Guía Completa: Configuración y Uso de ORB-SLAM3 con Docker y ROS2 Humble

Este documento explica paso a paso el proceso para configurar el *wrapper* de ORB-SLAM3 en un entorno Docker con ROS2 Humble, y cómo utilizarlo con la cámara de tu dron Tello.

Parte 1: Explicación del Proceso de Descarga y Configuración

Hemos utilizado un repositorio de GitHub que proporciona un *wrapper* de ORB-SLAM3 preconfigurado para ROS2 Humble dentro de un entorno Docker. La ventaja de usar Docker es que aísla todas las dependencias (ROS2, OpenCV, Pangolin, ORB-SLAM3 y sus librerías internas) en un contenedor, evitando conflictos con tu sistema operativo principal y simplificando enormemente la instalación.

1. Clonación del Repositorio y Submódulos

El primer paso fue obtener todo el código necesario. El repositorio principal (ORB-SLAM3-ROS2-Docker) no contiene todo el código de ORB-SLAM3 directamente, sino que lo referencia como un "submódulo" de Git.

- **Comando:**
  git clone https://github.com/suchetanrs/ORB-SLAM3-ROS2-Docker
  cd ORB-SLAM3-ROS2-Docker
  git submodule update --init --recursive --remote

- **Explicación:**
  - git clone: Descarga el repositorio principal a tu máquina.
  - cd ORB-SLAM3-ROS2-Docker: Te mueve al directorio recién clonado.
  - git submodule update --init --recursive --remote: Este comando es crucial. Descarga el código de todos los submódulos (como la propia librería de ORB-SLAM3 y otras dependencias) que el repositorio principal necesita. Sin esto, el código estaría incompleto.

2. Instalación de Docker en tu Sistema

Docker es la plataforma que nos permite crear y ejecutar los contenedores. El repositorio incluye un *script* para facilitar su instalación en Ubuntu.

- **Comando:**
  sudo chmod +x container_root/shell_scripts/docker_install.sh
  ./container_root/shell_scripts/docker_install.sh

- **Explicación:**

    - sudo chmod +x ...: Le da permisos de ejecución al *script* de instalación de Docker.
    - ./...docker_install.sh: Ejecuta el *script*. Este *script* se encarga de añadir los repositorios de Docker, instalar los paquetes necesarios y configurar tu usuario para que pueda usar Docker.
- Configuración de X11 Forwarding (para visualización):
  Para que las aplicaciones gráficas que se ejecutan dentro del contenedor (como el visor de ORB-SLAM3) puedan mostrarse en tu escritorio, configuramos X11 Forwarding.
  echo "xhost +" >> ~/.bashrc
  source ~/.bashrc

    - echo "xhost +" >> ~/.bashrc: Añade la línea xhost + a tu archivo de configuración de terminal, permitiendo conexiones X11 desde cualquier host (incluido tu contenedor Docker).
    - source ~/.bashrc: Recarga tu configuración de terminal para que los cambios surtan efecto inmediatamente.

3. Construcción de la Imagen de Docker con ORB-SLAM3

Una vez que Docker está instalado, construimos la "imagen". Una imagen de Docker es como una plantilla o un "molde" que contiene un sistema operativo (Ubuntu 22.04 en este caso), ROS2 Humble, ORB-SLAM3 compilado, y todas las librerías y configuraciones necesarias.

- **Comando:**
  sudo docker build --build-arg USE_CI=false -t orb-slam3-humble:22.04 .

- **Explicación:**
    - sudo docker build: Inicia el proceso de construcción de la imagen.
    - -t orb-slam3-humble:22.04: Asigna un nombre (orb-slam3-humble) y una etiqueta de versión (22.04) a la imagen.
    - .: Indica que el archivo Dockerfile (que contiene las instrucciones para construir la imagen) se encuentra en el directorio actual.
    - Este paso es el más largo, ya que descarga e instala todo lo necesario dentro de la imagen.

4. Configuración del ROS_DOMAIN_ID y la Red para Docker

Para que tu máquina anfitriona y el contenedor Docker puedan comunicarse a través de ROS2, es fundamental que compartan el mismo dominio ROS y que la configuración de red sea adecuada.

- **Configurar ROS_DOMAIN_ID en tu máquina anfitriona:**
  echo "export ROS_DOMAIN_ID=55" >> ~/.bashrc
  echo "export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp" >> ~/.bashrc
  source ~/.bashrc

    - ROS_DOMAIN_ID=55: El *wrapper* Docker usa el dominio 55 por defecto.
    - RMW_IMPLEMENTATION=rmw_cyclonedds_cpp: Fuerza el uso de CycloneDDS

como *middleware*.

- Modificar docker-compose.yml para el entorno del contenedor:
  En tu máquina anfitriona, abre el archivo docker-compose.yml en la raíz del repositorio:
  cd ~/ORB-SLAM3-ROS2-Docker
  nano docker-compose.yml

  Asegúrate de que la sección environment: dentro del servicio orb_slam3_22_humble contenga las siguientes líneas:

```
services:
  orb_slam3_22_humble:
    # ... otras configuraciones ...
    network_mode: "host" # Crucial para que comparta la red del host
    environment:
      - DISPLAY=${DISPLAY}
      - QT_X11_NO_MITSHM=1
      - ROS_DOMAIN_ID=55
      - RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
    # ...
```

  Guarda y cierra el archivo.

Parte 2: Tutorial de Uso de ORB-SLAM3 con Docker

1. Ejecutar el Contenedor de Docker

Para empezar a trabajar con ORB-SLAM3, inicia un contenedor a partir de la imagen que construimos.

- **Comando:**
  cd ~/ORB-SLAM3-ROS2-Docker # Asegúrate de estar en este directorio en tu máquina anfitriona
  sudo docker compose run --rm orb_slam3_22_humble

- **Explicación:**
  - sudo docker compose run --rm orb_slam3_22_humble: Inicia un nuevo contenedor. El --rm asegura que el contenedor se elimine al salir, evitando contenedores "huérfanos". Esto te dará una nueva línea de comando que indica que estás **dentro del contenedor** (ej. root@<ID_CONTENEDOR>:/#).
- Verificación de X11 Forwarding (Opcional):
  Una vez dentro del contenedor, puedes verificar que la visualización gráfica funciona:
  xeyes

  Si aparece una ventana con ojos que siguen tu cursor, la visualización está correcta.

2. Compilar ORB-SLAM3 y el Wrapper dentro del Contenedor

La compilación final de ORB-SLAM3 y del *wrapper* de ROS2 se hace *dentro* del contenedor.

- **Comandos (ejecutar dentro del contenedor):**
  cd /home/orb/ORB_SLAM3/ && sudo chmod +x build.sh && ./build.sh
  cd /root/colcon_ws/ && colcon build --symlink-install && source install/setup.bash

- **Explicación:**
  - cd /home/orb/ORB_SLAM3/ && ...: Navega al directorio de ORB-SLAM3 dentro del contenedor y lo compila.
  - cd /root/colcon_ws/ && ...: Navega al espacio de trabajo de ROS2, compila los paquetes ROS2 (incluido el *wrapper* orb_slam3_ros2_wrapper) y luego "fuentea" el entorno.

3. Configuración de Parámetros de Cámara y QoS para el Tello ⚙️

Para que ORB-SLAM3 reciba y procese las imágenes de tu Tello, necesitamos configurar los parámetros de la cámara y, crucialmente, la Calidad de Servicio (QoS) para que sea compatible. El Tello publica con BEST_EFFORT y VOLATILE.

- Modificar rgbd-ros-params.yaml (dentro del contenedor):
  Este archivo controla los parámetros ROS del wrapper, incluyendo el QoS del suscriptor.
  cd /root/colcon_ws/src/orb_slam3_ros2_wrapper/params/
  nano rgbd-ros-params.yaml

  Asegúrate de que la sección ros__parameters contenga:

```
ORB_SLAM3_RGBD_ROS2:
  ros__parameters:
    # ...
    rgb_image_topic_name: /tello/imagen # <-- Tu topic de imagen del Tello
    depth_image_topic_name: /depth_camera # <-- Topic para la imagen de profundidad falsa
    # ...
    mode: RGB-D # Mantener en RGB-D si usas la profundidad falsa
    # --- Configuración de QoS para el suscriptor de ORB-SLAM3 ---
    qos_history: 1      # Keep Last
    qos_depth: 1        # Buffer de 1 mensaje
    qos_reliability: 1  # RELIABLE (¡CRUCIAL para que coincida con el publicador!)
    qos_durability: 2   # VOLATILE
    # ...
```

  Guarda y cierra.
- Modificar gazebo_rgbd.yaml (dentro del contenedor):
  Este archivo define los parámetros intrínsecos de la cámara para ORB-SLAM3.
  nano gazebo_rgbd.yaml

- Camera.RGB: 1 si tu Tello publica en RGB, 0 si es BGR.
- Camera.width, Camera.height: Deben coincidir con la resolución de tu *topic* /tello/imagen (ej. 640x480).
- ThDepth y DepthMapFactor: Deben estar **descomentados** y con valores numéricos (ej. 40.0 y 1.0).
- Tbc: La matriz de transformación de cámara a IMU. Si no usas IMU o profundidad, puedes comentar todas las líneas de Tbc: !!opencv-matrix hasta el siguiente parámetro principal.
  Guarda y cierra.

- Modificar tu camara.py (Publicador del Tello) en tu máquina anfitriona:
  Para que el QoS del publicador del Tello coincida con el suscriptor de ORB-SLAM3 (RELIABLE).
  cd ~/DronTello/
  nano camara.py

  Asegúrate de que la importación incluya DurabilityPolicy:
  from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy
  Y que el perfil de QoS sea:

```
qos_profile_publisher = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE, # ¡CAMBIADO A RELIABLE!
    history=HistoryPolicy.KEEP_LAST,
    depth=1,
    durability=DurabilityPolicy.VOLATILE
)
```

  Guarda y cierra.
- Crear fake_depth_publisher.py (Publicador de Profundidad Falsa) en tu máquina anfitriona:
  Este publicador proporcionará una imagen de profundidad simulada que ORB-SLAM3 espera en modo RGB-D.
  cd ~/DronTello/
  nano fake_depth_publisher.py

  Pega el siguiente código:

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import numpy as np
import cv2
```

```
import time

# --- CONFIGURACIÓN ---
ROS_TOPIC_OUTPUT_DEPTH = '/depth_camera' # Asegúrate que coincida con
depth_image_topic_name en el YAML
TIMER_PERIOD_DEPTH = 1.0 / 30.0        # Publicar a 30 Hz

# Estas dimensiones DEBEN COINCIDIR con las de tu imagen RGB del Tello
# y con Camera.width/height en gazebo_rgbd.yaml
FRAME_WIDTH_DEPTH = 640
FRAME_HEIGHT_DEPTH = 480

# Valor de profundidad constante (ej. 1 metro)
FAKE_DEPTH_VALUE = 1.0 # En metros (float32)

class FakeDepthPublisher(Node):
   def __init__(self):
      super().__init__('fake_depth_publisher')
      self.get_logger().info("Iniciando nodo publicador de profundidad falsa.")
      self.get_logger().info(f"Publicando profundidad en:
{ROS_TOPIC_OUTPUT_DEPTH}")

      self.bridge = CvBridge()

      # Crear Publicador ROS con el mismo QoS que el Tello
      qos_profile_publisher = QoSProfile(
         reliability=ReliabilityPolicy.RELIABLE, # ¡CAMBIADO A RELIABLE para coincidir
con ORB-SLAM3!
         history=HistoryPolicy.KEEP_LAST,
         depth=1,
         durability=DurabilityPolicy.VOLATILE
      )
      self.depth_publisher_ = self.create_publisher(Image, ROS_TOPIC_OUTPUT_DEPTH,
qos_profile=qos_profile_publisher)
      self.get_logger().info("Publicador de profundidad falsa ROS creado.")

      # Crear Temporizador ROS para publicar frames
      self.timer = self.create_timer(TIMER_PERIOD_DEPTH, self.timer_callback)
      self.get_logger().info(f"Temporizador creado con periodo:
{TIMER_PERIOD_DEPTH:.3f}s")

   def timer_callback(self):
      """ Se ejecuta periódicamente para publicar un frame de profundidad falsa """
```

```
        fake_depth_array = np.full((FRAME_HEIGHT_DEPTH, FRAME_WIDTH_DEPTH),
                                   FAKE_DEPTH_VALUE, dtype=np.float32)

        ros_depth_msg = self.bridge.cv2_to_imgmsg(fake_depth_array, encoding="32FC1")

        ros_depth_msg.header.stamp = self.get_clock().now().to_msg()
        ros_depth_msg.header.frame_id = "tello_camera" # Coincide con el frame_id del
Tello

        self.depth_publisher_.publish(ros_depth_msg)
        # self.get_logger().debug("Frame de profundidad falsa publicado en ROS.",
throttle_duration_sec=1.0)


    def destroy_node(self):
        """ Se llama automáticamente al cerrar el nodo """
        self.get_logger().info("Cerrando nodo publicador de profundidad falsa...")
        if self.timer:
            self.timer.cancel()
        super().destroy_node()

def main(args=None):
    rclpy.init(args=args)
    node = FakeDepthPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        print("Ctrl+C detectado, cerrando nodo de profundidad falsa...")
    finally:
        node.destroy_node()
        rclpy.shutdown()
        print("Publicador de profundidad falsa finalizado.")

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

Hazlo ejecutable: chmod +x fake_depth_publisher.py

Cómo Ejecutar ORB-SLAM3 con tu Tello (Paso a Paso) 🚀

Para ejecutar ORB-SLAM3 y ver el SLAM en acción con tu dron Tello, sigue estos pasos en el orden indicado:

1. Detén y elimina cualquier contenedor Docker previo de ORB-SLAM3.
   Abre una terminal en tu máquina anfitriona y ejecuta:
   cd ~/ORB-SLAM3-ROS2-Docker
   sudo docker compose down --remove-orphans

2. Lanza los publicadores de ROS2 en tu máquina anfitriona.
   Abre dos terminales separadas en tu máquina anfitriona:
   - **Terminal 1 (Publicador RGB del Tello):**
     cd ~/DronTello/
     source /opt/ros/humble/setup.bash # Fuentea tu entorno ROS2
     python3 camara.py

     Asegúrate de que tu Tello esté encendido y conectado a la WiFi de tu ordenador.
   - **Terminal 2 (Publicador de Profundidad Falsa):**
     cd ~/DronTello/
     source /opt/ros/humble/setup.bash # Fuentea tu entorno ROS2
     python3 fake_depth_publisher.py

     Asegúrate de que ambos *scripts* se inicien sin errores y estén publicando.
3. Verifica la visibilidad de los Topics (Opcional pero recomendado).
   Abre una tercera terminal en tu máquina anfitriona y ejecuta:
   ros2 topic list

   Deberías ver /tello/imagen y /depth_camera en la lista. Si los ves, la comunicación de ROS2 en tu anfitrión está funcionando.
4. Inicia el Contenedor Docker de ORB-SLAM3.
   Abre una cuarta terminal en tu máquina anfitriona y ejecuta:
   cd ~/ORB-SLAM3-ROS2-Docker
   sudo docker compose run --rm orb_slam3_22_humble

   Esto iniciará el contenedor y te dará una línea de comando dentro de él (root@<ID_CONTENEDOR>:/#).
5. Lanza ORB-SLAM3 dentro del Contenedor.
   Una vez dentro del shell del contenedor (en la cuarta terminal):
   source /opt/ros/humble/setup.bash
   source /root/colcon_ws/install/setup.bash
   ros2 launch orb_slam3_ros2_wrapper unirobot.launch.py

   Si todo está configurado correctamente, el visor de ORB-SLAM3 debería aparecer y empezar a mostrar puntos y la trayectoria del dron a medida que muevas el Tello.

Parte 3: Opciones Avanzadas y Tutoriales de Uso del

Wrapper

El *wrapper* de ORB-SLAM3 en Docker ofrece varias opciones y funcionalidades que puedes aprovechar para un control más fino del SLAM y la integración con otros componentes de ROS2.

1. Ejecución con Diferentes Modos de ORB-SLAM3

ORB-SLAM3 soporta varios modos de operación (Monocular, Estéreo, RGB-D, Monocular-Inercial, Estéreo-Inercial). Aunque este *wrapper* está fuertemente configurado para RGB-D, la estrategia actual es alimentarle una profundidad falsa para que funcione. Si quisieras un modo monocular "puro", requeriría modificar el código C++ del *wrapper* o buscar uno que lo soporte de forma nativa.

2. Parámetros Importantes del Wrapper (ROS Parameters)

El README.md del repositorio lista varios parámetros ROS que puedes ajustar en los archivos YAML. Aquí se detallan algunos de los más relevantes, ubicados en rgbd-ros-params.yaml:

- robot_base_frame: Nombre del *frame* base de tu robot (por defecto base_footprint).
- global_frame: Nombre del *frame* global de referencia (por defecto map).
- odom_frame: Nombre del *frame* de odometría (por defecto odom).
- rgb_image_topic_name: El *topic* donde el *wrapper* espera recibir las imágenes RGB (configurado a /tello/imagen).
- depth_image_topic_name: El *topic* para imágenes de profundidad (configurado a /depth_camera para la imagen falsa).
- imu_topic_name: *Topic* para mensajes IMU (no usado en modo RGB-D).
- odometry_mode: true o false. Si es true, el sistema usa datos de odometría externa. Si es false (por defecto), publica la transformación directamente entre global_frame y robot_base_frame.
- visualization: true o false. Habilita/deshabilita el visor de ORB-SLAM3.
- publish_tf: true o false. Publica las transformaciones TF (map->odom o map->odom->base_link).
- map_data_publish_frequency: Intervalo de tiempo en ms para publicar los datos del mapa.
- do_loop_closing: true o false. Habilita/deshabilita el cierre de bucles en ORB-SLAM3.

3. Servicios ROS Disponibles (Services)

El *wrapper* de ORB-SLAM3 expone varios servicios ROS que puedes llamar para interactuar con el SLAM y obtener información:

- orb_slam3/get_map_data: Obtiene los datos del mapa.
- orb_slam3/get_landmarks_in_view: Dada una pose de entrada, publica los puntos característicos visibles desde esa pose.
- orb_slam3/get_all_landmarks_in_map: Publica todos los puntos característicos en el mapa.

- orb_slam3/reset_mapping: Reinicia la instancia de mapeo actual y borra todos los *keyframes*.

Puedes llamar a estos servicios desde otra terminal (dentro del mismo contenedor o en una máquina anfitriona con el ROS_DOMAIN_ID configurado) usando ros2 service call.

- **Ejemplo de uso de un servicio (dentro del contenedor):**
  ros2 service call /orb_slam3/reset_mapping std_srvs/srv/SetBool "{data: true}"

4. Topics ROS Publicados (Published Topics)

El *wrapper* también publica información útil en varios *topics* de ROS2:

- map_points: Nube de puntos que representa los puntos característicos del SLAM (publicado cuando se llama a get_all_landmarks_in_map).
- visible_landmarks: Nube de puntos de los puntos característicos visibles desde una pose dada (publicado cuando se llama a get_landmarks_in_view).
- slam_info: Información general relacionada con el SLAM.
- map_data: Datos del mapa generados continuamente por el algoritmo SLAM.
- robot_pose_slam: La pose del robot en el *frame* global.

Puedes monitorizar estos *topics* usando ros2 topic echo <topic_name> para ver los datos en tiempo real.

5. Uso del Paquete map_generator

Este repositorio incluye un paquete orb_slam3_map_generator que puede usarse para generar una nube de puntos global a partir de los datos del SLAM. Suscribe los datos del mapa (map_data) y una nube de puntos de entrada (LiDAR o cámara de profundidad) y los une basándose en la última pose.

- **Para usarlo:**
  1. Asegúrate de que el contenedor de ORB-SLAM3 esté ejecutándose y el SLAM esté activo.
  2. Dentro del contenedor, el README sugiere un *script* launch_slam.sh que puede lanzar el SLAM y el pointcloud_stitcher (parte del map_generator).
  3. Puedes lanzar el nodo pointcloud_stitcher por separado si lo necesitas.
  4. Para publicar la nube de puntos global en cualquier momento, puedes ejecutar el archivo Python correspondiente.
  5. La nube de puntos global se puede visualizar en RViz.

6. Integración con Simulaciones (Gazebo Sim)

El repositorio está diseñado para integrarse con simulaciones de Gazebo Sim. Si estás usando un simulador, asegúrate de que el ROS_DOMAIN_ID sea el mismo (por defecto 55) para que los *topics* se comuniquen correctamente entre el simulador y el contenedor de ORB-SLAM3.