Traffic sign detection and classification

Bruno Sousa - up201604145,
João Gonçalves - up201604245,
Pedro Silva - up201604470,

FEUP,
MIEIC - [EIC0104] Visão por Computador,
10/04/2020

Introduction

In this report we will talk about an application that we developed, which is able to detect traffic signs based on their color and shape. The detection of the traffic signs will happen on a road image acquired by using a computer connected camera, or by the selection of a pre-acquired image. Experimental results show the effectiveness of our approach.

The road signs to be detected will be the ones used in Portugal, and these can be divided in classes, those being: red circles, red triangles, blue circles, blue squares, yellow squares and red octagonals (stop sign).

To run this application (while in the project folder) on a pre-acquired image in the “res” folder named “multiple”, the command “python src/project.py res/multiple.jpg” needs to be executed. To run the application on a image acquired by using a computer connected camera, execute the command “python src/project.py”.

Process

To achieve the most accurate results possible, there were various techniques applied to the image in order to remove the noise and better seperate the colors in the image.

Our project is divided in very specific stages: reader, preprocessor, detector and printer. In the **reader** we collect the image, we then process it in the **preprocessor**. The result of the preprocessor is given to the **detector** which finds which signs there are in each image and stores that information in an object. That object is finally passed to our **printer** which draws the contours of the signs and writes their description as well as giving some information in the terminal.

1. Get an image (reader.py)

For demonstration purposes, we will show the preprocessing of the follow image (Fig 1)

This image was chosen because of its shape and color diversity.

2. Color Smoothing and Shape simplification (preprocessor.py)

In order to get the best results, we needed to preprocess the image in order to get rid of some background the noise and simplify the image as much as possible for our algorithm to work better. To get this result:

- Transformed the image from RGB to HSV
- Using a predefined range, filtered out the colors we did not want to have and standardized the colors used.
- Applied a small **dilate/erode** followed by a small **erode/dilate**. The first part is used to **consolidate colors** that were just barely out of scope of out filter and therefore may have some small inconsistencies. The second part is used to **remove small, isolated points of noise.**
- Afterwards, for each color of interest, each block of color is separated from the rest of the image using floodfill. This is used to isolate each sign.

- After having a **single sign** of a different color, it is then placed in a frame (a big, empty image) in order to perform a big **dilate/erode** sequence in order to **fill the interior of the sign**. This dilate/erode is done separately in a frame big enough not to distort the signs.
- When the previous step is done for each color, all the signs are merged into one image with a bitwise_or. This operation allows us to disregard signs that have other colors inside since their insides will become a color that is not regarded.

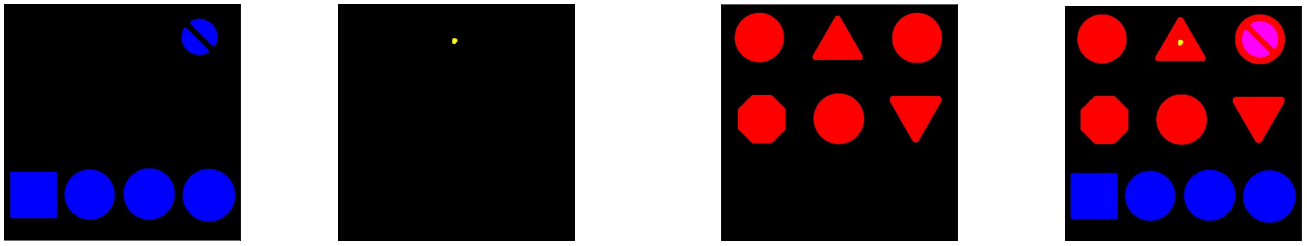


Fig2. Preprocessing step 1. Blue(1), Yellow(2) and Red(3) channel and final result of step 1 (4)

3. Shape and channel definition (preprocessor.py)

In the final step of the preprocessing, the shapes are read from the final result of step 1. (Fig2.4) During the second step, the image will be separated in the three different channels: red, blue and yellow; as well as per sign. The result will be an object mapping the color to an array of signs of said color. Furthermore, a **large dilute/erode pair is applied to each separate sign**, the goal of this operation is to completely fill and potentially recreate parts of the sign that may have been partially occluded. For this example, the blue inside the third sign will be omitted since it became lilac with the bitwise operation. The next image **show the top three signs** after being separated from the rest signs and colors and then applied the dilute/erode operation:

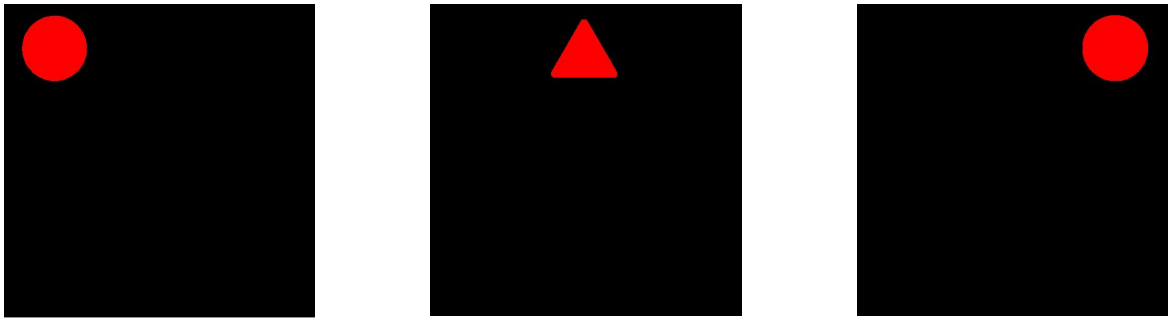


Fig 3. After the operations all the signs are uniform, separated in their own frame, and possibly reconstructed from occlusion.

NOTE: It may seem that the yellow in the triangle will be saved but it won't. Since it was created from a bitwise operation between red and yellow, it may remain yellow but it will be a different yellow and thus disregarded. The colors were standardized in Step 1 which allows us to do this.

4. Sign detection (detector.py)

In the detection part, each frame, each with one sign, will be searched for contours and their shape discovered. This is done using the "findContours" OpenCV function to discover the contours and the "approxPolyDP" one to detect polygons based on those contours. After that, the polygons are classified by the number of their sides: 3 for triangles, 4 for rectangles, 8 for STOP and more than 8 for circles. In the end, the area of each polygon is calculated to remove small polygons, whose area is smaller than a easily configurable percentage of the total image area, that are mostly false positives. While this last step may remove some smaller signs, we felt it was a good trade-off, because most signs are identified at the current percentage.

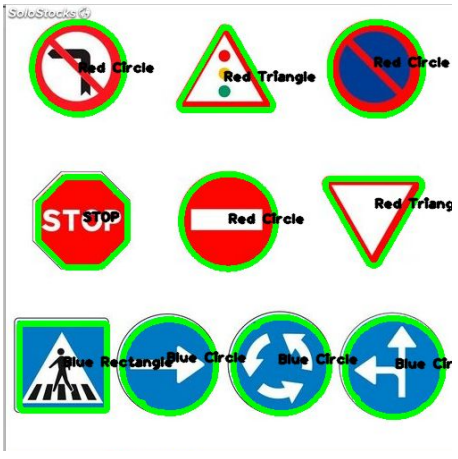
Depending on the type of shape that is being looked for, the operation that finds the contours of the image and the precision of the "approxPolyDP" function will vary.

5. Print the results (printer.py)

After the **detector** finds all the signs in the various frames, the printer takes that information and prints the contours and the information for each sign in the original image

In this example, there were detected (from left to right, top to bottom):

- Red Circle, Red Triangle, Red Circle
- Stop, Red Circle, Red Triangle
- Blue Rectangle, Blue Circle (x3)



Additional Improvements

The program also **detects Stop signs** as the previous example shows as well as **multiple signs in the same image.**

In order to read **signs in the shade**, we define each color (red, yellow and blue) as **intervals** in order to get multiple shades of each color and therefore reach signs that are in the shade or in the light.

Furthemore, our algorithm detects both **yellow signs** even if the optical axis of the **camera is not perpendicular** to the sign plane and even in **real situations** with other **objects in the background**. Finally, our algorithm allows us to detect **partially occluded** signs since it tries to autocomplete them in case of obstruction. The intermediate prints for the occlusion example can be found in Annex 1.

Conclusions

We developed an efficient application with structured code that is easy to maintain and continue to work on. For example, to add the ability to read yellow signs, we only had to define the interval for the color yellow.

We encountered some difficulties during the developing of this project, most of them were when we had to decide which is more important, to have wider color ranges and get more shading examples or narrow the intervals to reduce noise of the image. Furthermore, we had problems using the HoughCircles algorithm; we started our project, however, we found that the tuning variables were way too specific for each image and therefore was not a good algorithm for a generic project like ours.

In a future elaboration of this work, we would like it to improve its run speedy in order to make it viable for real time applications. Due to the number of dilute/erode operations in big frames in order to keep the signs shape, there is some overhead since the program has to create, for each sign, a frame, copy it to the frame, make the operations and then copy the sign back.

Finally, we are proud that we have managed to create an application that is able to fulfill not only the base requirements but all of the additional ones while still keeping the project as manageable and organized as possible, all while learning a new programming language.

References

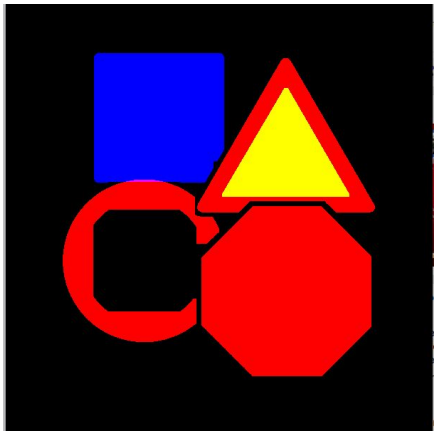
[1] “OpenCV shape detection“, https://www.pyimagesearch.com/2016/02/08/opencv-shape-detection/

[2] “Color Detection and Segmentation with OpenCV”, https://www.learnopencv.com/invisibility-cloak-using-color-detection-and-segmentation-with-opencv/

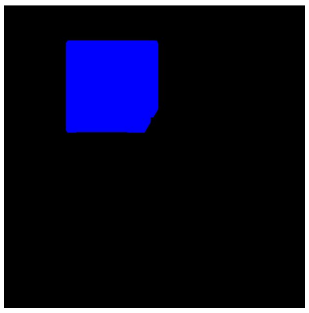
[3] “Traffic Signs: Portugal”, https://traffic-rules.com/en/portugal/traffic-signs/warning

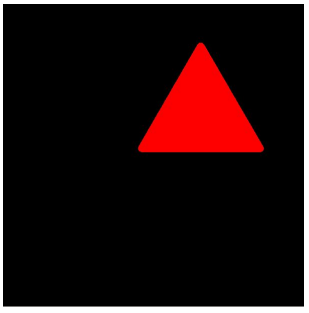
Appendix 1

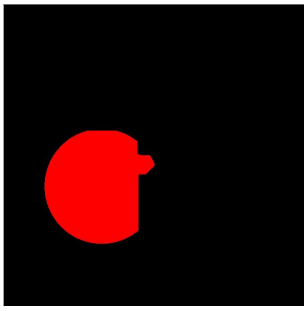
Original image and final of First Process:

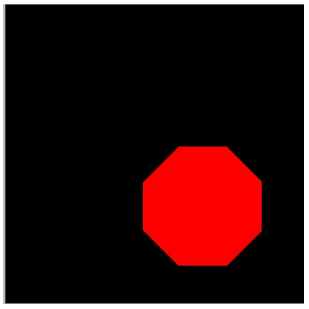


Final of Second Process:









Final Result:

Appendix 2

Interesting result from live capture with a computer camera:

Appendix 3

project.py:

```
import cv2
import sys
import numpy as np

# Local imports
from utils import *
from detector import Detector
from printer import Printer
from reader import Reader
from preprocessor import Preprocessor

# Read the image
reader = Reader()
img = reader.getImage()
original = img.copy()

# Preprocess the image
pre = Preprocessor(img)
img = pre.getProcessed()
coloredSigns = pre.getLists()

# Detect image Points of Interest (POI)
det = Detector(img, coloredSigns)
det.detectCircles("Blue")
det.detectCircles("Red")
det.detectRectangles("Blue")
det.detectTriangles("Red")

det.detectRectangles("Yellow")
det.detectStop()
ans = det.getDetected()

# Print the Detected POI's into the image
printer = Printer(original)
printer.printToSTDOUT(det.getDetectedSigns())
original = printer.printAllIntoImage(ans)
printer.showAndSave('output.png')
```

reader.py:

```
import cv2
import sys
import numpy as np

class Reader:

    """Reads an image from the user
    """

    def openImage(self, name):
        return cv2.imread(name,cv2.IMREAD_COLOR)

    def takeImageFromCamera(self):
        cap = cv2.VideoCapture(0, cv2.CAP_DSHOW)
        while(True):
            ret, frame = cap.read()
            img = frame
            cv2.imshow('Camera',img)
            if cv2.waitKey(int(1000/FPS)) != -1:
                break
        cap.release()
        cv2.destroyAllWindows()
        return img

    def getImage(self):
        if len(sys.argv) > 1:
            return self.openImage(sys.argv[1])
        else:
            return self.takeImageFromCamera()
```

preproceessor.py:

```
import cv2
import numpy as np

from utils import *

class Preprocessor:
    """ This class preprocesses the image by removing noise and simplifying the colors
    In a first step, the image is processed sign by sign with the intent to fill the sign making it simpler and clean some noise in the image.
    In a second step, the signs are already well defined. There will be a dilute/erode pair that will fill the signs as much as possible.
        The objective of this step is to interpret partially ocluded signs since they will gain shape.

    Attributes:
        img (image): Image that is going to get processed
    """

    """
    Returns the results obtained from the processing
    """
    def getLists(self):
        return {
            "Blue": self.blueList,
            "Red": self.redList,
            "Yellow": self.yellowList
        }

    """
    Returns the cleaned image
    """
    def getProcessed(self):
        return self.processedImg

    """
    Deletes the little noise in the image
    """
    def cleanImage(self, img):
        kernel1 = np.ones((5,5),np.uint8)
        img = cv2.dilate(img,kernel1,iterations = 1)
        img = cv2.erode(img,kernel1,iterations = 1)
        kernel2 = np.ones((5,5),np.uint8)
        img = cv2.erode(img,kernel2,iterations = 1)
        img = cv2.dilate(img,kernel2,iterations = 1)
        return img

    """
    Finds all the signs and executes the processing in the signs
    """
    def processSingleSign(self, img, val, x , y):
        h = img.shape[0]
        w = img.shape[1]
        temp = img.copy()
        cv2.floodFill(temp, None, (x,y),(0,255,0))
        temp = removeAllButOneColor(temp, "Green")
        x_offset = y_offset = val
        frame = np.zeros([h + y_offset*2, w + x_offset*2,3],dtype=np.uint8)
        frame[y_offset:y_offset+temp.shape[0], x_offset:x_offset+temp.shape[1]] = temp
        kernel = np.ones((val,val),np.uint8)
        frame = cv2.dilate(frame,kernel,iterations = 1)
```

```
        frame = cv2.erode(frame,kernel,iterations = 1)
        return frame[y_offset:y_offset+h, x_offset:x_offset+w]

    """
    Separates each sign according to its color and if the image needs to be cleaned or not
    An image needing to be cleaned means there is noise in the image that needs to be taken into account
    If an image does not need to be cleaned, then the erode/dilute will be bigger in order to fill the images
    """
    def processElements(self, color, clean = True):
        if clean:
            startImg = self.img
        else:
            startImg = self.processedImg
        img = removeAllButOneColor(startImg, color)
        everySign = []
        h = img.shape[0]
        w = img.shape[1]
        if clean:
            img = self.cleanImage(img)
            val = 100 # Smaller value to deal with the noise
        else:
            val = 300 # Bigger value to fill the signs properly

        # Look for signs
        for y in range(0, h):
            for x in range(0, w):
                if img[y][x][0] == RBG_PURE_COLOR[color][0] and img[y][x][1] == RBG_PURE_COLOR[color][1] and img[y][x][2] ==
RBG_PURE_COLOR[color][2]:
                    singleSign = self.processSingleSign(img, val, x, y)
                    everySign.append(singleSign)
                    cv2.floodFill(img, None, (x,y),(0,0,0))
        # Save computed results
        self.saveResuts(clean, img, everySign, color)

    """
    Saves results in everySign according to the 'clean' and 'color' settings
    """
    def saveResuts(self, clean, img, everySign, color):
        h = img.shape[0]
        w = img.shape[1]
        finalImgHSV = np.zeros([h,w,3], dtype=np.uint8)
        if clean:
            for singleSign in everySign:
                img = convertToHSV(singleSign)
                green_mask = create_mask(img, ["Green"])
                cv2.bitwise_or(finalImgHSV, img, finalImgHSV, mask=green_mask)

            finalImg = convertToRGB(finalImgHSV)
            finalImg[np.where((finalImg==RBG_PURE_COLOR["Green"]).all(axis=2))] = RBG_PURE_COLOR[color]
            if color == "Red":
                self.redProcessed = finalImg.copy()
            elif color == "Blue":
                self.blueProcessed = finalImg.copy()
            else:
                self.yellowProcessed = finalImg.copy()
        else:
            for i in range(len(everySign)):
                everySign[i][np.where((everySign[i]==RBG_PURE_COLOR["Green"]).all(axis=2))] = RBG_PURE_COLOR[color]
            if color == "Red":
                self.redList = everySign
            elif color == "Blue":
                self.blueList = everySign
```

```
        else:
            self.yellowList = everySign

    """
    Constructs and starts the preprocessing
    """
    def __init__(self, img):
        self.img = img
        h = img.shape[0]
        w = img.shape[1]
        self.blueList = []
        self.redList = []
        self.yellowList = []
        print("[PREPROCESSING] Cleaning the image")
        self.processElements('Red')
        self.processElements('Blue')
        self.processElements('Yellow')
        temp = cv2.bitwise_or(self.blueProcessed, self.redProcessed)
        temp = cv2.bitwise_or(temp, self.yellowProcessed)
        self.processedImg = temp.copy()
        self.print2(self.processedImg)
        print("[PREPROCESSING] Increasing quality of signs")
        self.processElements('Red', False)
        self.processElements('Blue', False)
        self.processElements('Yellow', False)

    """
    Displays the images in the array for debugging purposes
    """
    def printArray(self, array):
        for pic in array:
            self.print2(pic)

    """
    Displays an image for debugging purposes
    """
    def print2(self, img):
        cv2.imshow("Signs", img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
```

detector.py:

```
import cv2
import numpy as np

# Local imports
from utils import *

# Percentage of image size signs have to be to be considered
MINIMUM_SIGN_SIZE = 0.001

class Detector:
    """ Detetor runs shape detection algorithms to find signs in the image and saves it in 'detected'

    This class can detect:
        - cicles:     detectCircles(color)
        - triangles:  detectTriangles(color)
        - rectangles: detectRectangles(color)
        - stop:       detectStop()
    Attributes:
        img (image): Image where the detection algorithms will be run
        detected (:obj: type of signal -> information for the signal): The information of the image will be kept in this data structure and
can then be exported for further use
    """

    """
    Detect all the circles of the image
    """
    def detectCircles(self, color):
        circlesObj = self.getDefaultObj(color + " Circle")
        self.detected["c-" + color] = circlesObj
        for img in self.arrays[color]:
            self.detectCirclesInOneImage(color, img)
    """
    Detect all the triangles of the image
    """
    def detectTriangles(self, color):
        trianglesObj = self.getDefaultObj(color + " Triangle")
        self.detected["t"] = trianglesObj
        for img in self.arrays[color]:
            self.detectTrianglesInOneImage(color, img)
    """
    Detect all the rectangles of the image
    """
    def detectRectangles(self, color):
        rectanglesObj = self.getDefaultObj(color + " Rectangle")
        self.detected["r-" + color] = rectanglesObj
        for img in self.arrays[color]:
            self.detectRectanglesInOneImage(color,img)
    """
    Detect all the stops in the image
    """
    def detectStop(self):
        stopsObj = self.getDefaultObj("STOP")
        self.detected["STOP"] = stopsObj
        for img in self.arrays["Red"]:
            self.detectStopInOneImage(img)

    def __init__(self, img, arrayImg):
        self.arrays = arrayImg
```

```
    self.img = img
    h = img.shape[0]
    w = img.shape[1]
    self.minimumSignSize = MINIMUM_SIGN_SIZE * h * w
    print("Minimum Sign Size: " + str(self.minimumSignSize))
    self.detected = {}
    self.detectedSigns = []

def getDetected(self):
    return self.detected

def getDetectedSigns(self):
    return self.detectedSigns

"""
Prepare the image for detection
"""
def prepareImg(self, color, img):
    img = removeAllButOneColor(img,color)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    contours, _ = cv2.findContours(gray, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    return contours

"""
Construct and return a default obj
"""
def getDefaultObj(self, text):
    return {
        "info": [],
        "coordText": [],
        "text": text
    }

"""
Detect all the circles of a color in one image
"""
def detectCirclesInOneImage(self, color, img):
    contours = self.prepareImg(color, img)
    circles = []
    centers = []
    for cnt in contours:
        approx = cv2.approxPolyDP(cnt, 0.01*cv2.arcLength(cnt, True), True)
        if len(approx) > 8:
            circle = []
            for coords in approx:
                circle.append((coords[0][0],coords[0][1]))
            if calculateArea(circle) >= self.minimumSignSize:
                center = getCenter(circle)
                centers.append(center)
                circles.append([cnt])
                self.detectedSigns.append({
                    "name": self.detected["c-" + color]["text"],
                    "sign": circle
                })
    self.detected["c-" + color]["info"].append(circles)
    self.detected["c-" + color]["coordText"].append(centers)
    return contours, circles

"""
Detect all the triangles of a color in one image
"""
def detectTrianglesInOneImage(self, color, img):
```

```
    contours = self.prepareImg(color, img)
    triangles = []
    centers = []
    for cnt in contours:
        approx = cv2.approxPolyDP(cnt, 0.04*cv2.arcLength(cnt, True), True)
        if len(approx) == 3:
            triangle = [(approx[0][0][0],approx[0][0][1]),
                        (approx[1][0][0],approx[1][0][1]),
                        (approx[2][0][0],approx[2][0][1])]
            if calculateArea(triangle) >= self.minimumSignSize:
                center = getCenter(triangle)
                centers.append(center)
                triangles.append([cnt])
                self.detectedSigns.append({
                    "name": self.detected["t"]["text"],
                    "sign": triangle
                })
    self.detected["t"]["info"].append(triangles)
    self.detected["t"]["coordText"].append(centers)
    return contours, triangles

"""""
Detect all the rectangles of a color in one image
"""""
def detectRectanglesInOneImage(self, color, img):
    contours = self.prepareImg(color, img)
    rectangles = []
    centers = []
    for cnt in contours:
        approx = cv2.approxPolyDP(cnt, 0.04*cv2.arcLength(cnt, True), True)
        if len(approx) == 4:
            rectangle = [(approx[0][0][0],approx[0][0][1]),
                         (approx[1][0][0],approx[1][0][1]),
                         (approx[2][0][0],approx[2][0][1]),
                         (approx[3][0][0],approx[3][0][1])]
            if calculateArea(rectangle) >= self.minimumSignSize:
                center = getCenter(rectangle)
                centers.append(center)
                rectangles.append([cnt])
                self.detectedSigns.append({
                    "name": self.detected["r-" + color]["text"],
                    "sign": rectangle
                })
    self.detected["r-" + color]["info"].append(rectangles)
    self.detected["r-" + color]["coordText"].append(centers)
    return contours, rectangles

"""""
Detect all the stops in one image
"""""
def detectStopInOneImage(self, img):
    contours = self.prepareImg("Red", img)
    stops = []
    centers = []
    for cnt in contours:
        approx = cv2.approxPolyDP(cnt, 0.01*cv2.arcLength(cnt, True), True)
        if len(approx) == 8:
            stop = [(approx[0][0][0],approx[0][0][1]),
                    (approx[1][0][0],approx[1][0][1]),
                    (approx[2][0][0],approx[2][0][1]),
                    (approx[3][0][0],approx[3][0][1]),
                    (approx[4][0][0],approx[4][0][1]),
```

```
                (approx[5][0][0],approx[5][0][1]),
                (approx[6][0][0],approx[6][0][1]),
                (approx[7][0][0],approx[7][0][1])]
        if calculateArea(stop) >= self.minimumSignSize:
            center = getCenter(stop)
            centers.append(center)
            stops.append([cnt])
            self.detectedSigns.append({
                "name": self.detected["STOP"]["text"],
                "sign": stop
            })
    self.detected["STOP"]["info"].append(stops)
    self.detected["STOP"]["coordText"].append(centers)
    return contours, stops
```

printer.py:

```
import cv2
import sys
import numpy as np

# Local Imports
from utils import *

class Printer:
    """The Printer allows the user to print information on top of an image or to STDOUT

    The user can 'printLabels' and 'printShapes' on the image

    Attributes:
        img (image): The image where the contents should be printed
    """

    def __init__(self, img):
        self.img = img
    """
    Print all the labels into the image
    """
    def printLabels(self, img, obj):
        # Constants
        font = cv2.FONT_HERSHEY_PLAIN
        fontScale = 1
        color = (0, 0, 0)
        thickness = 2

        # Personalized for each shape
        textToPrint = obj["text"]
        coords = obj["coordText"]
        for coord in coords:
            if len(coord) > 0:
                img = cv2.putText(img, textToPrint, coord[0], font, fontScale, color, thickness, cv2.LINE_AA)
        return img
    """
    Print all the shapes into the image
    """
    def printShapes(self, img, obj):
        text = obj["text"]
        for sign in obj["info"]:
            if len(sign) > 0:
                cv2.drawContours(img, sign[0], 0, (0, 255, 0), 6)
        return img
    """
    Print all the information from 'answer' to the image
    """
    def printAllIntoImage(self, answer):
        print("[PRINTER] Preparing the final image")
        for signType in answer:
            obj = answer[signType]
            self.img = self.printShapes(self.img, obj)
            self.img = self.printLabels(self.img, obj)
        return self.img

    def printToSTDOUT(self, answer):
        for sign in answer:
            print(sign["name"] + ": " + str(sign["sign"]))
```

```
    """
    Show the image and print it in nameImage
    """
    def showAndSave(self, nameImage):
        cv2.imshow("Signs", self.img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
        cv2.imwrite(nameImage, self.img)
```

utils.py:

```
import cv2
import numpy as np

FPS = 60

"""
Convert the image from RGB to HSV
"""
def convertToHSV(img):
    return cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

"""
Convert the image from HSV to RGB
"""
def convertToRGB(img):
    return cv2.cvtColor(img, cv2.COLOR_HSV2BGR)

# RBG pure colors for comparison
RBG_PURE_COLOR = {
    "Red": [0,0,255],
    "Blue": [255,0,0],
    "Yellow": [0, 255, 255],
    "White": [255,255,255],
    "Green": [0,255,0]
}

# HSV pure colors for comparison
HSV_PURE_COLOR = {
    "Red": [0,100,255],
    "Blue": [255,0,0],
    "Yellow": [0, 255, 255],
    "Green": [120,100,100]
}

# HSV ranges for some possible colors of signs
HSV_RANGES = {
    # red is a major color
    'Red': [
        {
            'lower': np.array([0, 200, 100]),
            'upper': np.array([5, 255, 255])
        },
        {
            'lower': np.array([175, 200, 100]),
            'upper': np.array([180, 255, 255])
        }
    ],
    # yellow is a minor color
    'Yellow': [
        {
            'lower': np.array([15, 128, 230]),
            'upper': np.array([33, 255, 255])
        }
    ],
    # green is a major color
    'Green': [
        {
            'lower': np.array([41, 39, 64]),
```

```
            'upper': np.array([80, 255, 255])
        }
    ],
    # cyan is a minor color
    'Cyan': [
        {
            'lower': np.array([81, 39, 64]),
            'upper': np.array([100, 255, 255])
        }
    ],
    # blue is a major color
    'Blue': [
        {
            'lower': np.array([100, 200, 64]),
            'upper': np.array([141, 255, 255])
        }
    ],
    # violet is a minor color
    'Violet': [
        {
            'lower': np.array([141, 39, 64]),
            'upper': np.array([160, 255, 255])
        }
    ],
    # next are the monochrome ranges
    # black is all H & S values, but only the lower 25% of V
    'Black': [
        {
            'lower': np.array([0, 0, 0]),
            'upper': np.array([180, 255, 63])
        }
    ],
    # gray is all H values, lower 15% of S, & between 26-89% of V
    'Gray': [
        {
            'lower': np.array([0, 0, 64]),
            'upper': np.array([180, 38, 228])
        }
    ],
    # white is all H values, lower 15% of S, & upper 10% of V
    'White':    [
        {
            'lower': np.array([0, 0, 150]),
            'upper': np.array([180, 38, 255])
        }
    ]
}

"""
Creates a binary mask from HSV image using given colors.
"""
def create_mask(hsv_img, colors):
    mask = np.zeros((hsv_img.shape[0], hsv_img.shape[1]), dtype=np.uint8)

    for color in colors:
        for color_range in HSV_RANGES[color]:
            mask += cv2.inRange(
                hsv_img,
                color_range['lower'],
                color_range['upper']
            )
```

```
    return mask

def print2(img):
        cv2.imshow("Signs", img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

# Removes all the colors except for the one given
def removeAllButOneColor(img, color):
    img = convertToHSV(img)
    colored_mask = create_mask(img, [color])
    mask_img = cv2.bitwise_and(img, img, mask=colored_mask)
    mask_img = convertToRGB(mask_img)
    # print2(mask_img)
    h = mask_img.shape[0]
    w = mask_img.shape[1]
    for y in range(0, h):
        for x in range(0, w):
            if mask_img[y][x][0] != 0 or mask_img[y][x][1] != 0 or mask_img[y][x][2] != 0:
                mask_img[y][x] = RBG_PURE_COLOR[color]
    return mask_img

# Finds center of shape
def getCenter(shape):
    center = [ int(sum(x) / len(shape)) for x in zip(*shape) ]
    return (center[0], center[1])

# Calculates the area of a shape
def calculateArea(shape):
    soma = 0
    for i in range(len(shape)):
        if i == len(shape) - 1:
            soma += (shape[i][0] * shape[0][1]) - (shape[i][1] * shape[0][0])
        else:
            soma += (shape[i][0] * shape[i+1][1]) - (shape[i][1] * shape[i+1][0])
    return abs(soma) / 2
```