Folder: Asphalt

Folder: People Counter

Folder: Vehicle Count

Folder: Virtual Keyboard

Error processing: Ran out of input

File: directkeys.py

Documented code for directkeys.py:

**Code Explanation:**

**1. Import Statements (Line 1):**

```
import ctypes
import time
```

These lines import the necessary libraries for the script's functionality. ctypes is used for interacting with the Windows API, and time is used for controlling the sleep duration between keypresses.

**2. Variable Definitions (Lines 2-5):**

```
SendInput = ctypes.windll.user32.SendInput
```

This line defines a variable called SendInput, which is assigned the function pointer to the SendInput function from the user32.dll Windows library. This function is used to simulate keypresses and releases.

```
A = 0x1E
D = 0x20
Space = 0x39
```

```

These lines define constants representing the virtual key codes for the 'A,' 'D,' and 'Space' keys. These values are used as parameters to the PressKey and ReleaseKey functions.

**3. C Struct Redefinitions (Lines 6-40):**

These lines define C structures that are used to represent input events. These structures are necessary for interacting with the SendInput function.

**4. Actual Functions (Lines 41-55):**

```
def PressKey(hexKeyCode):
    ...
```

```
def ReleaseKey(hexKeyCode):
    ...
```

These functions are used to simulate keypresses and releases. They take a single parameter, hexKeyCode, which is the virtual key code of the key to be pressed or released.

**5. Main Program (Lines 56-60):**

```
if __name__ == '__main__':
    PressKey(0x11)
    time.sleep(1)
    ReleaseKey(0x11)
    time.sleep(1)
```
```

```

These lines represent the main program. It starts by pressing the 'A' key using the PressKey function, then waits for 1 second using the time.sleep function, and finally releases the 'A' key using the ReleaseKey function.

**Conclusion:**

This script uses ctypes to simulate keypresses and releases on a Windows system. It defines functions for pressing and releasing keys, and uses these functions to simulate pressing and releasing the 'A' key with a 1-second interval in between.

File: color.py
Documented code for color.py:
**Code Explanation:**

**1. Import Statements (Line 1):**

```
import ctypes
import time
```

- We import the `ctypes` library to interact with the Windows API and the `time` library to introduce delays in the program.

**2. Variable Declarations (Lines 2-5):**

```
SendInput = ctypes.windll.user32.SendInput

A = 0x1E
D = 0x20
```
```

```
Space = 0x39
```

- `SendInput`: This is a function pointer to the `SendInput` function from the Windows API. It allows us to simulate keyboard input.
- `A`, `D`, and `Space`: These are hexadecimal values representing the virtual key codes for the 'A', 'D', and 'Space' keys, respectively.

**3. C Struct Redefinitions (Lines 6-45):**

- These lines define various C structures that are used to represent keyboard, mouse, and hardware input events. These structures are necessary for interacting with the Windows API.

**4. Actual Functions (Lines 47-61):**

```
def PressKey(hexKeyCode):
    # Simulate a key press
    ...

def ReleaseKey(hexKeyCode):
    # Simulate a key release
    ...
```

- `PressKey` and `ReleaseKey`: These functions take a hexadecimal key code as an argument and simulate a key press or release, respectively. They use the `SendInput` function to send the input events to the system.

**5. Main Program (Lines 63-67):**

```
if __name__ == '__main__':
```

```
    PressKey(0x11)
    time.sleep(1)
    ReleaseKey(0x11)
    time.sleep(1)
```

- This is the entry point of the program. It first calls `PressKey(0x11)` to simulate pressing the 'Q' key. Then, it uses `time.sleep(1)` to introduce a one-second delay. Next, it calls `ReleaseKey(0x11)` to simulate releasing the 'Q' key. Finally, it uses `time.sleep(1)` again to introduce another one-second delay.

**Conclusion:**

This program demonstrates how to simulate keyboard input in Python using the Windows API. It defines functions to press and release keys and uses them to simulate pressing and releasing the 'Q' key with a one-second delay in between.

File: steering.py
Documented code for steering.py:
**Code Explanation:**

**1. Import Statements:**

```
import cv2
import imutils
from imutils.video import VideoStream
import numpy as np
from directkeys import A, D, Space, ReleaseKey, PressKey
```

- We import necessary libraries for computer vision (cv2), video streaming (imutils), image processing (numpy), and keyboard control (directkeys).

**2. Variable Declarations:**

```
cam = VideoStream(src=0).start()
currentKey = list()
```

- `cam`: VideoStream object initialized to capture video from the default camera (webcam).
- `currentKey`: List to keep track of currently pressed keys.

**3. User Input Handling:**

```
while True:
    key = False
```

- We enter an infinite loop to continuously process video frames.
- `key` is a flag to check if any key is currently pressed.

**4. Control Flow - Part 1:**

```
    img = cam.read()
    img = np.flip(img, axis=1)
    img = np.array(img)
```

- We read a frame from the video stream, flip it horizontally to match the game's perspective, and convert it to a NumPy array.

**5. Function Calls - Part 1:**

```
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    blurred = cv2.GaussianBlur(hsv, (11, 11), 0)
```

- We convert the image to HSV color space and apply Gaussian blur to reduce noise.

**6. Data Manipulation:**

```
    colorLower = np.array([31, 33, 153])
    colorUpper = np.array([100, 255, 255])
    mask = cv2.inRange(blurred, colorLower, colorUpper)
```

- We define color ranges for the steering wheel and nitro boost areas.
- We create a mask using `cv2.inRange` to isolate these areas.

**7. Looping Structures:**

```
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((5, 5), np.uint8))
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, np.ones((5, 5), np.uint8))
```

- We apply morphological operations (opening and closing) to remove noise and fill gaps in the mask.

**8. Control Flow - Part 2:**

```
    width = img.shape[1]
```

```
    height = img.shape[0]
```

- We get the width and height of the image.

**9. Function Calls - Part 2:**

```
    upContour = mask[0:height // 2, 0:width]
    downContour = mask[3 * height // 4:height, 2 * width // 5:3 * width // 5]
```

- We split the mask into two regions: the upper half for steering and the lower-right corner for nitro boost.

**10. Data Manipulation:**

```
    cnts_up = cv2.findContours(upContour, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    cnts_up = imutils.grab_contours(cnts_up)

                        cnts_down = cv2.findContours(downContour, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    cnts_down = imutils.grab_contours(cnts_down)
```

- We find contours in both regions to detect objects (steering wheel and nitro boost button).

**11. Control Flow - Part 3:**

```
    if len(cnts_up) > 0:
        c = max(cnts_up, key=cv2.contourArea)
```

```
        M = cv2.moments(c)
        cX = int(M["m10"] / M["m00"])

        if cX < (width // 2 - 35):
            PressKey(A)
            key = True
            currentKey.append(A)
        elif cX > (width // 2 + 35):
            PressKey(D)
            key = True
            currentKey.append(D)
```

- If a contour is detected in the upper region, we calculate its center of mass (cX).
- Based on the center's position, we press the 'A' key for left, 'D' key for right, and update `key` and `currentKey`.

**12. Control Flow - Part 4:**

```
    if len(cnts_down) > 0:
        PressKey(Space)
        key = True
        currentKey.append(Space)
```

- If a contour is detected in the lower-right region, we press the 'Space' key for nitro boost, update `key` and `currentKey`.

**13. Output Generation:**

```
    img = cv2.rectangle(img, (0, 0), (width // 2 - 35, height // 2), (0, 255, 0), 1)
```

```
    cv2.putText(img, "LEFT", (110, 30), cv2.FONT_HERSHEY_COMPLEX, 1, (139, 0, 0))

    img = cv2.rectangle(img, (width // 2 + 35, 0), (width, height // 2), (0, 255, 0), 1)
    cv2.putText(img, "RIGHT", (440, 30), cv2.FONT_HERSHEY_COMPLEX, 1, (139, 0, 0))

    img = cv2.rectangle(img, (2 * (width // 5), 3 * height // 4), (3 * width // 5, height), (0, 255, 0), 1)
     cv2.putText(img, "NITRO", (2 * (width // 5) + 20, height - 10), cv2.FONT_HERSHEY_COMPLEX,
1, (139, 0, 0))

    cv2.imshow("Steering Wheel [HAWK-AI]", img)
```

- We draw rectangles and labels on the image to visualize the steering wheel and nitro boost areas.
- We display the image in a window.

**14. Error Handling:**

```
    if not key and len(currentKey) != 0:
        for current in currentKey:
            ReleaseKey(current)
```

- If no key is currently pressed and there are keys in `currentKey`, we release all pressed keys.

**15. User Input Handling:**

```
    Key = cv2.waitKey(1) & 0xFF
    if Key == ord('q'):
        break
```

- We check for user input. If the 'q' key is pressed, we break out of the loop and exit the program.

**16. Cleanup:**

```
cv2.destroyAllWindows()
```

- We destroy all OpenCV windows before exiting the program.

**Conclusion:**

This program uses computer vision to detect the steering wheel and nitro boost button in a video game and controls the game accordingly. It continuously captures video frames, processes them to identify the control areas, and sends keystrokes to the game based on the detected positions. The program also provides visual feedback by displaying the detected areas on the screen.

File: main.py
Documented code for main.py:
**Code Explanation:**

**1. Import Statements (Line 1):**

```
import cvzone
import cv2
from cvzone.HandTrackingModule import HandDetector
from time import sleep
import numpy as np
from pynput.keyboard import Controller, Key
```

- We import necessary libraries and modules:

- `cvzone`: Computer Vision Zone library for hand tracking.
- `cv2`: OpenCV library for image processing and computer vision.
- `HandDetector`: Hand Detector class from `cvzone`.
- `sleep`: Used for adding delays in the program.
- `numpy`: For numerical operations and array handling.
- `Controller`, `Key`: From `pynput` library, used for keyboard control.

**2. Variable Declarations (Line 2-5):**

```
cap = cv2.VideoCapture(0)
detector = HandDetector(detectionCon=0.8, maxHands=2)
keys = [["Q", "W", "E", "R", "T", "Y", "U", "I", "O", "P"],
        ["A", "S", "D", "F", "G", "H", "J", "K", "L", ";"],
        ["Z", "X", "C", "V", "B", "N ", "M"]]
finaltext = ""
keyboard = Controller()
```

- `cap`: Video capture object to access the webcam.
- `detector`: Hand Detector object for hand tracking.
- `keys`: List of lists containing the keyboard layout.
- `finaltext`: String to store the typed text.
- `keyboard`: Controller object to control the keyboard.

**3. User Input Handling (Line 6-10):**

```
while True:
    success, img = cap.read()
    hands, img = detector.findHands(img)
```

- An infinite loop to continuously capture frames from the webcam.
- `success, img = cap.read()`: Reads a frame from the webcam.
- `hands, img = detector.findHands(img)`: Detects hands in the frame.

**4. Control Flow - Part 1 (Line 11-15):**

```
    if hands:
        # Hand 1
        hand = hands[0]
        lmList = hand["lmList"]  # List of 21 Landmarks points
        bbox = hand["bbox"]  # Bounding Box info x,y,w,h
        centerPoint = hand["center"]  # center of the hand cx,cy
        handType = hand["type"]  # Hand Type Left or Right
```

- If hands are detected, it extracts information about the first hand:
  - `hand`: First detected hand.
  - `lmList`: List of 21 landmark points of the hand.
  - `bbox`: Bounding box around the hand.
  - `centerPoint`: Center point of the hand.
  - `handType`: Type of hand (Left or Right).

**5. Function Calls - Part 1 (Line 16-20):**

```
    if len(hands) == 2:
        hand2 = hands[1]
        lmList2 = hand2["lmList"]  # List of 21 Landmarks points
        bbox2 = hand2["bbox"]  # Bounding Box info x,y,w,h
        centerPoint2 = hand2["center"]  # center of the hand cx,cy
        handType2 = hand2["type"]  # Hand Type Left or Right
```

- If two hands are detected, it extracts information about the second hand.

**6. Looping Structures (Line 21-25):**

```
    for button in buttonList:
        x, y = button.pos
        w, h = button.size
```

- Iterates through the list of buttons on the keyboard layout.

**7. Data Manipulation (Line 26-30):**

```
        if x < lmList[8][0] < x + w and y < lmList[8][1] < y + h:
            # Tip of finger is point no.8(index finger)
            cv2.rectangle(img, button.pos, (x + w, y + h), (175, 0, 175), cv2.FILLED)
              cv2.putText(img, button.text, (x + 20, y + 65), cv2.FONT_HERSHEY_PLAIN, 4, (255, 255, 255), 4)
            l, _, _ = detector.findDistance(centerPoint, centerPoint2, img)
```

- Checks if the tip of the index finger is within the bounds of a button.
- If so, it highlights the button and displays its text.
- It also calculates the distance between the two hands' center points.

**8. Control Flow - Part 2 (Line 31-35):**

```
            if l < 1000:
                # keyboard.press(button.text)
```

```
                keyboard.press(button.text.lower() if len(button.text) == 1 else Key.space)  # Press lowercase letter or space
                cv2.rectangle(img, button.pos, (x + w, y + h), (0, 255, 0), cv2.FILLED)
                cv2.putText(img, button.text, (x + 20, y + 65), cv2.FONT_HERSHEY_PLAIN, 4, (255, 255, 255), 4)
                finaltext += button.text
                sleep(1)
```

- If the distance between the two hands is less than 1000 pixels, it presses the corresponding key on the keyboard.
- It also updates the `finaltext` string with the pressed key.

**9. Function Calls - Part 2 (Line 36-40):**

```
    cv2.rectangle(img, (50, 350), (700, 450), (0, 0, 0), cv2.FILLED)
    cv2.putText(img, finaltext, (60, 430), cv2.FONT_HERSHEY_PLAIN, 4, (255, 255, 255), 4)
```

- Draws a rectangle to display the typed text.
- Displays the `finaltext` string within the rectangle.

**10. Output Generation (Line 41-45):**

```
    cv2.imshow("image", img)
    if cv2.waitKey(1) == 27:
        break
```

- Displays the frame with the hand tracking and keyboard layout.
- Checks if the 'Esc' key is pressed to exit the program.

**11. Error Handling (Line 46-50):**

```
cap.release()
cv2.destroyAllWindows()
```

- Releases the webcam capture object and destroys all OpenCV windows.

**Conclusion:**

This program uses hand tracking to control a virtual keyboard on the screen. It detects hands and their landmarks, and when the tip of the index finger is within the bounds of a button, it presses the corresponding key on the keyboard. The typed text is displayed on the screen. The program continues to run until the 'Esc' key is pressed.

Folder: __pycache__
File: distance.py
Documented code for distance.py:
**Code Explanation:**

**1. Import Statements:**

```
import cv2
import datetime
import imutils
import numpy as np
from centroidtracker import CentroidTracker
from itertools import combinations
import math
```

- We import necessary libraries for computer vision, date and time manipulation, image processing, and object tracking.

**2. Variable Declarations:**

```
protopath = "MobileNetSSD_deploy.prototxt"
modelpath = "MobileNetSSD_deploy.caffemodel"
detector = cv2.dnn.readNetFromCaffe(prototxt=protopath, caffeModel=modelpath)

tracker = CentroidTracker(maxDisappeared=80, maxDistance=90)

CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
          "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
          "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
          "sofa", "train", "tvmonitor"]
```

- We define paths to the MobileNetSSD model files and load the model for object detection.
- We initialize a CentroidTracker object for tracking people.
- We define a list of object class labels.

**3. User Input Handling:**

```
cap = cv2.VideoCapture("videos/testvideo2.mp4")
```

- We open a video capture object to read the input video.

**4. Control Flow - Part 1:**

```
while True:
    ret, frame = cap.read()
    if not ret:
        break
```

- We enter a loop to continuously read frames from the video.
- If there are no more frames, we break out of the loop.

**5. Function Calls - Part 1:**

```
frame = imutils.resize(frame, width=600)

blob = cv2.dnn.blobFromImage(frame, 0.007843, (600, 450), 127.5)
detector.setInput(blob)
person_detections = detector.forward()
```

- We resize the frame for faster processing.
- We convert the frame to a blob for object detection.
- We pass the blob to the object detector and get the detections.

**6. Looping Structures:**

```
rects = []

for i in np.arange(0, person_detections.shape[2]):
    confidence = person_detections[0, 0, i, 2]
    if confidence > 0.5:
        idx = int(person_detections[0, 0, i, 1])
```

```
        if CLASSES[idx] != "person":
            continue

        person_box = person_detections[0, 0, i, 3:7] * np.array([W, H, W, H])
        (startX, startY, endX, endY) = person_box.astype("int")
        rects.append(person_box)
```

- We iterate over the detections and filter out non-person detections.
- We extract the bounding box coordinates for each person detection.

**7. Data Manipulation:**

```
boundingboxes = np.array(rects)
boundingboxes = boundingboxes.astype(int)
rects = non_max_supression_fast(boundingboxes, 0.3)
```

- We convert the list of bounding boxes to a NumPy array and cast it to integers.
- We apply non-maximum suppression to remove overlapping bounding boxes.

**8. Control Flow - Part 2:**

```
objects = tracker.update(rects)
```

- We update the tracker with the remaining bounding boxes to get object IDs and their corresponding bounding boxes.

**9. Function Calls - Part 2:**

```
for (objectId, bbox) in objects.items():
    x1, y1, x2, y2 = bbox
    x1 = int(x1)
    y1 = int(y1)
    x2 = int(x2)
    y2 = int(y2)
    cX = int((x1 + x2) / 2.0)
    cY = int((y1 + y2) / 2.0)

    centroid_dict[objectId] = (cX, cY, x1, y1, x2, y2)
```

- We iterate over the tracked objects and extract their bounding box coordinates and centroids.
- We store the object ID and its corresponding data in a dictionary.

**10. Output Generation:**

```
red_zone_list = []
for (id1, p1), (id2, p2) in combinations(centroid_dict.items(), 2):
    dx, dy = p1[0] - p2[0], p1[1] - p2[1]
    distance = math.sqrt(dx * dx + dy * dy)
    if distance < 75.0:
        if id1 not in red_zone_list:
            red_zone_list.append(id1)
        if id2 not in red_zone_list:
            red_zone_list.append(id2)

for id, box in centroid_dict.items():
    if id in red_zone_list:
        cv2.rectangle(frame, (box[2], box[3]), (box[4], box[5]), (0, 0, 255), 2)
```

```
    else:
        cv2.rectangle(frame, (box[2], box[3]), (box[4], box[5]), (0, 255, 0), 2)
```

- We calculate the distance between each pair of centroids.
- If the distance is less than a threshold, we mark the corresponding objects as being in the red zone.
- We draw bounding boxes around the objects, coloring them red for those in the red zone and green for others.

**11. Error Handling:**

```
fps_and_time = datetime.datetime.now()
time_diff = fps_and_time - fps_start
if time_diff.seconds == 0:
    fps = 0.0
else:
    fps = (total_frames / time_diff.seconds)

fps_text = "FPS:{:.2f}".format(fps)

cv2.putText(frame, fps_text, (5, 30), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 255), 2)
cv2.imshow("FPS", frame)
key = cv2.waitKey(1)
if key == ord('q'):
    break
```

- We calculate the frames per second (FPS) and display it on the frame.
- We handle user input to allow quitting the program by pressing the 'q' key.

**Conclusion:**

This code demonstrates real-time object detection and tracking of people in a video. It uses a pre-trained MobileNetSSD model for object detection and a CentroidTracker for tracking people. The code also implements a social distancing check by calculating the distance between people and marking those who are too close as being in the red zone. The output is a video feed with bounding boxes around people and red zones indicating potential social distancing violations.

File: final.py
Documented code for final.py:
**1. Import Statements:**

```
import cv2
import datetime
import imutils
import numpy as np
from centroidtracker import CentroidTracker
import psycopg2
import psycopg2.extras
```

- We import essential libraries for computer vision, image processing, and database connectivity.

**2. Variable Declarations:**

```
protopath = "MobileNetSSD_deploy.prototxt"
modelpath = "MobileNetSSD_deploy.caffemodel"
detector = cv2.dnn.readNetFromCaffe(prototxt=protopath, caffeModel=modelpath)

tracker = CentroidTracker(maxDisappeared=80, maxDistance=90)

CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
```

```
        "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
        "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
        "sofa", "train", "tvmonitor"]
```

- We define paths to the pre-trained MobileNetSSD model and initialize the object detector.
- We create a CentroidTracker object for tracking people.
- We define a list of object classes for classification.

**3. User Input Handling:**

```
cap = cv2.VideoCapture(0)  # Use 0 for webcam or provide a video file path
```

- We initialize the video capture using the webcam or a video file.

**4. Control Flow - Part 1:**

```
while True:
    ret, frame = cap.read()
    if not ret:
        break
```

- We enter an infinite loop to continuously process video frames.
- We read the next frame from the video capture and check if it's valid.

**5. Function Calls - Part 1:**

```
frame = imutils.resize(frame, width=600)
```

```
blob = cv2.dnn.blobFromImage(frame, 0.007843, (600, 450), 127.5)
detector.setInput(blob)
person_detections = detector.forward()
```

- We resize the frame for faster processing.
- We create a blob from the frame for object detection.
- We pass the blob to the object detector to get person detections.

**6. Looping Structures:**

```
rects = []

for i in np.arange(0, person_detections.shape[2]):
    confidence = person_detections[0, 0, i, 2]
    if confidence > 0.5:
        idx = int(person_detections[0, 0, i, 1])

        if CLASSES[idx] != "person":
            continue

        person_box = person_detections[0, 0, i, 3:7] * np.array([W, H, W, H])
        (startX, startY, endX, endY) = person_box.astype("int")
        rects.append(person_box)
```

- We iterate through the person detections and filter out those with low confidence.
- We extract the bounding boxes of detected persons.

**7. Data Manipulation:**

```
boundingboxes = np.array(rects)
boundingboxes = boundingboxes.astype(int)
rects = non_max_supression_fast(boundingboxes, 0.3)
```

- We convert the list of bounding boxes to a NumPy array and cast it to integers.
- We apply non-maximum suppression to remove overlapping bounding boxes.

**8. Control Flow - Part 2:**

```
objects = tracker.update(rects)
```

- We update the CentroidTracker with the filtered bounding boxes to track people.

**9. Function Calls - Part 2:**

```
for (objectId, bbox) in objects.items():
    x1, y1, x2, y2 = bbox
    x1 = int(x1)
    y1 = int(y1)
    x2 = int(x2)
    y2 = int(y2)

    cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
    text = "ID:{}".format(objectId)
    cv2.putText(frame, text, (x1, y1 - 5), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 255, 0), 1)
```

- We iterate through the tracked objects and draw bounding boxes around them.

- We also display the object ID for each person.

**10. Output Generation:**

```
lpc_count = len(objects)
opc_count = len(object_id_list)
lpc_txt = "LIVE COUNT: {}".format(lpc_count)
opc_txt = "TOTAL COUNT:{}".format(opc_count)

cv2.putText(frame, lpc_txt, (5, 60), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
cv2.putText(frame, opc_txt, (5, 90), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

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

- We calculate the live person count (LPC) and the overall person count (OPC).
- We display the LPC and OPC on the frame.
- We display the frame in a window.

**11. Error Handling:**

```
key = cv2.waitKey(1)
if key == ord('q'):
    break
```

- We wait for a key press. If the 'q' key is pressed, we break out of the loop.

**12. Cleanup:**

```
```

```
cv2.destroyAllWindows()
```

- We destroy all OpenCV windows when the loop exits.

**Conclusion:**

This Python script uses OpenCV, CentroidTracker, and a pre-trained MobileNetSSD model to detect and track people in a video stream. It displays the live person count and the overall person count on the frame. The script also sends the count data to a PostgreSQL database in real time. This system can be used for various applications such as crowd counting, security surveillance, and people analytics.

File: centroidtracker.py
Documented code for centroidtracker.py:
**Code Explanation:**

**1. Import Statements:**

```
from scipy.spatial import distance as dist
from collections import OrderedDict
import numpy as np
```

- We import the necessary libraries and modules.
- `scipy.spatial.distance` provides functions for computing distances between points.
- `collections.OrderedDict` is used to maintain the order of elements in dictionaries.
- `numpy` is used for numerical operations.

**2. Class Definition:**

```
```

```
class CentroidTracker:
    def __init__(self, maxDisappeared=50, maxDistance=50):
        # initialize the next unique object ID along with two ordered
        # dictionaries used to keep track of mapping a given object
        # ID to its centroid and number of consecutive frames it has
        # been marked as "disappeared", respectively
        self.nextObjectID = 0
        self.objects = OrderedDict()
        self.disappeared = OrderedDict()
        self.bbox = OrderedDict()  # CHANGE

        # store the number of maximum consecutive frames a given
        # object is allowed to be marked as "disappeared" until we
        # need to deregister the object from tracking
        self.maxDisappeared = maxDisappeared

        # store the maximum distance between centroids to associate
        # an object -- if the distance is larger than this maximum
        # distance we'll start to mark the object as "disappeared"
        self.maxDistance = maxDistance
```

- We define a class called `CentroidTracker` that will be used to track objects in a video stream.
- The constructor initializes several instance variables:
  - `nextObjectID`: The next unique object ID to be assigned.
  - `objects`: An ordered dictionary that maps object IDs to their centroids.
  - `disappeared`: An ordered dictionary that maps object IDs to the number of consecutive frames they have been marked as "disappeared".
  - `bbox`: An ordered dictionary that maps object IDs to their bounding boxes. (CHANGE)
  - `maxDisappeared`: The maximum number of consecutive frames an object can be marked as "disappeared" before it is deregistered.
  - `maxDistance`: The maximum distance between centroids that is allowed for an object to be associated with an existing track.

**3. Registering an Object:**

```
    def register(self, centroid, inputRect):
        # when registering an object we use the next available object
        # ID to store the centroid
        self.objects[self.nextObjectID] = centroid
        self.bbox[self.nextObjectID] = inputRect  # CHANGE
        self.disappeared[self.nextObjectID] = 0
        self.nextObjectID += 1
```

- The `register()` method is used to register a new object in the tracker.
- It takes two arguments:
  - `centroid`: The centroid of the object.
  - `inputRect`: The bounding box of the object. (CHANGE)
- It assigns the next available object ID to the object and adds it to the `objects`, `bbox`, and `disappeared` dictionaries.

**4. Deregistering an Object:**

```
    def deregister(self, objectID):
        # to deregister an object ID we delete the object ID from
        # both of our respective dictionaries
        del self.objects[objectID]
        del self.disappeared[objectID]
        del self.bbox[objectID]  # CHANGE
```

- The `deregister()` method is used to deregister an object from the tracker.
- It takes one argument:

- `objectID`: The ID of the object to be deregistered.
- It removes the object from the `objects`, `bbox`, and `disappeared` dictionaries.

**5. Updating the Tracker:**

```
def update(self, rects):
    # check to see if the list of input bounding box rectangles
    # is empty
    if len(rects) == 0:
        # loop over any existing tracked objects and mark them
        # as disappeared
        for objectID in list(self.disappeared.keys()):
            self.disappeared[objectID] += 1

            # if we have reached a maximum number of consecutive
            # frames where a given object has been marked as
            # missing, deregister it
            if self.disappeared[objectID] > self.maxDisappeared:
                self.deregister(objectID)

        # return early as there are no centroids or tracking info
        # to update
        # return self.objects
        return self.bbox

    # initialize an array of input centroids for the current frame
    inputCentroids = np.zeros((len(rects), 2), dtype="int")
    inputRects = []
    # loop over the bounding box rectangles
    for (i, (startX, startY, endX, endY)) in enumerate(rects):
        # use the bounding box coordinates to derive the centroid
        cX = int((startX + endX) / 2.0)
```

```
            cY = int((startY + endY) / 2.0)
            inputCentroids[i] = (cX, cY)
            inputRects.append(rects[i])  # CHANGE

        # if we are currently not tracking any objects take the input
        # centroids and register each of them
        if len(self.objects) == 0:
            for i in range(0, len(inputCentroids)):
                self.register(inputCentroids[i], inputRects[i])  # CHANGE

        # otherwise, are are currently tracking objects so we need to
        # try to match the input centroids to existing object
        # centroids
        else:
            # grab the set of object IDs and corresponding centroids
            objectIDs = list(self.objects.keys())
            objectCentroids = list(self.objects.values())

            # compute the distance between each pair of object
            # centroids and input centroids, respectively -- our
            # goal will be to match an input centroid to an existing
            # object centroid
            D = dist.cdist(np.array(objectCentroids), inputCentroids)

            # in order to perform this matching we must (1) find the
            # smallest value in each row and then (2) sort the row
            # indexes based on their minimum values so that the row
            # with the smallest value as at the *front* of the index
            # list
            rows = D.min(axis=1).argsort()

            # next, we perform a similar process on the columns by
            # finding the smallest value in each column and then
```

```
        # sorting using the previously computed row index list
        cols = D.argmin(axis=1)[rows]

        # in order to determine if we need to update, register,
        # or deregister an object we need to keep track of which
        # of the rows and column indexes we have already examined
        usedRows = set()
        usedCols = set()

        # loop over the combination of the (row, column) index
        # tuples
        for (row, col) in zip(rows, cols):
            # if we have already examined either the row or
            # column value before, ignore it
            if row in usedRows or col in usedCols:
                continue

            # if the distance between centroids is greater than
            # the maximum distance, do not associate the two
            # centroids to the same object
            if D[row, col] > self.maxDistance:
                continue

            # otherwise, grab the object ID for the current row,
            # set its new centroid, and reset the disappeared
            # counter
            objectID = objectIDs[row]
            self.objects[objectID] = inputCentroids[col]
            self.bbox[objectID] = inputRects[col]  # CHANGE
            self.disappeared[objectID] = 0

            # indicate that we have examined each of the row and
            # column indexes, respectively
```

```
            usedRows.add(row)
            usedCols.add(col)

        # compute both the row and column index we have NOT yet
        # examined
        unusedRows = set(range(0, D.shape[0])).difference(usedRows)
        unusedCols = set(range(0, D.shape[1])).difference(usedCols)

        # in the event that the number of object centroids is
        # equal or greater than the number of input centroids
        # we need to check and see if some of these objects have
        # potentially disappeared
        if D.shape[0] >= D.shape[1]:
            # loop over the unused row indexes
            for row in unusedRows:
                # grab the object ID for the corresponding row
                # index and increment the disappeared counter
                objectID = objectIDs[row]
                self.disappeared[objectID] += 1

                # check to see if the number of consecutive
                # frames the object has been marked "disappeared"
                # for warrants deregistering the object
                if self.disappeared[objectID] > self
```