2024

# Computer Vision : Project Report

LOUIS JARDINET 195030

# Table of Contents

# 1. Introduction

During the course of Mr Delhaye titled “Computer Vision 2” we were given the task of implementing a small image processing project. I found it an interesting challenge to recognize patterns, shapes and colors in an image. At the beginning, I wanted to implement the detection of Waldo in the famous book-game ‘Where's Waldo’. However, I ended up implementing a way of cheating on the Ishihara test, which seemed more complete and gave me more the impression that I had the right tools in hand from the start.

The Ishihara test was developed in 1917 by Shinobu Ishihara to detect color blindness. It consists of dots placed randomly in a circle, some of which are colored in the shape of a number. If the patient is unable to identify which number is being displayed, then he or she probably suffers from color blindness. My code therefore had to be able to show the number on the image and to indicate which number corresponded to the shape on the image, both on a sample recorded in a file and shown on a webcam.

# 2. Key Challenges

Here are the key challenges I had to overcome to successfully implement this project:

a) Color detection : The code has to highlight predefined colors from determined color codes
b) Color identification : The code has to differentiate the color of the number from the color of the background.
c) Number detection : The code retrieves the number shown in the image
d) Ishihara test detection on a live feed : The aforementioned steps work and the number is retrieved when the test is showed on a live feed

# 3. Strategy of implementation

## a. Color Detection :

This step was one for which the laboratories gave all the keys. To retrieve the color intervals, I used a ChatGPT prompt. I created a dictionary for each color with their lower and upper intervals of HSV colors. As we have seen during laboratory 2, the input image is first converted in the HSV color space, then a mask is created for each color using cv2.inRange() function. The colors were arbitrarily chosen.

```python
def highlight_colors(image):

    # Convert the frame to HSV color space
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define HSV ranges for the desired colors
    color_ranges = {
        "red": [(0, 50, 50), (10, 255, 255)],
        "blue": [(100, 50, 50), (140, 255, 255)],
        "green": [(40, 50, 50), (80, 255, 255)],
        "yellow": [(20, 50, 50), (30, 255, 255)],
        "purple": [(130, 50, 50), (160, 255, 255)],
        "orange": [(10, 50, 50), (20, 255, 255)],
        "rose": [(160, 50, 50), (170, 255, 255)]
    }
    # Create masks for each color
    masks = {}
    for color, ranges in color_ranges.items():

        lower, upper = ranges
        masks[color] = cv2.inRange(hsv, np.array(lower), np.array(upper))

    # Display each color mask separately for visualization
    for color, mask in masks.items():
        cv2.imshow(f"{color} Mask", mask)


    # Break the loop on 'q' key press
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

*Figure 1 : Implementation of the function to highlight colors*

## b. Color Identification

This was the most challenging step as I had to use K-means clustering, which I was not very familiar with. Cluster analysis is an unsupervised learning technique which consists in regrouping data points into groups called clusters. The data is similar within a cluster and different between each clusters.

First, my approach was to retrieve the color palette using K-means clustering and then identify the 2 most distant color using the Euclidean distance between the RGB codes of the colors present on the picture. The two colors with the most distant color codes were thus computed and tolerance was applied on the color codes in order to separate all green circles from red circles, for example, whether the color is light or dark. A filter was also applied to remove the white background. Eventually, I found out it was not the best approach as the most distant colors were often little represented (see on figure 2) and because they were extreme RGB values, the tolerance applied to retrieve an interval of color of all green circles was high. A bigger interval means more errors and thus the strategy was not suitable.

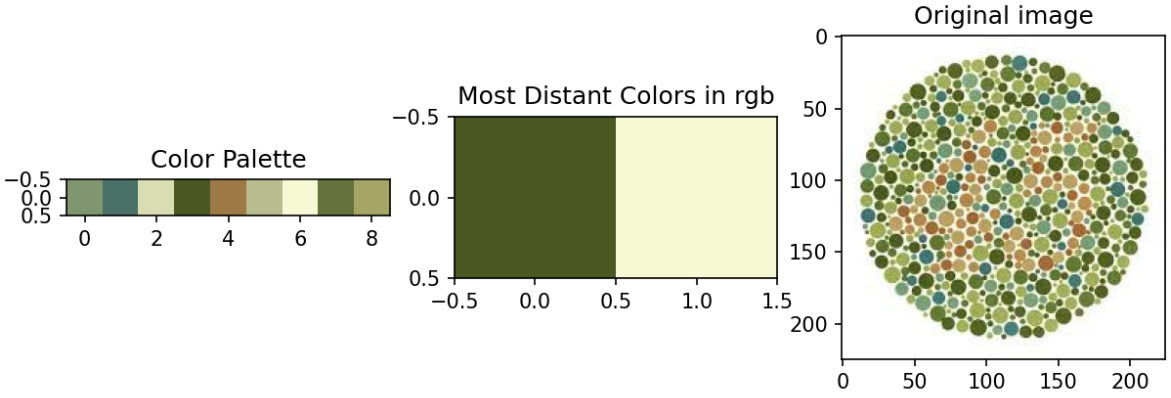


*Figure 2 : First implementation output in order to retrieve the color values*

After (quite a few) more tries with different ways to compute the distance between color codes, I came across a website in which they had implemented the same project. In this site was a comparison between color spaces and it clearly highlighted that working in the RGB color space didn't yield the best results as compared to the YCrCb color space.[1]

Also, I dug more into the concept of clustering. I realized that instead of taking the most extreme values out of a lot of clusters, I should only take 2 clusters as the center number would most probably be one separate cluster. It seems evident as I am writing these lines, but it wasn't for me at first.

My final working strategy was therefore:

1. Retrieve the red Chrominance channel
2. Prepare this data to be suitable for k-means operation
3. Apply K-means on the data with 2 clusters
4. Sort the pixels by cluster label : 0 or 1
5. Retrieve the black or white value as it is 255 * cluster label

Here is the implemented function :

[1] **Syed, N. R.** (2018, March 25). *Image Segmentation via K-Means Clustering to Decipher Color Blindness Tests.*

```
#########################################################################################################################
#STEP 3 : Cluster the image using K means algorithm to group each pixel with their closest color

def clustering(image):
    # Convert the image from BGR to YCrCb color space. YCrCb represents Luminance, Red Chrominance component, Blue Chrominance Component
    image_ycrcb = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)

    # Extract the Cr channel (index 1) for clustering
    cr_channel = image_ycrcb[:, :, 1]

    # Flatten the 2D Cr channel into a 1D array of pixels because K-means needs 1D data.
    pixels = cr_channel.reshape(-1, 1)

    # Since the images are dichromic, we will need 2 clusters
    num_clusters = 2

    # Apply K-means algorithm clustering to group pixels into 2 clusters where each pixel will be grouped in the cluster with the closest color
    kmeans = KMeans(n_clusters=num_clusters, n_init=30, max_iter=500, random_state=42)
    kmeans.fit(pixels)

    # Retrieve the cluster labels, which is the cluster to which each pixel belongs for each pixel
    # and reshape them back into the original image dimensions
    cluster_labels = kmeans.labels_.reshape(cr_channel.shape[:2])
    labels = [0,1]
    # Create an blank output image to represent the clusters with the right dimensions
    clustered_image = np.zeros_like(cluster_labels, dtype=np.uint8)

    # Assign black or white values to clusters based on their cluster labvel.
    for i, label in enumerate(labels):
        grayscale_value = 255  * i
        clustered_image[cluster_labels == label] = grayscale_value

    return clustered_image
```

*Figure 3 : Implementation of the color separation*

To yield the best results, I also had to resize the image to enhance the resolution using the cv2.resize() function.

## c. Number detection

For the text extraction I used the Pytesseract optical character recognition tool which I previously downloaded and imported the path as “tess”. Python-tesseract is a wrapper for Google's Tesseract-OCR Engine. Determining the config was a bit tricky, I had to try a lot of different configs to determine which worked best. Here is how it works [2]:

- psm 8 treats the image as a single word. It yielded better results with images displaying single numbers
- psm 6 assumes a single uniform block of text. It yielded better results with images displaying 2 numbers.
- oem 3 specifies the OCR Engine mode to default, which is standard as I understand
- -c tessedit_char_whitelist=0123456789 is used to whitelist all numbers thus to detect numbers only

Below is my implementation :

[2] **Horvay, D.** (n.d.). *Pytesseract Function Parameters*

```python
def extract_text(image):

    # Use PyTesseract to detect numbers. psm 8 recognises best images with a single number as it treats the image as a single word
    config = "--psm 8 --oem 3 -c tessedit_char_whitelist=0123456789"
    extracted_text = tess.image_to_string(image, config=config)

    #psm 10 detects best when 2 numbers are present even though it treats the image as a single character
    if extracted_text == "":
        config_single_word = "--psm 10 --oem 3 -c tessedit_char_whitelist=0123456789"
        extracted_text = tess.image_to_string(image, config=config_single_word)
    return extracted_text
```

*Figure 4 : Implementation to retrieve the displayed numbers*

Before this step, I applied a blur on the image to smoothen it using the cv2.medianBlur() method. The kernel size was important as some information could be lost depending on it. I will go further in detail on this topic in section 4.

## d. Test on a live feed

For this step I used the circle detection method seen during the second laboratory with the HoughCircles function. I created a custom zone to work in to which I applied all previous steps.

Here is my implementation :

```python
def Ishihara_on_video():
    cam = cv2.VideoCapture(0)
    cam.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    cam.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
    while True:
        kernel_size = 31
        ret, frame = cam.read()
        # Convert to grayscale for circle detection
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        # Apply blur to reduce noise
        blur = cv2.blur(gray, (5, 5))
        # Detect circles using HoughCircles method. It returns a tuple containing the x-coordinate of the circle's center (float),
        # the y-coordinate of the circle's center (float) and the radius of the circle (float)

        circles = cv2.HoughCircles(blur, method=cv2.HOUGH_GRADIENT,dp=1,minDist=400,param1=50,param2=13,minRadius=40,maxRadius=175)
        if circles is not None:
            # Process each detected circle
            for circle in circles[0, :]:
                #Convert the x center, y center and radius to int
                x, y, radius = map(int, circle)
                #Create a custom zone to work in
                working_zone = frame[y - radius:y + radius, x - radius:x + radius]
                if working_zone.size > 0:
                    #Use all previous operations on the circle
                    resized_image = resize_image(working_zone)
                    clustered = clustering(resized_image)
                    blurred_and_clustered = blur_image(clustered, kernel_size)
                    extracted_text = extract_text(blurred_and_clustered)
                    if (extracted_text is not None):
                        print(f"Extracted Text: {extracted_text}")

                    # Draw the circle on the original frame
                    cv2.circle(frame, (x, y), radius, (0, 255, 0), 2)
                    cv2.circle(frame, (x, y), 2, (0, 0, 255), 3)
        cv2.imshow('Detected', frame)
        # Press the 'ESC' key to stop
        if cv2.waitKey(1) & 0xFF == 27:
            break
    cam.release()
```

*Figure 5 : Implement of the previous steps on a live feed*

# 4. Results

After all the calibration and optimization of parameters and configurations, the program validated 9 out of the 10 tests I did on fixed images. The kernel size of the blurring method was a determining factor: too low, it had trouble detecting the 7 or 9 as some ends did not connect. Too high, the pointy end of the "1" was smoothened and it didn't recognize it either. On image "Ishi7.png", this kernel size parameter must be lowered to around 11 for the ocr to recognize the number. The results depend thus on the calibration of this factor according to the dataset. I determined a suitable kernel size factor for most images at around 31.

As previously stated, I also added a second configuration for the number recognition as it yielded better results. One config works best for single number images and the other for dual number images.

I tried to use different samples with different variations of colors and containing 0,1 or 2 numbers.

Concerning the live feed, the number is recognized after a few errors, mostly because of the blur induced by fast movements. When still, the image is recognized as shown in the video.

Follow the links below to view the tests:

Video_Still_Images.mkv

Video_Live_feed.mkv

# 5. Conclusion

In conclusion, the recognition of numbers of various colors works mostly well on a live feed as well as with still images. To take it further, first, the model could be tuned to work with a greater dataset. Then, other problematics regarding color blindness could be approached, for example detecting the ripeness of fruits or vegetables, detecting whether a food is cooked or not or detecting whether the traffic light is green or red. These programs could also be integrated in smart glasses to help colorblind people to highlight the colors the user has the most trouble with.

# 6. Additional Deliverables

Here is the link of my Github repository.

# 7. Bibliography

1. **Horvay, D.** (n.d.). *Pytesseract Function Parameters*. Retrieved from Kaggle.

2. **Syed, N. R.** (2018, March 25). *Image Segmentation via K-Means Clustering to Decipher Color Blindness Tests*. Retrieved from nrsyed.com.
3. **Nanonets Team.** (n.d.). *OCR with Tesseract*. Retrieved from Nanonets.
4. **Brownlee, J.** (n.d.). *K-Means Clustering for Image Classification using OpenCV*. Retrieved from Machine Learning Mastery.
5. **Brownlee, J.** (n.d.). *K-Means Clustering in OpenCV and Application for Color Quantization*. Retrieved from Machine Learning Mastery.
6. **Daltonien.free.fr.** (n.d.). *Le Daltonisme - Les Tests Ishihara*. Retrieved from daltonien.free.fr.
7. **Colorlite.** (n.d.). *Test Ishihara pour le Daltonisme*. Retrieved from Colorlite.
8. **GeeksforGeeks Team.** (n.d.). *Image Resizing using OpenCV | Python*. Retrieved from GeeksforGeeks.
9. **OpenCV Documentation.** (n.d.). *Filtering in OpenCV*. Retrieved from OpenCV.
10. **Rosebrock, A.** (2014, July 21). *Detecting Circles in Images using OpenCV and Hough Circles*. Retrieved from PyImageSearch.
11. **OpenAI Chatgpt :** “Give me a complete bibliography of these websites” prompt with the previous links pasted
12. **OpenAI Chatgpt :** “Give me the hsv color intervals for these colors : red blue green yellow purple orange rose”