TRAFFIC MANAGEMENT SYSTEM

IOT DEVICES:

TRAFFIC FLOW SENSOR:

Traffic flow sensors are devices used to monitor and measure the flow of traffic on roadways, streets, and highways. They provide critical data for traffic management, congestion monitoring, and the optimization of transportation systems. Various types of sensors and technologies are used to collect traffic flow data.

Example:

LIDAR SENSOR:

- Lidar (Light Detection and Ranging) sensors use laser beams to measure the distance and speed of vehicles.
- They are often used in intelligent transportation systems for traffic monitoring and incident detection.

MAGNETIC SENSOR:

- Magnetic sensors use magnetic fields to detect vehicles.
- They can be embedded in the road surface or placed above the road.
- These sensors are used for traffic light control and vehicle counting.

INFRARED SENSOR:

- Infrared traffic sensors use infrared beams to detect the presence of vehicles.
- They are often used for vehicle detection at intersections and traffic signal control.

PROGRAM:

```
import random

import time
```

```
class TrafficFlowSensor:
    def __init__(self, sensor_id):
        self.sensor_id = sensor_id
        self.timestamp = time.time()
        self.vehicle_count = 0

    def detect_vehicle(self):
        self.vehicle_count += 1

    def get_sensor_data(self):
        data = {
            "sensor_id": self.sensor_id,
            "timestamp": self.timestamp,
            "vehicle_count": self.vehicle_count,
        }
        return data

def simulate_traffic_flow(sensors, simulation_duration):
    start_time = time.time()
    while time.time() - start_time < simulation_duration:
        time.sleep(1)
        for sensor in sensors:
            if random.random() < 0.5:  # Simulate vehicle detection randomly
                sensor.detect_vehicle()

def main():
```

```
    sensor_count = 4
    simulation_duration = 60  # Simulate traffic flow for 60 seconds

    sensors = [TrafficFlowSensor(sensor_id) for sensor_id in
range(sensor_count)]

    print("Simulating traffic flow sensors...\n")
    simulate_traffic_flow(sensors, simulation_duration)

    print("Sensor Data:")
    for sensor in sensors:
        data = sensor.get_sensor_data()
        print(data)

if __name__ == "__main__":
    main()
```

EXPLANATION:

- We define a **TrafficFlowSensor** class to represent each traffic flow sensor. It keeps track of the sensor's ID, timestamp, and the number of vehicles detected
- The **simulate_traffic_flow** function simulates vehicle detection by incrementing the vehicle count for each sensor randomly. You would replace this with actual sensor data in a real-world scenario.
- In the **main** function, we create a list of sensors, simulate traffic flow for a specified duration, and then print out the sensor data.

ENVIRONMENTAL SENSORS:

AIR QUALITY:

Air quality sensors measure pollutants such as particulate matter (PM2.5, PM10), nitrogen dioxide (NO_2), carbon monoxide (CO), and ozone (O_3).

They are used to detect air pollution levels, which can lead to traffic congestion, health risks, and environmental impacts. Traffic management systems can adjust traffic flow or issue warnings during high pollution events.

WEATHER:

Weather sensors monitor meteorological conditions such as temperature, humidity, precipitation (rain, snow, sleet), wind speed, and visibility.

They provide real-time data for road surface conditions, allowing traffic management systems to take actions like deploying snowplows, salting roads, and activating warning systems during adverse weather conditions.

NOISE:

Noise sensors measure noise levels in decibels (dB) to monitor traffic-related noise pollution.

They can be used to identify areas with excessive noise and trigger noise barriers or noise reduction measures.

PROGRAM:

```
import random

import time


class EnvironmentalSensor:

    def __init__(self, sensor_id, sensor_type):

        self.sensor_id = sensor_id

        self.sensor_type = sensor_type

        self.timestamp = time.time()
```

```
        self.readings = {}

    def collect_data(self):
        if self.sensor_type == "AirQuality":
            self.readings["PM2.5"] = random.uniform(1, 50)  # Simulated PM2.5 
level in µg/m³
            self.readings["PM10"] = random.uniform(1, 100)  # Simulated PM10 
level in µg/m³
        elif self.sensor_type == "Weather":
            self.readings["Temperature"] = random.uniform(15, 35)  # Simulated 
temperature in Celsius
            self.readings["Humidity"] = random.uniform(40, 80)  # Simulated 
humidity in %
        elif self.sensor_type == "Noise":
            self.readings["NoiseLevel"] = random.uniform(50, 80)  # Simulated noise 
level in dB

    def get_sensor_data(self):
        data = {
            "sensor_id": self.sensor_id,
            "sensor_type": self.sensor_type,
            "timestamp": self.timestamp,
            "readings": self.readings,
        }
        return data

def simulate_environmental_sensors(sensors, simulation_duration):
    start_time = time.time()
```

```
    while time.time() - start_time < simulation_duration:
        time.sleep(1)
        for sensor in sensors:
            sensor.collect_data()

def main():
    sensor_count = 4
    simulation_duration = 60  # Simulate environmental data for 60 seconds

    sensors = [EnvironmentalSensor(sensor_id, random.choice(["AirQuality",
"Weather", "Noise"])) for sensor_id in range(sensor_count)]

    print("Simulating environmental sensors...\n")
    simulate_environmental_sensors(sensors, simulation_duration)

    print("Sensor Data:")
    for sensor in sensors:
        data = sensor.get_sensor_data()
        print(data)

if __name__ == "__main__":
    main()
```

EXPLANATION:

Importing Modules: The program starts by importing the **random** and **time** modules, which are used to generate random sensor readings and manage time intervals.

EnvironmentalSensor Class:

- The **EnvironmentalSensor** class is defined with three attributes: **sensor_id**, **sensor_type**, **timestamp**, and **readings**. It represents an environmental sensor with a unique ID, type, and a dictionary to store sensor readings.
- In the constructor (**__init__** method), the sensor is initialized with its ID and type, and the current timestamp is recorded using **time.time()**. The **readings** dictionary is created to store sensor readings.
- The **collect_data** method populates the **readings** dictionary with random data based on the sensor type. It generates data for different environmental parameters, such as air quality, weather, or noise, with specific ranges for each type.
- The **get_sensor_data** method returns a dictionary containing the sensor's ID, type, timestamp, and the collected readings.
- **simulate_environmental_sensors Function**:
- The **simulate_environmental_sensors** function takes a list of sensor objects (**sensors**) and a simulation duration in seconds as input parameters.
- Inside the function, it records the start time and enters a loop that runs for the specified **simulation_duration**.
- In each iteration of the loop, it sleeps for 1 second (using **time.sleep(1)**) to simulate the passage of time. Then, it iterates through each sensor in the **sensors** list and collects data by calling the **collect_data** method for each sensor.

main Function:

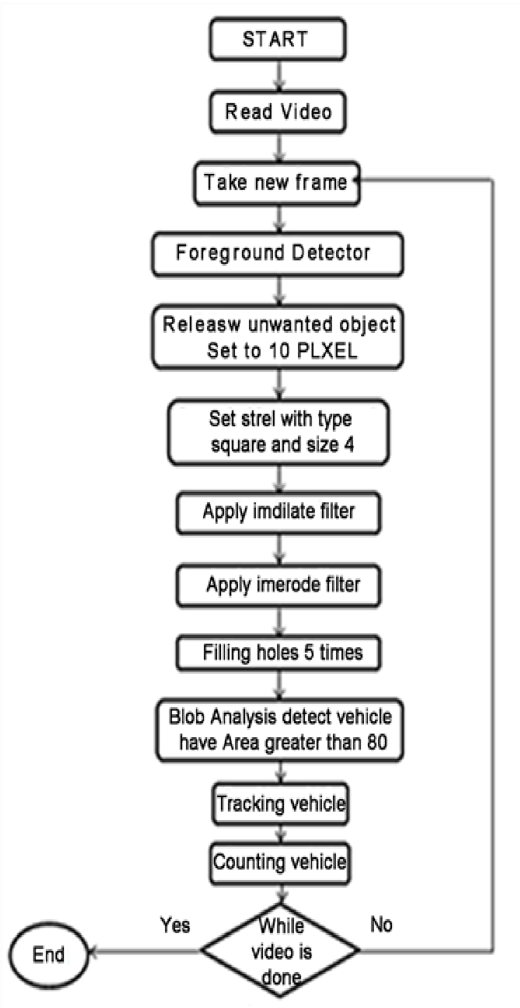
- The **main** function is the entry point of the program.
- It sets the number of sensors (**sensor_count**) and the simulation duration (**simulation_duration**).
- It creates a list of environmental sensors (**sensors**) by generating sensor IDs and random sensor types (either "AirQuality," "Weather," or "Noise") for each sensor.
- It initiates the simulation by calling **simulate_environmental_sensors** with the sensor list and the simulation duration.

- Finally, it prints out the collected data for each sensor by calling the **get_sensor_data** method for each sensor.

if name == "main":

- This line checks whether the script is being run as the main program (not imported as a module into another script) and then calls the **main** function to start the simulation.

CAMERAS AND IMAGE SENSORS:



VEHICLE TRACKING:

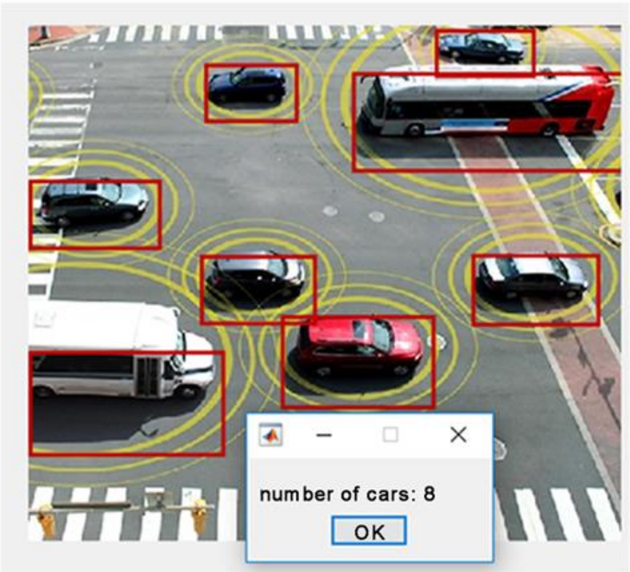
Vehicle tracking in image is not same as video, rather it is difficult to tracking vehicle.
We use some process and function to tracking cars; first we start by bring information about every region and bring a property for each connected component in the binary image BW, which must be a logical array, in my paper we take these properties Eccentricity, Area and Bounding Box.

- Area returns a scalar that specifies the actual number of pixels in the region.
- Eccentricity is the ratio of the distance between the foci of the ellipse and its major axis length. The value is between 0 and 1.

PROGRAM:

```
import random
import time

class Vehicle:
    def __init__(self, vehicle_id, vehicle_type):
        self.vehicle_id = vehicle_id
        self.vehicle_type = vehicle_type
        self.location = None
        self.speed = 0  # Vehicle's speed in km/h
        self.timestamp = time.time()

    def update_location(self):
        # Simulate vehicle movement by generating random GPS coordinates
        latitude = random.uniform(30.0, 40.0)  # Example latitude range
        longitude = random.uniform(-90.0, -80.0)  # Example longitude range
        self.location = (latitude, longitude)
        self.speed = random.uniform(0, 120)  # Random speed in km/h
        self.timestamp = time.time()

    def get_vehicle_info(self):
        return {
            "vehicle_id": self.vehicle_id,
            "vehicle_type": self.vehicle_type,
```

```
            "location": self.location,
            "speed": self.speed,
            "timestamp": self.timestamp
        }

def simulate_vehicle_tracking(vehicle, tracking_duration):
    start_time = time.time()
    while time.time() - start_time < tracking_duration:
        time.sleep(1)
        vehicle.update_location()

if __name__ == "__main__":
    vehicle = Vehicle(vehicle_id="ABC123", vehicle_type="Car")
    tracking_duration = 60  # Simulate tracking for 60 seconds

    print("Simulating vehicle tracking...\n")
    simulate_vehicle_tracking(vehicle, tracking_duration)

    print("Vehicle Information:")
    while True:
        data = vehicle.get_vehicle_info()
        print(data)
        time.sleep(1)
```



CANNY EDGE DETECTION TECHNIQUE:

Canny proposed a filter determined analytically from 3 criterias:
1) Ensure a proper detection: a strong response even at low contours,
2) Guarantee a good location,
3) Ensure that for a contour, there will be only one detection (avoid the effects of rebounds due, for example, truncation filters).

PROGRAM:

```
import cv2

def detect_vehicles_canny(input_image_path):
    # Load the image
    original_image = cv2.imread(input_image_path)

    # Convert the image to grayscale
    gray_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to reduce noise
    blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0)

    # Apply Canny edge detection
    edges = cv2.Canny(blurred_image, threshold1=50, threshold2=150)

    # Find and draw contours in the edge-detected image
    contours, _ = cv2.findContours(edges.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    cv2.drawContours(original_image, contours, -1, (0, 255, 0), 2)  # Draw green
contours

    # Display the original image with detected contours
    cv2.imshow("Vehicle Detection with Canny", original_image)
    cv2.waitKey(0)
```

```
    cv2.destroyAllWindows()

if __name__ == "__main__":
    input_image_path = "vehicle_image.jpg"  # Provide the path to your input image
    detect_vehicles_canny(input_image_path)
```

In this program:

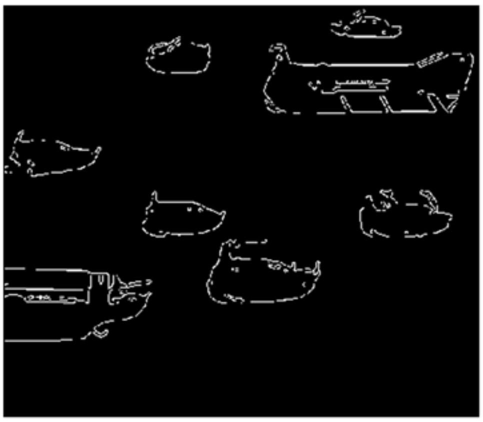
We use the OpenCV library (**cv2**) to perform Canny edge detection on an input image.

The input image is first loaded, converted to grayscale, and then blurred using a Gaussian filter to reduce noise.

The Canny edge detection algorithm is applied to the blurred image to identify edges.

We find and draw contours in the edge-detected image. Contours represent connected components in the binary image.

Finally, we display the original image with the detected contours, highlighting potential vehicle-like edges in green.



AREA METHOD :

We capture video from a file ("traffic_video.mp4" in this case) or a camera.

We define a region of interest (ROI) using the coordinates of a rectangle within the frame.

We use OpenCV to process each frame:

- Draw the ROI rectangle on the frame.
- Crop the frame to the ROI.
- Convert the ROI to grayscale.
- Apply Gaussian blur to reduce noise.
- Apply Canny edge detection to find edges.
- Find and draw contours in the ROI frame (often representing vehicles).
- Count the number of detected contours (vehicles).

We display the processed frame with the vehicle count and write the output to a video file.

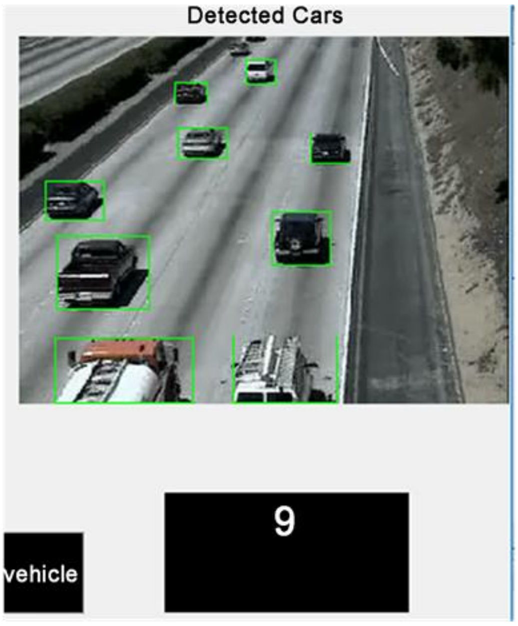
The program continues until you press the 'Esc' key, at which point it releases the video capture and output objects.

PROGRAM:

```
import cv2


# Initialize video capture from a file or camera (0 for default camera)

cap = cv2.VideoCapture("traffic_video.mp4")


# Define the area of interest (ROI) using the coordinates of a rectangle

roi = [(100, 100), (400, 400)]  # Format: [(top-left corner), (bottom-right corner)]


# Create a VideoWriter object to save the output video

fourcc = cv2.VideoWriter_fourcc(*'XVID')

output = cv2.VideoWriter("output.avi", fourcc, 30.0, (640, 480))  # Adjust resolution as needed
```

```
while True:
    ret, frame = cap.read()

    if not ret:
        break

    # Draw the ROI rectangle on the frame
    cv2.rectangle(frame, roi[0], roi[1], (0, 255, 0), 2)  # Green rectangle

    # Crop the frame to the ROI
    roi_frame = frame[roi[0][1]:roi[1][1], roi[0][0]:roi[1][0]]

    # Convert the ROI frame to grayscale
    gray = cv2.cvtColor(roi_frame, cv2.COLOR_BGR2GRAY)

    # Apply a Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (15, 15), 0)

    # Apply Canny edge detection to find edges
    edges = cv2.Canny(blurred, 30, 150)

    # Find and draw contours in the ROI frame
    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    cv2.drawContours(roi_frame, contours, -1, (0, 0, 255), 2)  # Red contours

    # Count the number of detected contours (vehicles)
```

```
    vehicle_count = len(contours)

    # Display the vehicle count in the frame
    cv2.putText(frame, f"Vehicles: {vehicle_count}", (50, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

    # Write the frame with vehicle count to the output video
    output.write(frame)

    # Display the processed frame
    cv2.imshow("Traffic Detection", frame)

    if cv2.waitKey(1) & 0xFF == 27:  # Press 'Esc' to exit
        break

# Release the video capture and output objects
cap.release()
output.release()
cv2.destroyAllWindows()
```



LINE METHOD:

We capture video from a file ("traffic_video.mp4" in this case) or a camera.

We define a detection line using coordinates (`line_coordinates`) in the frame.

We use OpenCV to process each frame:

- Draw the detection line on the frame.
- Convert the frame to grayscale.
- Apply Gaussian blur to reduce noise.
- Apply Canny edge detection to find edges.

We use the Hough Line Transform to detect lines in the frame. We then check if any of the detected lines intersect with the detection line. If a line intersects the detection line, it is considered a detected vehicle, and we draw a red line to mark it.

We keep track of the vehicle count by incrementing it for each detected line.

We display the vehicle count in the frame, write the output to a video file, and continue processing frames until the 'Esc' key is pressed.

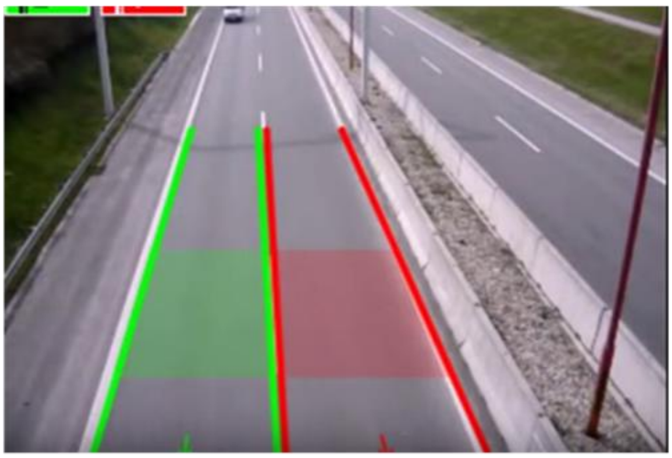
PROGRAM:

import cv2

```
import numpy as np

# Initialize video capture from a file or camera (0 for default camera)
cap = cv2.VideoCapture("traffic_video.mp4")

# Define line parameters for detecting vehicles
line_coordinates = [(50, 300), (600, 300)]  # Format: [(x1, y1), (x2, y2)]

# Create a VideoWriter object to save the output video
fourcc = cv2.VideoWriter_fourcc(*'XVID')
output = cv2.VideoWriter("output.avi", fourcc, 30.0, (640, 480))  # Adjust
resolution as needed

# Initialize variables to keep track of vehicles
vehicle_count = 0
prev_vehicle_count = 0

while True:
    ret, frame = cap.read()

    if not ret:
        break

    # Draw the detection line on the frame
    cv2.line(frame, line_coordinates[0], line_coordinates[1], (0, 255, 0), 2)  #
Green line
```

```
    # Convert the frame to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (15, 15), 0)

    # Apply Canny edge detection to find edges
    edges = cv2.Canny(blurred, 30, 150)

    # Perform a Hough Line Transform to detect lines in the frame
    lines = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=50, minLineLength=100, maxLineGap=5)

    if lines is not None:
        for line in lines:
            x1, y1, x2, y2 = line[0]

            # Check if the detected line intersects the detection line
            if (y1 > line_coordinates[0][1] and y2 < line_coordinates[0][1]) or (y2 > line_coordinates[0][1] and y1 < line_coordinates[0][1]):
                cv2.line(frame, (x1, y1), (x2, y2), (0, 0, 255), 2)  # Red line (detected vehicle)

                # Increment the vehicle count for each detected line
                vehicle_count += 1

    # Display the vehicle count in the frame
```

```
    cv2.putText(frame, f"Vehicles: {vehicle_count}", (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

    # Write the frame with vehicle count to the output video
    output.write(frame)

    # Display the processed frame
    cv2.imshow("Traffic Detection", frame)

    if cv2.waitKey(1) & 0xFF == 27:  # Press 'Esc' to exit
        break

    # Update the previous vehicle count
    prev_vehicle_count = vehicle_count

# Release the video capture and output objects
cap.release()
output.release()
cv2.destroyAllWindows()
```



CONCLUSION:

In conclusion, a well-implemented traffic management system through these program technologies is a critical component of modern urban and transportation infrastructure. It plays a pivotal role in ensuring efficient, safe, and sustainable mobility for both people and goods