Universidad Zaragoza
1542

Trabajo Fin de Grado

INTEGRACIÓN DE TÉCNICAS DE DETECCIÓN DE PERSONAS CON VISIÓN PARA ROBÓTICA COLABORATIVA

Autor/es

Daniel Cubel Gálvez

Director/es

José Jesús Guerrero Campo

Escuela de Ingeniería y Arquitectura

2018

RESUMEN

La robótica colaborativa es una tecnología en auge ya que cada vez más empresas demandan soluciones flexibles y compactas para la automatización de sus procesos productivos, algo que los robots tradicionales no pueden conseguir debido a las medidas de seguridad necesarias.

La norma ISO 10218-1/2 especifica los modos de colaboración entre persona y robot mientras que la norma ISO/TS 15066:2016 recoge las medidas de seguridad para los sistemas de robótica colaborativa y el entorno de trabajo.

La mayor capacidad de cálculo de los ordenadores permite el desarrollo de aplicaciones de visión por computador, capacidad de autoaprendizaje y la posibilidad de instalar un mayor número de sensores que pueden aplicarse a la robótica colaborativa a la hora de detectar a las personas en el entorno del robot.

Este TFG se centra en el reconocimiento del cuerpo humano con una cámara RGB-D, la Kinect V2 de Windows, que permite localizar a las personas respecto a la cámara y la utilización de los datos aportados por la Kinect (posición y orientación de las articulaciones del cuerpo humano) para programar de forma colaborativa un robot tradicional ABB.

Primero, se aborda el estado del arte de la robótica colaborativa y las técnicas de detección de personas mediante la obtención de la profundidad. Posteriormente se realiza un estudio del hardware utilizado en el trabajo, el robot ABB IRB 120 y la cámara Kinect y software utilizado tanto para programar la cámara como para programar el uso de los datos de la Kinect para programar el robot.

El siguiente problema abordado son las posibilidades de uso de los datos de la Kinect, como la captura de la posición de la mano y la vigilancia de la célula. Una vez abordado este problema, se programa la Kinect para obtener los datos de posición y orientación de las articulaciones del cuerpo y una aplicación que permita usar esos datos para trabajar de forma colaborativa con el robot.

Una vez se ha desarrollado la primera aplicación, se estudia la posibilidad de controlar la aplicación mediante los gestos reconocidos por la Kinect.

Por último, se analizan las conclusiones alcanzadas y el trabajo futuro.

ABSTRACT

Collaborative robotics is a booming technology as more and more companies demand flexible and compact solutions for the automation of their productive processes, something that traditional robots can't achieve because of the security measures they require.

ISO 10128-1/2 specifies safety requirements for industrial robots and describes the collaborative modes between human and robot while ISO 15066:2016 specifies the security requirements for collaborative robots and the work environment.

The greater computing capacity of computers allows the development of computer vision applications, machine-learning capacity and the possibility of installing more sensors, which can be applied to collaborative robots when detecting people in the robot's environment.

This degree final project focuses on the recognition of human body with an RGB-D camera, Microsoft's Kinect V2, which allows locating people relative to the camera and using its data (position and orientation of body joints) to collaboratively program a traditional ABB robot.

First, we approach the state of art in collaborative robots and people detection techniques through depth imaging. Later, the hardware and software used in the project is studied: on one hand, the ABB IRB 120 robot and the Kinect V2 camera and, on the other hand, the software used to program the camera and use the data provided by the camera.

The next problem addressed are the possibilities of using these data, for example, capturing the hand's position or monitoring the robotic cell. Once we have addressed this problem, we program the camera to obtain the body joint's position and orientation and the application to use that data to program a robot in a collaborative way.

Once we have programmed the application, we approach the possibility of controlling it with gestures recognized by the camera.

Finally, we analyse the conclusions of the project and the future work.

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

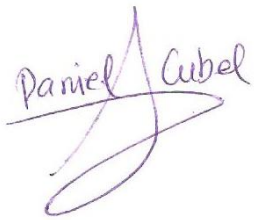
D./Dª. DANIEL CUBEL GÁLVEZ,

con nº de DNI 77132472V en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) GRADO, (Título del Trabajo) INTEGRACIÓN DE TÉCNICAS DE DETECCIÓN DE PERSONAS CON VISIÓN PARA ROBÓTICA COLABORATIVA,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 30 de agosto de 2018



Fdo: DANIEL CUBEL GÁLVEZ

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGOS

1. INTRODUCCIÓN

1.1. OBJETO

El objeto de este trabajo fin de grado es el desarrollo de una aplicación para robótica colaborativa que integre técnicas de detección de personas por visión por computador y utilice la información obtenida para la programación de un robot, de forma que este trabaje de forma colaborativa con una persona.

El trabajo está centrado en la aplicación de la visión por computador a la robótica colaborativa. La visión por computador nos permite tener localizada en todo momento a la persona dentro de una célula robótica, permitiendo eliminar barreras de seguridad y la colaboración entre el robot y la persona.

La utilización de sensores que son capaces de reconocer el cuerpo humano nos permite la interacción directa entre la persona y el robot una vez conozcamos la posición relativa sensor-robot.

Para la detección de personas se va a utilizar y programar la cámara Kinect V2 de Microsoft y los datos proporcionados se utilizarán para la programación del robot IRB 120 de ABB.

1.2. ALCANCE

Las actividades incluidas en el trabajo son:

1) Estudios previos:
   1.1) Revisión bibliográfica y estado del arte de la robótica colaborativa.
   1.2) Estudio y documentación del uso de la Kinect V2 para comprender su funcionamiento y conocer sus limitaciones.
   1.3) Estudio de las herramientas de desarrollo de la Kinect V2.
   1.4) Estudio del robot ABB IRB 120, el robot utilizado en el trabajo.
   1.5) Estudio del programa de simulación de ABB RobotStudio y de las herramientas de desarrollo que nos ofrece.
   1.6) Estudio del entorno de programación Microsoft Visual Studio
2) Estudio de las opciones disponibles de programar la Kinect V2 para la detección del cuerpo humano. Elección de la opción adecuada a las características del trabajo.
3) Estudio de las alternativas de uso de los datos obtenidos en la detección del cuerpo humano.
4) Desarrollo de un programa que utilice los datos obtenidos en la detección del cuerpo humano para la programación de un robot. Comprobación del correcto funcionamiento del programa en RobotStudio.
5) Conclusiones del trabajo desarrollado.

La aplicación del trabajo desarrollado a un robot real no se contempla en el alcance de este trabajo.

1.3. ORGANIZACIÓN DEL TRABAJO

En el capítulo 2 se aborda el estudio del estado del arte de la robótica colaborativa y el reconcomiendo del cuerpo humano con técnicas de visión a partir de la medida de la profundidad.

En el capítulo 3 se aborda el estudio del hardware utilizado, el robot ABB IRB 120 y la cámara RGB-D Kinect V2 de Microsoft.

En el capítulo 4 se aborda el software utilizado para el desarrollo del trabajo: el SDK de la cámara, el programa de simulación de ABB RobotStudio, los SDK de ABB y el entorno de desarrollo de Microsoft Visual Studio.

En el capítulo 5 se estudian las posibilidades del uso de los datos proporcionados por la Kinect para programar un robot de forma colaborativa.

En el capítulo 6 se especifican las formas de programación de la Kinect y se programa para obtener la posición y orientación de las articulaciones del cuerpo humano.

En el capítulo 7 se programa la aplicación de PC SDK que permite utilizar los datos de la Kinect para programar de forma colaborativa el robot.

En el capítulo 8 se propone otro enfoque para la aplicación del capítulo 7 mediante el control por gestos.

En el capítulo 9 se recogen las conclusiones del trabajo y las posibles líneas de trabajo futuro.

2. MARCO TEÓRICO

2.1. ROBÓTICA Y COLABORACIÓN

Los robots industriales tradicionales requieren una instalación y mantenimiento caros, además de necesitar barreras de seguridad para evitar posibles accidentes que pongan en peligro a los trabajadores. Estos requerimientos suponen que las pequeñas y medianas empresas no puedan permitirse la automatización de su producción y demandan soluciones asequibles y compactas. La robótica colaborativa supone una solución a este problema.

Además, la robótica colaborativa está enmarcada en la Industria 4.0, que busca una mayor eficiencia, un menor coste de producción y una mayor productividad a través de la automatización integrada.

La robótica colaborativa rompe con las barreras humano-robot de las células robóticas convencionales al permitir una interacción directa entre las personas y el robot. Los robots son capaces de realizar tareas repetitivas con exactitud, rapidez y repetibilidad, lo que es imposible de conseguir por los humanos, mientras que las personas tienen versatilidad, capacidad de adaptación a entornos dinámicos y capacidad de decisión, algo de lo que carecen los robots. La robótica colaborativa combina las ventajas de las dos partes, lo que resulta en una mayor productividad del trabajador mientras se reduce su estrés y su fatiga.

Los robots colaborativos suelen ser fáciles de controlar y programar, ya que cuentan con sistemas intuitivos que están basados en realidad aumentada, programación paso a paso o programación por demostración. Por el contrario, los robots industriales tradicionales necesitan expertos en ingeniería para ser programados.

Según el nivel de interacción robot-humano, [1] distingue tres niveles jerarquizados, de forma que el siguiente incluye al anterior:

- Seguridad: célula robótica sin barreras de seguridad, se utilizan sensores para evitar colisiones y reducir la fuerza del robot al impactar en caso de que se produzcan.
- Coexistencia: el robot y la persona comparten un espacio de trabajo seguro e incluso pueden trabajar sobre el mismo objeto, pero no hay contacto ni coordinación de las acciones e intenciones.
- Colaboración: permite que el robot y la persona realicen una tarea compleja juntos, existiendo coordinación e interacción. Puede ser conseguido estableciendo contacto intencionadamente o sin contacto mediante comandos de voz o gestos.

Por tanto, la colaboración incluye una arquitectura de control que evita las colisiones, capacidades de detección y reacción, al mismo tiempo que la colaboración humano-robot.

También puede distinguirse entre la colaboración humano-robot y la interacción humano-robot, con la diferencia de que en la colaboración la persona y el robot trabajan con un mismo objetivo mientras que la interacción no necesariamente trabajan con el mismo objetivo. La definición de interacción humano-robot es similar a la de coexistencia.

2.1.1. Modos de colaboración según norma ISO 10218-1/2

Con la introducción de la robótica colaborativa se ha dado gran importancia a los estándares de seguridad. La norma ISO 10128-1/2, que recoge los requisitos de seguridad para los robots industriales, propone cuatro modos de colaboración (Fig. 2.1).

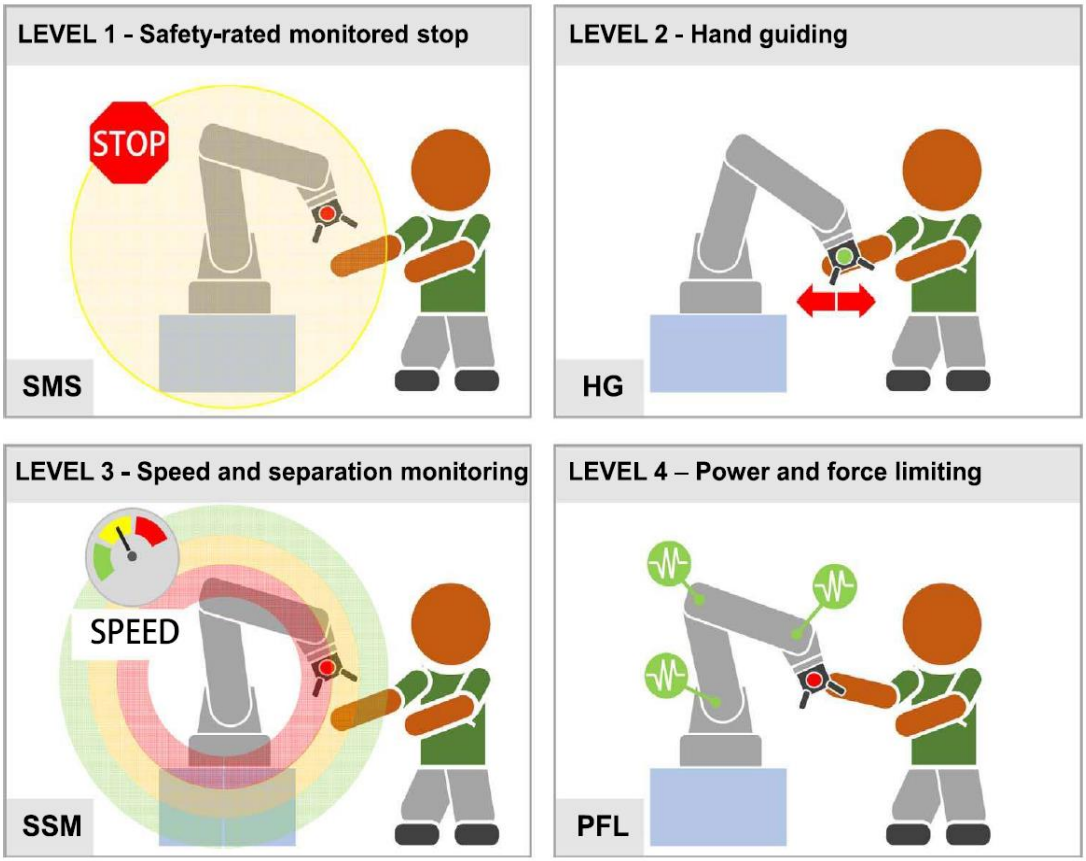


Fig. 2.1 – Los modos de colaboración según la norma ISO 10128-1/2 [2]

1) Parada supervisada de seguridad.

   Este modo de colaboración consiste en limitar el movimiento del robot, de forma que si hay una persona dentro del espacio de trabajo colaborativo el robot no puede acceder a éste o si el robot está ya dentro de este espacio el robot se detiene. Cuando la persona abandone el espacio de trabajo colaborativo el robot reiniciará su movimiento.

   Para ello, el sistema de seguridad del robot debe ser capaz de detectar la presencia de personas dentro del espacio de trabajo colaborativo.

   Si no hay personas dentro del espacio de trabajo colaborativo, el robot puede funcionar de forma no colaborativa.

   En esencia, esta forma de trabajo es similar la parada de seguridad tradicional, con la diferencia de que la parada supervisada de seguridad supervisada tiene una función que deja potencia en los actuadores del robot.

2) Guiado a mano.

   En este modo de colaboración el operador puede programar el robot moviéndolo con la mano. Para ello, el sistema de control requiere de la parada supervisada de seguridad y de una velocidad supervisada de seguridad.

   Cuando el robot está dentro de la zona de trabajo, el robot trabaja de forma no colaborativa, ejecutando el programa que tiene cargado. Si un operador entra a la zona de trabajo el programa y el movimiento se detienen y el robot cambia a la funcionalidad de velocidad supervisada de seguridad, que permite que el operador pueda mover el robot. Cuando el operador suelta el robot, este permanece en parada supervisada de seguridad hasta que el operador sale de la zona de trabajo. Entonces, el robot reanuda el movimiento y el programa que estaba ejecutando.

3) Supervisión de velocidad y posición

   En este modo de colaboración, dependiendo de la distancia entre la persona y el robot, este se moverá a velocidad normal, reducida o se parará.

   Un enfoque interesante de este modo de colaboración es el cálculo dinámico del espacio seguro, de forma que se calcula el mínimo espacio seguro dependiendo de la robustez y la posición del robot. Esto implica que el operador cuenta con el máximo espacio de trabajo posible.

4) Limitación de fuerza y potencia

   Este modo de colaboración consiste en limitar la potencia de los motores y la fuerza del robot de forma que una persona pueda trabajar con el robot. Se necesita equipo dedicado y modelos de control para gestionar las colisiones entre el robot y la persona sin que se dañe a la persona.

2.1.2. Retos de la robótica colaborativa

La robótica colaborativa es todavía una tecnología en auge, por lo que tiene varios retos a los que enfrentarse. En [2] encontramos algunos de ellos:

Por un lado, cualquier aplicación de colaboración tendrá que enfrentarse a problemas de seguridad, ya que al haber contacto directo entre el robot y a persona se debe garantizar que esta interacción es segura.

Además, las interfaces de usuario deberán ser diseñadas de forma que sean intuitivas para que interactuar con el robot sea fácil y puedan utilizarse las habilidades del trabajador al máximo. Si el operador puede proporcionar entradas al robot y programarlo fácilmente, el trabajo tendrá que preocuparse menos por la comunicación con el robot y si la realimentación es adecuada, podrá reaccionar rápidamente a situaciones imprevistas y saber el estado del sistema. El uso de nuevos enfoques de programación, como la realidad aumentada, programación paso a paso o programación por demostración evitan el uso de ratón, teclado, pantallas…

Para conseguirlo se necesitan nuevos métodos de diseño, que se traducen en leyes de control, sensores y enfoques de asignación y programación de tareas, que permitan a la persona estar cerca del robot, compartir el área de trabajo y las tareas y proporcionar el sistema de interacción con la flexibilidad requerida.

2.1.3. Términos y conceptos

Para la correcta comprensión del trabajo es necesario definir una serie de términos y conceptos relacionados con la robótica que irán apareciendo a lo largo del texto. Estas definiciones aparecen el manual de RobotStudio [3].

2.1.3.1. Conceptos de RAPID

Declaración de datos – Se utiliza para crear instancias de variables o tipos de datos, como num o tooldata.

Instrucción – Los comandos de código que hacen que ocurra algo, como por ejemplo un cambio de valor en una variable.

Instrucción de movimiento – Crea los movimientos del robot. Debe especificarse una posición y unos parámetros de movimiento, como la velocidad o la zona del movimiento.

Rutina – Normalmente, conjunto de declaración de datos e instrucciones utilizados para implementar una tarea. Pueden ser procedimientos, funciones o rutinas TRAP.

Procedimiento – Conjunto de instrucciones que no devuelve ningún valor

Función – Conjunto de instrucciones que devuelve un valor.

Rutina TRAP – Conjunto de instrucción asociado a una interrupción.

Módulo – Conjunto de declaración de datos y rutinas.

2.1.3.2. Conceptos de programación

Programación en línea – Programación con conexión al módulo de control. También implica el uso del robot la creación de posiciones y trayectorias.

Programación fuera de línea – Programación sin conexión al robot ni al módulo de control.

Programación real fuera de línea – Programación conectando un entorno de simulación a un controlador virtual. Permite programar, hacer pruebas y optimizar programas.

Controlador virtual – Software que emula al controlador del robot para poder ejecutar en el PC el mismo software que se ejecuta un robot real.

Sistema de coordenadas – Se utilizan para definir posiciones y orientaciones. Al programar un robot se pueden utilizar distintos sistemas de coordenadas para posicionar más fácilmente los objetos unos respecto de los otros.

2.1.3.3. Sistemas de coordenadas

Sistema de coordenadas mundo – Sistema de coordenadas de la estación.

Sistema de coordenadas de la base – Sistema de coordenadas del robot, situado en su base.

Sistema de coordenadas del punto central de la herramienta (TCP) – Sistema de coordenadas situado en el centro de la herramienta.

Sistema de coordenadas de un objeto de trabajo – El objeto de trabajo representa la pieza de trabajo física. A la hora de programar los objetivos del robot, estas siempre estarán referidos al sistema de coordenadas de un objeto de trabajo. Si no se especifica

ningún objeto de trabajo, el objeto de trabajo predeterminado (Wobj0) coindice con la base del robot.

Sistema de coordenadas del usuario (UCS) – Sistema de referencia definido por el usuario, utilizado para crear puntos de referencia.

Matriz de transformación – Se utiliza para cambiar el sistema de referencia.

2.1.3.4. Objetivos y trayectorias

Objetivo – Coordenada que debe ser alcanzada por el robot

Posición – La posición del objetivo, definida respecto a un sistema de coordenadas de un objeto de trabajo.

Orientación – La orientación del objeto, definida respecto a un sistema de coordenadas de un objeto de trabajo.

Configuración – Configuración de los ejes del robot para alcanzar el objetivo.

Trayectorias – Secuencia de instrucciones de movimiento. Sirve para que el robot se mueva a lo largo de una serie de objetivos.

Instrucciones de movimiento – Se componen de un objetivo, datos de movimiento (velocidad, zona de movimiento), referencia a datos de una herramienta, referencia a un objeto de trabajo.

2.1.3.5. Configuraciones del robot

Lo normal cuando se programa un objetivo es que el robot pueda alcanzarlo de distintas formas, con distintas configuraciones de los ejes del robot. Para distinguir entre configuraciones, todos los objetivos tienen un valor de configuración que indica en que cuadrante debe situarse cada eje.

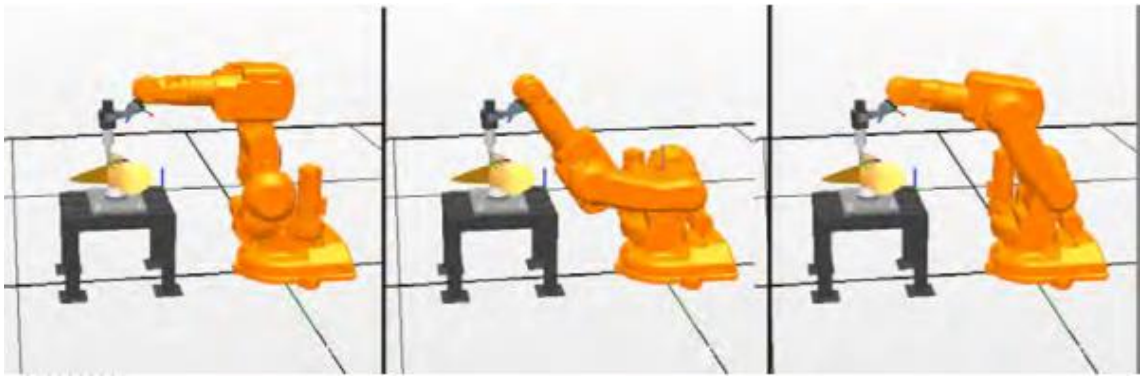


Fig. 2.2 – Distintas configuraciones para el mismo objetivo

2.2. RECONOCIMIENTO DEL ESQUELETO HUMANO CON VISIÓN

La principal ventaja que introduce la robótica colaborativa es la interacción directa entre la persona y el robot. Pero para que esta interacción sea segura, es necesario conocer en todo momento la posición relativa entre la persona y el robot.

La visión por computador nos permite analizar la imagen proporcionada por un sensor, con lo que podemos analizar la profundidad a la que se encuentran los objetos en la imagen y encontrar posibles obstáculos en el camino del robot. A partir de esos datos de profundidad también podemos realizar el reconocimiento del esqueleto humano.

La Fig. 2.3 muestra las técnicas de obtención de la profundidad de un objeto en el espacio respecto un dispositivo.

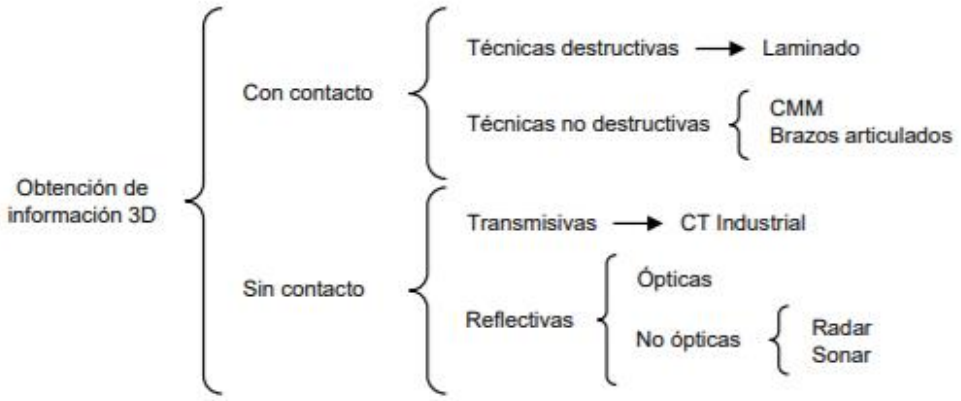


Fig. 2.3 – Métodos de obtención de la profundidad [4]

En el ámbito industrial, las técnicas ópticas son las que proporcionan mejor resultado y, por tanto, son las que van a ser explicada en este TFG.

Las técnicas ópticas pueden dividirse en dos grupos según el tipo de fuente emisora: técnicas activas y técnicas pasivas. Las técnicas pasivas no utilizan una fuente emisora propia, si no que emplean fuentes externas al dispositivo como pueden ser la luz solar, la luz interior o luz infrarroja generada por un emisor. Las técnicas activas si emplean una fuente emisora propia, como cámaras infrarrojas instaladas en el dispositivo.

Las principales técnicas de medición ópticos son:

- Visión estéreo
- Luz estructurada
- Interferometría
- Tiempo de vuelo (Time of flight, ToF)

2.2.1. Visión estéreo

La técnica de visión estero es un método pasivo que se basa en el sistema de visión del ser humano, tomando dos imágenes desde distintas posiciones y analizando las diferencias de proyección de un mismo punto. Estas diferencias son conocidas como disparidad y su análisis permite conocer la profundidad.

Para obtener la profundidad se debe establecer una correspondencia entre los puntos de ambas imágenes. Esta correspondencia se calcula con algoritmos que tratan de localizar las proyecciones de un punto en las dos imágenes. La principal limitación es que un solo píxel no es suficiente para identificar los puntos, por lo que es necesario analizar los pixeles vecinos.

Para poder obtener la profundidad es necesario conocer la configuración geométrica de las dos cámaras. Si las dos cámaras están alineadas se pueden calcular la profundidad mediante técnicas de triangulación.

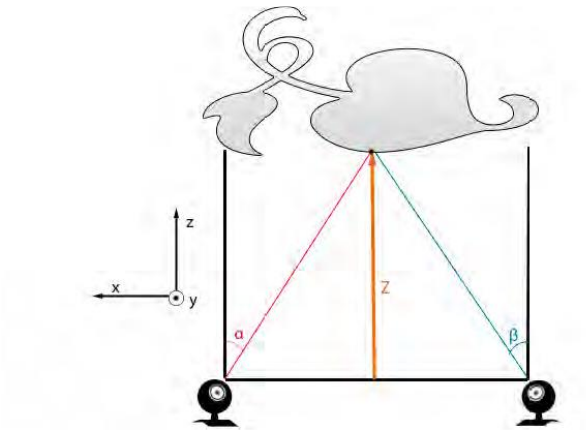


Fig. 2.4 – Triangulación en visión estéreo [5]

2.2.2. Luz estructurada

La técnica de luz estructurada es un método activo que se basa en la proyección de un patrón de luz proyectado conocido sobre la escena. Para obtener la profundidad se debe analizar la distorsión del patrón respecto al patrón inicial. Al ser el patrón conocido, se eliminan los problemas de correspondencia que se encontraban en la visión estéreo.

El codificado del patrón puede realizarse de diferentes formas: multiplexado temporal, vecindad espacial y codificación directa. El multiplexado temporal permite utilizar diferentes patrones a lo largo del tiempo, lo que permite utilizar patrones simples. La codificación basada en vecindad espacial crea un código para cada zona del patrón a partir de los códigos de las zonas vecinas. La codificación directa define un código según paletas de colores o escalas de grises.

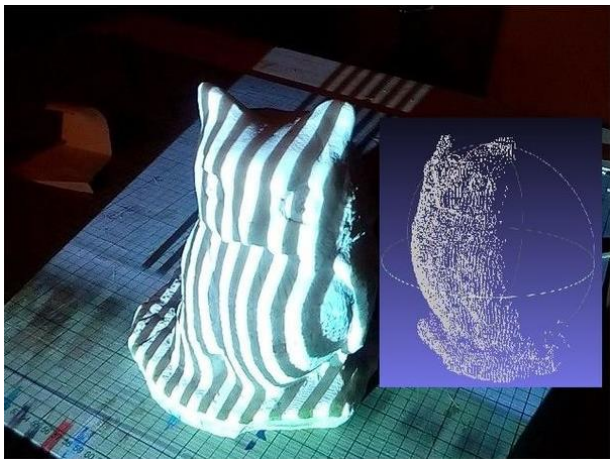


Fig. 2.5 – Ejemplo de patrón de luz estructurada

2.2.3. Tiempo de vuelo

La tecnología de tiempo de vuelo (Time of Flight, ToF) se basa en la emisión de ondas IR moduladas y la observación de las ondas reflejadas. Midiendo el desfase entre la onda emitida y la onda recibida podemos calcular la distancia a la que se encuentra un objeto.

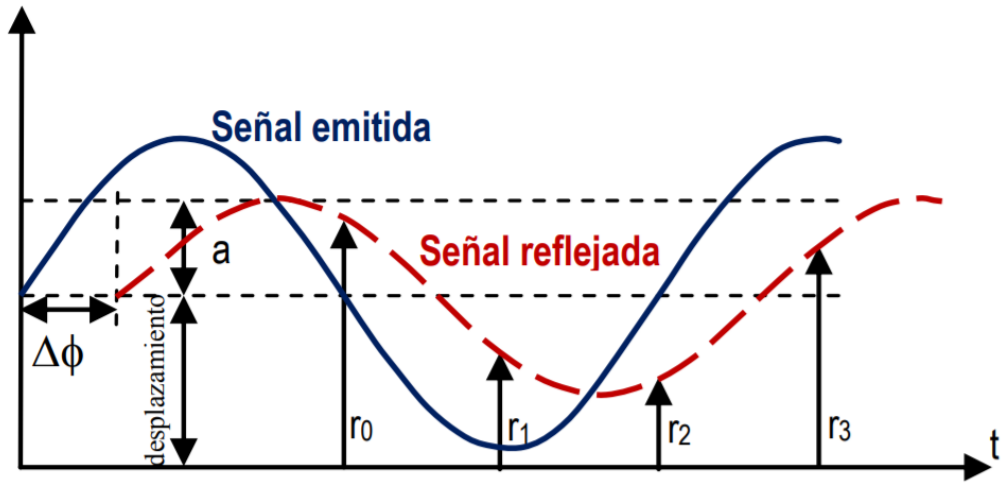


Fig. 2.6 – Principio de funcionamiento de la tecnología Tiempo de Vuelo (ToF) [6]

Para medir la distancia se utilizan varias muestras por medida, con señales desfasadas 90º. Entonces, la distancia se calcula de la siguiente forma:

$$d = \frac{c \cdot \Delta\phi}{4 \cdot f_{mod}} \tag{2.1}$$

Donde c es la velocidad de la luz, Δϕ el desfase entre ondas y f_{mod} la frecuencia de modulación.

El hecho de que la distancia se mida utilizando el desfase, que se repite cada 2π, provoca que se produzca aliasing y por lo tanto que exista una distancia máxima medible. A partir de (2.1) podemos calcular la distancia máxima medible, sabiendo que el desfase máximo es 2π:

$$d_{máx} = \frac{c}{2 \cdot f_{mod}} \tag{2.2}$$

Por tanto, podemos calcular la distancia máxima medible en función de la frecuencia de modulación utilizada, como puede observarse en la Tabla 2.1 – Distancia máxima en función de la frecuencia de modulación. Aunque podemos pensar que es mejor utilizar frecuencia de modulación bajas, ya que permiten calcular distancias mayores, debemos tener en cuenta que la onda recibida debe tener la suficiente intensidad para poder ser captada por el sensor y que, además, la precisión disminuye conforme más baja es la frecuencia de modulación.

f_{mod} (MHz)	40	35	30	25	20	18	16	14	12
d (m)	3.75	4.28	5	6	7.5	8.33	9.37	10.71	12.5
f_{mod} (MHz)	10	8	6	4	3	2	1	0.5	0.1
d (m)	15	19.75	25	37.5	50	75	150	300	1500

Tabla 2.1 – Distancia máxima en función de la frecuencia de modulación

Las tecnologías de ToF más avanzadas utilizan sistemas multifrecuencia, que permiten aumentar la distancia máxima medible sin reducir la modulación en frecuencia.

Respecto a la visión estéreo, la tecnología de tiempo de vuelo tiene un menor error de resolución al medir la distancia, ya que se puede aumentar la energía de emisión. La visión en estéreo tiene un error de resolución que depende del cuadrado de la distancia. La visión estéreo tiene algunas ventajas, ya que no necesita cámaras especializadas.

Respecto a la luz estructurada, la tecnología de tiempo de vuelo permite tener un frame-rate mayor, ya que la tecnología de luz estructurada necesita varios patrones para extraer una sola imagen de profundidad. Un frame-rate bajo implica que los objetos de la escena no pueden moverse muy rápido. La ventaja de la técnica de luz estructurada que se puede conseguir una gran resolución espacial sin necesidad de utilizar cámaras especializadas. En comparación, la tecnología ToF es menos sensible a las condiciones de iluminación y es más compacto.

2.2.4. Otras técnicas

Otras técnicas que se ha comentado para medir la profundidad es la interferometría. Esta técnica hace interferir la luz emitida por los objetos de la escena con la luz generada por otras fuentes. Por tanto, en la cámara de visión se obtiene un patrón de interferencia. Proporciona una alta precisión, pero solo sirve para medir distancias pequeñas, del orden de cm, por lo que se suelen utilizar en procesos de calidad.

3. ELEMENTOS DE LA CÉLULA ROBÓTICA

3.1. ROBOT ABB IRB 120

La célula de trabajo del laboratorio consta de un robot ABB IRB 120, un robot industrial de seis ejes, con una carga útil de 3 kg y está específicamente diseñado para aplicaciones de automatización flexible e industria 3C. Toda la información sobre el robot viene recogida en [7].

El robot viene equipado con el controlador IRC5 y software de control de robots RobotWare. Además, puede equiparse con software opcional: aplicaciones de adhesivos y soldadura, comunicación en red, procesamiento multitarea, control de sensores…

El robot con los seis ejes se puede observar en la Fig. 3.1.Tabla 2.1

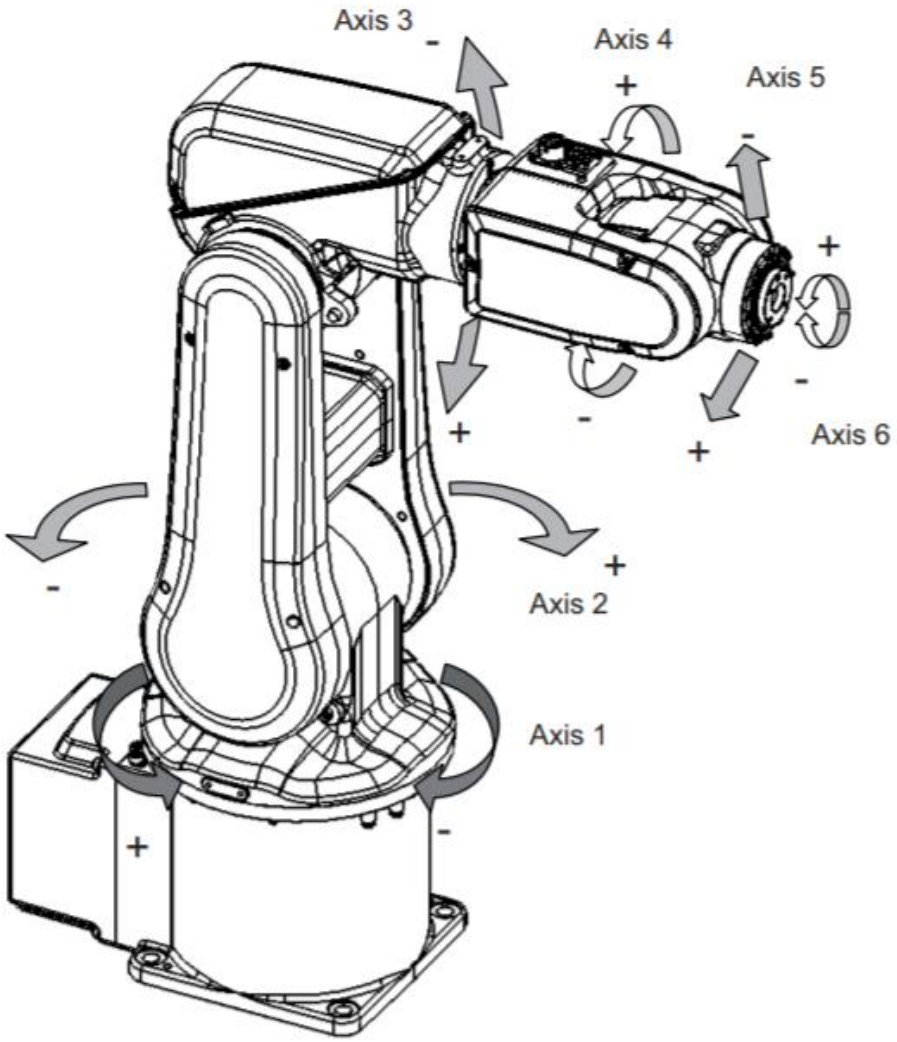


Fig. 3.1 – Robot ABB IRB 120

3.1.1. Características generales

La Tabla 3.1 recoge las características generales del robot.

Peso [kg]	25
Alcance [mm]	580
Capacidad de carga [kg]	3
Montaje	Suelo, pared o ángulo inclinado

Tabla 3.1 – Características generales del robot ABB IRB 120

3.1.2. Dimensiones

Las dimensiones del robot en mm se muestran en la Fig. 3.2.

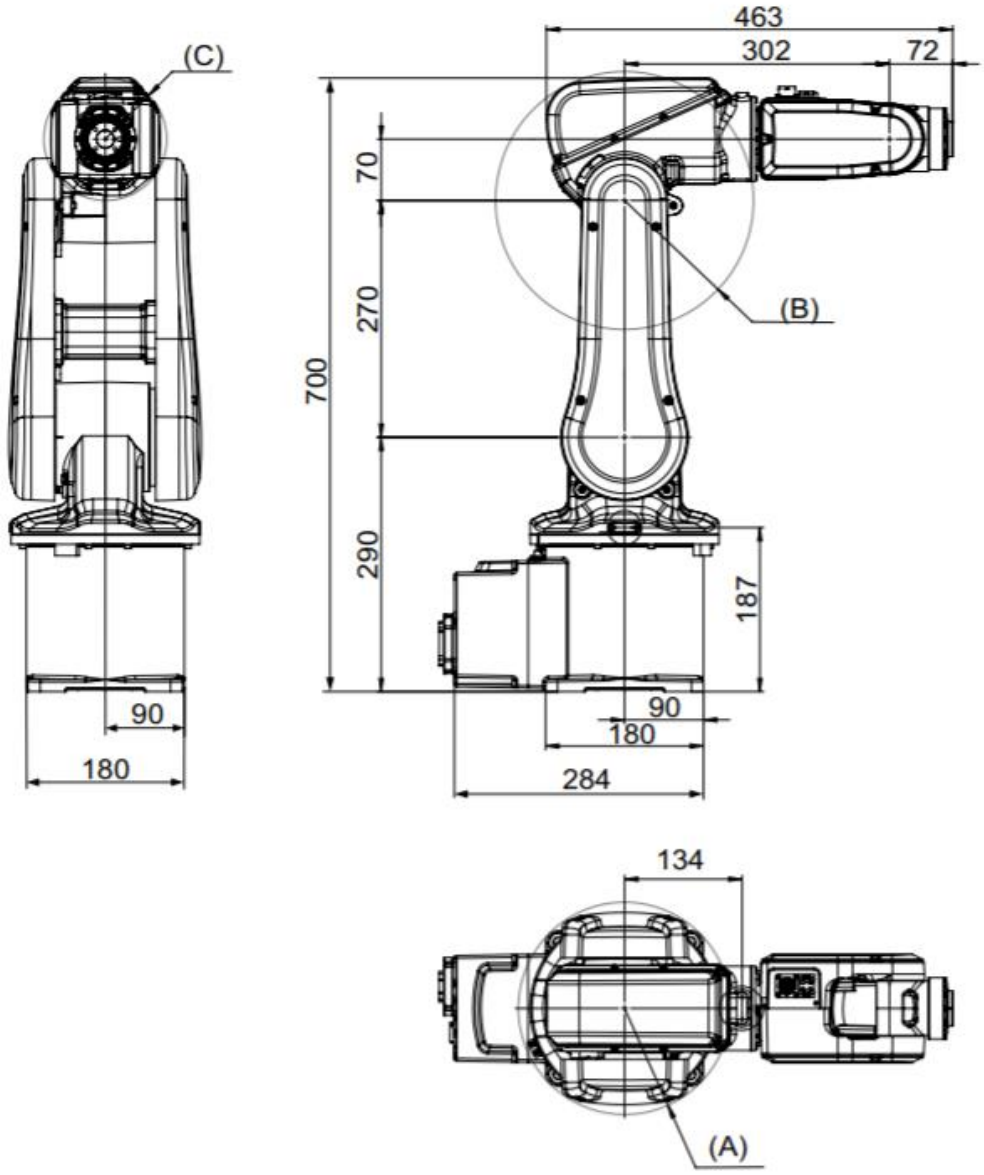


Fig. 3.2 – Dimensiones del robot

3.1.3. Movimiento del robot

3.1.3.1. Área de trabajo

La Fig. 3.3 y la Fig. 3.4 muestra el área de trabajo sin restricciones y el radio de giro del robot. Las posiciones extremas del brazo están especificadas desde el centro de la muñeca y las medidas están en mm.

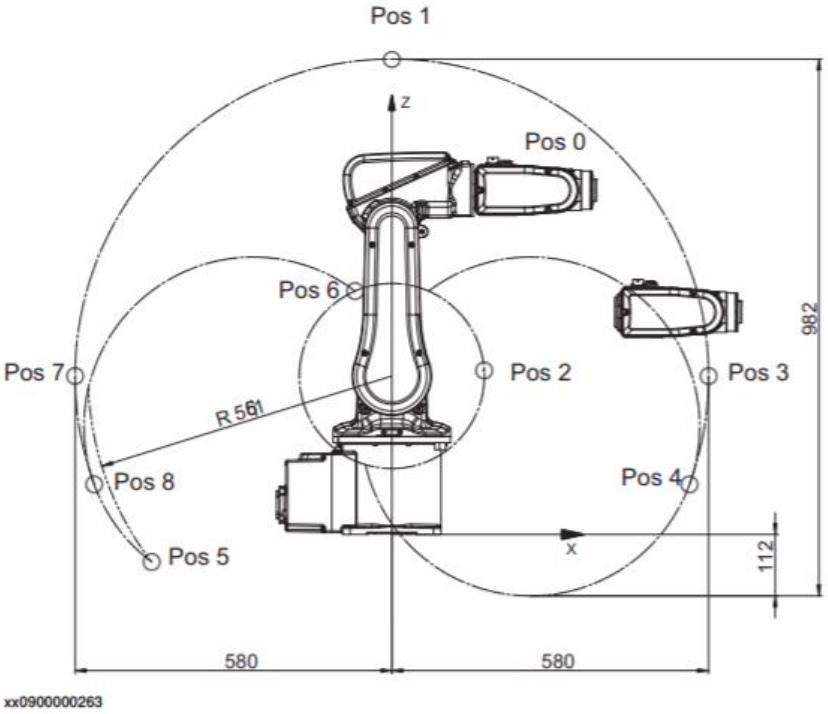


Fig. 3.3 – Área de trabajo del robot [7]

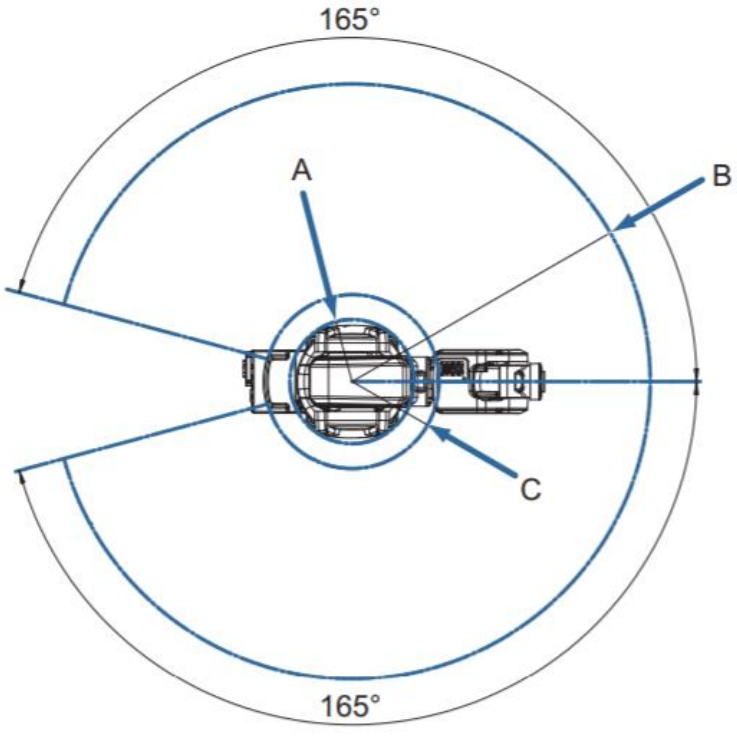


Fig. 3.4 – Radio de giro del robot

3.1.3.2. Tipo y rango de movimiento

En la Tabla 3.2 se especifica el tipo de movimiento de cada eje según la Fig. 3.2 y los límites de movimiento de cada eje.

Ubicación del movimiento	Tipo de movimiento	Rango de movimiento
Eje 1	Rotación	+165º a -165º
Eje 2	Mov. del brazo	+110º a -110º
Eje 3	Mov. del brazo	+70º a -110º
Eje 4	Mov. de la muñeca	+160º a -160º
Eje 5	Mov. de doblado	+120º a -120º
Eje 6	Mov. de giro	+400º a -400º

Tabla 3.2 – Tipos y límites de movimiento

3.1.3.3. Velocidad de los ejes

En la Tabla 3.3 se especifica la velocidad máxima de cada eje. La resolución es aproximadamente 0,01º en cada eje.

Eje 1	Eje 2	Eje 3	Eje 4	Eje 5	Eje 6
250 º/s	250 º/s	250 º/s	320 º/s	320 º/s	420 º/s

Tabla 3.3 – Velocidades de los ejes del robot

3.1.3.4. Rendimiento según normativa ISO 9283

La norma ISO 9283 marca los criterios de análisis de prestaciones y los métodos de ensayo relacionados. Los datos de repetibilidad y exactitud están referidos a carga máxima, offset máximo y velocidad de 1.6 m/s, en el plano de prueba ISO inclinado, con los seis ejes en movimiento. Debido a que los datos son resultados de la media obtenido con un número reducido de robots, pueden obtenerse resultados distintos dependiendo de la parte del área de trabajo en la que el robot este posicionándose, la velocidad, la configuración del robot, la dirección de aproximación y carga…

AP, RT, AT y RT se miden de acuerdo con la Fig. 3.5. La descripción de las letras se encuentra en la Tabla 3.4.

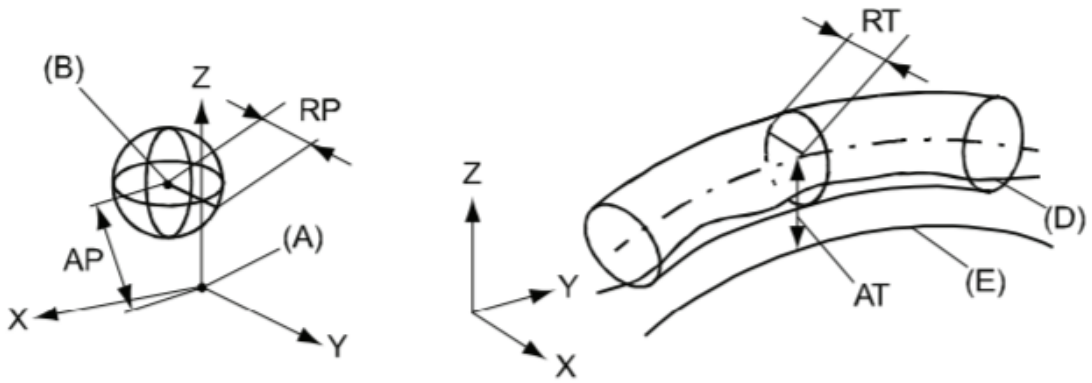


Fig. 3.5 – Repetibilidad y exactitud del robot

Posición	Descripción	Posición	Descripción
A	Posición programada	E	Trayectoria programada
B	Posición media durante la ejecución del programa	D	Trayectoria actual durante la ejecución del programa
AP	Distancia media desde la posición programada	AT	Desviación máxima de E con respecto a la trayectoria media
RP	Tolerancia de la posición B en caso de posicionamiento repetido	RT	Tolerancia de la trayectoria con la ejecución repetida del programa

Tabla 3.4 – Descripción de las letras de la Fig. 3.5

Los valores de AP, RP, AT y RT se encuentran en la Tabla 3.5.

Repetibilidad de pose, RP [mm]	0.01
Exactitud de pose, AP [mm]	0.02
Repetibilidad de trayectoria lineal, RT [mm]	0.07-0.016
Exactitud de trayectoria lineal, AT [mm]	0.21-0.38

Tabla 3.5 – Valores de repetibilidad y exactitud del robot

3.2. CONTROLADOR IRC5

El robot viene equipado con el controlador de ABB IRC5. Toda la información sobre el controlador viene recogida en [8]. El controlador IRC5 (Industrial Robot Controller) pertenece a la quinta generación de controladores de ABB y contiene todas las funciones necesarias para mover y controlar el robot. El controlador estándar cuenta con un único armario, pero también puede dividirse en dos, uno para cada módulo del controlador.

El controlador está dividido en dos módulos:

- Módulo de control, que contiene toda la electrónica de control como el ordenador principal, las tarjetas E/S y la memoria flash.
- Módulo de accionamiento, que contiene toda la electrónica de alimentación que alimenta los motores del robot. Un módulo de accionamiento puede contener nueve unidades de accionamiento y manejar seis ejes, y hasta dos o tres más en función del modelo.

Se puede controlar más de un robot con el mismo controlador (opción MultiMove), pero es necesario añadir módulo de accionamiento por cada robot añadido. El módulo de control es el mismo para todos los robots.

En la Fig. 3.6 se muestra la botonera del controlador y en la Tabla 3.6 se especifica la funcionalidad de los botones.

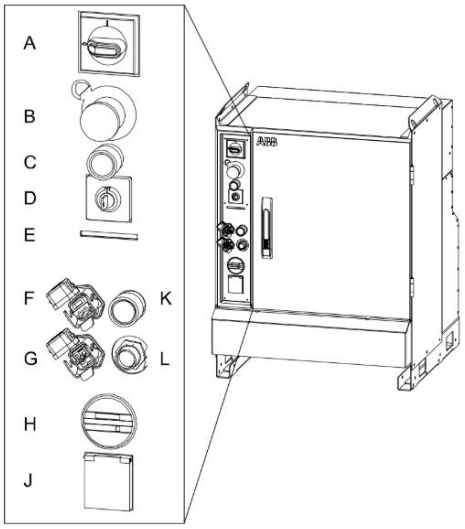


Fig. 3.6 – Controlador IRC5

A	Interruptor principal
B	Paro de emergencia
C	Motores ON
D	Selector de modo
E	LEDs de la cadena de seguridad (opción)
F	Puerto USB (opción)
G	Puerto de servicio para PC (opción)
H	Contador de tiempo de funcionamiento (opción)
J	Toma de servicio 115V/230V, 200W (opción)
K	Botón de conexión en caliente (opción)
L	Conector para FlexPendant

Tabla 3.6 – Funcionalidad de la botonera

3.2.1. FlexPendant

El controlador viene equipado con un dispositivo FlexPendant (TPU o unidad de programación) que sirve para manejar muchas de las funciones de uso del robot, como ejecutar programas, mover el robot, crear y editar programas…

Se compone de una parte hardware, como la pantalla y los botones, y una parte software. Se conecta al controlador a través de un cable con conector integrado.

En la Fig. 3.7 se pueden ver las partes principales del FlexPendant, especificadas en la Tabla 3.7.

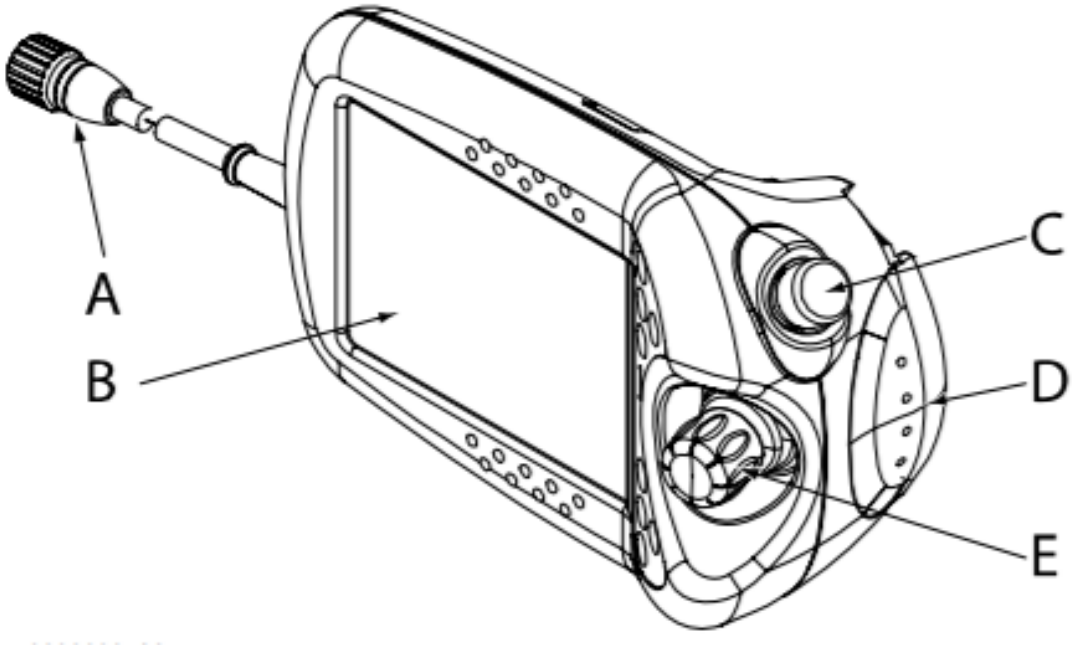


Fig. 3.7 – FlexPendant

Letra	Descripción
A	Conector
B	Pantalla táctil
C	Pulsador de paro de emergencia
D	Dispositivo de habilitación
E	Joystick

Tabla 3.7 – Partes principales del FlexPendant

La botonera del FlexPendant (Fig. 3.8) cuenta con cuatro botones programables y cuatro preprogramados, cuya funcionalidad se especifica en la Tabla 3.8.

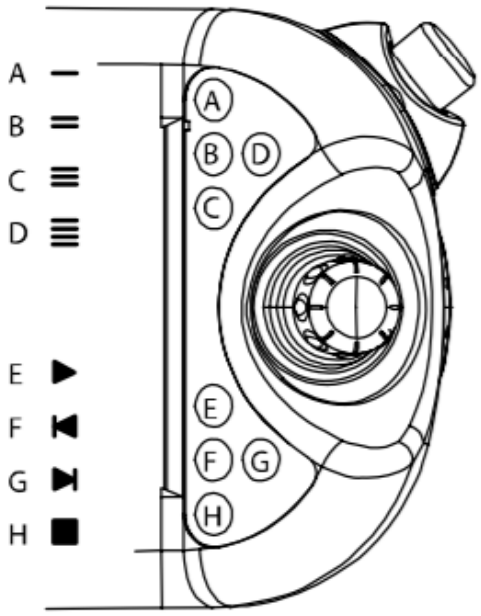


Fig. 3.8 – Botonera del FlexPendant

Tecla	Descripción
A-D	Teclas programables de 1 a 4.
E	Botón INICIAR. Inicia la ejecución del programa
F	Botón RETROCEDER un paso. Ejecuta el programa una instrucción hacia atrás
G	Botón AVANZAR un paso. Ejecuta el programa una instrucción hacia delante
H	Botón DETENER. Detiene la ejecución del programa.

Tabla 3.8 – Funcionalidad de los botones del FlexPendant

En la Fig. 3.9 se muestra la pantalla del FlexPendant y en la Tabla 3.9 se describen sus elementos.

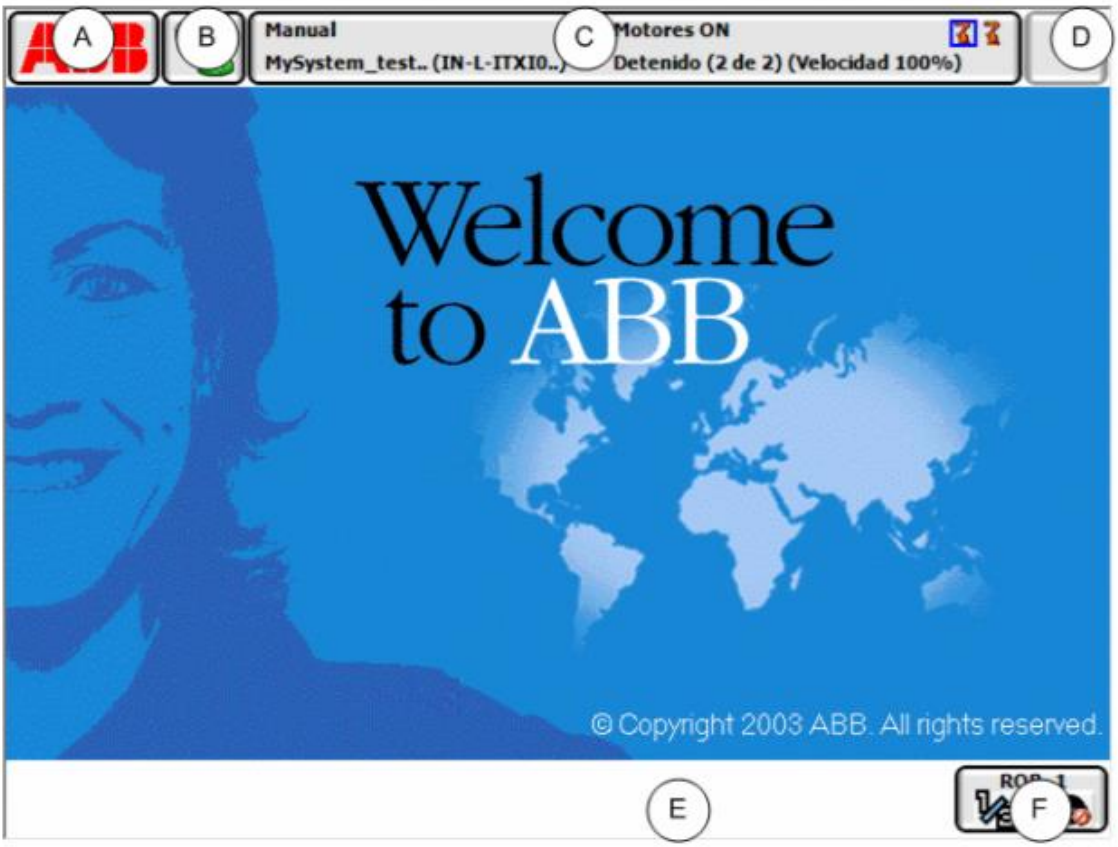


Fig. 3.9 – Pantalla del FlexPendant

Letra	Elemento
A	Menú ABB
B	Ventana de operador
C	Barra de estado
D	Botón Cerrar
E	Barra de tareas
F	Menú de configuración rápida

Tabla 3.9 – Elementos de la pantalla del FlexPendant

El *menú ABB* (A) permite seleccionar la ventana de producción, el editor de programas, los datos de programa, las copias de seguridad y restauración, el panel de control, el registro de errores y la información del sistema, entre otras opciones. Pueden abrirse varias vistas a la vez, pero solo se puede trabajar con una.

La *ventana del operador* (B) muestra mensajes del programa del robot, por ejemplo, cuando el programa necesita una respuesta del operador para continuar ejecutándose.

La *barra de estado* (C) muestra información importante acerca del estado del sistema, como por ejemplo el modo de funcionamiento, si los motores están encendidos o apagados, el estado del programa…

El *botón cerrar* (D) cierra la vista o la aplicación que este activa en ese momento.

La *barra de tareas* (E) muestra todas las vistas del menú ABB abiertas y se utilizar para cambiar entre ellas.

El *menú de configuración rápida* (F) contiene valores sobre el movimiento y la ejecución de programas.

3.3. MICROSOFT KINECT V2

La cámara de Microsoft Kinect V2 es el sensor utilizado para realizar la detección de personas y el reconocimiento del esqueleto humano. La Kinect V2 es la segunda versión de la cámara lanzada en 2013 como complemento para la consola Xbox One. Posteriormente, fue descatalogada en 2017 por su poca popularidad.

Pese a que fue lanzada como complemento de la Xbox One, sus características (detección de personas, gestos, voz, medida de la profundidad…) han permitido su uso en aplicaciones muy variadas, entre las que destaca el tema de estudio: la robótica.

El dispositivo cuenta con una cámara RGB, un sensor de infrarrojos y un arreglo de micrófonos. El sensor de infrarrojos permite medir la distancia mediante la tecnología de tiempo de vuelo (Time of Flight, ToF), estudiada anteriormente. También cuenta con un adaptador que permite utilizar la cámara con un ordenador con un sistema operativo Windows.

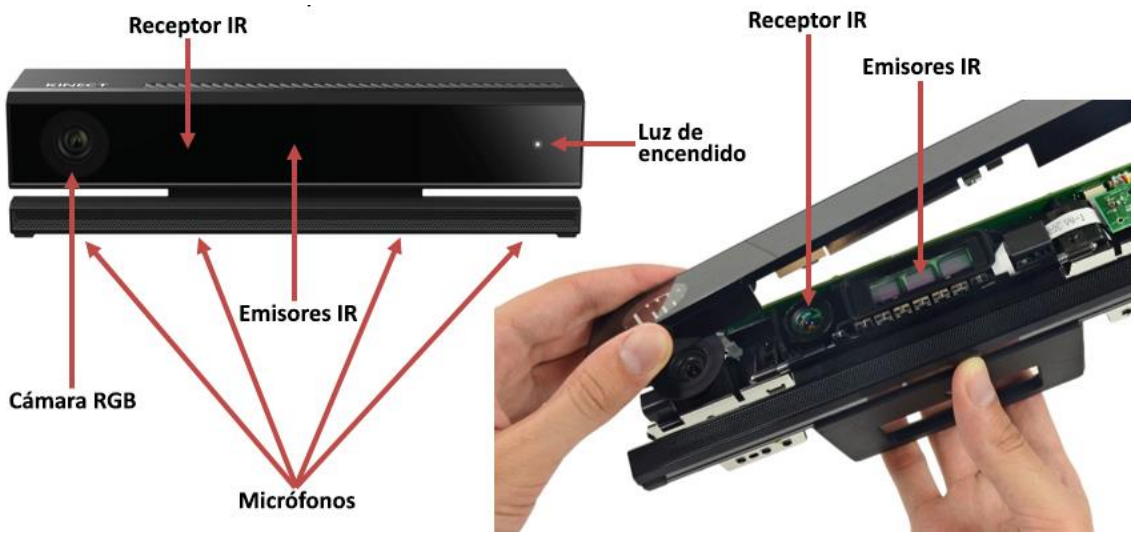


Fig. 3.10 – Kinect V2

Fig. 3.11 – Adaptador Kinect for Windows

3.3.1. Características de la Kinect V2.

Las características generales vienen resumidas en la Tabla 3.10 [9]. Como se puede observar, la cámara RGB tiene una resolución de 1080p mientras que la cámara RGB tiene una resolución cuatro veces menor, lo que limita sus aplicaciones.

Campo de visión (horizontal x vertical) [º]	70 x 60
Resolución RGB [pixeles]	1920 x 1080 30Hz
Resolución IR [pixeles]	512 x 424 30Hz
Rango de profundidad [m]	0.5 – 4.5
Articulaciones por esqueleto	25
Esqueletos reconocidos	6
USB	3.0
Sistema operativo	Windows 8 o superior

Tabla 3.10 – Características de la Kinect

El sensor IR de la Kinect tiene una resolución de 512 x 424 píxeles y utiliza diodos CMOS. Los parámetros del sensor se encuentran en la Tabla 3.11. Cada píxel tiene dos fotodiodos controlados por la misma señal de reloj, de forma que mientras un fotodiodo esta encendido, el otro está apagado. Operando con ambas señales puede obtenerse la profundidad, la imagen de escala de grises a luz ambiente y la imagen a escala de grises independiente de la luz ambiente [10].

Tecnología	TSMC 0.13 1P5M
Resolución [píxeles]	512 x 545
Rango dinámico	> 2500 = 68 dB
Modulación de contraste	68% @ 860 nm @ 50MHz
Modulación de frecuencia	10-130 MHz
Promedio de modulación en frecuencia	80 MHz
Incertidumbre de medición	< 0.5% del rango
Longitud de onda de operación	860 nm
ADC	2 Gs/s
Reflectividad	15-95%
Potencia	2.1 W

Tabla 3.11 – Características del sensor IR [10]

El sensor utiliza un filtro de banda estrecha que elimina toda la luz salvo el rango de longitud de onda de 580nm, que corresponde a la longitud de onda de funcionamiento.

3.3.2. Precisión de medida de la Kinect

Como todo sistema físico de medición, la Kinect está sujeta a un rango de medición y unos errores y precisiones variable dentro de este rango. Se va a destacar la influencia de la distancia, los materiales y los colores y, por último, la eficiencia en exteriores.

3.3.2.1. Influencia de la distancia

El error de profundidad posee la forma de un cono elíptico de 70 x 60º, de acuerdo con el campo de visión de la cámara, y es mayor conforme nos alejamos del eje del cono y de la cámara.

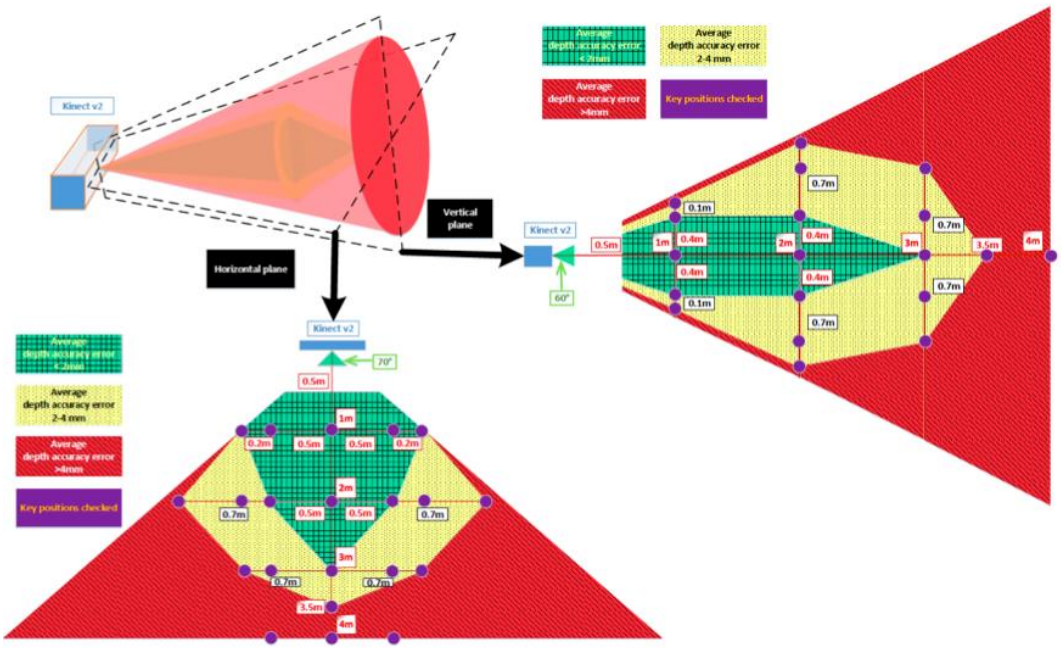


Fig. 3.12 – Distribución del error de medición según distancia [11]

En las zonas verdes el error de medición es de menos 2 mm y aumenta al alejarse hasta más de 4mm en las zonas rojas.

3.3.2.2. Influencia del material y del color.

Se estudia la influencia del material y el color de los objetos en las medidas de profundidad realizadas con la Kinect. El estudio realizado en [12] concluye que el color no tiene gran influencia ya que para todos los colores testados existía una desviación típica de 1 mm. De la influencia de los materiales se concluye que los materiales más reflectantes, como el aluminio, son percibidos más cerca mientras que superficies más ásperas, como la madera, con percibidas más lejanas. Los materiales muy reflectantes saturan la cámara y no son detectados a ninguna distancia.

Sin embargo, el estudio realizado en [13] concluye que los objetos más oscuros y de material más reflectante reflejan una onda de intensidad muy baja y son percibidos más lejos, habiendo hasta 6cm de error. El estudio no concluye nada sobre las superficies ásperas debido al ruido inherente a la nube de puntos.

Ambos estudios están realizados colocando diferentes colores y materiales en una superficie plana alineada con la cámara. En el primer estudio la cámara está situada a 775mm de la cámara mientras que en el segundo no se especifica. Una posible razón de los resultados discordantes entre los dos estudios puede ser que no se realizaran bajo las mismas condiciones (temperatura, distancia de la cámara…).

3.3.2.3. Influencia de exteriores

El estudio realizado en [13] también analiza la capacidad de funcionamiento de la Kinect cuando está expuesta a la luz del sol. La conclusión a la que se llega es que la Kinect puede trabajar con luz solar mientras ésta no ilumine directamente la cámara, ya que se puede desconectar la cámara del ordenador. Las capturas realizadas con luz solar tienen más pixeles flotantes en los límites del campo de visión y el número de puntos disminuye con la intensidad de la luz.

3.3.3. Viabilidad de captura dinámica

El trabajo de Juan Felipe Montesinos [5] lleva a cabo un análisis de la viabilidad de la captura dinámica de la Kinect teniendo en cuenta tres cuestiones:

- La captura de imagen por parte de la Kinect asume un entorno estático. El sistema de procesado de la Kinect realizada varias medidas de profundidad por cada píxel, para lo cual se supone que, durante el proceso de adquisición de la imagen, cada píxel se corresponde a un punto y este no cambia durante el proceso. El hecho de realizar una captura dinámica rompe esta premisa, por lo que existe una velocidad máxima que depende de la profundidad del objeto medido.
- Así mismo, también existe una limitación en el tamaño mínimo de objeto que puede medir la Kinect, debido al procesamiento de imagen.
- También debe tenerse en cuenta el contorno de la imagen, que es lugar geométrico de los puntos que tienen dos medidas de profundidad simultáneamente, la del objeto y la del fondo que se encuentra detrás. Esto provoca que se pueda mezclar la información y haya inconsistencias en las medidas.

Debido al modelo que utiliza la Kinect, cuanto más cerca está un objeto del cámara más tamaño ocupará en la imagen, lo que facilita el procesado y disminuye el error de distancia cometido. Por contra, la velocidad máxima a la que se pueda desplazar el objeto sin producir desenfoque es menor y limita el tamaño máximo de un objeto en la imagen.

Por ejemplo, para una persona de 1,90m a 2,5m de la cámara podría moverse a 20 cm/s. Las limitaciones de la Kinect se ven suplidas por la cámara RGB, ya que, aunque no sea posible obtener datos sobre la ejecución, si es posible recuperar el tracking del objeto antes y después del movimiento.

La distancia óptima es aquella lo más cercana posible sin producir desenfoque y captando el objeto entero.

3.3.4. Sistema de referencia de la cámara

Para programar la cámara en los próximos apartados es necesario conocer sus sistemas de referencia.

Fig. 3.13 – Sistema de referencia del sensor IR de la Kinect

El plano paralelo a la cámara está formado por los ejes X e Y mientras que el eje Z es perpendicular. La distancia se medirá entonces en el eje Z.

4. ENTORNO DE DESARROLLO SOFTWARE

4.1. INTRODUCCIÓN

El software utilizado para este trabajo fin de grado es el siguiente:

Para la parte de desarrollo de la aplicación:

- Microsoft Kinect SDK 2.0
- ABB PC SDK
- Microsoft Visual Studio 2017

Para la parte de simulación de la célula robótica:

- ABB RobotStudio

4.2. MICROSOFT KINECT SDK 2.0

El software de Microsoft Kinect SDK (Software Development Kit) 2.0 es el utilizado para procesar los datos obtenidos por el sensor Kinect, de forma que se pueda reconocer el esqueleto humano.

El SDK incorpora los drivers necesarios para utilizar la Kinect en un ordenador con sistema operativo Windows, así como las APIs necesarias para poder programar la Kinect. El SDK también incorpora algunas herramientas como Kinect Studio, que permite ver, grabar y reproducir los datos de Kinect para probar y debuggear aplicaciones, ya sea la imagen RGB, infrarroja o profundidad, y Visual Gesture Builder, que permite crear una base de datos de gestos, reconocer gestos en una imagen y enseñar a la aplicación distintos gestos.

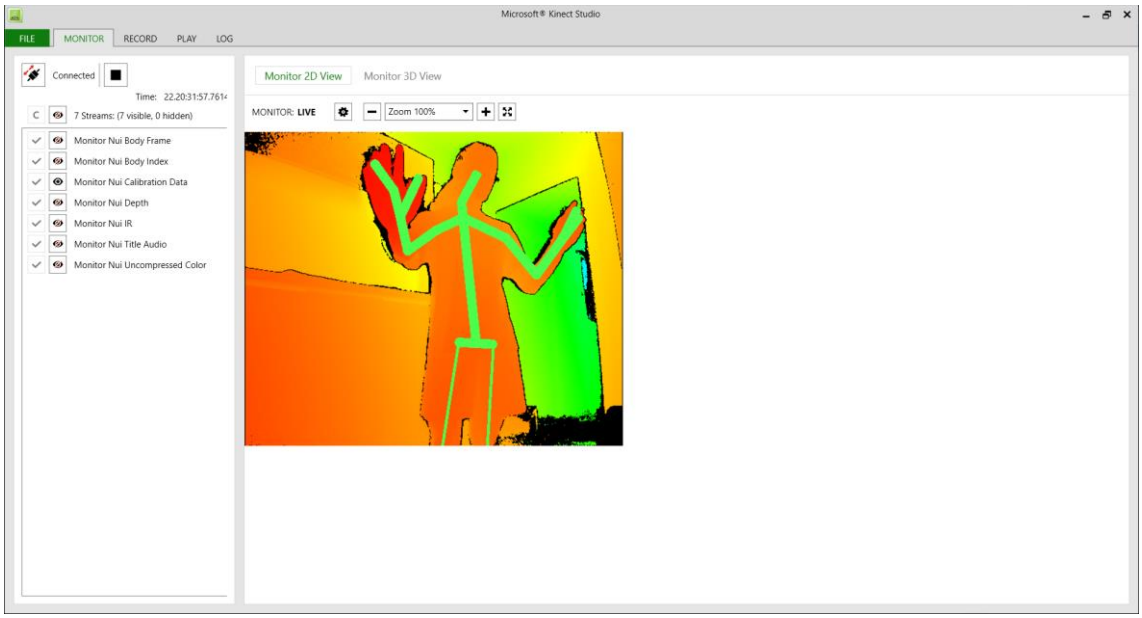


Fig. 4.1 – Interfaz de Kinect Studio

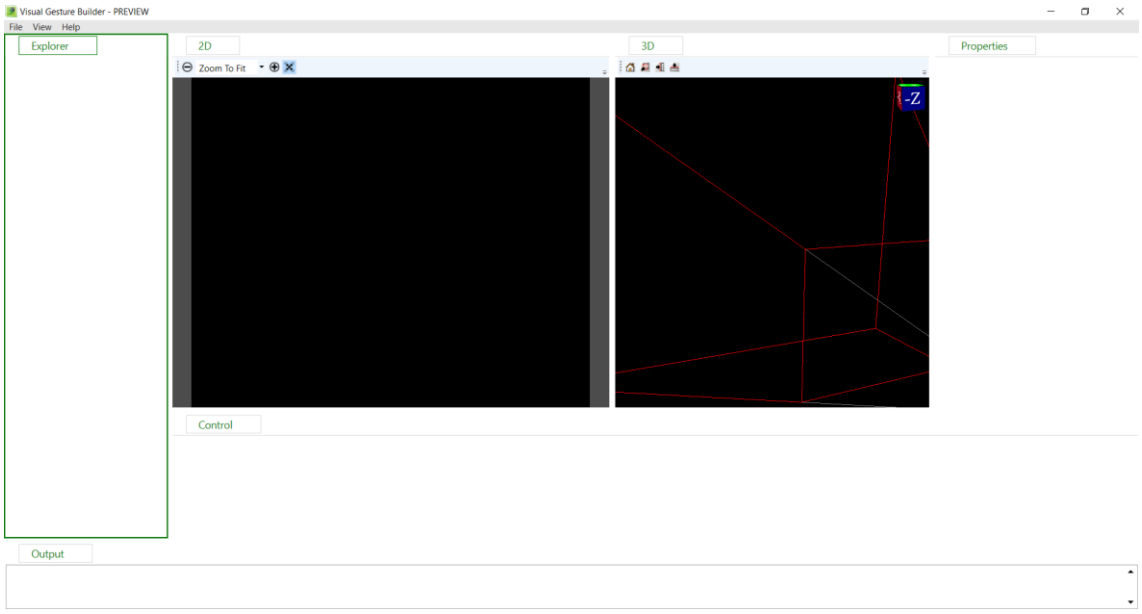


Fig. 4.2 – Interfaz de Visual Gesture Builder

Además, el SDK también incorpora una serie de ejemplos de código en diferentes lenguajes para programar la Kinect. Por ejemplo:

- Audio Basics: ejemplo de cómo obtener y mostrar el audio de los micrófonos de la Kinect.
- Body Basics: ejemplo de cómo obtener y mostrar el esqueleto humano.
- Color Basics: ejemplo de cómo obtener y mostrar la imagen RGB del sensor.
- Coordinate Mapping Basics: ejemplo de cómo eliminar el fondo (similar al croma)
- Discrete Gesture Basics: ejemplo de cómo detectar gestos almacenados en una base de datos de Visual Gesture Builder.

En concreto, partiremos del código de Body Basics para obtener los datos que necesitamos de posición y orientación de distintas partes del cuerpo humano.

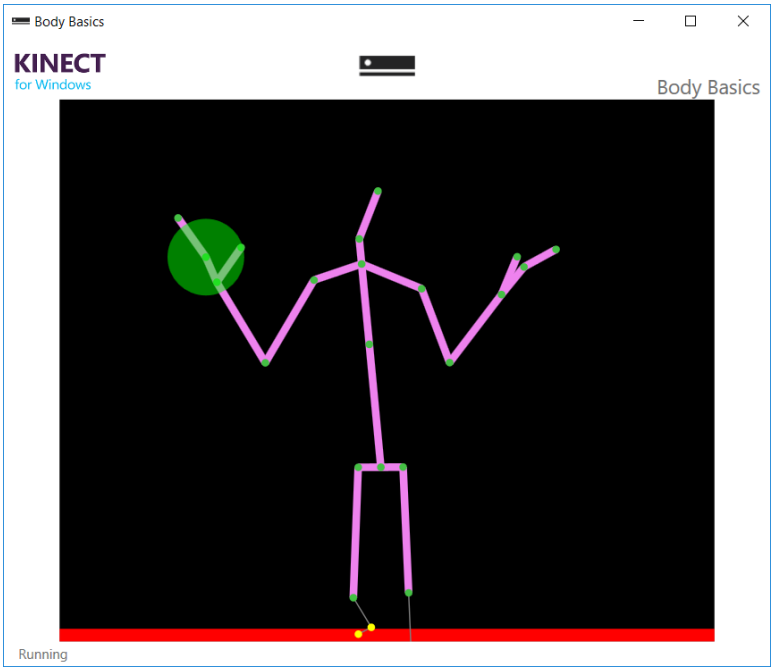


Fig. 4.3 – Body Basics

4.3. ROBOTSTUDIO

RobotStudio es un software de ABB que permite el modelado, la programación fuera de línea y la simulación de células robóticas. Toda la información de RobotStudio viene recogida en [14].

RobotStudio permite trabajar tanto con un controlador virtual como con un controlador real IRC5. Cuando se utiliza un controlador virtual o un controlador real sin conexión estamos trabajando en modo offline. Cuando se utiliza un controlador real conectado estamos trabajando en modo online.

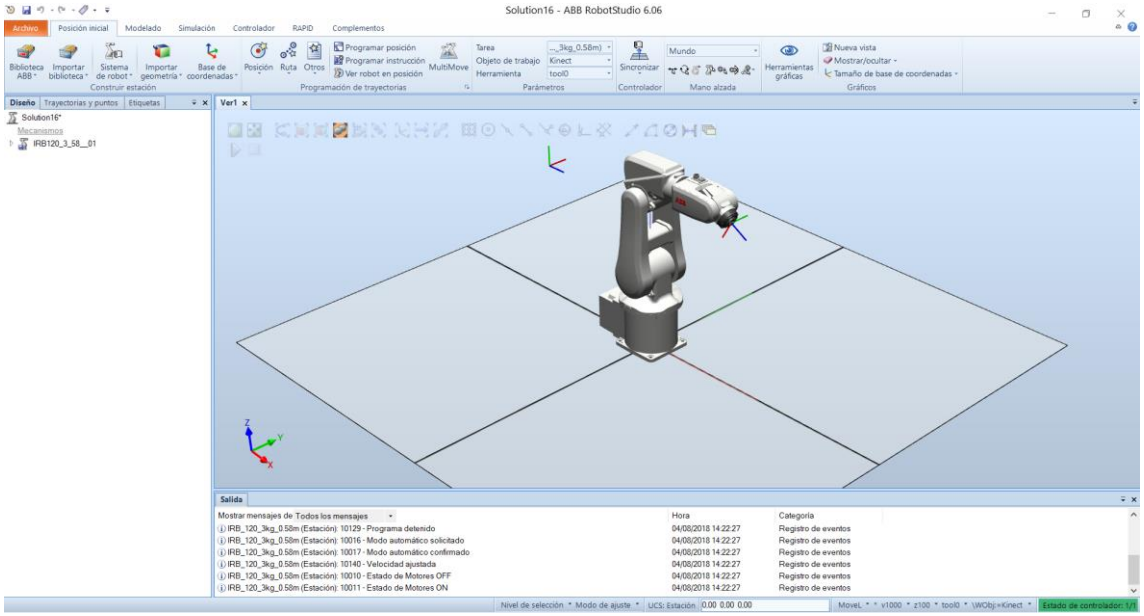


Fig. 4.4 – Interfaz de RobotStudio

4.3.1. Interfaz de usuario

En la Fig. 4.4 se puede ver la interfaz de RobotStudio en una estación ya creada. En la Fig. 4.5 se pueden ver la cinta, las pestañas y los grupos de la interfaz.

Fig. 4.5 – Cinta, pestañas y grupos de la interfaz

La pestaña *Archivo* (1) contiene las opciones necesarias para crear una estación, crear un sistema de robot, crear módulos de RAPID, conectarse a un controlador y más opciones de RobotStudio.

La pestaña *Posición inicial* (2) contiene todos los controles necesarios para construir una estación, con bibliotecas propias de ABB, programar posiciones y trayectorias y mover el robot.

La pestaña *Modelado* (3) contiene todos los controles necesarios para crear y agrupar componentes, crear cuerpos, mecanismos y realizar mediciones y operaciones de CAD.

La pestaña *Simulación* (4) contiene todos los controles necesarios para configurar, monitorizar y grabar simulaciones y analizar las E/S del sistema.

La pestaña *Controlador* (5) contiene todos los controles para crear y configurar un controlador, como reiniciar el controlador, acceder a la E/S del controlador... También permite la gestión de controladores reales.

La pestaña *RAPID* (6) contiene el Editor de RAPID utilizado para editar todas las tareas del robot que no sean las de movimiento. Permite editar, cargar programas y probar y depurar el código.

La pestaña *Complementos* (7) contiene todos los complementos de RobotStudio, como los PowerPacs o los AddIn, además de los paquetes instalados como RobotWare.

El menú de navegación *Diseño* es una representación jerárquica de los elementos físicos de la estación, como por ejemplo robots, herramientas, componentes, sólidos o sistemas de referencia.

El menú de navegación *Trayectorias y puntos* es una representación jerárquica de elementos no finitos, como por ejemplo los objetos de trabajo, las posiciones programadas, las trayectorias y los procedimientos.

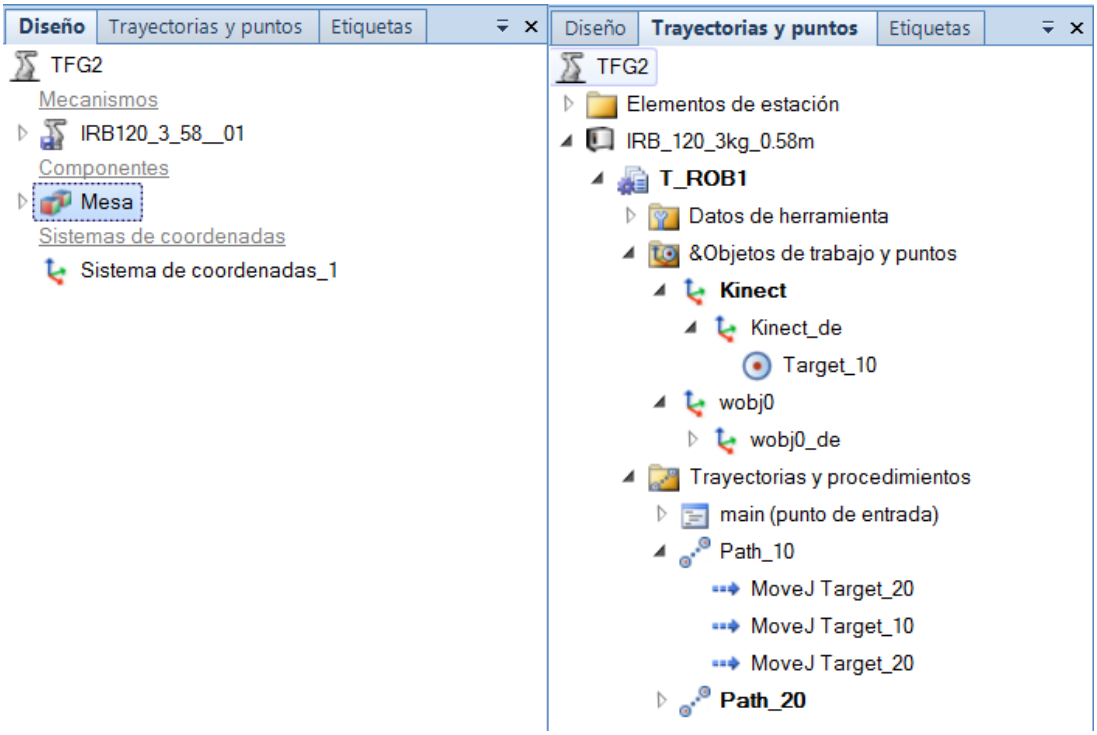


Fig. 4.6 – Menús de navegación Diseño (izda.) y Trayectorias y puntos (dcha.)

El menú de navegación *Controlador* de la pestaña *Controlador* es una representación jerárquica de los controladores de la estación y su configuración.

El menú de navegación *Archivos* de la pestaña *RAPID* permite gestionar los archivos RAPID y las copias de seguridad. También permite abrir y editar los módulos de RAPID.

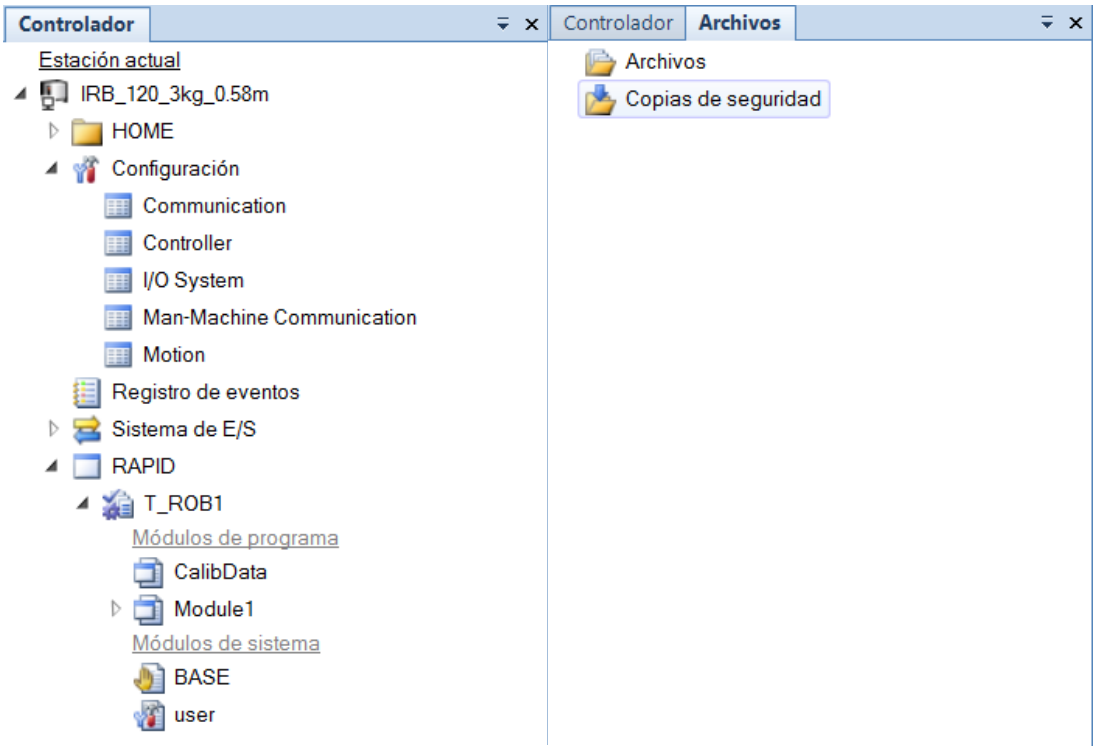


Fig. 4.7 – Menús de navegación Controlador (izda.) y Archivos (dcha.)

La ventana *Salida* muestra información sobre los eventos que se producen en la estación, como por ejemplo el inicio o la detención de la simulación, además de indicar la hora a la que se produce y la categoría del evento.

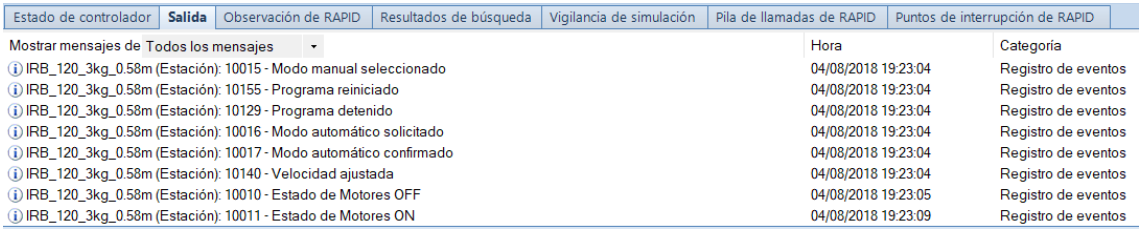


Fig. 4.8 – Ventana Salida

La ventana Estado del controlador muestra el estado de funcionamiento de los controladores, mostrando el nombre, el estado y el modo del controlador.

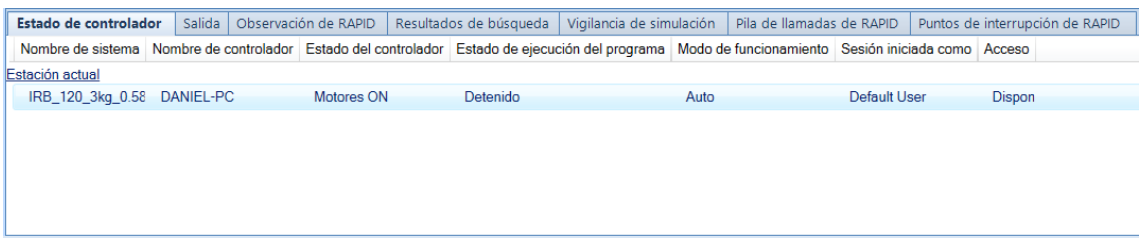


Fig. 4.9 – Ventana Estado de controlador

4.4. SDKS DE ABB

ABB deja a nuestra disposición una serie de herramientas de desarrollo. De las cinco opciones que aparecen en la página web del centro de desarrollo vamos a destacar tres de ellas:

- RobotStudio SDK
- PC SDK
- FlexPendant SDK

Las tres opciones utilizan el lenguaje de programación C#.

4.4.1. RobotStudio SDK

RobotStudio SDK es una herramienta de desarrollo que permite la creación de Add-Ins y de Smart Compoments.

Los Add-Ins son extensiones de una aplicación existente, en este caso RobotStudio, que permite añadir nuevas características a la interfaz del programa que no están disponibles por defecto. Podemos ver un ejemplo de Add-In en el trabajo de Arvid Boderg [15].

En este trabajo se crear un Add-In que implementa una nueva pestaña con tres botones. El primero de ellos sirve para abrir la ventana de la Kinect en la que aparece el esqueleto humano cuando es detectado y a partir de los gestos y el movimiento de la mano pueden moverse los brazos del robot colaborativo de ABB YuMi y crear objetivos. Con los otros dos botones se pueden crear trayectorias a partir de los objetivos guardados, según el brazo al que correspondan.

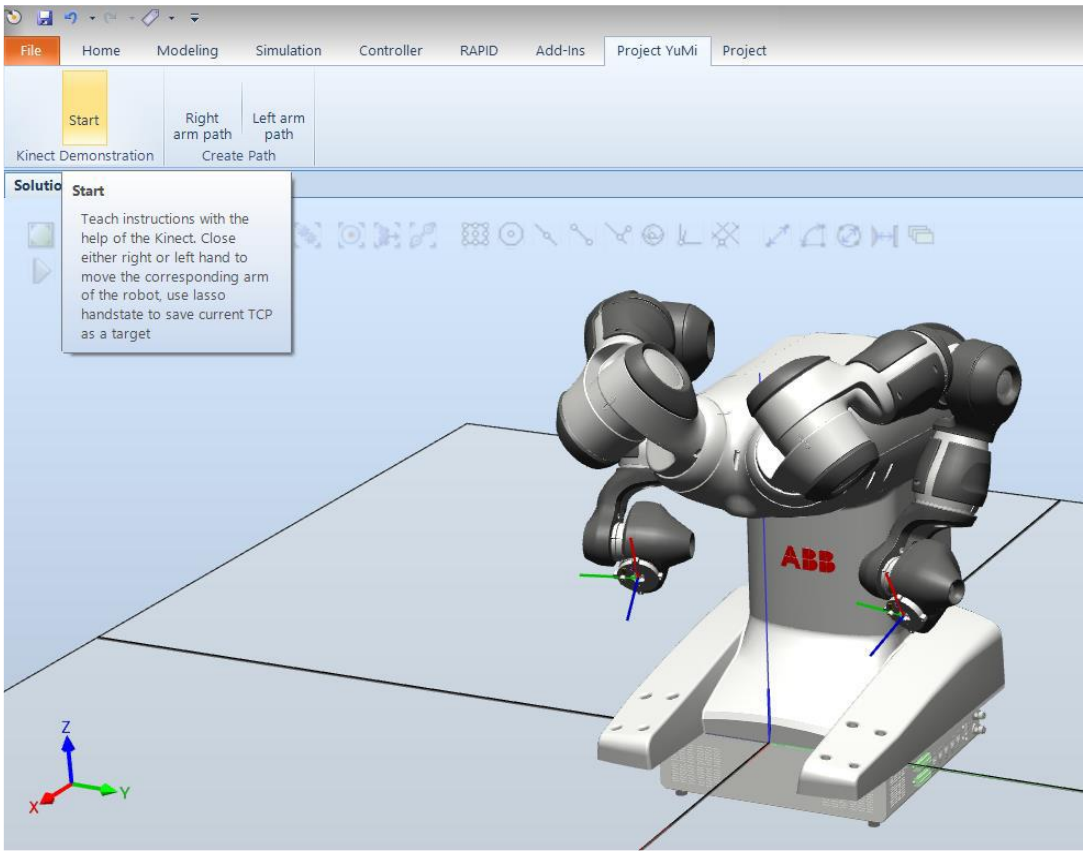


Fig. 4.10 – Ejemplo de Add-In [15]

Los Smart Component son objetos de estación que tienen comportamiento y estado. Algunos Smart Component simples pueden ser un sensor que detecte la presencia de objetos o un transportador que simule una línea de producción en la que los objetos se mueven por una cinta transportadora. Si el Smart Component es muy complejo o requiere una funcionalidad que no está disponible en los componentes que trae RobotStudio, es posible programarlo mediante RobotStudio SDK en Visual Studio.

El inconveniente de los Add-In es que al ser una aplicación que añade funcionalidades a RobotStudio no pueden ser implementados en un robot real.

4.4.2. PC SDK

La herramienta de desarrollo PC SDK permite crear aplicaciones de interfaces de operario personalizadas para el controlador IRC5 cuando sea necesaria una interfaz diferente de la interfaz general que viene por defecto. Las aplicaciones creadas con PC SDK se cargan en el PC.

El espacio de nombres Discovery permite buscar controladores en la red, detectar cuando se añaden o pierden controladores, crear un objeto de tipo Controlador o conectarse a un controlador de la red. El espacio de nombres RapidDomain permite acceder a los datos de RAPID del sistema.

Para obtener un dato RAPID primero debe crearse un objeto de tipo RapidData que almacena la ruta al dato. Para acceder al valor es necesario crear otro objeto que represente el valor. Dentro de RapidDomain hay varias clases que representan los tipos de datos de RAPID. Para crear el objeto necesario para representar el valor es necesario utilizar el método Value de RapidData y hacer un cast al tipo de dato correspondiente.

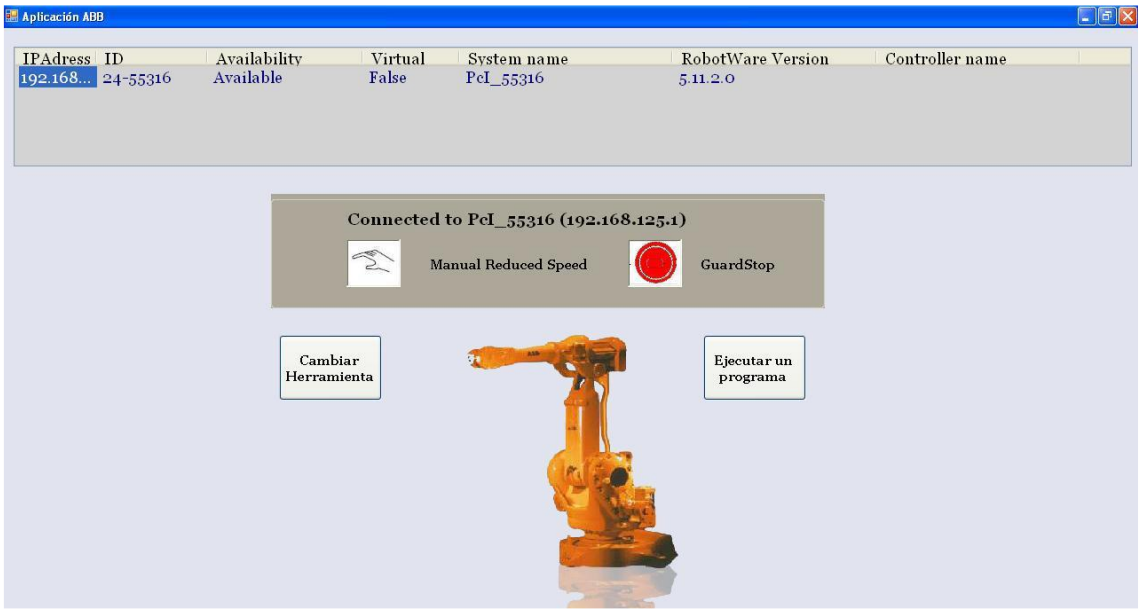


Fig. 4.11 – Ejemplo de una aplicación desarrollada por PC SDK [16]

4.4.1. FlexPendant SDK

La herramienta de desarrollo FlexPendant SDK es similar a PC SDK, ya que también sirve para crear interfaces de operario personalizadas para el controlador IRC5, pero estas se ejecutan y se cargan en el FlexPendant (3.2.1).

Otra de las diferencias entre PC SDK y FlexPendant SDK es que las aplicaciones de PC SDK son de cliente remoto y las aplicaciones de FlexPendant SDK son de cliente local, lo que implica que las aplicaciones de PC SDK deben solicitar explícitamente el control del controlador ya que los clientes remotos no tienen los privilegios de los clientes locales.

Una ventaja de las aplicaciones de PC SDK frente a las de FlexPendant SDK es que se puede monitorizar y acceder a varios controladores desde un mismo lugar, además de que el PC tiene mayor memoria.

Un ejemplo de aplicación desarrollada con PC SDK la encontramos en el trabajo de David Torremocha Ruano [16], que implementa una interfaz para el cambio de herramientas de un robot. Esta interfaz puede verse en la Fig. 4.11.

Una vez se han explicado las tres opciones de desarrollo de ABB se ha elegido PC SDK como la herramienta para desarrollar nuestra aplicación ya que esta puede ejecutarse en un robot real y además la Kinect debe ir conectada a un ordenador.

4.5. MICROSOFT VISUAL STUDIO

Visual Studio es un entorno de desarrollo integrado (IDE) que permite a los desarrolladores crear aplicaciones y juegos para Windows, aplicaciones para móvil en Android, iOS y Windows, aplicaciones de Azure, aplicaciones web, complementos de Office…

Visual Studio trabaja con cargas de trabajo según el lenguaje de programación y la aplicación que se quiera desarrollar. En función de la carga de trabajo elegida se instalarán unas herramientas u otras.

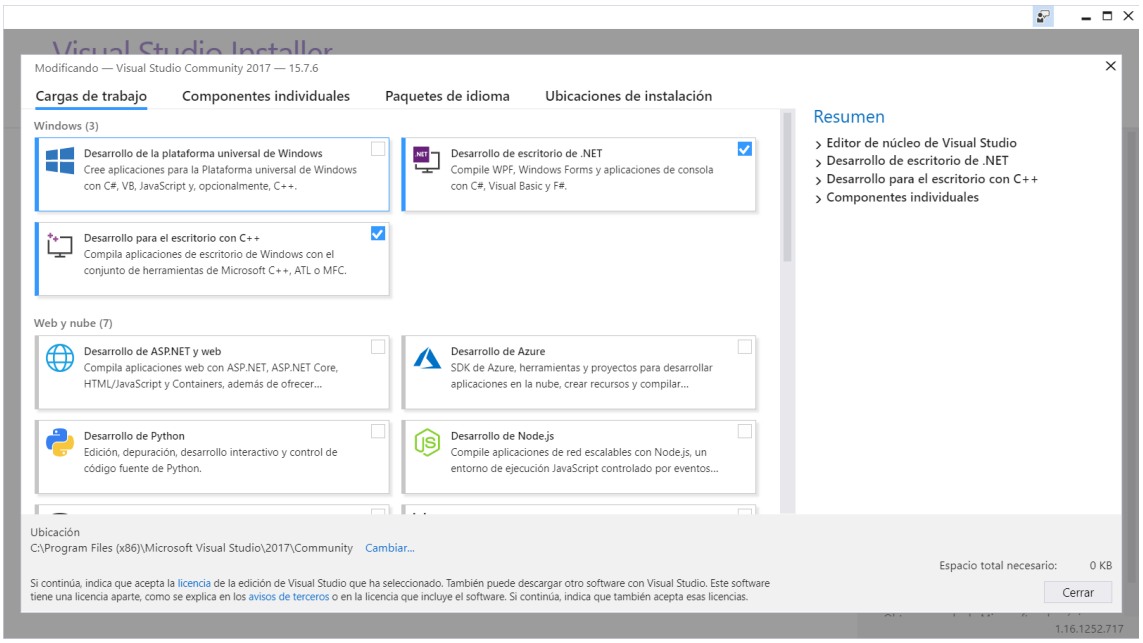


Fig. 4.12 – Menú de instalación de las cargas de trabajo

En nuestro caso trabajaremos esencialmente con la carga de trabajo *Desarrollo de escritorio de .NET* ya que la aplicación que vamos a desarrollar está en lenguaje C#.

Visual Studio también cuenta con herramientas que sirven de ayuda al desarrollador mientras está programando, como por ejemplo Microsoft IntelliSense, que sirve para autocompletar código además de servir como documentación y para desambiguación de nombre de variables, procedimientos y métodos.

4.5.1. Interfaz de Visual Studio

Podemos ver la interfaz de Visual Studio en la Fig. 4.13. Esta interfaz cuenta con varias ventanas:

- Explorador de soluciones
- Propiedades
- Cuadro de herramientas
- Salida
- Desarrollo

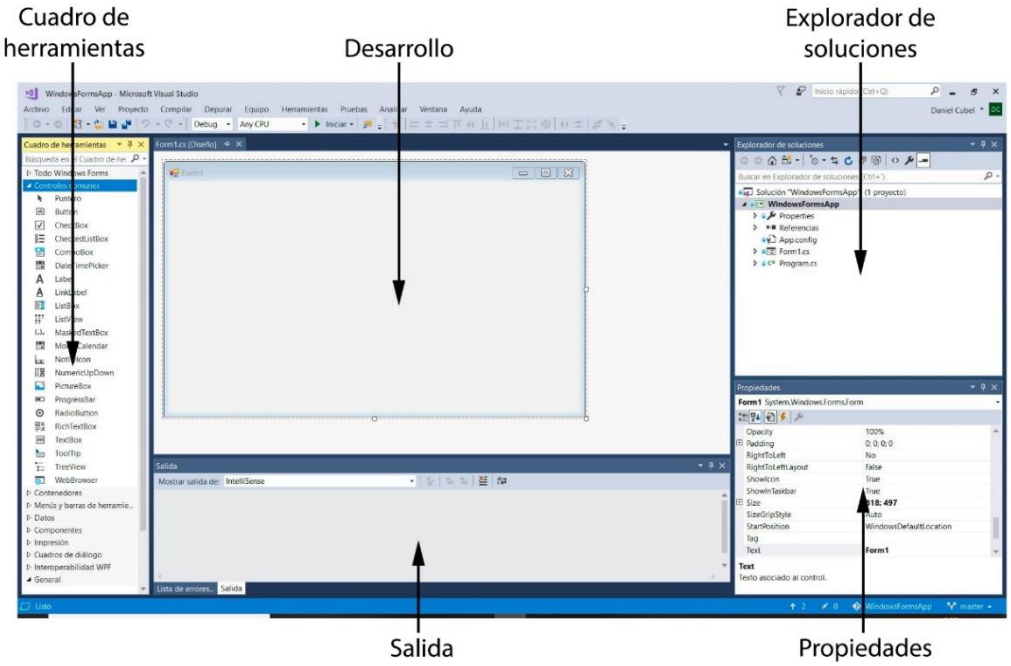


Fig. 4.13 – Interfaz de Visual Studio

La ventana *Explorador de soluciones* muestra de forma jerárquica los proyectos junto a sus archivos y permite el acceso directo a ellos. Cada proyecto contiene sus archivos, pero tienen algunos en común: propiedades, configuración y referencias. Dentro de referencias deben incluirse las bibliotecas que sean utilizadas posteriormente en el código.

La ventana *Cuadro de herramientas* muestra los controles que se pueden agregar al proyecto, como menús, barras de herramientas y otros elementos necesarios para la aplicación. Cada tipo de proyecto tiene controles distintos y depende de la versión de .NET Framework utilizada.

La ventana de *Salida* muestra los posibles errores semánticos y advertencias tanto por errores semánticos como sintácticos una vez se ha compilado la solución.

4.5.2. Windows Forms

Las aplicaciones de PC SDK se desarrollan sobre un formulario Windows Forms en C#. Un formulario es una superficie visual sobre la que se pueden agregar controles, un elemento de interfaz que permite mostrar o aceptar la entrada de datos. Cuando el usuario realiza una acción sobre un control se genera un evento. Mediante los controladores de eventos se programa que ocurre cuando se genera un evento.

Se van a destacar los controles disponibles para Windows Forms que van a ser utilizados en este trabajo:

- ListView: Muestra una lista de elementos. Esta lista se puede organizar de cinco formas distintas.
- Button: Presenta un botón estándar que genera un evento cuando el usuario hace click sobre él.
- Panel: permite agrupar varios controles.
- GroupBox: muestra un marco alrededor de los controles con título opcional.

Hay dos formas para desarrollar un formulario de Windows Forms, que son complementarias. Por un lado, el diseñador de formularios permite desarrollar la interfaz gráfica añadiendo controles y, por otro lado, está el código que implementa la funcionalidad de los controles de la interfaz gráfica.

5. ANÁLISIS DE LAS APLICACIONES DE LA DETECCIÓN DEL CUERPO HUMANO PARA COLABORACIÓN CON UN ROBOT

Una vez conocemos la capacidad del sensor Kinect para la detección del cuerpo humano y de los datos que nos aporta de posición y orientación, cabe plantear en que aplicaciones de colaboración con un robot pueden ser útiles.

En respuesta a la pregunta se ocurren dos respuestas:

- Colaboración directa entre la persona y el robot
- Vigilancia de la célula robótica.

5.1. COLABORACIÓN DIRECTA HUMANO-ROBOT

Una de las opciones es la colaboración directa entre la persona y el robot, en la que la persona y el robot están en contacto. A partir de los datos de posición de las partes del cuerpo humano proporcionadas por la Kinect, y conocida la posición relativa entre la cámara y el robot, se le puede comunicar al robot que vaya a una posición determinada, como por ejemplo la mano.

Dentro de esta opción podemos plantearnos dos formas de programación:

- Que sea el usuario el que elija cuando se captura una posición.
- Que sea el programa del robot el que elija cuando se captura una posición.

Por otro lado, también cabe plantearse si la captura se realiza una sola vez o se lleva a cabo un seguimiento del movimiento de la mano.

Fig. 5.1 – Ejemplo de colaboración directa humano-robot

5.2. VIGILANCIA DE LA CÉLULA ROBÓTICA

Otra de las opciones es la de vigilancia de la célula robótica, que permitiría eliminar las vallas de protección que existen actualmente. A partir de los datos de posición de las partes del cuerpo humano proporcionadas por la Kinect, y conocida la posición relativa entre la cámara y el robot, puede saberse la posición de la persona con respecto al robot y si esta distancia es menor a una distancia de seguridad se podría reducir la velocidad o parar el robot.

Para esta opción también cabrían dos formas de programación:

- Calcular la distancia de la persona a la herramienta del robot
- Calcular si la persona está dentro de la zona de trabajo del robot.

La principal desventaja de la primera forma de programación es que solo se garantiza la seguridad respecto a la herramienta, pero no respecto al resto de eslabones del robot, mientras que en la segunda opción si se garantiza esta seguridad.

Por ejemplo, para el robot ABB IRB 120 se podría modelar la zona de trabajo como una esfera de radio 580mm (3.1.3 - Movimiento del robot). En caso de que una persona entre en la zona delimitada por la esfera, el robot reduciría su velocidad o se pararía.

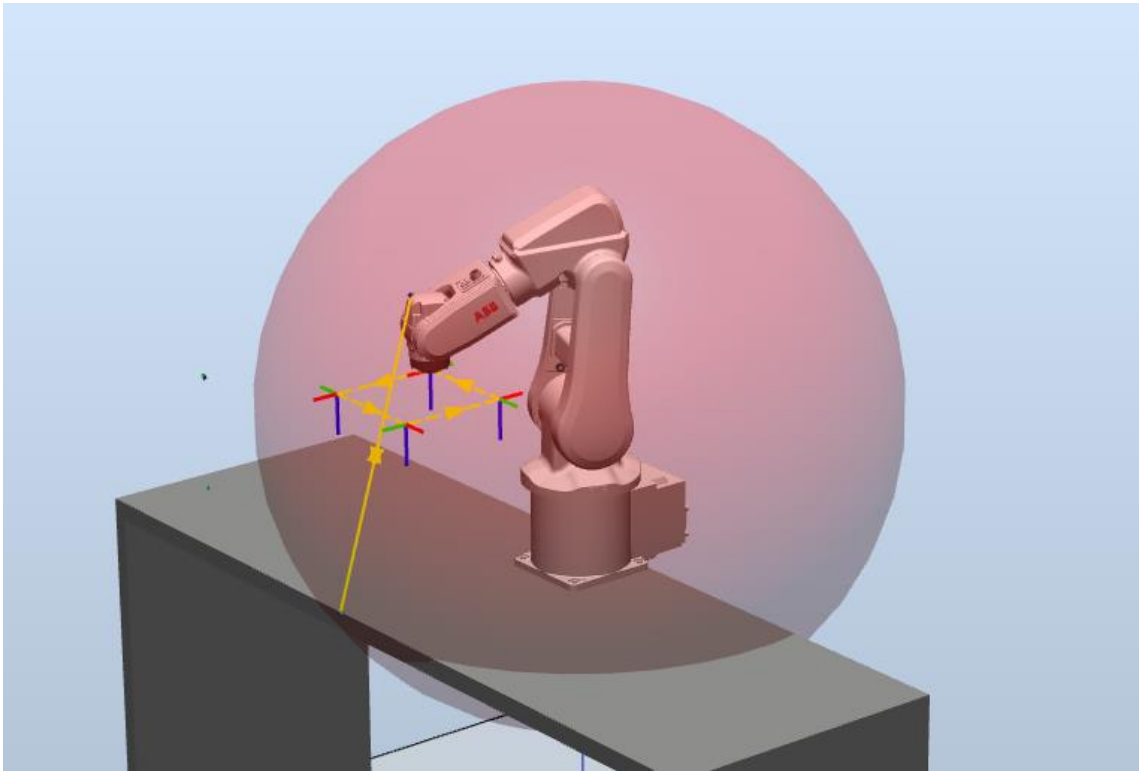


Fig. 5.2 – Zona de trabajo del robot ABB IRB 120

5.3. CASOS A ESTUDIAR

A la vista del análisis realizado, se van a estudiar las siguientes opciones:

- Captura de la posición de la mano según usuario.
- Captura de la posición de la mano según código RAPID.
- Seguimiento de la mano.
- Vigilancia de la célula robótica.

6. PROGRAMACIÓN DE LA KINECT

6.1. OPCIONES DE PROGRAMACIÓN

El SDK de la Kinect soporta los lenguajes de programación C++, C# y Visual Basic. Se van a analizar dos opciones de programación para la Kinect:

- C++ y bibliotecas Open CV
- C#

6.1.1. C++ y bibliotecas OpenCV

Para esta opción, el SDK de la Kinect nos proporciona las bibliotecas necesarias para poder programar la cámara. Adicionalmente se puede disponer de la biblioteca NtKinect [17] que reúne todas las herramientas del SDK en una clase que hace más fácil su utilización en el código. Las bibliotecas OpenCV nos proporcionan las herramientas necesarias para el correcto tratamiento de la imagen.

El trabajo de Juan Felipe Montesinos [5] programa la Kinect en C++ con estas bibliotecas. La aplicación desarrollada en este trabajo integra la obtención de las imágenes RGB, infrarroja y profundidad, la captura de imágenes y video, el reconocimiento del esqueleto humano y la obtención de las coordenadas de la imagen utilizando el cursor y su almacenamiento en un fichero de texto.

6.1.2. C#

Para esta opción, el SDK también nos proporciona las bibliotecas necesarias para poder programar la cámara, sin necesidad de utilizar bibliotecas adicionales.

Como se ha comentado anteriormente, el SDK de la Kinect proporciona códigos de muestra. El programa de muestra BodyBasics sirve para obtener y visualizar el esqueleto del cuerpo humano. Con un poco de código adicional podemos conseguir la posición y orientación de las articulaciones.

Por ello, se ha elegido esta opción para programar la Kinect. A continuación, se detalla el código implementado para obtener la posición y la orientación de las articulaciones.

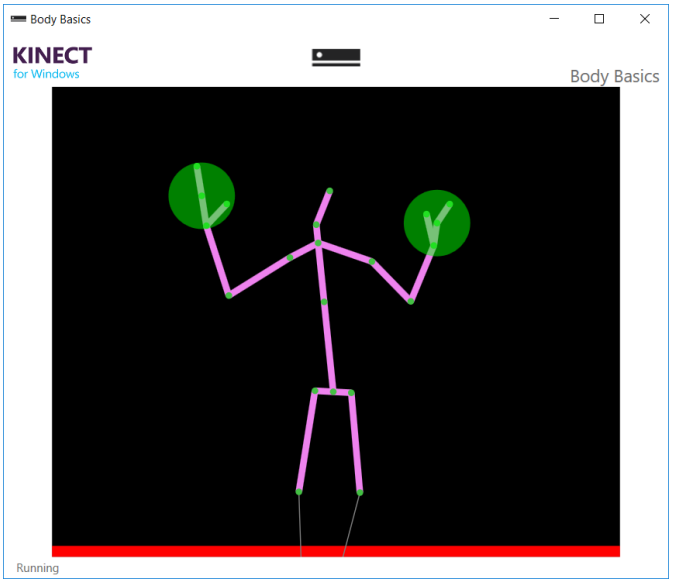


Fig. 6.1 – Ventana de Body Basics

6.2. CÓDIGO DESARROLLADO PARA LA KINECT

Aunque la Kinect nos permite definir el cuerpo humano con 25 articulaciones, en un principio solo vamos nos van a ser de interés las dos manos y la cabeza.

El primer paso es definir las variables al principio del código:

```
        private Joint pRH; // posición de la mano derecha
        private Joint pLH; // posición de la mano izquierda
        private Joint pHead; // posición de la cabeza

        private JointOrientation oRH; // orientación de la mano derecha
        private JointOrientation oLH; // orientación de la mano izquierda
```

Código 1 – Declaración de las variables de las articulaciones

El siguiente paso es definir las propiedades que nos permitirán dar y obtener el valor de estas variables:

```
        public Joint GetpRH { get => pRH; set => pRH = value; }
        public Joint GetpLH { get => pLH; set => pLH = value; }
        public Joint GetpHead { get => pHead; set => pHead = value; }

        public JointOrientation GetoRH { get => oRH; set => oRH = value; }
        public JointOrientation GetoLH { get => oLH; set => oLH = value; }
```

Código 2 – Declaración de las propiedades

Por último, habrá que darles valor a las variables cuando la Kinect detecte una persona:

```
        GetpRH = body.Joints[JointType.HandRight];
        GetpLH = body.Joints[JointType.HandLeft];
        GetpHead = body.Joints[JointType.Head];

        GetoRH = body.JointOrientations[JointType.HandRight];
        GetoLH = body.JointOrientations[JointType.HandLeft];
```

Código 3 – Asignación de valor a las variables

Una vez resuelta la programación de la Kinect, podemos pasar a desarrollar la aplicación de PC SDK para los casos del apartado 5.3 y del código de RAPID correspondiente.

7. PROGRAMACIÓN DE LA APLICACIÓN DE PC SDK

Una vez conocemos los elementos de la célula robótica, el software utilizado y ya está programada la Kinect para obtener las posiciones y orientaciones necesarias, podemos pasar al siguiente paso: el diseño y desarrollo de la aplicación de PC SDK.

Para este paso hay que tener en cuenta dos aspectos:

- Por un lado, el tratamiento de los datos obtenidos con la Kinect y la relación con el controlador ya sea virtual o real, para utilizar esos datos en la programación del robot.
- Por otro lado, el programa RAPID, que debe ser adecuado a la aplicación que se desee conseguir con los datos de la Kinect.

7.1. INTERFAZ DE USUARIO

Para desarrollar la aplicación de PC SDK se va a seguir las indicaciones del manual de aplicación del PC SDK [18]. Partimos de una aplicación de Windows Forms vacía (Fig. 7.1)

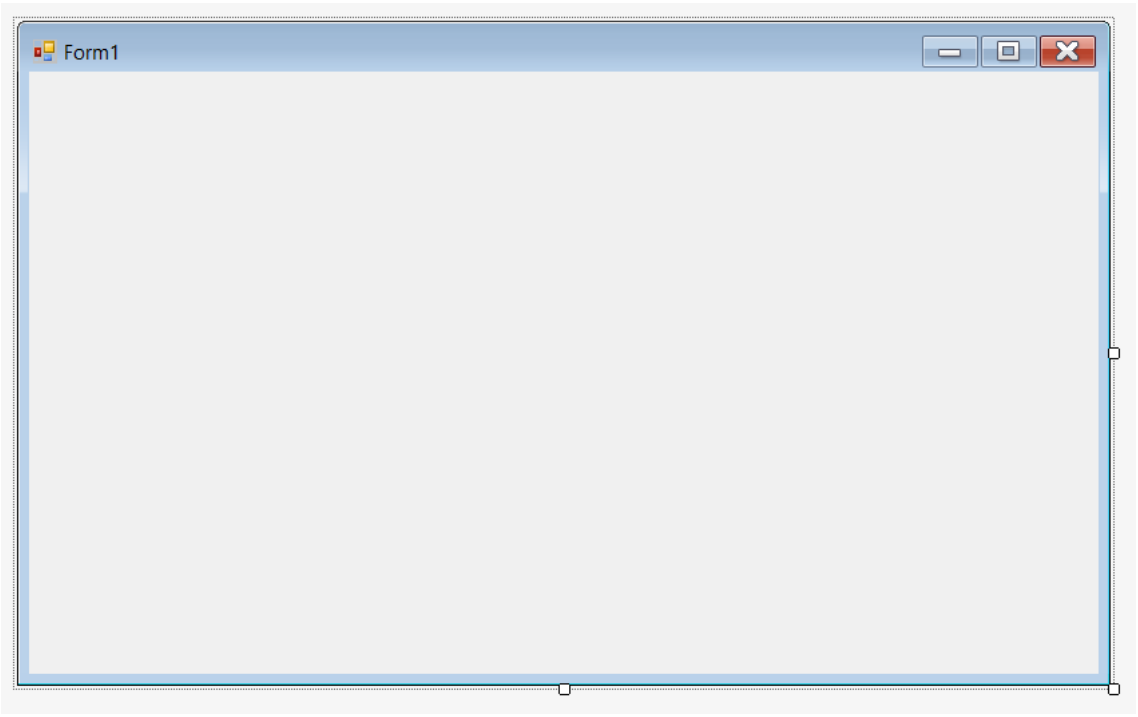


Fig. 7.1 – Ventana inicial de la aplicación de Windows Forms

7.1.1. Diseño y funcionalidad de la interfaz.

La interfaz de la aplicación tendrá dos partes. Por un lado, en la parte superior de la ventana habrá una lista con los controles activos en la red del ordenador junto con su información, que permitirá conectarse a un controlador al hacer doble click sobre él. Por otra parte, en la parte inferior de la ventana habrá un panel con botones que permitirá ejecutar el código correspondiente a los cuatros casos señalados en el apartado 5.3, además de ejecutar y para el código RAPID.

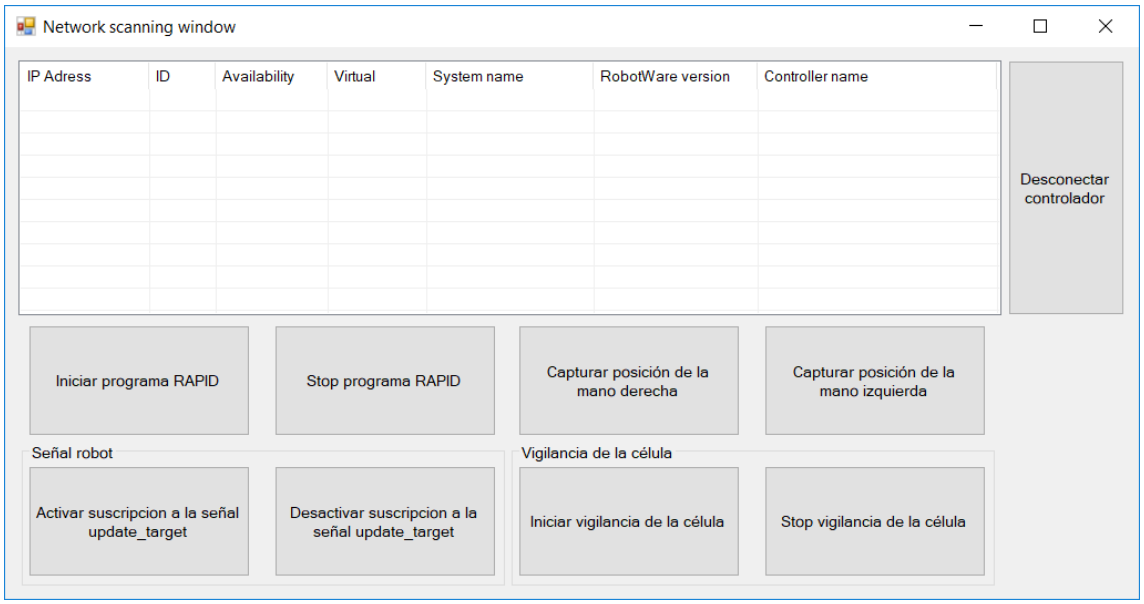


Fig. 7.2 – Interfaz de usuario

7.1.2. Desarrollo de la interfaz

El primer paso, una vez se ha creado el proyecto de Windows Forms, es añadir las referencias necesarias para poder utilizar las bibliotecas de PC SDK y Kinect SDK. Se deben añadir y declarar en el código las siguientes referencias:

```
using ABB.Robotics;
using ABB.Robotics.Controllers;
using ABB.Robotics.Controllers.Discovery;
using ABB.Robotics.Controllers.RapidDomain;
using ABB.Robotics.Controllers.IOSystemDomain;
using Microsoft.Kinect;
```

Código 4 – Referencias de PC SDK y Kinect SDK

En siguiente paso es añadir un control ViewList a la ventana. En la ventana de propiedades se debe seleccionar FullRowSelect como True, GridLines como True y View como Details. Después se añaden las columnas IP adress, ID, Availability, Virtual, System name, RobotWare versión y Controller name, que mostrarán la información del controlador.

El siguiente paso es desarrollar el código que nos permitirá escanear la red para encontrar los controladores activos y añadir su información a la ViewList.

Para encontrar todos los controladores de la red se deben declarar las siguientes variables:

```
private NetworkScanner scanner = null;
private Controller controller = null;
private Task[] tasks = null;
private NetworkWatcher networkwatcher = null;
```

Código 5 – Declaración de variables

Después se implementa el código que escanea la red. Como nos interesa que escaneo se realice nada más abrir la aplicación este código debe ir en la gestión de eventos de carga de la ventana, Form1_Load.

```
        // Encontrar los controladores activos.
        this.scanner = new NetworkScanner();
        this.scanner.Scan();
        ControllerInfoCollection controllers = scanner.Controllers;
        ListViewItem item = null;
        foreach (ControllerInfo controllerInfo in controllers)
        {
                item = new ListViewItem(controllerInfo.IPAddress.ToString());
                item.SubItems.Add(controllerInfo.Id);
                item.SubItems.Add(controllerInfo.Availability.ToString());
                item.SubItems.Add(controllerInfo.IsVirtual.ToString());
                item.SubItems.Add(controllerInfo.SystemName);
                item.SubItems.Add(controllerInfo.Version.ToString());
                item.SubItems.Add(controllerInfo.ControllerName);
                this.listView1.Items.Add(item);
                item.Tag = controllerInfo;
        }
```

Código 6 – Escaneo de controladores

El escáner solo sirve para recuperar los controladores que están activos cuando se carga la aplicación, pero puede ser que se añadan o se pierdan controladores mientras la aplicación se está ejecutando. Por ello, es necesario implementar un "vigilante" que detecte cuando se añaden o se pierden controladores. Para ello se crea un NetworkWatcher y se le suscribe a los eventos HandleFoundEvent, cuando se añade un controlador, o HandleLostEvent, cuando se pierde. El NetworkWatcher debe iniciarse cuando se cargue la aplicación, por lo que se inicializa en la gestión de eventos de carga de la ventana, Form1_Load.

```
        // Detectar cuando se conectan o desconectan controladores.
        this.networkwatcher = new NetworkWatcher(scanner.Controllers);
        this.networkwatcher.Found += new
EventHandler<NetworkWatcherEventArgs>(HandleFoundEvent);
        this.networkwatcher.Lost += new
EventHandler<NetworkWatcherEventArgs>(HandleLostEvent);
        this.networkwatcher.EnableRaisingEvents = true;
```

Código 7 – Inicialización del Network Watcher

Cuando se produzca un evento debe ser recibido en segundo plano y resultar en la actualización de los elementos de la ViewList:

```
        void HandleFoundEvent(object sender, NetworkWatcherEventArgs e)
        {
                this.Invoke(new
        EventHandler<NetworkWatcherEventArgs>(AddControllerToListView), new
        Object[] { this, e });
        }

        private void AddControllerToListView(object sender,
NetworkWatcherEventArgs e)
        {
                ControllerInfo controllerInfo = e.Controller;
                ListViewItem item = new
        ListViewItem(controllerInfo.IPAddress.ToString());
                item.SubItems.Add(controllerInfo.Id.ToString());
                item.SubItems.Add(controllerInfo.Availability.ToString());
                item.SubItems.Add(controllerInfo.IsVirtual.ToString());
                item.SubItems.Add(controllerInfo.SystemName);
                item.SubItems.Add(controllerInfo.Version.ToString());
                item.SubItems.Add(controllerInfo.ControllerName);
                this.listView1.Items.Add(item);
                item.Tag = controllerInfo;
        }
```

Código 8 – Gestión del evento cuando se añade un controlador

```
        void HandleLostEvent(object sender, NetworkWatcherEventArgs e)
        {
                this.Invoke(new
        EventHandler<NetworkWatcherEventArgs>(RemoveControllerFromListView),
        new Object[] { this, e });
        }

        private void RemoveControllerFromListView(object sender,
NetworkWatcherEventArgs e)
        {
                foreach (ListViewItem item in this.listView1.Items)
                {
                        if ((ControllerInfo)item.Tag == e.Controller)
                        {
                                this.listView1.Items.Remove(item);
                                break;
                        }
                }
        }
```

Código 9 – Gestión del evento cuando se pierde un controlador

Una vez se han cargado los datos de los controladores y estos aparecen en la lista, vamos a querer conectarnos a uno de ellos. Para conectarnos a un controlador debemos crear un evento al hacer doble click sobre la ViewList (DoubleClick event). En la gestión del evento debemos crear un objeto controlador que representa al controlador seleccionado y conectarnos a él. Previo a la conexión, habrá que comprobar que el controlador está disponible.

```
        ListViewItem item = this.listView1.SelectedItems[0];
        if (item.Tag != null)
        {
                ControllerInfo controllerInfo = (ControllerInfo)item.Tag;
                if (controllerInfo.Availability == Availability.Available)
                {
                        if (this.controller != null)
                        {
                                this.controller.Logoff();
                                this.controller.Dispose();
                                this.controller = null;
                        }
                        this.controller =
                ControllerFactory.CreateFrom(controllerInfo);
                        this.controller.Logon(UserInfo.DefaultUser);
                        MessageBox.Show("Estás conectado al controlador.");
                }
                else
                {
                        MessageBox.Show("El controlador seleccionado no está
                disponible.");
                }
        }
```

Código 10 – Conexión a un controlador

También puede interesarnos en algún momento desconectarnos del controlador. Para ello añadimos un botón que permita desconectarnos.

```
        private void DisconnectController_Click(object sender, EventArgs e)
        {
                if (this.controller != null)
                {
                        this.controller.Logoff();
                        this.controller.Dispose();
                        this.controller = null;
                        MessageBox.Show("Te has desconectado del controlador");
                }
        }
```

Código 11 – Desconexión de un controlador

Una vez estamos conectados al controlador podemos ejecutar y detener un programa RAPID y acceder al dominio RAPID, lo que nos permite leer y escribir variables.

Fig. 7.3 – Botones de inicio y stop del programa RAPID

Ahora vamos a implementar la funcionalidad de los botones que inician y detienen el código RAPID. Para ello será necesario que el robot esté en modo automático y que la aplicación pueda tomar el control del controlador. En caso de que no pueda tener el control, el robot no esté en modo automático o haya otro problema, saltará un mensaje de aviso.

```
try
{
	if (controller.OperatingMode == ControllerOperatingMode.Auto)
	{
		tasks = controller.Rapid.GetTasks();

		// Cambiar el valor de start_bool
		Bool start_AUX = new Bool();
		RapidData start =
	controller.Rapid.GetRapidData("T_ROB1", "Module1",
	"start_bool");

		if (start.Value is Bool)
		{
			start_AUX.Value = true;
		}

		using (Mastership m =
	Mastership.Request(controller.Rapid))
		{
			start.Value = start_AUX;
			tasks[0].SetProgramPointer("Module1", "main");
			controller.Rapid.Start();
		}
	}
	else
	{
		MessageBox.Show("Automatic mode is required to start
	execution from a remote client.");
	}
}
catch (System.InvalidOperationException ex)
{
	MessageBox.Show("Mastership is held by another client." +
ex.Message);
}
catch (System.Exception ex)
{
	MessageBox.Show("Unexpected error occurred: " + ex.Message);
}
```

Código 12 – Funcionalidad del botón Iniciar programa RAPID

```
try
{
	if (controller.OperatingMode == ControllerOperatingMode.Auto)
	{
		// Cambiar el valor de start_bool
		Bool stop_AUX = new Bool();
		RapidData stop_bool =
	controller.Rapid.GetRapidData("T_ROB1", "Module1",
	"start_bool");

		if (stop_bool.Value is Bool)
		{
			stop_AUX.Value = false;
		}
```

```
                    // Actualizar los valores de las variables de RAPID.
                    using (Mastership m =
              Mastership.Request(controller.Rapid))
                    {
                          stop_bool.Value = stop_AUX;
                          controller.Rapid.Stop();
                    }
              }
              else
              {
                    MessageBox.Show("Automatic mode is required to start
              execution from a remote client.");
              }
        }
        catch (System.InvalidOperationException ex)
        {
              MessageBox.Show("Mastership is held by another client." +
              ex.Message);
        }
        catch (System.Exception ex)
        {
              MessageBox.Show("Unexpected error occurred: " + ex.Message);
        }
```

Código 13 – Funcionalidad del botón Stop programa RAPID

Llegados a este punto hemos implementado la funcionalidad básica de la aplicación de PC SDK. Tenemos una aplicación que escanea los controladores activos de una red y los representa en una lista, que actualiza esta lista según se añaden o pierden controladores, que puede conectarse a un controlador de la lista y ejecutar o detener un programa RAPID del controlador al que se está conectado.

La siguiente parte será desarrollar la funcionalidad especifica de la aplicación en los casos del apartado 5.3:

- Captura de la posición de la mano según usuario.
- Captura de la posición de la mano según código RAPID.
- Seguimiento de la mano.
- Vigilancia de la célula robótica.

7.2. DESARROLLO DE LOS CASOS DE ESTUDIO

7.2.1. Célula robótica en RobotStudio

La célula robótica en RobotStudio estará formada por un robot ABB IRB 120 con una herramienta pinza estándar ABB Smart Gripper. La Kinect será simulada con un objeto de trabajo cuyo sistema de referencia sea el mismo que el de la cámara. En este caso, al ser una simulación y no estar trabajando con una célula robótica real, el sistema de referencia se colocará detrás del robot por comodidad.

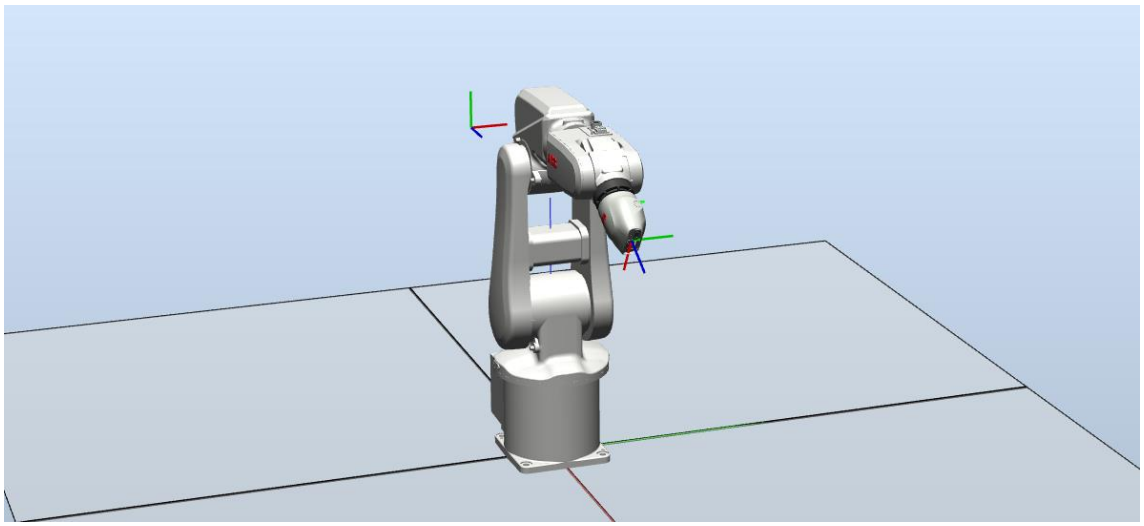


Fig. 7.4 – Célula robótica utilizada

Se necesitará crear un objetivo dentro de la referencia de la Kinect para después poder ser actualizada. El hecho de que el objeto de trabajo de la Kinect esté definido por nosotros nos evita tener que calcular la posición y orientación relativa entre el robot y la cámara.

En una aplicación real, en la que no conocemos la posición ni la orientación relativa, sería necesario llevar a cabo una calibración entre el robot y la cámara con el objetivo de conocer la matriz de transformación entre las dos bases.

7.2.1.1. Orientación de las manos

La programación de la Kinect nos devuelve la posición y la orientación de las manos, pero esta orientación no coincide con la orientación de la herramienta del robot, por lo que es necesario girar la orientación de la mano para que coincidan.

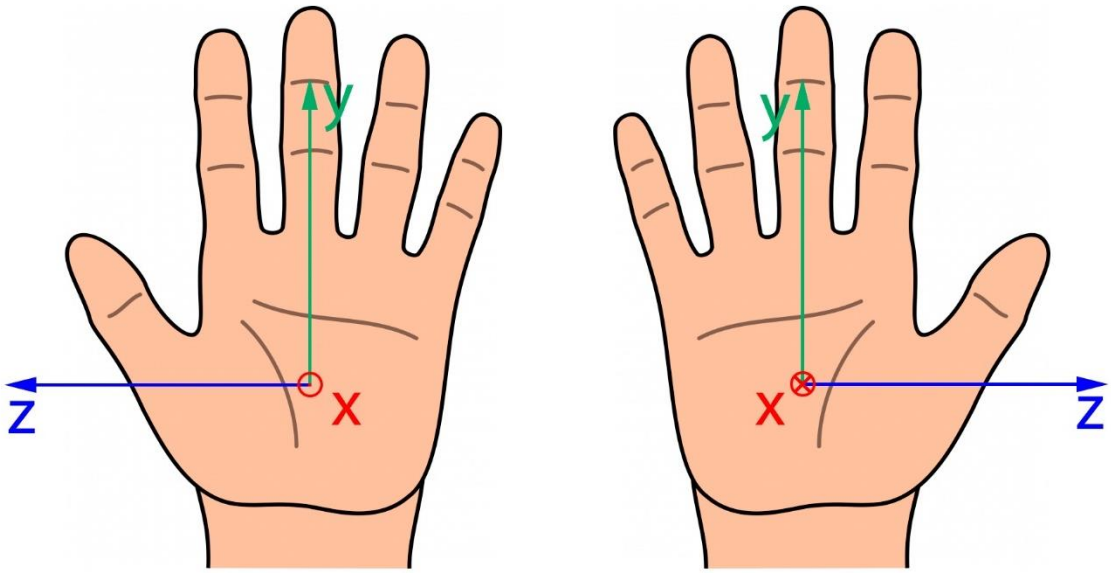


Fig. 7.5 – Orientación de las manos según la Kinect

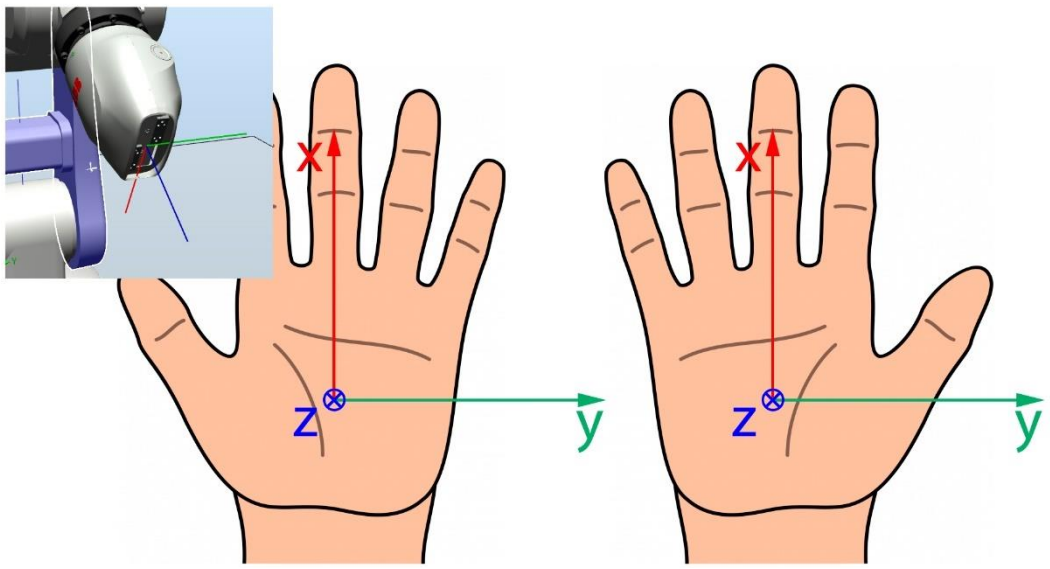


Fig. 7.6 – Orientación de la herramienta y orientación final de las manos

Observando las orientaciones de las manos que nos da la Kinect y la orientación de las manos que queremos obtener de forma que coincidan con la orientación del robot, llegamos a la conclusión de que:

- La orientación de la mano derecha hay que girarla 90º en *dirección Y* y 90º en dirección Z, estando ambos giros referidos al sistema de coordenadas original.
- La orientación de la mano izquierda hay que girarla -90º en *dirección Y* y -90º en dirección X, estando ambos giros referidos al sistema de coordenadas original.

7.2.2. Captura de la posición de la mano según usuario

7.2.2.1. Funcionalidad

El funcionamiento de esta opción debe ser el siguiente: al pulsar el botón de capturar posición, ya sea la mano derecha o izquierda, el robot debe desplazarse hasta esa posición. Esto será conseguido actualizando el valor de una posición.

La captura de la posición de la mano se debe producir después del usuario haga click con el cursor sobre el botón Capturar posición de la mano, ya sea derecha o izquierda. Para ello debe crearse un evento de Click para esos botones.

7.2.2.2. Código C#

La aplicación de PC SDK debe adquirir, por un lado, la posición y la orientación de la mano de la Kinect y, por otra parte, una variable de tipo RapidData que contenga los datos de una posición del código RAPID. El valor de la posición se casteará a una variable auxiliar de tipo RobTarget, que servirá para actualizar la posición según los datos de la Kinect. Posteriormente, se deberán actualizar los datos de la posición con los datos de la mano en la variable de tipo RapidData y enviarlos al controlador.

Este código irá dentro del gestor de eventos para el Click de los botones de Capturar posición de la mano. El siguiente código captura de la posición de la mano derecha.

```
try
{
	if (controller.OperatingMode == ControllerOperatingMode.Auto)
	{
		// Cambiar el valor de Target_10 / Posición de la mano.
		RobTarget RobTarget_AUX = new RobTarget();
		RapidData rd = controller.Rapid.GetRapidData("T_ROB1",
	"Module1", "Target_10");

		if (rd.Value is RobTarget)
		{
			RobTarget_AUX = (RobTarget)rd.Value;
		}

		Joint RH = myKinectWindow.GetpLH;
		JointOrientation ORH = myKinectWindow.GetoLH;

		// Parte de traslación
		Pos pos = new Pos();
		pos.X = RH.Position.X * 1000;
		pos.Y = RH.Position.Y * 1000;
		pos.Z = RH.Position.Z * 1000;

		//Parte de rotación
		//Orientación de la mano según ángulos de la Kinect
		Quaternion rot_AUX = new Quaternion();
		rot_AUX.W = ORH.Orientation.W;
		rot_AUX.X = ORH.Orientation.X;
		rot_AUX.Y = ORH.Orientation.Y;
		rot_AUX.Z = ORH.Orientation.Z;

		// Cuaternios de rotación
		Vector3 X_vector = Vector3.UnitX;
		Vector3 Y_vector = Vector3.UnitY;
		Quaternion quat_X = Quaternion.CreateFromAxisAngle(X_vector,
	(float)Math.PI / 2);
		Quaternion quat_Y = Quaternion.CreateFromAxisAngle(Y_vector,
	(float)Math.PI / 2);

		// Giro de los ángulos de la Kinect
		rot_AUX = rot_AUX * quat_X * quat_Y;

		Orient rot = new Orient();
		rot.Q1 = rot_AUX.W;
		rot.Q2 = rot_AUX.X;
		rot.Q3 = rot_AUX.Y;
		rot.Q4 = rot_AUX.Z;

		//Actualización de los valores en la variable auxiliar.
		RobTarget_AUX.Trans = pos;
		RobTarget_AUX.Rot = rot;

		// Cambiar el valor de update_target para poder actualizar
	Target_10 / Posición de la mano.
		Bool updatetarget_AUX = new Bool();
		RapidData updatetarget =
	controller.Rapid.GetRapidData("T_ROB1", "Module1", "update_target");
```

```
                if (updatetarget.Value is Bool)
                {
                        updatetarget_AUX.Value = true;
                }

                // Actualizar los valores de las variables de RAPID.
                using (Mastership m = Mastership.Request(controller.Rapid))
                {
                        rd.Value = RobTarget_AUX;
                        updatetarget.Value = updatetarget_AUX;
                }
        }
        else
        {
                MessageBox.Show("Automatic mode is required to start
        execution from a remote client.");
        }
}
catch (System.InvalidOperationException ex)
{
        MessageBox.Show("Mastership is held by another client." +
ex.Message);
}
catch (System.Exception ex)
{
        MessageBox.Show("Unexpected error occurred: " + ex.Message);
}
```

Código 14 – Código C# para la captura de la posición de la mano

7.2.2.3. Código RAPID

El código RAPID debe declarar una variable de tipo RobTarget que sea la posición a actualizar. Para poder actualizar la posición de la mano mientras la aplicación de PC SDK está activa se necesita un bucle WHILE que incorpore el movimiento a la posición. Para evitar que siempre se esté ejecutando la instrucción de movimiento, se puede declarar una variable booleana update_target, de forma que la instrucción de movimiento se ejecute cuando esta variable sea verdadera, dentro de un IF. Esta variable se pondrá en verdadero desde la aplicación de PC SDK y se pondrá en falso cuando se termine de ejecutar la instrucción de movimiento.

```
MODULE Module1
	CONST robtarget
Target_20:=[[364.353829072,0,594],[0.5,0,0.866025404,0],[0,0,0,0],[9E+09,9E+09,
9E+09,9E+09,9E+09,9E+09]];
	VAR robtarget
Target_10:=[[0,93.99994615,864.353821314],[0.683012701,0.183012706,0.183012
706,-0.683012701],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

	VAR bool start_bool := TRUE;
	VAR bool update_target:=FALSE;

	PROC main()
		MoveJ Target_20,v1000,z0,tool0\WObj:=wobj0;
		WHILE start_bool DO
			IF (update_target = TRUE) THEN
				MoveJ Target_10,v100,z0,Servo\WObj:=Kinect;
				update_target:=FALSE;
			ENDIF
		ENDWHILE
	ENDPROC
ENDMODULE
```

Código 15 – Código RAPID para la captura de la posición de la mano

7.2.3. Captura de la posición de la mano según código RAPID

7.2.3.1. Funcionalidad

El funcionamiento de esta opción debe ser el siguiente: en el momento en el que el código RAPID necesite actualizar una posición, requerirá a la aplicación de PC SDK la posición de la mano.

Esto se consigue por medio de una señal de salida digital en RobotStudio y una suscripción al cambio de valor de esta señal en la aplicación de PC SDK. De esta forma, cada vez que la señal cambie de valor será detectada por la aplicación y podrá ejecutar el código necesario.

Cuando la señal se ponga a 1, se deberá ejecutar el código de la aplicación PC SDK que actualice la posición de la mano y cuando termine pondrá la señal a 0. Mientras tanto, el código RAPID estará esperando el cambio de la señal a 0. Cuando se produzca se ejecutará la instrucción de movimiento.

La suscripción a la señal se explicitará con el botón Activar suscripción a la señal update_target. Así mismo, también deberá ser posible desactivar la suscripción con el botón Desactivar la suscripción a la señal update_target.

7.2.3.2. Código C#

En la aplicación de PC SDK se debe declarar una señal y asignarle la señal creada en RobotStudio. Posteriormente se debe añadir una suscripción al cambio de valor de la señal.

```
try
{
	if (controller.OperatingMode == ControllerOperatingMode.Auto)
	{
		// Declaración de la señal que nos indica cuando actualizar
	la posición.
		Signal updatetarget_signal =
	controller.IOSystem.GetSignal("update_target");

		// Añadir suscripción para detectar el cambio de valor de la
	señal.
		updatetarget_signal.Changed += new
	EventHandler<SignalChangedEventArgs>(Sig_Changed);
	}
	else
	{
		MessageBox.Show("Automatic mode is required to start
	execution from a remote client.");
	}
}
catch (System.InvalidOperationException ex)
{
	MessageBox.Show("Mastership is held by another client." +
ex.Message);
}
catch (System.Exception ex)
{
	MessageBox.Show("Unexpected error occurred: " + ex.Message);
}
```

Código 16 – Código C# de la declaración de la señal y suscripción al cambio de valor.

Cuando el valor de la señal cambie a 1, se ejecutará el código contenido en el método Sig_Changed, que es idéntico al código de captura de la posición de la mano (Código 15). La única diferencia es que deberá poner el valor de la señal a 0.

```
	//Declaración de la señal
	Signal updatetarget_signal =
controller.IOSystem.GetSignal("update_target");

	// Actualizar los valores de las variables de RAPID.
	using (Mastership m = Mastership.Request(controller.Rapid))
	{
		updatetarget_signal.Value = 0;
	}
```

Código 17 – Código C# añadido en Sig_Changed para reiniciar la señal

También es necesario programar la desactivación de la suscripción al cambio de valor de la señal. El código es similar al código de suscripción, pero con un pequeño cambio.

```
	updatetarget_signal.Changed -= new
EventHandler<SignalChangedEventArgs>(Sig_Changed);
```

Código 18 - Código C# para desactivar la suscripción a la señal

7.2.3.3. Código RAPID

El código RAPID debe poner la señal update_target a 1 y esperar a que la ejecución del código de la aplicación de PC SDK ponga el valor de la señal a 0. Cuando la señal se haya puesto a 0 debe ejecutarse la instrucción de movimiento que mueva el robot hasta la posición de la mano capturada.

```
MODULE Module1
	CONST robtarget
Target_20:=[[364.353826402,0,593.999995375],[0.499999968,0,0.866025422,0],[0,
0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

	VAR robtarget
Target_10:=[[0,93.999995375,864.353826402],[0.683012695,0.183012727,0.18301
2727,-0.683012695],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

	PROC main()
		MoveJ Target_20,v500,fine,tool0\WObj:=wobj0;

		SetDO update_target, 1;
		WaitTime 1;
		WaitDO update_target, 0;

		MoveJ Target_10,v1000,z0,tool0\WObj:=Kinect;
		MoveJ Target_20,v500,fine,tool0\WObj:=wobj0;
	ENDPROC
ENDMODULE
```

Código 19 – Código RAPID para la captura de la posición de la mano según RAPID

7.2.4. Seguimiento de la posición de la mano

7.2.4.1. Funcionalidad

El funcionamiento de esta opción debe ser el siguiente: cuando el usuario ejecute el programa RAPID el robot debería seguir la posición de la mano. Esto se consigue de una forma similar al caso anterior, pero con el movimiento a la posición de la mano dentro de un bucle WHILE en el código RAPID.

En este caso, los botones son los mismos que en el caso anterior, Activar suscripción a la señal update_target y Desactivar la suscripción a la señal update_target.

7.2.4.2. Código C#

El código de la aplicación de PC SDK es el mismo que para el caso anterior, suscripción a la señal que indica que se puede actualizar la posición, captura de la posición en Sig_Changed y posibilidad de desactivar la suscripción a la señal.

7.2.4.3. Código RAPID

El código RAPID para el seguimiento del movimiento de la mano es similar al programa para capturar la mano según código RAPID, pero encapsulado en un bucle WHILE.

Cuando se ejecute el programa, se entrará en el bucle. La señal para actualizar la posición se pondrá a 1 y se ejecutará el código de la aplicación de PC SDK que actualiza

la posición de la mano y pone la señal a cero. Una vez esta puesta la señal a cero se ejecuta la instrucción de movimiento. Una vez termina la instrucción de movimiento la señal se vuelve a poner a 1 y el proceso se repite.

```
PROC main()
        MoveJ Target_20,v500,fine,tool0\WObj:=wobj0;
        WHILE start_bool DO
                SetDO update_target, 1;
                WaitTime 1;
                WaitDO update_target, 0;
                MoveJ Target_10,v1000,z0,tool0\WObj:=Kinect;
        ENDWHILE
ENDPROC
```

Código 20 – Código RAPID para el seguimiento de la mano

A la hora de poner la señal update_target a 1 y esperar que la aplicación de PC SDK ponga esta señal a 0, es necesario añadir un tiempo de espera para que el código se ejecute correctamente y no se quede atascado.

7.2.5. Vigilancia de la célula

7.2.5.1. Funcionalidad

El funcionamiento de esta opción debe ser el siguiente: cuando el usuario este dentro de la zona de trabajo del robot, éste debe reducir su velocidad o pararse.

Por un lado, la aplicación de PC SDK debe ir comprobando que la posición del usuario no se encuentra dentro de la zona de trabajo del robot. Mientras tanto el código RAPID debe comprobar si la aplicación de PC SDK ha detectado a un usuario dentro de la zona de trabajo o no.

Para comprobar si el usuario está dentro de la zona de trabajo, se va a suponer que esta zona es una esfera de radio 580 mm situada a 310 mm de altura sobre la base del robot, de acuerdo con la zona de trabajo del robot ABB IRB 120 (Fig. 3.3).

Esta comprobación se realizará una vez cada un periodo de tiempo mediante un reloj. La comprobación de la aplicación de PC SDK se realizará cada 250 ms mientras que la comprobación por código RAPID se realizará cada 500 ms.

7.2.5.2. Código C#

El código de la aplicación de PC SDK debe implementar un reloj que cada 250 ms realice la comprobación. Los relojes de clase Timer generan un evento cada vez que transcurre un intervalo.

Cada vez que transcurra un intervalo se ejecutará el método CheckDistance, que comprobará la distancia desde las articulaciones al centro de la esfera de la zona de trabajo. En caso de que la distancia sea menor a 580 mm (distancia de seguridad), el detector de presencia se pondrá a 1, en caso contrario se pondrá a 0.

Para obtener la distancia hay que transformar las coordenadas de la Kinect a las coordenadas del robot. Esto no era necesario en las opciones anteriores, ya que la posición de la mano se almacenaba en el objeto de trabajo de la Kinect, que tiene el mismo sistema de referencia que la cámara.

Dentro del módulo CalibData se encuentra la definición de la base de la Kinect con respecto a la base del robot. Con esta definición podemos obtener la matriz de transformación de la base de la Kinect a la base del robot. Multiplicando la posición de las articulaciones en la base de la Kinect por la matriz de transformación obtenemos la posición de las articulaciones en la base del robot. Solo falta calcular la distancia hasta el centro de la esfera de la zona de trabajo.

```
try
{
	if (controller.OperatingMode == ControllerOperatingMode.Auto)
	{
		aTimer = new System.Timers.Timer(250);
		aTimer.Elapsed += CheckDistance;
		aTimer.AutoReset = true;
		aTimer.Enabled = true;
		MessageBox.Show("Vigilancia activada.");
	}
	else
	{
		MessageBox.Show("Automatic mode is required to start
execution from a remote client.");
	}
}
catch (System.InvalidOperationException ex)
{
	MessageBox.Show("Mastership is held by another client." +
	ex.Message);
}
catch (System.Exception ex)
{
	MessageBox.Show("Unexpected error occurred: " + ex.Message);
}
```

Código 21 – Código C# para implementar el reloj

```
try
{
	if (controller.OperatingMode == ControllerOperatingMode.Auto)
	{
		// Declaración de variables
		Num haypers_AUX = new Num(); // Variable para activar la
	interrupción en el programa RAPID.

		Vector3 RH_rbt = new Vector3();
		Vector3 LH_rbt = new Vector3();
		Vector3 Head_rbt = new Vector3();

		float distancia_seg = 580;

		// Obtención de la base de la cámara respecto de la base del
	robot.
		WobjData baseknt_AUX = new WobjData();
		Pos pos_knt = new Pos();
		Orient rot_knt = new Orient();
		RapidData baseknt = controller.Rapid.GetRapidData("T_ROB1",
	"CalibData", "Kinect");
```

```
            if (baseknt.Value is WobjData)
            {
                  baseknt_AUX = (WobjData)baseknt.Value;
                  pos_knt = baseknt_AUX.Uframe.Trans;
                  rot_knt = baseknt_AUX.Uframe.Rot;
            }

            // Obtener posición de las manos y de la cabeza.
            Joint RH_knt = myKinectWindow.GetpRH;
            Joint LH_knt = myKinectWindow.GetpLH;
            Joint Head_knt = myKinectWindow.GetpHead;

            // Declaración del booleano que indicará si hay una persona
      dentro del espacio de trabajo.
            RapidData haypers = controller.Rapid.GetRapidData("T_ROB1",
      "Module1", "haypers_AUX");

            if (haypers.Value is Num)
            {
                  haypers_AUX = (Num)haypers.Value;
            }

            // Cambio de coordenadas de las posiciones.
            // Mano derecha
            RH_rbt = FromKntToRbt(RH_knt, pos_knt, rot_knt);
            // Mano izquierda
            LH_rbt = FromKntToRbt(LH_knt, pos_knt, rot_knt);
            // Mano derecha
            Head_rbt = FromKntToRbt(Head_knt, pos_knt, rot_knt);

            //Obtención de las distancias a la base del robot.
            float distance_RH = Vector3.Distance(RH_rbt, Vector3.UnitZ *
      310);
            float distance_LH = Vector3.Distance(LH_rbt, Vector3.UnitZ *
      310);
            float distance_Head = Vector3.Distance(Head_rbt,
      Vector3.UnitZ * 310);

            // Cambiar el valor del booleano si la distancia es menor a
      la distancia de seguridad.
            if ((distance_Head > distancia_seg) & (distance_RH >
      distancia_seg) & (distance_LH > distancia_seg))
                  haypers_AUX.Value = 0;
            else
                  haypers_AUX.Value = 1;

            // Actualizar los valores de las variables de RAPID.
            using (Mastership m = Mastership.Request(controller.Rapid))
            {
                  haypers.Value = haypers_AUX;
            }
      }
      else
      {
            MessageBox.Show("Automatic mode is required to start
      execution from a remote client.");
      }
}
```

```
catch (System.InvalidOperationException ex)
{
	MessageBox.Show("Mastership is held by another client." +
ex.Message);
}
catch (System.Exception ex)
{
	MessageBox.Show("Unexpected error occurred: " + ex.Message);
}
```

Código 22 – Código C# para comprobar la presencia en la zona de trabajo

```
	private Vector3 FromKntToRbt (Joint articulacion, Pos pos_knt,
Orient rot_knt)
	{
		// Sacar vector desplazamiento
		Vector3 pos_AUX = new Vector3();
		pos_AUX.X = (pos_knt.X / 1000);
		pos_AUX.Y = (pos_knt.Y / 1000);
		pos_AUX.Z = (pos_knt.Z / 1000);

		// Sacar cuaternio de rotación
		Quaternion quat_AUX = new Quaternion();
		quat_AUX.W = (float)rot_knt.Q1;
		quat_AUX.X = (float)rot_knt.Q2;
		quat_AUX.Y = (float)rot_knt.Q3;
		quat_AUX.Z = (float)rot_knt.Q4;

		// Sacar la matriz transformación como rotación * traslacion
		Matrix4x4 mat_trans =
	Matrix4x4.CreateFromQuaternion(quat_AUX) *
	Matrix4x4.CreateTranslation(pos_AUX);

		// Sacar vector de la articulación. 4 dimensiones para
	multiplicar por la matriz de transformación.
		System.Numerics.Vector4 art_AUX = new
	System.Numerics.Vector4();
		art_AUX.W = 1;
		art_AUX.X = articulacion.Position.X;
		art_AUX.Y = articulacion.Position.Y;
		art_AUX.Z = articulacion.Position.Z;

		// Sacar articulación cambiada de base.
		System.Numerics.Vector4 art_RBT = new
	System.Numerics.Vector4();
		art_RBT = System.Numerics.Vector4.Transform(art_AUX,
	mat_trans);

		// Pasar coordenadas a un vector de 3 componentes.
		Vector3 articulacion_RBT = new Vector3();
		articulacion_RBT.X = art_RBT.X * 1000;
		articulacion_RBT.Y = art_RBT.Y * 1000;
		articulacion_RBT.Z = art_RBT.Z * 1000;

		return articulacion_RBT;
	}
```

Código 23 – Método para realizar el cambio de coordenadas entre la cámara y el robot

7.2.5.3. Código RAPID

El código RAPID debe implementar un reloj que genere una interrupción cada 500 ms. Dentro de esa interrupción debe comprobar si el indicar de presencia está a 1 o está a 0. En caso de que este a 1 el robot reducirá su velocidad o se parará y si está a 0 el robot continuará o volverá a la velocidad normal.

Las rutinas TRAP permiten programar interrupciones cuando una variable se ponga a 1. Esa variable va a ser controlada por un reloj de tipo ITimer que pondrá la variable a 1 cada 500 ms. Dentro de la rutina TRAP se programará el cambio de velocidad. Por comodidad se ha omitido la declaración de las posiciones.

```
MODULE Module1

    VAR num haypers_AUX;
    VAR intnum time_int;

    PROC main()
        CONNECT time_int WITH ChangeVelocity;
        ITimer 0.5, time_int;
        ISleep time_int;
        WHILE TRUE DO
            IWatch time_int;
            Path_20;
        ENDWHILE
    ENDPROC

    PROC Path_20()
        MoveJ Target_30,v500,z0,Servo\WObj:=wobj0;
        MoveJ Target_40,v500,z0,Servo\WObj:=wobj0;
        MoveJ Target_50,v500,z0,Servo\WObj:=wobj0;
        MoveJ Target_60,v500,z0,Servo\WObj:=wobj0;
        MoveJ Target_70,v500,z0,Servo\WObj:=wobj0;
    ENDPROC

    TRAP ChangeVelocity
        IF haypers_AUX = 1 THEN
            SpeedRefresh 0;
        ELSE
            SpeedRefresh 100;
        ENDIF
    ENDTRAP
ENDMODULE
```

Código 24 – Código RAPID para la vigilancia de la célula

7.3. LIMITACIONES

La programación de las aplicaciones que necesitan capturar la posición de la mano tiene ciertas limitaciones, ya que no se comprueba la alcanzabilidad de las posiciones capturadas. Esto puede producir problemas a la hora de ejecutar el programa RAPID, ya que si el robot no puede llegar a una posición el programa se para.

Una posible solución para comprobar la alcanzabilidad antes del mover el robot, que finalmente no se ha implementado, sería comprobar los ángulos de los ejes del robot para la posición capturada con la instrucción CalcJointT. En caso de que no sea posible alcanzar la posición capturada, esta instrucción nos devuelve una variable que indica si la posición no se podía alcanzar por estar fuera del área de trabajo o por exceder los límites de los ángulos.

En caso de que la posición fuera alcanzable se ejecutaría la instrucción de movimiento mientras que si no fuera alcanzable el robot no se movería.

```
Joint_10:=CalcJointT(Target_10,tool0\ErrorNumber:=nErrorNum);

IF nErrorNum = ERR_ROBLIMIT THEN
        reachable:=FALSE;
ELSEIF nErrorNum = ERR_OUTSIDE_REACH THEN
        reachable:=FALSE;
ELSE
        reachable:=TRUE;
ENDIF

IF reachable THEN
        MoveJ Target_10,v100,z0,Servo\WObj:=Kinect;
ENDIF
```

Código 25 – Código RAPID para la comprobación de alcanzabilidad

Por otra parte, a la hora de realizar el seguimiento de la mano, el robot no reproduce fielmente el movimiento, ya que el robot no puede seguir todos los puntos de posición de la mano, debido principalmente a dos motivos: primero, el delay introducido entre la puesta a 1 de la señal update_target y la espera a que la aplicación PC SDK la ponga a 0, y segundo, al tiempo de ejecución de las instrucciones de movimiento del robot, que depende de la velocidad del robot y la distancia entre puntos.

Otro aspecto a tener en cuenta es que algunos de los casos desarrollados simplemente buscan capturar la posición de la mano, pero no se han aplicado a situaciones reales en un sistema productivo, por lo que a la hora de ser utilizados puede ser que necesiten código adicional para un correcto funcionamiento.

Con esto, ya se habrían programado los cuatros casos es estudio del apartado 5.3.

8. CONTROL POR GESTOS DE LA APLICACIÓN

En los apartados anteriores se ha desarrollado la aplicación PC SDK que implementa los cuatro casos de estudio (captura de la posición de la mano según usuario y según RAPID, seguimiento de la mano y vigilancia) de tal forma que el operador tenía que hacer click en unos botones.

Tal como se ha explicado en el marco teórico, uno de los retos de la robótica colaborativa es el desarrollo de interfaces intuitivas de forma que la interactuación con el robot sea lo más fácil posible, algo que no se consigue si el operador tiene que hacer click que el robot vaya a la posición de la mano, por ejemplo, ya que tiene que colocar la mano al mismo tiempo que hace click en el botón.

Una de las soluciones, también comentada en el marco teórico, es el control por gestos. En concreto la Kinect reconoce tres estados de la mano: abierta, cerrada y lasso. También puede ser que el estado de la mano sea NotTracked o Unknown, pero no tienen utilidad para la aplicación.

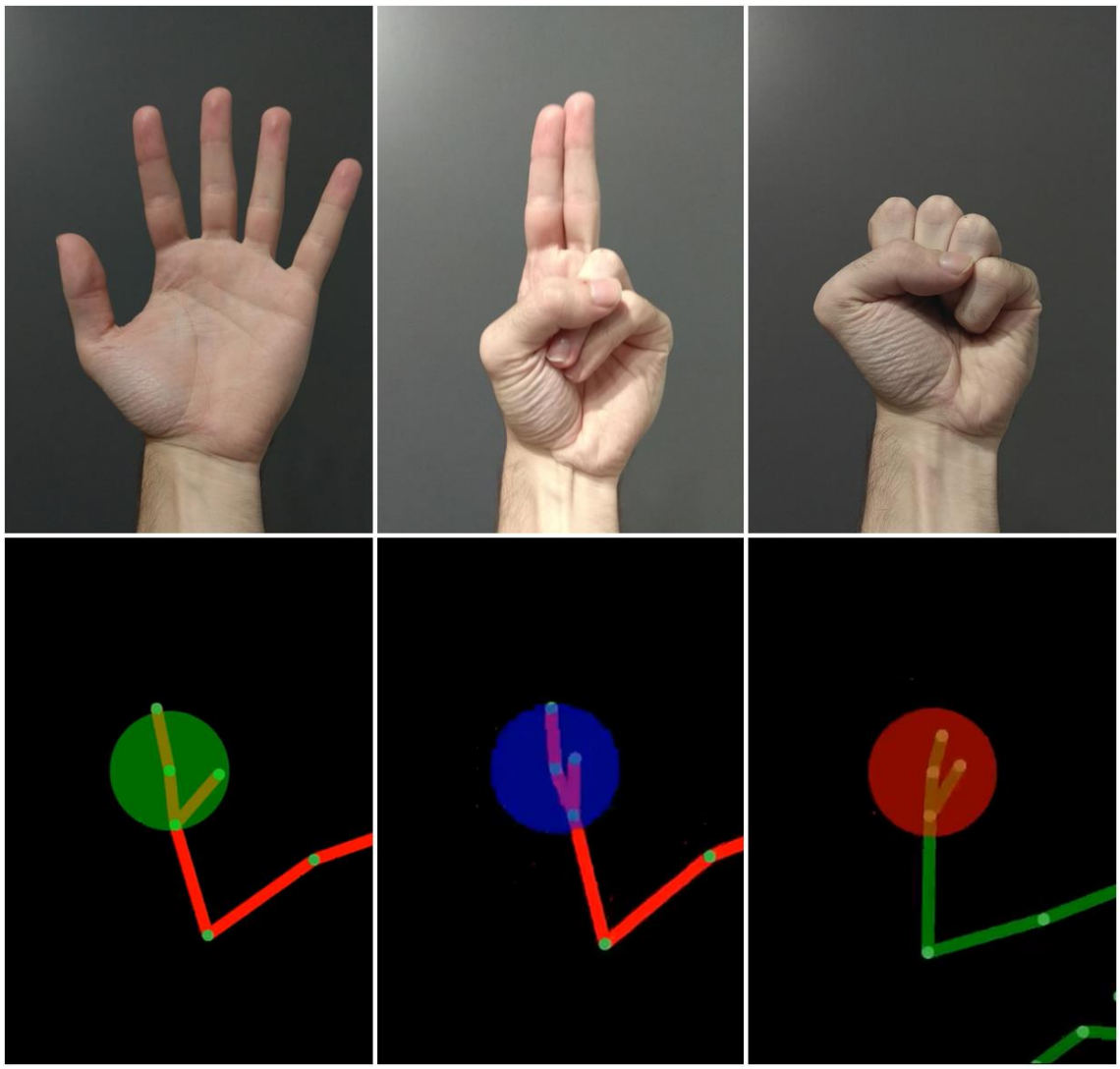


Fig. 8.1 – Estados de la mano reconocidos por la Kinect

A partir de los gestos de la mano reconocidos por la Kinect se puede reprogramar la aplicación anterior para que no se necesite botones.

8.1. PROGRAMACIÓN DE LA KINECT

Al igual que para el caso de las posiciones y las orientaciones, definimos dos variables al principio del código que representen el estado de la mano derecha e izquierda y los métodos para cambiar y obtener el valor de las variables.

```
private HandState rightHS;
private HandState leftHS;

public HandState GetRightHS { get => rightHS; set => rightHS = value; }
public HandState GetLeftHS { get => leftHS; set => leftHS = value; }
```

Código 26 – Declaración de las variables y los métodos para obtener el estado de la mano

Por último, habrá que darles un valor a las variables cuando se detecte un cuerpo.

```
GetLeftHS = body.HandLeftState;
GetRightHS = body.HandRightState;
```

Código 27 – Asignación del valor a las variables del estado de la mano

8.2. PROGRAMACIÓN DE LA APLICACIÓN DE PC SDK

Dentro de la aplicación de PC SDK, habrá que comprobar el estado de la mano y en función de ese estado ejecutar una parte del código u otra.

Para comprobar el estado de la mano se va a implementar un timer que cada 500 ms compruebe el estado de la mano. El timer debe iniciar cuando se cargue la aplicación, por lo que se implementa dentro del gestor de eventos de carga, Form1_Load.

```
aTimer_Hand = new System.Timers.Timer(500);
aTimer_Hand.Elapsed += CheckHandState;
aTimer_Hand.AutoReset = true;
aTimer_Hand.Enabled = true;
```

Código 28 – Código para implementar el timer que compruebe el estado de la mano

Cada 500 ms se ejecutará el código desarrollado en CheckHandState, que debe comprobar el estado de la mano y ejecutar el código correspondiente. Solo utilizaremos los estados de la mano lasso y cerrada, ya que es más habitual tener la mano abierta.

```
HandState rightHS = myKinectWindow.GetRightHS;
HandState leftHS = myKinectWindow.GetLeftHS;
switch (rightHS)
{
       case HandState.Closed:
              CaptureLHPosition();
              break;
}
switch (leftHS)
{
       case HandState.Closed:
              CaptureRHPosition();
              break;
       case HandState.Lasso:
              StartStopRapid();
              break;
}
```

Código 29 – Código de CheckHandState

Con la mano derecha únicamente controlaremos la captura de la posición de la mano izquierda, mientras que con la mano izquierda controlaremos la captura de la mano derecha y el inicio y detención del programa RAPID.

La vigilancia de la célula seguirá siendo activada mediante un botón, ya que una mala detección del estado de la mano puede activar la vigilancia de la célula cuando no se está ejecutando el código RAPID correspondiente.

Para el inicio y detención de la ejecución del código RAPID se ha añadido un booleano que cambia de valor cada vez que se ejecuta el método StartStopRapid(), de esta forma puede controlarse con el mismo gesto. Cuando el usuario haga el gesto lasso con la mano izquierda la ejecución del programa RAPID se iniciará o se detendrá.

El código de los métodos CaptureLHPosition() y CaptureRHPosition() son idénticos a los de la captura de la mano según usuario del capítulo anterior. Estos códigos se ejecutan cuando el usuario cierra la mano.

Para el caso de captura de posición según RAPID y seguimiento de la mano es un poco diferente. Se propone aprovechar la conexión y desconexión del controlador para suscribirse y desuscribirse a la señal update_target, de forma que el operador no tenga que hacerlo manualmente. En ese caso bastaría con cargar el programa RAPID correspondiente y ejecutarlo.

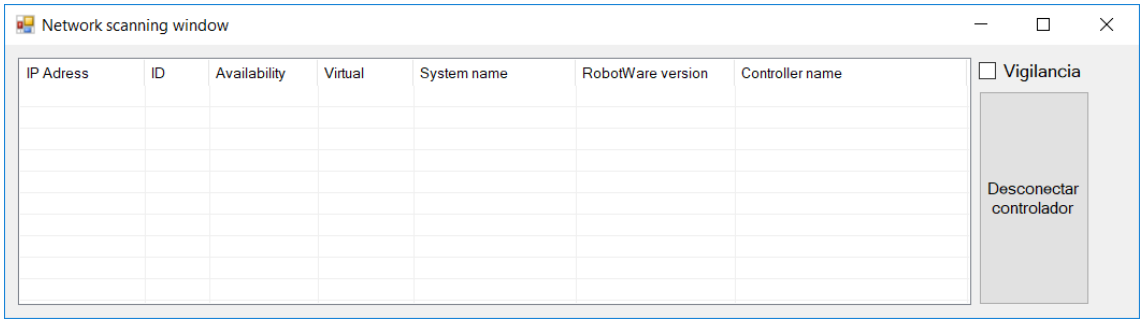


Fig. 8.2 – Interfaz del control por gestos

8.3. LIMITACIONES

El control por gestos implementado en este apartado tiene ciertas limitaciones a la hora de usar la aplicación.

Por un lado, la Kinect solo detecta tres estados que son útiles a la hora de programar el control por gestos: manos abierta, cerrada y lasso. Si suponemos que el operador va a tener la mano abierta mientras utiliza la aplicación solo podemos utilizar la mano cerrada y lasso para controlar la aplicación, lo que nos deja solo cuatro opciones, dos por cada mano.

Por otro lado, el reconocimiento de los gestos por parte de la Kinect es limitado, ya que pueden ocurrir oclusiones, un mal posicionamiento, que produzcan un reconocimiento erróneo de los gestos, algo que puede afectar al funcionamiento de la aplicación.

Una vez comentadas las limitaciones, ya habríamos terminado la programación del control por gestos de la aplicación.

9. CONCLUSIONES

En este trabajo fin de grado se ha llevado a cabo el desarrollo de una aplicación que utiliza las posiciones de partes del cuerpo humano obtenidas con la cámara Kinect V2 para programar un robot como el ABB IRB 120.

En primer lugar, se ha llevado a cabo una revisión bibliográfica sobre el desarrollo de la robótica y la colaboración humano-máquina, además de definir algunos términos y conceptos que serán necesarios para entender buena parte del trabajo. También se ha estudiado diversas técnicas de reconocimiento del esqueleto humano, destacan la tecnología de tiempo de vuelo, que es la utilizada por la Kinect V2.

Posteriormente, se ha llevado a cabo una recopilación de las especificaciones técnicas de los elementos que van a formar la célula robótica: el robot ABB IRB 120 con el controlador IRC5 y la cámara Kinect V2. Del robot se especifican la zona de trabajo, que será necesaria para apartados posteriores. De la cámara Kinect V2 se describen sus prestaciones técnicas, tanto de la cámara RGB como IR y se estudian las condiciones en las que su utilización es más adecuada. También se incluye un análisis de viabilidad de captura dinámica que permite conocer las velocidades a las que se pueden mover los objetos en función de su tamaño y la distancia a la cámara.

A continuación, se incluye una explicación del software utilizado para llevar a cabo el proyecto. Se incluyen los SDKs de la Kinect y de ABB que permitirán programar la cámara y la aplicación de interfaz de operario, el simulador de ABB RobotStudio, que permitirá simular la célula robótica y Microsoft Visual Studio que servirá como entorno de programación.

Una vez descritos los elementos de la célula y el software utilizado se ha realizado un análisis de las posibles aplicaciones de los datos que obtenemos de la Kinect, llegando a la conclusión de que se puede realizar la captura de la mano de dos formas distintas, según usuario y según código RAPID, el seguimiento de la mano y la vigilancia de la célula.

Después, se ha llevado a cabo la decisión de programar la Kinect en lenguaje C# a partir de un código de muestra que viene con el SDK de la Kinect y se ha añadido el código necesario para obtener las posiciones de las manos y de la cabeza.

El siguiente paso ha sido desarrollar la interfaz de operador con el PC SDK. Se ha implementado una aplicación que permite listar los controladores activos en la red y actualizar la lista si se añade o pierde un controlador y que permite conectarse a un controlador de la lista. También cuenta con botones que permiten ejecutar y detener programas. Después se han implementado las cuatro aplicaciones para los datos de la Kinect especificando el funcionamiento, el código C# de la aplicación de PC SDK y el código RAPID necesario para que funcione en la simulación de RobotStudio. Cada aplicación cuenta con un par de botones para que el usuario pueda ejecutar la aplicación requerida. Se han analizado las limitaciones de la aplicación, ya que no se comprueba la alcanzabilidad de las posiciones al capturar las posiciones de las articulaciones, y se propone un método para solucionarlo, además de que en el seguimiento de la mano el robot no realiza el mismo movimiento que la mano y los casos no se han programado para una aplicación concreta de un sistema productivo.

Por último, se ha llevado a cabo la programación de la aplicación para ser controlada por gestos, con el objetivo de conseguir una interfaz de usuario más intuitiva de acuerdo con los retos de la robótica colaborativa. También se han estudiado las limitaciones de esta opción de programación.

9.1. TRABAJO FUTURO

En este trabajo fin de grado se ha trabajo sobre todo en cómo llevar a cabo las cuatro aplicaciones para los datos de la Kinect: realizar la captura de la mano de dos formas distintas, según usuario y según código RAPID, el seguimiento de la mano y la vigilancia de la célula, llevando a cabo una simulación en RobotStudio. Pero no se ha estudiado su implantación en un robot real, ni se han estudiado las incertidumbres existentes al hacerlo: el movimiento del robot no es exacto, por habrá un error al moverse a la posición capturada y esta posición también estará sujeta a los errores de la Kinect al capturar la imagen.

También se han comentado algunas limitaciones del control por gestos de la aplicación. Una línea de trabajo futuro puede ser la mejora de este control. Por un lado, puede programarse para que la aplicación solo reconozca los gestos en una determinada posición de las manos, comprobando la posición y orientación. Por otro lado, se ha comentado que el SDK de la Kinect incorpora un programa que permite la creación de bases de datos de gestos y reconocimiento de gestos en una imagen, lo que ayudaría a aumentar las posibilidades de programación al tener más gestos de la mano reconocibles por la cámara.

BIBLIOGRAFÍA

[1] A. De Luca y F. Flacco, «Integrated control for pHRI: Collisioin avoidance, detection, reaction and collaboration.,» de *Proceeding of the IEEE RAS & EMBS international conference biomedical robotics and biomechatronics (BioRob)*, 2012.

[2] V. Villani, «Survey on human-robot collaboration in industrial settings: Safety, intuitive interfaces and applications,» *Mechatronics,* 2018.

[3] ABB Robotics, Manual del operador - RobotStudio.

[4] J. Molleda Meré, Técnicas de visión por computador para la reconstrucción en tiempo real de la forma 3D de productos laminados, Gijón: Tesis doctoral, Universidad de Oviedo, 2008.

[5] J. F. Montesinos García, Visión por computador con cámara RGB-D par detección de personas en aplicaciones de robótica, Trabajo Fin de Grado, Universidad de Zaragoza, 2017.

[6] P. Gil, T. Kisler, G. J. García, C. Jara y J. A. Corrales, «Calibración de cámaras de tiempo de vuelo: Ajuste adaptatico del tiempo de integración y análisis de la frecuencia de modulación,» *Revista Iberoamericana de Automática e Informática Industrial RIAI,* nº 10, pp. 453-454, 2013.

[7] ABB Robotics, Especificaciones del producto - IRB 120.

[8] ABB Robotics, Manual del operador - IRC5 con FlexPendant.

[9] S. Demitsheva, Gesture basec computer controlling using Kinect camera, Tartu: Bachelor's thesis, Tartu University, 2015.

[10] L. Valgma, 3D reconstruction using Kinect v2 camera, Tartu: Bachelor's thesis, University of Tartu, 2016.

[11] L. Yang, L. Zhang, H. Dong, A. Alelaiwi y A. E. Saddik, «Evaluating and Improving the Depth Accuracy of Kinect for Windows v2,» *IEEE Sensor Journal,* vol. 15, nº 8, pp. 4275-4285, August 2015.

[12] A. Corti, S. Giancola, G. Mainetti y R. Sala, «A metrological characterization of the Kinect V2 time-of-flight camera,» *Robotics and Autonomous Systems,* vol. 75, nº Part B, pp. 584-594, 2016.

[13] E. Lachat, H. Macher, M. -A. Mitet, T. Landes y P. Grussenmeyer, «First experiences with kinect V2 sensor for close range 3D modelling,» Ávila, Spain, 2015.

[14] ABB Robotics, Manual del operador - RobotStudio.

[15] A. Boberg, Virtual lead-through programming: programming virtual robot by demonstration, Bachelor Degree Project, University of Skövde, 2015.

[16] D. Torremocha Ruano, Realización de una interfaz gráfica para el cambio de herramientas en un robot ABB, Proyecto Final de Carrera, Universidad Carlos III de Madrid, 2010.

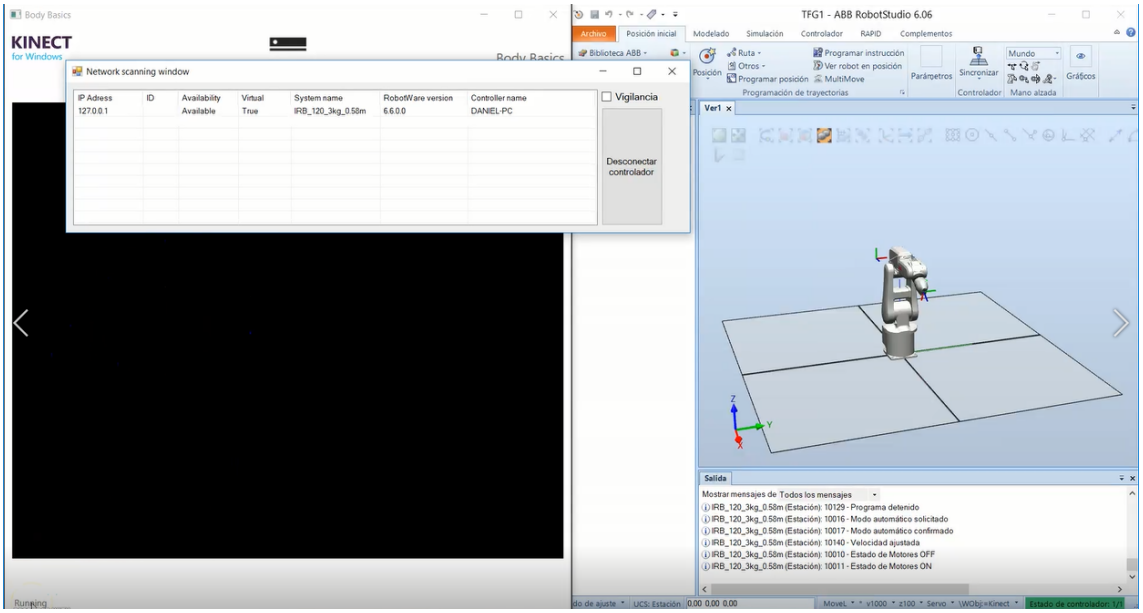
[17] Y. Nitta, «NtKinect - Kinect V2 C++ Programming with OpenCV on Windows10,» [En línea]. Available: http://nw.tsuda.ac.jp/lec/kinect2/.

[18] ABB Robotics, Application manual - PC SDK.

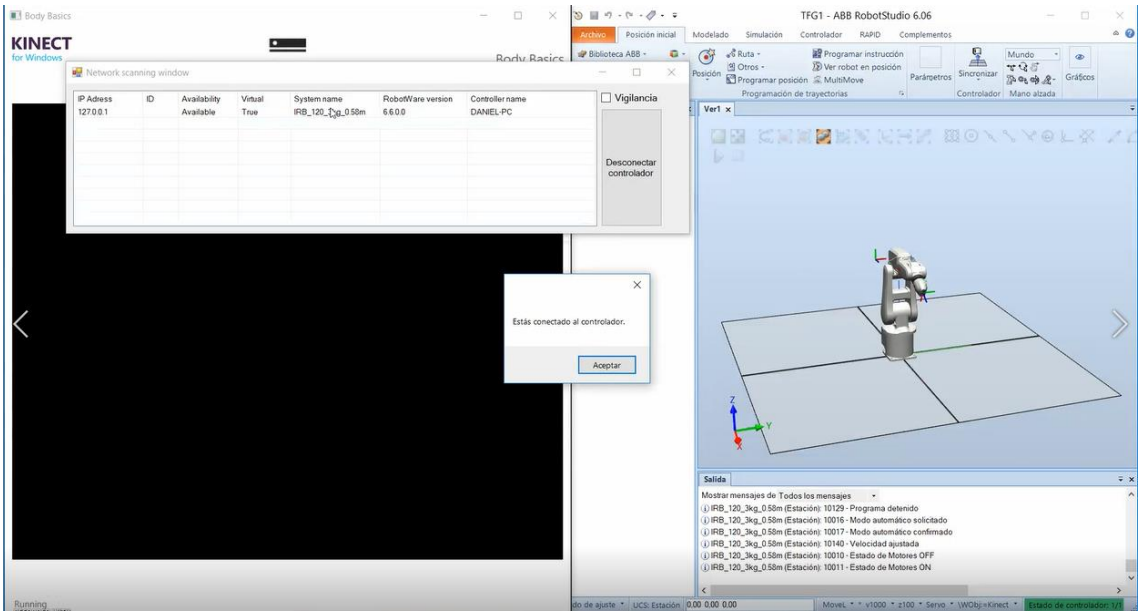
ANEXO I: FUNCIONAMIENTO DE LA APLICACIÓN

Se va a explicar el funcionamiento de la aplicación mediante el control por gestos, con el objetivo de explicar los resultados de los capítulos 7 y 8. Se ha decidido anexarlo para no sobrecargar la última parte del trabajo. Se va a explicar el funcionamiento de la aplicación para los casos de captura de la posición de la mano según usuario, seguimiento de la mano y vigilancia. El caso de captura de la posición de la mano según código RAPID no se explica ya que es similar al caso del seguimiento de la mano.

El primer paso es cargar la aplicación y la estación de RobotStudio. Una vez se haya cargado el controlador virtual de la estación de RobotStudio, este aparecerá en la lista de controladores de la aplicación.



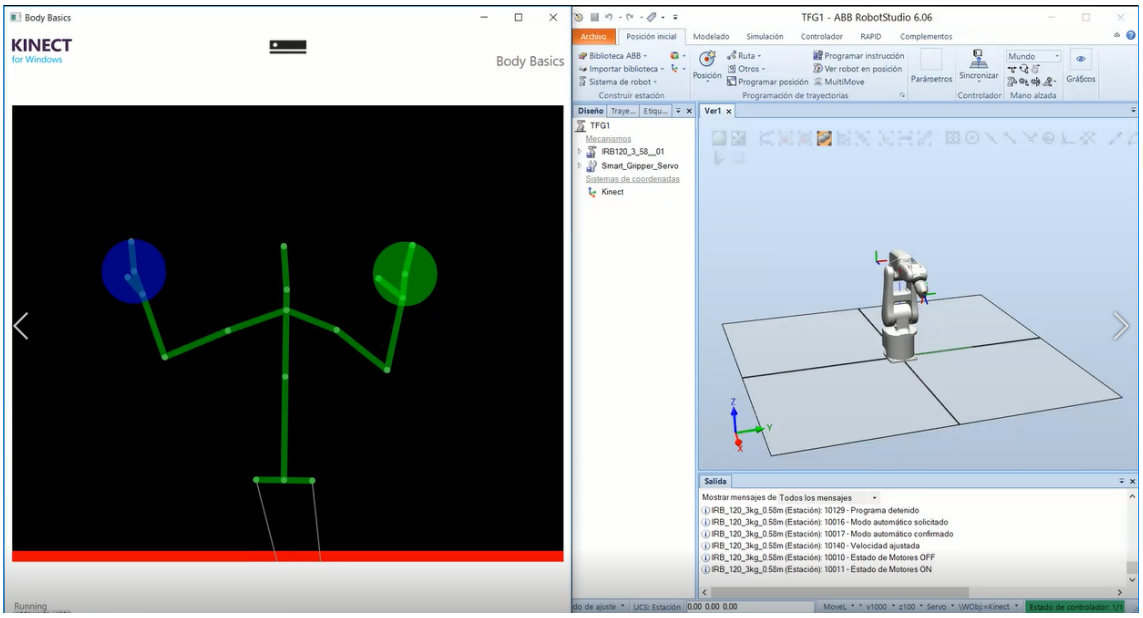
Haciendo doble click sobre el controlador nos conectamos a él. En ese momento aparecerá un mensaje como que estamos conectados.



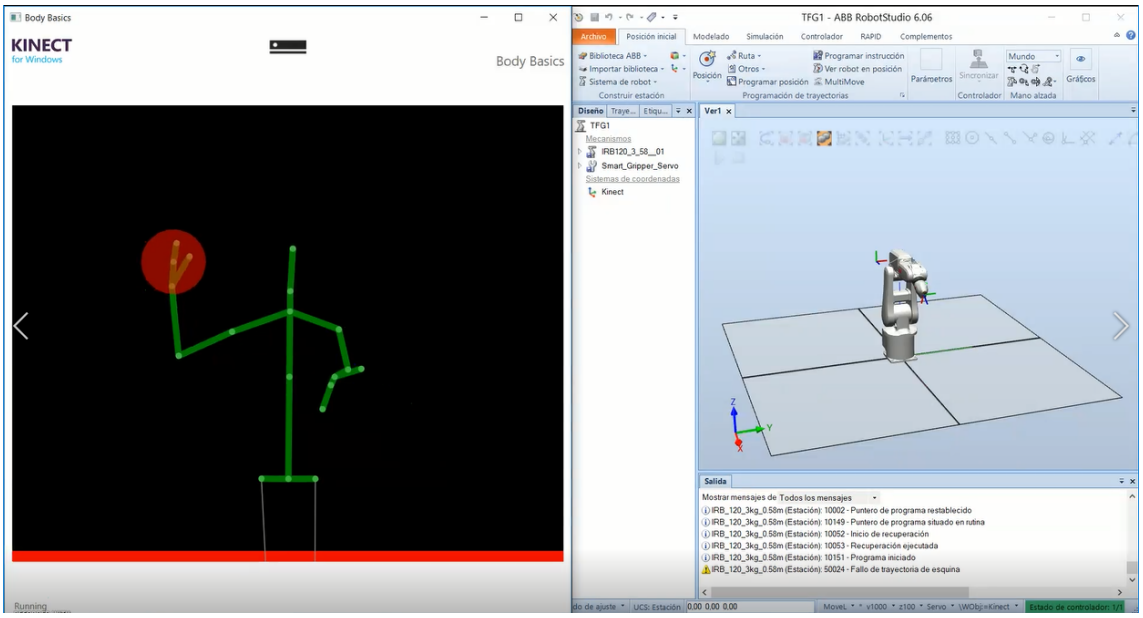
A partir de ahora, los pasos a dar dependerán de la opción que queramos ejecutar: captura de la mano, seguimiento de la mano o vigilancia. Según la opción deberá cargarse el módulo de programa RAPID correspondiente.

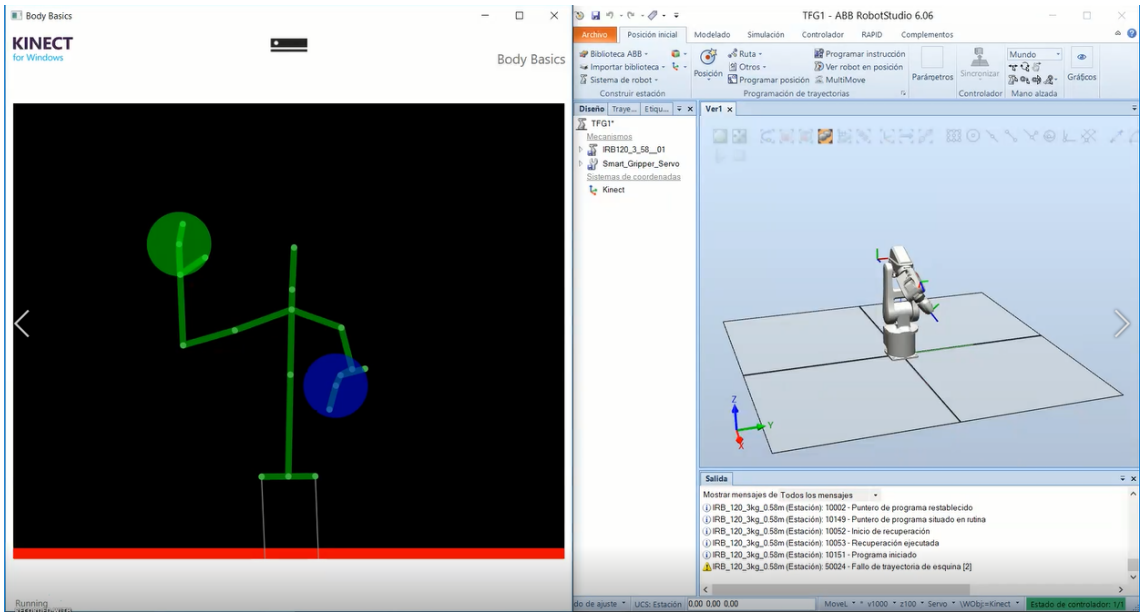
CAPTURA DE LA MANO

En caso de que el módulo de programa cargado en RobotStudio sea el de captura de la mano, el primer paso será iniciar el programa RAPID, lo que se consigue poniendo la mano izquierda en posición lasso:



Una vez se inicia el programa RAPID, éste espera a que el usuario actualice la posición. Para ello, el usuario debe cerrar una de las manos. Si el usuario cierra la mano izquierda, el robot irá a la mano derecha mientras que si cierra la mano derecha irá a la posición de la izquierda.

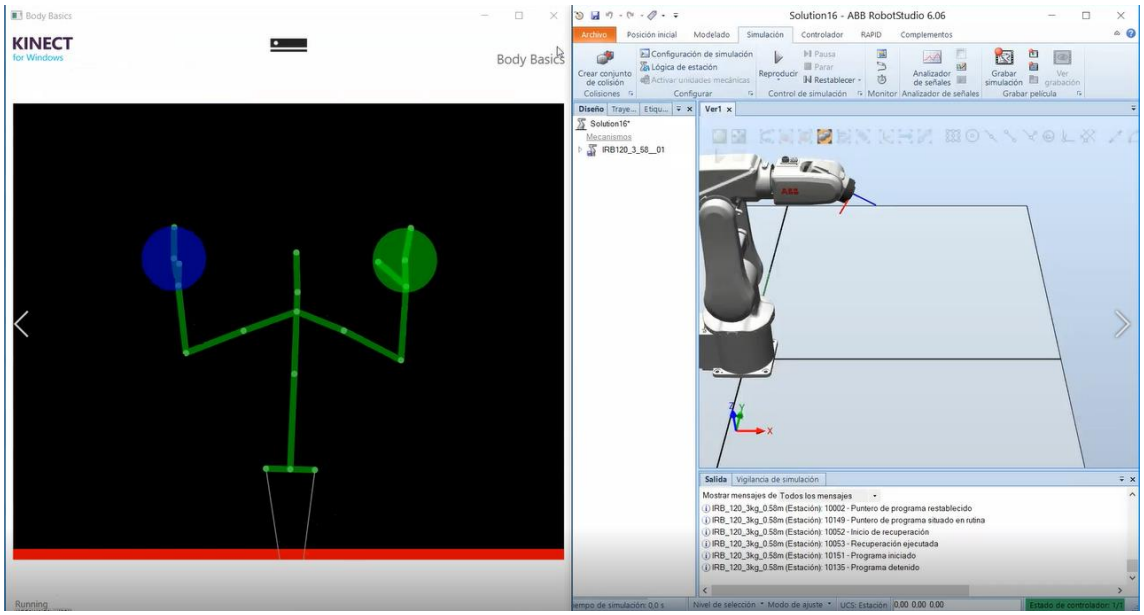




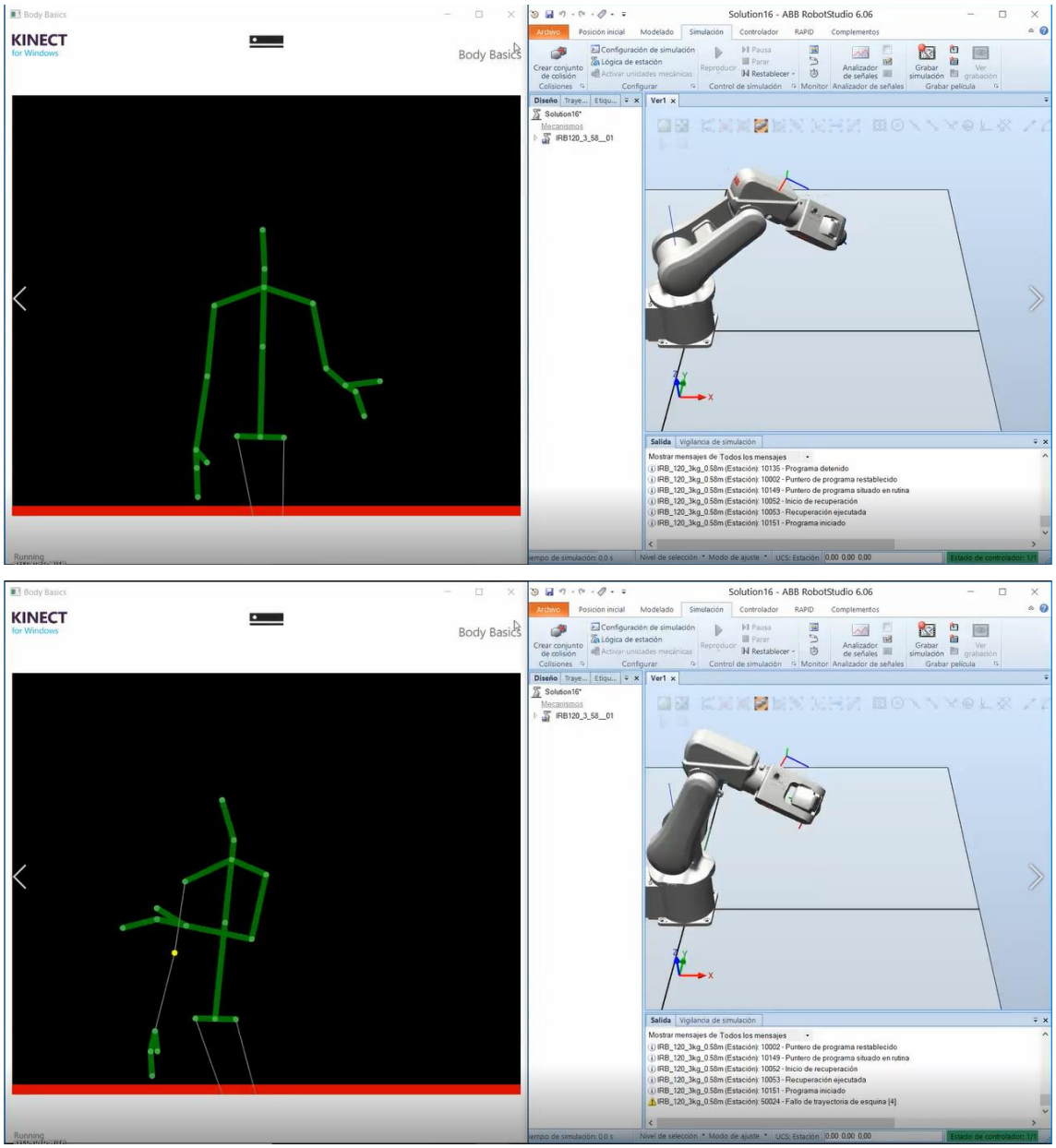
El usuario podrá actualizar las veces que quiera la posición de la mano mientras la posición esté dentro de la zona de trabajo del robot y el movimiento del robot no sobrepase el límite de giro de sus ejes. En caso de que el usuario quiera detener la ejecución del programa RAPID deberá volver a poner la mano izquierda en posición lasso.

SEGUIMIENTO DE LA MANO

En caso de que el módulo de programa cargado en RobotStudio sea el de seguimiento de la mano, el primer paso también será iniciar el programa RAPID, poniendo la mano izquierda en posición lasso.



Nada más iniciarse el programa RAPID, se ejecuta el código de la aplicación que captura la posición de la mano y el código RAPID ejecuta la instrucción de movimiento. Una vez ejecutada la instrucción de movimiento se volverá a capturar la posición. El tiempo entre captura y captura es asíncrono y depende del tiempo que tarda en moverse el robot de una posición a otra.



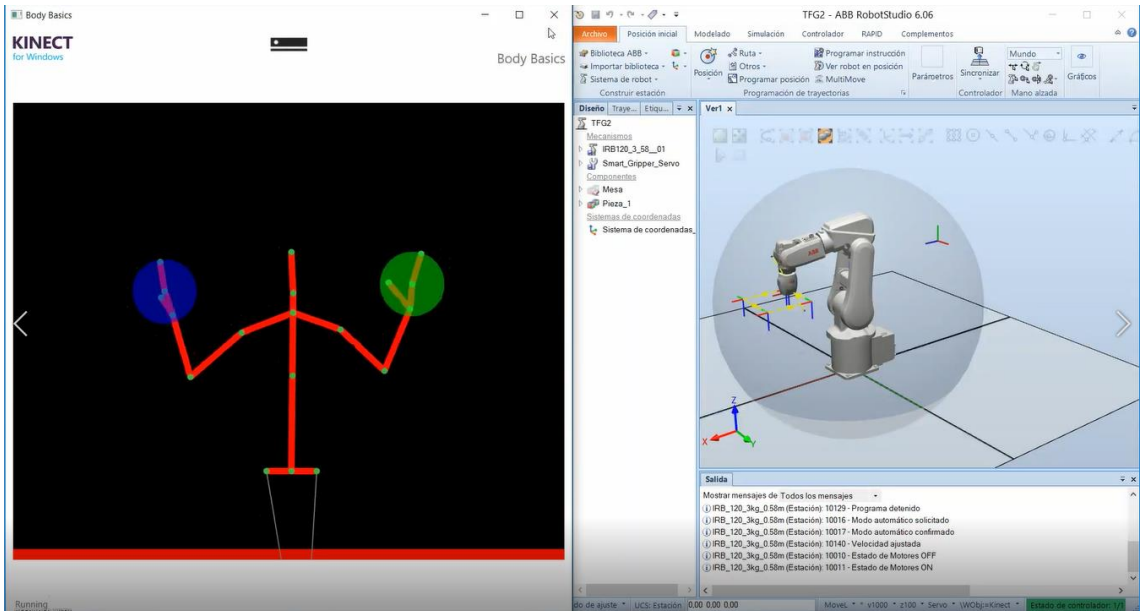
El robot seguirá la posición de la mano mientras se ejecute el programa. Al volver a poner la mano izquierda en posición lasso el programa se detiene.

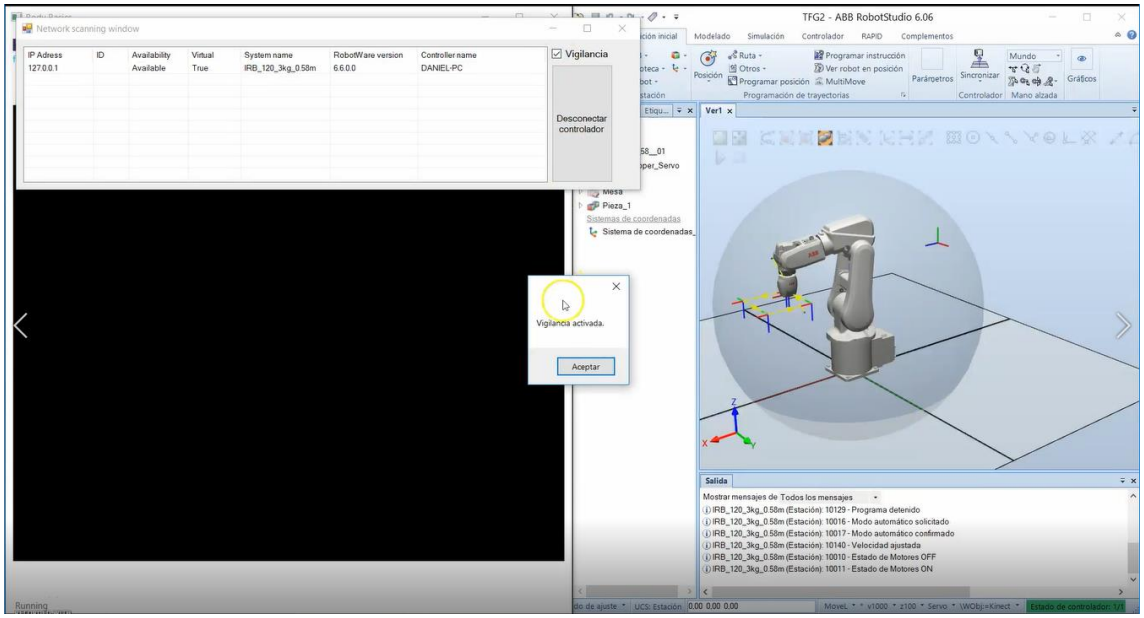
VIGILANCIA DE LA CÉLULA

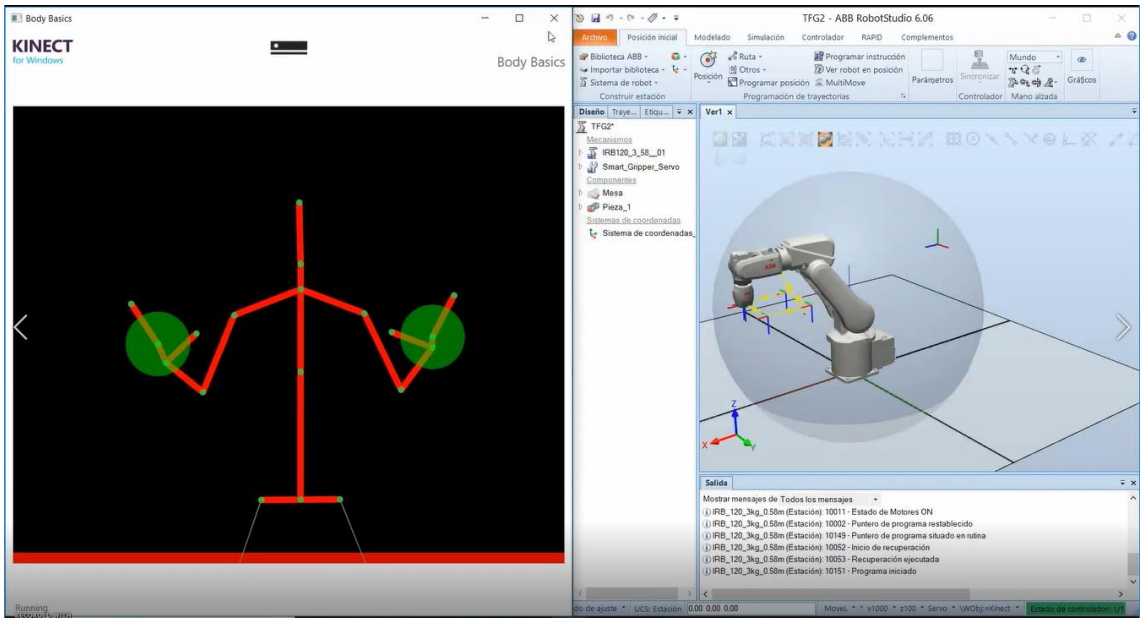
En caso de que el módulo de programa cargado en RobotStudio sea el de vigilancia de la célula, el primer paso también será iniciar el programa de RAPID, poniendo la mano izquierda en posición lasso. Una vez iniciado el programa, el robot ejecuta una trayectoria dentro de un bucle while.

Para inicar la vigilancia de la célula, seleccionamos el botón de vigilancia y aparecerá una ventana indicandonos que la vigilancia esta activada.

Ahora, debemos tener en cuenta la posición de la cámara en la célula, que está situada detrás del robot. Por tanto, al acercarnos a la cámara estaremos acercandonos a la zona de trabajo del robot, por lo que el movimiento de robot se detendrá. Cuando salgamos de la zona de trabajo el robot restablecerá su movimiento.


Body Basics
KINECT
for Windows
Body Basics
TFG2 - ABB RobotStudio 6.06
Archivo
Posición inicial
Modelado
Simulación
Controlador
RAPID
Complementos
Salida
Mostrar mensajes de Todos los mensajes
IRB_120_3kg_0.58m (Estación): 10129 - Programa detenido
IRB_120_3kg_0.58m (Estación): 10016 - Modo automático solicitado
IRB_120_3kg_0.58m (Estación): 10017 - Modo automático confirmado
IRB_120_3kg_0.58m (Estación): 10140 - Velocidad ajustada
IRB_120_3kg_0.58m (Estación): 10010 - Estado de Motores OFF
IRB_120_3kg_0.58m (Estación): 10011 - Estado de Motores ON
Running


Network scanning window
IP Adress
ID
Availability
Virtual
System name
RobotWare version
Controller name
127.0.0.1
Available
True
IRB_120_3kg_0.58m
6.6.0.0
DANIEL-PC
Vigilancia
Desconectar controlador
Vigilancia activada.
Aceptar
TFG2 - ABB RobotStudio 6.06
Salida
Mostrar mensajes de Todos los mensajes
IRB_120_3kg_0.58m (Estación): 10129 - Programa detenido
IRB_120_3kg_0.58m (Estación): 10016 - Modo automático solicitado
IRB_120_3kg_0.58m (Estación): 10017 - Modo automático confirmado
IRB_120_3kg_0.58m (Estación): 10140 - Velocidad ajustada
IRB_120_3kg_0.58m (Estación): 10010 - Estado de Motores OFF
IRB_120_3kg_0.58m (Estación): 10011 - Estado de Motores ON
Running


Body Basics
KINECT
for Windows
Body Basics
TFG2 - ABB RobotStudio 6.06
Archivo
Posición inicial
Modelado
Simulación
Controlador
RAPID
Complementos
Salida
Mostrar mensajes de Todos los mensajes
IRB_120_3kg_0.58m (Estación): 10011 - Estado de Motores ON
IRB_120_3kg_0.58m (Estación): 10002 - Puntero de programa restablecido
IRB_120_3kg_0.58m (Estación): 10149 - Puntero de programa situado en rutina
IRB_120_3kg_0.58m (Estación): 10052 - Inicio de recuperación
IRB_120_3kg_0.58m (Estación): 10053 - Recuperación ejecutada
IRB_120_3kg_0.58m (Estación): 10151 - Programa iniciado
Running

Al volver a seleccionar el botón de vigilancia de la célula desactivamos la vigilancia y aparece un mensaje que así lo indica. Si desactivamos la opción de vigilancia el movimiento del robot no se detiene aunque nos acerquemos a la cámara.