bev_projection.py:

```
import numpy as np
import json


class BEVProjector:
    def __init__(self, config_path):
        cfg = json.load(open(config_path))
        self.cameras = {
            c['id']: np.array(c['homography'], dtype=np.float32)
            for c in cfg['cameras']
        }


    def image_to_bev(self, cam_id, point):
        """Proyecta punto (u,v) de la imagen a coordenadas BEV (x,y)"""
        H = self.cameras[cam_id]
        uv1 = np.array([point[0], point[1], 1.0])
        xyw = H.dot(uv1)
        return (xyw[0]/xyw[2], xyw[1]/xyw[2])


    def bev_to_image(self, cam_id, bev_point):
        """Proyecta punto BEV (x,y) de vuelta a coordenadas de imagen"""
        H = self.cameras[cam_id]
        H_inv = np.linalg.inv(H)
        xy1 = np.array([bev_point[0], bev_point[1], 1.0])
        uvw = H_inv.dot(xy1)
        return (int(uvw[0]/uvw[2]), int(uvw[1]/uvw[2]))
```

camera_stream.py:

```
import cv2

import threading

import json


class RTSPCamera:

    def __init__(self, cam_id, source):

        self.cam_id = cam_id

        self.source = source

        self.cap = cv2.VideoCapture(self.source)

        self.frame = None

        self.stopped = False

        self.lock = threading.Lock()


    def start(self):

        threading.Thread(target=self._update, daemon=True).start()

        return self


    def _update(self):

        while not self.stopped:

            ret, frame = self.cap.read()

            if not ret:

                continue

            with self.lock:
```

```
        self.frame = frame

  def read(self):
    with self.lock:
      return self.frame.copy() if self.frame is not None else None

  def stop(self):
    self.stopped = True
    self.cap.release()

def load_cameras_from_config(config_path):
  with open(config_path, 'r') as f:
    config = json.load(f)

  cameras = []
  for cam_cfg in config['cameras']:
    cam = RTSPCamera(cam_cfg['id'], cam_cfg['source']).start()
    cameras.append(cam)

  return cameras
```

config.json:

```
{
 "cameras": [
  {
    "id": 1,
    "rtsp_url": "http://192.168.0.5:4747/video",
    "homography": [
      [0.5, 0, -100],
      [0, 0.5, -50],
      [0.001, 0.002, 1]
    ]
  },
  {
    "id": 2,
    "rtsp_url": "http://192.168.0.6:4747/video",
    "homography": [
      [0.6, 0, -90],
      [0, 0.6, -60],
      [0.001, 0.002, 1]
    ]
  },
  {
    "id": 3,
    "rtsp_url": 0,
    "homography": [
      [0.55, 0, -80],
      [0, 0.55, -55],
      [0.001, 0.002, 1]
    ]
  }
  ],
  "bev": {
    "grid_resolution": 0.05
  },
  "tracker": {
    "max_age": 30,
    "dist_threshold": 1.0
  }
}
```

detector.py

```
from ultralytics import YOLO

class YOLOv11Detector:
    def __init__(self, model_path='yolo11n.pt', threshold=0.5):
        self.model = YOLO(model_path)
        self.threshold = threshold

    def detect(self, frame):
        results = self.model(frame)[0]
        detections = []
        for box, cls, conf in zip(results.boxes.xyxy, results.boxes.cls, results.boxes.conf):
            if int(cls) == 0 and conf >= self.threshold:  # Clase 0 = 'person'
```

```
                x1, y1, x2, y2 = map(int, box.tolist())
                detections.append({
                    'bbox': (x1, y1, x2 - x1, y2 - y1),
                    'score': float(conf)
                })
        return detections
```

index.py:

```
import time
import json
import cv2
import numpy as np
from camera_stream import RTSPCamera
from detector import YOLOv11Detector
from bev_projection import BEVProjector
from tracker import MultiCameraTracker
from visualization import draw_detections, draw_trajectories
from logger import CSVLogger

# 1) Carga de configuración
dir_cfg = 'config.json'
config = json.load(open(dir_cfg))

# 2) Inicializar cámaras (asegúrate de que para la webcam local uses un entero 0, no "0")
```

```
cams = [
    RTSPCamera(c['id'], c['rtsp_url']).start()
    for c in config['cameras']
]

detector = YOLOv11Detector()
projector = BEVProjector(dir_cfg)
tracker = MultiCameraTracker(
    config['tracker']['max_age'],
    config['tracker']['dist_threshold']
)
logger = CSVLogger('trajectories.csv')

# 3) Crear lienzo BEV
bev_h, bev_w = 500, 500
bev_canvas = np.ones((bev_h, bev_w, 3), dtype=np.uint8) * 255

try:
    while True:
        frames = []
        all_dets = []

        # 4) Leer cada cámara y detectar
        for cam in cams:
            frame = cam.read()
            if frame is None:
```

```
        # placeholder negro si no llega frame
        frame = np.zeros((240, 320, 3), dtype=np.uint8)
    frames.append((cam.cam_id, frame))

    # Detector + proyección BEV
    dets_img = detector.detect(frame)
    for d in dets_img:
        x, y, w, h = d['bbox']
        base = (x + w // 2, y + h)
        bev_pt = projector.image_to_bev(cam.cam_id, base)
        all_dets.append((bev_pt[0], bev_pt[1], cam.cam_id))

# 5) Actualizar tracker global
tracks = tracker.update(all_dets)

# 6) Dibujar BEV unificado
bev_canvas[:] = 255
for trk in tracks:
    if trk.history:
        pts = np.array([
            (int(x * 100 + bev_w/2), int(bev_h - y * 100))
            for x, y in trk.history
        ], dtype=np.int32)
        cv2.polylines(bev_canvas, [pts], False, (0,0,255), 2)
        cx, cy = pts[-1]
        cv2.circle(bev_canvas, (cx, cy), 5, (0,0,255), -1)
        cv2.putText(bev_canvas, f"ID:{trk.id}",
              (cx+5, cy-5),
              cv2.FONT_HERSHEY_SIMPLEX, 0.4,
              (0,0,0), 1)

    # 7) Visualizar detecciones + trayectorias en cada frame
    vis_frames = []
    for cam_id, frame in frames:
      dets = detector.detect(frame)
      vis = draw_detections(frame.copy(), dets, tracks, projector, cam_id)
      vis = draw_trajectories(vis, tracks, projector, cam_id)
      vis_frames.append(cv2.resize(vis, (320, 240)))

    # 8) Combinar todos los frames en una sola imagen
    combined = cv2.hconcat(vis_frames)

    # 9) Mostrar ventanas
    cv2.imshow('Multi-Camera Tracking', combined)
    cv2.imshow('Bird Eye View (BEV)', bev_canvas)

    # 10) Registrar última posición de cada track
    for trk in tracks:
      if trk.history:
        logger.log(trk.id, trk.history[-1], cam_id)

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

        time.sleep(0.03)

finally:
    for cam in cams:
        cam.stop()
    cv2.destroyAllWindows()
```

logger.py:

```
import csv
from datetime import datetime

class CSVLogger:
    def __init__(self, path):
        self.file = open(path, 'w', newline='')
        self.writer = csv.writer(self.file)
        self.writer.writerow([
            'track_id','timestamp','x_bev','y_bev','cam_id'
        ])

    def log(self, track_id, pos, cam_id):
        ts = datetime.utcnow().isoformat()
        self.writer.writerow([
            track_id, ts, pos[0], pos[1], cam_id
        ])
        self.file.flush()
```

tracker.py:

```
import numpy as np
from filterpy.kalman import KalmanFilter
from scipy.optimize import linear_sum_assignment

class Track:
    def __init__(self, track_id, init_pos, max_age):
        self.id = track_id
        self.kf = KalmanFilter(dim_x=4, dim_z=2)
        self.kf.x = np.array([init_pos[0], init_pos[1], 0, 0])  # [x, y, vx, vy]

        # Matriz de transición de estado (modelo de velocidad constante)
        self.kf.F = np.array([
            [1, 0, 1, 0],
            [0, 1, 0, 1],
            [0, 0, 1, 0],
            [0, 0, 0, 1]
        ])

        # Matriz de observación (solo observamos posición)
        self.kf.H = np.array([
            [1, 0, 0, 0],
            [0, 1, 0, 0]
        ])

        # Covarianza del error de estimación inicial
        self.kf.P = np.eye(4) * 100

        # Covarianza del ruido de medición
        self.kf.R = np.eye(2) * 5

        # Covarianza del ruido del proceso
        self.kf.Q = np.eye(4) * 0.1

        self.age = 0
        self.max_age = max_age
        self.history = []
        self.cam_id = init_pos[2] if len(init_pos) > 2 else None

    def predict(self):
        self.kf.predict()
        self.age += 1
        return self.kf.x[:2]

    def update(self, pos):
        self.kf.update(np.array(pos[:2]))
        self.history.append((pos[0], pos[1]))
        self.age = 0
        if len(pos) > 2:
            self.cam_id = pos[2]

class MultiCameraTracker:
    def __init__(self, max_age=30, dist_threshold=1.0):
        self.max_age = max_age
        self.dist_threshold = dist_threshold
        self.next_id = 1
        self.tracks = []

    def update(self, detections):
        # Paso 1: Predecir la posición de cada track
        for trk in self.tracks:
            trk.predict()

        # Si no hay detecciones, solo actualiza los tracks existentes
        if not detections:
            self.tracks = [trk for trk in self.tracks if trk.age <= self.max_age]
            return self.tracks

        # Si no hay tracks, crea nuevos para todas las detecciones
        if not self.tracks:
            for det in detections:
                self.tracks.append(Track(self.next_id, det, self.max_age))
                self.next_id += 1
    return self.tracks

# Paso 2: Asignación de detecciones a tracks existentes
pred_positions = np.array([trk.kf.x[:2] for trk in self.tracks])
det_positions = np.array([d[:2] for d in detections])

# Matriz de costos (distancia entre predicciones y detecciones)
cost_matrix = np.linalg.norm(pred_positions[:, None] - det_positions, axis=2)

# Resolver el problema de asignación
row_ind, col_ind = linear_sum_assignment(cost_matrix)

# Paso 3: Actualizar tracks asignados
assigned_tracks = set()
assigned_dets = set()

for r, c in zip(row_ind, col_ind):
    if cost_matrix[r, c] < self.dist_threshold:
        self.tracks[r].update(detections[c])
        assigned_tracks.add(r)
        assigned_dets.add(c)

# Paso 4: Crear nuevos tracks para detecciones no asignadas
for i, det in enumerate(detections):
    if i not in assigned_dets:
        self.tracks.append(Track(self.next_id, det, self.max_age))
```

```
                self.next_id += 1

        # Paso 5: Eliminar tracks perdidos
        self.tracks = [trk for trk in self.tracks if trk.age <= self.max_age]

        return self.tracks
```

visualization.py:

```
import cv2
import numpy as np

def draw_detections(frame, detections, tracks, projector, cam_id):
    for det in detections:
        x,y,w,h = det['bbox']
        cv2.rectangle(frame, (x,y), (x+w, y+h), (0,255,0), 2)
        base = (x + w//2, y + h)
        bev_pt = projector.image_to_bev(cam_id, base)
        for trk in tracks:
            if np.linalg.norm(trk.kf.x[:2] - np.array(bev_pt)) < 0.5:
                cv2.putText(
                    frame, f"ID:{trk.id}", (x,y-10),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    0.5, (0,255,0), 2
                )
                break
    return frame


def draw_trajectories(frame, tracks, projector, cam_id):
    for trk in tracks:
        pts = [
            projector.bev_to_image(cam_id, (p[0], p[1]))
            for p in trk.history
        ]
        for i in range(1, len(pts)):
            cv2.line(frame, pts[i-1], pts[i], (255,0,0), 2)
    return frame
```