PYTHONIC LENSES

Introduction to OpenCV with Python

PYTHONIC LENSES

OPENCV WITH PYTHON

BY

AHMDAJON AHMEDOV

INTRODUCTION

Welcome to the world of computer vision with OpenCV using Python! Computer vision is one of the most exciting and interesting topics in computer science. This field is a rapidly growing field that focuses on how machines perceive and process data from image and video, and understand visual information from the world around us. The technologies of this field are essential and rewarding for our future, with internet and artificial intelligence becoming more and more popular. From self-driving cars to surveillance systems, computer vision technologies are transforming industries and our daily lives.

HOW TO READ THIS BOOK?

In this book, you are going to learn how to build interesting computer vision applications, using OpenCV for Python, as this technology is used in other programming languages as well. This book is designed to be beginner-friendly yet comprehensive enough for intermediate users. Each chapter builds upon the previous one, gradually introducing new concepts and techniques. Therefore, I highly recommend you to read each chapter attentively. Moreover, you are encouraged to follow along with the examples, as practical coding experience is key to mastering OpenCV.

ABOUT THIS BOOK

This book covers various aspects of OpenCV, including:

1. **Introduction to OpenCV and Environment Setup:** We begin by setting up your development environment quickly, by installing OpenCV and other necessary libraries.
2. **Image and Video Processing:** Learn to load images and videos, display them and do various operations on them. Furthermore, we will talk about how to make live video recorder using your camera.
3. **Image Manipulation:** Understand the structure of images and how to manipulate them by drawing shapes, lines and text on them.
4. **Enhancing Image Quality:** Explore methods to enhance image quality using masks and thresholds.

5. **Advanced Techniques:** Dive into more advanced techniques such as corner detection, motion detection as well as edge detection. These techniques are fundamental for applications ranging from robotics to surveillance.
6. **Object Recognition:** Delve into the fascinating world of computer vision with techniques for face recognition and object detection in images.

EXERCISES AND EXAMPLES

Throughout the book, you will find some challenges to reinforce your learning. Each chapter includes practical examples and code snippets, accompanied with illustrative images. These examples will help you understand how to apply OpenCV techniques in real-world scenarios.

By the end of this book, you will have a solid understanding of OpenCV and the practical skills to tackle a wide range of image and video processing tasks in the realm of computer vision. Whether you are interested in building your own computer vision projects, learning AI and machine, or simply expanding your programming skills, this book will be a valuable resource.

1 – INTRODUCTION TO OPENCV AND ENVIRONMENT SETUP

HISTORY OF OPENCV

OpenCV, short for Open Source Computer Vision Library, is an open-source computer vision and machine learning library. It was initially developed in 1999 as a research project to advance CPU-intensive applications such as real-time object detection, recognition and image processing.

Over the years, OpenCV has evolved into one of the widely used libraries for computer vision tasks due to its efficiency, speed and extensive functionality. It is written in C++ and has interfaces for Python, Java, Matlab and many other programming languages, which makes it accessible to a wide range of developers and researchers worldwide. As I mentioned, we are going to explore OpenCV for only Python!

CAPABILITIES OF OPENCV

OpenCV offers a vast array of capabilities that empower developers to build robust computer vision application, such as image and video processing, feature detection and description, object detection and tracking, machine learning and deep learning, camera calibration and many more.

WHY LEARN OPENCV?

Understanding OpenCV is essential for anyone interested in computer vision, machine learning, robotics and AI. These fields are evolving rapidly, so in the future any experienced programmer should know them.

INSTALLING LIBRARIES

For this book, I assume you have basic python knowledge, and have python interpreter installed on your computer. Besides, in this book we will need a couple of libraries that are not part of the default core stack of Python. This means that we need to install them separately using pip.

First of all, we are going to install the data science libraries:

pip install numpy

pip install matplotlib

pip install colorthief

pip install pandas

As mentioned earlier, you should be comfortable using these libraries already. They are not the main focus of this book, but they are going to help us a lot with some tasks. The main library we will need for this book, as you already know, is OpenCV. We also install it using pip:

pip install opencv-python

All these libraries will help us greatly, otherwise we have to do a lot of work ourselves.

2 – LOADING AND HANDLING IMAGES AND VIDEOS

Before proceeding processing and manipulating images and videos, we will need to learn how to load the respective data to our computer. This is exactly what we are going to learn in this chapter.

LOADING IMAGES

First, let's import the modules that we need:

```
import cv2 as cv
import matplotlib.pyplot as plt
```

Notice that OpenCV was named opencv-python when installing it using pip, but here in order to use it we import cv2. Also we use an alias cv, so that in the future when a new release comes in, cv3 for example, our code stays compatible. Also we import matplotlib, it will come in handy in certain tasks.

In order to load an image, we have to have one in our working directory. Make sure that you use a license free images, you either need to download them from internet that have no copyright, or use your own ones.

```
img = cv.imread('image.jpeg', -1)
cv.imshow('Image', img)

cv.waitKey(0)
cv.destroyAllWindows()
```

In this code, I loaded an image from my working directory. First, I declared a variable called img to access my image later. To this variable, I assigned an actual image, using imread function, to which you have to pass the name of the image file I want to load. The second parameter here is a digit. This digit can be the followings:

-1: Loads an ordinary image as it is. cv.IMREAD_COLOR

0: loads image in grayscale mode. cv.IMREAD_GRAYSCALE

1: Loads including alpha channel. cv.IMREAD_GRAYSCALE

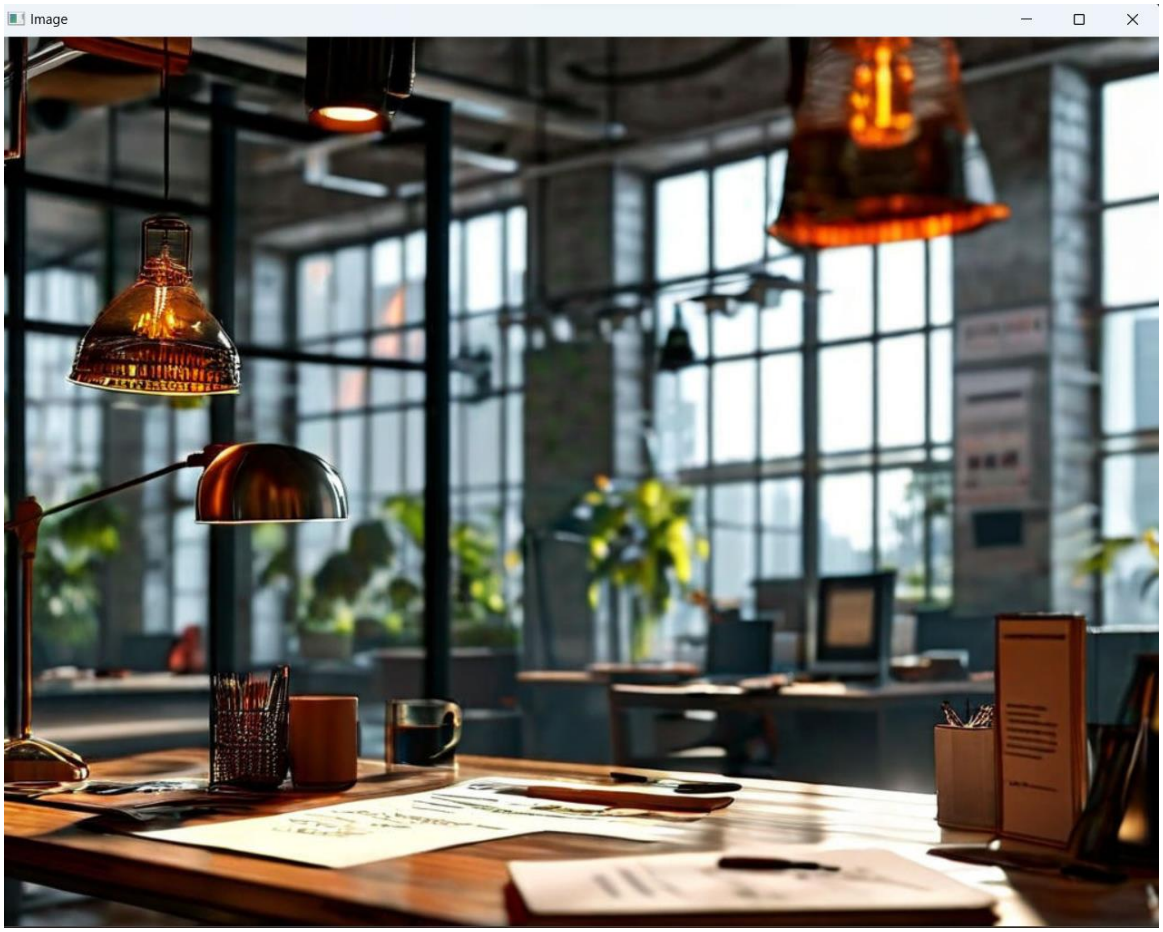
You could also ignore this parameter, and the image would be loaded in the first way (original image).

The next line displays this image on the screen. The first parameter here is the title of the window on which the image will be illustrated, and the second parameter is the actual image. The final two lines wait for user input for infinite amount of time, and when the user presses the close button on the window, the window disappears, which is done by the final line cv.destroyAllWindows. This number is a delay. If I would have passed other positive number to waitKey function, 5 for example, our window would automatically close after 5 seconds.

Otherwise, the window would appear and instantly disappear. The result will as follows:

In grayscale mode:

Including alpha channel:



As you can see the result do not differ in alpha mode, that is because there are no transparent pixels in our image.

CUSTOMIZING LOADED IMAGES

There are multiple options for us to customize images: resizing, rotating, scaling, cropping and so on.

Resizing:

```
img = cv.imread('image.jpeg')

img = cv.resize(img, (0, 0), fx=0.5, fy=0.5)
cv.imshow('Image', img)
```

Here to resize the image we use resize function, which takes three parameters: the image to resize, the size we want to resize, and the relative sizes. If we want to double or halve the image, we should use the fx and fy parameters, in which we specify the ratio of the new image. However, if we want to hardcode the sizes of the image, we should use the second parameter, in which we specify the new sizes as a tuple.

Rotating:

```
img = cv.imread('image.jpeg')

img = cv.rotate(img, cv.ROTATE_90_CLOCKWISE)
cv.imshow('Image', img)
```

To rotate an image, we use the respective function, by passing it our image and an information on how to rotate the image. Instead of `cv.ROTATE_90_CLOCKWISE`, we could also use the following rotation options:

```
cv.ROTATE_180

cv.ROTATE_90_COUNTERCLOCKWISE
```

SAVING IMAGES

Writing processed modified images back to files is easy in opencv, we just need to use the imwrite function, pass it a name of a new image and the image object itself.

```
cv.imwrite('new_image.jpg', img)
```

After running this code, you will see a new file in your current working directory, which will have the same name as you have specified.

LOADING VIDEO

Besides images, we could also load videos into our script:

```
video = cv.VideoCapture('video.mp4')
while True:
    ret, frame = video.read()
    if ret:
        cv.imshow('City Video', frame)
        if cv.waitKey(20) == ord('q'):
            break
    else:
        video = cv.VideoCapture('video.mp4')

video.release()
cv.destroyAllWindows()
```

Here we use VideoCapture class and pass the file path of our video to it. After that, we run an endless loop, which every iteration reads a frame of the video, using the read method. This method returns the boolean data type indicating the success in capturing the frame from a video, and the frame of the video. The former data is super useful to track whether the video was finished or not. Then we show the returned frame of the video using imshow, and then at the end of our loop, we use waitKey that checks if the ‘q’ key was pressed. If we press this key, the script gets out of the loop and terminates. You could set this key to any other key in your keyboard, but q makes most sense here as it stands for quit. As a delay we used 20 (20 milliseconds), which means that we wait 20 milliseconds before showing the next frame. One second has 1000 milliseconds. You can easily modify this number if you want to have higher or lower FPS of a video.

This code effectively loads video and displays it on the screen.

LOADING CAMERA DATA

Finally, let's look at how to getting our camera data into our script. Instead of specifying a file path in VideoCapture class, we specify a digit, which is the index of your camera, in order to view the data of our camera in real time:

```
video = cv.VideoCapture(0)
while True:
    ret, frame = video.read()
    if ret:
        cv.imshow('Camera data', frame)
        if cv.waitKey(10) == ord('q'):
            break
    else:
        video = cv.VideoCapture(0)

video.release()
cv.destroyAllWindows()
```

Here I have set the index to 0, which means the first camera. This assumes that your webcam is connected to the system, and if you have multiple cameras, you can set this number to another one, 1 or 2 for example depending on the number of cameras you have. When you run this code, you will see your yourself, as your computer loads frames directly from your camera, once again in endless while loop.

3 – IMAGE STRUCTURE

Images are fundamental to computer vision, so understanding the structure of images is crucial for effectively manipulating and analyzing them. In this chapter, we will delve into the foundational aspects of image representation and processing. We will cover image analysis, cropping parts of them and pasting them to other coordinates.

STRUCTURE OF IMAGES

In OpenCV, images are represented as NumPy arrays, which provide a versatile and efficient structure for storing and manipulating image data in Python. NumPy is a high performance array and math library for python, and it allows us to perform more optimized operations on arrays. These two libraries are very closely correlated. NumPy arrays are particularly well suited due to their ability to store multidimensional data.

```
import cv2 as cv

img = cv.imread('image.jpeg')
print(img.shape)
```

Output: (1024, 1024, 3)

This output means that our image's size is 1024 by 1024, and that it has three types of colors or channels: red, green and blue (RGB). Specifically, in OpenCV, these values are not standard (RGB), they are BGR, the reverse order. The amount of these colors range from 0 to 255. The mixture of these three colors give us around 16 million different colors. The image is represented by single pixels, which have various colors. In our case, we have 1024 rows of pixels and 1024 columns of pixels. First value is height of image, whereas the second is the width of an image.

```
print(type(img))
```

<class 'numpy.ndarray'>

The type of image is numpy array.

You don't have to understand numpy to follow along this book, but if you know numpy, this is probably useful information.

```
import cv2 as cv

img = cv.imread('image.jpeg')
print(img)
```

Output:

[[[134 132 124]

[132 130 122]

[132 130 122]

…

[122 119 114]

[123 120 115]

[123 120 115]] …

As you can, OpenCV loads an image and stores data of an image in numpy array.

Inside of this array, we have other nested arrays (rows), which by themselves have another arrays inside of them (columns), which hold three values inside of them from 0 to 255. And these final values are the colors of individual pixels. Take this array as an example:

```
[
[[0, 0, 0], [255, 255, 255]],
[[0, 0, 0], [255, 255, 255]]
]
```

Hopefully that is clear. You just have three values, representing each pixel. Regularly, it is red, green and blue, but in OpenCV it is actually going to be blue, green and red.

This is really important to understand, because in the previous chapter when we were modifying the images, we were just modifying this array. There was not any magic here. For instance, we did not just took the image and rotated it, all that actually happened in lower-level, when I wrote cv.rotate, the opencv took this array that we has and changed the values inside of it in a certain way, so that we had a rotated image. You do not have to use this opencv library to do this actions, you could create your own algorithms to do this actions, but opencv provided us with all these ready algorithms, so it is really convenient to utilize.

So let’s look at more practical example, lets look at the values of our array at the index of 0:

```
print(img[0])
```

This gives us the following result:

```
[[134 132 124]
 [132 130 122]
 [132 130 122]
 ...
 [122 119 114]
 [123 120 115]
 [123 120 115]]
```

COPYING THE PART OF AN IMAGE

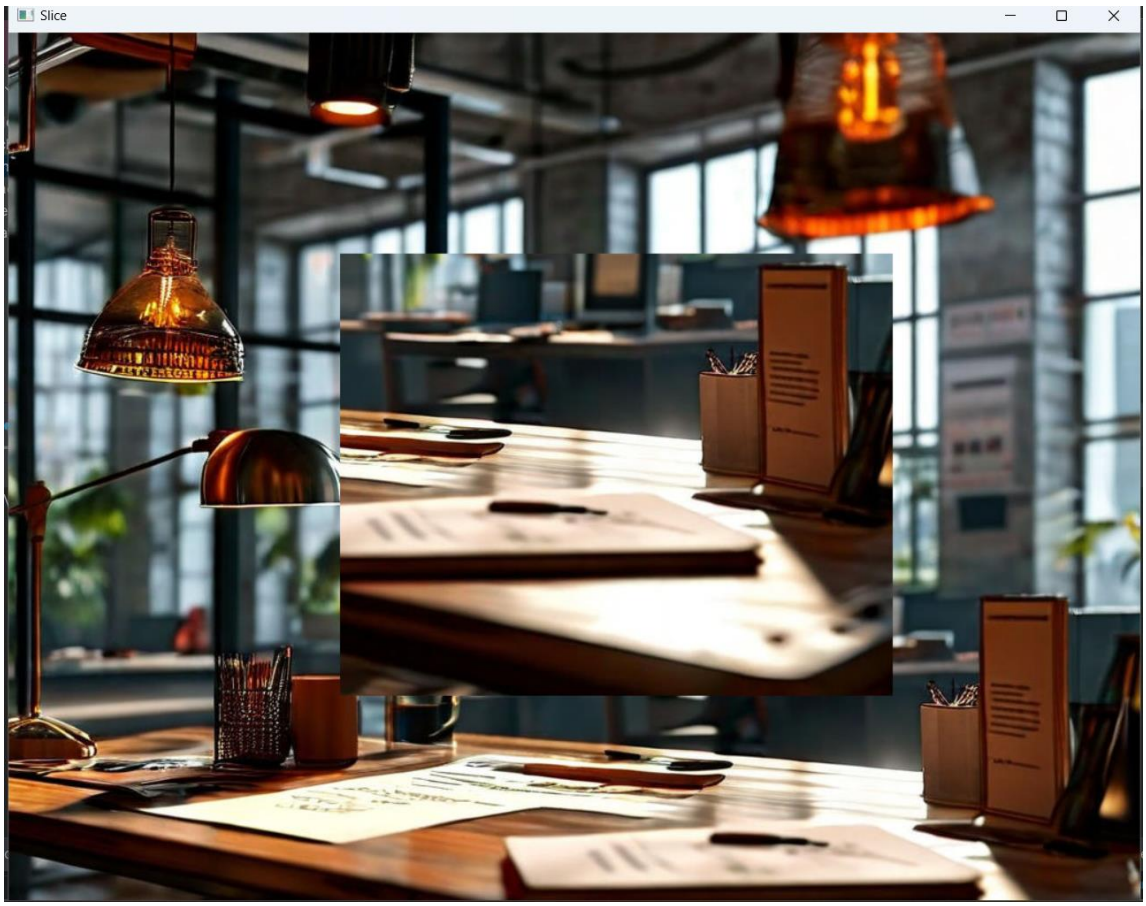
With this basic knowledge gained about images, we can look at how to slice a certain part of an image:

```
img = cv.imread('image.jpeg')
slice = img[500:900, 500:1000]
img[200:600, 300:800] = slice

cv.imshow('Slice', img)

cv.waitKey(0)
cv.destroyAllWindows()
```

First we read an image file. Next, we extracted a specific region or slice of the image we just loaded. img[500:900, 500:1000] selects rows from index 500 to 899 and columns from index 500 to 999 (not including 1000). This extracted slice of the image is stored in the variable slice.

img[200:600, 300:800] = slice

This line pastes (or replaces) the previously extracted slice into another region of the original image. Once again, it selects rows from index 200 to 599 and columns from index 300 to 799 (not including 800). At the end, when we display the image, we see this output:



This example showcases how to perform simple image processing tasks.

CHANGING RANDOM PIXEL COLORS

Let’s look at another interesting example to consolidate what we have learned in this chapter:

```
import cv2 as cv
import random

img = cv.imread('image.jpeg')

for i in range(300):
    for j in range(img.shape[1]):
        img[i][j] = [random.randint(0, 255), random.randint(0, 255),
random.randint(0, 255)]

cv.imshow('Example', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

For this on we will also need random module to generate random pixel colors.

After import statements and reading the image, we iterate through the first 300 rows of the image and all columns in each row, as the number of columns in other rows is equal to the umber of columns in the first row.

For each pixel (img[i][j]), a new random color is assigned using random.randint(0, 255), for each channel: Blue, Green and Red, for BGR format used in OpenCV. This example assigns a random color to the first 300 rows of the image. We can see this from the following ugly picture:

DRAWING ON IMAGES

Now that we know how to load images, video and camera data, and their structures as well, we can start talking about fundamental editing.

Before finishing up this chapter, lets take a look at how draw on our images. With OpenCV we can paint simple geometrical shapes like lines, circles and rectangles onto our pictures.

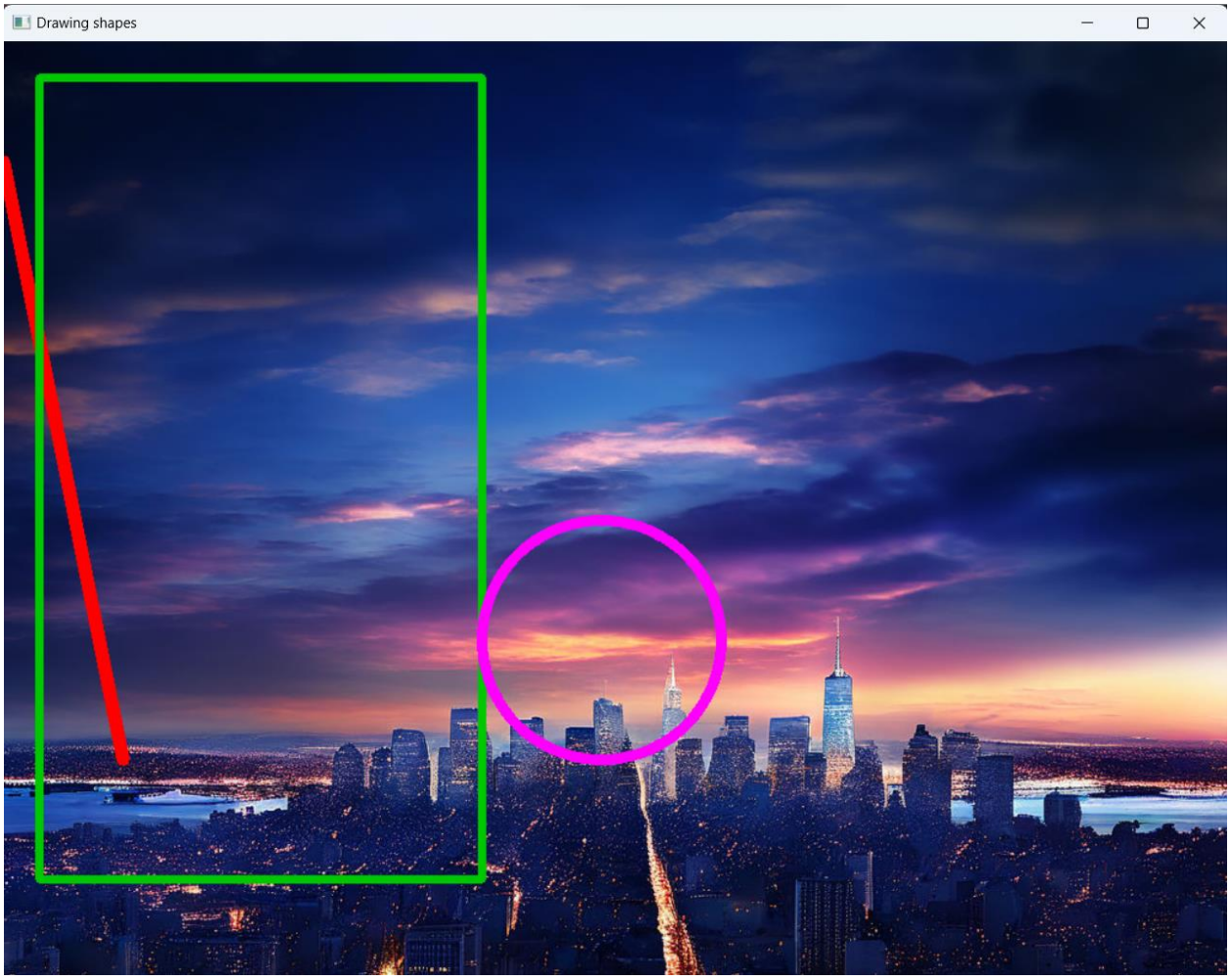
```
img = cv.imread('city.jpeg')

cv.line(img, (0, 100), (100, 600), (0, 0, 255), 10)
cv.rectangle(img, (30, 30), (400, 700), (0, 200, 0), 5)
cv.circle(img, (500, 500), 100, (255, 0, 255), 7)

cv.imshow('Drawing shapes', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

Here is an example of drawing these shapes to a new image of the city.

For this we use functions line, rectangle and circle. But of course we have other functions as well like fillPoly to draw polygons.

The parameters of these functions vary. To line and rectangle we pass the two points right after giving it our image in a form of a tuple. These are the coordinates of a starting and the end points. The two values are the x- and the y-coordinates. Then we also pass a color of the shape in BGR. Finally, we specify the thickness of the line. Note that to fill the whole rectangle, we need to set the thickness value to -1.

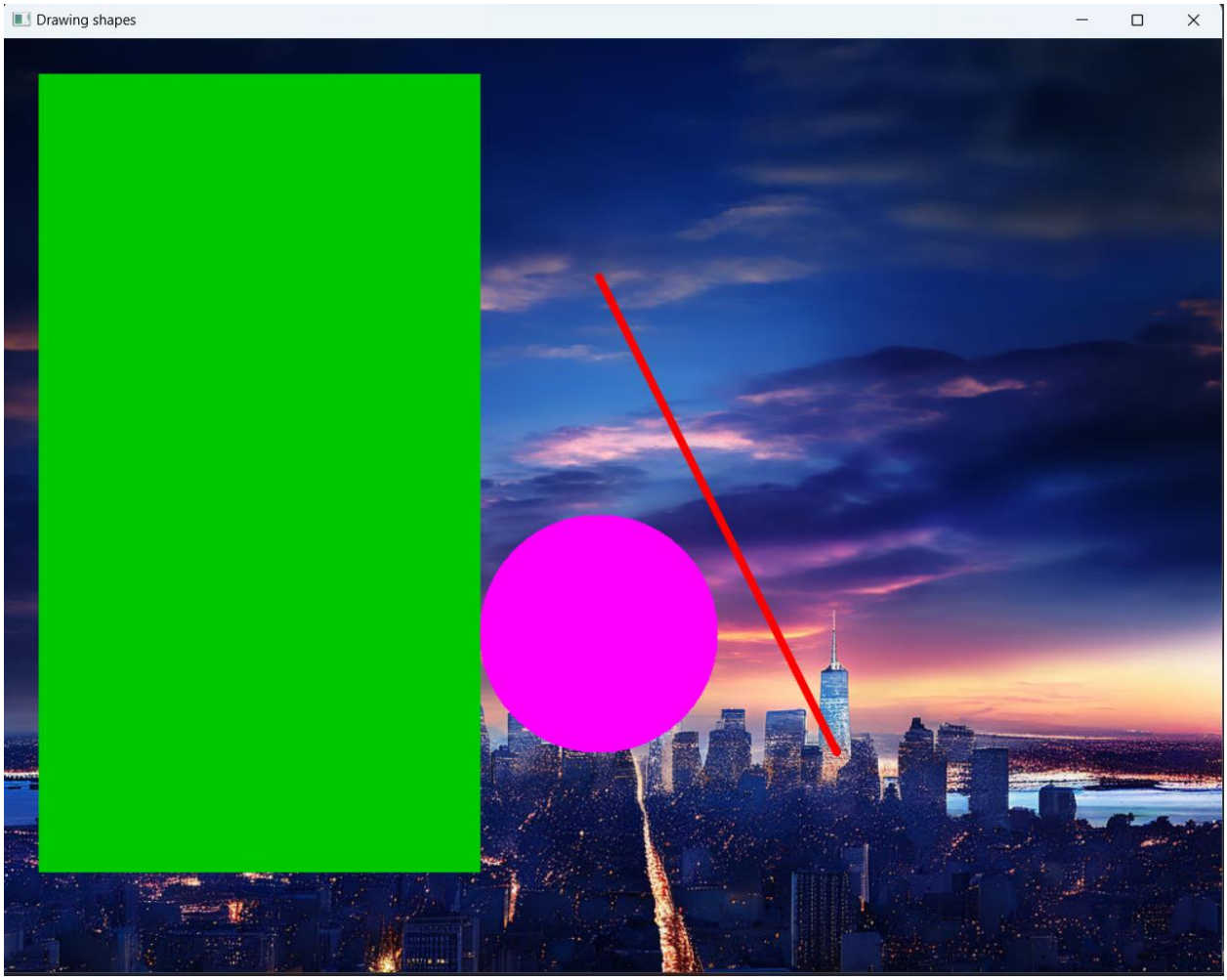
Regarding the circle, we first need to specify only one point, which is the center, and then the radius of the circle. At the end, we once again set the line thickness. The notion for rectangle also applies to circles, when it comes to line thickness.



In the figure above you can see what the result looks like. This primitive drawing is particularly useful when we want to highlight something specific on the screen. For example, in the future chapters we are going to draw shapes around recognized objects in an image.

That is the code of how to fill the shapes using -1 for the thickness:

```
import cv2 as cv

img = cv.imread('city.jpeg')

cv.line(img, (500, 200), (700, 600), (0, 0, 255), 5)
cv.rectangle(img, (30, 30), (400, 700), (0, 200, 0), -1)
cv.circle(img, (500, 500), 100, (255, 0, 255), -1)

cv.imshow('Drawing shapes', img)
cv.waitKey(0)
cv.destroyAllWindows()
```



I changed the coordinates of the line so it does not get overlapped by the rectangle.

You can definitely play around with the coordinates to see where your figures will appear.

DRAWING WITH MATPLOTLIB

It is also possible to use Matplotlib library to draw on our pictures. I am not going to get deep into the details of Matplotlib, since it is out of the scope of this book. And there are too many lessons on this on the Internet. However, lets look at a quick example of plotting function onto our image.

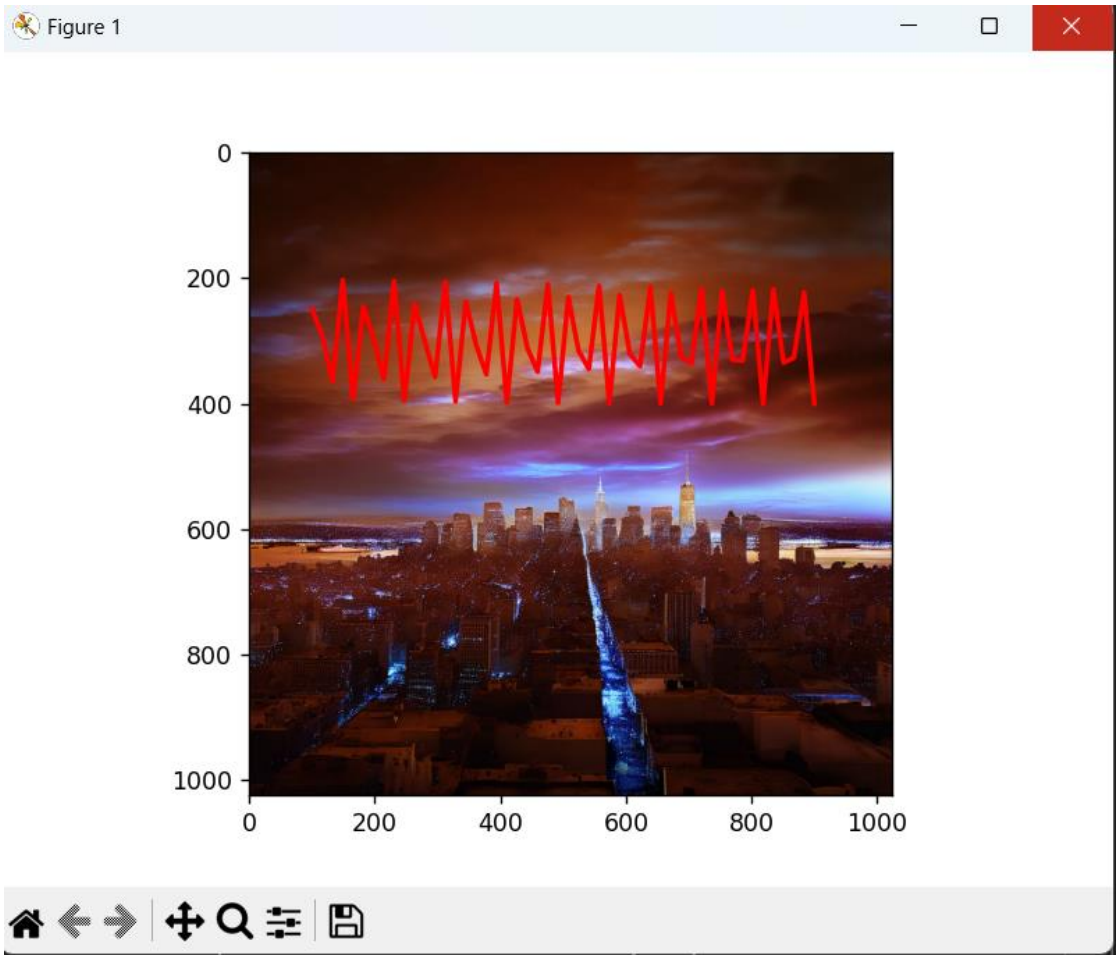
```
import cv2 as cv
import matplotlib.pyplot as plt
import numpy as np

img = cv.imread('city.jpeg')

x_values = np.linspace(100, 900, 50)
```

```
y_values = np.sin(x_values) * 100 + 300

plt.imshow(img, cmap='gray')
plt.plot(x_values, y_values, 'r', linewidth=2)
plt.show()
```

Here I combined the usage of OpenCV, Matplotlib as well as NumPy to display an image alongside a plot of a sine wave superimposed on top of it.

After importing the respective libraries, we generate data for plotting. In this case, np.linspace(100, 900, 50) generates an array pf 50 equally spaced points between 100 and 900. np.sin(x_values) * 100 + 300 computes the sine of each element in x_values, scales it by 100 and shifts it upwards by 300. This creates a sine wave that varies smoothly across the specified range.

plt.imshow(img, cmap='gray') uses matplotlib's imshow function to display the image in grayscale, following which we plot the sine wave in red with a linewidth of 2. Finally, plt.show() displays the combined image and plot using Matplotlib.



So, these were the all the things that I wanted to teach you in this chapter. I know that these things are far from the promised topics in this book, but I believe that

before proceeding to more complicated, yet interesting themes, you have to understand this concept, which is core for the upcoming topics. Also, do not forget to practice as you read this book on your own, to discover new methods of accomplishing the same task and so on, as we best learn by experimenting.

4 – IMAGE ENHANCEMENT TECHNIQUES

In the realm of image processing, the ability to enhance and refine images is essential for extracting meaningful information and improving visual quality. This is exactly what this chapter concentrates on. We are going to learn how masks, filters and thresholds work in OpenCV. At the end of this chapter, we will look at some practical examples to learn about this topic more.

MASKS

Masks in OpenCV enable precise control over which parts of an image are manipulated, analyzed or enhanced. Essentially, a mask is a binary image that acts as a stencil, specifying where operations should be applied based on pixel values. In OpenCV, masks are typically represented as arrays of the same size as the original image, where each pixel's value determines its influence.

Lets look at an example to understand how masks work:

```
import cv2 as cv
import numpy as np

video = cv.VideoCapture(0)
while True:
    ret, frame = video.read()
    width = int(video.get(3))
    height = int(video.get(4))

    hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
    lower_blue = np.array([90, 50, 50])
    upper_blue = np.array([130, 255, 255])

    mask = cv.inRange(hsv, lower_blue, upper_blue)
    result = cv.bitwise_and(frame, frame, mask=mask)

    cv.imshow('Result', result)
    cv.imshow('Mask', mask)

    if cv.waitKey(10) == ord('x'):
        break

video.release()
cv.destroyAllWindows()
```

As always, we first import necessary modules. Then, we open the camera for capturing the video, which is followed by a while loop to capture frames from a video. Following this process, we get the width and the height of our screen using

```
int(video.get(3))
```

and

```
int(video.get(4))
```

Here 3 stands for the width and 4 stands for the height of our screen.

hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV) converts the captured BGR color frame to HSV (Hue, Saturation, Value) color space (hsv). This is useful for color segmentation. Then the following lines of code define a range of blue color in HSV (lower blue to upper blue) where pixels within this range are white (255) and others are black (0):

lower_blue = np.array([90, 50, 50])

upper_blue = np.array([130, 255, 255])

mask = cv.inRange(hsv, lower_blue, upper_blue)

Subsequently, we apply our mask, using bitwise AND operation to the original frame. This results in result, where only pixels within blue color range remain visible:

result = cv.bitwise_and(frame, frame, mask=mask)

Finally, we display our masked result (result variable) and the mask (mask variable) in separate window titled ‘Result’ and ‘Mask’ respectively.

The output will consist of two main windows: in first, only the pixels falling within the specified blue color range are visible, while the second window displays the binary mask, in which pixels that match the blue color range are shown as white and non-matching pixels black.

Following this order, you can create your own masks and use them in your larger projects.

THRESHOLDING

Thresholding is another fundamental technique in image processing used to segment images into regions based on pixel intensity values. It simplifies images by converting grayscale or color images into binary images, where images are either classified as foreground or background based on a specified threshold value. This technique finds wide application in tasks such as object detection, image segmentation and feature extraction.

Thresholding involves setting a threshold value, above or below which pixel values are classified as foreground or background, respectively. It's a straightforward yet powerful technique that helps extract meaningful information from images by separating objects or features of interest from the background.

Here is a basic example of how to perform thresholding using OpenCV:

```
import cv2 as cv

img = cv.imread('city.jpeg', 0)

_, thresh = cv.threshold(img, 120, 255, cv.THRESH_BINARY)

cv.imshow('Original Image', img)
cv.imshow('Thresholded Image', thresh)
cv.waitKey(0)
cv.destroyAllWindows()
```

Here we load the image in grayscale mode, which simplifies the thresholding operation since it operates on a single channel (intensity).

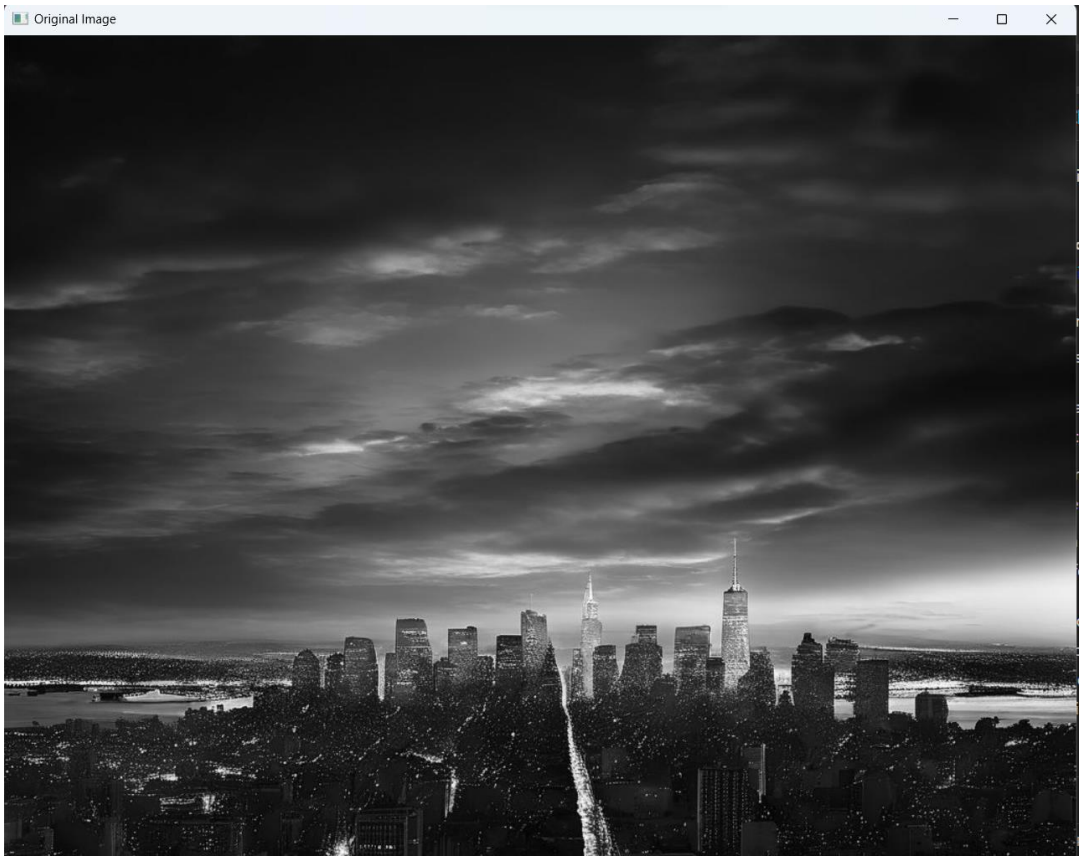
cv.threshold function is used to apply thresholding to our image. First argument is our image to which we want apply thresholding. The second argument we passed is thresholding value. The next one is maximum value used with cv.THRESH_BINARY, where pixels above the threshold become 255 (white) and below become 0 (black). Eventually, cv.THRESH_BINARY parameter specifies the type of thresholding. There are some simple options here, between which you can choose:

- THRESH_BINARY.
- THRESH_BINARY_INV.
- THRESH_TRUNC.
- THRESH_TOZERO.
- THRESH_TOZERO_INV.

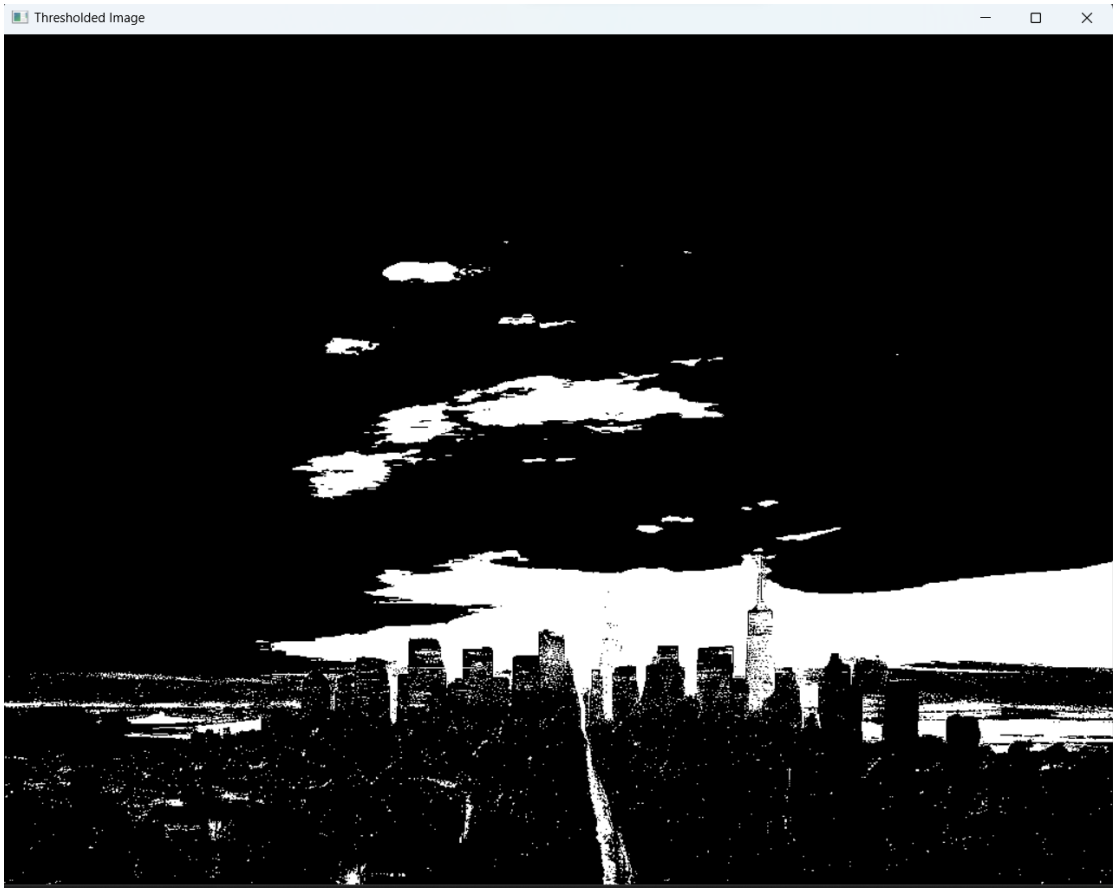
You can experiment them and choose what best suits for your task.

Back to the code, after the threshold function, we display original grayscale image and the threshold binary image. The output is as follows:

Original Image:



Thresholded Image:



Thresholding is extensively used in various applications:

Object Detection: It separates objects from the background, making it easier to detect and analyze them.
Segmentation: It helps in partitioning an image into meaningful regions for further processing.
Image Enhancement: Thresholding can improve image clarity by emphasizing important features or reducing noise.

PROJECT: ENHANCING TEXT READABILITY IN IMAGES

In this project we will explore how to make text in images readable again using image processing techniques we learned, specifically focusing on thresholding.

Text in images often suffers from poor visibility due to factors such as low contrast, uneven lightning, or background interference. The goal of this mini project is to develop a method that can automatically enhance the readability of text by improving its contrast and reducing background noise. As a challenge, you can try to realize this project on your own, it is not too difficult to implement. Just you need to use thresholding to image of the text that is complex to read.

```
import cv2 as cv

img = cv.imread('poor_quality.png')
img = cv.resize(img, (0, 0), fx=0.5, fy=0.5)
img = cv.cvtColor(img, cv.COLOR_RGB2GRAY)

_, result = cv.threshold(img, 32, 255, cv.THRESH_BINARY)

adaptive = cv.adaptiveThreshold(img, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C,
cv.THRESH_BINARY, 81, 4)

cv.imshow('Image', img)
cv.imshow('Result', result)
cv.imshow('Adaptive', adaptive)
cv.waitKey(0)
cv.destroyAllWindows()
```

Here is the full code for achieving such quality of the book. Initially, we load and resize the image of the book which has really poor quality, as its size was too big. After converting it to grayscale, we apply to the image simple global thresholding. Following this, we apply adaptive thresholding, which is much more flexible. We pass 6 arguments in total. First and second ones are grayscale image and maximum value that can be assigned to pixel accordingly. cv.ADAPTIVE_THRESH_GAUSSIAN_C is adaptive thresholding method using the weighted sum of neighborhood values. Instead, you could also pass ***cv2.ADAPTIVE_THRESH_MEAN_C*** and many others, but I think for these task Gaussian C works best. cv.THRESH_BINARY is a thresholding type where pixel values are set to either 0 or 255 as mentioned earlier. Next is a size of pixel neighborhood used to calculate the threshold value (this number must be odd), and finally a constant subtracted from the mean or weighted mean.

After displaying the images, we see impressive results:

Original Image:

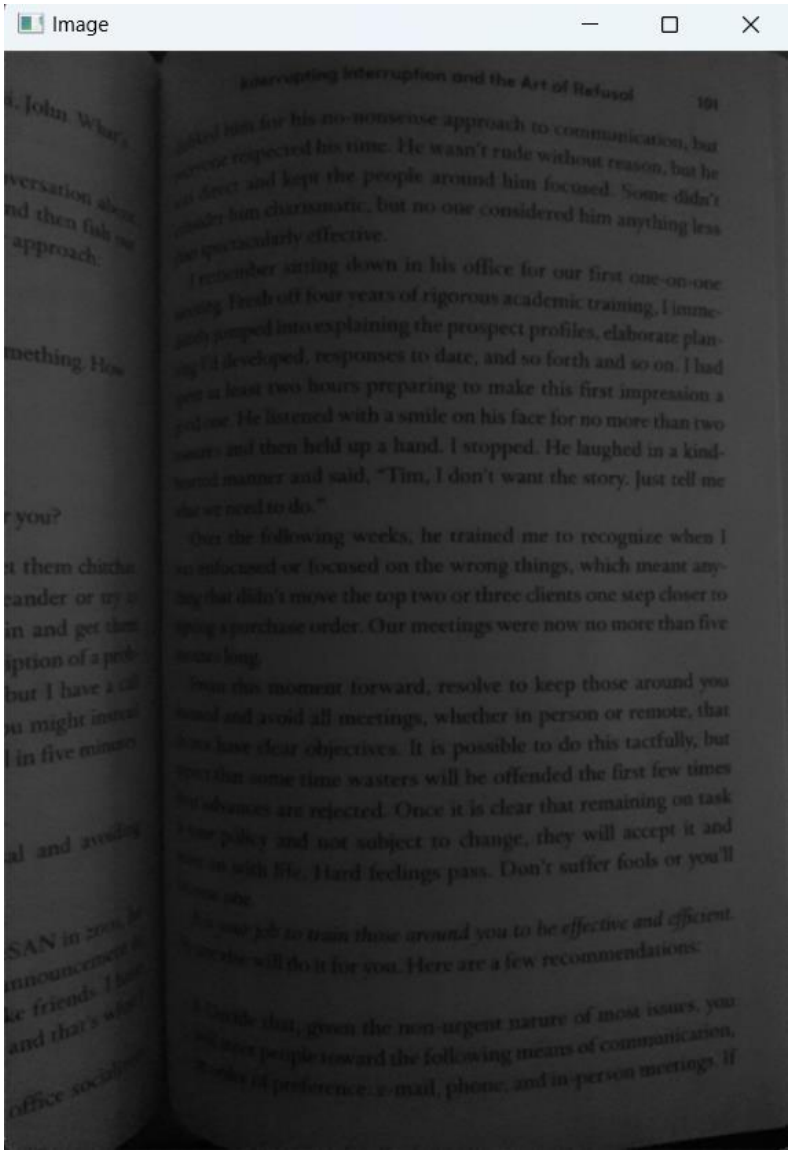
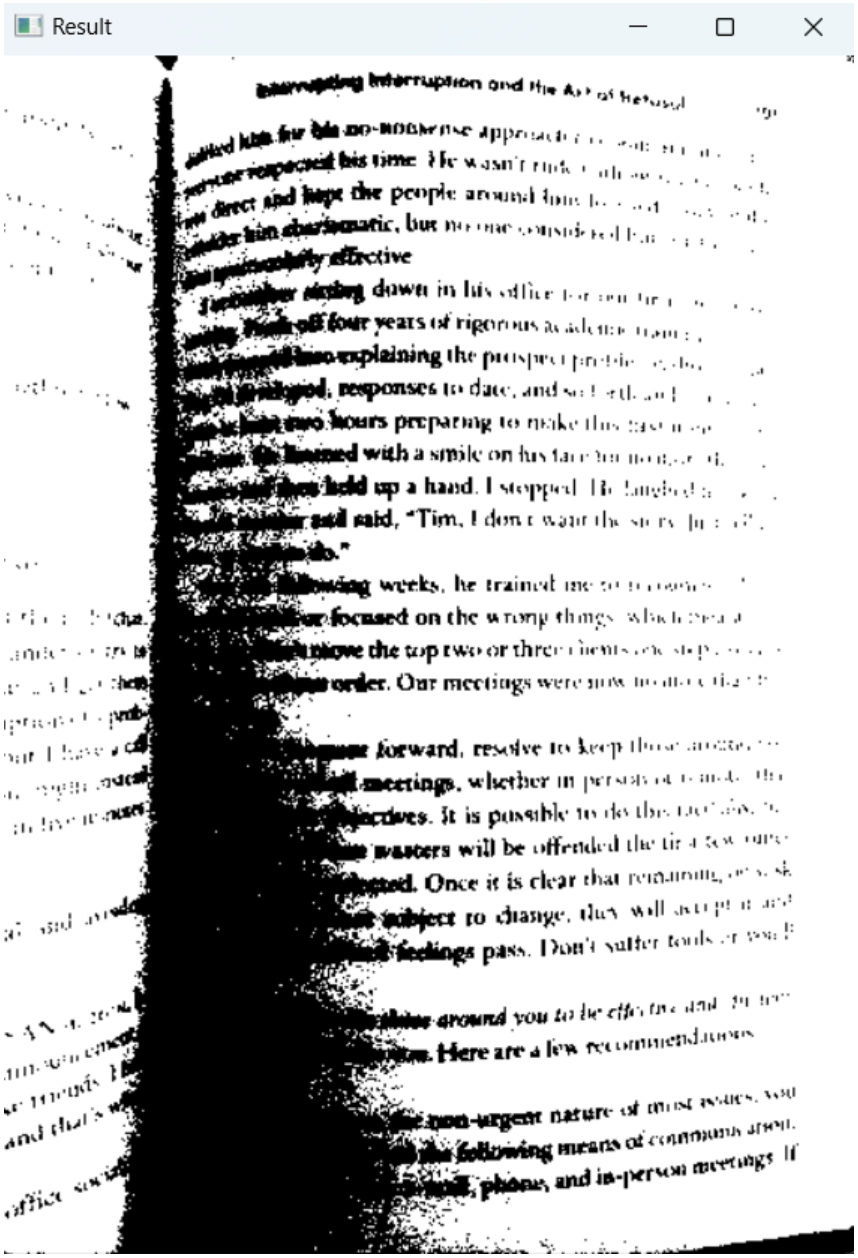
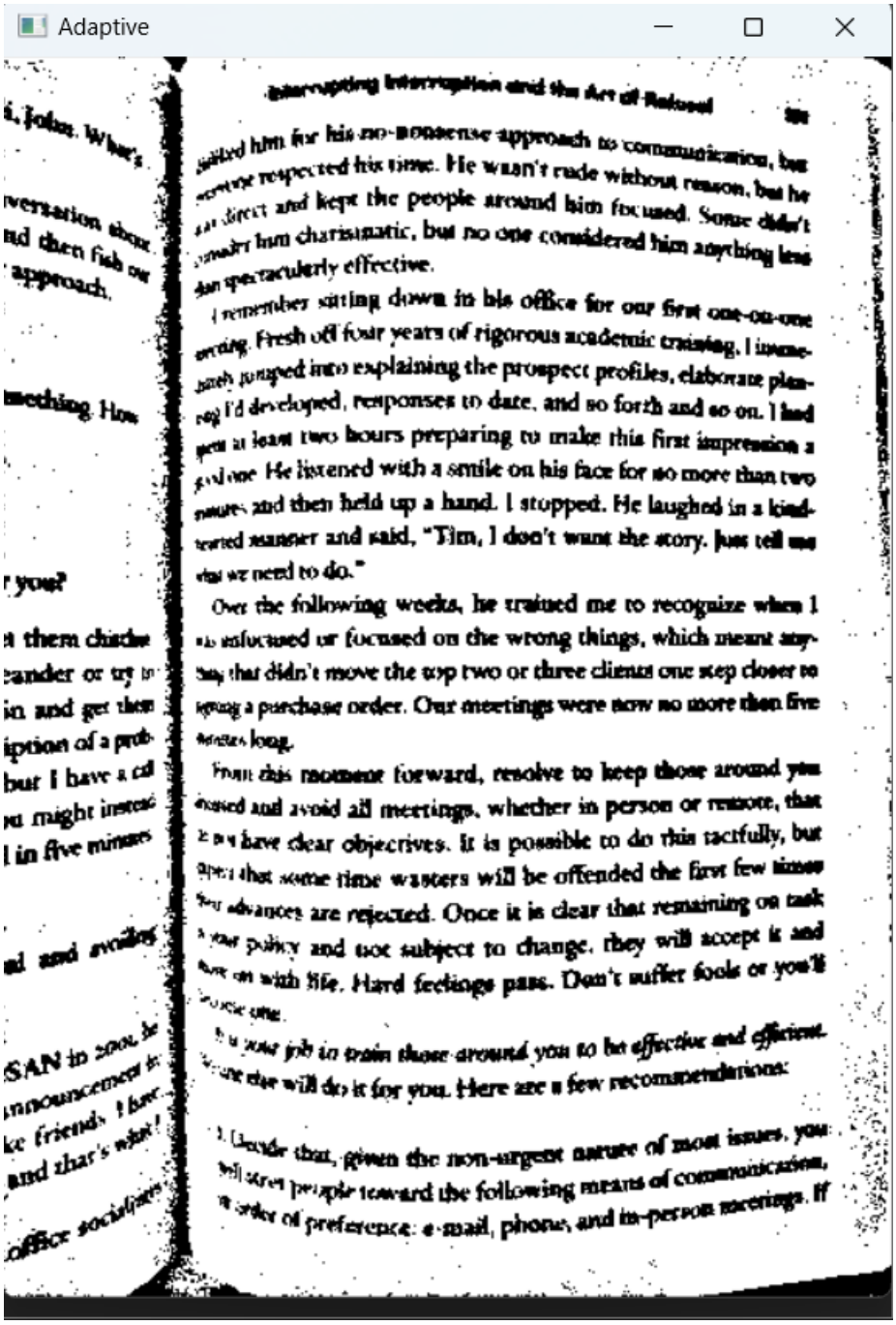


Image after applying simple global thresholding, its quality is really poor as you can see. Most parts are totally black, whereas others are too light:



And finally, the adaptive thresholding, which gives us the best result so far:



This project illustrates how thresholding techniques can significantly improve the readability of text in images, demonstrating a practical applications in document enhancement and OCR (Optical Character Recognition). I recommend you experimenting with different parameters and constant numbers, to find out which works best for your image.

5 – FEATURE DETECTION AND TRACKING

Having acquired foundational knowledge in image processing and computer vision, it is now time to harness this expertise to delve into the fascinating realm of feature detection and tracking real life objects. In this chapter, we explore advanced methods that enable computers to perceive, analyze and interpret data with precision. We will cover 4 topics in this chapter:

Corner Detection, Motion Detection, Edge Detection and Dominant Color Extraction, either from images or videos.

This chapter builds upon your existing knowledge, empowering you to leverage advanced techniques effectively, so if you are not familiar with previous topics, I definitely recommend you reading them first.

CORNER DETECTION

Corner detection is crucial for tasks like image stitching, 3D reconstruction and object recognition. To implement it, we are not going to realize algorithms from scratch, but use ready algorithms like Harris or goodFeaturesToTrack algorithm.

This is the original image that we are going to work with.

```
import cv2 as cv
import numpy as np

img = cv.imread('skyscraper.jpeg')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

corners = cv.goodFeaturesToTrack(gray, 100, 0.01, 10)
corners = np.intc(corners)

for corner in corners:
    x, y = corner.ravel()
    cv.circle(img, (x, y), 5, (255, 0, 0), -1)
    for i in range(len(corners)):
        for j in range(i + 1, len(corners)):
            corner1 = tuple(corners[i][0])
            corner2 = tuple(corners[j][0])
            color = tuple(map(lambda a: int(a), np.random.randint(0, 255,
size=3)))
            cv.line(img, corner1, corner2, color, 1)

cv.imshow('Frame', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

After the initial setup steps, we convert the loaded image to grayscale, which simplifies the corner detection since it operates on a single channel intensity values. In this example, we utilize corners = cv.goodFeaturesToTrack(gray, 100, 0.01, 10) algorithm, which detects corners in the grayscale image using the Shi-Tomasi corner detection method. Parameters here are: 1) grayscale input image 2) 100 – maximum number of corners to detect 3) 0.01 - Quality level, indicating the minimum accepted quality of corners. If you were to increase this value, our program would find less corners, but with high level of accuracy. Else, it will have a lot of corners, but most of them would be incorrect guesses, so in order to escape from these problems, we need to find the medium value.

4) 10 – minimum Euclidean distance between detected corners.

Thereupon, we draw detected corners and connections between them on an image. The first outer loop iterates over each detected corner.

x, y = corner.ravel() extracts the x and y coordinates of each corner.

cv.circle(img, (x, y), 5, (255, 0, 0), -1) draws a filled blue circle at each corner position.

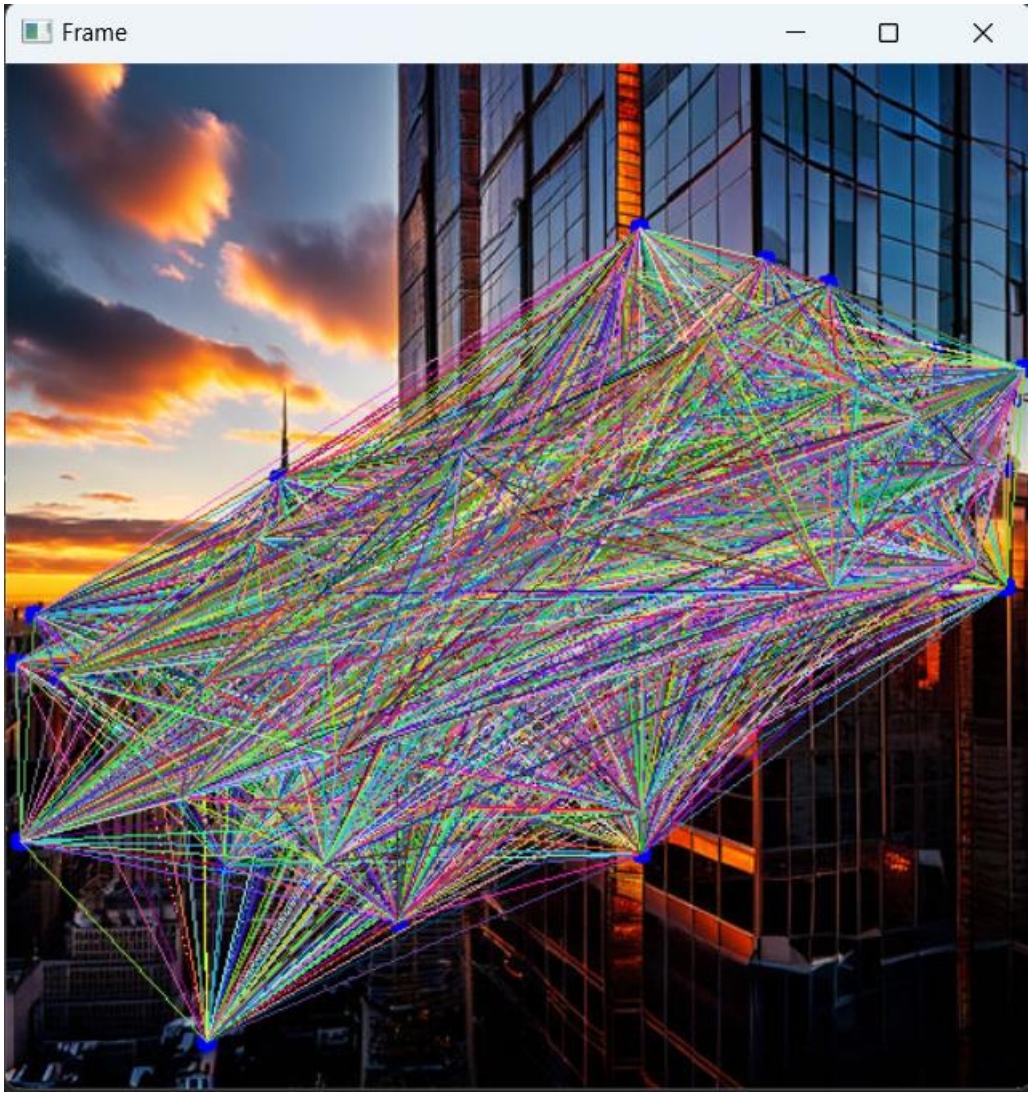
The following nested loops iterate over each pair of corners.

`corner1` and `corner2` extract and define the positions of each corner pair.

color generates a random BGR color tuple for each line connecting the corners using the respective NumPy function.

cv.line(img, corner1, corner2, color, 1): Draws a line between each pair of corners in a random color.

After displaying the image we see the following output:



As you see, the program drew lines between each detected corner and visualized them with blue circles. Corner detection is vital in computer vision for tasks such as image stitching, structure from motion, and object tracking. Definitely play around with the numbers we have hardcoded to find out the most relevant result for your image.

MOTION DETECTION

Now! Lets talk about motion detection using OpenCV. Motion detection enables systems to perceive changes in the environment by analyzing sequences of images or video frames. It plays a crucial role in various applications. The primary goal of it is to identify regions within the image or video where there is movement compared to a reference frame or background.

```
import cv2 as cv

video = cv.VideoCapture(0)
subtractor = cv.createBackgroundSubtractorMOG2(300, 200)

while True:
    _, frame = video.read()
    mask = subtractor.apply(frame)
    cv.imshow('Mask', mask)
    if cv.waitKey(10) == ord('q'):
        break

video.release()
cv.destroyAllWindows()
```

This is the second example of feature detection.

In here, we capture video from a camera, to detect user's motion in real time. As always, you can replace 0 in VideoCapture with the path to a video file to process video instead.

Next, we define a so-called subtractor object, using the MOG2 (Mixture of Gaussians) method: 300 is a history -> specifies the number of previous frames used to build the background model. 200 is a varThreshold -> sets the threshold on the squared Mahalanobis distance to decide whether a pixel is part of the background (higher values mean less sensitivity to noise and more to slow moving objects).

In the while loop, read frames, apply background subtractor we defined earlier to the current frame to obtain a binary foreground mask, where white pixels represent detected motion, and display the result using imshow function.

Last but not least, we exit the loop and do a little bit a cleanup.

This technique effectively identifies moving objects by comparing each frame with dynamically updated background model. Identically, you could apply the same method to detect motion in videos of the crowd, where the motion is easy to detect.

Such models can be used in real life in gesture control for example, where users can interact with devices and interfaces through hand gestures, which enhances user experience in gaming, virtual reality and smart TVs.

EDGE DETECTION

Alright! Edges are another fundamental features in images that cover important information about object boundaries and shapes. Edge detection forms the bedrock of many computer vision applications, enabling tasks such as object recognition and image segmentation.

OK, lets get into the code!

```
import cv2 as cv
import numpy as np

video = cv.VideoCapture(0)

while True:
    _, frame = video.read()
    cv.imshow(Original, frame)

    laplacian = cv.Laplacian(frame, cv.CV_64F)
    laplacian = np.uint8(laplacian)
    cv.imshow('Laplacian', laplacian)

    edges = cv.Canny(frame, 100, 100)
    cv.imshow('Canny', edges)

    if cv.waitKey(10) == ord('q'):
        break

video.release()
cv.destroyAllWindows()
```

Here, to detect edges we setup live video recorder nicely and use two types of filters: Laplacian and Canny. Former one has a lot of background noise, whereas the latter one displays images in plain black and white surface.

To compare these filters with original video, we display original captured frame. Then we apply Laplacian edge detection to our frame. The cv.CV_64F argument specifies the output image depth (64-bit float). You do not need to understand what is it, it is just the type of image. Then, we convert the Laplacian result to an unsigned 8-bit integer for proper display, after which we display the edge-detected image in a new window.

As for Canny, we apply it to frame as well with thresholds of 100 for both low and high thresholds. After displaying this image as well, we can see that the result of it is much more precise.

These edge detection techniques can be applied to your videos as well, you just need to load the in VideoCapture instead of your web camera.

EXTRACTING DOMINANT COLORS FROM IMAGES

So let's at the end of this chapter take a look at another topic, which is not related to OpenCV, but is closely correlated with computer vision. That is to find out the most dominant colors from images.

For these purpose we do need to install extra package colorthief:

pip install colorthief

```
from colorthief import ColorThief
import matplotlib.pyplot as plt

ct = ColorThief('color.jpeg')
dominant_color = ct.get_color(quality=1)

plt.imshow([[dominant_color]])
plt.show()

palette = ct.get_palette(color_count=5)
plt.imshow([[palette[i] for i in range(5)]])
plt.show()
```

Here we use modules colorthief, which has many useful functions, and matplotlib to display the output.

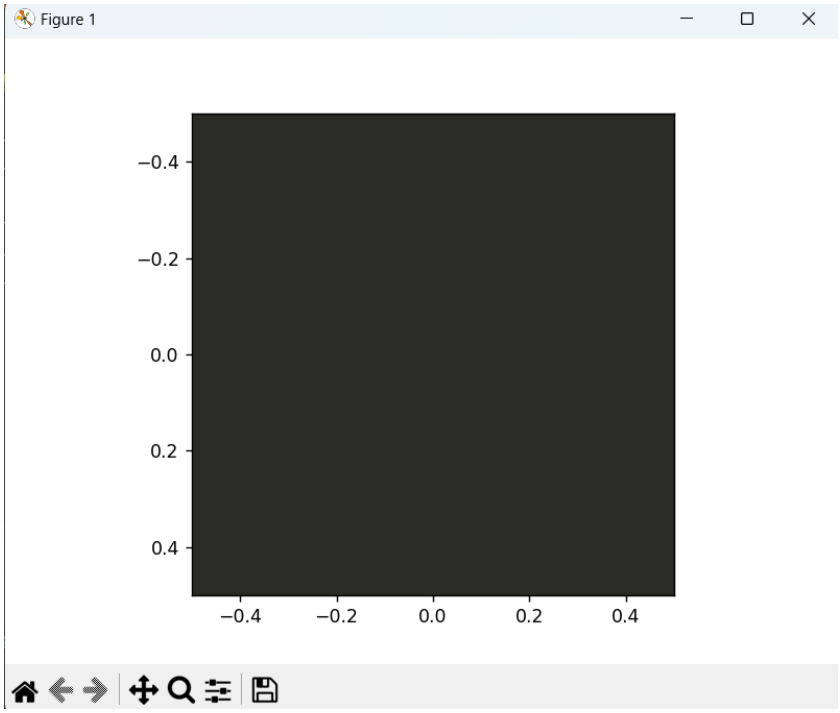
We first import both of them, then initialize colorthief object, by passing it our image file. To extract dominant color from our image, we use ct.get_color(quality=1) method of ColorThief class. The quality parameter (set to 1 here) controls the quality of the color extraction; higher values may result in more accurate but slower processing.

Subsequently, we display our dominant color using imshow function of matplotlib. The double square brackets are used to create a 2D, which is necessary for imshow to interpret as an image.

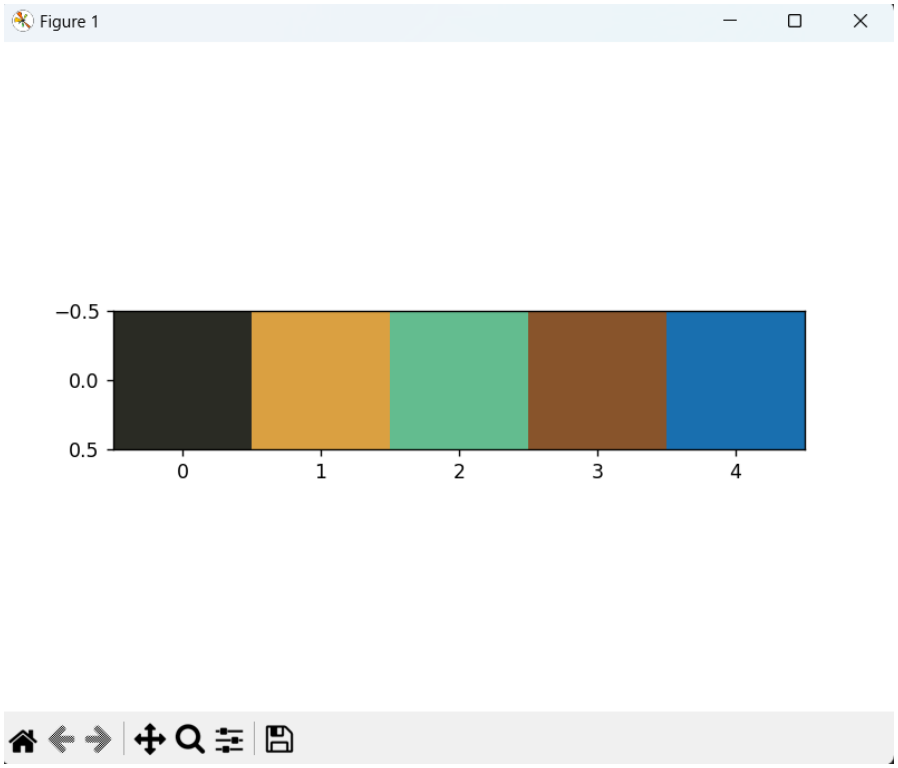
Last but not least, we extract a color palette of dominant colors from the image, using get_palette method. color_count specifies that we want a palette containing 5 dominant colors. In order to illustrate it, we use imshow again, where a list comprehension extracts each color from the palette list and forms a 2D array for nice visualization.

The initial image itself looks like this. I deliberately chose image with multiple colors:

Dominant color in this image is dark grey:



Five dominant palettes:



This example demonstrates a simple yet powerful use of the ColorThief library to extract and visualize dominant colors and a color palette from an image file. This library greatly complements OpenCV in terms of computer vision.

6 – FACE RECOGNITION AND OBJECT DETECTION

In the end of this book, we are going to cover remaining two key areas of computer vision: Face recognition and Object detection from either images or videos. These two fields have revolutionized how machines perceive and interact with the visual world. These technologies can be used in games for example, particularly when distinguishing enemies and facilitating attacks. By this you can create sort of a neural network that can play games. You just need to give commands to a machine of what to do if it finds enemies or if it does not find them.

FACE AND EYE DETECTION, CASCADE CLASSIFIER

So, let's start by looking how to detect face from a video, as the same principle can be applied to images.

```
import cv2 as cv

video = cv.VideoCapture(0)
face_cascade = cv.CascadeClassifier(cv.data.haarcascades +
'haarcascade_frontalface_default.xml')
eye_cascade = cv.CascadeClassifier(cv.data.haarcascades +
'haarcascade_eye.xml')

while True:
    _, frame = video.read()
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)

    for x, y, w, h in faces:
        cv.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 5)
        roi_grey = gray[y:y + w, x: x + w]
        roi_color = frame[y: y + h, x: x + w]
        eyes = eye_cascade.detectMultiScale(roi_grey, 1.3, 5)

        for ex, ey, ew, eh in eyes:
            cv.rectangle(roi_color, (ex, ey), (ex + ew, ey + eh), (0, 255,
0), 5)
    cv.imshow('Face Detection', frame)
    if cv.waitKey(1) == ord('q'):
        break

video.release()
cv.destroyAllWindows()
```

With this simple code, we can easily detect faces in video or image.

We are using webcam here to detect the user's face. We initialize Haar Cascades:

face_cascade = cv.CascadeClassifier(cv.data.haarcascades + 'haarcascade_frontalface_default.xml') loads the Haar cascade classifier for frontal face detection, while eye_cascade = cv.CascadeClassifier(cv.data.haarcascades + 'haarcascade_eye.xml') loads the Haar cascade classifier for eye detection. These are the built in data of eyes and faces of people that we are using to track of ours. The are stored in xml files and in the code above we are accessing them. In the main processing loop, we first convert the current frame to grayscale. Second, using these line of code, faces = face_cascade.detectMultiScale(gray, 1.3, 5), we detect faces in the grayscale frame using the detectMultiScale method of face_cascade. There are also other methods like detectMultiScale2 and detectMultiScale3, but they are mostly used for images with different number of dimensions. Parameters are adjusted for optimal face detection there. First number is scale factor and the second one is the number of minimum neighbors. The scale factor specifies how much the image size should be reduced at each image scale. Because the built in images that we have loaded initially have certain sizes, but our image can be too large, 10000 pixels by 10000 pixels for example. In this case, the program would not be able to detect faces correctly, so for that reason scale factor is used. The recommended value is 1.1. The next minNeighbors specifies how many neighbors each candidate rectangle should have to retain. This parameter will influence the quality of the detected faces, higher value results in fewer detections but higher quality, whereas lower value results in more detections which are less precise. Definitely try adjusting these numbers and see what results you get.

Following this, we get the x and y positions of the face and the width and height of the face. We iterate over the faces to get those values, and draw rectangles around each detected face on the original color frame.

Similarly, we detect eyes within each detected face region: `roi_gray` and `roi_color`

eyes = eye_cascade.detectMultiScale(roi_gray, 1.3, 5) detects eyes using the same function and arguments. Then we again iterate over the detected eyes and draw green rectangles around each detected eye on the roi_color frame.

Finally, we display these frames on the image. The output of this displays the real-time face and detection, for which we used Haar cascades in OpenCV. It continuously captures frames from the webcam, detects faces, draws rectngles around them, extracts eye regions within each detected face and draws rectangles around the eyes.

The result will be pretty impressive. You can also use built in cascades to OpenCV, some of them are lower body, upper body, smile, left eye and right eye.

OBJECT DETECTION

I am going to use this picture in order to recognize objects, specifically a clock.

OpenCV does not have any default Haar Cascades for clock, so we need to download them from the internet. Here I found this nice GitHub where you can download cascade for clock and many other objects:

https://github.com/gale31/ObjectDetector/blob/master/return-wallclock/classifierWallClock.xml

If for some reason the link do not work in the future, you just can look up in the internet for some Haar Cascades for the objects that you want to recognize.

However, take care of the licensing of these files.

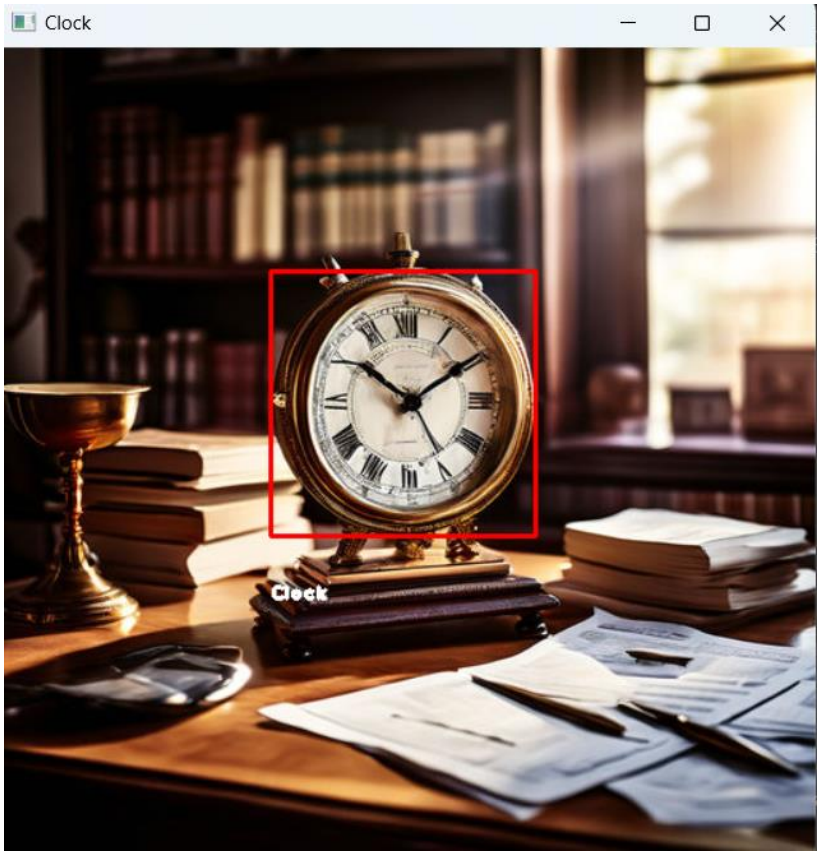
But anyways, in the picture above you can see a large clock, which we want to recognize with our script.

```
import cv2 as cv

img = cv.imread('clock.jpeg')
clock_cascade = cv.CascadeClassifier('classifierWallClock.xml')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

```
clocks = clock_cascade.detectMultiScale(gray, 1.3, 10)
for (x, y, w, h) in clocks:
    cv.rectangle(img, (x, y), (x + w, y + h), (0, 0, 255), 2)
    cv.putText(img, 'Clock', (x, y + h + 40), cv.FONT_HERSHEY_PLAIN, 0.8,
(255, 255, 255), 2)

cv.imshow('Clock', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

Initial steps of the code are loading image and the cascade that we have downloaded. Make sure that this xml file is in your current working directory, otherwise it will not work either. Next we convert image to grayscale as OpenCV is much more comfortable working with grayscale images. Then we use detectMultiScale function of our classifier in order to recognize clocks with the help of our XML file. Here we again pass two optional parameters, excluding our grayscale image. First is the scaling factor of the image, which is the same as the previous example, and the second is the minimum amount of the neighbor classification for a match. That's actually it. It is just as simple to detect objects in images using OpenCV. We now have to visualize everything using the for loop, where we draw rectangle around the clock and imshow function. Note that we have drawn a text on our image along with the rectangle. To draw a text, we just use the putText function and pass it our image, Text, positions, font, fontscale, color and the thickness. The font is also built in to the OpenCV library: `cv.FONT_HERSHEY_PLAIN`.

With this method, you could also draw the names of the people in a certain image. That can be a nice challenge for you: Load the pictures of people, or well pictures of their faces and save them in a separate folder. Then load image where all these people appear at the same time. Then you could detect faces and check if they match with the faces of people that you have prepared. If so, you can do certain stuff, like specifying his/her name with the putText function.



As you can see this also works pretty well. This technique also works with videos with camera data and normal videos. You just need to start a while loop and extract every frame of the video as always.

As always, experiment around with the concepts and terms you have just learned about in this chapter, as programmers learn fastest when they try experimenting and trying with various methods. Just be creative and constantly ask yourself a question: What the result will be if I change something in my code?

What Next?

Congratulations on completing this journey through Python OpenCV! By now, you have explored the fundamental concepts and practical applications of computer vision using one of the most powerful libraries available. You have learned how to manipulate images, detect objects, track motion and even apply machine learning algorithms to enhance visual data analysis.

As you reflect on your accomplishments with OpenCV, consider the following advice for you ongoing learning. Since no book in the world could teach you everything, so you need to learn some extra IT skills:

1) Machine Learning and Artificial Intelligence: Expand your knowledge by getting right into machine learning because computer vision and machine learning are closely correlated concepts. In addition, machine learning is one of the most fast growing industries and people who work in AI fields are earning a lot more money then programmers who work in other IT fields.
2) Game development: As you know computer vision, you can start learning one of the most engaging fields of programming, which is game development. In this field, you will have to learn how to create your own images, which is kind of related to image processing with OpenCV.
3) Master Video Processing and Real-time Applications**:** Move beyond static images and focus on video processing. Experiment with techniques for video stabilization, object tracking across frames, and real-time analysis for applications in surveillance, autonomous systems, and augmented reality.
4) Data Science: Also consider exploring data science with python libraries like NumPy, Pandas and Matplotlib. Python is dominating programming language in this field and its future looks bright.

Remember, proficiency in Python OpenCV is just the beginning of your journey into the expansive field of computer vision. Whether you aspire to build innovative applications, contribute to research breakthroughs, or simply enjoy exploring the visual world through code, your dedication and curiosity will continue to drive your success.

Thank you for choosing this book as your guide. Embrace the challenges ahead with enthusiasm, and let your passion for learning propel you toward new heights in the exciting realm of computer vision.

Happy coding and may your vision always be clear and insightful!