Computer Vision
Degree in Information Technology
2º Semester 2021/2022

Worksheet 5

Goals:

- Gradient and Edge detection
- Counters

Exercises

Part I – Gradient and Edge Detection

1- **Sobel**. Write and run the following code in a python file:

```
import numpy as np
import cv2

image = cv2.imread("ponte.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Image", image)

sobelX = cv2.Sobel(image, cv2.CV_64F, 1, 0)
sobelY = cv2.Sobel(image, cv2.CV_64F, 0, 1)
sobelX = np.uint8(np.absolute(sobelX))
sobelY = np.uint8(np.absolute(sobelY))
sobelCombined = cv2.bitwise_or(sobelX, sobelY)

cv2.imshow("Sobel X", sobelX)
cv2.imshow("Sobel Y", sobelY)
cv2.imshow("Sobel Combined", sobelCombined)
cv2.waitKey(0)
```

- Sobel method: cv2.Sobel function.
- The first argument to the Sobel operator is the image we want to compute the gradient representation.
- The second argument is our data type for the output image. We will use a 64-bit float (cv2.CV64F) because of the transition of black-to-white and white-to-black in the image.
- Transitioning from black-to-white is considered a positive slope, whereas a transition from white-to-black is a negative slope. If you don't use a floating-point data type when computing the gradient magnitude image, you will miss edges, specifically the white-to-black transitions.
- Thus, use a floating-point data type, then take the absolute value of the gradient image and convert it back to an 8-bit unsigned integer.

- The last two arguments are the order of the derivatives in the x and y direction, respectively.
- Specify a value of 1 and 0 to find vertical edge-like regions and 0 and 1 to find horizontal edge-like regions.
- Then we find all edges by taking the absolute value of the floating-point image.
- Then convert the result to an 8-bit unsigned integer.
- To combine the gradient images in both the x and y direction, we can apply a bitwise OR.
- The gradient images are shown at the end of the code.

2- **Prewitt**. Write and run the following code in a python file:

```
import cv2
image = cv2.imread("ponte.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Image", image)

kernelx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
kernely = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
prewittx = cv2.filter2D(image, -1, kernelx)
prewitty = cv2.filter2D(image, -1, kernely)
prewittCombined = cv2.bitwise_or(prewittx, prewitty)

cv2.imshow("Prewitt Y", prewittx)
cv2.imshow("Prewitt X", prewitty)
cv2.imshow("Prewitt Combined", prewittCombined)
cv2.waitKey(0)
```

- We perform the prewitt method with cv2.filter2D function. It convolves a kernel with an image.
- The first argument is the mask, the second argument is the output depth. -1 implies the same depth as the input image. The last argument is the kernel.

3- **Laplacian**. Write and run the following code in a python file:

```
import numpy as np
import cv2

image = cv2.imread("ponte.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Image", image)

lap = cv2.Laplacian(image, cv2.CV_64F)
lap = np.uint8(np.absolute(lap))
cv2.imshow("Laplacian", lap)
cv2.waitKey(0)
```

- Laplacian method: cv2.Laplacian function.

- The first argument is our grayscale image.
- We use a floating-point data type in the second argument: cv2.CV64F.
- Take the absolute value of the gradient image.
- Convert it back to an 8-bit unsigned integer.

4- **Canny**. Write and run the following code in a python file:

```
import numpy as np
import cv2

image = cv2.imread("coins.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Image", image)

image = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Blurred", image)

canny = cv2.Canny(image, 30, 150)
cv2.imshow("Canny", canny)
cv2.waitKey(0)
```

- Import the packages, load the image, convert it to grayscale, and blur it using the Gaussian blurring method.
- Applying the Canny edge detector: cv2.Canny function.
- The first argument is the grayscale image.
- Then provide the two values: threshold 1 and threshold 2.
- Any gradient value larger than threshold 2 is an edge.
- Any value below threshold 1 is considered not to be an edge.
- Values in between threshold 1 and threshold 2 are either classified as edges or non-edges based on how their intensities are “connected”. In this case, any gradient values below 30 are considered non-edges whereas any values above 150 are considered edges.

Part II– Counters

1- Write and run the following code in a python file

```
Import cv2
image = cv2.imread("coins.jpg")

gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (11, 11), 0)
edged = cv2.Canny(blurred, 30, 150)
cv2.imshow("Edges", edged)
cv2.waitKey(0)

(cnts, _) = cv2.findContours(edged.copy(),
cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
print("I count {} coins in this image". format(len(cnts)))
coins = image.copy()

cv2.drawContours(coins, cnts, -1, (0, 255, 0), 2)
cv2.imshow("Coins", coins)
cv2.waitKey(0)
```

- To find contours in an image, we need to first obtain a binarization of the image, using either edge detection methods or thresholding.
- In this case, we use the canny edge detector. And like in the previous subsection we first blurred the image.
- Any gradient values below 30 are considered non-edges whereas any values above 150 are considered sure edges.
- We find the contours of the outlines using the cv2.findContours function. This method
- returns a 2-tuple of: the contours themselves, cnts, and the remaining image which will be discarded.
- The first argument is our edged image. It is important to note that this function is destructive to the image you pass in. If you intend using that image later in your code, it is best to make a copy of it, using the NumPy copy method.
- The second argument is the type of contours we want. We use cv2.RETR_EXTERNAL to retrieve only the outermost contours.
- The last argument is how we want to approximate the contour. We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments into their endpoints only. This saves both computation and memory.
- Our contours cnts is simply a Python list. We can use the len() function to count the number of contours that were returned.
- cv2.drawContours draws the actual contours on our image.:
    - The first argument is the image we want to draw on.
    - The second is our list of contours.

- The third argument is the contour index. -1 indicates that we want to draw all the contours.
- The fourth argument is the colour of the line.
- The last argument is the thickness of the line.

2- Complete the next code to count the dices in the image:

a. #step 1: Convert the image for gray scale
b. #step 2: Image Bluring
c. #step 3: Segmentation using Otsu
d. #step 4: Edge detection with Canny
e. #step 5: Image Counter
f. #stept 6: complete the calling of write function to write the appropriate messages in the corresponding images
g. #stept 7: Complete the Draw Counter in the objects which you found
h. #step 8: Verify how many objects are in that image

```
import numpy as np
import cv2
import mahotas

#Function to facilitate the image writting
def write(img, texto, cor=(255,0,0)):
    fonte = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(img, texto, (10,20), fonte, 0.5, cor, 0,cv2.LINE_AA)

imgColour = cv2.imread('dice.jpg')


#step 1: Convert the image for gray scale

….

#step 2: Image Bluring

….

#step 3: Segmentation using Otsu

….

#step 4: Edge detection with Canny

…

#step 5: Image Counter

….

write(…, "Gray Image", 0)
write(…, "Blurring", 0)
write(…, "Otsu Method", 255)
write(…, "Canny Edge detector", 255)
temp = np.vstack([ np.hstack([…, …]), np.hstack([…, …]) ])

cv2.imshow("Amount of Objects: "+str(len(…)), temp)
cv2.waitKey(0)

imgC2 = imgColour.copy()
cv2.imshow("Original Image", imgColour)


cv2.drawContours(imgC2, …, -1, (255, 0, 0), 2)
write(imgC2, str(len(…))+" object founded!")

cv2.imshow("Result", imgC2)
cv2.waitKey(0)
```

Ideas for the practical work:

Implement the several filters as functions.

Using this knowledge count the object in an image like this.