UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Grado en Informática Industrial y Robótica

Asignatura: ROM

"Robot de reparto autónomo en CoppeliaSim con ROS 2: seguimiento de línea y evasión de obstáculos"

Autores del proyecto:

ALBERTO ANDRÉS GÓMEZ

DIEGO RUIZ ROCA

ANDRÉS TOLEDO CERVERA

SALVADOR ISASI

Profesor responsable:

LEOPOLDO ARMESTO ÁNGEL

Índice

Tabla de Figuras

Descripción general del proyecto

Este proyecto consiste en el desarrollo de una simulación en el entorno CoppeliaSim utilizando ROS2, donde un robot móvil TurtleBot3 Burger realiza la tarea de entrega autónoma de paquetes siguiendo un recorrido predefinido. El robot dispone de sensores específicos, como un sensor infrarrojo para seguir una línea negra trazada en el suelo y un sensor LIDAR para la detección de obstáculos dinámicos, como peatones, permitiendo al robot detenerse cuando detecta una posible colisión. Adicionalmente, se implementa un sistema de semáforo interactivo para controlar el paso de peatones simulados, consiguiendo así un entorno de simulación dinámico, realista y seguro. Todo ello se realiza mediante programación propia de nodos ROS2 y modificación sustancial del entorno simulado.

Propósito del trabajo

El objetivo principal de este trabajo es el de implementar un robot seguidor de líneas completamente autónomo capaz de evitar colisiones con los peatones del entorno y capaz de poder entregar el paquete al cliente sin daños en el mismo y sin salirse de la ruta.

Requisitos funcionales

A continuación, se describen los requisitos funcionales que la simulación debe cumplir para alcanzar su objetivo:

1. **Seguimiento de línea**
   El robot debe ser capaz de seguir de forma continua una línea negra trazada en el suelo mediante el uso de un sensor infrarrojo.

2. **Entrega de paquetes**
   El robot debe transportar un paquete desde un punto de inicio hasta un destino específico, identificado por la proximidad a un marcador (dummy) que representa al destinatario. Al llegar al destino, el sistema debe mostrar un mensaje confirmando la entrega.

3. **Detección de obstáculos**
   Mediante un sensor LIDAR, el robot debe identificar objetos cercanos en su trayectoria y detenerse automáticamente si se aproxima a un obstáculo dentro de un umbral de seguridad.

4. **Interacción con peatones**
   El sistema incluye un modelo de peatón animado (Bill) que cruza la trayectoria del robot. El robot debe detenerse si detecta al peatón en el paso de cebra, reanudando su marcha cuando este se haya retirado.

5. **Sistema interactivo con semáforos**
   Se implementa un nodo semáforo que regula el movimiento del peatón. El peatón solo se mueve cuando el semáforo está en verde, al estar en rojo los peatones desaparecen para evitar ruido con el sensor LIDAR del robot. Este estado puede ser modificado manualmente durante la simulación.

6. **Notificación de fin de recorrido**
   Al completar el trayecto y entregar el paquete, el sistema debe emitir un mensaje indicando que el recorrido ha finalizado con éxito, permitiendo identificar claramente la finalización de la tarea y una vez mostrado el mensaje, la simulación debe finalizar.

Estos requisitos permiten la evaluación del comportamiento autónomo del robot en un entorno simulado, combinando percepción, navegación y control en un flujo funcional integrado.

Descripción del escenario de simulación

Para el desarrollo y validación del proyecto se ha diseñado un entorno de simulación en CoppeliaSim que recrea una escena urbana simplificada. Este entorno permite poner a prueba el comportamiento autónomo del robot en condiciones similares a un escenario real, incluyendo elementos como caminos con líneas de guía, pasos de peatones, obstáculos, y zonas de entrega. La escena está adaptada específicamente para comprobar las distintas funcionalidades del sistema, garantizando una evaluación completa de los sensores, algoritmos de navegación y lógica de interacción con el entorno. A continuación, se detalla la estructura general de la escena y componentes principales de esta simulación.

Vista general de la escena

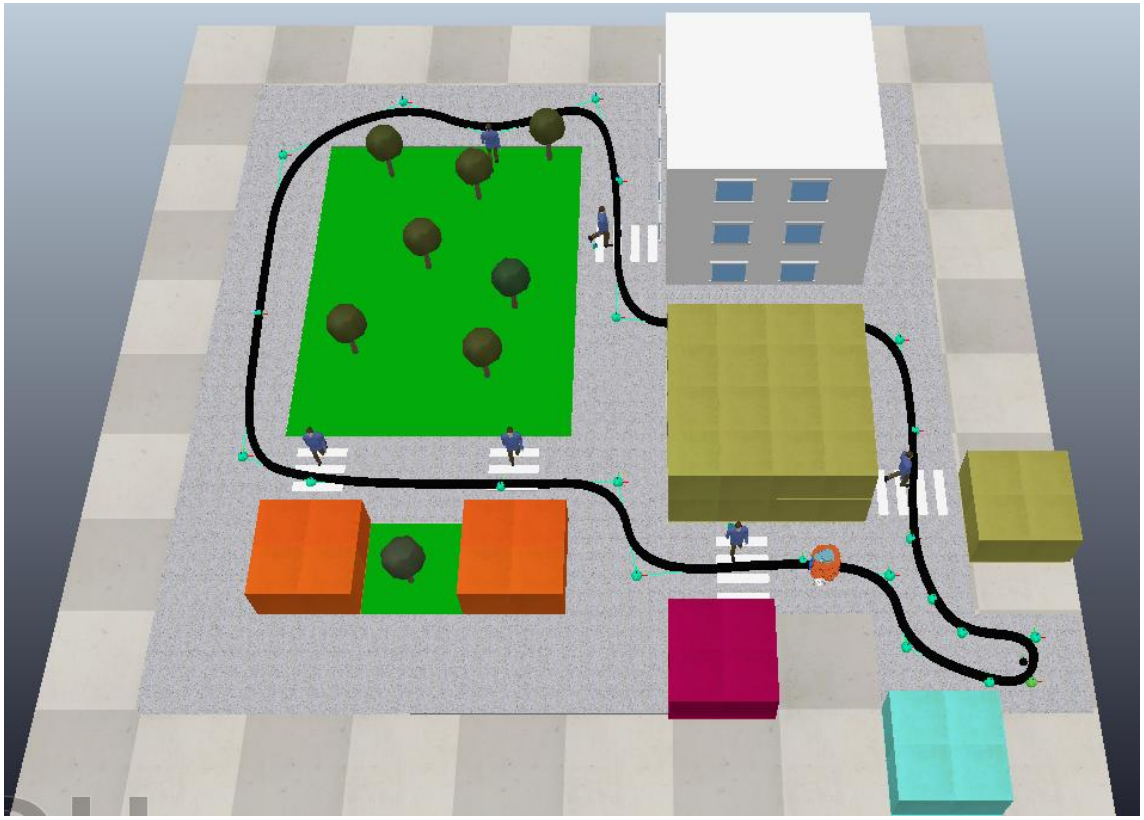

Figura 1: Vista general de la simulación.

La vista principal del entorno de simulación representa una pequeña ciudad organizada para probar el comportamiento autónomo del robot de reparto. Se puede distinguir entre los **objetos de decoración**, que ambientan la escena sin influir directamente en la lógica de navegación, y los **objetos principales**, que sí interactúan directamente con el sistema del robot.

Objetos de decoración

- **Árboles**
  Distribuidos principalmente en la zona verde del parque, aportan variedad visual y ayudan a delimitar áreas no transitables sin representar un obstáculo físico.

Figura 2: Modelo de árbol de coppeliaSim.

Para el diseño de los árboles hemos utilizado el modelo 3D que nos proporciona Coppelia en la carpeta **nature** del model browser.

- **Pasos de cebra**
  Ubicados a lo largo del mapa donde el robot debe ceder el paso a peatones simulados, tienen valor visual y funcional como referencia de zonas de cruce.

Figura 3: Modelo de paso de cebra.

Para la implementación del paso de cebra, como CoppeliaSim no tiene un modelo 3D predefinido, hemos optado por el uso de paredes de 20 cm de ancho que incorpora el simulador que al colocarlas de manera vertical y modificando la coordenada Z de estas obtenemos de forma visual y funcional un paso de cebra. A parte, también hemos tenido que desactivar las propiedades dinámicas para anular el efecto de la gravedad y las físicas (que sea colisionable, medible y detectable) para que no interfieran con el flujo de la simulación.

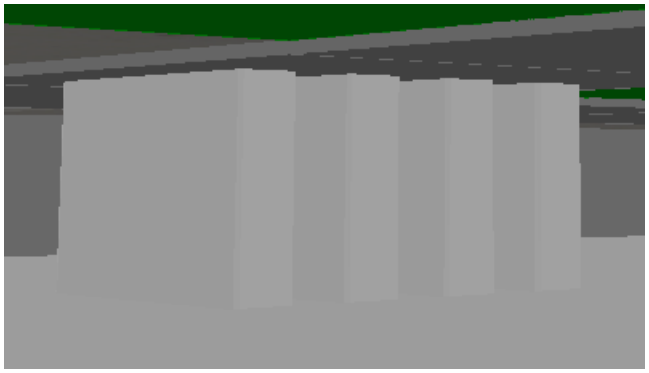

Figura 4: Diseño del paso de cebra.

- **Casas**

Representadas por conjuntos de **cuboids** (figuras primitivas que nos ofrece el simulador) de distintos colores (naranja, rosa, cian…), aportan un entorno urbano reconocible, simulando edificios residenciales o pequeños comercios.

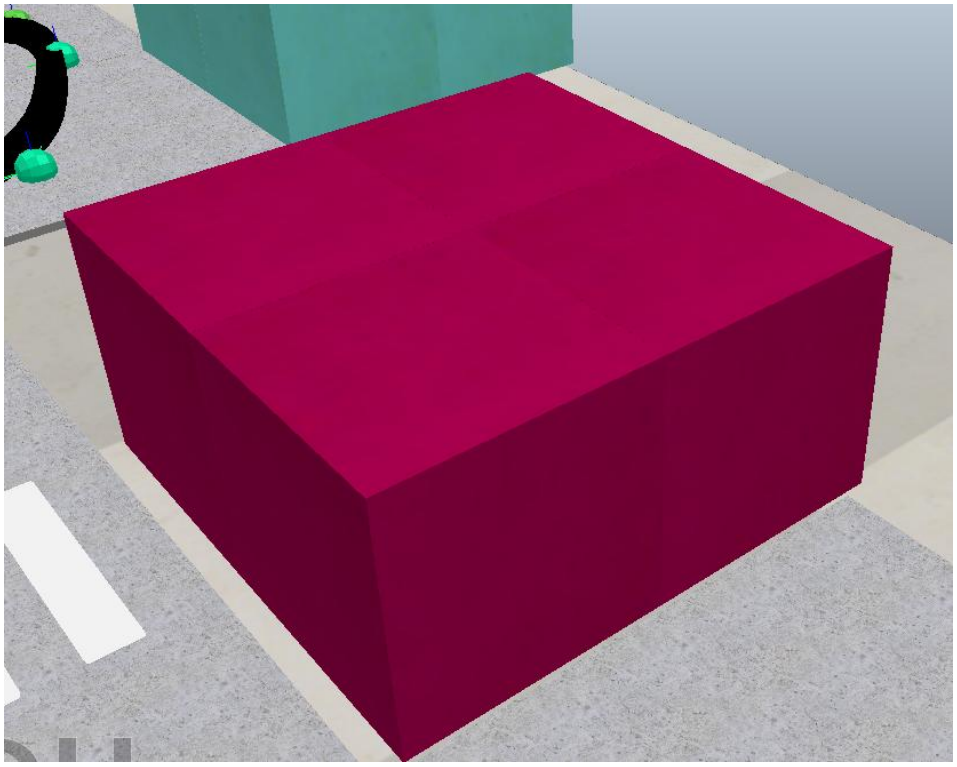

Figura 5: Vista del modelo casa.

- **Edificio principal**
  Edificio de mayor tamaño y color blanco con el detalle de incluir ventanas, situado en la parte superior derecha, simula un bloque de oficinas o sede institucional, dando contexto al entorno urbano.

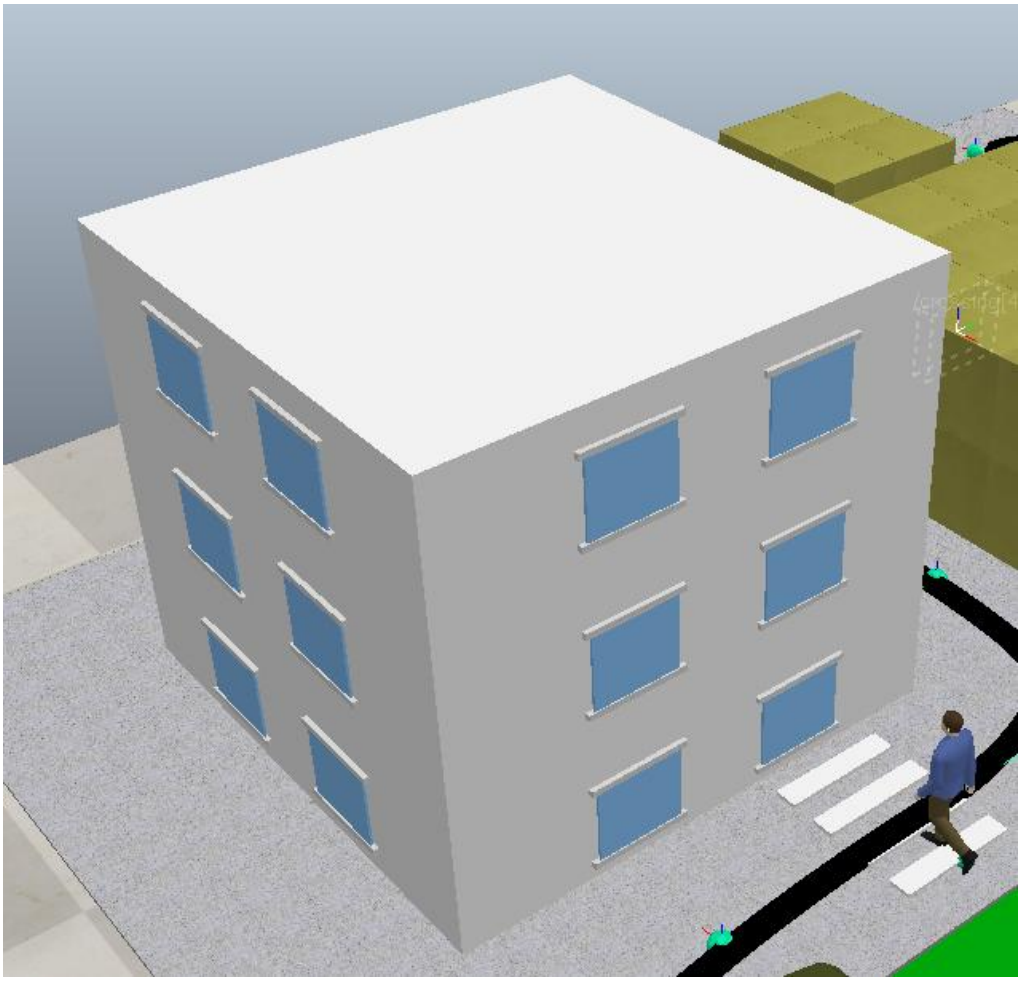

Figura 6: Vista del edificio.

Para la implementación del edificio hemos utilizado el modelo de Coppelia **Customizable building** que podemos encontrar en **/infrastructure/urban** en el model browser.

- **Suelo**
  Dividido en zonas de **cemento gris**, que representa las aceras y vías, y zonas de **césped verde**, que definen parques y zonas recreativas, delimitando visualmente los caminos transitables.

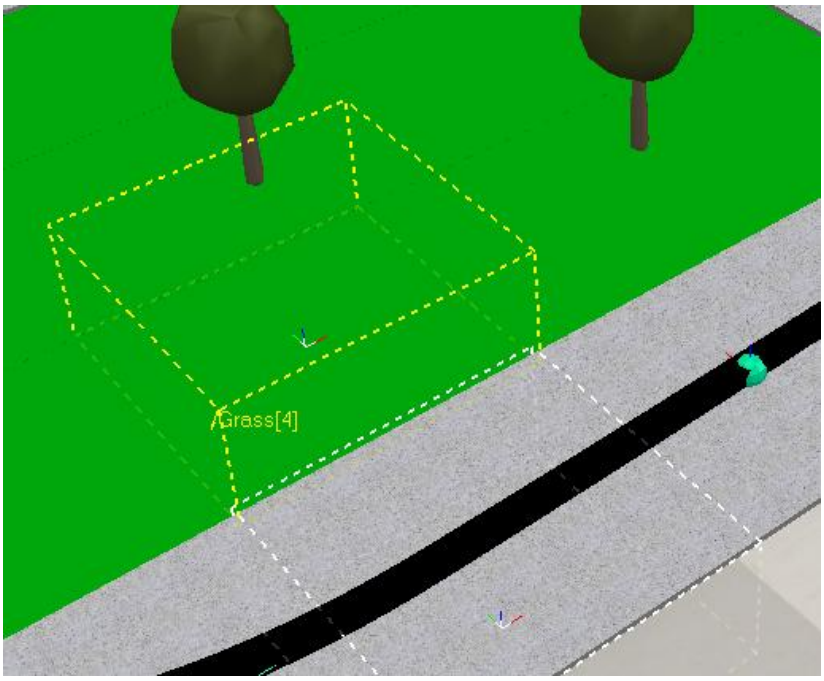

Figura 7: Imagen de un bloque cemento y un bloque césped.

Para el diseño del suelo hemos utilizado el modelo 3D de Coppelia **5m x 5m concrete floor** localizado en **infrastructure/floors**.

Para el césped simplemente hemos cambiado el color del objeto a verde.

Estos elementos no solo aportan estética, sino que también ayudan a guiar al observador en la interpretación de la escena y delimitan claramente las áreas por las que el robot puede o no circular. En el siguiente apartado se detallarán los objetos principales que permiten la ejecución de la tarea.

Objetos principales

1. **Robot**

   Es un modelo **TurtleBot3 Burger**, equipado con sensores como el seguidor de línea (sensor infrarrojo) y un LIDAR 2D. Es el encargado de seguir el camino definido por una línea negra, detenerse ante obstáculos y entregar el paquete a su destino. Su comportamiento está controlado mediante nodos ROS 2 personalizados que coordinan su movimiento, percepción y toma de decisiones.

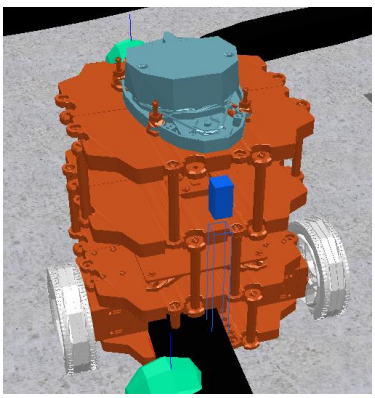

Figura 8: Imagen del modelo 3D TurtleBot3 Burger simplificado.

El modelo 3D TurtleBot3 Burger lo hemos importado de la carpeta **turtlebot3_coppeliasim** de las prácticas de la asignatura, por lo que no hemos tenido que importar los diferentes componentes del robot y luego unirlos.

2. **Camino (Path)**

Está representado por una línea negra continua que recorre toda la ciudad, cruzando zonas peatonales, bordeando edificios y atravesando parques. Es la guía visual que el robot sigue usando su sensor infrarrojo, actuando como su única referencia para navegación básica.

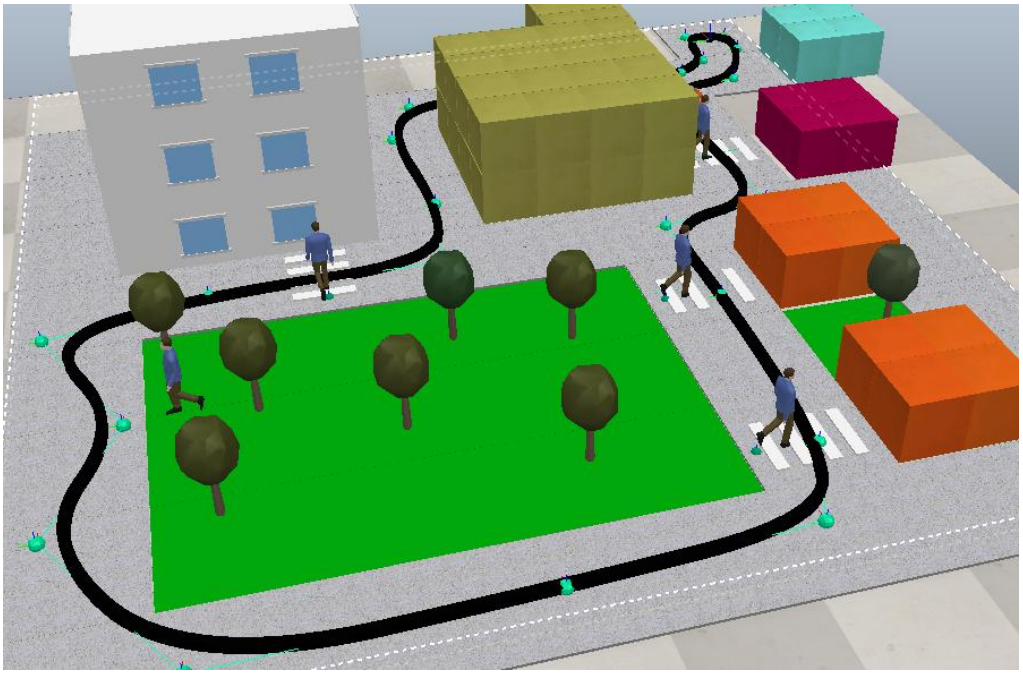

Figura 9: Vista general del path.

Para la implementación del path, hemos añadido un path de tipo cerrado del apartado "add" en la barra del menú principal.

En las propiedades del path hemos incrementado las subdivisiones del path de 100 a 300 para que el camino se vea más fluido, no tanto como líneas rectas juntas. También se ha marcado la opción de **generate extruded shape** para mostrar el camino más amplio y hacer la tarea del seguidor de líneas más fácil para el robot.

3. **Walking Bill**

Es un personaje animado incorporado en Coppelia que simula un peatón caminando constantemente de un lado al otro en los pasos de cebra. El modelo lo podemos encontrar en la carpeta **people/Walking Bill** en el model browser.

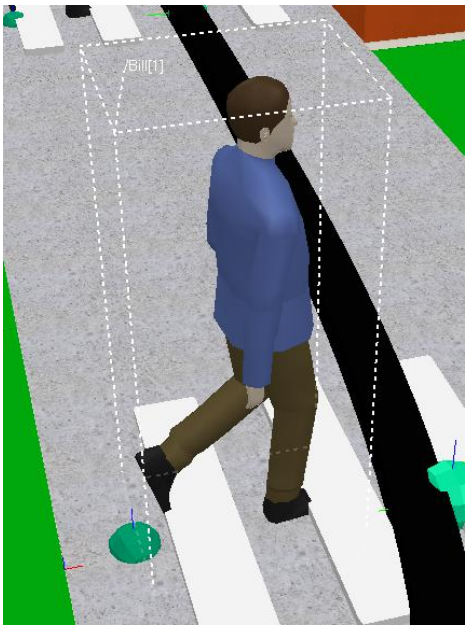

Figura 10: Modelo 3D – Walking Bill.

Este modelo de Coppelia incorpora un script para el movimiento del muñeco, el cual, ha tenido que ser modificado para que Bill se mueva en un path de manera constante de un lado a otro.

Aquí os dejamos la función modificada:

```
function sysCall_thread()
    sim.setStepping(true)

    for _ = 1, maxJourneys do
        ──────────────── IDA ────────────────
        local dist = 0
        local st   = sim.getSimulationTime()

        while dist < pathL do
            ──────────────────────────────────────────────────────────
            -- Semáforo: si está rojo (canMove == false) Bill se queda quieto
            ──────────────────────────────────────────────────────────
            if not canMove then
                sim.step()                          -- espera un ciclo
                st = sim.getSimulationTime()        -- re-sincroniza el reloj
                goto continue_ida                   -- salta al próximo bucle
            end

            local dt = sim.getSimulationTime() - st
            dist     = velocity * dt

            local p  = sim.getPathInterpolatedConfig(pathPositions, pathLengths, dist)
            local pp = sim.getObjectPosition(BillHandle)
            local dx, dy = p[1]-pp[1], p[2]-pp[2]

            sim.setObjectOrientation(BillHandle, {0, 0, math.atan2(dy, dx)})
            sim.setObjectPosition(BillHandle, p)
            moveBody(dist)
            sim.step()

            ::continue_ida::
        end
```

Figura 11: Script Lua de Walking Bill para el movimiento constante de un punto a otro (viaje ida).

```
        ---------- VUELTA ----------
        dist = pathL
        st   = sim.getSimulationTime()

        while dist > 0 do
            if not canMove then
                sim.step()
                st = sim.getSimulationTime()
                goto continue_vuelta
            end

            local dt = sim.getSimulationTime() - st
            dist     = pathL - velocity * dt

            local p  = sim.getPathInterpolatedConfig(pathPositions, pathLengths, dist)
            local pp = sim.getObjectPosition(BillHandle)
            local dx, dy = p[1]-pp[1], p[2]-pp[2]

            sim.setObjectOrientation(BillHandle, {0, 0, math.atan2(dy, dx)})
            sim.setObjectPosition(BillHandle, p)
            moveBody(pathL - dist)
            sim.step()

            ::continue_vuelta::
        end

        moveBody()          -- postura neutral al terminar viaje
    end
end
```

Figura 12: Script Lua de Walking Bill para el movimiento constante de un punto a otro (viaje vuelta).

4. **Bill Standing (receptor)**: es un personaje estático que representa al destinatario del paquete. Se encuentra en una posición concreta del mapa esperando que el robot llegue con el paquete. Su función principal es marcar el punto de entrega.
   Este modelo no tiene ninguna función interactiva, simplemente visual, para facilitar el entendimiento del programa al observador simulando que se trata de un cliente que está esperando un paquete.

Figura 13: Modelo Bill Standing.

5. **Paquete**

   Es un objeto cúbico primitivo de Coppelia que representa físicamente el elemento a entregar. Está inicialmente cargado sobre el robot y, tras completarse la entrega, desaparece de la escena simulando que ha sido recibido correctamente por Bill Standing. También sirve como referencia visual del objetivo de la misión del robot.

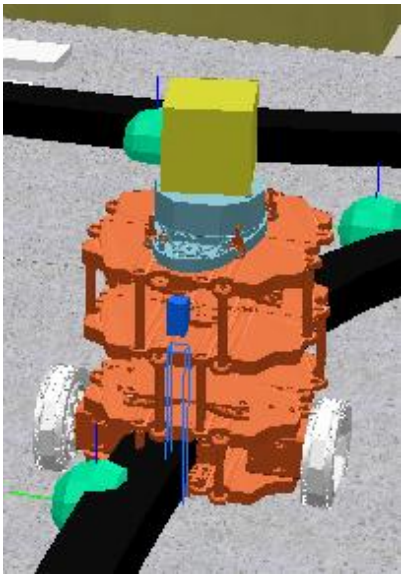

Figura 14: Imagen del robot transportando el paquete.

Estos elementos componen el núcleo funcional del entorno de simulación y son fundamentales para recrear una escena de reparto autónomo realista y controlada.

Sensores incorporados

En este apartado se detallan los sensores utilizados por el robot para percibir y reaccionar ante su entorno durante la simulación. Estos sensores permiten al robot seguir el recorrido predefinido, detectar obstáculos y actuar en consecuencia para cumplir su misión de entrega. Su correcta integración y configuración en ROS 2 ha sido clave para garantizar un comportamiento autónomo fiable y coherente con los objetivos del proyecto.

Sensor LIDAR

El sensor LIDAR (Light Detection and Ranging) juega un papel fundamental en la capacidad del robot para detectar obstáculos a su alrededor. Este sensor realiza un escaneo en 360 grados, proporcionando una nube de puntos con distancias a los objetos circundantes. Este sensor no ha tenido que ser implementado por nosotros ya que venía incorporado en el modelo 3D del robot importado.

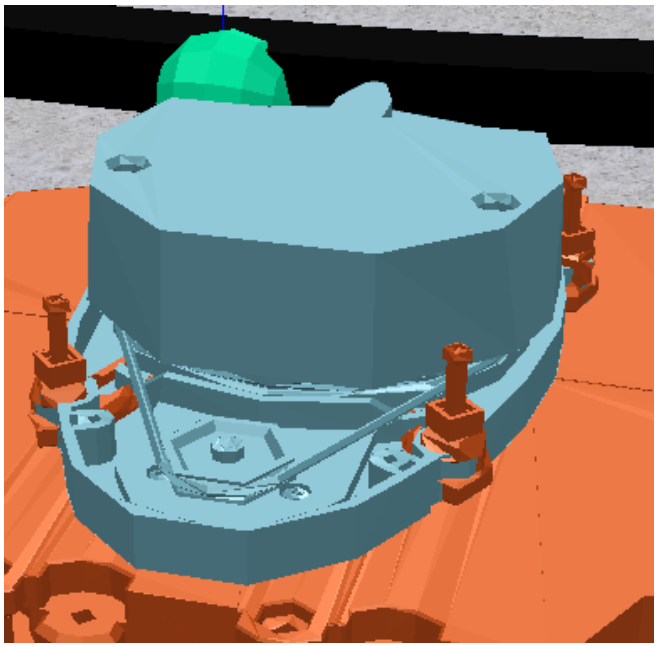

Figura 15: Imagen del sensor LIDAR incorporado en el robot.

Este sensor está montado en la parte superior del robot, como se puede observar en la imagen anterior, y se encarga de realizar un escaneo 2D del entorno mediante los tres emisores láser que incorpora de una distancia máxima de 8 metros en la simulación. Gracias a este escaneo, el robot puede detectar obstáculos a su alrededor dentro del rango máximo de escaneo, facilitando así la navegación segura dentro del entorno simulado.

Laser Script

El script principal encargado de la medición del escenario es el **Laser Script** que ya venía incorporado en el sensor. Este script se encarga de medir las distancias a los objetos detectados por el LIDAR en el rango predefinido y enviarlos mediante ROS 2 para su posterior utilización en las lógicas del programa.

```
if #measuredData>0 then
--print(#measuredData)
    local msg={
        header={
            stamp=simROS2.getTime(),
            frame_id='base_scan',
        },
        angle_min=0.5*math.pi/180,
        angle_max=359.5*math.pi/180,
        angle_increment=math.pi/180,
        time_increment=0,
        scan_time=sim.getSimulationTimeStep(),
        range_min=0.001,
        range_max=maxScanDistance,
        ranges=measuredData,
        intensities={},
    }
    simROS2.publish(pub,msg)
end
```

Figura 16: Envío de datos.

Este fragmento de código se encarga de publicar los datos del sensor LIDAR en un tópico de ROS 2. El proceso comienza comprobando que existan datos medidos válidos. A continuación, se construye un mensaje del tipo **LaserScan**, el cual contiene toda la información necesaria para representar un escaneo completo del entorno. Este mensaje incluye el sello temporal, el marco de referencia del sensor, el rango angular cubierto, la resolución entre muestras, el tiempo de escaneo, las distancias mínimas y máximas detectables, así como el conjunto de distancias medidas.

Una vez estructurado el mensaje, se publica mediante ROS 2 utilizando la función **simROS2.publish**, lo que permite que otros nodos en el sistema reciban esta información y puedan tomar decisiones en base a los obstáculos detectados. De esta forma, el sensor LIDAR proporciona en tiempo real un mapa 2D del entorno que puede ser procesado para tareas como evitar colisiones o detener el robot en situaciones de peligro.

Cabe recalcar que los datos son enviados al tópico **/scan** creado en la función **sysCall_init()** de este mismo con la siguiente instrucción:

```
pub=simROS2.createPublisher('/scan','sensor_msgs/msg/LaserScan',0,false,
    {reliability=simROS2.qos_reliability_policy.reliable,durability=simROS2.qos_durability_policy.volatile,
        history=simROS2.qos_history_policy.keep_last,liveliness=simROS2.qos_liveliness_policy.automatic,
        deadline={sec=3,nanosec=0},lifespan={sec=0,nanosec=500000000},depth=10,
        liveliness_lease_duration={sec=3,nanosec=0},avoid_ros_namespace_conventions=false})
```

Figura 17: Creación del tópico "scan".

Funcionamiento visual del LIDAR en el entorno de simulación

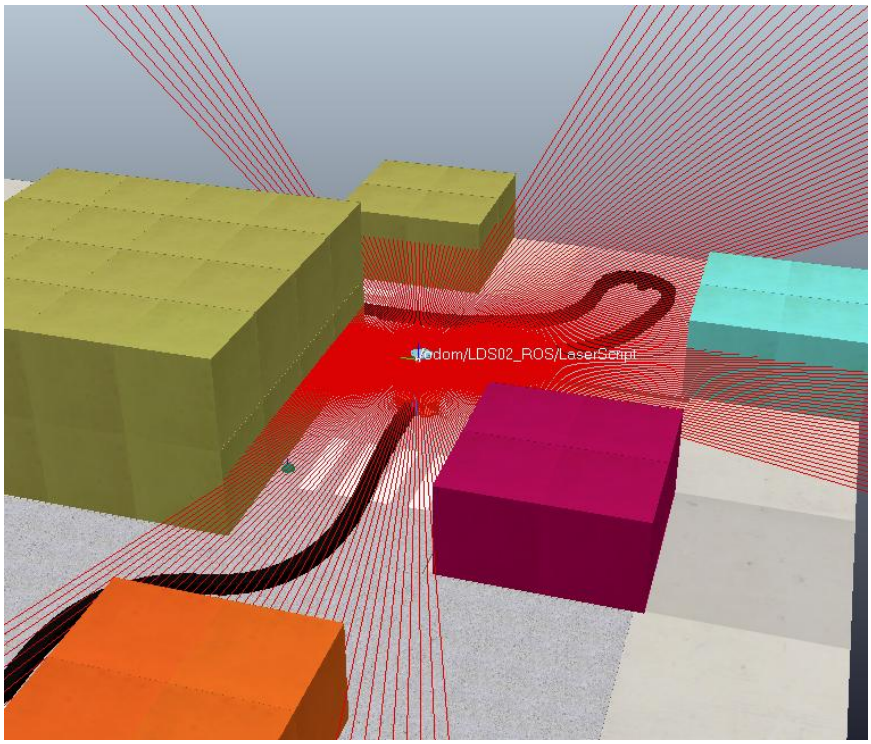

Figura 18: Imagen funcionamiento visual del LIDAR.

En esta imagen se puede observar el funcionamiento del LIDAR, el cual hace un escaneo de 360º del entorno con el uso de los láseres pudiendo identificar los objetos de la escena.

Detección y prevención de colisiones

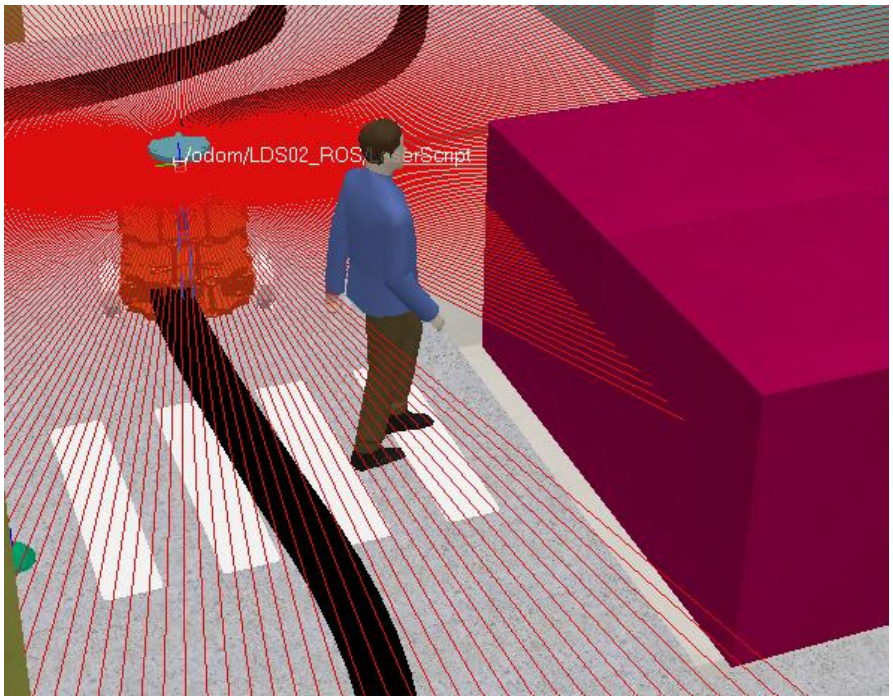

Figura 19: Visualización de la detección del peatón por el sensor LIDAR.

Para la detección de objetos y la prevención de colisiones el sensor publica los datos de las distancias respectivas de los distintos objetos detectados en el escaneo en el tópico ROS2 **/scan**. Estos datos son analizados por un nodo que evalúa la mínima distancia detectada y determina si el robot debe detenerse para evitar una colisión. En la imagen se puede ver claramente cómo los rayos del LIDAR alcanzan tanto al muñeco que cruza como a los edificios cercanos, permitiendo al sistema anticiparse a obstáculos imprevistos.

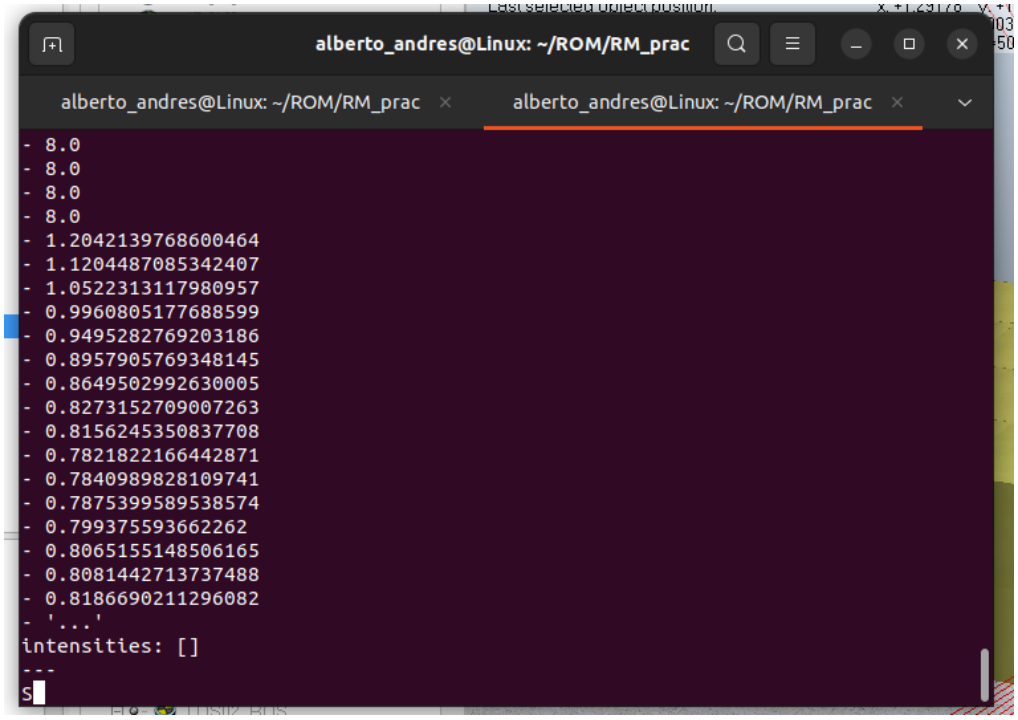

Figura 20: Datos publicados en /scan

En esta imagen podemos ver las distancias de los objetos que detecta el sensor LIDAR. Para poder ver esta información en una terminal nueva introducciones el siguiente comando: **ros2 topic echo /scan**.

Con esta información, luego un nodo ros2 que está suscrito a este tópico, se encarga de la lógica de las colisiones y de las posteriores órdenes de parar o mover al robot, pero esto lo veremos más detalladamente en el apartado de los nodos ROS 2.

Sensor TCRT5000 (Infrarrojo)

El sensor TCRT5000 es un componente esencial en la lógica de seguimiento de línea del robot. Se trata de un sensor óptico infrarrojo que detecta la presencia o ausencia de una línea negra sobre una superficie clara, lo que permite al robot orientarse y desplazarse a lo largo de un recorrido predefinido. Su funcionamiento se basa en la reflexión de luz: las superficies blancas reflejan la luz infrarroja emitida por el sensor, mientras que las líneas negras la absorben, generando así un cambio en la señal de salida.

En el proyecto, el sensor no venía integrado en el robot, por lo que ha tenido que ser implementado. Para ello hemos hecho uso de un sensor de visión de tipo ortogonal que podemos encontrar en el apartado **add->vision sensor->orthogonal type** de Coppelia el cual se ha colocado en la parte delantera central del robot.

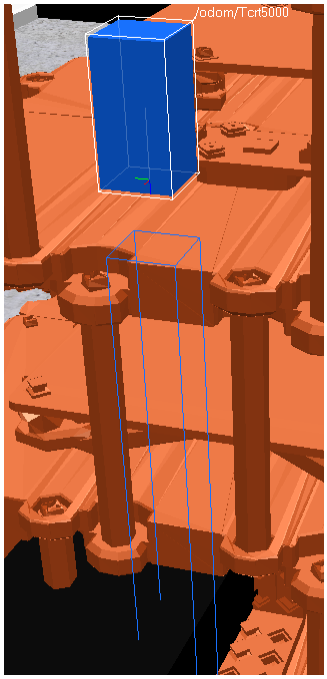

Figura 21: Vista del sensor TCRT5000.

En las propiedades del sensor hemos modificado la resolución de la cámara a 1x1 px para que solo detecte la presencia o ausencia de la línea bajo el sensor, simulando el comportamiento de un sensor digital que no necesita interpretar imágenes complejas, sino únicamente distinguir entre blancos y negros para seguir el camino trazado.

Además, se ha ajustado el rango de detección de 0.01 m a 0.15 m para que no detecte elementos del entorno que estén alejados o fuera de la línea, garantizando así una detección precisa únicamente del trazado negro sobre el suelo.

En esta imagen podemos ver donde se han hecho los cambios en las propiedades del sensor.

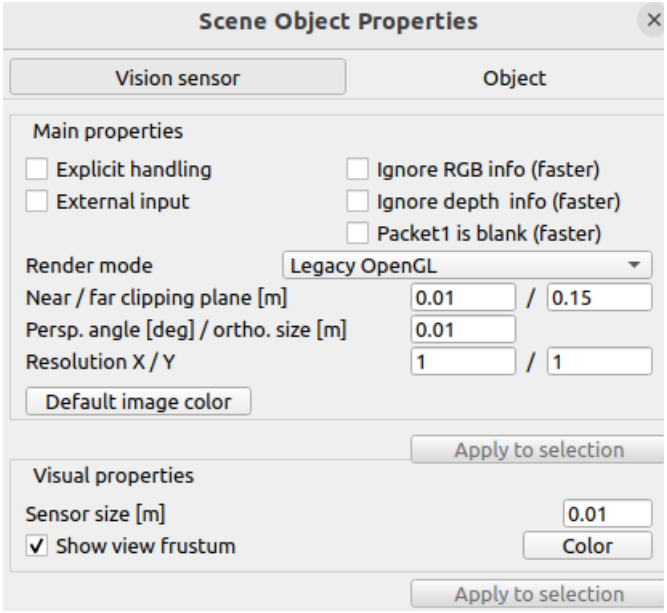

Figura 22: Propiedades del sensor TCRT5000.

Script

La lógica del sensor infrarrojo consiste en obtener el valor de la luminancia de una imagen dada. Aquí os dejamos el código para detectar la luminancia (cuanto más cerca esté el valor de 0 más oscura es la imagen y viceversa) de la imagen.

```
function getLum()
    local img = sim.getVisionSensorImg(sensor, 1)
    if img then
        -- img es un string; 1er byte = luminancia
        local lumByte = string.byte(img, 1)         -- => entero 0..255
        local lum = lumByte / 255.0                 -- normaliza 0..1
        return lum
    else
        return 1.0
    end
end

function sysCall_sensing()
    local value = getLum()
    simROS2.publish(irPub, {data = value})
end
```

Figura 23: Función getLum().

Si el sensor ha detectado una imagen, la función almacena el valor de la luminancia de la imagen y la normaliza para tener valores entre 0 y 1 ya que, al tratarse de un valor entero el rango de valores es de 0 a 255 lo cual dificulta saber si la imagen detecta colores claros u oscuros.

Una vez analizada la imagen y tener el valor de la luminancia, lo publicamos en el tópico anteriormente creado en el mismo script **/ir_line**, al cual le pasamos un valor de tipo Float con el valor de la luminancia de la imagen.

Luego mediante un nodo de ROS2, con la luminancia, programamos la lógica del seguimiento de líneas que explicaremos en el siguiente apartado.

Arquitectura ROS 2

ROS2 (Robot Operating System 2) es un conjunto de herramientas, bibliotecas y convenciones diseñado para facilitar el desarrollo de software para robots. Se trata de una plataforma modular y distribuida que permite gestionar de forma eficiente la comunicación entre los distintos componentes de un sistema robótico, como sensores, actuadores y algoritmos de control. En esencia, ROS2 actúa como el sistema operativo del robot, proporcionando una infraestructura para que los diferentes módulos puedan intercambiar datos de forma segura y en tiempo real.

En este proyecto, ROS2 ha sido la base sobre la cual se ha construido toda la lógica del comportamiento del robot. Gracias a sus mecanismos de publicación/suscripción (topics), servicios y parámetros, se ha podido estructurar el sistema de forma flexible, escalable y mantenible. Cada función del robot —como el seguimiento de líneas, la detección de obstáculos o la gestión del cruce peatonal— se implementa como un nodo independiente que se comunica con el resto mediante mensajes, lo que permite un alto grado de modularidad y reutilización del código.

Descripción de los nodos y su funcionamiento

El proyecto implementa 3 nodos ROS2 principales, 2 trabajan de forma conjunta para el movimiento del robot sobre la línea y la detección de obstáculos, el tercer nodo implementado es independiente y se encarga de activar o desactivar a los peatones como una especie de semáforo manual. A continuación, se detalla la lógica interna de cada uno:

Ir_line_follower_node

El nodo **ir_line_follower_node** es una de las piezas fundamentales de la arquitectura del proyecto, ya que se encarga de controlar el comportamiento de seguimiento de la línea negra por parte del robot. Su funcionamiento se basa en la lectura de la luminancia obtenida por el sensor TCRT5000.

El nodo recibe las mediciones del sensor a través del topic **/ir_line**, que publica datos de tipo **Float32** en el rango de 0.0 a 1.0 dependiendo del nivel de luminancia detectado.

Este nodo también se encuentra conectado con el sistema de seguridad del robot. A través del topic **/safety_stop**, el nodo puede recibir una señal de parada que interrumpe cualquier movimiento en caso de detectar un obstáculo mediante el sensor LIDAR.

- **Subscripciones**
    a. /ir_line
        i. Tipo de mensaje**: std_msgs/msg/Float32**
        ii. Función: Recibe la luminancia detectada por el sensor TCRT5000. Se utiliza para determinar si el robot está sobre la línea negra o no.
    b. /safety_stop
        i. Tipo de mensaje: **std_msgs/msg/Bool**
        ii. Función: Señal enviada por el nodo del LIDAR. Si es true, indica que se ha detectado un obstáculo y el robot debe detenerse inmediatamente.
- **Publicaciones**
    a. /cmd_vel
        i. Tipo de mensaje: **geometry_msgs/msg/Twist**
        ii. Función: Comandos de velocidad lineal y angular para controlar el movimiento del robot.

```
// ── Suscriptor a la luminancia ──────────────────────────
lum_sub_ = create_subscription<std_msgs::msg::Float32>(
    "/ir_line", rclcpp::SensorDataQoS(),
    std::bind(&IrLineFollower::lumCallback, this, std::placeholders::_1));

safety_sub_ = create_subscription<std_msgs::msg::Bool>(
    "/safety_stop", rclcpp::SensorDataQoS(),
    std::bind(&IrLineFollower::safetyCallback, this, std::placeholders::_1));

// ── Publicador /cmd_vel ─────────────────────────────────
cmd_pub_ = create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);
```

Figura 24: Subscripciones y publicaciones de los diferentes tópicos.

Callbacks

Para gestionar la lectura de los valores de luminancia y la señal de parada por colisión, se utilizan funciones callback. Estas funciones se ejecutan automáticamente cada vez que se recibe un nuevo mensaje en los tópicos a los que están suscritas, permitiendo una respuesta inmediata a los cambios del entorno.

Aquí os dejamos una imagen con la definición de ambas funciones:

```
void lumCallback(const std_msgs::msg::Float32::SharedPtr msg)
{
  last_lum_ = msg->data;              // guarda el valor (0-1)
  have_lum_ = true;
}

void safetyCallback(const std_msgs::msg::Bool::SharedPtr msg)
{
  safety_stop_ = msg->data;
}
```

Figura 25: Funciones callback.

Estas funciones sólo almacenan los valores leídos en el tópico para poder utilizarlos para la lógica de seguimiento de líneas y la parada por colisión.

Algoritmo de seguimiento de líneas

El algoritmo se basa en la siguiente ecuación:

$$\omega = -k_{LINE}(lineColour - grey)$$

Figura 26: Ecuación seguimiento de líneas

Componentes de la ecuación

- **ω (omega)**
  Es la velocidad angular del robot. Es decir, cuánto y en qué dirección debe girar.

- **k_{line} (k_LINE)**
  Es una constante de proporcionalidad que regula la sensibilidad del robot a los cambios de color detectados. Un valor mayor hace que el robot gire más bruscamente. En nuestro proyecto k equivale a 5 llegando a este valor mediante prueba y error.

- **lineColour**
  Es el valor de intensidad de color detectado por el sensor que se lee en el tópico **/ir_line**.

- **Grey**
  Es el valor de referencia que representa el color intermedio (gris) entre la línea y el suelo. Se usa como umbral para saber si el sensor está alineado o no. El valor de **Grey** en nuestra simulación es de 0.35, ya que el suelo tiene una luminancia de 0.7 y la línea negra de 0.1 aproximadamente.

Funcionamiento del algoritmo

El algoritmo compara el color actual detectado (lineColour) con el valor esperado para el gris (grey). La diferencia entre ambos se multiplica por la constante k_LINE, y el resultado se usa para ajustar la velocidad angular.

- Si el sensor ve **más oscuro que el gris** (es decir, está sobre la línea negra), la diferencia será negativa ⇒ ω será positiva ⇒ el robot gira hacia un lado.
- Si el sensor ve **más claro que el gris** (es decir, está fuera de la línea), la diferencia será positiva ⇒ ω será negativa ⇒ el robot gira hacia el otro lado.
- Si el sensor ve exactamente el gris ⇒ ω = 0 ⇒ el robot sigue recto.

```
void timerCb()
{
  if (!have_lum_) return;              // espera primera lectura

  if (safety_stop_)
  {
    cmd_pub_->publish(geometry_msgs::msg::Twist());  // Twist(0,0)
    return;
  }

  const double v   = get_parameter("v_forward").as_double();
  const double k   = get_parameter("k_line").as_double();
  const double ref = get_parameter("grey_ref").as_double();

  double omega = -k * (last_lum_ - ref);   // ley de control ω

  // Publica Twist (cmd_vel)
  geometry_msgs::msg::Twist cmd;
  cmd.linear.x  = v;
  cmd.angular.z = omega;
  cmd_pub_->publish(cmd);
}
```

Figura 27: Función timerCb().

En esta imagen observamos la implementación del algoritmo.

Con la condición del **safety_stop_** podemos controlar si el sensor LIDAR ha detectado un peatón, ya que, en el tópico correspondiente, se publicará True en el caso de detección y False en el otro caso, por lo que, si se ha detectado un objeto, la condición se cumplirá y se publicará en el tópico **/cmd_vel** encargado del movimiento del robot (0, 0) para que no se mueva.

Una vez el LIDAR deje de detectar obstáculo, se publicará False y por tanto no entrará en la condición. En el tópico **/cmd_vel** se publica el valor angular correspondiente calculado por el algoritmo y el valor de la velocidad lineal correspondiente.

Implementación en Coppelia -> cmdVelScript

```
function sysCall_init()
    left_motor=sim.getObject('../left_motor')
    right_motor=sim.getObject('../right_motor')
    cmdVelSub=simROS2.createSubscription('/cmd_vel','geometry_msgs/msg/Twist','setCmdVel_cb')
    r=0.033
    b=0.08
    maxVel=0.2
    -- do some initialization here
end

function setCmdVel_cb(msg)
    local v=msg.linear.x
    local w=msg.angular.z
    local vL=v-b*w
    local vR=v+b*w
    if (vL>maxVel) then
        vL=maxVel
    elseif (vL<-maxVel) then
        vL=-maxVel
    end
    if (vR>maxVel) then
        vR=maxVel
    elseif (vR<-maxVel) then
        vR=-maxVel
    end
    local wL=vL/r
    local wR=vR/r
    sim.setJointTargetVelocity(left_motor,wL)
    sim.setJointTargetVelocity(right_motor,wR)
end
```

Figura 28: Script cmdVel.

Como se puede observar en la imagen siguiente, se establece una suscripción al tópico /cmd_vel, que está asociado a la función *callback* **setCmdVel_cb(msg).** Esta función se activa automáticamente cada vez que se publica un nuevo mensaje en dicho tópico, lo cual permite al robot responder de forma reactiva a las órdenes de movimiento.

El mensaje recibido contiene tanto la velocidad lineal como la velocidad angular deseadas para el robot. A partir de estos valores, la función **setCmdVel_cb** calcula la velocidad adecuada para cada una de las ruedas del robot diferencial, distribuyendo el movimiento de manera que se pueda avanzar en línea recta, girar o realizar trayectorias curvas, según lo especificado.

El script en la jerarquía del robot es hijo de ambos motores.

Lidar_bridge_node

El nodo **lidar_bridge_node** actúa como un puente de comunicación entre el entorno de simulación CoppeliaSim y el ecosistema ROS2 para transmitir los datos capturados por el sensor LIDAR. Su principal responsabilidad es recoger las mediciones de distancia generadas por el LIDAR virtual y empaquetarlas en mensajes del tipo **sensor_msgs/msg/LaserScan**, que son estándar en ROS2 para este tipo de sensores. Gracias a este nodo, es posible integrar fácilmente los datos del LIDAR en la arquitectura del sistema robótico, permitiendo que otros nodos, como el encargado de la detección de obstáculos o el de parada de emergencia, puedan suscribirse y utilizar esta información para tomar decisiones en tiempo real.

Este nodo se convierte en una pieza clave para dotar al robot de una percepción precisa del entorno, contribuyendo directamente a su capacidad de navegación segura y autónoma.

Este nodo actúa únicamente como **publisher**, enviando los datos del sensor virtual al resto del sistema.

- **Subscripciones**
  a. /scan
    i. Tipo de mensaje**: sensor_msgs/msg/LaserScan**
    ii. Función: Recibe un listado de todas las distancias a objetos detectadas por el sensor LIDAR
- **Publicaciones**
  a. /cmd_vel
    i. Tipo de mensaje: **geometry_msgs/msg/Twist**
    ii. Función: Comandos de velocidad lineal y angular para controlar el movimiento del robot.
  b. /safety_stop
    i. Tipo de mensaje: **std_msgs/msg/Bool**
    ii. Función: Enviar True si el LIDAR detecta objeto y False en el otro caso.

```
laser_sub_ = create_subscription<sensor_msgs::msg::LaserScan>(
    "/scan", qos,
    std::bind(&LidarBridge::scanCb, this, std::placeholders::_1));

cmd_pub_  = create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);
stop_pub_ = create_publisher<std_msgs::msg::Bool>("/safety_stop", 10);
```

Figura 29: Subscripciones y publicaciones de los diferentes tópicos.

Callbacks

Para gestionar la lectura de los diferentes valores detectados por el sensor LIDAR y comprobar que esos valores son válidos, se utiliza la función callback **scanCb**. Esta funcione se ejecuta automáticamente cada vez que se recibe un nuevo mensaje en el tópico **/scan** al que está suscrito el nodo, permitiendo una respuesta inmediata a los cambios del entorno.

Aquí os dejamos una imagen con la definición la función:

```
void scanCb(const sensor_msgs::msg::LaserScan::SharedPtr msg)
{
        double best = std::numeric_limits<double>::infinity();

        for (float d : msg->ranges)
        {
          if (std::isfinite(d))            // descarta inf y NaN
            best = std::min(best, static_cast<double>(d));
        }
        min_dist_ = best;

 }
```

Figura 30: Función callback.

Esta función analiza todos los valores del listado que se han publicado en el tópico, descarta los valores infinitos y Nan y guarda en la variable **min_dist_** la distancia mínima detectada.

Luego, con la distancia mínima del mensaje podemos continuar con la lógica de la detección de objetos.

Función para la lógica de la detección de objetos y detención del robot

```
void timerCb()
{
  const double safe = get_parameter("safe_distance").as_double();

  if (min_dist_ < safe && !stopped_)            // obstáculo → STOP
  {
    geometry_msgs::msg::Twist stop;             // todo a 0
    cmd_pub_->publish(stop);
    stopped_ = true;
    RCLCPP_WARN(get_logger(),
                "Obstáculo a %.2fm → STOP", min_dist_);
  }
  else if (min_dist_ >= safe && stopped_)       // libre otra vez
  {
    stopped_ = false;
    RCLCPP_INFO(get_logger(), "Camino libre de nuevo");
    // El seguidor de línea retomará con su próximo /cmd_vel
  }
  std_msgs::msg::Bool flag;
  flag.data = stopped_;
  stop_pub_->publish(flag);
}
```

Figura 31: Función timerCb().

En primer lugar, se obtiene el valor del parámetro **safe_distance**, que representa la distancia mínima segura a la que el robot puede acercarse a un obstáculo antes de detenerse, siendo de 0.17 m. Esta distancia se compara con la distancia mínima **min_dist_** detectada por el LIDAR.

- Si se detecta un obstáculo a una distancia menor que la segura **min_dist_ < safe_distance** y el robot aún no estaba detenido **!stopped_**, se genera un mensaje de velocidad Twist con todos los valores a cero para detener el robot. A continuación, se establece la variable **stopped_** a True y se muestra un mensaje de advertencia informando de que se ha activado la parada.
- Por el contrario, si el camino vuelve a estar libre **min_dist_ >= safe_distance** y el robot estaba detenido **stopped_ == true**, se marca la variable **stopped_** como False y se imprime un mensaje informativo indicando que el camino vuelve a estar despejado. Esto permite que el robot retome su movimiento cuando reciba un nuevo mensaje de velocidad en el topic correspondiente.

```
[lidar_bridge_node-4] [WARN] [1751476669.768020274] [lidar_bridge]: ⚠️ Obstáculo a 0.17m →STOP
[lidar_bridge_node-4] [INFO] [1751476671.246730640] [lidar_bridge]: Camino libre de nuevo
```

Figura 32: Output terminal detección obstáculo y camino libre.

En esta imagen, vemos los mensajes que se imprimen por pantalla al detectar un obstáculo y cuando este obstáculo desaparece.

Traffic_ligth_node

El nodo del semáforo **traffic_light_node** se encarga de simular el control de paso en los pasos de peatones del entorno de simulación. Su función principal es regular el movimiento de los personajes animados (como Bill) que cruzan el paso de cebra, mediante el envío de una señal de tipo booleano que indica si el paso está permitido o no. Este nodo actúa como un sistema de control externo que permite habilitar o deshabilitar el movimiento de los peatones según una lógica que puede ser activada manualmente. Gracias a esta implementación, se logra un mayor realismo en la simulación, añadiendo interacciones dinámicas entre el entorno y el robot móvil, como la necesidad de ceder el paso o esperar ante una situación de cruce.

- **Publicaciones**
    a. /crossing_green
        i. Tipo de mensaje: **std_msgs/msg/Bool**
        ii. Función: Publicar True si queremos que los peatones se muevan y aparezcan y False si queremos que desaparezcan.

```
declare_parameter<bool>("green", true);          // arranca en verde
pub_ = create_publisher<std_msgs::msg::Bool>("/crossing_green", 10);
```

Figura 33: Declaración del publicador

Función principal

```
void timerCb()
{
  std_msgs::msg::Bool msg;
  msg.data = get_parameter("green").as_bool();
  pub_->publish(msg);
}
```

Figura 34: Función timerCb().

Esta función se encarga simplemente de publicar o True o False en el tópico de manera manual, ya que se trata de un semáforo manual, para ello, necesitamos abrir una nueva terminal y poner el siguiente comando: **ros2 param set /traffic_ligth green true/false**.

Lo que conseguimos con este comando es poner la variable **green** declarada en el nodo en verdadero o falso.

Luego en el script del **Walking Bill** tenemos una suscripción al tópico **/traffic_light** con una función callback.

Función callback

```
function green_cb(msg)
    local newState = msg.data     -- true = verde, false = rojo
    if newState == canMove then return end   -- nada que hacer si no cambia
    canMove = newState

    if not canMove then

        storedOrient = sim.getObjectOrientation(BillHandle, -1)
        local p = sim.getObjectPosition(BillHandle, -1)
        sim.setObjectPosition(BillHandle, -1, {p[1], p[2], p[3] + downOffset})
    else

        local p = sim.getObjectPosition(BillHandle, -1)
        sim.setObjectPosition(BillHandle, -1, {p[1], p[2], p[3] - downOffset})
        sim.setObjectOrientation(BillHandle, -1, storedOrient)
    end
end
```

Figura 35: Función green_cb(msg).

La función **green_cb(msg)** es una función de tipo *callback* que se activa automáticamente al recibir un nuevo mensaje en el tópico del semáforo **/crossing_green**.

En primer lugar, la función extrae el nuevo estado del mensaje **newState = msg.data** y, si no hay cambio respecto al estado anterior **canMove**, termina de inmediato, ya que no es necesario hacer ninguna modificación.

Si el estado ha cambiado y el semáforo ha pasado a rojo **not canMove**, el personaje Bill es "ocultado" visualmente y dejado fuera del alcance del LIDAR. Para ello, se guarda su orientación actual **storedOrient**, se obtiene su posición, y se le traslada hacia abajo en el eje Z mediante un pequeño desplazamiento **downOffset**, de forma que queda bajo el plano de la simulación.

Cuando el semáforo vuelve a verde **else**, Bill es restaurado a su posición y orientación originales. Primero se le traslada de nuevo hacia arriba (revirtiendo el desplazamiento aplicado), y a continuación se le restaura la orientación previamente almacenada.

Este mecanismo permite simular de forma efectiva la desaparición y reaparición del peatón, coordinada con el estado del semáforo, mejorando la integración con el robot móvil y el entorno de simulación.

Detección y confirmación de entrega

En esta sección se describe la lógica implementada para la entrega del paquete por parte del robot móvil a un personaje estático del entorno. Este proceso forma parte fundamental del objetivo final del proyecto, ya que representa la culminación de la ruta del robot. La entrega se produce cuando el robot alcanza una posición específica cerca del destinatario, lo que se detecta mediante un cálculo de proximidad. Una vez cumplida esta condición, se genera un mensaje visual que indica que el paquete ha sido entregado correctamente, y se simula la entrega eliminando el objeto del paquete de la escena. Este comportamiento mejora el realismo de la simulación y refuerza el propósito logístico del sistema.

```
function sysCall_init()
    dummy      = sim.getObject('.')
    robotBase  = sim.getObject('/odom/base_footprint')
    cuboidPaquete = sim.getObject('/odom/turtlebot3_burger/Paquete')
    threshold  = 0.10          -- distancia para entregado (m)
    pub        = simROS2.createPublisher('/package_delivered',
                                         'std_msgs/msg/Bool')
end

local function round2(x)
    return math.floor(x * 100 + 0.5) / 100     -- 2 decimales
end

function sysCall_sensing()
    if delivered then return end

    local r = sim.getObjectPosition(robotBase, -1)
    local d = sim.getObjectPosition(dummy, -1)
    local dist = math.sqrt((r[1]-d[1])^2 + (r[2]-d[2])^2)
    print(dist)
    --if dist > 2.73 then
        --sim.displayDialog('Ruta Finalizada','Ruta Finalizada',
          --      sim.dlgstyle_ok,false)
        --sim.stopSimulation()
    --end
    if dist > 2.73 then
        delivered = true
        sim.displayDialog('Entrega realizada','Paquete ENTREGADO',
                  sim.dlgstyle_ok,false)
        simROS2.publish(pub,{data=true})
        sim.setObjectInt32Param(cuboidPaquete,
                        sim.objintparam_visibility_layer, 0)
    end
end
```

Figura 36: Script entrega del paquete.

Este script, dada la distancia desde un **control point** del camino a la distancia actual del robot, con la condición **if dist > 2.73** que concuerda con el punto visual donde se encuentra el muñeco, muestra por pantalla un cuadro para informar que la entrega se ha realizado, a continuación publica en el tópico **/package_delivered** un valor True para indicar que el paquete se ha entregado y por tanto ya no entre en la lógica y por último, desmarca la casilla de la capa de visibilidad 0 para que visualmente haga el efecto que el paquete se ha entregado.

A continuación, os dejamos una imagen en la que se ve el correcto funcionamiento de la entrega del pedido.

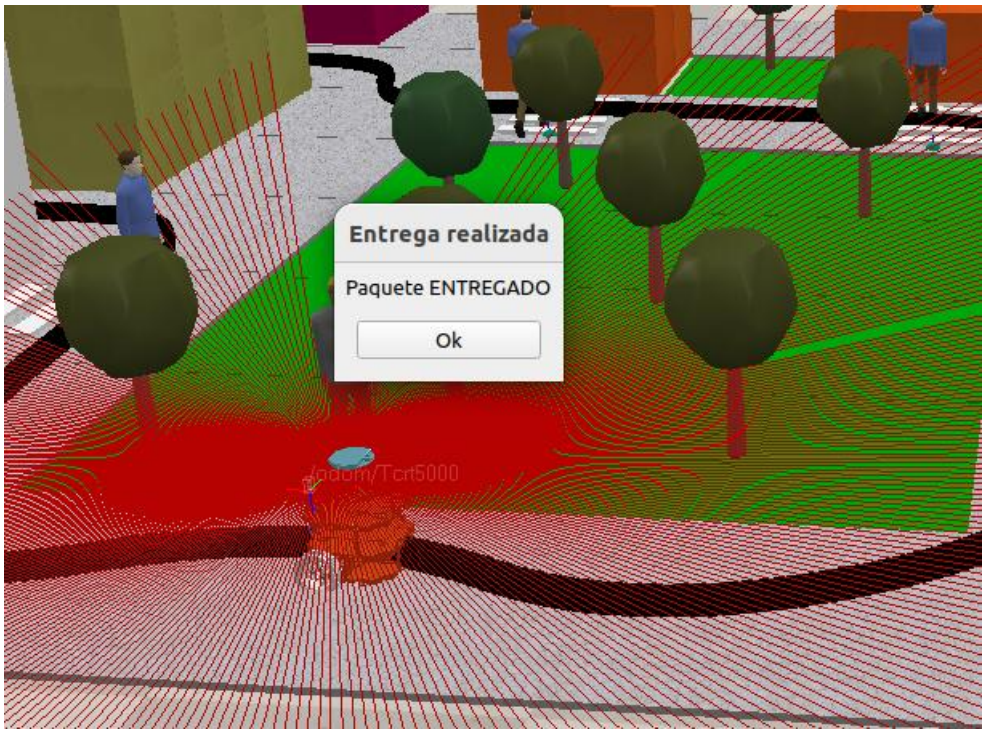

Figura 37: Visualización de la entrega del pedido.

Coordinación mediante archivo launch

El archivo launch.py, ubicado en la carpeta launch/, permite iniciar todos los elementos esenciales del sistema con un solo comando. Este archivo coordina la ejecución de los diferentes nodos y la simulación, facilitando una puesta en marcha rápida y organizada del entorno. Entre los componentes que se lanzan automáticamente se incluyen:

- **Simulación en CoppeliaSim**
  Cargando de forma automática la escena.ttt previamente diseñada, que contiene tanto los elementos de decoración como los componentes funcionales del sistema (robot, caminos, sensores y personajes).

- **Publicación de datos del sensor LIDAR**
  Gracias a un script LUA que extrae las mediciones del escáner láser y las publica en un tópico de ROS2.

- **Nodo ir_line_follower_node**
  Responsable de la lógica de seguimiento de línea mediante la lectura del sensor TCRT5000 y el control de velocidades del robot.

- **Nodo lidar_bridge_node**
  Encargado de analizar las distancias obtenidas por el LIDAR y detener el robot si se detectan obstáculos peligrosamente cercanos.

- **Nodo traffic_light_node**
  Simula un semáforo virtual controlando el movimiento del personaje "Bill" en los pasos de cebra.
- **Visualización con RViz**
  Permitiendo observar la odometría del robot, las trayectorias planificadas y las detecciones de los sensores en tiempo real.

Visualización opcional con herramientas de consola, como ros2 topic echo, que permiten monitorizar en tiempo real el estado de los sensores y el comportamiento del sistema.

Además, este archivo launch permite la configuración de parámetros clave mediante **LaunchConfiguration**, como por ejemplo la ruta absoluta del archivo de escena **scene_dir** o la activación del uso de tiempo simulado **use_sim_time**. Esto proporciona flexibilidad a la hora de ajustar la simulación sin necesidad de modificar el código fuente.

Conclusión

El desarrollo de este proyecto ha permitido implementar un sistema robótico funcional y autónomo dentro de un entorno simulado complejo y realista, utilizando como base la arquitectura ROS2 y la plataforma de simulación CoppeliaSim. A lo largo del trabajo, se ha construido un entorno urbano completo con elementos de decoración y obstáculos, así como con múltiples componentes dinámicos, como pasos de cebra y peatones móviles.

El robot ha sido diseñado para seguir una línea negra en el suelo utilizando un sensor TCRT5000, integrando además un sensor LIDAR que le permite detectar obstáculos y detenerse automáticamente cuando existe riesgo de colisión. Gracias a la comunicación entre nodos mediante ROS2, ha sido posible establecer un comportamiento coordinado entre distintos componentes del sistema, como la gestión del semáforo virtual y la lógica de entrega del paquete a un personaje estático.

Se han desarrollado múltiples nodos personalizados en C++ y scripts en LUA para controlar tanto el hardware simulado como la lógica de comportamiento. La interacción entre estos nodos demuestra un uso adecuado de las capacidades de ROS2 en la gestión de múltiples procesos concurrentes, incluyendo la publicación/suscripción de datos, temporizadores, y la parametrización desde archivos de configuración o launch.

En resumen, el proyecto cumple con los requisitos funcionales y técnicos establecidos: simula de forma efectiva una entrega autónoma de paquetes en un entorno urbano con obstáculos y peatones, aplicando conceptos avanzados de robótica, percepción y control en un marco de desarrollo profesional. Además, la modularidad y organización del código permiten una fácil extensión futura para añadir nuevas funcionalidades o sensores.

Referencias bibliográficas

- https://chatgpt.com
- https://www.coppeliarobotics.com/
- https://stackoverflow.com/questions
- https://www.youtube.com/watch?v=l32IiRkCwxg&list=PLjzuoBhdtaXOYfcZOPS98uDTf4aAoDSRR&ab_channel=LeopoldoArmesto