UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Mecanismos de atención visual en RoboComp.

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Mecanismos de atención visual en RoboComp.

Autor: Cristina Mendoza Gutiérrez

Tutor: Pablo Bustos García de Castro

Co-Tutor: Pilar Bachiller Burgos

Resumen

El objetivo principal de este proyecto es dotar a Shelly, el robot social con el que se está trabajando actualmente en el RoboLab (grupo de investigación de la Universidad de Extremadura), de un sistema predictivo de atención visual que le permita situar y conocer la posición de los objetos de su entorno en el espacio, de una forma "semiconsciente".

El presente documento muestra el planteamiento, el desarrollo y la solución final a la que se ha llegado, al igual que algunos de los problemas o dudas que han ido surgiendo en el transcurso de la investigación. Consta de tres partes: una introductoria que comprende los capítulos 1 y 2, una teórica que se desarrolla en el capítulo 3 y una práctica que se expone en los capítulos 4, 5 y 6.

En la primera sección se trata sobre los conceptos necesarios para la comprensión de los diferentes algoritmos implementados: visión artificial del robot, diferentes tipos de cámaras existentes para plantear la solución, atención y consciencia en un robot, arquitectura de Shelly y elementos más importantes de su desarrollo para este apartado y explicación del entorno RoboComp.

En la segunda parte, la práctica, se explica el proceso hasta la implementación del objetivo final, qué dudas se han planteado, qué información debe almacenar el sistema y cuál no, los sistema de referencia existentes en el entorno que utilizaremos, el movimiento de los motores del cuello de nuestro robot, etc.

Por último, se analizan los resultados obtenidos, las mejoras realizadas, los experimentos llevados a cabo, las conclusiones a las que se ha llegado y las medidas pertinentes que hemos tomado para la corrección de algunos errores.

La memoria del trabajo concluye con algunas ideas sobre otras posibles mejoras a desarrollar en el futuro.

Abstract

The goal of this project is to endow Shelly, the social robot with which Robolab (Extremadura University research group) is currently working, with a predictive system of visual attention that allows it to situate and know the spatial position of its environment, in a ”semi-conscious” way.

This document shows the approach, the development and the final solution that has been reached, as well as some of the problems or doubts that have arisen during the course of this research.

The first section is dedicated to expose the necessary previous concepts in order to properly understand the different implemented algorithms: robot's artificial vision, different existing cameras to approach the solution, attention and consciousness of a robot, Shelly's architecture and most important elements of its development for this section and RoboComp environment explanation.

On the second section, the practice section, it is explained the process from the beginning to the final goal, what doubts have been raised, what information should the system store and what not, the existing reference systems in the environment that we will use, the movement of the neck engines of our robot, etc.

To finish up, the obtained results, the improvements made, the experiments carried out, the conclusions reached and the relevant measures that we have taken to correct some errors are analyzed.

The report of the work concludes with some ideas about other possible improvements to be developed in the future.

Índice

Índice de tablas

Índice de figuras

1. INTRODUCCIÓN

A día de hoy, los robots cada vez tienen una carga de trabajo mayor, más importante y más sofisticada, pero para que exista esta sofisticación, necesitamos que sean robots bastante avanzados, que implementen técnicas poco experimentadas hasta ahora y contando con la ayuda de todos los elementos hardware existentes hasta el momento.

Este trabajo se centra en dotar a un sistema sin atención, una cierta atención visual y consciencia sobre dónde está situado en el mundo y dónde se encuentran situados los objetos que le rodean. Para ello, utiliza técnicas de reconocimiento de objetos y posicionamiento.

Las pruebas se han realizado en el robot asistencial Shelly, un sistema que, como su propio nombre indica, sirve para asistir a personas mayores o discapacitados en algunas tareas motoras que no puedan realizar físicamente. Por tanto, el objetivo es que, si una persona, en un momento dado, interrumpe la trayectoria del robot, éste sea capaz de prever que la persona se va a interponer en su camino y consiga esquivarla o pararse con el fin de no obstaculizar su paso ni chocarse con ella. Para esto, comenzamos con ejemplos sencillos de objetos inertes que no presentan ningún tipo de movimiento ni velocidad de avance.

El sistema debe reconocer que, si aparece un agente externo y mueve un objeto de sitio, se trata del mismo objeto y sólo debe actualizar su posición. Pero, si alguien elimina o pone un objeto en su campo de visión debe borrarlo o crearlo instantáneamente en su memoria. Con el tiempo la complejidad del sistema inicial irá aumentando visiblemente.

La memoria del sistema utiliza los ficheros .xml a los que tiene acceso el robot y que representan el entorno que le rodea en el mundo real. Por tanto, si varía algo en el mundo real, este mundo sintético debe anotar tal modificación para que el robot sea plenamente consciente de los cambios.

En este documento se explica la teoría necesaria y los pasos a seguir para implementar un sistema de atención visual sencillo, obteniendo unos resultados fiables.

Finalmente, se presentan los experimentos del sistema implementado realizados sobre el robot Shelly, para observar la bondad y los fallos de los resultados obtenidos.

En el siguiente capítulo se exponen los objetivos que se han marcado para realizar la implementación del sistema de atención visual.

2. OBJETIVOS

Como objetivo principal hemos planteado estudiar, diseñar e implementar un mecanismo predictivo de atención visual para Shelly, el robot asistencial de RoboLab, con un sistema de consciencia básico con el que pueda situar un objeto y a sí mismo en el espacio y predecir hechos sencillos. Este mecanismo le permite al robot mantener un estado de preconsciencia sobre su entorno inmediato, haciéndole sensible a cualquier cambio que se produzca en él. Para ello, el robot mantendrá un modelo de su entorno con el que realizará predicciones en el futuro inmediato. El contraste de estas predicciones con la realidad percibida a través de un algoritmo basado en redes neuronales profundas, permitirá tomar decisiones sobre los cambios ocurridos.

Este gran objetivo se divide en varias partes específicas enumeradas a continuación:

- A partir de un fichero ".xml", previamente creado, que representa el mundo real de forma sintética, obtener la posición real del robot y de los objetos que lo rodean, respecto al eje de coordenadas del mundo.

- Capturar la imagen de la escena que percibe el robot a través de una cámara RGBD y procesarla para obtener los objetos que ve y su posición en el espacio.

- Comparar la escena que se percibe del mundo real con la que estaría percibiendo Shelly en el mundo sintético, para ello, ambos mundos deben ser lo más parecidos posible.

- No modificar la pose de los objetos que permanezcan en la misma posición, pero sí actualizar, en el mundo sintético, la de los objetos que hayan cambiado de posición en un intervalo de tiempo, indefinido hasta el momento. Este intervalo dependerá del tiempo de atención que consideremos suficiente tras las pruebas.

- En el caso de que un objeto que estaba en una escena anterior, desapareciera en una escena posterior, se debe actualizar el mundo sintético borrando dicho objeto.

- En el caso de que apareciera un objeto nuevo en la escena, se debe obtener su posición en el mundo real, convertirla al mundo sintético y crear un nuevo objeto en éste con la posición asociada.

- Este estado de atención mantendría al robot inmóvil, únicamente, actualizando la escena que percibe, pero no el resto del entorno. Para solucionarlo, dotamos al robot de movilidad con dos motores en el cuello:

  - Flexo-extensión: puede mover la cabeza (cámara) hacia arriba y hacia abajo.
  - Rotación: le permite mover la cabeza (cámara) hacia la derecha y hacia la izquierda.

A partir de este momento, el robot puede variar la escena de visión en un entorno cercano y, así, actualizar más partes del mundo.

- No tener en cuenta las imágenes percibidas mientras los motores del cuello de nuestro robot están rotando, pues, estas imágenes no están bien definidas y podrían introducir errores en nuestro sistema.

- Estudiar los diferentes algoritmos de actualización que podemos implantar y obtener el que más se ajuste a nuestras necesidades.

3. ESTADO DEL ARTE

La atención según la Real Academia de la Lengua Española es “la capacidad de aplicar voluntariamente el entendimiento a un objeto espiritual o sensible” y la consciencia es “la capacidad del ser humano de reconocer la realidad circundante y de relacionarse con ella”. Ambos conceptos son difíciles de definir, entender, pero sobre todo, aplicar. La consciencia como fenómeno biológico o filosófico, dependiendo del autor, existe tanto en humanos como en animales, lo experimentamos de forma natural desde muy temprana edad. Significa experiencia subjetiva, tiene contenidos, pero aunque pueda tener una enorme variedad de contenidos no puede tener muchos al mismo tiempo. No es un fenómeno pasivo como respuesta a estímulos, sino un proceso activo de interpretación y construcción de datos externos y de la memoria, relacionándolos entre sí. Los actos voluntarios y la toma de decisiones son aspectos importantes de la experiencia consciente. Por ello, uno de los significados más comunes de consciencia es que es un sistema de control ejecutivo que supervisa y coordina las actividades del organismo. Además, se ha considerado a la consciencia íntimamente relacionada con la memoria operativa, la atención y el procesamiento controlado. La memoria operativa es importante para la solución de problemas, la toma de decisiones y la iniciación de la acción. Su relación con la atención es clara: prestar atención a algo es ser consciente de ese algo. El ejemplo más clásico de atención selectiva es el conocido como “efecto cocktail party”, por el que seleccionamos información interesante en medio de un gran ruido de fondo.

William James, filósofo y profesor de psicología norteamericano, en sus Principios de Psicología describió cinco características de alto nivel de la consciencia que aún siguen vigentes:

1. Subjetividad: Todos los pensamientos son subjetivos, pertenecen a un individuo y son sólo conocidos por ese individuo. Esta subjetividad la obtenemos por el hecho de que un robot es distinto de otro y se encuentra en un espacio diferente, por tanto, su percepción del mundo es distinta.

2. Cambio: Dentro de la consciencia de cada persona, el pensamiento está siempre cambiando. El robot deberá actualizar continuamente su memoria, como ya hemos indicado, porque el conocimiento sobre el entorno que le rodea debe estar actualizado en tiempo real.

3. Intencionalidad: La consciencia es siempre de algo, apunta siempre a algo. La intencionalidad no las podemos crear con los descubrimientos realizados hasta ahora, porque un autómata es un ser no reflexivo, pero se podría simular.

4. Continuidad: James utilizó siempre la expresión "curso de la consciencia" para dar a entender que la consciencia parece ser siempre algo continuo. Muy relacionada con el cambio. Como el entorno es siempre cambiante, la consciencia del robot debe estar continuamente activa y receptiva.

5. Selectividad: Aquí James se refirió a la presencia de la atención selectiva, es decir, que en cada momento somos conscientes de sólo una parte de todos los estímulos. En cada momento, el robot seleccionará los estímulos que percibe, sencillamente, porque sólo puede recibir estímulos que entren en su campo de visión, es decir, en el campo de visión de su cámara.

A pesar de la enorme variedad de percepciones y pensamientos de naturaleza siempre cambiante, tenemos la impresión de que nuestra consciencia es algo unificado y continuo [1].

Por otro lado, el concepto de atención ha variado enormemente a lo largo del tiempo, considerándose desde un mecanismo selector de información hasta un conjunto limitado de recursos de procesamiento asignados a las distintas tareas. Sin embargo, durante los últimos años han surgido una serie de trabajos que han llevado a integrar muchos de los aspectos asociados tradicionalmente al término atención. Por ejemplo, James (1890) enfatizó la función selectiva de la atención, consistente en controlar el acceso a la consciencia de sólo aquel estímulo que ha sido atendido. Esta idea se encuentra también con mayor o menor claridad en autores del siglo XIX como Wundt, quien diferenció entre el foco y el campo de la consciencia, constituyendo el

foco la información apercibida, esto es, la atendida. No es raro, por tanto, que cuando la atención retorna con fuerza a la Psicología tras el paréntesis conductista, durante los primeros años de la década de 1950, su función selectiva sea la que ocupe el interés de los investigadores. De esta forma, se propone que la atención es necesaria debido a que el procesamiento de información es realizado por un mecanismo de capacidad limitada. Este mecanismo se colapsaría si la información accediese simultáneamente. La atención funcionaría como un filtro que deja pasar sólo un elemento cada vez. De acuerdo con este planteamiento teórico, la atención tiene un claro carácter pasivo [2].

Como podemos observar, extrapolar ambos términos a un sistema automático es una ardua tarea. Pero se pretende dar al robot asistencial la capacidad de la atención visual, lo que implica una consciencia sencilla sobre su entorno, a través de un sistema de control.

Pero ¿cuál es el objetivo último de esta mejora? Dotar a las máquinas de los mecanismos necesarios para que realicen su función, incluso cuando sufren perturbaciones no esperadas en su entorno. Un simple ejemplo de reacción sería un frigorífico que a partir de su sensor de temperatura percibe que las calorías internas son superiores a las indicadas, por alguna causa externa, por ejemplo, el usuario ha olvidado la puerta abierta, por tanto, se activa automáticamente el circuito de refrigeración [3].

En nuestro caso, una perturbación en el medio sería el desplazamiento de un objeto de una posición a otra, esto perturbaría la estabilidad de nuestro sistema, que tendría que actualizar la posición de dicho objeto.

3.1. Conceptos previos

3.1.1. Sistema de control

Un sistema de control es un conjunto de dispositivos encargado de administrar, ordenar, dirigir o regular el comportamiento de otro sistema. Está formado por tres partes básicas:

- El sensor: en nuestro caso una cámara RGBD que captura el entorno en tiempo real y permite decidir cuándo actuar.

- El actuador: es el software que modifica la memoria del robot, es decir, el software que modifica el archivo ".xml" que contiene la descripción del mundo. Además, como ya se ha comentado tendríamos los motores como actuadores secundarios en el caso de que no se perciba ningún cambio en el entorno en un determinado periodo de tiempo o que un estímulo externo nos haga moverlos.

- El controlador: propiamente dicho, que establece la estrategia a seguir en la operación del actuador en función de los valores del sensor, es decir, el software implementado, que explicaremos en capítulos posteriores.

3.1.2. Máquina de estados

Una máquina de estados es un modelo de comportamiento de un sistema con entradas y salidas, en donde las salidas dependen, no sólo de las señales de entradas actuales, sino también de las anteriores.

En esta implementación se aplica al software principal de control. Existen distintos estados según las condiciones en las que se encuentra en un determinado momento el sistema.

En nuestro caso, la máquina de estados es de tipo binaria, lo que significa que las salidas de los diferentes estados son binarias, pueden resultar como máximo dos salidas por cada estado.

3.1.3. Distancia de Manhattan

La distancia de Manhattan entre dos puntos X_1 y X_2 es la suma de las diferencias absolutas entre sus coordenadas.

$$D(\vec{X_1},\vec{X_2}) = \|X_1 - X_2\| = \sum_{i=1}^{n} |X_{1i} - X_{2i}|$$

En nuestro caso concreto, en dos dimensiones sería:

$$D(\vec{X_1},\vec{X_2}) = \|X_1 - X_2\| = \sum_{i=1}^{2} |X_{1i} - X_{2i}|$$

En nuestro sistema se utiliza para calcular el error de las coordenadas de un objeto guardadas en el sistema, respecto a las obtenidas, en un instante posterior, por YOLO.

3.1.4. Transformación de coordenadas en tres dimensiones

Las coordenadas en tres dimensiones (3D) de un punto en un sistema de referencia $R1$, pueden transformarse a coordenadas 3D en otro sistema de referencia $R2$. Aunque existen otras transformaciones geométricas, para este estudio sólo tenemos en cuenta: la traslación y la rotación [4].

Traslación

Como su nombre indica, traslada el punto de una posición en el sistema de referencia inicial $R1$ a otra en el sistema de referencia final $R2$.

La matriz de traslación 3D sería:

$$T(t_x,t_y,t_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

De forma gráfica:

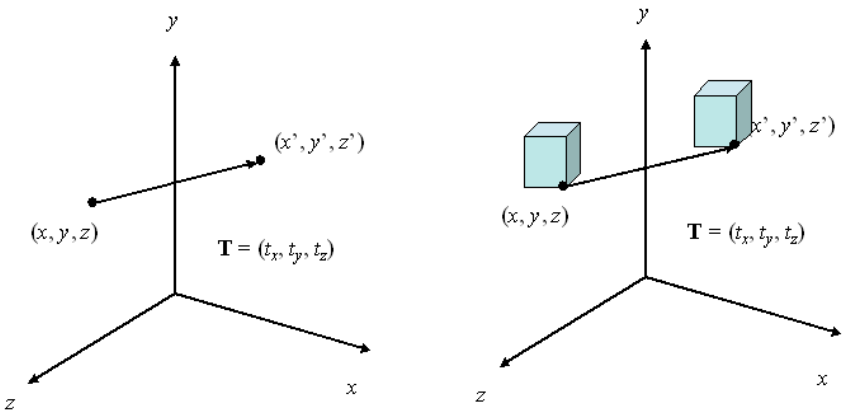


Figura 1: Traslación en tres dimensiones.

Rotación

La rotación puede definirse alrededor de cualquier recta en el espacio y, como su nombre indica, rota un punto desde $R1$ hasta $R2$. Existen tres rotaciones elementales que se definen alrededor de los ejes coordenados [5]. Las matrices correspondientes son las siguientes:

$$R_x(\theta) = \begin{bmatrix} cos\theta & sen\theta & 0 & 0 \\ -sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & -sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ sen\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & sen\theta & 0 \\ 0 & -sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ambas transformaciones se encuentran implementadas en la librería *innermodel*

de RoboComp.

3.1.5. Proyección de coordenadas 3D sobre un plano 2D

Para el desarrollo del sistema necesitamos la proyección del objeto 3D o de sus coordenadas en el plano 2D de la cámara. Esto nos permite saber si un objeto en un momento dado se encuentra en el plano de visión de la cámara o si el objeto está fuera de su alcance.

Hay dos tipos de proyección: paralela y perspectiva, en este caso, utilizamos la proyección perspectiva. En ésta, los rayos proyectores (rayos que unen cada punto del objeto 3D con el centro de proyección) tienen una dirección diferente para cada punto del objeto a proyectar y todos confluyen en el centro de proyección. Los puntos 2D son los puntos que se forman al cortar cada rayo de proyección con el plano de proyección, o plano sobre el que queremos proyectar la imagen, en nuestro caso, el plano de imagen o sensor de la cámara.

En 3D, la proyección perspectiva se obtiene de la siguiente forma. Sea $c = (c_1, c_2, c_3)$ el centro de proyección, X_1X_2 el plano de proyección y $p = (x_1, x_2, x_3) \in \mathbb{R}^3$ un punto cuya proyección en el plano X_1X_2 es $p' = (x'_1, x'_2, 0)$.

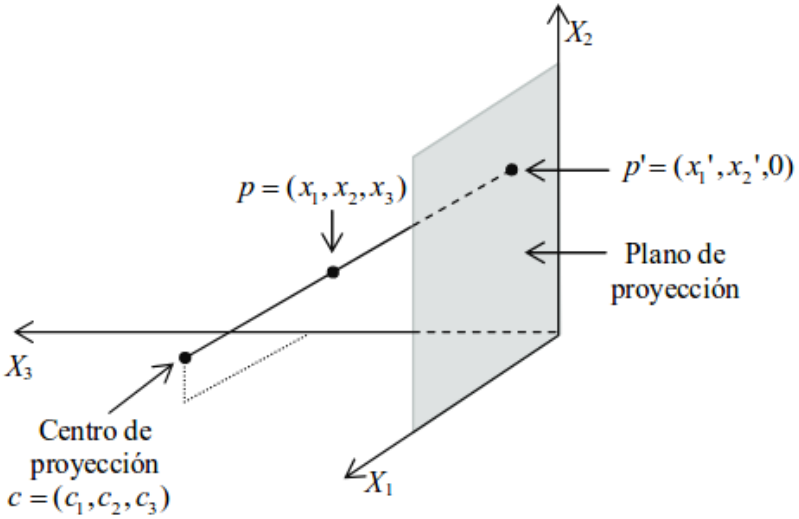


Figura 2: Proyección perspectiva de un punto p en el plano X_1X_2.

Los valores de x'_1 y x'_2 se pueden calcular haciendo uso de la ecuación paramétrica de la recta de la siguiente forma:

$$x'_1 = x_1 + t(x_1 - c_1)$$
$$x'_2 = x_2 + t(x_2 - c_2)$$
$$0 = x_3 + t(x_3 - c_3)$$

La última ecuación asegura que p' está en el plano X_1X_2, y de aquí se puede despejar el parámetro t como $t = x_3/(c_3 - x_3)$, sustituyendo este valor en el resto de ecuaciones se tiene:

$$\begin{cases} x'_1 = \frac{c_3x_1 - c_1x_3}{c_3 - x_3} \\ x'_2 = \frac{c_3x_2 - c_2x_3}{c_3 - x_3} \end{cases}$$

De esta forma, se tiene que en coordenadas homogéneas el punto p' está dado por:

$$p' = \begin{pmatrix} \frac{c_3x_1 - c_1x_3}{c_3 - x_3} & \frac{c_3x_2 - c_2x_3}{c_3 - x_3} & 0 & 1 \end{pmatrix}$$

Multiplicando por $(c_3 - x_3)$ se elimina el denominador y, así, nuevamente en coordenadas homogéneas se tiene que:

$$p' = \begin{pmatrix} c_3x_1 - c_1x_3 & c_3x_2 - c_2x_3 & 0 & c_3 - x_3 \end{pmatrix}$$

Entonces, en forma matricial utilizando coordenadas homogéneas se tiene:

$$\begin{bmatrix} x'_1 & x'_2 & x'_3 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_3 & 0 & 0 & 0 \\ 0 & c_3 & 0 & 0 \\ -c_1 & -c_2 & 0 & -1 \\ 0 & 0 & 0 & c_3 \end{bmatrix}$$

Esta proyección se encuentra implementada en la librería *innermodel* de RoboComp.

3.1.6. Tipos de movimientos del ojo humano

Existen tres tipos de movimientos oculares [6]:

- Movimientos sacádicos: se trata de un movimiento rápido del ojo de una posición a otra. En los movimientos sacádicos, los ojos se desplazan durante un intervalo de tiempo de entre 20 y 200 milisegundos ms. En esta fase, los ojos no extraen información del estímulo. A cada sácada o sacudida brusca del ojo, le sigue una fijación ocular en la que los ojos permanecen casi estáticos durante 250 ms, teniendo lugar la extracción de información en esta fase.

- Movimientos de seguimiento: una vez el objeto ha sido fijado, los movimientos de seguimiento lo mantienen en la visión foveal, tanto si se mueve él mismo como si se mueve el observador.

- Movimientos de convergencia: si cambia la distancia del objeto respecto al observador, los movimientos de convergencia lo mantienen fijado por las fóveas de ambos ojos. A medida que el objeto se acerca, los movimientos de convergencia cambian las direcciones de la mirada de ambos ojos hacia la nariz.

Para nuestro experimento, el robot sólo debe realizar movimientos sacádicos para centrar la atención primero en un objeto y luego en otro. Esto se simula con los dos motores que mueven la cámara de una posición a otra distinta.

3.1.7. Obtención de un ángulo a partir de las coordenadas de un punto

Para la rotación de los motores hacia una posición en la que la cámara detecte un objeto, utilizamos la trigonometría básica, como se puede ver en la siguiente figura:

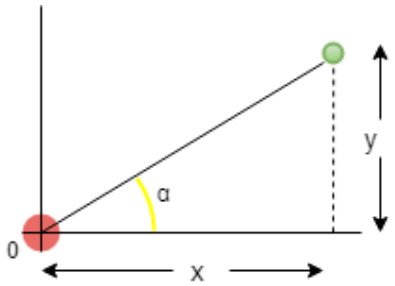


Figura 3: Ángulo que forma un punto (x, y) respecto a cero.

El ángulo se calcuaría como:

$$\alpha = atan(y/x)$$

Tomando como referencia el punto rojo (que sería la posición de los motores en cero), se calcula el ángulo de giro que debe realizar un motor para desplazar la cámara hasta que el punto verde aparezca en su escena.

Existe un pequeño error que no tenemos en cuenta, porque la cámara se encuentra situada encima de los motores, es decir, el sistema de referencia de la cámara no es exactamente el mismo que el de los motores, pero los tomamos como iguales.

3.1.8. Ecualización de una imagen

La ecualización del histograma de una imagen es una transformación que pretende obtener para una imagen un histograma con una distribución uniforme. Es decir, que exista el mismo número de píxeles para cada nivel de gris del histograma de una imagen monocroma. En teoría, la aplicación de esta operación debería transformar el histograma en otro con una forma perfectamente uniforme sobre todos los niveles de gris. Sin embargo, en la práctica esto no se va a poder conseguir pues se estaría trabajando con funciones de distribución discretas en lugar de continuas. En la transformación, todos los pixels de un mismo nivel de gris se transformarán a otro nivel de gris, y el histograma se distribuirá en todo el rango disponible separando en lo posible las ocupaciones de cada nivel.

En primer lugar, hay que obtener el histograma acumulado, para ello, utilizamos la siguiente función:

$$H(i) = \sum_{k=0}^{i} h(k)$$

En segundo lugar, igualar el histograma acumulado al modelo ideal:

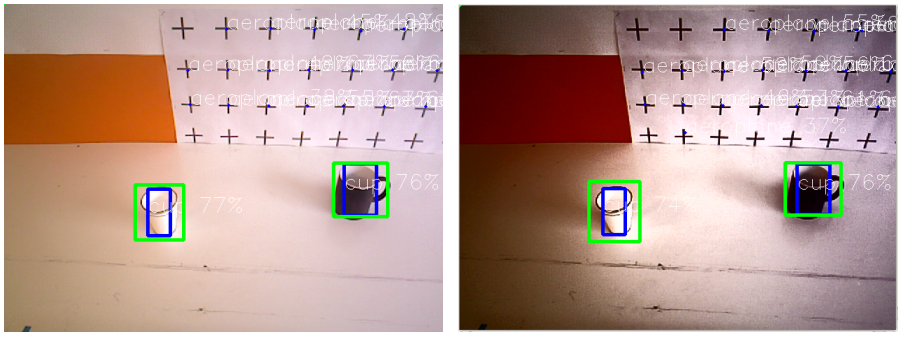
$$H(i) = G(i') = (i'+1)\frac{NM}{n}$$

Donde N y M son las dimensiones de la imagen y n el número de niveles de gris.

Por último, debemos calcular el nuevo nivel de gris i' asociado con cada nivel de gris actual i:

$$i' = \frac{n}{NM}H(i) - 1$$

El resultado de la ecualización maximiza el contraste de una imagen sin perder información de tipo estructural, como se ve en la siguientes imágenes:



(a) Sin ecualizar (b) Ecualizada

Figura 4: Imagen obtenida por el robot sin ecualizar y ecualizada.

Para ecualizar la imagen de entrada de nuestra cámara hemos utilizado las funciones de ecualización de la librería OpenCV.

3.2. Servidor YOLO

You only look once (YOLO) [7] es un sistema de detección de objetos en tiempo real de última generación. YOLO es extremadamente rápido y preciso gracias

al buen rendimiento de GPU que se obtiene con la versión para GPU de dicho programa. Además, puede fácilmente sacrificar la velocidad y la precisión simplemente cambiando el tamaño del modelo, no se requiere reentrenamiento.

Para su funcionamiento, aplica una red neuronal única a la imagen completa. Esta red divide la imagen en regiones y predice cuadros de límite y probabilidades para cada región. Estos cuadros delimitadores están ponderados por las probabilidades pronosticadas. Este modelo, tiene varias ventajas sobre otros sistemas:

- Mira la imagen completa en el momento de la prueba, de modo que sus predicciones se basan en el contexto global de la imagen.
- Hace predicciones con una única evaluación de red a diferencia de sistemas como el R-CNN (Region-based Convolutional Neural Netwoks) que requieren miles para una sola imagen. Esto lo hace extremadamente rápido, más de 1000 veces más rápido que el R-CNN.

El uso complementario de YOLO es imprescindible para esta aplicación porque nos permite detectar con suficiente exactitud dónde se encuentra un objeto y qué clase objeto es.

3.3. El robot Shelly

Shelly es el robot más avanzado con el que cuenta RoboLab actualmente. El propósito de este robot es asistencial, extiende la autonomía de personas con movilidad reducida, ya sea por algún tipo de discapacidad o por la avanzada edad, para permitirles llevar una vida independiente en sus hogares. Actualmente, el robot se encuentra en fase de pruebas en un laboratorio que emula un apartamento habitado en la Escuela Politécnica de la Universidad de Extremadura, diseñado específicamente para ello.

Su desarrollo está soportado por varios proyectos de investigación, incluyendo el proyecto del Plan Nacional de Investigación, en su convocatoria Retos de la Sociedad, TIN2015-65686-C5-5-R de nombre Fusión de las habilidades de navegación y manipulación para robots sociales en SmartHomes, por el plan de ayuda a

grupos de investigación de la Junta de Extremadura, GR15120, por la Red de Excelencia del Ministerio de Economía y Competitividad, Red de Agentes Físicos TIN2015-71693-REDT, por el proyecto de colaboración internacional España-Brasil del Ministerio de Educación, Cultura y Deporte PHBP14/00083 y por el proyecto europeo de la convocatoria POPTEC, EuroAGE.

En este momento, procederemos a realizar una descripción física de Shelly, es decir, de los componentes físicos de los que dispone.

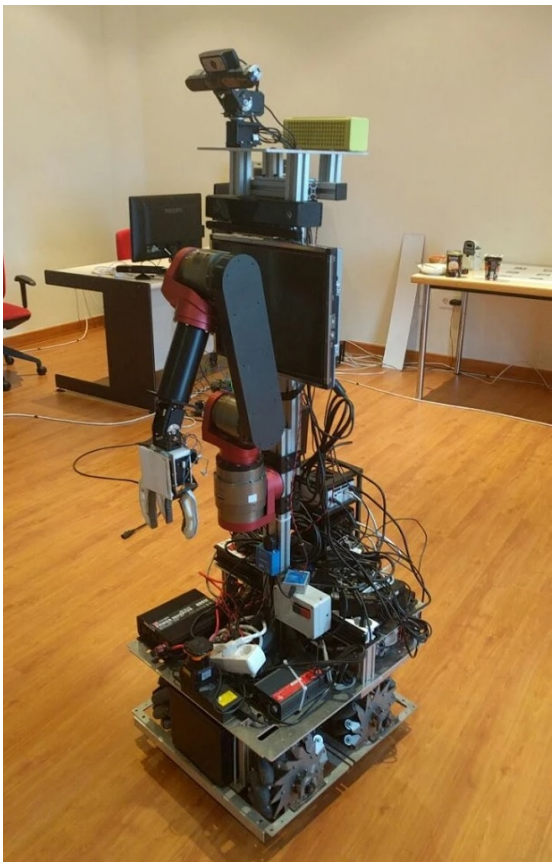


Figura 5: Imagen del robot Shelly.

Como se puede observar en la imagen 5, Shelly dispone de unas ruedas Mecanum que le permiten desplazarse en cualquier dirección o rotar sobre sí mismo. También, tiene un lidar (Laser Imaging Detection and Ranging) que le permite conocer la distancia a la que se encuentran obstáculos cercanos; un brazo robótico situado en el centro que le permite manipular objetos; una cámara ASUS Xtion Pro mediante

la que obtiene imágenes RGBD; una cámara Kinect 2 para localizar y seguir a las personas de su entorno; 4 mini-ordenadores NUC conectados entre sí a través de un conmutador sobre los cuales se ejecutan los componentes; una pantalla táctil; baterías y la electrónica de potencia y recarga.

Actualmente, está dotado con un sistema de reconocimiento y localización de objetos, pero el objetivo ahora es dotar al sistema de consciencia sobre su entorno, es decir, aunque el robot no se encuentre mirando hacia una zona de la sala, si ya la ha visto previamente o ha sido precargado un mapa de la sala, él ha de saber qué objetos existen y su situación en el mundo. Este sistema la permitirá prever cambios que le afecten para el desarrollo normal de sus tareas.

3.4. RoboComp

El framework de desarrollo para robótica RoboComp [8] está basado en la programación orientada a componentes (COP), una técnica utilizada para la implementación de sistemas software. Este tipo de programación aumenta el nivel de abstracción respecto a técnicas como la orientación a objetos, lo que mejora notablemente dos de los grandes problemas de la creación de software: la reutilización y la escalabilidad.

Un componente [9] es un programa software que forma parte de un sistema software mayor y ofrece servicios mediante una interfaz predefinida, a través de la cual, es capaz de comunicarse con otros componentes. Dicho componente será totalmente reemplazable por otro que cumpla con las interfaces declaradas.

Las características principales de un componente son:

- Ser reutilizable.
- Ser intercambiable.
- Poseer interfaces definidas.
- Ser cohesivos.

En este caso, RoboComp utiliza Ice (framework completo de llamada a procedimiento remoto con soporte para C++).

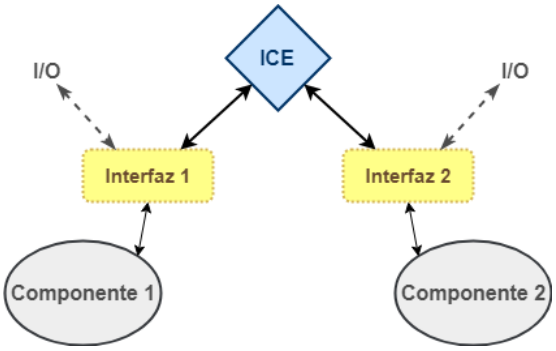


Figura 6: Representación genérica de componentes

Para realizar la conexión entre dos componentes, sólo se necesita: el nombre de la interfaz, la dirección IP de la máquina donde se aloja el componente y el número de puerto asociado.

<Nombre de la Interfaz> : <TCP / UDP> -p puerto -h host

Tras esta breve introducción a la programación orientada a componentes, procederemos a explicar Robocomp.

Robocomp, es un framework de desarrollo para robótica de código abierto que ofrece la posibilidad de crear componentes de una manera fácil y sencilla, comunicándose éstos a través de interfaces públicas. Las comunicaciones, como he explicado anteriormente, se realizan mediante el middleware Ice. Para generar un componente de RoboComp se utiliza un lenguaje de dominio específico, llamado CDSL, un ejemplo de este lenguaje sería:

```
import "import1.idsl";
import "import2.idsl";
Component ComponenteName
{
  Communications
  {
    implements interfaceName;
```

```
    requires otherName;
    subscribesTo topicToSubscribeTo;
    publishes topicToPublish;
  };
  language Cpp/Python;
  gui Qt(QWidget);
};
```

De esta manera, un componente generado con RoboComp puede suscribirse o publicar un tópico e implementar o hacer uso de una interfaz, mediante los lenguajes de programación Python o C++.

Los componentes generados con RoboComp tienen un archivo de configuración en el que se puede configurar el puerto y la dirección IP de una interfaz implementada en otro componente, el tamaño de los mensajes Ice, añadir variables de configuración, etc. Además, RoboComp cuenta con otro lenguaje específico de dominio para la definición de interfaces, IDSL, que genera los archivos .idsl. Gracias al framework de Ice, los componentes escritos con RoboComp pueden comunicarse con componentes escritos en otros lenguajes, como Java, PHP, etc. RoboComp utiliza otras herramientas como CMake, Qt4, Qt5, IPP, OpenScenegraph y OpenGL. También cuenta con un simulador.

El objetivo de RoboComp es lograr una mayor eficiencia, simplicidad y capacidad de reutilización de esos componentes. Todos los componentes que controlan a Shelly han sido creados utilizando RoboComp.

3.5. Componentes previos de los que disponemos

En nuestro sistema aparecen actuadores físicos como los dos motores situados en el cuello del robot que permiten los movimientos de la cámara hacia los lados, hacia arriba y hacia abajo. Para trabajar con ellos, utilizamos el componente *dynamixel*, que tiene todo lo necesario para conectar nuestro programa con los motores físicos y nos permite controlarlos.

Además, tenemos el sensor principal, la cámara RGBD, para la que existe otro

componente llamado *rgbd*. Éste nos permite obtener la capturas de la cámara en tiempo real y en un formato útil para poder procesarlas.

A continuación, para procesar las imágenes y extraer todo la información que vamos a utilizar, se necesita el servidor YOLO. Éste se despliega iniciando el componente *yoloServer*, que inicia el servidor y lo conecta al sistema.

Para emular la "memoria" nos apoyamos en archivos .xml de la clase *innermodel*, que nos proporciona la información del mundo simulado. Este mundo simulado representa el laboratorio real dónde se encuentra el robot, el apartamento de investigación. Esta simulación se carga a través del componente *rcis* que carga el archivo "autonomyLab.xml" que representa el apartamento, éste a su vez, carga el fichero de la base del robot. Para desplegar el componente se puede utilizar *faulhaber* o *base* en el *rcManager*, como se mostrará a continuación. Así, obtenemos la simulación de la base del robot dentro del apartamento de investigación, lo que nos permite trabajar con el mundo real.

Innermodel es un sistema de representación interno basado en ficheros .xml. Estos ficheros almacenan árboles de transformaciones de coordenadas, planos y mallas que, en conjunto, forman el mundo o la simulación con la que vamos a trabajar. Para ver la evolución del entorno en tiempo real, utilizamos un componente llamado *innerviewer*. Éste es capaz de representar gráficamente un fichero .xml asociado de forma continua. Cada vez que se producen cambios en el fichero de memoria del sistema se muestra gráficamente, lo que nos permite una mejor comprensión de lo que el robot tiene almacenado internamente. Dicho componente, no hace falta desplegarlo dentro del sistema, nuestro componente *ObjectDetection*, crea una instancia de este tipo y lo utiliza directamente.

El componente *ObjectDetection* es el componente creado por mí para añadir atención visual al robot. Se explicará posteriormente.

Para finalizar, existen tres componentes principales que nos permiten el funcionamiento de todo el sistema:

- *commonJoint*: es el componente central que conecta los motores reales con los

simulados, y, a través de ellos, el sistema con la simulación, lo que permite que el programa funcione correctamente. Si alguno de los componentes anteriores no se ha ejecutado o iniciado bien, este componente no puede funcionar.

- *rcnode(IS)*: es el componente que permite la comunicación entre todas las interfaces de los componentes y, al fin y al cabo, de los componentes en sí. Es el encargado de inicializar el middleware Ice.

- *rcRemote*: administra y monitoriza la ejecución organizada de la red de componentes del robot. Es el primero que se debe iniciar y no depende de ningún otro componente. El conjunto de herramientas rcremote se compone de dos herramientas: rcremoteserver y rcremote, servidor y cliente, respectivamente. El servidor se ejecuta en cada una de las computadoras que tiene el robot y el cliente debe ser invocado por rcmanager. Cuando un comando se ejecuta de forma remota utilizando rcremote, el servidor utiliza una pestaña en "yakuake" (terminal de Ubuntu) para cada uno de los componentes ejecutados, de modo que los desarrolladores puedan ver el resultado textual. Como el servidor se ejecuta localmente, no hay limitaciones con respecto a la salida gráfica.

Resumiendo, como se observa en el esquema de la figura 7, la conexión entre los distintos componentes es la siguiente:

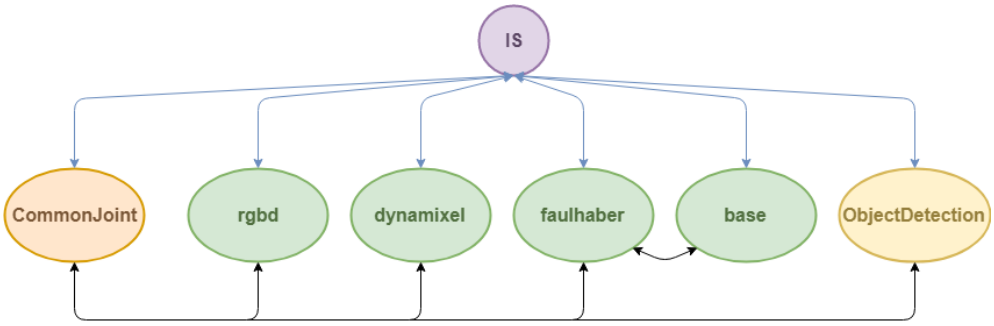


Figura 7: Gráfico de comunicaciones del sistema completo.

Además, hay tres componentes o herramientas que han sido muy útiles para el desarrollo del sistema:

- *rcManagerSimple*: es una herramienta sencilla que te permite ver el grafo de dependencias del sistema y, así, desplegarlo fácilmente. Como se puede imaginar, teniendo en cuenta la cantidad de componentes relatados anteriormente, el despliegue completo de uno en uno es un poco lento. Gracias a este componente podemos controlarlos de forma más rápida y clara. El utilizado en este desarrollo muestra este grafo de dependencias en su interfaz gráfica:

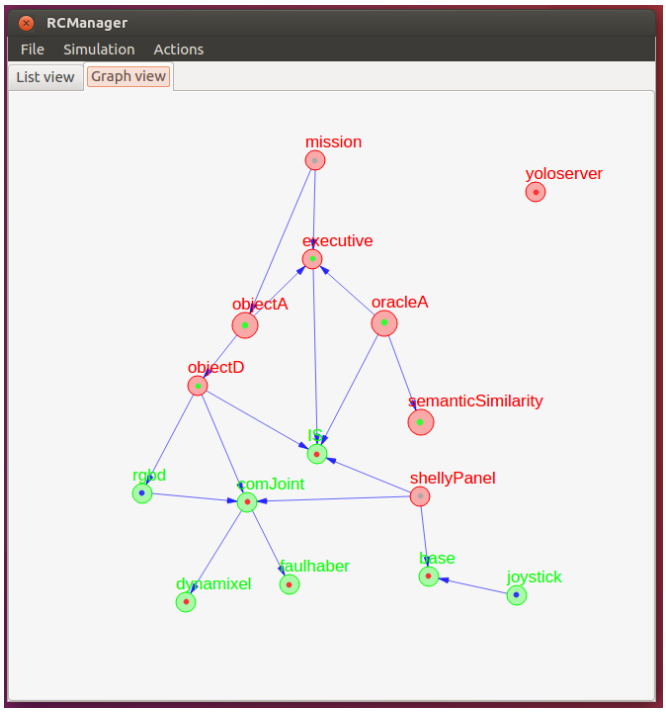


Figura 8: Interfaz gráfica del componente *rcManagerSimple*.

Te permite conocer, además, si los componentes están activos (verde) o no (rojo).

- *rcMonitor*: es una herramienta gráfica muy útil que, dependiendo de la interfaz a la que te conectes, te permite mover los motores simulados, los reales o ambos a la vez. Esta herramienta carga todos los motores que estén utilizando un puerto determinado y los muestra en su interfaz con su posición actual. Podemos cambiar el valor de esta posición y la velocidad a la que se desplazan los motores para realizar pruebas de los movimientos de la cámara.

- *joystick*: este componente conecta un joystick físico para poder manipular la base

del robot simulada dentro del apartamento simulado. Es muy útil porque, para el funcionamiento correcto del sistema, el mundo real y el simulado que observa el robot debe ser lo más parecido posible al iniciarlo. Éste nos permite situar al robot delante de una mesa respecto a la que hemos realizado las pruebas.

4. UN SISTEMA DE ATENCIÓN VISUAL Y CONSCIENCIA SOBRE EL ENTORNO PARA UN ROBOT

Hasta ahora, conocemos la situación actual de Shelly, tanto física como internamente, y, también, qué es y para qué sirve RoboComp. En esta sección, explicaremos el sistema de atención, consciencia y percepción de objetos creado. Como se ha explicado en la sección anterior, es un componente llamado *ObjectDetection*.

En RoboLab (laboratorio de investigación de robótica de la Universidad de Extremadura) se ha trabajado anteriormente en detección de objetos, por tanto, ya se dispone de un componente llamado *ObjectDetection* en el que me he basado para crear mi nuevo componente.

A continuación, describiré de forma más precisa la creación del componente citado y la del sistema completo para la correcta ejecución.

4.1. Planteamiento inicial

Inicialmente, la idea fue crear una máquina de estados que fuese leyendo y actualizando un archivo del *innermodel*. De esta forma, conseguíamos una "memoria" actualizada continuamente en la que basar nuestro desarrollo. El planteamiento era el siguiente: el robot debe ser capaz de saber en todo momento qué le rodea e ir actualizando su memoria conforme surjan cambios.

En este caso, todos los cambios son un estímulo externo para que el robot actualice su memoria. Esta memoria, como ya hemos dicho, es un fichero .xml que tiene una representación inicial del mundo que rodea al robot y se carga al iniciar el programa. Para que funcione correctamente, la posición del robot en el mundo real y en el simulado debe ser lo más parecida posible, así que, con el joystick movemos el robot simulado a dónde nos convenga o iniciamos el robot en la posición adecuada para las

pruebas. Para facilitar la implementación inicial, sólo tenemos en cuenta los objetos de tipo taza o vaso que se encuentran situados sobre una mesa del apartamento.

Los estados que conforman la máquina de estados son:

- *Predict*: estado inicial que captura la imagen y comprueba hacia dónde está mirando la cámara para comparar el mundo real y el sintético más adelante. Pasa al estado *YoloInit* al acabar su tarea.

- *YoloInit*: estado intermedio que envía la imagen capturada al servidor de Yolo para que la procese y obtener la información de ella. Pasa al estado *YoloWait*.

- *YoloWait*: estado intermedio que obtiene la imagen ya procesada de Yolo y, así, etiqueta los distintos objetos que aparecen en la escena. Pasa al estado *Compare*.

- *Compare*: compara lo obtenido en el primer estado de lectura del mundo sintético con lo obtenido por el servidor Yolo, es decir, compara el mundo simulado con el real. Pasa al estado *Stress*.

- *Stress*: si no se ha producido ningún cambio, entre ambos mundos, no hace nada. Pero, si, por el contrario, ha habido algún cambio en la posición de un objeto o algún cambio, en general, se actualiza el fichero .xml con las novedades percibidas. Pasa al estado *Predict*.

- *Moving*: es un estado especial al que se puede saltar desde cualquier otro estado. Para esto, el hilo principal comprueba continuamente si los motores del "cuello" de nuestro robot se están moviendo, en caso afirmativo, se pasa a estado *Moving* y se mantiene en él hasta que termine el movimiento de los motores. A partir de él, la máquina de estados pasa siempre a *Predict* porque, al acabar el desplazamiento de la cámara, el robot tiene que recalcular hacia donde está mirando para poder obtener los datos correctamente.

En la siguiente figura, se muestra una representación sencilla de la máquina de estados que controla el flujo del programa.

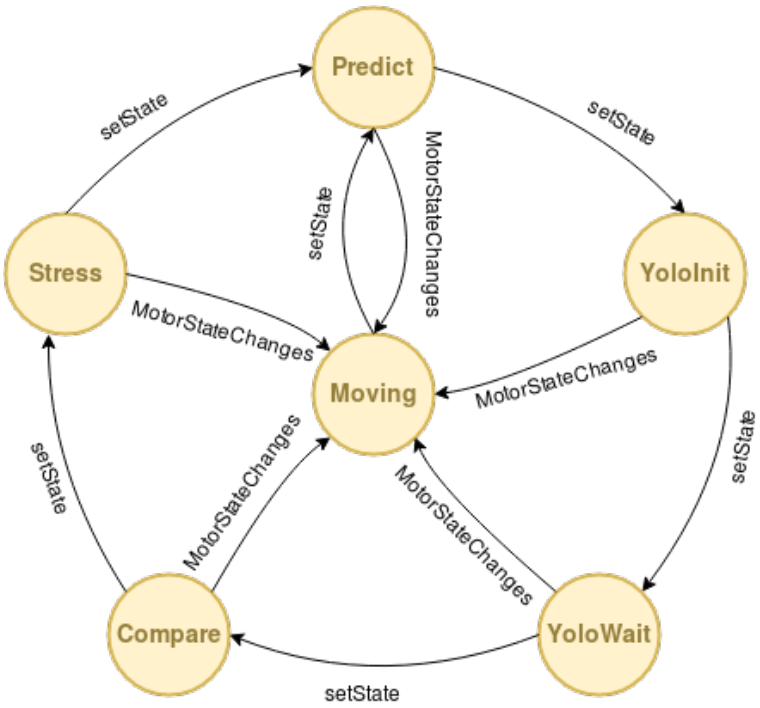


Figura 9: Diagrama de la máquina de estados del componente ObjectDetection.

4.2. Tratamiento de la información

4.2.1. Obtención de la información del *innermodel*

El *innermodel*, como ya se ha explicado, es el sistema interno de representación que tiene RoboComp para la simulación de entornos. En nuestro caso, debemos cargar o crear un entorno que simule el mundo real donde se encuentra el robot en el momento de las pruebas. Este fichero, dónde se encuentran los datos de la simulación, es de tipo .xml y está formado por sentencias como la siguiente:

<transform id = "objName" tx = "6.2" ty="54.8" tz="33.5" rx="1.0" ry="0.054" rz="0.033" >

Almacena el nombre del objeto y las transformaciones de coordenadas respecto al sistema de coordenadas del padre. En nuestro caso, el sistema de coordenadas principal es el del mundo, su hijo es el sistema de la mesa en la que realicemos las pruebas y, a su vez, su hijo sería el objeto que ponemos encima de esa mesa con su respectivo eje de coordenadas. En el sistema de coordenadas del mundo, el eje x con el z forman el plano

del suelo de la sala y el eje *y* es el eje vertical. A su vez, el sistema de coordenadas de la mesa de pruebas se encuentra centrado sobre la mesa con sus ejes alineados con los del mundo, *x* y *z* formando el plano del tablero de la mesa e *y* el eje vertical.

Este sistema interno de representación tiene implementadas algunas funciones muy útiles para nuestro desarrollo, como son: *getNode*, devuelve un nodo de este árbol; *getRotationMatrixTo*, devuelve la matriz de rotación de un sistema respecto a otro; *getTranslationVectorTo*, devuelve el vector de traslación de un objeto; *getCamera*, devuelve la cámara que pidamos del sistema; *transform* transforma las coordenadas de un sistema de referencia a otro, etc.

4.2.2. Información a almacenar

Llegados a este punto, sabemos que debemos almacenar los objetos, pero ¿qué información exactamente?.

Tras estudiar detenidamente la información que tenemos y que necesitamos, por cada objeto debemos almacenar: el tipo y el nombre del objeto, su identificador, su cubo delimitador o bounding-box, la proyección del bounding-box sobre la cámara, las coordenadas del objeto, un contador para saber cuánto tiempo lleva la cámara sin detectar el objeto y dos booleanos necesarios para el procesamiento del sistema. Todos estos datos se encuentran dentro de una estructura llamada *TObject* y se detallarán sus funciones más adelante.

El cubo delimitador es un vector de coordenadas que definen al cubo que contiene al objeto. Como en este sistema, por ahora, sólo trabajamos con tazas y vasos, el cubo delimitador no se calcula automáticamente y, por defecto, se toma un cubo de dimensiones 80x80x80 mm, lo suficientemente grande para que la imagen del vaso quepa completamente.

Como la estructura de estos ficheros .xml no nos permite almacenar toda la información que necesitamos para la implementación del sistema, utilizamos vectores de la clase *std::vector* que almacenan los objetos de tipo *TObject* durante la ejecución.

4.3. Desarrollo

Para crear una consciencia artificial, lo primero que necesitamos es la ”memoria”, la base en dónde vamos a apoyar toda la implementación posterior. Como ya hemos dicho, en el caso ideal, esta memoria sería un fichero .xml en el que guardar los datos durante la ejecución, pero, como todos estos datos no son compatibles con nuestro fichero, hemos realizado un híbrido entre este fichero .xml y los cinco listas auxiliares de tipo *std::vector*:

- *listObjects*: lista de objetos totales que el robot controla como existentes en el mundo.

- *listYoloObjects*: lista de objetos que detecta el servidor de Yolo tras analizar la imagen capturada por la cámara en un momento determinado.

- *listCreate*: lista de objetos nuevos que aparecen en una captura, ya sea porque, anteriormente, el servidor de Yolo no los había detectado o no con una probabilidad superior al 35 %, o porque alguien externo ha puesto sobre el campo de visión del robot un nuevo objeto. Estos objetos se crearán en el momento oportuno.

- *listDelete*: lista de objetos que, en alguna captura previa, existían y ya no. Puede ser debido a que el servidor Yolo detecte un objeto donde realmente no existía, a que alguien quite un objeto del campo de visión de la cámara o a que la memoria que hemos cargado al iniciar no estaba completamente actualizada.

- *listVisible*:lista de objetos visibles en un momento determinado, es decir, los objetos que en un instante entran completos en el campo de visión de la cámara. Si, por ejemplo, se ve sólo una parte del objeto en la captura, éste no se toma como visible.

4.3.1. *Predict*

Este estado inicial recorre la lista de completa de objetos *listObjects* y, por cada uno de ellos, comprueba si se encuentra dentro de los límites de la captura de la cámara, es decir, si el robot lo vería en ese momento. Si se cumple esta condición, se crea la caja que se imprime posteriormente sobre la imagen en color verde, como se muestra en la figura 10 y, además, se añade el objeto a la lista de objetos visibles.

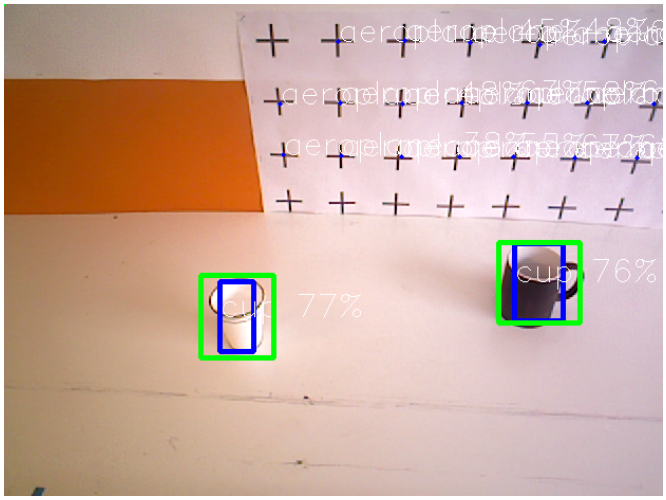


Figura 10: Captura del componente ObjectDetection.

Para realizar esta comprobación, obtenemos del *innermodel* la cámara sobre la que queremos trabajar, después, por cada vector de coordenadas que forma el bounding-box del objeto, se transforma del sistema de referencia del objeto al de la cámara y se proyecta sobre la cámara. En código:

```
QVec res = c->project(innermodel->transform("rgbd", o.bb.at(i), o.name));
```

Si las coordenadas que obtenemos (res) están delimitadas entre 0 y 640, en el caso de la *x*, y 0 y 480, en el caso de la *y*, este objeto se encuentra en la escena de visión en el momento en el que se está calculando.

Como se puede ver, lo que realiza realmente este estado introductorio, es un filtro entre los objetos que va a procesar o tener en cuenta el sistema en la siguiente iteración de la máquina de estados y los que no, es decir, los objetos que entran dentro del plano

de visión del robot y los que no. Estos objetos, son sintéticos, es decir, son los objetos que el robot tenía almacenados en su "memoria" como válidos y bien posicionados, pero, a continuación, debemos comprobar si esta situación es real o no.

4.3.2. *YoloInit*

En este estado sólo se envía la imagen capturada de 640x480 píxeles al servidor de Yolo para procesarla. Si el servidor de Yolo funciona correctamente, la imagen se envía a través del proxy. Si no funciona, por cualquier razón, el estado devolvería una excepción, pero no se pararía el programa.

4.3.3. *YoloWait*

Si el servidor de Yolo está funcionando correctamente, se obtienen los resultados del procesado de la imagen, se vacía la lista *listYoloObjects* y, por cada objeto obtenido a partir del servidor Yolo, se crea una instancia de tipo *TObject* que se añade a la lista de objetos de Yolo. Estos objetos son los que la cámara realmente está viendo en el momento de la captura. Yolo, por cada objeto, nos devuelve el tipo de objeto, las coordenadas (x, y) de la esquina superior izquierda, el ancho y el alto y la probabilidad de ser del tipo de objeto que ha sido detectado.

Si el servidor no funcionase, el programa devuelve una excepción pero no se detiene.

4.3.4. *Compare*

Al principio, se vacían las listas *listCreate* y *listDelete*. A continuación, por cada objeto de la lista de visibles (objetos sintéticos), se crea una lista de candidatos llamada *listCandidates*. Estos candidatos son de tipo *TCandidate*, una estructura que almacena un número real con el área del candidato, un *QPoint* con el error y un puntero al objeto de tipo *TObject* que se está tratando. A su vez, por cada objeto de la lista del servidor de Yolo (objetos reales), si el objeto es del mismo tipo, en nuestro caso "cup", y todavía

no ha sido asignado a ninguno de los objetos sintéticos, se procesa. Este procesamiento consiste en:

1. Se crea un rectángulo de tipo *QRect* con las coordenadas del objeto real.
2. Se crea un rectángulo de tipo *QRect* con las coordenadas del objeto sintético.
3. Se calcula la intersección entre ambos rectángulos y el área de esta intersección. La intersección se obtiene aplicando directamente la función *intersected* de la clase *QRect*, que devuelve un rectángulo formado por la intersección de los dos anteriores. A partir de este rectángulo, podemos hallar fácilmente su área multiplicando su ancho por su alto.
4. Se calcula el vector de desplazamiento entre los dos rectángulos, restando el centro de ambos.
5. Si el área de la intersección es mayor que cero y el vector de desplazamiento, medido mediante la distancia de Manhattan, es menor que el doble del ancho del rectángulo del objeto sintético, se crea un nuevo candidato que se añade a la lista de objetos candidatos ordenándolo de mayor área de intersección a menor.

Tras finalizar este proceso, si el objeto sintético tiene algún candidato, un objeto real que se encuentre muy cerca de su posición, se toma el primer objeto real candidato y se marca el sintético como asignado y como explicado.

Además, si sobra algún objeto visible sin explicar, este objeto sintético debe ser borrado porque, por alguna razón, ha desaparecido en esta iteración de la captura de la cámara. Este borrado se realiza posteriormente, por ahora, sólo se añade el objeto sintético a la lista *listDelete*.

Por último, si queda algún objeto en la lista de Yolo sin asignar, significa que hay un objeto que el sistema no contemplaba, pero ha aparecido en la escena. Entonces, se crea una nueva instancia *TObject* con el tipo de objeto y las coordenadas devueltas por el servidor de Yolo. Finalmente, este nuevo objeto se añade a la lista de creación de objetos *listCreate*.

4.3.5. *Stress*

En este estado, se realizan tres procesos claramente diferenciados: actualización de la posición los objetos previamente existentes, borrado de los objetos que ya no se encuentran en la escena y creación de los objetos que han aparecido por primera vez en la escena.

Respecto a la actualización de la posición de objetos ya existentes, se toma la lista de objetos visibles o sintéticos y, por cada objeto, si está explicado, es decir, si hay un objeto en el mundo real semejante a él y en una posición parecida, se actualiza el *innermodel* o la "memoria" del robot con las coordenadas obtenidas en esta iteración por el servidor de Yolo. Antes de actualizarlas hay que obtener el sistema de referencia que se quiere actualizar y se actualiza después.

En cuanto al borrado de objetos, se recorre la lista de objetos a borrar (*listDelete*) y, por cada objeto de tipo "cup", se elimina el nodo referente a este objeto del árbol del .xml. También, se borra de la lista de objetos visibles en este instante y de la de objetos totales del mundo.

Por último, para crear nuevos objetos, se comprueba que no se supera el máximo de objetos que puede mantener el robot en memoria (en este caso, sería cinco, pero es un parámetro que se puede ajustar). Si se puede añadir un nuevo objeto, se recorre la lista de objetos a crear (*listCreate*), por cada objeto, se obtiene la transformación de coordenadas de la posición del objeto según Yolo en la cámara respecto a la mesa sobre la que se encuentra, se obtiene un identificador propio que no esté previamente cogido por otro objeto y se crea el nuevo objeto. Para la transformación anterior utilizamos el siguiente código:

```
QVec ot = innermodel->transform("countertopA", n.pose, "rgbd");
```

Dónde *pose* es el vector de coordenadas de la posición del objeto, *"countertopA"* es el nombre de la mesa en el *innermodel* y *"rgbd"* es el nombre de la cámara en el *innermodel*.

Este nuevo objeto de tipo *TObject* no se puede añadir al .xml, pero sus coordenadas

sí, como una nueva transformación que cuelga de la tranformación de la mesa sobre la que se encuentre el objeto en el árbol de representación interna:

```
innermodel->newTransform(to.name, "", innermodel->getNode("countertopA"), ot.x
```

Dónde *to.name* es el nombre del nuevo objeto, el nodo padre es el devuelto por *getNode* y x, y y z son las coordenadas del nuevo objeto.

Además, se crea un bounding-box genérico como dijimos al principio de 80x80x80 mm y se proyecta sobre la cámara, si esta proyección resulta fuera de los límites de la captura, debe borrarse el nodo del árbol del .xml porque no es completamente visible. Esta comprobación es muy importante porque, si no, como en los bordes de la imagen capturada el servidor Yolo suele fallar más, se producen errores en nuestro sistema. Finalmente, si el objeto es plenamente visible y se ha creado correctamente, se añade a las listas de objetos visibles y de objetos totales del mundo.

4.3.6. Inicio de sistema

El sistema se inicia al levantar el componente *ObjectDetection* tras tener todos los demás iniciados. Al comenzar, realiza una serie de pasos:

- Lee el fichero del *innermodel* y lo carga en el programa para usarlo como “memoria”.

- Crea la instancia de *innerviewer* a partir del fichero anterior para que, durante la ejecución, se muestre gráficamente cómo varía el fichero .xml.

- Se leen las posiciones de las tazas del *innermodel* y se almacenan en objetos de tipo ”TObject”. Se rellenan el resto de campos de este tipode objeto con valores por defecto.

- Finalmente, se posicionan los motores en (0,0) como punto de partida, para que, inicialmente el robot esté siempre mirando en la misma dirección.

4.3.7. Hilo principal

El hilo principal de ejecución ejecuta continuamente la máquina de estados mientras no se produzca ningún error. Pero, antes de comenzar la ejecución del estado en el que se encuentre, realiza algunos cálculos:

1. Obtiene de la cámara RGBD, el sensor del sistema, la matriz de la imagen en color y la matriz de distancias. Esto es posible gracias a que la cámara es de tipo RGBD y, por tanto, obtiene a la vez la imagen en color, como la distancia a cada objeto de los que aparece en la escena. En realidad, obtiene cuatro canales por cada píxel: nivel de rojo (R), nivel de verde (G), nivel de azul (B) y distancia del pixel al punto en la realidad (D).

2. Se actualiza el *innermodel* con los valores de posicionamiento del robot en el mundo obtenidos en la iteración anterior, en caso de ser la primera iteración, se mantendrían igual. También se actualiza el estado de los motores del cuello del robot. Para actualizar se utilizan las funciones del *innermodel* *updateTransformValues* y *updateJointValue* respectivamente.

3. Se ecualiza la imagen de la cámara y se dibujan los rectángulos pertenecientes a los objetos obtenidos en la iteración anterior, tanto por el servidor de Yolo, como por nuestro sistema. En el caso de Yolo se dibuja un cuadro de color azul si su probabilidad es superior a 35 y en nuestro sistema de color verde.

4. Se comprueba si alguno de los objetos lleva más de 15 segundos sin ser visto y si los motores no se acaban de mover. Si esto se cumple, se calcula el ángulo que debe desplazarse cada motor (apartado 3.1.7) para que, el objeto que lleva más de 15 segundos sin ser visto, entre en el campo de visión del robot; se reinicia el contador de todos los objetos del sistema y se mueven los motores hasta la posición calculada. Esta posición es absoluta desde la posición cero de ambos motores, así, el cálculo es más sencillo y el movimiento es directo hacia una posición. Cuando comience la nueva iteración de la máquina de estados,

se comprobará si el objeto ya es visible para el robot. Para mover los motores utilizamos los siguientes métodos del componente *jointMotor*:

```
jointmotor_proxy->setPosition(head_yaw_joint);
jointmotor_proxy->setPosition(head_pitch_joint);
```

5. Se calcula la diferencia de posición entre la posición actual de los motores y la de la iteración anterior. Si esta diferencia es significativa, se pasa al estado *Moving* de la máquina de estados. Es por esto que, como se ha explicado anteriormente, desde cualquier otro estado se puede pasar al estado *Moving*, pues sólo depende de si un objeto lleva un determinado intervalo de tiempo sin ser observado.

4.3.8. *Moving*

Este estado, como hemos explicado con anterioridad, es sólo un estado auxiliar en el que se mantiene la máquina de estados cuando los motores del cuello del robot se están moviendo. Es necesario porque permite dejar de procesar lo que se captura durante el movimiento: imágenes distorsionadas que añaden errores a nuestro sistema. Es la forma de mantener al sistema activo pero sin procesar, hasta que la captura de la cámara sea de nuevo estable. Su única funcionalidad es pasar al estado *Predict*. Si el motor siguiese en movimiento en la siguiente iteración, la máquina de estados pasaría directamente a *Moving*, porque la comprobación del movimiento de los motores es previa a la ejecución de cualquier estado. Después volvería a pasar a *Predict* hasta que los motores finalicen su movimiento.

4.4. Primera mejora: mapa térmico

Para que el movimiento de la cámara sea más correcto, he modificado el planteamiento inicial de tener un contador por cada objeto. En su lugar, he añadido un mapa calorífico que es el que controla el movimiento de la cámara de un sitio a otro de la mesa, de forma que no deje zonas sin visitar, o sólo visite zonas donde hay o ha habido algún objeto.

Este mapa está planteado de la siguiente manera:

- El plano de control de objetos se reduce a la mesa en la que esté enfocando su atención el robot al iniciar el programa.
- Se divide este plano en celdas de 40x40 en coordenadas de la mesa. Es decir se plantea una cuadrícula sobre la que se va a ir modificando el valor de temperatura como si de un mapa térmico se tratara.
- Cada celda de las anteriores, tiene un valor inicial de temperatura a cero que se irá incrementando o decrementando dependiendo de la situación de los objetos sobre la mesa en cada momento y de la posición de la cámara.
- El mapa se enfriará continuamente con una temperatura de descenso constante homogénea para todo el mapa, excepto, para las zonas dónde se encuentre un objeto. Las celdas que toque un objeto, se enfriarán más rápidamente pues son las que requieren, en principio, mayor atención.
- El mapa se calentará dependiendo de la zona dónde esté enfocada la cámara con una función de Gauss. Esto quiere decir que, si una celda entra dentro de la proyección de la cámara en un determinado momento, esta celda verá incrementada su temperatura en función de su posición en la imagen. Si la celda se encuentra en un borde se calentará menos que si se encontrase en la parte central de la imagen que se calentaría más.
- La cámara debe moverse siempre hacia las zonas más frías de la mesa para que, así, no deje ninguna parte del plano de la mesa sin visitar.

Para crear el mapa, he utilizado la clase $std :: unordered\_map < Key, Value, KeyHasher >$, dónde Key es un par que almacena las coordenadas (x, y) de la esquina superior izquierda de cada celda en la que se divide la mesa; $Value$ es una estructura que almacena las coordenadas (x, y, z) de la celda y su temperatura; y $KeyHasher$ es el código hash utilizado para indexar el mapa.

Inicialmente, se debe rellenar el mapa para saber la posición de cada celda y, posteriormente, poder procesarlas. Se inicia dividiendo la mesa en celdas de 40x40 mm y almacenando las coordenadas de su esquina superior izquierda y su temperatura a cero.

El método del hilo principal que decidía si la cámara se movía según los temporizadores, debe ser modificado para que utilice el mapa térmico como guía para el movimiento. En este momento, se recorren todas las celdas del mapa de la mesa a la que se está prestando atención y se calcula la que tenga la mínima temperatura. Cuando la temperatura de la celda con menor temperatura es menor que un umbral fijado, se calculan los ángulos de giro que necesita la cámara para apuntar a dicha celda (3.1.7). De esta manera, vamos recorriendo todas las zonas de la mesa actualizando el *innermodel*. A su vez, se va mostrando el la simulación del *innerviewer* lo que se va añadiendo o borrando en el .xml.

Para mover la cámara dependiendo del mapa he creado tres métodos:

1. *getMotorState()*: toma la posición de los motores, llama al método *findPointAttention()* y, después, al método *centerAttention()*. Si esta posición varía respecto a la de la iteración anterior, pasa al estado *Moving*.

2. *findPointAttention()*: Es el método que realmente recorre el mapa buscando la menor temperatura. Cuando la encuentra y es menor que un umbral fijado, debe modificar un atributo del componente llamado *posAttention*, que es el encargado de almacenar el todo momento la posición hacia la que debe mirar el robot en coordenadas del mundo.

3. *centerAttention()*: centra la atención del robot, es decir, modifica la posición de los motores todo lo que sea necesario para que la zona con menor temperatura, o dicho de otra manera, *posAttention*, esté siempre centrado en la imagen que captura el robot a través de la cámara.

La función de Gauss de calentamiento se ha simulado con una implementación que calcula dónde se encuentra cada celda de la mesa perceptible en ese instante por

la cámara y según su posición respecto a la proyección en el plano de la cámara se calienta más o menos. Visualmente, esta función se mostraría como en la siguiente imagen:

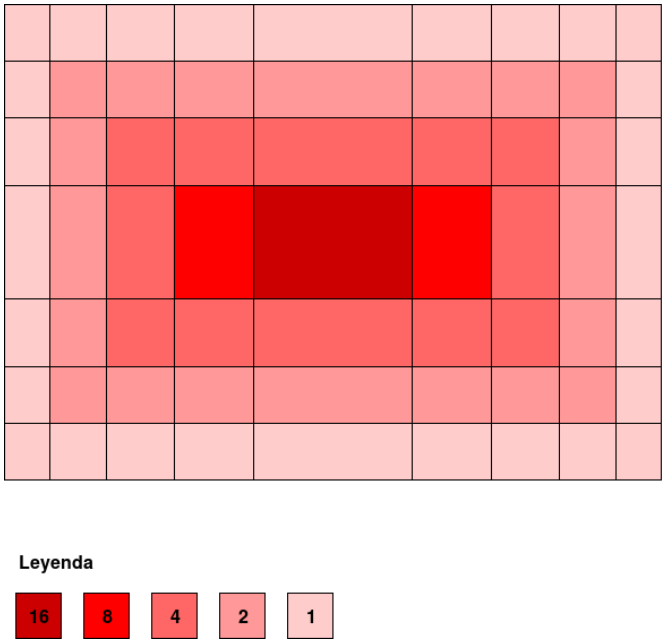


Figura 11: Filtro de imagen para la función de calentamiento.

Sobre la captura de la cámara se proyectarían las diferentes celdas de la cuadrícula de la mesa que se observa en ese momento y, según la posición dentro del filtro en la que estuviesen, la temperatura aumentaría de una forma u otra. En la parte central, como se puede apreciar, se calentarían las celdas con mayor intensidad que en los bordes, de forma exponencial.

Para enfriar, necesitamos tomar la posición de cada objeto sobre la mesa y calcular según sus dimensiones (80x80x80) qué celdas del mapa calorífico estaría tocando. Dichas celdas serían las que se enfriarían a mayor velocidad, aunque el resto del mapa, se enfriaría, como ya he dicho, de forma constante hasta un umbral fijado.

Estas variaciones en la temperatura son las que finalmente sigue el algoritmo del sistema que mueve la posición de los motores. La cámara siempre se desplaza hacia las zonas más frías del mapa para que no existan zonas sin supervisión.

El resultado de aplicar las funciones de enfriamiento y calentamiento sobre el mapa

es mejor que la forma anterior de temporizadores, porque aquellos sólo dependían de los objetos que hubiese detectado anteriormente o que estuviesen precargados. De esa forma, si la cámara detectaba por casualidad objetos en zonas de la mesa visibles en algún momento de la ejecución , los tenía en cuenta, pero si no eran visibles para el sistema, ni precargados, no iba a llegar a saber de su existencia porque dejaría zonas de la mesa sin explorar. Con esta mejora, conseguimos que, aunque no existan objetos en una zona de la mesa, ésta se explore por completo consiguiendo información más veraz o mayor cantidad de información.

4.5. Segunda mejora: un nivel más

La segunda mejora que se ha planteado en el sistema es que el robot sea capaz de controlar otras mesas de su entorno y no se centre sólo en una. Este ascenso de nivel, nos permitiría en un futuro continuar ascendiendo de forma que al final el robot sea capaz de controlar todo su entorno. Al igual que el tipo de objetos que controlamos hasta ahora es reducido, el tipo de superficies dónde apoyar dicho objetos también es reducido con esta segunda mejora, sólo se tienen en cuenta las mesas, ninguna otra superficie como una estantería u otro mueble es manejado por el momento.

Para conseguir un nuevo nivel he creado una nueva estructura que es la que manejaría las distintas mesas del entorno. La estructura sería la siguiente:

```
struct Table
       {
         map tableMap;
         int id;
         QVec pose;
         QVec dim;
         long temperature;
         QString name;
         TObjects listObjects;
         TObjects listYoloObjects;
```

```
    TObjects listCreate;
    TObjects listDelete;
    TObjects listVisible;
};
```

Como podemos ver, esta estructura contiene las listas anteriores para controlar a los objetos. Estas listas, ahora, pertenecen a cada mesa porque la atención diferencia una mesa de otra, por tanto, cada mesa sólo debe responder por los objetos que estén sobre ella. También tenemos por cada mesa el mapa de temperatura que controla el movimiento de los motores y, al fin y al cabo, de la cámara. Además, almacenamos la posición y dimensiones de la mesa, su nombre, su identificador y un número que contiene la temperatura global de la mesa. Para cargar toda esta información, al iniciar el programa, leemos del *innermodel* lo que necesitamos, como las dimensiones de la mesa, el nombre...

Con el parámetro de temperatura el sistema debe decidir a qué mesa debe mirar. Para tomar la decisión de cambiar de mesa es necesario un umbral que indique cuándo la temperatura es lo suficientemente baja para tener que atender a otra mesa. El recorrido es bastante limitado porque los motores sólo permiten 180º de movimiento, es decir, 90º a la derecha y 90º a la izquierda. Como no tenemos todavía el control de la base del robot, no podemos mover otros motores y no tenemos la posibilidad de explorar otras zonas del entorno.

En primer lugar, se deben cargar las mesas del laboratorio que vamos a usar en un vector de mesas, que es un atributo de la clase para que todos los métodos tengan acceso a él. Para ello, se crean instancias de tipo *Table* y se lee el contenido en el *innermodel* de cada una de ellas para cargar toda la información necesaria.

En segundo lugar, se crea un entero llamado *processTable* atributo de la clase. Este entero guarda en todo momento el identificador de la mesa a la que el robot está prestando atención. Este entero varía cuando la temperatura de alguna mesa sobrepasa un umbral de temperatura fijado.

Conseguir que pase de una mesa a otra supone un método añadido al hilo principal

para que se ejecute tantas veces como se ejecuta el método “compute”. Dicho método se llama *changeTable()* y se ejecuta dentro del método *getMotorState* justo antes de buscar el punto de atención y centrarlo. Este algoritmo recorre el vector de mesas completo y por cada una de ellas comprueba si es la mesa a la que se está prestando atención. Si es así, se va aumentando la temperatura y, si no, se va disminuyendo. Este algoritmo también comprueba por cada mesa que no se supere el umbral fijado, en este caso negativo, pues cuanta menor temperatura, más tiempo lleva sin ser atendido. Cuando la temperatura de una mesa es menor que el umbral fijado, se actualiza el entero *processTable* con el identificador de la nueva mesa y se calculan los ángulos de giro que deben moverse los motores para apuntar hacia la mesa que requiere atención (3.1.7). Como ya se ha comentado, este movimiento es limitado, si la mesa se encuentra más allá de su alcance, nunca podrá llegar a verla.

5. RESULTADOS

En esta sección se expondrán los experimentos realizados para probar la validez de nuestro sistema y su funcionamiento. Comenzaremos analizando los escenarios de pruebas y, más tarde, las pruebas en sí.

5.1. Escenario de pruebas

Durante el desarrollo del sistema, hay varios puntos de inflexión significativos: la creación inicial sin los motores con su fase de pruebas, la ampliación del sistema añadiendo el movimiento de los motores, la mejora del sistema con el mapa térmico y la mejora al añadir un nuevo nivel más alto de objetos.

El escenario de pruebas físico ha sido el mismo en todos los experimentos: en el apartamento de investigación *RoboLab* un escenario con dos mesas en L y la cámara del robot montada sobre los dos motores en un trípode, puesto que no es necesario tener la base del robot real. Además, tenemos varias tazas para realizar las pruebas.

En cuanto al software, el primer escenario de pruebas era sobre una máquina de estados con sólo cinco estados: *Predict*, *YoloInit*, *YoloWait*, *Compare* y *Stress*. Éstos son los estados que realmente obtienen la imagen, la procesan y manejan la información. Cuando esta detección sencilla funcionaba correctamente, comenzamos a añadir el movimiento de los motores, integrando el estado *Moving* a la máquina de estados. A partir de aquí, realizamos las pruebas convenientes.

En ambos escenarios, hay que tener en cuenta que el archivo .xml que cargamos debe contener la representación del entorno que rodea al robot, con medidas exactas. Por ejemplo, la altura, el ancho y el largo de las mesas en la simulación deben ser los mismos que en el mundo real, medidos sobre éste. En este caso, tenemos más información de la necesaria, porque el fichero que cargamos, llamado "autonomyLab.xml", contiene una representación del laboratorio completo de pruebas, como se puede apreciar en la siguiente figura:

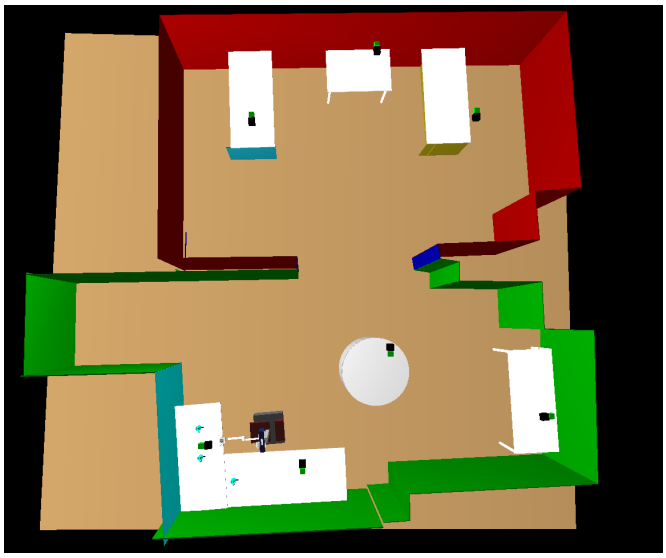


Figura 12: Simulación con RCIS del laboratorio.

Para ver esta simulación, utilizamos la herramienta RCIS de RoboComp que nos permite visualizar en tres dimensiones el fichero cargado.

Como ya comentamos con anterioridad, esta simulación necesita, tanto el archivo de simulación del laboratorio, como el de la base del robot *Shelly* con el que estamos trabajando. Esta segunda simulación es un fichero .xml, que contiene un árbol de componentes que conforman el robot. El resultado de ejecutar la base muestra la simulación del robot:

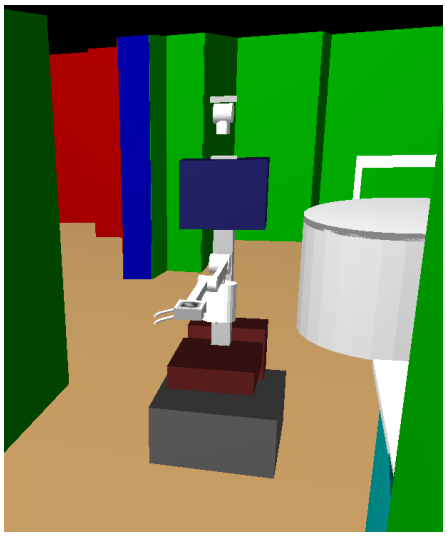


Figura 13: Simulación con RCIS del robot *Shelly*.

Para nuestros experimentos, sólo necesitamos saber la situación del robot en el mundo, de las mesas en el mundo y sus respectivos ejes de coordenadas. Aunque en la figura 12 no se aprecie con claridad, los ejes del apartamento no están centrados. El eje x con el z forman el plano del suelo de la sala, mientras el eje y es el eje vertical de la simulación. A su vez, el sistema de coordenadas de las mesas de pruebas se encuentra centrado sobre las mesas con sus ejes alineados con los del mundo, x y z formando el plano del tablero de la mesa e y el eje vertical, como ya se ha comentado. La mesa inicial respecto al robot se vería de la siguiente forma:

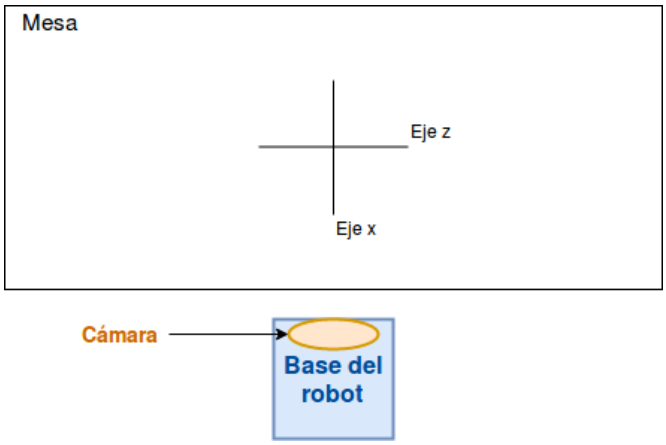


Figura 14: Ejes de la mesa en el modelo respecto al robot.

Al comenzar la ejecución de nuestro sistema, el robot se encuentra bien posicionado, porque hemos ajustado los valores del mundo simulado con los del real, para que el entorno de pruebas sea lo más parecido posible al real. La posición inicial es la de la figura 12, frente a la mesa inicial dónde están situadas las tazas.

Además, al iniciar el programa, siempre se modifica la posición de los motores para ponerlos a (0,0), de forma que el sistema siempre comience con los mismos valores iniciales.

5.2. Experimentos

5.2.1. Experimentos iniciales

Comenzamos realizando capturas estáticas de la cámara, por tanto, el sistema sólo puede detectar cambios que se produzcan en su escena de visión, no puede tener en cuenta ninguno de los objetos que no perciba por la cámara. Si existe un vaso en la mesa que está observando el robot pero fuera de su campo de visión, lo cargará del fichero .xml, pero no podrá verificar su existencia en el mundo real. Este es el caso de la figura 15, en la que el robot sólo percibe una taza aunque en el modelo existan dos y él cargue ambas al iniciar la simulación.

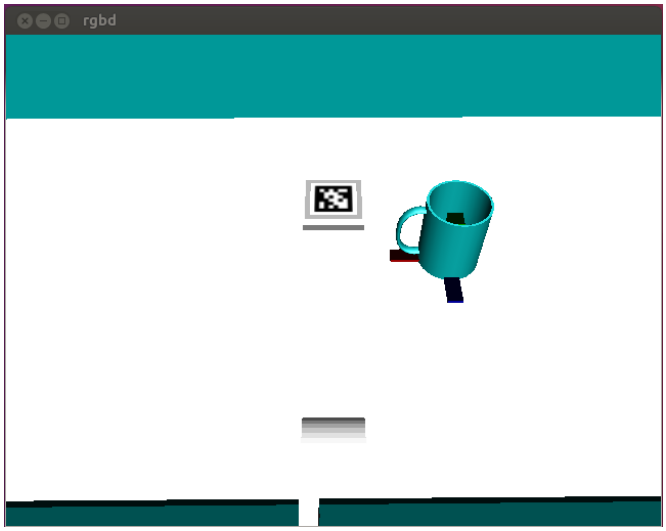


Figura 15: Visión de la simulación a través de la cámara.

En la primera configuración de pruebas, el mundo simulado sólo tiene una taza sobre la mesa y se encuentra en su campo de visión, como en la figura 15. Los casos que pueden aparecer son:

1. En el mundo real no hay ningún objeto en su campo de visión. Entonces, el robot borra del .xml el nodo correspondiente a la taza y actualiza su lista de objetos totales. También lo elimina de su lista de visibles y se actualiza el *innerviewer* borrando la taza de la simulación.

2. En el mundo real existe un objeto.

   a) Puede ser que este objeto se encuentre en una posición muy parecida a la que estaba almacenada en el *innermodel*, entonces, solamente actualiza su posición, y se añade a la lista de objetos visibles.

   b) Puede ser que el objeto se encuentre en una posición totalmente distinta a la almacenada en el .xml, entonces, borra el objeto antiguo del *innermodel* y crea uno nuevo, pues lo toma como nuevo, así que se añade, sustituyendo al anterior, a la lista de objetos totales , a la de objetos visibles y al *innerviewer*.

   c) La última posibilidad es que el servidor de Yolo no detecte correctamente el objeto o se encuentre justo en los límites de la captura, que como hemos observado, provoca muchos errores y no lo tenemos en cuenta, es decir, se trata como si el objeto real no existiera y se borra de la lista de objetos totales y del *innerviewer*.

3. En el mundo real no existe ningún objeto al comenzar, pero luego alguien lo añade. En este caso la simulación comenzaría comprobando el objeto que obtenemos al leer el .xml. Como no está, lo borra del sistema. Posteriormente, alguien pone una nueva taza sobre la mesa en su campo de visión, entonces, detecta la introducción de la mano en la imagen y la taza que se añadiría al *innermodel* y al *innerviewer* en la posición que reciba el sistema del servidor de Yolo.

4. En el mundo real existe el objeto, pero a lo largo de la prueba alguien lo quita. El sistema comenzaría detectando correctamente el objeto, pero, cuando desaparece, percibe ese cambio y actualiza su modelo borrando la taza de su campo de visión.

5. En el mundo real hay más de un objeto.

a) Si alguno de los objetos se parece lo suficiente al cargado de la memoria, se toma éste como el antiguo y el resto se posiciona en el .xml según los datos obtenidos por el servidor de Yolo.

b) Si ninguno de los objetos se encuentra en una situación similar al cargado, se borra el antiguo objeto y los nuevos detectados se añaden a la memoria, a las listas del sistema y al *innerviewer* en las posiciones obtenidas por el servidor de Yolo.

Entre los problemas más frecuentes que nos hemos encontrado está el problema de que Yolo no detecta correctamente un objeto. Este problema viene derivado de la iluminación del entorno en el que se encuentre el robot. Como el sistema está sujeto a cambios en la luz, hemos ecualizado la imagen, aumentando los contrastes de las capturas obtenidas y obteniendo así mejores resultados con el servidor de Yolo.

5.2.2. Experimentos con motores

En esta segunda sección experimental, se añade movimiento a la cámara, para que, así, el robot pueda controlar más partes de su entorno. El planteamiento inicial consistía en que el sistema modificaba su estado de reposo sólo cuando algo le hacía moverse, por ejemplo, el temporizador que tenía cada objeto. Así, cuando el temporizador de un objeto de la mesa sobrepasaba un umbral fijado, se obtenía la posición de ese objeto, la de los motores y lo que tendrían que desplazarse cada uno para poder posicionar en el centro de la imagen ese objeto que llevaba más tiempo sin ser atendido. Se realizaba el movimiento y volvía al estado *Predict*.

Esto causaba dos errores posibles:

- El primero era que dejaba zonas de la mesa sin explorar, porque si no había un objeto en una zona, nada iba a hacer que el robot mirase hacia esa zona durante la ejecución.

- A su vez, este problema implicaba otro, si existía un objeto en una parte de la mesa sin explorar, el sistema nunca podría saber de su existencia, porque no

exploraría esa zona de la mesa nunca.

En este tipo de pruebas, por tanto, sólo se podía obtener información de objetos que estuviesen cargados en el *innermodel* inicialmente o de objetos que de manera fortuita, se encontrasen en el campo de visión de la cámara en algún momento y fuesen correctamnete detectados por Yolo.

5.2.3. Experimentos con el mapa térmico

Para subsanar los dos errores anteriores, añadimos el mapa térmico, como se explica en la sección 4.5.. Este mapa es una cuadrícula proyectada sobre el plano de la mesa, que nos indica zonas más o menos calientes. La cámara debe moverse siempre hacia las zonas más frías, es decir, se va a explorar toda la mesa, sin dejar ninguna parte sin visitar y, por tanto, ninguna taza sin identificar.

Al añadir este mapa térmico ya se pueden realizar pruebas más complejas y explicativas.

La primera prueba que realizamos es el *tracking*, es decir, mantener la atención del robot fija sobre un objeto durante el tiempo que éste se esté moviendo. Esto implica que el robot va recalculando continuamente la posición de sus motores para que el objeto se mantenga centrado sobre la imagen capturada por la cámara RGBD durante todo el recorrido. Para esta prueba debe estar activo el mapa de temperatura, pues, el seguimiento es posible gracias a que el objeto, al moverse de posición, enfría intensamente una nueva zona y esto hace que, al recalcular la temperatura mínima de la mesa, sea la nueva zona dónde se ha posicionado el objeto la de menor temperatura. Si el objeto cesa en su movimiento, el robot mantiene la atención unos segundos, pero, a continuación, la desviará hacia otra zona de la mesa más fría. Para realizar esta prueba, hemos utilizado una sola taza que se ha ido desplazando sobre la mesa con ayuda de una persona. Para ver esta prueba en vídeo, puede visitar el siguiente enlace: `https://drive.google.com/file/d/1qqWS8MPFFiCnnsjstGkuPHFAjh3u63wN/view?usp=sharing`.

La segunda prueba fue la atención intermitente entre dos tazas. Para ello, se colocan

dos tazas sobre una mesa y se ejecuta el programa. El robot comienza a observar su entorno, reconoce la posición de la primera taza que encuentra, se centra sobre ella y, como no se está moviendo, puede desviar su atención a otras zonas más frías. Como existen dos tazas precargadas, el mapa va a enfriar esas zonas de manera más intensa hasta que compruebe que no existen en el mundo real. Cuando la cámara deja de atender a la primera taza que encontró, va a ir a la zona más fría, que será la zona de la taza de la simulación. Si en el camino encontrase una taza real, se centraría sobre ella hasta que su temperatura aumente bastante como para volver a desviar su atención. Entonces, volvería a buscar la primera taza que se encontrará en la misma posición. Así, continuaría un tiempo alternando entre ambas hasta que otra zona de la mesa llevase mucho tiempo sin ser atendida y se enfriase más que las tazas, entonces, requeriría atención esta zona y la cámara se centraría en ella. Al no existir ningún otro objeto en esta tercera zona, se calentaría rápidamente y las tazas anteriores volverían a captar toda la atención. Si al mirar esta tercera zona, encontrase un tercer objeto, centraría la atención sobre él durante un tiempo, más tarde, comenzaría a alternar entre los tres objetos existentes. Para ver esta prueba en vídeo, puede visitar el siguiente enlace: `https://drive.google.com/file/d/1vZfDDiHoUPx0x-lh7pPFZ6p7P12hme0m/view?usp=sharing`.

Por último, unimos las dos pruebas: mantenimiento de la atención y *tracking* con más de un objeto. Comenzamos haciendo *tracking* con una taza. Ésta se para y añadimos una nueva taza a la escena. Como es nueva, capta la atención del robot. Comenzamos a desplazar esta segunda taza y, aunque siguen ambas en la escena, la que se está desplazando es la que capta la atención al ir variando la zona de mínima temperatura. Cuando paramos la segunda taza, vuelve la atención a la primera taza. Si comenzamos a desplazar la primera, ésta mantiene la atención hasta que se pare. Paramos una taza al lado de la otra, esto implica que ambas deben recibir la misma atención, porque están las dos estáticas. Entonces, comienza a alternar la atención. `https://drive.google.com/file/d/1EaU8tDK7tXGEApe_QGouSyKY65eCI_hC/view?usp=sharing`.

5.2.4. Experimentos con más de una mesa

Para finalizar las pruebas, añadimos un nivel más, lo que significa que, ahora, el sistema puede prestar atención a más de una mesa, pero sólo a una en un instante concreto. Esto quiere decir que la atención se centra sobre una única mesa, aunque el sistema capte más información y siga actualizando su entorno.

Cambiar de mesa implica superar un umbral global mínimo de temperatura. Cuando esto ocurre, significa que hay otra mesa que requiere más atención. Entonces, se recorre el mapa de la nueva mesa y se busca la zona más fría, que es a dónde se moverá la cámara. En nuestro caso, cuando la cámara llega a la segunda mesa, se encuentra vacía. Entonces, añadimos una taza, se centra en la imagen y mantiene la atención un tiempo hasta que la primera mesa vuelve a requerir atención. El tiempo de atención sobre cada mesa se puede ajustar. El tiempo de atención sobre cada mesa puede variar, en el siguiente enlace puede ver una prueba con un intervalo de tiempo pequeño en el cambio de mesa: `https://drive.google.com/file/d/1zkX6pImNgvkQBkh8xS_if1SZXzx7REYo/view?usp=sharing`.

Como el movimiento de los motores es limitado, la segunda mesa no se puede ver completa, porque sólo se permiten 90º de giro hacia la izquierda o hacia la derecha. Esto supone que las pruebas sobre varias mesas también sean limitadas.

6. CONCLUSIONES Y TRABAJO FUTURO

En este proyecto se ha desarrollado una posible forma de atención para robots avanzados. Este tipo de robots es en el que cada vez más gente confía y, por tanto, cada vez deben ser mejores en todos los aspectos. Aunque no parezca demasiado importante, la consciencia sobre sí mismo y la atención sobre su entorno puede ayudar a desarrollar o complementar muchas otras tareas a las que se dedica este tipo de robot. En su uso asistencial por ejemplo, podría determinar en cualquier momento dónde hay un objeto si alguien se lo pidiese, sin necesidad de tener que buscarlo, porque su modelo interno del entorno estaría continuamente actualizado.

Como se puede apreciar en las pruebas, el sistema se debe mejorar para que sea más rápido y receptivo, pero es una primera aproximación muy interesante a un modelo mucho más complejo de atención. También, hay que tener en cuenta que la carga computacional del programa es muy alta, además, de que no hay nada paralelizado, lo que aligeraría bastante el proceso. El número de frames que recoge la cámara por segundo es muy alto, por tanto, muy pesado para ser procesado por Yolo, pero hay disponible actualmente una nueva versión del mismo que procesa de forma más ligera y precisa.

Otra conclusión a mencionar sería que el entorno en el que se encuentra el robot debe estar previamente creado en un fichero .xml, porque la memoria, con el sistema actual no crea el entorno dinámicamente, sino que se basa en un entorno ya creado que se va actualizando.

A modo de resumen, se puede decir que el sistema que se ha implementado obtiene unos resultados con una alta fiabilidad, como se ha podido comprobar en las pruebas que se han realizado con las diferentes configuraciones. Para el sistema que se ha implementado en el robot Shelly algunas posibles mejoras podrían ser, por ejemplo:

- Añadir el control de otros motores del robot al sistema, como la base, que nos permitan obtener más información viendo otras partes del entorno.
- Ampliar el número de tipos de objetos que detecta el sistema, que no fuesen

sólo tazas y vasos, sino personas, libros, platos... en definitiva, más diversidad. Esto sería posible dentro de los límites de Yolo, puesto que es el detector de objetosutilizado. Si aparecen en la escena objetos irreconocibles por Yolo, no podríamos procesarlos. No obstante, la variedad de objetos que reconoce el servidor de Yolo es muy amplia y nos sería suficiente para pruebas posteriores.

- Añadir una propiedad "velocidad" a los objetos visibles porque, así, se podría prever dónde estaría el objeto en la siguiente iteración de la máquina de estados. Esta ampliación sería muy útil con personas, para saber su posición y trayectoria durante el tiempo que se encuentre en el campo de visión del robot.

- Añadir el movimiento ocular de seguimiento para que la cámara sea capaz de seguir un objeto en movimiento continuo con el fin de mantenerlo siempre en su campo de visión sin saltos bruscos.

- Abstraer a niveles mayores la comprensión del *innermodel* por parte del sistema, para que se pueda integrar el mundo completo, no sólo una parte de éste.

- Controlar la temperatura del mapa con ecuaciones diferenciales, que nos permitan eliminar los umbrales. Así, los movimientos serían más lentos y no tan bruscos al cambiar de una zona de la mesa a otra.

- Separar en varios hilos de ejecución la ejecución del programa, puesto que, cada vez es más pesado y tarda más en procesar. Por ejemplo, el proceso de imágenes por parte de Yolo podría correr en un hilo distinto que quitase carga de trabajo al hilo principal. También, se podría paralelizar el uso del *innerviewer* pues sólo sería un hilo ejecutándose que mostrase todos los cambios sufridos por el fichero .xml del *innermodel*.

- No añadir o borrar los objetos hasta que exista certeza de presencia o de ausencia. Esto quiere decir que el objeto debe aparecer o desaparecer en varios frames consecutivos para añadirse o eliminarse. En la implementación actual, si Yolo en

un frame no detecta un objeto aunque realmente exista, el sistema lo borra de su memoria.

El modelo ideal de este sistema observaría el entorno real y crearía dinámicamente su propio modelo interno. Aunque se necesitan muchas mejoras para llegar hasta eso, el sistema desarrollado constituye un paso inicial en esta dirección.

Referencias

[1] Francisco J. Rubia Vila. La consciencia es el mayor enigma de la ciencia y la filosofía. *Tendencias21: REVISTA ELECTRÓNICA DE CIENCIA, TECNOLOGÍA, SOCIEDAD Y CULTURA.*, 2010.

[2] José Mª Colmenero; Andrés Catena y Luis J. Fuentes. Atención visual: Una revisión sobre las redes atencionales del cerebro. *Servicio de Publicaciones de la Universidad de Murcia, Anales de psicología*, 2001.

[3] Ricardo Sanz. Hacia las máquinas conscientes. (Spanish) [towards conscious machines]. *III Jornadas de Teoría y Psicología. Aspectos de la consciencia.*, pages 1–19, 2005.

[4] Héctor E. Medellín Anaya. Tranformaciones en 3D. *Sitio web de la Universidad Autónoma San Luis Potosí*.

[5] Irving Alberto Cruz Matías. Proyecciones. In *Rotaciones multidimensionales generales*, volume 5, pages 1–22, Cholula, Puebla, México, 2007. Escuela de Ingeniería y Ciencias, Universidad de las Américas Puebla.

[6] J. Antonio Aznar Casanova. Bases neuro-fisiológicas de la visión. In *Psicología de la percepción visual*, Universidad de Barcelona, 2017. Pirámide.

[7] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. *University of Washington*, 2018.

[8] Pablo Bustos García de Castro. RoboComp. *GitHub*, 2017.

[9] Wikipedia. Componente de Software. *Wikipedia*, 2017.