OpenCV tutorial Documentation

Release 2019

Raphael Holzer

Feb 09, 2020

Contents:

CHAPTER 1

Introduction

In this section we look at the basic operations for displaying images in a window and reacting to mouse and keyboard events.

1.1 Load and show an image

OpenCV is a library for image processing. We start this tutorial by opening a file and displaying it in a window.

First we import the OpenCV library `cv2` and give it the shortcut `cv`.

```
import cv2 as cv
```

Then we load an image from the current folder with the function `cv.imread` and display it with the function `cv.imshow` in a window called **window**.

```
img = cv.imread('messi.jpg')
cv.imshow('window', img)
```

You can download the image here:

`messi.jpg`

Without calling the `cv.waitKey()` no window is displayed. The parameter of this function is the number of miliseconds the function waits for a keypress. With a value of 0 the function waits indefinitely. Once a key is pressed, the program advances to the last line and destroys all windows.

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

Clicking the *window close* button closes the window, but does not quit the program. After closing the window, a key press has no effect anymore and the only way to quit the program is by choosing **Quit** from the (Python) menu, or by pressing the shortcut **cmd+Q**.

Listing 1: Here is the complete code.

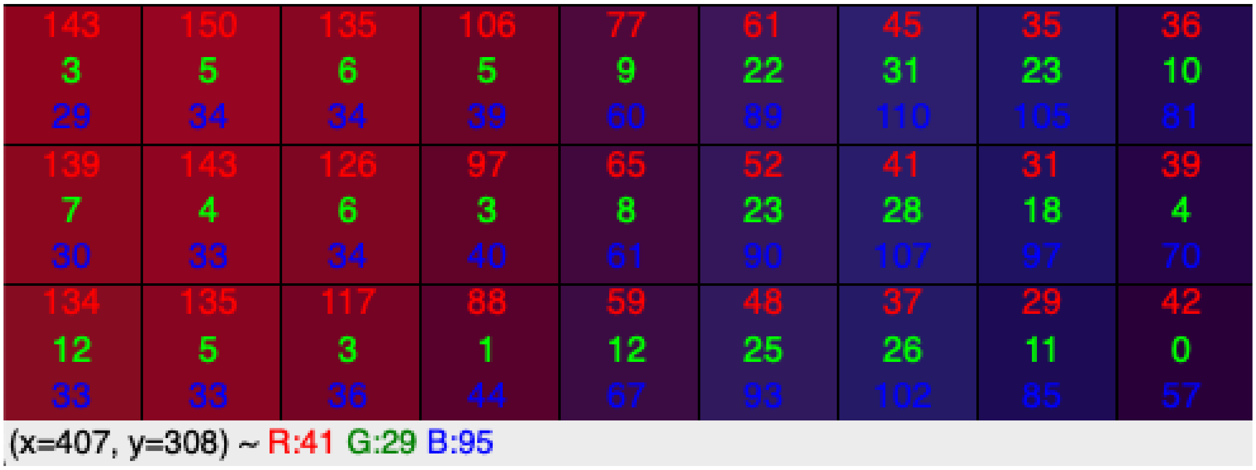
```
import cv2 as cv

img = cv.imread('messi.jpg')
cv.imshow('window', img)

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

`intro1.py`

1.2 What's a pixel?

Images are made of pixels. They are the colored dots that compose an image. If you zoom into an image you can see squares of uniform color. Use the mouse wheel and try to zoom into an OpenCV image.

It shows also the RGB color values at the mouse position (currently at R=41, G=29, B=95). To the left are reddish pixels, to the right are blueish pixels.



The status line shows the mouse position (currently at x=470, y=308). Move the mouse to explore the coordinate system. The origine (0, 0) is at the top left position.

- The x coordinate increases from left to right
- The y coordinate increases from top to bottom

The highest values are at the bottom right corner, which gives you the size of the image.

1.3 Save an image

Saving an image is very simple. Just use `imwrite(file, img)` and supply the file name with a recognized image format extension (.jpg, .png, .tiff). OpenCV automatically converts to the desired format.

To change the image to a grayscale image use this function:

```
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

```
import cv2 as cv

img = cv.imread('messi.jpg')
cv.imshow('window', img)

cv.imwrite('messi.png', img)
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imwrite('messi_gray.png', gray)

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

`intro1_save.py`

1.4 Capture live video

To capture video we must create a `VideoCapture` object. The index 0 refers to the default camera (built-in webcam):

```
cap = cv.VideoCapture(0)
```

Inside a loop we read the video capture to get frames. We then operate on the frame (convert to grayscale), then display the result, and then loop back. The loop finishes when **q** is pressed:

```
while True:
    # Capture frame-by-frame
    ret, frame = cap.read()

    # Our operations on the frame come here
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    # Display the resulting frame
    cv.imshow('window', frame)
    if cv.waitKey(1) & 0xFF == ord('q'):
        break
```

At the end the video stream is relased and all windows are closed:

```
# When everything done, release the capture
cap.release()
cv.destroyAllWindows()
```

Listing 2: Here is the complete code.

```
"""Capture video from camera."""
import cv2 as cv

cap = cv.VideoCapture(0)

while True:
    # Capture frame-by-frame
    ret, frame = cap.read()

    # Our operations on the frame come here
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    # Display the resulting frame
    cv.imshow('frame', frame)
    if cv.waitKey(1) & 0xFF == ord('q'):
        break

# When everything done, release the capture
cap.release()
cv.destroyAllWindows()
```

`intro2.py`

1.5 Add an overlay

An overlay can be added to a window to add a line of text during a certain time delay. This is the fonction:

```
cv.displayOverlay(window, text, delay=0)
```

The overlay text is white on black background, centered and can be displayed on multiple lines:

```
cv.displayOverlay('window', 'line 1\nline 2\nline 3')
```

```
# Add an overlay
import cv2 as cv

file = 'messi.jpg'
img = cv.imread(file, cv.IMREAD_COLOR)

cv.imshow('window', img)
cv.displayOverlay('window', f'file name: {file}')

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

`overlay.py`

The following program adds the following information:

- the file name
- the width of the image (in pixels)
- the height of the image
- the number of channels (3 for RGB)

OpenCV images are Numpy arrays:

```
>>>type(img)
```

<class 'numpy.ndarray'>

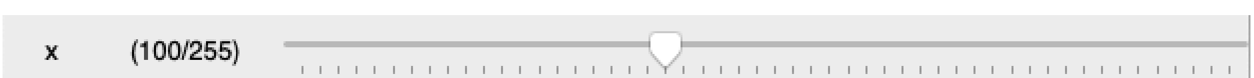
Such an array has the attribute `shape` which returns the array dimensions.


```
# Add an overlay
import cv2 as cv

file = 'messi.jpg'
img = cv.imread(file, cv.IMREAD_COLOR)

cv.imshow('window', img)
text = f'file name: {file}\n\
        width: {img.shape[1]}\n\
        height: {img.shape[0]}\n\
        channels: {img.shape[2]}'

cv.displayOverlay('window', text)

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

overlay2.py

1.6 Add a trackbar

A trackbar is a slider added at the bottom of the window.



The function takes the following arguments:

```
cv.createTrackbar(name, window, value, maxvalue, callback)
```

- the trackbar **name**
- the **window** where to add the trackbar
- the initial **value**
- the maximum value **maxvalue** on a scale starting at 0
- the **callback** function called if the slider is moved

The `createTrackbar` command adds a trackbar below the main image. It goes from 0 to 255 and we set the initial value to 100. When the trackbar is moved, it calls a callback function named `trackbar`:

```
cv.createTrackbar('x', 'window', 100, 255, trackbar)
```

The callback function `trackbar` displays the trackbar position in the overlay region on getTrackbarPosof the window:

```
def trackbar(x):
    """Trackbar callback function."""
    text = f'Trackbar: {x}'
    cv.displayOverlay('window', text, 1000)
    cv.imshow('window', img)
```

The function `cv.imshow` is used to force an update of the window.

```
# Add a trackbar
import cv2 as cv

def trackbar(x):
    """Trackbar callback function."""
    text = f'Trackbar: {x}'
    cv.displayOverlay('window', text, 1000)
    cv.imshow('window', img)

img = cv.imread('messi.jpg', cv.IMREAD_COLOR)
cv.imshow('window', img)
cv.createTrackbar('x', 'window', 100, 255, trackbar)

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

`trackbar.py`

1.7 Compose an RGB color

We can use three trackbars for composing a color. First we use the Numpy `zero()` function to create a black image with a dimension of (100, 600).

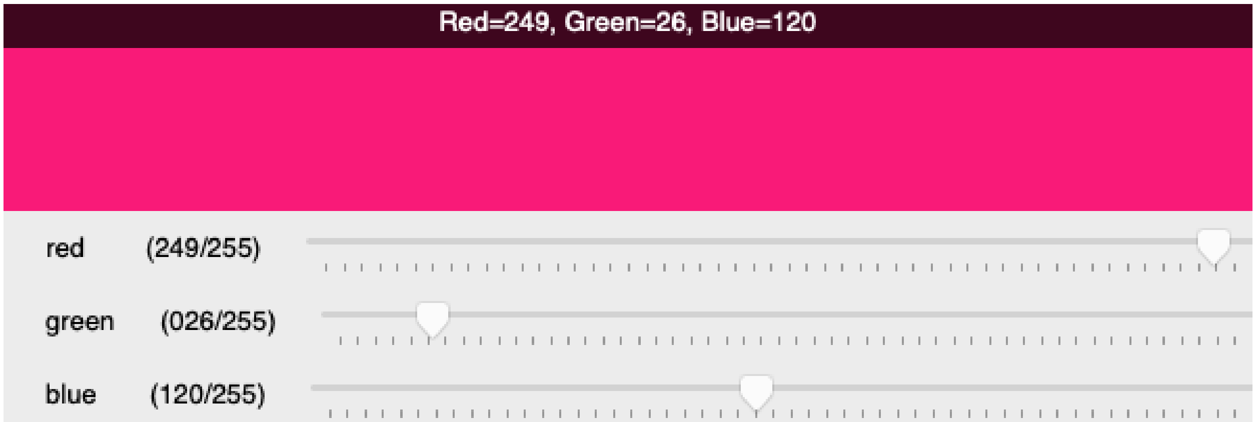
```
img = np.zeros((100, 600, 3), 'uint8')
```

Inside the trackbar callback function `rgb` we get the 3 trackbar positions with the red, green and blue color components which can vary from 0 to 255.

```
r = cv.getTrackbarPos('red','window')
g = cv.getTrackbarPos('green','window')
b = cv.getTrackbarPos('blue','window')
```

Then we reset the image array with the new color value. OpenCV uses the BGR order. Be careful to use the right order.

```
img[:] = [b, g, r]
```




```
# Compose an RGB color with 3 trackbars
import cv2 as cv
import numpy as np

def rgb(x):
    """Trackbar callback function."""
    r = cv.getTrackbarPos('red','window')
    g = cv.getTrackbarPos('green','window')
    b = cv.getTrackbarPos('blue','window')
    img[:] = [b, g, r]
    cv.displayOverlay('window', f'Red={r}, Green={g}, Blue={b}')
    cv.imshow('window', img)

img = np.zeros((100, 600, 3), 'uint8')
cv.imshow('window', img)
cv.createTrackbar('red', 'window', 200, 255, rgb)
cv.createTrackbar('green', 'window', 50, 255, rgb)
cv.createTrackbar('blue', 'window', 100, 255, rgb)
rgb(0)

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

`trackbar_rgb.py`

1.8 Catch mouse events

The `setMouseCallback` function attaches a mouse callback function to the *image* window:

```
cv.setMouseCallback('window, mouse)
```

This is the callback definition:

```
def mouse(event, x, y, flags, param):
    """Mouse callback function."""
    text = f'mouse at ({x}, {y}), flags={flags}, param={param}'
    cv.displayStatusBar('window', 'Statusbar: ' + text, 1000)
```

Listing 3: Here is the complete code.

```
"""Catch mouse events and write to statusbar."""
import cv2 as cv

def mouse(event, x, y, flags, param):
    """Mouse callback function."""
    text = f'mouse at ({x}, {y}), flags={flags}, param={param}'
    cv.displayOverlay('window', 'Overlay: ' + text, 1000)

img = cv.imread('messi.jpg')
cv.imshow('window', img)
cv.setMouseCallback('window', mouse)

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

`intro3.py`

1.9 Draw with the mouse

Now we can use the mouse to change the pixel color at the mouse position. We can make a simple drawing program. When the mouse button is pressed, the flag is set to 1. We use an `if` statement to set the current pixel at (x, y) to red when the mouse button is pressed.

```
if flags == 1:
    img[y, x] = [0, 0, 255]
```

Notice: OpenCV uses the color ordering BGR, so you must specify the red component last.

This is an image with a red outline drawn with the mouse.


```
"""Draw pixels with the mouse."""
import cv2 as cv

def mouse(event, x, y, flags, param):
    text = f'Mouse at ({x}, {y}), flags={flags}, param={param}'
    cv.displayOverlay('window', text, 1000)
    if flags == 1:
        img[y, x] = [0, 0, 255]
    cv.imshow('window', img)

img = cv.imread('messi.jpg')
cv.imshow('window', img)
cv.setMouseCallback('window', mouse)

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

intro3b.py

1.10 Access a slice of the image

The slice operator (:) allows to address rectangular areas of a Numpy array. The command:

```
img[250:300, 50:550] = (0, 255, 0)
```

specifies the rectangle with y values from 250 to 300 and x values from 50 to 500. It sets these pixels to green.

Next We use it to extract the area containing the face. This sub-region is then inserted elsewhere in the image.


```
"""Acces a slice of the image."""
import cv2 as cv
img = cv.imread('messi.jpg')

img[250:300, 50:550] = (0, 255, 0)
face = img[80:230, 270:390]
img[0:150, 0:120] = face

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

intro3c.py

1.11 Object-Oriented Programming

From now on we will use object-oriented programming (OOP) techniques. We define an App class which loads an image and creates a window.

```
class App:
    def __init__(self):
        img = cv.imread('messi.jpg')
        Window('image', img)

```

(continues on next page)

(continued from previous page)

```
    def run(self):
        """Run the main event loop."""
        k=0
        while k != ord('q'):
            k = cv.waitKey(0)
            print(k, chr(k))

        cv.destroyAllWindows()
```

The `run` method prints the key code and the key character. When a `q` is pressed the program quits.

The `Window` class stores window name and image and shows the image in a window.

```
class Window:
    """Create a window with an image."""
    def __init__(self, win, img):
        self.win = win
        self.img = img
        cv.imshow(win, img)
```

The last two lines instantiate the app with `App()` and call the `run()` method:

```
if __name__ == '__main__':
    App().run()
```

`intro4.py`

1.12 Patterns

These are the patterns for reading, displaying and saving images:

```
img = cv.imread('file', type)
cv.imshow('win', img)
cv.imwrite('file', img)
```

Interface:

```
cv.namedWindow('win', type)
cv.waitKey(ms)
cv.destroyAllWindows()
```

Video capture:

```
cap = cv.VideoCapture(0)
cap.isOpened()
cap.get(id)
cap.set(id, val)
ret, frame = cap.read()
cap.release()

img2 = cv.cvtColor(img, type)
```

Drawing functions:

```
cv.line(img, p0, p1, col, d)
cv.circle(img, c0, r, col, d)
cv.ellipse(img, p0, (w, h), a
cv.polylines(img, [pts], True, col)

font = cv.FONT_
cv.putText(img, str, pos, font, size, col)
```

Mouse callback:

```
cv.setMouseCallback('img', cb)
cb(evt, x, y, flags, param)

cv.createTrackbar('name', 'win', 0, max, cb)
cv.getTrackbarPos('name', 'win')
```

CHAPTER 2

Drawing shapes

OpenCV has different drawing functions to draw:

- lines
- circle
- rectangle
- ellipse
- text

2.1 Using Numpy

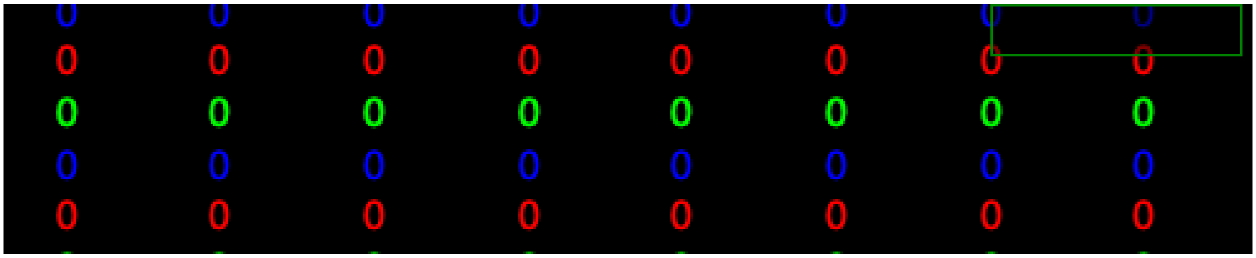
Numpy is a very powerful math module for dealing with multi-dimensional data such as vectors and images. The OpenCV images are represented as Numpy arrays. At the start of a program we import both:

```
import cv2 as cv
import numpy as np
```

To create an empty color image we create a 3D array of zeroes:

```
img = img = np.zeros((100, 600, 3), np.uint8)
cv.imshow('RGB', img)
```

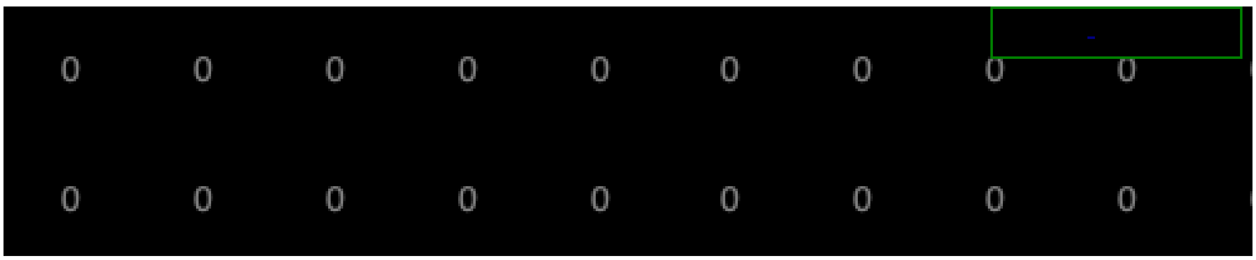
When zooming, we can see the 3 color components.



To create an empty gray-scale image we create a 2D array of zeroes:

```
gray_img = np.zeros((100, 600), np.uint8)
cv.imshow('Gray', gray_img)
```

The grayscale values for each pixel go from 0 to 255. In a black image all pixel values are 0.



```
import cv2 as cv
import numpy as np

img = img = np.zeros((100, 500, 3), np.uint8)
cv.imshow('RGB', img)

gray_img = np.zeros((100, 500), np.uint8)
cv.imshow('Gray', gray_img)

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

draw1.py

2.2 Define colors

Colors are defined by three base colors: Blue, Green and Red. All three put to zero gives black, all three at the maximum gives white:

```
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
```

Different from the more common RGB ordering, OpenCV uses the ordering BGR:

```
RED = (0, 0, 255)
GREEN = (0, 255, 0)
BLUE = (255, 0, 0)
```

Mixing color components results in more colors:

```
CYAN = (255, 255, 0)
MAGENTA = (255, 0, 255)
YELLOW = (0, 255, 255)
```

2.3 Draw a line

The function `cv.line()` adds a line to an image:

```
cv.line(image, p0, p1, color, thickness)
```

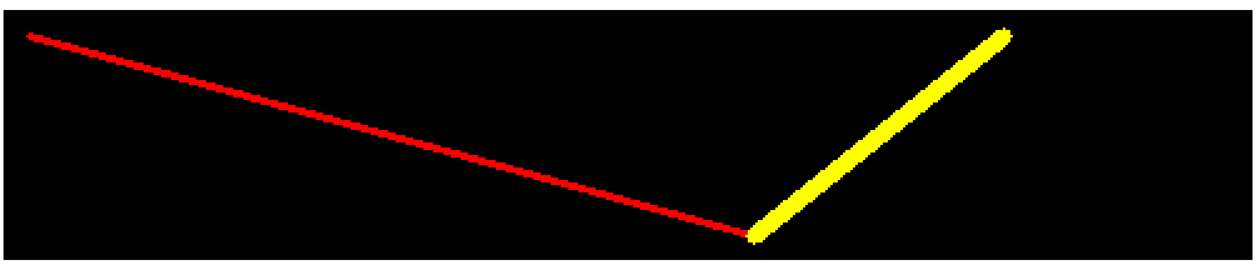
- **image** where the line is added
- start point **p0**
- end point **p1**
- line **color**
- line **thickness**

Lets define three points:

```
p0 = 10, 10
p1 = 300, 90
p2 = 500, 10
```

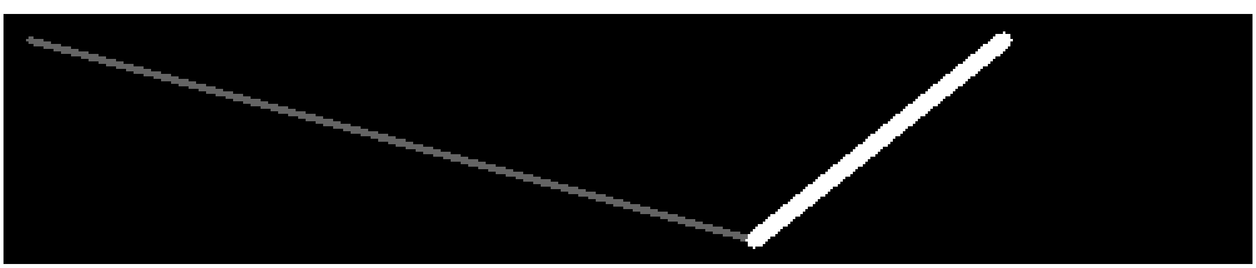
Now we can draw two colored lines:

```
cv.line(img, p0, p1, RED, 2)
cv.line(img, p1, p2, YELLOW, 5)
```



If the image is a gray-scale image, instead of the color triplet, a grayscale value from 0 (black) to 255 (white) is used:

```
cv.line(gray_img, p0, p1, 100, 2)
cv.line(gray_img, p1, p2, 255,5)
```



```
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
YELLOW = (0, 255, 255)

p0, p1, p2 = (10, 10), (300, 90), (400, 10)

img = img = np.zeros((100, 500, 3), np.uint8)
cv.line(img, p0, p1, RED, 2)
cv.line(img, p1, p2, YELLOW, 5)
cv.imshow('RGB', img)

gray_img = np.zeros((100, 500), np.uint8)
cv.line(gray_img, p0, p1, 100, 2)
cv.line(gray_img, p1, p2, 255,5)
cv.imshow('Gray', gray_img)

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

line1.py

2.4 Select thickness with a trackbar

We can use a trackbar to set the thickness of the line. The trackbar callback function has one argument - the line thickness. Trackbar arguments are always integer and they always start at 0. However, for lines the smallest line thickness is 1. Therefore we use the `max` function to make the thickness at least 1. To have a numeric feedback, we display the thickness value also in the overlay:

```
def trackbar(x):
    x = max(1, x)
    cv.displayOverlay('window', f'thickness={x}')
```

Next we have to redraw the line. We start by resetting the image to 0. Then we draw the line and display the new image:

```
img[:] = 0
cv.line(img, p0, p1, RED, x)
cv.imshow('window', img)
```

```
# Select line thickness with a trackbar
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
p0, p1 = (100, 30), (400, 90)

def trackbar(x):
    x = max(1, x)
    cv.displayOverlay('window', f'thickness={x}')
    img[:] = 0
    cv.line(img, p0, p1, RED, x)
    cv.imshow('window', img)

img = np.zeros((100, 500, 3), np.uint8)
cv.line(img, p0, p1, RED, 2)
cv.imshow('window', img)
cv.createTrackbar('thickness', 'window', 2, 20, trackbar)

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

line2.py

2.5 Select color with a trackbar

We can use a trackbar to set the line color. The trackbar is used to select an index into a color list which we define with 7 colors:

```
colors = (RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW, WHITE)
```

The trackbar is defined to return an integer value from 0 to 6:

```
cv.createTrackbar('color', 'window', 0, 6, trackbar)
```

Inside the trackbar callback function we use the index to look up the color. To give numeric feedback we display the color RGB value in the overlay:

```
def trackbar(x):
    color = colors[x]
    cv.displayOverlay('window', f'color={color}')
```

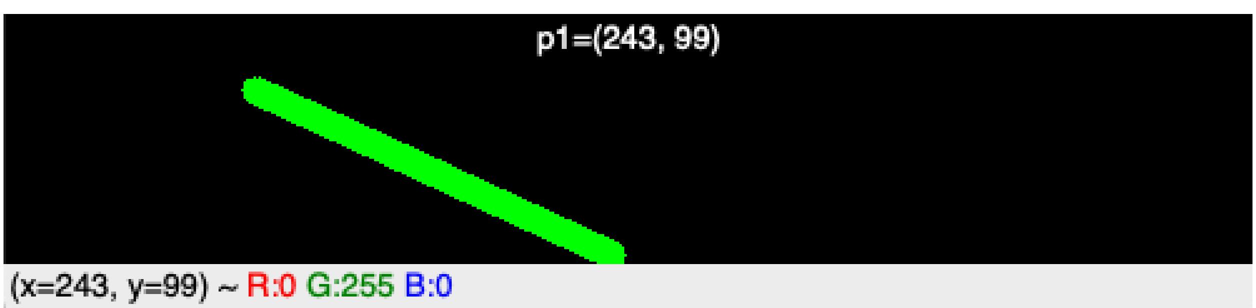
```
# Select line color with a trackbar
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
GREEN = (0, 255, 0)
BLUE = (255, 0, 0)
CYAN = (255, 255, 0)
MAGENTA = (255, 0, 255)
YELLOW = (0, 255, 255)
WHITE = (255, 255, 255)

colors = (RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW, WHITE)
p0, p1 = (100, 30), (400, 90)

def trackbar(x):
    color = colors[x]
    cv.displayOverlay('window', f'color={color}')
    img[:] = 0
    cv.line(img, p0, p1, color, 10)
    cv.imshow('window', img)

img = np.zeros((100, 500, 3), np.uint8)
cv.line(img, p0, p1, RED, 10)
cv.imshow('window', img)
cv.createTrackbar('color', 'window', 0, 6, trackbar)

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

line3.py

2.6 Select end point with the mouse

Let's use the mouse for selecting the end point of the line. The mouse callback function has the x and y coordinates as arguments. We are only interesed in the mouse movement when a button is pressed (flags==1). The current mouse coordinates will be the new end point for the line. We display this coordinates in the overlay:

```
def mouse(event, x, y, flags, param):
    if flags == 1:
        p1 = x, y
        cv.displayOverlay('window', f'p1=({x}, {y})')
```



```
# Select end point with the mouse
import cv2 as cv
import numpy as np

GREEN = (0, 255, 0)
p0, p1 = (100, 30), (400, 90)

def mouse(event, x, y, flags, param):
    if flags == 1:
        p1 = x, y
        cv.displayOverlay('window', f'p1=({x}, {y})')
        img[:] = 0
        cv.line(img, p0, p1, GREEN, 10)
        cv.imshow('window', img)

img = np.zeros((100, 500, 3), np.uint8)
cv.line(img, p0, p1, GREEN, 10)
cv.imshow('window', img)
cv.setMouseCallback('window', mouse)

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

line4.py

2.7 Draw a complete line

Let's now draw a complete line with the mouse. Now we need to distinguish between mouse down, mouse move and mouse up events. When the mouse is down, we start a new line and set both points `p0` and `p1` to the current mouse position:

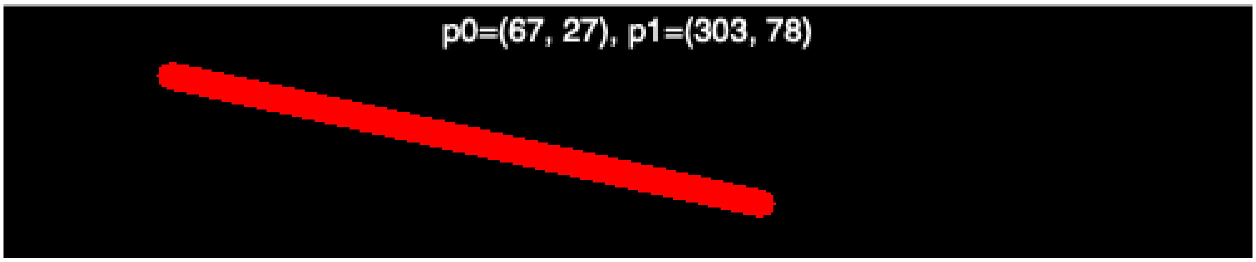
```
if event == cv.EVENT_LBUTTONDOWN:
     p0 = x, y
     p1 = x, y
```

If the mouse moves (with the button pressed) or if the mouse button goes up, we only set `p1` to the new mouse position:

```
elif event == cv.EVENT_MOUSEMOVE and flags == 1:
    p1 = x, y

elif event == cv.EVENT_LBUTTONUP:
    p1 = x, y
```

At the end of the mouse callback function we reset the image to zero (black), draw the line, display the new image and show the two points in the overlay:

```
img[:] = 0
cv.line(img, p0, p1, RED, 10)
cv.imshow('window', img)
cv.displayOverlay('window', f'p0={p0}, p1={p1}')
```




```
# Draw a complete line with the mouse
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
p0, p1 = (100, 30), (400, 90)

def mouse(event, x, y, flags, param):
    global p0, p1

    if event == cv.EVENT_LBUTTONDOWN:
        p0 = x, y
        p1 = x, y

    elif event == cv.EVENT_MOUSEMOVE and flags == 1:
        p1 = x, y

    elif event == cv.EVENT_LBUTTONUP:
        p1 = x, y

    img[:] = 0
    cv.line(img, p0, p1, RED, 10)
    cv.imshow('window', img)
    cv.displayOverlay('window', f'p0={p0}, p1={p1}')

img = np.zeros((100, 500, 3), np.uint8)
cv.imshow('window', img)
cv.setMouseCallback('window', mouse)

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

line5.py

2.8 Draw multiple lines

How do we do to draw multiple lines to an image? First we need to have a temporary copy `img0` which contains the lines of the previous stage of the drawing:

```
img0 = np.zeros((100, 500, 3), np.uint8)
img = img0.copy()
```

When the mouse button is down, we set the two points `p0` and `p1` to the current mouse position:

```
if event == cv.EVENT_LBUTTONDOWN:
    p0 = x, y
```

(continues on next page)

(continued from previous page)

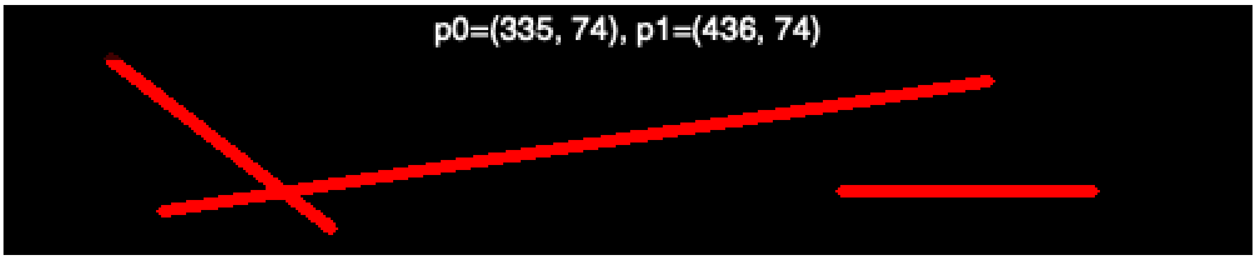
```
    p1 = x, y
```

When the mouse moves, we reset the current image to the previous image `img0` and draw a blue line of thickness 2:

```
elif event == cv.EVENT_MOUSEMOVE and flags == 1:
    p1 = x, y
    img[:] = img0
    cv.line(img, p0, p1, BLUE, 2)
```

When the mouse goes up, we reset the image to the previous image `img0`, draw a red line of thickness 4, and save this new image as `img0`:

```
elif event == cv.EVENT_LBUTTONUP:
    img[:] = img0
    cv.line(img, p0, p1, RED, 4)
    img0[:] = img
```




```
# Draw multiple lines with the mouse
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
BLUE = (255, 0, 0)
p0, p1 = (100, 30), (400, 90)

def mouse(event, x, y, flags, param):
    global p0, p1

    if event == cv.EVENT_LBUTTONDOWN:
        p0 = x, y
        p1 = x, y

    elif event == cv.EVENT_MOUSEMOVE and flags == 1:
        p1 = x, y
        img[:] = img0
        cv.line(img, p0, p1, BLUE, 2)

    elif event == cv.EVENT_LBUTTONUP:
        img[:] = img0
        cv.line(img, p0, p1, RED, 4)
        img0[:] = img

    cv.imshow('window', img)
    cv.displayOverlay('window', f'p0={p0}, p1={p1}')

img0 = np.zeros((100, 500, 3), np.uint8)
img = img0.copy()
```

(continues on next page)

(continued from previous page)

```
cv.imshow('window', img)
cv.setMouseCallback('window', mouse)

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

line6.py

2.9 Draw a rectangle

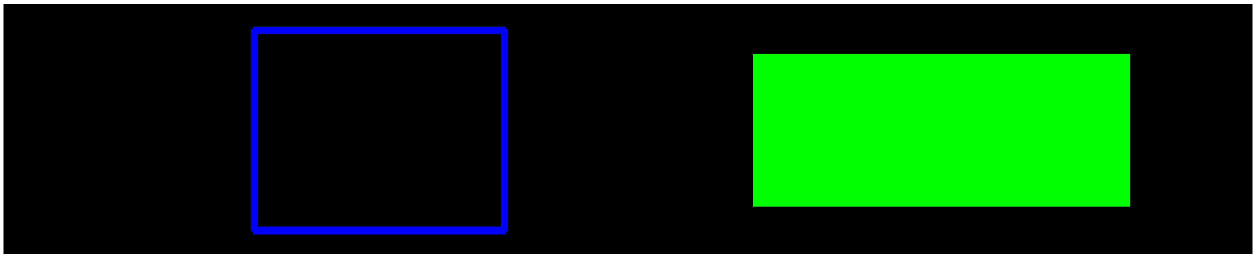
The function `cv.rectangle()` adds a rectangle to an image:

```
cv.rectangle(image, p0, p1, color, thickness)
```

- **image** where the rectangle is added
- corner point **p0**
- corner point **p1**
- ouline **color**
- line **thickness**

If the line thickness is negative or `cv.FILLED` the rectangle is filled:

```
cv.rectangle(img, p0, p1, BLUE, 2)
cv.rectangle(img, p2, p3, GREEN, cv.FILLED)
```



```
import cv2 as cv
import numpy as np

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

RED = (0, 0, 255)
GREEN = (0, 255, 0)
BLUE = (255, 0, 0)

CYAN = (255, 255, 0)
MAGENTA = (255, 0, 255)
YELLOW = (0, 255, 255)

p0 = 100, 10
p1 = 200, 90
p2 = 300, 20
p3 = 450, 80
```

(continues on next page)

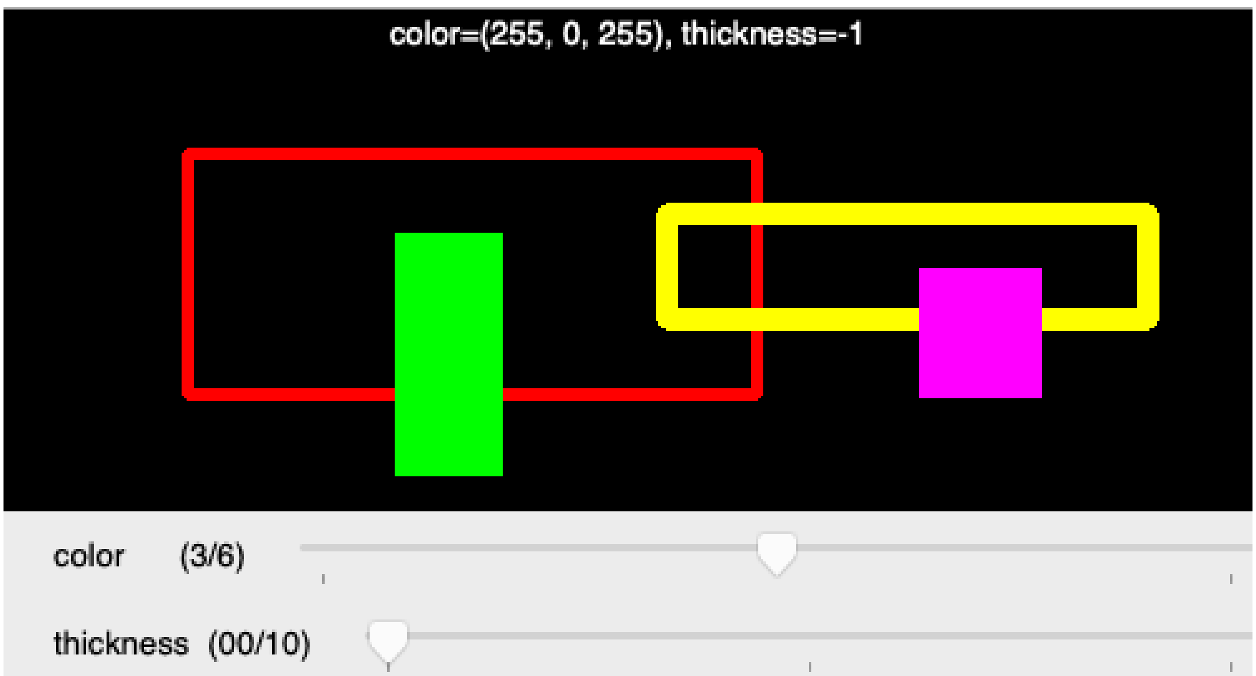
(continued from previous page)

```
img = img = np.zeros((100, 500, 3), np.uint8)
cv.rectangle(img, p0, p1, BLUE, 2)
cv.rectangle(img, p2, p3, GREEN, cv.FILLED)
cv.imshow('RGB', img)

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

rect1.py

2.10 Draw multiple rectangles

Now we combine thickness and color trackbar as welle as the mouse callback to create multple rectangles.




```
import cv2 as cv
import numpy as np
from draw import *

def draw(x):
    global p0, p1
    d = cv.getTrackbarPos('thickness', 'window')
    d = -1 if d==0 else d
    i = cv.getTrackbarPos('color', 'window')
    color = colors[i]
    img[:] = img0
    cv.rectangle(img, p0, p1, color, d)
    cv.imshow('window', img)
    text = f'color={color}, thickness={d}'
    cv.displayOverlay('window', text)
```

(continues on next page)

(continued from previous page)

```
def mouse(event, x, y, flags, param):
    global p0, p1
    if event == cv.EVENT_LBUTTONDOWN:
        img0[:] = img
        p0 = x, y
    elif event == cv.EVENT_MOUSEMOVE and flags == 1:
        p1 = x, y
    elif event == cv.EVENT_LBUTTONUP:
        p1 = x, y
    draw(0)

cv.setMouseCallback('window', mouse)
cv.createTrackbar('color', 'window', 0, 6, draw)
cv.createTrackbar('thickness', 'window', 0, 10, draw)

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

`rect2.py`

The common code such as color definitions and image creation has been placed in a separate file.

```
import numpy as np
import cv2 as cv

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

RED = (0, 0, 255)
GREEN = (0, 255, 0)
BLUE = (255, 0, 0)

CYAN = (255, 255, 0)
MAGENTA = (255, 0, 255)
YELLOW = (0, 255, 255)

colors = (RED, GREEN, BLUE, MAGENTA, CYAN, YELLOW, WHITE)
p0 = p1 = 0, 0

img0 = np.zeros((200, 500, 3), np.uint8)
img = img0.copy()
cv.imshow('window', img)
```

`draw.py`

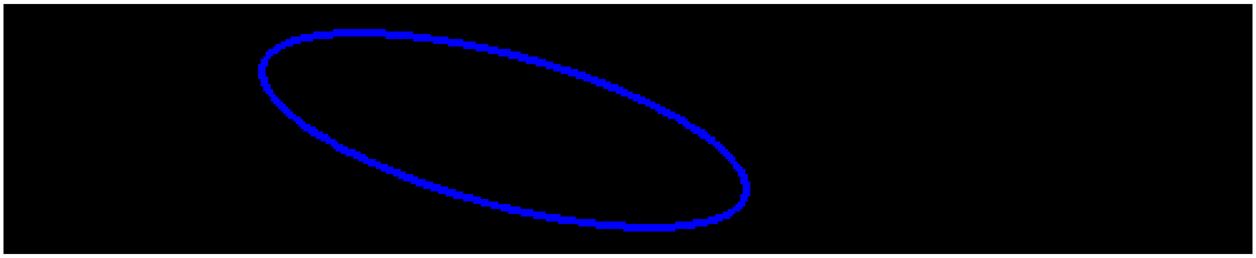
2.11 Draw an ellipse

The function `cv.ellipes()` adds an ellipse to an image:

```
cv.ellipse(img, center, axes, angle, a0, a1, color, thickness)
```

- **image** where the ellipse is added
- **center** point
- the two **axes**

- the axis orientation **angle**
- the beginning angle **a0**
- the ending angle **a1**
- ouline **color**
- line **thickness**



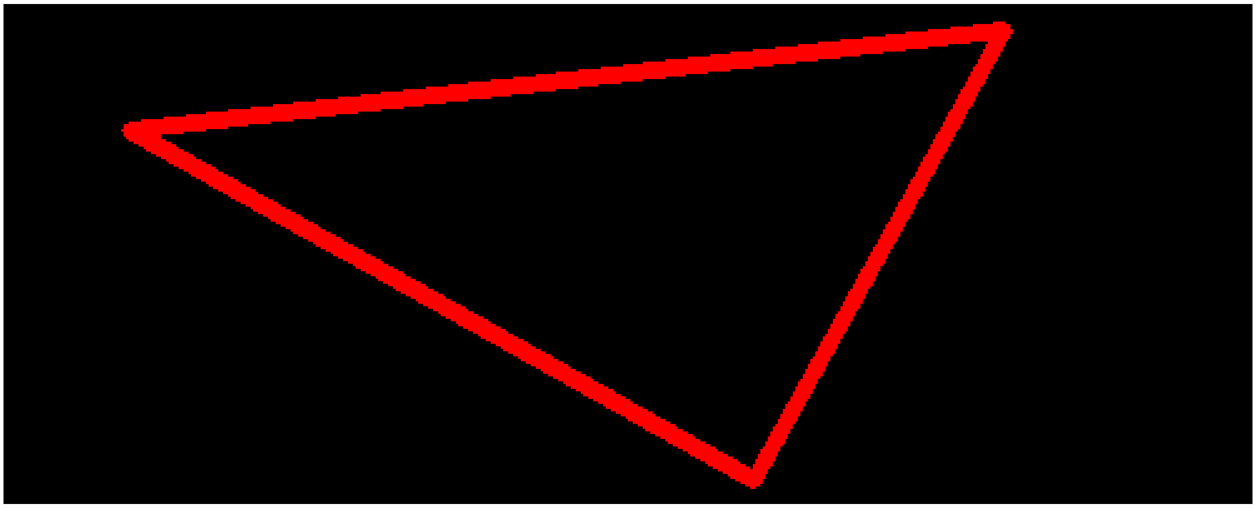
```
import cv2 as cv
import numpy as np

BLUE = (255, 0, 0)
center = 200, 50
axes = 100, 30
angle = 15

img = img = np.zeros((100, 500, 3), np.uint8)
cv.ellipse(img, center, axes, angle, 0, 360, BLUE, 2)
cv.imshow('RGB', img)

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

draw4.py

2.12 Draw a polygon

The `polylines` function expects a Numpy array for the point list:

```
pts = [(50, 50), (300, 190), (400, 10)]
cv.polylines(img, np.array([pts]), True, RED, 5)
```



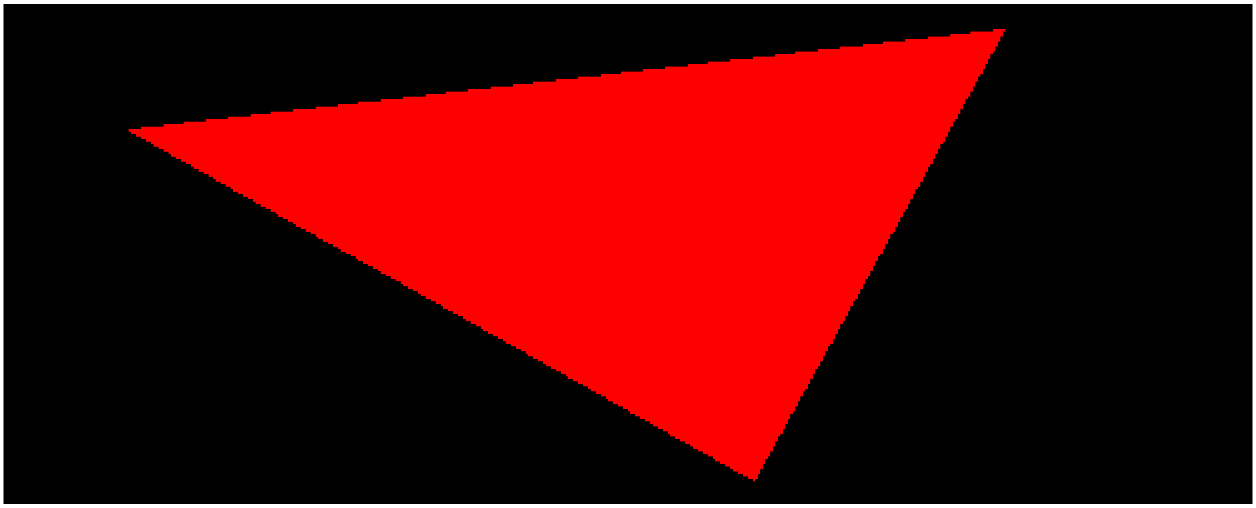
```
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
pts = [(50, 50), (300, 190), (400, 10)]

img = img = np.zeros((200, 500, 3), np.uint8)
cv.polylines(img, np.array([pts]), True, RED, 5)
cv.imshow('window', img)

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

`polygon1.py`

2.13 Draw a filled polygon

The `polylines` function expects a Numpy array for the point list:

```
pts = [(50, 50), (300, 190), (400, 10)]
cv.polylines(img, np.array([pts]), True, RED, 5)
```



```
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
pts = [(50, 50), (300, 190), (400, 10)]

img = img = np.zeros((200, 500, 3), np.uint8)
cv.fillPoly(img, np.array([pts]), RED)
cv.imshow('window', img)

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

polygon2.py

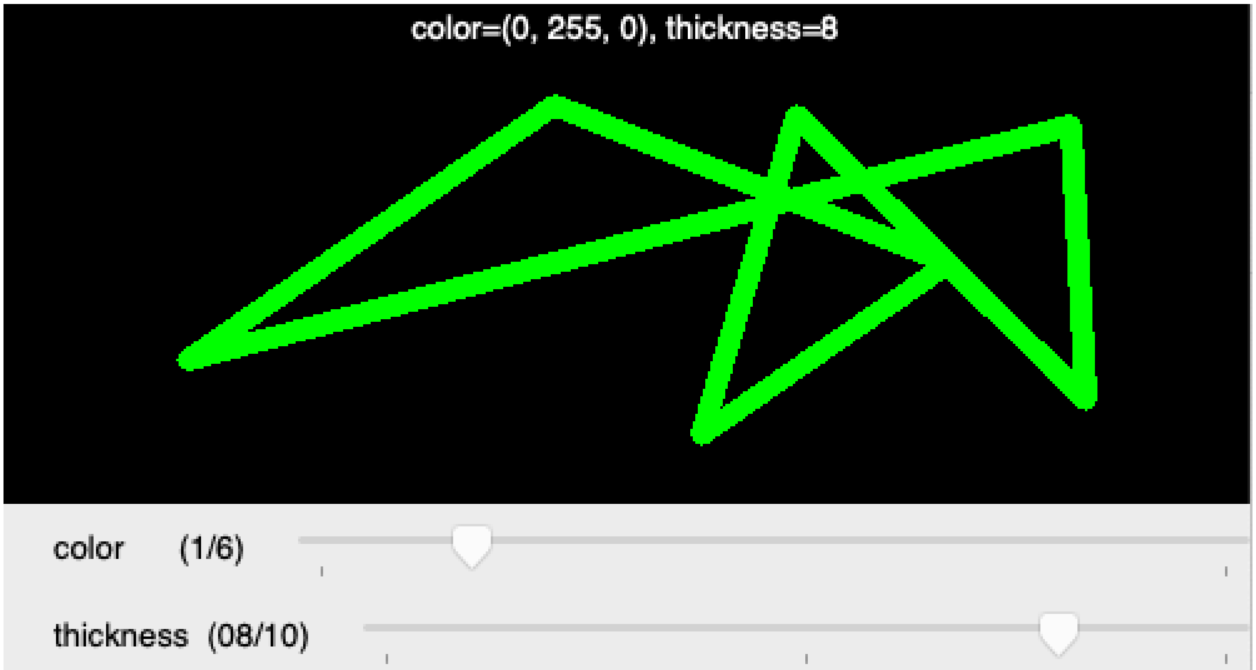
2.14 Draw a polygon with the mouse

Combining the previous techniques, it is rather simple to draw a polygon just by clicking into the window. First we define an empty list:

```
pts = []
```

Each time we click with the mouse we append a point:

```
def mouse(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        pts.append((x, y))
        draw(0)
```




```
import cv2 as cv
import numpy as np
from draw import *

pts = []

def draw(x):
    d = cv.getTrackbarPos('thickness', 'window')
    d = -1 if d==0 else d
    i = cv.getTrackbarPos('color', 'window')
    color = colors[i]
    img[:] = img0
    cv.polylines(img, np.array([pts]), True, color, d)
    cv.imshow('window', img)
    text = f'color={color}, thickness={d}'
    cv.displayOverlay('window', text)

def mouse(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        pts.append((x, y))
        draw(0)

cv.setMouseCallback('window', mouse)
cv.createTrackbar('color', 'window', 0, 6, draw)
cv.createTrackbar('thickness', 'window', 2, 10, draw)
draw(0)

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

polygon3.py

2.15 Draw text


```
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
p0 = (10, 100)

font = cv.FONT_HERSHEY_SIMPLEX
img = np.zeros((200, 500, 3), np.uint8)
cv.putText(img,'OpenCV', p0, font, 4, RED, 2, cv.LINE_AA)
cv.imshow('window', img)

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

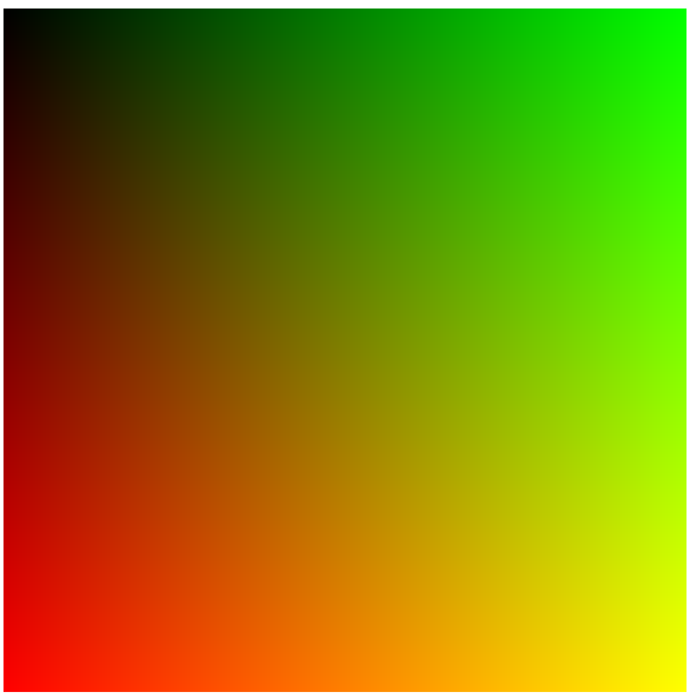
text1.py

Reference: https://docs.opencv.org/4.2.0/d6/d6e/group__imgproc__draw.html

CHAPTER 3

Color spaces

In the following image BGR = (z, h, v) blue is zero green increases in the horizontal direction, red increases in the horizhontal direction. We have black, red, green and yellow in the 4 corners.



```
# Compose an RGB color with 3 trackbars
import cv2 as cv
import numpy as np

v = np.fromfunction(lambda i, j: i, (256, 256), dtype=np.uint8)
h = np.fromfunction(lambda i, j: j, (256, 256), dtype=np.uint8)
z = np.zeros((256, 256), dtype=np.uint8)
```

(continues on next page)

(continued from previous page)

```
img = cv.merge([z, h, v])
cv.imshow('window', img)

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

rgb2.py

3.1 Sliding through the color cube

The RGB colors space is a cube of dimension $256 x 256 x 256$. In the following program we display **Blue** and **Green** and use the trackbar to select the **Red** component.

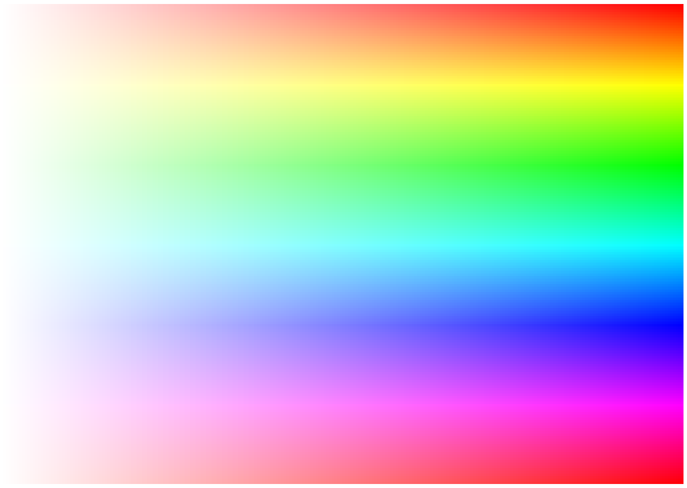
color/rgb3.png

```
# Trackbar to go through 1 axis
import cv2 as cv
import numpy as np

def trackbar(x):
    img[:, :, 2] = x
    cv.imshow('window', img)

img = np.zeros((256, 256, 3), dtype=np.uint8)

for i in range(256):
    img[i, :, 0] = i
    img[:, i, 1] = i

cv.imshow('window', img)
cv.createTrackbar('red', 'window', 0, 255, trackbar)

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

rgb3.py

3.2 The HSV colorspace

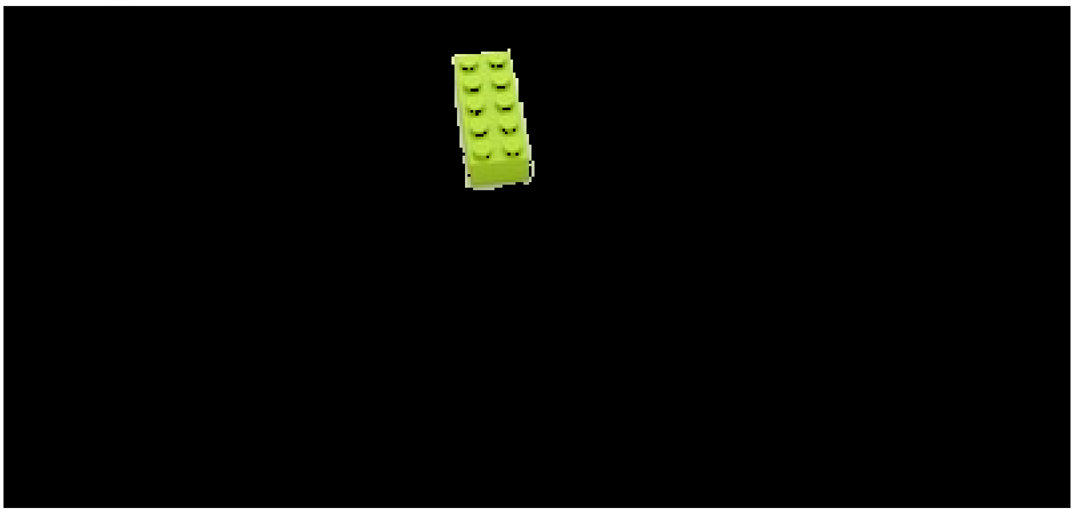
The HSV color space is a cube of dimension 180x256x256.



```
# Trackbar to go through 1 axis
import cv2 as cv
import numpy as np

def trackbar(x):
    img[:, :, 2] = x
    rgb = cv.cvtColor(img, cv.COLOR_HSV2BGR)
    cv.imshow('window', rgb)

img = np.zeros((180, 256, 3), dtype=np.uint8)

for i in range(180):
    img[i, :, 0] = i

for i in range(256):
    img[:, i, 1] = i

cv.imshow('window', img)
cv.createTrackbar('saturation', 'window', 0, 255, trackbar)

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

hsv2.py

3.3 Extracting an object based on hue



```
# Extract an object in HSV color space based on hue
import cv2 as cv
import numpy as np

img = cv.imread('legos.jpg')
hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)

def trackbar(x):
    lower = (x, 30, 30)
    upper = (x+5, 250, 250)
    mask = cv.inRange(hsv, lower, upper)
    img2 = cv.bitwise_and(img, img, mask=mask)
    cv.imshow('window', np.vstack([img, img2]))

cv.imshow('window', img)
cv.createTrackbar('hue', 'window', 0, 179, trackbar)

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

`hsv3.py`

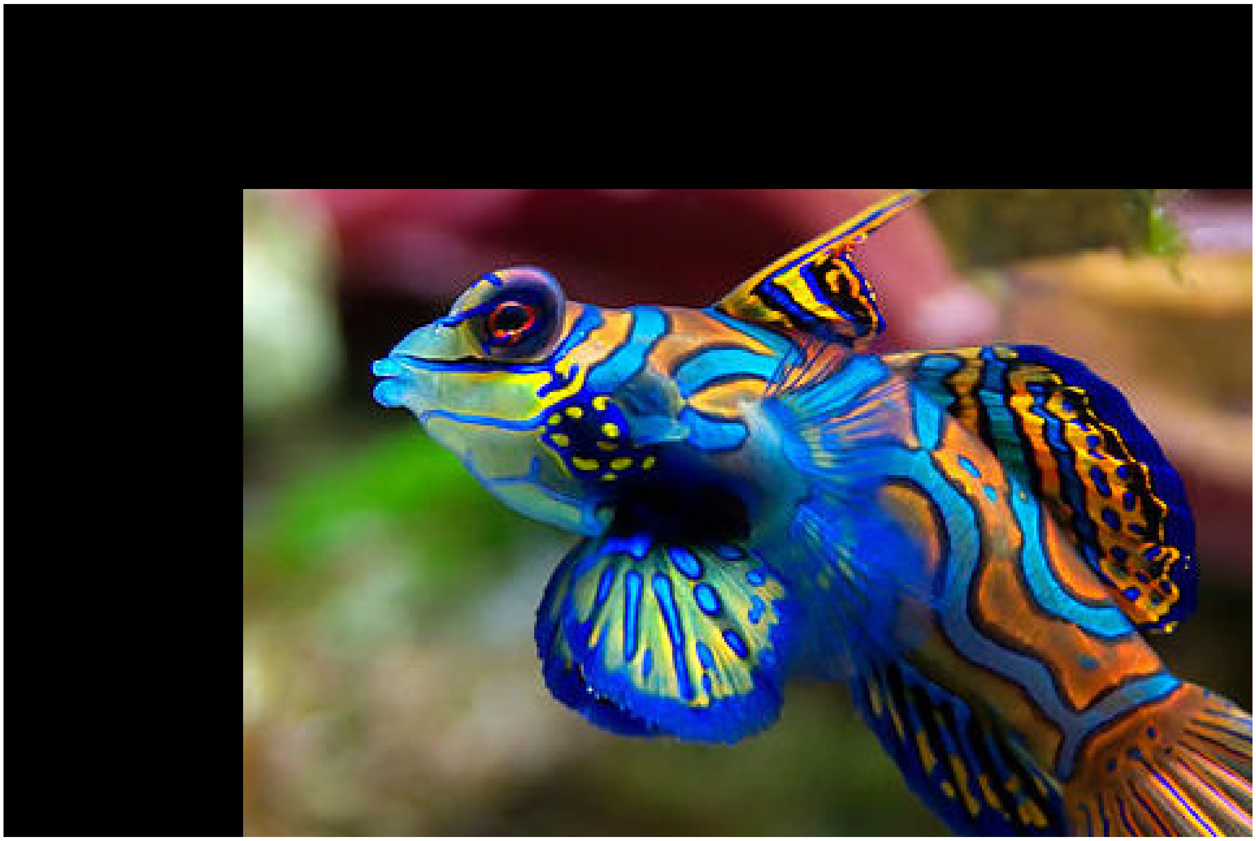
CHAPTER 4

Image transformation

4.1 Translation

Translating an image is shifting it along the x and y axes. A affine transformation can be obtained by using a transformation matrix `M`. It is a translation matrix which shifts the image by the vector (x, y). The first row of the matrix is [1, 0, x], the second is [0, 1, y]

```
M = np.float32([[1, 0, x], [0, 1, y]])
shifted = cv.warpAffine(img, M, size)
```



fish.jpg

transform1.py

4.2 Rotation

When we rotate an image we need to specify the center of rotation. Here we take the center of the image:

```
h, w = img.shape[:2]
center = w//2, h//2
```

To obtain the rotation matrix we use the function `cv.getRotationMatrix2D`. It takes three arguments:

- the rotation center,
- the roation angle and
- the scale factor

```python
"""rotation an image using the trackbar."""
import cv2 as cv

def trackbar(angle):
    M = cv.getRotationMatrix2D(center, angle, 1.0)
    rotated = cv.warpAffine(img, M, (w, h))
    cv.imshow('window', rotated)

img = cv.imread('fish.jpg')
h, w = img.shape[:2]
center = w//2, h//2

cv.imshow('window', img)
cv.createTrackbar('angle', 'window', 0, 180, trackbar)

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

`transform2.py`

4.3 Scale

Scaling an image is to change its dimension.

```python
"""scale an image using the trackbar."""
import cv2 as cv

def trackbar(scale):
    M = cv.getRotationMatrix2D(center, 0, scale/10)
    rotated = cv.warpAffine(img, M, (w, h))
    cv.imshow('window', rotated)

img = cv.imread('fish.jpg')
h, w = img.shape[:2]
center = w//2, h//2

cv.imshow('window', img)
cv.createTrackbar('scale', 'window', 10, 30, trackbar)

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

transform3.py

4.4 Flipping

Horizontally or vertically using a key.

```python
"""Flip an image horizontally and vertically using keys."""
import cv2 as cv

img = cv.imread('fish.jpg')
cv.imshow('window', img)

while True:
    k = cv.waitKey(0)
    if k == ord('q'):
        break

    elif k == ord('v'):
        img = cv.flip(img, 0)

    elif k == ord('h'):
        img = cv.flip(img, 1)

    cv.imshow('window', img)

cv.destroyAllWindows()
```

transform4.py

4.5 Image artithmetic

The operation `add` and `subtract` allow to add two images. The `add` function is limited to 255. The `subtract` function is limit to 0. In the example below we add or subtract the value (40, 40, 40) to each pixel. As a result, the image becomes brighter or darker.

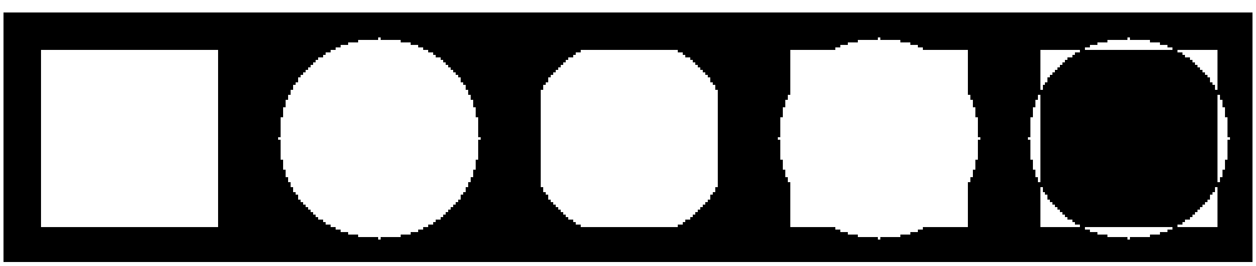
```
"""Add and subtract"""
import cv2 as cv
import numpy as np

img = cv.imread('fish.jpg')
img = cv.resize(img, None, fx=0.5, fy=0.5, interpolation=cv.INTER_CUBIC)
M = np.ones(img.shape, dtype='uint8') * 40

brighter = cv.add(img, M)
darker = cv.subtract(img, M)

img2 = np.hstack([img, brighter, darker])

cv.imshow('window', img2)
cv.waitKey(0)
cv.destroyAllWindows()
```

transform5.py

4.6 Bitwise operations

Bitwise operations act on grayscale images. Most often it is used on black and white images. We start wtih a circle and a square shape and calculate this three bitwise operations:

- and
- or
- xor (exclusive or)



```
"""Bitwise and, or and xor operation"""
import cv2 as cv
import numpy as np

d = 15
```

(continues on next page)

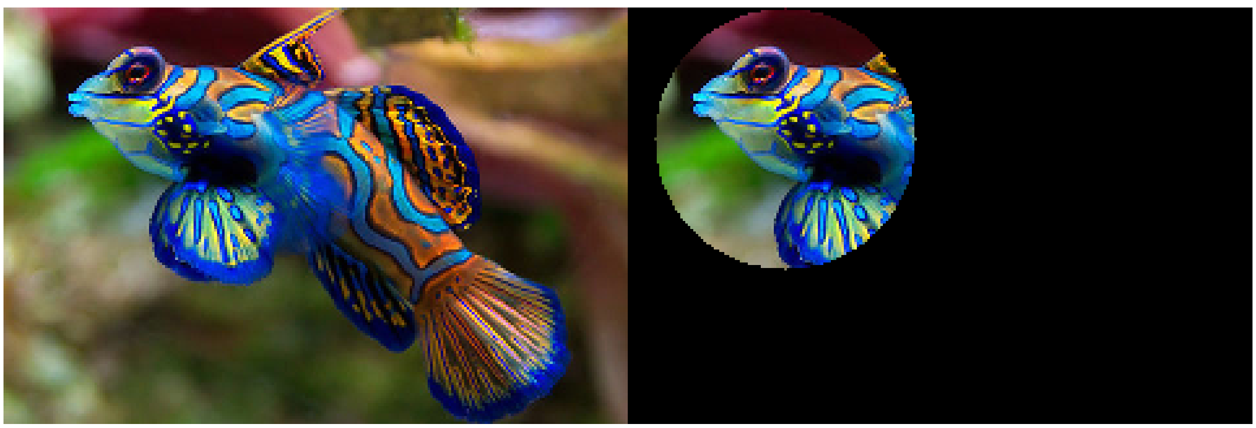
(continued from previous page)

```
rect = np.zeros((100, 100), np.uint8)
cv.rectangle(rect, (d, d), (100-d, 100-d), 255, -1)

circle = np.zeros((100, 100), np.uint8)
cv.circle(circle, (50, 50), 40, 255, -1)

bit_and = cv.bitwise_and(rect, circle)
bit_or = cv.bitwise_or(rect, circle)
bit_xor = cv.bitwise_xor(rect, circle)

img = np.hstack([rect, circle, bit_and, bit_or, bit_xor])

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

bitwise.py

4.7 Masking

We can use a mask to extract only a certain part of an image.

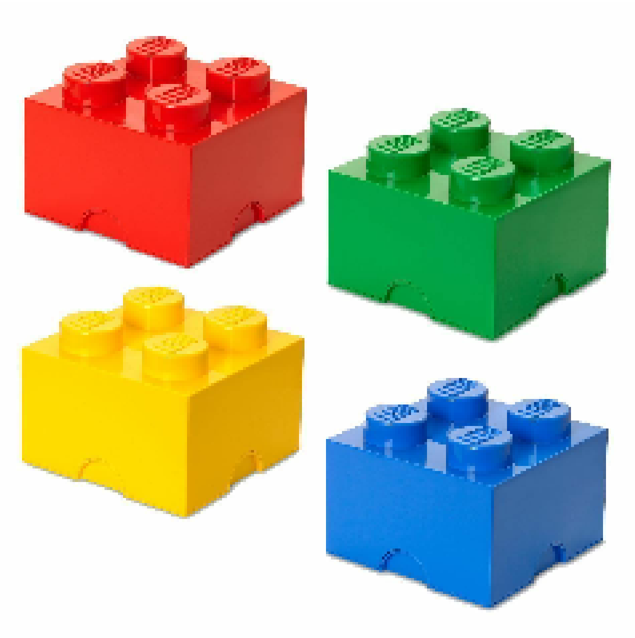


```
"""Masking."""
import cv2 as cv
import numpy as np

img = cv.imread('fish.jpg')
img = cv.resize(img, None, fx=0.5, fy=0.5, interpolation=cv.INTER_CUBIC)
mask = np.zeros(img.shape[:2], dtype='uint8')
cv.circle(mask, (60, 50), 50, 255, -1)

masked = cv.bitwise_and(img, img, mask=mask)
img2 = np.hstack([img, masked])

cv.imshow('window', img2)
cv.waitKey(0)
cv.destroyAllWindows()
```

masking1.py

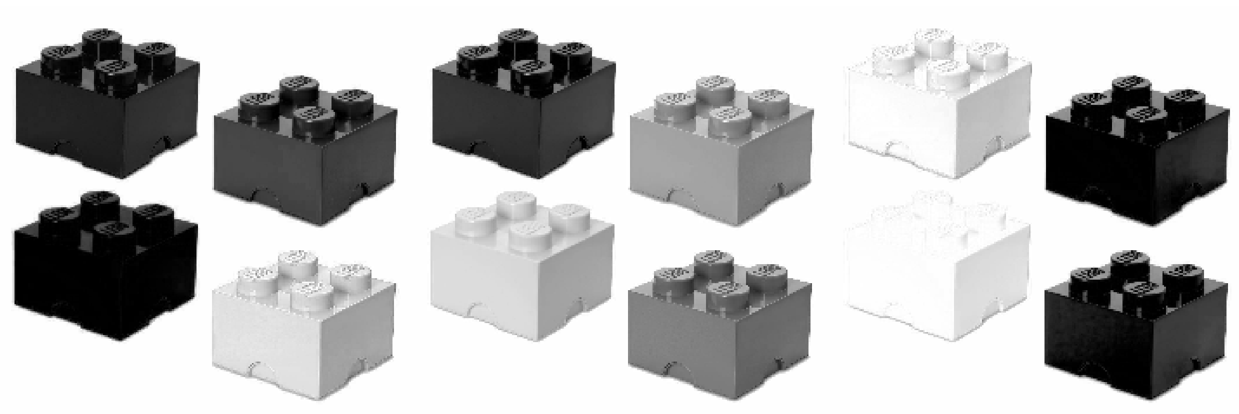
4.8 Splitting channels

We can split an RGB image into it's components. Let's use an image which contains the three base colors.



lego.png

We find each color component in the separate channel Blue-Green-Red.

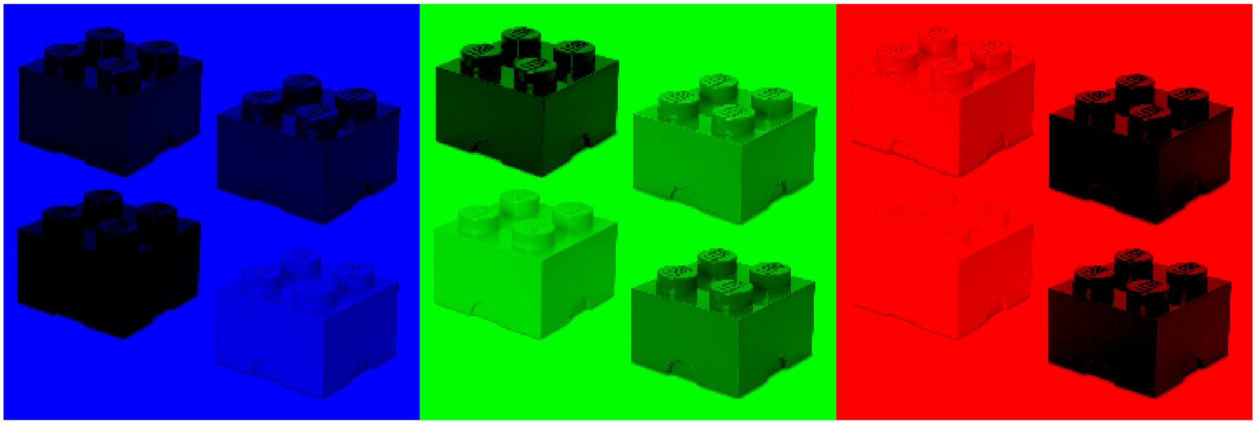


```
"""Splitting into 3 channels"""
import cv2 as cv
import numpy as np

img = cv.imread('lego.png')
b, g, r = cv.split(img)
img2 = np.hstack([b, g, r])

cv.imshow('window', img2)
cv.waitKey(0)
cv.destroyAllWindows()
```

splitting1.py

4.9 Merging channels

We can merge channels.



```
"""Merging 3 channels"""
import cv2 as cv
import numpy as np

img = cv.imread('lego.png')
z = np.zeros(img.shape[:2], 'uint8')

b, g, r = cv.split(img)
blue = cv.merge([b, z, z])
green = cv.merge([z, g, z])
red = cv.merge([z, z, r])

img2 = np.hstack([blue, green, red])

cv.imshow('window', img2)
cv.waitKey(0)
cv.destroyAllWindows()
```

splitting2.py

A different and faster way of keeping only one color channel and setting the others to zero is to act directly on the Numpy array using slice indexing.

```
"""Numpy indexing."""
import cv2 as cv
import numpy as np

img = cv.imread('lego.png')
blue = img.copy()
green = img.copy()
red = img.copy()

blue[:, :, 1:] = 0
green[:, :, 0] = 0
green[:, :, 2] = 0
red[:, :, :2] = 0

img2 = np.hstack([blue, green, red])
```

(continues on next page)

(continued from previous page)

```
cv.imshow('window', img2)
cv.waitKey(0)
cv.destroyAllWindows()
```

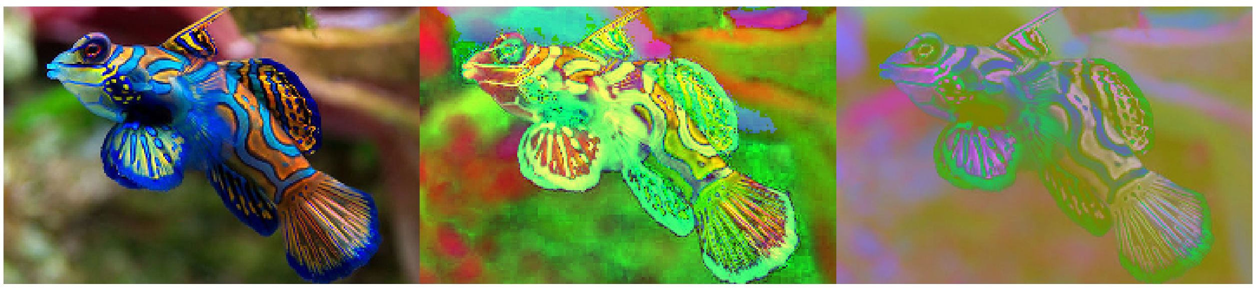
splitting3.py

4.10 Color spaces

So far we have seen the RGB color space. However there are many other spaces.

The example below shows:

- HSV (Hue-Saturation-Value)
- L*a*b

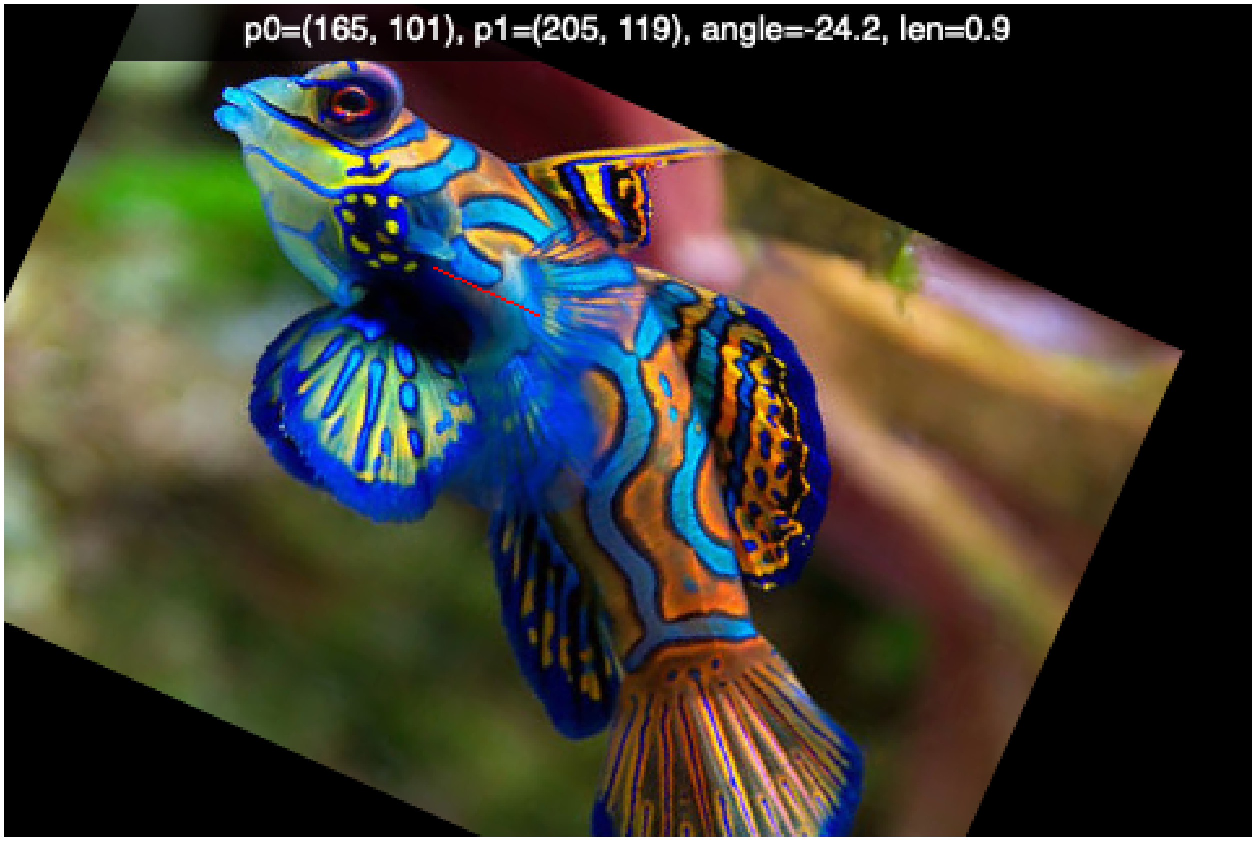


```
"""Change the color space."""
import cv2 as cv
import numpy as np

img = cv.imread('fish.jpg')
img = cv.resize(img, None, fx=0.5, fy=0.5, interpolation=cv.INTER_CUBIC)
M = np.ones(img.shape, dtype='uint8') * 40

hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
lab = cv.cvtColor(img, cv.COLOR_BGR2LAB)
img2 = np.hstack([img, hsv, lab])

cv.imshow('window', img2)
cv.waitKey(0)
cv.destroyAllWindows()
```

transform7.py

4.11 Affine transformation

Here we use the mouse to rotate and scale.




```python
"""Rotate and scale image with mouse."""
import cv2 as cv
import numpy as np

RED = (0, 0, 255)
p0, p1 = (100, 30), (400, 90)

def mouse(event, x, y, flags, param):
    global p0, p1

    if event == cv.EVENT_LBUTTONDOWN:
        p0 = x, y
        p1 = x, y

    elif event == cv.EVENT_MOUSEMOVE and flags == 1:
        p1 = x, y

    elif event == cv.EVENT_LBUTTONUP:
        p1 = x, y


    dx = p1[0] - p0[0]
    dy = p1[1] - p0[1]
    angle = -np.degrees(np.arctan2(dy, dx))
    len = np.sqrt(dx**2 + dy**2) / 50
    cv.displayOverlay('window', f'p0={p0}, p1={p1}, angle={angle:.1f}, len={len:.1f}')

    M = cv.getRotationMatrix2D(p0, angle, len)
```

(continues on next page)

(continued from previous page)

```
    img2 = cv.warpAffine(img, M, (w, h))
    cv.line(img2, p0, p1, RED, 2)
    cv.imshow('window', img2)

img = cv.imread('fish.jpg')
h, w = img.shape[:2]
cv.imshow('window', img)
cv.setMouseCallback('window', mouse)

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

affine1.py

CHAPTER 5

Histograms

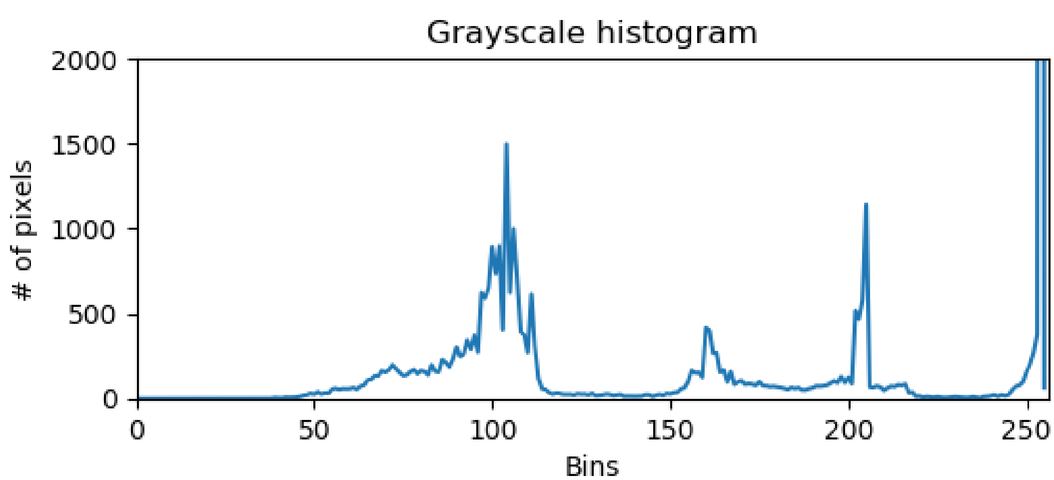
Histograms are

5.1 Grayscale histogram

The `calcHist` function takes these arguments:

```
cv.calcHist([img], channels, mask, bins, ranges)
```

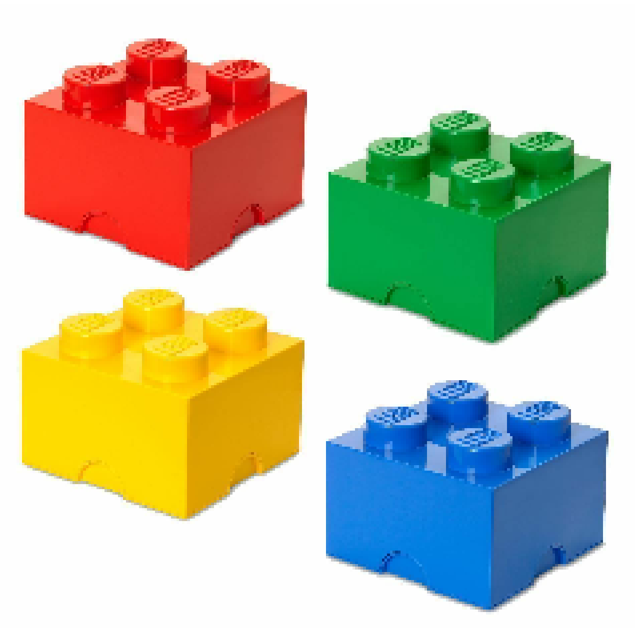
- image list
- channel list
- mask
- the number of **bins**
- ranges, typically [0, 255]



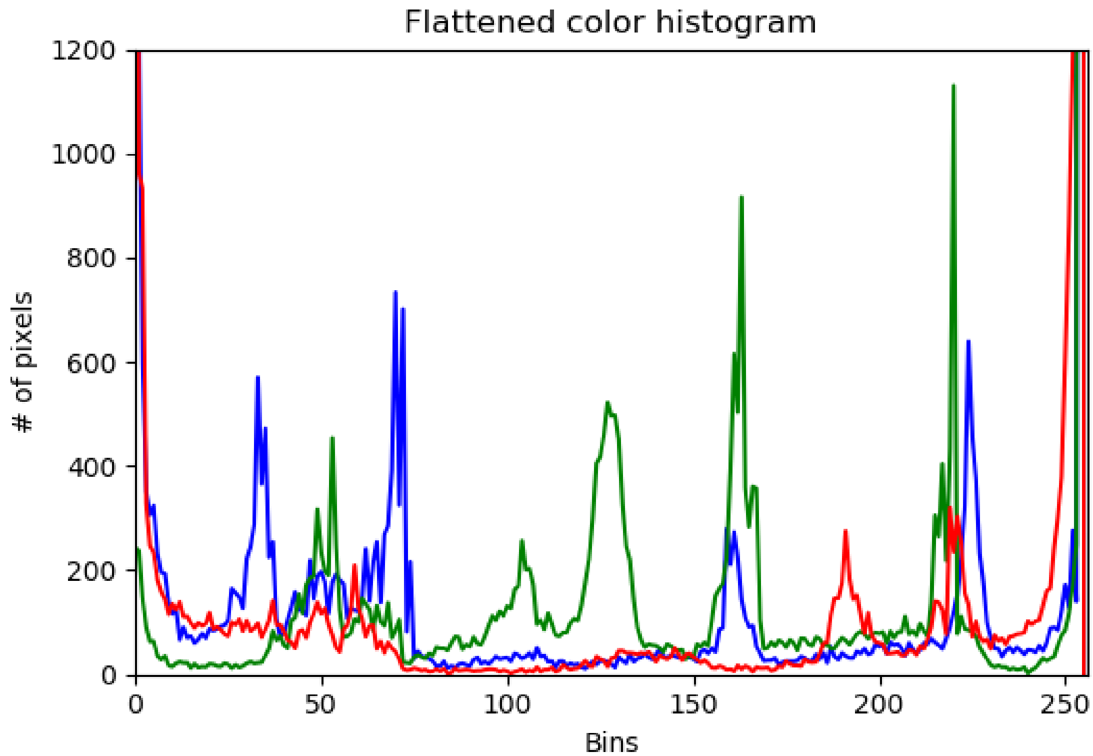
```
from matplotlib import pyplot as plt
import cv2 as cv

img = cv.imread('lego.png')
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

hist = cv.calcHist([gray], [0], None, [256], [0, 256])

plt.figure()
plt.title('Grayscale histogram')
plt.xlabel('Bins')
plt.ylabel('# of pixels')
plt.plot(hist)
plt.xlim([0, 256])
plt.ylim([0, 2000])
plt.show()

cv.waitKey(0)
```

histogram1.py

5.2 Color histogram



Here is the histogram




```
# Color histogram
from matplotlib import pyplot as plt
import cv2 as cv

img = cv.imread('lego.png')
chans = cv.split(img)
colors = 'b', 'g', 'r'

plt.figure()
plt.title('Flattened color histogram')
plt.xlabel('Bins')
plt.ylabel('# of pixels')

for (chan, color) in zip(chans, colors):
    hist = cv.calcHist([chan], [0], None, [256], [0, 255])
    plt.plot(hist, color=color)
    plt.xlim([0, 256])
    plt.ylim([0, 1200])

plt.show()
cv.waitKey(0)
```

histogram2.py

5.3 Blurring


```
# Blurring
import cv2 as cv

def trackbar(x):
    x = cv.getTrackbarPos('blur x','window')
    y = cv.getTrackbarPos('blur x','window')
    blurred = cv.blur(img, (x, y))
    cv.imshow('window', blurred)
    cv.displayOverlay('window', f'blur = ({x}, {y})')

img = cv.imread('lego.png')
cv.imshow('window', img)
cv.createTrackbar('blur x', 'window', 0, 4, trackbar)
cv.createTrackbar('blur y', 'window', 0, 4, trackbar)

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

blur1.py

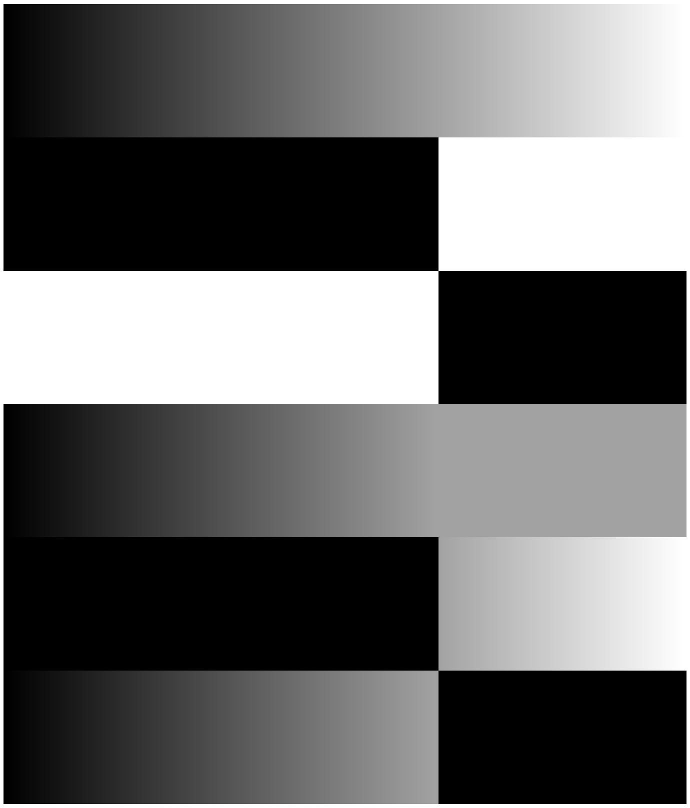
CHAPTER 6

Filters and convolution

https://docs.opencv.org/master/d2/d96/tutorial_py_table_of_contents_imgproc.html

6.1 Simple thresholding

For every pixel, the same threshold is applaied. If the pixel is smaller then the threshold, it is set to 0, otherwise it is set to the maximum.



```
# Add a trackbar
import cv2 as cv
import numpy as np

img = np.fromfunction(lambda i, j: j, (50, 256), dtype='uint8')

modes = (cv.THRESH_BINARY,
        cv.THRESH_BINARY_INV,
        cv.THRESH_TRUNC,
        cv.THRESH_TOZERO,
        cv.THRESH_TOZERO_INV)

def trackbar(x):
    """Trackbar callback function."""
    text = f'threshold={x}'
    cv.displayOverlay('window', text, 1000)

    ret, img1 = cv.threshold(img, x, 255, cv.THRESH_BINARY)
    ret, img2 = cv.threshold(img, x, 255, cv.THRESH_BINARY_INV)
    ret, img3 = cv.threshold(img, x, 255, cv.THRESH_TRUNC)
    ret, img4 = cv.threshold(img, x, 255, cv.THRESH_TOZERO)
    ret, img5 = cv.threshold(img, x, 255, cv.THRESH_TOZERO_INV)

    cv.imshow('window', np.vstack([img, img1, img2, img3, img4, img5]))

cv.imshow('window', img)
trackbar(100)
cv.createTrackbar('threshold', 'window', 100, 255, trackbar)

cv.waitKey(0)
```

(continues on next page)

(continued from previous page)

```
cv.destroyAllWindows()
```

threshold0.py

6.2 Binary thresholding

```
# binary thresholding
import cv2 as cv
import numpy as np

def trackbar(x):
    ret, img1 = cv.threshold(img, x, 255, cv.THRESH_BINARY)
    ret, img2 = cv.threshold(img, x, 255, cv.THRESH_BINARY_INV)
    cv.imshow('window', np.hstack([img, img1, img2]))

    text = f'threshold={x}, mode=BINARY, BINARY_INV'
    cv.displayOverlay('window', text, 1000)

img = cv.imread('eye.jpg')
trackbar(100)
cv.createTrackbar('threshold', 'window', 100, 255, trackbar)

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

threshold1.py

6.3 To zero

```
# threshold to zero
import cv2 as cv
import numpy as np
```

(continues on next page)

(continued from previous page)

```
def trackbar(x):
    """Trackbar callback function."""
    text = f'threshold={x}, mode=TOZERO, TOZERO_INV'
    cv.displayOverlay('window', text, 1000)

    ret, img1 = cv.threshold(img, x, 255, cv.THRESH_TOZERO)
    ret, img2 = cv.threshold(img, x, 255, cv.THRESH_TOZERO_INV)
    cv.imshow('window', np.hstack([img, img1, img2]))

img = cv.imread('eye.jpg')
cv.imshow('window', img)
cv.createTrackbar('threshold', 'window', 100, 255, trackbar)

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

threshold2.py

6.4 Adaptive thresholding

https://docs.opencv.org/master/d7/d1b/group__imgproc__misc.html#ga72b913f352e4a1b1b397736707afcde3

6.5 2D convolution

https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html


```
# convolution
import cv2 as cv
import numpy as np

kernel = np.ones((5, 5), 'float32')/25

def trackbar(x):
    """Trackbar callback function."""
    d = 2*x + 1
    kernel = np.ones((d, d), 'float32')/(d**2)
```

(continues on next page)

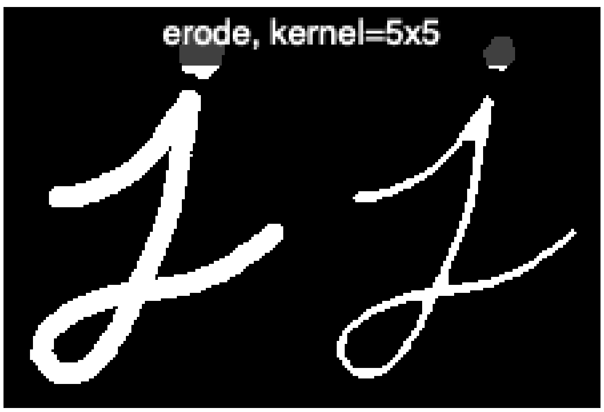
(continued from previous page)

```
    img1 = cv.filter2D(img, -1, kernel)
    cv.imshow('window', np.hstack([img, img1]))

    text = f'kernel=({d}x{d})'
    cv.displayOverlay('window', text)

img = cv.imread('eye.jpg')
trackbar(2)
cv.createTrackbar('threshold', 'window', 2, 7, trackbar)

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

convolution1.py

6.6 Morphological Transformations

6.6.1 Erosion




```
# morphological transformation : erode
import cv2 as cv
import numpy as np

def trackbar(x):
    n = 2*x + 1
    kernel = np.ones((n, n), np.uint8)

    img1 = cv.erode(img, kernel, iterations=1)
    cv.imshow('window', np.hstack([img, img1]))

    text = f'erode, kernel={n}x{n}'
    cv.displayOverlay('window', text)

img = cv.imread('j.png')
trackbar(2)
cv.createTrackbar('kernel', 'window', 2, 5, trackbar)

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

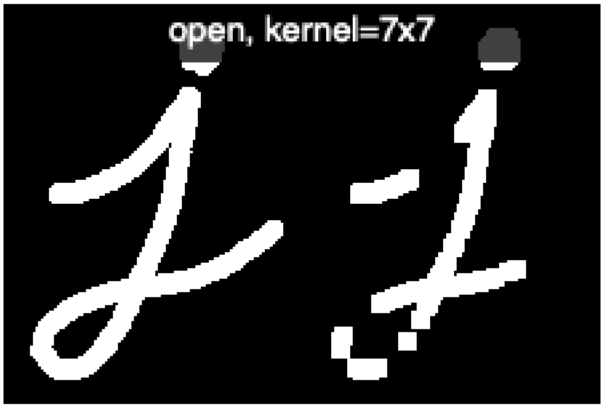
morph1.py

6.6.2 Dilation


```
# morphological transformation : dilation
import cv2 as cv
import numpy as np

def trackbar(x):
    n = 2*x + 1
    kernel = np.ones((n, n), np.uint8)

    img1 = cv.dilate(img, kernel, iterations=1)
    cv.imshow('window', np.hstack([img, img1]))

    text = f'dilate, kernel={n}x{n}'
    cv.displayOverlay('window', text)

img = cv.imread('j.png')
trackbar(2)
cv.createTrackbar('kernel', 'window', 2, 5, trackbar)

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

morph2.py

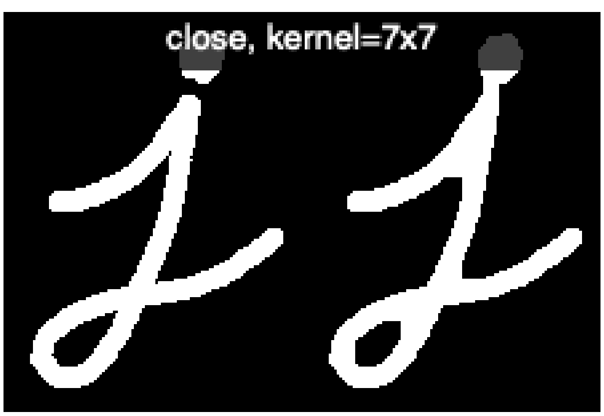
6.6.3 Opening




```
# morphological transformation : opening
import cv2 as cv
import numpy as np

def trackbar(x):
    n = 2*x + 1
    kernel = np.ones((n, n), np.uint8)

    img1 = cv.morphologyEx(img, cv.MORPH_OPEN, kernel)
    cv.imshow('window', np.hstack([img, img1]))

    text = f'open, kernel={n}x{n}'
    cv.displayOverlay('window', text)

img = cv.imread('j.png')
trackbar(2)
cv.createTrackbar('kernel', 'window', 2, 5, trackbar)

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

morph3.py

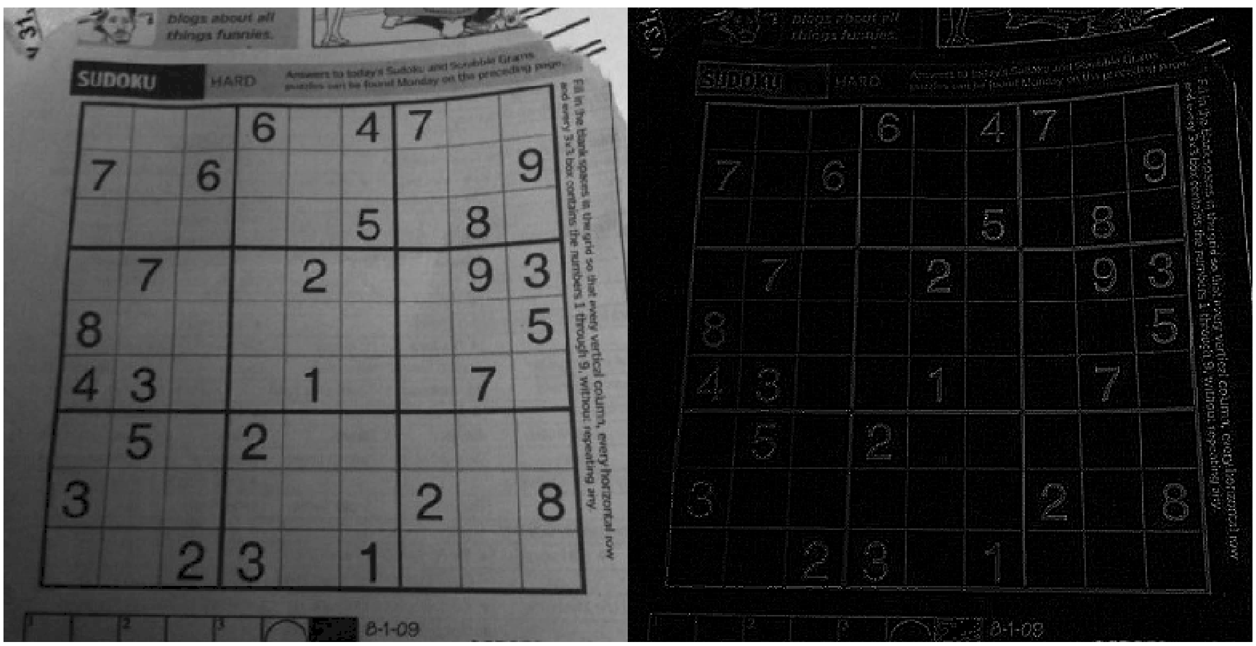
6.6.4 Closing



```
# morphological transformation : closing
import cv2 as cv
import numpy as np

def trackbar(x):
    n = 2*x + 1
    kernel = np.ones((n, n), np.uint8)

    img1 = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel)
    cv.imshow('window', np.hstack([img, img1]))

    text = f'close, kernel={n}x{n}'
    cv.displayOverlay('window', text)

img = cv.imread('j.png')
trackbar(2)
cv.createTrackbar('kernel', 'window', 2, 5, trackbar)

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

morph4.py

6.7 Image gradient - Laplacian



```
# image gradient - Laplacian
import cv2 as cv
import numpy as np

img = cv.imread('sudoku.png', cv.IMREAD_GRAYSCALE)
img1 = cv.Laplacian(img.copy(), 8)
cv.imshow('window', np.hstack([img, img1]))
```

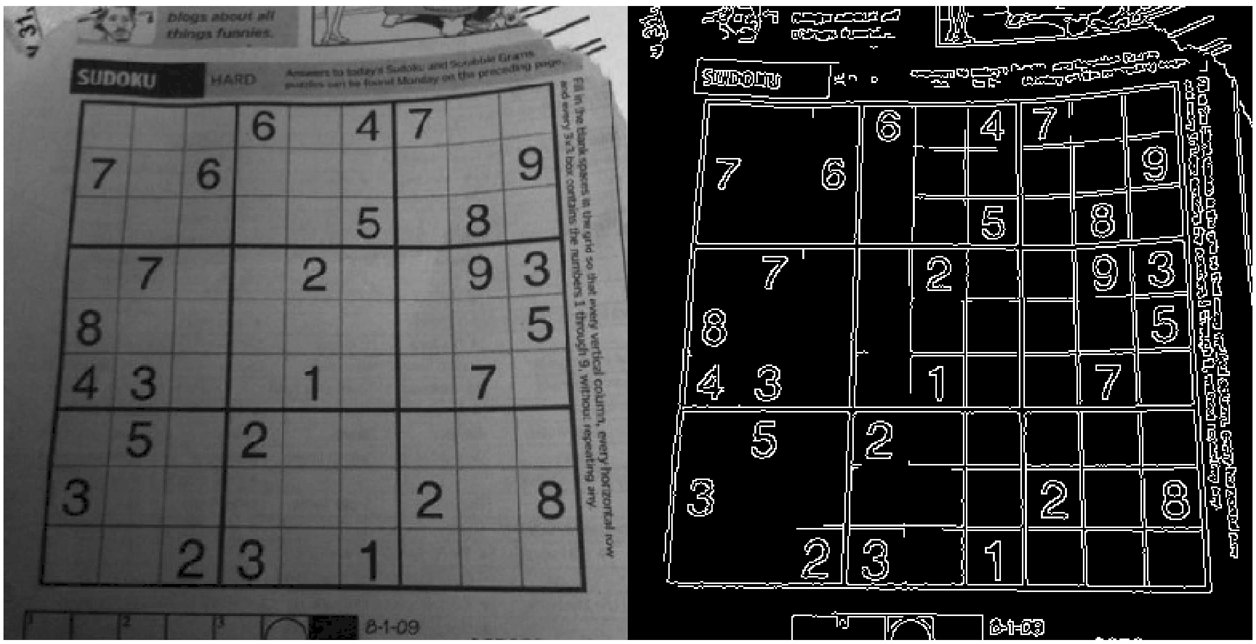
(continues on next page)

(continued from previous page)

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

gradient.py

6.8 Canny edge detection




```
# image gradient - Laplacian
import cv2 as cv
import numpy as np

img = cv.imread('sudoku.png', cv.IMREAD_GRAYSCALE)
img1 = cv.Canny(img, 100, 200)
cv.imshow('window', np.hstack([img, img1]))

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

canny.py

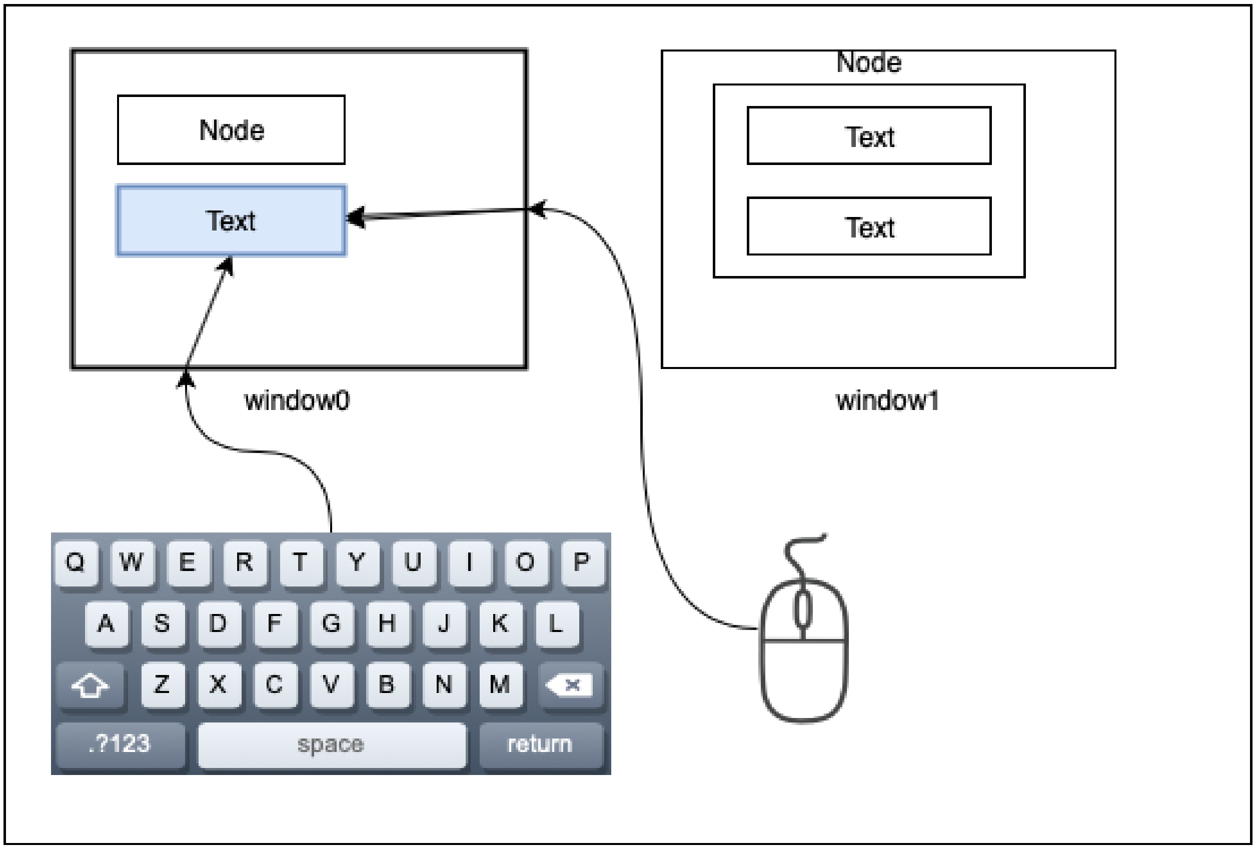
CHAPTER 7

Creating an application

In this section we are going to create an application from zero. Our goal to establish a general framework which can be the basis for different kinds of applications, such as editors, browsers or video games.

The image below shows an schematic of how an app works. The app is the overall place where keyboard and mouse input is detected. An app can open one or several windows. Only one is the active window at any given time. To make a window the active window, one has to click with the mouse in it. Each window has several objects. These objects are organized as hierarchical nodes. Each window can have one active node. The keyboard events are sent from the application to the active window, and from the active window to the active node (node, widget, or shape).

A Text node can use keyboard input to edit the widget. If there is no active node, then the window can use the keyboard input as shortcuts. If neither an active node, nor the window handles the keyboard event, it falls back to the application, which can handle it.



App

We start by importing the **OpenCV** and the **numpy** module and give them the usual abbreviations:

```
import cv2 as cv
import numpy as np
```

Then we declare the `App` class which creates a named window with `namedWindow`. Without any window the `waitKey` function does not work:

```
class App:
    def __init__(self):
        cv.namedWindow('window0')
```

Now we need a method to run the application:

```
def run(self):
    key = ''
    while key != 'q':
        k = cv.waitKey(0)
        key = chr(k)
        print(k, key)

    cv.destroyAllWindows()
```

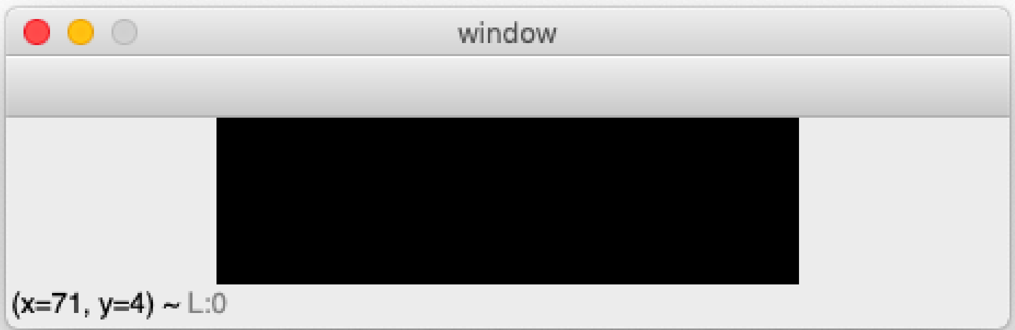
We wait for a key from the keyboard. The argument of the `waitKey` function means the timeout period in miliseconds. A value of 0 means to wait without a time limit. If we wanted to display the frames of a video stream every 25 miliseconds, we could write `cv.waitKey(25)`. If no key is pressed during this period, a -1 integer value is returned.

Typing *q* quits the event loop and closes all windows.

At the end of the program we add code to instantiate the App and to call the `run()` method:

```
if __name__ == '__main__':
    App().run()
```

With `cv.namedWindow('window0')` OpenCV opens a smll black image such as shown below.



On a Mac the some of the key presses do not give a result and some of the keys have a code value of 0. These keys have:

- no key code: cmd, fn, Up, Down, Left, Right
- key code 0: alt, ctrl, shift

All letters are lower-case only. We will see later how we can use the code 0 of the alt/ctrl/shift key to toggle between lower case and upper case letters.

7.1 Shortcut keys

It is convienent for an application to have shortcut keys. The most efficient way to define to associate certain keys with a function is to use a dictionary. In the App class **init** function we add:

```
self.shortcuts = {  'h': help,
                    'i': self.inspect,}
```

This dictionary associates the letter **h** with the function `help()` and the letter **i** with the function `self.inspect()`. Later we will add more shortcut functions.

In the App class we define the `key` handler:

def key(self, k):

if k in self.shortcuts: self.shortcuts[k]()

The function **help** is defined as a global function:

```
def help():
    print('--- HELP ---')
```

The function **inspect** is defined as a method of the App class:

```
def inspect(self):
    print('--- INSPECT ---')
    print('App.wins', App.wins)
    print('App.win', App.win)
```

This kind of inspect function is usefull for debugging.

7.2 Create the Window class

Some applications have only one window, but often an appliation can have any number of windows. To track all the windows of an application and specify the currently active window, we add these two class variables to the `App`` class:

```
class App:
    wins = []
    win = None
```

App.wins is the list of opended windows. **App.win** is the currently active window.

The `Window` class is defined below:

```
class Window:
    """Create a window."""
    def __init__(self, win=None, img=None):
```

First, the new window is added to the Apps window list. Then it is made the currently active window:

```
App.wins.append(self)
App.win = self
```

Then the windows object list `self.objs` is set to the empty list. Currently there is now active object, so currently active object `self.obj` it's set to `None`:

```
self.objs = []
self.obj = None
```

If no image is given, the constructer creates a 200 x 600 pixel default image with all pixels being black:

```
if img==None:
    img = np.zeros((200, 600, 3), np.uint8)
```

If no window name is given, a new string is formed from the window id. Afterwards the id is incremented to the next higher value:

```
if win == None:
    win = 'window' + str(App.win_id)
App.win_id += 1
```

The window name and the image are stored as an instance attribute:

```
self.win = win
self.img = img>
```

As the window is directly modified by adding graphics objects to it, we need to keep a copy of the original image:

```
self.img0 = img.copy()
```

Finally we show the image:

```
cv.imshow(win, img)
```

7.3 Handle the mouse

The mouse is handled separately by each window. We set a mouse callback function to the window's `mouse` handler function:

```
cv.setMouseCallback(win, self.mouse)
```

Inside the `Window` class we define a `mouse` function which receives the parameters:

- eventy type (mouse down, up, double-click, move)
- postion (x, y)
- flags (3 mouse bottons, 3 modifier keys)

    def mouse(self, event, x, y, flags, param): text = 'mouse event {} at ({}, {}) with flags {}'.format(event, x, y, flags) cv.displayStatusBar(self.win, text, 1000)

We display these parameters for 1 second in the status bar.

Inside the `mouse` callback function, we dispatch the events, according to the event type. There are 12 different types of mouse events:

```
EVENT_LBUTTONDBLCLK 7
EVENT_LBUTTONDOWN 1
EVENT_LBUTTONUP 4
EVENT_MBUTTONDBLCLK 9
EVENT_MBUTTONDOWN 3
EVENT_MBUTTONUP 6
EVENT_MOUSEHWHEEL 11
EVENT_MOUSEMOVE 0
EVENT_MOUSEWHEEL 10
EVENT_RBUTTONDBLCLK 8
EVENT_RBUTTONDOWN 2
EVENT_RBUTTONUP 5
```

There are 3 buttons:

- left (LBUTTON)
- middle (MBUTTON)
- right (RBUTTON)

and there are 3 event types:

- down (DOWN)
- up (UP)
- doubleclick (DBLCLK)

Furthermore there are 6 event flags which can be combined together. For exemple, pressing the left button and the ctrl key simultaneously would result in 9, the sum of 1+8:

```
EVENT_FLAG_LBUTTON 1
EVENT_FLAG_MBUTTON 4
EVENT_FLAG_RBUTTON 2

EVENT_FLAG_CTRLKEY 8
EVENT_FLAG_SHIFTKEY 16
EVENT_FLAG_ALTKEY 32
```

When a mouse is clicked in a window, this window becomes the active window and this must be signalled to the App:

```
if event == cv.EVENT_LBUTTONDOWN:
    App.win = self
```

7.4 Create the Object class

An app can have multiple windows, and each window can have multiple objects. Only one object is the active object in any one window. We add this code to the constructor of the Window class:

```
self.objs = []
self.obj = None
```

Initially the object list is empty, and there is no active object yet.

Now we can create the Object class:

```
class Object:
    """Add an object to the current window."""
    def __init__(self, **options):
        App.win.objs.append(self)
        App.win.obj = self
        self.img = App.win.img
```

We append the new object to the object list of the currently active window. We go through two levels: the app knows the currently active window, and the currently active window keeps track of its objects.

The expression `App.win.obj` means the currently active object of the currently active window. There is always an active window, which is also the top window. The window which had been clicked last, becomes the active window.

Finally we set the windows image as the target for the object.

To specify the default options for a new object we use a dictionary:

- default position (pos)
- default size (size)
- inital id

This default dictionary defined as a Window **class attribute**, and is the same for all windows:

```
obj_options = dict(pos=(20, 20), size=(100, 30), id=0)
```

The current object obtions are defined as Window **instance attribute** and is independent for each window. We must be careful to copy the dictionary, and not just make a reference to it:

```
self.obj_options = Window.obj_options.copy()
```

Inside the Object constructor we update the object options with the new options received as argument:

```
d = App.win.obj_options
d.update(options)
```

Then we assign the id, position and size of the object:

```
self.id = d['id']
self.pos = x, y = d['pos']
self.size = w, h = d['size']
```

Then we increment the object id:

```
d['id'] += 1
```

Often objects (buttons, text) are placed in a vertical layout, with a small gap, we calculate a new position for the next object automatically:

```
d['pos'] = x, y + h + 5
```

In order name the object, we give define the **str** method:

```
def __str__(self):
    return 'Object {} at ({}, {})'.format(self.nbr, *self.pos)
```

7.5 Drawing an object

Each object knows how to draw itself. At this point we need to define some colors at the beginning of the program. Remember that OpenCV uses the BGR color format:

```
BLACK = (0, 0, 0)
RED = (0, 0, 255)
GREEN = (0, 255, 0)
BLUE = (255, 0, 0)
WHITE = (255, 255, 255)
```

In the **Object** class we add a **draw** method which draws the object by placing a thin rectangle on the image to mark the region occupied by the object:

```
def draw(self):
    cv.rectangle(self.img, (*self.pos, *self.size), RED, 1)
```

In the **Window** class add a **draw** method which draws all the objects. First we restore the image from the stored original image. Then we draw all the objects and finally we show the updated image:

```
def draw(self):
    self.img[:] = self.img0[:]

    for obj in self.objs:
        obj.draw()

    cv.imshow(self.win, self.img)
```

At this point, we can redraw the window, whenever there is a mouse event. So we add this as the last line in the `mouse` handler:

```
self.draw()
```

7.6 Adding new windows and new objects

The constructors of the Window and the Object class both have default parameters. This allows us to add shortcuts to automatically create new windows and new objects:

```
self.shortcuts = {  'h': help,
                    'i': self.inspect,
                    'w': Window,
                    'o': Object,}
```

7.7 Passing the mouse click to an object

When a mouse click happens inside an object, this should be handled by that object. Therefore we need to know if the mouseclick happend inside the object:

```
def is_inside(self, x, y):
    x0, y0 = self.pos
    w, h = self.pos
    return x0 <= x <= x0+w and y0 <= y <= y0+h
```

Inside the Window **mouse** method we add this code:

```
if event == cv.EVENT_LBUTTONDOWN:
    App.win = self

    self.obj = None
    for obj in self.objs:
        obj.selected = False
        if obj.is_inside(x, y):
            obj.selected = True
            self.obj = obj
```

7.8 Select an object

In order to act on an object we need to select it. This can be done by clicking with the mouse on the object. At the creation of a new object it is not selected:

```
self.selected = False
```

We draw the selected object with a colored contour. This is the modiefied `draw` method:

```
def draw(self):
    x, y = self.pos
    w, h = self.size
    cv.rectangle(self.img, (x, y, w, h), WHITE, 1)
    if self.selected:
        cv.rectangle(self.img, (x-2, y-2, w+2, h+2), RED, 2)
```

7.9 Moving an object

If the mouse is clicked over an object, the name of the object is printed:

```
def mouse(self, event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        print(self)
```

The moving of an object has to be defined in the Window mouse handler and not in the Object mouse handler. Depending on the direction we move the object, the mouse coordinates can be outside the object.

If the mouse moves and the ALT key is pressed, the current object is moved to the cursor position (x, y):

```
if event == cv.EVENT_MOUSEMOVE:
    if flags == cv.EVENT_FLAG_ALTKEY:
        self.obj.pos = x, y
```

7.10 Add window custom options

To make our application as customizable as possible, we should give all parameters such as the window background color, the default object color, the selection color as options to the app class.

In the App class we add this line:

```
options = dict( win_color=GRAY, obj_color=YELLOW, sel_color=BLUE)
```

In the Window **init** method we add this:

```
if img == None:
    img = np.zeros((200, 600, 3), np.uint8)
    img[:,:] = App.options['win_color']
```

We update the Object **draw** method to this:

```
cv.rectangle(self.img, (x, y, w, h), App.options['obj_color'], 1)
if self.selected:
    cv.rectangle(self.img, (x-2, y-2, w+2, h+2), App.options['sel_color'], 2)
```

7.11 Displaying information in the status bar

The status bar is a convenient place to display feedback information during program development.

Compared to printing to the console the statusbar has a double advantage:

- the info appears in the associated window
- the info disappears after a timeout

This code is added to the `key` handler in the Window class:

```
text = 'key {} ({})'.format(k, ord(k))
cv.displayStatusBar(self.win, text, 1000)
```

This code is added to the `mouse` handler in the Window class:

```
def mouse(self, event, x, y, flags, param):
     text = 'mouse event {} at ({}, {}) with flags {}'.format(event, x, y, flags)
     cv.displayStatusBar(self.win, text, 1000)
```

7.12 Create the Text class

In order to add text to an object, we subclass the Object class and we add the text options as a class attribute:

```
class Text(Object):
    """Add a text object to the current window."""
    options = dict( fontFace=cv.FONT_HERSHEY_SIMPLEX,
                    fontScale=1,
                    color=BLUE,
                    thickness=1,
                    lineType=cv.LINE_8, )
```

In the constructor method we update the options, copy them to the Text object, then we call the parent (Object class) constructor:

```
def __init__(self, text='Text', **options):

    for k, v in options.items():
        if k in Text.options:
            Text.options[k] = v
```

7.13 Send key events to windows and objects

In order to send key events to a specfic object, we must first send the key event from the app level to the currently active window by modifying the App event loop like this:

```
def run(self):
    while True:
        key = cv.waitKey(0)

        if key >= 0:
            k = chr(key)
            if not App.win.key(k):
                self.key(k)
```

We first the key event to the Window level by calling `App.win.key` handler. If the upper level handles the event, it is returning True. In that case the App level has does not need to call its own `key` handler.

On the app level the letters **w, o, t, i, h** have associated shortcuts. However, when an object is active for editing, the key press has to go to the active object, and should not be treated as a shortcut.

In the Window class we add a `key` event handler which treats certain keys as special:

- the TAB key to advance to the next object
- the ESCAPE key to unselect the current object
- the CMD/SHFT key to toggle upper and lower case

Again we use a dictionary to associate the keys with their respective actions:

```
self.shortcuts = {  '\t': self.select_next_obj,
                    chr(27): self.unselect_obj,
                    chr(0): self.toggle_case, }
```

At the window level we first see if the key is part of the shortcut keys. If this is the case, the associated function is called, the image redrawn, and the `key` handler returns True, to signal to the caller that the event has been dealed with:

```
def key(self, k):
    if k in self.shortcuts:
        self.shortcuts[k]()
        self.draw()
        return True

    elif self.obj != None:
        self.obj.key(k)
        self.draw()
        return True

    return False
```

If the key is not a shortcut key and if there exists an active object, the key is sent the **key(k)** handler at the Object level. There the key events are used for editing the text attribute.

7.14 Use the tab key to advance to the next object

It is convenient to use the tab key to move between objects. The following function tries to `find` the index of the currently selected object, if there is one, and increments it by one:

```
def select_next_obj(self):
    """Select the next object, or the first in none is selected."""
    try:
        i = self.objs.index(self.obj)
    except ValueError:
        i = -1
    self.objs[i].selected = False
    i = (i+1) % len(self.objs)
    self.objs[i].selected = True
    self.obj = self.objs[i]
```

7.15 Use the escape key to unselect

The escape key can serve to unselect an object. We add the following code to the Window class:

```
def unselect_obj(self):
    if self.obj != None:
        self.obj.selected = False
        self.obj = None
```

7.16 Toggle between upper case and lower case

The OpenCV module does not allow to get upper-case letters. To be able to input upper case letters we use the keys which result in a key code of 0 to switch between upper case and lower case. To implement this we add the following code to the Window `key` handler:

```
elif k == chr(0):  # alt, ctrl, shift
    self.upper = not self.upper
    if self.upper:
        cv.displayStatusBar(self.win, 'UPPER case', 1000)
    else:
        cv.displayStatusBar(self.win, 'LOWER case', 1000)
    return True
```

7.17 Update size of the text object

When text is edited, the size of the object changes. We use this function to get the new size:

```
def get_size(self):
    """Returns the text size and baseline under the forme (w, h), b."""
    d = self.text_options
    return cv.getTextSize(self.text, d['fontFace'], d['fontScale'],d['thickness'])
```

7.18 Creating the Node class

To place geometric elements into the window we are creating a **Node** class which has the following attributes:

- position (top left corner)
- size
- direction of the next object
- gap between adjacent objects

We store the default options as Node **class attribute**:

```
class Node:
    options = dict( pos=np.array((20, 20)),
                    size=np.array((100, 40)),
                    gap=np.array((10, 10)),
                    dir=np.array((0, 1)),
                    )
```

In the Node constructor, we can change these 4 options by specifing a named parameter. If the parameter is given in the form of a tuple, such as `size=(50, 20)` the tuple needs to be transformed into an **np.array**. Only the 4 elements of the options dictionary are updated:

```
def __init__(self, parent, **options):

    # update node options from constructor options
    for k, v in options.items():
        if k in Node.options:
```

(continues on next page)

(continued from previous page)

```
            if isinstance(v, tuple):
                v = np.array(v)
            Node.options[k] = v
```

Then we create empty instance attributes:

```
# create instance attributes
 self.pos = None
 self.size = None
 self.gap = None
 self.dir = None
```

We give them values from the node optionss:

```
# update instance attributes from node options
self.__dict__.update(Node.options)
```

Finally we calculate the next node position:

```
pos = self.pos + (self.size+self.gap)*self.dir
Node.options['pos'] = pos
```

7.18.1 Drawing the node

Nodes need to be drawn recursively. If a node has chidren, these need to be drawn as well. The `draw` method needs a position argument to draw the children with respect to the parent position. The default position is (0, 0). If the node is selected, a selection rectangle is drawn around it:

```
def draw(self, pos=np.array((0, 0))):
    x, y = pos + self.pos
    w, h =  self.size
    cv.rectangle(self.img, (x, y, w, h), RED, 1)
    if self.selected:
        cv.rectangle(self.img, (x-2, y-2, w+4, h+4), GREEN, 1)

    for child in self.children:
        child.draw(self.pos)
```

7.18.2 Checking if a position is inside

Using the **numpy** library makes 2D calculation easy. We can compare the components of a vector at once, such as `self.pos < pos`, which results in a boolean vector of the form `[True False]`. The function `all()` returns True if all vector components are True:

```
def is_inside(self, pos):
    """Check if the point (x, y) is inside the object."""
    pos = np.array(pos)
    return all(self.pos < pos) and all(pos < self.pos+self.size)
```

7.18.3 Finde the enclosure for children

If several nodes are placed inside another node, at the end the size of the parent nodes needs to be adapted to enclose all children. Here the `np.maximum` function finds the maximum coordinates of two vectors:

```
def enclose_children(self):
    p = np.array((0, 0))
    for node in self.children:
        p = np.maximum(p, node.pos+node.size)
    self.size = p
```

CHAPTER 8

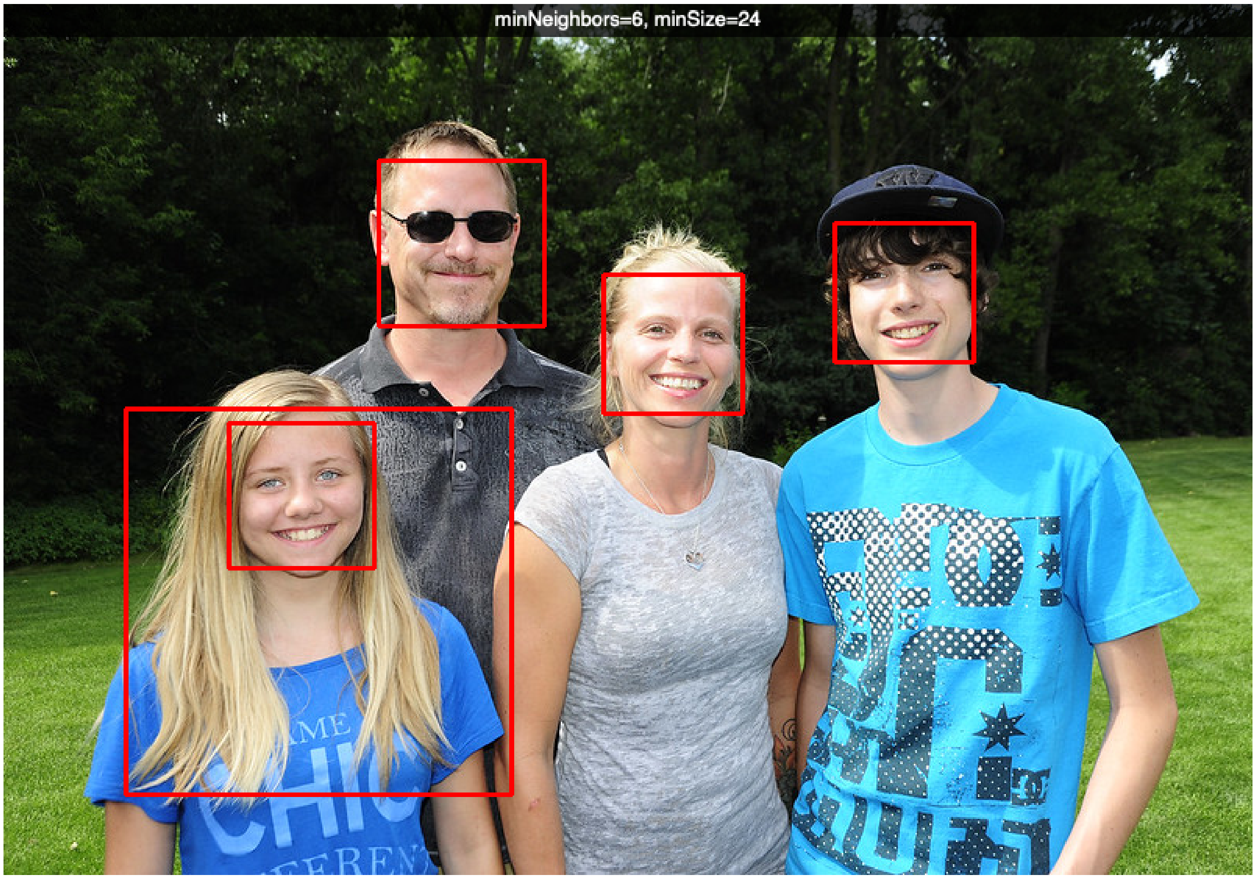
Detect faces

face/face1.png

```
import cv2 as cv

print(cv.__version__)
RED = (0, 0, 255)

img = cv.imread('family2.jpg')
cv.imshow('window', img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('window2', gray)

path = 'cascades/haarcascade_frontalface_default.xml'
face_detector = cv.CascadeClassifier(path)

face_rects = face_detector.detectMultiScale(gray,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30),
        flags = cv.CASCADE_SCALE_IMAGE)

print(f'found {len(face_rects)} face(s)')

for rect in face_rects:
    cv.rectangle(img, rect, RED, 2)

cv.imshow('window', img)
cv.waitKey(0)
```

8.1 Use trackbars to select parameters

The cascade detector allows to detect faces in an image.




```
import cv2 as cv

print(cv.__version__)
RED = (0, 0, 255)
scaleFactor = 1.1
minNeighbors = 5
minSize = (30, 30)

def detect():
    rects = face_detector.detectMultiScale(gray,
        scaleFactor=scaleFactor,
        minNeighbors=minNeighbors,
        minSize=minSize,
        flags=cv.CASCADE_SCALE_IMAGE)

    print(f'found {len(rects)} face(s)')

    img = img0.copy()
    for rect in rects:
        cv.rectangle(img, rect, RED, 2)
    cv.imshow('window', img)

def trackbar(x):
```

(continues on next page)

(continued from previous page)

```
    global minSize, minNeighbors, scaleFactor
    i = cv.getTrackbarPos('size','window')
    d = (24, 30, 60, 120)[i]
    minSize = (d, d)

    n = cv.getTrackbarPos('neighbors','window') + 1
    minNeighbors = n

    i = cv.getTrackbarPos('scale','window')
    s = (1.05, 1.1, 1.4, 2)[i]
    scaleFactor

    text = f'minNeighbors={n}, minSize={d}, scaleFactor={s}'
    cv.displayOverlay('window', text)
    detect()

img0 = cv.imread('family2.jpg')
img = img0.copy()
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

path = 'cascades/haarcascade_frontalface_default.xml'
face_detector = cv.CascadeClassifier(path)

detect()

cv.createTrackbar('neighbors', 'window', 0, 10, trackbar)
cv.createTrackbar('size', 'window', 0, 3, trackbar)
cv.createTrackbar('scale', 'window', 0, 3, trackbar)
cv.waitKey(0)
```

8.2 Video face detection

Now let's use the video camera to do face detection.

```
import cv2 as cv
import numpy as np
import time

path = 'cascades/haarcascade_frontalface_default.xml'
face_detector = cv.CascadeClassifier(path)

def detect():
    rects = face_detector.detectMultiScale(gray_s,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30),
        flags=cv.CASCADE_SCALE_IMAGE)

    for rect in rects:
        cv.rectangle(gray_s, rect, 255, 2)

cap = cv.VideoCapture(0)
t0 = time.time()

M = np.float32([[0.5, 0, 0], [0, 0.5, 0]])
size = (640, 360)

while True:
    # Capture frame-by-frame
    ret, frame = cap.read()

    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    gray_s = cv.warpAffine(gray, M, size)

    detect()

    cv.imshow('window', gray_s)
```

(continues on next page)

(continued from previous page)

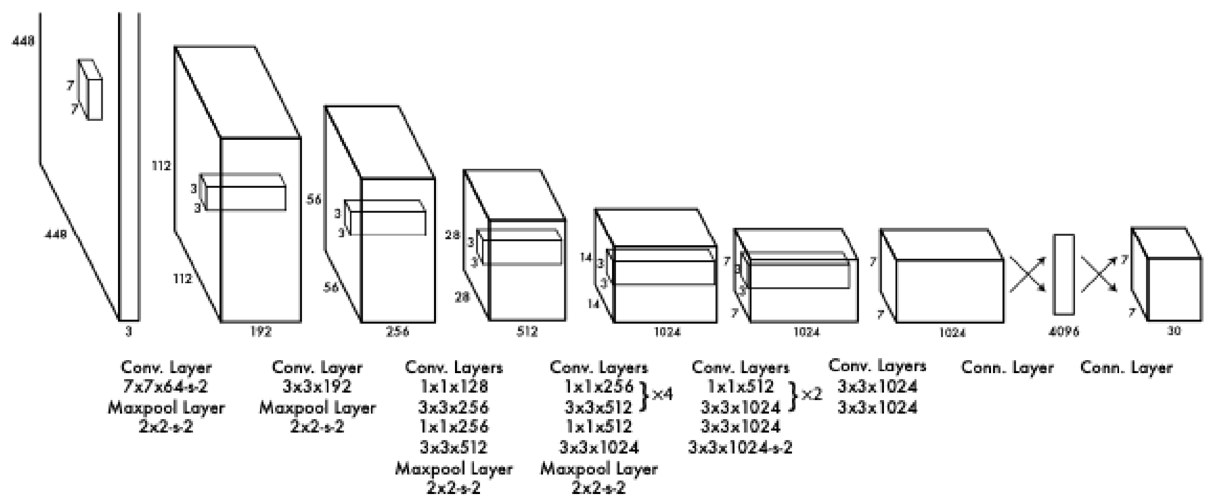
```
    t = time.time()
    cv.displayOverlay('window', f'time={t-t0:.3f}')
    t0 = t

    if cv.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv.destroyAllWindows()
```

CHAPTER 9

YOLO - object detection

YOLO — You Only Look Once — is an extremely fast multi object detection algorithm which uses convolutional neural network (CNN) to detect and identify objects.

The neural network has this network architecture.



Source: https://arxiv.org/pdf/1506.02640.pdf

This is our input image:

horse.jpg

9.1 Load the YOLO network

In order to run the network you will have to download the pre-trained YOLO weight file (237 MB). https://pjreddie.com/media/files/yolov3.weights

Also download the the YOLO configuration file.

yolov3.cfg

You can now load the YOLO network model from the harddisk into OpenCV:

```
net = cv.dnn.readNetFromDarknet('yolov3.cfg', 'yolov3.weights')
net.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV)
```

The YOLO neural network has 254 components. You can print them to the console with:

```
ln = net.getLayerNames()
print(len(ln), ln)
```

The 524 elements consist of convolutional layers (`conv`), rectifier linear units (`relu`) etc.:

```
254 ['conv_0', 'bn_0', 'relu_0', 'conv_1', 'bn_1', 'relu_1', 'conv_2', 'bn_2',
'relu_2', 'conv_3', 'bn_3', 'relu_3', 'shortcut_4', 'conv_5', 'bn_5', 'relu_5',
'conv_6', 'bn_6', 'relu_6', 'conv_7', 'bn_7', 'relu_7', 'shortcut_8', 'conv_9',
'bn_9', 'relu_9', 'conv_10', 'bn_10', 'relu_10', 'shortcut_11', 'conv_12', 'bn_12',
...
```

9.2 Create a blob

The input to the network is a so-called **blob** object. The function `cv.dnn.blobFromImage(img, scale, size, mean)` transforms the image into a blob:

```
blob = cv.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)
```

It has the following parameters:

- the **image** to transform
- the **scale** factor (1/255 to scale the pixel values to [0..1])
- the **size**, here a 416x416 square image
- the **mean** value (default=0)
- the option **swapBR=True** (since OpenCV uses BGR)

A blob is a 4D numpy array object (images, channels, width, height). The image below shows the red channel of the blob. You notice the brightness of the red jacket in the background.


```
# YOLO object detection
import cv2 as cv
import numpy as np
import time

img = cv.imread('images/horse.jpg')
cv.imshow('window',  img)
cv.waitKey(1)

# Give the configuration and weight files for the model and load the network.
net = cv.dnn.readNetFromDarknet('yolov3.cfg', 'yolov3.weights')
net.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV)
# net.setPreferableTarget(cv.dnn.DNN_TARGET_CPU)

ln = net.getLayerNames()
print(len(ln), ln)

# construct a blob from the image
blob = cv.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)
```

(continues on next page)

(continued from previous page)

```
r = blob[0, 0, :, :]

cv.imshow('blob', r)
text = f'Blob shape={blob.shape}'
cv.displayOverlay('blob', text)
cv.waitKey(1)

net.setInput(blob)
t0 = time.time()
outputs = net.forward(ln)
t = time.time()

cv.displayOverlay('window', f'forward propagation time={t-t0}')
cv.imshow('window',  img)
cv.waitKey(0)
cv.destroyAllWindows()
```

The blob object is given as input to the network:

```
net.setInput(blob)
t0 = time.time()
outputs = net.forward(ln)
t = time.time()
```

The forward propagation takes about 2 seconds on an MacAir 2012 (1,7 GHz Intel Core i5).

`yolo1.py`

And the 80 COCO class names.

`coco.names`

9.3 Identifiy objects

These two instructions calculate the network response:

```
net.setInput(blob)
outputs = net.forward(ln)
```

The `outputs` object are vectors of lenght 85

- 4x the bounding box (centerx, centery, width, height)
- 1x box confidence
- 80x class confidence

We add a slider to select the BoundingBox confidence level from 0 to 1.

The final image is this:


```
# YOLO object detection
import cv2 as cv
import numpy as np
import time

img = cv.imread('images/food.jpg')
cv.imshow('window',  img)
cv.waitKey(1)

# Load names of classes and get random colors
classes = open('coco.names').read().strip().split('\n')
np.random.seed(42)
colors = np.random.randint(0, 255, size=(len(classes), 3), dtype='uint8')

# Give the configuration and weight files for the model and load the network.
net = cv.dnn.readNetFromDarknet('yolov3.cfg', 'yolov3.weights')
net.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV)
# net.setPreferableTarget(cv.dnn.DNN_TARGET_CPU)

# determine the output layer
ln = net.getLayerNames()
ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]

# construct a blob from the image
blob = cv.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)
r = blob[0, 0, :, :]

cv.imshow('blob', r)
```

(continues on next page)

(continued from previous page)

```
text = f'Blob shape={blob.shape}'
cv.displayOverlay('blob', text)
cv.waitKey(1)

net.setInput(blob)
t0 = time.time()
outputs = net.forward(ln)
t = time.time()
print('time=', t-t0)

print(len(outputs))
for out in outputs:
    print(out.shape)

def trackbar2(x):
    confidence = x/100
    r = r0.copy()
    for output in np.vstack(outputs):
        if output[4] > confidence:
            x, y, w, h = output[:4]
            p0 = int((x-w/2)*416), int((y-h/2)*416)
            p1 = int((x+w/2)*416), int((y+h/2)*416)
            cv.rectangle(r, p0, p1, 1, 1)
    cv.imshow('blob', r)
    text = f'Bbox confidence={confidence}'
    cv.displayOverlay('blob', text)

r0 = blob[0, 0, :, :]
r = r0.copy()
cv.imshow('blob', r)
cv.createTrackbar('confidence', 'blob', 50, 101, trackbar2)
trackbar2(50)

boxes = []
confidences = []
classIDs = []
h, w = img.shape[:2]

for output in outputs:
    for detection in output:
        scores = detection[5:]
        classID = np.argmax(scores)
        confidence = scores[classID]
        if confidence > 0.5:
            box = detection[:4] * np.array([w, h, w, h])
            (centerX, centerY, width, height) = box.astype("int")
            x = int(centerX - (width / 2))
            y = int(centerY - (height / 2))
            box = [x, y, int(width), int(height)]
            boxes.append(box)
            confidences.append(float(confidence))
            classIDs.append(classID)

indices = cv.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
if len(indices) > 0:
    for i in indices.flatten():
        (x, y) = (boxes[i][0], boxes[i][1])
```

(continues on next page)

(continued from previous page)

```
        (w, h) = (boxes[i][2], boxes[i][3])
        color = [int(c) for c in colors[classIDs[i]]]
        cv.rectangle(img, (x, y), (x + w, y + h), color, 2)
        text = "{}: {:.4f}".format(classes[classIDs[i]], confidences[i])
        cv.putText(img, text, (x, y - 5), cv.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)

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

yolo2.py

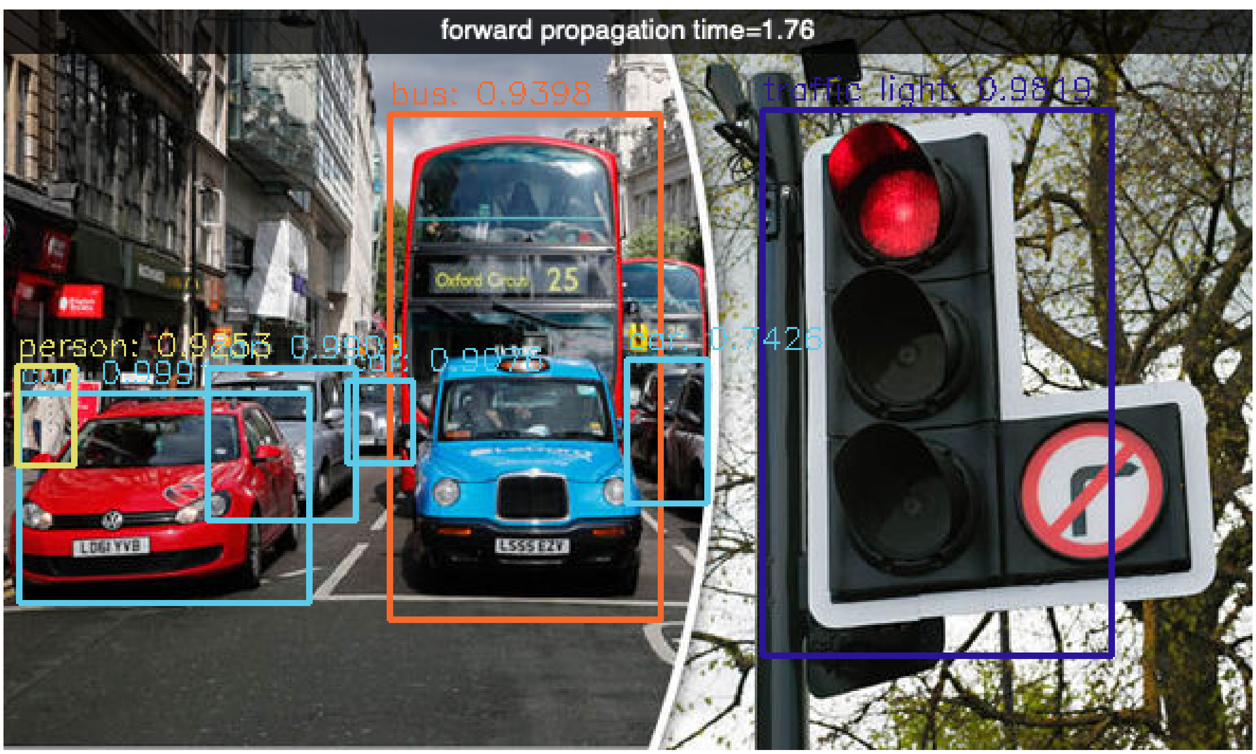
9.4 3 Scales for handling different sizes

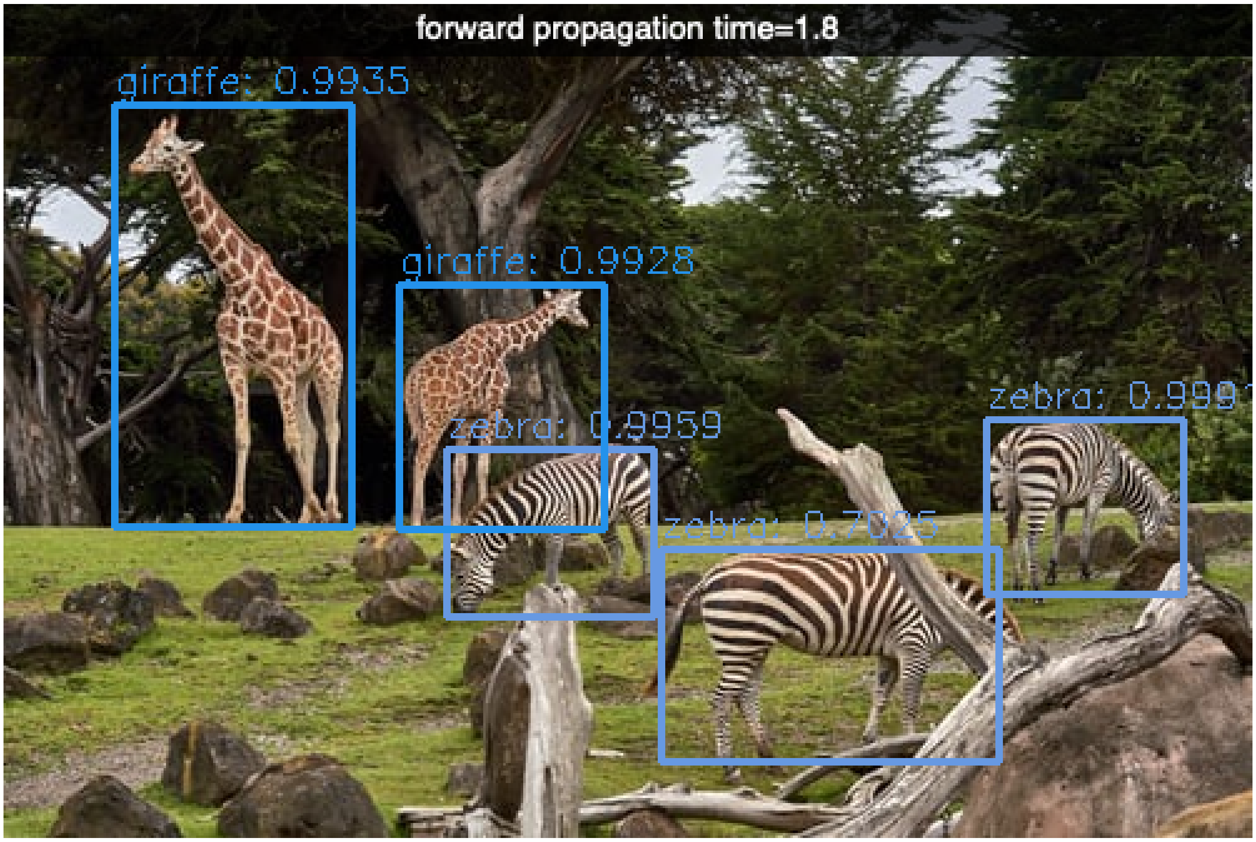
The YOLO network has 3 outputs:

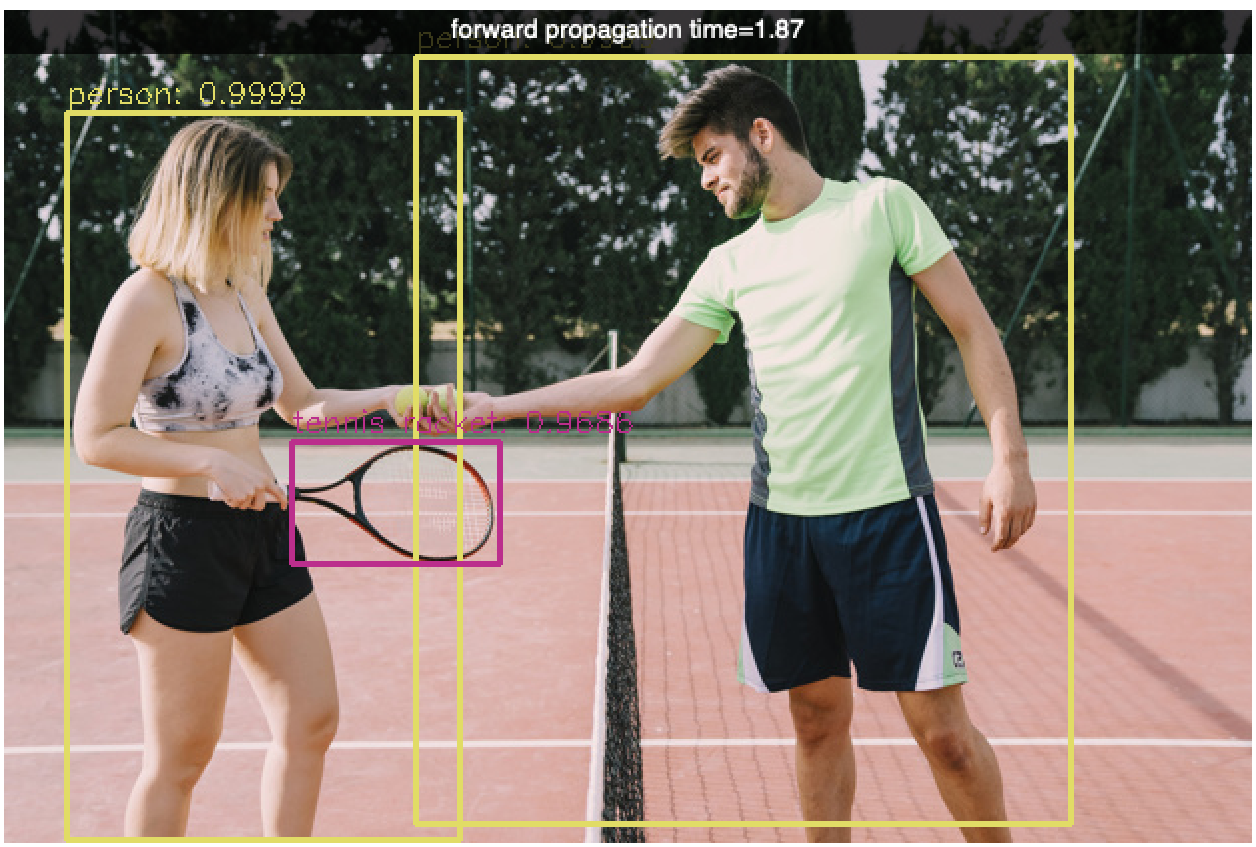
- 507 (13 x 13 x 3) for large objects
- 2028 (26 x 26 x 3) for medium objects
- 8112 (52 x 52 x 3) for small objects

9.5 Detecting objects

In this program example we are going to detect objects in multiple imgages.




forward propagation time=1.8
giraffe: 0.9935
giraffe: 0.9928
zebra: 0.9959
zebra: 0.7025


forward propagation time=1.87
person: 0.9999

```
# YOLO object detection
import cv2 as cv
import numpy as np
import time

WHITE = (255, 255, 255)
img = None
img0 = None
outputs = None

# Load names of classes and get random colors
classes = open('coco.names').read().strip().split('\n')
np.random.seed(42)
colors = np.random.randint(0, 255, size=(len(classes), 3), dtype='uint8')

# Give the configuration and weight files for the model and load the network.
net = cv.dnn.readNetFromDarknet('yolov3.cfg', 'yolov3.weights')
net.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV)
# net.setPreferableTarget(cv.dnn.DNN_TARGET_CPU)

# determine the output layer
ln = net.getLayerNames()
ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]

def load_image(path):
    global img, img0, outputs, ln

    img0 = cv.imread(path)
    img = img0.copy()

    blob = cv.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)

    net.setInput(blob)
    t0 = time.time()
    outputs = net.forward(ln)
    t = time.time() - t0

    # combine the 3 output groups into 1 (10647, 85)
    # large objects (507, 85)
    # medium objects (2028, 85)
    # small objects (8112, 85)
    outputs = np.vstack(outputs)

    post_process(img, outputs, 0.5)
    cv.imshow('window',  img)
    cv.displayOverlay('window', f'forward propagation time={t:.3}')
    cv.waitKey(0)

def post_process(img, outputs, conf):
    H, W = img.shape[:2]

    boxes = []
    confidences = []
    classIDs = []

    for output in outputs:
        scores = output[5:]
```

(continues on next page)

(continued from previous page)

```
        classID = np.argmax(scores)
        confidence = scores[classID]
        if confidence > conf:
            x, y, w, h = output[:4] * np.array([W, H, W, H])
            p0 = int(x - w//2), int(y - h//2)
            p1 = int(x + w//2), int(y + h//2)
            boxes.append([*p0, int(w), int(h)])
            confidences.append(float(confidence))
            classIDs.append(classID)
            # cv.rectangle(img, p0, p1, WHITE, 1)

    indices = cv.dnn.NMSBoxes(boxes, confidences, conf, conf-0.1)
    if len(indices) > 0:
        for i in indices.flatten():
            (x, y) = (boxes[i][0], boxes[i][1])
            (w, h) = (boxes[i][2], boxes[i][3])
            color = [int(c) for c in colors[classIDs[i]]]
            cv.rectangle(img, (x, y), (x + w, y + h), color, 2)
            text = "{}: {:.4f}".format(classes[classIDs[i]], confidences[i])
            cv.putText(img, text, (x, y - 5), cv.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)

def trackbar(x):
    global img
    conf = x/100
    img = img0.copy()
    post_process(img, outputs, conf)
    cv.displayOverlay('window', f'confidence level={conf}')
    cv.imshow('window', img)

cv.namedWindow('window')
cv.createTrackbar('confidence', 'window', 50, 100, trackbar)
load_image('images/horse.jpg')
load_image('images/traffic.jpg')
load_image('images/zoo.jpg')
load_image('images/kitchen.jpg')
load_image('images/airport.jpg')
load_image('images/tennis.jpg')
load_image('images/wine.jpg')
load_image('images/bicycle.jpg')

cv.destroyAllWindows()
```

yolo3.py

9.6 Sources

- https://arxiv.org/pdf/1506.02640.pdf
- https://en.wikipedia.org/wiki/Object_detection
- https://github.com/StefanPetersTM/TM
- https://www.cyberailab.com/home/a-closer-look-at-yolov3

Tutorials:

- https://www.pyimagesearch.com/2017/08/21/deep-learning-with-opencv/

- https://www.learnopencv.com/deep-learning-based-object-detection-using-yolov3-with-opencv-python-c/

CHAPTER 10

Nodes

Nodes are the elements of the window which are used as the base class to create widgets and shapes. Nodes are the elements of a tree graph. Each window is a root of the tree and can have multiple nodes as children. Nodes can have children of their own.

Widgets are user control items such as text fields, buttons, comboboxes, entry fields, and listboxes.

Shapes are the geometrical formes such as markers, lines, arrows, rectangles, circles, ellipses and polygones.

class node1.**Demo**

10.1 Node options

Each node has 4 attributes (options):

- position
- size
- gap between objects

- direction towards the next object

The Node options are reset at the creation of a new window. They are in the format of **numpy** int64 arrays. The advantage of using numpy arrays is that we can do vector addition. For example the lower right corner is simply:

```
p1 = pos + size
```

The node options are stored as a dictionary inside the Window class:

```
class Window:
    """Create a window for the app."""
    node_options = dict(pos=np.array((20, 20)),
                        size=np.array((100, 20)),
                        gap=np.array((10, 10)),
                        dir=np.array((0, 1)),
                        )
```

When creating a new window, the initial node options are reassigned:

```
Node.options = Window.node_options.copy()
```

10.2 Parents and children

The window is the parent of the first-level children. At window creation an empty children list is created:

```
self.children = []  # children
```

At that point the window is the parent of the children to add. Parents are stored in a stack. Initialy the window is the parent for the first-level children. So at window creation, the window itself is added to the parent stack:

```
self.stack = [self]  #  parent stack
```

At that point no node exists, so the active node is set to None:

```
self.node = None  # currently selected node
```

The attribute *level* decides the point of attachement of the new node:

level = 0 The last parent remains the parent and a new sibling to the last is created.

level = 1 The level is increased and the last child becames the new parent. The new child is a great-child of the previos parent.

level = -1 The level is decreased and the grand-parent becomes the new parent. The new child is a sibling to the previous parent

10.3 Enclosing nodes

The following exemple shows a first node, folloed by 3 nodes at a child level, then 4 nodes at the parent level, with a change of direction:

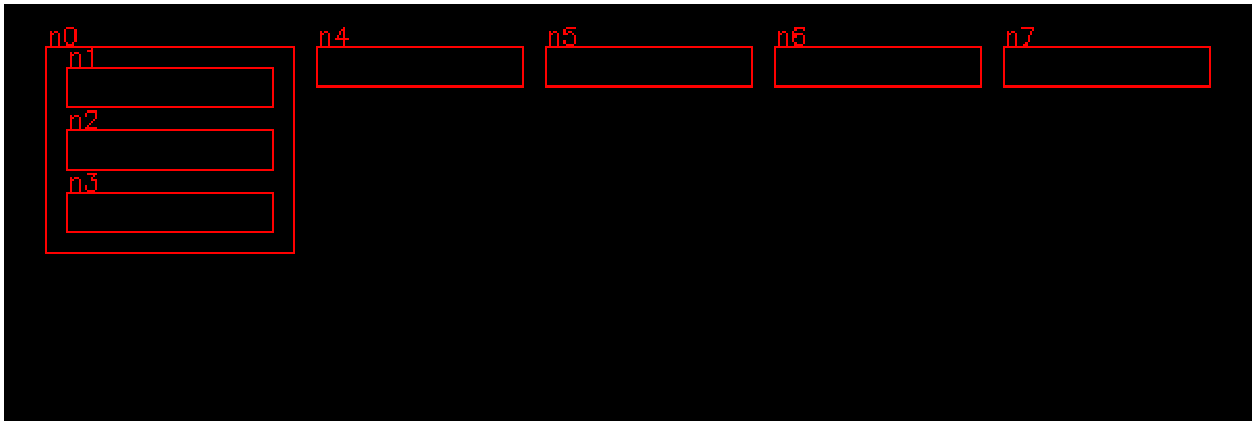
```
class Demo(App):
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        Node()
        Node(level=1)
        Node()
        Node()
        Node(level=-1, dir=(1, 0))
        Node()
        Node()
        Node()
```

class node2.**Demo**



In the next example node 6 increases level again, and changes direction to vertical. The parent of the last nodes is forced to enclcose its children:

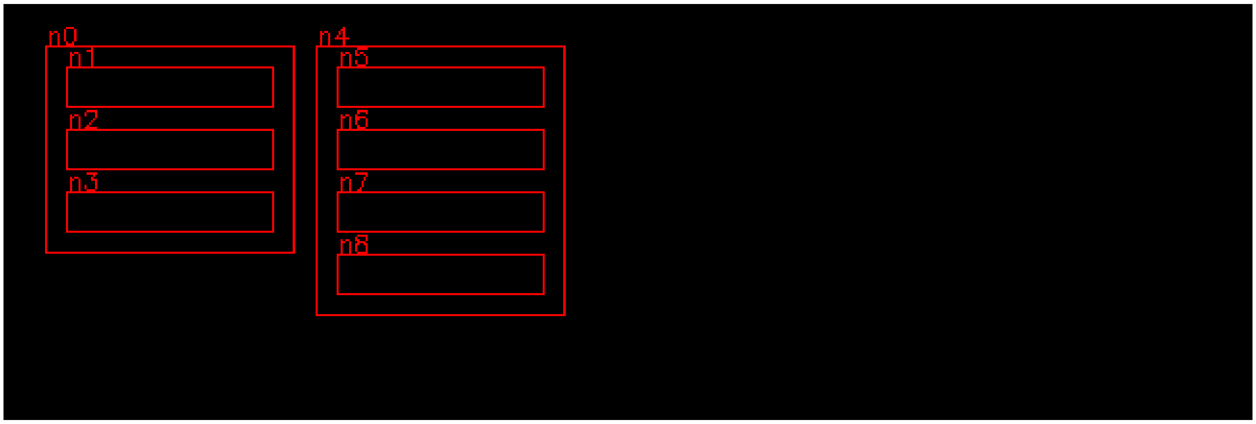
```
Node()
Node(level=1)
Node()
Node()
Node(level=-1, dir=(1, 0))
Node(level=1, dir=(0, 1))
Node()
Node()
Node().parent.enclose_children()
```

10.4 Embedded nodes

Nodes can be embedded in other nodes. In the example below node 1 is embedded in node 0, node 3 and 4 is embedded in node 2. This is the code:

```
Node()
Node(level=1)
Node()
Node(level=1)
Node()
Node(level=-1)
Node()
Node(level=-1)
Node()
```

class node4.**Demo**



In the following example, we go down 3 levels:

- node 1 is embedded in node 0
- node 2 is embedded in node 1
- node 3 is embedded in node 2

This is the code:

```
Node()
Node(level=1)
Node(level=1)
Node(level=1)
Node(level=-1)
Node(level=-1)
Node(level=-1)
```

class node5.**Demo**

10.5 Decrease multiple levels

While we can go down at most one level, it is possible to go up multiple levels at once. If level is negative we repeat this:

- enclose the children of the current parent

- make the grand-parent the current parent

This is the code:

```
for i in range(-level):
    self.win.current_parent.enclose_children()
    self.parent = self.win.current_parent.parent
    self.win.current_parent = self.parent
```

Here is the previous example where we go up 3 levels at once, instead of one by one:

```
Node()
Node(level=1)
Node(level=1)
Node(level=1)
Node(level=-3)
Node()
Node()
```

class node6.**Demo**

10.6 Changing the direction of node placement

New nodes are placed according to the direction *dir* vector. This can be:

- vertical (0, 1)
- horizontal (1, 0)
- diagonal (1, 1)

Here is an example of 5 nodes placed in vertical, horizontal and two diagonal directions:

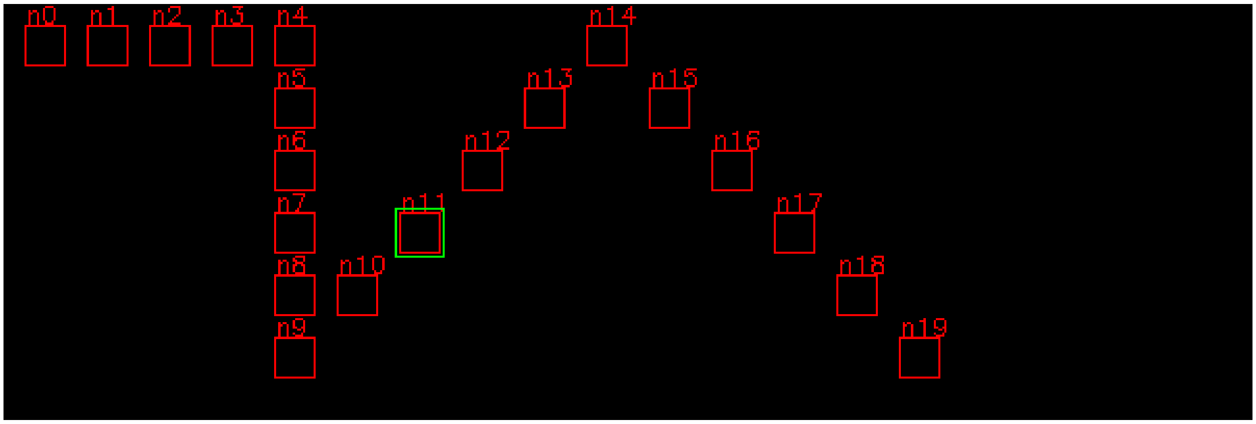
```
for i in range(5):
    Node(dir=(1, 0), size=(20, 20))

for i in range(5):
    Node(dir=(0, 1))

for i in range(5):
    Node(dir=(1, -1))
```

(continues on next page)

(continued from previous page)

```
for i in range(5):
    Node(dir=(1, 1))
```

class node7.**Demo**



10.7 Toggle frames

Displaying frames is mostly needed for understanging the node frame structure, and during debugging. It is convenient to turn it off or on either the window level or the node level. For this we create a new Window instance attribute:

```
self.frame = True
```

and a Node instance attribute:

```
self.frame = True
```

Inside the Node `draw()` method we are using both flags:

```
if self.win.frame and self.frame:
    cv.rectangle(self.img, (x, y, w, h), RED, 1)
    label = 'n{}'.format(self.id)
    cv.putText(self.img, label, (x, y-1), 0, 0.4, RED, 1)
```

Inside the Window class we define a new method to toggle the flag:

```
def toggle_frame(self):
    self.frame = not self.frame
```

Finally we add a new shortcut to the Window class:

```
self.shortcuts = {'\t': self.select_next_node,
                  chr(27): self.unselect_node,
                  chr(0): self.toggle_shift,
                  'f': self.toggle_frame }
```

10.8 Nodes based on points

We are going to create a new Node class which is defined by a list of points.

10.9 Executing commands when clicking a node

In order to react to a mouse click inside a node, we add a **cmd** attribute. There are several places to modify. First we add it to the default node options in the Window class:

```
node_options = dict(pos=np.array((20, 20)),
                    size=np.array((100, 20)),
                    ...
                    cmd=None)
```

Then we add a new attribute in the Node class:

```
self.cmd = options.get('cmd', None)
```

and finally we call it in the **mouse** callback method:

```
def mouse(self, event, pos, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        self.cmd()
        for child in self.children:
            child.selected=False
            if child.is_inside(pos-self.pos):
                child.selected=True
                child.cmd()
```

The following example we association three callback functions to three nodes:

- nodo 0 - prints the help menu
- node 1 - toggles visibility
- node 2 - creates a new Text instance

This is the code:

```
Node(cmd=help)
Node(cmd=App.win.toggle_visible)
Node(cmd=Text)
```

class node11.**Demo**

CHAPTER 11

Basic shapes

In this section we going create classes to add basic shapes to an image:

- Marker
- Line
- Arrow
- Rectangle
- Circle
- Ellipse
- Polygon

11.1 Finding OpenCV attributes

OpenCV has 1912 attributes, which can be verified with the following command:

```
>>> len(dir(cv))
2190
```

We define a small function for matching this large attribute list with a regular expression:

```
def cv_dir(regex):
    atts = dir(cv)
    return [s for s in atts if re.match(regex, s)]
```

We use it to find the markers

```
>>> cv_dir('MARKER.*')
['MARKER_CROSS',
 'MARKER_DIAMOND',
```

(continues on next page)

(continued from previous page)

```
'MARKER_SQUARE',
'MARKER_STAR',
'MARKER_TILTED_CROSS',
'MARKER_TRIANGLE_DOWN',
'MARKER_TRIANGLE_UP']
```

11.2 Marker

We base the Marker class on the Node class. At first we set the options as class attribute of the Marker class:

```
class Marker(Node):
    options = dict( color=GREEN,
                    markerType=cv.MARKER_CROSS,
                    markerSize=20,
                    thickness=1,
                    line_type=8)
```

Then we define the `__init__()` method, which only has options. Four of them (pos, size, gap, dir) are applied to Node, and the rest are specific to the Marker class (color, markerType, markerSize, thickness, line_type). The method `set_class_options()` sets these options:

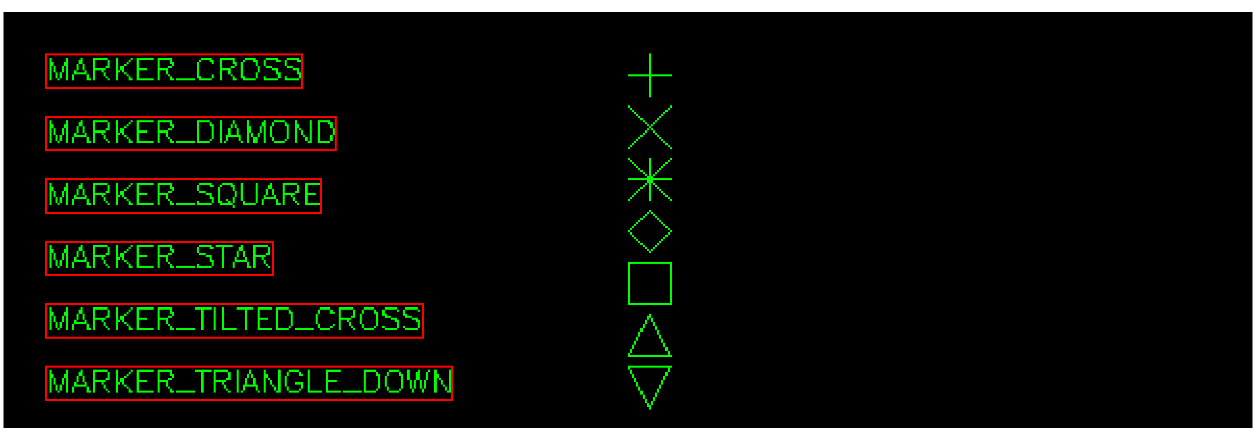
```
def __init__(self, **options):
    super().__init__(**options)
    self.set_class_options(options)
```

We set the size to 20x20 which is the size of the markers. To better see the markers, we do not display the frame:

```
self.size = np.array((20, 20))
self.frame = False
cv.imshow(self.win.win, self.img)
```

Finally we create the `draw()` method:

```
def draw(self, pos=np.array((0, 0))):
    super().draw(pos)
    x, y = pos + self.pos + (10, 10)
    cv.drawMarker(self.img, (x, y), **self.options)
```



```
"""Show the different markers."""
from cvlib import *

class Demo(App):
    def __init__(self):
        super().__init__()

        Window()
        markers = cv_dir('MARKER.*')

        for marker in markers:
            Text(marker, fontScale=0.5, thickness=1)

        for m in range(7):
            Marker(pos=(300, m*25+20), markerType=m)

if __name__ == '__main__':
    Demo().run()
```

https://docs.opencv.org/master/d6/d6e/group__imgproc__draw.html

CHAPTER 12

Widgets

A widget is a control element in a graphial user interface. The trackbar is the only native widget OpenCV has. In this section we are going to add:

- Text
- Button
- Listbox
- Entry
- Spinbox

12.1 Trackbar

The only GUI element OpenCV provides is a trackbar. This is an example to add a trackbar to the Window and call the trackbar callback function:

```
class Demo(App):
    def __init__(self):
        super().__init__()

        Window()
        Text('Trackbar')
        cv.createTrackbar('x', App.win.win, 50, 100, self.trackbar)

    def trackbar(self, pos):
        print(pos)
```

12.2 Text

Displaying text is important. OpenCV uses the Hershey fonts:

```
class Text(Node):
    options = dict(fontFace=cv.FONT_HERSHEY_SIMPLEX,
                fontScale=1,
                color=GREEN,
                thickness=2,
                lineType=cv.LINE_8,
                bottomLeftOrigin=False)

    def __init__(self, text='Text', **options):
        super().__init__(**options)
        self.set_class_options(options)
        self.text = text
        (w, h), b = self.get_size()
        self.size = np.array((w, h+b))
```

12.2.1 Font scale

The size of the font is given to the text as the `fontScale` argument. In the example below we display 4 different scales:

```
for scale in (0.5, 1, 2, 3):
    text = 'fontScale={}'.format(scale)
    Text(text, fontScale=scale, thickness=1, color=YELLOW)
```

12.2.2 Font type

OpenCV uses the **Hershey fonts** which are a collection of fonts developped in 1967 by Dr. Allen Vincent Herschey at the Naval Weapons Laboratory, to be rendered on early cathod ray tube displays[1].

class text2.**Demo**

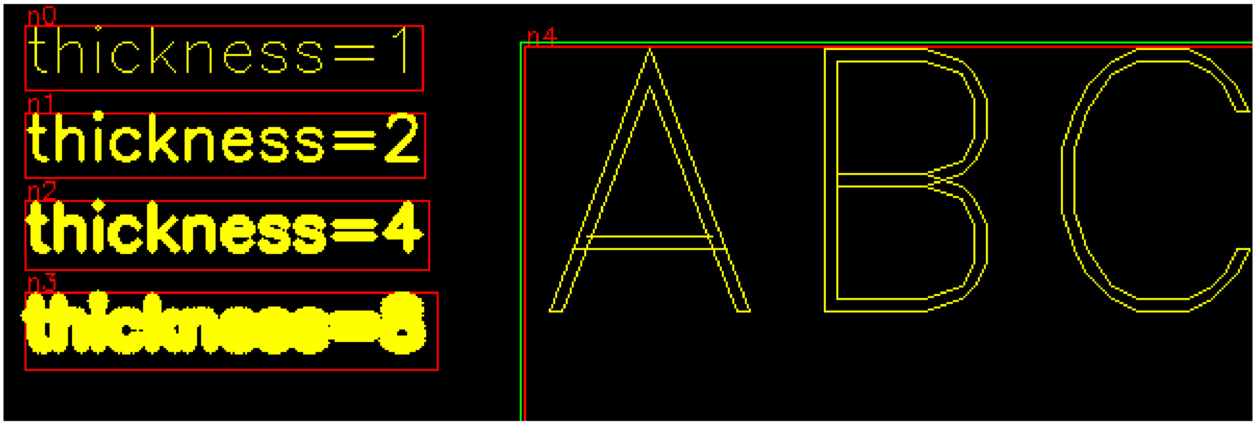
12.2.3 Font thickness

The following code displays different thickness for the font:

```
for t in (1, 2, 4, 8):
    text = 'thickness={}'.format(t)
    Text(text, thickness=t, color=YELLOW)

Text('ABC', pos=(250, 20), fontScale=6, thickness=1,
    fontFace=cv.FONT_HERSHEY_DUPLEX)
```

class text3.**Demo**

[1] https://en.wikipedia.org/wiki/Hershey_fonts

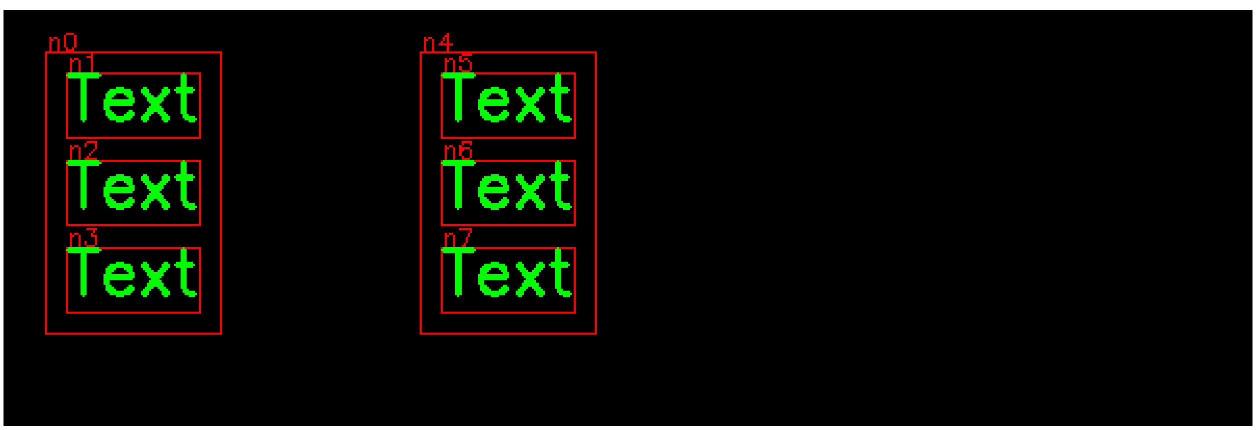


12.2.4 Placing text inside a Node

Text can be placed and grouped inside a node. The elements inside the encloser move together. In the exemple below we have two groups with 3 text fields inside:

```
Node()
Text(level=1)
Text()
Text()

Node(level=-1, pos=(200, 20))
Text(level=1)
Text()
Text().parent.enclose_children()
```

class text4.**Demo**



12.3 Button

12.4 Entry

12.5 Combobox

12.6 Listbox

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Index

D