Robótica 2021-2022

1st Andrés Ventura Valiente
aventurav@alumnos.unex.es

2nd Carlos Sánchez García
csancheznf@alumnos.unex.es

3rd Yeray Bravo Díaz
ybravodi@alumnos.unex.es

I. INTRODUCCIÓN

Los robots autónomos tienen la aplicabilidad y capacidad de poder ejecutar actividades y tareas sin la necesidad de algún tipo de comando y control directamente explícito de los humanos. Son funcionales en industrias, comercios, espacios personales... además de áreas de trabajo tan diferentes como el agua, el aire y el espacio.

Somos tres estudiantes de Ingeniería Informática en Ingeniería de Computadores de la Escuela Politécnica de Cáceres. Este documento trata la implementación de un robot de limpieza de la asignatura Robótica, el que contiene las diferentes fases de diseño, implementación y pruebas.

El documento va a contener apartados según vayamos mejorando el modelo, los cuales se mencionan a continuación:

- Sesión 1: instalación del software necesario e introducción a C/C++.
- Sesión 2: instalación del software necesario y la implementación del robot para que recorra el mayor espacio espacio posible en un tiempo dado.
- Sesión 3: la implementación del robot para que se desplace a lugar especifico seleccionado.
- Sesión 4: la misma implementación anterior pero pudiendo detectar y evitar los obstáculos que se encuentre el robot por el camino.
- Sesión 5: la implementación del robot para que detecte y diferencie las diferentes secciones de una sala.

II. SESIÓN 1

Para esta primera sesión se van a explicar que herramientas hemos tenido que instalar y la introducción a C/C++.

A. Herramientas

La principal herramienta para poder llevar a cabo la implementación del robot es instalar Ubuntu 20.04 ya que todas las librería relacionadas con el robot sólo son compatibles con ese sistema operativo. Otra herramienta que vamos a necesitar es el compilador GCC/G++ para C++10. También vamos a tener que instalar las libreias de Qt5 para poder crear interfaces de usuario. Después tenemos que instalar Git que es el lenguaje de creación y mantenimiento de repositorios que es dónde vamos a tener almacenada toda nuestra implementación. Para poder implementar el robot, vamos a instalar CLion que es un IDE de programación de JetBrains. También tenemos que instalar Ice de ZeroC que es un middleware de comunicación utilizado como parte de RoboComp. Y para poder crear la documentación vamos a tener que crearnos una cuenta de Overleaf, que es una página web de documentos en Latex.

B. Objetivos de la sesión

Para esta priemra sesión se han realizado cambios en dos ejercicios distintos propuestos, el primero consta modificar el código para que el contador comience la cuenta y habilitar el botón de STOP para poder parar la cuenta en un momento en concreto. Y la segunda parte consta en entender el código fuente proporcionado y poder cambiar el periodo del contador.

- Primera propuesta:

  En primer lugar en counterDlg.ui hemos añadido los botones de START y un dial, hemos realizado cambios para que en la frecuencia que se selecciona salga en el display y start comience con la frecuencia seleccionada. A continuación hemos creado un método llamado ”doButtonStart” que su función es que funcione el boton START guardando la nueva frecuencia seleccionada con el día en una variable y que se muestre por pantalla, iniciando el contador con la nueva frecuencia al pulsar en el nuevo botón de Start, estos cambios se han realizado en ”ejemplo1.cpp”, y en ”ejemplo1.h” hemos declarado las variables numfrecuencia, QTimer, y el método doButtonStart.

- Segunda propuesta:

  Para la segunda propuesta, tras analizar y entender el código, se nos pedía que hubiese un contador desde el inicio y que se pueda cambiar el periodo igual que en el primer ejercicio. Para realizar estos cambios, en ”ejemplo1.h” declaramos la variable para el nuevo contador y el método ”cuentaLong”, y en ”ejemplo.cpp” declaramos un connect con el nuevo método para que arranque el contador de segundos desde el inicio, mostramos el valor del periodo en su pantalla correspondiente y el método cuentaLong que se encarga de incrementar el contador de segundos. Por último en la clase ”timer.h” hemos creado un getPeriod que devuelve el periodo actual.

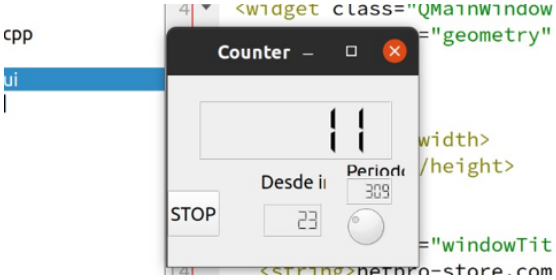

Fig. 1. Imagen QTimer

III. SESIÓN 2

A. Herramientas

En primer lugar, antes de comenzar a realizar los cambios necesarios para que la segunda sesión fuese correcta hemos tenido que instalar el simulador de CoppeliaSim que es el entorno de simulación del robot con el que vamos a probar las implementaciones durante la siguientes sesiones.También hemos tenido que instalar la librería PyRep, que es necesaria para poder ejecutar el simulador CoppeliaSim. A parte de instalar estas dos herramientas, también hemos tenido que instalar la librería del repositorio de Robocomp, en la cual incluyen algunos componentes. Principalmete el componente para representar el espacio de trabajo, para visualizar el espacio recorrido, para el correcto funcionamiento y detección del láser, para poder controlarlo con hardware externo y para controlar la velocidad de movimiento.

B. Objetivos de la sesión

Para esta sesión se precisa de que nuestro robot de limpieza sea capaz en avanzar y en poder detectar obstáculos para así evitar colisiones, y el principal objetivo es que limpie el mayor espacio posible en el menor tiempo posible. Para la obtención de la sesión se han seguido unos pasos:

- Creación del componente Controlador:

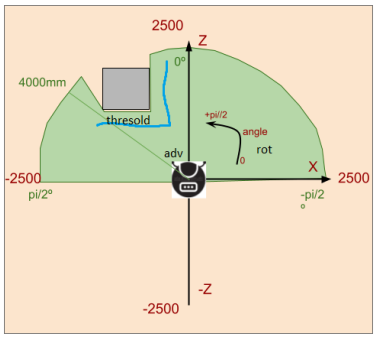

Fig. 2. Imagen del láser y ángulos de rotación del robot

Hemos tenido que crear el componente "Controlador" con el editor robocompdsl. Este componente es que condoce al robot entre llos obtáculos evitando colisiones. En el módulo compute de la clase SpecificWorker.cpp es donde hemos realizado cambios para que sea posible el objetivo. Tenemos una constante "threshold" declarada para que el robot se pare a la distancia especificada, que en nuestro caso es de 365 milímetros. Una variable "rot" para la rotación del robot, que en nuestro caso es de 0.8 radianes por segundo. Y otra variable "adv" para la velocidad de avance del robot, que en nuestro caso es de 220 milímetros por segundo.

- Si el robot encuentra un obstáculo:

Para poder controlar si el robot choca, tenemos en cuenta en todo momento los datos que obtiene el láser del propio robot(RoboCompLaser::TLaserData ldata = laserproxy-getLaserData()). Para este caso, hacemos la comprobación de si nuestra primera posición válida del vector de datos del láser "ldata" es menor a nuestra constante de "threshold". Si es menor, lo que hacemos es que a través de una estructura switch, que en el primer caso el robot retroceda

```
differentialrobotproxy->setSpeedBase(-120, 0)
```

y que después rote

```
differentialrobotproxy->setSpeedBase(0, rot)
```

si no le es posible rotar a causa de una colisión, que entre en el siguiente caso del switch, y que haga exactamente lo mismo pero rotando en sentido contrario

```
differentialrobotproxy->setSpeedBase(0, -rot)
```

En ambos casos hacemos que el robot retroceda durante 2 segundos. Y dónde la rotación varía según la figura 2.

- Si el robot no encuentra ningún obstáculo:

En este caso tenemos controlado que la primera posición del vector de datos del láser sea mayor que nuestra constante "threshold". Cómo el objetivo de la entrega es que recorra el mayor espacio posible en el menor tiempo posible, el primer movimiento que hace el robot es una espiral

```
differentialrobotproxy->setSpeedBase(adv,0.8)
```

y dónde

$$adv = adv + 16$$

hasta que choque o hasta que alcanca la velocidad de 1000(hemos decidido hacer una espiral ya que si hay un gran espacio libre, puede ser una buena opción para que barra el mayor espacio posible de ese hueco), el segundo

y tercer movimiento es el mismo con la idea de evitar que se quede en alguna esquina o perdiendo eficacia entre objetos

```
differentialrobotproxy->setSpeedBase(800, 0)
```

Y el cuarto movimiento es que avance con una pequeña rotación para evitar siempre el avance en línea recta, y así puede avanzar con una diferente inclinación

```
differentialrobotproxy->setSpeedBase(800, -0.11)
```

C. Pruebas

Tras el correcto funcionamiento del robot, hemos procedido a realizar varias pruebas. Hemos concluido tres pruebas de 5 minutos cada una. En la primera prueba hemos obtenido un recorrido de un 26.96. En la segunda prueba un recorrido de 27.2. Y en la tercera prueba un recorrido de 27.68. En el que de media hemos tenido un recorrido de 27.28.

- Prueba 1:

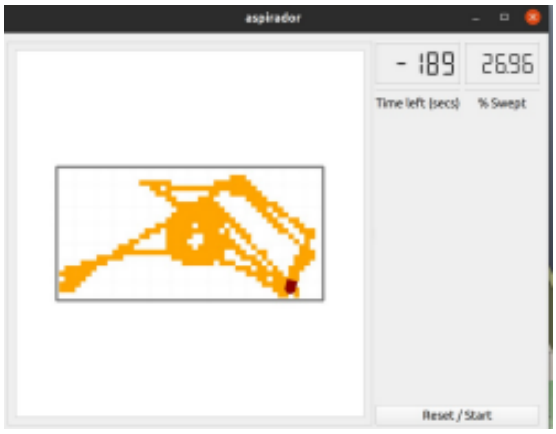

Fig. 3. Imagen de la prueba 1

- Prueba 2:

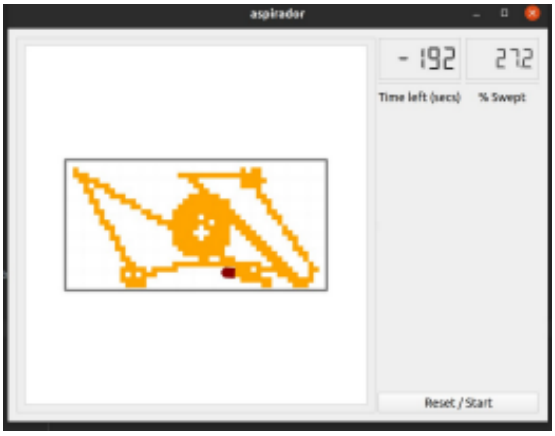

Fig. 4. Imagen de la prueba 2

- Prueba 3:

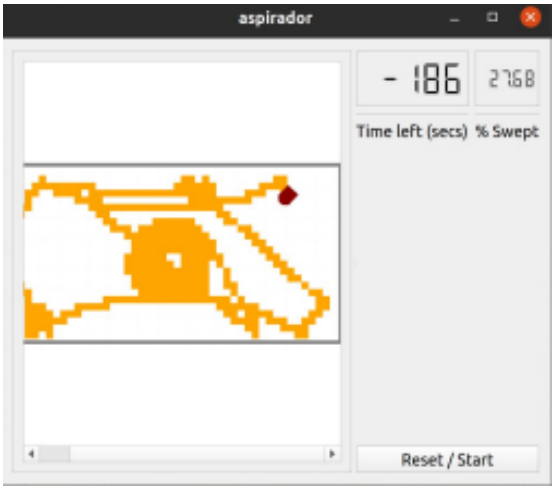

Fig. 5. Imagen de la prueba 3

IV. SESIÓN 3

Para esta segunda sesión se va a explicar la implementación del robot para que se desplace a lugar especifico seleccionado

A. Herramientas

En primer lugar, antes de comenzar a realizar los cambios necesarios para que la tercera práctica fuese correctamente tuvimos que añadir un componente nuevo, llamado gotoxy. Y con el cual tuvimos que crear la definición del componente: robocompdsl gotoxy.cdsl. Este componente nos permite poder hacer click en una parte del mapa, y que nuestro robot se desplace hasta ese punto.

B. Objetivos de la entrega

Para esta entrega se precisa de que nuestro robot de limpieza sea capaz de que si hacemos click en una parte del mapa vaya

hacía ese punto automaticamente. Y también se precisa de que en nuestro componente recién creado, también se visualice el láser del propio robot. Para la obtención de esta práctica se han seguido unos pasos:

- Visualizar el láser del robot:

  Hemos tenido que crear el componente "gotoxy" con el editor robocompdsl. Para que el componente pueda dibujar el láser hemos creado el método drawlaser, que tiene como parámetro la información del láser. Está representado en un póligono, por lo que creamos un póligono: QPolygonF poly, y dentro del póligono le vamos metiendo todos los datos que va procesando:

```
poly << QPointF(l.dist * sin(l.angle), l.dist * cos(l.
    angle))
```

  Pero para poder desarrollar correctamente el polígono, debemos añadir la coordenada 0,0, para que pueda empezar y terminar el polígono desde la ubicación del láser:

```
poly << QPointF(0,0)
```

  Y poder visualizar el láser en todo momento, lo declaramos en nuestro método compute, que está en todo momento en funcionamiento después de obtener los datos del láser:

```
1. RoboCompLaser::TLaserData ldata = laserproxy->
    getLaserData()
2. drawlaser(ldata)
```

- Capturar el click:

  Para poder conseguir que nuestro robot aspirador pueda ir a la ubicación que queramos, primero tenemos que capturar las coordenadas de la ubicación deseada. En primer lugar, creamos una estructura donde vamos a tener tres atributos: un T, que lo inicializamos como un Vector2f, para almacenar las coordenadas, un QPointF que según hacemos click las guarda y un booleano que si hacemos click se pone a true y cuando llega al punto se pone a false. Primero conseguimos las coordenadas a través del método click:

```
void SpecificWorker::click(QPointF punto)
```

  que lo que hace coge el click y lo guarda en el primer atributo de nuestro struct:

```
t1.content=Eigen::Vector2f (punto.x(),punto.y())
```

  y pone el booleano a true ya que ha detectado un click.

- Calcular la rotación y la distancia:

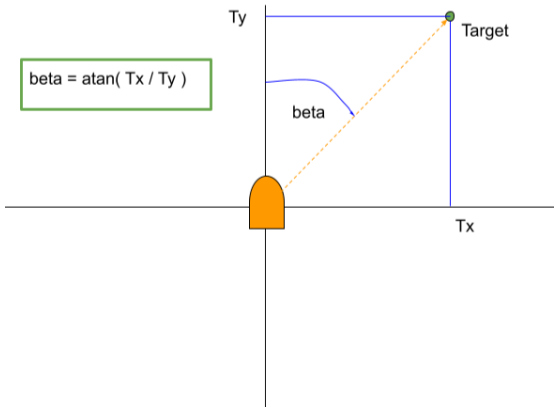

Fig. 6. Imagen del cálculo de beta

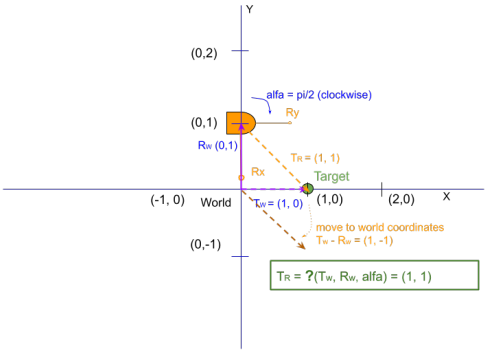

Fig. 7. Imagen de como se obtiene la rotación

Después de obtener las coordenadas, calculamos la rotación y la distancia que el robot deberá realizar para poder llegar a la ubicación deseada. Para ello, como el robot puede estar en cualquier sitio del mapa, primero guardamos el estado del robot:

```
Eigen::Vector2f rw(bState.x,bState.z)
```

después debemos calcular la orientación del robot que para ello debemos calcular la diferencia vectorial geométrica, que obtemos el punto opuesto:

$$Tr = R^t * (Tw - Rw)$$

donde R es:

$$\begin{matrix} cos(a) sin(a) \\ -sin(a) cos(a) \end{matrix}$$

y finalmete la rotación y la distancia teniendo Tr:

La rotación:

$$beta = atan2(Tx, Ty)$$

La distancia:

$$dist = ||Tr||$$

- Reducir la velocidad cuando se aproxima al punto:

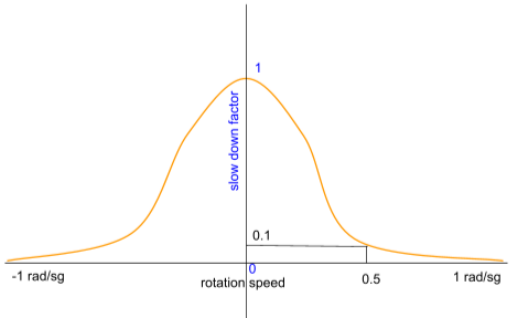

Fig. 8. Imagen de representación de la velocidad

El último punto a calcular, es la velocidad de avance ya que la de rotación la hemos calculado anteriormente en beta. Para calcular la velocidad de avance, tenemos que controlar que según esté llegando al punto vaya reduciendo la velocidad, y para ello aplicamos la siguiente fórmula:

$$rot_speed = beta$$

```
advspeed = MAXADVSPEED * reducespeedifturning *
     reducespeedifclosetotarget
```

Dónde MAXADVSPEED está limitada a 1000. Los otros dos factores están comprendidos entre 0 y 1, que son los factores que van a hacer que reduzca la velocidad de avance. Si esta acercándose al punto, compara con la velocidad máxima de avance, y si es mayor devuelve uno para que no influya en la velocidad, pero si es menor, devuelve el valor de la distancia entre la velocidad máxima de avance, que ahi ya implica que empiece a disminuir la velocidad. El otro factor implica a la rotación, si está rotando a 1 rad/s hace que el robot no avance, ya que devuelve un 0, y si la velocidad de rotación es de 0 rad/s hace que el factor sea 1, y no bloquea el avance. Para este último factor lo calculamos así:

$$y = exp((-beta * beta)/s)$$

y dónde s es:

$$s = pow(0.5, 2)/log(0.1)$$

dónde equivale a la siguiente fórmula:

$$F(angle) = e^{x^2/\lambda}$$

- Funcionamiento:

  Si el robot no detecta ningún click o ya ha llegado al destino, el robot tiene que estar parado:

```
differentialrobotproxy->setSpeedBase(0, 0)
```

  Y cuando detecta un click, lo que hace el robot es calcular la rotación en beta y introducirlo al robot para que comience con el movimientode rotación pero sin que avance:

```
differentialrobotproxy->setSpeedBase(0, beta)
```

  Una vez que el robot ya está orientado al robot, hay que calcular el avance:

```
adv = MAXADVANCE * stopifturning(beta) * stopifAttarget
     (dist)
```

  E introducirlo al movimiento del robot:

```
differentialrobotproxy->setSpeedBase(adv, 0)
```

V. SESIÓN 4

A. *Herramientas*

Para esta entrega vamos a seguir utilizando las mismas herramientas y el mismo componente que hemos usado en la sesión anterior, por lo que no va a haber ningún cambio en este aspecto.

B. *Objetivos de la entrega*

Para esta sesión nuestro robot de limpieza debe ser capaz de que si hacemos click en una parte del mapa vaya hacía ese punto automaticamente y si por el camino se encuentra un obstáculo, tiene que ser capaz de bordearlo y terminar en la ubicación del click.

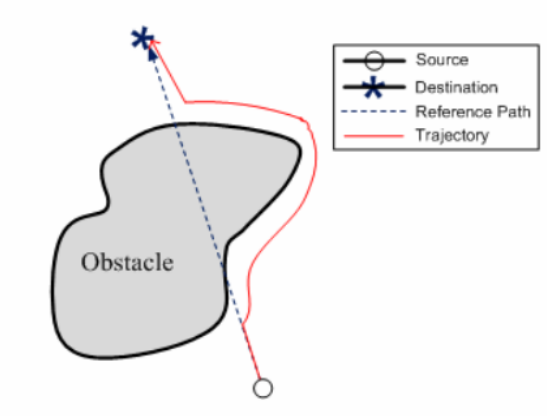

Fig. 9. Imagen de la idea del objetivo de la entrega

Es el mismo objetivo que la entrega anterior pero evitando cualquier obtáculo que se interponga en el camino. Para la obtención de esta práctica se ha implementado el algoritmo BUG, en el que se basa en diferentes estados por los que tiene que ir pasando el robot. A continuación en la imagen, se puede observar los estados que tiene y que condiciones tiene para pasar de un estado a otro:

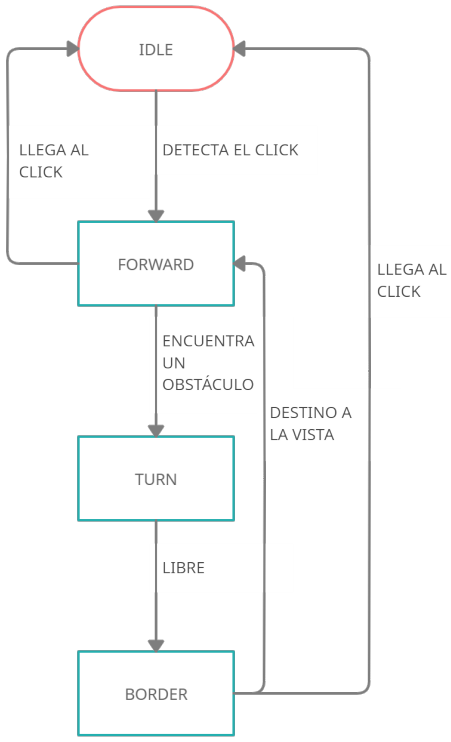

Fig. 10. Imagen de los estados del robot

A continuación, vamos a explicar como hemos implementado cada uno de los estados:

- IDLE:

  En este primer estado es en el que el robot tiene que comenzar y finalizar. El robot no hace nada, tan solo espera a que se haga un click para que se desplace como se ha hecho en la práctica anterior. Una vez que el robot recibe el click, se pasa del estado IDLE al estado FORWARD.

- FORWARD:

  En este segundo estado, el robot avanza hasta que sucedan dos cosas. La primera es que si llega a la distancia menor que 500 al obstáculo que pase al estado TURN, y sino llama al método FORWARD con los parámetros de estado, distancia y beta:

```
Forward(bState,distan,beta)
```

  que se explica a continuación, y la otra es que si llega a la ubicación del click que se pase al estado IDLE. El estado FORWARD se basa en lo siguiente: comprueba que si la distancia es menor que 120 se para el robot

```
differentialrobotproxy->setSpeedBase(0, 0)
```

  y si es mayor que 120 calculamos la velocidad de avance como en la práctica anterior. Como velocidad de giro usamos la beta ya calculada:

```
differentialrobotproxy->setSpeedBase(adv, beta)
```

- TURN:

  En este tercer estado, el robot gira hasta que tenga el frente libre y pase al estado BORDER. Primero comprueba que la distancia sea menor que 800 y llama a la función TURN. Esta función lo único que hace es lo siguiente:

```
differentialrobotproxy->setSpeedBase(10, 0.5)
```

- BORDER:

  En este cuarto estado, el robot bordea el obstáculo hasta que nos encontremos en dos casos diferentes. El primero es que al bordear se ponga en la ubicación deseada y pase al estado IDLE. Y el segundo es que cuando este bordeando tenga a la vista la ubicación deseada y pase al estado FORWARD. El estado lo que hace es llamar a la función border con los parámetros: los datos del láser, la distancia y el estado:

```
Border(ldata,distan,bState)
```

Lo primero que hace la función es almacenar las coordenadas de (x,y) de la ubicación destino en dos vectores. Después calculamos la rotación que tiene que hacer el robot como en la práctica anterior. Una vez que lo tenemos calculado, calculamos la distancia que tiene el robot hacia la izquierda y hacia la derecha:

```
min = std::minelement( ldataX.begin() + 10, ldataX.end
    () - 300, [](auto a, auto b){ return a.dist < b.
    dist; })
```

Después de tener la distancia izquierda y derecha calculadas, comparamos la distancia normal con 460, y si es menor hacemos que el robot rote:

```
differentialrobotproxy->setSpeedBase(0, 0.6)
```

si es mayor, compara la distancia derecha con 310, si es menor, y también compara la distancia izquierda con 460, y si es mayor, avanza y rota a la vez:

```
differentialrobotproxy->setSpeedBase(190, -0.6)
```

y si es menor, avanza y rota en sentido contrario:

```
differentialrobotproxy->setSpeedBase(190, 0.6)
```

Una vez que haya terminado de hacer estos pasos, calcula si las coordenadas de la ubicación destino están en el láser del robot, y para ello llama al método checkpoint con los parámetros de los datos del láser, y las coordenadas (x,y). Si el método devuelve true, cambia al estado FORWARD para que avance hasta el punto. Si el método delvuelve false, calcula la distancia hacía el punto destino:

```
distLinea = abs((VectorLinea[0] * bState.x) + (
    VectorLinea[1] * bState.z) + VectorLinea[2]) /
    sqrt(pow(VectorLinea[0],2) + pow(VectorLinea[1],2)
```

y el resultado lo compara con 30 y si es menor pasa al estado FORWARD.

VI. SESIÓN 5

A. Herramientas

Para esta entrega hay que hacer algunos cambios. Primero debemos clonar la repo https://github.com/robocomp/robocomp-giraff.git para poder cargar el nuevo mapa en Coppelia. Después en la clase specificworker.h debemos añadir robocomp/classes/grid2d/grid.h y grid.cpp. En el CMakelistsSpecific.txt hay que añadir grid.cpp en el sources block. Y finalmente crear una variable privada de tipo grid e inicializarla con un QRect de dimensiones varibles.

B. Objetivos de la entrega

En esta entrega, el objetivo es explorar el nuevo mapa, y que sea capaz de detectar lo que es pared y lo que es puerta. La función del robot se basa en dos estados, explorar la sala, que la explora y detecta las paredes y las puertas. Y el segundo estado, pasar de una sala a otra sala, pasando por el punto medio de la puerta.

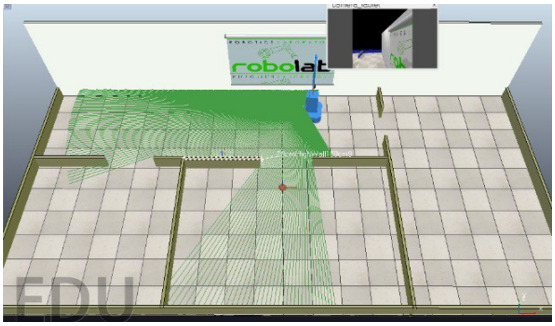

Fig. 11. Imagen del nuevo mapa

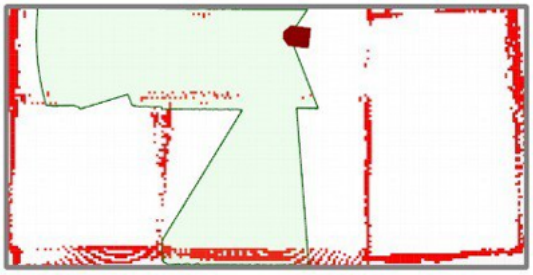

Fig. 12. Imagen del mapa detectado por el robot

- Explorar la sala:

  Lo primero que se debe hacer es explorar la sala. Para ello el robot gira sobre sí mismo, y en nuestro caso, si detecta algo a menor distancia que 4000 lo interpreta como pared y lo pinta en el mapa, y si no detecta nada significa que es puerta. Una vez que esta explorada la primera sala, detecta los dos puntos que demilitan la puerta, y en la que se calcula su punto medio.

- Pasar de una sala a otra:

  Como en el estado anterior ya tenemos el punto medio de la puerta calculado, tenemos que obtener la rotación del robot hacía ese punto y que después avance hacia la siguiente sala. Aquí es dónde no hemos podido continuar, ya que no hemos podido conseguir que el robot avance por la puerta.