Folder: Asphalt

Folder: People Counter

Folder: Vehicle Count

Folder: Virtual Keyboard

File: main.py

Documented code for main.py:

Code Explanation:

1. Import Statements:

```
import cv2
import numpy as np
import psycopg2
```

- `cv2`: OpenCV library for computer vision tasks.
- `numpy`: NumPy library for numerical operations.
- `psycopg2`: Psycopg2 library for connecting to PostgreSQL databases.

2. Variable Declarations:

```
cap = cv2.VideoCapture('video.mp4')
min_width_rect = 80
min_height_rect = 80
count_line_position = 550
```

- `cap`: VideoCapture object for reading the input video file.
- `min_width_rect` and `min_height_rect`: Minimum width and height thresholds for detecting vehicles.
- `count_line_position`: Position of the line used for counting vehicles.

3. User Input Handling:

~~~
# No user input handling in this code.
```
~~~

4. Control Flow - Part 1:

~~~
while True:
    # Code inside the loop will continue to execute until 'break' is encountered.
```
~~~

5. Function Calls - Part 1:

~~~
ret, frame1 = cap.read()
```
~~~

- `cap.read()`: Reads the next frame from the video file.

6. Looping Structures:

~~~
while True:
    # Code inside the loop will continue to execute until 'break' is encountered.
```
~~~

7. Data Manipulation:

````
grey = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
````

~~~
blur = cv2.GaussianBlur(grey, (3, 3), 5)
img_sub = algo.apply(blur)
dilat = cv2.dilate(img_sub, np.ones((5, 5)))
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
dilate = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel)
dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)
```
~~~

- Color conversion, Gaussian blur, background subtraction, dilation, and morphological operations are applied to the video frame for vehicle detection.

8. Control Flow - Part 2:

~~~

for (i, c) in enumerate(contour):
    # Code inside the loop will execute for each contour detected in the frame.
```
~~~

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

~~~

(x, y, w, h) = cv2.boundingRect(c)
```
~~~

- `cv2.boundingRect(c)`: Calculates the bounding rectangle for the contour.

10. Output Generation:

````

cv2.line(frame1, (25, count_line_position), (1200, count_line_position), (255, 255, 255), 3)
cv2.rectangle(frame1, (x, y), (x + w, y + h), (0, 255, 0), 2)
cv2.putText(frame1, "Vehicle" + str(counter), (x, y - 20), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 0, 0), 2)
````

```
cv2.circle(frame1, counter1, 4, (0, 0, 255), -1)
cv2.putText(frame1, "Vehicle Counter:" + str(counter), (450, 70), cv2.FONT_HERSHEY_COMPLEX, 2, (0, 0, 255), 5)
```

- Drawing lines, rectangles, text, and circles on the frame for visualization.

11. Error Handling:

```

# No explicit error handling in this code.
```

Conclusion:

This code performs vehicle counting using background subtraction and contour analysis. It reads frames from a video file, processes them to detect vehicles, and displays the count on the frame. Additionally, it connects to a PostgreSQL database and inserts the vehicle count into a table.

File: vehicle.py
Documented code for vehicle.py:
Code Explanation:

1. Import Statements:

```

import cv2
import numpy as np
import psycopg2
```

- We import necessary libraries for computer vision (cv2), numerical operations (numpy), and database connectivity (psycopg2).

2. Variable Declarations:

~~~
cap = cv2.VideoCapture('video.mp4')
min_width_rect = 80
min_height_rect = 80
count_line_position = 550
```
~~~

- `cap`: VideoCapture object to read video frames from a file named 'video.mp4'.
- `min_width_rect`, `min_height_rect`: Minimum width and height thresholds for detected rectangles.
- `count_line_position`: Position of the line used for vehicle counting.

3. User Input Handling:

~~~
# No user input is handled in this code.
```
~~~

4. Control Flow - Part 1:

~~~
while True:
    # Code inside this loop will continuously run until a break statement is encountered.
```
~~~

5. Function Calls - Part 1:

~~~
ret, frame1 = cap.read()
```
~~~

- `cap.read()`: Reads the next frame from the video capture object and returns a boolean indicating success (ret) and the captured frame (frame1).

6. Looping Structures:

```
while True:
    # Code inside this loop will continuously run until a break statement is encountered.
```

7. Data Manipulation:

```
grey = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(grey, (3, 3), 5)
img_sub = algo.apply(blur)
dilat = cv2.dilate(img_sub, np.ones((5, 5)))
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
dilate = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel)
dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)
```

- We perform various image processing operations to detect moving objects:
  - Convert the frame to grayscale.
  - Apply Gaussian blur for noise reduction.
  - Apply background subtraction to identify moving objects.
  - Dilate and erode the image to remove noise and fill gaps.

8. Control Flow - Part 2:

```
for (i, c) in enumerate(contour):
    # Code inside this loop iterates through the contours (detected objects) in the frame.
```

~~~~
```
~~~~

9. Function Calls - Part 2:

~~~~
(x, y, w, h) = cv2.boundingRect(c)
```
~~~~

- `cv2.boundingRect(c)`: Calculates the bounding rectangle for the contour `c`.

10. Output Generation:

~~~~
cv2.rectangle(frame1, (x, y), (x + w, y + h), (0, 255, 0), 2)
cv2.putText(frame1, "Vehicle" + str(counter), (x, y - 20), cv2.FONT_HERSHEY_COMPLEX, 1, (255,
0, 0), 2)
```
~~~~

- We draw a green rectangle around detected vehicles and display the vehicle count.

11. Error Handling:

~~~~
# No error handling is implemented in this code.
```
~~~~

Conclusion:

This code uses computer vision techniques to detect and count vehicles in a video. It continuously reads frames from a video file, processes them to identify moving objects, and draws rectangles around detected vehicles. The vehicle count is displayed on the frame and updated whenever a vehicle crosses a predefined line. Additionally, the vehicle count is stored in a database. This code demonstrates basic image processing and object detection concepts.

File: directkeys.py

Documented code for directkeys.py:

1. Import Statements:

```
import directkeys
import keyboard
import time
```

* The `directkeys` module is imported to enable direct keypresses on the keyboard.
* The `keyboard` module is imported to capture keypresses from the keyboard.
* The `time` module is imported to handle time-related operations.

2. Function Definition:

```
def press_keys(key_sequence):
    for key in key_sequence:
        directkeys.press(key)
        time.sleep(0.1)
        directkeys.release(key)
```

* The `press_keys` function is defined to press a sequence of keys one after the other.
* The function takes a `key_sequence` as an argument, which is a list of keys to be pressed.
* The function uses the `directkeys` module to press each key in the sequence, waits for 0.1 seconds, and then releases the key.

3. Keypress Capture:

~~~
while True:
    key = keyboard.read_key()
    if key == 'esc':
        break
    elif key == 'a':
        press_keys(['a', 's', 'd', 'f', 'j', 'k', 'l', ';'])
    elif key == 's':
        press_keys(['s', 'd', 'f', 'j', 'k', 'l', ';', 'a'])
```
~~~

* The `while True` loop is used to continuously capture keypresses from the keyboard.
* The `keyboard.read_key()` function is used to capture the keypress.
* If the keypress is `esc`, the loop is broken and the program exits.
* If the keypress is `a`, the `press_keys` function is called with a sequence of keys to be pressed.
* If the keypress is `s`, the `press_keys` function is called with a different sequence of keys to be pressed.

4. Keypress Simulation:

~~~

press_keys(['a', 's', 'd', 'f', 'j', 'k', 'l', ';'])
```
~~~

* This line of code simulates a sequence of keypresses.
* The keys are pressed one after the other with a delay of 0.1 seconds between each keypress.

**5. Program Termination:**

~~~

break
```
~~~

* The `break` statement is used to terminate the `while True` loop and exit the program.

6. Release Captured Resources:

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

* The `cap.release()` function is used to release the captured video resource.
* The `cv2.destroyAllWindows()` function is used to destroy all open windows.

Summary:

The provided code demonstrates how to simulate keypresses on the keyboard using the `directkeys` and `keyboard` modules. The program captures keypresses from the keyboard and simulates a sequence of keypresses based on the captured keypress. The program exits when the `esc` key is pressed.

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

1. Import Statements:

```
import cv2
import numpy as np
import psycopg2
```

- We import necessary libraries for computer vision (cv2), numerical operations (numpy), and database connectivity (psycopg2).

2. Variable Declarations:

```
cap = cv2.VideoCapture('video.mp4')
min_width_rect = 80
min_height_rect = 80
count_line_position = 550
```

- `cap`: VideoCapture object to read video frames from a file named 'video.mp4'.
- `min_width_rect`, `min_height_rect`: Minimum width and height thresholds for detected rectangles.
- `count_line_position`: Position of the line used for vehicle counting.

3. User Input Handling:

No user input is handled in this code.

4. Control Flow - Part 1:

```
algo = cv2.bgsegm.createBackgroundSubtractorMOG()
```

- We initialize a background subtractor algorithm (`algo`) to separate moving objects from the background.

5. Function Calls - Part 1:

```
def center(x, y, w, h):
    x1 = int(w / 2)
    y1 = int(h / 2)
    cx = x + x1
```

~~~
    cy = y + y1
    return cx, cy
```
~~~

- We define a `center()` function to calculate the center of a rectangle given its coordinates and dimensions.

6. Looping Structures:

~~~

while True:
    # Video frame processing and vehicle detection logic
```
~~~

- We enter an infinite loop to continuously process video frames.

**7. Data Manipulation:**

~~~

ret, frame1 = cap.read()
grey = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(grey, (3, 3), 5)
img_sub = algo.apply(blur)
dilat = cv2.dilate(img_sub, np.ones((5, 5)))
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
dilate = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel)
dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)
```
~~~

- We read a frame from the video (`frame1`), convert it to grayscale (`grey`), apply Gaussian blur (`blur`), perform background subtraction (`img_sub`), dilate the image (`dilat`), and apply morphological operations (`dilate`) to enhance the detected objects.

8. Control Flow - Part 2:

```
contour, h = cv2.findContours(dilate, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

- We find contours in the processed image (`dilate`) to identify potential vehicles.

9. Function Calls - Part 2:

```
for (i, c) in enumerate(contour):
    (x, y, w, h) = cv2.boundingRect(c)
    validate_counter = (w >= min_width_rect) and (h >= min_height_rect)
    if not validate_counter:
        continue
```

- We iterate through the contours, calculate bounding rectangles for each contour, and filter out small objects based on minimum width and height thresholds.

10. Output Generation:

```
cv2.line(frame1, (25, count_line_position), (1200, count_line_position), (255, 255, 255), 3)
cv2.putText(frame1, "Vehicle Counter:" + str(counter), (450, 70), cv2.FONT_HERSHEY_COMPLEX,
2, (0, 0, 255), 5)
cv2.imshow("frame", frame1)
```

- We draw a line at the counting position, display the vehicle count, and show the processed frame.

11. Error Handling:

No specific error handling mechanisms are implemented in this code.

Conclusion:

This code demonstrates a vehicle counting system using background subtraction, contour detection, and line crossing detection. It reads frames from a video file, processes them to detect moving objects, and counts vehicles that cross a predefined line. The vehicle count is displayed on the frame and can be stored in a database.

File: steering.py

Documented code for steering.py:

1. Import Statements:

```
import cv2
import numpy as np
import psycopg2
```
```

- `cv2`: OpenCV library for computer vision tasks.
- `numpy`: NumPy library for numerical operations.
- `psycopg2`: Psycopg2 library for connecting to PostgreSQL databases.

2. Variable Declarations:

```
cap = cv2.VideoCapture('video.mp4')
min_width_rect = 80
min_height_rect = 80
count_line_position = 550
algo = cv2.bgsegm.createBackgroundSubtractorMOG()
detect = []
```
```

~~~~
offset = 6
counter = 0
```
~~~~

- `cap`: VideoCapture object to read the input video file.
- `min_width_rect`, `min_height_rect`: Minimum width and height thresholds for detected rectangles.
- `count_line_position`: Position of the counting line.
- `algo`: Background subtractor object for motion detection.
- `detect`: List to store the detected vehicles' center coordinates.
- `offset`: Allowed error margin for counting vehicles crossing the line.
- `counter`: Counter for the number of vehicles that have crossed the line.

3. User Input Handling:

There is no user input handling in this code.

4. Control Flow - Part 1:

~~~~

while True:
    ret, frame1 = cap.read()
    # ...
    if cv2.waitKey(0) == 13:
        break
```
~~~~

- The code enters an infinite loop to continuously process video frames.
- It reads each frame from the video using `cap.read()`.
- The loop continues until the user presses the 'Enter' key, which breaks the loop.

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

```
grey = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(grey, (3, 3), 5)
img_sub = algo.apply(blur)
dilat = cv2.dilate(img_sub, np.ones((5, 5)))
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
dilate = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel)
dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)
contour, h = cv2.findContours(dilate, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

- `cv2.cvtColor()`: Converts the frame to grayscale.
- `cv2.GaussianBlur()`: Applies Gaussian blur to reduce noise.
- `algo.apply()`: Applies background subtraction to detect moving objects.
- `cv2.dilate()`: Dilates the binary image to fill gaps in the detected objects.
- `cv2.morphologyEx()`: Performs morphological operations to remove noise and improve object shapes.
- `cv2.findContours()`: Finds the contours of the detected objects.

6. Looping Structures:

```
for (i, c) in enumerate(contour):
    # ...
for (x, y) in detect:
    # ...
```

- The first loop iterates through the detected contours.
- The second loop iterates through the detected vehicles' center coordinates.

7. Data Manipulation:

```
(x, y, w, h) = cv2.boundingRect(c)
validate_counter = (w >= min_width_rect) and (h >= min_height_rect)
```

- `cv2.boundingRect()`: Calculates the bounding rectangle for each contour.
- `validate_counter`: Checks if the detected object meets the minimum width and height requirements.

8. Control Flow - Part 2:

```

if not validate_counter:
    continue
```

- If the detected object does not meet the minimum size requirements, it is ignored.

9. Function Calls - Part 2:

```

cv2.rectangle(frame1, (x, y), (x + w, y + h), (0, 255, 0), 2)
cv2.putText(frame1, "Vehicle" + str(counter), (x, y - 20), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 0, 0), 2)
counter1 = center(x, y, w, h)
detect.append(counter1)
cv2.circle(frame1, counter1, 4, (0, 0, 255), -1)
```

- `cv2.rectangle()`: Draws a green rectangle around the detected vehicle.
- `cv2.putText()`: Draws the vehicle count text on the frame.
- `center()`: Calculates the center coordinates of the detected vehicle.
- `detect.append()`: Adds the vehicle's center coordinates to the `detect` list.
- `cv2.circle()`: Draws a blue circle at the vehicle's center.

10. Output Generation:

~~~
cv2.line(frame1, (25, count_line_position), (1200, count_line_position), (255, 255, 255), 3)
cv2.putText(frame1, "Vehicle Counter:" + str(counter), (450, 70), cv2.FONT_HERSHEY_COMPLEX, 2, (0, 0, 255), 5)
cv2.imshow("frame", frame1)
```
~~~

- `cv2.line()`: Draws a white line indicating the counting line.
- `cv2.putText()`: Draws the vehicle count text on the frame.
- `cv2.imshow()`: Displays the processed frame.

11. Error Handling:

There is no explicit error handling in this code.

Conclusion:

This code uses OpenCV for vehicle detection and counting in a video. It subtracts the background, detects moving objects, filters out small objects, and draws rectangles and circles around detected vehicles. When a vehicle crosses the counting line, its count is incremented and displayed on the frame. The vehicle count is also stored in a PostgreSQL database.

Folder: __pycache__
File: centroidtracker.py
Documented code for centroidtracker.py:
Code Explanation:

1. Import Statements:

```
import cv2
import numpy as np
import psycopg2
```

- `cv2`: OpenCV library for computer vision tasks.
- `numpy`: NumPy library for numerical operations.
- `psycopg2`: Psycopg2 library for connecting to PostgreSQL databases.

2. Variable Declarations:

```
cap = cv2.VideoCapture('video.mp4')
min_width_rect = 80
min_height_rect = 80
count_line_position = 550
```

- `cap`: VideoCapture object to read video frames from a video file.
- `min_width_rect` and `min_height_rect`: Minimum width and height thresholds for detected rectangles.
- `count_line_position`: Position of the counting line.

3. User Input Handling:

```
# No user input handling in this code.
```

4. Control Flow - Part 1:

```
while True:
```

~~~~
    # Code inside the loop will execute continuously until 'break' is encountered.
```
~~~~

5. Function Calls - Part 1:

~~~~
ret, frame1 = cap.read()
```
~~~~

- `cap.read()`: Reads the next frame from the video file and returns a boolean `ret` indicating success and the frame `frame1`.

6. Looping Structures:

~~~~
while True:
    # Code inside the loop will execute continuously until 'break' is encountered.
```
~~~~

7. Data Manipulation:

~~~~
grey = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(grey, (3, 3), 5)
img_sub = algo.apply(blur)
dilat = cv2.dilate(img_sub, np.ones((5, 5)))
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
dilate = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel)
dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)
```
~~~~

- Converting the frame to grayscale, applying Gaussian blur, background subtraction, dilation, and morphological operations to enhance vehicle detection.

8. Control Flow - Part 2:

```
for (i, c) in enumerate(contour):
    # Loop through each contour detected in the frame.
```

9. Function Calls - Part 2:

```
(x, y, w, h) = cv2.boundingRect(c)
```

- `cv2.boundingRect(c)`: Calculates the bounding rectangle for the contour `c`.

10. Output Generation:

```
cv2.line(frame1, (25, count_line_position), (1200, count_line_position), (255, 255, 255), 3)
cv2.putText(frame1, "Vehicle Counter:" + str(counter), (450, 70), cv2.FONT_HERSHEY_COMPLEX,
2, (0, 0, 255), 5)
cv2.imshow("frame", frame1)
```

- Drawing the counting line, displaying the vehicle count, and showing the processed frame.

11. Error Handling:

```
if cv2.waitKey(0) == 13:
    break
```

- Waiting for a key press (in this case, the 'Enter' key) to break out of the loop and stop the program.

Conclusion:

This code uses OpenCV for vehicle detection and counting in a video. It applies background subtraction, morphological operations, and contour analysis to identify vehicles and count them as they cross a predefined counting line. The vehicle count is displayed on the frame and can be stored in a database.

File: final.py

Documented code for final.py:

Code Explanation:

1. Import Statements:

```
import cv2
import numpy as np
import psycopg2
```

- We import necessary libraries:
  - `cv2`: OpenCV library for computer vision tasks.
  - `numpy`: NumPy library for numerical operations.
  - `psycopg2`: Psycopg2 library for connecting to PostgreSQL databases.

2. Variable Declarations:

```
cap = cv2.VideoCapture('video.mp4')
min_width_rect = 80
min_height_rect = 80
```

```
count_line_position = 550
```

- `cap`: VideoCapture object to read video frames from a video file named 'video.mp4'.
- `min_width_rect`, `min_height_rect`: Minimum width and height thresholds for detecting vehicles.
- `count_line_position`: Position of the counting line on the video frame.

3. User Input Handling:

```

# No user input handling in this code.
```

- There is no user input handling in this code.

4. Control Flow - Part 1:

```

algo = cv2.bgsegm.createBackgroundSubtractorMOG()
```

- We initialize a background subtractor algorithm (`algo`) using OpenCV's MOG (Mixture of Gaussians) method.

5. Function Calls - Part 1:

```

def center(x, y, w, h):
    x1 = int(w / 2)
    y1 = int(h / 2)
    cx = x + x1
    cy = y + y1
    return cx, cy
```

~~~
```
~~~

- We define a helper function `center` to calculate the center of a rectangle given its coordinates and dimensions.

6. Looping Structures:

~~~

while True:
    # Read a frame from the video capture object
    ret, frame1 = cap.read()

    # Preprocess the frame for vehicle detection
    grey = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(grey, (3, 3), 5)
    img_sub = algo.apply(blur)
    dilat = cv2.dilate(img_sub, np.ones((5, 5)))
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
    dilate = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel)
    dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)

    # Find contours in the preprocessed frame
    contour, h = cv2.findContours(dilate, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```
~~~

- We enter an infinite loop to continuously process video frames.
- We read each frame using `cap.read()`.
- We preprocess the frame by converting it to grayscale, applying Gaussian blur, background subtraction, dilation, and morphological operations to enhance vehicle detection.
- We find contours in the preprocessed frame using OpenCV's contour detection algorithm.

**7. Data Manipulation:**

```
detect = []
offset = 6
counter = 0
```

- We initialize an empty list `detect` to store the detected vehicles' centers.
- We define an `offset` variable to allow a small margin of error when vehicles cross the counting line.
- We initialize a counter variable `counter` to keep track of the number of vehicles that have crossed the line.

8. Control Flow - Part 2:

```
cv2.line(frame1, (25, count_line_position), (1200, count_line_position), (255, 255, 255), 3)
```

- We draw a horizontal line on the frame to indicate the counting line.

9. Function Calls - Part 2:

```
for (i, c) in enumerate(contour):
    (x, y, w, h) = cv2.boundingRect(c)
    validate_counter = (w >= min_width_rect) and (h >= min_height_rect)
    if not validate_counter:
        continue

    cv2.rectangle(frame1, (x, y), (x + w, y + h), (0, 255, 0), 2)
    cv2.putText(frame1, "Vehicle" + str(counter), (x, y - 20), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 0, 0), 2)
    counter1 = center(x, y, w, h)
    detect.append(counter1)
    cv2.circle(frame1, counter1, 4, (0, 0, 255), -1)
```

- We iterate through the detected contours.
- For each contour, we extract its bounding rectangle coordinates and dimensions.
- We check if the contour meets the minimum width and height requirements to be considered a vehicle.
- If it's a valid vehicle, we draw a green rectangle around it, display the vehicle count, and store its center in the `detect` list.

10. Output Generation:

```

for (x, y) in detect:
    if y < (count_line_position + offset) and y > (count_line_position - offset):
        counter += 1
        cv2.line(frame1, (25, count_line_position), (1200, count_line_position), (0, 0, 255), 3)
        detect.remove((x, y))
        count = str(counter)
        print("Vehicle Counting:" + str(counter))
```

- We iterate through the detected vehicle centers.
- If a vehicle's center is within the `offset` range of the counting line, we increment the counter, draw a red line to indicate the vehicle crossing the line, remove the vehicle's center from the `detect` list, and print the updated vehicle count.

11. Error Handling:

```

# No error handling in this code.
```

~~~~
```
~~~~

- There is no explicit error handling in this code.

Conclusion:

This Python script uses OpenCV for vehicle detection and counting in a video. It preprocesses video frames, detects vehicles using background subtraction and contour analysis, and counts vehicles crossing a predefined line. The vehicle count is displayed on the frame and printed to the console. Additionally, the script can insert the vehicle count into a PostgreSQL database using the Psycopg2 library.

File: distance.py

Documented code for distance.py:

Line 1:

~~~~
import cv2
import numpy as np
```
~~~~

This code imports the OpenCV and NumPy libraries. OpenCV is a library for computer vision and image processing, while NumPy is a library for scientific computing.

Line 3:

~~~~
cap = cv2.VideoCapture('video.mp4')
```
~~~~

This code creates a VideoCapture object that can be used to capture video from a webcam or a video file. The 'video.mp4' argument specifies the path to the video file that we want to capture.

Line 5:

~~~
min_width_rect = 20
min_height_rect = 20
```
~~~

These lines define the minimum width and height of a rectangle that will be used to detect objects in the video.

Line 7:

~~~
count_line_position = 500
```
~~~

This line defines the position of a horizontal line in the video frame. This line will be used to count the number of objects that cross it.

Line 9:

~~~
def center_coordinates(x, y):
    int_x = int(x)
    int_y = int(y)
    cx = x - int_x
    cy = y - int_y
    return int_x, int_y
```
~~~

This code defines a function that takes two arguments, x and y, and returns the integer values of x and y. This function is used to calculate the center coordinates of an object.

Line 15:

```
count = []
offset = 50 # allowing error between pixel
counter = 0
```

These lines define a list called 'count' to store the number of objects that cross the line, an offset of 50 pixels to allow for some error in the detection, and a counter variable to keep track of the number of objects.

Line 19:

```
while True:
```

This line starts a while loop that will continue until the loop is broken.

Line 20:

```
ret, frame = cap.read()
```

This line reads the next frame from the video capture object. The 'ret' variable is a boolean that indicates whether a frame was successfully read. The 'frame' variable is a NumPy array that contains the image data of the frame.

Line 22:

~~~
gray = cv2. cvtColor(frame, cv2.COLOR_BGR2GRAY)
```
~~~

This line converts the frame from BGR (blue, green, red) color space to grayscale.

Line 24:

~~~

blur = cv2.GaussianBlur(gray, (5, 5), 0)
```
~~~

This line applies a Gaussian blur to the grayscale image. This helps to reduce noise and make it easier to detect objects.

**Line 26:**

~~~

algo = cv2.createBackgroundSubtractor()
```
~~~

This line creates a BackgroundSubtractor object. This object is used to extract the background from the image.

Line 28:

~~~

img_sub = algo.apply(blur)
```
~~~

This line applies the background subtractor to the blurred image. The result is an image that contains only the objects in the frame.

**Line 30:**

```
dilate = cv2.dilate(img_sub, np.ones((5, 5)))
```

This line dilates the image, which expands the objects in the image. This helps to make them easier to detect.

Line 32:

```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
```

This line creates a kernel that is used for morphological operations. The kernel is a 5x5 square kernel with a cross-shaped structure.

Line 34:

```
dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)
```

This line performs a morphological closing operation on the image. This operation fills any holes in the objects in the image.

Line 36:

```
dilate = cv2.morphologyEx(dilate, cv2.MORPH_CLOSE, kernel)
```

This line performs a second morphological closing operation on the image. This helps to make the objects in the image more solid.

Line 38:

```
contours, _ = cv2.findContours(dilate, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

This line finds the contours of the objects in the image. The 'contours' variable is a list of NumPy arrays, where each array contains the points that make up the contour of an object.

Line 40:

```
for (x, y, w, h) in contours:
```

This line iterates through the contours of the objects in the image.

Line 42:

```
rect = cv2.Rect(x, y, w, h)
```

This line creates a rectangle object that represents the object in the image. The rectangle is defined by its x and y coordinates, its width, and its height.

Line 44:

```
validate_counter = (w >= min_width_rect and h >= min_height_rect)
```

This line checks if the object is larger than the minimum width and height of a rectangle. This is used to filter out small objects that are not likely to be vehicles.

Line 46:

```
if not validate_counter:
    continue
```

This line continues the loop if the object is not larger than the minimum width and height of a rectangle.

Line 48:

```
cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

This line draws a rectangle around the object in the image. The rectangle is drawn with a green color and a thickness of 2 pixels.

Line 50:

```
cv2.putText(frame, "Count: " + str(counter), (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
```

This line adds a label to the image that shows the number of objects that have crossed the line. The label is drawn with a green color and a thickness of 2 pixels.

Line 52:

~~~
counter += center_coordinates(x + w/2, y + h/2)
```
~~~

This line calculates the center coordinates of the object and adds them to the counter variable.

Line 54:

~~~
count.append(counter)
```
~~~

This line appends the counter variable to the 'count' list.

Line 56:

~~~
cv2.circle(frame, (x + w/2, y + h/2), 5, (0, 255, 0), 2)
```
~~~

This line draws a circle at the center of the object. The circle is drawn with a green color and a thickness of 2 pixels.

Line 58:

~~~
for i in detect:
```
~~~

This line iterates through the list of detected objects.

Line 60:

~~~~
if y > count_line_position - offset and y < (count_line_position + offset):
```
~~~~

This line checks if the object is within the offset range of the line.

Line 62:

~~~~
counter += 1
```
~~~~

This line increments the counter variable.

Line 64:

~~~~
cv2.line(frame, (x, count_line_position), (x + w, count_line_position), (0, 255, 0), 2)
```
~~~~

This line draws a line from the object to the line. The line is drawn with a green color and a thickness of 2 pixels.

Line 66:

~~~~
detect.remove((x, y))
```
~~~~

This line removes the object from the list of detected objects.

Line 68:

~~~
count_str = str(counter)
```
~~~

This line converts the counter variable to a string.

Line 70:

~~~
print("Count: ", count_str)
```
~~~

This line prints the number of objects that have crossed the line.

Line 72:

~~~
cv2.putText(frame, "Count: " + count_str, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255,
0), 2)
```
~~~

This line adds a label to the image that shows the number of objects that have crossed the line. The label is drawn with a green color and a thickness of 2 pixels.

Line 74:

~~~
if cv2.waitKey(1) == 27:
    break
```
~~~

This line checks