to take into account how strong the human eye perceives each color component, resulting in a more natural effect. [9]

$$Y = 0.299R + 0.587G + 0.114B \tag{2.1}$$

The specific weights are taken from the OpenCV documentation [22], though

$$Y = 0.2126R + 0.7152G + 0.0722B \tag{2.2}$$

is also widely used, with other approaches to calculating the luminance resulting in different weights [21].

2.1.3 Binarization

Binarization is the process of transforming a grayscale or color image, which would either be reduced to a grayscale image or taken apart into its color channels, into a binary black and white image. This binary black and white image can also be interpreted as being separated into a foreground and background, with the foreground encapsulating all the white pixels, while the background encompasses all the black pixels. This image can then be used as a map or to clearly separate elements in an image. As a map, the binarized image would be used to apply other computer vision operations to only the marked areas. Separating objects is useful for a number of further operations, such as comparing the two different classes with each other or simply isolating certain elements in an image. Normally, this separation is done via a threshold, which would divide the entire content of an image into two classes. If a pixel has a value lower than a specific threshold, it will be set to black, while pixels with a value higher than the threshold will be white. [9]

The threshold can, depending on the use case, be both manually set or automatically calculated via an algorithm such as Otsu's method. Otsu's method calculates a threshold by going through all the possible intensities and dividing all the pixels into two groups, according to whether they are above or below the current intensity threshold. After separating the two groups, the intra-class variance, meaning the weighted sum of all the variances of the two groups, is calculated for each threshold. The final threshold is the intensity value that leads to the the lowest intra-class variance. [23]

2.1.4 Brightness and Contrast

Contrast can be described as a high difference of values between neighbouring pixels, or a measure of distance between entire regions of an image, making segments of the image more distinguishable from each other. Each hue, saturation and brightness value influences the contrast of the image. Two different hues are clearly distinguishable on the basis of being at a different point on the color spectrum, two differently saturated colors are distinguishable by how close their respective RGB values are to another and a difference in brightness is distinguishable by how close the colors are to black and white respectively. In a grayscale image, only the brightness is used. [9]

A high contrast and balanced exposure can be important, as some operations in an object detection pipeline depend on high contrast to more accurately separate individual features from an object, such as the background from the object, as well as features on the object itself. [9]

2.1.5 Brightness and Contrast Correction

The idea behind brightness and contrast correction is to manipulate the image in a way that the different intensity values are spread more evenly across the image, instead of being clumped together. Normally, this would lead to darker areas to become darker, while bright areas become brighter respectively, resulting in an image with higher contrast and better exposure. [9]

This change in intensity can be calculated via the following linear equation,

$$g(x) = \alpha f(x) + \beta \tag{2.3}$$

with $f(x)$ and $g(x)$ being the original and new intensities of pixel x respectively and α and β being the gain and bias to control the brightness and contrast. To automatically adjust the brightness and contrast of an image, α and β need to be chosen to map the lowest intensity value to the minimum intensity of the image and the highest intensity to the maximum. [9]

2.1.6 Convolution and Kernels

A kernel is an additional element for computer vision operations, whose application can give rise to many different effects, depending on its composition. Most kernels have a square shape, but they are theoretically not limited by any shape or size. A convolution is an image operation using a kernel, which can be used to blur or sharpen an image, as well as to perform edge detection and many other types of operations. Specifically, a convolution with a rectangular kernel can be expressed as

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{2.4}$$

with $f(x,y)$ being the value of a pixel in the original image f at coordinates (x,y). $g(x,y)$ is defined accordingly for the resulting image g and w describes the kernel matrix with a and b denoting the width and height of the kernel. In practice, the kernel is shifting along every pixel

in the original image and the pixel in the resulting image is then calculated by the sum of the surrounding pixels times the respective corresponding value in the kernel. When applying the kernel to pixels close to the edges, the kernel contains pixels that are outside the image, for which there are no determined values. Because of this, the image is typically padded before applying a convolution. [9]

2.1.7 Blur

The Blur operation makes an image appear less sharp and thus reduces the level of detail, preserving only the shape of larger areas, but also making the transition between areas, i.e. edges, softer and less clear. The blur operation uses a kernel to normalize the value of a pixel through the values of all the pixels around it. The size of the kernel dictates the strength of the respective blur effect. [9]

In a simple box blur, the kernel is filled homogeneously, meaning every entry in the kernel has the same value of 1 divided by the number of entries. In a 3×3 kernel, this means that every entry has the value of $\frac{1}{9}$. In a Gaussian blur, the kernel approximates a Gaussian distribution to fill its kernel, resulting in the middle of the kernel, and by proxy the pixel in the original image, to have a larger influence on the resulting image, than the pixels around it. This causes a smoother effect than a box blur, which may sometimes lead to small rectangular artifacts. [8, 9]

2.1.8 Matrix Transformation

Matrix transformation in computer vision describes an operation to transform an image in space. In a 2-dimensional context, this includes either a single operation or a combination of translation, rotation, shear and scale implemented as a multiplication of the image with a matrix describing the transformation. [2]

A subset of such transformations are perspective transformations, which change the apparent perspective of an image and can be used to either distort or realign an image. An example of this can be found in scanner applications that transform a flat object, i.e. a piece of paper, into a top-down view, as if seen from straight above, even if the original image was viewed from an angle, with the top part of the paper appearing farther away than the bottom and the sides at an angle. [39]

For perspective transformation in a 2-dimensional space, there must exist a transformation matrix, that can map an area of the original image to another area in a new image, with each area being delimited by a series of four points. This matrix can be calculated by utilizing the four specific points from the original and the new image. If the original image and the transformation matrix are multiplied, everything inside the four points in the original image will be transformed to fit inside the four points of the new image. [2]

$$g(x,y) = M \cdot f(x,y) \tag{2.5}$$

where f and g denote the old and new image respectively, x and y denote coordinates on the respective axes, and M is the transformation matrix.

This idea can be extended to 3-dimensional objects, for instance a box, in a 2-dimensional image, with each face of the object acting as a single 2-dimensional plane. By isolating the plane of interest and detecting the face's corner points, a flat area on a 3-dimensional object can be transformed into a flat 2-dimensional top-down view. [2]

2.2 Object Detection

Object detection is the task in computer vision to detect, locate and identify an object in an image. There are several different approaches to detecting objects, such as classic computer vision techniques, or more modern approaches such as those using a neural network trained to identify certain objects. There are several steps included to achieve this task, such as various types of preprocessing of the initial image, the actual detection algorithm which classifies the different features of an image, and potential post-processing of the resulting data. Object detection can be used to recognize specific features of an object or classify it into predetermined categories. [42]

2.2.1 Morphological Operations

Morphological operations in computer vision describe operations using a structuring element, i.e. a kernel, to manipulate the shape and form of objects inside an image. The four most common morphological operations include dilation, erosion, opening and closing. The former two grow and shrink a selection of pixels, for example two groups divided into foreground and background, while the latter two, as the name implies, opens and close holes and gaps in and between pixel groups. These operations are usually applied to binary images, but can also be applied to grayscale images. [9]

The morphological operations work similarly to a convolution with the structuring element shifting along every pixel of the image and manipulating the output pixel at the same coordinates depending on the specific operation. Dilation works by setting the value of the output pixel to white, if the structuring element overlaps with at least one foreground element of the input image, thus expanding the foreground selection. Erosion operates in the same way, but shrinking the area of the foreground instead, by setting the output pixel to black, if the structuring element contains at least one background pixel. [9]

Opening and closing use the successive application of the dilation and erosion operation to achieve their effects. Opening first uses an erosion, followed by a dilation to remove any area of foreground between two background areas smaller than the structuring element, as the erosion first shrinks the foreground area to connect the two background areas and thus creating a larger overall background area that can not be fully dilated back into the foreground. The closing operation is done by first applying dilation, then erosion, with any background area inside a foreground area smaller than the structuring element not being able to erode back, as it was completely dilated into the foreground in the first step. It should be noted that opening and closing do not change the size of the foreground and background beyond the opened and closed areas, as the combination of dilation and erosion keeps all the unaffected areas the same due to the second operation reverting the effects of the first one. Figure 2.1 shows the effect of dilation, erosion, opening and closing on a black and white image with a 3×3 square structuring element. [9]

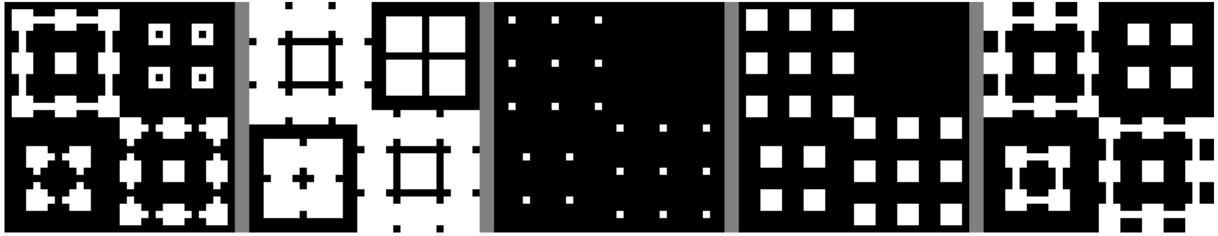

Figure 2.1: From left to right: Original, dilation, erosion, opening and closing

The morphological operations can also be applied to a grayscale and, by extension, an RGB image, with grayscale dilation resulting in the pixel being set to the brightest value contained in the structuring element and with grayscale erosion to the darkest value respectively. Thus, the opening and closing operation does not specifically result in the opening of connected areas or closing of gaps, but rather effectively smoothing out uniform areas, resulting in an image with less detail, but, in contrast to blurring, preserving a sharper transition for edges between areas. [9]

2.2.2 Edge Detection

An edge can be described as an abrupt change of intensity in the image. Edge detection can be used to isolate such areas of high contrast for further processing. Similar to blurring, edge detection uses convolution with a specific kernel to achieve this effect. There are different kernels for specific operators, such as Roberts, Prewitt and Sobbel, which create an intensity gradient. To more accurately find edges and not just areas of high contrast introduced through artifacts, the image is typically denoised first, i.e. through blurring, to reduce the amount of wrongly identified edges, which could impact further operations down the line. [9]

One such edge algorithm is called the Canny edge detector [4]. The Canny edge detector consists of several steps:

1. reduce the image to grayscale
2. apply a Gaussian filter
3. determine the intensity gradient of the image
4. use gradient thresholding and double thresholding to thin out the edges and remove noise
5. filter edges via hysteresis

First of all, the image is reduced to grayscale. It is possible to use a color image, in which case each channel would act in the same way as a single grayscale image with the respective red, green and blue channel values acting as the intensity. Afterwards, a Gaussian blur is applied to remove noise [4].

In the next step, the intensity gradients are determined, as shown by equation 2.6 and 2.7. The intensity gradient is based on the vertical and horizontal intensity detected by applying an edge detector such as Sobbel, with the gradient strength G being the hypotenuse of the vertical and horizontal values G_y and G_x. The gradient angle Θ is computed by using the atan2, the arctangent utilizing two values, of the same vertical and horizontal value. The gradient angle is

then rounded to the the nearest angle of either $0°, 45°, 90°$ or $135°$, with angles between $157.5°$ and $180°$ wrapping around and being rounded to $0°$

$$G = \sqrt{{G_x}^2 + {G_y}^2} \tag{2.6}$$

$$\Theta = \text{atan2}(G_y, G_x), \quad \text{rounded to the nearest value of } 0°, 45°, 90° \text{or } 135° \tag{2.7}$$

Next, the algorithm thins out the edges by applying a lower bound cut-off suppression, which compares each pixel with the pixels in both the direction of its angle, as well as the opposite direction, i.e. if the pixel has the angle of 45°, it is compared to its lower left and upper right neighbour. The aim of this operation is to reduce the edge thickness to one pixel to better determine the correct spatial location of the edge. If the current pixel has the strongest gradient compared to its two neighbors, it will be preserved, otherwise discarded. For example, if the edge currently has a thickness of three pixels, the algorithm looks along those three pixels and discards the two pixels with the lowest gradient strength, leaving an edge with a thickness of one pixel. [4]

Afterwards, double thresholding is applied by determining a higher and a lower threshold. If a pixel has a gradient value below the lower threshold, it will be discarded, as it will most likely be an edge pixel introduced by noise. Meanwhile, whether the gradient value is between the lower and higher threshold or above the higher threshold will determine if the edge pixel will be marked as a strong or weak edge pixel. [4]

Lastly, the algorithm determines whether a weak edge pixel should be discarded or not via hysteresis. It works on the assumption that a weak edge pixel connected to a strong edge pixel is most likely indicative of a real edge in the image, while a weak edge pixel with no connection to another strong edge pixel could either be noise or not a real edge. Whether a weak edge pixel is connected to a strong edge pixel is determined by looking at all the eight directly neighbouring pixels. If any of those pixels is a strong edge pixel, the weak edge pixel will also be considered a strong edge pixel. [4]

2.2.3 Contour Detection

A contour can be described as a line, more specifically a curve, connecting all the continuous points along any kind of boundary with the same intensity value. This differentiates a contour from an edge, as an edge only indicates a local change in intensity, while a contour indicates a closed region. As such, contours can be used to determine the shape and position of an object inside an image. Normally, contour detection is applied on a binary image, obtained either done via thresholding or edge detection, as the boundaries are then more clearly defined, which increases accuracy. [9]

Contours can also be filtered by their properties, such as the total area they encompass, which can be used to automatically determine the largest detected shape. Since a contour contains every point along a boundary, the contour can be simplified to a polyline by reducing all the points along a straight line to its start and end point, for example by using the Ramer–Douglas–Peucker algorithm [7], with a polyline being a sequence of points connected by straight lines forming a curve. [9]

The Ramer–Douglas–Peucker algorithm is a recursive algorithm used to simplify a polyline by reducing the amount of points the polyline consists of and thus creating a similar polyline with fewer points. The algorithm works by setting a starting and ending point, beginning with the first and last point along the line, marking them and selecting the point farthest away from the line connecting these two points. If the selected point is farther away than a predefined distance ε, the point is kept and marked, otherwise discarded. The algorithm then calls itself alternating with the original starting and end point and the lastly marked points, until every point along the line is either marked and kept or discarded. [7]

2.2.4 Intersection over Union

Intersection over Union, also known as Jaccard index, is a measurement for determining how similar two sets are, as described in equation 2.8.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.8}$$

with A and B being the respective two sets and J the Jaccard index function. The similarity will be one if the two sets are identical, while it will be zero if the two sets have no intersection at all. [28]

In object detection, this measurement can be used to evaluate how accurate the object detection is by comparing the prediction of the object detection with the ground truth of the original object. For example, the two bounding boxes of the detection and the ground truth, indicating the position and area of the object, could be used as the respective two sets. This would lead to a low accuracy, if the predicted bounding box barely intersects with the ground truth, while the prediction becomes more accurate, the more the bounding boxes overlap. [28]

2.2.5 Object Detection utilizing Neural Networks

While object detection can be achieved using classical computer vision operations, modern approaches are utilizing neural networks to detect and classify objects in an image by training these networks on large data sets to produce prediction models for the respective classes.

YOLO

YOLO (You Only Look Once) uses a convolutional neural network to predict class probabilities, as well as bounding box coordinates for objects in images. The model was trained on the ImageNet 1000-class competition data set and produces a top-5 accuracy of 88%, meaning the correct class of the object is in the top 5 most likely contenders, and a mean average precision (mAP) of 63.4%. YOLO can also be used to detect objects in a specific context by training it on a custom data set prepared for the task. YOLO has a high performance and can run at 45 frames per second, while the less accurate version of Fast YOLO can process 155 frames a second, making it applicable in real-time contexts. [26]

Mask R-CNN

Mask R-CNN [11] is an extension of Faster R-CNN [27], which normally uses a convolutional neural network to detect objects and returns class probability and bounding box coordinates. Mask R-CNN extends the latter framework by also semantically segmenting the object inside the bounding box and thus creating a mask for each object of a different class for an overall more precise localisation. When tested on the COCO image data set [17], Mask R-CNN achieved an average precision of 35.7% for its segmentation mask, while processing at five frames per second.

2.3 Optical Character and Text Recognition

In computer science, optical character recognition (OCR) is the process of extracting text from an image source to convert it into text that can be processed by a computer, with examples for real life applications being the automatic conversion of a book into a digital copy or the detection of specific text in an image.

Optical character recognition and, by extension, text recognition can be achieved both via classical computer vision approaches and operations such as pattern and template matching for characters [30], or through more modern approaches, in particular the utilization of neural networks trained to detect text [37].

2.3.1 OCR Programs

This section introduces the OCR programs that will be later used to apply text recognition for the implementation in chapter 4.

Tesseract

Tesseract [36] is an OCR engine with its current version utilizing a long short-term memory (LSTM) neural network [12] to detect lines of text. Tesseract can detect text in over 100 different languages and scripts, detect text orientation and return a confidence value for the detected text. Additionally, Tesseract can be trained on a custom data set. Bindings for Tesseract are available in many programming languages, including `C++`, `JavaScript` and `Python`.

EasyOCR

EasyOCR [1] is developed by Jaided AI and uses a CRAFT algorithm [3] for the text detection and utilizes a CRNN architecture [32] for its recognition model. It supports over 80 different languages and all popular writing scripts. EasyOCR also provides the option to use a custom-made model for both text detection and text recognition.

Keras-OCR

Keras-OCR [20] uses the official Pytorch implementation of the CRAFT algorithm [3] and the Keras Convolutional Recurrent Neural Network [41] for scene text recognition. It also comes with an end-to-end training pipeline for custom models.

2.3.2 String Matching and Text Similarity

String Matching is the task of finding a string of text in another string of text. One subtask of this is the approximate string matching, which, instead of finding a string that matches exactly, is looking for a string similar to the input. How similar, or rather how dissimilar, two strings are is denoted by the edit distance. Some edit distance measures are calculated using operations such as insertion and deletion of a character in the string, the substitution of a character in the string with another character and the transposition of two characters in a string. The edit distance, and by extension, the specific text similarity can be computed in several different ways. [14]

Hamming Distance

Under the Hamming distance [10], edits are defined as substitutions of characters and as such, the edit distance is the sum of characters not equal to the character in the original string at the same position. For example, the strings `approximate` and `epporximata` have a distance of 4 and a similarity of $1 - \frac{4}{11} \approx 0.63$.

Jaro Similarity

The Jaro similarity, only allows transposition and calculates text similarity as seen in the following equation

$$\mathrm{sim}_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3} \cdot \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) & \text{otherwise} \end{cases} \tag{2.9}$$

with sim_j being the text similarity, m the amount of matching characters, t the number of transpositions and $|s_1|$ and $|s_2|$ the length of the respective strings. A character is considered matching, if the same character is not further away than $\left\lfloor \frac{\max(|s_1|,|s_2|)}{2} \right\rfloor$ characters. The number of transpositions is the number of non-matching characters divided by two. The text similarity is 0 if no character matches. [40]

For example, the strings `approximate` and `epporximata` have a distance of $\mathrm{sim}_j = \frac{1}{3} \cdot \left(\frac{9}{11} + \frac{9}{11} + \frac{9-1}{9} \right) \approx 0.84$, as there are nine matching characters and one transposition of the characters `a` and `e` from the beginning and end of the string.

Gestalt Pattern Matching

The gestalt pattern matching [25], also known as Ratcliff/Obershelp pattern recognition, uses no edit operations and instead calculates the text similarity based on the longest common substrings according to the following equation

$$D_{ro} = \frac{2K_m}{|S_1| + |S_2|} \tag{2.10}$$

with D_{ro} being the text similarity, K_m the sum of the characters of the common substrings and $|S_1|$ and $|S_2|$ being the length of the respective strings. For example, the strings `approximate` and `epporximata` have a distance of $D_{ro} = \frac{2\cdot(2+5)}{11+11} \approx 0.63$.

Levenshtein Distance

The Levenshtein distance [15] allows for the deletion, insertion and substitution of characters and defines the edit distance as the minimum number of operations needed to transform one string into the other. The Levenshtein distance can be computed by recursively splitting the two strings into substrings and stopping when the two substrings are identical, with the final distance being the least amount of times the strings had to be split.

For example, the strings `approximate` and `epporximata` have a Levenshtein distance of 4, as the minimum steps required to transform `epporximata` into `approximate` is 4. Since the two strings have the same length, this is achieved by substituting the four characters that do not match at a given position and replacing them with the correct one.

2.4 ROS and PR2

The Robot Operating System (ROS) is an open-source framework and provides a standard for robotic applications that can be used on any robot. ROS is not an operating system, but still provides basic and core functionalities, in particular the abstraction of hardware, low-level device control and communication between individual components, upon which higher level operations can be built upon. It also provides access to libraries, meaning not every aspect of the robot operation, such as controlling and motion planning, needs to be programmed from scratch and can be used across different robot platforms. [29]

The PR2 is a robot platform developed by Willow Garage. The robot consist of a moving platform, a height-adjustable spine, two freely movable arms that can be equipped with special hands, for example a gripper or Shadow Hands, and a head, equipped with a wide- and narrow-angle camera, capable of rotating and tilting. [24]

3 Related Work

In this chapter, we summarize the work of research related to this bachelor's thesis and its topic. First, we introduce the paper Trixi the Librarian, laying some necessary foundations and introducing the capabilities and limitations of the system previously developed at the TAMS group, which gives the background and context for this thesis' groundwork. Afterwards, we explore some of the current state of the art approaches to object detection, text detection and optical character recognition.

3.1 Trixi the Librarian

Trixi the Librarian [38] is a system developed at the TAMS group at the University of Hamburg. It utilizes the PR2 robot through the ROS-framework and further makes use of the MoveIt framework [5] and BioIK solver [31] to detect the spines of books and sort them along a shelf. For this task specifically, the PR2 was equipped with a Shadow Hand, which emulates the movement and functionality of a human hand and an additional two-finger gripper [38]. For vision, the PR2 used an Azure Kinect camera on top of the robot, which not only returns an RGB color image, but also depth information (RGBD). While the PR2 is capable of movement along the floor, for this task, it was placed statically in front of the shelf, only changing the elevation of the torso to switch between the two different sections of the shelf when picking up and placing the books.

3.1.1 Task, Approaches and Results

Trixi the Librarian was developed to test the feasibility of utilizing a service robot to automate part of the work of a human librarian, in this case sorting books on a shelf [38]. The specific shelf for the test setup consists of two separate sections, with the top section of the shelf being used as the storage space for all the books, while the lower section is utilized to place the sorted books one after the other. The shelf was also marked with an AprilTag. The presented approach splits the task into two parts, where the first part is concerned with the detection and recognition of the book spines, followed by the manipulation of the books by placing them ordered on the lower section.

Firstly, the images provided by the camera are preprocessed by aligning the book spines to the image axis using a perspective transformation for each shelf [38]. The AprilTag is used to determine the perspective transformation, resulting in an easier identification of the books. The spines were detected using the neural network object detection model YOLO. The default model for detecting books was not utilized, due to problems detecting the spine, as well as multiple spines standing close together being interpreted as a single book spine. Instead, the authors created their own data set to fine-tune the model for the specific task at hand. While there were attempts to use text recognition and SIFT-Matching, these were ultimately rated as insufficient, with the final classification method using an HSV-histogram without the value

component, comparing the identified book spine with the spines inside a pre-arranged database of all books.

The system starts the perception process, detecting and recognizing the book spines, creating a world state representation, under the assumption, that the environment does not change and the action of moving the book is deterministic [38]. The perception process is then stopped. The OMPL planner is used to calculate the needed motion to extract the book from the upper section of the shelf and then place it on the lower section. The book is extracted by tilting it to the front using the Shadow Hand and pulling it out of the shelf via the gripper hand. Before placing the book on the lower section, the book is placed in front of the camera focus area and rotated to show the front and back of the book for potential future perception tasks. Lastly, the book is then placed on the lower section of the shelf by leaning it against the left-side wall of the shelf and then pushing it further left, until it stands upright.

In the final demo, the system was able to sort three books, by detecting the spines and identifying the respective books, extracting them from the top section of the shelf and successfully placing them on the lower section in the sorted order [38].

3.1.2 Possible Improvements

The paper acknowledges some limitations of its approach and describes methods for the system to become more robust. For the perception process, it is explained how a larger data set would prevent overfitting and allow the system to generalize better, while more fine-tuned metrics for the spine identification would improve the matching of the books [38].

What is not described is how the shortcomings of the text recognition as a feature could be improved, either by seeking out other text recognition models, or by incorporating the text recognition later into the process. After retrieving the book from the top section of the shelf, the book is already brought into the view of the camera, which would greatly improve the readability of the book title compared to the view into the shelf after the perspective transformation. This additional step would also solve the problem of the misidentification of a book, if there are several similar looking books on the shelf or a book series whose only identification feature would be the volume number.

3.2 State of the Art

This section focuses on the current state of the art concerning the topic of object detection and text recognition, especially different approaches on how to approach the challenge of detecting and recognizing text in a scene recognition environment.

As mentioned in section 2.2.5, there already exist promising approaches for accurate object detection in the form of YOLO [26] and Mask R-CNN [11]. The first one frames the detection of an object as a regression problem and eliminates all the regions in an image with a low probability of the object being inside a certain area using only a single pass of the input image. On the other hand Mask R-CNN is built on Faster R-CNN [27], which creates region proposals for the object in the image, a feature map for each Region of Interest (ROI) and performs the object classification using a fully connected layer. It is then extended by a segmentation of the object by utilizing an FCN [6] on each Region of Interest of the R-CNN to create a mask of the object. Similarly, Alexander Kirillov et al [13] introduced a new model and large data set specifically for the task of image segmentation that can successfully segment most objects in a scene. Their work is currently available as an interactive demo online[19].

For text detection, Shijian Lu and Chew Lim Tan [18] propose an approach using perspective invariants of characters to detect text on flat and curved surfaces under different perspectives and transformations, such as rotation, skew and scale. These invariants include the character ascender and descender, the number of vertical character runs, meaning how many times a vertical line intersects with the character and the water reservoir of the character. The 26 lowercase characters of the Latin alphabet are then structured under a classification and regression tree (CART) and being separated based on the three character invariants for classification.

Baoguang Shi et al [33] are using a Spatial Transformer Network (STN) to rectifiy irregular and hard to read text and subsequently apply a Sequence Recognition Network (SRN) for text reconition. The first step includes transforming irregular text, characterized by being distorted through perspective or rotation, into a regular text with the characters being aligned horizontally and written left to right by predicting a set of fiducial points along the characters and calculating the rectified image via a grid generator. It then uses an SRN to detect the text by utilizing an RCNN for classification.

The approach of Minghui Liao et al [16] is based on predicting text in a two-dimensional space, instead of a one-dimensional sequence and thus performing text recognition directly on an image with the text being in an irregular state, such as being rounded along a curve. This is achieved by devising a Character Attention Fully Convolutional Network (CA-FCN), which first predicts the location of the characters and subsequently recognizes them to form the detected text.

4 Implementation

In this chapter, we explain the process for implementing the text recognition for objects on recorded images, starting with the setup and environment. Next, we present the overall structure and steps of the developed image processing pipeline and the text recognition using different OCR programs.

4.1 Environment and Setup

The presented approach only requires a standard RGB color image as its input, which makes it compatible with both pre-recorded images, as well as a live-feed from a normal camera. The code itself is written in Python 3.9.13 and using the OpenCV Python-wrapper opencv-python version 4.7.0.72. The program was tested on both a Windows 10 system, as well as the Manjaro Linux distribution.

4.2 Overview of the Image Processing Pipeline

This section focuses on the overall image processing pipeline developed for detecting and processing text on an object in an image. The pipeline can be roughly divided into two parts, with the first part focusing on preparing the image for text recognition by detecting the object and transforming the face containing the text into a top-down view. The second part specifically focuses on text detection by utilizing the OCR programs introduced in section 2.3.1. Figure 4.2 gives an overview over the entire processing pipeline.

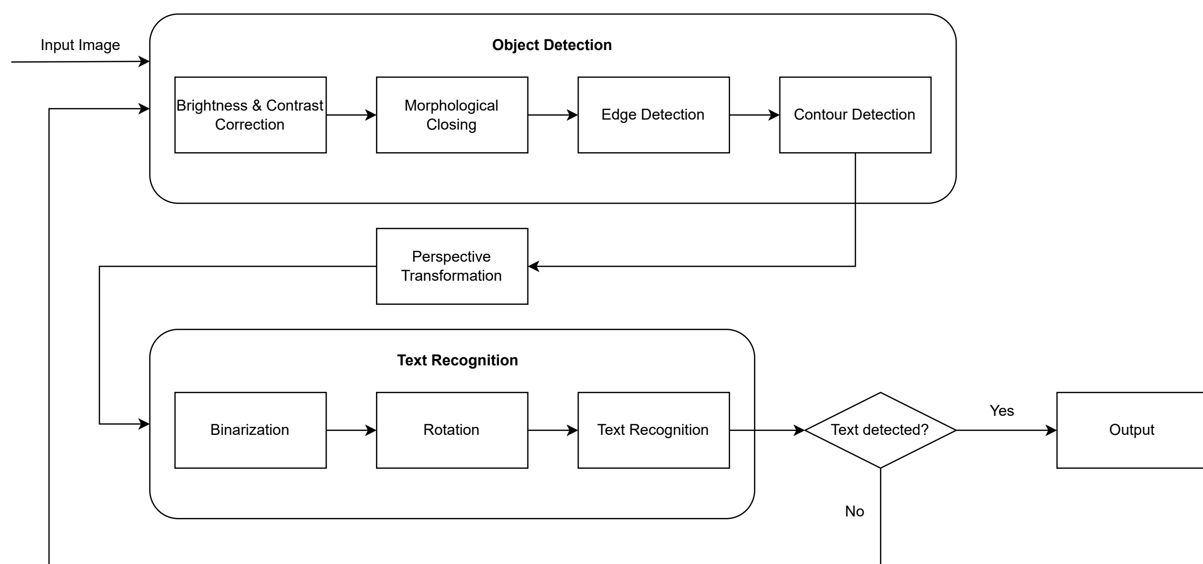

Figure 4.1: Diagram of the processing pipeline

Depending on whether any text is recognized in the latter step, the image processing step will be repeated using different parameters for the preprocessing and edge detection, to increase the chance of correctly detecting the object and respective text over several passes. The change in parameters includes the size for the structuring element for the closing operation, as well as the blur radius and the thresholding value for the canny edge detector, starting from a small kernel size and a high thresholding value and increasing the size and and lowering the threshold after each pass. The amount of total passes is limited to five.

The pipeline is built on several core assumption about both the object and the image as a whole. The object needs to face the camera with at least one face of the object resembling a rectangle for the perspective transformation to work correctly. As such, the pipeline will be able to detect flat objects such as a sheet of paper, or box-shaped objects, for instance books and most common packaging, but will likely fail for rounded surfaces, such as bottles. The background should also be mostly homogeneous, as patterns and other objects in the background could lead to the primary object being wrongly detected. Lastly, the image should have enough pixel information to make text recognition possible. If the text is out of focus, smeared, obscured or simply too small, the text will not be correctly recognized.

4.3 Preprocessing, Object Detection and Perspective Transformation

To guarantee the best possible result for the text recognition, the text should be read from a top-down view. This can be achieved by applying a perspective transformation on the original image, which would reverse the effect perspective has on the image introduced by the camera looking at the object at an angle. To calculate the matrix for the perspective transformation, the program needs to extract the four corner points, which define the area in which the text we want to detect is located. These corner points can be computed by detecting the object inside the image via a combination of edge and contour detection. Before this detection can be attempted, the image should be preprocessed to improve the results of all the subsequent steps.

As such, we first manipulate the input image by automatically adjusting the brightness and contrast for a clearer distinction between object and background. Next, we apply a morphological closing operation to eliminate unnecessary details, which suppresses smaller areas to be detected. Next, edge detection via the Canny edge detector is applied, which builds the basis upon which we apply the contour detection that yields the detected object. Finally, the output is approximated to a rectangle and the image undergoes a perspective transformation into a top-down view, at which point the resulting image is prepared for the text recognition.

4.3.1 Automatic Brightness and Contrast Correction

To improve the results of the subsequent edge and contour detection, the brightness and contrast for the input image needs to be corrected, as this will yield more detectable edges along the object and background, as well as improve the readability of any text on the object.

To calculate the new intensities, equation 2.3 is used to change the brightness and contrast of the input image. One can derive the α and β value by solving the equations 4.1 and 4.2 for mapping the old intensity range of $[I_{min}, I_{max}]$ to the new range of $[0, 255]$.

$$\alpha = \frac{255}{I_{max} - I_{min}} \tag{4.1}$$

$$\beta = -I_{min} \cdot \alpha \tag{4.2}$$

However, as a single black and white pixel in the original image would result in no change overall, one can clip the intensities in the input image beforehand to initially only include a certain range of intensities and ignore all the pixels outside this range, thus increasing the α and β value, which will lead to a stronger effect of the correction. The threshold for clipping the lowest and highest intensities can be set arbitrarily, or calculated by excluding a set percentage of pixels in the image. In the latter case, the upper and lower clip threshold would be determined by calculating the histogram of the image and starting at the highest and lowest intensity. From there on, one calculates the sum of all the pixels outside the currently defined range. This process is repeated by decreasing the range until a set percentage of pixels in the image has been reached, with the last intensity thresholds denominating the minimum and maximum intensity value for which the image should be clipped. After some testing, we have determined that a value of two percent yields good results.

Figure 4.2 shows how a more even distribution of the intensities creates a stronger contrast and better exposure in the image.

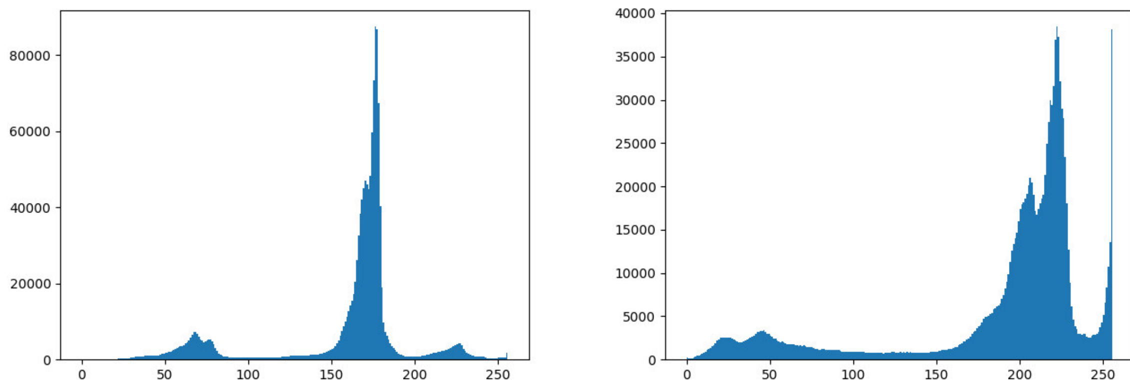

Figure 4.2: Before (left) and after (right) applying automatic brightness and contrast correction. The top row shows the image itself and a histogram is included below.

4.3.2 Morphological Closing

Morphological closing, as described in section 2.2, is used to reduce the amount of detail in an area. In this context, the morphological closing is utilized to reduce the detection of possible edges inside an object, that do not matter in the detection of the object itself. It also decreases the amount of noise that was introduced by the brightness and contrast correction.

Figure 4.3 shows how the closing removes artefacts, such as smaller text, while leaving the overall shape of the object intact.

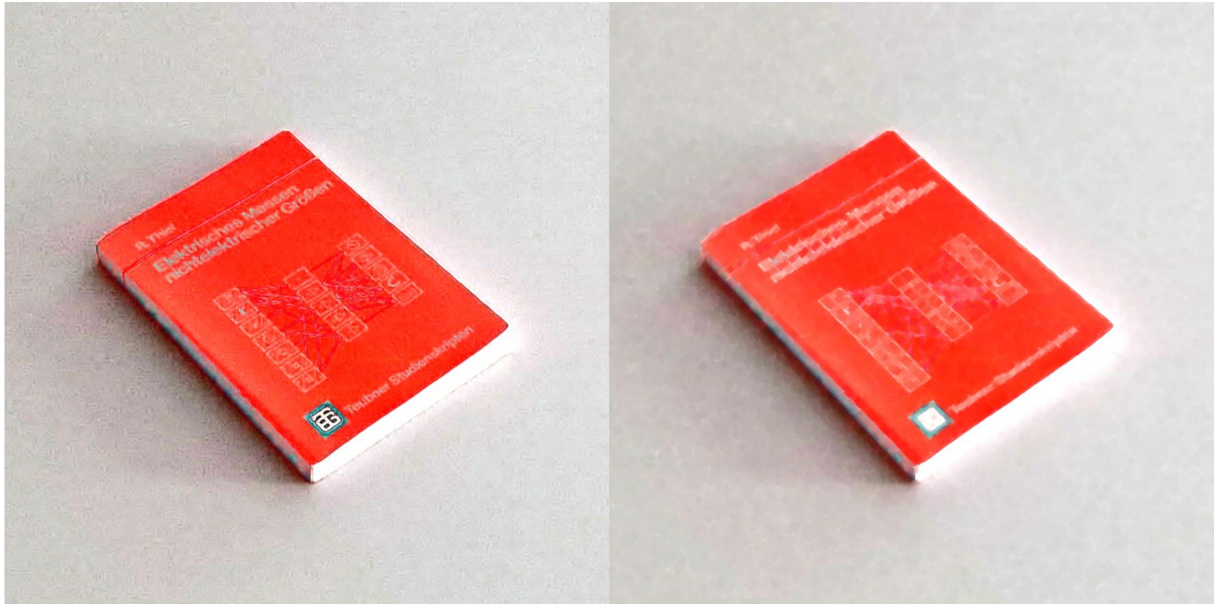

Figure 4.3: Before (left) and after (right) morphological closing

4.3.3 Edge and Contour Detection

After the image is preprocessed, edge detection is applied to locate all the remaining edges left in the image. This is done under the assumption that, while large areas in the image were already simplified, the edges making up the contour of the object should still be noticeable to differentiate between object and background.

The actual edge detection on the preprocessed image is done using the Canny edge detector. Due to unclear edges or similar colors between object and background, the resulting binary image is closed to connect small gaps between the edges. The result of the edge detection can be seen in figure 4.4.

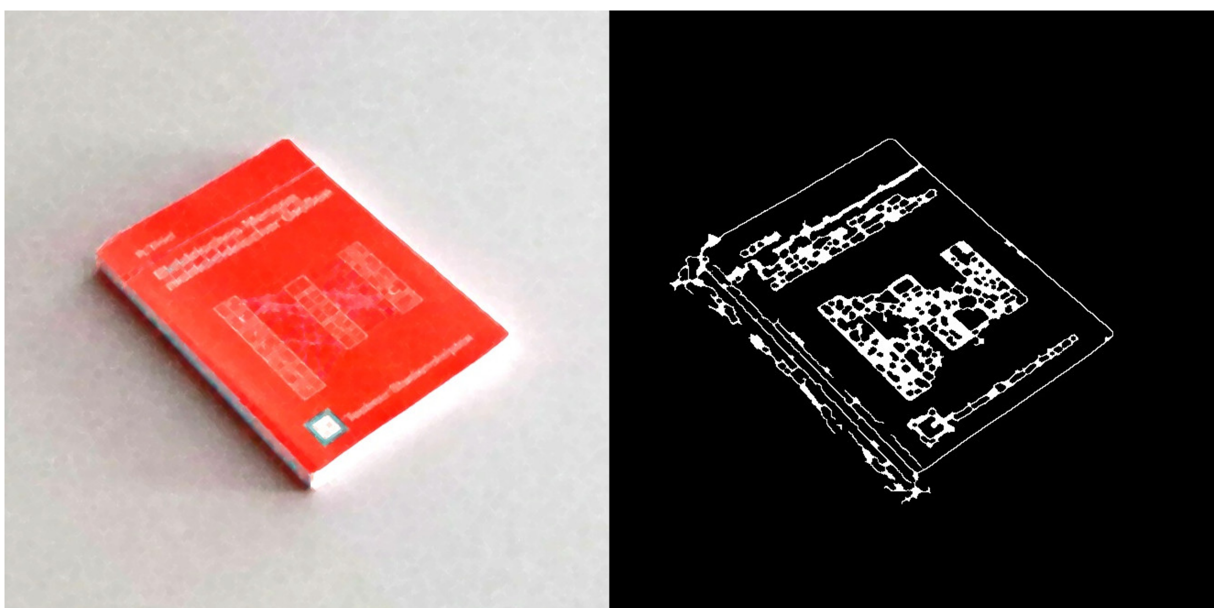

Figure 4.4: Input (left) and output (right) of edge detection

Afterwards, the closed edge image is used to detect the contours of the object. Specifically, this is done using the OpenCV method `findContours`, which returns all contours in an image. For the purpose of the task at hand, we need all the points along the contour and without a hierarchy, as the subsequent step will only take the largest contour into consideration and as such no relationships between contours need to be noted.

```
1 contours, hierarchy = cv2.findContours(closed_canny_output,
     cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
```

Listing 4.1: Contour Detection after Canny edge detection using OpenCV's findContours()

This operation will retrieve all the contours found among the detected edges using border following [35]. In the next step, we locate the largest contour in the shape of a rectangle from these contours. For this, the contours are sorted according to their area and then approximated using the Ramer–Douglas–Peucker algorithm, by interpreting them as a polyline.

In the case of a rectangular boundary, the polyline is defined with all the points defining the curve aligning with the contour and starting from one point, eventually closing in on itself at the end. Using the Ramer–Douglas–Peucker algorithm, this curve will be continuously reduced to only include the four corner points of the contour, which can subsequently be interpreted as the corners of the biggest rectangular face of the object.

The maximum distance between the original curve and the simplified curve is denoted by ε, and acts as a parameter for accuracy. Because of this, ε is set to a percentage of the total length of the contour.

```
1 for contour in object:
2     epsilon = 0.02 * cv2.arcLength(contour, True)
3     corners = cv2.approxPolyDP(contour, epsilon, True)
4     if len(corners) == 4:
5         break
```

Listing 4.2: Approximation of the contour to a rectangle

Overall, the program loops through a list of all the contours found by the contour detection process, sorted by area size, and returns the four corner points for the largest contour that could be first approximated to a polyline of four points. This process should not be done on only the largest contour, but on the next largest ones too, in our case five, to mitigate the case of a wrongly detected contour with a large area that can't be approximated to a rectangle, thus potentially missing out on the object detection altogether.

For the last step, the corner points are sorted to correctly map them to the top-left, top-right, bottom-left and bottom-right corner of the rectangle. This can be easily achieved by first dividing the four corners into two groups, left and right, and then dividing these two groups again depending on whether a corner is at the top or the bottom. The groups can be formed by sorting the corners along their x-coordinates, which sorts the two corners with the smallest x-coordinate into the left group and the two corners with the biggest x-coordinate into the right group. Then, in both the left and right group, the two y-coordinates of the corners are compared with each other and the corner with the smaller y-coordinate is the top corner, while the larger y-coordinate is the bottom corner. After sorting, all four corners are mapped to their representative position.

Figure 4.5 shows the individual steps, going from the edge detection to the contour detection and lastly the final positions of the corners used for the perspective transformation in the next step.

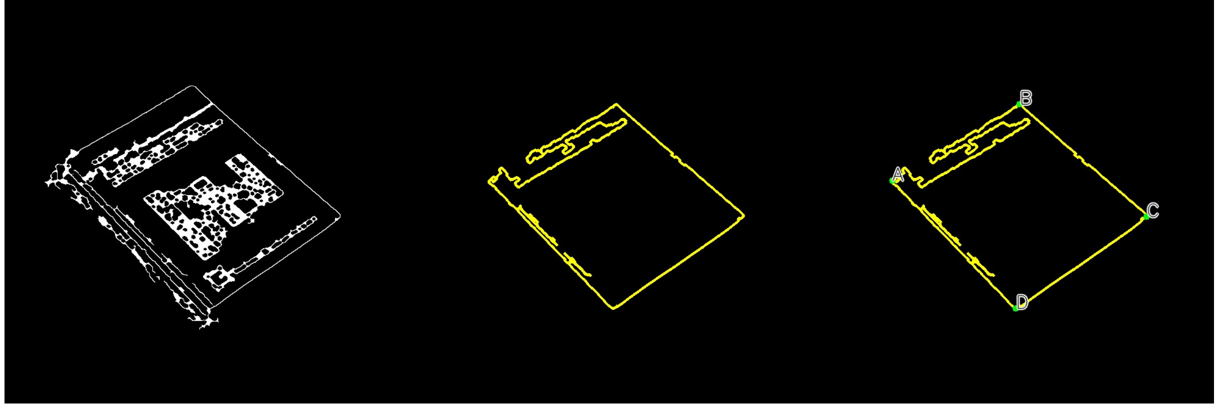

Figure 4.5: Before and after contour and corner detection

Mapping the corners correctly is important, because the calculation of the matrix for the perspective transformation assumes a correct correlation between the corner points of the original image and the transformed image. On the other hand, a wrong mapping of the corners can result in the transformed image being mirrored, rotated or twisted and thus making the text unrecognizable.

4.3.4 Perspective Transformation

Using the four corner points found through the contour detection, the original input image can be transformed to create a top-down view of the detected area. Since the original dimensions are unknown, the new dimensions of the transformed output image have to be calculated via the length of the sides of the detected rectangle, specifically the longer sides. The height is calculated by taking the longer of the distance between the top-left and and bottom-left and the top-right and bottom-right corner. Similarly, the width is calculated by comparing the distance between the top-left and top-right corner and the bottom-left and bottom-right corner and using the longer distance.

The longer of the respective two heights and widths are chosen to make sure that as much information as possible from the original image is retained after transformation, as no part of the image will need to be scaled down. This method does not retain the original aspect ratio of the detected area, which might lead to the resulting image being distorted, depending on the original angle of the object, with a sharper angle leading to a stronger distortion.

After determining the dimension of the output image, the homography, meaning the matrix for the perspective transformation, can be calculated, using the four corners from the contour detection and the corners from the output image. The corners from the output image would be denoted as $(0,0)$, $(0, width)$, $(height, 0)$, and $(height, width)$ for the top-left, top-right, bottom-left and bottom-right corner respectively. Equation 4.3 describes how the old points of the input image would be transformed into the new top-down view image.

$$M \cdot \begin{bmatrix} TL_{old} \\ TR_{old} \\ BL_{old} \\ BR_{old} \end{bmatrix} = \begin{bmatrix} TL_{new} \\ TR_{new} \\ BL_{new} \\ BR_{new} \end{bmatrix} \tag{4.3}$$

This equation can be transformed into a series of linear equations and solved for M. After the transformation matrix M has been determined, the original image can be transformed into the top-down view. We use the corresponding OpenCV functions for this step.

```
1  M = cv2.getPerspectiveTransform(np.float32(input_corners),
       np.float32(destination_corners))
2  top_down_image = cv2.warpPerspective(original, M, (newWidth,
       newHeight), flags=cv2.INTER_LINEAR)
```

Listing 4.3: Calculating homography and transforming the image into a top down view

Figure 4.6 shows an example of such a perspective transformation. As can be seen, the corner points of the image have been successfully transformed to the new corner positions and all the points between the original corners have been interpolated in between them to form the top-down view.

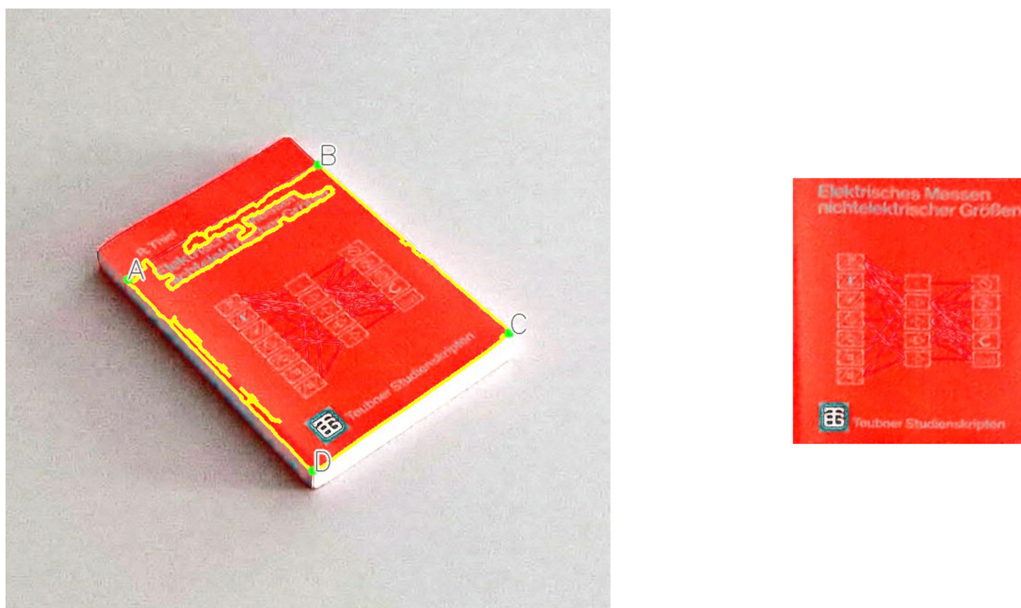

Figure 4.6: Image (left) is transformed to the top-down perspective (right)

4.4 Text Recognition

In this section, we look at how text recognition is applied on the transformed input image, by further separating the text from the background using binarization, handling the orientation of the text and lastly using OCR to detect the text of the object.

4.4.1 Binarization

As an additional processing step, the newly transformed image can be binarized to further separate the text from the background, as the text tends to be written in a color with stark contrast to its surroundings. We first apply a Gaussian blur to soften the edges of the letters and automatically calculate the threshold for the binarization using Otsu's method. Using the determined threshold, we binarize the image, creating a black and white image, separating the text into the foreground and the rest of the image into the background.

4.4.2 Rotations

Depending on the original orientation of the object in the image, the perspective transformation may result in a rotated image. However, with the exception of Tesseract, the other OCR programs are not able to detect whether a piece of text is rotated or not. While this case could be mitigated by detecting the location of the text and calculating its angle to rotate the image back to align with the image axis, an easier approach exists, that is less prone to error.

Due to the nature of the contour detection and perspective transformation, the image can only be rotated by either 0°, 90°, 180° or 270°. As such, if no text is initially recognized by the text detection model, the transformed image can be simply rotated by 90°. If again no text was recognized, this step can be repeated until all four possible rotations were considered.

4.4.3 Text Recognition using OCR

After detecting the area of the text, the last step in the pipeline is the text recognition on the transformed image. All the text recognition models introduced in section 2.3 simply take an RGB image and use their default models to begin their text recognition. With the exception of Keras-OCR, all the OCR programs also have the ability to be set to a specific language to detect different kinds of writing systems, i.e. Chinese characters or Cyrillic script. For the purpose of this thesis, the language falls back to detect Latin characters.

As output, all three OCR programs return an object for every recognized string in the image containing the recognized text, its bounding box and, in the case of Tesseract and EasyOCR, a confidence value. For Tesseract and Keras-OCR, each individual word is returned as its own object, while EasyOCR will return an entire line of text. Figure 4.7 shows the result of the text recognition using EasyOCR.

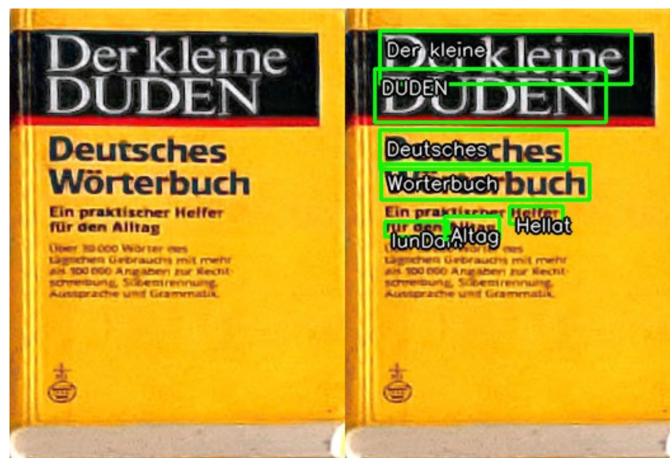

Figure 4.7: Text Recognition using EasyOCR on a transformed image

5 Evaluation

In this chapter, we present the results obtained by testing the developed methodology and implementation of the text recognition. First, we introduce a way to quantify the results by determining a metric for the object detection and text similarity. Next, we describe the environment and setup under which the test data was recorded. Lastly, we evaluate the individual parts of the image processing pipeline, describing potential sources of errors, as well as the overall performance of the pipeline as a whole. Additionally, we take a look how well the text recognition performs without the preprocessing part of the pipeline.

5.1 Evaluating Object Detection

As described in section 2.2.4, to evaluate the accuracy of an object detection implementation, one typically computes the corresponding Intersection over Union, which would indicate how big the overlap between the detection and the ground truth is. However, due to the nature of the results and the specific approach chosen to detect the object, the Intersection over Union is not applicable to accurately calculate the accuracy of the object detection for the intended purpose of subsequent text recognition.

The results can be generally grouped into three categories, depending on whether the object was not detected, the object was fully detected or the object was only partially detected.

For the first case, the evaluation of the object detection is trivial, as the object was either not detected at all, or another object was wrongly detected instead, leading to an accuracy of zero.

The second case is the result of the object detection being successful, by transforming the largest contour in the image, which would naturally be the object. While the detection could still cut off parts of the object or include an area beyond it, for the purpose of the text recognition task, it only matters whether all the text is still inside the detected area and not cut off or outside it. While Intersection over Union could still be applied to measure the accuracy of the object detection, it would only indicate a sufficiency, not a necessity for the task at hand.

Lastly, a partial detection of the object is caused by the detection of the largest contour failing, but falling back on one of the the next largest contours. The reason Intersection over Union cannot be applied in this case is that, while the partially detected area is a subset of the complete object, the area of the partially detected object is solely dependant on how the object is segmented. As such, different objects cannot be compared with each other, as their partially detected areas have no relation to the accuracy of the detection.

Instead, the evaluation of the object detection simply categorizes the detection into the aforementioned cases of whether the object was detected as a whole, only partially detected with the detected area being a subset of the whole object, or not detected, either due to a wrong detection outside the object, or not being detected at all.

5.2 Quantifying and Evaluating Text Similarity

In this section, we evaluate the different means to compare strings with each other introduced in section 2.3.2 for our specific use case, how to calculate the specific text similarity of the test images and how to determine whether a text was correctly identified or not.

5.2.1 String Matching

To evaluate the accuracy of the text recognition at the end of the processing pipeline, there needs to be a metric to determine the similarity between the recognized text and the actual text inside the image. As mentioned in section 2.3.2, there are several ways to determine this similarity, though considering one cannot assume anything about the recognized text, or even if any text was recognized to begin with, this limits us to only consider approximate string matching methods to determine the text similarity. Due to the uncertain nature of the results, some of the distance measures for text are not applicable to both the test images, as well as other real-life applications.

Hamming Distance

Since we do not know whether we detect any text or how many characters the string will consist of, the Hamming distance cannot be used as a measure of distance in our case. It only allows substitutions for determining the text similarity and as such relies on the compared strings to be of the same length. Theoretically, the strings can be padded to the same length, however, it is unlikely that an otherwise correctly detected string was cut before its end or includes more characters than it should after it.

Jaro Similarity

The Jaro similarity is also a bad pick for the given context, as the similarity score of two words is mostly dependent on the same characters being at a similar position in the respective string. This is unlikely, as a wrongly detected text will most likely contain different characters entirely, rather than the correct characters transposed at slightly different positions inside the string. Even when this case does occur, a string with its characters shuffled should not be considered similar for a strict reading task, as shuffled characters are more indicative of being recognized incorrectly, than recognized correctly at a different position.

Gestalt Pattern Matching

The gestalt pattern matching could be used, as the similarity score is based on the largest common substrings, which one could expect from a correctly recognized string. This holds even if the recognized string is only partially recognized correctly, as mistakes such as transposed characters next to each other are unlikely. Similarly, it is unlikely that entire substrings are in the wrong order in the otherwise correctly recognized string. Additionally, it can also be applied to strings of different lengths.

Levenshtein Distance

Similar to the gestalt pattern matching, the Levenshtein distance can also be used as a measure for the similarity of recognized text, as it is based on reconstructing the actual word from the recognized string with the minimum operations needed. This means that a wrongly identified character can simply be replaced, a wrongly added character removed and a missing one be added, while all the correctly identified substrings, regardless of their position, are left alone.

5.2.2 Determining Text Similarity

Functionally, the Levenshtein distance and the gestalt pattern matching can be used interchangeably, as the sum of the longest common substrings between two strings is equal to the Levenshtein distance subtracted from the length of the longer string. This stems from the fact that the longest common substrings can be formed by removing every character not part of these substrings and thus trivially leading to a Levenshtein distance of the length of all the longest common substrings subtracted from the length of the longer string. The only difference in text similarity would be caused by how the length of the two compared strings would influence the final text similarity. As such, both methods are used to determine the text similarity for the test images.

Using both the Levenshtein Distance and the text similarity of the gestalt pattern matching as a measure for text similarity, the results of the processing pipeline were compared to the strings on the objects themselves, specifically the most prominent and most recognizable piece of text, such as the book title and author, or the brand and product name for packaging. Specifically, entire blocks of text are compared to each other, instead of every single word. The reason for this is that each block of text should be able to be recognized as a whole and not just by its individual parts, in the case only parts of a block can be recognized. Beforehand, all strings are converted into lowercase, as case sensitivity is not an important factor for determining whether the actual content of the string is the same or not. However, this methodology would need to be changed if applied in a context in which the case sensitivity of the text is crucial, for instance the recognition of product IDs.

Determining whether a string is recognized or correct is not a trivial problem, as it highly depends on the context of the text recognition, as well as what a robot considers to be a similar string, as opposed to a human. The first approach was to allow an absolute amount of errors in the text detection, as given by the Levensthein distance. This decision was based on the assumption that everything beyond a certain threshold of errors could not be considered the same string of text anymore, as the text recognition already incorrectly recognized a set of characters. This thinking is similar to how a human can misspell a word in a sentence and it be considered inside an error tolerance, as errors are to be expected, but already two misspelled words could be indicative of something being wrong or unexpected. However this idea was quickly discarded, as it does not scale with longer strings and causes shorter, incorrect strings to be wrongly identified as correct. As such, instead of using an absolute value, a relative value based on the text similarity is used.

While the gestalt pattern matching is already measured as a value between 0 and 1, denoting the similarity of the two strings in percentages, the Levenshtein distance returns an absolute value of the minimum amount of changes needed. To calculate the text similarity, the ratio between the Levenshtein distance and the maximum length of the two strings is computed and

then subtracted from one, as shown in equation 5.1, thus giving the ratio of correctly identified characters.

$$\text{sim}_L = 1 - \frac{L(s_1, s_2)}{\max(|s_1|, |s_2|)} \tag{5.1}$$

with sim_L being the text similarity using the Levenshtein distance function L and $|s_1|$ and $|s_2|$ being the length of the two strings respectively. This would map a Levenshtein distance of zero to a text similarity of 100%, while a word without a single matching character has a text similarity of 0%, as the Levenshtein distance would be equal to the length of the compared string. The difference between the text similarity using the Levenshtein distance and the gestalt pattern similarity is in how the value is normalized, with the gestalt pattern similarity using the average length of the two strings, while the Levenshtein distance uses the length of the longest string.

The next problem is setting a threshold for whether two different strings are to be considered similar or different. This value could be set arbitrarily in accordance to what one would either expect or demand as criteria for success. However, as text recognition is highly dependent on its surroundings, the threshold of the text similarity for whether a string can be considered to be correctly identified, was determined by applying Otsu's method for the respective test results. This is done by interpreting the text similarities as entries in a grayscale image with a high text similarity being a pixel with a high intensity and a low text similarity with a lower intensity. This threshold can be further adjusted to be less or more strict with the identification, depending on the task at hand.

5.3 Testing Environment

The images for the evaluation process were taken using an Azure Kinect DK, which is fitted on top of the PR2 robot platform, tested in the TAMS department of the University of Hamburg. Overall, three different types of images were taken.

For the first type, the objects were placed into the Shadow Hand of the PR2 robot in front of the camera. For the second type, the objects were placed on top of a table in front of the PR2. The objects were then recorded from three different angles by adjusting the height of the PR2 body to its minimum height, its maximum height and a height in the middle. Lastly, for the third type of image, the objects were placed on top of a whiteboard and orthogonal to the camera angle, enabling a direct front view onto the top of the object. Figure 5.1 and figure 5.2 show the general recording setup and the different views of the PR2 on the objects. Overall, 75 individual images were recorded.

Figure 5.1: Image recording setup using the PR2 platform

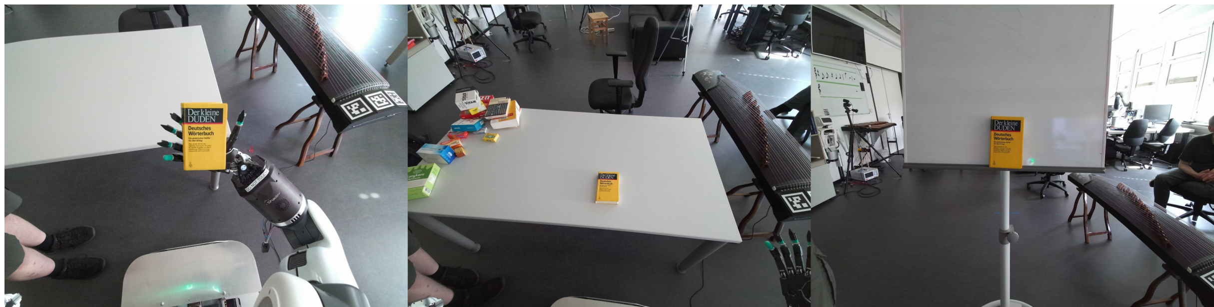

Figure 5.2: Object in the Shadow Hand, on a table and on a whiteboard

The images have an output resolution of 4096×3072 pixels, however due to the nature of the wide angle lens used by the camera, the objects only occupy a small portion of the image and were manually cropped to exclude everything clearly outside the area of interest. This was also done to reduce the background to a homogeneous area for reasons explained in section 4.2.

The collection of objects used for the purpose of testing consists of different books and box-shaped packaging of different colors and sizes. For the second type of image, the objects were placed randomly on the table, laid down with some degree of variation in position and rotation, though always facing the camera. When the object allowed it, it was also placed on another side standing up, creating a different view of the same object and text.

5.4 Evaluation of the Pipeline

In this section, we go over the individual parts of the processing pipeline and evaluate how well each critical step, as well as the pipeline as a whole, succeeds in detecting the objects and recognizing the respective text of the test images. An excerpt of the test images in their respective stages in the pipeline and the results of the text recognition can be found in appendix A.

5.4.1 Evaluating Individual Steps of the Pipeline

To better explain the results of the whole processing pipeline, we first evaluate the individual steps and assess how well they work, as well as name potential causes for a step to fail and to subsequently cascade into a bad result. Specifically, we go over the steps of the edge and contour detection, as well the text detection, as they are the most critical steps.

Edge Detection

While the boundary of the objects were most often detected after the respective amount of passes of the preprocessing, they were not always correctly isolated with many different artifacts plaguing the edge detection. Most of these artifacts include unclean and distorted edges caused by noise and the shadow of the object onto the background. Larger details on the object itself were not always blended into the background, causing additional edges inside the object to appear. Depending on how clear an edge in the object was, gaps between edge segments could not always be closed, with segments in the object of a similar color to the background sometimes not even having their edge detected at all, leading to an incomplete outline. Lastly, actual physical edges inside the object, such as between the front face and the top face of the object, were only seldom recognized, leading to the individual faces of the object not having their own boundary.

Contour Detection

As the contour detection relies on the successful edge detection, any image with the previous step failing will also automatically result in either a wrongly identified contour or no recognized contour at all. Additionally, the face for the object, on which the text is located, might not be able to be recognized as a rectangle all together, either due to missing edges of the face, or noise extending an edge outside the accuracy parameter set for the reduction of the polyline. A result of this is the potential subdivision of the original object into smaller areas, which might not contain all the necessary information to correctly identify the object, beyond also returning wrong information for the position of the object, which could lead to further errors down the line not associated with the text recognition.

Depending on whether every edge along the face of the object with the text was detected, it may happen that the recognized contour includes another face, whose edges are inside the accuracy bound for the detection of the rectangle. This will result in the corners of the rectangle to be positioned alongside a face other than the actual face with the text, leading to a distortion of the subsequent perspective transformation along the area of interest, as the perspective transformation is based on the idea of transforming one 2-dimensional plane, in this case the

largest rectangle detected on the object, into another plane with a top-down perspective. In the aforementioned case, two planes would be combined into one not perfectly aligning with the original edges of the object and thus being distorted. The amount of distortion depends on the difference in size of the two planes, but is technically limited by the accuracy parameter of the detection for the rectangle. In the case of the test images, this would lead to another side of the object to appear, for example the pages visible from the side of a book, or the top part of a packaging, but with the front of the object still being transformed into the mostly correct perspective and rotation. While this leads to a slightly imperfect transformation, the text recognition is robust enough for this error to not have any strong impact, as seen in the results of section 5.5.1.

Binarization

While the binarization generally succeeded in separating the text from the background, due to the size of the transformed image and the text, it also closed the general text area together as a single area. As such, instead of separating the characters of the text and the background into black and white, it often turned individual letters into unrecognizable blobs and the whole text into similar colored areas, which could at best be used as a mask. Since the text recognition is not limited to a full binarization of the text and its background and also works successfully on color images, the final binarization step was removed from the pipeline entirely, as its inclusion is debatable and can actively make the results worse if applied on smaller text.

Text Recognition

Similar to the contour detection, the text recognition step also relies on the success of the previous steps and as such will yield either wrong or no results for any image whose object was not correctly detected and thus either does not contain the text for the text recognition, or was not properly transformed into the top-down view. Beyond this, the text recognition was mostly successful on all the tested OCR programs, with a failed recognition mostly being caused by the text of the transformed area being either too small, partially obscured or generally illegible.

Other difficulties include stylized text, characters and logos, which were usually only partially recognized correctly, as well as numbers and special characters, such as umlaute, which were recognized as standard characters from the 26 letter Latin alphabet. Other common mistakes include the mix-up of the letters `i` and `l`, `ß` and `B` as well as of the letters `M`, `N` and `V`.

The confidence values of the OCR programs were unhelpful, as they did not always correlate with a correct detection of the text. For example, a recognized string with a low text similarity could still be denoted by a high confidence value, while a correctly identified string was sometimes given low confidence. As such, the returned confidence value could not be used to reliably filter out badly recognized words, while keeping the results with a high confidence value.

Additionally, there also exist false positives and false negatives. False positives denote the case of any text being recognized, despite the transformation being unsuccessful and the image not containing any text, while false negatives denote text on correctly transformed objects not being recognized at all. This further worsens the results, as correct passes for the preprocessing step, in which the text should be recognized, would either be prematurely skipped or ignored altogether.

5.4.2 Examples of Common Errors

This section shows common errors found in several places along the processing pipeline for the test images.

Unrecognized and Unclosed Edge

Depending on the color of the object and background, it is possible that an edge is either not detected at all or could not be properly closed. Figure 5.3 shows an example with the color of the box matching the background in two places, which led to the top edge not being correctly detected. Meanwhile, figure 5.4 shows how an edge could not be closed, as indicated by the red circles, and thus did not form a boundary for the contour detection.

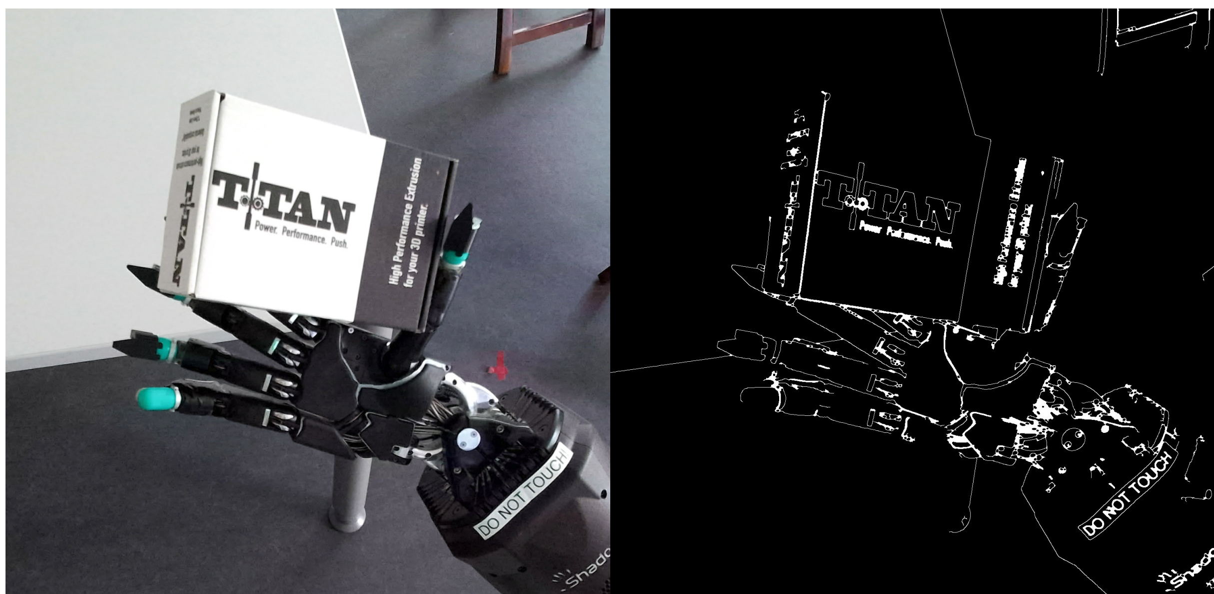

Figure 5.3: Upper edge of the object is not recognized due to similar colors of the background

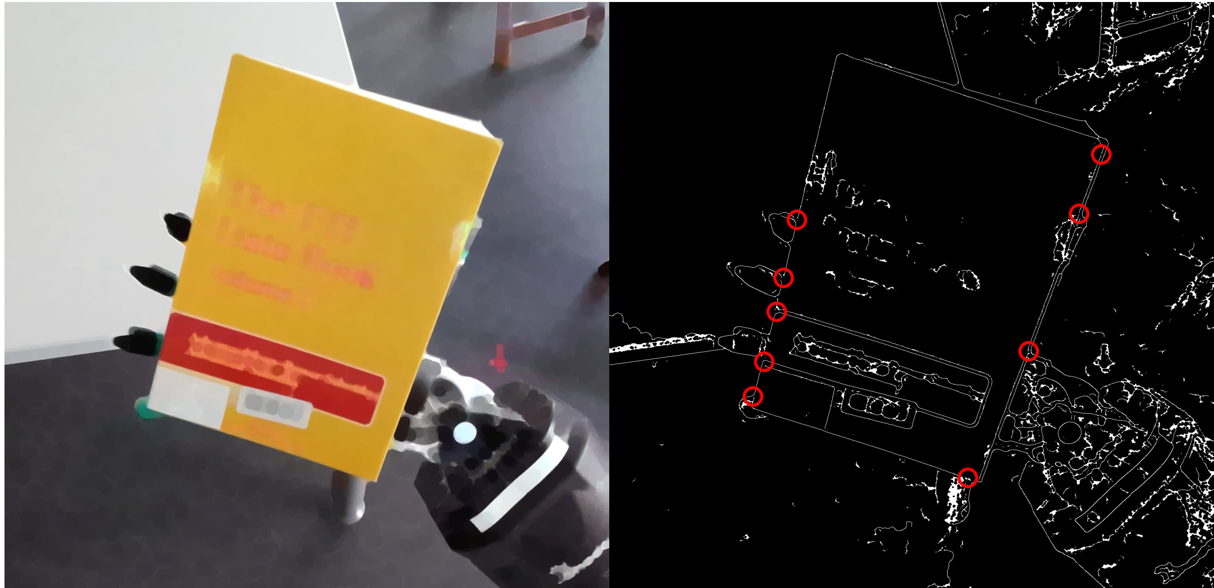

Figure 5.4: Edges are not fully closed in areas denoted by red circles

Partial Recognition

When the edges were not successfully recognized or closed, the contour detection would fail for the object as a whole, but could still detect the next largest contour. In the case of the object being subdivided, this would cause a rectangle inside the object to be detected, though this rectangle will most likely not contain all the necessary information for the identification of the object. Figure 5.5 shows how only a part of the book was recognized and thus only contains a part of the title, but not all the text visible on the book.

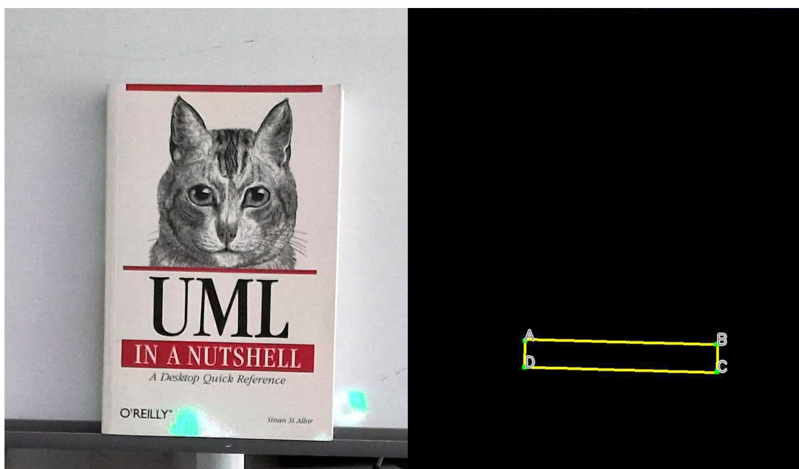

Figure 5.5: Only the part of the title inside a separate rectangle was recognized

Several Faces of the Object

Whether every edge alongside the face of interest was detected correctly or not, it could happen that the contour detection also includes another face of the object, aside from the one with the

text. This will cause a slightly distorted output image, as the perspective transformation will transform not just the area of interest, but also another area into a top down view. Since these two faces have different orientations in space, their corner positions will not align with each other.

However, as explained in section 5.4.1, this possible distortion is limited and small enough to not impair the results. Figure 5.6 shows the book cover along its side view of the pages. The distortion is noticeable, but not strong enough to make text recognition fail, as the lines on the side of the cover have been mostly straightened to be parallel to the image axis.

Figure 5.6: Contour also includes the side of the book. Text on the cover is still recognized

Noisy Edge

Due to noise at the edge of the side of the object, some contours are unable to include the correct edge into the actual boundary of the object, leading to a failed transformation. Figure 5.7 shows how the noise, introduced through the shadow of the book, leads to the contour not being able to be reduced to a rectangle, as the left edge is only closed through the edges introduced by noise.

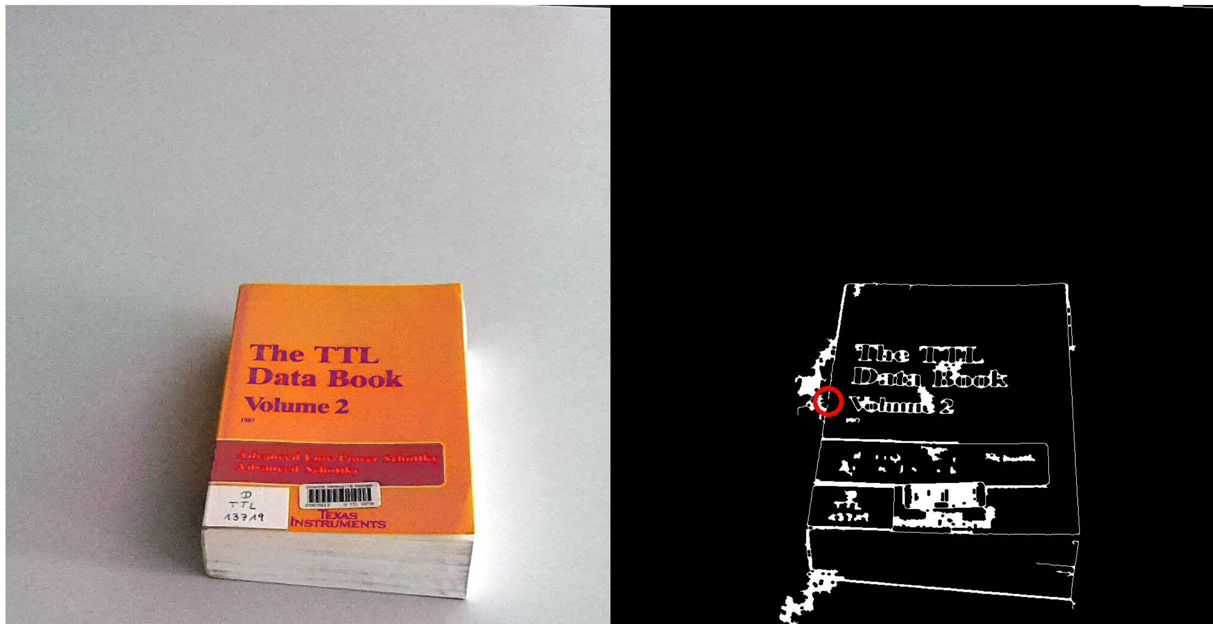

Figure 5.7: Noise at the side of the edge detection. Edge along the side is not actually closed, as indicated by the red circle

Unrecognized Text

If the text in the original input image was already illegible or hard to recognize, the transformed image will not be able to improve the text, as the transformation does not create new information, but only transforms the already existing pixel data. Figure 5.8 shows how the small size of the text also makes it hard to read the title for humans.

Figure 5.8: Text in the transformed image is too illegible for EasyOCR, as the text in the original image is already hard to read

False Positive and False Negative

In some cases, the text recognition either recognizes text where there was none, or didn't recognize any text, despite it being clearly visible. On the left side, figure 5.9 shows how the bar codes are being recognized as the strings `A` and `wil` respectively, while on the right side, an otherwise correctly transformed book cover is detected with no text beyond a seemingly recognized whitespace character.

Figure 5.9: A false positive (left) and a false negative (right)

It should be noted that this type of error occurred exclusively when using Tesseract for text recognition.

Error on Stylized Logo

In the case of stylized logos or strings of text, the text recognition might not recognize specific characters or mistake them as another. In figure 5.10, the `I` in `TITAN` is stylized as a mechanical part and thus not recognized as a normal character, leading to the logo being recognized as `trlAv` and `TTAN` respectively.

Figure 5.10: Stylized character not being recognized

Binarization

If the characters on the object are either small or close together, the binarization causes the characters to either close in on themselves or to be reduced to the background. In figure 5.11, one can see how the upper `CHEEZE-IT` is not readable anymore by its individual chararcters, as the small letter spacing causes the binarized characters to connect with each other.

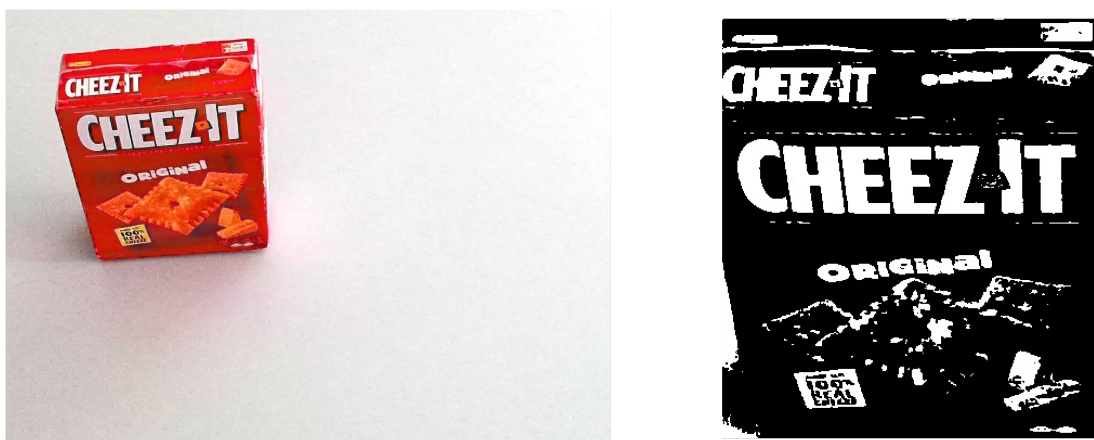

Figure 5.11: Binarization making the characters of the upper text indistinguishable

5.5 Evaluation of the Text Recognition and Results

In this section, we evaluate the text recognition by comparing the results of the aforementioned pipeline according to the specific OCR program used. Additionally, we compare how each OCR program fares when the text recognition is applied not to the preprocessed image, but the original unprocessed input image. Lastly, we determine how effective the proposed pipeline is in improving text recognition by comparing the results of the first two evaluations with each other.

5.5.1 Evaluation of the Entire Pipeline

Overall, the results of the pipeline are largely dependant on the correct detection of the object, as the actual text recognition yields good results. Table 5.1 compares the three OCR programs used to each other and it clearly shows that, while the general text similarity between the recognized text and actual text is relatively high, the amount of correctly or at least partially transformed objects is very low.

The pipeline was executed on a set of 75 test images, created as described in section 5.3, with Tesseract, EasyOCR and Keras-OCR acting as the text recognition step at the end. The results were grouped by whether the object was either detected, partially detected, meaning only a part of all the text was in the transformed image, or not detected at all. From the detected and partially detected images, it is documented whether the text is recognized by the text detection or not. Lastly, for all the text that was recognized, the arithmetic mean, as well as the threshold for correctly identified text, for the text similarity using the Levenshtein distance and the gestalt pattern matching is noted.

OCR	Object detected		Text	Levenshtein		Gestalt	
Engine	Full	Partial	Detected	Mean	Thresh.	Mean	Thresh.
Tesseract	6	10	12	0.934	0.875	0.950	0.875
EasyOCR	10	15	23	0.787	0.796	0.818	0.803
Keras-OCR	9	16	24	0.722	0.281	0.748	0.330

Table 5.1: Comparison of Tesseract, EasyOCR and Keras-OCR on detected text

As can be seen in table 5.1 the performance, or rather the robustness, of the object detection part of the pipeline is rather poor with only a third of the total objects being recognized at all using EasyOCR and Keras-OCR and only $\sim 20\%$ using Tesseract. Additionally, even among the recognized objects, more than a half of them have only been detected partially, meaning a complete identification of the object is not guaranteed.

On the other hand, in the cases the object detection and subsequent perspective transformation was correctly applied, the text detection was rather successful. While not every text was detected, this can mostly be attributed to the text being hard to read, rather than an inherent problem of the OCR program itself. The average text similarity is also high enough for a human to still recognize the detected text, even with a wrongly identified character among four others.

Especially EasyOCR performed very consistently with no detected string having a text similarity lower than 60%, except for two that were completely wrong, while Keras-OCR has some outliers, but was otherwise also consistent and mostly correctly recognized the text to an acceptable level.

On the other hand, despite the surprisingly high text similarity of the recognized text, Tesseract performed the worst overall. Compared to the other two OCRs, Tesseract detected a third less objects, leading to a way worse result overall when taking into account all the objects that were not recognized at all. This can also be extended to the general text detection, as only 75% of the texts were recognized as text, while EasyOCR and Keras-OCR only have two and one outlier respectively. This all leads to the high text similarity for using Tesseract being mostly in part due to survivorship bias of the data, as it did not have to be applied to text that was already filtered out beforehand. As such, the high text similarity of Tesseract should be seen critically, as it applies to less text samples than the other OCR programs.

Table 5.2 shows how the individual OCR programs fare when normalized on all 75 test images, instead of only the recognized objects and texts. The text similarity for every object and text not recognized is assumed to be 0.

OCR Engine	Mean Levenshtein	Mean Gestalt
Tesseract	0.149	0.152
EasyOCR	0.231	0.239
Keras-OCR	0.231	0.239

Table 5.2: Comparison of Tesseract, EasyOCR and Keras-OCR for all images

In this table, it is more apparent how Tesseract produces less accurate results than EasyOCR and Keras-OCR. It is also more apparent how the general robustness of the pipeline is too low to reliably detect an object and recognize the text on the object itself.

5.5.2 Evaluation of Text Recognition without Perspective Transformation

To see how well the text recognition works outside the proposed processing pipeline, we next compare the three OCR programs on the same test images as in the previous section, but without the object detection and perspective transformation. This means that all text on the objects needs to be recognized from a variety of angles and rotations from the original image.

The evaluation of these results is the same as in section 5.5.1, with the exception of the object detection not being a factor anymore, while whether the text was detected or not is still documented. Table 5.3 shows the results of the text detection without the preprocessing pipeline.

OCR	Text	Levenshtein		Gestalt	
Engine	Detected	Mean	Thresh.	Mean	Thresh.
Tesseract	17	0.882	0.678	0.912	0.822
EasyOCR	73	0.845	0.658	0.868	0.749
Keras-OCR	74	0.872	0.800	0.895	0.841

Table 5.3: Comparison between Tesseract, EasyOCR and Keras-OCR without object detection and perspective transformation

Similar to the results in table 5.1, there also exists a large discrepancy between Tesseract and the other OCR programs concerning how much of the text was detected. While EasyOCR and Keras-OCR could not detect the text of only two and one object respectively, Tesseract only

recognized the text from 17 out of the 75 images. Similarly to the results of the images with the perspective transformation applied, Tesseract has the best results in terms of text similarity. But these results have to be questioned as well, since the same survivorship bias still applies, as Tesseract only detected a fraction of the text on the object, but it achieves a good text similarity on those it does recognize. Table 5.4 takes into consideration all the 75 original test images, again assigning a text similarity of 0 for any text not recognized.

OCR Engine	Mean Levenshtein	Mean Gestalt
Tesseract	0.200	0.206
EasyOCR	0.822	0.845
Keras-OCR	0.861	0.883

Table 5.4: Comparison of Tesseract, EasyOCR and Keras-OCR for all images without object detection and perspective transformation

This table gives a much clearer and comparable look of the robustness of the respective OCR programs overall. Tesseract, while returning good results on a successful detection, has been shown to be unreliable in this specific context, while EasyOCR and Keras-OCR can consistently recognize the text of an object with a good text similarity and only performing slightly worse than Tesseract for recognized text.

5.5.3 Evaluation of Text Similarity with and without Perspective Transformation

Lastly, we compare the results of section 5.5.1 and 5.5.2 to determine whether a successful object detection and subsequent perspective transformation yields an overall better result for text recognition. Due to the nature of each approach not recognizing any text in a certain subset of all the original 75 test images, this comparison is limited to the intersection of images where both approaches recognized the text on the same object. For EasyOCR and Keras-OCR, this reduces the sample size to 18 and 17 images respectively. In the case of Tesseract, the intersection includes only two images and is thus excluded from the comparison for a lack of meaningful data.

OCR Engine	Mean Levenshtein (Pipeline)	Mean Gestalt (Pipeline)	Mean Levenshtein	Mean Gestalt
Tesseract	N/A	N/A	N/A	N/A
EasyOCR	0.872	0.902	0.864	0.895
Keras-OCR	0.897	0.923	0.913	0.928

Table 5.5: Comparison of Text similarities between transformed and unprocessed images

As table 5.5 shows, in the context of the test images, the perspective transformation, when successful, results in a similar text similarity as if no perspective transformation was applied. As such, the perspective transformation is only useful when the text could not otherwise be detected on the original input image. This can be seen in figure 5.12 with the text detection on the original image barely recognizing the title, while the recognized text on the transformed image still has a text similarity of $\sim 67.5\%$.

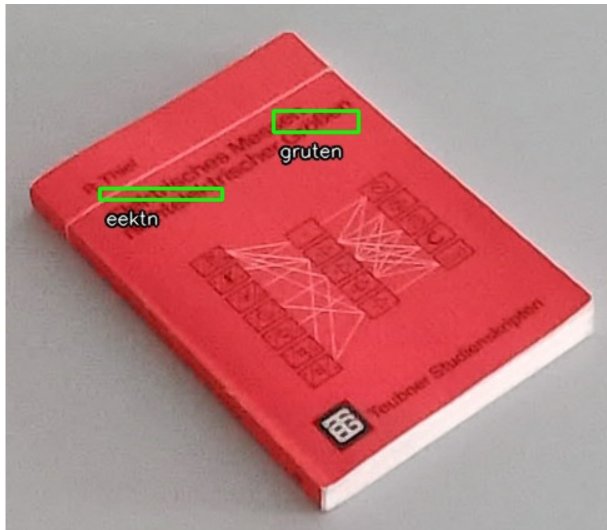

Figure 5.12: Title being recognized on the transformed image, but not the original

6 Summary and Outlook

This thesis explored the possibility of generalized object detection and text recognition in a robotics context using only normal image data recorded by a camera. This was done by introducing a processing pipeline for detecting an object in a given context, isolating it from the background and transforming it into a top-down view for text recognition using several different OCR programs to determine the final output. It was also discussed how to evaluate the recognized text by comparing it to the actual text in the image and determining the text similarity.

While the object detection part of the processing pipeline has been proven to be mostly unreliable in consistently producing an accurate top-down view of the object, the text recognition still proved to be generally robust enough to accurately identify text with a text similarity of around $\sim 85\%$, even without the perspective transformation. However, tests have also shown that the transformation into the top-down view might be required for text to be detected when rotated or seen from a sharp angle.

In general, the tested OCR programs perform especially well if the text has been properly prepared, either with the original object in the image being recorded facing the camera or being successfully transformed into the top-down view. However, some of the OCR programs still suffer from occasional errors, such as not recognizing any text on an otherwise properly prepared image, with Tesseract not being recommendable on its default model, despite its good results on successfully recognized text, while EasyOCR and Keras-OCR are more robust and applicable in more general tasks and environments, such as those introduced in section 3.1.

As the object detection part of the processing pipeline is the one most prone to errors, the general approach could be improved by incorporating neural network techniques for detecting objects, as the current one solely relies on classical computer vision. Systems such as YOLO for object detection and Mask R-CNN have already shown good results, with a large variety of different classes of objects being categorizable and they can be expected to become more robust with the further development of other object detection models and larger training sets. These systems have also already proven their ability to detect several objects at the same time, unlike the proposed approach, which only allows for the detection of a single object.

Additionally, further information can be utilized for the perspective transformation of the objects, as the current approach solely takes an RGB color image into account, but could also be extended by a depth map of the current view, as well as information from the robot, for instance tilt and angle of the camera and positional information. This could also be extended to give feedback to the robot, in the case of an unsuccessful object detection or text recognition, with instructions to create a better view of the object.

Appendices

A Excerpt of Test Images

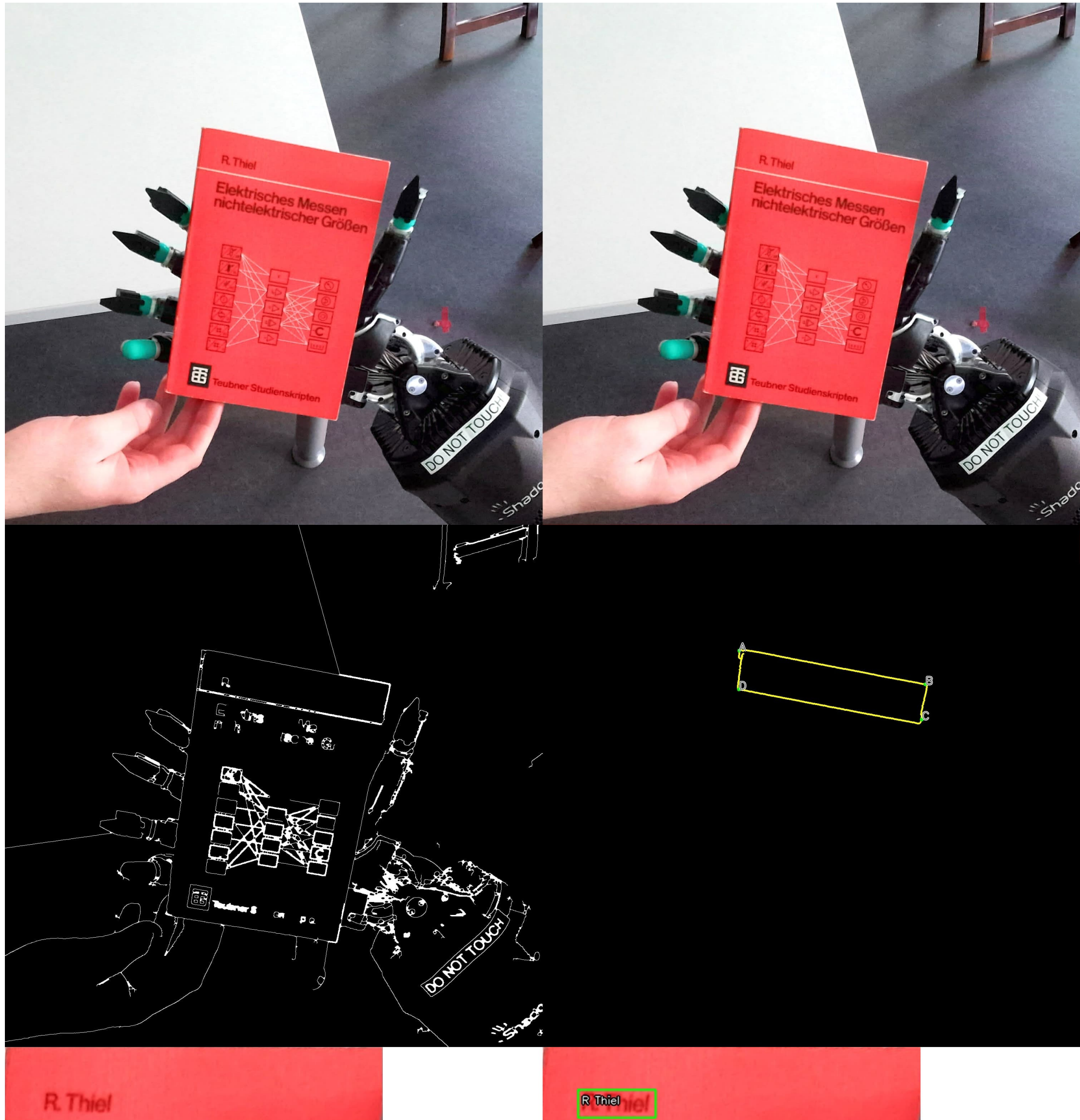

Figure A.1: Recognized text: R Thiel

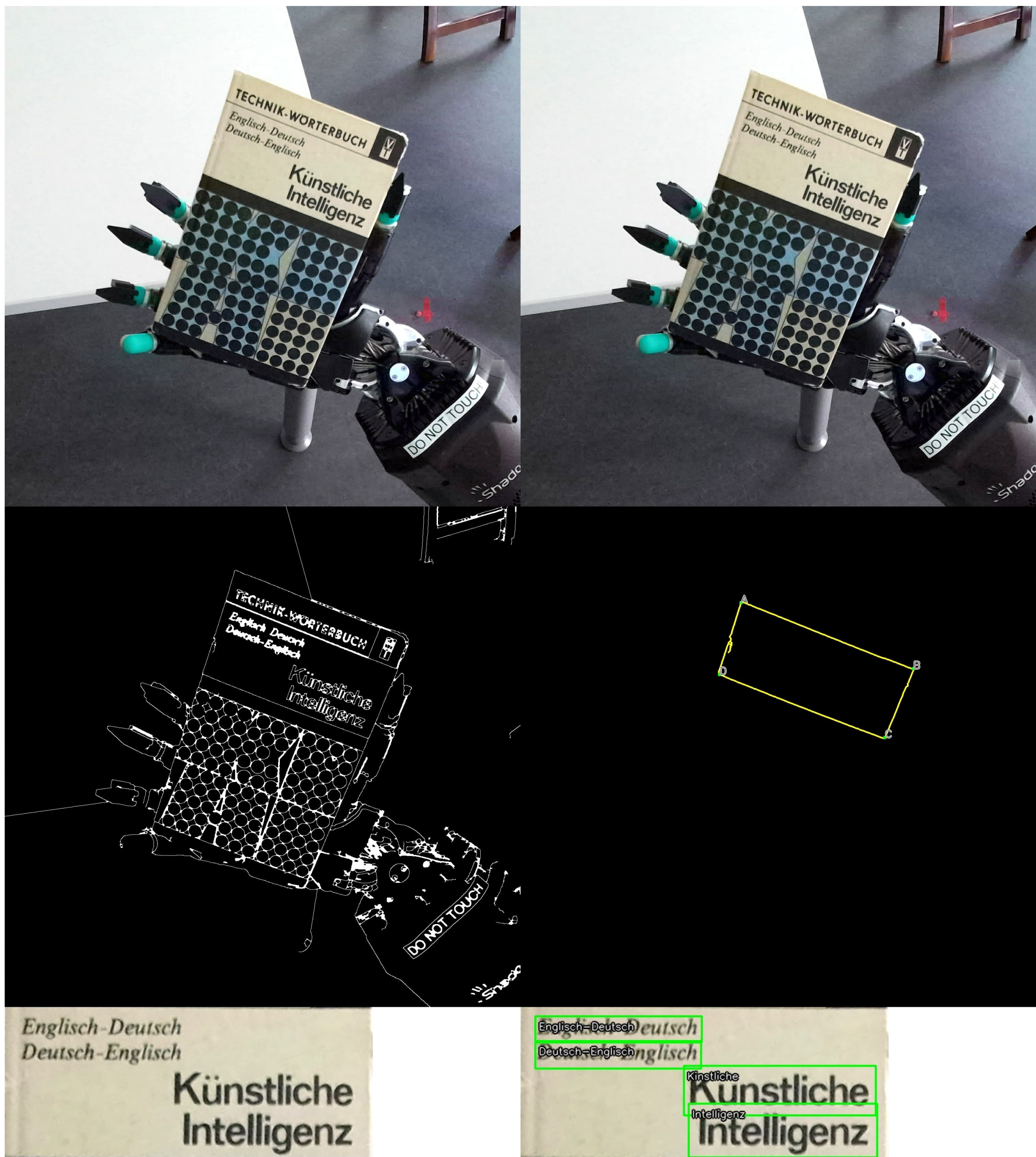

Figure A.2: Recognized text: Englisch-Deutsch Deutsch-Englisch Kinstliche Intelligenz

Figure A.3: Recognized text: Der kleine DUDEN Deutsches Worterbuch Hellat lunDani Altag

Figure A.4: Recognized text: The TTL Data Bookk Vlumne 2 ScmPoner Sctotvy TTL 437/9 INSDORENTS

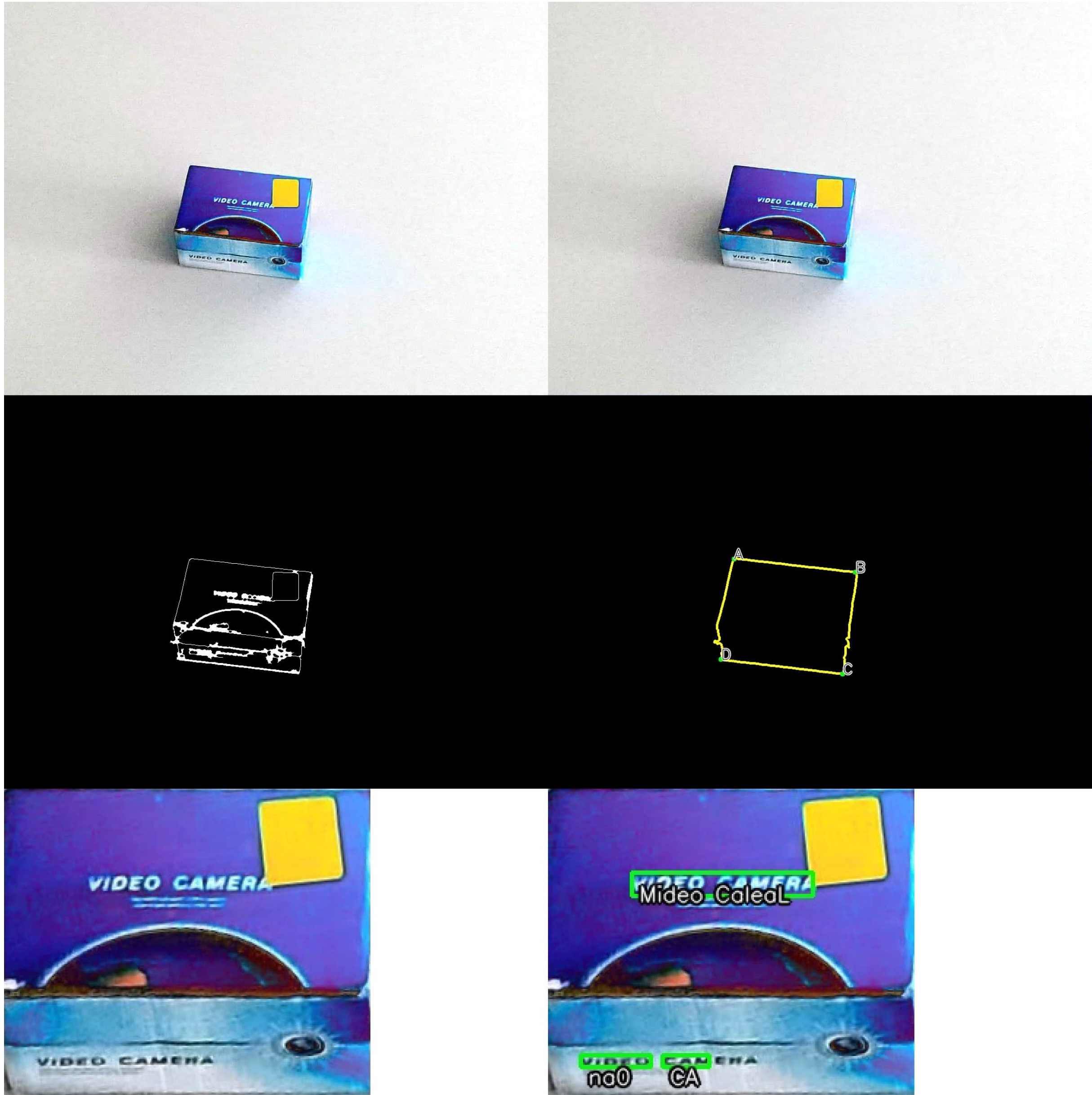

Figure A.5: Recognized text: Mideo CaleaL na0 CA

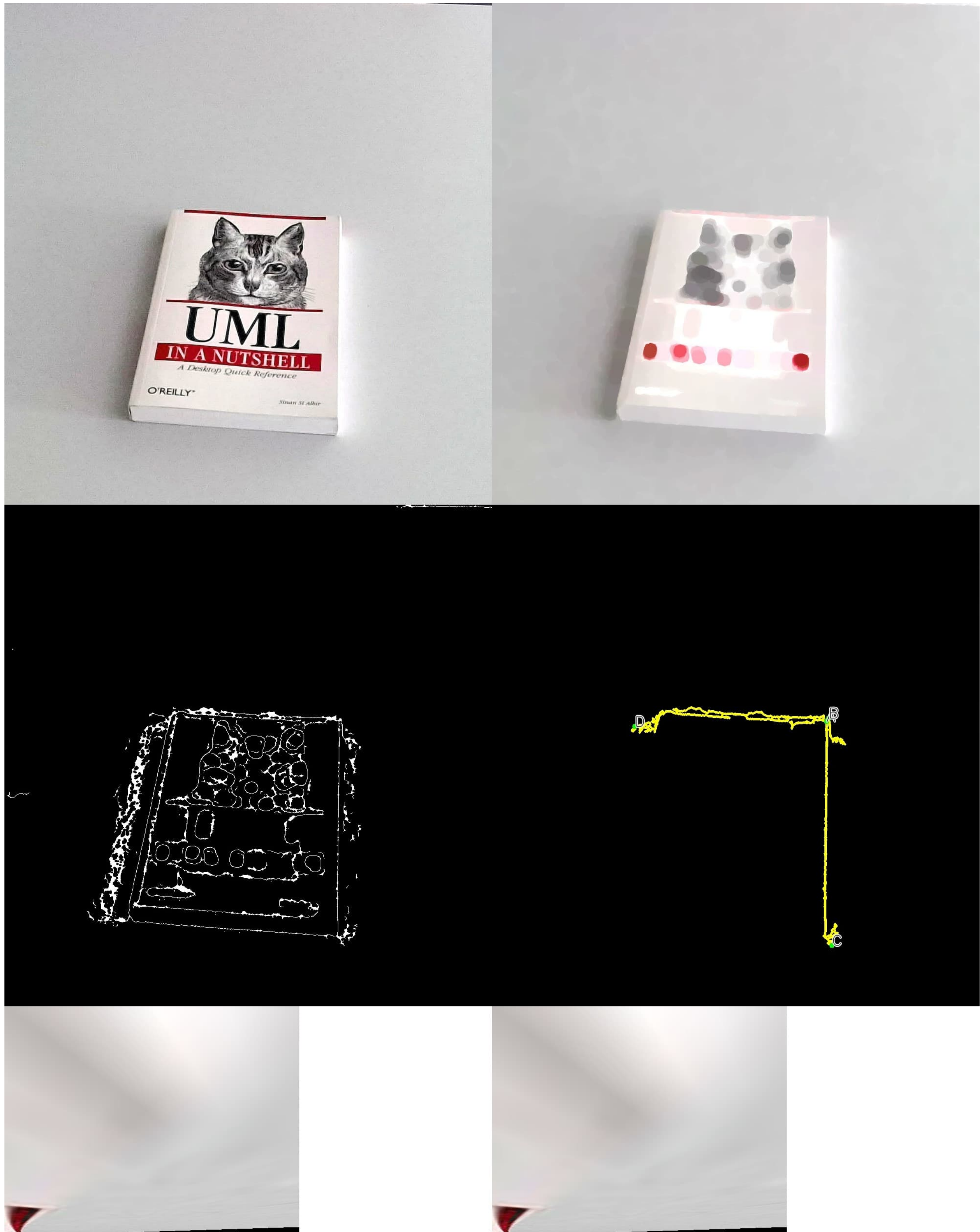

Figure A.6: Recognized text: N/A

List of Figures

2.1 From left to right: Original, dilation, erosion, opening and closing 7

4.1 Diagram of the processing pipeline 16
4.2 Before (left) and after (right) applying automatic brightness and contrast correction. The top row shows the image itself and a histogram is included below. 19
4.3 Before (left) and after (right) morphological closing 20
4.4 Input (left) and output (right) of edge detection 20
4.5 Before and after contour and corner detection 22
4.6 Image (left) is transformed to the top-down perspective (right) 23
4.7 Text Recognition using EasyOCR on a transformed image 24

5.1 Image recording setup using the PR2 platform 29
5.2 Object in the Shadow Hand, on a table and on a whiteboard 29
5.3 Upper edge of the object is not recognized due to similar colors of the background 32
5.4 Edges are not fully closed in areas denoted by red circles 33
5.5 Only the part of the title inside a separate rectangle was recognized 33
5.6 Contour also includes the side of the book. Text on the cover is still recognized 34
5.7 Noise at the side of the edge detection. Edge along the side is not actually closed, as indicated by the red circle 34
5.8 Text in the transformed image is too illegible for EasyOCR, as the text in the original image is already hard to read 35
5.9 A false positive (left) and a false negative (right) 35
5.10 Stylized character not being recognized 36
5.11 Binarization making the characters of the upper text indistinguishable 36
5.12 Title being recognized on the transformed image, but not the original 40

A.1 Recognized text: R Thiel 43
A.2 Recognized text: Englisch-Deutsch Deutsch-Englisch Kinstliche Intelligenz 44
A.3 Recognized text: Der kleine DUDEN Deutsches Worterbuch Hellat lunDani Altag 45
A.4 Recognized text: The TTL Data Bookk Vlumne 2 ScmPoner Sctotvy TTL 437/9 INSDORENTS 46
A.5 Recognized text: Mideo CaleaL na0 CA 47
A.6 Recognized text: N/A 48

List of Tables

5.1 Comparison of Tesseract, EasyOCR and Keras-OCR on detected text 37
5.2 Comparison of Tesseract, EasyOCR and Keras-OCR for all images 38
5.3 Comparison between Tesseract, EasyOCR and Keras-OCR without object detection and perspective transformation 38
5.4 Comparison of Tesseract, EasyOCR and Keras-OCR for all images without object detection and perspective transformation 39
5.5 Comparison of Text similarities between transformed and unprocessed images 39

List of Listings

4.1 Contour Detection after Canny edge detection using OpenCV's findContours() 21
4.2 Approximation of the contour to a rectangle 21
4.3 Calculating homography and transforming the image into a top down view 23

7 References

[1] Jaided AI. *EasyOCR*. Accessed: 2023-05-15. URL: https://github.com/JaidedAI/EasyOCR.

[2] K.Geetha Arulmani and Murali S. *Automatic Rectification of Perspective Distortion from a Single Image Using Plane Homography*. In: *International Journal on Computational Science & Applications* 3 (Oct. 2013), pp. 47–58.

[3] Youngmin Baek et al. *Character Region Awareness for Text Detection*. 2019.

[4] John Canny. *A Computational Approach to Edge Detection*. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698.

[5] David Coleman et al. *Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study*. 2014.

[6] Jifeng Dai et al. *R-FCN: Object Detection via Region-based Fully Convolutional Networks*. 2016.

[7] David H Douglas and Thomas K Peucker. *Algorithms for the reduction of the number of points required to represent a digitized line or its caricature*. In: *Cartographica: the international journal for geographic information and geovisualization* 10.2 (1973), pp. 112–122.

[8] Estevao Gedraite and M. Hadad. *Investigation on the effect of a Gaussian Blur in image filtering and segmentation*. In: Jan. 2011, pp. 393–396.

[9] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. 4. Pearson, 2018.

[10] R. W. Hamming. *Error detecting and error correcting codes*. In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160.

[11] Kaiming He et al. *Mask R-CNN*. 2018.

[12] MVV Kantipudi, Sandeep Kumar, Ashish Kumar Jha, et al. *Scene text recognition based on bidirectional LSTM and deep neural network*. In: *Computational Intelligence and Neuroscience* 2021 (2021).

[13] Alexander Kirillov et al. *Segment Anything*. 2023.

[14] Praveen Kumar et al. *Approximate string matching Algorithm*. In: *International Journal on Computer Science and Engineering* 2 (May 2010).

[15] Vladimir Iosifovich Levenshtein. *Binary codes capable of correcting deletions, insertions and reversals.* In: *Soviet Physics Doklady* 10.8 (Feb. 1966), pp. 707–710.

[16] Minghui Liao et al. *Scene Text Recognition from Two-Dimensional Perspective*. In: *CoRR* (2018).

[17] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015.

[18] Shijian Lu and Chew Lim Tan. *Camera Text Recognition based on Perspective Invariants*. In: vol. 2. Jan. 2006, pp. 1042–1045.

[19] Meta. *Segment Anything*. Accessed: 2023-05-16. URL: https://segment-anything.com/.

[20] Fausto Morales. *Keras-OCR*. Accessed: 2023-05-15. URL: https://github.com/faustomorales/keras-ocr.

[21] Rang M. H. Nguyen and Michael S. Brown. *Why You Should Forget Luminance Conversion and Do Something Better*. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 5920–5928.

[22] OpenCV. *OpenCV Documentation Version 4.7*. Accessed: 2023-05-10. URL: https://docs.opencv.org/4.7.0/.

[23] Nobuyuki Otsu. *A Threshold Selection Method from Gray-Level Histograms*. In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66.

[24] *PR2 IEEE Robotics and Automation Society*. Accessed: 2023-05-14. URL: https://robots.ieee.org/robots/pr2/.

[25] John W. Ratcliff and David E. Metzener. *Pattern Matching: the Gestalt Approach*. In: *Dr. Dobb's Journal* (July 1988).

[26] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016.

[27] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016.

[28] Hamid Rezatofighi et al. *Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression*. 2019.

[29] ROS. *ROS Documentation*. Accessed: 2023-05-14. URL: http://wiki.ros.org/Documentation.

[30] Adrian Rosebrock. *Credit card OCR with OpenCV and Python*. Accessed: 2023-05-16. URL: https://pyimagesearch.com/2017/07/17/credit-card-ocr-with-opencv-and-python/.

[31] Philipp Sebastian Ruppel. *Performance optimization and implementation of evolutionary inverse kinematics in ROS*. MA thesis. University of Hamburg, 2017.

[32] Baoguang Shi, Xiang Bai, and Cong Yao. *An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition*. 2015.

[33] Baoguang Shi et al. *Robust Scene Text Recognition With Automatic Rectification*. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[34] Matteo Sostero. *Automation and Robots in Services: Review of Data and Taxonomy*. JRC Working Papers on Labour, Education and Technology 2020-14. Joint Research Centre (Seville site), Dec. 2020.

[35] Satoshi Suzuki and Keiichi Abe. *Topological structural analysis of digitized binary images by border following*. In: *Computer Vision, Graphics, and Image Processing* 30.1 (1985), pp. 32–46.

[36] Tesseract. *Tesseract OCR*. Accessed: 2023-05-15. URL: https://github.com/tesseract-ocr/tesseract.

[37] Tao Wang et al. *End-to-end text recognition with convolutional neural networks*. In: *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*. 2012, pp. 3304–3308.

[38] Fabian Wieczorek et al. *Trixi the Librarian*. 2022.

[39] Williem et al. *Fast and Robust Perspective Rectification of Document Images on a Smartphone*. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2014, pp. 197–198.

[40] William Winkler. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. In: *Proceedings of the Section on Survey Research Methods* (Jan. 1990).

[41] Jan Zdenek. *CRNN*. Accessed: 2023-05-15. URL: https://github.com/janzd/CRNN.

[42] Zhong-Qiu Zhao et al. *Object Detection with Deep Learning: A Review*. 2019.