Elevi: Diana Trif, Andreea Tomescu
IA3

Barcode Detection

Proiectul nostru este compus dintr-o aplicație OpenCV în Python care detectează și extrage coduri QR în orice orientare.

1. *Funcționalități:*

- detectează rapid mai multe coduri QR dintr-o imagine sau un cadru video
- oferă link-ul codurilor QR
- am folosit OpenCV
- am folosit numpy

2. *Workflow:*

Etapa 1:

- am eliminat zgomotul din imaginea eșantion și am convertit imaginea originală în tonuri de gri
- am aplicat estompare gaussiană pentru a reduce zgomotul
- am aplicat filtrul Canny Edge pentru a elimina distragerile

Etapa 2:

- am restrâns căutarea la numai localizatori de coduri QR și am căutat toate contururile rămase
- am filtrat toate contururile cu un număr de vârfuri aproximativ de patru
- am filtrat toate patrulaterele care sunt aproximativ pătrate

Etapa 3:

- am localizat locatoarele

- am căutat toate pătratele care au dimensiuni similare pătratului curent iar dacă cele mai apropiate două pătrate de pătratul curent sunt la distanțe similare, estimăm că acest pătrat este locatorul din stânga sus
- pentru a determina orientarea codului QR trebuie să găsim vârful din stânga sus

Etapa 4:

- găsim pătratul modelului de aliniere
- în timp ce căutăm printre pătrate, le stocăm pe cele care au mai puțin de jumătate din dimensiunea pătratelor de localizare
- după ce am determinat orientările locatorului, calculăm punctul de mijloc al codului QR
- determinăm că al patrulea colț al codului QR este o distanță rațională față de modelul de aliniere (în direcția opusă punctului de mijloc) iar în cazul în care nu se găsește niciun model de aliniere (codurile QR mai mici nu au acest lucru sau este posibil ca camera să nu îl detecteze)
- determinăm marginile locatoarelor care se află de-a lungul marginilor întregului cod QR care s-ar intersecta pentru a forma al patrulea colț
- găsim intersecția liniei și determinăm că acel punct este al patrulea colț

Etapa 5:

- am compensat deformarea perspectivei și am extras codul și pentru fiecare cod, am deformat vârfurile într-un pătrat pentru a fixa alinierea
- am redus cu interpolare cubică în pătrat de 29 x 29 pixeli
- am convertit într-un bit alb-negru prin limitarea imaginii și returnăm codurile formatate în listă

3. **Codul aplicației:**

//qr_extractor.py

```
import math

import cv2
import numpy as np

BLUR_VALUE = 3
```

```
SQUARE_TOLERANCE = 0.15
AREA_TOLERANCE = 0.15
DISTANCE_TOLERANCE = 0.25
WARP_DIM = 300
SMALL_DIM = 29


def count_children(hierarchy, parent, inner=False):
    if parent == -1:
        return 0
    elif not inner:
        return count_children(hierarchy, hierarchy[parent][2], True)
            return 1 + count_children(hierarchy, hierarchy[parent][0], True) +
count_children(hierarchy, hierarchy[parent][2], True)


def has_square_parent(hierarchy, squares, parent):
    if hierarchy[parent][3] == -1:
        return False
    if hierarchy[parent][3] in squares:
        return True
    return has_square_parent(hierarchy, squares, hierarchy[parent][3])


def get_center(c):
    m = cv2.moments(c)
    return [int(m["m10"] / m["m00"]), int(m["m01"] / m["m00"])]


def get_angle(p1, p2):
    x_diff = p2[0] - p1[0]
    y_diff = p2[1] - p1[1]
    return math.degrees(math.atan2(y_diff, x_diff))


def get_midpoint(p1, p2):
    return [(p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2]


def get_farthest_points(contour, center):
```

```
    distances = []
    distances_to_points = {}
    for point in contour:
        point = point[0]
        d = math.hypot(point[0] - center[0], point[1] - center[1])
        distances.append(d)
        distances_to_points[d] = point
    distances = sorted(distances)
    return [distances_to_points[distances[-1]], distances_to_points[distances[-2]]]


def line_intersection(line1, line2):
    x_diff = (line1[0][0] - line1[1][0], line2[0][0] - line2[1][0])
    y_diff = (line1[0][1] - line1[1][1], line2[0][1] - line2[1][1])

    def det(a, b):
        return a[0] * b[1] - a[1] * b[0]

    div = det(x_diff, y_diff)
    if div == 0:
        return [-1, -1]

    d = (det(*line1), det(*line2))
    x = det(d, x_diff) / div
    y = det(d, y_diff) / div
    return [int(x), int(y)]


def extend(a, b, length, int_represent=False):
    length_ab = math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)
    if length_ab * length <= 0:
        return b
     result = [b[0] + (b[0] - a[0]) / length_ab * length, b[1] + (b[1] - a[1]) / length_ab
* length]
    if int_represent:
        return [int(result[0]), int(result[1])]
    else:
        return result
```

```
def extract(frame, debug=False):
    output = frame.copy()

    # Remove noise and unnecessary contours from frame
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    gray = cv2.bilateralFilter(gray, 11, 17, 17)
    gray = cv2.GaussianBlur(gray, (BLUR_VALUE, BLUR_VALUE), 0)
    edged = cv2.Canny(gray, 30, 200)

        contours, hierarchy = cv2.findContours(edged.copy(), cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

    squares = []
    square_indices = []

    i = 0
    for c in contours:
        # Approximate the contour
        peri = cv2.arcLength(c, True)
        area = cv2.contourArea(c)
        approx = cv2.approxPolyDP(c, 0.03 * peri, True)

        # Find all quadrilateral contours
        if len(approx) == 4:
                            # Determine if quadrilateral is a square to within
SQUARE_TOLERANCE
            if area > 25 and 1 - SQUARE_TOLERANCE < math.fabs((peri / 4) ** 2) /
area < 1 + SQUARE_TOLERANCE and count_children(hierarchy[0], i) >= 2
and has_square_parent(hierarchy[0], square_indices, i) is False:
                squares.append(approx)
                square_indices.append(i)
        i += 1

    main_corners = []
    east_corners = []
    south_corners = []
    tiny_squares = []
    rectangles = []
    # Determine if squares are QR codes
    for square in squares:
```

```
        area = cv2.contourArea(square)
        center = get_center(square)
        peri = cv2.arcLength(square, True)

        similar = []
        tiny = []
        for other in squares:
            if square[0][0][0] != other[0][0][0]:
                                # Determine if square is similar to other square within
AREA_TOLERANCE
                                if math.fabs(area - cv2.contourArea(other)) / max(area,
cv2.contourArea(other)) <= AREA_TOLERANCE:
                    similar.append(other)
                elif peri / 4 / 2 > cv2.arcLength(other, True) / 4:
                    tiny.append(other)

        if len(similar) >= 2:
            distances = []
            distances_to_contours = {}
            for sim in similar:
                sim_center = get_center(sim)
                d = math.hypot(sim_center[0] - center[0], sim_center[1] - center[1])
                distances.append(d)
                distances_to_contours[d] = sim
            distances = sorted(distances)
            closest_a = distances[-1]
            closest_b = distances[-2]

            # Determine if this square is the top left QR code indicator
                    if max(closest_a, closest_b) < cv2.arcLength(square, True) * 2.5 and
math.fabs(closest_a - closest_b) / max(closest_a, closest_b) <=
DISTANCE_TOLERANCE:
                # Determine placement of other indicators (even if code is rotated)
                        angle_a = get_angle(get_center(distances_to_contours[closest_a]),
center)
                        angle_b = get_angle(get_center(distances_to_contours[closest_b]),
center)
                if angle_a < angle_b or (angle_b < -90 and angle_a > 0):
                    east = distances_to_contours[closest_a]
                    south = distances_to_contours[closest_b]
```

```
            else:
                east = distances_to_contours[closest_b]
                south = distances_to_contours[closest_a]
            midpoint = get_midpoint(get_center(east), get_center(south))
            # Determine location of fourth corner
            # Find closest tiny indicator if possible
            min_dist = 10000
            t = []
            tiny_found = False
            if len(tiny) > 0:
                for tin in tiny:
                    tin_center = get_center(tin)
                            d = math.hypot(tin_center[0] - midpoint[0], tin_center[1] -
midpoint[1])
                    if d < min_dist:
                        min_dist = d
                        t = tin
                tiny_found = len(t) > 0 and min_dist < peri

            diagonal = peri / 4 * 1.41421

            if tiny_found:
                # Easy, corner is just a few blocks away from the tiny indicator
                tiny_squares.append(t)
                offset = extend(midpoint, get_center(t), peri / 4 * 1.41421)
            else:
                 # No tiny indicator found, must extrapolate corner based off of other
corners instead
                    farthest_a = get_farthest_points(distances_to_contours[closest_a],
center)
                    farthest_b = get_farthest_points(distances_to_contours[closest_b],
center)
                # Use sides of indicators to determine fourth corner
                offset = line_intersection(farthest_a, farthest_b)
                if offset[0] == -1:
                    # Error, extrapolation failed, go on to next possible code
                    continue
                offset = extend(midpoint, offset, peri / 4 / 7)
                if debug:
```

```
                                cv2.line(output, (farthest_a[0][0], farthest_a[0][1]),
(farthest_a[1][0], farthest_a[1][1]), (0, 0, 255), 4)
                                cv2.line(output, (farthest_b[0][0], farthest_b[0][1]),
(farthest_b[1][0], farthest_b[1][1]), (0, 0, 255), 4)

            # Append rectangle, offsetting to farthest borders
                    rectangles.append([extend(midpoint, center, diagonal / 2, True),
extend(midpoint, get_center(distances_to_contours[closest_b]), diagonal / 2,
True), offset, extend(midpoint, get_center(distances_to_contours[closest_a]),
diagonal / 2, True)])
            east_corners.append(east)
            south_corners.append(south)
            main_corners.append(square)

    codes = []
    i = 0
    for rect in rectangles:
        i += 1
        # Draw rectangle
        vrx = np.array((rect[0], rect[1], rect[2], rect[3]), np.int32)
        vrx = vrx.reshape((-1, 1, 2))
        cv2.polylines(output, [vrx], True, (0, 255, 255), 1)
        # Warp codes and draw them
        wrect = np.zeros((4, 2), dtype="float32")
        wrect[0] = rect[0]
        wrect[1] = rect[1]
        wrect[2] = rect[2]
        wrect[3] = rect[3]
        dst = np.array([
            [0, 0],
            [WARP_DIM - 1, 0],
            [WARP_DIM - 1, WARP_DIM - 1],
            [0, WARP_DIM - 1]], dtype="float32")
             warp = cv2.warpPerspective(frame, cv2.getPerspectiveTransform(wrect,
dst), (WARP_DIM, WARP_DIM))
        # Increase contrast
        warp = cv2.bilateralFilter(warp, 11, 17, 17)
        warp = cv2.cvtColor(warp, cv2.COLOR_BGR2GRAY)
                    small = cv2.resize(warp, (SMALL_DIM, SMALL_DIM), 0, 0,
interpolation=cv2.INTER_CUBIC)
```

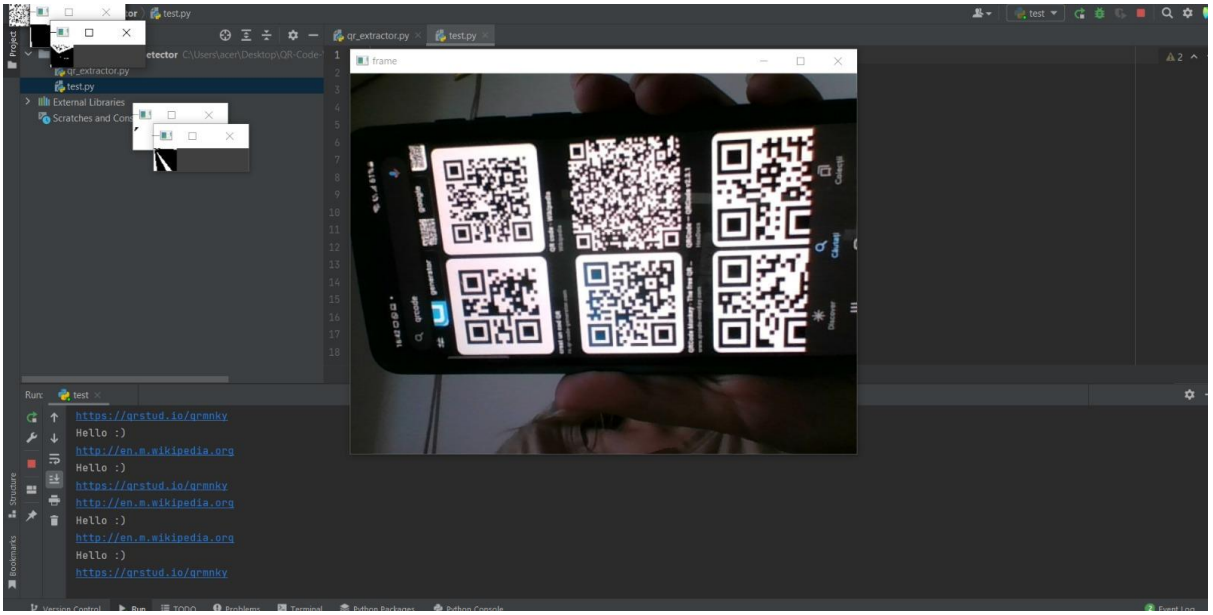
```
        _, small = cv2.threshold(small, 100, 255, cv2.THRESH_BINARY)
        codes.append(small)
        if debug:
            cv2.imshow("Warped: " + str(i), small)

    if debug:
        # Draw debug information onto frame before outputting it
        cv2.drawContours(output, squares, -1, (5, 5, 5), 2)
        cv2.drawContours(output, main_corners, -1, (0, 0, 128), 2)
        cv2.drawContours(output, east_corners, -1, (0, 128, 0), 2)
        cv2.drawContours(output, south_corners, -1, (128, 0, 0), 2)
        cv2.drawContours(output, tiny_squares, -1, (128, 128, 0), 2)

    return codes, frame
```

//test.py

```
import cv2

import qr_extractor as reader
import numpy as np
from pyzbar.pyzbar import decode

cap = cv2.VideoCapture(0)
cap.set(3,640)
cap.set(4,480)

while True:
    success, img = cap.read()
    for barcode in decode(img):
        myData = barcode.data.decode('utf-8')
        print(myData)
        pts = np.array([barcode.polygon], np.int32)
        pts = pts.reshape((-1, 1, 2))
        cv2.polylines(img, [pts], True, (255, 0, 255), 5)
        pts2 = barcode.rect
            cv2.putText(img, myData, (pts2[0], pts2[1]),
cv2.FONT_HERSHEY_SIMPLEX,
                    0.9, (255, 0, 255), 2)

    _, frame = cap.read()
    codes, frame = reader.extract(frame, True)
```

```
    cv2.imshow("frame", frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        print("I quit!")
        break


# When everything done, release the capture
cap.release()
cv2.destroyAllWindows()
```

3.1. Date de ieșire:



4. *Indicatori de performanță:*

Metrica	***Indicator***	***Informații Suplimentare***	***Modalitate de calcul***
Timp	Încadrare în termene		Numărul de zile de întârziere:0
Calitate	Numărul de erori apărute	Indicatorul măsoară	

Portabilitatea		ușurința cu care o aplicație poate fi executată pe diverse platforme hardware și software, de obicei este dependentă de tehnologia folosită pentru implementare	☑
Testabilitate		cât de ușor poate fi testată o aplicație; arhitectura să fie cât mai simplă;	☑
Scalabilitatea		cât de bine se comportă sistemul dacă dimensiunea problemei pentru care el a fost proiectat să o rezolve creşte.	☑
Timpul de Răspuns (Response Time)		timpul necesar unui sistem software pentru a răspunde la o anumită modificare	timp relativ mic

5. Îmbunătățiri:

- să poată extrage coduri QR colorate
- să poată distinge coduri QR cu diferite ornamente