Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Simulación de comportamientos de robots móviles en Unity

Simulation of mobile robot behaviors in Unity

Carlos Socas García

D. **Jonay Tomás Toledo Carrillo**, con N.I.F 78698554Y profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Bibiana Fariña Jerónimo**, con N.I.F. 54063772H investigadora Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Simulación de comportamientos de robots móviles con Unity"

ha sido realizada bajo su dirección por D. **Carlos Socas García**,

con N.I.F. 45981460K.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 12 de septiembre de 2022

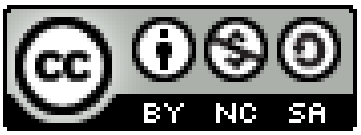
La Laguna, 12 de septiembre de 2022

Agradecimientos

Quiero dar mi mayor agradecimiento a mi familia y amigos por su incondicional apoyo sobre mis decisiones y tropiezos a lo largo de esta etapa.

También, por otro lado, expresar mi más sentido agradecimiento a mis tutores Jonay Tomás Toledo Carrillo y Bibiana Fariña Jerónimo por su gran apoyo dado a la hora de realizar el proyecto.

Licencia



Índice general

Índice de figuras

Índice de tablas

Resumen

Unity es un motor de desarrollo de videojuegos (game engine) el cual tiene una serie de rutinas de programación que permiten el diseño, creación y funcionamiento de un entorno interactivo. Es un software que centraliza todo lo necesario para poder desarrollar videojuegos. En este caso, se utilizará para generar nuestra simulación. Por otro lado, ROS es un framework para el desarrollo de robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo.

La idea principal de este proyecto es plantear una simulación donde sea posible analizar el comportamiento de un robot móvil, en este caso una silla con tracción diferencial, mientras se mueve por determinados espacios. La finalidad es generar mediante ecuaciones odométricas a partir de las velocidades angulares de las ruedas la trayectoria estimada, y compararla con el recorrido que realmente realiza el robot en la simulación. Esto permitirá obtener un ejemplo visual de situaciones que se pueden dar en entornos reales cuando la silla desliza o derrapa, lo que conlleva a diferencias en la trayectoria estimada por las ruedas respecto a la de referencia.

Palabras clave: ROS, Unity, Simulación, tracción diferencial, robot, espacio, velocidades angulares

Abstract

Unity is a game development engine (game engine) which has a series of programming routines that allow the design, creation, and operation of an interactive environment. It is a software that centralizes everything necessary to develop video games. In this case, it will be used to generate our simulation. On the other hand, ROS is a framework for the development of robots that provides the functionality of an operating system in a heterogeneous cluster.

The main idea of this project is to propose a simulation where it is possible to analyze the behavior of a mobile robot, in this case a chair with differential traction, while it moves through certain spaces. The purpose is to generate the estimated trajectory using odometric equations from the angular wheel velocities and, compare it with the route carried out by the robot in the simulation. This will allow to obtain a visual example of situations that can occur in real environments when the chair slides or skids, which leads to differences in the trajectory estimated by the wheels with respect to the reference one.

Keywords: ROS, Unity, Simulation, differential drive, robot, track, angular velocities

Capítulo 1
Introducción

1.1 Robots

Para realizar este proyecto, el robot presentado es una silla de ruedas. En este caso, esta silla de ruedas tendrá una serie de sensores capaces de detectar la velocidad de giro de sus ruedas traseras que son las encargadas de generar el torque para el movimiento.

El movimiento del robot está basado en una tracción diferencial, la cual se trata de una configuración bastante común para sistemas utilizados en interiores, ya que permite al robot girar sobre su propio eje. De esta manera, el robot puede moverse en espacios congestionados con cierta facilidad. La tracción diferencial utiliza dos ruedas controladas individualmente con una o varias ruedas locas como puntos de apoyo, en este caso son dos puntos de apoyo [1].

Figura 1: Silla de ruedas autónoma

1.2 Motivación

El objetivo final de este proyecto es la necesidad de un motor de simulación que permita validar algoritmos de localización de una silla de ruedas autónoma sin necesidad de utilizar ningún prototipo físico. Para ello se busca crear una simulación del prototipo digital capaz de moverse entre distintos tipos de superficie con condiciones de deslizamiento reales y adaptarse a medida que va avanzando en un entorno simulado para poder plantearlo en la realidad con mayor facilidad.

El interés por el que se crea este proyecto proviene de comparar, de la manera más precisa posible, la capacidad de obtención de datos de los sensores implantados en nuestro robot y ver los defectos al comparar diferentes modelos que se presenten en distintos terrenos y ver cómo es capaz de influir en la trayectoria que lleve el robot.

1.3 Antecedentes y Estado actual

Unity es un motor de desarrollo de videojuegos (game engine) el cual tiene una serie de rutinas de programación que permiten el diseño, creación/funcionamiento de un entorno interactivo. Es un software que centraliza todo lo necesario para poder desarrollar videojuegos. En este caso, se utilizará para generar nuestra simulación [2].

ROS (Robot Operating System) es un framework de desarrollo software Open Source para diseñar aplicaciones para robots. ROS ofrece una plataforma de desarrollo software estándar para todos los desarrolladores para poder enseñar desde los fundamentos básicos/prototipado hasta la producción de este [3].

Para la simulación, se ha preparado un modelo de silla con tracción diferencial en Unity. Este tipo de direccionamiento viene dado por la diferencia de velocidades de las ruedas laterales. La tracción se consigue por dos ruedas montadas que son independientemente propulsadas y controladas, proporcionando ambas tracción y direccionamiento. Por otro lado, se presentan dos ruedas de apoyo a las ruedas motrices, pero sin un movimiento fijo.

En resumen, este proyecto nació de la necesidad de obtener un modelo capaz de imitar la realidad para poder realizar pruebas y ajustes sin la necesidad de probar en el prototipo final. Además, es posible simular el comportamiento sin tener que utilizar sujetos de pruebas reales en condiciones de riesgo, pudiendo así comprobar el comportamiento con distintos parámetros como materiales o pesos.

1.4 Estructura de la memoria

En este caso, la memoria consta de siete capítulos. Se puede observar cómo, en el capítulo 1 se lleva a cabo una breve introducción del proyecto junto con la descripción de los objetivos a conseguir. Asimismo, en el capítulo 2 se detalla el software requerido para llevar a cabo la simulación en estudio. El capítulo 3 describe la metodología seguida y sus características. El capítulo 4 resume los diferentes resultados obtenidos en la simulación y su validación en el entorno real. Finalmente, se desarrollan unas conclusiones con líneas a futuros proyectos emergentes en español e inglés en el capítulo 5 y 6 respectivamente, junto a un presupuesto general del proyecto presentado en el capítulo 7.

Capítulo 2
Software

2.1 Tecnologías usadas

2.2.1 **Unity**

Unity es un motor gráfico multiplataforma creado por Unity Technologies en el año 2001. El término motor de videojuego del que hace uso Unity se refiere a un software que tiene una serie de rutinas de programación que permiten el diseño, creación y funcionamiento de un entorno interactivo. Unity está disponible como plataforma de desarrollo para Windows, OS X y Linux y este permite crear proyectos para los anteriormente mencionados, además de consolas como pueden ser PS4, Xbox One, también incluyendo móviles IOS y Android [2].

Figura 1: Logo Unity

Unity es probablemente la tecnología de desarrollo de videojuegos que mayor uso tiene en estos momentos. Uno de los puntos fuertes que tiene es su gran comunidad. Esto permite tener acceso a multitud de documentación, foros, assets y comunidades, además de ser uno de los motores predilectos para aprender el desarrollo de videojuegos [2].

En Unity, es destacable dos versiones, una limitada o gratuita y una versión de pago o versión completa [4]. En este caso, se utiliza la versión de gratuita, ya que es la enfocada principalmente al aprendizaje o para el desarrollo de videojuegos pequeños. Una de las mayores ventajas que aporta Unity 3D es que permite desarrollar juegos para distintas plataformas partiendo de un proyecto general. A diferencia de otras tecnologías, con Unity 3D, prácticamente no existe necesidad de ajustar parámetros dependiendo de la plataforma, salvo si fuese necesario usar funcionalidades específicas de la plataforma objetivo.

El motor gráfico usado por Unity es OpenGL (en Windows, Mac y Linux), Direct3D (solo en Windows) y OpenGL ES (Android y IOS). También cuenta con soporte para mapeado de relieve, sombras dinámicas utilizando mapas de sombra, render a textura y efectos de post-procesamiento [2].

Para la programación de la lógica del proyecto, se tiene a disposición un amplio abanico de posibilidades. Las posibilidades que nos brinda son el uso de JavaScript, C# y un derivado de Python llamado Boo. Además, Unity hace uso de su IDE integrado, MonoDevelop [4]. El entorno de desarrollo que presenta Unity se trata de uno de los más sencillos y potentes.

Como se nombró anteriormente, uno de los lenguajes utilizados por Unity es C#. Como se podrá observar, este es el lenguaje elegido para el desarrollo del proyecto, ya que es el predilecto durante el uso de Unity y el recomendado por los creadores de las librerías preestablecidas para el desarrollo de este proyecto [5].

Sus características principales son que se trata de un lenguaje potente y sencillo, es una evolución de los lenguajes C y C++ que, a su vez, aprovecha las utilidades de sus anteriores versiones. Además, presenta mejoras en áreas como seguridad de tipos y control de versiones, así como su acceso a las APIs más comunes como es el caso de .NET Framework [4].

2.2.2 ROS

ROS es un meta sistema operativo de código abierto para robots. Provee de servicios que se esperarían de un sistema operativo, incluyendo abstracción de hardware, control de dispositivos de bajo nivel, implementación de funcionalidades comunes, pasaje de mensaje entre procesos y manejo de paquetes. También brinda herramientas y librerías para obtener, construir, escribir y correr código a través y mediante varios sistemas [3].

Figura 2: Logo ROS

Las características principales de ROS son:

- **Liviano**: está diseñado para ser lo más liviano posible, de modo que el código escrito se pueda usar con otros frameworks. Uno de los mejores puntos de esto es que ROS es fácil de integrar con otros marcos de software de robot como OpenRAVE, Orocos y Player [5].
- **Bibliotecas agnósticas ROS**: el modelo de desarrollo preferido es escribir bibliotecas agnósticas en ROS con interfaces funcionales limpias [5].
- **Independencia del lenguaje**: es fácil de implementar en cualquier lenguaje de programación moderno. Ya se ha implementado en Python, C ++ y Lisp y existen bibliotecas experimentales en Java y Lua [5].

- **Pruebas sencillas**: tiene un marco de prueba de integración/unidad incorporada llamado “rostest” que facilita la activación y desactivación de dispositivos de prueba [5].
- **Escalado**: es apropiado para grandes sistemas de rutinas de ejecución y para grandes procesos de desarrollo [5].

Es necesario asentar unas bases con respecto a términos necesarios para la compresión de la tecnología.

- **Paquetes**:

  El software de ROS se organiza en paquetes y cada paquete contiene una combinación de código, datos y documentación. Estos paquetes se encuentran dentro del directorio de trabajo, dentro de la carpeta ***src***. Cada paquete debe tener obligatoriamente un CMakeList.txt que se trata del fichero de compilación, y el fichero package.xml que describe el contenido del paquete y como se debe interactuar con él [5].

- **Nodos**:

  Dentro de los paquetes ROS, se pueden crear archivos ejecutables llamados nodos. Los nodos son unidades de procesamiento software de datos donde se implementan todos los requisitos funcionales para una aplicación de ROS. Estas unidades de software utilizan las bibliotecas de ROS para comunicarse con otros nodos que se encargan de publicar o subscribirse al mismo tópico o tema, además de proporcionar o utilizar algún servicio [5].

- **Tópicos**:

  Para comunicarse entre nodos, se usan los Topics o Tópicos. Tiene un funcionamiento sencillo donde, desde un nodo, se publica el tópico y, desde el otro nodo, se subscribe. Gracias a eso, es posible conectar varios nodos y pueden interaccionar entre ellos enviando y recibiendo mensajes o servicios [5].

  Es importante que los mensajes o servicios que envíe un nodo y que espere recibir otro nodo sean del mismo tipo y que presenten la misma estructura, si no existirá un error en la comunicación [5].

- **Mensajes**:

  Los mensajes se programan para que se envíen de forma constante, con o sin tiempo de separación entre uno y otro. Además, la perdida de algún mensaje debe tener poca relevancia al ser una comunicación corta y repetitiva [5].

- **Servicios**:

  Los servicios se diferencian de los mensajes, ya que tienen una mayor relevancia y necesitan de confirmación de que la comunicación se ha realizado con éxito, ya que solo se envían a petición de algún evento determinado. Esto ayuda a no sobrecargar el sistema [5].

2.2.3 Docker

Para la realización de este proyecto, se ha aconsejado el uso de la herramienta Docker. Este es un sistema operativo para contenedores. De manera similar a cómo una máquina

virtual virtualiza el hardware del servidor, los contenedores virtualizan el sistema operativo de un servidor [6].

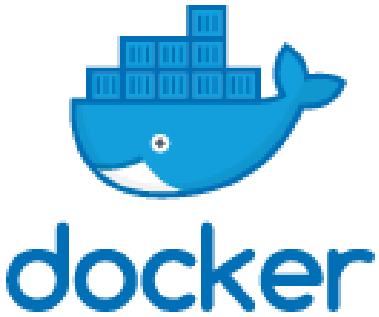


Figura 3: Logo Docker

Los contenedores Docker son una abstracción en la capa de aplicación que empaqueta código y dependencias juntas. Además, cuando varios contenedores están creados en una misma máquina, estos pueden compartir el sistema operativo del kernel corriendo en cada uno de ellos como procesos aislados. Esto hace que, si creamos varios contenedores iguales, sólo uno de ellos ocupe espacio en el servidor. Docker proporciona comandos sencillos que se pueden utilizar para crear, iniciar o detener contenedores [6].

Los contenedores Docker se generan mediante el uso de imágenes que se forman a partir de ficheros de configuración llamados Dockerfile. En estos Dockerfiles, se describen todo el software que encapsulará los contenedores [6].

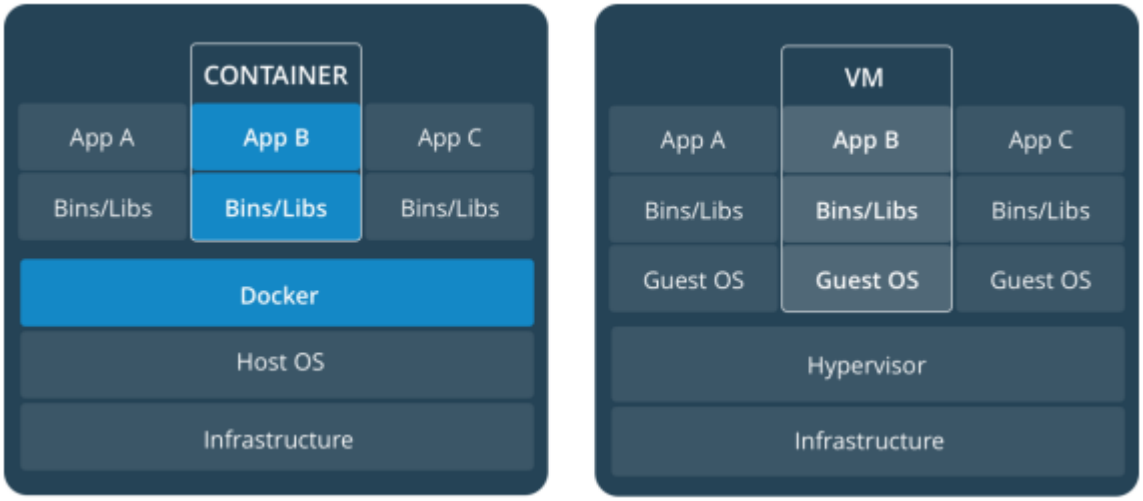


Figura 4: Estructura Docker vs Máquina virtual

Como se puede observar en la Figura 4, la mayor diferencia reside en las necesidades de instalación. Por un lado, una máquina virtual necesita instalar todo un sistema operativo que se comunica a través del hipervisor que virtualiza el hardware de la máquina. Por otro lado, los contenedores son procesos Docker dentro del propio sistema operativo de la máquina que aíslan la aplicación junto a sus dependencias [6].

Docker permite generar y enviar código con mayor rapidez, estandarizar las operaciones de las aplicaciones, transferir el código con facilidad y ahorrar en recursos [6].

Capítulo 3
Desarrollo del proyecto

3.1 Recorrido

Para el recorrido, se ha generado un modelo del pasillo inferior del edificio de Física y Matemáticas. Este recorrido consiste en un movimiento a través del pasillo, dar una media vuelta y volverá al punto inicial (ver Figura 5).

El modelo se ha generado tomando una referencia de los materiales allí presentes y de la referencia catastral para poder generar las medidas aproximadas de su estructura.

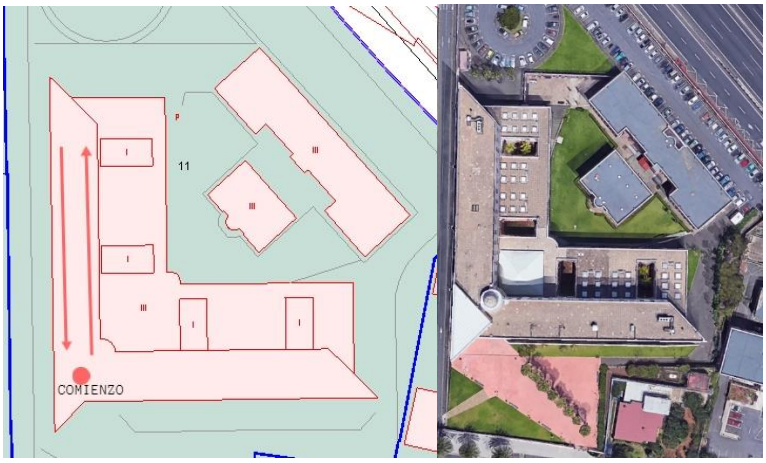


Figura 5: Vista aérea del recorrido

3.2 Metodología

3.2.1 Unity

3.2.1.1 Importación de librerías Unity-ROS y modelo.

Para comenzar con el proyecto en Unity, se debe importar la silla de ruedas modelo. Como se puede observar en la Figura 6, se trata de un modelo de silla que sigue el movimiento de tracción diferencial y es muy similar al modelo real que se busca usar.

Para que la silla se mueva con total libertad, es necesario usar las librerías presentes en Unity como es el caso de la librería de ***WheelCollider***. ***WheelCollider*** permite generar una rueda que se superpone a la rueda normal de nuestra silla [7]. Gracias a este método, es posible controlar el movimiento y dirección de las ruedas.

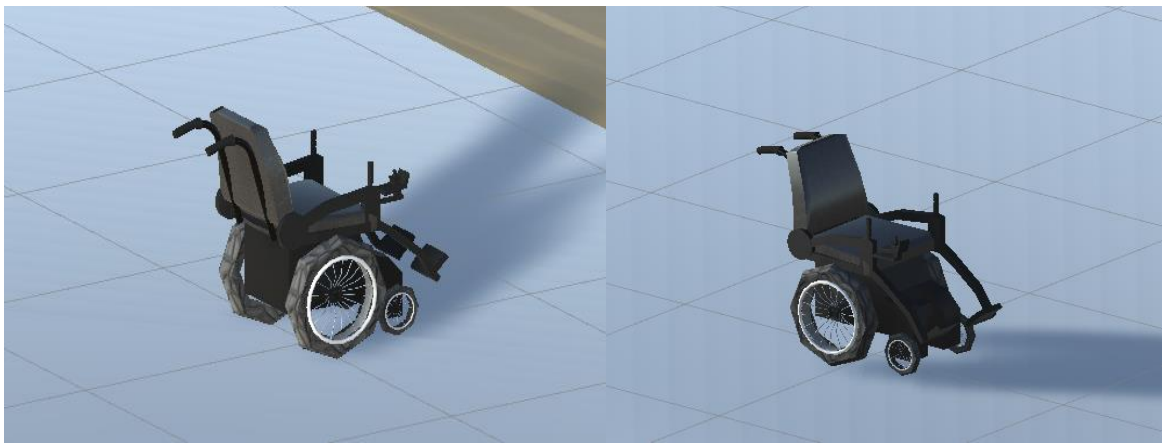


Figura 6: Modelo en Unity

Siguiendo las recomendaciones de Unity para la implementación de ROS, es necesario descargarse unas librerías de Unity Robotics Hub para poder crear la conexión vía TCP Unity-ROS. Gracias a esta funcionalidad añadida por parte de Unity e instalada mediante el uso de Git, podemos acceder a las funciones básicas que nos permitirán conectar Unity con ROS [8].

En este caso, se nos añade un desplegable adicional como se puede ver en la Figura 7 donde se pueden definir los parámetros para la conexión TCP Unity-ROS, así como generar msg y srv creados en ROS para poder usarlos en Unity.

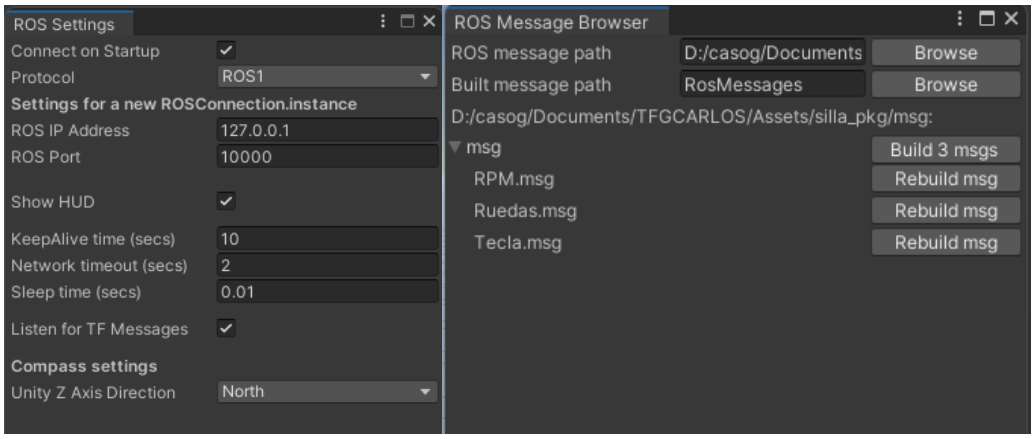


Figura 7: Opciones de conexión con ROS y Generación de mensajes

3.2.1.2 Creación conexión Unity-ROS (Parte de Unity)

Para empezar, se utilizará la opción de **TCP** Unity-ROS para definir los parámetros para poder conectarse a ROS. En este caso, es necesario establecer el puerto a **10000** usando la IP local **127.0.0.1** (loopback). Con la configuración básica realizada, es necesario generar la conexión desde Unity con código en C# (ver Figura 8).

```
public class RosSubscriber : MonoBehaviour
{
    ROSConnection ros;
    private float timeElapsed;

    public SimpleCarController Silla;

    void Start()
    {
        ROSConnection.GetOrCreateInstance().Subscribe<PosRotMsg>("mv",
Movement);
    }
    void Movement(PosRotMsg move)
    {
```

Figura 8: Creación de conexión

Como se puede observar en la Figura 8, es necesario generar el objeto "**ros**" para poder crear el acceso a los recursos que da ROS desde Unity. Con el uso de "***ROSConnection***", se establece el nodo creado desde donde partimos y busca o crea el tópico a subscribirse.

3.2.1.3 Estructuras de la simulación

Con ello, se obtiene la silla para poder ser usada en el proyecto de Unity. Asimismo, es necesario generar un plano donde se realicen los movimientos generados por la silla. Para ello, se debe establecer un objeto "***3D Plane***" para poder implantar ese contexto. Este plano va a estar dotado de una dimensión de **100 x 100 unidades**, ya que, para el buen funcionamiento y realizado de la prueba del modelo, es necesario bastante espacio por si fuese necesario, para posteriormente, pasar al modelo del pasillo de la facultad de física.

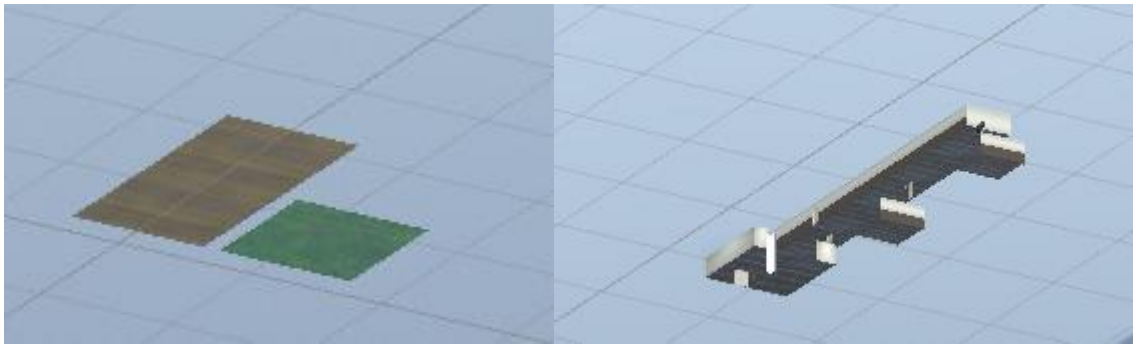


Figura 9: Planos ejemplo y Modelo pasillo Facultad Física

Como se puede observar en la Figura 9, se presentan dos escenarios básicos. Por un lado, se trata de un entorno básico con diferencias en las texturas de los planos para conocer a simple vista las diferencias en **fricción** utilizadas. Por otro lado, se observa el mapeado final del modelo basado en el pasillo de la facultad de Física anteriormente mencionada.

Unity permite modificar la fricción de los objetos mediante el uso del denominado "***Physic Material***" [7]. Gracias a este componente, es posible modificar la fricción dinámica y estática que un plano pueda ejercer sobre un objeto. Este coeficiente varía entre **0 y 1**. En el caso de la fricción dinámica que sucede mientras el objeto se encuentra en movimiento, el valor de 0 se corresponde a una superficie resbaladiza como pudiera ser conducir sobre hielo, mientras que, el valor de 1 se trata de una superficie capaz de dejar el objeto estable, sin movimiento, rápidamente.

Asimismo, en el caso de la fricción estática, el valor de 0 se correspondería a una superficie sin

capacidad de oponer fricción para detener el objeto a mover, y el valor de 1 hará que sea más difícil mover el objeto de su posición inicial.

3.3.1.4 Creación de scripts

Para este proyecto, fue necesario generar, desde ROS, varios scripts capaces de recibir el input de control generado en ROS y transformarlo a los movimientos representados en la simulación propia de Unity, así como tratar y enviar los datos generados por esos movimientos para poder generar el modelo simulado por ecuaciones desde ROS (ver Figura 10).

```
public class RosSubscriber : MonoBehaviour
{
    ROSConnection ros;
    private float timeElapsed;

    public SimpleCarController Silla;

    void Start()
    {
        ROSConnection.GetOrCreateInstance().Subscribe<PosRotMsg>("mv", Movement);
    }
    void Movement(PosRotMsg move)
    {

        Silla.InputValues(move.pos_z, move.rot_z);

    }
}
```

Figura 10: Input de ROS

En el caso de Unity, primero se trata el input recibido desde ROS para conseguir el control deseado del robot. Como se puede observar en la Figura 10, se busca o crea el tópico "***mv***". En este caso, se busca el tópico ya generado para poder recibir los mensajes de tipo "***PosRotMsg***" publicados desde el sistema ROS. Esos mensajes se redirigen a la función "***Movement***" que recibe una variable de tipo "***PosRotMsg***".

Como segundo paso para poder recibir y utilizar los inputs generados en **ROS**, podemos ver en la Figura 11 como el objeto de clase "***SimpleCarController***" tiene 4 valores a rellenar. Se ha creado una clase previa para poder almacenar las ruedas a las que se les va a introducir el input proveniente de **ROS** donde se puede observar como la clase recoge las ruedas lógicas, tanto derecha como izquierda, el torque del motor y el giro.

```
[System.Serializable]
public class AxleInfo {
    public WheelCollider leftWheel;
    public WheelCollider rightWheel;
    public bool motor; // is this wheel attached to motor?
    public bool steering; // does this wheel apply steer angle?
}
```

Figura 11: Clase para administrar ruedas del robot

A partir de las indicaciones de control proporcionadas desde ROS se procede a simular el movimiento objetivo. Se accede a cada par de ruedas en cada iteración del bucle (ver Figura 12), en este caso, solo es una iteración al solo existir un par de ruedas que se puedan controlar. El modelo de silla con la tracción diferencial establece que solo dos ruedas funcionen de forma motriz y de giro, mientras que el otro par de ruedas se establece como ruedas sin ningún tipo de control.

```
public void InputValues(float pos_z, float rot_z)
{

    motor = maxMotorTorque * pos_z; //Lineal
    steering = maxSteeringAngle * rot_z; //Angular

    foreach (AxleInfo axleInfo in axleInfos)
    {
        if (axleInfo.motor)
        {
            axleInfo.leftWheel.motorTorque = motor - steering;
            axleInfo.rightWheel.motorTorque = motor + steering;
        }
    }
```

Figura 12: Función de control del robot

Para conseguir el movimiento objetivo, se reducirá la velocidad de giro en la rueda izquierda en el caso de girar a la izquierda, y aumentará la velocidad de giro de la derecha para compensar y realizar ese giro y viceversa. El input generado en ROS está basado en datos del intervalo de [0, 1], por ello, al recibir ese dato desde ROS, se multiplica por el valor de la potencia ejercida sobre las ruedas para poder mover las ruedas en el entorno de Unity. Esto permite el correcto movimiento de la silla simulada siguiendo la trayectoria deseada descrita por los comandos introducidos en ROS.

Mientras la silla se mueve se recoge dos tipos de información diferentes que serán enviados a ROS para su posterior procesamiento: (1) La posición real de la silla simulada en cada instante que extrae la información de su posición de los valores a tiempo real dados desde Unity. En este caso, se crea un tópico nuevo "***DatosStamped***" donde se mandarán los mensajes de las posiciones y rotaciones de la silla de Unity. (2) La velocidad angular de las ruedas en cada instante que representará los resultados de la odometría de la silla de ruedas, que son extraídos de las propias revoluciones dadas por las ruedas lógicas generadas en Unity "**WheelCollider**" y que se traspasan mediante mensajes separados y formateados a radianes por segundo.

```
private void Update()
{

    wRight = 2 * 3.14159 * (Silla_root.axleInfos[0].rightWheel.rpm / 60);
    wLeft = 2 * 3.14159 * (Silla_root.axleInfos[0].leftWheel.rpm / 60);
```

Figura 13: Cálculo de velocidad angular

Gracias a esto, se puede tener en cuenta los datos a tiempo real cuando se haga la comparación con la posición generada en ROS mediante cálculos matemáticos. Finalmente, con el script mostrado en la Figura 14, se generan nuevos tópicos desde el nodo y se publican las velocidades angulares pasadas a radianes por segundos.

```
            RuedasMsg mensaje = new RuedasMsg(
                wRight
            );

        // Finally send the message to server_endpoint.py running in

            ros.Publish("wheel_rpm_right", mensaje);

            RuedasMsg mensaje_left = new RuedasMsg(
                wLeft
            );

            ros.Publish("wheel_rpm_left", mensaje_left);

}
```

Figura 14: Publicación de mensajes

3.2.2 ROS

3.2.2.1 Creación de entorno (Docker).

Como se comenta anteriormente, lo recomendado por Unity para realizar el conjunto Unity – ROS es el uso de Docker (ver Figura 15).

```
docker build -t melodic -f ros_docker/Dockerfile .
docker run -it --rm -p 10000:10000 melodic /bin/bash
```

Figura 15: Comandos para instalación Docker

Como se puede observar en la Figura 15, al seguir los pasos recomendados de la instalación de *ROS*, se recurre al uso de *Docker* que permitirá montar una imagen de *ROS Melodic*. ***ROS Melodic Morenia*** o únicamente *ROS Melodic* se trata de la doceava versión de *ROS* [9]. Una vez se prosiguen con los comandos necesarios, se crea con "***Docker build"*** el contenedor con el software de ROS Melodic. El último paso del setup es establecer el directorio base para la compilación y ejecución de los programas.

Gracias a la imagen ya creada por Unity para generar este contenedor de software, se obtienen una serie de scripts, mensajes ROS que permiten establecer la comunicación de ROS tanto dentro como fuera [8]. La guía finaliza estableciendo uno de los comandos ROS más importantes que es la ejecución del servidor para que pueda conectarse desde Unity a ROS y poder enviar mensajes entre ellos (ver Figura 16).

```python
#!/usr/bin/env python

import rospy

from ros_tcp_endpoint import TcpServer


def main(args=None):
    # Start the Server Endpoint
    rospy.init_node("unity_endpoint", anonymous=True)
    tcp_server = TcpServer(rospy.get_name())
    tcp_server.start()
    rospy.spin()
```

Figura 16: Servidor ROS en Python

3.2.2.2 Creación y uso de scripts para control.

Para el uso de *ROS*, es necesario preestablecer una serie de scripts para poder generar, tanto el output a la silla generada en *Unity*, como para poder recabar la información para generar la propia simulación. En este caso, será necesario usar un script de librerías de *ROS*. Para ello, se usará *teleop_twist_keyboard* [10] que se trata de un input genérico para el control de dirección.

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Gracias al anterior comando, se obtiene un input layout desde la línea de comandos para poder generar la dirección que va a recibir nuestra silla simulada en Unity, como podemos observar en la Figura 17.

```
Reading from the keyboard  and Publishing to Twist!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
anything else : stop

CTRL-C to quit
```

Figura 17: Esquema de control

Para el control sobre el modelo de la silla se seguirán las instrucciones dadas por el propio programa. En este caso, se utilizará la "**i**" y la "***coma***" para poder controlar la dirección hacia a delante y hacia detrás, respectivamente. Asimismo, con el uso de "***J***" y la "***L***" para poder controlar la dirección de izquierda y derecha respectivamente.

Así, el mensaje que genera este script es de tipo *Twist*, por ello, se realiza un paso de conversión de los datos a estructuras más simples y se redirige mediante un mensaje que ya comparten entre *ROS* y *Unity*: "*PosRotMsg*". Para realizar esa conversión, se ha generado un script donde se recoge los datos necesarios del mensaje enviado (ver Figura 18). Con esto, ya se crean las señales de control perfectamente entendibles por Unity para el control de la silla en la simulación.

```python
def callback(data):

    ### Posicion
    msg.pos_x = data.linear.z
    msg.pos_y = data.linear.y
    msg.pos_z = data.linear.x
    ### Angulo
    msg.rot_x = data.angular.x
    msg.rot_y = data.angular.y
    msg.rot_z = data.angular.z

    pub.publish(msg)


if __name__ == "__main__":
    try:
        msg = PosRot()
        rospy.init_node('cmd_vel_to_unity')
        pub = rospy.Publisher('mv', PosRot, queue_size=10)
        sub = rospy.Subscriber('cmd_vel', Twist, callback)
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

Figura 18: Transformador de mensajes

3.2.2.3 Creación y uso de scripts para ecuaciones.

En este momento, ya se tienen todos los datos necesarios para generar, mediante ecuaciones odométricas (ver Figura 19) y junto al uso de la velocidad angular de las ruedas, los datos necesarios para la creación de un modelo odométrico de la silla simulada en Unity.

$$v_{r/l} = w_{r/l} R_{l/r} = \frac{2\pi c k_{r/l}}{encRes \triangle t} R_{l/r}$$

$$v = \frac{v_l + v_r}{2} \qquad w = \frac{(v_l - v_r)}{D}$$

$$\theta_{k+1} = \theta_k + w\triangle t$$
$$X_{k+1} = X_k + vcos(\theta_{k+1})\triangle t$$
$$Y_{k+1} = Y_k + vsin(\theta_{k+1})\triangle t$$

Figura 19: Ecuaciones odométricas a partir de la velocidad angular

Para generar el modelo a partir de las ecuaciones, es necesario calcular las velocidades angulares de ambas ruedas de la silla partiendo de las revoluciones por minuto a radianes por segundo. Asimismo, al dividir por el radio de las ruedas, se obtiene la velocidad lineal de nuestra silla. Gracias a ello, es posible establecer los tres datos (x, y, theta) necesarios para conocer la posición a tiempo real a partir de un sensor que capta la velocidad angular de las ruedas. Como se puede observar, las ecuaciones representadas en la Figura 19 se desarrollan en la Figura 20 por medio del lenguaje de programación de Python.

```python
def odom(data):
    global primero, x, y, th, v, w, dt, my_velocity_l, my_velocity_r, t0, t1

    if primero == 0:
    t0 = rospy.Time.now().to_sec()
    primero = 1
    return

    timeStamp = rospy.Time.now()

    my_odom = data

    t1 = rospy.Time.now().to_sec()

    vl = (my_velocity_l) * 0.25
    vr = (my_velocity_r) * 0.25

    dt = (t1 - t0)

    v = (vl + vr) / 2
    w = (vl - vr) / 0.5

    x = x + ((v * cos(th)) * dt)
    y = y + ((v * sin(th)) * dt)
    th = th + (w * dt)

    odom_quat = tf.transformations.quaternion_from_euler(0, 0, th)
```

Figura 20: Cálculos odométricos en ROS

Finalmente, mediante la posición y giro obtenida por las ecuaciones, se puede generar la posición a tiempo real de la silla. Para poder representarlo y compararlo, se crea un mensaje de ***Odometry*** para poder enviar los resultados a la herramienta ***RViz***, donde es necesario enviar la posición X e Y junto al quaternion calculado con anterioridad, así como los valores de la velocidad lineal (v) y la velocidad angular (w) (ver Figura 21).

```python
odometry = Odometry()
odometry.header.stamp = timeStamp
odometry.header.frame_id = "odom"
odometry.child_frame_id = "base_link"


odometry.pose.pose.position.x = x
odometry.pose.pose.position.y = y
odometry.pose.pose.position.z = 0.0

odometry.pose.pose.orientation.x = odom_quat[0]
odometry.pose.pose.orientation.y = odom_quat[1]
odometry.pose.pose.orientation.z = odom_quat[2]
odometry.pose.pose.orientation.w = odom_quat[3]

odometry.twist.twist.linear.x = v
odometry.twist.twist.linear.y = 0
odometry.twist.twist.angular.z = w


pub.publish(odometry)
```

Figura 21: Mensaje de posición desde ROS

4 Resultados

En este apartado, se presentan los resultados obtenidos por cada lado del proyecto.

4.1 Resultados Unity

Se obtiene como resultado un entorno donde la silla, con tracción diferencial, es capaz de realizar el recorrido en el modelo recreado del pasillo de física (ver Figura 22). Durante ese recorrido, es capaz de obtener los comandos u órdenes de control provenientes desde el sistema con ROS donde se encuentra el controlador, además de poder enviar al sistema ROS toda la información de posición y velocidad angular de sus ruedas a tiempo real.

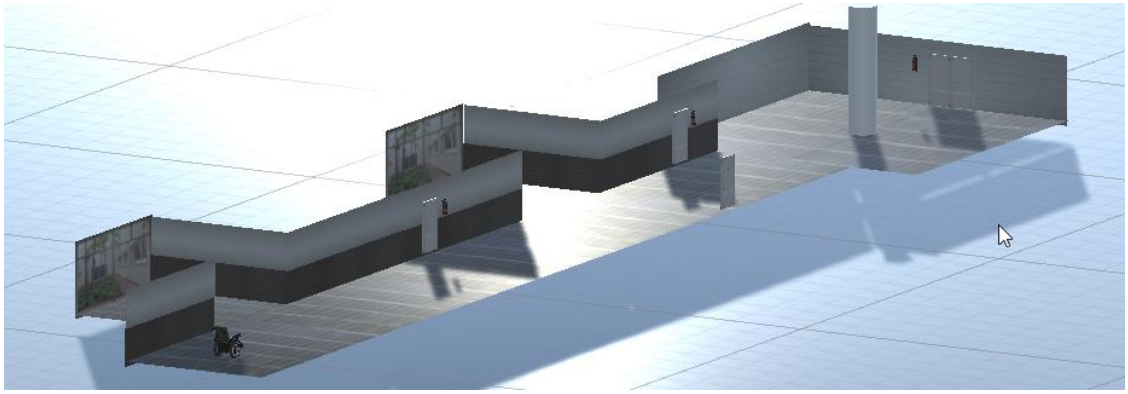


Figura 22: Modelo de pasillo de Facultad de Física

4.2 Resultados ROS

En el sistema ROS, se obtiene como resultado una simulación generada con ***RViz*** que compara la trayectoria real que la silla de ruedas sigue en la simulación de Unity, con la trayectoria estimada por la odometría de las ruedas calculada en ROS.

Esta comparación de ambos resultados, en este caso posiciones, aportará una información capaz de representar cómo evolucionan ambas trayectorias dado que, como se ha mencionado anteriormente, la trayectoria de la Odometría se ve afectada por errores debido al deslizamiento o derrape de las ruedas, lo que genera entradas diferentes.

Como se puede observar en las Figura 23, 24 y 25, se tiene ejemplos de recorridos cortos donde se representan la posición del modelo generado en Unity (Flecha roja) y el modelo simulado a partir de los datos de ROS (Flecha verde). Si se observa detenidamente, la trayectoria de la flecha roja es más precisa y tiene menor cantidad de excesos o correcciones comparada a la flecha verde. Esto se debe a la precisión continua de los mensajes provenientes desde Unity. Por otro lado, se encuentra la flecha verde que, al generar su posición en base a datos sin tener en cuenta entorno o dificultades del terreno, genera una posición similar, pero con ligeras desviaciones. Estas imágenes con recorridos realizados que se pueden distinguir por la diferencia de rozamiento en la superficie que se recorre, estas presentan un valor entre 0 y 1, como ya se ha nombrado anteriormente. Se puede observar como en la Figura 23 se presenta un rozamiento medio, 0.6 en este caso. Este rozamiento permite mantener un estado donde el robot va a poder circular correctamente, pero puede generar desliz en sus ruedas capaces de modificar la trayectoria. Por otro lado, se presenta el caso de un rozamiento de 0 (Figura 24). En este caso, a diferencia del caso anterior, presenta un rozamiento nulo que permite al robot realizar un recorrido sin una gran cantidad de deslizamientos, desvíos que puedan afectar a su recorrido.

Para finalizar los recorridos cortos, se presenta el caso de rozamiento 1 (Figura 25). La aplicación de este rozamiento en la simulación afecta al robot de manera que sea poco capaz de mantener un contacto firme y seguro con la superficie. Este caso hace que el robot sea más propenso a deslizarse. Se puede observar como el recorrido rojo presenta una cantidad grande de variaciones en su recorrido, a diferencia del primer o segundo recorrido.

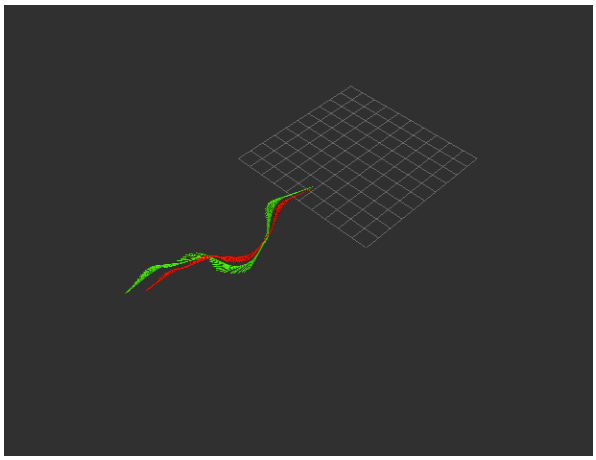


Figura 23: Comparación de posiciones con rozamiento 0.6 con velocidad 1

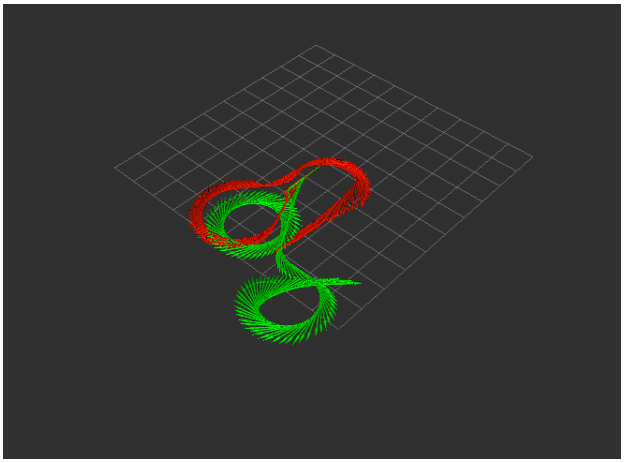


Figura 24: Comparación de posiciones con rozamiento 0 con velocidad 5

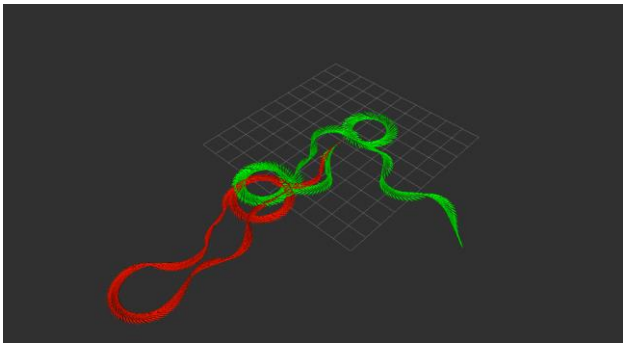


Figura 25: Comparación de posiciones con rozamiento 1 con velocidad 5

Asimismo, si se da uso al entorno objetivo (el pasillo de la facultad de Física), podemos observar unas trayectorias que se van diferenciando cada vez más a medida que pasan los instantes de tiempo y aumenta el recorrido hecho, hasta llegar al punto de diferir completamente a la posición real. Esto se debe a diferencias como la generación de ambas trayectorias, fricciones, deslizamientos, velocidades, comportamientos respecto al terreno o de la propia silla que son ajenos a los datos recibidos por la silla simulada desde ROS (ver Figuras 26 y 27).

En este caso, se presentan varios recorridos realizados esta vez a velocidades diferentes, manteniendo un rozamiento medio donde se puede generar un recorrido del robot correcto y presente múltiples obstáculos como podría pasar en un modelo real. Se puede observar en las Figuras 26 y 27 como el robot es capaz de generar un recorrido similar al original, pero a medida que el tiempo incrementa, la velocidad de giro de las ruedas dada al robot hace que exista una diferencia a la hora de representar el recorrido simulado. Si bien en un primer instante es capaz de seguir fielmente al original, cuando se realiza un evento que requiere un control más preciso, el recorrido simulado en ROS presenta un recorrido que difiere al modelo original ya que obtiene unas instrucciones donde no está presente los obstáculos que acarrea el modelo original. Además, las velocidades dadas aumentan la diferencia entre recorridos al necesitar mas tiempo para realizar las acciones.

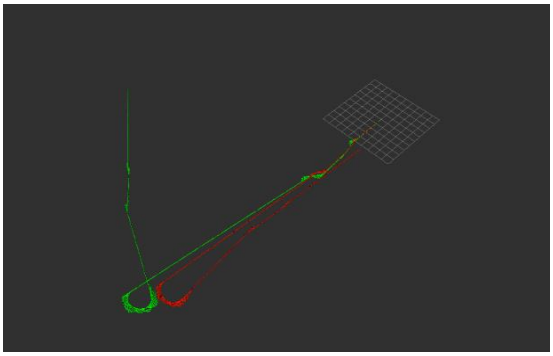


Figura 26: Comparación de posiciones con rozamiento 0.6 con velocidad 5

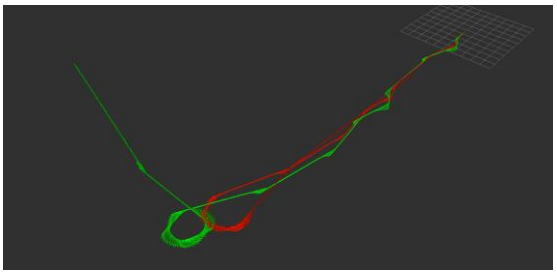


Figura 27: Comparación de posiciones con rozamiento 0.6 con velocidad 3

Finalmente, se puede observar en la Figura 28 un ejemplo dado por un modelo real. Este modelo presenta un recorrido similar a los anteriormente mencionados (Figura 26 y 27). Asimismo, podemos ver como en el inicio presenta ya una diferencia en la trayectoria del robot al contrario que el simulado desde Unity. En cambio, es presenta una diferencia en la trayectoria menor que realiza durante el giro, que se trata de la columna del pasillo de Físicas. Aun así, sigue acarreando problemas similares al simulado al representar el modelo sin obstáculos, velocidades de giro que no reflejan el movimiento actual o diferencias en el terreno.

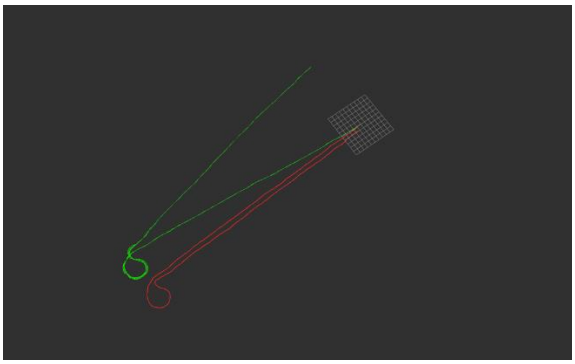


Figura 28: Resultado de ejemplo real montado actualmente

5 Conclusiones y líneas futuras

En este proyecto se ha conseguido simular el comportamiento de la localización de una silla de ruedas atendiendo a las posibles características del entorno, por ello, tras la elaboración de este proyecto, se puede observar como la simulación generada por Unity es viable, ya que nos permite generar un modelo a nuestro placer para poder realizar pruebas en un entorno seguro y fiable. Además, presenta un contexto que facilita y ayuda a la generación de un mecanismo fácilmente exportable para poder dar uso en un modelo real y prever posibles fallos o diferencias que puedan suceder en un modelo real.

Por ello, es interesante crear un entorno capaz de simular contextos o situaciones donde se desarrolle la acción en formato digital para poder tenerlo como referencia y, cuando sea necesario, diseñar algoritmos de localización que puedan corregirlos antes de implementarse en el prototipo físico o rediseñar aspectos físicos propios del robot.

Durante este proyecto, solo se tuvo en cuenta un modelo plano (el pasillo de Física), con lo que queda también abierto para un futuro trabajo la realización de este tipo de modelos en diferentes terrenos para observar cómo se puede llegar a comportar o robots para ver sus comportamientos a tiempo real, poder simular los resultados de posición estimados por otros sensores como puede ser LIDAR o la IMU o directamente buscar una solución capaz de buscar un realismo a la hora de representar la posición de la silla, consiguiendo resultados muchos más fiables.

6 Summary and Conclusions

It has been possible to simulate the behavior of the location of a wheelchair considering the possible characteristics of the environment, therefore, after the elaboration of this project, it can be observed how the simulation generated by Unity is viable, since it allows us to generate a model at our pleasure to be able to carry out tests in a safe and reliable environment. In addition, it presents a context that facilitates and helps the generation of an easily exportable mechanism to be used in a real model and anticipate possible failures or differences that may occur in a real model.

Therefore, it is interesting to create an environment capable of simulating multiple contexts or situations to have as an early reference and, when it is necessary, design localization algorithms that can correct them before being implemented in the physical prototype or redesign physical aspects of the robot.

Likewise, during this project, only one model was taken into account (the Physics building corridor), which also leaves open for future work the realization of this type of models in different terrains to observe how they can behave or robots to see their behavior in real time, to be able to simulate the position results estimated by other sensors such as LIDAR or the IMU or to directly search for a solution capable of seeking realism when representing the position of the chair, achieving many results more reliable.

7 Presupuesto

Con este capítulo, se busca establecer un presupuesto global del desarrollo del proyecto.

7.1 Personal

Para los siguientes cálculos, se toma el valor de 15€ la hora para la jornada de un ingeniero informático mostrados en la Tabla 1.

Tareas	Horas	Coste €
Montaje de sistema Unity	40	600
Preparación de la simulación	30	450
Generación de scripts Unity	120	1800
Montaje de sistema ROS	15	225
Generación de scripts ROS	80	1200
Conectividad	20	300
Pruebas y depuración de errores	160	2400
Redacción de la memoria	50	750
Total	**515**	**7725**

Tabla 1: Tabla de tiempos

7.2 Materiales

Los cálculos mostrados en la Tabla 2 establecen todo el material necesario para el desarrollo del proyecto.

Componentes	Coste €
Sensores	200
Raspberry	80
PC	2000
Total	**2280**

Tabla 2: Tabla de Materiales

7.3 Costes totales

En la Tabla 3, se plantea el coste total de realizar este proyecto.

Tareas	**Coste €**
Costes: Personales	80
Costes: Componentes	2000
Total	**2580**

Tabla 3: Tabla de Total

Bibliografía

[1] Mexicana De Mecatrónica, A., Arreguín, P., Israel, J., Saúl, A., Giovanni, U., Carrillo, V., Efrén, H., Ortega, P., Carlos, J., Soto, V., Emilio, J., Arreguín, R., Manuel, J., & Sotomayor, O. (n.d.). Mecamex.net. Retrieved July 17, 2022, from http://mecamex.net/anterior/cong10/trabajos/art18.pdf

[2] Asensio, I. (n.d.). *Qué es Unity y para qué sirve*. MasterD. Retrieved September 4, 2022, from https://www.masterd.es/blog/que-es-unity-3d-tutorial

[3] Documentation - ROS Wiki. (n.d.). Ros.org. Retrieved July 17, 2022, from http://wiki.ros.org/Documentation

[4] (N.d.). Upm.Es. Retrieved September 4, 2022, from https://oa.upm.es/43395/1/TFC_GUAMAN_COCHA_CHRISTIAN.pdf

[5] (N.d.-b). Idus.Us.Es. Retrieved September 4, 2022, from https://idus.us.es/bitstream/handle/11441/68456/TFG_Felipe%20P%c3%a9rez%20Molina.pdf?sequence=1&isAllowed=y

[6] (N.d.-b). Amazon.com. Retrieved September 4, 2022, from https://aws.amazon.com/es/docker/

[7] Unity Technologies. (n.d.). *Unity user manual 2021.3 (LTS)*. Unity3d.com. Retrieved July 17, 2022, from http://docs.unity3d.com/Manual/index.html

[8] *Unity-Robotics-Hub: Central repository for tools, tutorials, resources, and documentation for robotics simulation in Unity*. (n.d.).

[9] melodic - ROS Wiki. (n.d.). Ros.org. Retrieved July 17, 2022, from http://wiki.ros.org/melodic

[10] Teleop_twist_keyboard: Generic Keyboard Teleop for ROS. (n.d.).