2 TYPES OF IMAGES

1. Raster Images

Raster (or bitmap) images are generally what you think about when thinking of images. These are the types of images that are produced when scanning or photographing an object. Raster images are compiled using pixels, or tiny dots, containing unique color and tonal information that come together to create the image.

Since raster images are pixel based, they are resolution dependent. The number of pixels that make up an image as well as how many of those pixels are displayed per inch, both determine the quality of an image. As you may have guessed, the more pixels in the image and the higher the resolution is, the higher quality the image will be.

For example, if we scale a raster image to enlarge it, without changing resolution, it will lose quality and look blurry or pixilated. This is because we are *stretching* the pixels over a larger area, thus making them look less sharp

**Common Raster Image Types: JPG, TIFF, GIF, PNG, BMP**

1. Vector Images

Instead of trying to keep track of the millions of tiny pixels in a raster image, vector images, or line art, keep track of points and the equations for the lines that connect them. Generally speaking, vector images are made up of paths or line art that can infinitely scalable because they work based on algorithms rather than pixels.

One of the greatest things about **vector images is that you can re-size them infinitely larger or smaller, and they will still print out just as clearly, with no increase (or decrease) in file size.**

***Common Vector File Types: EPS, SVG****. Also,* almost all computer font files are based on vector images of the letters - that's why it's possible to scale them WAY up or WAY down and still have the letters be clear.  All Microsoft Office clipart uses vector art and most charts and graphs produced by Office or by statistical analysis software are vector-based.

## **VideoWriter() : Used to write a video file**

|  |  |  |  |
| --- | --- | --- | --- |
| cv::VideoWriter::VideoWriter | ( | const [**String**](https://docs.opencv.org/3.4/d1/d8f/classcv_1_1String.html) & | filename, |
|  |  | int | fourcc, |
|  |  | double | fps, |
|  |  | [**Size**](https://docs.opencv.org/3.4/dc/d84/group__core__basic.html#ga346f563897249351a34549137c8532a0) | frameSize, |
|  |  | bool | isColor = true |
|  | ) |  |  |

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**Parameters**

|  |  |
| --- | --- |
| **filename** | Name of the output video file. |
| **fourcc** | 4-character code of codec used to compress the frames. For example, **[VideoWriter::fourcc](https://docs.opencv.org/3.4/dd/d9e/classcv_1_1VideoWriter.html" \l "afec93f94dc6c0b3e28f4dd153bc5a7f0" \o "Concatenates 4 chars to a fourcc code. )**('P','I','M','1') is a MPEG-1 codec, **[VideoWriter::fourcc](https://docs.opencv.org/3.4/dd/d9e/classcv_1_1VideoWriter.html" \l "afec93f94dc6c0b3e28f4dd153bc5a7f0" \o "Concatenates 4 chars to a fourcc code. )**('M','J','P','G') is a motion-jpeg codec etc. List of codes can be obtained at [MSDN](https://docs.microsoft.com/en-us/windows/win32/medfound/video-fourccs) page or with this [archived page](https://web.archive.org/web/20220316062600/http:/www.fourcc.org/codecs.php) of the fourcc site for a more complete list). FFMPEG backend with MP4 container natively uses other values as fourcc code: see [ObjectType](http://mp4ra.org/" \l "/codecs), so you may receive a warning message from OpenCV about fourcc code conversion. |
| **fps** | Framerate of the created/ OUTPUT video stream. (ur choice) |
| **frameSize** | Size of the video frame as a tuple (width, height). (INPUT Video frame size has to be passed here) |
| **isColor** | If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames (the flag is currently supported on Windows only). |

HINT:

* Use, cv2.VideoWriter\_fourcc(\*'XVID') ---- for avi formats
* Use, cv2.VideoWriter\_fourcc(\*'mp4v') ---- for mp4 formats
* Use cv2.VideoWriter\_fourcc('p', 'n', 'g', ' ') ------ for lossless compressions/ writing

1. cv2.waitKey(10) & 0xFF == ord('q'):

# *if cv2.waitKey(10) & 0xFF == ord('q'):*

*break*

cv2.waitKey() returns the value -1 if no key is pressed.

But, When you press a key, it returns the ASCII value of that key. So if you do

k = cv2.waitKey(10)

if k == ord('q'):

break

when the q key is pressed, the value of k will be 27 which is equal to the value of ord('q'), ie 27 and this conditional will be true resulting in break. Also, Thus, cv2.waitKey(10) -- means that what ever is your output will stay on screen for 10 ms, at most.

Also, we can have ,

# *if cv2.waitKey(0) & 0xFF == ord('q'):*

*break*

1. Thus, cv2.waitKey(0) -- 0 means that what ever is your output will stay on screen for 0ms i.e infinite time period
2. 0xFF == ord('q') -- means taking keyboard input. here its 'q'

in normal term i say this is trying to say keep output open until user press 'q' in its keyboard.

Example 2:

**import cv2**

**cvim2disp = cv2.imread('data/home.jpg')**

**cv2.imshow('HelloWorld', cvim2disp)**

**cv2.waitKey(0)**

**cv2.destroyWindow('HelloWorld')**

waitKey(0) method is waiting for an input infinitely. When you see a frame of the corresponding image, **do not try to close the image using close in top right corner. Instead press some key**. Then, waitkey method will take that as an input and it will return back a value. Further you can also check which key was pressed to close the frame.

Additionally waitKey(33) will keep the frame active for 33 ms and then close it automatically.

destroyWindow() will destroy the current frame if there

& destroyAllWindows() will destroy all the frames currently present.

COLOR CONVERSION OF AN IMAGE:

1. **RGB->gray conversion** --- Here, different channels are weighed differently.

Specifically:

gray\_pixel = 0.114 \* blue\_pixel + 0.299 \* red\_pixel + 0.587 \* green\_pixel

Code:

gray=cv2.cvtColor(image2, cv2.COLOR\_BGR2GRAY) # converts BGR to Greyscale Image

plt.imshow(gray) # gives the image as output

1. new =cv2.cvtColor(image2, cv2.COLOR\_BGR2HSV)

plt.imshow(new)

# Using HSV color space. HSV(hue saturation value) color space is mostly used for object tracking algorithms.

**NOTE:** **A RGB image consists of the color intensity of different color channels, i.e. the intensity and color information are mixed in RGB color space but in HSV color space the color and intensity information are separated from each other. This makes HSV color space more robust to lighting changes.**

1. new2 =cv2.cvtColor(image2, cv2.COLOR\_RGB2BGR) # converts RGB to BGR

plt.imshow(new2)

etc. etc. etc.

ROTATING AN IMAGE ABOUT ITS CENTRE:

1. At 45 degress clockwise:

Rotating an image about the center point requires that we first calculate the center coordinates of the image.

h,w,d = image2.shape

center = (w // 2, h // 2) # using "//" to get integer (no floating pt values)

#let's rotate an image 45 degrees clockwise using OpenCV by first computing the image center and then constructing the rotation matrix M, and then finally applying the affine warp (cv2.warpAffine()).

**Recall from Maths**: that “ -ve “ degree rotation is for clockwise rotation and “+ve” degree is for counter-clockwise rotation

M = cv2.getRotationMatrix2D(center, -45, 1.0)

rotated = cv2.warpAffine(image2, M, (w, h))

plt.imshow(rotated)

FACE DETECTION USING OPEN CV:

#### **Loading Haar-cascade in OpenCV**

**Step 1** :We can load haar-cascade XML file Classifiers using cv2.CascadeClassifier function which will be used for face and eye detection

face\_detector=cv2.CascadeClassifier(‘haarcascade\_frontalface\_default.xml’)

eye\_dectector = cv2.CascadeClassifier(‘haarcascade\_eye.xml’)

**Step 2**: Then, we need to load test image and convert to greyscale image:

img = cv2.imread('Indian\_team.jpg')

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

**Step 3:** We can call the detector function once the XML file is loaded .

faces = face\_detector.detectMultiScale(gray, 1.03, 4)

Parameters are:

cv2.CascadeClassifier.detectMultiScale(image[, scaleFactor[, minNeighbors[, flags[, minSize[, maxSize]]]]])

1. **image** : (= gray scale image only) Matrix of the type CV\_8U containing an image where objects are detected.
2. **scaleFactor** : Parameter specifying how much the image size is reduced at each image scale. Suppose, the scale factor is 1.03, it means we're using a small step for resizing, i.e. reduce size by 3 %, we increase the chance of a matching size with the model for detection is found, while it's expensive. (use trial and error to get value)
3. **minNeighbors** : Parameter specifying how many neighbors each candidate rectangle should have to retain it. **This parameter will affect the quality of the detected faces: higher value results in less detections but with higher quality**. We're using 4 in the code. (use trial and error to get value)
4. Output = faces, contains a list of coordinates for the rectangular regions where faces were found. We use these coordinates to draw the rectangles in our image. **If faces are found, it returns the positions of detected faces as Rect(x,y,w,h).**

**Step 4 :** Draw rectangle around the detected faces

for (x,y,w,h) in faces:

cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0), 2 )

***Syntax:****cv2.rectangle(image, start\_point, end\_point, color, thickness)*

*Here, start\_point is (x,y) is taken as input from faces because “faces” would return 4 values which are stored in x,y,w,h and later used as input arg to cv2.rectangle().*

*Slly, end\_point is x+w, y+h is calculated using above 4 values .*

***color:****It is the color of border line of rectangle to be drawn. For****example****, we pass a tuple. eg: (255, 0, 0) for blue colored rectangle.****thickness:****It is the thickness of the rectangle border line .*

**Step 5**: plot the image with rectangles

plt.imshow(img)