



TAREA 5

Jaime Mendez

Universidad Santo Tomas de Aquino

Bogotá, Colombia

jaimemendezj@usantotomas.edu.co

I.INTRODUCCIÓN

Este documento detalla el proceso de análisis, personalización y despliegue de tres algoritmos de software distintos: una simulación de un vehículo seguidor de línea, un videojuego de naves espaciales y un nodo del Sistema Operativo para Robots (ROS). El objetivo principal es demostrar un flujo de trabajo moderno de desarrollo y operaciones (DevOps) que abarca desde la adaptación del código hasta su encapsulamiento en contenedores Docker para garantizar la portabilidad y la reproducibilidad. Finalmente, se describe la distribución del código fuente a través de GitHub y de las imágenes de contenedor mediante Docker Hub, completando así un ciclo de entrega de software robusto y estandarizado.

Términos Clave— Docker, Containerización, Git, GitHub, Python, ROS, Tkinter.

II. DESARROLLO DE CONTENIDOS

A partir de esta sección, se desarrollan los contenidos del tema, de una forma ordenada y secuencial.

A. Flujo de Trabajo para el Desarrollo y Despliegue

El proyecto se ejecutó siguiendo un proceso estructurado enfocado en la automatización y la reproducibilidad:

1.Análisis y Personalización del Algoritmo: Se seleccionaron tres algoritmos base y se modificaron para cumplir con los requisitos y añadir una "esencia propia". Esto implicó desde el diseño de nuevas pistas de carreras hasta la alteración de la mecánica de juego.

2.Contenerización con Docker: Para cada proyecto, se creó un Dockerfile que define el entorno de ejecución completo, incluyendo el sistema operativo base, las

dependencias del sistema (ej. tk para Tkinter), las librerías de Python (ej. pygame) y el código fuente de la aplicación.

Control de Versiones con Git y GitHub: Todo el código fuente, junto con los Dockerfiles y la documentación (README.md), se gestionó utilizando el sistema de control de versiones Git. El repositorio se alojó en GitHub para facilitar el acceso público y el seguimiento de cambios [6].

Distribución de Imágenes con Docker Hub: Las imágenes de Docker construidas a partir de los Dockerfiles se publicaron en Docker Hub. Esto permite que cualquier persona con Docker instalado pueda descargar y ejecutar las aplicaciones con un único comando, garantizando que funcionen de manera idéntica en cualquier entorno [7].

B. Especificación y Despliegue de Proyectos

Se detallan a continuación las particularidades de cada uno de los cuatro proyectos desarrollados.

1. Simulación de Vehículo Seguidor de Línea:

Este proyecto se originó a partir de un código base que simulaba un vehículo siguiendo una pista simple en forma de L. A continuación, se describe el funcionamiento de ese algoritmo inicial y las mejoras sustanciales que fueron implementadas.

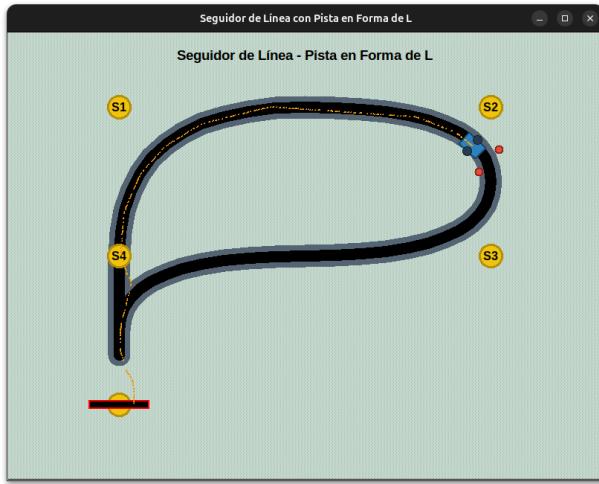
a) Análisis del Algoritmo Base: El código original (carrito.py) presentaba las siguientes características:

Entorno Gráfico: Utilizaba la librería Tkinter de Python para renderizar la pista y el vehículo.

Diseño de Pista: Generaba una pista estática en forma de L con cinco puntos de control o "paradas".

Lógica de Control: El vehículo usaba tres puntos de sensado (izquierdo, derecho y central) para detectar una línea guía. El control se basaba en un autómata de estados ('left', 'right', 'center', 'both', 'none') que alimentaba a un controlador **PID (Proporcional-Integral-Derivativo)** para calcular el ángulo de giro.

Objetivo: El vehículo se desplazaba por la pista hasta alcanzar una barra de detención al final del recorrido, momento en el cual la simulación se detenía.



b) Personalización y Mejoras Implementadas: El código base fue extensamente modificado para crear una simulación más desafiante, robusta y con una identidad visual propia. Las principales novedades del código final ([carrito_pista_desafiante.py](#)) son:

- Diseño de Pista Avanzado:** Se reemplazó la pista en L por un circuito cerrado **complejo inspirado en una pista de Fórmula 1**, con 25 curvas definidas. Se utilizó el suavizado de líneas (`smooth=True`) de Tkinter para crear un trazado fluido y orgánico, lo que representa un desafío de control mucho mayor.
- Lógica de Control Refinada:** Se simplificó y robusteció drásticamente la lógica de los sensores. En lugar de un autómata de estados, el sistema ahora genera un **error numérico directo** (`-3, 3, 0`). Si el vehículo pierde la línea, utiliza el último error registrado para decidir la dirección de búsqueda, un método más eficiente que el comportamiento aleatorio del código original.
- Ajuste de Parámetros PID:** Las constantes del controlador PID (`Kp, Ki, Kd`) fueron **recalibradas** de manera significativa para adaptar el comportamiento

del vehículo a la alta velocidad y a las curvas cerradas de la nueva pista, mejorando la estabilidad y reduciendo las oscilaciones.

- **Funcionalidad de Vuelta Completa:** Al ser un circuito cerrado, el objetivo ya no es detenerse al final. Ahora, el vehículo debe **completar una vuelta entera**, detectando el paso por una línea de meta con un diseño de bandera a cuadros. Al lograrlo, se despliega el mensaje "**¡VUELTA COMPLETA!**".

- **Mejoras Visuales y de Presentación:** Se renovó por completo la estética del proyecto. Se implementó una nueva paleta de colores (vehículo rojo, pista gris sobre fondo verde) y se añadieron textos informativos directamente sobre el lienzo, como el título de la pista y el nombre del autor, otorgando al proyecto una presentación más profesional y personalizada.



El resultado final es la transformación de una demostración básica a una simulación de carreras personalizada y técnicamente más avanzada, lista para su despliegue en un contenedor Docker.

c) Containerización para Despliegue Reproducible: Para garantizar que la simulación pudiera ejecutarse en cualquier sistema anfitrión con Docker instalado, independientemente de sus librerías gráficas o versiones de Python, se procedió a su containerización. El proceso se definió mediante un archivo Dockerfile, cuyo contenido se detalla a continuación.

El Dockerfile orquesta la construcción de una imagen autocontenido. Primero, se define una imagen base oficial y ligera (`FROM python:3.9-slim`), lo que proporciona un entorno Python limpio y minimiza el tamaño final de la imagen.

El paso más crítico fue la gestión de dependencias. La instrucción `RUN apt-get update && apt-get install -y tk` es fundamental, ya que instala la librería de sistema Tcl/Tk, una

dependencia externa requerida por el módulo tkinter de Python para poder renderizar la interfaz gráfica. Este comando demuestra la capacidad de Docker para gestionar no solo dependencias de lenguaje, sino también de sistema operativo.

```
GNU nano 7.2
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ ls
Usa una imagen base de Python. La versión 'slim' es más ligera.
FROM python:3.9-slim

# Instala las dependencias necesarias para tkinter.
# 'tk' es la librería gráfica que Python necesita.
RUN apt-get update && apt-get install -y tk

# Establece el directorio de trabajo dentro del contenedor.
WORKDIR /app

# Copia tu script de Python al directorio de trabajo del contenedor.
COPY carrito_pista_desafiante.py .

# Define el comando que se ejecutará cuando inicie el contenedor.
CMD ["python3", "carrito_pista_desafiante.py"]
```

Posteriormente, se establece un directorio de trabajo (WORKDIR /app) y se copia el script de la simulación (COPY carrito_pista_desafiante.py .) dentro de la imagen. Finalmente, la instrucción CMD ["python3", "carrito_pista_desafiante.py"] define el comando por defecto que se ejecutará al iniciar un contenedor, lanzando así la aplicación.

```
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ ls
assets carrito_pista_desafiante.py carrito.py Dockerfile
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ docker build -t f1-car-app .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
Install the buildx component to build images with BuildKit:
https://docs.docker.com/go/buildx

Sending build context to Docker daemon 1.818MB
Step 1/5 : FROM python:3.9-slim
--> 1be4b628ef55
Step 2/5 : RUN apt-get update && apt-get install -y tk
--> Using cache
--> 0f0f5fe36a61f6
Step 3/5 : WORKDIR /app
--> Using cache
--> b779326fbba5
Step 4/5 : COPY carrito_pista_desafiante.py .
--> Using cache
--> 3e0ff8984852
Step 5/5 : CMD ["python3", "carrito_pista_desafiante.py"]
--> Using cache
--> 1ca0a9fb04a8
Successfully built dd4dfaafbf04a
Successfully tagged f1-car-app:latest
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ nano Dockerfile
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ ]
```

La ejecución final del contenedor requirió superar el desafío de conectar la interfaz gráfica del contenedor con el servidor de video del host. Esto se logró con el comando docker run, utilizando los flags --network="host" para compartir la pila de red y -e DISPLAY=\$DISPLAY para dirigir la salida de video, una solución robusta para el despliegue de aplicaciones gráficas containerizadas.

```
--> 0d40f8a81094a
Successfully built dd4dfaafbf04a
Successfully tagged f1-car-app:latest
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ nano Dockerfile
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ xhost +local:docker
non-network local connections being added to access control list
gixer155@gixer155-Latitude-3440:~/Corte_3/TAREAS_5/Carrito_Desafante_Carpeta$ docker run
-it --rm --network="host" -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix --name f1-simulation f1-car-app
```

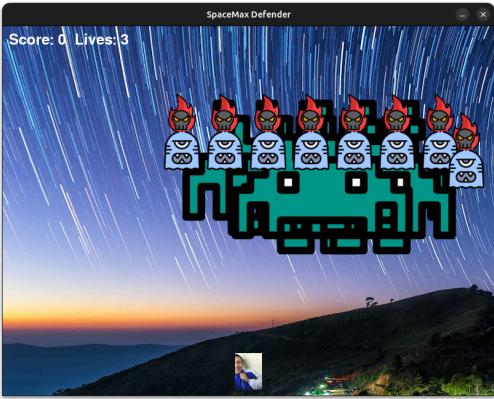
2. Videojuego de Naves Espaciales:

El segundo proyecto consistió en la evolución de un videojuego. Se partió de un prototipo funcional con mecánicas clásicas (Juegos_Naves_Espaciales.py) y se lo transformó en una experiencia de juego más completa, robusta y dinámica (Juegos_Naves_Espaciales_Mejorado.py).

Se trata de un juego clásico de naves espaciales desarrollado con Pygame. La "esencia propia" se añadió mediante la modificación de los sprites (gráficos) de la nave del jugador y de los enemigos, la implementación de un sistema de puntuación acumulativo y el aumento progresivo de la dificultad. La containerización fue más directa que en el caso anterior, ya que las dependencias de Pygame se manejan fácilmente a través del gestor de paquetes pip dentro del Dockerfile.

a) Análisis del Prototipo Inicial: El código base era una implementación minimalista del género *shoot 'em up*, inspirado en clásicos como Space Invaders. Su funcionamiento se basaba en:

- **Plataforma y Estructura:** Desarrollado en Pygame, con una estructura de código lineal dentro de un único bucle de juego principal que gestionaba todos los eventos, actualizaciones y renderizado.
- **Mecánicas de Juego:** El jugador controlaba una nave en la parte inferior de la pantalla, con movimiento horizontal y la capacidad de disparar proyectiles hacia arriba. El objetivo era destruir un enjambre de naves enemigas que se movían como un bloque rígido, de lado a lado, descendiendo un paso cada vez que el enjambre alcanzaba un borde de la pantalla.
- **Gestión de Recursos:** La carga de imágenes y sonidos se realizaba directamente. La ausencia de un solo archivo de recurso (**asset**) resultaría en una terminación abrupta del programa por un error no controlado.
- **Limitaciones:** El juego carecía de estados definidos (como menús o pantallas de "Game Over"), música de fondo, y presentaba una dificultad estática. Las mecánicas eran simples, sin cadencia de disparo para el jugador ni capacidad ofensiva por parte de los enemigos.



b) Evolución a un Juego Completo: Mejoras y Personalización: El prototipo fue sometido a una reingeniería integral para dotarlo de profundidad jugable y robustez técnica. Las mejoras clave incluyen:

1. Arquitectura del Bucle Principal: De un Bucle Monolítico a una Máquina de Estados

- **Versión Inicial:** El flujo del programa residía en un único bucle `while running`:. Dentro de este, toda la lógica —manejo de eventos, actualización de estados de todos los objetos, y renderizado— se ejecutaba de manera secuencial y sin distinción de fases. Esta estructura monolítica es simple pero carece de escalabilidad y dificulta la introducción de nuevas fases de juego como menús o finales.
- **Versión Mejorada:** Se refactorizó el bucle principal para implementar una **máquina de estados finitos (FSM)**, controlada por la variable `game_state`. El bucle `while running`: ahora actúa como un despachador que ejecuta bloques de código específicos según el estado actual ('`INTRO`', '`PLAYING`', '`GAME_OVER`'). Este patrón de diseño es superior, ya que:
 - **Encapsula la lógica:** El código para la pantalla de título está completamente separado del código de juego activo.
 - **Mejora la legibilidad y el mantenimiento:** Añadir nuevos estados (ej. una pantalla de pausa) es trivial y no interfiere con la lógica existente.
 - **Controla las transiciones:** El cambio de estado se gestiona explícitamente (ej. `if player.lives <= 0: game_state = 'GAME_OVER'`), resultando en un flujo de programa claro y predecible.

2. Refinamiento del Diseño Orientado a Objetos (Clases `Player` y `Enemy`)

- **Clase `Player`:**

- **Inicial:** La clase era un simple contenedor de datos (`rect`, `lives`, `score`) con un método `update()` que solo gestionaba el movimiento horizontal. La lógica de disparo estaba externalizada en el bucle principal, violando el principio de encapsulamiento.
- **Mejorada:** La clase `Player` asume más responsabilidades. Se añaden atributos para gestionar estados complejos como la cadencia de disparo (`self.last_shot`, `self.shoot_delay`) y la invulnerabilidad (`self.invulnerable`, `self.hide_timer`). Se introducen métodos como `shoot()` y `get_hit()`, que encapsulan la lógica de acción y reacción del jugador. El método `update()` ahora también gestiona los efectos visuales de la invulnerabilidad (parpadeo), centralizando todo el comportamiento del jugador en su propia clase.

- **Clase `Enemy`:**

- **Inicial:** El movimiento era colectivo. La clase `Enemy` solo contenía un `speed` y una `direction`, y su `update()` ejecutaba un simple desplazamiento lateral. El descenso era una lógica global manejada en el bucle principal.
- **Mejorada:** El comportamiento se individualiza. Cada instancia de `Enemy` ahora posee atributos únicos para un **movimiento sinusoidal parametrizado** (`self.horizontal_range`, `self.horizontal_angle`,

`self.horizontal_speed`), que son inicializados con valores aleatorios. Su método `update(self, dy)` fue rediseñado para calcular su posición horizontal individualmente y aceptar el desplazamiento vertical `dy` como un parámetro externo. Esto transforma a los enemigos de un bloque pasivo a una colección de agentes con comportamiento emergente.



3. Gestión de Recursos (Assets) y Robustez

- **Inicial:** Se utilizaban llamadas directas y desprotegidas a `pygame.image.load()` y `pygame.mixer.Sound()`. Esta aproximación es frágil; la ausencia de un solo archivo causa una excepción no controlada (`pygame.error` o `FileNotFoundException`) que detiene la ejecución del programa por completo.
- **Mejorada:** Se implementaron **funciones contenedoras (wrapper functions)**, `load_image()` y `load_sound()`, que encapsulan la lógica de carga dentro de bloques `try...except`. Si la carga falla, en lugar de terminar el programa, se captura la excepción, se notifica al usuario por consola y se retorna un **objeto sustituto** (un `pygame.Surface` de color magenta para las imágenes o una instancia de la clase `DummySound` para los sonidos). Este enfoque de programación defensiva incrementa drásticamente la robustez y la portabilidad de la aplicación.



4. Lógica de Juego y Dinámicas Procedurales

- **Inicial:** La creación de enemigos estaba codificada directamente en el script principal. La dinámica era estática y no cambiaba durante la partida.
- **Mejorada:** Se introdujeron elementos procedurales. La función `create_wave()` abstrae la generación de enemigos, permitiendo reiniciar el enjambre fácilmente entre niveles. Se añadieron variables para controlar la dificultad de forma dinámica (`downward_move_interval`, `enemy_shoot_chance`), las cuales se ajustan al final de cada oleada, creando una curva de dificultad progresiva. La introducción de probabilidad (`random.random() < enemy_shoot_chance`) para el disparo enemigo añade un factor de imprevisibilidad que enriquece la experiencia de juego.

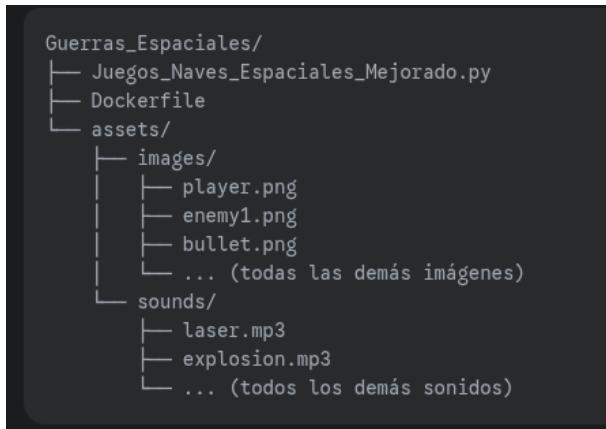


d) Proceso de Containerización para "Guerras Espaciales"

El objetivo es crear una imagen de Docker que contenga el juego, todas sus dependencias (como Pygame) y sus recursos (imágenes y sonidos), para que pueda ser ejecutado con un solo comando en cualquier máquina que tenga Docker.

Paso 1: Organizar la Estructura de Archivos

Para que Docker pueda construir la imagen correctamente, es fundamental que los archivos estén organizados de una manera específica. Asegúrate de que tu directorio del proyecto se vea así:



Paso 2: Creación del Dockerfile

Dentro del directorio `Guerras_Espaciales/`, crea un archivo llamado `Dockerfile` (sin extensión) con el siguiente contenido:

```
2 de jun 02:21 gixer155@gixer155-Latitude-3440: ~/Corte_3
GNU nano 7.2
# Usar una imagen base de Python. Python 3.11 es una buena opción.
FROM python:3.11-slim-bookworm

# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app

# Instalar python3-pip y las dependencias de sistema cruciales para Pygame (librerías SDL)
RUN apt-get update && apt-get install -y \
    python3-pip \
    libbsd12-2.0.0 \
    libbsd12-image-2.0.0 \
    libbsd12-mixer-2.0.0 \
    libbsd12-ttf-2.0.0 \
    # Las siguientes son a veces necesarias para funcionalidades completas de Pygame/SDL
    # y para evitar otros posibles errores de "missing library" más adelante.
    # Si la construcción es muy lenta o falla por alguna de estas, puedes probar comentándolas.
    libportmidi \
    libjpeg-dev \
    libpng-dev \
    libtiff-dev \
    libwebp-dev \
    # ttf-mscorefonts-installer # Este puede requerir interacción, mejor evitarlo si es posible
    libsm \
    libxext \
    ffmpeg \
    && rm -rf /var/lib/apt/lists/*

# Ahora, instalar pygame usando pip
RUN pip3 install pygame

# Copiar todos los archivos del directorio actual (donde está el Dockerfile)
# al directorio de trabajo /app dentro del contenedor.
# Esto incluye tu script .py y la carpeta 'assets'.
COPY . /app

# Comando para ejecutar el juego cuando el contenedor se inicie
CMD ["python3", "Juegos_Naves_Espaciales_Mejorado.py"]
```

Análisis Técnico del Dockerfile:

`FROM python:3.9-slim`: Inicia desde un sistema operativo Linux mínimo con Python 3.9 preinstalado.

`RUN pip install pygame`: Este es el comando clave. Ejecuta el instalador de paquetes de Python (pip) para descargar e instalar la librería Pygame dentro del entorno del contenedor.

`WORKDIR /app`: Fija el directorio de trabajo a `/app`. Todos los comandos siguientes se ejecutarán desde esta ruta dentro del contenedor.

`COPY ./assets ./assets`: Copia la carpeta `assets` de tu máquina local a la carpeta `/app/assets` dentro del contenedor. Es fundamental para que las rutas relativas del código (`os.path.join('assets', ...)`) sigan funcionando.

`COPY Juegos_Naves_Espaciales_Mejorado.py ./`: Copia el script del juego a `/app/`.

`CMD [...]`: Especifica el comando por defecto para iniciar el juego cuando se ejecute el contenedor.

Paso 3: Construir la Imagen de Docker

Abre una terminal en el directorio `Guerras_Espaciales/` y ejecuta el siguiente comando:

```
Bash
docker build -t guerras-espaciales .
```

`docker build`: Es el comando para construir una imagen a partir de un Dockerfile.

`-t guerras-espaciales`: La opción `-t` (de "tag") le asigna un nombre legible a tu imagen. En este caso, la hemos llamado `guerras-espaciales`.

`./`: El punto al final es muy importante. Le indica a Docker que el contexto de la construcción (donde se encuentra el Dockerfile y los archivos a copiar) es el directorio actual.

Paso 4: Ejecutar el Contenedor del Juego

Debido a que Pygame crea una ventana gráfica, necesitamos conectar la pantalla del contenedor con la de tu máquina (host), un proceso conocido como **X11 forwarding**.

```
Bash
docker run -it --rm \
    -e DISPLAY=$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    --device /dev/snd \
    --name="guerras-espaciales-game" \
    guerras-espaciales
```

-it --rm: Ejecuta el contenedor en modo interactivo (**-it**) y lo elimina automáticamente al cerrar el juego (**--rm**).

-e DISPLAY=\$DISPLAY y **-v /tmp/.X11-unix:/tmp/.X11-unix**: Son los parámetros clave para el reenvío de video, permitiendo que la ventana de Pygame aparezca en tu escritorio.

--device /dev/snd: Este flag es **importante para el audio**. Conecta el dispositivo de sonido de tu máquina al contenedor, permitiendo que los efectos de sonido y la música del juego se reproduzcan correctamente.

--name="guerras-espaciales-game": Asigna un nombre a la instancia del contenedor en ejecución.

guerras-espaciales: Es el nombre de la imagen que construiste en el paso anterior y que quieras ejecutar.

3.Simulación y Containerización de Sistemas Robóticos con ROS:

El tercer pilar de este trabajo se centró en el modelado, simulación y despliegue de sistemas robóticos utilizando el Sistema Operativo para Robots (ROS). Se abordó una secuencia de proyectos de complejidad creciente para demostrar la versatilidad de ROS y la viabilidad de su containerización.

a) Proyecto Fundacional: Brazo Robótico de 3 Grados de Libertad (GDL). Como punto de partida, se desarrolló un brazo robótico simple de 3 GDL. Este proyecto sirvió para establecer las bases del flujo de trabajo en ROS. Su implementación consistió en la creación de un archivo **URDF (Unified Robot Description Format)** para describir la estructura cinemática del brazo: una base rotatoria, una articulación de hombro y una de codo. El control se implementó mediante un nodo de Python que publicaba los ángulos deseados para cada articulación en el tópico **/joint_states**. La visualización y la depuración del modelo se realizaron exclusivamente en **RViz**, la herramienta de visualización 3D de ROS, que permitió verificar la correcta definición de los eslabones, las articulaciones y los sistemas de coordenadas del robot sin la sobrecarga computacional de una simulación física.



Descripción de Componentes

El sistema se compone de dos nodos principales de ROS2.

Nodo de Simulación (sim_node.py) Este nodo actúa como el "gemelo digital" del robot y su entorno. Sus responsabilidades son:

Inicialización del Entorno: Utiliza la librería **PyBullet** para crear una instancia de simulación física, estableciendo parámetros como la gravedad. Carga los modelos 3D de un plano y del brazo robótico KUKA iiwa a partir de sus archivos de descripción **URDF**.

Suscripción a Comandos: Crea un suscriptor de ROS2 que escucha en el tópico **/kuka_joint_commands**. Este tópico está definido para recibir mensajes del tipo **Float64MultiArray**, que contienen una lista de 7 valores angulares (radianes).

Actuación en Simulación: Al recibir un mensaje, la función de callback del suscriptor extrae las posiciones angulares objetivo y las aplica a las articulaciones del brazo simulado mediante la función **p.setJointMotorControl2** de PyBullet.

Avance de la Simulación: Un temporizador interno se ejecuta a 240 Hz para llamar a **p.stepSimulation()**, asegurando que la física del entorno se actualice de manera constante y fluida.

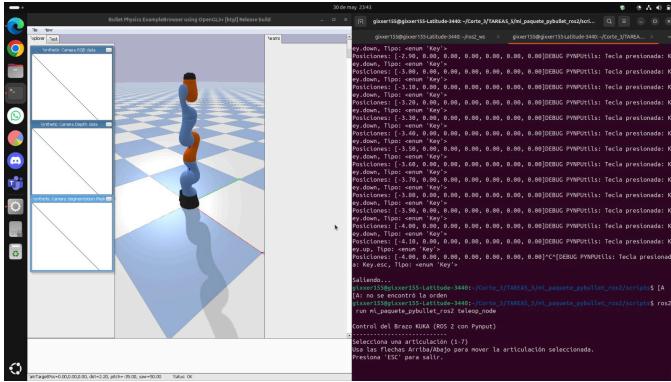
```

if __name__ == '__main__':
    main()
pixxer155@pixxer155-Latitude-3440:~/Corte_3/TAREAS_5/mi_paquete_pybullet_ros2/scripts$ ls
__init__.py sim_node.py teleop_node.py
pixxer155@pixxer155-Latitude-3440:~/Corte_3/TAREAS_5/mi_paquete_pybullet_ros2/scripts$ cat sim_node.py
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
import pybullet as p
import pybullet_data
from std_msgs.msg import Float64MultiArray
import time
# --- NUEVAS IMPORTACIONES DE QOS ---
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy
# --- PERFIL DE QOS DEFINIDO (DEBE SER COMPATIBLE CON EL PUBLICADOR) ---
qos_profile_comandos = QoSProfile(
    reliability=ReliabilityPolicy.RELIABLE,
    history=HistoryPolicy.KEEP_LAST,
    depth=10,
    durability=DurabilityPolicy.VOLATILE
)
class PybulletKukaNode(Node):
    def __init__(self):
        super().__init__('pybullet_kuka_simulator')
        self.get_logger().info('Iniciando simulación en PyBullet...')
        try:
            self.physics_client = p.connect(p.GUI)
        except p.error as e:
            self.get_logger().error(f'No se pudo conectar al servidor GUI de PyBullet: {e}')
            self.get_logger().info("Intentando conectar en modo DIRECT (sin GUI)...")
        try:
            self.physics_client = p.connect(p.DIRECT)
            self.get_logger().info("Conectada a PyBullet en modo DIRECT.")
        except p.error as e_direct:
            self.get_logger().fatal(f'No se pudo conectar a PyBullet en modo DIRECT tampoco: {e_direct}')
            rclpy.shutdown()
            return
        p.setAdditionalSearchPath(pybullet_data.getDataPath())
        p.setGravity(0, 0, -9.81)
        p.loadURDF("plane.urdf")
        # --- Configuración de los servos ---

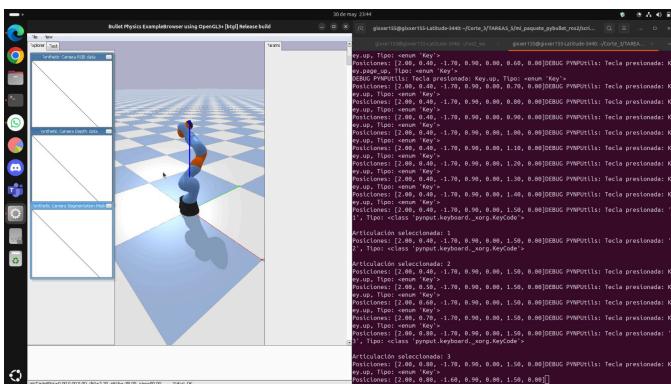
```

Nodo de Teleoperación (teleop_node.py): Este nodo sirve como la interfaz hombre-máquina (HMI), traduciendo las acciones del usuario en comandos para el robot.

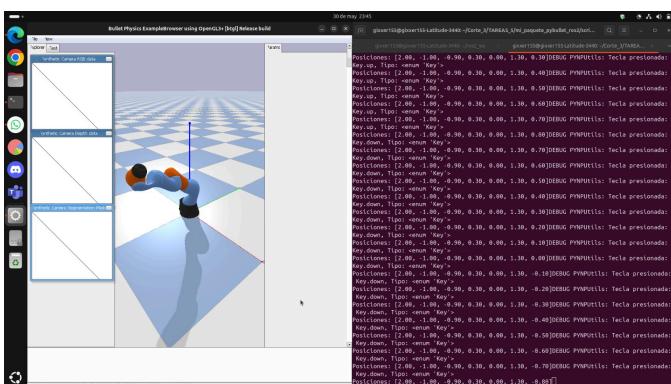
Captura de Entrada: Utiliza la librería `Pynput` para capturar las pulsaciones del teclado. Para no bloquear el procesamiento de ROS2, el listener de teclado se ejecuta en un hilo de ejecución (thread) independiente.



Procesamiento de Comandos: El usuario puede seleccionar una de las 7 articulaciones (teclas '1' a '7') y modificar su ángulo objetivo con las flechas de 'Arriba' y 'Abajo'. Las posiciones de todas las articulaciones se almacenan en una lista interna.



Publicación de Comandos: Un temporizador periódico (ejecutado a 10 Hz) toma la lista de posiciones articulares, la empaqueta en un mensaje `Float64MultiArray` y la publica en el tópico `/kuka_joint_commands`.



Mecanismo de Comunicación: Calidad de Servicio (QoS)

Un aspecto técnico fundamental de esta implementación es el manejo explícito de la **Calidad de Servicio (QoS)**, una característica central de ROS2.

Tópico: El tópico `/kuka_joint_commands` actúa como el canal de comunicación abstracto entre los nodos.

Perfil de Fiabilidad: Tanto el publicador (en `teleop_node.py`) como el suscriptor (en `sim_node.py`) se configuran con un perfil de QoS que especifica una **política de fiabilidad RELIABLE**. Esto instruye a ROS2 a garantizar la entrega de cada mensaje. Para una aplicación de control robótico, esta configuración es crítica, ya que la pérdida de un comando de movimiento podría resultar en un comportamiento inesperado o inseguro del robot.

Despliegue y Containerización

Para asegurar la portabilidad y la facilidad de despliegue, todo el sistema fue encapsulado en un contenedor Docker.

Dockerfile: Se creó un `Dockerfile` a partir de una imagen base de ROS2 (ej. `ros:humble-desktop`). Este archivo automatiza la instalación de todas las dependencias del sistema y de Python (como `pybullet` y `pynput`).

Lanzamiento: Dentro del contenedor, un archivo de lanzamiento de ROS2 (`.launch.py`) se encarga de ejecutar ambos nodos (`sim_node.py` y `teleop_node.py`) de forma concurrente.

Visualización: La ejecución del contenedor se configura con reenvío **X11 forwarding**, permitiendo que la ventana gráfica de la simulación de PyBullet, generada dentro del contenedor, se muestre directamente en el escritorio del sistema anfitrión.

```

#!/bin/bash
# Usar una imagen base oficial de ROS 2 Jazzy (más minimalista)
FROM ros:jazzy

# Evitar preguntas interactivas durante la instalación de paquetes
ENV DEBIAN_FRONTEND=noninteractive

# Instalar dependencias del sistema:
# - python3-pip: para instalar paquetes de Python
# - x11-apps: utilidades X11, bueno para probar la GUI
# - libgl1, libegl1, mesa-utils: Librerías gráficas estándar para OpenGL/GUI
# - python3-pynput: para el control por teclado
RUN apt-get update & apt-get install -y \
    python3-pip \
    x11-apps \
    libgl1 \
    libegl1 \
    mesa-utils \
    python3-pynput \
    && rm -rf /var/lib/apt/lists/*

# Instalar PyBullet usando pip3
COPY ./mi_paquete_pybullet_ros2 ./src/mi_paquete_pybullet_ros2

# Crear y establecer el directorio de trabajo para el workspace de ROS 2
WORKDIR /ros2_ws

# Copiar tu paquete ROS 2 al directorio 'src' del workspace
COPY ./mi_paquete_pybullet_ros2 ./src/mi_paquete_pybullet_ros2

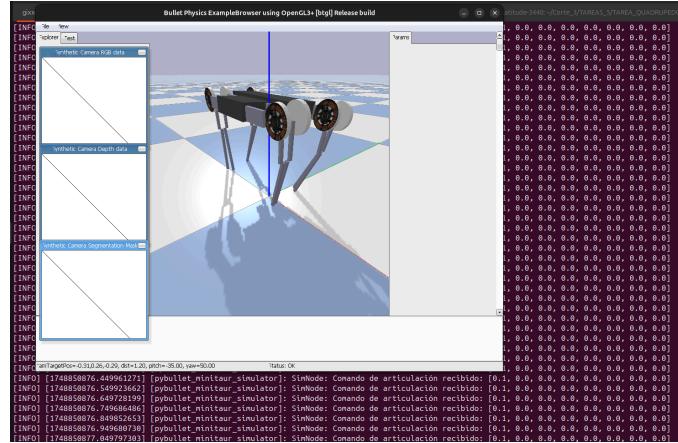
# Construir el workspace de ROS 2
# El comando './opt/ros/jazzy/setup.sh' carga el entorno de ROS antes de compilar
RUN ./opt/ros/jazzy/setup.sh && colcon build --symlink-install

# Configurar el entorno para que se cargue al iniciar un contenedor
# Y establecer el comando por defecto a bash para poder ejecutar comandos manualmente
CMD ["bash"]

```

b) Incremento de Complejidad: Robot Bípedo. (FALLIDO) El siguiente paso fue el diseño y simulación de un robot bípedo. Este sistema representa un salto cualitativo en complejidad debido a los desafíos inherentes al equilibrio y la locomoción dinámica. Se definió un modelo URDF más complejo para el bípedo y se desarrolló un nodo de control central encargado de la **generación de patrones de marcha** (*gait generation*). Este nodo calculaba las trayectorias periódicas de las articulaciones de las piernas para producir un ciclo de caminata estable. A diferencia del brazo robótico, la validación de este sistema exigió el uso del simulador físico **Gazebo**. La simulación en Gazebo fue crucial para probar la interacción del robot con la gravedad, las fuerzas de contacto con el suelo y, en última instancia, su capacidad para mantener el equilibrio dinámico durante la marcha.

c) Sistema Avanzado: Robot Cuadrúpedo. El proyecto culminó con el desarrollo de un robot cuadrúpedo, el sistema más complejo de los tres. La locomoción cuadrúpeda, aunque más estable que la bípeda, requiere una coordinación precisa de múltiples cadenas cinemáticas. El control de este robot se basó en un enfoque jerárquico: un nodo de alto nivel definía la trayectoria del cuerpo y seleccionaba el tipo de marcha (ej. trote), mientras que los nodos de bajo nivel utilizaban **cinemática inversa** para calcular los ángulos articulares necesarios para posicionar cada pie en el punto deseado del espacio. La simulación en Gazebo fue, de nuevo, indispensable para validar la coordinación entre las cuatro patas y la estabilidad del robot en movimiento.



1. sim_quadrupedo.py - El Nodo de Simulación

Este script es el corazón de la simulación. Se encarga de:

Crear un mundo virtual usando PyBullet.

Cargar el modelo del robot cuadrúpedo Minitaur.

Escuchar los comandos de movimiento que vienen de otros nodos ROS 2.

Aplicar esos comandos al robot en la simulación.

Publicar el estado actual de las articulaciones del robot.

```
Python

#!/usr/bin/env python3
import rclpy # Librería principal de ROS 2 para Python
from rclpy.node import Node # Clase base para crear un nodo
import pybullet as p # Librería de simulación física
import pybullet_data # Datos y modelos para PyBullet (como el suelo, robots)
from std_msgs.msg import Float64MultiArray # Tipo de mensaje para enviar arrays de
from sensor_msgs.msg import JointState # Tipo de mensaje para publicar el estado de
import time
import math
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy

# Nombres exactos de los 8 motores del Minitaur que queremos controlar.
# Estos nombres deben coincidir con los definidos en el archivo URDF del robot.
ACTUAL_MINITAUR_MOTOR_JOINT_NAMES = [
    "motor_front_rightR_joint", "motor_front_rightL_joint",
    "motor_back_rightR_joint", "motor_back_rightL_joint",
    "motor_front_leftL_joint", "motor_front_leftR_joint",
    "motor_back_leftL_joint", "motor_back_leftR_joint"
]

# Perfil de Calidad de Servicio (QoS) para la comunicación.
# BEST_EFFORT es más flexible si la red no es perfecta. RELIABLE es más estricto.
qos_profile_communications = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    history=HistoryPolicy.KEEP_LAST,
    depth=10,
    durability=DurabilityPolicy.VOLATILE
)
```

```
# Función principal que inicia el nodo ROS 2
def main(args=None):
    rclpy.init(args=args) # Inicializar ROS 2
    node = PyBulletMinitaurNode() # Crear una instancia de nuestro nodo
    if p.isConnected(node.physics_client): # Solo si PyBullet se conectó bien
        try:
            rclpy.spin(node) # Mantiene el nodo vivo, procesando callbacks (suscripciones)
        except KeyboardInterrupt: # Si se presiona Ctrl+C
            node.get_logger().info('Cerrando por KeyboardInterrupt')
        finally: # Limpieza al cerrar
            if p.isConnected(node.physics_client): p.disconnect(node.physics_client)
            if hasattr(node, 'is_valid') and node.is_valid(): node.destroy_node()
            if rclpy.ok(): rclpy.shutdown()
    else: # Si PyBullet no se pudo conectar
        node.get_logger().error("No se pudo inicializar PyBullet. Terminando nodo.")
        if rclpy.ok():
            if hasattr(node, 'is_valid') and node.is_valid(): node.destroy_node()
            rclpy.shutdown()

    if __name__ == '__main__':
        main() # Ejecutar la función principal
```

2. teleop_quadrupedo.py - El Nodo de Control por Teclado

Este script te permite:

Seleccionar uno de los 8 motores principales del Minitaur usando las teclas numéricas (1-8).

Aumentar o disminuir el ángulo objetivo para el motor seleccionado usando las teclas de flecha arriba/abajo.

Publica continuamente el array completo de los 8 ángulos objetivo al tópico /minitaur_joint_commands.

```

#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float64MultiArray # Mismo tipo de mensaje que el suscriptor
import sys
from pynput import keyboard # Librería para leer el teclado de forma avanzada
import threading
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy, DurabilityPolicy

# Nombres de los motores, DEBEN COINCIDIR EN ORDEN con los que sim_quadrupedo usa
MOTOR_NAMES = [
    "motor_front_rightR_joint", "motor_front_rightL_joint",
    "motor_back_rightR_joint", "motor_back_rightL_joint",
    "motor_front_leftR_joint", "motor_front_leftL_joint",
    "motor_back_leftR_joint", "motor_back_leftL_joint"
]

# Texto de instrucciones que se muestra en la terminal
instructions_header = """
Control del Minitaur (ROS 2 con Pynput) - 8 Motores
-----
Selecciona un motor (1-8) y luego usa Flechas Arriba/Abajo.
Presiona 'ESC' para salir.

Motores (1-8):
"""
motor_instructions = "\n".join(["{} {}".format(i+1, name) for i, name in enumerate(MOTOR_NAMES)])
instructions = instructions_header + motor_instructions

```

```

def on_key_press(self, key):
    # Esta función es llamada por pynput cada vez que se presiona una tecla
    # print(f"DEBUG PYNPUT: Tecla presionada: {key}, Tipo: {type(key)}") # !!!!!
    if not self.running: return False # Si se indicó salir, no hacer nada y detener

    # Intentar leer la tecla como un carácter (para los números 1-8)
    if hasattr(key, 'char') and key.char is not None:
        if '1' <= key.char <= str(self.num_motors): # Si es un número entre 1 y 8
            self.selected_motor_index = int(key.char) - 1 # Seleccionar el motor
            print(f"\nControlando Motor {self.selected_motor_index + 1}: {MOTOR_NAMES[self.selected_motor_index]}")
        # Si no fué un carácter, o no fue un número válido, ver si es una tecla especial
    elif key == keyboard.Key.up: # Flecha arriba
        self.joint_positions[self.selected_motor_index] += 0.1 # Aumentar ángulo
    elif key == keyboard.Key.down: # Flecha abajo
        self.joint_positions[self.selected_motor_index] -= 0.1 # Disminuir ángulo
    elif key == keyboard.Key.esc: # Tecla Escape
        print("\nSaliendo...")
        self.running = False # Indicar que el nodo debe parar
        return False # Detener el listener de pynput

    # Actualizar la linea de Posiciones en la terminal después de cada acción
    sys.stdout.write("\rPosiciones: [{} ".format(self.joint_positions[0]))
    sys.stdout.flush()
    return True # Mantener el listener activo (a menos que se presione Esc)

def publish_commands_periodically(self):
    # Esta función es llamada por el timer de ROS 2
    if not self.running: # Si se indicó salir, no publicar
        if rclpy.ok(): pass # Podríamos añadir lógica de cierre más robusta aquí
        return

    # Función principal que inicia el nodo ROS 2
def main(args=None):
    rclpy.init(args=args)
    teleop_node = MinitaurTeleopNode()
    try:
        # Este bucle mantiene el nodo vivo y permite que los callbacks de ROS (timers y la lógica de salida (self.running) funcionen.
        while rclpy.ok() and teleop_node.running:
            rclpy.spin_once(teleop_node, timeout_sec=0.1) # Procesar eventos de ROS
    except KeyboardInterrupt: # Si se presiona Ctrl+C en esta terminal
        teleop_node.get_logger().info('Cerrando por KeyboardInterrupt')
    finally: # Limpieza al cerrar
        teleop_node.running = False # Asegurar que el listener de pynput sepa que el nodo ya no existe
        if teleop_node.key_listener.is_alive():
            teleop_node.key_listener.stop()
        # Esperar un poco a que el hilo del listener termine
        if hasattr(teleop_node, 'listener_thread') and teleop_node.listener_thread.is_alive():
            teleop_node.listener_thread.join(timeout=0.2)

        if rclpy.ok():
            # Comprobar si el nodo aún es válido antes de intentar destruirlo
            if hasattr(teleop_node, 'is_valid') and teleop_node.is_valid():
                try:
                    teleop_node.destroy_node()
                except Exception: pass # Evitar error si ya se destruyó por otra vía
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

Flujo de Trabajo General:

teleop_quadrupedo.py:

Se inicia.

Espera a que presiones teclas.

Cuando presionas '1'-'8', selecciona un motor.

Cuando presionas flechas, cambia el ángulo objetivo para el motor seleccionado en su lista interna self.joint_positions.

Cada 0.1 segundos, publica el contenido completo de self.joint_positions (un array de 8 números) en el tópico /minitaur_joint_commands.

sim_quadrupedo.py:

Se inicia y carga el Minitaur en PyBullet.

Identifica los 8 motores principales del URDF y guarda sus índices.

Se suscribe al tópico /minitaur_joint_commands.

Cuando llega un mensaje de teleop_quadrupedo (el array de 8 ángulos), la función command_callback se ejecuta.

Esta función actualiza self.current_joint_targets con los nuevos ángulos.

Un temporizador en sim_quadrupedo llama a simulation_step_callback muy rápidamente (240 veces por segundo).

En simulation_step_callback, se leen los self.current_joint_targets y se usa p.setJointMotorControl2 para aplicar esos ángulos (con una cierta fuerza y ganancias) a cada uno de los 8 motores en la simulación PyBullet.

También publica el estado real de las articulaciones del robot (leído de PyBullet) en /minitaur_joint_states.

El objetivo es empaquetar tu código ROS 2, junto con todas sus dependencias (PyBullet, Pynput, librerías del sistema, etc.), en una imagen de Docker. Esta imagen luego se puede usar para ejecutar tu aplicación en contenedores de forma aislada y portable.

```

gixxer155@gixxer155-Latitude-3440:~/Corte_3/TAREAS... x gixxer155@gixxer155-Latitude-3440:~/Corte_3/TAREAS... x gixxer155@gixxer155-Latitude-3440:~/Corte_3/TAREAS...
GNU nano 7.2
Usar una imagen base oficial de ROS 2 Jazzy (más genérica)
FROM ros:jazzy
# Evitar preguntas interactivas durante la instalación de paquetes
ENV DEBIAN_FRONTEND=noninteractive

# Instalar dependencias del sistema
RUN apt-get update && apt-get install -y \
    python3-pip \
    x11-apps \
    libglib1 \
    libgegl1 \
    mesa-utils \
    python3-pynput \
    && rm -rf /var/lib/apt/lists/*

# Instalar PyBullet usando pip3
RUN pip3 install pybullet --break-system-packages

# Crear y establecer el directorio de trabajo para el workspace de ROS 2
WORKDIR /ros2_ws

# ... CAMBIO IMPORTANTE AQUÍ ...
# 1. Crear explícitamente el directorio src dentro del workspace
RUN mkdir src

# 2. Copiar tu paquete ROS 2 al directorio 'src' que acabamos de crear
# La ruta de destino es relativa al WORKDIR (/ros2_ws), por lo que será /ros2_ws/src/mi_paquete_quadrupedo_ros2
COPY ./mi_paquete_quadrupedo_ros2 ./src/mi_paquete_quadrupedo_ros2
# ... FIN DEL CAMBIO IMPORTANTE ...

# Construir el workspace de ROS 2
RUN ./opt/ros/jazzy/setup.sh && colcon build --symlink-install

# Comando por defecto
CMD ["bash"]

```

Una vez que tienes tu Dockerfile y tu carpeta de paquete (mi_paquete_quadrupedo_ros2) en el mismo directorio (ej. TAREA_QUADRUPEDO_DOCKER), construyes la imagen con:

```

Bash

cd ~/Corte_3/TAREAS_5/TAREA_QUADRUPEDO_DOCKER
docker build -t ros2_quadrupedo_app .

```

Ahora que la imagen está construida, puedes lanzar tu aplicación. Como tienes dos nodos ROS 2 (**sim_quadrupedo** y **teleop_quadrupedo**) y uno de ellos (el simulador) tiene una interfaz gráfica, necesitas:

- **Dos contenedores separados**, uno para cada nodo.
- **Configuración especial para la GUI** y la comunicación de red.

Los comandos que usamos fueron:

Para el Simulador (**sim_quadrupedo_gui**):

```

Bash

# (Después de xhost + en tu máquina anfitriona si estás en Linux)
docker run -it --rm --name sim_quadrupedo_gui \
--net=host \
-e DISPLAY=$DISPLAY \
-e ROS_DOMAIN_ID=0 \
-e ROS_LOCALHOST_ONLY=1 \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-v $HOME/.Xauthority:/root/.Xauthority:rw \
-v /dev/dri:/dev/dri \
ros2_quadrupedo_app \
bash -c "source install/setup.bash && ros2 run mi_paquete_quadrupedo_ros2 sim"

```

Para el Control por Teclado (**teleop_quadrupedo_control**):

```

Bash

docker run -it --rm --name teleop_quadrupedo_control \
--net=host \
-e DISPLAY=$DISPLAY \
-e ROS_DOMAIN_ID=0 \
-e ROS_LOCALHOST_ONLY=1 \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-v $HOME/.Xauthority:/root/.Xauthority:rw \
ros2_quadrupedo_app \
bash -c "source install/setup.bash && ros2 run mi_paquete_quadrupedo_ros2 teleop_quadrupedo_control"

```

d) Proceso Unificado de Despliegue con Docker: Una vez validados funcionalmente, los tres sistemas robóticos fueron encapsulados en contenedores Docker para garantizar un despliegue universal y reproducible. Se estandarizó un **Dockerfile** que utilizaba una imagen base oficial de ROS (**ros:noetic-desktop-full**), la cual incluye el ecosistema completo de ROS, Gazebo y RViz. El proceso de construcción consistió en copiar el espacio de trabajo del proyecto (**catkin_ws**) al interior del contenedor, instalar sus dependencias específicas mediante **rosdep**, compilar el código fuente con **catkin_build**, y configurar el entorno mediante el **source** del archivo **setup.bash**.

IV.CONCLUSIONES

La realización de la TAREA 5 ha permitido una exploración profunda y práctica de diversos algoritmos y tecnologías, abarcando desde la creación de interfaces gráficas simples y juegos, hasta la simulación y control de robots utilizando ROS 2, con un énfasis crucial en el despliegue de estas aplicaciones mediante Docker. A continuación, se detallan las principales conclusiones derivadas de este proceso:

Exploración y Adaptación de Algoritmos:

Se demostró la capacidad de modificar y extender algoritmos existentes, como se evidenció en el circuito seguidor de línea en Tkinter, donde una pista simple fue transformada en un circuito cerrado con múltiples curvas complejas, requiriendo ajustes en la lógica de control del vehículo.

En el juego de naves espaciales con Pygame, se logró incorporar una "esencia propia" mediante la implementación de un sistema de olas progresivas, aumentando la dificultad con enemigos más rápidos y patrones de movimiento más caóticos, lo que enriqueció la jugabilidad original.

Comprensión y Aplicación de ROS 2:

Se adquirió una comprensión fundamental de los conceptos clave de ROS 2, incluyendo nodos, tópicos, mensajes y perfiles de Calidad de Servicio (QoS). Esto se materializó en el desarrollo de una aplicación para controlar un robot cuadrúpedo (Minitaur) simulado en PyBullet.

Se implementó con éxito la comunicación entre dos nodos ROS 2: un nodo de teleoperación (`teleop_quadrupedo.py`) que captura la entrada del teclado (usando `pynput` para mayor robustez) y publica comandos de articulación, y un nodo de simulación (`sim_quadrupedo.py`) que se suscribe a estos comandos y los aplica al modelo del robot en PyBullet.

Se evidenció la importancia de la correcta identificación y mapeo de las articulaciones del robot (URDF) para un control efectivo.

Despliegue en Docker y Gestión de Entornos Complejos:

Una conclusión central del proyecto es la exitosa dockerización de aplicaciones complejas, incluyendo aquellas con interfaces gráficas (PyBullet, Pygame) y comunicación en red (ROS 2). Se crearon Dockerfiles específicos para cada proyecto, gestionando dependencias del sistema (librerías SDL, X11, OpenGL) y de Python (Pygame, PyBullet, Pynput, rclpy).

Se superaron múltiples desafíos técnicos inherentes al despliegue de GUI y aplicaciones ROS 2 en Docker, tales como:

La configuración del reenvío X11 (-e DISPLAY, -v /tmp/.X11-unix, -v \$HOME/.Xauthority).

El acceso a hardware gráfico (-v /dev/dri).

La resolución de problemas de red entre contenedores para ROS 2 (DDS discovery), optando finalmente por --net=host y la configuración explícita de `ROS_DOMAIN_ID` y `ROS_LOCALHOST_ONLY` para asegurar la comunicación.

El manejo de la protección externally-managed-environment (PEP 668) en Ubuntu 24.04 y en las imágenes Docker base, utilizando `apt install python3-<paquete>` cuando fue posible, y `pip3 install --break-system-packages` como alternativa necesaria dentro de Docker.

Se demostró el valor de Docker para crear entornos consistentes y portables, especialmente para proyectos con dependencias complejas como ROS 2, evitando los problemas de configuración que surgieron durante los intentos de instalación nativa en Ubuntu 24.04 (ej. la incompatibilidad de ROS Noetic y la necesidad de migrar a ROS 2 Jazzy).

RECONOCIMIENTOS

Quisiera expresar mi más sincero agradecimiento al Profesor Diego , por su invaluable dedicación y guía a lo largo de este curso y, en particular, durante el desarrollo de las actividades. Su compromiso con la enseñanza, su paciencia para resolver dudas y su capacidad para explicar conceptos complejos fueron fundamentales para superar los desafíos presentados y llevar este proyecto a buen término. Su entusiasmo por la materia ha sido una gran fuente de motivación.

REFERENCIAS

- [9] dialejobv, "Ejemplo de Carro Seguidor de Línea en Tkinter (Sistemas_Operativos)," (Consultado en 2024). [En línea]. Disponible: https://github.com/dialejobv/Sistemas_Operativos/tree/main/2%20Carro_tkinter
- [10] dialejobv, "Ejemplo de Juego de Nave Espacial (Sistemas_Operativos)," (Consultado en 2024). [En línea]. Disponible: https://github.com/dialejobv/Sistemas_Operativos/tree/main/3%20Nave_espacial
- [11] Python Software Foundation, "Python Language Reference, version 3.12," (Consultado en 2024). [En línea]. Disponible: <http://www.python.org>
- [12] Pygame Developers, "Pygame Documentation," (Consultado en 2024). [En línea]. Disponible: <https://www.pygame.org/docs/>
- [13] Open Robotics, "ROS 2 Jazzy Jalisco Documentation," (Consultado en 2024). [En línea]. Disponible: <https://docs.ros.org/en/jazzy/>
- [14] E. Coumans y Y. Bai, "PyBullet Physics SDK," (Consultado en 2024). [En línea]. Disponible: <https://pybullet.org>
- [15] Docker, Inc., "Docker Official Website," (Consultado en 2024). [En línea]. Disponible: <https://www.docker.com>
- [16] M. Tatra, "pynput - Control and monitor input devices," (Consultado en 2024). [En línea]. Disponible: <https://pypi.org/project/pynput/>
- [17] The Tkinter Development Team, "Tkinter Python interface to Tcl/Tk," en *Python Standard Library Documentation*, Python Software Foundation, (Consultado en 2024). [En línea]. Disponible: <https://docs.python.org/3/library/tkinter.html>
- [18] Canonical Ltd., "Ubuntu 24.04 LTS (Noble Numbat) Documentation," (Consultado en 2024). [En línea]. Disponible: <https://ubuntu.com/desktop/docs>
- [19]