UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Desarrollo de un Laboratorio Virtual de Robots Utilizando un Motor Gráfico para Videojuegos

Trabajo Fin de Máster

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Atahualpa Guerrero, Anibal

Tutor/a: Gracia Calandin, Luis Ignacio

CURSO ACADÉMICO: 2022/2023

UNIVERSIDAD POLITÉCNICA DE VALÉNCIA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Máster Universitario en Automática e Informática Industrial

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

"Desarrollo de un Laboratorio Virtual de Robots Utilizando un Motor Gráfico para Videojuegos"

TRABAJO FIN DE MÁSTER

Autor:
Aníbal Atahualpa Guerrero

Tutor:
Luis Ignacio Gracia Calandin

VALÉNCIA, ESPAÑA

septiembre, 2023

Índice general

Índice de figuras

Lista de Scripts

Resumen

En el presente trabajo se documenta el desarrollo de un entorno virtual de simulación en tiempo real para aplicaciones robóticas, que aprovecha, por una parte, las funciones y herramientas del sistema ROS (Robotic Operating System) y el motor gráfico para videojuegos de Unreal Engine 5.

Se realiza una integración entre ROS y Unreal a partir del desarrollo de una API en Python que utiliza la librería ROSLIBPY para conseguir la comunicación en tiempo real entre ambos entornos usando websockets. La API ha sido desarrollada implementando un framework conocido como FASTAPI. El producto del desarrollo se ha nombrado UROS API (Unreal - ROS - API).

El software (UROS API) desarrollado en el presente trabajo se ejecuta gracias a Uvicorn un servidor ASGI (Asynchronous Server Gateway Interface), la API permite gestionar las comunicaciones, también con clientes web, como punto de partida para la transición que los sistemas industriales requieren en relación con los cambios que se producen en la llamada industria 4.0. Para ello, se desarrolló una aplicación web utilizando el framework React.js y Vite que consume la API y los datos que se obtienen de los tópicos de ROS. Esta aplicación web se ha nombrado (UROS APP) y permite interactuar de forma gráfica con la API desarrollada.

UROS API es capaz de almacenar los datos obtenidos de cada tópico en ROS en colecciones dentro de una base de datos NoSQL como MongoDB y UROS APP es capaz de visualizar esta información utilizando Chart.js como librería para gráficos.

El entorno se ha construido a partir de recursos y modelos 3D libres disponibles en línea y en el marketplace de Unreal Engine. Se realizaron las configuraciones de los modelos 3D creando clases con blueprints, que es el lenguaje gráfico de programación en el entorno de Unreal Engine. De esta manera, los robots simulados usan juntas con restricciones físicas para cada unión entre cuerpos rígidos, formando sistemas que describen el comportamiento y movimiento real de los sistemas robóticos que se evalúan en el entorno.

Finalmente Se llevaron a cabo pruebas genéricas de comunicación y control simple de los sistemas, control manual de teleoperaciones en un robot sumo de dos ruedas. Y una tarea de pick and place en un robot industrial con 6 grados de libertad.

Capítulo 1

Objetivos

1.1. General

Desarrollar un entorno de simulación en tiempo real que integre el sistema ROS (Robotic Operating System) con Unreal Engine 5, con el propósito de facilitar la interacción y comunicación entre sistemas robóticos virtuales y entornos gráficos de alta calidad explorar su potencial en aplicaciones industriales y de investigación en el contexto de la industria 4.0.

1.2. Especificos

1. Diseñar y desarrollar una API en Python (UROS API), que permita la comunicación en tiempo real entre el sistema ROS y Unreal Engine 5.
2. Diseñar y desarrollar un entorno 3D para un laboratorio de robotica en Unreal Engine 5, aprovechando recursos libres disponibles en linea y en el marketplace de Unreal Engine.
3. Configurar los modelos 3D para representar de manera precisa las restricciones físicas y el comportamiento de sistemas robóticos en el entorno de simulación.
4. Implementar una aplicación web (UROS APP) utilizando el framework React.js para consumir la API UROS y visualizar los datos obtenidos de los tópicos de ROS.
5. Almacenar la información obtenida de los tópicos de ROS en una base de datos NoSQL, como MongoDB, utilizando UROS API.
6. Realizar pruebas de comunicación y control simple de sistemas robóticos simulados en el entorno, para un robot sumo de dos ruedas y un robot industrial con 6 grados de libertad.
7. Documentar el proceso de desarrollo, integración y pruebas, proporcionando un recurso útil para futuros trabajos y aplicaciones similares.

Capítulo 2

Conceptos Previos

2.1. Robótica

La robótica es una rama de la ingeniería que se enfoca en el diseño, construcción, programación y operación de robots. Los robots son dispositivos automáticos o semiautomáticos que pueden realizar tareas de manera autónoma o controlada por humanos en una variedad de entornos, desde fábricas industriales hasta entornos médicos y espaciales. [1, 2]

2.2. Entorno Virtual y Realidad Virtual

Un entorno virtual es un espacio digital simulado que puede representar un ambiente real o ficticio. En este trabajo, el entorno virtual se refiere al espacio de simulación donde interactúan los sistemas robóticos y los entornos gráficos creados con Unreal Engine 5. [3]

La realidad virtual (RV) es una tecnología que crea entornos digitales simulados que pueden ser explorados y experimentados por usuarios generalmente a través de dispositivos como cascos de RV. Estos entornos pueden ser simulaciones altamente inmersivas que permiten a los usuarios interactuar con objetos y escenarios. [4]

2.3. Tiempo Real, Websockets y DDS

El procesamiento en tiempo real se refiere a la capacidad de un sistema para responder a eventos o entradas en un tiempo concreto, normalmente con un retraso mínimo o imperceptible. En el contexto de este trabajo, la simulación en tiempo real significa que la simulación de sistemas robóticos ocurre de manera sincronizada con el tiempo real, lo que permite la interacción y el control en tiempo real. [5]

Websockets es un protocolo de comunicación bidireccional que permite una comunicación interactiva en tiempo real entre un servidor y un cliente a través de una conexión persistente. Se utiliza en este trabajo para lograr la comunicación en tiempo real entre ROS y Unreal Engine 5. [6, 7]

DDS (Data Distribution Service) es un estándar de comunicación para sistemas distribuidos en tiempo real que permite la transferencia de datos entre componentes de manera eficiente y confiable. DDS es relevante en el contexto de sistemas robóticos y la comunicación entre ellos. [8]

2.4. Simulación y Algoritmos de control

La simulación es una técnica que se utiliza para modelar y emular el comportamiento de sistemas y procesos en un entorno controlado. En el contexto de este trabajo, la simulación se enfoca en modelos digitales de sistemas robóticos que asemejan su funcionamiento a sistemas similares en el mundo real. [9]

Un algoritmo de control es un conjunto de reglas y procedimientos diseñados para guiar el comportamiento de un sistema o dispositivo, como un robot. En este contexto, se utilizan algoritmos de control para dirigir el movimiento y la interacción de los sistemas robóticos simulados. [2]

2.5. Industria 4.0 y Contenedores

La Industria 4.0 se refiere a la cuarta revolución industrial, que implica la digitalización y automatización de procesos de fabricación y producción. En este contexto, se busca la interconexión de sistemas, la recopilación y el análisis de datos en tiempo real y la aplicación de tecnologías avanzadas como la inteligencia artificial y el Internet de las cosas (IoT) para mejorar la eficiencia y la toma de decisiones en la industria. [1]

Los contenedores (Docker, Kubernetes, etc.) son entornos de ejecución ligeros que contienen aplicaciones y sus dependencias, lo que facilita la implementación y el despliegue de software en diferentes entornos. Los contenedores pueden utilizarse para gestionar componentes de software en proyectos que requieran alta escalabilidad, compatibilidad, estabilidad y portabilidad. Esto para la revolución digital de la industria es imprescindible. [10]

2.6. ROS, Rosbridge, Paquete (ROS), UE5, Motor Gráfico

- ROS (Robotic Operating System): Es un sistema operativo de código abierto utilizado comúnmente en la robótica. ROS no es un sistema operativo en el sentido tradicional, sino una plataforma de software que proporciona herramientas y bibliotecas para el desarrollo de aplicaciones robóticas. Está diseñado para facilitar la programación y control de robots, ofreciendo funciones como comunicación entre componentes, control de hardware y simulación. [11]
- Rosbridge: Es una interfaz que permite la comunicación entre ROS y otros sistemas a través de protocolos como JSON y Websockets. [7]
- Paquete (ROS): En el contexto de ROS, un paquete es una unidad de organización de código que puede contener nodos, bibliotecas y recursos relacionados con una funcionalidad específica. Los paquetes se utilizan para estructurar y modularizar el software en ROS. [11]
- UE5 (Unreal Engine 5): UE5 es un motor gráfico de videojuegos de alto rendimiento desarrollado por Epic Games. Se utiliza en la creación de videojuegos, simulaciones interactivas y experiencias de realidad virtual. UE5 es conocido por su capacidad para crear entornos visuales altamente realistas y es ampliamente utilizado en la industria del entretenimiento y en aplicaciones de simulación, incluida la simulación de sistemas robóticos en tiempo real. [12]
- Motor gráfico: Es un software que se utiliza para crear y renderizar gráficos en aplicaciones, como videojuegos y simulaciones. En este trabajo, Unreal Engine 5 se utiliza como el motor gráfico para generar entornos visuales realistas. [12]

2.7. API, Framework, Librería

- API (Interfaz de Programación de Aplicaciones): Una API es un conjunto de reglas y protocolos que permiten que diferentes aplicaciones se comuniquen entre sí. [13, 14]
- Framework (Marco de Trabajo): Un framework es un conjunto de herramientas y bibliotecas que proporciona una estructura para el desarrollo de aplicaciones. En este trabajo se usa React.js y FASTAPI como frameworks para desarrollar la aplicación web y la API respectivamente. [15, 14]
- Librería (JavaScript, Python, etc): Son conjuntos de funciones y utilidades predefinidas que pueden ser utilizadas por programadores para facilitar tareas específicas en la programación. En este trabajo, se utilizan librerías como roslibpy para la comunicación con ROS en Python y Chart.js en el desarrollo de la aplicación web (UROS APP). [16]

2.8. Servidor Web, Petición HTTP, Servidor ASGI, WebApps y NoSQL

- Servidor web: Es un software que aloja sitios web y responde a solicitudes de los clientes a través del protocolo HTTP (Hypertext Transfer Protocol). En este contexto, se utilizan servidores web para alojar la aplicación web UROS APP y servir contenido a los usuarios. [16]
- Petición HTTP: Es un mensaje que un cliente envía a un servidor web para solicitar recursos o realizar acciones en un sitio web. Se utiliza en el contexto de la comunicación entre el cliente web (UROS APP) y el servidor web para obtener datos de la API. [16]
- Servidor ASGI (Asynchronous Server Gateway Interface) : Es un tipo de servidor web que admite aplicaciones web asincrónicas, lo que significa que puede manejar múltiples solicitudes simultáneamente de manera eficiente. Se utiliza en este trabajo para ejecutar UROS API y gestionar la comunicación entre ROS y la aplicación web. [17]
- WebApps (Aplicaciones Web): Las aplicaciones web son aplicaciones informáticas que se ejecutan en un navegador web. En este trabajo UROS APP es un ejemplo de una aplicación web que interactúa con la API UROS para visualizar datos y permitir la interacción con los sistemas robóticos simulados. [16]
- NoSQL (Bases de Datos NoSQL): Las bases de datos NoSQL son sistemas de gestión de bases de datos diseñados para el almacenamiento y la recuperación eficiente de datos no relacionales, como documentos, gráficos y datos de series temporales. MongoDB es un ejemplo común de una base de datos NoSQL. [16, 17]

Capítulo 3

Antecedentes y Estado del Arte

En la era actual de la tecnología y la información, la convergencia de múltiples disciplinas ha dado lugar a un cambio fundamental en la forma en que interactuamos en el mundo digital y físico. La Industria 4.0 viene impulsando la automatización, la interconexión de dispositivos y la toma de decisiones basada en datos a un nivel sin precedentes. En este contexto, la ciencia de datos y el big data desempeñan un papel crucial al permitirnos extraer información valiosa de grandes volúmenes de datos, optimizando así procesos y mejorando la toma de decisiones. [1]

A medida que avanzamos hacia el futuro, dos conceptos emergentes se destacan: el metaverso y los digital twins. El metaverso representa un espacio virtual tridimensional compartido donde la interacción y la colaboración se vuelven más inmersivas, extendiendo las fronteras de la realidad digital. Los digital twins, por otro lado, son réplicas digitales precisas de objetos, sistemas y procesos del mundo real, que permiten la simulación, supervisión y control en tiempo real. [18]

En este panorama de cambio constante, es fundamental explorar y desarrollar herramientas y plataformas que faciliten la creación y simulación de sistemas robóticos avanzados en entornos virtuales. Esto se vuelve aún más relevante en un mundo impulsado por la Industria 4.0, donde la automatización y la robótica desempeñan un papel fundamental en la transformación de la industria. En este contexto, surge la idea de concretar una forma de integración multiplataforma entre ROS y Unreal Engine, para ello se evaluaron algunos desarrollos anteriores:

3.1. Gazebo

Gazebo es un simulador de código abierto ampliamente utilizado en la robótica. Proporciona un entorno de simulación 3D que permite modelar y simular robots y entornos virtuales. Gazebo ofrece herramientas avanzadas para la simulación de robots, incluyendo la capacidad de definir dinámicas, sensores y controladores personalizados. Ha sido una opción popular para el desarrollo y prueba de algoritmos de control y planificación en entornos simulados. Pero sus capacidades gráficas son limitadas y no es completamente compatible y soportado en sistemas operativos diferentes a Linux.

3.2. CARLA Simulator

CARLA Simulator es un entorno de simulación de conducción de código abierto que se enfoca en la simulación de vehículos autónomos. Además de la simulación realista de vehículos y entornos, CARLA ofrece una amplia variedad de sensores virtuales, que permiten el desarrollo y pruebas de algoritmos de percepción y conducción autónoma.

3.3. AirSim

AirSim es un simulador de vuelo y vehículos autónomos desarrollado por Microsoft. Ofrece un entorno de simulación para drones y vehículos terrestres. AirSim se destaca por su capacidad para simular el comportamiento realista de sensores y su integración con plataformas de desarrollo como ROS. Ha sido utilizado en aplicaciones de investigación de drones y vehículos autónomos. [4]

3.4. ROSIntegration

ROSI o ROSIntegration, es un plugin propuesto y desarrollado inicialmente por la Universidad de Bremen, que utiliza comunicación por sockets especialmente enfocado en Unreal Engine 4, e inicialmente permitía una comunicación usando el protocolo TCP, aunque existe una comunidad que ha venido aportando y desarrollando a la implementación de websockets y extender su compatibilidad con Unreal Engine 5. [19]

3.5. rclUE

Es rclUE un proyecto que busca integrar ROS 2 con Unreal Engine. Esto permite la comunicación y control de sistemas robóticos simulados en Unreal Engine utilizando las capacidades avanzadas de ROS 2. rclUE es un proyecto desarrollado por rapyuta robotics, que utiliza comunicación DDS (Data Distribution Service) e integra Unreal Engine en sistemas operativos Linux. [20]

3.6. foxglove

Foxglove es una plataforma de desarrollo de robótica que combina herramientas de simulación y desarrollo de software. Ofrece un entorno unificado para el diseño, simulación y control de robots. La plataforma foxglove se centra en la simplicidad y la facilidad de uso, lo que la hace atractiva para desarrolladores y diseñadores de sistemas robóticos.

Capítulo 4

Introducción a Unreal Engine

Unreal Engine, es una plataforma de desarrollo líder en simulación y gráficos en 3D. A continuación se documenta brevemente la exploración del entorno y las herramientas esenciales que Unreal Engine ofrece para la creación de simulaciones interactivas.

Se abordan los conceptos básicos que constituyen la base de esta plataforma, incluyendo conceptos como de su UI (Interfaz de usuario) como el viewport, los event graphs, sus herramientas de programación gráfica como los blueprints que permiten crear lógica y comportamientos de manera visual y efectiva, además de la apertura a desarrollos personalizados para ampliar las capacidades de Unreal Engine como son los plugins. [12]

4.1. Entorno y herramientas

Sobre el entorno, interfaz de usuario, la organización de proyectos y las herramientas esenciales que facilitan la creación de simulaciones en 3D. [12]

- Epic Launcher: Es el punto de partida, permite gestionar la instalación de diferentes versiones de UE, como también el acceso al marketplace y administrar mis proyectos.

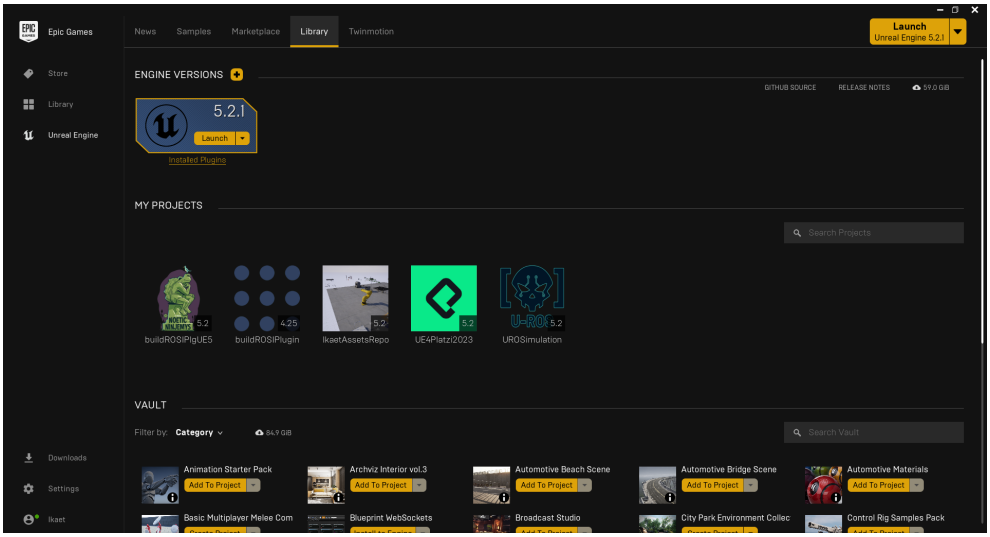

Figura 4.1: Epic Games Launcher

- UE Marketplace: Permite el acceso a recursos y plugins para UE.

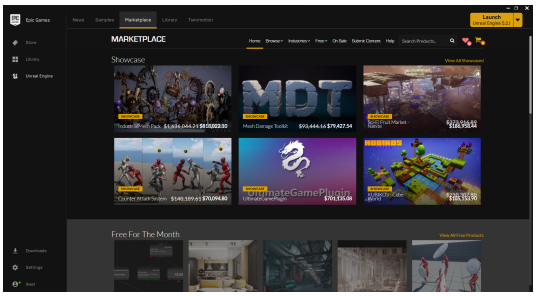

Figura 4.2: Epic Games Marketplace

- UE5 Interfaz de usuario: Es el conjunto inicial de ventanas y funciones de UE.

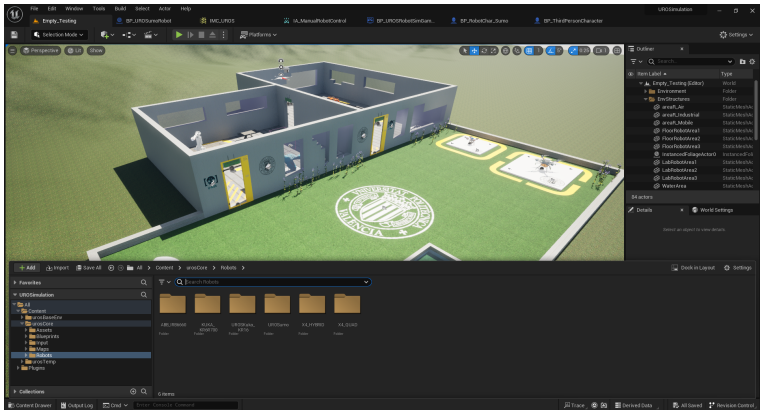

Figura 4.3: UE5 Interfaz de Usuario

- UE5 Viewport: Tanto el viewport principal como cada ventana de viewport permite una visualización de nuestros modelos y recursos.

Figura 4.4: UE5 Viewport

- UE5 Content Drawer: Permite la organización y administración de los recursos del proyecto y su fácil implementación en los proyectos.

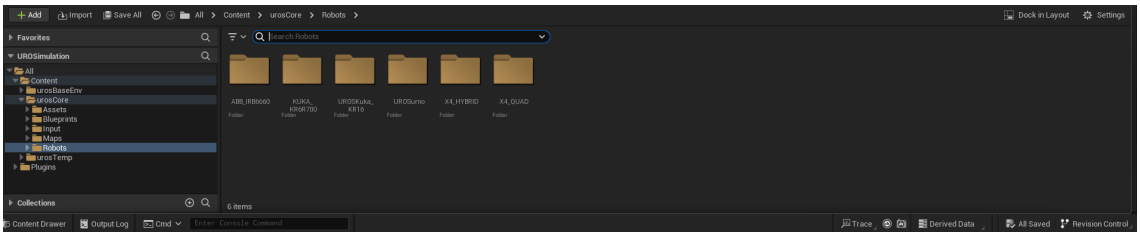

Figura 4.5: UE5 Content Drawer

- UE5 Outliner: Es el espacio en donde visualizamos las instancias de los actores que se encuentran en nuestro nivel o mapa y cómo estan organizados.

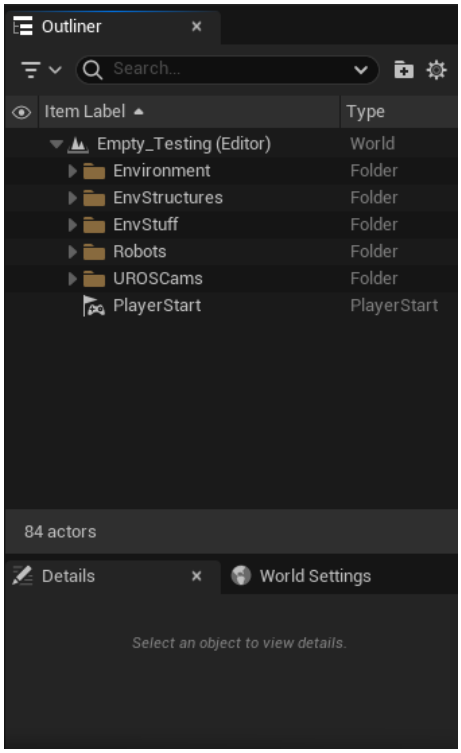

Figura 4.6: UE5 Outliner

4.2. Actores y Blueprints

Sobre los actores se consideran actores la clase base de la que derivan todos los objetos o entidades que se emplean en el entorno 3D. Existen diferentes actores que extienden sus caracteristicas y permiten otras funcionalidades. [12]

Figura 4.7: UE5 Actores (Actors)

Sobre los blueprints de Unreal Engine, son una herramienta esencial para el desarrollo interactivo en Unreal Engine. Son bloques de programación que agiliza la construcción de objetos interactivos en el entorno, estos bloques se conectan a traves de nodos y permitien controlar el flujo de ejecución de la aplicación o funcionalidad que se busca implementar. [12]

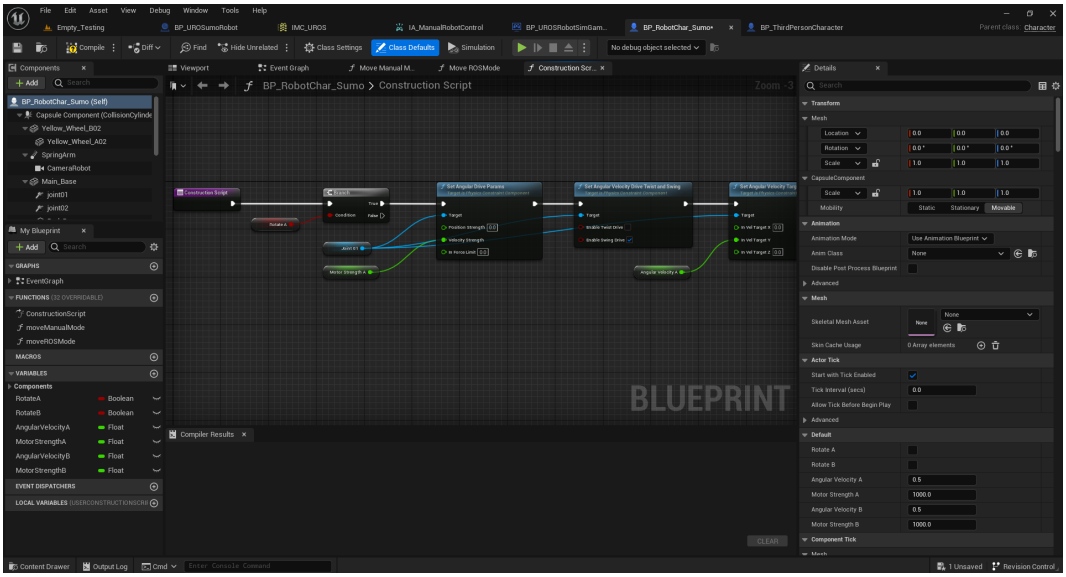

Figura 4.8: UE5 Blueprints

4.3. Scripts - Functions y Eventgraphs

Las funciones y scripts, como en cualquier lenguaje de programación se crean deacuerdo a las necesidades de la aplicación, UE cuenta además con un manejo del flujo de ejecución que promueve buenas prácticas enfocado en el rendimiento para entornos de producción. [12]

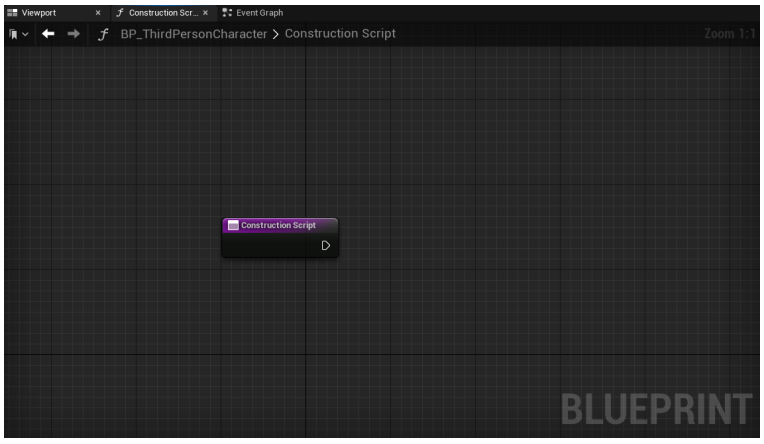

Figura 4.9: UE5 Construction Script - Functions

los Eventgraphs por su parte, permiten separar los eventos interactivos y manejar de forma clara y consistente lo que ocurre en tiempo de ejecución.

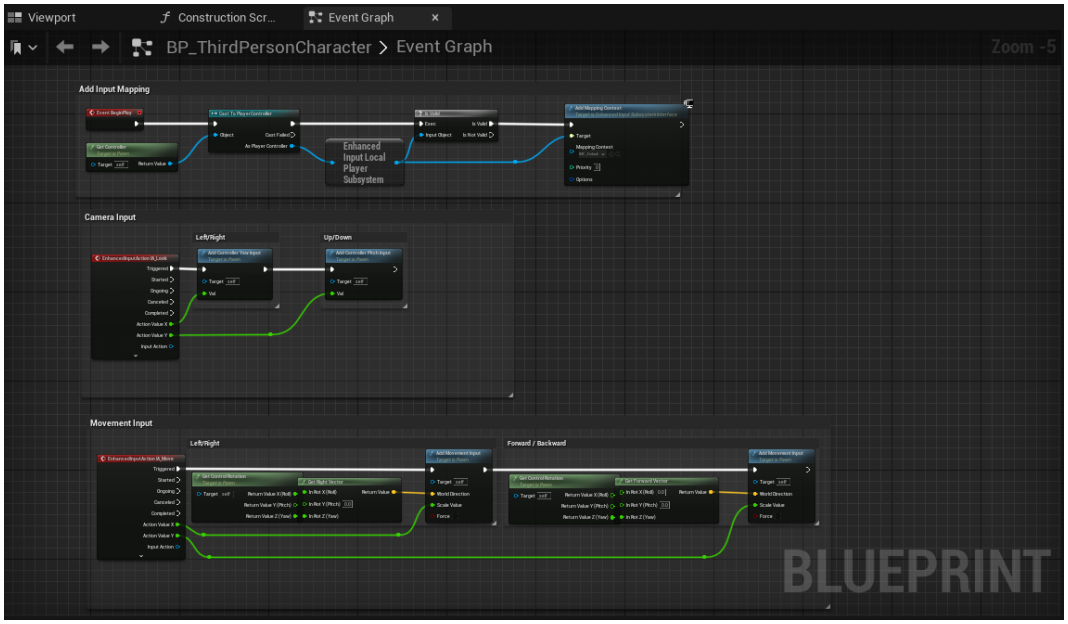

Figura 4.10: UE5 Eventgraphs (Gráfico de eventos)

4.4. Plugins

Los plugins son herramientas que pueden extender las capacidades de la plataforma, permitiendo la creación de simulaciones más complejas y personalizar funcionalidades del entorno de Unreal Engine. [12]

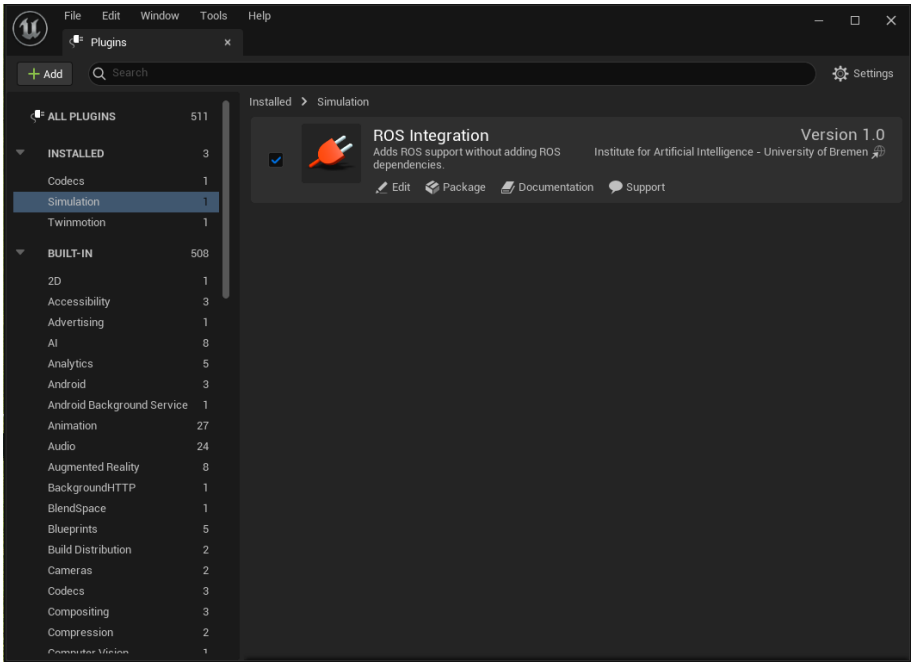

Figura 4.11: UE5 Plugins

Es el caso del plugin ROSIntegration desarrollado por la Universidad de Bremen para integrar ROS con Unreal Engine, que permite extender las funcionalidades de Unreal y realizar conexiónes mediante sockets con UE version 4 especialmente. [19]

Capítulo 5

Introducción a ROS

Robotic Operating System (ROS) es una plataforma ampliamente utilizada en el campo de la robótica. Los conceptos básicos de ROS, incluye comentar acerca de su modelo de comunicaciones basado en publicaciones y subscripciones, empleando nodos, tópicos y servicios, elementos esenciales para la comunicación y control en entornos robóticos. [2]

5.0.1. Conceptos básicos

- Modelo Subscriptor-Publicador: El modelo de comunicación en ROS se basa en el principio de publicación y suscripción. En este modelo, los nodos (componentes de software) pueden publicar mensajes en tópicos específicos y suscribirse a otros tópicos para recibir mensajes. Esto permite que los nodos se comuniquen de manera asincrónica, lo que facilita la modularidad y la flexibilidad en la construcción de sistemas robóticos. [2]
- Nodos: Los nodos son unidades de ejecución en ROS. Cada nodo es un programa independiente que realiza una tarea específica en el sistema robótico. Los nodos pueden comunicarse entre sí mediante la publicación y suscripción a tópicos. Esta arquitectura de nodos distribuidos permite desarrollar sistemas robóticos altamente escalables y modularizados.
- Tópicos: Los tópicos son canales de comunicación en ROS. Los nodos pueden publicar mensajes en tópicos y suscribirse a tópicos para recibir mensajes. Un nodo que controla un sensor podría publicar datos de este sensor en un tópico miSensor y otros nodos podrían suscribirse a ese tópico para procesar dichos datos. Los tópicos permiten la comunicación eficiente entre nodos en tiempo real. [2]
- Servicios: Los servicios son un mecanismo de comunicación en ROS que permiten una comunicación bidireccional específica entre nodos. A diferencia de los tópicos, que se usan principalmente para la transmisión de datos, los servicios están diseñados para actuar en respuesta a una solicitud. [2, 11]

5.0.2. Simulación

ROS se integra con simuladores para permitir la creación de entornos virtuales realistas y la evaluación de algoritmos de control en un contexto seguro. Esto incluye herramientas como gazebo para construir dichos entornos. [11]

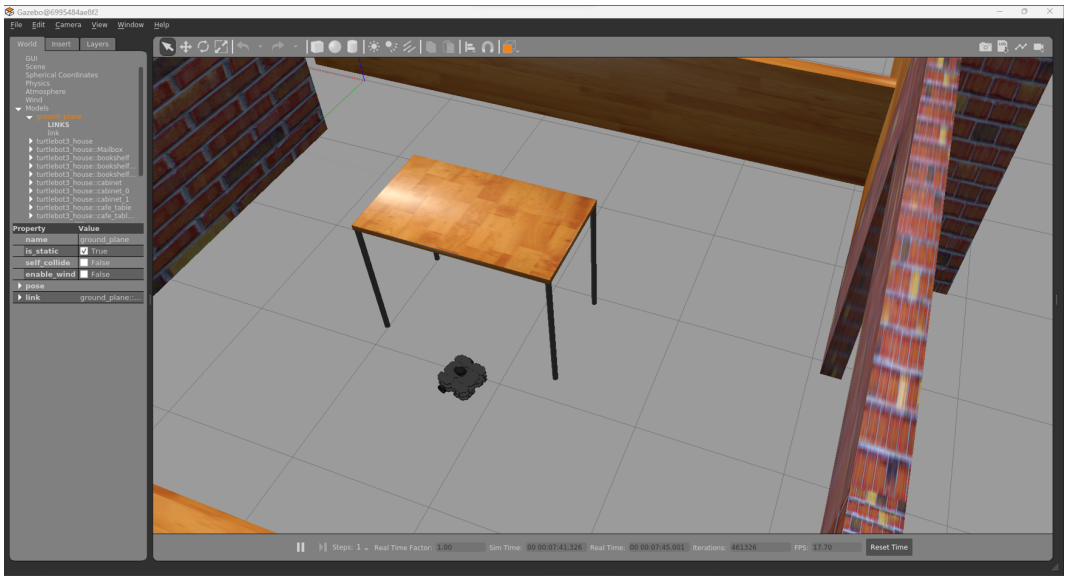

Figura 5.1: Gazebo Entorno 3D

5.0.3. Comunicaciones externas ROS

Hay desarrollos en ROS que facilitan la comunicación con dispositivos y sistemas externos. Esta capacidad de comunicación es crucial para la interoperabilidad y la integración de sistemas robóticos en aplicaciones del mundo real. Uno de estos desarrollos es conocido como ROSBRIDGE, que es un paquete de comunicaciones que incluye un servidor websockets para que ROS tanto su versión 1 como 2 puedan compartir datos y comunicarse con sistemas externos a ROS. [7, 21]

```
2023-09-14 13:28:13+0000 [-] WebSocketServerFactory start
ing on 9090
2023-09-14 13:28:13+0000 [-] Starting factory <autobahn.t
wisted.websocket.WebSocketServerFactory object at 0x7f3eb
d4efa00>
2023-09-14 13:28:13+0000 [-] [INFO] [1694698093.570041]:
Rosbridge WebSocket server started at ws://0.0.0.0:9090
```

```
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [rosbridge_websocket-1]: process started with pid
[192]
[INFO] [rosapi_node-2]: process started with pid [194]
[rosbridge_websocket-1] [INFO] [1694698185.960585380] [ro
sbridge_websocket]: Rosbridge WebSocket server started on
 port 9090
```

Figura 5.2: Rosbridge websocket ROS Noetic derecha / ROS Humble izquierda

```
1    roslaunch rosbridge_server rosbridge_websocket.launch
```

Script 5.1: Comando bash iniciar servidor websocket - ROS Noetic

```
1    ros2 launch rosbridge_server rosbridge_websocket_launch.xml
```

Script 5.2: Comando bash iniciar servidor websocket - ROS Humble

5.1. Diferencias ROS 1 y ROS 2

ROS tiene multiples distribuciones o versiones a medida que ha evolucionado con el tiempo, en este trabajo se utilizó la versión ROS Noetic de ROS 1, para realizar diferentes pruebas. Aunque exiten otras distribuciones de ROS 1, esta versión es una de las que tienen un soporte más amplio actualmente. [22, 23]

Por otra parte ROS 2, es una versión de ROS que ha cambiado en muchos aspectos, incluido las herramientas para compilar paquetes y los comandos para ejecutar las funcionalidades de ROS. Basa sus comunicaciones en un sistema DDS (Data Distribution Service) que permite escalar facilmente en sistemas distribuidos, además se consideran DDS, sistemas robustos que permiten la comunicación en tiempo real de alto rendimiento y fiabilidad. [22, 23]

```
source /opt/ros/noetic/setup.bash

roslaunch gazebo_ros empty_world.launch

mkdir -p ~/mi_workspace/src
cd ~/mi_workspace/
catkin_make
```

Script 5.3: Comandos diferencias - ROS Noetic

```
source /opt/ros/humble/setup.bash

ros2 launch my_gazebo_ros my_gazebolaunch.py

mkdir -p ~/mi_workspace/src
cd ~/mi_workspace/
colcon build
```

Script 5.4: Comandos diferencias - ROS Humble

En los anteriores ejemplos para comandos en la terminal, se pueden observar diferencias importantes en la manera en como se lanzan los servicios y como se compilan los espacios de trabajo (workspace), incluso también existen diferencias cuando se compilan los paquetes en las diferentes versiones.

Por ello es importante tener en cuenta la documentación y considerar las herramientas compatibles para cada versión de ROS cuando se incluyen en un proyecto. [22, 23]

Capítulo 6

Trabajo Previo

En primera instancia se hizo una revisión de los diferentes entornos de ejecución, sistemas operativos y herramientas disponibles para ejecutar ROS en un ambiente común para cualquier usuario como Windows o MAC OS y se llegó a la conclusión que la forma más rápida para desarrollar en un ambiente controlado para ROS es utilizar contenedores docker que permiten configurar libremente las redes en las que opera el sistema ROS, como también administrar rapidamente aplicaciones y levantar servicios de diferentes tipos, por ejemplo MongoDB.

Como solución para permitir la ejecución de aplicaciones con interfaz gráfica se optó por usar un servidor Xserver (VcXsrv) que le permite a los contenedores de docker usar las pantallas del sistema host para visualizar las aplicaciones que se ejecutan en dichos contenedores. [24]

Se utiliza también el sistema WSL2 (Windows Subsystem For Linux), que permite ejecutar un entorno Linux en paralelo a Windows, usando el kernel de Linux integrado a Windows, en este trabajo se utilizó la version Ubuntu 22.04 y Windows 11.

Se revisaron dos desarrollos anteriores, el plugin ROSIntegration y rclUE, para comprobar su funcionamiento y la facilidad para ser implementados en un proyecto como este. [19, 20]

6.0.1. ROS con Docker

Docker y su aplicación Docker Desktop permiten administrar eficazmente entornos de desarrollo de ROS, de manera rápida y controlada. Esta práctica garantiza una configuración consistente y simplifica el proceso de desarrollo de sistemas robóticos.

Se puede crear una imagen propia del sistema, o partir de una imagen base en Docker Hub, modificar dicha imagen para conseguir un sistema con una configuración específica. Esto es aplicable a diferentes sistemas operativos y servicios.

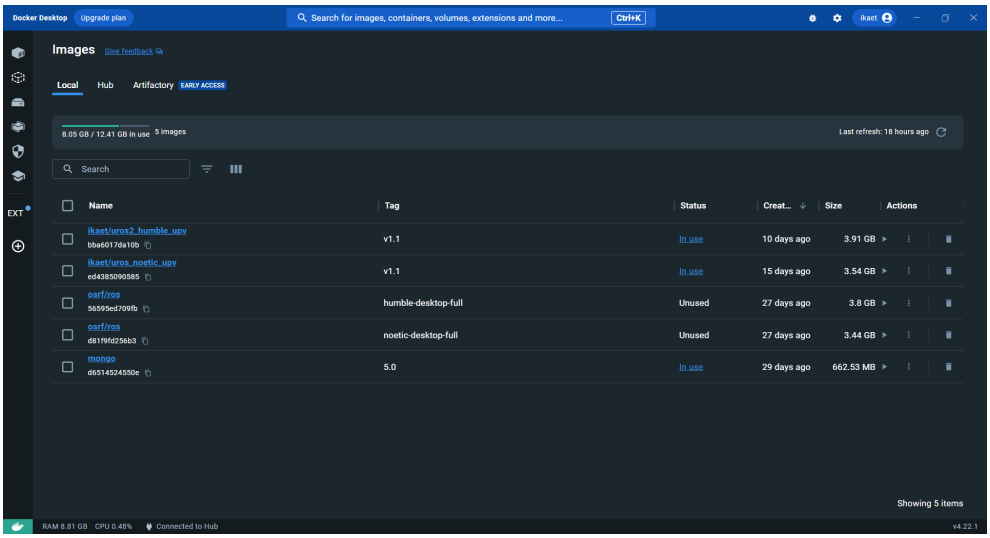

Figura 6.1: Docker Imagenes

A partir de las imágenes se pueden levantar contenedores como servicios, configurando sus respectivos entornos de red, puertos, usuarios y variables de entorno entre otros.

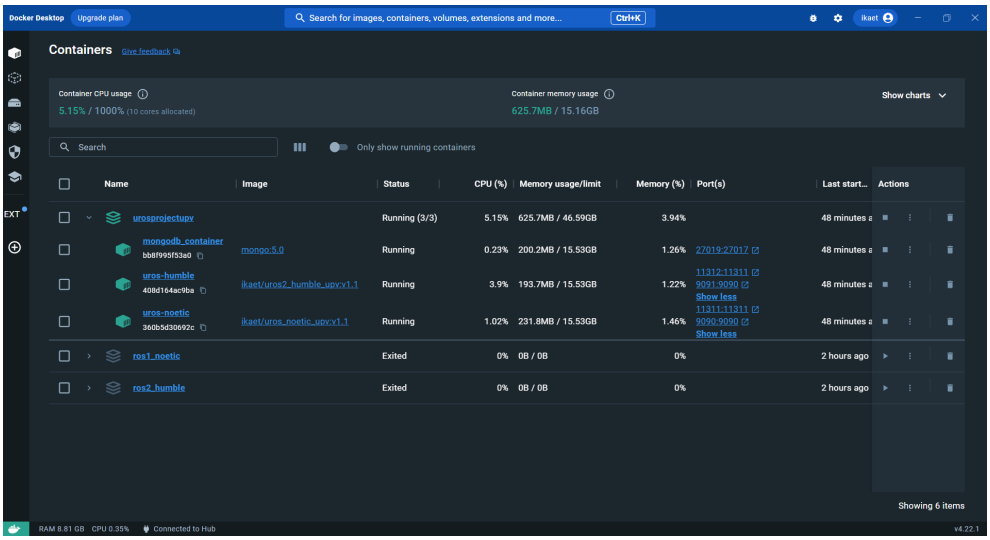

Figura 6.2: Docker Contenedores

Ejemplo de un dockerfile para ROS Noetic:

```
# Utilizar la imagen base de ROS Noetic FULL
FROM osrf/ros:noetic-desktop-full

# Actualizar
RUN apt-get update

# Instalar pip, rosbridge, git
RUN apt-get install -y python3-pip
RUN apt-get install -y ros-noetic-rosbridge-server
RUN pip install roslibpy
RUN apt-get install -y git

# Establecer el directorio de trabajo
WORKDIR /root

# Comando al ejecutar el contenedor
CMD ["bash"]
```

Script 6.1: Dockerfile - ROS Noetic

Finalmente con docker se puede mantener la persistencia de los datos a través de volúmenes, que pueden montarse como unidades compartidas entre los diferentes contenedores y el sistema operativo Host.

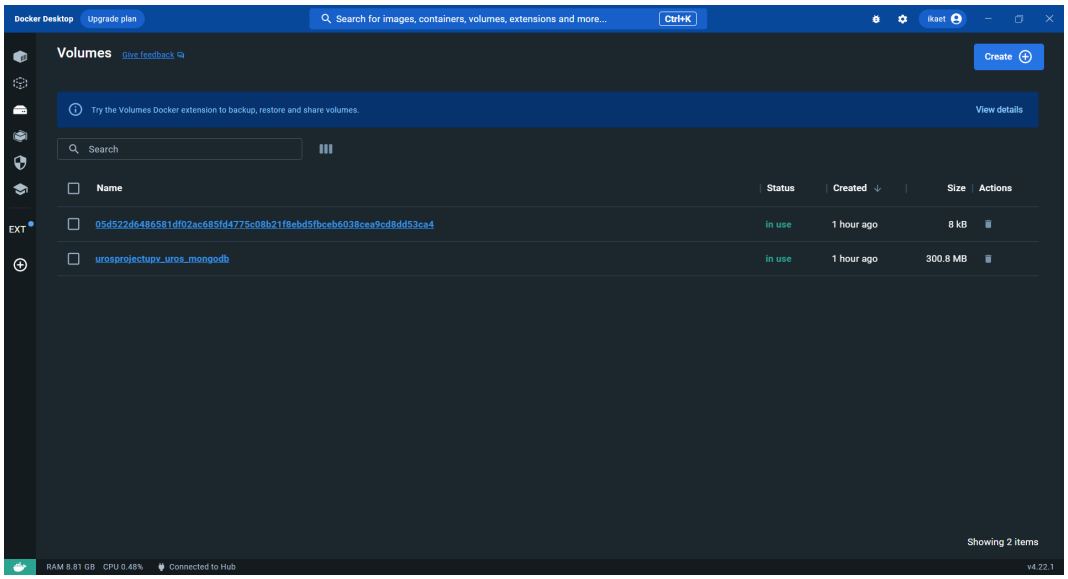

Figura 6.3: Docker Volumenes

6.0.2. XServer GUI

El servidor, Xserver (VcXsrv), es una herramienta que permite la visualización de aplicaciones gráficas de cualquier contenedor docker, incluido ROS. Esta configuración es fundamental para la simulación permitiendo el control visual y aprovechar muchas de las herramientas desarrolladas para ROS como rViz entre otras. [24]

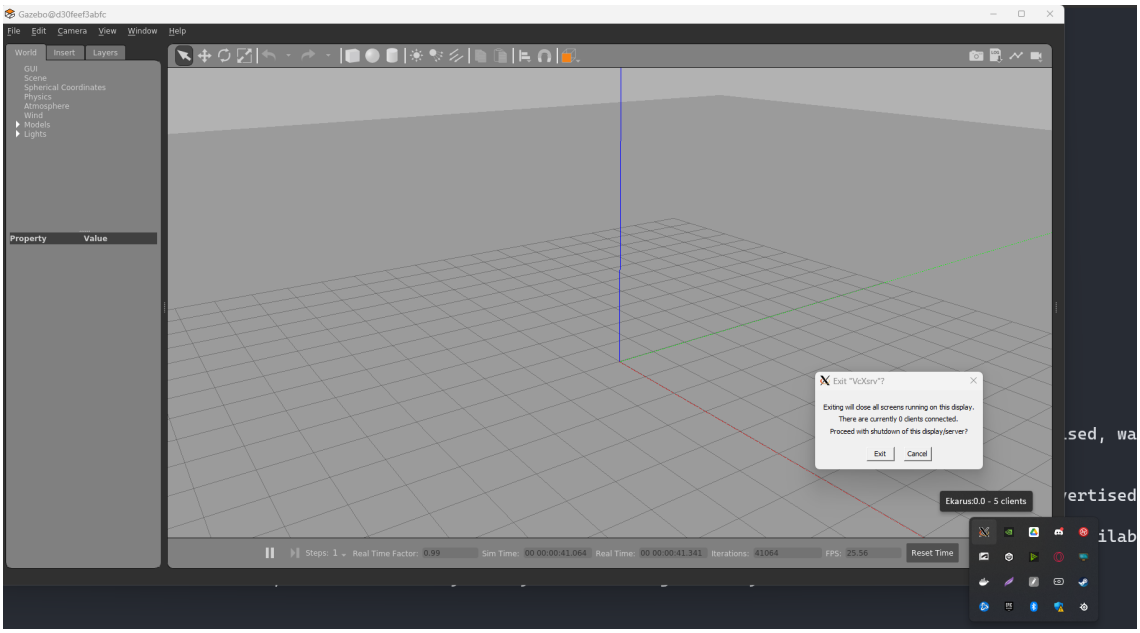

Figura 6.4: VcXsrv - Xserver

6.0.3. Prueba ROSIntegration

Se realizaron pruebas prácticas con el plugin ROSIntegration, mencionado anteriormente, estas pruebas demostraron su funcionamiento para Sockets TCP en UE4 y UE5. Sin embargo con diferentes limitaciones de compatibilidad entre las diferentes versiones de UnrealEngine y las pruebas del uso de websockets con ROSIntegration no fueron satisfactorias. Implica un esfuerzo adicional por la falta de documentación y tutoriales oficiales disponibles sobre la herramienta. No implica que la herramienta no pueda tener un buen desempeño si se estudia detalladamente como se ha realizado su implementación y desarrollo. [19]

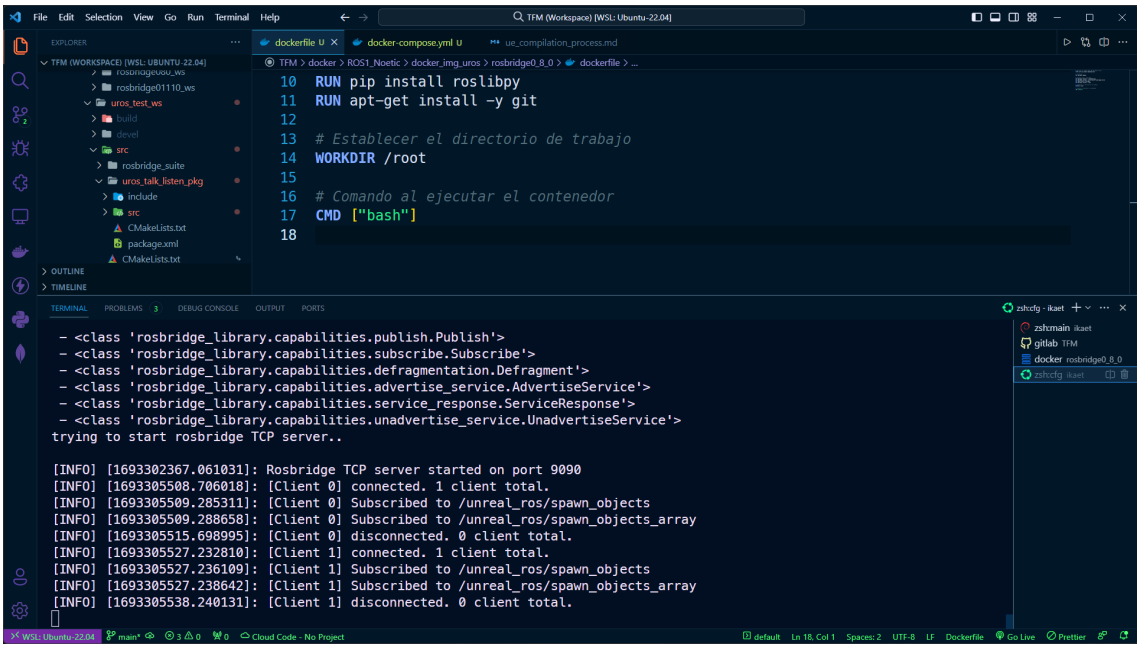

Figura 6.5: ROSIntegration Plugin - Consola ROS

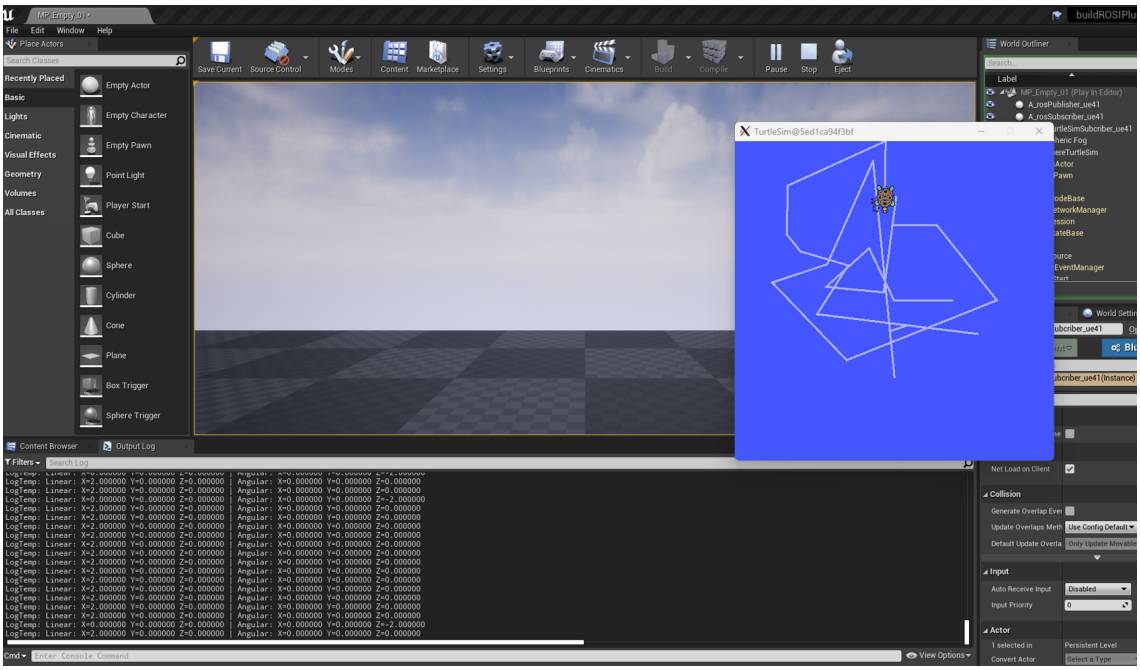

Figura 6.6: ROSIntegration Plugin - TurtleSim - Prueba

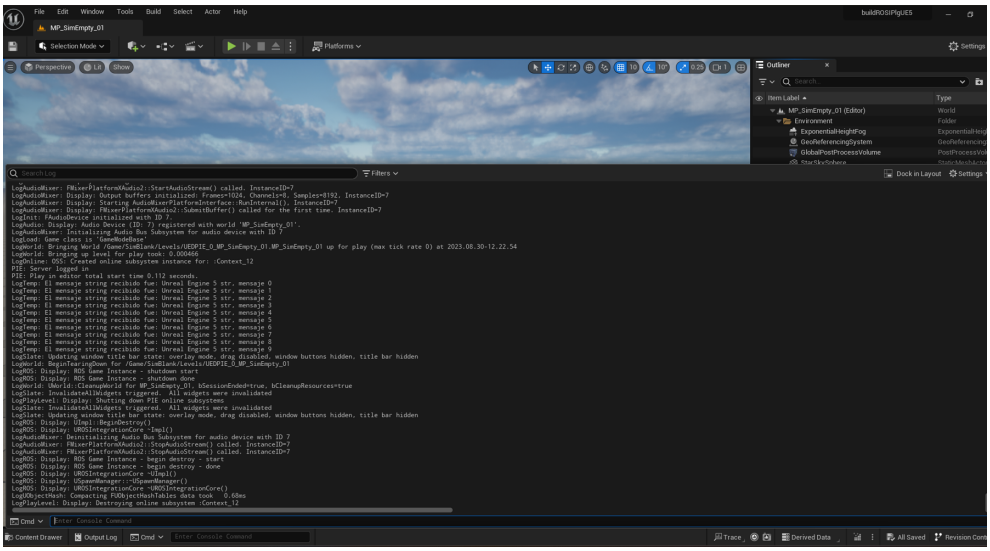

Figura 6.7: ROSIntegration Plugin - Consola Unreal

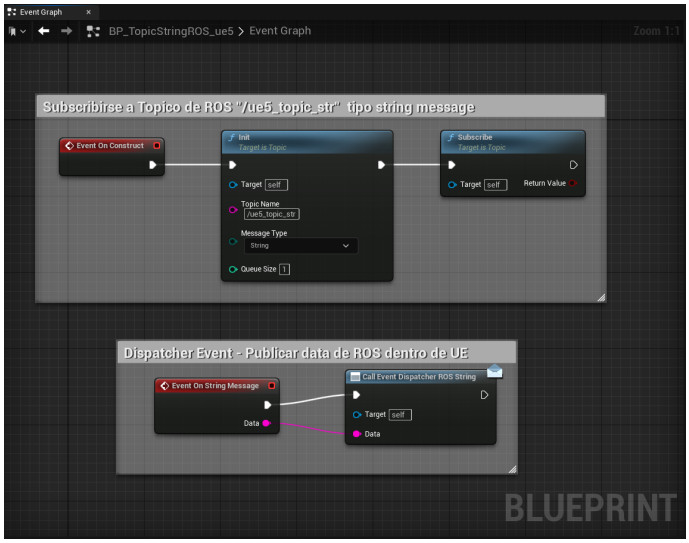

Figura 6.8: ROSIntegration Plugin - Blueprint topicos

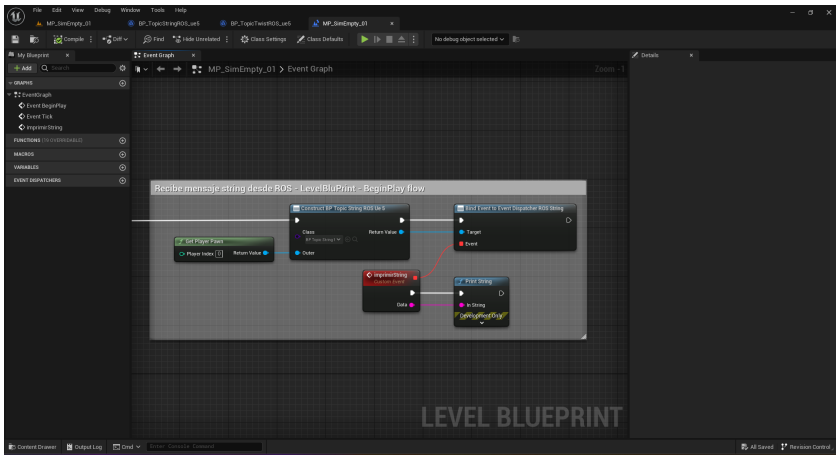

Figura 6.9: ROSIntegration Plugin - Blueprint Datos Unreal

6.0.4. Prueba rclUE

Se intentaron realizar pruebas de rclUE, sin embargo durante el proceso, se evidencia que esta solución requiere una instalación de Linux nativa, que permita la instalación de los controladores gráficos de Nvidia o la tarjeta gráfica que use el equipo, ya que Unreal Engine debe correr usando estos controladores, existen soluciones que pueden permitir la ejecución de estas pruebas en contenedores docker, pero requieren un esfuerzo adicional. Sin embargo este plugin, utiliza un sistema DDS que puede ser una solución muy completa para el propósito de trabajos similares a este. [20]

6.0.5. MongoDB con Docker

Para este trabajo se configuro MongoDB como un servicio local corriendo desde un contenedor de docker, permitiendo modularizar y separar los servicios en el ambiente de desarrolo, lo que permite identificar facilmente los problemas respecto al flujo de los datos.

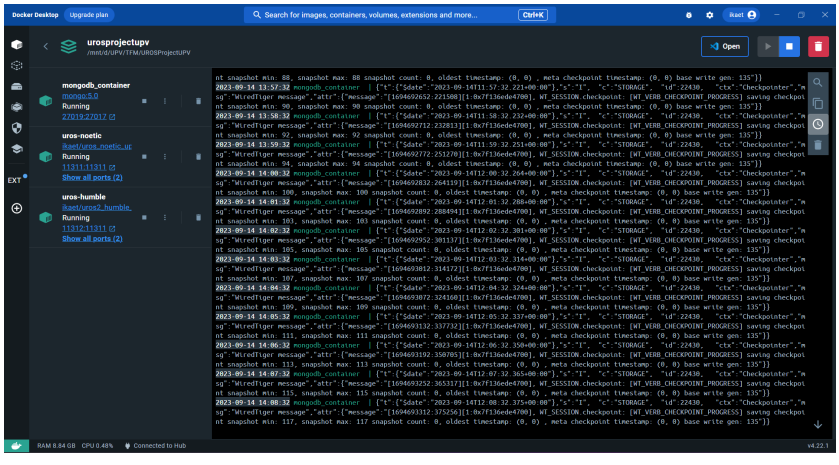

Figura 6.10: Docker Servicio MongoDB y Stack UROS

Capítulo 7

UROS API

7.0.1. Introducción

UROS API, como se ha llamado el software desarrollado, es una interfaz de programación de aplicaciones, que ha sido desarrollada en el lenguaje Python. Aprovecha la librería ROSLIBPY para gestionar la comunicación a traves de websockets con ROS 1 y ROS 2 usando convirtiendose en un cliente del servidor websockets que permite rosbridge. Es una herramienta experimental que se seguirá desarrollando para conseguir el soporte DDS, de tal forma que usuarios con perfiles menos técnicos tengan la posibilidad de aprovechar ROS como sistema operativo para robots dentro de entornos controlados. [21]

Se ha utilizado FASTAPI como framework para manejar peticiones Http, construir las rutas y prestar servicio a cualquier cliente con arquitectura REST que quisiera consumir los datos que ROS produce, algunas de las caracteristicas de la API ya estan implementadas y en funcionamiento, otras se publicaran a futuro en la primera versión completa de este software. [14]

Esta API presta soporte para ROS Noetic (ROS 1) y ROS Humble (ROS 2), para comunicaciones websocket y actualmente tiene las funcionalidades básicas para suscribir, publicar en topicos y consumir servicios de ROS.

La API puede administrar la comunicación websocket con Unreal Engine 5 y ROS, se ha probado especificamente en la versión 5.2 , sirviendo como puente entre ambos sistemas o simplemente convirtiendose en un cliente más de ROS que puede monitorear los datos que se publican en tópicos específicos.

El código fuente completo estará disponible en el repositorio git `https://gitlab.com/Ikaet/uros_api`

7.0.2. Funciones

- Administrar la conexión con ROS 1 y 2 con websockets
- Suscribir y desuscribirse de topicos, almacenarlos en objetos JSON
- Suscribirse y desuscribirse de topicos, almacenando datos en MongoDB
- Publicar datos en topicos nuevos o existentes y dejar de publicar
- Crear servicio websocket para conexión de Unreal Engine 5.2
- Administrar conexiones websocket generadas por la API
- Servir datos de MongoDB cuando es solicitada por un cliente

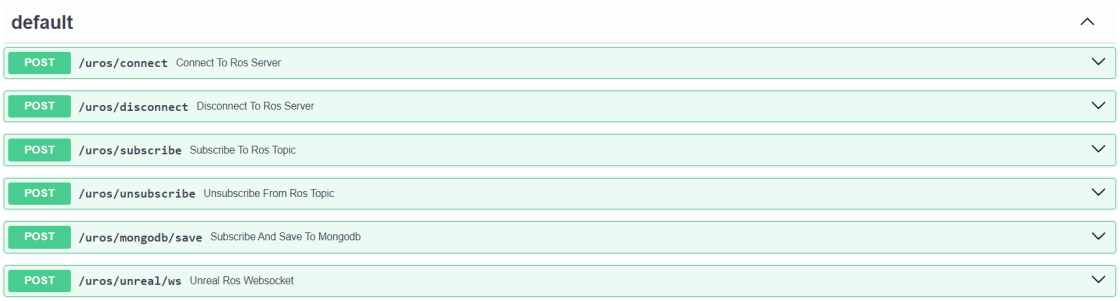

Figura 7.1: UROS API - Funciones v0.1

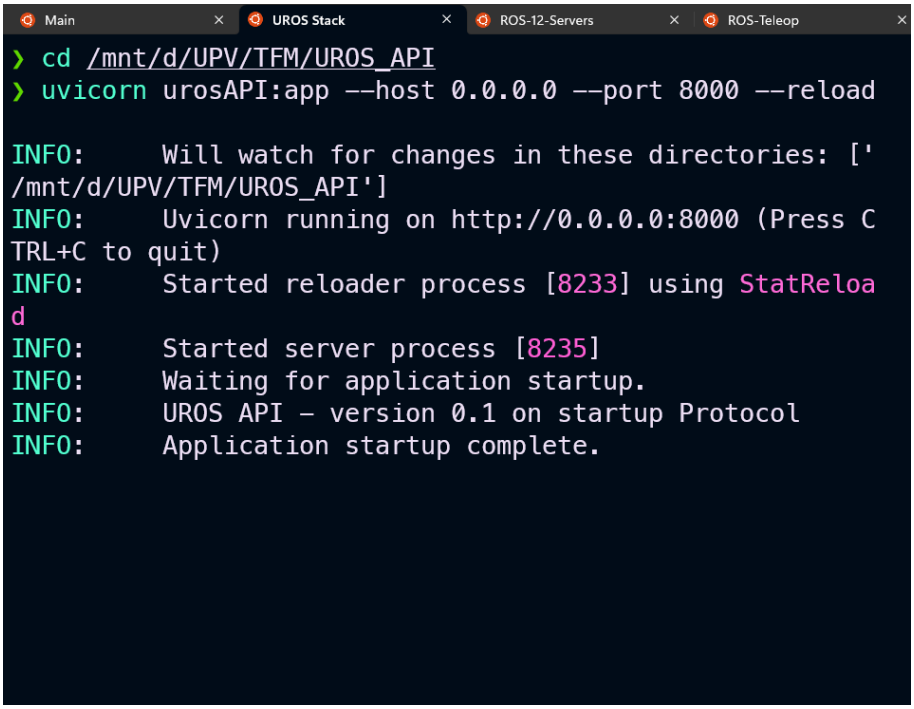

Figura 7.2: UROS API - Levantando el servicio v0.1

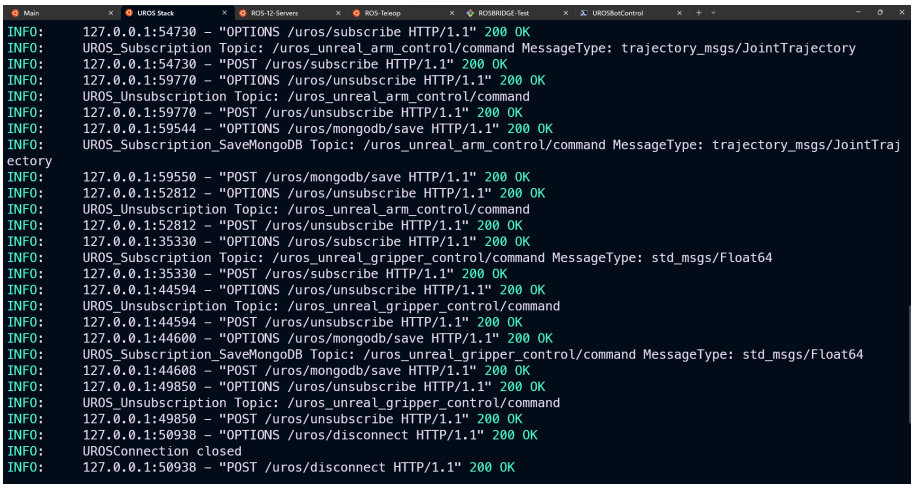

Figura 7.3: UROS API - Log de funcionamiento v0.1

7.0.3. Conexión a ROS

Para la comunicación con ROS se utilizó la libreria roslibpy principalmente.

```
ros_connections: Dict[str, Ros] = {}  # Dict conn ROS

def is_valid_ip(ip):
    if ip == "localhost":
        return True
    ip_pattern = r'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$'
    return bool(re.match(ip_pattern, ip))

def is_valid_port(port):
    port_pattern = r'^(?:[1-9]\d{0,4}|[1-5]\d{4}|6[0-5][0-5][0-3][0-5])$'
    return bool(re.match(port_pattern, str(port)))

def ros_begin_connection(ip, port):
    uvicorn_logger.info(f"UROSConnection Host: {ip} Port: {port}")
    try:
        if "single_ros_server" not in ros_connections:
            ros = Ros(host=ip, port=port)
            ros_connections["single_ros_server"] = ros
            ros.run()
            uvicorn_logger.info(f"UROSConnection open on {ip}:{port}")
            return "success", None
        else:
            ros_connections["single_ros_server"].connect()
            uvicorn_logger.info(f"UROSConnection reopen on {ip}:{port}")
            return "already exist - restarted", None

    except Exception as e:
        uvicorn_logger.error(f"Error UROS creating ROS connection: {str(e)}")
        return "ERROR", str(e)

def ros_end_connection():
    ros = ros_connections.get("single_ros_server")

    if ros and ros.is_connected:
        try:
```

```
                ros.close()
                uvicorn_logger.info(f"UROSConnection closed")
                return "success", None
            except Exception as e:
                uvicorn_logger.error(f"Error UROS closing ROS connection: {str(e)}")
                return "ERROR", str(e)
        return "no active connection", None
```

Script 7.1: UROS API - Conexión ROS

7.0.4. Websocket Unreal Engine

```

    # WEBSOCKETS simple
    active_connections: Set[WebSocket] = set()

    def message_callback(message):
        global message_data_ws  # Utiliza la variable global para almacenar el mensaje
        message_data_ws = message['data']

    @app.websocket("/ws")
    async def websocket_endpoint(websocket: WebSocket):
        await websocket.accept()
        active_connections.add(websocket)
        uvicorn_logger.info("UROSConnection Host: localhost Port: 9090")
        ros = Ros(host='localhost', port=9090)
        ros.run()
        uvicorn_logger.info("UROSConnection open")
        global message_data_ws
        topic = Topic(ros, '/testUPV_string', 'std_msgs/String')
        topic.subscribe(message_callback)

        try:
            while True:
                if message_data_ws:
                    await websocket.send_text(message_data_ws)
                    message_data_ws = ""
        except WebSocketDisconnect:
            if ros.is_connected:
                uvicorn_logger.info("UROSConnection Host: localhost Port: 9090")
                uvicorn_logger.info("UROSConnection closed")
                ros.terminate()
                active_connections.remove(websocket)
        except Exception as e:
            print(f"Error en websocket_endpoint: {str(e)}")

            if ros.is_connected:
                uvicorn_logger.info("UROSConnection Host: localhost Port: 9090")
                uvicorn_logger.info("UROSConnection closed")
                ros.terminate()
            active_connections.remove(websocket)
```

Script 7.2: UROS API - Websocket ROS-Unreal Conexión

La conexión con websockets desde unreal se pueden lograr desarrollando clases C++ y funciones directamente, o aprovechando plugins para comunicación websocket que existen disponibles en el marketplace de Unreal Engine, sin embargo se deben construir los blueprints necesarios deacuerdo a la función que tendra Unreal durante la simulación. [25]

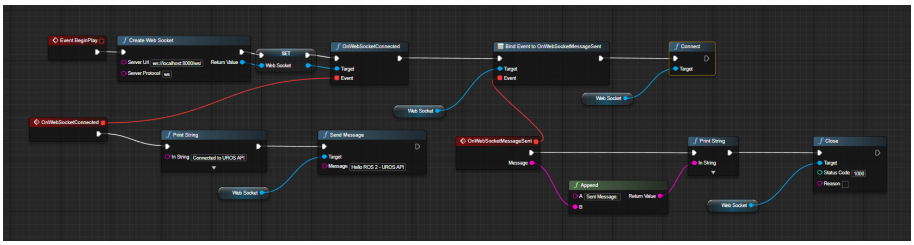

Figura 7.4: UROS API - Unreal Connection Websockets v0.1

7.0.5. Conexión a MongoDB

La conexión con mongoDB se consigue usando el cliente disponible para python, esto permite a la API aprovechar los datos para posteriormente ser analizados y servir en diferentes aplicaciones por ejemplo la ciencia de datos. [26]

```
1    mongo_client = MongoClient('mongodb://root:root123@localhost:27019/?authMechanism=DEFAULT')
2    db = mongo_client['uros_api']
```

Script 7.3: UROS API - Conexión MongoDB

7.0.6. Arquitectura

Se utiliza para el funcionamiento de UROS API, un servidor web conocido como uvicorn, es un servidor ASGI (Asynchronous Server Gateway Interface) desarrollado en Python. ASGI es un estándar que permite la creación de aplicaciones web asincrónicas en Python, lo que significa que puede manejar múltiples conexiones simultáneamente de manera eficiente y escalable. Un servidor ASGI puede gestionar solicitudes y respuestas de manera concurrente sin bloquear el hilo principal del servidor. Uvicorn soporta conexiones en tiempo real por medio de websockets. [25, 20, 17]

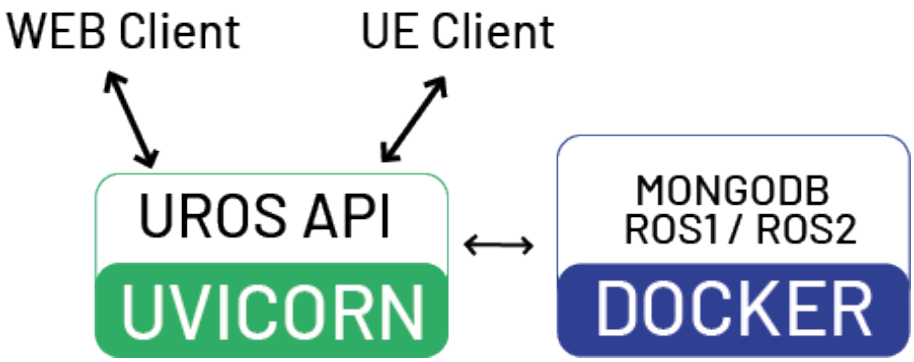

Figura 7.5: UROS API - Arquitectura v0.1

Capítulo 8

UROS APP

8.0.1. Introducción

UROS APP, es una aplicación web construida usando React.JS , es un cliente web que permite consumir los servicios de la API documentada anteriormente, es una aplicación de frontend que utiliza componentes y maneja rutas que presentan una forma de consumir datos desde ROS, especialmente enfocada en un posible usuario final.

UROS APP, puede conectarse a ROS por medio de UROS API, o también si se implementa, podria conectarse directamente utilizando websockets con javascript. Esta aplicación es desarrollada en javascript con los estandares de ECMAScript ampliamente conocidos. [15]

En este trabajo se presenta un ejemplo del codigo empleado en cada una de las principales características, de la aplicación. El código fuente completo estará disponible en el repositorio de git `https://gitlab.com/Ikaet/uros_app`

8.0.2. Funciones

- Servir de interfaz gráfica para UROS API
- Visualizar gráficamente los datos obtenidos de los topicos de ROS
- Controlar UROS API a partir de las peticiones http

8.0.3. Rutas

```
import { useRoutes, BrowserRouter} from 'react-router-dom'

import Home from '../Home'
import UROSConnection from '../UROSConnection'
import UROSubscribe from '../UROSubscribe'
import UROSPublish from '../UROSPublish'
import UROSMonitor from '../UROSMonitor'
import UROSNotFound from '../UROSNotFound'
import Credits from '../Credits'

import Disclaimer from '../../components/Disclaimer'
import './App.css'

const AppRoutes = () =>{
  let routes = useRoutes([
    { path: '/', element: <Home /> },
```

```
        { path: '/UROSConnection', element: <UROSConnection /> },
        { path: '/UROSubscribe', element: <UROSubscribe /> },
        { path: '/UROSPublish', element: <UROSPublish /> },
        { path: '/UROSMonitor', element: <UROSMonitor /> },
        { path: '/Credits', element: <Credits /> },
        { path: '/*', element: <UROSNotFound /> },
      ])

      return routes
    }

    const App = () => {

      return(
        <BrowserRouter>
          <AppRoutes />
          <Disclaimer />
        </BrowserRouter>
      )
    }

    export default App
```

Script 8.1: UROS APP - Rutas

8.0.4. Paginas

```
    import Navbar from '../../components/Navbar'
    import ConnectionForm from '../../components/ConnectionForm'
    import UROSBranding from '../../components/Branding'

    import '../App/App.css'

    function UROSConnection() {

      return (
        <>
          <div>
            <ConnectionForm />
          </div>
          <UROSBranding />
          <Navbar />
        </>
      )
    }

    export default UROSConnection
```

Script 8.2: UROS APP - Paginas

8.0.5. Componentes

```
import { useState } from "react";

const ConnectionForm = () => {
  const [formData, setFormData] = useState({
    ip: "",
    port: "",
  });
  const [connected, setConnected] = useState(false); // Handle connection status

  const handleConnect = async () => {
    try {
      const response = await fetch("http://localhost:8000/uros/connect", {
        method: "POST",
        body: JSON.stringify(formData),
        headers: {
          "Content-Type": "application/json",
        },
      });
      if (response.ok) {
        // If connection ok
        const data = await response.json();
        // Handle answer
        console.log(data.message);
        setConnected(true); // Is connected
      } else {
        console.error("Error en la solicitud de conexion:", response.statusText);
      }
    } catch (error) {
      console.error(error);
    }
  };

  const handleDisconnect = async () => {
    try {
      const response = await fetch("http://localhost:8000/uros/disconnect", {
        method: "POST",
        body: JSON.stringify(formData),
        headers: {
          "Content-Type": "application/json",
        },
      });
      if (response.ok) {
        // If disconnection ok
        const data = await response.json();
        // Handle answer
        console.log(data.message);
        setConnected(false); // Is disconnected
      } else {
        console.error("Error en la solicitud de desconexion:", response.statusText);
      }
    } catch (error) {
      console.error(error);
    }
  };

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

```

```
    return (
      <div className="card form-div">
        <label htmlFor="ip">IP ROS Server:</label>
        <input type="text" id="ip" name="ip" required value={formData.ip} onChange={handleChange}
/><br />

        <label htmlFor="port">Port ROS Server:</label>
        <input type="number" id="port" name="port" required value={formData.port} onChange={
handleChange} /><br />

        <div className="card">
          <button className="button bg-zinc-900 text-cyan-200" type="button" onClick={handleConnect
} disabled={connected}>
            Connect to ROS
          </button>
        </div>

        <div className="card">
          <button className="button bg-zinc-900 text-cyan-200" type="button" onClick={
handleDisconnect} disabled={!connected}>
            Disconnect
          </button>
        </div>

      </div>
    );
};

export default ConnectionForm;
```

Script 8.3: UROS APP - Componentes

8.0.6. Vistas

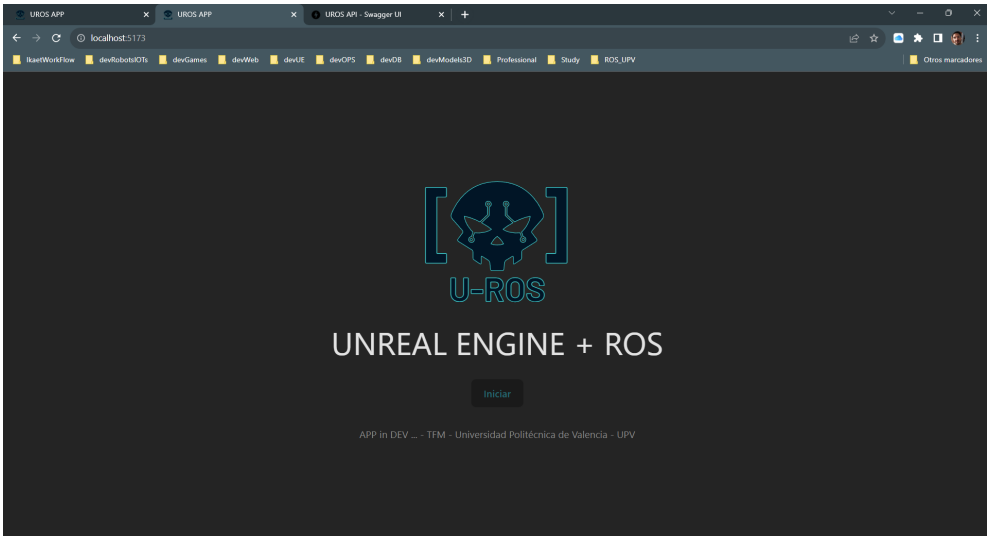

Figura 8.1: UROS APP - Vista Inicio - v0.1

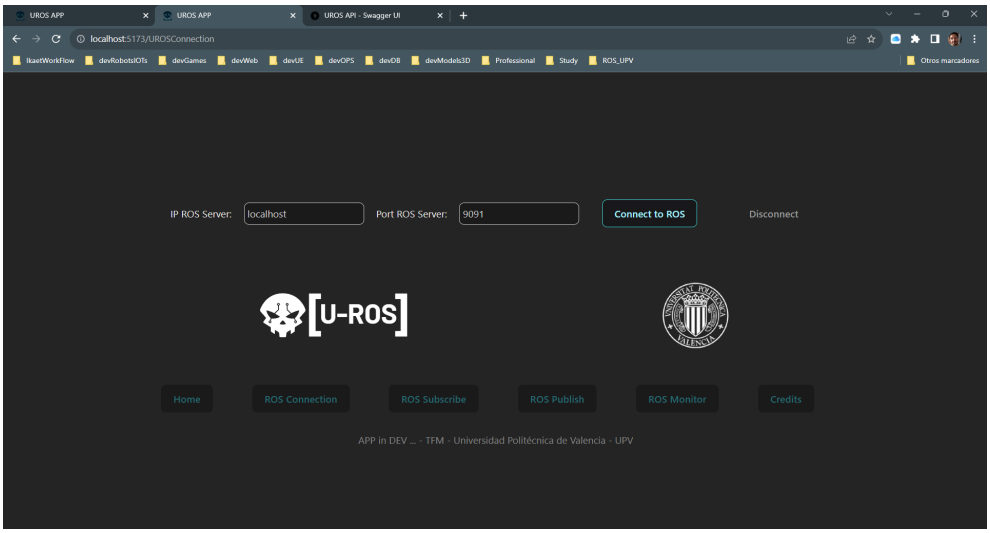

Figura 8.2: UROS APP - Vista Conexión - v0.1

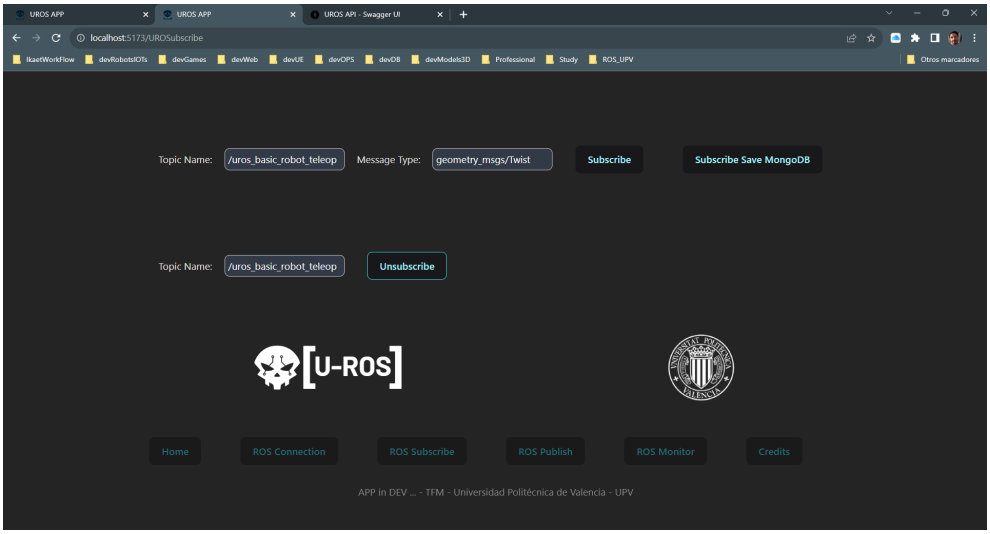

Figura 8.3: UROS APP - Vista Suscribir - v0.1

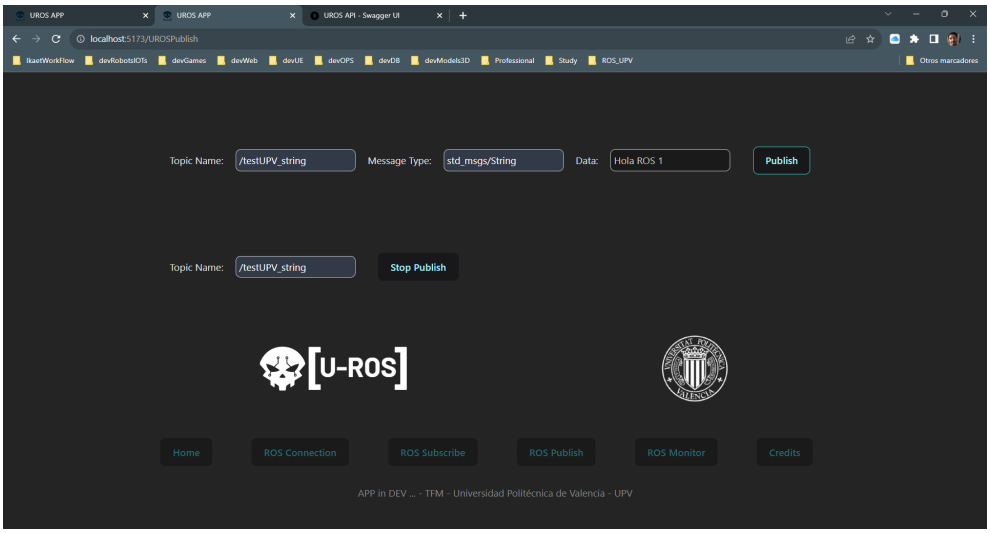

Figura 8.4: UROS APP - Vista Publicar - v0.1

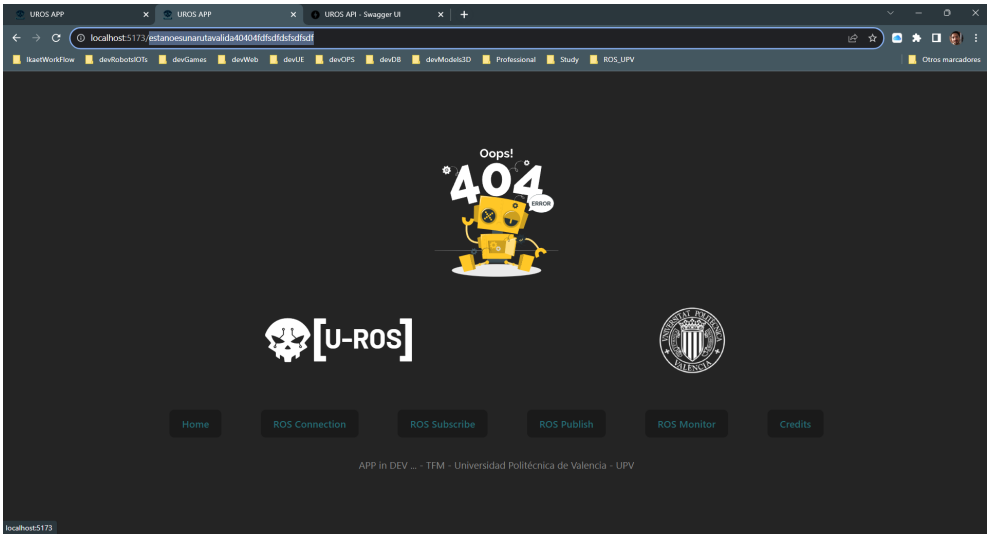

Figura 8.5: UROS APP - Vista No Encontrado - v0.1

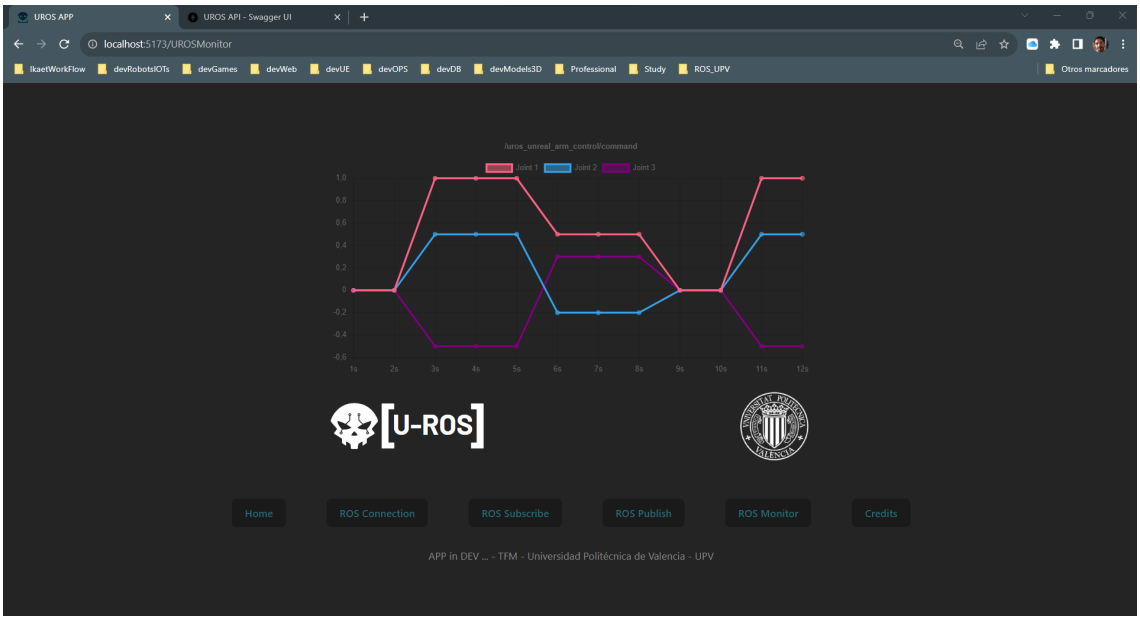

Figura 8.6: UROS APP - Vista Monitor v0.1

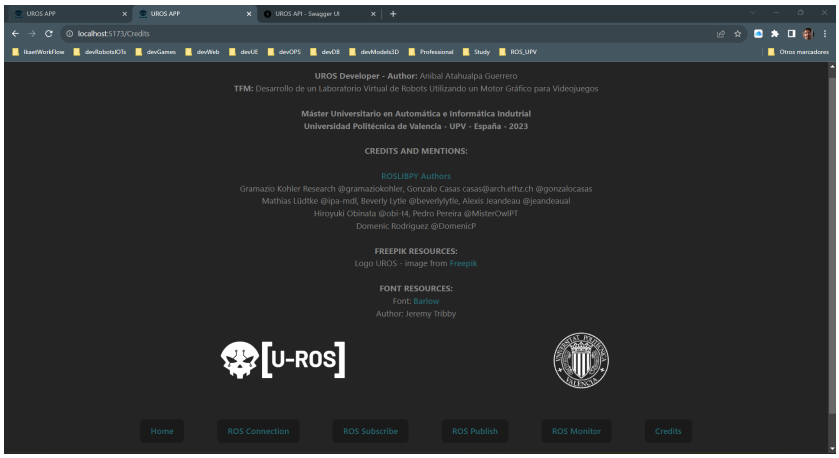

Figura 8.7: UROS APP - Vista Creditos - v0.1

8.0.7. Arquitectura

Se utiliza un modelo vista componente, que controla la API, la aplicación se corre iniciando el servicio sobre un servidor web local integrado con vite.

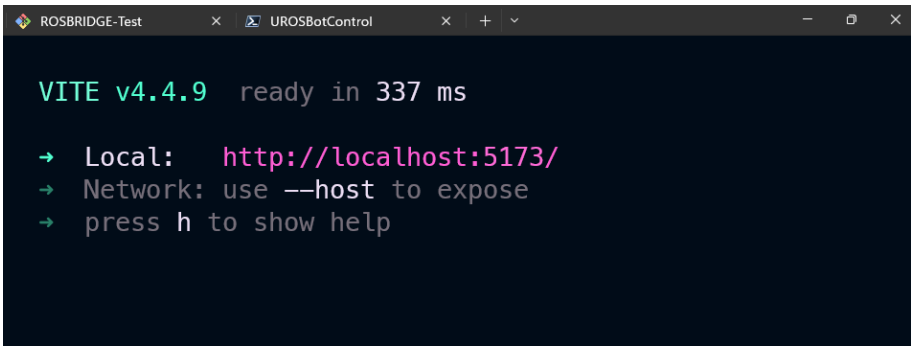

Figura 8.8: UROS APP - Servicio - v0.1

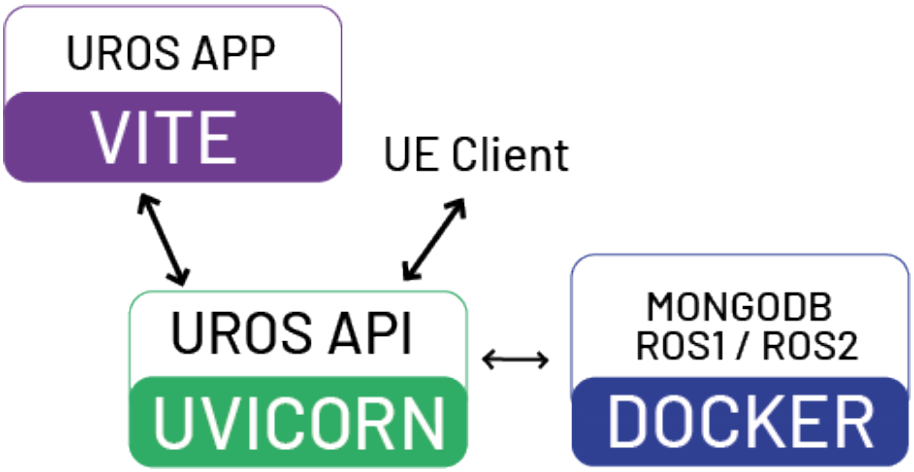

Figura 8.9: UROS APP - Arquitectura - v0.1

Capítulo 9

UROS Simulacion y pruebas

9.0.1. Introducción

Acontinuación se documenta el entorno de simulación y laboratorio de robótica virtual, en el que se realizaron pruebas y se simularon dos robots con controles simples de posición y velocidad, este entorno permite generar una experiencia que es suceptible de adaptarse en una versión de realidad virtual aprovechando Unreal Engine para este objetivo. [27]

Actualmente el entorno de laboratorio cuenta con 2 modelos de robot con 6 grados de libertad, 2 modelos tipo Drone, 1 tipo sumo, un Hexapodo y un robot Cuadrupedo. Los recursos no fueron modelados para el trabajo si no una recopilación de diferentes recursos disponibles de forma libre en sitios web como grabcad y en el marketplace de Unreal Engine.

9.0.2. Laboratorio de robótica

Figura 9.1: UROS SIM - Entorno vista 1 - v0.1

Figura 9.2: UROS SIM - Entorno vista 2 - v0.1

Figura 9.3: UROS SIM - Entorno vista 3 - v0.1

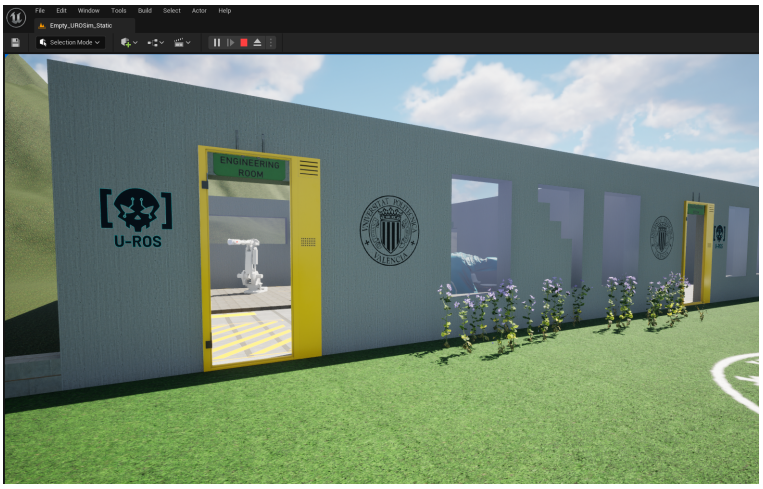

Figura 9.4: UROS SIM - Entorno vista 4 - v0.1

Area 1

Esta área corresponde al espacio para los robots aereos en donde se probará a futuro la simulación de fluidos como el aire.

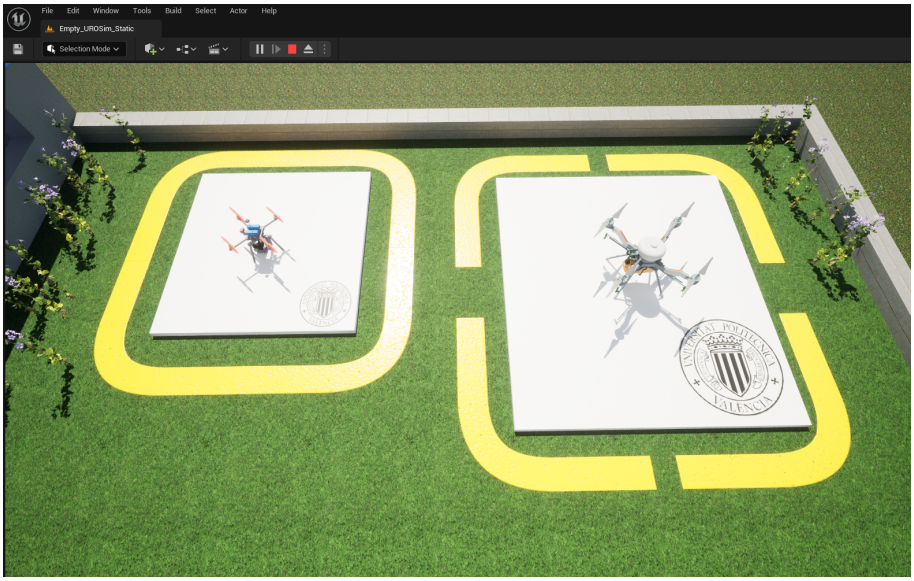

Figura 9.5: UROS SIM - Entorno vista 5 - v0.1

Area 2

Esta área corresponde al espacio para robots móviles, en donde se pueden utilizar obstaculos y otros elementos para probar algoritmos de control y trayectorias.

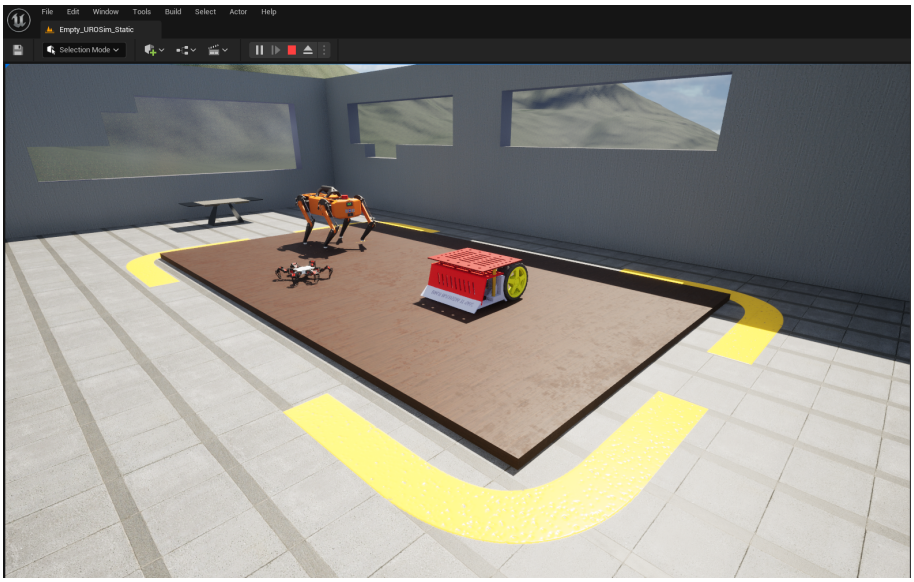

Figura 9.6: UROS SIM - Entorno vista 6 - v0.1

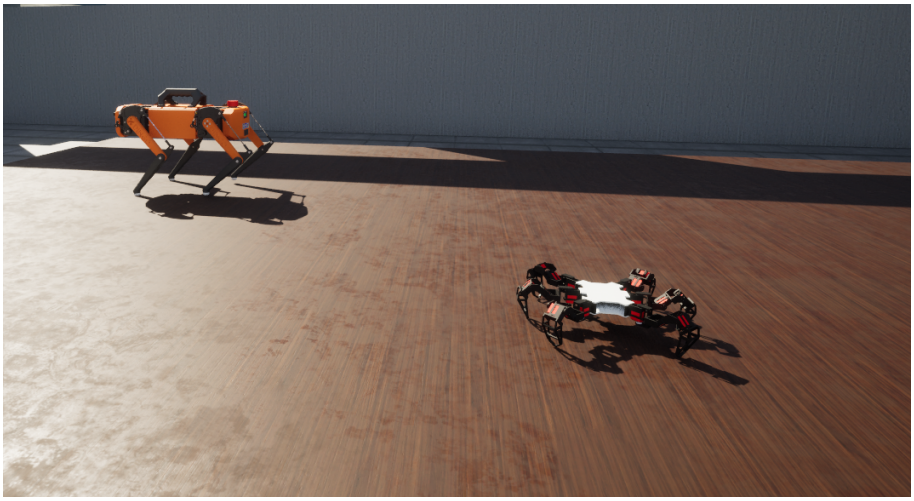

Figura 9.7: UROS SIM - Entorno vista 6A - v0.1

Area 3

Finalmente el area para robots industriales, se pueden realizar diversas actividades y pruebas, entre ellas tareas simples de pickand place, controles por posición o implementando el control adecuado simular controles de fuerza.

Figura 9.8: UROS SIM - Entorno vista 7 - v0.1

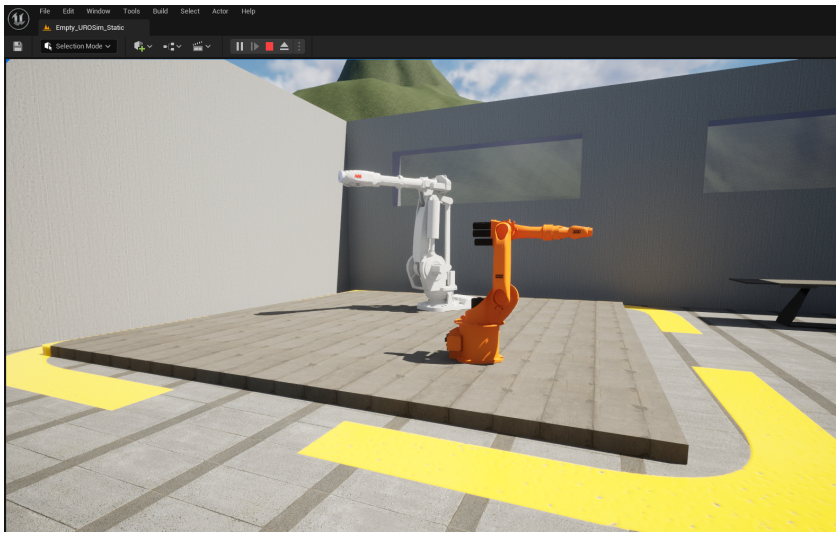

Figura 9.9: UROS SIM - Entorno vista 8 - v0.1

9.0.3. Modelos 3D y Configuración

Para utilizar los modelos 3D en el entorno de simulación, se requiere configurar correctamente los eslabones correspondientes de cada robot, incluidas las restricciones físicas de rotación y traslación, además de los factores a simular, por ejemplo la fricción, la masa, los limites físicos de colisiones entre objetos, entre otras caractéristicas.

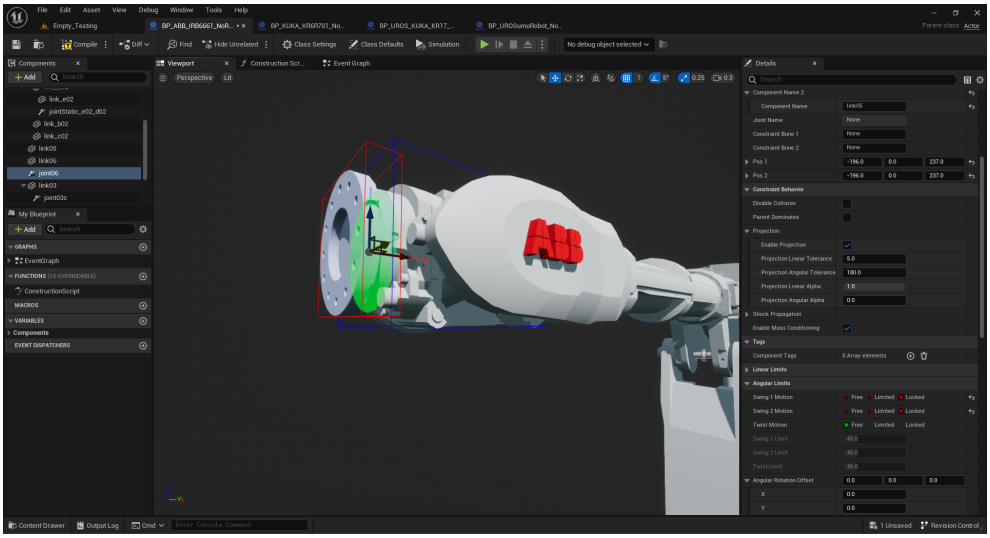

Figura 9.10: UROS SIM - Configuraciones Robot 1 - v0.1

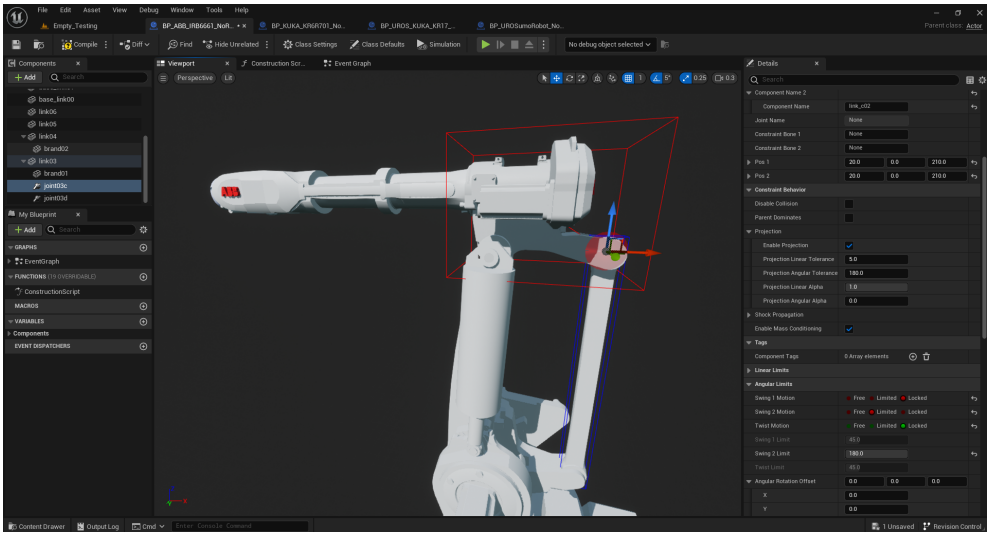

Figura 9.11: UROS SIM - Configuraciones Robot 2 - v0.1

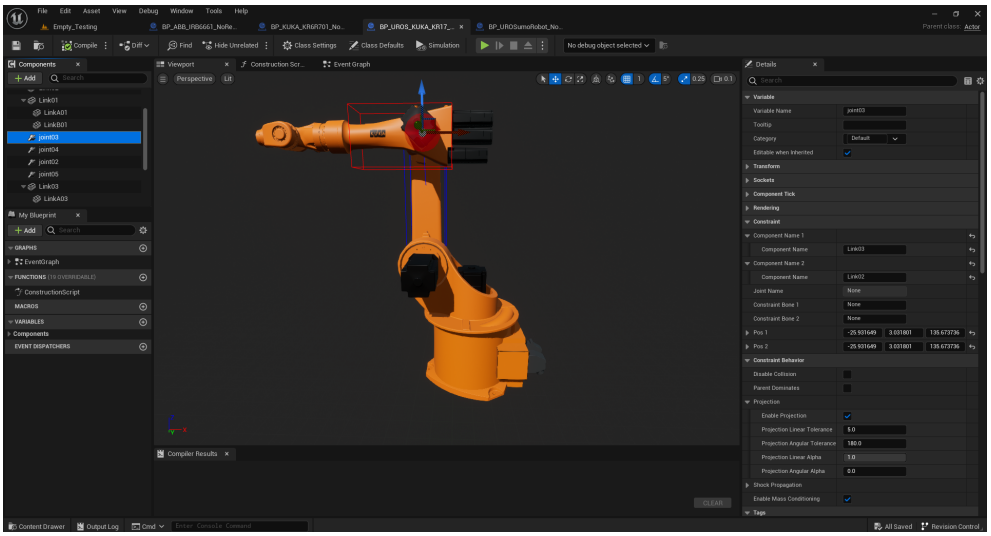

Figura 9.12: UROS SIM - Configuraciones Robot 3 - v0.1

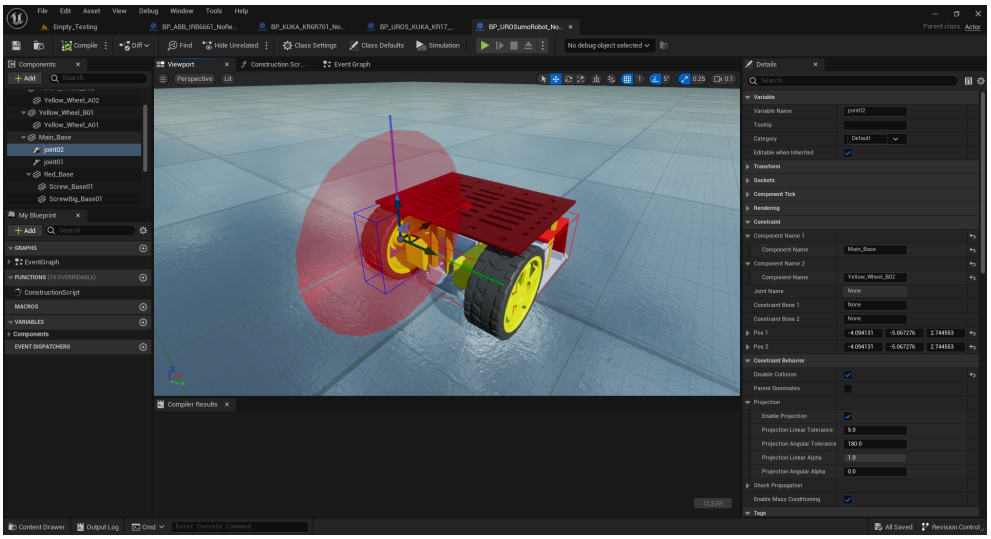

Figura 9.13: UROS SIM - Configuraciones Robot 4 - v0.1

9.0.4. URDF

Se utilizaron herramientas de ROS como rViz y otros plugins por ejemplo en solidworks para extraer y configurar modelos URDF de prueba que permitieran la simulación de los robots del entorno 3D.

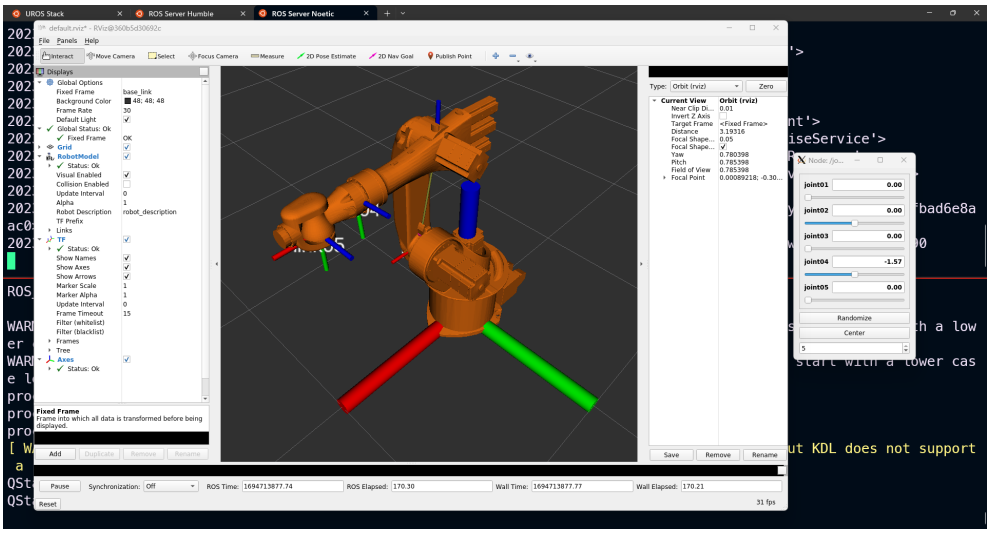

Figura 9.14: UROS SIM - URDF rViz Robot Kuka KR16 - v0.1

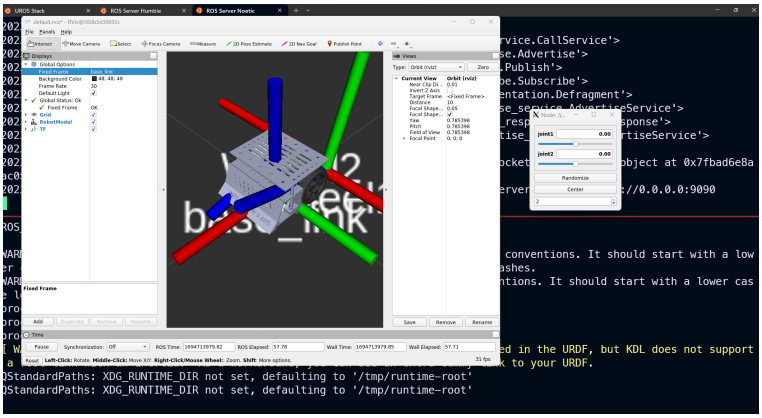

Figura 9.15: UROS SIM - URDF rViz Robot Sumo - v0.1

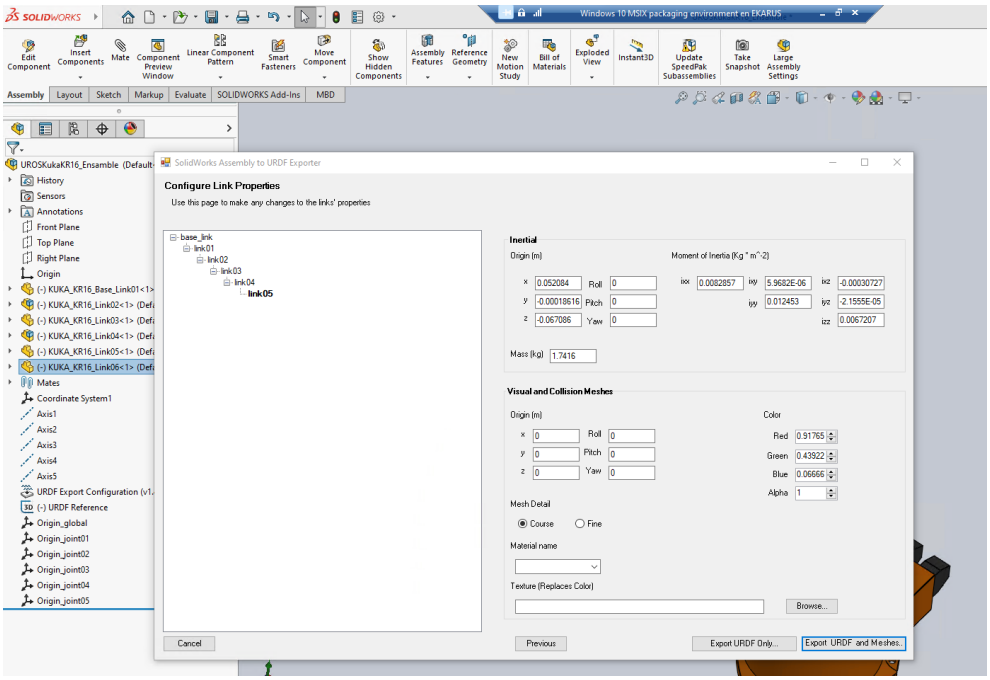

Figura 9.16: UROS SIM - URDF plugin solidworks - v0.1

9.0.5. Algoritmos de Control

Se utilizaron controles manuales escritos en Python simples de velocidad y posición para los robots simulados, estos controles se pueden construir usando blueprints en unreal engine, o utilizando lenguaje Python para publicar en los tópicos correspondientes de ROS.

9.0.6. Algoritmo teleoperación

```
1
2     def main():
3         # UrosPublisher for topic /urosBasicRobotController
4         cmd_vel_publisher = roslibpy.Topic(client, '/urosBasicRobotController', 'geometry_msgs/Twist'
      )
5
6         twist_msg = roslibpy.Message({
7             'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0},
8             'angular': {'x': 0.0, 'y': 0.0, 'z': 0.0}
9         })
10
11        def on_key_event(e):
12            twist_msg['linear']['x'] = 0.0
13            twist_msg['linear']['y'] = 0.0
14            twist_msg['linear']['z'] = 0.0
15            twist_msg['angular']['x'] = 0.0
16            twist_msg['angular']['y'] = 0.0
17            twist_msg['angular']['z'] = 0.0
18
19            if e.event_type == keyboard.KEY_DOWN:
20
21                if e.name == 'w':
```

```
                    twist_msg['linear']['x'] = 1.0
                elif e.name == 's':
                    twist_msg['linear']['x'] = -1.0
                elif e.name == 'd':
                    twist_msg['angular']['z'] = -1.0
                elif e.name == 'a':
                    twist_msg['angular']['z'] = 1.0

                if (
                    twist_msg['linear']['x'] != 0.0 or
                    twist_msg['linear']['y'] != 0.0 or
                    twist_msg['linear']['z'] != 0.0 or
                    twist_msg['angular']['x'] != 0.0 or
                    twist_msg['angular']['y'] != 0.0 or
                    twist_msg['angular']['z'] != 0.0
                ):
                    cmd_vel_publisher.publish(twist_msg)

            if e.name == 'space' and e.event_type == keyboard.KEY_DOWN:
                # UROS Controller Custom Future Functionality
                pass

            if e.event_type == keyboard.KEY_UP:

                if e.name == 'w' or e.name == 's' or e.name == 'a' or e.name == 'd':
                    if e.name == 'w' or e.name == 's':
                        twist_msg['linear']['x'] = 0.0
                    elif e.name == 'a' or e.name == 'd':
                        twist_msg['angular']['z'] = 0.0
                    cmd_vel_publisher.publish(twist_msg)
                else:
                    pass

        keyboard.hook(on_key_event)
        keyboard.wait()

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

Script 9.1: UROS SIM - Teleoperacion

9.0.7. Algoritmo tarea pick and place

El siguiente es un ejemplo de un script para ejecutar una tarea simple de pick and place con el robot industrial KUKA KR16, dentro del entorno de simulación se realizaron pruebas simples de comunicación y ejecución de tareas similares usando blueprints en Unreal Engine.

```

    def main():
        # UrosPublishers for control topics
        arm_publisher = roslibpy.Topic(client, '/uros_unreal_arm_control/command', 'trajectory_msgs/JointTrajectory')
        gripper_publisher = roslibpy.Topic(client, '/uros_unreal_gripper_control/command', 'std_msgs/Float64')

        while not client.is_connected:
            time.sleep(1)

        # Home position
        home_position = {
            'joint_names': ['joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', 'joint_6'],
```

```
            'points': [
                {
                    'positions': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                    'velocities': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                    'time_from_start': {'secs': 1, 'nsecs': 0}
                }
            ]
        }

        # Pick position
        pick_position = {
            'joint_names': ['joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', 'joint_6'],
            'points': [
                {
                    'positions': [1.0, 0.5, -0.5, 0.0, 0.0, 0.0],
                    'velocities': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                    'time_from_start': {'secs': 1, 'nsecs': 0}
                }
            ]
        }

        # Place position
        place_position = {
            'joint_names': ['joint_1', 'joint_2', 'joint_3', 'joint_4', 'joint_5', 'joint_6'],
            'points': [
                {
                    'positions': [0.5, -0.2, 0.3, 0.0, 0.0, 0.0],
                    'velocities': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                    'time_from_start': {'secs': 1, 'nsecs': 0}
                }
            ]
        }

        try:
            while True:  # Bucle infinito
                # Move to home position
                arm_publisher.publish(home_position)
                time.sleep(2)

                # Perform pick and place
                arm_publisher.publish(pick_position)
                time.sleep(2)

                # Activate gripper (adjust value)
                gripper_publisher.publish({'data': 0.5})
                time.sleep(2)

                arm_publisher.publish(place_position)
                time.sleep(2)

                # Deactivate gripper (adjust value)
                gripper_publisher.publish({'data': 0.0})
                time.sleep(2)

        except KeyboardInterrupt:
            pass

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

Script 9.2: UROS SIM - Pick Place

9.0.8. Pruebas

Sobre las pruebas hechas se hicieron dos simulaciónes, en primer lugar un robot sumo de 2 ruedas, usando un control sencillo de velocidad y el control por posición en el robot de 6 grados de libertad para efectuar una tarea básica de pick and place.

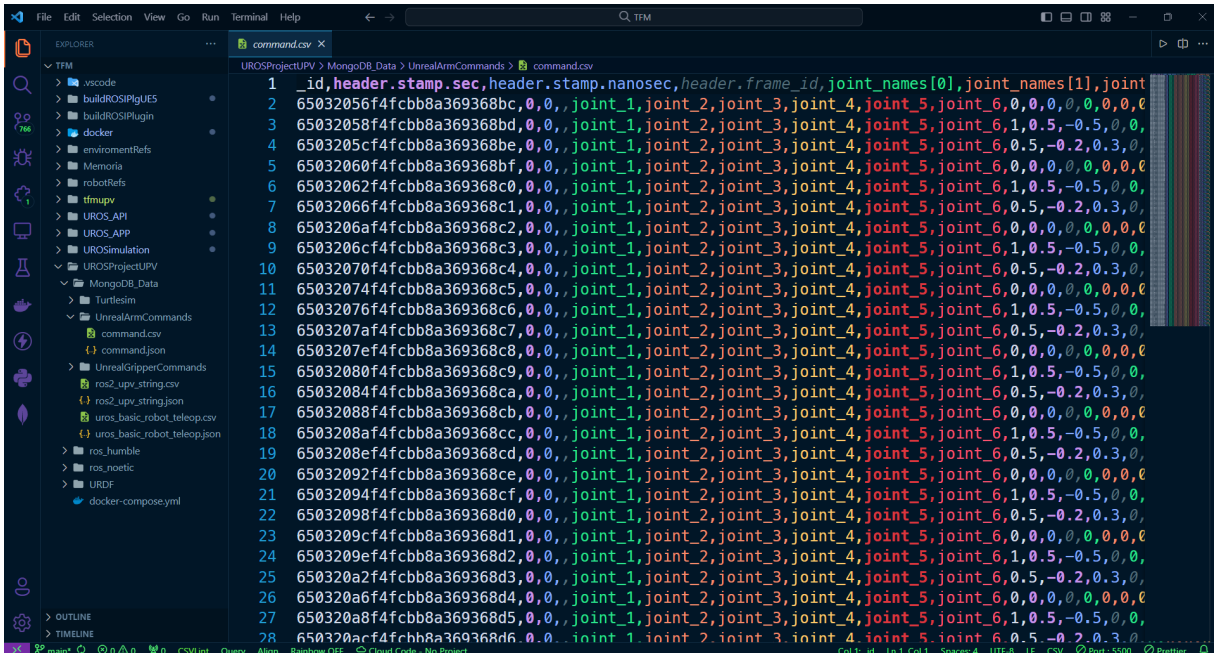

Figura 9.17: UROS SIM - Datos Pruebas CSV - v0.1

Las Pruebas realizadas evidenciaron el correcto funcionamiento y la viabilidad de la propuesta, con muchas oportunidades para implementar y ampliar las funcionalidades del software desarrollado, no se perciben ciertos retardos en la conexión entre ROS con los demás sistemas, sin embargo es necesario implementar un sistema de informes que refiera la latencia en tiempo de ejecución.

Los robots simulados y modelos 3D se comportaron como se esperaba incluso presentaron los problemas propios relacionados con configuracion correcta de ejes para realizar las tranformaciones y lograr el resultado buscado.

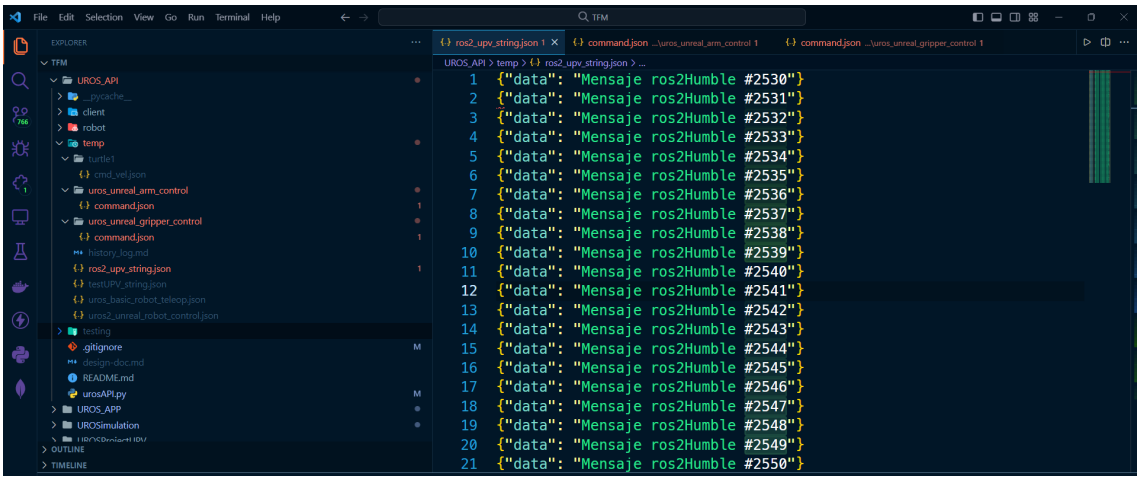

Figura 9.18: UROS SIM - Datos Pruebas JSON - v0.1

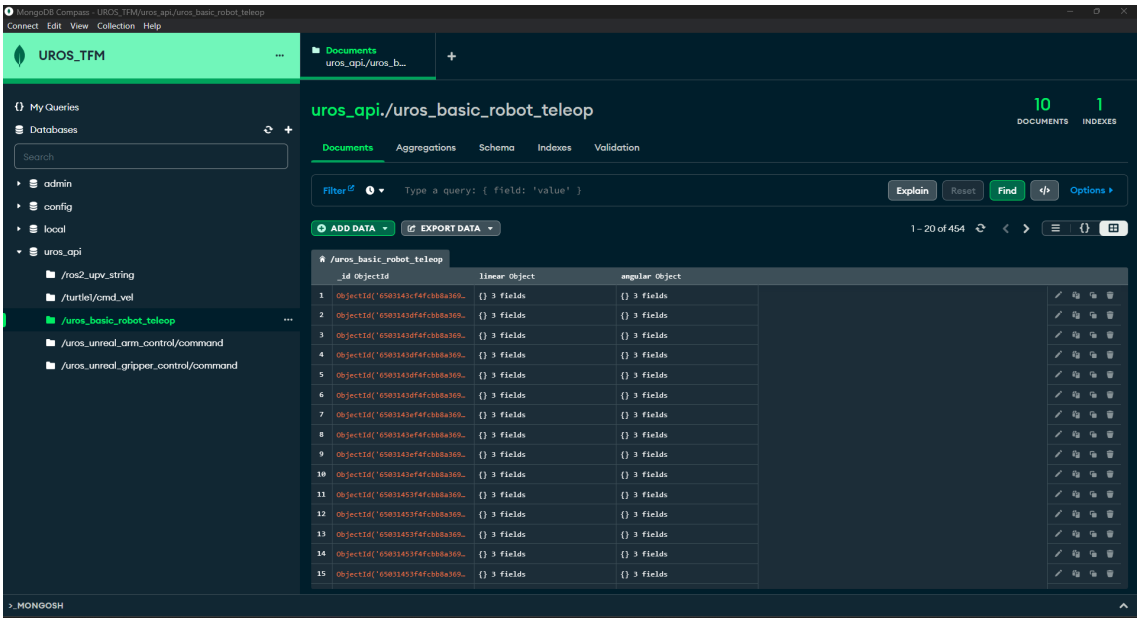

Figura 9.19: UROS SIM - Datos Pruebas MongoDB - v0.1

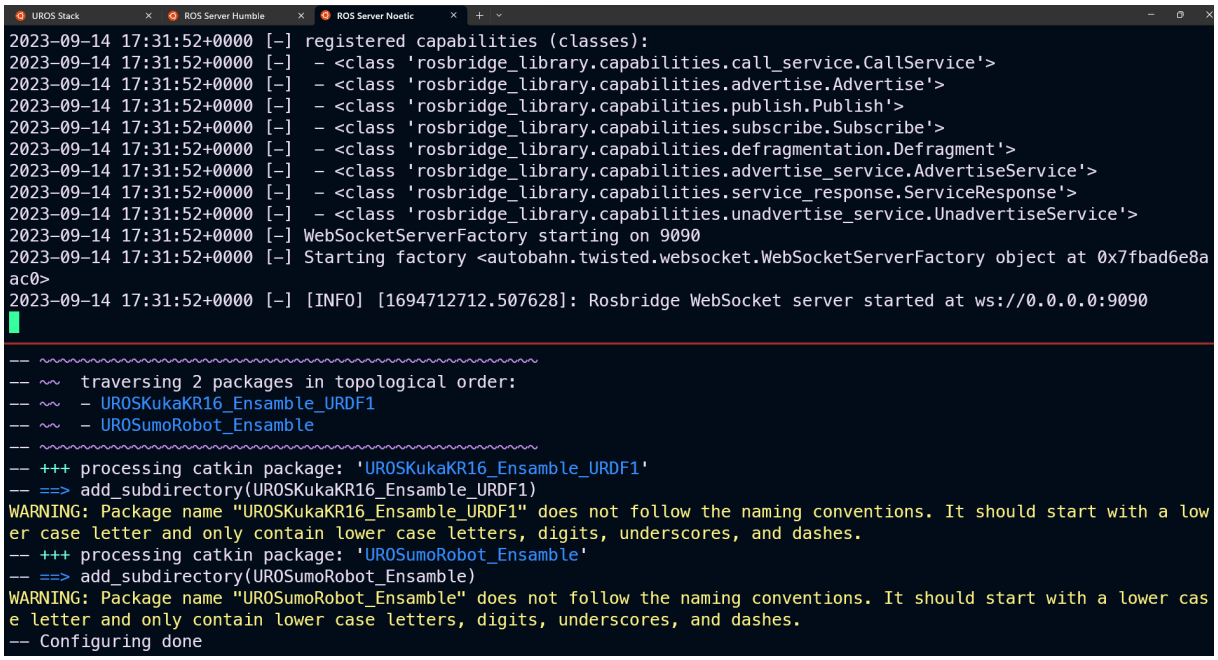

Figura 9.20: UROS SIM - Pruebas Paquetes URDF Robots - v0.1

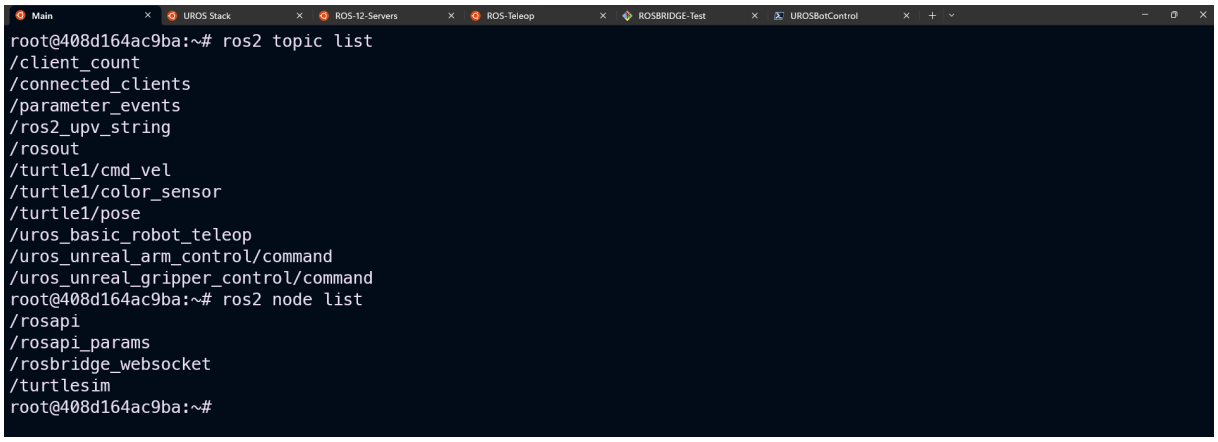

Figura 9.21: UROS SIM - Pruebas Topicos en Unreal - ROS - v0.1

Capítulo 10

Objetivos Desarrollo Sostenible (ODS)

En el marco de los objetivos para el desarrollo sostenible de las naciones unidas para 2030, el presente trabajo representa en potencia, una aportación significativa, especialmente en relación a los derechos de igualdad y disminución de la pobreza para latinoamerica.

10.1. Aportes sobre ODS 2020-2030

10.1.1. Acceso

El acceso a equipos de ultima tecnología en el campo de la robótica a nivel industrial, es bastante dificil, esencialmente por el coste que significan la adquisición, el transporte e instalación para poblaciones en circunstancias de pobreza en latinoamerica y otras partes del mundo.

En este sentido, UROS API y UROS APP, puede potencialmente ofrecer soluciones de acceso y formación educativa en experiencias de realidad virtual (VR) con unreal engine, que disminuyen significativamente los costes asociados y potencian la adopción de estas tecnologías de forma segura en estas comunidades.

10.1.2. Ubicuidad

La ubicuidad como concepto en el desarrollo de la tecnología, es decir poder utilizar la tecnología en cualquier momento y lugar, permite escenarios revolucionarios en el contexto de un mundo digitalizado y la industria 4.0. La posibilidad de participar de ambientes virtuales y a futuro poder interactuar con gemelos digitales a nivel industrial, permiten el trabajo remoto aumentando la empleabilidad de personas que se encuentran en circustancias de pobreza, sin la necesidad de cambiar su lugar de residencia con todo lo que ello implica.

10.1.3. Actualización

Muchas pequeñas y medianas empresas que por el volumen de mercado que manejan no se pueden permitir la actualización de sus equipos para sus procesos productivos, requieren evaluar en sus propios contextos estas adquisiciones, de tal forma que puedan optar por actualizar su maquinaria en la medida de sus necesidades reales.

Con trabajos como este, podrian evaluar sus líneas de producción o montaje y buscar soluciones que se encuentren al alcance de sus proyecciones económicas, reduciendo los costos asociados con la adopción de nuevas tecnologías y consiguiendo infraestructuras resilientes, además de promover la industralización inclusiva, de acuerdo a los ODS.

Capítulo 11

Conclusiones

- Unreal Engine en su versión 5 es una herramienta potente, capaz de ofrecer una experiencia virtual con gráficos realistas y es versatil al permitir del desarrollo de plugins para aprovechar y adoptar otras tecnologías y conseguir productos de alta calidad, ya se en el ambito industrial, educativo o del entretenimiento.
- Unreal Engine simplifica por medio de blueprints y acorta significativamente la curva de aprendizaje de personas que no tienen formación explicita en programación, esto permite acercar herramientas como ROS a otros usuarios que desconocen sus herramientas y podrían aprovecharlo en procesos de aprendizaje, formación profesional o desarrollo económico.
- UROS API como producto de cara a usuarios finales, no esta en una versión completa, requiere más desarrollo para poder ser aprovechado en aplicaciones reales de industria, esto incluye el desarrollar el soporte para DDS y pulir las conexiones websockets realizando pruebas exhustivas y medidas para validar su fiabilidad en ambientes de producción.
- UROS API, cumple con las funcionalidades y la intención de verificar la experiencia de integración entre ROS y Unreal Engine 5.
- UROS APP es un ejemplo de cliente, que igual que la API requiere mayor desarrollo, especialmente respecto a la seguridad del sistema y cifrado de comunicaciones.
- UROS APP cumple con las funciones que se buscaban en los objetivos del trabajo, respecto a la visualización de los datos y servir de interfaz gráfica para la API desarrollada.
- El proyecto Unreal ROS (UROS) se puede continuar orientado a convertir el entorno y laboratorio de Robótica en una aplicación VR que permita la formación de personal en el area industrial y universitario.
- Los datos recolectados y la posibilidad de exportar datos de manera transparente y simple, beneficia la transición de la industria 4.0 en empresas que todavía no han adoptado estos modelos dentro de sus líneas productivas.
- En el contexto de los objetivos del desarrollo sostenible, este trabajo brinda un punto de partida para contribuir a disminuir la desigualdad y la pobreza, permitiendo el acceso ubicuo a tecnología que su adquisición física puede ser inviable, sobre todo en comunidades de bajos recursos.

Bibliografía

[1] Sabina Jeschke and et al. *Industrial Internet of Things (IIoT): Smart Manufacturing and Industry 4.0*. Springer, 2017.

[2] Lentin Joseph. *Robot Operating System (ROS) for Absolute Beginners*. Apress, 2019.

[3] W. Qiu and A. Yuille. Unrealcv: Connecting computer vision to unreal engine. *arXiv preprint arXiv:1609.01326*, 2016.

[4] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[5] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[6] Andrew Lombardi. *WebSocket: Lightweight Client-Server Communications*. O'Reilly Media, 2015.

[7] C. Crick, G. Jay, S. Osentoski, B. Pitzer, and O. C. Jenkins. Rosbridge: Ros for non-ros users. In *Robotics Research*, pages 493–504, 2017.

[8] OpenDDS Community. OpenDDS Documentation. `https://opendds.readthedocs.io/en/latest-release/`.

[9] O. Ganoni and R. Mukundan. A framework for visually realistic multi-robot simulation in a natural environment. *arXiv preprint arXiv:1708.01938*, 2017.

[10] Karl Matthias and Sean P. Kane. *Docker: Up and Running*. O'Reilly Media, 2015.

[11] Lentin Joseph. *Mastering ROS for Robotics Programming*. Packt Publishing, 2015.

[12] Unreal Engine 5 Documentation. `https://docs.unrealengine.com/5.0/en-US/`.

[13] World Wide Web Consortium (W3C). WebSocket API Specification. `https://www.w3.org/TR/websockets/`.

[14] FastAPI Documentation. `https://fastapi.tiangolo.com/`.

[15] Stoyan Stefanov. *React Up and Running*. O'Reilly Media, 2016.

[16] Mozilla Developer Network. MDN Mozilla Documentation. `https://developer.mozilla.org/en-US/docs/Web`.

[17] uvicorn. ASGI and uvicorn. `https://www.uvicorn.org/#why-asgi`.

[18] Michaela Kümpel, Christian A. Mueller, and Michael Beetz. Semantic digital twins for retail logistics. In *Dynamics in Logistics*, page 129, 2021.

[19] Patrick Mania and Michael Beetz. A framework for self-training perceptual agents in simulated photorealistic environments. In *International Conference on Robotics and Automation (ICRA)*, Montreal, Canada, 2019.

[20] Rapyuta Robotics. rclUE a tool to enable cloud robotics simulation in Unreal Engine 4. https://www.rapyuta-robotics.com/2021/10/22/rosconjp2021blog/.

[21] Roslibpy official docs. https://roslibpy.readthedocs.io/en/latest/.

[22] Ros noetic official docs. http://wiki.ros.org/noetic.

[23] Ros humble official docs. https://docs.ros.org/en/humble/.

[24] Ros2 docker and gui. https://youtu.be/qWuudNxFGOQ?si=CRdQP32i_7wu3FiN.

[25] Marc René Zofka, Lars Töttel, Maximilian Zipfl, Marc Heinrich, Tobias Fleck, Patrick Schulz, and J. Marius Zöllner. Pushing ros towards the dark side: A ros-based co-simulation architecture for mixed-reality test systems for autonomous vehicles. In *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 204–211, 2020.

[26] Wes McKinney. *Python for Data Analysis*. O'Reilly Media, 2017.

[27] Marco Sewtz, Hannah Lehner, Yunis Fanger, Jan Eberle, Martin Wudenka, Marcus G. Müller, Tim Bodenmüller, and Martin J. Schuster. Ursim - a versatile robot simulator for extra-terrestrial exploration. In *2022 IEEE Aerospace Conference (AERO)*, pages 1–14, 2022.