I'm starting a new project using my drone with px4 and ardupilot firmware. Im going to setup a raspberrypi onboard and connect a DJI Action 3 Camera to it. drone will connected via mavros. using ros2 jazzy. my idea is to connect action 3 to pi and connect it to opencv and ros2. wheer im going to navigae using cam feen and another node. so what is best nodes to have
my idea is mavros node, camera to ROS node, camera feed to openCV node (so i can change the open cv funtionality), control node based on openCV.

my first nagivation idea is to rotate my drone in360 arraound a person while keeping same distance and keep person center

awesome project. here's a clean, swappable node layout for **ROS 2 Jazzy + MAVROS** on a Raspberry Pi, with a DJI Action 3 feeding OpenCV, and a first behavior: **360° orbit around a person while keeping constant distance and centered in view**.

Minimal node graph (recommended)

```
bash

[dji_camera_driver]  →  [image_proc]  →  [person_tracker]  →  [orbit_controller]  →
/mavros/setpoint_raw/local
                                          │                        │
                                          └──── /tf (camera→base_link)   └── /mavros/state,
/mavros/local_position/pose, /mavros/extended_state
```

1) `dji_camera_driver` (camera → ROS)

- **Role:** Publish `sensor_msgs/Image` + `CameraInfo`.
- **How:**
  - If Action 3 supports **UVC** on USB: open `/dev/videoX` via `v4l2src/gscam` or `image_tools`.
  - If using **HDMI out** + USB capture dongle: same, it still appears as `/dev/videoX`.
- **Topics**
  - `/camera/image_raw` (`sensor_msgs/Image`)
  - `/camera/camera_info` (`sensor_msgs/CameraInfo`)
- **Key params**
  - `device: /dev/video0`
  - `width/height/fps`
  - `exposure/iso` (lock if your lighting is stable)
- **Notes**
  - Prefer `image_transport` (e.g., `compressed`) if bandwidth to offboard is tight.

2) `image_proc` (optional but recommended)

- **Role:** Rectify, resize, color-convert; publish `/image_rect`.
- **Topics**
  - in: `/camera/image_raw`, `/camera/camera_info`
  - out: `/camera/image_rect` (and/or `/camera/image_mono`)
- **Why:** keep your tracker simple/fast and consistent.

3) `person_tracker` (OpenCV/AI → bearing + range)

- **Role:** Find the person in the frame and output:
  - **bearing error** (px or radians from image center)
  - **range estimate** (meters) — see notes below
  - **tracking confidence**
- **Topics**
  - in: `/camera/image_rect`
  - out:
    - `/vision/target_pixel` (`geometry_msgs/Point` → x,y = pixel center; z unused)
    - `/vision/target_bearing` (`std_msgs/Float32` radians)
    - `/vision/target_range` (`std_msgs/Float32` meters, NaN if unknown)
    - `/vision/ok` (`std_msgs/Bool`)
- **How to get range**
  - Best: add a small **range sensor** (e.g., TFmini/TF-Luna) forward-facing → publish `/range/front`.
  - Or **AprilTag/Aruco** on the person for calibrated pose.
  - Or **monocular**: rough scale via known target height—works but drifts; acceptable for a demo with a safety limit.
- **Model choice**
  - CPU-friendly: lightweight person detector (e.g., MobileNet-SSD, Nano-YOLO). Keep it under ~10–15 ms at your target FPS.

4) `orbit_controller` (vision servo → MAVROS)

- **Role:** Use bearing + range (and local pose) to publish **velocity setpoints** that:
  - keep the person centered (minimize bearing error)
  - regulate distance to a target radius **Rdes**
  - maintain altitude **z_des** (or track current)
  - maintain tangential speed to orbit at **ω_des**
- **Control outputs (choose one)**

  - `/mavros/setpoint_raw/local` (`mavros_msgs/PositionTarget`) — robust and expressive
  - or `/mavros/setpoint_velocity/cmd_vel` (`geometry_msgs/TwistStamped`)
- **Other I/O**
  - in: `/mavros/state`, `/mavros/local_position/pose`, `/mavros/extended_state`
  - optional in: `/range/front`
  - out: `/mavros/setpoint_raw/local`
- **Parameters**
  - `R_des` (m), e.g., 3.0
  - `z_des` (m AGL or relative)
  - `omega_des` (rad/s) or orbit period
  - `k_center`, `k_radius`, `k_yaw` (gains)
  - `frame_id` (`base_link` vs `map`)
  - `px4_mode: OFFBOARD` / `apm_mode: GUIDED` (pick once at runtime)
  - `failsafe: true` (stops on target lost; optionally RTL on prolonged loss)

5) `mavros` (flight bridge)

- **Role:** Mode/arming, state, local pose, setpoint interface.
- **Key topics**
  - `/mavros/state`, `/mavros/local_position/pose`
  - `/mavros/set_mode`, `/mavros/cmd/arming`
  - `/mavros/setpoint_raw/local` or `/mavros/setpoint_velocity/cmd_vel`

Orbit control sketch (practical, robust)

Use **image-based visual servoing** with a simple velocity controller in the **body frame**:

Let:

- `e_bearing` = person bearing error (radians, +left/–right) from camera center.
- `d` = current range (m) (from range sensor or estimate).
- `R_des` = desired radius.
- `ω_des` = desired angular rate (rad/s) (positive = CCW).
- `z_des` = desired altitude.

Command (body frame):

- **Yaw rate:** `r = k_yaw * e_bearing`
- **Forward/back (x):** `vx = k_radius * (d - R_des)` (move forward if too far)
- **Lateral (y) for orbit:** `vy = ω_des * R_des` (constant tangential speed)

- **Vertical:** `vz = k_z * (z_des - z)` (from local_position/pose)

Pack into `mavros_msgs/PositionTarget`:

- Set `coordinate_frame = FRAME_BODY_NED` (ArduPilot) or body ENU equivalent for PX4 (double-check frame flags).
- Set `type_mask` to **ignore position** and **use velocity + yaw_rate**.

Add **saturations** (e.g., |vx|,|vy| ≤ 1.5 m/s; |r| ≤ 0.8 rad/s) and a **soft-start** ramp. If `person_tracker` drops below confidence or `range` is NaN for >N frames, **hold** or **land/RTL** based on your `failsafe` policy.

Launch layout (single UAV)

```bash
ros2 launch your_pkg orbit_demo.launch.py \
  fcu_url:=serial:///dev/ttyUSB0:921600  \
  use_apm:=true R_des:=3.0 z_des:=5.0 omega_des:=0.4 \
  camera_device:=/dev/video0 width:=1280 height:=720 fps:=30
```

Example `orbit_demo.launch.py` (trimmed for clarity)

```python
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        DeclareLaunchArgument('fcu_url', default_value='serial:///dev/ttyUSB0:921600'),
        DeclareLaunchArgument('use_apm', default_value='true'),  # false => PX4
        DeclareLaunchArgument('R_des', default_value='3.0'),
        DeclareLaunchArgument('z_des', default_value='5.0'),
        DeclareLaunchArgument('omega_des', default_value='0.4'),
        DeclareLaunchArgument('camera_device', default_value='/dev/video0'),
        DeclareLaunchArgument('width', default_value='1280'),
        DeclareLaunchArgument('height', default_value='720'),
        DeclareLaunchArgument('fps', default_value='30'),

        # MAVROS
        Node(
            package='mavros', executable='mavros_node', output='screen',
            parameters=[{
                'fcu_url': LaunchConfiguration('fcu_url'),
                'gcs_url': '',  # optional
            }],
        ),

        # Camera driver (V4L2 → Image)
        Node(
            package='v4l2_camera', executable='v4l2_camera_node', output='screen',
            namespace='camera',
            parameters=[{
                'video_device': LaunchConfiguration('camera_device'),
                'image_size': [LaunchConfiguration('width'), LaunchConfiguration('height')],
                'frame_rate': LaunchConfiguration('fps'),
            }],
            remappings=[('image_raw', '/camera/image_raw')],
        ),

        # (Optional) image_proc
```

```
        Node(
            package='image_proc', executable='image_proc', name='image_proc',
            remappings=[('image', '/camera/image_raw'),
                        ('image_rect', '/camera/image_rect'),
                        ('camera_info', '/camera/camera_info')],
        ),

        # Person tracker
        Node(
            package='orbit_vision', executable='person_tracker', output='screen',
            parameters=[{'min_conf': 0.5, 'publish_debug': True}],
            remappings=[('image', '/camera/image_rect')],
        ),

        # Orbit controller
        Node(
            package='orbit_control', executable='orbit_controller', output='screen',
            parameters=[{
                'R_des': LaunchConfiguration('R_des'),
                'z_des': LaunchConfiguration('z_des'),
                'omega_des': LaunchConfiguration('omega_des'),
                'use_apm': LaunchConfiguration('use_apm'),
                'failsafe_timeout': 1.0,
                'k_yaw': 2.0, 'k_radius': 0.8, 'k_z': 1.0,
            }],
        ),
    ])
```

Tiny code skeletons (ROS 2 Jazzy, Python)

`person_tracker.py` (publishes bearing & rough range)

```python
import rclpy, math, cv2
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
from std_msgs.msg import Float32, Bool
from geometry_msgs.msg import Point

class PersonTracker(Node):
    def __init__(self):
        super().__init__('person_tracker')
        self.bridge = CvBridge()
        self.sub = self.create_subscription(Image, '/camera/image_rect', self.cb, 10)
        self.pub_pix = self.create_publisher(Point, '/vision/target_pixel', 10)
        self.pub_brg = self.create_publisher(Float32, '/vision/target_bearing', 10)
        self.pub_rng = self.create_publisher(Float32, '/vision/target_range', 10)
        self.pub_ok  = self.create_publisher(Bool, '/vision/ok', 10)
        self.fx = 800.0  # calibrate
        self.cx = 640.0  # center x
        self.conf_thresh = 0.5
        # TODO: load lightweight detector; placeholder is center of frame
    def cb(self, msg):
        img = self.bridge.imgmsg_to_cv2(msg, 'bgr8')
        h, w = img.shape[:2]
        # TODO: run detector; fake bbox at image center for skeleton:
        x_c = w/2; y_c = h/2; conf = 1.0
        ok = Bool(); ok.data = conf >= self.conf_thresh
        self.pub_ok.publish(ok)
        if not ok.data:
            return
        # bearing (radians): small-angle approx using pinhole model
        bearing = math.atan2((x_c - self.cx), self.fx)  # +left
        self.pub_brg.publish(Float32(data=float(bearing)))
        # rough range (unknown): publish NaN for now
        self.pub_rng.publish(Float32(data=float('nan')))
        self.pub_pix.publish(Point(x=float(x_c), y=float(y_c), z=0.0))
```

```
def main():
    rclpy.init()
    rclpy.spin(PersonTracker())
    rclpy.shutdown()

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

`orbit_controller.py` (body-frame velocity + yaw rate)

```
python


import rclpy, math
from rclpy.node import Node
from geometry_msgs.msg import PoseStamped, TwistStamped
from mavros_msgs.msg import PositionTarget, State
from std_msgs.msg import Float32, Bool

class OrbitController(Node):
    def __init__(self):
        super().__init__('orbit_controller')
        self.R_des = self.declare_parameter('R_des', 3.0).value
        self.z_des = self.declare_parameter('z_des', 5.0).value
        self.omega = self.declare_parameter('omega_des', 0.4).value
        self.k_yaw = self.declare_parameter('k_yaw', 2.0).value
        self.k_rad = self.declare_parameter('k_radius', 0.8).value
        self.k_z   = self.declare_parameter('k_z', 1.0).value
        self.use_apm = self.declare_parameter('use_apm', True).value

        self.bearing = 0.0
        self.range_m = float('nan')
        self.has_target = False
        self.alt = 0.0

        self.sub_brg = self.create_subscription(Float32, '/vision/target_bearing', self.cb_brg,
10)
        self.sub_rng = self.create_subscription(Float32, '/vision/target_range', self.cb_rng,
10)
        self.sub_ok  = self.create_subscription(Bool, '/vision/ok', self.cb_ok, 10)
        self.sub_pose= self.create_subscription(PoseStamped, '/mavros/local_position/pose',
self.cb_pose, 10)
        self.pub_raw = self.create_publisher(PositionTarget, '/mavros/setpoint_raw/local', 10)
        self.timer = self.create_timer(0.05, self.tick)  # 20 Hz

    def cb_brg(self, m): self.bearing = m.data
    def cb_rng(self, m): self.range_m = m.data
    def cb_ok (self, m): self.has_target = m.data
    def cb_pose(self, m): self.alt = m.pose.position.z

    def tick(self):
        if not self.has_target:
            return
        # Controller
        r = max(-0.8, min(0.8, self.k_yaw * self.bearing))  # yaw rate (rad/s)
        if math.isnan(self.range_m):
            vx = 0.0  # no radial control without range; you can hold existing radius
        else:
            vx = max(-1.5, min(1.5, self.k_rad * (self.range_m - self.R_des)))  # body x
        vy = max(-1.5, min(1.5, self.omega * self.R_des))  # tangential
        vz = max(-1.0, min(1.0, self.k_z * (self.z_des - self.alt)))

        pt = PositionTarget()
        # Choose a body frame; FRAME_BODY_NED=8 (APM), FRAME_BODY_FRD for PX4 as appropriate
        pt.coordinate_frame = PositionTarget.FRAME_BODY_NED
        IGN_POS = (PositionTarget.IGNORE_PX | PositionTarget.IGNORE_PY |
PositionTarget.IGNORE_PZ)
        USE_VEL = 0
        IGN_ACC = (PositionTarget.IGNORE_AFX | PositionTarget.IGNORE_AFY |
PositionTarget.IGNORE_AFZ)
        IGN_YAW = PositionTarget.IGNORE_YAW  # we will use yaw_rate
```

```
        # do not ignore YAW_RATE
        pt.type_mask = IGN_POS | USE_VEL | IGN_ACC | IGN_YAW
        pt.velocity.x = vx
        pt.velocity.y = vy
        pt.velocity.z = -vz  # NED: up is negative; adjust per frame you use
        pt.yaw_rate = r
        self.pub_raw.publish(pt)

def main():
    rclpy.init()
    rclpy.spin(OrbitController()); rclpy.shutdown()

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

⚠️ Frame notes: PX4 vs ArduPilot differ in body/NED/ENU details. If your velocities go the "wrong way," flip signs or switch to the appropriate `coordinate_frame` (PX4 often prefers **FRD**; ArduPilot uses **NED**). Test in SITL first.

Practical wiring for DJI Action 3 → Pi

1. **Try UVC mode first**
   Settings → USB mode → "Webcam/UVC" (if available on Action 3). On the Pi:

```
bash

v4l2-ctl --list-devices
# Expect /dev/video0
```

Then launch `v4l2_camera` as above.

2. **If no UVC**, use **HDMI out** from the Action 3 → **USB HDMI capture** dongle → Pi; it will still appear as `/dev/videoX`.
3. For low latency, consider **gscam** with a GStreamer pipeline (H264 → decode → appsink) if you end up using the Action 3 over an IP stream. Keep total pipeline latency < 80 ms.

Safety & reliability checklist (field-ready)

- **Mode gating:** Don't switch to OFFBOARD/GUIDED until your controller publishes at ≥20 Hz for ≥0.5 s and target is valid.
- **Geofence:** Set radius/altitude limits in FCU and in node.
- **Failsafe:** If `/vision/ok == false` for >1.0 s → hold / brake; if >5 s → RTL (your call).
- **SRx rates:** Ensure MAVLINK stream rates for **LOCAL_POSITION_NED** and **EXTENDED_SYS_STATE** are non-zero.
- **CPU:** Pin detector to one core, and run controller in another; target 30 FPS vision, 20–50 Hz control.

What you'll implement first

1. Bring up **MAVROS** (serial or UDP) and confirm `/mavros/local_position/pose`.

2. Bring up **camera** to `/camera/image_rect`.
3. Run the **tracker** (start with a dummy center target; then plug in a lightweight person detector).
4. Verify the **orbit controller** publishes sane velocity values in SITL.
5. Flight test with **very small** `omega_des` and **R_des ≥ 3 m** in a wide, safe area.

If you want, I can tailor the `person_tracker` to a specific detector you prefer (e.g., YOLO-NAS-S, MobileNet-SSD, or ArUco/AprilTag for a marker-based demo), and give you a drop-in `requirements.txt` + optimized OpenCV build flags for the Pi.