Université Bourgogne Franche-Comté

Master 2 in Smart Integrated Systems

Report 2D Computer vision

Grover ARUQUIPA | grover _aruquipa@femto-st.fr
MEZIANE Sameh | Sameh _Meziane@edu.univ-fcomte.fr

March 20, 2023

Introduction

Computer vision being the ability of a computer to recognize and understand what it sees, is an AI technique allowing to process visual inputs principally images and videos captured using a camera, allowing the recognition of shapes, colors and other features, interpretation and processing of the extracted data[1] [2][3]. The aim is to make decisions based on the latter inputs. This is a growing field finding applications in different sectors from research to industry and even security[4] [5].
The goal of this project is to implement a python code as introduction to machine learning specifically supervised learning, to detect shapes and colors in a video segment and show the results, the computer not knowing what a color is and only knowing RGB *(Red, Green, Blue)* or HSV *(Hue, Saturation, Value)* values.

In this work we present a computer vision application based on basic recognition techniques, We first calibrate the colors values, meaning teach the computer to interpret simple data *(HSV values)* extracted from images into intervals representing color names comprehensible to the human brain, i.e., assign each HSV interval to the corresponding color. To facilitate this process a GUI *(Graphical User Interface)* is to be implemented to choose the calibration picture containing all the colors present in the video, all the user has to do is to choose a color and the corresponding HSV intervals. All the calibration data are stored in CSV files.

Additionally, we use the OpenCV[1] library to detect the contours thus recognizing the different shapes (rectangles) present in the video to process. From which the length and width of the rectangles are extracted and used to calculate corresponding sections.
Finally, we show on the processed video, the contour of each recognized shape and print at its middle both its section, color and number.

[1] https://github.com/opencv/opencv

Implementation

Notes to run the code

- GUI.py: For the main interface or the general interface
- Main2.py: For just the detection interface
- Tool.py: For the color Calibration interface
- limits.py: For the limits calibration interface

In order to implement all the desired functionalities, the code can be mainly divided into two sections like it's observed in the Fig. 2.1, The calibration and the detection part.
The calibration data will be extracted and stored in order to be used later on in the detection part and printed on the video.

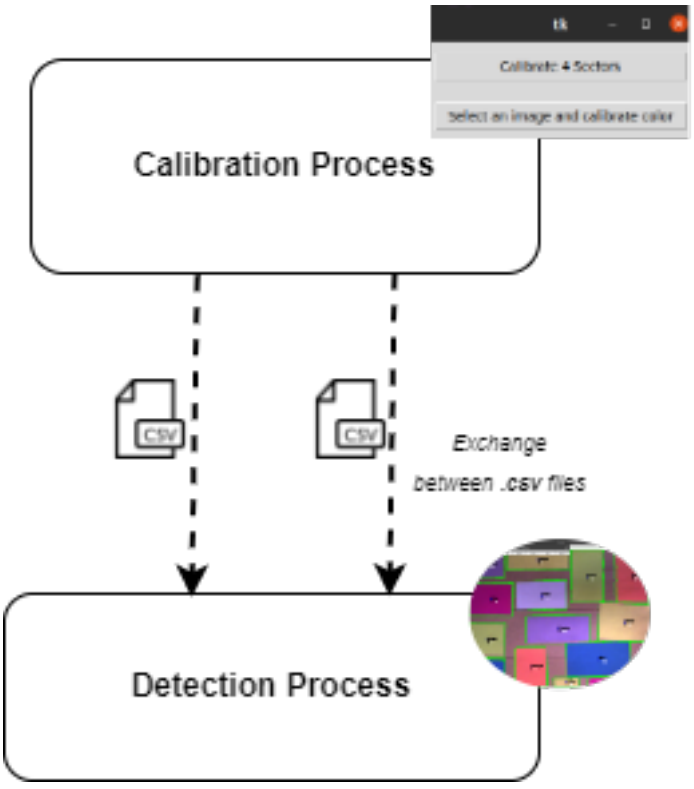


Figure 2.1: Principal process of the application.

It is necessary to mention that the exchange between these two programs is done through local variables which are stored in external memory files based on *.csv* files, which contain the calibration values of each color as is showed in the Fig.2.1.

On the other hand, for the development and implementation of this program, the numpy libraries were used, which is the fundamental python library for the manipulation of mathematical elements such as double or arrays, in the same way the opencv library was used, which entails a series of functions that simplify image processing such as morphological operations or contour detection, classification and feature extraction algorithms such as SWIFT and in the latest version basic deep-learning architectures such as YOLOv1.
In the same way, pandas was used, which is an accepted and popular library in the data science community, for the manipulation of Dataframes, growth of csv or txt files, as an extra for the development of our graphical interfaces, the use of PyQT5 was initially proposed, but due to cross-compilation problems with opencv and therefore the cross-platform problems between windows and linux, it was migrated to the use of Tkinter, which is the basic interface development tool.
Finally, the OS library is used, for the manipulation of scripts, in the same way for the display of images in tkinter, the use of the Pillow library (PIL) was chosen, which helps to manipulate different image formats and presents a compatibility stable with Tkinter.

2.1 Comparison between the HSV space and the RGB space

First of all, in order to perform an optimal color detection application, it is necessary to analyze the behavior of the frames, in many cases an image with noise and distortions is presented, which totally harms the separation and detection of colors in a space such as RGB, in order to be able to observe these characteristics as is explained in [5], it is possible to observe the image in a 3 vectors, *(H,S,V)* or *(R,G,B)*, in this way in Fig. 2.2 a correct representation is shown for a single frame.

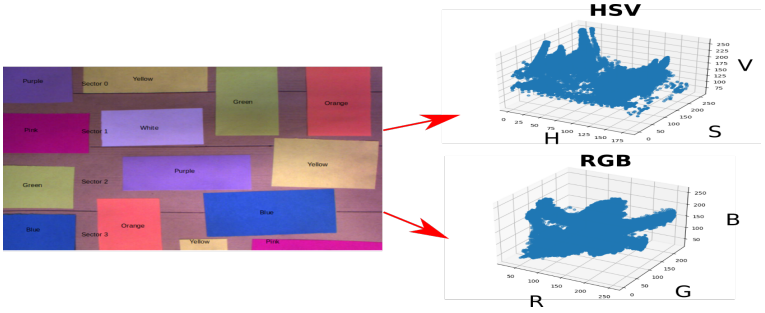


Figure 2.2: Color representation.

Thus, in Fig. 2.2 it is observed how the separation between pixels is better presented in the HSV space, for this frame which gives us an initial step to be able to work in this color field, on the other hand in the RGB image, it is more visible a single clustering, although there is a perceptible grouping, it does not show a clear separation, which consequently would make it difficult to detect colors correctly.

To better show the difference, we implemented the script *compare.py*, using matplotlib[1] library, which is a plotting library for python allowing to produce high quality figures. we scattered the RGB and HSV data to obtain the figure bellow, even though the scatter function is a bit slow when manipulating large data since matplotlib was not designed with speed in mind, other libraries where proposed but since most use either third party software*(browsers)* to plot or require installing packages, we went back to matplotlib so it should be considered that it might take few seconds to couple minutes to load. *(20s on a windows i7-11 16GB RAM machine).*

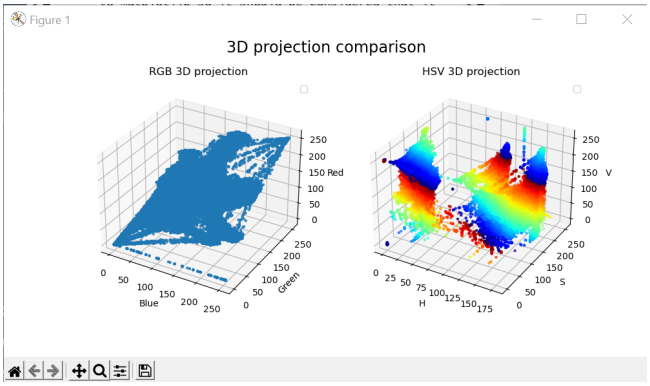


Figure 2.3: 3D Projection Domain of the Image.

In the Fig. 2.3, we can see a clear difference in data projection from RGB and HSV data, in the first one data is all cluttered in the same area. But to further analyse the data, we implemented a histogram Fig .2.4, where the data dispersion is much more obvious.

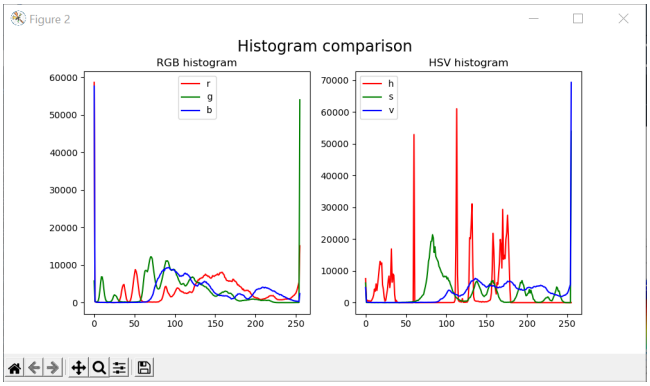


Figure 2.4: Histogram RGB vs HSV comparison

2.2 Design of the script Calibration

The color calibration is done through an interface to facilitate communication with the user instead of dealing directly with the code to extract the calibration data from the calibration image.

[1] *https : //matplotlib.org/*

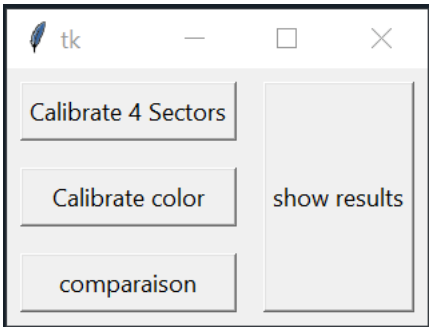


Figure 2.5: Calibration interface.

The interface contains 4 buttons, the first to calibrate the sections calling *limits.py* program, the second one calibrate color to select the calibration image calling the select-image function and do the color calibration. We can also show the results of the comparison explained in the section above calling *compare.py*. The fourth and main button is to show the final result, i.e. shape and color detection on the video with the position of each frame.

```
def limits():
    os.system('python limits.py')
def main():
    os.system('python main2.py')
def comparision():
    os.system('python comparehuh2.py')
```

The select image function in the script bellow allows to select the calibration image from the computer and process it using the script *tools.py*. which is a function to process the called image, convert it to HSV and extract calibration data of the selected color using a track-bar moving it right or left to define the Hue,Saturation, and Value values for each color interval.

```
def select_image():
    global panelA, panelB
    path = tkFileDialog.askopenfilename()
    if len(path) > 0:
        image = cv2.imread(path)
        cv2.imwrite('paperEval.png',image)
        os.system('python tool.py')
```

The human brain not being able to distinguish each value in the HSV spectrum, for each color we take an interval corresponding to the maximum and minimum values, and each color within the interval is considered the same.

2.2.1 Limits Calibration

First, the limits script allows to divide the frame into four separate sections from top to bottom. The frame segmentation will allow us to latter better track the of edge positions and thus rank the rectangles in a chronological order.

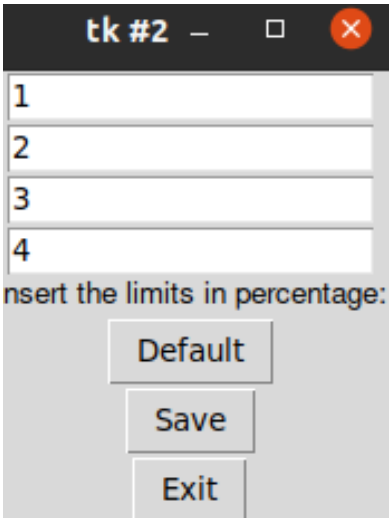


Figure 2.6: Limits Interface.

The bounds calibration script does not entail an extensive development as such, due to the basic widgets used. So, for this section, the 4 *Entry* widgets are first defined for each entry, a *Save* button to save the values, an *Exit* to exit the interface and a Default $Tosetthevaluestothedefaultlimits.$, to finally be able to save the limits in external CSV files that will be read by the recognition program.

```
self.Master=Tk()
self.Entry1=Entry(self.Master)
self.Entry1.pack()
self.Entry2=Entry(self.Master)
self.Entry2.pack()
self.Entry3=Entry(self.Master)
self.Entry3.pack()
self.Entry4=Entry(self.Master)
self.Entry4.pack()
```

Finally, in the previous lines of the program, it is shown how to instantiate these widgets for use in the interface.

2.2.2 Color Calibration

Additionally, for the calibration of colors, the attached interface was added in fig. 2.7, where you can see that a canvas was created, where through slides for the HSV values, the maximum and minimum values of each parameter are changed, on the other hand also buttons are used to save these parameters for each color.

Figure 2.7: Interface for the calibration of the colors.

The functional structure of these designs will be explained in a timely manner, in such a way as to show the totally important sections for each part of the program.
So, in order to be able to execute the interface in python with tkinter, we first define an object of type Tk, with a geometry of $800x600+300+300$ As shown below, a class that we will execute with the name of App and the *mainloop* which implies running the program continuously, as shown below.

```
root = Tk()
root.geometry("800x600")
app = App(root)
app.mainloop()
```

On the other hand, in the python execution class, we use the *init* construct, where we define the buttons, the canvas and the slides, so for the button definition an execution objective function defined by *hsvthreshold* and belonging to type frame2 as shown below, in the same way for the button defaults, which is calling the predefined values.

```
self.green = Button(self.frame2, text="Green", command=self.hsvthreshold)
self.blue = Button(self.frame2, text="Blue", command=self.hsvthreshold)
self.white = Button(self.frame2, text="White", command=self.hsvthreshold)
self.yellow = Button(self.frame2, text="Yellow", command=self.hsvthreshold)
self.orange = Button(self.frame2, text="Orange", command=self.hsvthreshold)
self.purple = Button(self.frame2, text="Purple", command=self.hsvthreshold)
self.pink = Button(self.frame2, text="Pink", command=self.hsvthreshold)
self.defv = Button(text='Default values', command=self.hsvthreshold, bg='green',
    fg='white', font=('helvetica', 9, 'bold'))
self.close = Button(text='Close', command=self.close_window, bg='brown', fg='
    white', font=('helvetica', 9, 'bold'))
```

Likewise for each slide, for the Hmax, Hmin, Smax, Smin, Vmax, and Vmin, where we need define the range of the values for each slider, as shown below.

```
self.SliderHmin = Scale(self.frame2, from_=0, to=255, orient=HORIZONTAL, command
    =self.hsvthreshold)
self.SliderHmax = Scale(self.frame2, from_=0, to=255, orient=HORIZONTAL, command
    =self.hsvthreshold)
self.SliderSmin = Scale(self.frame2, from_=0, to=255, orient=HORIZONTAL, command
    =self.hsvthreshold)
self.SliderSmax = Scale(self.frame2, from_=0, to=255, orient=HORIZONTAL, command
    =self.hsvthreshold)
```

```
self.SliderVmin = Scale(self.frame2, from_=0, to=255, orient=HORIZONTAL, command
    =self.hsvthreshold)
self.SliderVmax = Scale(self.frame2, from_=0, to=255, orient=HORIZONTAL, command
    =self.hsvthreshold)
```

Following the same procedure, It is also mentioned that a difficult section to manipulate is the Tkinter Canvas, because Tkinter is not designed directly for Computer Vision application such as PyQT5, the frame read for calibration must be transformed into a different variable, so the frame is instantiated. canvas using the following line of code:

```
self.frame1 = Frame(self)
self.original = Image.open('paperEval.png')
self.hsv = cv2.cvtColor(np.array(self.original), cv2.COLOR_RGB2HSV)
self.image = ImageTk.PhotoImage(Image.fromarray(self.hsv))
self.display = Canvas(self.frame1)
self.frame1.pack(side=LEFT, fill=BOTH, expand=1)
```

In accordance, in the code shown above, we first carry out the instance of the canvas type object, we open the image with the *open* instruction of the PIL library, on the other hand we also carry out a conversion to an array type with the *np.array* function, and the conversion to HSV, through opencv functions, in order to obtain the accepted format for canvas, which is of type $PhotoImage$ and finally we display the image with $self.display = Canvas(self.frame1)$. On the other hand, the *hsvthrtreshold* function, we mainly obtain the values of each slide with the command $self.SliderHmin.get()$ for each value, in order to create an array for the values with the commands shown below, in the same way is created a mask.

```
lower = np.array([Hmin,Smin,Vmin])
upper = np.array([Hmax,Smax,Vmax])
mask = cv2.inRange(self.hsv, lower, upper)
```

Subsequently, the image is displayed on the canvas, a size change of the image is made, in such a way as to have an acceptable format in the interface.

```
res= cv2.bitwise_and(self.bgr2,self.bgr2, mask= mask)
self.image = ImageTk.PhotoImage(Image.fromarray(res))
self.display.delete("IMG")
self.display.create_image(self.display.winfo_width()/1.2, self.display.
    winfo_height()/1.2, anchor=CENTER, image=self.image, tags="IMG")
self.resize()
self.display.pack(fill=BOTH, expand=1)
self.frame1.pack(fill=BOTH, expand=1)
```

```
    if self.pink['state'] == 'active':
            print('pink')
            #Print the values of the sliders
            dfpink.loc[0] = [Hmin, Hmax, Smin, Smax, Vmin, Vmax, 7]
            dfpink.to_csv('pink.csv', index=False)
        if self.defv['state'] == 'active':
            print('default')
            #Print the values of the sliders
            dfgreen.loc[0] = [25, 91, 75, 250, 86, 255, 3]
            dfgreen.to_csv('green.csv', index=False)
            dfblue.loc[0] = [90, 120, 50, 255, 70, 255, 0]
            dfblue.to_csv('blue.csv', index=False)
            dfwhite.loc[0] = [90, 133, 35, 115, 160, 255, 0]
```

```
            dfwhite.to_csv('white.csv', index=False)
            dfyellow.loc[0] = [11, 21, 86, 255, 132, 255, 1]
            dfyellow.to_csv('yellow.csv', index=False)
            dforange.loc[0] = [144, 179, 123, 176, 105, 255, 2]
            dforange.to_csv('orange.csv', index=False)
            dfpurple.loc[0] = [129, 140, 120, 150, 100, 255, 0]
            dfpurple.to_csv('purple.csv', index=False)
            dfpink.loc[0] = [150, 180, 180, 255, 120, 255, 5]
            dfpink.to_csv('pink.csv', index=False)
```

Contrasting, it is necessary to add conditions that show the states of 1 or 0 for each button, so that each time the button of each color is pressed, it saves the values of HSV ranges in the CSV files, in the same way. *Default* button is added, which changes the HSV data to the best found for color recognition, specifically for this image.

2.3 Design of the Detection script

The detection section was implemented through four main phases, observed in the Fig. 2.8: The extraction of the frames to convert to the HSV domain for better processing, the detection of colors using the calibration data and the shape detection after filtering to find the contours used to calculate the section and the center of the rectangles, interpretation of the extracted data and the last phase is to save all the processed data in different external files, These functions can be found in the attached *github* repository, specifically in the scripts$main2.py$[2] and $base_functions.py$[3]

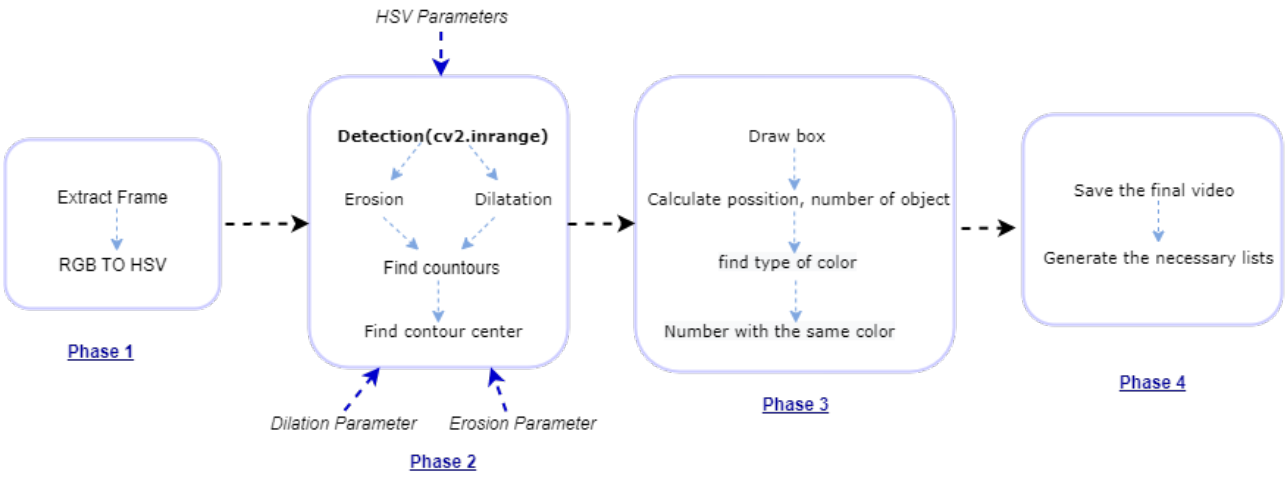
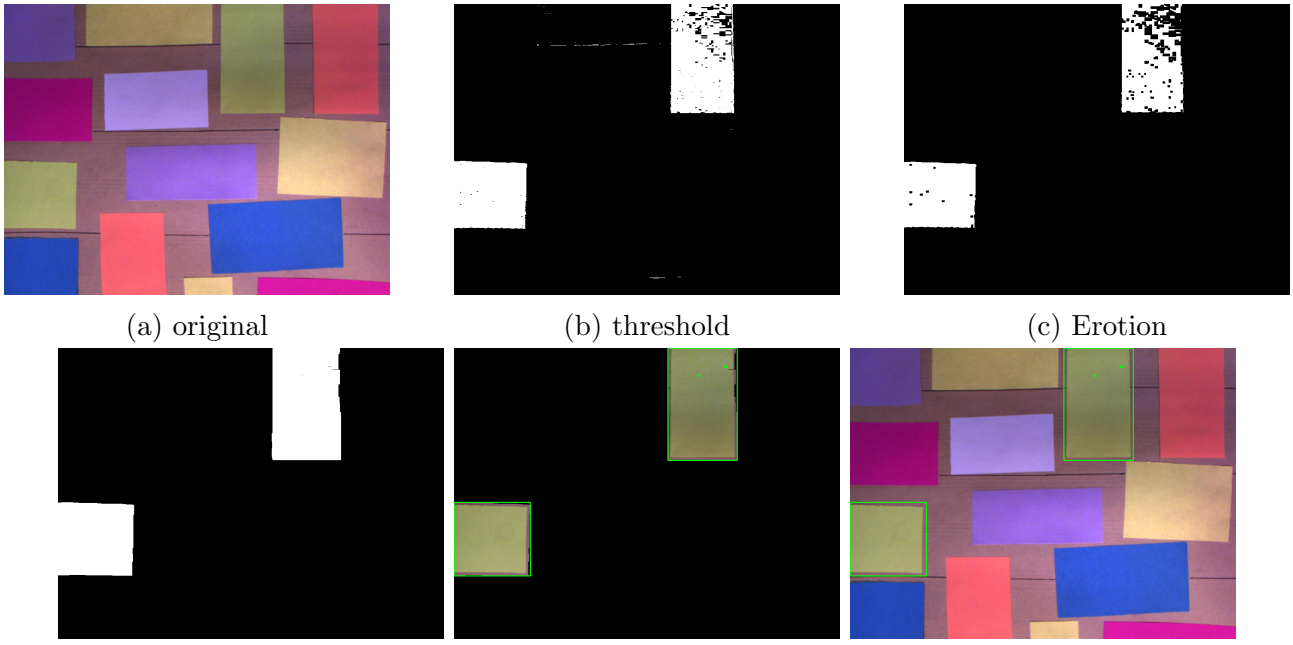


Figure 2.8: Detection process

The HSV parameters are the calibration data obtained from the process mentioned before saved in CSV files, these parameters will allow the computer to interpret data obtained from the video to colors by comparison.
Multiple filters have been used to enhance image resolution thus reducing noise and improving shape recognition, by applying a different mask for each color.

[2] $https://github.com/GroverAruquipa/color_task_ubfc/blob/main/main2.py$
[3] $https://github.com/GroverAruquipa/color_task_ubfc/blob/main/base_functions.py$



(a) original (b) threshold (c) Erotion

(d) Dilatation (e) Contour (f) Rectangle

Figure 2.9: Color recognition steps

We can observe in Fig. 2.9 multiple filters used to reduce noise :
Threshold fig.2.9b : a technique in openCV which is the assignment of pixel values in relation to the threshold value provided. In threshold, each pixel value is compared with the threshold value, if the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value (generally 255).It is sensitive to lighting changes and gradients, color heterogeneity, etc. It is better applied on relatively clean pictures, after blurring them to reduce noise, without much color variance in the objects you want to segment.
Erotion fig.2.9c and Dilatation fig.2.9d, which are basic morphological processing tools that produce contrasting results when applied to either gray-scale or binary images. Dilation has the opposite effect to erosion, i.e. it adds a layer of pixels to both the inner and outer boundaries of regions, while erosion removes pixels on object boundaries.

Later on, opening and closing morphological tools were used to enhance noise filtering. The morphological opening being basically an erosion followed by a dilation,while morphological closing of an image, is the reverse, i.e. dilation followed by an erosion with the same structuring element. we also used openCV findcontour function to detect the contours of the shapes present in the frame fig.2.9f and then openCV bounding rectangle to improve shape recognition fig.2.9e

2.3.1 Principal Functions

In such a way to capture a frame of the video proposed for the evaluation the following lines are used, which are stored in two variables that are *ret* and *frame*, ret will return a boolean response of 1 if the variable is available and frame contains the arrays of frame colors, so in this way the image extracted for the first frame is showed in the fig. 2.9a.

```
cap = cv2.VideoCapture('paperEval.mp4')
ret, frame = cap.read()
```

```
frame = cv2.medianBlur(frame, 5)
```

On the other hand, to perform the conversion from RGB to HSV, the following line of code is used, which is predefined by opencv [4], regarding the theoretical equivalent between this conversion it is possible to find it in [4] .

```
hsvimage = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

Thus, once the color space is defined, we proceed to perform the thresholding with the following function available by opencv[5], in which a lower and upper limit is defined, where internally from a kernel the image is analyzed and becomes **1** all the pixels that are in the specified range and in other case **0**, thus creating the first mask of the image and the result of the conversion is observed in Fig. 2.9b.

```
full_mask = cv2.inRange(hsvimage, minv, maxv)
```

Furthermore, because it is a simple color detection, this problem can be solved by performing morphological transformations in the mask, because it is intended to have an analysis with the minimum latency, we tried to use the minimum number of operations. Thus, the first option applied is *eroding*, where we must define a kernel that evaluates the mask and, based on this size, will apply the operation, resulting in fig. 2.9c

```
def eroding_image(img,ker):
    kernel = np.ones ((ker, ker ), np.uint8 )
    erosion = cv2.erode (img, kernel, iterations = 1)
    return erosion
```

Subsequently, the dilation application is applied, which eliminates the noise in a superficial way, although it is not a robust method, in systems with controlled lighting, it can function relatively correctly, the result of this operation is shown in Fig. 2.9d.

```
def dilate_image(img,ker):
    kernel = np.ones ((ker, ker ), np.uint8 )
    dilate = cv2.dilate(img, kernel, iterations=1)
    return dilate
```

Additionally, Morphological operations of *opening* and *closing* are applied through which the framing of boxes is improved, these operations become complementary and redundant, because they only help refine the result, they must be considered depending on the application in which they are applied. seeks to apply.

```
def opening_image(img,ker):
    kernel = np.ones ((ker, ker ), np.uint8 )
    opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
    return opening
def closing_image(img,ker):
    kernel = np.ones ((ker, ker ), np.uint8 )
    closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
    return closing
```

[4] *https://docs.opencv.org/4.x/df/d9d/tutorial_py_colorspaces.html*
[5] *https://docs.opencv.org/3.4/da/d97/tutorial_threshold_inRange.html*

Consequently, from the previously treated mask, we proceed to find the contour of the regions in the mask, although there are many techniques in the state of the art to approximate an exact solution, in this case we focus on the implementation of the traditional method $ContourAprroximatedmethod$[6], it should be noted that the output of this function will be variables that represent the outline of the image, which is showed in the Fig. 2.9f.

```
def findContourseOfTheMask(img,mask):
    contours, hierarchy = cv2.findContours (mask, cv2.RETR_TREE, cv2.
    CHAIN_APPROX_SIMPLE)
    return contours
```

From the previously obtained contours, the centers of each contour must be found, because the image can have different objects with the same color, for this step the function $cv2.moments$ is used, which returns the center of each contour, where from the application the center is calculated for x and y, to later save it in a list and draw a circle on each center. found[7].

```
def findCenter ( img, contours ):
    # Find the center of the contour
    centers=[]
    for cnt in contours:
        M = cv2.moments (cnt)
        cx = int (M['m10'] / M['m00'])
        cy = int (M['m01'] / M['m00'])
        #save centers in vector
        centers.append([cx,cy])
        cv2.circle (img, (cx, cy), 5, (255, 0, 0), -1)
    return centers,img
```

Subsequently, a frame is made in a rectangle of the contour by means of the $boundingrect$[8] function from the upper left part to the lower part of the image as shown in Fig. 2.9e.

```
def drawBoxAroundObjectWithMask(img,mask):
    # Find contours
    contours, hierarchy = cv2.findContours (mask, cv2.RETR_TREE, cv2.
    CHAIN_APPROX_SIMPLE)
    #Draw a rectangle around the contour
    for cnt in contours:
        x, y, w, h = cv2.boundingRect (cnt)
        cv2.rectangle (img, (x, y), (x + w, y + h), (0, 255, 0), 2)
    return img
```

Once the correct detection of each color has been carried out, the variables are saved in external files, although according to the instructions it is recommended to save the data in .txt files.
In practice this fact is not recommended, due to the fact that in many cases it is necessary to share the data obtained between different applications, and in the process of reading from Java.
For example, many times line terminators must be taken into account or in other cases tabulations or spaces between the data are omitted, in this way it is recommended to use databases based on SQL, or higher levels such as mongoDB, for this simple case the use of saving in *.txt*

[6] $https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html$
[7] $https://www.pythonpool.com/opencv-moments/$
[8] $https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html$

is respected, but at the same time the *.csv* files are used, which become a standard in simple apps like this.

```
df = pd.DataFrame({'frame_number': counterframe, 'x': listcentersx, 'y':
    listcentersy, 'color': listcolor, 'sector': listsector})
df.to_csv('centers.csv', mode='a', header=False, index=False, encoding='utf-8')
# save df in txt file
df.to_csv('centers.txt', mode='a', header=False, index=False, encoding='utf-8')
```

On the wise, the code presented above uses the Pandas library, which is a standard in data science applications, through which the centroids, color, sector and frame number are saved for each detected frame.
Based on the functions presented above, the following code arrangement is proposed, where the same procedure is carried out for each color, only changing the values of HSV max, min, the value of erosion and dilation.

```
minvgr,maxvgr =colorvalues('green')# Change for different colors(green,yellow,
    etc)
dilgr=40
erogr=6
full_maskgr,contoursgr=detection_color.colori(hsvimage, minvgr, maxvgr, dilgr,
    erogr)
kernel=np.ones((100,100),np.uint8)
full_maskgr = cv2.morphologyEx(full_maskgr, cv2.MORPH_OPEN, kernel)
centersgr,imggr=color_functions.findCenter(image,contoursgr)
text="Green"
imggr=color_functions.writeTextInTheCenterOfTheContour(imggr,contoursgr,text,0)
sectorvgr, image=color_functions.findsector(image,contoursgr)
imaux=color_functions.drawBoxAroundObjectWithMask(imaux,full_maskgr)
print("sectorvgreen",sectorvgr)
countourscounter.append(centersgr)
colorcounter.append("Green")
sectorcounter.append(sectorvgr)
```

Finally, the code of this part of the program is divided into two scripts, the first script *main.py*contains the image reading statements and the second script which is $base_functions.py$ module that contains the functions and classes necessary for the correct operation of the program.

2.3.2 Secondary functions

Additionally, secondary functions of the project are explained, as necessary according to the qualification rubric.
In such a way to be able to save the data of each center, lists are created, which are filled in each frame evaluation by means of the append function, shown below, thus being able to obtain the saved data.

```
countourscounter.append(centerspu)
colorcounter.append("Purple")
sectorcounter.append(sectorvpu)
centers.append([cx,cy])
```

Thereby, in the previous lines of code, the lists created for the object counter, a color identifier, the sector in which it is and the centroid that is saved through the $findcenter$ function in the

same way for each iteration, are saved.

Meanwhile, the following function is used to display the information for each frame, where the centroids of the contours to be evaluated for a specific color are calculated, and a specific text is placed using the predefined offset on the X axis, for each color.

```
def writeTextInTheCenterOfTheContour(img,contours,text,offset):
# Find the center of the contour
for cnt in contours:
    M = cv2.moments(cnt)
    cx = int (M['m10'] / M['m00'])
    cy = int (M['m01'] / M['m00'])
    cv2.putText (img, text, (cx, cy+offset), cv2.FONT_ITALIC, 0.6, (0, 0, 0), 2,
    cv2.LINE_AA)
    return img
```

Similarly, the same function is performed for printing and displaying the sector information.

```
    cv2.putText (img, str(sector), (cx, cy+20), cv2.FONT_ITALIC, 0.6, (0, 0, 0),
    2, cv2.LINE_AA)
```

Results Discussion

The result after compiling all the scripts used above is shown in the fig. **??**, in this figure we can observe that all the requirements of the project have been implemented perfectly : shapes and colors are detected and shown in the image.

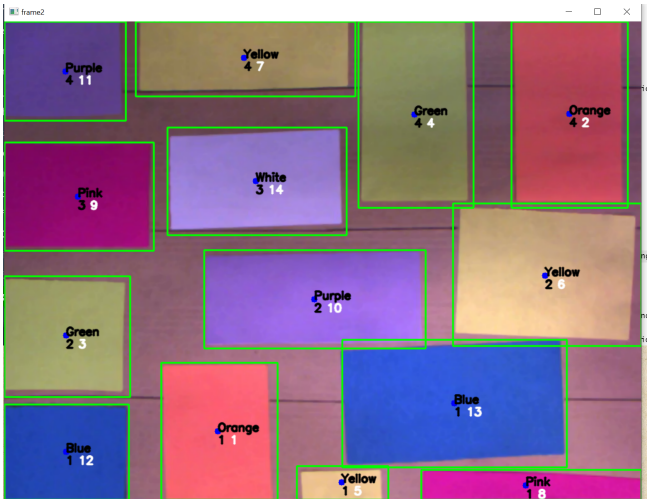


Figure 3.1: final result.

To go further in our work, we have decided to implement a script *histogram.py*, that allows us to show the noise in the frames before and after filtering and then study the noise for each color filter.

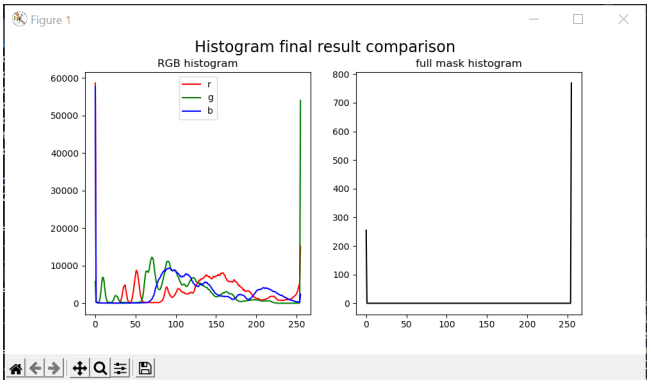


Figure 3.2: Histogram comparison before and after.

We can observe clearly in the two histograms that the noise has been reduced after applying all the filters to our frame compared to the first frame in *RGB* before filtering. To better observe this filter, the fig. 3.2 bellow shows the histogram for each color.

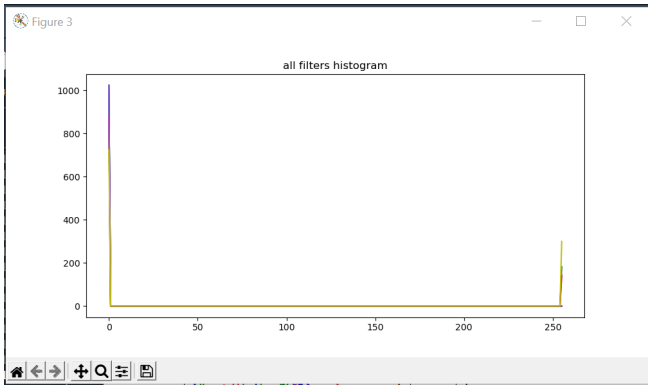


Figure 3.3: All figures histogram

We can see that the noise is almost not visible, in fact reducing it to none would not be of great interest in our application and it's already optimized enough.

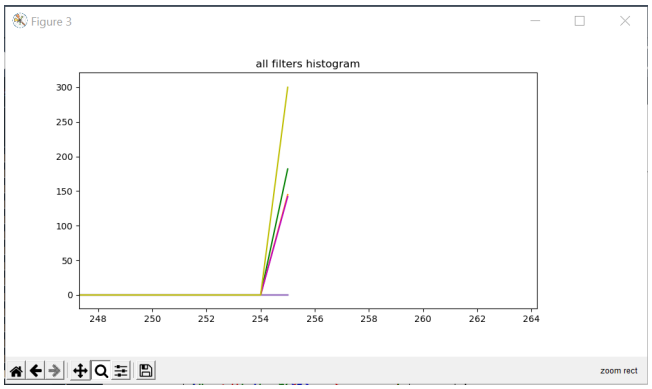


Figure 3.4: All figures zoom 1

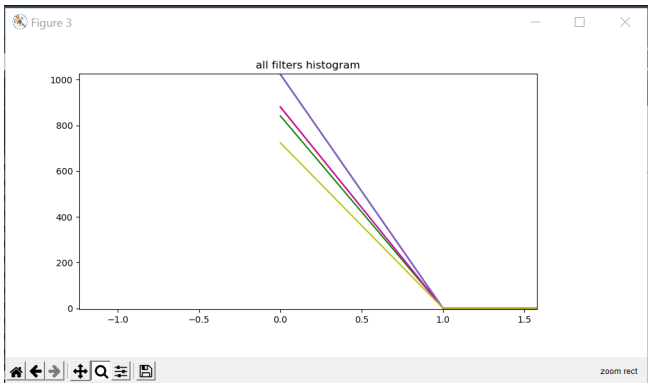


Figure 3.5: All figures histogram zoom 2

The zoom in, in the figures 3.4, 3.5shows better the negligible presence of noise after filtering.

Discussion

In order to implement all the work mentioned above, (factors that helped obtain the results) there was a lot of trial and error and a lot of documentation reading.
We were struck with a lot of compatibility problems jumping from Linux to windows due to the use of very specific functions not so common. This usually does not occur often with python, so finding the solutions to overcome these bugs was kind of challenging and not much help was found on forums, we were thus obliged many times to edit codes that run perfectly or switch libraries to ensure a cross platforms compilation and not much installation of new packages. It was also hard to try to optimize some parts of the code to run faster and avoid bugs with the length of the code and all the specific libraries used.
Gladly, all the main functionalities we were asked to implement were implemented and new features were added beyond the requirements. But still, in further prospects, There might be room to improve the code if needed for other applications, such as a better-looking interface to begin with and thus requiring the use of non-basic packages (not provided directly with python and require installation), recognition of different shapes other than rectangles, adding more color to the color identification library and also applying this recognition to live footage. This project being not more than an introduction to basic machine learning and computer vision, it opens the door to much more applications. If improved, it can jump to real AI code with neural networks and deep-learning to detect complex shapes and figures.

Conclusion

In this work, a basic application of color classification was successfully developed, fulfilling all the necessary objectives and in the same way applying additional concepts to have a quality result. Although this application is a first step in the application of computer vision, it provides all the essentials to scale to possible robust applications based on image color segmentation, whether supervised or unsupervised.
Since in many cases it is necessary to perform a manual segmentation, which can not be done simply by making or obtaining points but by using intervals, although calibration by points was explored, this method turned out to be totally useless, due to the need to consider detection intervals. since when selecting multiple points and finding the average, it does not represent as such the entire spectrum of a color in an image.
On the other hand, the centroid calculation correctly represents the position of each image, although this was not validated (because it is not a requirement) the correct functioning of this tool is visually observed. This application as such, developed by our team, has the necessary flexibility, modularity and robustness to be able to be implemented in simple applications such as the detection of painting errors in assembly parts or the detection of different packages by color in a production line for its manipulation by the centroid and a common application the tracking of mobile robots of given colors.
Finally, it is recommended to be able to propose in future sessions challenges such as the application of har-cascade or HOG filters, or unsupervised learning using KNN or K-MEANS in such a way as to enrich this application much more.

Bibliography

[1] D. Cazzato, C. Cimarelli, J. L. Sanchez-Lopez, H. Voos, and M. Leo, "A Survey of Computer Vision Methods for 2D Object Detection from Unmanned Aerial Vehicles," en, *Journal of Imaging*, vol. 6, no. 8, p. 78, Aug. 2020, ISSN: 2313-433X. DOI: 10.3390/jimaging6080078. [Online]. Available: https://www.mdpi.com/2313-433X/6/8/78 (visited on 11/01/2022).

[2] G. D. Finlayson, "Colour and illumination in computer vision," en, *Interface Focus*, vol. 8, no. 4, p. 20180008, Aug. 2018, ISSN: 2042-8898, 2042-8901. DOI: 10.1098/rsfs.2018.0008. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsfs.2018.0008 (visited on 11/01/2022).

[3] A. Gagalowicz and W. Philips, Eds., *Computer Vision/Computer Graphics CollaborationTechniques: 4th International Conference, MIRAGE 2009, Rocquencourt, France, May 4-6, 2009. Proceedings* (Lecture Notes in Computer Science), en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5496, ISBN: 978-3-642-01810-7 978-3-642-01811-4. DOI: 10.1007/978-3-642-01811-4. [Online]. Available: http://link.springer.com/10.1007/978-3-642-01811-4 (visited on 11/01/2022).

[4] A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIX* (Lecture Notes in Computer Science), en. Cham: Springer International Publishing, 2020, vol. 12364, ISBN: 978-3-030-58528-0 978-3-030-58529-7. DOI: 10.1007/978-3-030-58529-7. [Online]. Available: https://link.springer.com/10.1007/978-3-030-58529-7 (visited on 11/01/2022).

[5] Computer Engineering and Informatics Department, Bandung State Polytechnic, Bandung,Indonesia, P. Hidayatullah, Computer Engineering and Informatics Department, Bandung State Polytechnic, Bandung,Indonesia, and M. Zuhdi, "Color-Texture Based Object Tracking Using HSV Color Space and Local Binary Pattern," *International Journal on Electrical Engineering and Informatics*, vol. 7, no. 2, pp. 161–174, Jun. 2015, ISSN: 20856830, 20875886. DOI: 10.15676/ijeei.2015.7.2.1. [Online]. Available: http://www.ijeei.org/docs-216833929559cc78b15561.pdf (visited on 11/01/2022).