The World of Computer Vision: An introduction to OpenCV

Dharmik Shah ©
dharmik.shah@mail.utoronto.ca
September 5, 2020

ABSTRACT

The purpose of this paper is to introduce the reader to the popular computer vision library called OpenCV, short for Open Source Computer Vision. OpenCV was released as a C++ library by Intel back in 1999. The main use cases revolved around image and video analysis, although OpenCV can easily be combined with many other machine learning libraries like Tensorflow and Numpy to build complex systems. In 2009, a major revampment took place to the C++ code, which resulted in OpenCV 2. After the publishing of the library on GitHub, over 1,100 individuals contributed to making over 2,500 algorithms. The library also has popular wrappers in the Python and Java languages. As of 2020, the library has had over 18 million downloads and over 40,000 active users. In computer vision, OpenCV is a must know, and learning only 20% of the material can produce 80% of the results. This paper intends to deliver comprehensive information about OpenCV and it's usage in Python.

Keywords

Computer Vision, Machine Learning, Artificial Intelligence

1. INTRODUCTION

Computer vision is a sub-field of artificial intelligence which deals with how computers are able to gain a high-level understanding of images and videos. Put simply, computer vision deals with performing analysis on images and videos through a program [1]. Simple tasks like object recognition are trivial in the eyes of a human, but for a computer, it wasn't always this way. Due to advancements in the field, tasks like these are suddenly possible with very little code. One such library that allows this is called OpenCV, short for Open Source Computer Vision Library, released by Intel in 1999 as a C++ library [2]. Back in 2009, a major change took place to the C++ library, resulting in OpenCV version 2, with it's code published on GitHub under the BSD license [3]. Over the past decade, more than 1,100 individuals contributed to the project, implementing even faster algorithms to perform image analysis [4]. Overall the library has over 2,500 algorithms used in many areas of computer vision [5]. Although the project started off as a C++ library, individual efforts led to wrappers in the Python and Java language. OpenCV, when combined with other libraries like Google's Tensorflow can prove to be very powerful. A library that has over 18 million downloads and more than 45,000 active users deserves some recognition. As the Pareto principle states, about 80% of the results come from 20% of the efforts [6]. In other words, learning only 20% of OpenCV is enough to complete 80% of computer vision tasks, and this paper will introduce the user to key concepts that help create that 80%.

2. LAYOUT OF PACKAGE

OpenCV follows roughly the same file structure in all 3 major languages. In Python particularly, OpenCV is imported as *cv2*. OpenCV is separated into modules, with each module handling a certain area of computer vision [7]. In Python, modules are referred to using the . (dot). For example, computer vision dealing with machine learning can be used by first typing the prefix *cv2.ml*. If the interests of the user lie more in deep neural networks, the module *cv2.dnn* would be of more use [8]. Realistically however, very common tasks like image filters, edge detection and more are simply available with the prefix *cv2*. As mentioned previously, only 20% of OpenCV will likely be used, and particularly this refers to the main module [9].

3. READING AN IMAGE OR VIDEO

Before diving deep into how OpenCV works under the hood, it is important to discuss two actions that will appear in every project: reading and showing an image. To read an image, the *imread* method is used as follows:
img = cv2.imread('cat.png')
To show an image, the *imshow* method is used:
cv2.imshow('window1', img)
The first parameter is the window name, which should be unique if multiple images want to be shown at once. By default, Python will show the window and close it right away because there is nothing blocking the termination of the program. To make sure that the window(s) don't close right away, an infinite delay is added using the *waitKey* method [10]. Reading an image can also be done through other Python libraries like Pillow and Matplotlib, but this paper will focus on using OpenCV the entire route. What follows is the code and the illustration of the cat:

```
import cv2
img = cv2.imread('cat.png')
cv2.imshow('window1', img)
cv2.waitKey(0)
```

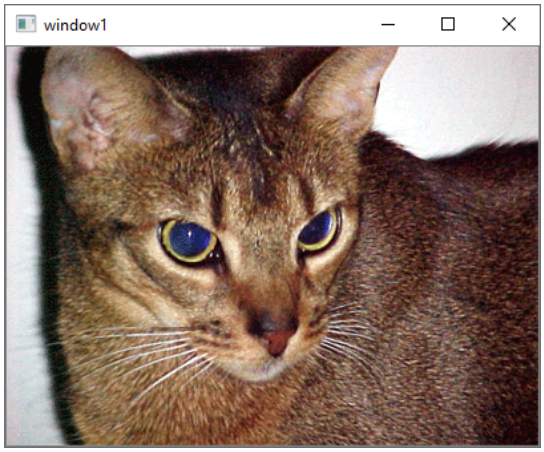

Figure 1: A 400x300 cat image

But what about a video? Remember that a video is simply a continuous set of images, or frames [11]. To read a video, the user need to specify the source of input through *cv2.VideoCapture(0)*, and just keep reading forever. The parameter *0* refers to the primary video source, which is the built-in webcam on most laptops, or a USB webcam on a PC:

```
import cv2
cap = cv2.VideoCapture(0)
while True:
    success, img = cap.read()
    cv2.imshow("Result", img)
    key = cv2.waitKey(1) & 0xFF
    if key == ord("q"):
        break

cv2.destroyAllWindows()
```

If the user presses the character *'q'*, the loop will break and all existing windows will be destroyed.

4. HOW PIXELS ARE STORED

When an image is read by *cv2*, not much can be learned without modifications to it. Applying filters, resizing, warping, all require a basic understanding how of how an image is stored when *imread* is called. To understand this, it is beneficial to think about the human eye. The eye has 3 'cones', which are essentially photo-receptor cells that respond to different amounts of light. The 3 cones respond to red light, green light and blue light respectively (RGB) [12]. In this way, every colour is a combination of these 3 channels, with varying amounts of lights for each. The computer does not have cones, so instead, it understands an image with numbers ranging from 0 - 255 for each channel. For instance in the red channel, if a computer reads 0, this means there is no red light, in contrast to 255, which is full red light. The computer can also read the green and blue channel respectively [13]. Overall, when the user types in *img = cv2.imread('cats.png')*, the *img* is a matrix of numbers with dimension Height x Width x 3 [14]. This matrix is a *numpy* array, so certain operations can easily be performed on it [15]. For example, *numpy* arrays have dimensions which are clearly visible with *img.shape*:

```
import cv2
img = cv2.imread('cat.png')
print(img.shape) # (300, 400, 3)
```

Notice that although dimensions are often represented as Width x Height, *imread* stores the Height first, then the Width. Specifically regarding indexing, *(0,0)* refers to the top left of the image, and *(height - 1, width-1)* refers to the bottom right. Indexing *img[25][34]* will return *[117, 121, 152]*. This can be interpreted as a blue level of *117*, green level of *121* and red level of *152* for the pixel located *25* units right of *(0,0)*, and *34* units down of *(0,0)*. This translates to an overall brownish colour. Notice that *imread* stores format as BGR instead of RGB. What about grayscale images? Grayscale simply means that individual pixels have the same blue, green and red light level. When blue, green and red light of the same level are combined, this basically creates a color between white and black [16]. In this sense, a grayscale image only needs to have *0* or *1* channel, depending on the way one looks at it. If it is *0*, then *img[25][34]* is *130*. If it is *1*, then *img[25][34]* is *[130]*. This is easily verifiable when the image is converted to grayscale:

```
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
print(imgGray.shape) # (300, 400)
cv2.imshow('window2', imgGray)
cv2.waitKey(0)
```

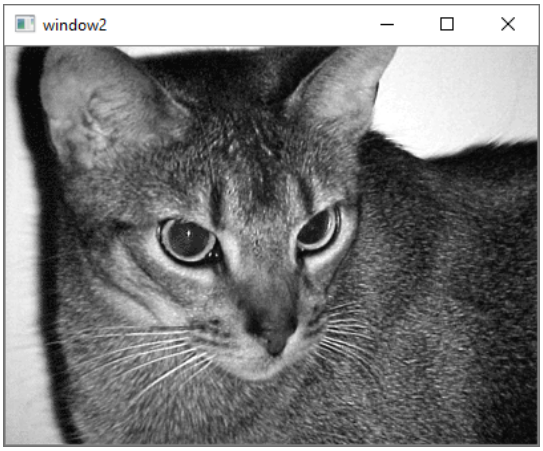

Figure 2: The same cat image, but grayscale

5. BASIC FILTER ON IMAGES

Now that there is sufficient knowledge on how images are actually stored, it is time to talk about some basic filters that are applied to nearly all projects. One such filter is the Gaussian Blur. It is widely used to reduce noise in an image [17], and it can be applied as follows:
imgBlur = cv2.GaussianBlur(imgGray, (3,3), 0)

The first parameter in the function is the image to apply the filter on. This means that the `imgGray` will be the one blurred, and the resulting image will be stored in a new variable called *imgBlur*. Note that doing this does not modify

the original *imgGray*, it simply uses it as a starting point. The next parameter controls how much blur is applied. The *(3,3)* is referred to as a kernel, and there are two rules it must satisfy: the values of the kernel must be positive, and the values must be odd. What this means is that kernels like *(1,3)*, *(0,0)* [exception to odd rule], *(3,1)* are allowed, but kernels like *(-1, 3)*, *(2,3)* are not. Note that these rules depend on the filter the user is trying to perform. But what is kernel actually doing?

Firstly, the idea of a kernel is not just unique to Gaussian Blur. A kernel, also called a convolution matrix, is just a matrix of numbers [18]. Let's assume that the kernel is the *identity*. What this means is that it looks like the following:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The kernel is *applied* to each pixel of the image. For example, let's say that the kernel has reached the pixel located at *(24,35)*, which is a white pixel. The kernel operation will consider an area around the pixel that matches its own dimension. For example, since the kernel has size *(3,3)*, it will consider the color values in the following area:

$$\begin{pmatrix} (23,34) & (24,34) & (25,34) \\ (23,35) & (24,35) & (25,35) \\ (23,36) & (24,36) & (25,36) \end{pmatrix} \rightarrow \begin{pmatrix} 123 & 23 & 40 \\ 100 & 255 & 200 \\ 85 & 23 & 49 \end{pmatrix}$$

After finding the corresponding area, the kernel operation performs a dot product between the kernel and the color values:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 123 & 23 & 40 \\ 100 & 255 & 200 \\ 85 & 23 & 49 \end{pmatrix}$$

$$= 0 \cdot 123 + 0 \cdot 23 + 0 \cdot 40 + 0 \cdot 100 + 1 \cdot 255 + 0 \cdot 200 + 0 \cdot 85 + 0 \cdot 23 + 0 \cdot 49 = 255$$

The value calculated becomes the new value of the pixel [19]. In the case of the identity kernel, there is no change to the value, but different kernels will do different things. The result is an image that has been transformed by a filter.

For example, a regular blur kernel looks something like:

$$\begin{pmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{pmatrix}$$

The process of applying a kernel on an image is called convolution, and it can be said that the image is convolved by the kernel. When applied to an image, it can drastically change the look, as is shown below. The process works similar for color images.

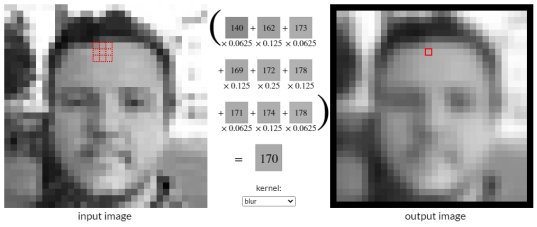

Figure 3: A blur filter applied to an image

The larger the filter, the more affected a pixel will be by its surrounding, and hence the effect is magnified. There are well known kernels that are suitable for many tasks, including blurring, edge detection, sharpening, and more [20]. These kernels are integrated in many of the *cv2* functions, some of which will be discussed later. The Gaussian kernel values are based off the Gaussian distribution, which is like the normal distribution [21]. That being said, it has an associated standard deviation value, which is what the final parameter is for. Most people tend to leave it as *0*, and increase the size of the filter if more or less blur is required. Another filter, called thresholding, also uses kernel, and will be discussed in section 7.

Filters require the knowledge of a kernel, but if the user wants to perform modifications like changing brightness and contrast, adding a tint or color detection, then it is important to understand HSV. HSV is just another representation of the RGB model [22]. HSV is an acroynm for Hue, Saturation and Value. The HSV model is essentially a cylinder that showcases the colors of the rainbow, ROYGBIV [23].

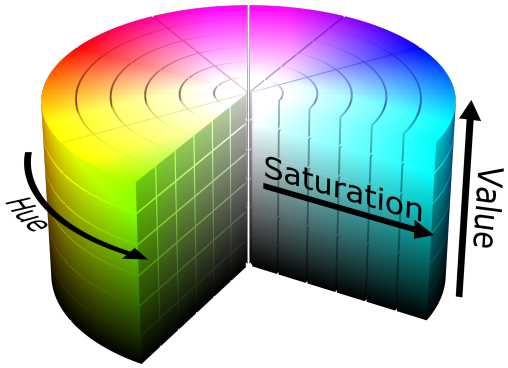

Figure 4: The HSV model, visualized

The hue is the dominant color, and ranges from *0* to *360* degrees. Red is *0* degrees, orange is around *50*, green is around *100*, etc. In *cv2*, the hue channel is expressed from *0* to *180* degrees, so whatever the value in the cylinder is, it must be divided in half. The saturation can be thought of as the amount of gray that is absent in a color. For instance, *0* saturation means no gray is absent, thus giving a grayscale image. The max value, *255*, means that all gray is absent, so the color is the purest form of itself. Saturation is often divided by *255* and represented on a *0 - 1* scale. Thus, saturation is often expressed as a percentage, from *0 - 100*, but in reality the numeric values range from *0* to *255*. Finally there is value, which can be thought of as the

amount of white or black in the image. Value *0* refers to a black image, and value *255* means a white image. Value can be thought of representing how light or dark and image is. Value is also often expressed as a percentage [24].

Now why is this important? It is important because changing any one of these channels, or a combination of them can result in many different effects. For instance, increasing the value channel can brighten or darken an image. Tinting the image is just changing the hue channel, etc [25].

HSV is also heavily used when performing color detection, and this is the focus of the next topic.

6. COLOR DETECTION

Detecting color is an essential prerequisite for detecting objects in an image. The human brain takes in color as input to distinguish objects in an environment. At the beach, the water can be classified as a blue color, and the sand as beige. The brain can understands that these must be two different objects [26]. But why is this important? Color detection has many applications, including isolating a feature in an image, tracking the motion based on color and countless others. Detecting a stop sign is as simple as detecting the red color and an octagon shape. In *cv2*, detecting color is a very easy task: all the computer needs is a range of HSV values. Why HSV? Well HSV works really well to represent color as opposed to RGB. In RGB, everything about a color, such as the main shade, grayness, lightness, etc, are all represented by a single value ranging from *0 - 255*. It is much better to separate these, and that is what HSV is able to do [27].

Consider the following image, which will be used to detect the color green:

Figure 5: A forest with a lake

The first step is to convert the image to HSV format using *hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)*

After this, the computer needs a range of HSV values in which the color lies. This can be easily found out using a software like paint. First select the green that appears the lightest and grab the RGB of it. This location is shown as the red circle above. This grass pixel has an RGB value of *(204, 203, 113)*. This can then be inserted into a simple RGB to HSV converter online [28]. This yields that *(204, 203, 113)* corresponds to a hue of *59*, a saturation of *45%* and a value of *80%*. One important thing to remember is that OpenCV stores hue value from *0 - 180*, so the hue of *59* should actually be entered as it's half *(29)* in the program [29].

The final step is to figure out the max hue value of this specified color. For example, a hue chart can prove helpful:

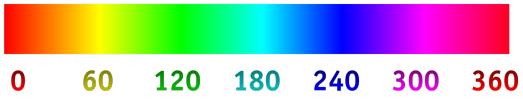

Figure 6: Hue range for ROYGBIV

There are two things to notice. The first is that green seems to have a max hue value of *150*, and second, this chart goes to *360*. Therefore, the hue channel in the upper range will be *75*. The saturation and value are kept at max value *255* in the upper channel. The two arrays are stored in variables, and then the *cv2.inRange* function is applied:

```
import cv2

img = cv2.imread('forest.png')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
lower = (29, 45, 80)
upper = (75, 255, 255)
mask = cv2.inRange(hsv, lower, upper)
cv2.imshow('mask', mask)
cv2.waitKey(0)
```

The *inRange* function will scan the input image (*hsv*), and determine whether or not each pixel lies within the low and high values [30]. If yes, the value *255* (white) is stored, and *0* (black) otherwise. This in turn creates a binary mask:

Figure 7: The white regions represent the green areas

It is a good mask, but there are quite a few dark green areas not covered. This is easily fixable by decreasing the value in the lower limit. Why value? Remember that value refers to amount of white or black in an image. Decreasing the value minimum will allow more darker areas to be visible. The value change from *80* to *40* is much nicer:

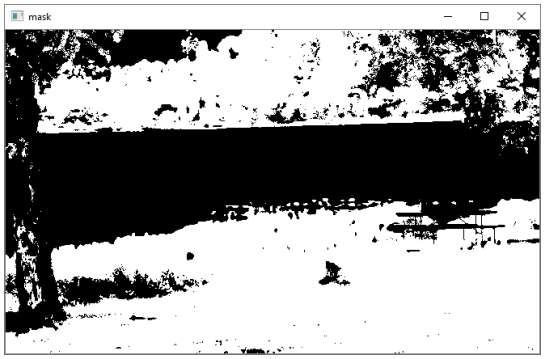

Figure 8: The value lower bound had been decreased, allowing more darker areas to show

All that remains is to use this mask on the original image. The method *cv2.bitwise_and* is of use here [31]. Simply *and* the original image with the mask:

```
import cv2

img = cv2.imread('forest.png')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
lower = (29, 45, 40)
upper = (75, 255, 255)
mask = cv2.inRange(hsv, lower, upper)
result = cv2.bitwise_and(img, img, mask=mask)
cv2.imshow('result', result)
cv2.waitKey(0)
```

Figure 9: The resulting image after mask has been applied

This technique effectively allows the user to isolate objects in the foreground and background. After isolated, all efforts can be focused towards it without other distractions.

7. EDGE DETECTION

Although isolating objects based on color can be helpful, there are a lot of problems when it becomes to the simplest of items. If the user wanted to detect a book in a frame, well, not all books are the same. Some are larger, some are smaller, but mainly, the color is different. What is helpful here is to consider the edges, not the color. Books have 4 corners, and finding the edges that intersect these corners is a very easy way to isolate the book. From this, the user can make a contour, which is discussed in the next section, and find out more information about the object, like area, aspect ratio, to determine if this is the target. Edge detection in *cv2* works using an algorithm created by John Canny, appropriately named the Canny Edge Detection. Fortunately, the algorithm is already implemented in *cv2* in a method called *cv2.Canny(img, threshold1, threshold2)*. The first parameter is the source image, and is usually best in grayscale. The second and third parameters are thresholds, which are a mystery right now, but should be clear in the following description of how the algorithm works.

The Canny algorithm works by detecting discontinuities in the brightness of an image, and this is why grayscale works best [32]. The Canny algorithm is an extension of something called the Sobel Operator [33]. There are 4 steps in the Canny algorithm, in which the first two are exactly the same as the Sobel Operator. In this sense, Canny is simply an extension of the Sobel Operator.

The first step of the Canny (and Sobel) is to perform noise reduction on the image. Random spots and places of noise can influence the algorithm to falsely predict that the location is an edge, when it is not. To counter this, in the first step the algorithm applies a small Gaussian Blur to smooth out the image [34]. This will remove excess noise.

The second step is to apply a filter to the grayscale image. Remember that filters in *cv2* are just kernels that are convolved through the image. There are two important kernels in the Canny (and Sobel) which help with edge detection. The kernels are as follows:

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, G_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

The G_x and G_y kernels represent the derivatives of the grayscale image in the x and y directions [35]. Although derivatives are usually thought of regarding curves, it is important to remember what a derivative is doing. A derivative just measures the change between two values, and in the case for Canny, it is just the differences between two adjacent pixels. In a grayscale image, a large change in the value signifies an edge. As a result, the derivative between the two points will be large. When it is higher, the new image will show a white line, indicating an edge boundary. If the derivative is near 0 or a small number, then this is not an edge boundary, and so a gray or black line is shown. The following dictates the original image, which is a rose. After that, the rose is convolved using both kernels, which in turn showcase important features of the image:

Figure 10: Grayscale image of a rose

Figure 11: G_x convolution showcasing the edges in the horizontal direction

Figure 12: G_y convolution showcasing the edges in the vertical direction

The two kernels (also called gradients since both are derivatives), are then added together to produce the following:

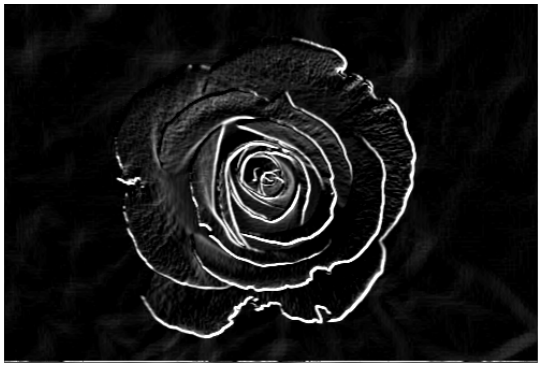

Figure 13: Added image from both the horizontal and vertical kernels

This completes the steps of the Sobel operator, but Canny still has two more steps.

The third step in Canny is to perform something called non-maximum suppression. The wording maybe a little confusing, but the idea is simple. The edges detected in Sobel are still quick thick. Canny does not care about the size of an edge, but rather, where the edge is. A way to fix this is to go through each pixel, and check a small area around it. If this pixel is the maximum (whitest) pixel in that area, the pixel is kept as is. If it is not, this pixel is set to 0 [36]. What this effectively is doing is cleaning out any small edges or smudges that should not be there. What results is the following:

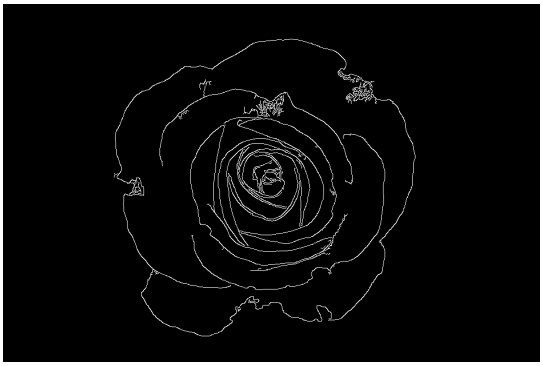

Figure 14: Thin edges remain after a lot of the non-edges are removed

The last step is to really narrow down on what counts as an edge and what does not. In the previous step, the edges were thinned, but there might still be places that should not be considered edges. This is where thresholding comes into play, and where the user has to supply two values, representing the lower and higher thresholds. For the purpose of illustration, let's say the user provides the lower threshold as *115*, and the upper at *240*. This is the result:

```
import cv2

img = cv2.imread('rose.jpeg')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
result = cv2.Canny(img, 115, 240)
cv2.imshow('result', result)
cv2.waitKey(0)
```

Figure 15: Threshold values applied

Notice how in the new image, there are none of those weird edges in the top right corner or a little above the middle. All the smaller, insignificant edges have been erased. However, what exactly do the numbers *115* and *240* mean? The logic is that any pixel that is above *240* in value automatically is considered an edge. Any pixel that is below *115* is not an edge. Any pixel in between is considered an edge if it is connected to an area that goes above the *240* value [37]. In other words, it is a continuous edge.

These are the 4 steps to the Canny algorithm, and for the user, only two values should be known, the thresholds.

However, determining threshold values by hand is quite hard, so *cv2* offers trackbars which can be adjusted by the user to provide an insight:

```
import cv2

name = "window"
cv2.namedWindow(name)
cv2.resizeWindow(name, 500,100)

def empty(x): return

cv2.createTrackbar("th1", name, 0, 255, empty)
cv2.createTrackbar("th2", name, 0, 255, empty)
```

First the user must create a window to host the trackbars in. The above will create two trackbars named *th1* and *th2* with starting value *0* and max value *255*. The *empty* is a function that is called on every slide of the trackbar, but most of the time, the user does not need to do anything, so just return [38].

The next task is physically reading the data from the trackbar and updating the source image:

```
img = cv2.imread('rose.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

while True:
    th1_pos = cv2.getTrackbarPos("th1", name)
    th2_pos = cv2.getTrackbarPos("th2", name)

    canny = cv2.Canny(img, th1_pos, th2_pos)
    cv2.imshow(name, canny)

    key = cv2.waitKey(1) & 0xFF
    if key == ord("q"):
        break

cv2.destroyAllWindows()
```

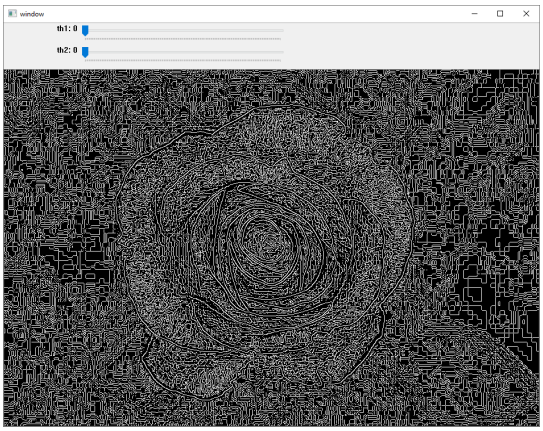

Figure 16: Both thresholds are 0 when the program first runs

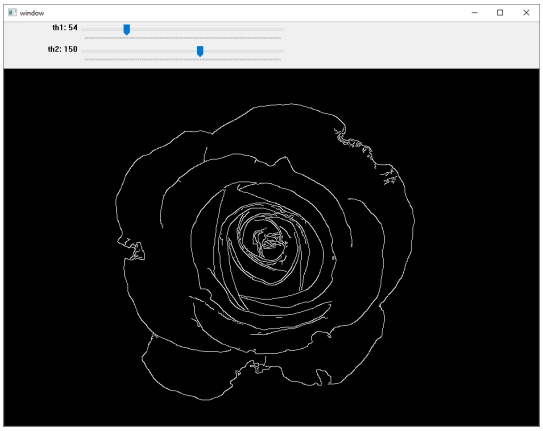

Figure 17: The user can slide the thresholds to find a good match

8. CONTOUR DETECTION

Edge detection is important because it is able to separate the objects in the image. However, edge detection alone using Canny does not allow the user to refer to a specific item. For example, consider the shapes below:

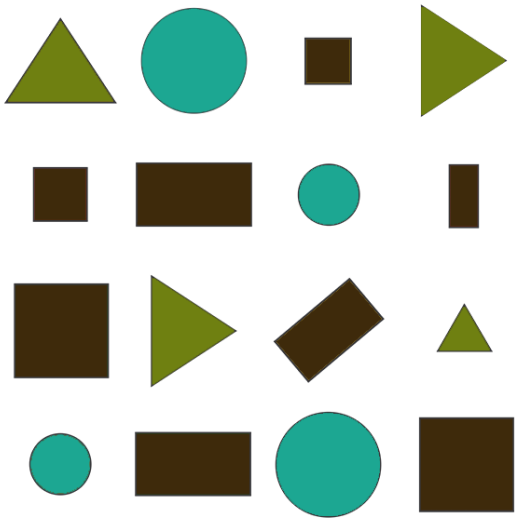

Figure 18: A series of shapes which will be used to demonstrate contour detection

The user can use the following piece of code to determine the edges of the shape:

```
import cv2

img = cv2.imread('shapes.png')
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
imgBlur = cv2.GaussianBlur(imgGray, (7, 7), 0)
result = cv2.Canny(imgBlur, 50, 200)
cv2.imshow('result', result)
cv2.waitKey(0)
```

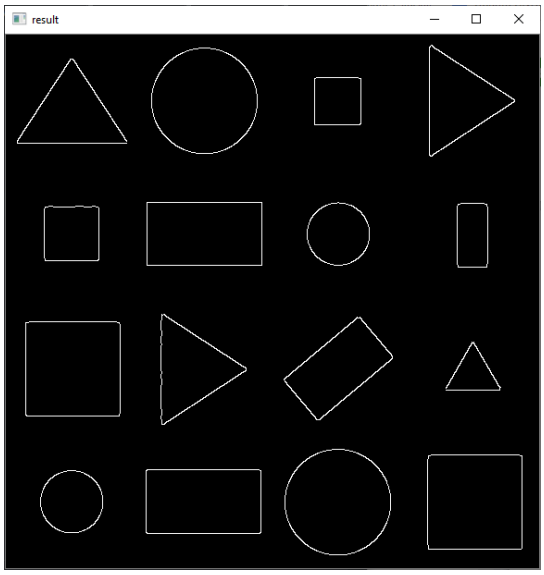

Figure 19: A series of filters applied to detect the edges

This is good, but how can the user refer to the individual shapes? How can the user know which ones are triangles? This is where contours come in. A contour can simply be thought of as an outline of something, which represents or bounds the shape within. Contours should always been applied on a canny image because it is very easy if there are only edges [39]. There is a method called *cv2.findContours* which can do this [40]. Here is the method in action:

```
# result = cv2.Canny(imgBlur, 50, 200) and above
contours, _ = cv2.findContours(result,
                               cv2.RETR_EXTERNAL,
                               cv2.CHAIN_APPROX_NONE)
for cnt in contours:
    cv2.drawContours(img, cnt, -1, (0, 0, 255), 3)

cv2.imshow('result', img)
cv2.waitKey(0)
```

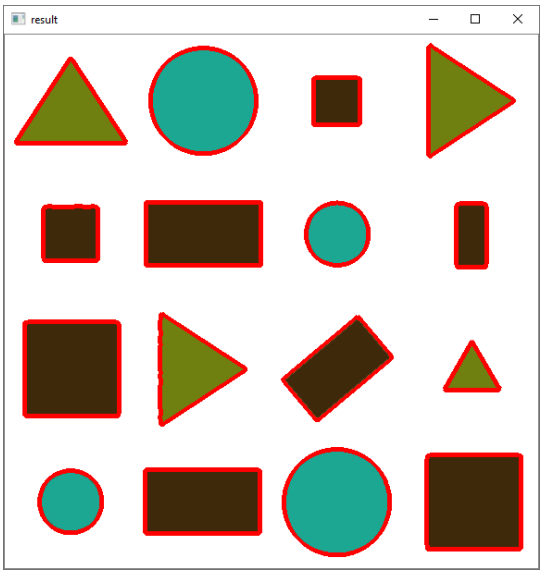

Figure 20: All contours printed on the main image

Notice how *cv2* is able to recognize the shapes and draws a border around each one. The *findContours* method should always take in a canny image as the first input. The second argument is the retrieval type of the contours. Here, *cv2.RETR_EXTERNAL* is used when only the outer contours are wanted, which is most of the time. For example, if the second circle had an inner circle, then *cv2.RETR_EXTERNAL* would not detect it, because it is overshadowed by the larger circle. The last parameter tells the method what to store in the return value. Contours are essentially just points. The option *cv2.CHAIN_APPROX_NONE* simply says to store each point and bundle them together in a list. There is another option called *cv2.CHAIN_APPROX_SIMPLE* which will to store the location of the corners, but it is not always accurate [41].

The next section simply loops through each contour, and draws it on the main image with red, and a thickness of 3. The *-1* says to draw all the points in *cnt*.

Contours can be used in a number of geometric applications, but are particularly good for calculating perimeter, area, number of corners and a bounding box [42]. Area is often used as a filter to decide whether or not this shape is

what the user is looking for, or just a smaller shape that is not important.

```
for cnt in contours:
    per = cv2.arcLength(cnt, True)
    area = cv2.contourArea(cnt)
    approx = cv2.approxPolyDP(cnt, 0.02*per, True)
    numCorners = len(approx)
```

The *arcLength* simply takes in the contour, and a boolean dictating whether or not it is a closed shape. Almost always, this will be True. The method *contourArea* can be used as a filter. For instance, *if area > 5000: [do stuff]*. The third method *approxPolyDP* approximates curves to represent the contour. The number of curves it approximates represents the number of corners. For example, a triangle can be approximated using 3 lines, a square, 4 lines. For something like a circle, checking whether corners are larger than 4 is often enough [43]. Once the user is certain that the shape or object is found, a bounding box should be created around it. A bounding box is better than a set of contours because a bounding box is just defined by 4 points. Contours are defined by many, and it is much easier to track with a bounding box. Here is how that will work:

```
for cnt in contours:
    per = cv2.arcLength(cnt, True)
    approx = cv2.approxPolyDP(cnt, 0.02*per, True)
    x, y, w, h = cv2.boundingRect(approx)
    cv2.rectangle(img, (x, y), (x+w, y+h),
                            (255, 0, 0), 3)
```

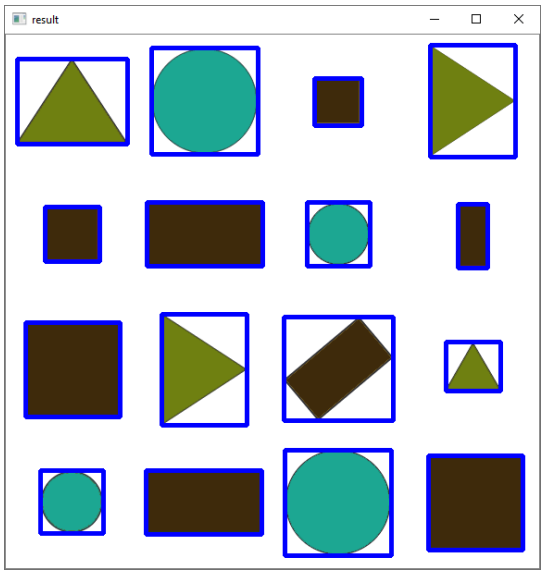

Figure 21: Bounding boxes around each shape

The *boundingRect* quite literally bounds the object inside. The *x, y* refer to the coordinate of the top left coordinate of the rectangle, and the *w, h* represents how far in the x and y direction the rectangle extends to. Therefore, the bottom left is *x+w, y+h*.

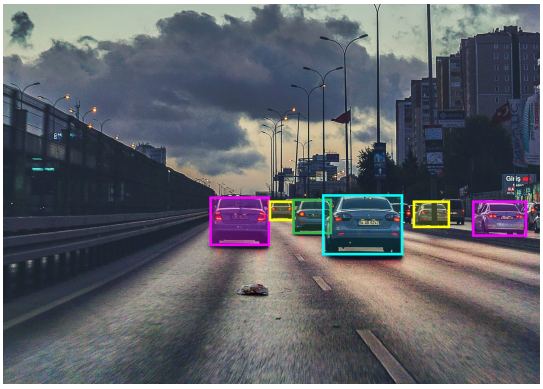

Figure 22: Bounding boxes can be used to detect and track a range of objects

With a bounding box, the user can easily rotate the object, warp it, find its center, change the color of the bounding box to match a certain label, and perform so many more actions [44]. The next section will dive into how to warp an image that isn't to the users liking.

9. WARPING

The user until now has learned how to detect colors, perform edge detection as well select images using contours. If the user wants to only focus on a certain image, it is important to make sure that the image is 'straight'. For example, consider the following image:

Figure 23: A business card that is rotated

The edges and the contours of the business card can be find in a manner similar to the previous section, using canny and contours:

```
import cv2
import numpy as np
import operator

img = cv2.imread('card.jpg')
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
imgBlur = cv2.GaussianBlur(imgGray, (3,3), 0)
canny = cv2.Canny(imgBlur, 60, 160)

contours, _ = cv2.findContours(canny,
                               cv2.RETR_EXTERNAL,
                               cv2.CHAIN_APPROX_NONE)

contours = sorted(contours, key=cv2.contourArea,
                                      reverse=True)

if len(contours) > 0:
    largestContour = contours[0]
    cv2.drawContours(img, largest_contour, -1,
                                (255,0,0), 2)

cv2.imshow('result', img)
cv2.waitKey(0)
```

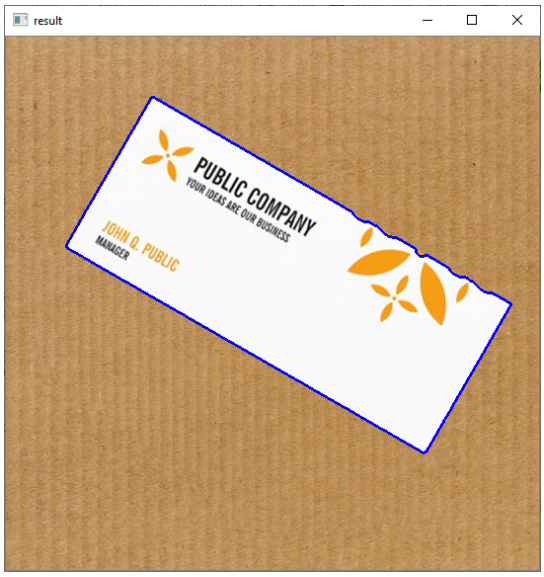

Figure 24: The contours of the business card highlighted

Now that the user has isolated the business card through the use of contours, perhaps the next step is to read the name of the company. To the human eye, it does not matter what orientation the text is in. The human brain is able to understand text that is at multiple angles, ie, it does not have to be straight. However to a computer, often the image needs to be straightened so it is easier to perform analysis. This is where the idea of warping comes into play. This can be done using the methods *cv2.getPerspectiveTransform* and *cv2.warpPerspective*.

The method *cv2.getPerspectiveTransform* takes in two lists which contain the same amount of points each. The purpose of this method is to map a point from the original image to another image of a different size. Basically, it will map it from the original image to the new warped image. The points passed in are usually corners of the image, so in the case of the business card, the first list could be *[top left coord, top right coord, bot right coord, bot left coord]*. This can be easily found by the contours, because contours are just points. For example, the top left coordinate is the contour which has the smallest *x+y* value. The top right coordinate is the contour which has the largest *x-y* value, etc. This can be done by using a little bit of *numpy* [45]:

```
def find_extreme_corners(largest_contour,
                         limit_fn, compare_fn):
    section, _ = limit_fn(enumerate(
                [compare_fn(pt[0][0], pt[0][1])
                    for pt in largest_contour]),
                      key=operator.itemgetter(1))

    return largest_contour[section][0][0],
            largest_contour[section][0][1]
```

Above is a helper function will take in the largest contour, a limit function which is either *min* or *max*, as well as a compare function, which is either *np.add* or *np.subtract*. Here is how it can be integrated with the main code:

```
if len(contours) > 0:
    largest_contour = contours[0]
    top_left = find_extreme_corners(
        largest_contour, min, np.add)
    top_right = find_extreme_corners(
        largest_contour, max, np.subtract)
    bot_right = find_extreme_corners(
        largest_contour, max, np.add)
    bot_left = find_extreme_corners(
        largest_contour, min, np.subtract)

    cv2.circle(img, tuple(bot_right), 5,
                            (0, 0, 255), 3)
```

The idea is very simple. The top left corner will have the smallest *x+y* value, and that is why the functions *min* and *np.add* are being passed. Similarly, the top right corner will have the largest *x-y* value, and that is why the functions *max* and *np.subtract* are being passed [46]. These functions are not available in standard python, and this is why packages like *numpy* are used.

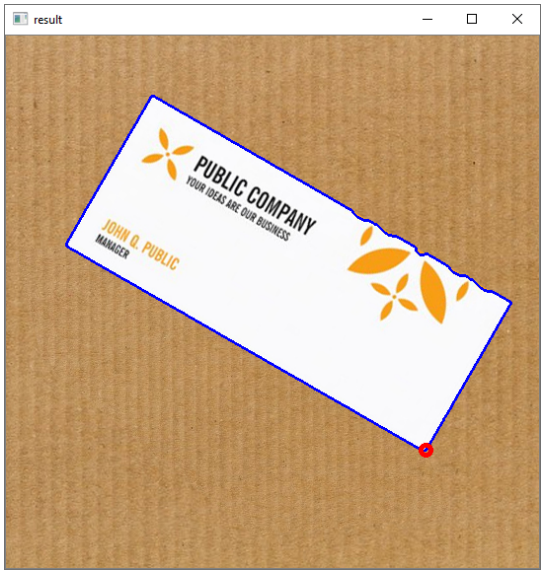

Figure 25: The bottom right corner detected from the contours

All that remains now is the second list. In the second list, the user has to supply 4 points in this case which would correspond to the *[top left coord, top right coord, bot right coord, bot left coord]* in the warped image. Note that the order must match. If the corners in the previous list were stored in another manner, then the corners in the second list have to be stored in the same manner. Commonly the warped image size should be the same as the subject that is being warped. This means that the width and height of the card should be known [47]. Based on the first list, the user is able to find out the width and the height of the card in the following way:

```
w = top_right[0] - top_left[0]
h = top_right[1] - top_left[1]
```

Now, the warped image size is easily determinable and the warped image can be shown:

```
pts1 = np.float32(
    [top_left, top_right, bot_right, bot_left])
pts2 = np.float32([[0,0],[w,0],[w,h],[0,h]])

matrix = cv2.getPerspectiveTransform(pts1,pts2)
warped = cv2.warpPerspective(img, matrix, (w,h))

cv2.imshow('result', warped)
```

The *getPerspectiveTransform* method is applied given the two sets of points. This will create a matrix which is used in the *warpperspective* method to perform the transformation. The math behind this operation is out of the scope for a simple introductory paper. The final image will look like the following:

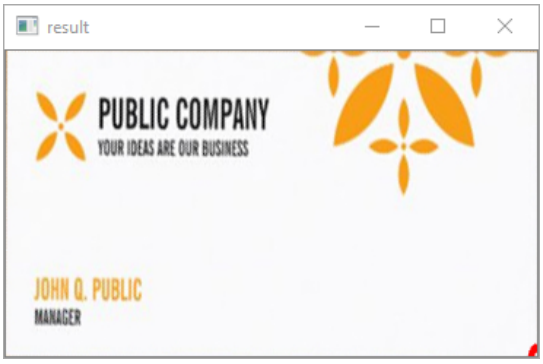

Figure 26: The business card, warped

Notice that now the image is straight, and the user can easily perform tasks like text analysis, which on a tilted image would not be very trivial.

10. SHAPES AND TEXT

There are many instances when drawing a shape on the frame would be appropriate. The most common use case is to allow the user to know that *cv2* has found the region of interest. This region of interest is almost always represented by a bounding rectangle, which was discussed in the last section. Another shape which is as popular as the rectangle is a circle. The method associated with the circle is *cv2.circle*, and can be used as follows: The first parameter as always is the image to draw the shape on. The second parameter is *(x, y)* and represents the center of the circle. This means that if the user puts *(x, y)* as *(0, 0)*, the bottom right quarter of the circle will still be partly visible. The next parameter is the radius, followed by the color. Notice how the first circle is not filled in, but this can easily be done with the *cv2.FILLED* option at the end. Circles are often used to showcase important points, like the corners of the object as was seen in the last section.

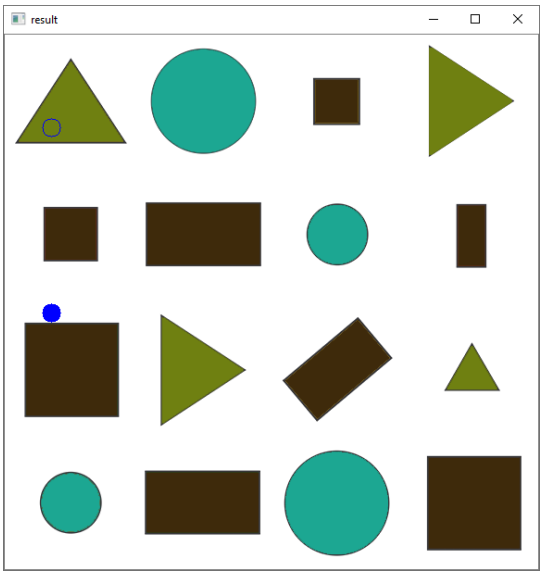

Figure 27: Two circles drawn on the shapes image

There are other shapes including lines, ellipses, polygons and more. Once a frame is loaded and processed by *cv2*, the final step is to tell the user the result, whatever it might be. For example, consider the following example:

```
import cv2

img = cv2.imread('shapes.png')
x_pos, y_pos = img.shape[1]//2, img.shape[0]//2
font = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText(img, "test", (x_pos, y_pos),
            font, 1, (0, 0, 255), 2)
cv2.imshow('result', img)
cv2.waitKey(0)
```

The first parameter is as always the image the effect should be applied on. After that follows the text itself, and the position. Notice that although the center of the image is passed, the text seems to be a little to the right of it in the following image. This is because *putText* uses this coordinate as the bottom-left starting point. Notice how the bottom left of *test* touches the center. What follows next is the font. The user can import personal fonts via a *ttf* file, but for now, it is set to a *cv2* default [48]. What remains after is the font scale, color, and font thickness. The font scale is important because it dictates how big the font should be. The value *1* is the default, and decimals are allowed here to either increase or decrease the size of the font. The font thickness is almost always set to a number greater than *1* because *1* is really thin in the *cv2.FONT_HERSEY_SIMPLEX*.

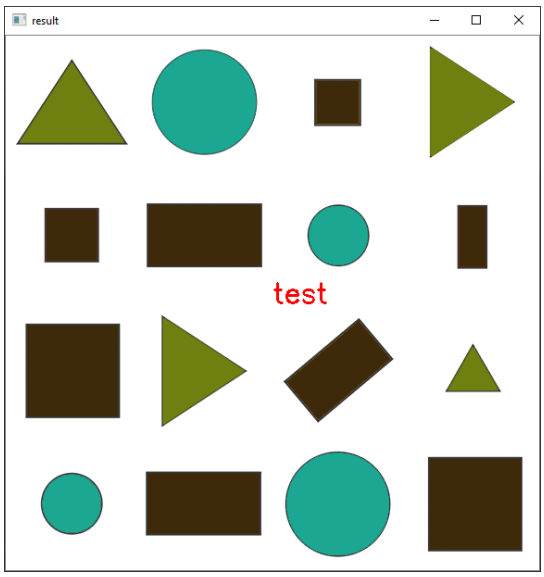

Figure 28: Text displayed on an image

Thankfully, there is an elegant way to truely center the text, using a method called *getTextSize*:

```
textsize = cv2.getTextSize(text, font, 1, 2)[0]
textX = (img.shape[1] - textsize[0]) // 2
textY = (img.shape[0] + textsize[1]) // 2
cv2.putText(img, "test", (textX, textY),
            font, (0, 0, 255), 1, 2)
```

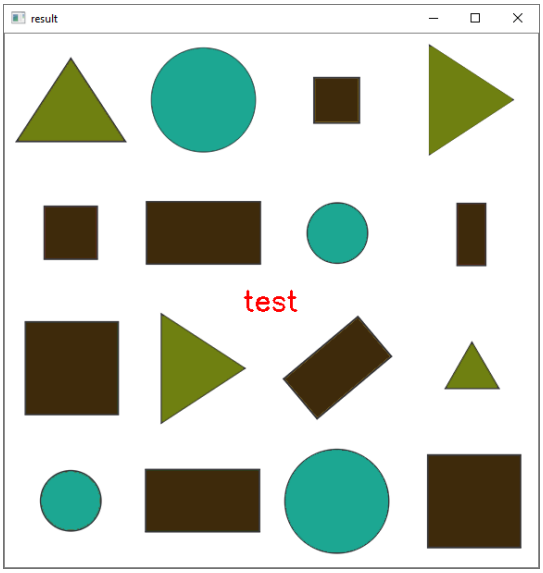

Figure 29: Text displayed on an image, centered

11. CONCLUSION

With these skills of *cv2*, suddenly many computer vision problems become possible. Learning how images are stored helps to understand how to manipulate them with filters. Understanding color modes like RGB and HSV allow for tasks like color detection and canny edge detection. Contour detection along with drawing shapes and text allow the user to isolate the objects and return feedback to the user. Often *cv2* is combined with tools including Keras and Tensorflow to perform object recognition tasks. For example, recognizing the suit and rank of a card, the ethnicity of a person, a sudoku solver, all use principles taught in this paper. Often the data is processed using *cv2*, sent to a neural network in Tensorflow, and displayed to the user in the form of text. The next step of the user should be to integrate *cv2* with said technologies above. The possibilities are truly endless, and this paper has illustrated only 20% of what *cv2* is capable of doing. OpenCV has become a staple tool in the world of Computer Vision, and due to open source, anyone can help to make it better.

12. REFERENCES

[1] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[2] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Realtime computer vision with opencv. *Queue*, 10(4):40–56, April 2012.

[3] Opencv - license agreement.

[4] Opencv. opencv/opencv on github.

[5] Opencv - about.

[6] Nick Bunkley. Joseph juran, 103, pioneer in quality control, dies, Mar 2008.

[7] Naveenkumar Mahamkali and A Vadivel. Opencv for computer vision applications. 03 2015.

[8] Opencv modules.

[9] Thiago Santos. Scipy and opencv as an interactive computing environment for computer vision. *Revista de Informática Teórica e Aplicada*, 22:154–189, 05 2015.

[10] Getting started with images.

[11] An introduction to the principles of video 101.

[12] Principles of neural science. *McGraw-Hill*, pages 507–513, 2000.

[13] Noor Ibraheem, Mokhtar Hasan, Rafiqul Zaman Khan, and Pramod Mishra. Understanding color models: A review. *ARPN Journal of Science and Technology*, 2, 01 2012.

[14] nkmk. Reading and saving image files with python, opencv (imread, imwrite), 05 2019.

[15] Stéfan van der Walt, S. Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13:22 – 30, 05 2011.

[16] Saravanan Chandran. Color image to grayscale image conversion. pages 196 – 199, 04 2010.

[17] Estevao Gedraite and M. Hadad. Investigation on the effect of a gaussian blur in image filtering and segmentation. pages 393–396, 01 2011.

[18] Victor Powell. Image kernels explained visually.

[19] Zhun Sun, Mete Ozay, and Takayuki Okatani. Design of kernels in convolutional neural networks for image classification. 11 2015.

[20] Prakhar Ganesh. Types of convolution kernels: Simplified, Oct 2019.

[21] Jogikalmat Krithikadatta. Normal distribution. *Journal of conservative dentistry : JCD*, 17:96–97, 02 2014.

[22] P. Ganesan, V. Rajini, B. S. Sathish, and K. B. Shaik. Hsv color space based segmentation of region of interest in satellite images. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 101–105, 2014.

[23] Bhubneshwar Sharma and Rupali Nayyer. Use and analysis of color models in image processing. *International Journal of Advances in Scientific Research*, 1:329, 10 2015.

[24] Real Python. Image segmentation using color spaces in opencv python, Aug 2020.

[25] A. Vadivel, Shamik Sural, and Arun Majumdar. Human color perception in the hsv space and its application in histogram generation for image retrieval. *Proceedings of SPIE - The International Society for Optical Engineering*, 01 2005.

[26] James Dicarlo, Davide Zoccolan, and Nicole Rust. How does the brain solve visual object recognition? *Neuron*, 73:415–34, 02 2012.

[27] Dibya Bora, Anil Gupta, and Fayaz Khan. Comparing the performance of l*a*b* and hsv color spaces with respect to color image segmentation. 06 2015.

[28] Rgb to hsv conversion | color conversion.

[29] Raghav Puri and Archit Gupta. Contour, shape & color detection using opencv-python. 01 2018.

[30] Thresholding operations using inrange.

[31] nkmk. Alpha blending and masking of images with python, opencv, numpy.

[32] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[33] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[34] Sofiane Sahir. Canny edge detection step by step in python- computer vision, Jan 2019.

[35] Sobel edge detector.

[36] Sambasivarao. K. Non-maximum suppression (nms), Oct 2019.

[37] R. Reddy, Chiluka Nagaraju, and I Raja Sekhar Reddy. Canny scale edge detection. *IJETT*, 01 2016.

[38] Manivannan Murugavel. Set trackbar on image using opencv python, Jan 2018.

[39] Akula Hemanth Kumar. Contours in images, Mar 2020.

[40] Zhahir, Meng Xipeng, Adrian Rosebrock, Erkki Nyfors, Sasa, Ashwin, Praveen, Shelly, Patrick, Sai Pattnaik, and et al. Python and opencv: Finding opencv contours with cv2.findcontours, Apr 2020.

[41] Raqueeb Shaikh. Opencv (findcontours) detailed guide, Jun 2020.

[42] Christoph Rasche. Rapid contour detection for image classification. *IET Image Processing*, 12, 11 2017.

[43] John Fisher, John Lowther, and Ching-Kuang Shene. Curve and surface interpolation and approximation: knowledge unit and software tool. volume 36, pages 146–150, 01 2004.

[44] Jie-Zhi Cheng, Feng-Ju Chang, Kuang-Jui Hsu, and Yen-Yu Lin. Knowledge leverage from contours to bounding boxes: A concise approach to annotation. pages 730–744, 11 2012.

[45] Boguslaw Cyganek and Jan Siebert. *Image Warping Procedures*, pages 409–428. 01 2009.

[46] Anshul Sachdev. How to project an image in perspective view of a background image - opencv, python, May 2019.

[47] Daryl Tan. Image geometric transformation in numpy and opencv, Mar 2020.

[48] Hmrishav Bandyopadhyay. Drawing in opencv, Oct 2019.