CP1

Anastasia Sharangia

Code:

```
import cv2
import numpy as np

# Function to apply Sobel edge detection
def sobel_edge_detection(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    sobel_x = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=np.float32)
    sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=np.float32)

    grad_x = cv2.filter2D(gray, -1, sobel_x)
    grad_y = cv2.filter2D(gray, -1, sobel_y)

    magnitude = np.sqrt(grad_x**2 + grad_y**2)
    magnitude = np.uint8(np.clip(magnitude, 0, 255))

    return magnitude

# DBSCAN clustering function
def dbscan(data, eps, min_samples):
    n_samples = data.shape[0]
    labels = -1 * np.ones(n_samples, dtype=int)  # Initialize all points as noise
(-1)
    cluster_id = 0

    # Function to calculate Euclidean distance between two points
    def euclidean_distance(p1, p2):
        return np.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

    # Function to find the neighbors of a point
    def region_query(point_idx):
        neighbors = []
        for i in range(n_samples):
            if euclidean_distance(data[point_idx], data[i]) <= eps:
                neighbors.append(i)
        return neighbors
```

```
    for point_idx in range(n_samples):
        if labels[point_idx] != -1:  # Skip if already visited
            continue

        neighbors = region_query(point_idx)

        if len(neighbors) < min_samples:
            labels[point_idx] = -1  # Mark as noise
        else:
            labels[point_idx] = cluster_id
            i = 0
            while i < len(neighbors):
                neighbor_idx = neighbors[i]
                if labels[neighbor_idx] == -1:
                    labels[neighbor_idx] = cluster_id  # Change noise to border
point
                elif labels[neighbor_idx] == -2:  # If unvisited
                    labels[neighbor_idx] = cluster_id
                    # Expand neighbors
                    new_neighbors = region_query(neighbor_idx)
                    if len(new_neighbors) >= min_samples:
                        neighbors.extend(new_neighbors)
                i += 1
            cluster_id += 1

    return labels

# Function to calculate the center of the bounding box (mid-point)
def calculate_bounding_box_centroid(cluster_points):
    # Find the bounding box by getting min/max coordinates of the points
    min_x = np.min(cluster_points[:, 0])
    max_x = np.max(cluster_points[:, 0])
    min_y = np.min(cluster_points[:, 1])
    max_y = np.max(cluster_points[:, 1])

    # The centroid is the center of the bounding box
    centroid = ((min_x + max_x) // 2, (min_y + max_y) // 2)
    return centroid

# Load the video
video_path = './content/cp1.mp4'
cap = cv2.VideoCapture(video_path)
```

```
# Background Subtractor
fgbg = cv2.createBackgroundSubtractorMOG2(history=600, varThreshold=20,
detectShadows=True)

# Parameters for DBSCAN and object tracking
min_contour_area = 1100
eps = 200  # DBSCAN epsilon
min_samples = 200  # DBSCAN minimum samples

previous_centroids = {}  # Store centroids from previous frame
cluster_speeds = {}  # Store speeds for each cluster

if not cap.isOpened():
    print("Error: Could not open video.")
else:
    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Resize the frame
        frame = cv2.resize(frame, (640, 360))

        # Background subtraction
        fgmask = fgbg.apply(frame)

        # Apply Sobel edge detection
        edges = sobel_edge_detection(frame)

        # Combine background mask and edges
        combined = cv2.bitwise_and(fgmask, edges)

        # Find non-zero pixels in the combined image (moving object pixels)
        non_zero_coords = np.column_stack(np.where(combined > 0))

        if len(non_zero_coords) > 0:
            # Apply DBSCAN on the non-zero coordinates
            labels = dbscan(non_zero_coords, eps, min_samples)

            # Visualize the clusters
            output_frame = frame.copy()
```

```
            # Store current frame centroids
            current_centroids = {}

            unique_labels = np.unique(labels)
            for cluster_id in unique_labels:
                if cluster_id == -1:  # Skip noise
                    continue
                color = (0, 0, 255)
                cluster_points = non_zero_coords[labels == cluster_id]
                centroid = calculate_bounding_box_centroid(cluster_points)  #
Calculate centroid as bounding box center
                current_centroids[cluster_id] = centroid

                # Draw the cluster points
                for point in cluster_points:
                    cv2.circle(output_frame, tuple(point[::-1]), 1, color, -1)

                # Calculate the speed if we have a previous centroid for this
cluster
                if cluster_id in previous_centroids:
                    prev_centroid = previous_centroids[cluster_id]
                    speed = np.linalg.norm(np.array(centroid) -
np.array(prev_centroid))  # Euclidean distance
                    # Store the speed for this cluster
                    if cluster_id not in cluster_speeds:
                        cluster_speeds[cluster_id] = []
                    cluster_speeds[cluster_id].append(speed)

            # Update previous centroids for the next frame
            previous_centroids = current_centroids

            # Show the output frame with clusters
            cv2.imshow('Clustered Frame', output_frame)

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

    # Calculate and print average speed of each cluster
    print("Average speeds of clusters:")
    for cluster_id, speeds in cluster_speeds.items():
        average_speed = np.mean(speeds)
        print(f"Cluster {cluster_id}: {average_speed:.2f} pixels/frame")
```

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

Input:

Video of a single moving ball

Output:

Speed output

```
Average speeds of clusters:
Cluster 0: 4.97 pixels/frame
```

Video working output kinda looks like this

Video not working output kinda looks like this

Explanation:

I started by implementing an edge detection function using the Sobel operator to detect edges in video frames. The function first converts the image to grayscale, then applies Sobel filters (`sobel_x` and `sobel_y`) to compute the horizontal and vertical gradients. These gradients are combined to calculate the magnitude of edges, which is returned as an 8-bit image representing the edge intensity.

Next, I wrote a function to calculate the Euclidean distance between two points, which is crucial for both object tracking and clustering. This distance metric is used to identify the movement of objects (such as balls) across frames and is also essential for grouping objects in the clustering process.

For clustering, I implemented the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm. This function labels points based on their density and groups them into clusters. It initializes all points as noise, then iteratively processes each point. If a point has enough neighbors within a given distance (`eps`), it's considered a core point, and a new cluster is formed. The algorithm then expands the cluster by checking neighbors of core points. Points that don't meet the density requirement are

marked as noise. The clusters are identified based on spatial proximity without the need for predefining the number of clusters.

I then applied these functions to process video frames. The video is loaded using OpenCV, and background subtraction is applied to isolate moving objects. The background subtraction is enhanced using a mask and combined with edge detection to highlight the moving objects and their contours.

For object tracking, the algorithm matches the current object's position with its previous position using Euclidean distance. If a match is found, the object's track is updated with the new position; otherwise, a new track is initiated. The centroids that don't match existing tracks are treated as new objects and tracked accordingly.

Finally, I calculate the average speed of each tracked object. The program sums the Euclidean distances between consecutive centroids in each object's track and computes the total distance traveled. The total time is calculated using the number of frames in the track and the frame rate. The average speed is then computed as the total distance divided by the total time. To reduce false positives caused by random movement (such as camera shake), tracks lasting fewer than a set number of frames are ignored.

This approach provides a basic method for detecting, tracking, clustering, and calculating the speed of objects in a video, and it is NOT reliable and can be affected by various factors like noise and false tracking.

Constraints:

Does not work on moving camera

Does not work on shaky camera

Does not work on a video that has too much objects

Does not work with sudden movements

And has a lot of other problematic constraints, but well i tried :’) .