Hong Trinh_438443
Embedded Vision

Assignment 4

Recognize circles and squares with the formula for roundness

With the OpenCV, I found the method called "Canny Edge Detection". This theory is an edge detection algorithm which was developed by John F. Canny in 1986. There are many stages of this method:

- Function canny_edge_detector: Convert image to grayscale, remove noise, detect edge.

```python
from math import sqrt, atan2, pi
import numpy as np

def canny_edge_detector(input_image):
    input_pixels = input_image.load()
    width = input_image.width
    height = input_image.height

    # Transform the image to grayscale
    grayscaled = compute_grayscale(input_pixels, width, height)

    # Blur it to remove noise
    blurred = compute_blur(grayscaled, width, height)

    # Compute the gradient
    gradient, direction = compute_gradient(blurred, width, height)

    # Non-maximum suppression
    filter_out_non_maximum(gradient, direction, width, height)

    # Filter out some edges
    keep = filter_strong_edges(gradient, width, height, 20, 25)

    return keep
```

- Function compute_grayscale: will be called in canny_edge_detector. Transfer the input to grayscale image

```python
def compute_grayscale(input_pixels, width, height):
    grayscale = np.empty((width, height))
    for x in range(width):
        for y in range(height):
            pixel = input_pixels[x, y]
            grayscale[x, y] = (pixel[0] + pixel[1] + pixel[2]) / 3
    return grayscale
```

First, noise reduction uses for removing to the noise in the image with a 5x5 Gaussian filter. Function compute_blur: will be called in canny_edge_detector to blur the image and remove noise.

```python
def compute_blur(input_pixels, width, height):
    # Keep coordinate inside image
    clip = lambda x, l, u: l if x < l else u if x > u else x

    # Gaussian kernel
    kernel = np.array([
        [1 / 256,  4 / 256,  6 / 256,  4 / 256, 1 / 256],
        [4 / 256, 16 / 256, 24 / 256, 16 / 256, 4 / 256],
        [6 / 256, 24 / 256, 36 / 256, 24 / 256, 6 / 256],
        [4 / 256, 16 / 256, 24 / 256, 16 / 256, 4 / 256],
        [1 / 256,  4 / 256,  6 / 256,  4 / 256, 1 / 256]
    ])

    # Middle of the kernel
    offset = len(kernel) // 2

    # Compute the blurred image
    blurred = np.empty((width, height))
    for x in range(width):
        for y in range(height):
            acc = 0
            for a in range(len(kernel)):
                for b in range(len(kernel)):
                    xn = clip(x + a - offset, 0, width - 1)
                    yn = clip(y + b - offset, 0, height - 1)
                    acc += input_pixels[xn, yn] * kernel[a, b]
            blurred[x, y] = int(acc)
    return blurred
```

- Second, finding intensity gradient of the image. After smooth the image, the image is filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in two directions. Function compute_gradient will return edge gradient and direction for each pixel:

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

```python
def compute_gradient(input_pixels, width, height):
    gradient = np.zeros((width, height))
    direction = np.zeros((width, height))
    for x in range(width):
        for y in range(height):
            if 0 < x < width - 1 and 0 < y < height - 1:
                magx = input_pixels[x + 1, y] - input_pixels[x - 1, y]
                magy = input_pixels[x, y + 1] - input_pixels[x, y - 1]
                gradient[x, y] = sqrt(magx**2 + magy**2)
                direction[x, y] = atan2(magy, magx)
    return gradient, direction
```

Third, non-maximum suppression is a full scan of image to remove any unwanted pixels which may not constitute the edge. The result of this stage is a binary image with thin edges.

```python
def filter_out_non_maximum(gradient, direction, width, height):
    for x in range(1, width - 1):
        for y in range(1, height - 1):
            angle = direction[x, y] if direction[x, y] >= 0 else direction[x, y] + pi
            rangle = round(angle / (pi / 4))
            mag = gradient[x, y]
            if ((rangle == 0 or rangle == 4) and (gradient[x - 1, y] > mag or gradient[x + 1, y] > mag)
                    or (rangle == 1 and (gradient[x - 1, y - 1] > mag or gradient[x + 1, y + 1] > mag))
                    or (rangle == 2 and (gradient[x, y - 1] > mag or gradient[x, y + 1] > mag))
                    or (rangle == 3 and (gradient[x + 1, y - 1] > mag or gradient[x - 1, y + 1] > mag))):
                gradient[x, y] = 0
```

Final, hysteresis thresholding is the stage deciding which are edges or not. This stage is important in choosing the minVal and max Val. Any edges with intensity gradient more than maxVal are sure to be edges and below the minVal are sure to be non-edges. Also consider the edges lie between based on their connection. Function filter_strong_edges removes small pixels noises and the final result is strong edges in the image.

```python
def filter_strong_edges(gradient, width, height, low, high):
    # Keep strong edges
    keep = set()
    for x in range(width):
        for y in range(height):
            if gradient[x, y] > high:
                keep.add((x, y))

    # Keep weak edges next to a pixel to keep
    lastiter = keep
    while lastiter:
        newkeep = set()
        for x, y in lastiter:
            for a, b in ((-1, -1), (-1, 0),  (parameter) low: Any 1), (1, -1), (1, 0), (1, 1)):
                if gradient[x + a, y + b] > low and (x+a, y+b) not in keep:
                    newkeep.add((x+a, y+b))
        keep.update(newkeep)
        lastiter = newkeep

    return list(keep)
```

Hong Trinh_438443
Embedded Vision

Import necessary library and open image. Create an output image to save the result

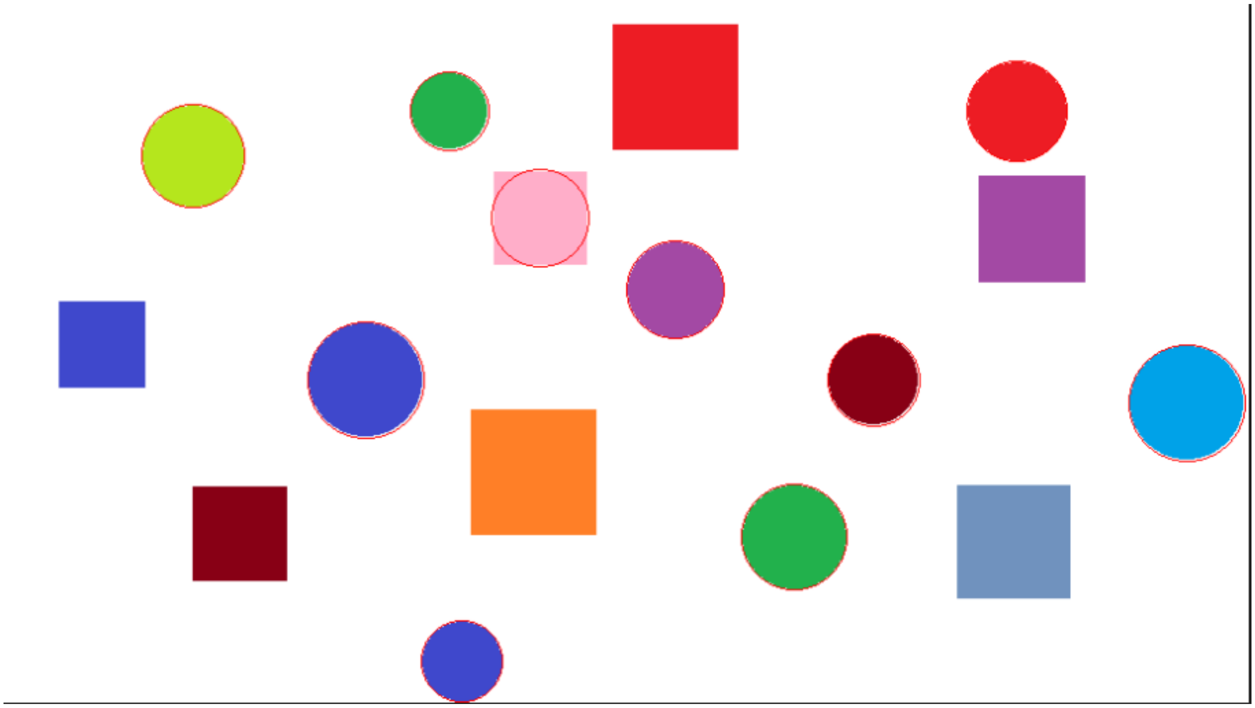
```python
from PIL import Image, ImageDraw
from math import pi, cos, sin
from canny import canny_edge_detector
from collections import defaultdict

# Load image:
input_image = Image.open("ci.png")

# Output image:
output_image = Image.new("RGB", input_image.size)
output_image.paste(input_image)
draw_result = ImageDraw.Draw(output_image)
```

Determine the maximum and minimum radius of circle can have. Using the function canny_edge_detector with the input image as parameter. Draw the border line at the circles with are detected.

```python
# Find circles
rmin = 18
rmax = 50
steps = 100
threshold = 0.4

points = []
for r in range(rmin, rmax + 1):
    for t in range(steps):
        points.append((r, int(r * cos(2 * pi * t / steps)), int(r * sin(2 * pi * t / steps))))

acc = defaultdict(int)
for x, y in canny_edge_detector(input_image):
    for r, dx, dy in points:
        a = x - dx
        b = y - dy
        acc[(a, b, r)] += 1

circles = []
for k, v in sorted(acc.items(), key=lambda i: -i[1]):
    x, y, r = k
    if v / steps >= threshold and all((x - xc) ** 2 + (y - yc) ** 2 > rc ** 2 for xc, yc, rc in circles):
        print(v / steps, x, y,  r)
        circles.append((x, y, r))

for x, y, r in circles:
    draw_result.ellipse((x-r, y-r, x+r, y+r), outline=(255,0,0,0))

# Save output image
output_image.save("resulti.png")
```

The result:

Hong Trinh_438443
Embedded Vision



Detecting circles and squares by using OpenCV:

Read and convert the image into grayscale mode. Finding the edges around the object become easy when we work with the grayscale image.

```
import cv2 as cv
#import image
image = cv.imread("ci.png")

#convert image into greyscale mode
gray_image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
```

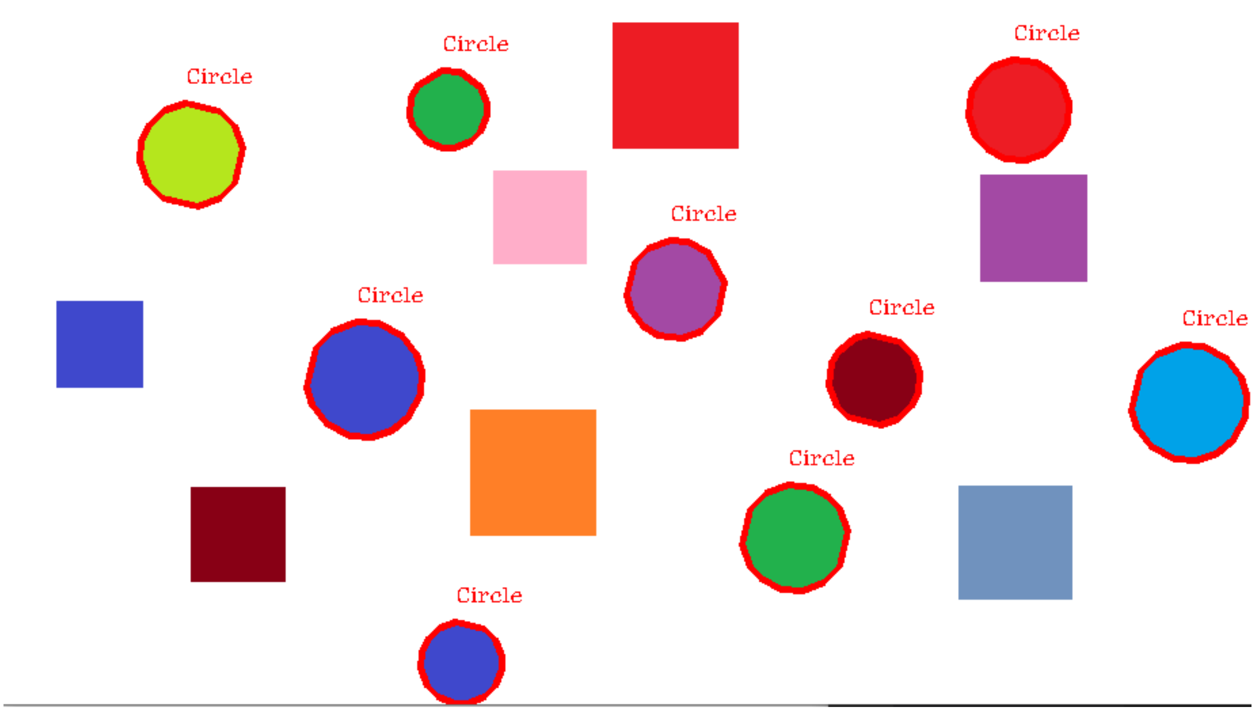
Find the threshold by using function threshold, this function finds out the threshold frequency of the gray image. 240 is the threshold value, and 250 is the maximum threshold value. Based on the thresholds the function will find out all the contours present in the grayscale image. The contours are the boundaries of the object or the continuous line around an object.

```
#find threshold of the image
_, thrash = cv.threshold(gray_image, 240, 255, cv.THRESH_BINARY)
contours, _ = cv.findContours(thrash, cv.RETR_TREE, cv.CHAIN_APPROX_NONE)
```

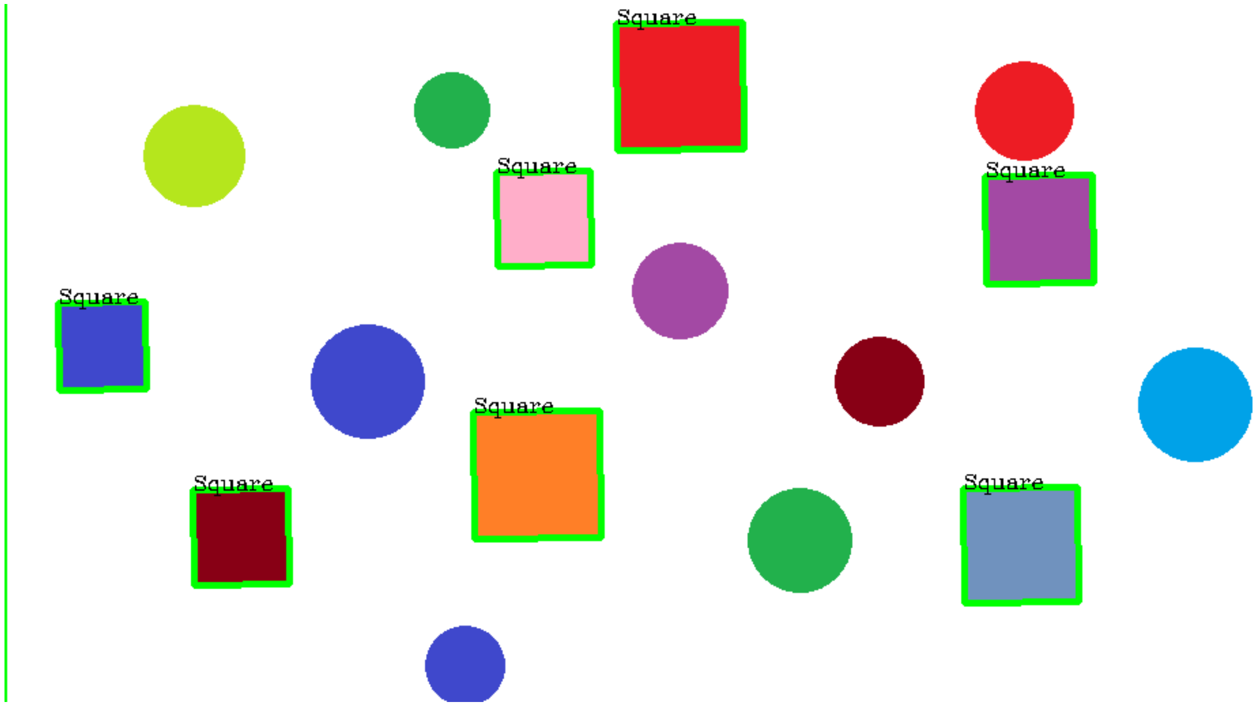
Using for loop for every contour and detect the shape. The function approPoLyDP() return s all polygons curve based on the contour with precision. True parameters specify close contour and curve. This function returns the approximates curves. Ravel()[0] return the x coordinates of the contour and ravel()[1] return y coordinates of the contour. The len() function can find out the total number of curves present in the close loop. With the square we have specified with 4 curves or edges. For the squares, the width and height aspect ratio is 1.

```
for contour in contours:
    shape = cv.approxPolyDP(contour, 0.01*cv.arcLength(contour, True), True)
    x_cor = shape.ravel()[0]
    y_cor = shape.ravel()[1]-15

    if len(shape) >12:
        cv.drawContours(image, [shape], 0, (0,0,255), 4)
        cv.putText(image, "Circle", (x_cor, y_cor), cv.FONT_HERSHEY_COMPLEX, 0.5, (0,0,255))
```

```
for contour in contours:
    shape = cv.approxPolyDP(contour, 0.01*cv.arcLength(contour, True), True)
    x_cor = shape.ravel()[0]
    y_cor = shape.ravel()[1]

    if len(shape) ==4:
        #shape cordinates
        x,y,w,h = cv.boundingRect(shape)

        #width:height
        aspectRatio = float(w)/h
        cv.drawContours(image, [shape], 0, (0,255,0), 4)
        if aspectRatio >= 0.9 and aspectRatio <=1.1:
            cv.putText(image, "Square", (x_cor, y_cor), cv.FONT_HERSHEY_COMPLEX, 0.5, (0,0,0))
```



Hong Trinh_438443
Embedded Vision



Refereneces:

https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html