COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE LA REALIDAD VIRTUAL AL DISEÑO Y PROGRAMACIÓN DE LA AUTOMATIZACIÓN DE PROCESOS INDUSTRIALES: VERSIÓN ROBOTS

Autor: Pablo Santiago Giménez Suárez

Director: José Antonio Rodríguez Mondéjar

Madrid

Agosto de 2020

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

APLICACIÓN DE LA REALIDAD VIRTUAL AL DISEÑO Y PROGRAMACIÓN DE LA AUTOMATIZACIÓN DE PROCESOS INDUSTRIALES: VERSIÓN ROBOTS

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2019/20 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.:  Fecha: 23/08/ 2020

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.:  Fecha: 23/08/2020

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE LA REALIDAD VIRTUAL AL DISEÑO Y PROGRAMACIÓN DE LA AUTOMATIZACIÓN DE PROCESOS INDUSTRIALES: VERSIÓN ROBOTS

Autor: Pablo Santiago Giménez Suárez

Director: José Antonio Rodríguez Mondéjar

Madrid

Agosto de 2020

Agradecimientos

En esta sección me gustaría hacer mención especial a una gran cantidad de personas que me han apoyado a lo largo de estos y cuyo apoyo ha sido inestimable.

En primera instancia a mis padres y hermano. Me supieron dar ánimos en los momentos más duros y nunca dejaron de creer en mí.

Por supuesto a mis amigos Adrian García y Jaime Cibran por su conocimiento de Unity, el cual ha sido de gran ayuda.

También querría agradecer a mis compañeros de clase por su ayuda y consejo. Entre ellos me gustaría mencionar a Manuel Trabado de la Cruz, Carlos Geûens, Felipe Araujo, Javier Colinas, Nora Segura, Ángel Sánchez y Pablo Bobo.

APLICACIÓN DE LA REALIDAD VIRTUAL AL DISEÑO Y PROGRAMACIÓN DE LA AUTOMATIZACIÓN DE PROCESOS INDUSTRIALES: VERSIÓN ROBOTS

Autor: Giménez Suárez, Pablo Santiago.
Director: Rodríguez Mondéjar, José Antonio.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Debido al avance de tecnologías como la realidad virtual y software de simulación avanzado se busca desarrollar un programa que aproveche las ventajas que estas tecnologías pueden aportan en la programación y simulación de procesos para la automatización de las cadenas de producción que emplean brazos robóticos, en este caso enfocado a la minifábrica de ICAI.

Palabras clave: ABB, Robot Studio, Unity, Realidad Virtual, TCP/UDP, C#, RAPID.

1. Introducción

Desde la aparición de la realidad virtual se ha buscado la forma de integrarla con los métodos convencionales de diseño y verificación por el gran realismo que aporta. Los motores gráficos como Unreal Engine [**1**] o *Unity* [**2**], que son principalmente usados en el mundo del videojuego, nos aportan una gran libertad a la hora de implementar la tecnología de la realidad virtual y un enorme control sobre las simulaciones que se quieran realizar.

Cabe mencionar que esta libertad no es gratuita puesto que se requiere un gran número de horas de programación y conocimiento sobre estos sistemas para poder sacar el máximo provecho y crear programas completos y a prueba de errores.

2. Definición del proyecto

En este proyecto se han buscado principalmente dos objetivos:

- Aprovechar la potencia del motor gráfico de *Unity* para crear simulaciones controladas
- Implementar la realidad virtual para agilizar la programación de procesos industriales

Sin embargo, debido a las circunstancias de la cuarentena no se ha tenido acceso al hardware de realidad virtual y por ello no se ha podido desarrollar una interfaz e interacciones con el entorno que son típicas de esta tecnología.

Por tanto, se ha enfocado el proyecto en crear un entorno virtual en *Unity* que mejore respecto a los anteriores proyectos en lo referente a la similitud con las prestaciones que aporta la aplicación de *RobotStudio* [**3**] y el sistema FlexPendant.

Por ejemplo, en el proyecto de Francisco Javier Domínguez Sánchez-Girón [**4**] se usa un sistema que solo permite un movimiento cada vez lo que limita la precisión y tipos de movimientos disponibles.

3. Descripción del sistema

El sistema desarrollado consta de dos bloques claramente diferenciados que se comunican entre sí de dos formas diferentes.

El primer bloque está constituido por el programa de *Unity* desarrollado prácticamente desde cero a excepción del modelo del brazo robótico. Este es el bloque principal y es en el que se manejará el usuario, que se moverá por un entorno 3D en primera persona con dos tipos de movimiento diferentes. Uno de estos movimientos estará ligado a las físicas del programa y el otro proporciona un movimiento libre en los 3 ejes del espacio.

El usuario contará con una interfaz que le permitirá interactuar con el entorno y acceder de forma visual a la información relevante.

Una vez el usuario haya configurado el entorno y el proceso a su gusto podrá sincronizar la información con *RobotStudio* que es el encargado de gestionar el movimiento del robot. Para ello Unity usará dos tipos de comunicación.

El primero es usando el *Protocolo TCP/IP*, un tipo de comunicación segura pero lenta comparada con el *Protocolo UDP/IP*, aunque este último no garantiza la recepción de los datos.

El segundo bloque de programa, que se ha desarrollado en RAPID (el lenguaje de programación de *RobotStudio)* se encargará de acomodar la información recibida de *Unity* a los parámetros de *RobotStudio* para proporcionar las funcionalidades deseadas.

Mientras el proceso configurado se esté simulando *RobotStudio* usará el *Protocolo UDP/IP*, por su velocidad, para actualizar los ángulos de las articulaciones del robot a *Unity* de tal manera que el usuario podrá ver como se está desarrollando el proceso en tiempo real.

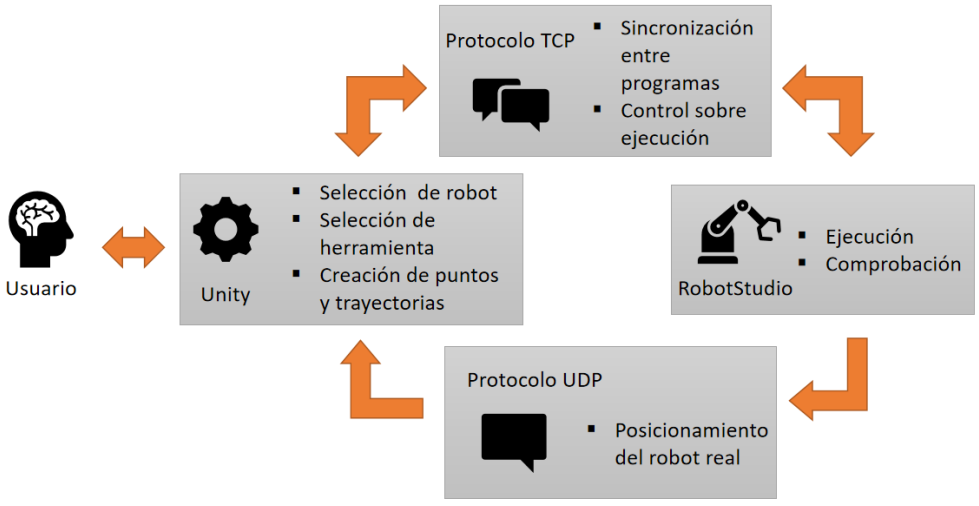

Figura 1 - Esquema de funcionamiento del programa desarrollado

4. Resultados

- Se ha creado un programa base que puede ir escalando en complejidad con más tiempo de desarrollo.
- Se ha superado la limitación del programa anterior para la precisión y tipo de movimientos
- Se ha optimizado el código para llegar a la tasa de fotogramas por segundo que requiere un hardware de realidad virtual.

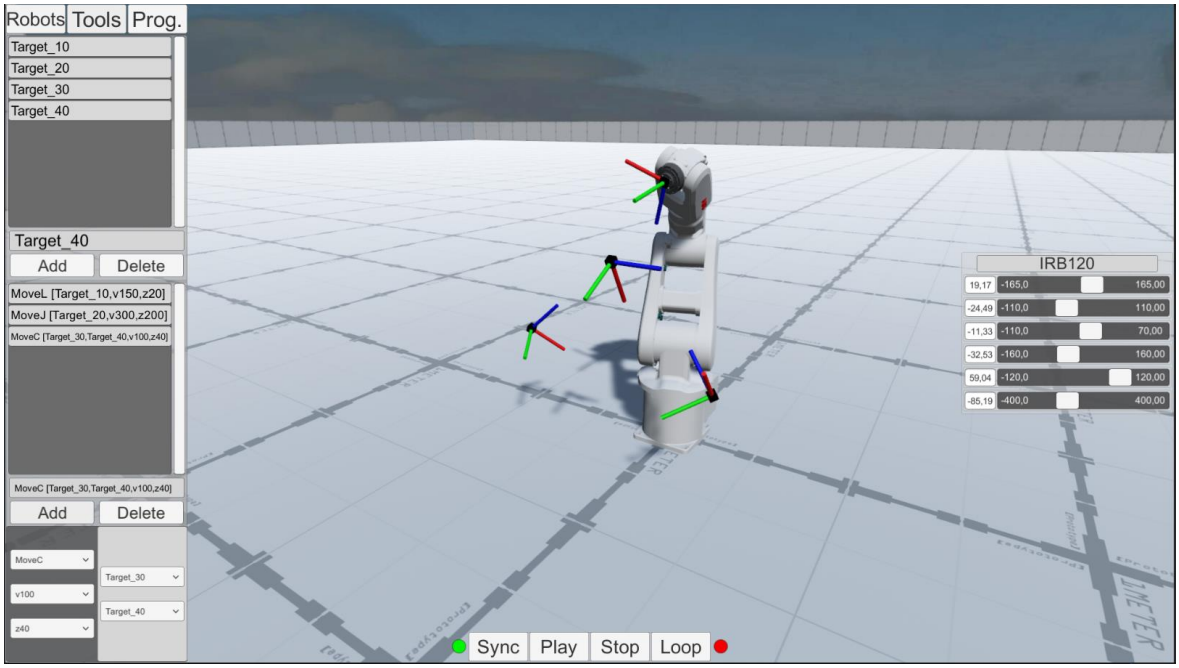

Figura 2 - Visual del programa desarrollado en Unity

5. Conclusiones

Existen algunas limitaciones en el programa de *Unity* debidas a *RobotStudio*. En concreto cabe destacar que, al usar tipos de movimiento complejos como *MoveC* se pueden producir errores que no se pueden tratar y paran la ejecución del código RAPID.

Por otro lado, se han logrado la mayor parte de los objetivos buscados y queda margen de mejora para el programa en general. Algunas de las funcionalidades que se deberían añadir en un futuro son:

- Interfaz e interacción de realidad virtual, puesto que era uno de los objetivos originales del proyecto y que por circunstancias de fuerza mayor no se ha podido desarrollar.
- Más herramientas para el robot, como una de tipo pinza, por ejemplo.
- Capacidad de seleccionar objetos de la escena para poder moverlos y rotarlos libremente para recrear una escena más realista.
- Debería estudiarse la posibilidad de reducir el número de clases usadas rediseñándolas para ser más genéricas.

APPLICATION OF VIRTUAL REALITY TO THE DESIGN AND PROGRAMMING OF INDUSTRIAL PROCESS AUTOMATION: ROBOT VERSION

Author: Giménez Suárez, Pablo Santiago.
Supervisor: Rodríguez Mondéjar, José Antonio.
Collaborating Entity: ICAI - Comillas Pontifical University

ABSTRACT

Due to the evolution of technologies such as virtual reality and advanced simulation software, the aim is to develop a program that takes advantage of the benefits that these technologies can provide in the programming and simulation of processes for the automation of production lines that use robotic arms, in this case focused on ICAI's mini-factory.

Keywords: ABB, Robot Studio, Unity, Virtual Reality, TCP/UDP, C#, RAPID.

1. Introduction

Since the arrival of virtual reality, we have sought to integrate it with conventional design and verification methods because of the great realism it brings. Graphic engines such as Unreal Engine [**1**] or *Unity* [**2**], which are mainly used in the world of video games, give us great freedom when implementing virtual reality technology and enormous control over the simulations that we want to make.

It is worth mentioning that this freedom is not free since they require a great number of hours of programming and knowledge about these systems to get the most out of them and create complete and error-proof programs.

2. Project definition

Two main objectives have been pursued in this project:

- To harness the power of *Unity*'s graphics engine to create controlled simulations
- Implementing virtual reality to speed up industrial process scheduling.

However, due to the circumstances of the quarantine, there has been no access to the virtual reality hardware, so it has not been possible to develop an interface and interactions with the environment that are typical of this technology.

Therefore, the project has focused on creating a virtual environment in *Unity* that improves with respect to the previous projects in terms of similarity with the features provided by the *RobotStudio* [**3**] application and the FlexPendant system.

For example, in Francisco Javier Domínguez Sánchez-Girón's project [**4**] the system used allows only one movement at a time, which limits the accuracy and types of movements available.

3. System description

The system developed consists of two clearly differentiated blocks that communicate with each other in two different ways.

The first block is made up of *Unity*'s program developed practically from scratch apart from the model of the robotic arm. This is the main block and it is where the user will operate, moving through a 3D environment in first person with two different types of movement. One of these movements will be linked to the physics of the program and the other provides free movement on all 3 axes of space.

The user will have an interface that will allow him/her to interact with the environment and visually access the relevant information.

Once the user has configured the environment and process to his liking, he can synchronize the information with *RobotStudio*, which manages the robot's movement. To do this, *Unity* will use two types of communication.

The first is by using the *Protocolo TCP/IP*, a safe but slow communication compared to the *Protocolo UDP/IP*, although the second one does not guarantee reception of the data.

The second program block which is developed in RAPID, *RobotStudio*'s programming language, will accommodate the information received from *Unity* to *RobotStudio*'s parameters to provide the desired functionality.

While the configured process is being simulated, *RobotStudio* will use the *Protocolo UDP/IP*, because of its speed, to update the robot's joints angles to *Unity* so that the user can see how the process is running in real time.

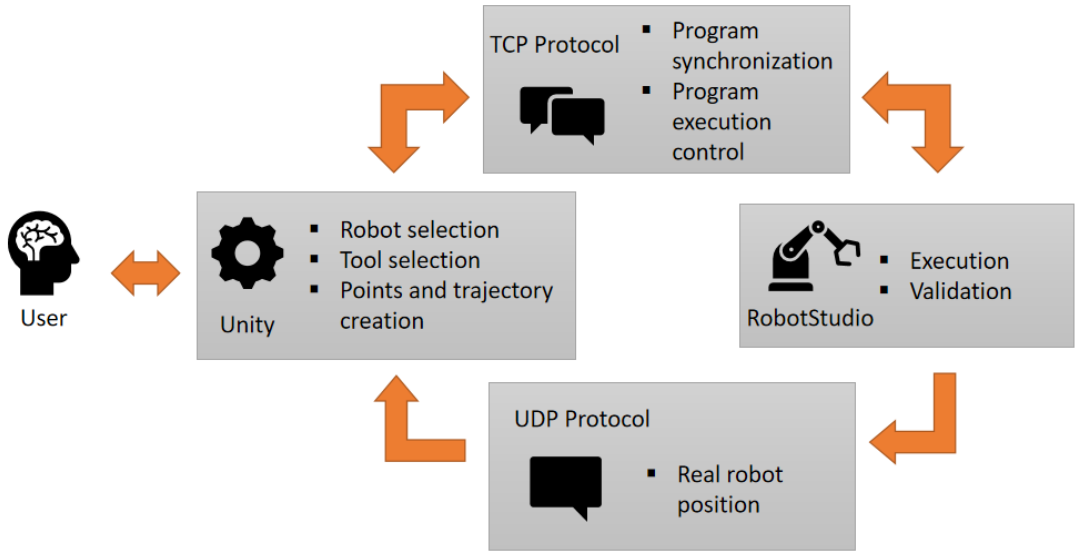

Figure 3 - Diagram of the operation of the developed program

4. Results

- A base program has been created that can be scaled in complexity with more time of development.
- The limitation of the previous program for the precision and type of movements has been overcome.
- The code has been optimized to achieve the frame rate required by virtual reality hardware

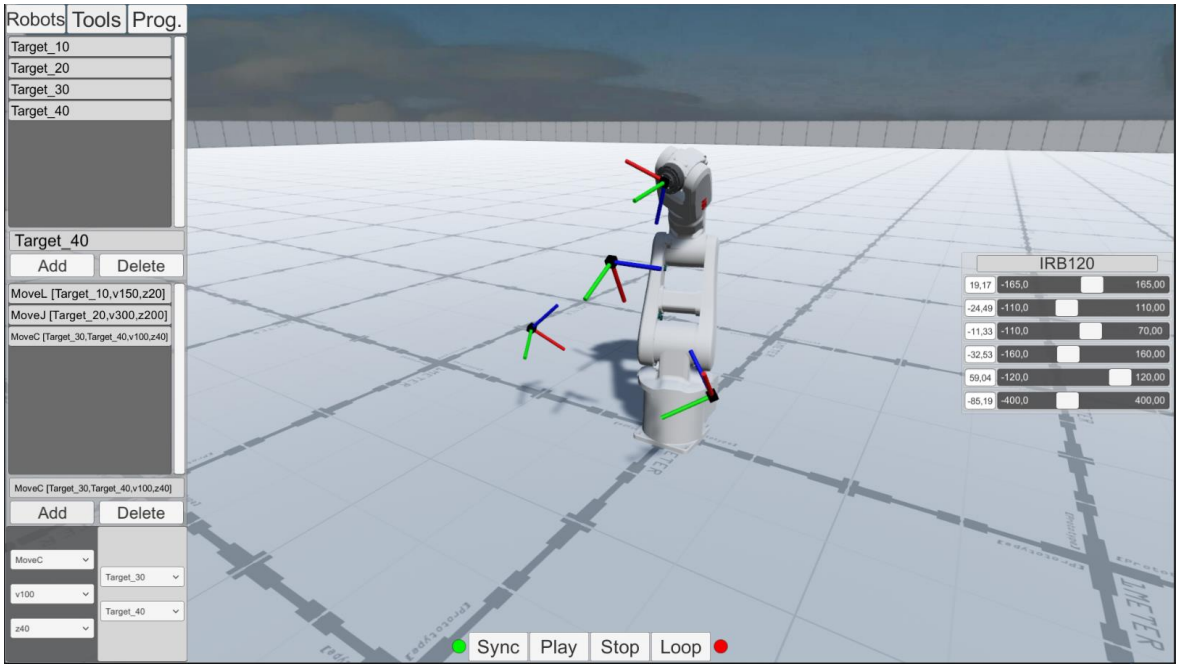

Figure 4 - View of the program developed in Unity

5. Conclusions

There are some limitations in the *Unity* program due to *RobotStudio*. In particular, when using complex motion types like *MoveC*, errors can occur that cannot be handled and will stop RAPID code execution.

On the other hand, most of the desired targets have been achieved and there is room for improvement for the program as a whole. Some of the functionalities that should be added in the future are

- Virtual reality interface and interaction, since it was one of the original objectives of the project and due to major circumstances, it has not been possible to develop it.
- More tools for the robot, such as a gripper type, for example.
- Ability to select objects from the scene to be able to move and rotate them freely to recreate a more realistic scene.
- Consideration should be given to reducing the number of classes used by redesigning them to be more generic.

Índice de la memoria

ÍNDICE DE FIGURAS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

Capítulo 1. INTRODUCCIÓN

Desde el origen de la era industrial la automatización de los procesos industriales ha ido evolucionando y mejorando la manera en la que hemos controlado las máquinas para que realicen las tareas para las que las diseñamos. Parece que ya queda lejos cuando entre 1960 y 1970 se usaban tarjetas perforadas para programar [**5**].

Desde entonces hemos ido actualizando nuestro hardware y software para facilitar y agilizar tanto al programador como a los usuarios convencionales el uso de los equipos y el desarrollo de sistemas muchísimo más complejos. En este campo podemos hablar, por ejemplo, de mejoras en las pantallas, tanto en resolución como número de fotogramas por segundo, el tamaño, etc.

Estos avances nos han facilitado la forma en la que diseñamos, controlamos y simulamos los procesos. Y aunque es verdad que actualmente lo más extendido es el uso de monitores y software avanzado que usan el teclado y ratón convencional, cada vez más se usan sistemas táctiles para el diseño y control de información a través de la pantalla.

Pero hay una tecnología diferente que se ha ido abriendo paso en la industria (tanto en el ambiente propiamente industrial como en el del entretenimiento) desde hace unos años, que es la realidad virtual. Así como otra que ha pasado algo más desapercibida, la realidad aumentada, que a diferencia de la RV que crea un mundo simulado al que el usuario accede, superpone el mundo virtual al real.

Según se ha ido desarrollando la tecnología, la realidad virtual se ha ido extendiendo en un gran número de sectores, pero donde más se ha hecho notar ha sido en el mundo del entretenimiento (ya sea por videojuegos o por turismo) seguido de la sanidad, donde supone un cambio de paradigma para los doctores a la hora de afrontar operaciones o para los hospitales a la hora del entrenamiento de nuevos cirujanos. Una función que tampoco ha

pasado desapercibida para las fuerzas militares, que emplean esta tecnología para el entrenamiento de sus soldados [**6**].

Como no podía ser de otra manera, la realidad virtual también ha hecho acto de presencia en el mundo de la ingeniería. La NASA, por ejemplo, ha recreado la Estación Espacial Internacional para así poder preparar a los nuevos astronautas. Pero uno de los sectores donde es realmente útil es en el de la automoción donde los ingenieros son capaces de ver un vehículo a tamaño real totalmente montado para así comprobar si las proporciones, el diseño y su funcionamiento son los esperados [**7**].

El proyecto por desarrollar aprovechará las ventajas de la realidad virtual respecto a la inmersión del usuario en un entorno creado o simulado para crear una mejora significativa respecto al típico programa de ordenador que se gestiona a través de una consola de mandos o teclado y ratón. Para ello, se tomará como base el trabajo de fin de grado de Javier Domínguez Sánchez-Girón en 2019 [**4**].

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

2.1 UNITY

Unity es un motor gráfico multiplataforma para la creación de videojuegos desarrollado por Unity Technologies [**2**]. Unity será el núcleo de este proyecto.

Se ha seleccionado esta tecnología debido a varios motivos:

- Versatilidad y compatibilidad con programas externos.
- Existencia de una licencia gratuita
- Proporciona paquetes de recursos externos con gran variedad de funcionalidades, *Prefab* y componentes ya desarrollados que agilizan la producción de proyectos específicos.

Será con ayuda de esta plataforma que se desarrollará el entorno de programación del brazo robótico *ABB IRB120*. De esta manera será clave para el desarrollo de las comunicaciones con *RobotStudio* y la interacción con el usuario.

Se puede consultar una lista de conceptos básicos de Unity aquí: ANEXO I: Conceptos básicos de Unity

2.1.1 PROTOCOLO TCP/IP

Se trata de un protocolo de comunicación de red [**8**]. Permite la comunicación en ambos sentidos.
Este protocolo aporta fiabilidad a cambio de velocidad. Es decir, a cambio de unas comunicaciones más lentas asegura que la información se ha recibido correctamente en el destino.
En este protocolo emisor y receptor tienen que establecer conexión antes de poder enviarse información, si no se producirán errores a la hora de intentar enviar mensajes por el puerto correspondiente.

2.1.2 PROTOCOLO UDP/IP

Se trata de un protocolo de comunicación de red [**9**]. Permite la comunicación en único sentido.

Este protocolo aporta velocidad a cambio de fiabilidad. Es decir, a cambio de unas comunicaciones más rápidas no asegura que la información se ha recibido correctamente en el destino.

En este protocolo se envía información por un puerto de comunicaciones y a este se pueden conectar varios receptores sin necesidad de hacérselo saber al emisor. De esta manera el receptor recibirá mensajes a la máxima velocidad que pueda. Esto ocasiona que si se envían mensajes a una velocidad superior a la que el receptor es capaz de procesar se pierda el exceso.

2.2 ROBOTSTUDIO

Es una de las herramientas de programación de robots más utilizada en el mundo [**3**].

Es propiedad de ABB que es la misma empresa que desarrolla el brazo robótico que se emplea en la minifábrica de ICAI, el *ABB IRB120*.

2.2.1 LENGUAJE RAPID

Es el lenguaje de programación que utiliza RobotStudio para la programación de los robots.

Se puede consultar una lista de conceptos básicos de RAPID aquí: *ANEXO II: Conceptos básicos de RAPID*

2.2.2 ABB IRB120

Modelo básico de brazo robótico de la familia ABB [**10**].

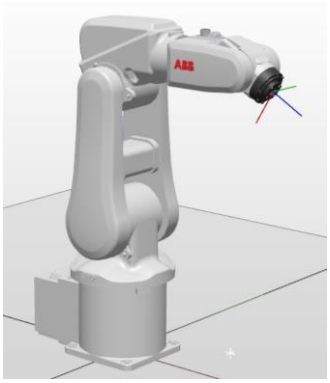

Ilustración 1 - ABB IRB120

El IRB120 es un brazo robótico con seis grados de libertad. Es el brazo robótico más pequeño de la gama ofertada por ABB.

Puesto que es el brazo robótico disponible en la minifábrica de ICAI, será el usado para desarrollar este proyecto.

2.3 AUTODESK FUSION 360

Aplicación de modelado que unifica diseño, ingeniería, electrónica y fabricación en una sola aplicación [**11**].

Se usará esta aplicación para simplificar las mallas del robot. De esta manera se consumirán menos recursos cuando se detecten colisiones. Este es un paso importante puesto que en la realidad virtual se requiere una tasa de fotogramas alto.

Capítulo 3. ESTADO DE LA CUESTIÓN

Actualmente existen diferentes proyectos con enfoques distintos a la hora de utilizar la realidad virtual/aumentada para la programación. Algunos de ellos son bastante avanzados, pero en general es una tecnología que aún tiene que perfeccionarse antes de llegar a un estándar extendido en la industria. En el sector de la robótica, a pesar de que lleva un par de años sin haber publicaciones de gran relevancia, podemos destacar los siguientes:

- Proyecto del laboratorio CSAIL del MIT basado en la aplicación de la realidad virtual para el control remoto de robots donde se usa la realidad virtual para controlar de forma sencilla un par de brazos robóticos. [**12**]
- Artículo del departamento de mecánica aplicada y robótica de la facultad de ingeniería mecánica y aeronáutica Rzeszow, Polonia. En el que se desarrolla la automatización del proceso de pulimiento de piezas de cerámica a través de la captura del movimiento de un artesano que posteriormente es digitalizado e implementado en un robot. [**13**]
- Artículo del departamento de ingeniería mecánica de la universidad nacional de Singapur en el que se desarrolla la programación de robots asistida por equipo de realidad aumentada. [**14**]
- Herramienta de simulación con realidad virtual desarrolladas por ABB e integradas en su programa de control de robots, *RobotStudio*. Permite una gran inmersión y es altamente preciso, pero no permite la programación directa de sus brazos robóticos. [**15**]

Capítulo 4. DEFINICIÓN DEL TRABAJO

4.1 OBJETIVOS

El proyecto tenía inicialmente como objetivos:

- Solventar la limitación presente en el proyecto anterior [**4**]. Esta forzaba a realizar una acción por vez, lo que implica que el robot se detenía brevemente en cada punto programado y que se ignorara la configuración de precisión del movimiento.
- Proporcionar al usuario una programación más sencilla y ágil que la presente en *RobotStudio* sin perder funcionalidad.
- Mejorar la interacción e interfaz presente en el proyecto anterior [**4**].

Debido a la pandemia algunos de estos puntos se han visto modificados tal y como se desarrollará en el *Capítulo 11*.

4.2 METODOLOGÍA

Para resolver el problema planteado usaremos motores gráficos normalmente empleados a la hora de crear videojuegos. Se seguirán los siguientes pasos para diseñar los diferentes bloques del proyecto:

- Protocolos de comunicación entre los diferentes programas que formaran el proyecto completo, *RobotStudio* y *Unity*.
- Una interfaz práctica que permita el acceso a todas las funciones del programa.
- El programa será creado principalmente en C# y aprovechará las funciones que este lenguaje de programación proporciona.
- Se implementará el modelo CAD del brazo robótico IRB120 de ABB para darle más realismo a la simulación que representará el proceso.

4.3 PLANIFICACIÓN

En la *Figura 5* se puede observar la planificación inicial del proyecto.

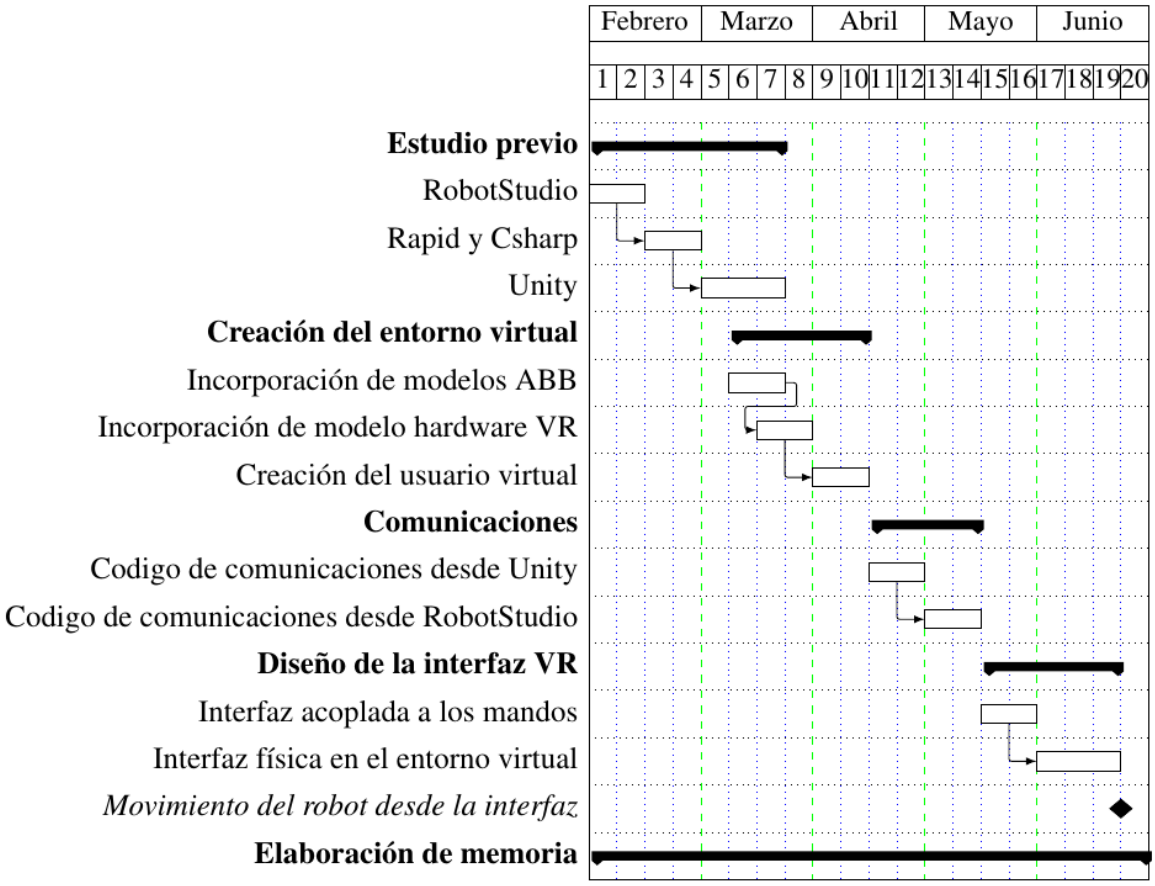

Figura 5 - Planificación del proyecto

Debido a la pandemia y problemas organizativos se han ido produciendo retrasos y cambios en ella. Por ejemplo, el tiempo dedicado a la sección de realidad virtual ha sido modificado para el diseño de la interfaz de primera persona.

Capítulo 5. ARQUITECTURA DEL SISTEMA

5.1 ÁRBOL DE FUNCIONES

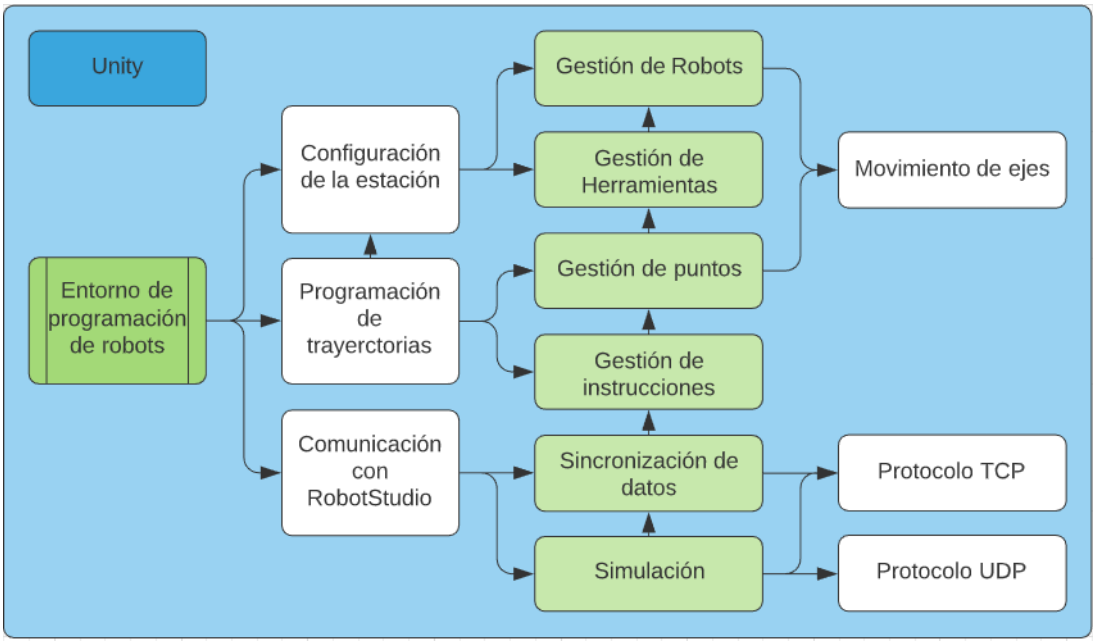

Figura 6 - Árbol de funciones del sistema.

En la *Figura 6* se muestra el árbol de funciones que se ha seguido para desarrollar el proyecto. Este árbol responde en el sentido de las flechas el ¿Cómo se hace? Mientras que el recorrerlo en el sentido contrario responde al ¿Por qué? Se remarca cada elemento.

Los elementos remarcados en verde marcan las funcionalidades principales que definen el programa desarrollado.

Todas estas secciones se irán explicando con mucho más detalle a lo largo de la memoria.

5.2 PROTOCOLO DE COMUNICACIONES

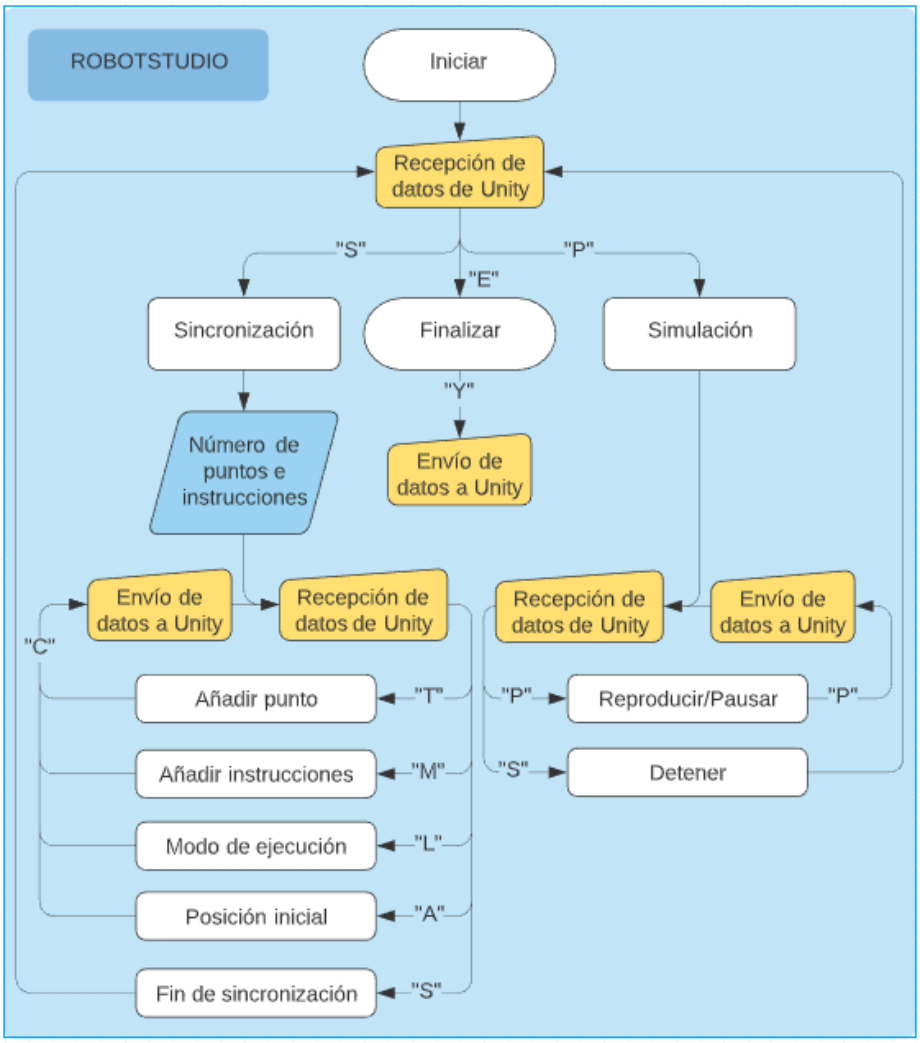

Figura 7 - Protocolo de comunicaciones visto desde RobotStudio.

Se ha desarrollado la *Figura 7* enfocando las comunicaciones desde el punto de vista de *RobotStudio* porque permite entender los bloques del sistema como distintos estados del mismo.

Capítulo 6. ENTORNO DE ROBOTSTUDIO

6.1 CONFIGURACIÓN DE LA SIMULACIÓN

Para ejecutar correctamente el programa desarrollado es necesario configurar una serie de parámetros en *RobotStudio*. A continuación, se encuentran los pasos a seguir*:*

- Una vez iniciado el programa creamos una estación vacía.
- A continuación, añadimos un controlador virtual a la estación. (*Posición inicial* → *Construir estación* → *Controlador virtual* → *Nuevo controlador*)

Esto abrirá una ventana de configuración. Solo habrá que seleccionar la versión del controlador y el tipo de robot que queremos usar. También es necesario marcar la opción de "Personalizar opciones*"*.

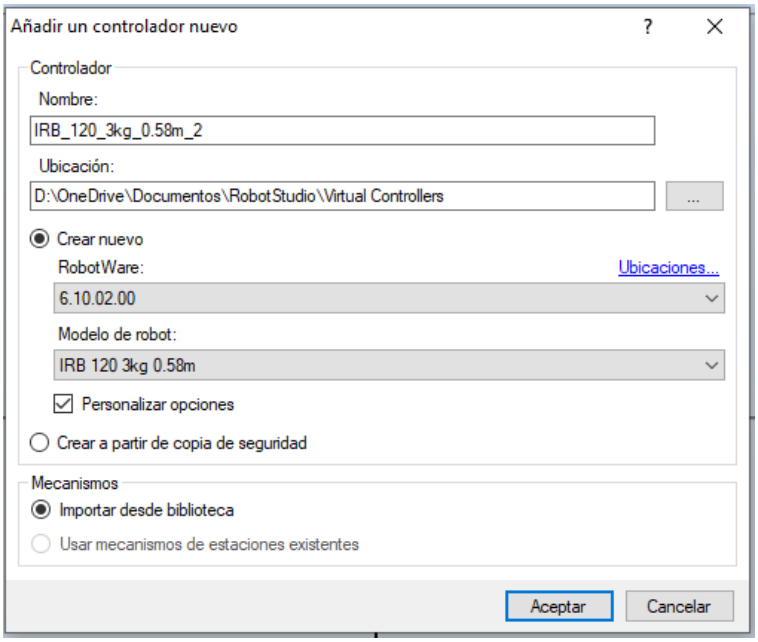

Figura 8 - Configuración de RobotStudio. Añadir el controlador virtual

Al hacerlo se abrirá una ventana de opciones, este paso también se puede hacer una vez creado el controlador virtual (*Controlador → Controlador virtual → Cambiar opciones).*

- Habrá que habilitar las comunicaciones de PC y la opción de multitasking:

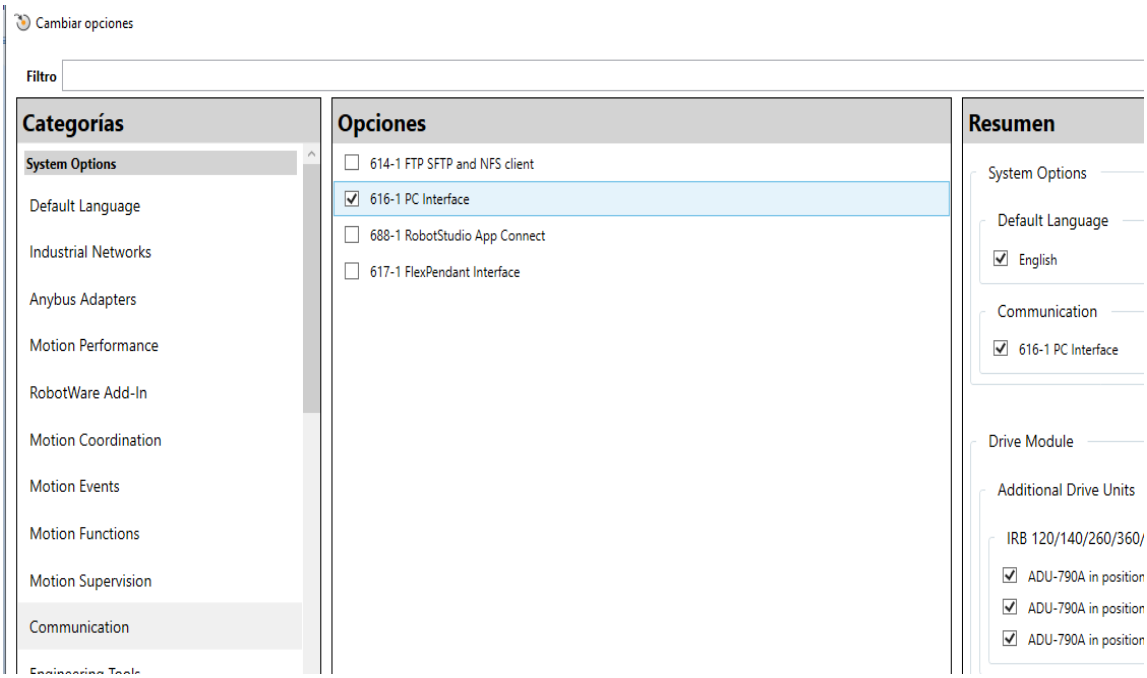

Figura 9 - Configuración de RobotStudio. PC interface

- Primero añadiremos la opción de comunicaciones (*Communication → PC Interface*).

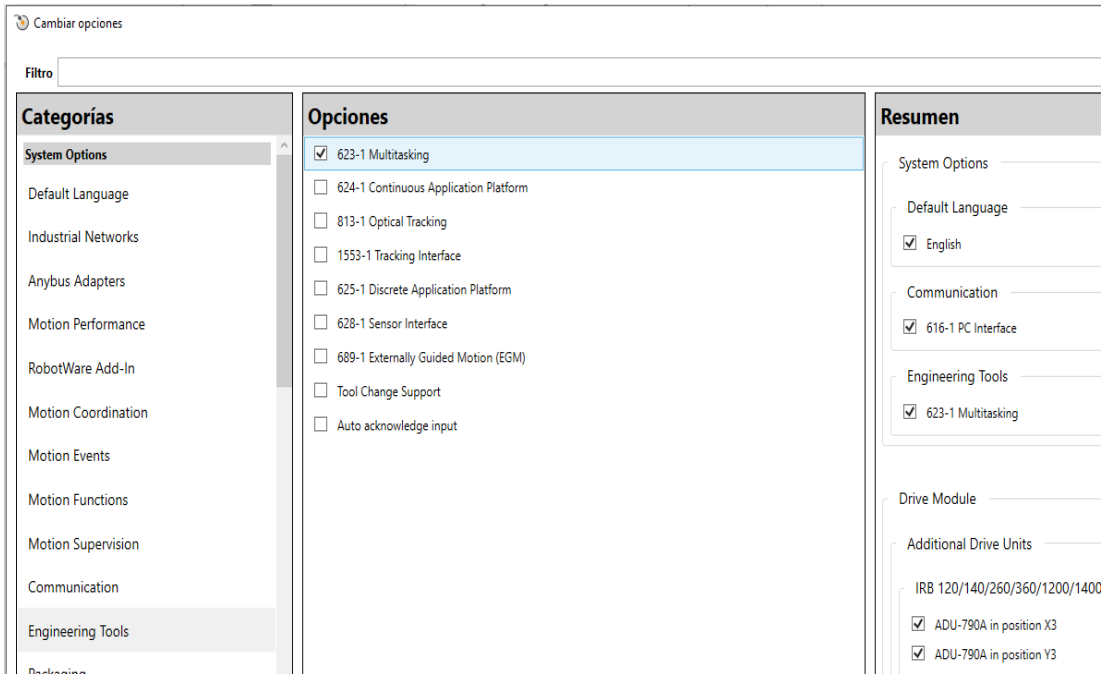

Figura 10 - Configuración de RobotStudio. Multitasking

La opción de multitasking nos permite tener varias tareas en el proyecto.

- A continuación, hay que asegurarse de que el modo de simulación sea de un solo ciclo.

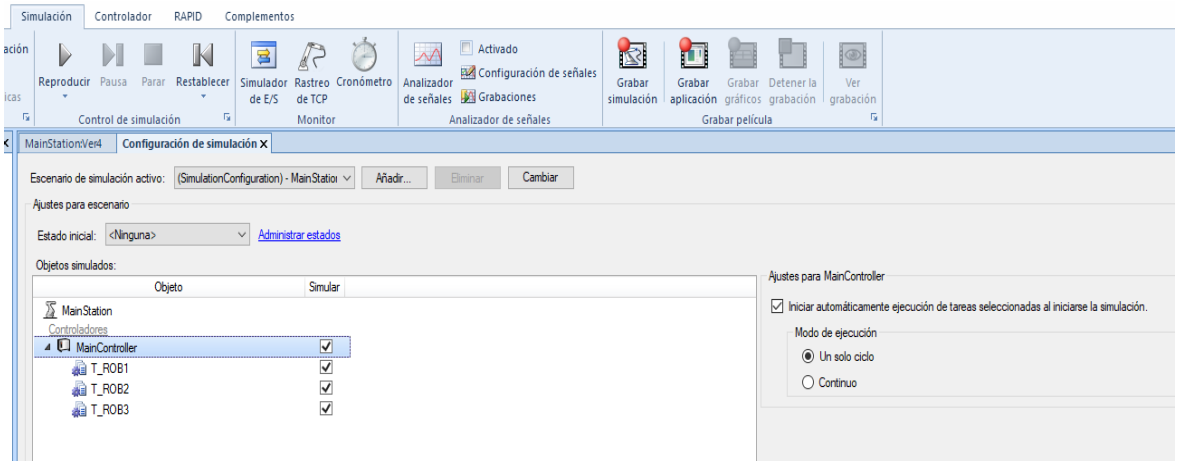

Figura 11 - Configuración de RobotStudio. Modo de ejecución.

- Después pasamos a configurar el código RAPID para que pueda ejecutar los módulos desarrollados en los próximos apartados.

Para acceder a la ventana que se muestra en la *Figura 12:* (*RAPID → Estación actual → MainController → Configuración → Controller → Task*)

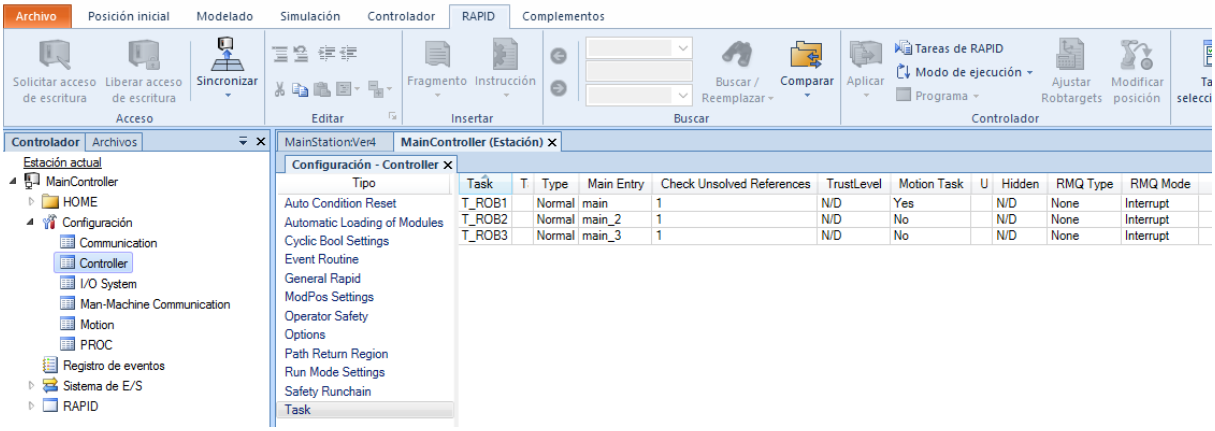

Figura 12 - Configuración de RobotStudio. Configuración de las tareas.

Como se puede ver en la *Figura 12* el proyecto consta de tres tareas diferentes.

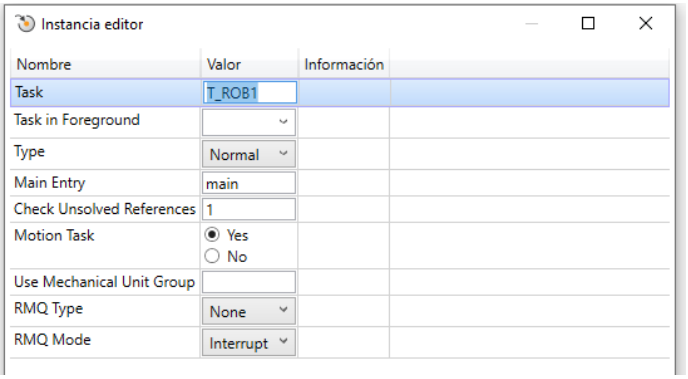

Figura 13 - Configuración de RobotStudio. Configuración de la tarea 1

Solo las tareas que están vinculadas a un robot se pueden configurar como "Motion Task", puesto que las tareas 2 y 3 son de comunicaciones no se configuran como tal.

- Por último, es necesario configurar la carga automática de los módulos para cuando sea necesario reiniciar el código por su actualización.

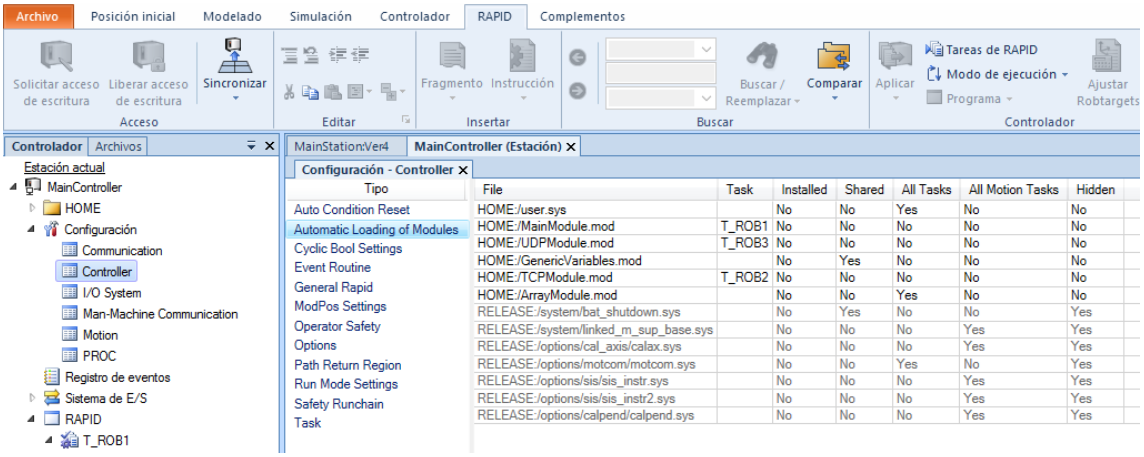

Figura 14 - Configuración de RobotStudio. Automatic Loading of Modules

Esta sección es muy importante porque la configuración de los módulos determinará su funcionalidad en la ejecución:

- La tarea principal se asignará en la tarea 1.
- El módulo de comunicaciones TCP se asigna a la tarea 2
- El módulo de comunicaciones UDP se asigna a la tarea 3
- El módulo de variables genéricas se configura como "Shared" sin asignarse a ninguna tarea. Esto implica que no aparece como modulo cargado en la jerarquía del código RAPID. Se cargará como modulo compartido entre todas las tareas. Esto permite que todas las tareas puedan acceder a las variables definidas en él.
- El módulo de variables matriciales hace referencia a variables que guardan vectores de variables. Este módulo se carga de forma simultánea en todas las tareas. Aquí definimos las variables como *Persistentes* puesto que es la única forma de definir vectores de datos a los que se pueda acceder desde todas las tareas.

6.2 MÓDULOS DE VARIABLES

Por cómo funciona el lenguaje de programación RAPID ha sido necesario incluir las variables en dos tipos de módulos distintos. En el módulo *GenericVariables* se incluyen todas las variables de control y numéricas simples que comparten los módulos principales del sistema.

Mientras, en el módulo *ArrayModule* se incluyen las variables vectoriales. No se almacenan en el mismo porque al configurar el módulo *GenericVariables* como "Shared" no es capaz de tener variables complejas definidas en él.

6.2.1 MÓDULO GENERICVARIABLES

Este módulo consiste únicamente en bloques de variables simples.

```
! VARIABLE PARAMETERS FOR VECTORS
CONST num maxNumOfPoints:=100;
PERS num nPoints;
PERS num pPoints;
! Number of points used

CONST num maxNumOfMoves:=100;
PERS num nMoves;
PERS num pMoves;
! Number of moves (J,L or C) used

VAR num errorNum:= 0;
```

Figura 15 - Módulo GenericVariables. Variables de vectores.

En la *Figura 15* - Módulo GenericVariables. Variables de vectores. aparecen variables que definen el tamaño máximo de los vectores de datos, así como variables auxiliares para poder recorrerlos y almacenar el número real de datos almacenados en ellos. También aparece la variable auxiliar "errorNum" que sirve para almacenar errores producidos durante la ejecución.

```
! RUNTIME VARIABLES
    PERS bool abort:=FALSE;
    ! TRUE if the program must end

    PERS bool actSim:=FALSE;
    ! TRUE if simulation is active

    PERS bool holdSim:=FALSE;
    ! TRUE if the simulation must be paused

    PERS bool actSync:=FALSE;
    ! TRUE if sincronization is active

    PERS bool actLoop:=FALSE;
    ! TRUE if there are loops

    PERS bool setPos:= FALSE;
    ! TRUE if we are going to set the instant position of the robot

    PERS bool canPause:= FALSE;
    ! TRUE if the simulation can be paused
```

Figura 16 - Módulo GenericVariables. Variables de control.

En este último bloque aparecen las variables que regulan el comportamiento del programa.

- La variable “abort” controla si el programa se debe seguir ejecutando.
- La variable “actSim” controla si la simulación esta activa.
- La variable “holdSim” controla si la simulación esta pausada.
- La variable “actSync” controla si la sincronización esta activa.
- La variable “actLoop” informa de si la ejecución debe ser en bucle o no.
- La variable “setPos” se utiliza para iniciar el reposicionamiento del robot durante la sincronización.
- La variable “canPause” controla si la simulación puede pausarse o no.

Se puede estudiar el código con más detenimiento en *GenericVariables*

6.2.2 MÓDULO ARRAYMODULE

Este módulo consta únicamente de variables vectoriales que se utilizan a lo largo de la ejecución del programa.

```
! DATA ARRAYS
    PERS speeddata speed{maxNumOfMoves};
    ! Speed mode for each movement

    PERS zonedata zone{maxNumOfMoves};
    ! Precision mode for each movement

    PERS num moveType{maxNumOfMoves};
    ! Stores the 3 type of movements

    PERS robtarget targets{maxNumOfPoints};
    ! Number of targets

    PERS jointtarget actTargetPos;
```

Figura 17 - ArrayModule. Variables vectoriales

Estos vectores almacenan la velocidad, la precisión, el tipo de movimiento y el objetivo a alcanzar que requieren las instrucciones de desplazamiento para ejecutarse. También hay una variable que almacena la posición inicial del robot.

Se puede analizar el código en detalle aquí: *ArrayModule*

6.3 MÓDULO PRINCIPAL

El módulo principal es el asignado al brazo robótico *ABB IRB120*. Se encarga de ejecutar los comandos de movimiento asignados en el *Módulo de Comunicaciones TCP*.

Se puede analizar el código en detalle aquí: *MainModule*

El módulo tiene definidas al inicio algunas variables para depurar la ejecución.

```
    PERS robtarget tTarget_1:=[[0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0,0,0]];
    PERS robtarget tTarget_2:=[[0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0,0,0]];
    PERS speeddata spd:=[0,0,0,0];
    PERS zonedata zn:=[FALSE,0,0,0,0,0,0];
```

Figura 18 - Variables definidas en el MainModule

Además, cuenta con el proceso main en donde se ejecuta todas sus funciones.

```
PROC main()

    ! With confJ off the robot will go to the nearest possible conf
    ConfJ\Off;
    ConfL\Off;
    CornerPathWarning(FALSE);

    WHILE (NOT abort) DO
        …
    ENDWHILE

    ! Error handling
ERROR
  …
ENDPROC
```

Figura 19 - Esquema general de MainModule

En la *Figura 19* - Esquema general de MainModule se puede ver el esquema general del proceso. Lo primero que se ejecuta son los comando “ConfJ” y “ConfL” con los que se permite al sistema alcanzar puntos con el formato de la variable *Confdata* más próximo al definido. Se hace de esta manera para ahorrar posibles problemas en la ejecución.

Después comienza el bucle de ejecución del programa que se repetirá hasta que cambie el valor de la variable “abort”. Por último, aparece la sección de manejo de errores, en donde se gestionan los posibles errores que se puedan producir en esta sección del código.

La *Figura 20* muestra los dos bloques principales de comportamiento de este módulo.

```
IF setPos THEN
     MoveAbsJ actTargetPos,v500,fine,tool0;
     WaitRob\InPos;
     setPos:=FALSE;
ENDIf

WHILE actSim DO
  …
     WaitRob\ZeroSpeed;
     canPause:=TRUE;
ENDWHILE
```

Figura 20 - MainModule. Funciones principales.

El primer bloque "IF" se usa durante la sincronización para situar el robot. El comando "WaitRob" permite detener el código hasta que el robot alcance la posición.

El segundo bloque corresponde al comportamiento durante la simulación. Se trata de un bucle "while" para permitir detener la simulación si así lo desea el usuario. Dentro del bucle se define la variable "canPause" que permite controlar cuando puede el usuario detener la simulación, que es solo después de que el robot llegue a un punto de velocidad cero. Dentro del bucle "while" controlado por "actSim" se encuentra el código mostrado en la Figura 21 - MainModule. Bloque de simulación

```
        WHILE (NOT holdSim) DO
                    canPause:=FALSE;

                    tTarget_1:=targets{pPoints};
                    IF pPoints<nPoints THEN
                        tTarget_2:=targets{pPoints+1};
                    ENDIF
                    spd:=speed{pMoves};
                    zn:=zone{pMoves};

                    TEST moveType{pMoves}

                    CASE 0:
                        MoveL tTarget_1,spd,zn,tool0;
                        pPoints:=pPoints+1;
                        pMoves:=pMoves+1;

                    CASE 1:
                        MoveJ tTarget_1,spd,zn,tool0;
                        pPoints:=pPoints+1;
                        pMoves:=pMoves+1;

                    CASE 2:
                        MoveC tTarget_1,tTarget_2,spd,zn,tool0;
                        pPoints:=pPoints+2;
                        pMoves:=pMoves+1;

                    DEFAULT:

                    ENDTEST

                    IF ((actLoop) AND (pMoves>nMoves)) THEN
                        pMoves:=1;
                        pPoints:=1;
                    ELSEIF (pMoves>nMoves) THEN
                        holdSim:=TRUE;
                        actSim:=FALSE;
```

```
                    ENDIF

                ENDWHILE
```

Figura 21 - MainModule. Bloque de simulación

Este bloque define el comportamiento disponible durante la simulación.

Con cada ciclo se actualiza los datos de la instrucción a realizar teniendo en cuenta el número máximo de puntos que hay definidos. Después se realiza un "TEST", un proceso similar al "switch case" de Csharp para realizar un tipo de movimiento u otro. Antes de finalizar el bucle se realiza una comprobación de si se ha llegado al número máximo de puntos definidos y dependiendo de si se ha configurado la simulación para ser en bucle o no poder reiniciar las variables que sirven para recorrer los vectores de datos.

```
ERROR
        IF ERRNO=ERR_ROBLIMIT THEN
            SkipWarn;
            errorNum:=1;
            holdSim:=TRUE;
            actSim:=FALSE;
            TRYNEXT;

        ELSEIF ERRNO=ERR_OUTSIDE_REACH THEN
            SkipWarn;
            errorNum:=2;
            holdSim:=TRUE;
            actSim:=FALSE;
            TRYNEXT;
        ENDIF
```

Figura 22 - MainModule. Gestión de errores

En la sección MainModule se pueden producir dos errores diferentes. Ambos están explicados en el *ANEXO III: Lista de errores.*

El primero es el *Error 001: Limite del robot*

El segundo es el *Error 002: Robot fuera de rango*

Cuando se producen la simulación se detiene de forma controlada para no detener la ejecución del código RAPID. Para esto se usan los comandos "SkipWarn" y "TRYNEXT" que permiten ignorar la advertencia que detiene la ejecución.

6.4 MÓDULO DE COMUNICACIONES TCP

Este módulo es el cerebro de todo el bloque de *RobotStudio*, se encarga de manipular todas las variables de control según el mensaje recibido.

Se puede analizar el código en detalle aquí: *TCPModule*

El primer bloque de código está compuesto por las variables que solo se emplean en este módulo.

```
! Local Communication Variables
VAR string key;
VAR string k;

VAR socketdev serverTCP;
VAR socketdev clientTCP;

CONST num portTCP:=5000;
CONST string ip:="127.0.0.1";

PERS string msgTCP;

! Variables to take specific numbers from the msg str
PERS num tempV1;
PERS num tempV2;
PERS bool ok;
PERS string tString;
```

Figura 23 - TCPModule. Variables.

La mayor parte de las variables definidas en la *Figura 23* sirven para gestionar las comunicaciones y por tanto son temporales.

- Las variables “key” y “k” sirven para obtener el dato que se usará para elegir entre un proceso u otro
- Las variables socket corresponden a variables necesarias para la creación de las comunicaciones
- “portTCP” e “ip” son el puerto y la dirección de conexión que se empleará para las comunicaciones TCP.
- “msgTCP” es la variable donde se almacenará el mensaje recibido.

A continuación, comienza el proceso main y se inicializan las variables de comportamiento.

```
PROC main_2()
        abort:=FALSE;
        ! TRUE if the program must end

        actSim:=FALSE;
        ! TRUE if simulation is active

        holdSim:=FALSE;
        ! TRUE if the simulation must be paused

        actSync:=FALSE;
        ! TRUE if sincronization is active

        actLoop:=FALSE;
        ! TRUE if there are loops

        setPos:=FALSE;
```

Figura 24 - TCPModule. Inicialización de variables

En la siguiente sección se inicializan las comunicaciones

```
! SERVER INITIALIZATION
        ! Create server socket and bind it to ip address and port
        SocketCreate serverTCP;
        SocketBind serverTCP,ip,portTCP;
        SocketListen serverTCP;

        ! Accept incoming connections from server socket into client socket
        SocketAccept serverTCP,clientTCP,\Time:=WAIT_MAX;
```

Figura 25 - TCPModule. Inicio del servidor.

En RAPID, para iniciar las comunicaciones TCP es necesario el uso de varios comandos. Primero se crea la variable y después se vincula a un puerto.

Después comienza el bloque de ejecución del programa, el cual está dentro de un bucle "while" controlado por la variable "abort".

```
        WHILE (NOT abort) DO
            msgTCP:=ReceiveMsgTCP();

            IF (StrLen(msgTCP)>0) THEN
                k:=StrPart(msgTCP,1,1);

                TEST k
                    CASE "S": ! Synchronization
                              …
                    CASE "P": ! Simulation
                              …
                    CASE "E": ! End Program
                              …
                    CASE "N": ! Nothing received
                              …
                    DEFAULT:
                    ! Wrong option
                     SocketSend clientTCP\str:="ERROR 007: Wrong generic option:"
+ k;
                ENDTEST
            ENDIF


            TEST errorNum

            CASE 1:
                SocketSend clientTCP\str:="ERROR 001: Robot limit. The position
is reachable, but outside joint/s limits.";
                errorNum:=0;

            CASE 2:
                SocketSend clientTCP\str:="ERROR 002: Robot out of reach. The
position is outside the working range.";
                errorNum:=0;

            ENDTEST

        ENDWHILE
```

Figura 26 - TCPModule. Bucle principal.

Al entrar en el bucle se llama al comando *ReceiveMsgTCP*, se comprueba el mensaje recibido y se usa la primera letra del mensaje para saber qué ejecutar a continuación.

Actualmente las opciones configuradas son:

- "S" para iniciar la sincronización
- "P" para iniciar la simulación
- "E" para finalizar el programa

- "N" si no se ha recibido ningún mensaje

Después hay un pequeño módulo de gestión de errores para los que se pueden producir en *Módulo Principal*. Con ayuda del número de error que se le asoció a la variable "errorNum" se mandan sendos mensajes de error a *Unity*.

6.4.1 SINCRONIZACIÓN

```
CASE "S": ! Synchronization
! Checks that the number of points and moves are less than the limit  S;num;num;
S;0;0;
                    tempV1:=StrFind(msgTCP,1,";");
                    tempV2:=StrFind(msgTCP,tempV1+1,";");
                  ok:=StrToVal(StrPart(msgTCP,tempV1+1,tempV2-tempV1-1),nPoints);

                    tempV1:=tempV2;
                    tempV2:=StrFind(msgTCP,tempV1+1,";");
                  ok:=StrToVal(StrPart(msgTCP,tempV1+1,tempV2-tempV1-1),nMoves);

                    pPoints:=1;
                    pMoves:=1;
                    ! Pointers to go through the vectors

                    IF nPoints<=maxNumOfPoints AND nMoves<=maxNumOfMoves THEN
                        ! Synchronize the robtargets and/or paths
                        actSync:=TRUE;

                        socketSend clientTCP\str:="R";
                        ! Ready to synchronize
                    ELSE
                        actSync:=FALSE;
                        socketSend clientTCP\str:="ERROR 003: Too many
points/moves. (max:" + numToStr(maxNumOfPoints,2) + "/" +
numToStr(maxNumofMoves,2) + ")";
                    ENDIF

                    WHILE actSync DO
                        msgTCP:=ReceiveMsgTCP();
                        IF (StrLen(msgTCP)>0) THEN
                            key:=StrPart(msgTCP,1,1);

                            TEST key

                            CASE "T": ! Robtargets
                            CASE "M": ! Movements
                            CASE "L": ! Loop
                            CASE "A": ! Initial Position
                            CASE "S": ! End
                            CASE "N": ! Nothing received
                            DEFAULT:
                                SocketSend clientTCP\str:="ERROR 005: Wrong
synchronization choice";

                            ENDTEST
                        ENDIF
                    ENDWHILE
```

Figura 27 - TCPModule. Sincronización

En caso de haberse seleccionado la sincronización, el mensaje completo recibido sería parecido a esto: S;num;num; donde el primer y segundo número hacen referencia al número de puntos e instrucciones programados. El primer bloque de la *Figura 27* corresponde a la asignación de estos valores a las variables correspondientes. A continuación, se confirma que estos valores son menores que el máximo permitido. En caso afirmativo se manda confirmación a *Unity* para continuar con la sincronización, en caso erróneo se comunica un error y se finaliza la sincronización.

Posteriormente se entra en un bucle para sincronizar puntos e instrucciones:

6.4.1.1 Sincronización de Robtargets

En caso de haber recibido una "T" al inicio del bucle, se ejecuta el código de la *Figura 28*

```
! Adds robtargets in order. T[num;num;num; num;num;num; num;num;num;num;]
                                tempV1:=StrFind(msgTCP,1,"[");
                                tempV2:=StrFind(msgTCP,1,"]");

targets{pPoints}:=StrToTarget(StrPart(msgTCP,tempV1+1,tempV2-tempV1-1));
                                pPoints:=pPoints+1;
                                SocketSend clientTCP\str:="C";
```

Figura 28 - TCPModule. Sincronización de Robtargets.

El comando más relevante en esta sección corresponde a *Método* StrToTarget. Convierte la cadena de caracteres en datos, lo almacena y responde con un mensaje confirmando recepción.

6.4.1.2 Sincronización de Instrucciones

En caso de haber recibido una "M" al inicio del bucle, se ejecuta el código de la *Figura 29*

```
! Adds movement type, speeddata and precision M[letter;num;num;]
                                tempV1:=StrFind(msgTCP,1,"[");
                                tempV2:=StrFind(msgTCP,1,"]");

                        ok:=StrToInstr(StrPart(msgTCP,tempV1+1,tempV2-tempV1-1));

                                pMoves:=pMoves+1;
                                SocketSend clientTCP\str:="C";
```

Figura 29 - TCPModule. Sincronización de instrucciones.

El comando más relevante en esta sección corresponde a *Método* StrToInstr.

6.4.1.3 Sincronización de bucle de ejecución

```
        ! Tells if the program is in loop
        actLoop:=NOT actLoop;
        SocketSend clientTCP\str:="C";
```

Figura 30 - TCPModule. Sincronización de bucle.

En caso de haber recibido una "L" se actualiza el estado de la variable "actLoop" que confirma si las instrucciones deben ejecutarse en bucle.

6.4.1.4 Sincronización de posición inicial

```
! Sets the initial position of the robot A[num,num,num,num,num,num]
   tempV1:=StrFind(msgTCP,1,"[");
   tempV2:=StrFind(msgTCP,1,"]");

! This section should be modified to escalate depending on the number of axis of
each robot
   tString:="["+StrPart(msgTCP,tempV1,tempV2-
tempV1+1)+",[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]]";

   ok:=StrToVal(tString,actTargetPos);

   IF ok THEN
         setPos:=TRUE;
         WaitUntil(NOT setPos),\MaxTime:=WAIT_MAX;
         SocketSend clientTCP\str:="C";
   ELSE
         SocketSend clientTCP\str:="ERROR 004: Couldn't set the initial position
of the robot";
   ENDIF
```

Figura 31 - TCPModule. Sincronización de posición inicial.

En caso de recibir una "A" se sincronizará la posición del robot. Como este proceso no es instantáneo se detiene la ejecución del programa hasta confirmarse que el robot está en posición. En caso de no ser posible se enviará un mensaje de error.

6.4.1.5 Finalización de la sincronización

```
! Ends the sychronization
   actSync:=FALSE;
```

Figura 32 - TCPModule. Finalización de la sincronización.

Este sencillo bloque únicamente actualiza la variable de la simulación cuando se reciba una "S".

6.4.2 SIMULACIÓN

```
            CASE "P": ! Simulation
                  pPoints:=1;
                  pMoves:=1;
                  holdSim:=FALSE;
                  actSim:=TRUE;
                  SocketSend clientTCP\str:="S";

                  WHILE actSim DO
                      msgTCP:=ReceiveMsgTCP();
                      IF StrLen(msgTCP)>0 THEN
                          key:=StrPart(msgTCP,1,1);

                          TEST key

                          CASE "P":
                              !Pauses/replay the simulation
                              holdSim:=NOT holdSim;
                              WaitUntil(canPause),\MaxTime:=WAIT_MAX;
                              SocketSend clientTCP\str:="P";

                          CASE "S":
                              !Stops de simulation
                              holdSim:=TRUE;
                              WaitUntil(canPause),\MaxTime:=WAIT_MAX;
                              actSim:=FALSE;

                          CASE "N":
                              ! Nothing received

                          DEFAULT:
                              SocketSend clientTCP\str:=("ERROR 006: Wrong
simulation choice: " + key);

                          ENDTEST
                      ENDIF
                  ENDWHILE

                  ! Message to confirm the end of the simulation
                  SocketSend clientTCP\str:="E";
```

Figura 33 - TCPModule. Simulación.

Su comportamiento es parecido al bloque de sincronización. Tiene una sección inicial donde se inicializan las variables para recorrer los vectores de simulación y las variables de comportamiento justo antes de confirmar por mensaje que la simulación ha comenzado.

Después comienza el bucle de simulación en el que existen dos posibles opciones de acción:

- En caso de recibir una "P" se pausa la simulación si se estaba ejecutando y viceversa. Se envía confirmación por mensaje cuando el robot se ha detenido.
- En caso de recibir una "S" se detiene la simulación. Se sale del bucle y en caso de volver a entrar se comienza la ejecución de la simulación de nuevo.

6.4.3 FINALIZACIÓN DE LA SIMULACIÓN

```
            ! Ends the program
                holdSim:=TRUE;
                actSim:=FALSE;
                abort:=TRUE;
```

Figura 34 - TCPModule. Finalización de la simulación.

En este pequeño bloque se actualiza el valor de las variables de comportamiento para detener la ejecución del programa.

6.4.4 FINAL DE LA RUTINA PRINCIPAL

```
        SocketSend clientTCP\str:="Y";
        SocketClose serverTCP;
        SocketClose clientTCP;
        stop;
```

Figura 35 - TCPModule. Final de la rutina principal.

Al cerrarse el bucle principal el código confirma la finalización del programa por mensaje antes de cerrar los puertos de comunicación.

6.4.5 MÉTODOS SECUNDARIOS

6.4.5.1 Método ReceiveMsgTCP

```
FUNC string ReceiveMsgTCP()
        VAR string tMsg;
        VAR rawbytes rawData;

        ! Receive robotTarget from Unity in string format
        SocketReceive clientTCP,\RawData:=rawData\Time:=2;
        UnpackRawBytes rawData,1,tMsg,\ASCII:=RawBytesLen(rawData);

        RETURN tMsg;

        ! Error handler
    ERROR
        IF ERRNO=ERR_FNCNORET THEN
            skipWarn;
            tMsg:="N";
            RETURN tMsg;

        ELSEIF ERRNO=ERR_SOCK_TIMEOUT THEN
            SkipWarn;
            TRYNEXT;

        ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
            SkipWarn;
            holdSim:=TRUE;
            actSim:=FALSE;
            abort:=TRUE;
            TRYNEXT;
        ENDIF
    ENDFUNC
```

Figura 36 - TCPModule. Método ReceiveMsgTCP.

Esta función trata de recibir un mensaje durante dos segundos cada vez que es llamada. La información se recibe en forma de bytes brutos y hace falta convertirlos a caracteres.

Puesto que la función tiene que devolver algún mensaje, se ha desarrollado una pequeña sección de gestión de errores. Esto corresponde al primer error gestionado, "ERR_FNCNORET". Se le asigna un valor a la variable correspondiente y se sale de la función con un "RETURN".

El resto de los errores son convencionales: Que se haya superado el tiempo de espera y que se haya producido un cese de la conexión de las comunicaciones de forma abrupta.

6.4.5.2 Método JointTargetToString

```
FUNC string jointTargetToString(jointtarget target)
        VAR string tempString;
        tempString:=NumToStr(target.robax.rax_1,0);
        ConcatenateString tempString,target.robax.rax_2;
        ConcatenateString tempString,target.robax.rax_3;
        ConcatenateString tempString,target.robax.rax_4;
        ConcatenateString tempString,target.robax.rax_5;
        ConcatenateString tempString,target.robax.rax_6;
        RETURN tempString;
    ENDFUNC
```

Figura 37 - TCPModule. Método JointTargetToString.

Esta función se encarga de convertir una variable del tipo *Jointtarget* en una cadena de caracteres. Esta función también está presente en el *Módulo de Comunicaciones UDP*

6.4.5.3 Método ConcatenateString

```
PROC ConcatenateString(INOUT string inString,num value)
        inString:=inString+";"+NumToStr(value,0);
    ENDPROC
```

Figura 38 - TCPModule. Método ConcatenateString.

Esta función se encarga de concatenar diferentes variables numéricas en una única cadena de caracteres.

6.4.5.4 Método GetMove

```
! Function that returns moveType
FUNC num GetMove(string data)
    VAR num type;

    TEST data
    CASE "L":
        type:=0;

    CASE "J":
        type:=1;

    CASE "C":
        type:=2;
    DEFAULT:

    ENDTEST

    RETURN type;
ENDFUNC
```

Figura 39 - TCPModule. Método GetMove.

Función que engloba el típico “switch case” de C para obtener el tipo de movimiento de una cadena de caracteres.

6.4.5.5 Método GetSpeed

```
! Function that returns speeddata variable type for movement commands
    FUNC speeddata GetSpeed(string data_3)
        VAR speeddata tempSpeed;
        TEST data_3
        CASE "0":
            tempSpeed:=v5;
        CASE "1":
            tempSpeed:=v10;
        CASE "2":
            tempSpeed:=v20;
        CASE "3":
            tempSpeed:=v30;
        CASE "4":
            tempSpeed:=v40;
        CASE "5":
            tempSpeed:=v50;
        CASE "6":
            tempSpeed:=v60;
        CASE "7":
            tempSpeed:=v80;
        CASE "8":
            tempSpeed:=v100;
        CASE "9":
            tempSpeed:=v150;
        CASE "10":
            tempSpeed:=v200;
        CASE "11":
            tempSpeed:=v300;
        CASE "12":
            tempSpeed:=v400;
        CASE "13":
            tempSpeed:=v500;
        CASE "14":
            tempSpeed:=v600;
        CASE "15":
            tempSpeed:=v800;
        CASE "16":
            tempSpeed:=v1000;
        CASE "17":
            tempSpeed:=v1500;
        CASE "18":
            tempSpeed:=v2000;
        CASE "19":
            tempSpeed:=v2500;

        DEFAULT:
        ENDTEST
        RETURN tempSpeed;
    ENDFUNC
```

Figura 40 - TCPModule. Método GetSpeed.

Función que engloba el típico “switch case” de C para obtener el nivel de velocidad de una cadena de caracteres.

6.4.5.6 Método GetPrecision

```
! Function that returns zonedata variable type for movement commands
    FUNC zonedata GetPrecision(string data_2)
        VAR zonedata tempPrec;

        TEST data_2
        CASE "0":
            tempPrec:=fine;
        CASE "1":
            tempPrec:=z0;
        CASE "2":
            tempPrec:=z1;
        CASE "3":
            tempPrec:=z5;
        CASE "4":
            tempPrec:=z10;
        CASE "5":
            tempPrec:=z15;
        CASE "6":
            tempPrec:=z20;
        CASE "7":
            tempPrec:=z30;
        CASE "8":
            tempPrec:=z40;
        CASE "9":
            tempPrec:=z50;
        CASE "10":
            tempPrec:=z60;
        CASE "11":
            tempPrec:=z80;
        CASE "12":
            tempPrec:=z100;
        CASE "13":
            tempPrec:=z150;
        CASE "14":
            tempPrec:=z200;

        DEFAULT:
        ENDTEST
        RETURN tempPrec;
    ENDFUNC
```

Figura 41 - TCPModule. Método GetPrecision.

Función que engloba el típico “switch case” de C para obtener el nivel de precisión de una cadena de caracteres.

6.4.5.7 Método StrToInstr

```
FUNC bool StrToInstr(string data_1)
        !M[letter;num;num;] M[L;13;9;]
        VAR num pMove;
        VAR num pSpeed;
        VAR num pPrec;

        ! Position index
        pMove:=StrFind(data_1,1,";");
        pSpeed:=StrFind(data_1,pMove+1,";");
        pPrec:=StrFind(data_1,pSpeed+1,";");

        moveType{pMoves}:=GetMove(StrPart(data_1,1,pMove-1));
        speed{pMoves}:=GetSpeed(StrPart(data_1,pMove+1,pSpeed-pMove-1));
        zone{pMoves}:=GetPrecision(StrPart(data_1,pSpeed+1,pPrec-pSpeed-1));
        ok:= TRUE
        RETURN ok;
    ENDFUNC
```

Figura 42 - TCPModule. Método StrToInstr.

Esta instrucción es capaz de convertir el mensaje recibido que codifica una instrucción en datos almacenados en los vectores correspondientes en el orden correcto devolviendo un "TRUE" si se ha realizado con éxito. En realidad, hacer que esta función devuelva confirmación solo es una formalidad para que pueda usarse como función, ya que RAPID no permite la creación de funciones que no devuelvan algún tipo de variable.

6.4.5.8 Método StrToTarget

```
FUNC robtarget StrToTarget(string data)
        VAR robtarget tempTarget;
        VAR bool bResult;

        VAR num pX;
        VAR num pY;
        VAR num pZ;

        VAR num tempRX;
        VAR num pRX;
        VAR num tempRY;
        VAR num pRY;
        VAR num tempRZ;
        VAR num pRZ;

        VAR num pcf1;
        VAR num pcf4;
        VAR num pcf6;
        VAR num pcfx;
```

```
        ! Position index
        pX:=StrFind(data,1,";");
        pY:=StrFind(data,pX+1,";");
        pZ:=StrFind(data,pY+1,";");

        pRX:=StrFind(data,pZ+1,";");
        pRY:=StrFind(data,pRX+1,";");
        pRZ:=StrFind(data,pRY+1,";");

        pcf1:=StrFind(data,pRZ+1,";");
        pcf4:=StrFind(data,pcf1+1,";");
        pcf6:=StrFind(data,pcf4+1,";");
        pcfx:=StrFind(data,pcf6+1,";");

        ! Position data
        bResult:=StrToVal(StrPart(data,1,pX-1),tempTarget.trans.x);
        bResult:=StrToVal(StrPart(data,pX+1,pY-pX-1),tempTarget.trans.y);
        bResult:=StrToVal(StrPart(data,pY+1,pZ-pY-1),tempTarget.trans.z);

        ! Orientation data
        bResult:=StrToVal(StrPart(data,pZ+1,pRX-pZ-1),tempRX);
        bResult:=StrToVal(StrPart(data,pRX+1,pRY-pRX-1),tempRY);
        bResult:=StrToVal(StrPart(data,pRY+1,pRZ-pRY-1),tempRZ);
        tempTarget.rot:=OrientZYX(tempRZ,tempRY,tempRX);

        ! Configuration data
        bResult:=StrToVal(StrPart(data,pRZ+1,pcf1-pRZ-1),tempTarget.robconf.cf1);
        bResult:=StrToVal(StrPart(data,pcf1+1,pcf4-pcf1-
1),tempTarget.robconf.cf4);
        bResult:=StrToVal(StrPart(data,pcf4+1,pcf6-pcf4-
1),tempTarget.robconf.cf6);
        bResult:=StrToVal(StrPart(data,pcf6+1,pcfx-pcf6-
1),tempTarget.robconf.cfx);

        ! External axii data
        tempTarget.extax.eax_a:=9E+09;
        tempTarget.extax.eax_b:=9E+09;
        tempTarget.extax.eax_c:=9E+09;
        tempTarget.extax.eax_d:=9E+09;
        tempTarget.extax.eax_e:=9E+09;
        tempTarget.extax.eax_f:=9E+09;

        RETURN tempTarget;
    ENDFUNC
```

Figura 43 - TCPModule. Método StrToTarget.

Esta función se encarga de convertir una cadena de caracteres es un variable del tipo *Robtarget* y devolverla. Para esto se usan variables auxiliares que almacenan las posiciones de los datos y posteriormente se almacenan por segmentos en una única variable temporal.

La codificación del mensaje es muy sencilla:

T[num;num;num; num;num;num; num;num;num;num;]

Básicamente consta de números seguidos de ";". De esta manera podemos buscar el símbolo que separa los números y al conocer el orden de éstos, guardarlos de forma correcta.

6.5 MÓDULO DE COMUNICACIONES UDP

Este módulo se encarga de enviar la posición de los ángulos de los ejes del robot a *Unity*.

Se puede analizar el código en detalle aquí: *UDPModule*

Este módulo usa dos funciones que ya se han explicado: *Método JointTargetToString* y *Método ConcatenateString*

```
! Temporal variables for sending data
    PERS jointtarget jointangles:=[[0,0,0,0,0,0],[0,0,0,0,0,0]];

    ! Local Communication Variables
    VAR socketdev serverUDP;

    CONST num portUDP:=6000;
    CONST string ip:="127.0.0.1";

    PROC main_3()
        SocketCreate serverUDP\UDP;

        WHILE (NOT abort) DO
            IF actSim THEN
                jointangles:=CJointT();
                SocketSendTo
serverUDP,ip,portUDP\Str:=jointTargetToString(jointangles);
            ENDIF

            WaitTime 0.005;
        ENDWHILE
        SocketClose serverUDP;
    ENDPROC
```

Figura 44 - UDPModule.

Como se muestra en la *Figura 44* el módulo cuenta con una primera sección donde se definen variables locales de comunicación y de almacenamiento temporal de datos.

En la función main se crea el socket de comunicación UDP y posteriormente se entra en un bucle que se ejecuta mientras el programa este activo. Éste enviará por el puerto de comunicaciones los ángulos del robot mientras que la simulación este activa esperando 0.005 segundos entre cada mensaje.

Se deja un margen de tiempo porque el programa de *Unity* no procesa la información recibida de forma instantánea y el exceso de información simplemente se perdería por usar el *Protocolo UDP/IP*.

Capítulo 7. ENTORNO DE UNITY

Todo el código desarrollado en las próximas secciones se encuentra aquí: *ANEXO IV: Código Fuente de Unity.*

7.1 CONFIGURACIÓN DE UNITY

Antes de usar el entorno de Unity hay que configurar una serie de parámetros, desde la iluminación hasta los accesos de teclado que se usaran durante la ejecución del programa.

7.1.1 PAQUETES Y ASSETS IMPLEMENTADOS

El único asset externo implementado es gratuito y se puede descargar desde la "Asset Store" de *Unity*. Se llama "Skybox" y se ha usado para configurar el fondo de la escena.

Los paquetes instalados son:

- Post Processing v2.3.0
- ProBuilder v4.2.3
- ProGrids v3.0.3

Mientras que el paquete de post procesado se ha añadido para mejorar la parte visual de la escena, los paquetes ProBuilder y ProGrids se han empleado por agilizar la construcción del escenario sin perjudicar al rendimiento. Son de uso extendido entre los usuarios de *Unity*.

A diferencia de los assets, para añadir paquetes de datos hace falta acceder a la ventana de Package Manager. *(Window → Package Manager).*

El resto de los paquetes aparecen en la *Figura 45*

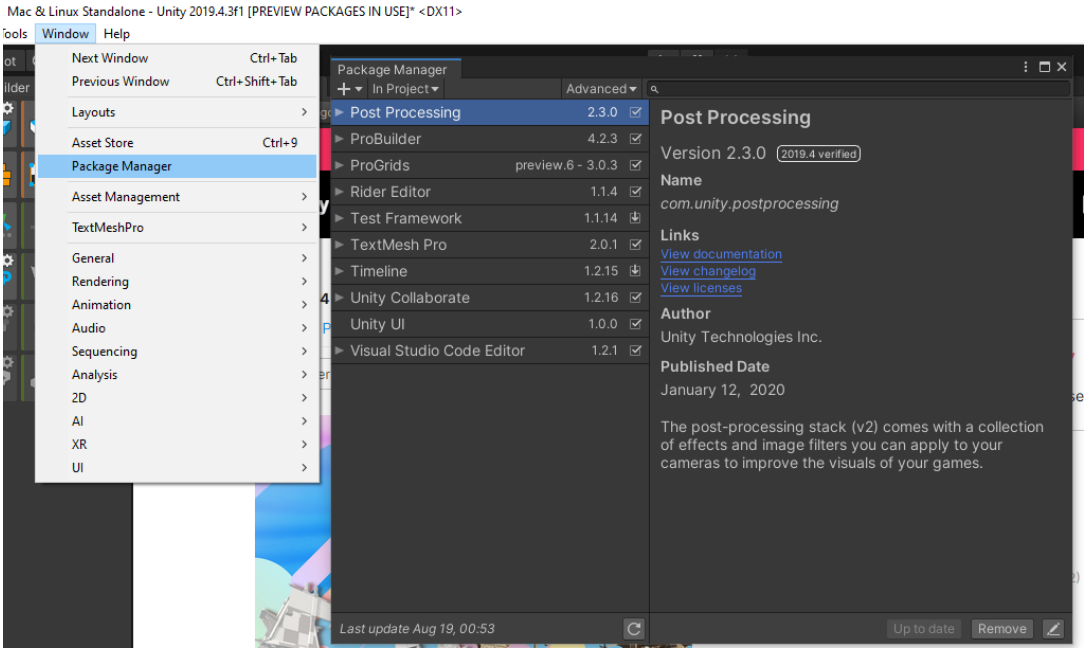

Figura 45 - Package Manager.

7.1.2 CONFIGURACIÓN DE ILUMINACIÓN

La configuración de la iluminación es sencilla. Para acceder a ella hace falta seguir los siguientes pasos: *(Window → Rendering → Lightning Settings).*

Para configurar el cielo basta con arrastrar el material deseado a la ventana de "Skybox material".

También es necesario añadir una fuente de luz, pero ésta ya viene por defecto.

Para terminar de configurarlo basta con darle a "auto-generate" en la parte inferior. Es la forma más rápida y sencilla de configurarlo y en general hace un buen trabajo compensando calidad y rendimiento.

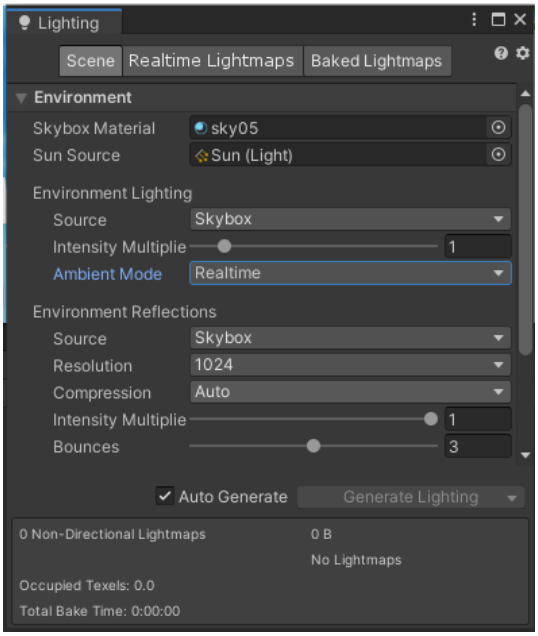

Figura 46 - Lighting Settings.

7.1.3 CONFIGURACIÓN DE INPUT MANAGER

En la versión que se ha utilizado para el programa se configuran diferentes bloques para los distintos accesos por teclado. Para acceder a la ventana correspondiente hay que seguir los siguientes pasos: *(Edit → Project Settings → Input Manager).*

Además de los configurados de serie se han agregado estos "ejes":

- X
- Y
- Z

Originalmente existen "Horizontal" y "Vertical" que sustituyen a X y Z, pero para aclarar el código se ha decidido añadir unos personalizados. No se puede modificar los existentes porque existen otros subsistemas internos que dependen de ellos.

En la *Figura 47* se muestra un ejemplo de cómo se configuran estos "ejes". Lo único relevante es que donde aparece "Axis" no hay que hacer nada, solo es relevante en la configuración de componentes como el ratón.

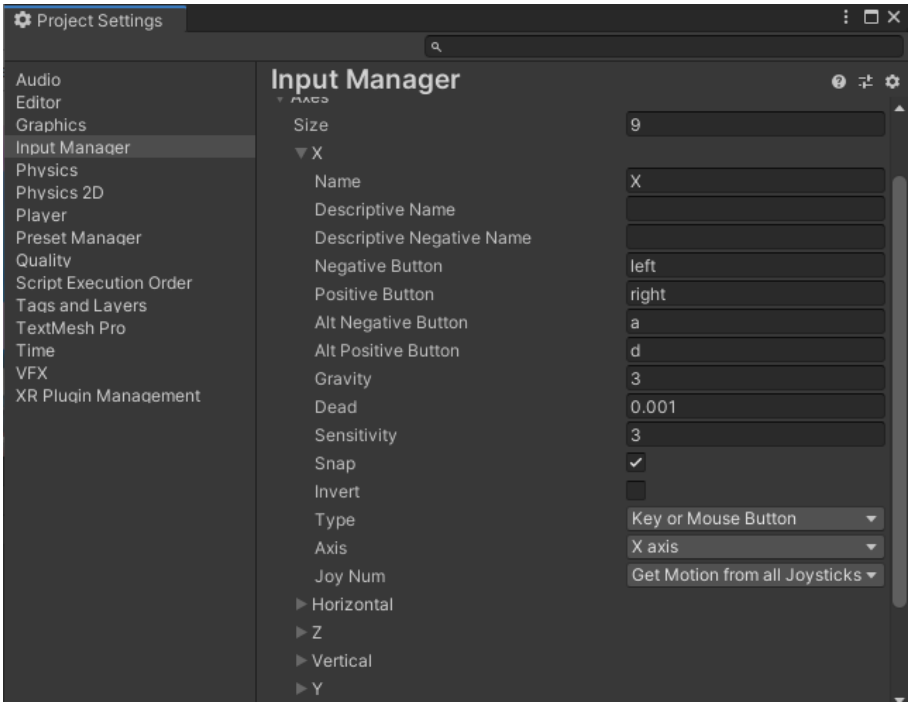

Figura 47 - Input Manager.

7.1.4 ZONA DE TRABAJO

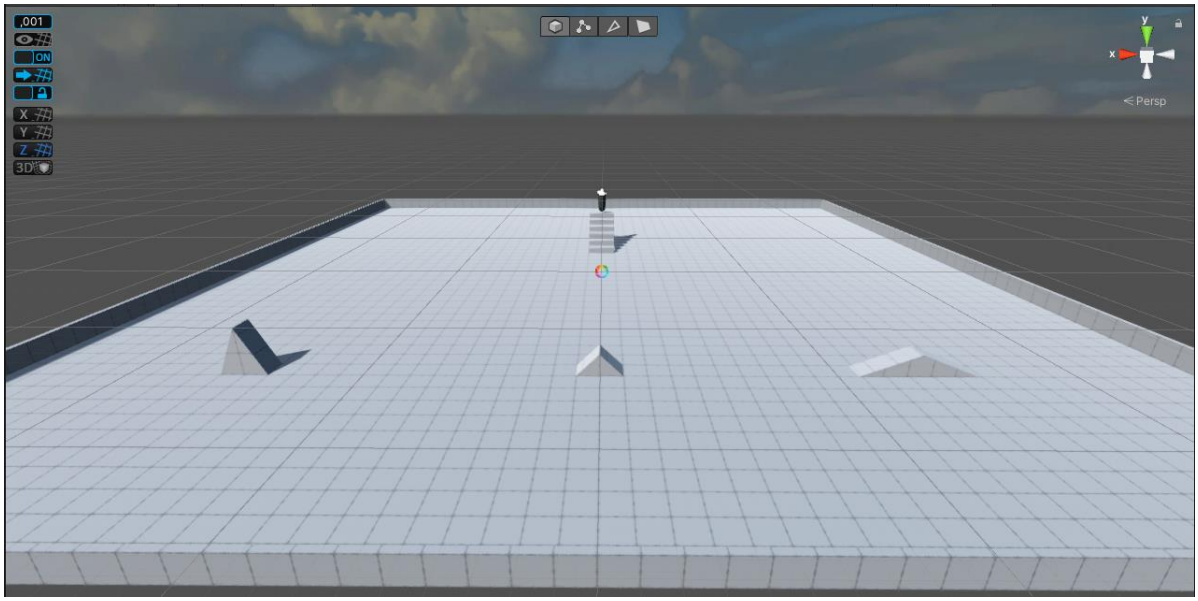

Figura 48 - Zona de trabajo.

En la *Figura 48* se muestra la zona de trabajo diseñada. La jerarquía de los *GameObject* que lo componen se muestra en la *Figura 49*.

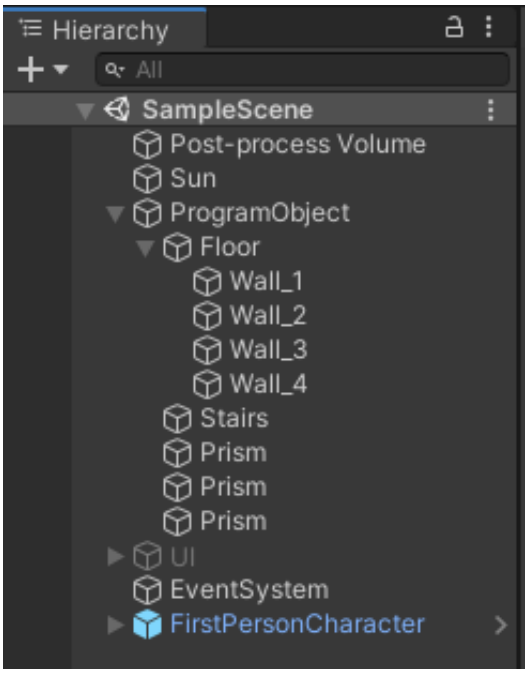

Figura 49 - Jerarquía de la zona de trabajo.

Los elementos que conforman la escena se han añadido como hijos del objeto "ProgramObject" y han sido creados con el paquete ProBuilder. Este es el objeto principal puesto que contiene los componentes principales del programa. En la *Figura 50* se muestran estos componentes.

El *GameObject* llamado "Floor" tiene vinculados los objetos de las paredes. Tanto las escaleras como los prismas han sido añadidos para mostrar la configuración de movimiento que se le ha añadido al objeto del usuario.

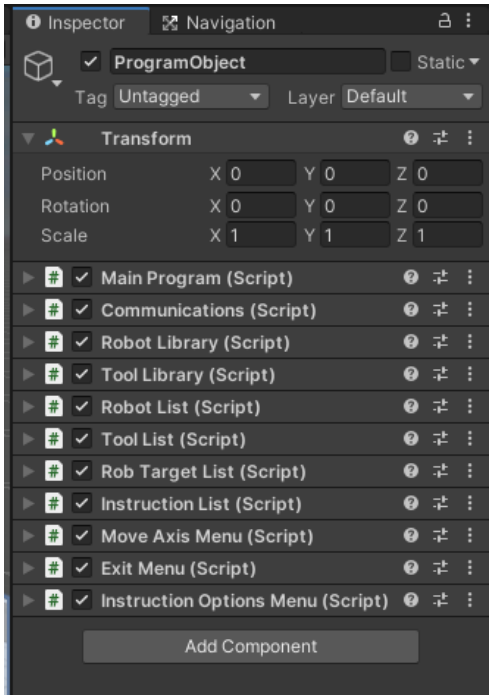

Figura 50 - Componentes añadidos a ProgramObject.

7.2 IMPLEMENTACIÓN DE ROBOTSTUDIO EN UNITY

En esta sección se desarrollan las distintas clases que se han empleado para la funcionalidad directa del programa.

7.2.1 IMPLEMENTACIÓN DE ROBOTS

La implementación de los robots se ha desarrollado con dos objetivos en mente:

- Sistema modular para poder incorporar más modelos
- Bajo consumo de recursos a la hora de simular para no afectar al rendimiento

En los siguientes apartados se irá explicando paso a paso los códigos principales relacionados con la parte funcional de los robots.

7.2.1.1 Ejes del robot

Para gestionar los ejes del robot se ha creado una clase que almacena el ángulo actual y el rango de funcionamiento de dicho eje. Para leer el código con más detalle: *CJoint*

```
using UnityEngine;

public class CJoint : MonoBehaviour
{
    // Public variables
    // Each joint has rotation angle
    public float angle;

    // Each joint has a max and min angle that can rotate
    public float maxAngle;
    public float minAngle;

    // Methods
    public float GetAngle() { return angle; }
    public void SetAngle(float a)
    {
        angle = a;
        // Clamps the angle of the joint
        if (angle > maxAngle)
            angle = maxAngle;
        else if (angle < minAngle)
            angle = minAngle;
    }
}
```

Figura 51 - CJoint

La clase cuenta con dos métodos únicamente. La única que varía de la norma es “SetAngle”, la cual tras asignar el ángulo tiene que comprobar que no supera el ángulo máximo o mínimo de ese eje del robot y en caso de hacerlo, corregirlo.

7.2.1.2 Clase GenericRobot

Clase abstracta que engloba a todos los robots y sus funcionalidades básicas. *GenericRobot.*

```
public abstract class GenericRobot : MonoBehaviour
{
    // Private parameters of Robots
    // Robot identifier
    [Header("Robot Identity")]
    public string displayName;
    public float robType;
    public int id;

    [Header("Joints Parameters")]
    // Number of joints
    public int nJoints;

    // Vector of Joints
    public List<CJoint> joints;

    // Joints transform
    public Transform[] tJoints;

    [Header("Relevant Transforms")]
    // Robot transforms
    public Transform robotPivot;
    public Transform toolReference;
```

Figura 52 - GenericRobot. Parte 1.

En la Figura 52 aparece la primera mitad de este bloque. En éste se definen las variables más importantes de la clase. Para diferenciar los robots se les asigna un nombre para mostrar en pantalla, el tipo de robot y la variable "id" sirve a la hora de gestionar la creación de robots.

Después se definen el número de ejes que tiene el robot y una lista para gestionarlos. También se almacena la variable *Transform* del eje para poder gestionar su movimiento más adelante. También se guardan la variable *Transform* del objeto padre de todo el robot y del objeto que servirá para vincular la herramienta.

```
    // Public methods
    // Get functions
    public int GetId() { return id; }
    public float GetRobType() { return robType; }
    public int GetNumJoints() { return nJoints; }
    public List<CJoint> GetJoints() { return joints;  }
    public CJoint GetJoint(int n)
    {
        if (n < nJoints)
            return joints[n];
        else
            return null;
    }

    // Set functions
    public void SetId(int i) { id = i; }
    public void SetName(string s) { displayName = s; }

    virtual public void MoveAxis(float newAngle, int n) { }
    virtual public void AxesMovement(float[] newAngles) { }
    virtual public void SetRobotPosition(Vector3 newPosition) { }
    virtual public void SetRobotRotation(Vector3 newRotation) { }
}
```

Figura 53 - GenericRobot. Parte 2.

En la *Figura 53* se muestran principalmente funciones de tipo “Get” y “Set”.

El método “get” más relevante es la de los ejes, puesto que hay que seleccionar el número de eje buscado. Las funciones set no tienen nada relevante.

Por último, están definidas como “virtual” una serie de funciones que corresponden principalmente al movimiento del robot. De esta manera se fuerza a las clases hijas a definir funciones concretas para su tipo de robot.

7.2.1.3 Clase IRB120

Define el comportamiento concreto del robot IRB120 del laboratorio. *IRB120.*

La clase consiste en sobrescribir los métodos “virtual” de la clase madre.

```
public override void MoveAxis(float newAngle, int n) // Moves an axis of the
robot
    {
        joints[n].SetAngle(newAngle);
        switch (n)
        {
            case 0:
           tJoints[0].localEulerAngles = new Vector3(0, joints[0].GetAngle(), 0);
                break;

            case 1:
           tJoints[1].localEulerAngles = new Vector3(0, 0, joints[1].GetAngle());
                break;

            case 2:
           tJoints[2].localEulerAngles = new Vector3(0, 0, joints[2].GetAngle());
                break;

            case 3:
           tJoints[3].localEulerAngles = new Vector3(joints[3].GetAngle(), 0, 0);
                break;

            case 4:
           tJoints[4].localEulerAngles = new Vector3(0, 0, joints[4].GetAngle());
                break;

            case 5:
           tJoints[5].localEulerAngles = new Vector3(joints[5].GetAngle(), 0, 0);
                break;

            default:
                Debug.LogError("Error 021: The robot has less joints");
                break;
        }
    }
```

Figura 54 - IRB120. MoveAxis.

En la Figura 54 se sobrescribe el método “MoveAxis”. Se le pasa un ángulo y el número del eje a rotar. Se pasa por un “switch case” y se actualiza el ángulo local de la articulación correspondiente con el parámetro *LocalEulerAngles*. Para esto hay que conocer el eje en el que puede rotar cada articulación.

```
    override public void AxesMovement(float[] newAngles) // Moves all the robot
axis
    {
        for (int i = 0; i < nJoints; i++)
        {
            joints[i].SetAngle(newAngles[i]);
        }

        tJoints[0].localEulerAngles = new Vector3(0, joints[0].GetAngle(), 0);
        tJoints[1].localEulerAngles = new Vector3(0, 0, joints[1].GetAngle());
        tJoints[2].localEulerAngles = new Vector3(0, 0, joints[2].GetAngle());
        tJoints[3].localEulerAngles = new Vector3(joints[3].GetAngle(), 0, 0);
        tJoints[4].localEulerAngles = new Vector3(0, 0, joints[4].GetAngle());
        tJoints[5].localEulerAngles = new Vector3(joints[5].GetAngle(), 0, 0);
    }
```

Figura 55 - IRB120. AxesMovement.

El método "AxesMovement" hace algo parecido al método anterior, pero en vez de tener que seleccionar una articulación concreta, se actualizan todas a la vez.

```
    override public void SetRobotPosition(Vector3 newPosition) // Set the robot
position
    {
        robotPivot.position = newPosition;
    }
    override public void SetRobotRotation(Vector3 newRotation) // Set the robot
rotation
    {
        robotPivot.eulerAngles = newRotation;
    }
```

Figura 56 - IRB120. Métodos de posicionamiento.

Estos métodos únicamente actualizan la posición y rotación de la variable *Transform* del conjunto del robot.

7.2.1.4 Implementación del IRB120

Para implementar el brazo robótico primero se ha tenido que obtener el modelo 3D del mismo. Esto se ha hecho directamente desde la propia aplicación de *RobotStudio*, que te permite descargar el modelo como archivo .fbx, que es compatible con *Unity*.

Sin embargo, el archivo contenía unas mallas de colisión demasiado complejas y ralentizaban la ejecución de forma inadmisible para un proyecto de realidad virtual.

Por tanto, se han simplificado las colisiones de cada segmento del robot con cubos que se aproximan a su forma aproximada. Inicialmente se intentaron simplificar con *Autodesk*

Fusion 360 pero no se conseguía mejorar el rendimiento significativamente y modificaba el aspecto visual del robot.

También se ha tenido que reasignar materiales a cada segmento del robot porque al importarlo los diferentes elementos se habían vuelto monocromos.

En la *Figura 57* se muestra el resultado final en *Unity* con los *Collider* finales.

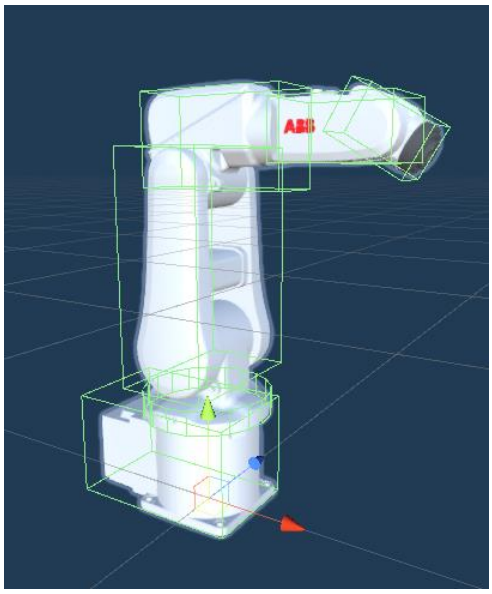

Figura 57 - Prefab del IRB120 con colliders

En la *Figura 58* aparece la jerarquía del *Prefab*. En el elemento de mayor orden se ha añadido el código *IRB120* y configurado sus parámetros en el inspector acorde al modelo de brazo robótico, lo cual consiste en arrastrar los elementos definidos anteriormente y marcar el *IRB120* como robot de tipo 0.

Jerarquizar los elementos del robot es fundamental para vincular su movimiento y que así el robot se mueva de forma conjunta.

Los elementos nombrados “LinkX” corresponden a la componente visual de cada parte del robot. En cada elemento nombrado “Joint_X” se han añadido *CJoint* y configurado los ángulos máximos de cada articulación:

- Joint_1: [-165,165]
- Joint_2: [-110,110]
- Joint_3: [-70,110]
- Joint_4: [-160,160]
- Joint_5: [-120,120]
- Joint_6: [-400,400]

Figura 58 - Jerarquía del IRB120.

7.2.1.5 Librería de robots

Se ha implementado una especie de librería de robots en donde se almacenan los diferentes tipos de robots que están disponibles en el programa. *RobotLibrary*.

```
// List of the different robots in the program
    [Header("Robot library data")]
    public List<RobotLibraryIcon> robotIcons = new List<RobotLibraryIcon>();
    public List<GenericRobot> robotsPrefab = new List<GenericRobot>();

    [Header("Robot types")]
    public int actRobType;

    // Visual variables
    [Header("Library Display")]
    public RobotLibraryDisplay robotLibraryDisplay;
```

Figura 59 - RobotLibrary. Variables.

El código almacena en una lista los *Prefab* de los diferentes robots configurados y la información de los iconos que se mostraran en ventana en la sección correspondiente de la interfaz.

La variable "actRobType" permite al Sistema conocer qué tipo de robot ha sido seleccionado desde la interfaz. Por último, se tiene una variable relacionada con la parte visual de este bloque que se encuentra explicado aquí: *7.5.2.7.5*

```
    void Awake()
    {
        // RobotIcons
        robotIcons[0] = Resources.Load<RobotLibraryIcon>("ABB
Library/Robots/UIIconPrefab/Icon_IRB120");

        // robotsPrefabs
        robotsPrefab[0] = Resources.Load<GenericRobot>("ABB
Library/Robots/ScenePrefab/IRB120/IRB120");
    }

    // Event management
    void Start()
    {
        robotLibraryDisplay.Prime(this);
        RobotLibraryIconDisplay.onClick += HandleonClick;
    }

    void OnDestroy()
    {
        RobotLibraryIconDisplay.onClick -= HandleonClick;
    }

    void HandleonClick(RobotLibraryIcon robot)
    {
        if (robotIcons.Contains(robot))
            actRobType = robot.robType;
    }
```

Figura 60 - RobotLibrary. Métodos.

En la *Figura 60* se encuentran definidos los métodos propios de *Unity* y el método "HandleOnClick".

Para mejorar el rendimiento se ha añadido en el método *Awake* la carga de los recursos que se usaran posteriormente con el método "Resources.Load".

Después se encuentra el método *Start*. Aquí se inicializa la variable visual *RobotLibraryDisplay* que se explica en la sección *7.5.2.7.5*. También se subscribe a los eventos producidos por la clase *RobotLibraryIconDisplay*. El funcionamiento de eventos esta explicado aquí: *Eventos*.

Cuando se seleccione alguno de los iconos mostrados en pantalla se llamará al último método, que actualizará la variable interna que corresponde con el tipo de robot seleccionado.

7.2.1.6 Lista de robots

Se ha creado una clase que manejara la lista de robots en el sistema: *RobotList*.

```
// List of robots
public List<GenericRobot> robots = new List<GenericRobot>();
public int actRobot;

// Display
public RobotScrollListDisplay scrollListDisplay;
public MoveAxisMenu moveAxisMenu;

// Delegate
public delegate void RobotListDelegate(RobotList robotList);
public static event RobotListDelegate OnChanged;
```

Figura 61 - RobotList. Variables.

Los parámetros que contiene esta clase son:

- Una lista de robots.
- Una variable que almacena la posición en la lista del robot activo.
- *MoveAxisMenu* y *RobotScrollListDisplay* son clases que se encargan de representar visualmente a esta clase en la escena.
- El método "RobotListDelegate" y la variable "OnChanged" están relacionados con el sistema de *Eventos*.

```
#region Unity Methods
void Start()
{
    RobotUIDisplay.OnClick += HandleOnClick;
}

void OnDestroy()
{
    RobotUIDisplay.OnClick -= HandleOnClick;
}
#endregion

// Functional Methods
void HandleOnClick(GenericRobot robot)
{
    if (robots.Contains(robot))
    {
        actRobot = robots.IndexOf(robot);
        scrollListDisplay.ShowRobot(actRobot);
        moveAxisMenu.Prime(robot.joints, robot.displayName);
    }
}
```

Figura 62 - RobotList. Métodos de Unity y eventos.

El código de la *Figura 62* sirve para supeditar la llamada al método "HandleOnClick" a los eventos producidos en la clase *RobotUIDisplay*. Este método se encarga de actualizar el robot activo y la interfaz.

```
public void Add(GenericRobot robot)
{
    // Debug functionality
    if (robot == null)
        return;

    robots.Add(robot);
    if (OnChanged != null)
        OnChanged.Invoke(this);

    actRobot = robots.Count - 1;
    moveAxisMenu.Prime(robot.joints, robot.displayName);
    scrollListDisplay.ShowRobot(actRobot);
}
```

Figura 63 - RobotList. Método Add.

El método "Add" sirve para añadir el robot que se le pasa a la lista. Al hacerlo se actualiza el robot activo y la parte visual correspondiente.

Además, se usa el método *Invoke* del evento estático definido en las variables. De esta manera todas las clases subscritas a este evento sabrán cuando se ha actualizado la lista de robots.

```
public void Delete(GenericRobot robot)
{
    // Debug functionality
    if (robot == null)
        return;
    if (!robots.Contains(robot))
        return;

    robots.Remove(robot);
    if (OnChanged != null)
        OnChanged.Invoke(this);

    if (robots.Count > 0)
    {
        actRobot = robots.Count - 1;
        moveAxisMenu.Prime(robot.joints, robot.displayName);
        scrollListDisplay.ShowRobot(actRobot);
    }
    else
    {
        moveAxisMenu.moveAxisMenuDisplay.canvas.enabled = false;
        scrollListDisplay.EraseShowRobot();
    }
}
```

Figura 64 - RobotList. Método Delete.

El código de la *Figura 64* tiene el funcionamiento opuesto al método "Add" definido anteriormente. Se encarga de eliminar robots de la lista. Tras comprobar que el robot a eliminar esta en la lista, se borra y llama de nuevo al evento de la clase. Por último, se vuelve a actualizar el robot activo y la interfaz.

7.2.2 IMPLEMENTACIÓN DE ROBTARGETS

7.2.2.1 Clase RobTarget

Esta clase reproduce las funciones que tendría un *Robtarget* en *RobotStudio*. Se puede analizar el código en detalle aquí: *RobTarget.*

```
    // Name
    public string displayName;
    private int id;

    // Object
    public Transform target;

    // Transform data
    private Vector3 robotStudioPosition = Vector3.zero;
    private Vector3 robotStudioRotation = Vector3.zero;

    // Robot config data
    private int[] cf = new int[] { 0, 0, 0, 0 };

    // Joint data
    public float[] angles;
```

Figura 65 - RobTarget. Método Variables.

Los parámetros que contiene esta clase son:

- Un nombre para identificar el objeto.
- Un identificador para gestionar su creación en listas.
- Una variable tipo *Transform* que almacena la posición y rotación del objeto en *Unity*.
- Dos vectores para almacenar la posición y rotación correspondientes a *RobotStudio*.
- Un vector de enteros que funciona como el tipo de variable *Confdata*.
- Un vector para almacenar el ángulo de las articulaciones del robot para alcanzar dicho objetivo.

Es necesario usar variables diferentes para almacenar la posición del objetivo debido a que *Unity* y *RobotStudio* usan un sistema de ejes diferente.

```
    public void Prime(int d, float robType, List<CJoint> joints, Transform t,
string name)
    {
        // Target name
        displayName = name;

        // Identifier
        id = d;

        angles = new float[joints.Count];
        // Joints
        for(int i = 0; i < joints.Count; i++)
        {
            angles[i] = joints[i].angle;
        }

        // World position
        target.position = t.position;
        target.rotation = t.rotation;

        // robTarget position
        robotStudioPosition.x = t.position.x * 1000;
        robotStudioPosition.y = t.position.z * 1000;
        robotStudioPosition.z = t.position.y * 1000;

        // robTarget global rotation with quaternions
        robotStudioRotation.x = -t.eulerAngles.x;
        robotStudioRotation.y = -t.eulerAngles.z;
        robotStudioRotation.z = -t.eulerAngles.y;

        // Robot configuration data
        if (robType == 0)
        {
            // cf1
            cf[0] = RobConf(-joints[0].GetAngle());

            // cf4
            cf[1] = RobConf(-joints[3].GetAngle());

            // cf6
            cf[2] = RobConf(-joints[5].GetAngle());

            // cfx
            cf[3] = 0; // cfx is not used in IRB120 robots
        }
    }
```

Figura 66 - RobTarget. Método Constructor.

El método “Prime” de esta clase es llamada para inicializar la variable.

Para esto hace falta pasarle:

- Un entero para el identificador.
- Un entero que le indique el tipo de robot para el que se está creando el objetivo.
- Una lista con las articulaciones del robot.

- Una variable tipo *Transform* que coincida con la posición en la que se desea crear el objetivo.
- Una cadena de caracteres que sirvan de nombre.

Para el vector que almacena la posición del objetivo en *RobotStudio* hace falta cambiar el eje "Z" por el "Y" con respecto a *Unity*. También es necesario multiplicar por 1000 el valor porque Unity almacena la distancia en milímetros y *RobotStudio* en metros.

Para el vector que almacena la rotación hay que volver a cambiar el eje "Z" por el "Y" además de cambiarlos de signo.

```
// Rob conf selector
private int RobConf(float a)
{
    if (a >= 0)
    {
        return (int)(a/90f);
    }
    else
    {
        return (int)((a/90f) - 1);
    }
}
```

Figura 67 - RobTarget. Método RobConf.

El método "RobConf" sirve para obtener la configuración de los ejes del robot con los mismos resultados que aportaría la variable *Confdata* de *RobotStudio*.

```
// Get functions
public string GetName() { return displayName;  }
public int GetId() { return id; }
public Vector3 GetRSPosition() { return robotStudioPosition; }
public Vector3 GetRSRotation() { return robotStudioRotation; }
public int GetRSConfig(int n) { return cf[n]; }
public Transform GetRobTargetTransform() { return target; }
```

Figura 68 - RobTarget. Get Methods.

El resto de los métodos sirven para acceder a las variables internas de la clase.

7.2.2.2 Prefab de RobTargets

Para su representación en la escena se ha creado un *Prefab* al que se ha añadido el código *RobTarget*.

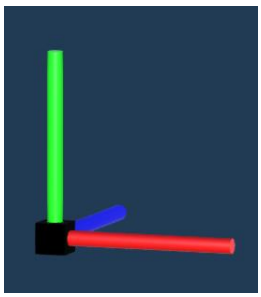

Ilustración 2 - Prefab de RobTarget

Simplemente se ha creado un cubo y tres cilindros para crear la representación visual de un punto.

7.2.2.3 Lista de RobTargets

De forma similar a la clase *RobotList* explicada anteriormente, se ha creado una clase que sirve para gestionar una lista de los *RobTarget* que se crean durante la ejecución. Se puede analizar el código en detalle aquí: *RobTargetList.*

```
    // List of targets
    public List<RobTarget> targets = new List<RobTarget>();
    public int actTarget;

    // Extra functional variables
    public bool placeRobot;

    // Display
    public RobTargetScrollListDisplay ScrollListDisplay;

    // Delegate
    public delegate void RobTargetListDelegate(RobTargetList robTargetList);
    public static event RobTargetListDelegate OnChanged;

    // Prefab
    public RobTarget robtargetPrefab;
```

Figura 69 - RobTargetList. Variables.

Los parámetros que componen a esta clase son:

- Una lista de *RobTarget*.
- Un parámetro para conocer el objetivo activo de la lista.
- Una variable booleana que sirve para posicionar el robot desde la rutina principal cuando se ha seleccionado alguno de los puntos objetivo.

- *RobTargetScrollListDisplay* es la clase que se encarga de representar esta clase en la escena. *7.5.2.9.2*
- El método "RobTargetListDelegate" y la variable "OnChanged" están relacionados con el sistema de *Eventos*.
- Por último, cuenta con el *Prefab* que se ha diseñado para los *RobTarget*

```
#region Unity Methods
void Start()
{
    RobTargetUIDisplay.OnClick += HandleonClick;
}

void OnDestroy()
{
    RobTargetUIDisplay.OnClick -= HandleonClick;
}
#endregion

void HandleonClick(RobTarget target)
{
    if (targets.Contains(target))
    {
        actTarget = targets.IndexOf(target);
        ScrollListDisplay.ShowTarget(actTarget);

        placeRobot = true;
    }
}
```

Figura 70 - RobTargetList. Métodos de Unity y eventos.

El código de la *Figura 70* se usa para supeditar la llamada a la función "HandleonClick" a los eventos producidos en la clase *RobTargetUIDisplay*. Este método se encarga de actualizar el valor del objetivo activo, actualizar la interfaz y cambiar el valor de la variable "placeRobot".

```
// Functional Methods
public void Add(RobTarget target)
{
    // Debug functionality
    if (target == null)
        return;

    targets.Add(target);
    if (OnChanged != null)
        OnChanged.Invoke(this);

    actTarget = targets.Count - 1;
    ScrollListDisplay.ShowTarget(actTarget);
}
```

Figura 71 - RobTargetList. Método Add.

El método "Add" sirve para añadir el *RobTarget* que se le pasa a la lista. Al hacerlo se actualiza el *RobTarget* activo y la parte visual correspondiente.

Además, se usa el método *Invoke* del evento estático definido en las variables. De esta manera todas las clases subscritas a este evento sabrán cuando se ha actualizado la lista de *RobTarget*.

```
public void Delete(RobTarget target)
{
    // Debug functionality
    if (target == null)
        return;
    if (!targets.Contains(target))
        return;

    targets.Remove(target);
    if (OnChanged != null)
        OnChanged.Invoke(this);

    if (targets.Count > 0)
    {
        actTarget = targets.Count - 1;
        ScrollListDisplay.ShowTarget(actTarget);
    }
    else
        ScrollListDisplay.EraseShowTarget();
}
```

Figura 72 - RobTargetList. Método Delete.

El código de la *Figura 72* tiene el funcionamiento opuesto al método "Add" definido anteriormente. Se encarga de eliminar *RobTarget* de la lista. Tras comprobar que el *RobTarget* a eliminar esta en la lista, se borra y llama de nuevo al evento de la clase. Por último, se vuelve a actualizar el *RobTarget* activo y la interfaz.RobTarget

7.2.3 IMPLEMENTACIÓN DE HERRAMIENTAS

La forma en la que se han implementado las herramientas tiene como objetivo poder añadir más en el futuro. En los siguientes apartados se irá explicando paso a paso los códigos principales relacionados con la parte funcional de las herramientas.

7.2.3.1 Clase Tool

Clase abstracta que englobaba a todas las herramientas y sus funcionalidades básicas. *Tool.*

```
    // Identifying variables
    public string displayName;
    public int id;
    public int toolType;

    // Boolean that verifies if the robot holds the tool
    [SerializeField]
    protected bool robHold;

    // Transform of the TCP of the tool
    [SerializeField]
    protected Transform tcp;

    // Transform of the wrist of the tool
    public Transform wrist;
    private Vector3 pastPosition;
    private Quaternion pastRotation;
```

Figura 73 - Tool. Variables.

Los parámetros que contiene esta clase son:

- "displayName" que sirve para almacenar el nombre del objeto.
- "id" que se utiliza para gestionar la creación de las herramientas.
- "toolType" que sirve para identificar los distintos tipos de herramientas.
- La variable "robHold" sirve para comprobar si algún robot está sosteniendo la herramienta.
- Las variables "TCP" y "wrist" sirven para almacenar posiciones clave de la herramienta. La variable "TCP" hace referencia al punto *TCP* que tendría en *RobotStudio*. La variable "wrist" es el punto de anclaje con el robot.
- También se almacenan la posición y rotación de la herramienta antes de vincularse con el robot para cuando se desvincule.

```
    // Get functions
    public int GetId() { return id; }
    public Transform GetTcp() { return tcp; } // Returns the tcp of the tool
    public bool GetToolState() { return robHold; } // Returns the robHold state
```

Figura 74 - Tool. Métodos Get.

Los métodos de la *Figura 74* sirven para acceder a variables internas de la clase.

```
    public void BindTool(Transform newParent) // Binds the tool to the robot
    {
        pastPosition = wrist.position;
        pastRotation = wrist.rotation;

        robHold = true;
        wrist.position = newParent.position;
        wrist.rotation = newParent.rotation;
        wrist.SetParent(newParent);
    }
```

Figura 75 - Tool. Método BindTool.

El método "BindTool" sirve para vincular la herramienta al elemento definido como "ToolReference" en el robot. Para esto se guarda la posición anterior, se actualiza la posición y rotación de la herramienta y por último se convierte a la variable *Transform* "wrist" en hija del objeto pasado a la función.

```
    public void UnBindTool() // Unbinds the tool from the robot
    {
        robHold = false;
        wrist.SetParent(null);
        wrist.position = pastPosition;
        wrist.rotation = pastRotation;
    }
```

Figura 76 - Tool. Método UnBindTool.

El método "UnBindTool" es la operación inversa al método "BindTool".

```
    // Set functions
    public void SetId(int n) { id = n; }
    public void SetName(string s) { displayName = s; }
    public void SetTcp(Vector3 pos, Quaternion rot) // Set the tcp position of
the tool
    {
        tcp.position = pos;
        tcp.rotation = rot;
    }
    public void SetWrist(Vector3 pos, Quaternion rot) // Public function to move
the tool
    {
        wrist.position = pos;
        wrist.rotation = rot;
    }
```

Figura 77 - Tool. Métodos Set.

Los métodos de la *Figura 77* sirven para cambiar los valores de las variables internas.

7.2.3.2 Clase Tool0

Esta clase sirve para definir el comportamiento específico de la herramienta básica y para añadirse al *Prefab* de la herramienta que se explica más adelante. Como las funcionalidades básicas ya están incluidas en la clase *Tool* la clase *Tool0* está vacía.

7.2.3.3 Implementación de Tool0

Para implementar la herramienta básica se ha creado un *GameObject* con los mismos elementos visuales que tiene un *RobTarget.*

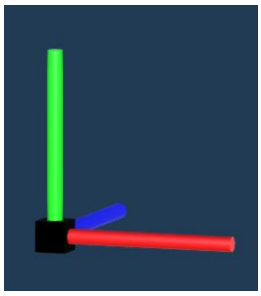

Ilustración 3 - Herramienta Tool0

7.2.3.4 Librería de herramientas

Se ha implementado una especie de librería de herramientas en donde se almacenan los diferentes tipos de herramientas que están disponibles en el programa. *ToolLibrary*.

```
    // List of the different robots in the program
    [Header("Tool library data")]
    public List<ToolLibraryIcon> toolIcons = new List<ToolLibraryIcon>();
    public List<Tool> toolsPrefab = new List<Tool>();

    public int actToolType;

    // Display
    [Header("Library Display")]
    public ToolLibraryDisplay toolLibraryDisplay;
```

Figura 78 - ToolLibrary. Variables.

El código almacena en una lista los *Prefab* de las diferentes herramientas configuradas y la información de los iconos que se mostraran en ventana en la sección correspondiente de la interfaz.

La variable "actToolType" permite al Sistema conocer qué tipo de herramienta ha sido seleccionado desde la interfaz. Por último, se tiene una variable relacionada con la representación visual de esta clase en la sección *7.5.2.8.5*

```
    void Start()
    {
        toolLibraryDisplay.Prime(this);
        ToolLibraryIconDisplay.onClick += HandleonClick;
    }

    void OnDestroy()
    {
        ToolLibraryIconDisplay.onClick -= HandleonClick;
    }
    #endregion

    void HandleonClick(ToolLibraryIcon tool)
    {
        if (toolIcons.Contains(tool))
            actToolType = tool.toolType;
    }
```

Figura 79 - ToolLibrary. Métodos de Unity y eventos.

A continuación, se encuentran definidos los métodos propios de *Unity*. Para mejorar el rendimiento se ha añadido en el método *Awake* la carga de los recursos que se usarán posteriormente con el método "Resources.Load".

Después se encuentra el método *Start*. Aquí se inicializa la variable visual *ToolLibraryDisplay* que se explica en la sección *7.5.2.8.5*. También se subscribe a los eventos producidos por la clase *ToolLibraryIconDisplay*. El funcionamiento de eventos esta explicado aquí: *Eventos.*

Cuando se seleccione alguno de los iconos mostrados en pantalla se llamará al último método, que actualizará la variable interna que corresponde con el tipo de robot seleccionado.

7.2.3.5 Lista de herramientas

Se ha creado una clase que manejará la lista de herramientas en el sistema: *ToolList.*

```
// List of tools
    public List<Tool> tools = new List<Tool>();
    public int actTool;

    // Display
    public ToolScrollListDisplay ScrollListDisplay;

    // Delegate
    public delegate void ToolScrollListDelegate(ToolList toolList);
    public static event ToolScrollListDelegate OnChanged;
```

Figura 80 - ToolList. Variables.

Los parámetros que contiene esta clase son:

- Una lista de herramientas.
- Una variable que almacena la posición en la lista de la herramienta activa.
- *ToolScrollListDisplay* es una clase que se encarga de representar visualmente a esta clase en la escena. *7.5.2.8.2*
- El método "ToolListDelegate" y la variable "OnChanged" están relacionados con el sistema de *Eventos.*

```
    #region Unity Methods
    void Start()
    {
        ToolUIDisplay.OnClick += HandleOnClick;
    }

    void OnDestroy()
    {
        ToolUIDisplay.OnClick -= HandleOnClick;
    }
    #endregion

    // Functional Methods
    void HandleOnClick(Tool tool)
    {
        if (tools.Contains(tool))
        {
            actTool = tools.IndexOf(tool);
            ScrollListDisplay.ShowTool(actTool);
        }
    }
```

Figura 81 - ToolList. Métodos de Unity y eventos.

El código de la *Figura 81* sirve para supeditar la llamada al método "HandleOnClick" a los eventos producidos en la clase *ToolUIDisplay*. Este método se encarga de actualizar la herramienta activa y la interfaz.

```
public void Add(Tool tool)
{
    // Debug functionality
    if (tool == null)
        return;

    tools.Add(tool);
    if (OnChanged != null)
        OnChanged.Invoke(this);

    actTool = tools.Count - 1;
    ScrollListDisplay.ShowTool(actTool);
}
```

Figura 82 - ToolList. Método Add.

El método "Add" sirve para añadir la herramienta que se le pasa a la lista. Al hacerlo se actualiza la herramienta activa y la parte visual correspondiente.

Además, se usa el método *Invoke* del evento estático definido en las variables. De esta manera todas las clases subscritas a este evento sabrán cuando se ha actualizado la lista de herramientas.

```
public void Delete(Tool tool)
{
    if (tool == null)
        return;
    if (!tools.Contains(tool))
        return;

    tools.Remove(tool);
    if (OnChanged != null)
        OnChanged.Invoke(this);

    if (tools.Count > 0)
    {
        actTool = tools.Count - 1;
        ScrollListDisplay.ShowTool(actTool);
    }
    else
        ScrollListDisplay.EraseShowTool();
}
```

Figura 83 - ToolList. Método Delete.

El código de la *Figura 83* tiene el funcionamiento opuesto al método "Add" definido anteriormente. Se encarga de eliminar herramientas de la lista. Tras comprobar que la

herramienta a eliminar esta en la lista, se borra y llama de nuevo al evento de la clase. Por último, se vuelve a actualizar la herramienta activa y la interfaz.

7.2.4 IMPLEMENTACIÓN DE INSTRUCCIONES

7.2.4.1 Clase Instruction

Esta clase almacena la información necesaria para crear una instrucción en *RobotStudio*.

El código completo se encuentra aquí: *Instruction.*

```
// Variables for instruction
    public string displayName;
    private string moveType;
    private string displaySpeed;
    private string displayZone;
    private int speeddata;
    private int zonedata;
    public RobTarget[] targets;
```

Figura 84 - Instruction. Variables.

Los parámetros que componen esta clase son bastante sencillos. Varias cadenas de caracteres para la información que se mostrará en pantalla, dos enteros que determinan el nivel de velocidad y precisión en la instrucción y un vector que almacena los *RobTarget* que se necesitan para la instrucción.

```
    // Constructor
    public Instruction(string n, RobTarget[] r, int m, int s, int z)
    {
        targets = r;
        // MoveType selector
        switch (m)
        {
            …
        }

        if (s >= 0 && s <= 19)
        {
            speeddata = s;
            // Speed display switch
            switch (speeddata)
            {
                …
            }
        }
        else
        {
            Debug.Log("ERROR 023: That speed doesn't exist");
            return;
        }

        if (z >= 0 && z <= 14)
        {
            zonedata = z;
            // Precision display switch
            switch (zonedata)
            {
                …
            }
        }
        else
        {
            Debug.Log("ERROR 024: That precision doesn't exist");
            return;
        }

        if (m == 2)
        {
            displayName = n + " [" + targets[0].displayName + "," +
targets[1].displayName + "," + displaySpeed + "," + displayZone + "]";
        }
        else
        {
            displayName = n + " [" + targets[0].displayName + "," + displaySpeed
+ "," + displayZone + "]";
        }
    }
```

Figura 85 - Instruction. Constructor.

El constructor simplemente asigna los valores que se le pasan al método a las variables internas. Para asignar el valor de las cadenas de caracteres se usan “switch case”. Pero solo

en el caso de la velocidad y la precisión, de esta manera se muestran de la misma forma que lo harían en *RobotStudio.*

Para asignar el nombre que se mostrará en pantalla se tiene en cuenta el tipo de movimiento y se irán añadiendo los diferentes parámetros de la clase.

```
    // Get Functions
    public int GetSpeed() { return speeddata; }
    public int GetPrecision() { return zonedata; }
    public string GetMoveType() { return moveType; }
```

Figura 86 - Instruction. Métodos Get.

Los últimos métodos definidos en la clase sirven para poder acceder a las variables internas.

7.2.4.2 Lista de instrucciones

Se ha creado una clase que manejará la lista de herramientas en el sistema: *InstructionList.*

```
    // List of instructions
    public List<Instruction> orders = new List<Instruction>();

    // List of the targets used in the instructions in order
    public List<RobTarget> targets = new List<RobTarget>();

    // Selected Instruction
    public int actInstr;

    // List Display
    public InstructionScrollListDisplay scrollListDisplay;

    // Delegate
    public delegate void InstructionListDelegate(InstructionList
InstructionList);
    public static event InstructionListDelegate OnChanged;
```

Figura 87 - InstructionList. Variables.

Los parámetros que componen a esta clase son:

- Una lista de *Instruction*.
- Una lista de *RobTarget.*
- Un parámetro para conocer la instrucción activa de la lista.
- *InstructionScrollListDisplay* es la clase que se encarga de representar esta clase en la escena. *7.5.2.9.4*

- El método "InstructionListDelegate" y la variable "OnChanged" están relacionados con el sistema de *Eventos*.

```
#region Unity Methods
// Event management
void Start()
{
    InstructionUIDisplay.OnClick += HandleOnClick;
}

void OnDestroy()
{
    InstructionUIDisplay.OnClick -= HandleOnClick;
}

#endregion

void HandleOnClick(Instruction order)
{
    if (orders.Contains(order))
    {
        actInstr = orders.IndexOf(order);
        scrollListDisplay.ShowInstruction(actInstr);
    }
}
```

Figura 88 - InstructionList. Métodos de Unity y eventos.

El código de la *Figura 88* se usa para supeditar la llamada a la función "HandleonClick" a los eventos producidos en la clase *InstuctionUIDisplay*. Este método se encarga de actualizar el valor de la instrucción activa y actualizar la interfaz

```
// Public functions
public void Add(Instruction order)
{
    // Debug functionality
    if (order == null)
        return;

    orders.Add(order);

    if (order.GetMoveType() == "C")
    {
        targets.Add(order.targets[0]);
        targets.Add(order.targets[1]);
    }
    else
        targets.Add(order.targets[0]);

    if (OnChanged != null)
        OnChanged.Invoke(this);

    actInstr = orders.Count - 1;
    scrollListDisplay.ShowInstruction(actInstr);
}
```

Figura 89 - InstructionList. Método Add.

El método “Add” sirve para añadir la instrucción que se le pasa a la lista. Al hacerlo se actualiza la instrucción activa y la parte visual correspondiente.

Además, se usa el método *Invoke* del evento estático definido en las variables. De esta manera todas las clases subscritas a este evento sabrán cuando se ha actualizado la lista de *Instruction*.

```
public void Delete(Instruction order)
{
    // Debug functionality
    if (order == null)
        return;
    if (!orders.Contains(order))
        return;

    orders.Remove(order);

    if (order.GetMoveType() == "C")
    {
        targets.Remove(order.targets[0]);
        targets.Remove(order.targets[1]);
    }
    else
        targets.Remove(order.targets[0]);

    if (OnChanged != null)
        OnChanged.Invoke(this);

    if (orders.Count > 0)
    {
        actInstr = orders.Count - 1;
        scrollListDisplay.ShowInstruction(actInstr);
    }
    else
        scrollListDisplay.EraseShowInstruction();
}
```

Figura 90 - InstructionList. Método Delete.

El código de la *Figura 90* tiene el funcionamiento opuesto al método "Add" definido anteriormente. Se encarga de eliminar instrucciones de la lista. Tras comprobar que la instrucción a eliminar esta en la lista, se borra y llama de nuevo al evento de la clase. Por último, se vuelve a actualizar la instrucción activa y la interfaz.

RobTarget

7.3 RUTINA PRINCIPAL

En esta sección se explicará el funcionamiento y la composición del código principal del programa. *MainProgram*

```
    public Communications comms;
    public RobTargetList targetList;
    public RobotList robotList;
    public RobotLibrary robotLibrary;
    public MoveAxisMenu moveAxisMenu;
    public ToolList toolList;
    public ToolLibrary toolLibrary;
    public InstructionList instructionList;
    public InstructionOptionsMenu instructionOptions;
    public float lerpTime = 0.05f;
```

Figura 91 - MainProgram. Variables.

Las variables que componen esta clase son principalmente elementos diseñados especialmente para el programa:

- Una clase *Communications* que engloba todo el sistema de comunicaciones del programa. Se explica en la sección 7.4.
- Una clase *RobTargetList* que se explicó en la sección 7.2.2.3.
- Una clase *RobotList* que se explicó en la sección 7.2.1.6.
- Una clase RobotLibrary que se explicó en la sección 7.2.1.5.
- Una clase MoveAxisMenu que se explica en la sección 7.5.2.4.2
- Una clase ToolList que se explicó en la sección 7.2.3.5
- Una clase ToolLibrary que se explicó en la sección 7.2.3.4.
- Una clase InstructionList que se explicó en la sección 7.2.4.2.
- Una clase InstructionOptionsmenu que se explicó en la sección 7.5.2.9.5.
- La variable “lerpTime” se emplea en la ejecución de movimiento en el método *Update*.

```
    public void AddTarget()
    {
        RobTarget t;
        string name;
        int id;
        if (robotList.robots.Count > 0)
        {
            t = (RobTarget)Instantiate(targetList.robtargetPrefab);

            if (targetList.targets.Count > 0)
            {
                id = targetList.targets[targetList.targets.Count - 1].GetId() +
1;
                name = "Target_" + (id + 1).ToString() + "0";
            }
            else
            {
                name = "Target_10";
                id = 0;
            }

            t.Prime(id, robotList.robots[robotList.actRobot].GetRobType(),
robotList.robots[robotList.actRobot].joints,
toolList.tools[toolList.actTool].GetTcp(), name);

            targetList.Add(t);
        }
        else
        {
            Debug.Log("ERROR 015: There are no robots");
        }
    }
```

Figura 92 - MainProgram. Método AddTarget.

El método "AddTarget" será llamado desde la interfaz. Su funcionamiento consiste en añadir *RobTarget* a la escena a partir de un *Prefab*. Al hacerlo tiene en cuenta los *RobTarget* ya creados modificando nombre e identificador en consecuencia. Una vez creado lo inicializa y posteriormente añade a la lista.

```
    public void DeleteTarget()
    {
        if (robotList.robots.Count > 0)
        {
            RobTarget t;

            if (targetList.targets.Count > 0)
            {
                t = targetList.targets[targetList.actTarget];
                targetList.Delete(t);

                Destroy(t.gameObject);
            }
            else
                Debug.Log("ERROR 016: There are no targets to remove");
        }
    }
```

Figura 93 - MainProgram. Método DeleteTarget.

El método "DeleteTarget" realiza la operación contraria a "AddTarget". Es importante recalcar que para eliminar correctamente el *RobTarget* hay que borrarlo de la lista y a continuación invocar al destructor de la clase *GameObject.*

```
    public void AddRobot()
    {
        GenericRobot t;
        string name;
        switch (robotLibrary.actRobType)
        {
            case 0:
                name = "IRB120";
                t = (IRB120)Instantiate(robotLibrary.robotsPrefab[0]);

                if (robotList.robots.Count > 0)
                {
                    t.SetId(robotList.robots[robotList.robots.Count - 1].GetId()
+ 1);
                    t.SetName(name + "_" +
(robotList.robots[robotList.robots.Count - 1].GetId() + 1).ToString());
                }

                robotList.Add(t);
                break;

            default:
                Debug.Log("ERROR 017: No robot selected");
                break;
        }
    }
```

Figura 94 - MainProgram. Método AddRobot.

El método "AddRobot" será llamado desde la interfaz. Su funcionamiento consiste en añadir el tipo de robot activo a la escena a partir de un *Prefab*. Al hacerlo tiene en cuenta los robots

ya creados modificando nombre e identificador en consecuencia. Una vez creado lo inicializa y posteriormente añade a la lista.

```
    public void DeleteRobot()
    {
        if (robotList.robots.Count > 0)
        {
            GenericRobot t = robotList.robots[robotList.actRobot];
            robotList.Delete(t);
            Destroy(t.gameObject);
        }
        else
            Debug.Log("ERROR 015: There are no robots");
    }
```

Figura 95 - MainProgram. Método DeleteRobot.

El método "DeleteRobot" realiza la operación contraria a "AddRobot". Es importante recalcar que para eliminar correctamente el robot hay que borrarlo de la lista y a continuación invocar al destructor de la clase *GameObject.*

```
    public void AddTool()
    {
        Tool t;
        string name;
        switch (toolLibrary.actToolType)
        {
            case 0:
                name = "Tool0";
                t = (Tool0)Instantiate(toolLibrary.toolsPrefab[0]);

                if (toolList.tools.Count > 0)
                {
                    t.SetId(toolList.tools[toolList.tools.Count - 1].GetId() +
1);
                    t.SetName(name + "_" + (toolList.tools[toolList.tools.Count -
1].GetId() + 1).ToString());
                }

                toolList.Add(t);;
                break;

            default:
                Debug.Log("ERROR 018: No tool selected");
                break;
        }
    }
```

Figura 96 - MainProgram. Método AddTool.

El método "AddTool" será llamado desde la interfaz. Su funcionamiento consiste en añadir el tipo de herramienta activa a la escena a partir de un *Prefab*. Al hacerlo tiene en cuenta las

herramientas ya creadas modificando nombre e identificador en consecuencia. Una vez creado la inicializa y posteriormente añade a la lista.

```
    public void DeleteTool()
    {
        if (toolList.tools.Count > 0)
        {
            Tool t = toolList.tools[toolList.actTool];
            toolList.Delete(t);
            Destroy(t.gameObject);
        }
        else
            Debug.Log("ERROR 019: There are no tools");
    }
```

Figura 97 - MainProgram. Método DeleteTool.

El método "DeleteTool" realiza la operación contraria a "AddTool". Es importante recalcar que para eliminar correctamente la herramienta hay que borrarlo de la lista y a continuación invocar al destructor de la clase *GameObject.*

```
    public void BindTool()
    {
        if (toolList.tools.Count > 0 && robotList.robots.Count > 0)

toolList.tools[toolList.actTool].BindTool(robotList.robots[robotList.actRobot].to
olReference);
        else
            Debug.Log("ERROR 019: There are no tools");
    }
```

Figura 98 - MainProgram. Método BindTool.

El método "BindTool" es llamada desde la interfaz. Su funcionamiento consiste en vincular la herramienta activa al robot activo con el método "BindTool" definido en la clase *Tool*.

```
    public void UnBindTool()
    {
        if (toolList.tools.Count > 0 && robotList.robots.Count > 0)
            toolList.tools[toolList.actTool].UnBindTool();
        else
            Debug.Log("ERROR 019: There are no tools");
    }
```

Figura 99 - MainProgram. Método UnBindTool.

El método "UnBindtool" realiza la operación contraria a "BindTool".

```
    public void AddInstruction()
    {
        RobTarget[] targets;
        Instruction t;
        string name = "MoveL";

        if (targetList.targets.Count > 0)
        {
            if (instructionOptions.GetActMoveType() == 2)
            {
                name = "MoveC";
                targets = new RobTarget[] {
targetList.targets[instructionOptions.GetActTarget1()],
targetList.targets[instructionOptions.GetActTarget2()] };
            }
            else
            {
                if (instructionOptions.GetActMoveType() != 0)
                    name = "MoveJ";
                targets = new RobTarget[] {
targetList.targets[instructionOptions.GetActTarget1()] };
            }

            t = new Instruction(name, targets,
instructionOptions.GetActMoveType(), instructionOptions.GetActSpeed(),
instructionOptions.GetActPrecision());

            instructionList.Add(t);
        }
    }
```

Figura 100 - MainProgram. Método AddInstruction.

El método "AddInstruction" será llamado desde la interfaz. Su funcionamiento consiste en añadir una instrucción a la lista teniendo en cuenta los parámetros seleccionados por el usuario.

```
    public void DeleteInstruction()
    {
        Instruction t;
        if (instructionList.orders.Count > 0)
        {
            t = instructionList.orders[instructionList.actInstr];
            instructionList.Delete(t);
        }
        else
            Debug.Log("ERROR 020: There are no Instructions to remove");
    }
```

Figura 101 - MainProgram. Método DeleteInstruction.

El método "DeleteInstruction" realiza la operación opuesta al método "AddInstruction". Al no haber un elemento que represente a la clase *Instruction* en la escena no es necesario un tipo de eliminación especial como en los casos anteriores.

```
    // Start is called before the first frame update
    void Start()
    {
        // Initializing components
        comms = gameObject.GetComponent(typeof(Communications)) as
Communications;
        targetList = gameObject.GetComponent(typeof(RobTargetList)) as
RobTargetList;
        robotList = gameObject.GetComponent(typeof(RobotList)) as RobotList;
        robotLibrary = gameObject.GetComponent(typeof(RobotLibrary)) as
RobotLibrary;
        moveAxisMenu = gameObject.GetComponent(typeof(MoveAxisMenu)) as
MoveAxisMenu;
        toolList = gameObject.GetComponent(typeof(ToolList)) as ToolList;
        toolLibrary = gameObject.GetComponent(typeof(ToolLibrary)) as
ToolLibrary;
        instructionList = gameObject.GetComponent(typeof(InstructionList)) as
InstructionList;
        instructionOptions =
gameObject.GetComponent(typeof(InstructionOptionsMenu)) as
InstructionOptionsMenu;

        Application.targetFrameRate = 300;
    }
```

Figura 102 - MainProgram. Método Start.

El método *Start* se ha usado para inicializar las variables internas con los demás componentes del objeto. También se ha fijado un objetivo de fotogramas por segundo. Como el programa está orientado a la realidad virtual tiene que garantizar ser superior a los 90 fotogramas por segundo. Para asegurarse que los fotogramas serán lo suficientemente altos se ha fijado un valor de 300.

```
    // Update is called once per frame
    void Update()
    {
        // Movement of the active robot with RobotStudio angles received with UDP
communications
        if(comms.sim && (comms.rAngles.Length ==
robotList.robots[robotList.actRobot].GetNumJoints()))
        {
            float[] tAngles = new
float[robotList.robots[robotList.actRobot].GetNumJoints()];

            for (int i = 0; i <
robotList.robots[robotList.actRobot].GetNumJoints(); i++)
                tAngles[i] =
Mathf.Lerp(robotList.robots[robotList.actRobot].GetJoint(i).GetAngle(),
comms.rAngles[i], Time.deltaTime/0.005f);

            if (comms.inPlay)
            {
                robotList.robots[robotList.actRobot].AxesMovement(tAngles);

moveAxisMenu.UpdateValues(robotList.robots[robotList.actRobot].GetJoints(),
robotList.robots[robotList.actRobot].displayName);
            }
        }

        // Specific angle selected with input fields
        if(moveAxisMenu.needChange)
        {
            for (int i = 0; i <
robotList.robots[robotList.actRobot].GetNumJoints(); i++)
            {

robotList.robots[robotList.actRobot].MoveAxis(moveAxisMenu.joints[i].GetAngle(),
i);
            }

            moveAxisMenu.needChange = false;
        }

        // Move the robot to the selected robtarget
        if(targetList.placeRobot && !comms.sim)
        {

robotList.robots[robotList.actRobot].AxesMovement(targetList.targets[targetList.a
ctTarget].angles);
            moveAxisMenu.Prime(robotList.robots[robotList.actRobot].joints,
robotList.robots[robotList.actRobot].displayName);

            targetList.placeRobot = false;
        }
    }
```

Figura 103 - MainProgram. Método Update.

El método *Update* se ha dividido en tres partes, que cumplen tres funciones distintas.

El primer bloque "IF" corresponde a la ejecución del programa durante la simulación. Se guardan los ángulos recibidos a través del *Protocolo UDP/IP* de las comunicaciones y teniendo en cuenta el espacio de tiempo entre los mensajes recibidos se realiza una interpolación entre el ángulo actual de los ejes del robot y el ángulo objetivo. Por último, se actualizan los valores mostrados en la interfaz.

El segundo bloque "IF" se utiliza cuando el usuario ha introducido por teclado el ángulo que quiere que alcance una articulación concreta.

El tercer y último bloque "IF" se usa cuando el usuario selecciona uno de los *RobTarget* creados desde la interfaz. Al ejecutarse sitúa el robot en la posición en la que estaba al crear el *RobTarget* en cuestión.

7.4 COMUNICACIONES

En esta sección se explicará el funcionamiento del código *Communications* que se encarga de gestionar los protocolos *Protocolo TCP*/IP y *Protocolo UDP*/IP en *Unity*. El código es largo y complejo por lo que se ha dividido en varias regiones.

En esta clase es importante mencionar los módulos de Csharp que se han empleado:

- System.Globalization que contiene información relativa a la referencia cultura [**16**].
- System.Net que será necesario para el *Protocolo UDP/IP* [**17**].
- System.Net.Sockets que será necesario para crear los clientes de los protocolos de comunicación [**18**].
- System.Threading que sirve para tener varios procesos ejecutándose de forma simultánea [**19**].
- UnityEngine y UnityEngine.UI son librerías propias de *Unity* [**20**].

7.4.1.1 Variables

La definición de variables está distribuida según su funcionamiento en el código.

```
    [Header("Behaviour Variables")]
    [Tooltip("Boolean that tells if the program is running or not")] public bool
abort;
    [Tooltip("Flag to start the synchronization with RobotStudio")] public bool
sendSync;
    [Tooltip("Bollean that tells if the simulation state must be updated")]
public bool upPlay;
    [Tooltip("Bollean that tells if the simulation must be in loop")] public bool
upLoop;
    [Tooltip("Flag to start a break in the simulation")] public bool stopSim;
    [Tooltip("Flag to start the end of the program")] public bool endProg;

        // Number of points and instructions
        private int nPoints;
        private int nMoves;

        // Position through the list of points and moves
        private int pPoints;
        private int pMoves;
```

Figura 104 - Communications. Variables de comportamiento.

Las variables de la *Figura 104* sirven como flags para controlar tanto el programa de *Unity* como el de *RobotStudio*. Su relación esta explicada en la sección [Referencia a arquitectura del sistema].

```
    [Header("Communication Variables")]
    private static string ip = "127.0.0.1";
    NetworkStream streamRobotStudio = default;
    [Tooltip("Boolean that tells if the program is connected to RobotStudio")]
public bool connected;
    [Tooltip("Boolean that tells if a message can be sent")] public bool canSend;
    [Tooltip("Boolean that tells if the program can end after robotstudio has
been stopped")] public bool canEnd;
    public string inMsg;
    public string outMsg;
    public float[] rAngles = null;

        // TCP variables
        private int pRobotStudioTCP = 5000;
        Thread threadRobotStudioTCPReceive = null;
        Thread threadRobotStudioTCPSend = null;
        TcpClient clientRobotStudioTCP = new TcpClient();

        // UDP variables
        private int pRobotStudioUDP = 6000;
        Thread threadRobotStudioUDP = null;
        UdpClient clientRobotStudioUDP;
```

Figura 105 - Communications. Variables de comunicación.

Las variables que se definen en la *Figura 105* son usadas en el sistema de comunicaciones. Se usan diferentes variables para el mensaje de entrada y salida del *Protocolo TCP/IP* y para el *Protocolo UDP/IP* porque se ha usado el sistema "multithreading" que permite ejecutar varios procesos de forma simultánea.

```
    [Header("Simulation Variables")]
    public RobotList robotList;
    public InstructionList instructionList;
    [Tooltip("State of the simulation")] public bool sim;
    [Tooltip("State of the simulation play in RobotStudio")] public bool inPlay;
```

Figura 106 - Communications. Variables de simulación.

Las variables de la *Figura 106* están relacionadas con el estado del programa en relación al proceso de simulación. La lista de robots e instrucciones corresponden a la de la rutina principal y contienen la información que se necesitará en la sincronización.

```
    [Header("Syncrhonization Variables")]
    [Tooltip("State of the synchronization with RobotStudio")] public bool sync;
    public bool robTargetSent;
    public bool instructionsSent;
    public bool initialPositionSent;
    [Tooltip("State of the simulation is in loop or not")] public bool inLoop;
```

Figura 107 - Communications. Variables de sincronización.

Las variables de la *Figura 107* corresponden con el estado de la sincronización y se utilizan en el desarrollo de ésta para saber qué bloque de información se ha sincronizado y cuál no.

```
    // UI Elements
    public Text playButtonText;
    public Image syncImage;
    public Image loopImage;
    public Image connectImage;
```

Figura 108 - Communications. Variables de interfaz.

Las variables de la *Figura 108* están relacionadas con la parte visual de la clase y sirven para informar al usuario del estado de la ejecución interna.

7.4.1.2 Métodos propios de Unity

```
    // Start is called before the first frame update
    void Start()
    {
        // Initialize compontents
        robotList = gameObject.GetComponent(typeof(RobotList)) as RobotList;
        instructionList = gameObject.GetComponent(typeof(InstructionList)) as
InstructionList;

        InitComms();

        // Create and start RobotStudio Receive TCP communications thread
        threadRobotStudioTCPReceive = new Thread(CommsRobotStudioTCPReceive);
        threadRobotStudioTCPReceive.Start();

        // Create and start RobotStudio Send TCP communications thread
        threadRobotStudioTCPSend = new Thread(CommsRobotStudioTCPSend);
        threadRobotStudioTCPSend.Start();

        // Create and start RobotStudio UDP communications thread
        threadRobotStudioUDP = new Thread(CommsRobotStudioUDP);
        threadRobotStudioUDP.Start();
    }
```

Figura 109 - Communications. Método Start.

En el método *Start* se inicializan las variables del sistema y se inician los distintos hilos de ejecución. Estos funcionan asignándoles un método concreto que deban ejecutar.

```
    void Update()
    {
        // Most of this tasks are here because they depend on the secondary
threads and uses functions only available in the main thread
        if(sync)
            syncImage.color = Color.red;
        else
            syncImage.color = Color.green;

        if (connected)
            connectImage.color = Color.green;
        else
            connectImage.color = Color.white;

        if(!inPlay)
            playButtonText.text = "Play";
        else
            playButtonText.text = "Pause";

        if(canEnd)
            ExitMenu.Quit();
    }
```

Figura 110 - Communications. Método Update.

El método *Update* es usado en este caso porque para realizar modificaciones sobre la escena es necesario encontrarse en el hilo principal de ejecución. No se pueden ejecutar estas

operaciones desde los procesos secundarios creados para las comunicaciones. Su funcionalidad consiste principalmente en actualizar la interfaz acorde al estado del sistema.

```
    // Activated if user stops Unity
    void OnApplicationQuit()
    {
        abort = true;
        // If the client or the stream are not null yet , close them
        if ((clientRobotStudioTCP != null || streamRobotStudio != null) &&
connected)
        {
            connected = false;
            streamRobotStudio.Close();
            clientRobotStudioTCP.Close();
        }
    }
```

Figura 111 - Communications. Método OnApplicationQuit.

El método "OnApplicationQuit" es llamada cuando se cierra la ejecución de *Unity*. Se usa este método para cerrar de forma controlada todos los procesos, incluida la comunicación con *RobotStudio*.

7.4.1.3 Protocolo UDP

Este módulo se encarga de la recepción de datos durante la simulación. Su trabajo es actualizar el vector de datos en el que se almacena la última posición recibida desde *RobotStudio*. Se ha diseñado con un bucle para que el usuario pueda decidir cuándo detener su ejecución [**4**].

```
    // UDP Comms
    void CommsRobotStudioUDP() // receives the real robot's joint angles from
RobotStudio
    {
        clientRobotStudioUDP = new UdpClient(pRobotStudioUDP);
        IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, pRobotStudioUDP);
        while (!abort)
        {
            try
            {
                if (sim & clientRobotStudioUDP.Available > 0)
                {
                    byte[] angles;
                    angles = clientRobotStudioUDP.Receive(ref remoteEP);

                    if (angles != null)
                        rAngles =
AnglesStrToFloat(System.Text.Encoding.ASCII.GetString(angles));
                }
            }
            catch (System.IO.IOException ex)
            {
                if(sim)
                    Debug.Log("ERROR 008: Unable to receive joint angles.  " +
ex.ToString());
            }
        }
        return;
    }
```

Figura 112 - Communications. Método CommsRobotStudioUDP.

7.4.1.4 Protocolo TCP

El protocolo empleado está dividido en dos bloques diferentes. Se ha diseñado de esta manera para separar la recepción y envío de datos.

7.4.1.4.1 Envió de datos

En la *Figura 113* se muestra el esquema general del método "CommsRobotStudioTCPSend".

```
    void CommsRobotStudioTCPSend()
    {
        while(!abort)
        {
            if (connected)
            {
                try // Sending data
                {
                    if (canSend)
                    {
                        …
                    }
                }

                catch (System.IO.IOException e)
                {
                    Debug.Log("ERROR 011: Can't send message to RobotStudio. " +
e.ToString());
                    connected = false;
                }

            }
        }
    }
```

Figura 113 - Communications. Método CommsRobotStudioTCPSend.

Primero comprueba que se haya realizado la conexión con *RobotStudio*. Después se intenta enviar información con la operación Try-Catch [**21**]. En caso de haberse activado el flag "canSend" se ejecutarán los protocolos de comunicación explicados a continuación.

```
                if (sync) // Syncrhonize robtargets and instructions
                {
                    if (!robTargetSent)
                    {
                        if (instructionList.targets.Count > 0)
                        {
                // Update outMsg to confirm that a robtarget is going to be sent
                outMsg = RobTargetToStr(instructionList.targets[pPoints]);
                            pPoints++;

                            if (pPoints == nPoints)
                                robTargetSent = true;

                            Debug.Log("Sending Point: " + outMsg);
                        } else {
                         Debug.Log("ERROR 009: There are no points created");
                            robTargetSent = true;
                            canSend = true;
                            outMsg = null;
                        }
                    } else if (!instructionsSent) {
                        if (instructionList.orders.Count > 0)
                        {
               // Update outMsg to confirm that a instruction is going to be sent
                outMsg = InstructionToStr(instructionList.orders[pMoves]);
                            pMoves++;

                            if (pMoves == nMoves)
                                instructionsSent = true;

                            Debug.Log("Sending Instruction: " + outMsg);
                        } else {
                            Debug.Log("ERROR 010: There are no orders");
                            instructionsSent = true;
                            canSend = true;
                            outMsg = null;
                        }
                    } else if (upLoop != inLoop) {
                        inLoop = upLoop;
                        outMsg = "L";
                    } else if (!initialPositionSent) {
                outMsg = JointTargetToStr(robotList.robots[robotList.actRobot]);
                        initialPositionSent = true;
                    } else {
                        sync = false;
                        outMsg = "S";
                    }
                }
```

Figura 114 - Communications. Bloque de sincronización.

En la *Figura 114* se muestra el proceso de sincronización del sistema. El programa se ejecutará varias veces recorriendo las listas de instrucciones y objetivos creados.

El orden de sincronización es el siguiente: *RobTarget* → *Instruction* → Modo de ejecución → Posición inicial del Robot.

```
                        if (sendSync)
                        {
                            sendSync = false;

                            // Data from the lists
                            pPoints = 0;
                            pMoves = 0;
                            nPoints = instructionList.targets.Count;
                            nMoves = instructionList.orders.Count;

                            outMsg = AddStrToMsg("S") + AddNumToMsg(nPoints) +
AddNumToMsg(nMoves);
                        }

                        if (stopSim)
                        {
                            stopSim = false;
                            outMsg = "S";
                        }

                        if (endProg)
                        {
                            endProg = false;
                            outMsg = "E";
                        }

                        if (upPlay)
                        {
                            upPlay = false;
                            outMsg = "P";
                        }

                        // Send Message
                        if (outMsg != null)
                        {
                            Debug.Log("Data Sent: " + outMsg);
                            byte[] outStream =
System.Text.Encoding.ASCII.GetBytes(outMsg);
                            streamRobotStudio.Write(outStream, 0,
outStream.Length);
                            streamRobotStudio.Flush();

                            canSend = false;
                        }
```

Figura 115 - Communications. Envío de datos.

En la *Figura 115* aparecen varios bloques if. Cada uno de ellos se activa con una variable diferente que actúa como flag.

- Variable “sendSync”. Variable que inicia el proceso de sincronización mandando a *RobotStudio* el número de puntos e instrucciones a sincronizar. Mensaje de salida: “[S;nPoints;nMoves]”.

- Variable "stopSim". Variable que inicia la finalización de la simulación. Mensaje de salida: "S".
- Variable "endProg". Variable que inicia la finalización el programa. Mensaje de salida: "E".
- Variable "upPlay". Variable que inicia y pausa la simulación. Mensaje de salida: "P".

En caso de habérsele dado un valor al mensaje de salida se mandará dicho mensaje por el puerto de comunicaciones.

7.4.1.4.2 Recepción de datos

En la *Figura 116* se muestra el esquema general del método "CommsRobotStudioTCPSend".

```
    void CommsRobotStudioTCPReceive()
    {
        while (!abort)
        {
            if (connected)
            {
                streamRobotStudio = clientRobotStudioTCP.GetStream();

                try // Try receiving
                {
                    …

                    Thread.Sleep(100);
                }

                catch (System.IO.IOException e)
                {
                    Debug.Log("ERROR 012: Not receiving message from RobotStudio.
" + e.ToString());
                    connected = false;
                }
            }
        }
    }
```

Figura 116 - Communications. Método CommsRobotStudioTCPReceive.

En cada ejecución primero se comprueba que se haya realizado la conexión con *RobotStudio*. A continuación, se almacena la información en bruto que haya en el puerto. Después se intenta recibir el mensaje con la operación Try-Catch [**21**].

Este proceso se pausa durante 100 milisegundos para que dé tiempo al resto del sistema a realizar las operaciones que se deriven.

```
var buffsize = clientRobotStudioTCP.ReceiveBufferSize;
byte[] inStream = new byte[buffsize];

streamRobotStudio.Read(inStream, 0, buffsize);
inMsg = System.Text.Encoding.ASCII.GetString(inStream);

Debug.Log("Data Received: " + inMsg);
if (!inMsg.Contains("ERROR"))
    inMsg = inMsg[0].ToString();

switch (inMsg)
{
    case "R": // Starts syncronization
        InitSync();
        sync = true;
        canSend = true;
        break;

    case "C": // Confirms that it can send again
        canSend = true;
        break;

    case "S": // Starts the simulation
        sim = true;
        inPlay = true;
        break;

    case "P": // Play/Pauses the simulation
        inPlay = !inPlay;
        break;

    case "E": // Ends the simulation
        sim = false;
        inPlay = false;
        break;

    case "Y":
        Debug.Log("Finalizing...");
        canEnd = true;
        break;

    default:
        Debug.Log(inMsg);
        break;
}
```

Figura 117 - Communications. Recepción de datos.

El mensaje recibido se procesa y convierte en una cadena de caracteres. En caso de que el mensaje no contenga “ERROR” se modificara el mensaje para que solo contenga el primer carácter almacenado en él. Después se entrará en un “switch case”.

- En caso de recibir "R" se inicia la sincronización activando la variable "Sync".
- En caso de recibir "C" el sistema se encontraba en proceso de sincronización y *RobotStudio* avisa de que está listo para recibir el siguiente mensaje.
- En caso de recibir "S" *RobotStudio* está avisando de que la simulación se ha iniciado.
- En caso de recibir "P" RobotStudio está avisando de que la simulación está cambiando entre pausa y reproducción y viceversa.
- En caso de recibir "E" *RobotStudio* está avisando de que la simulación se ha detenido.
- En caso de recibir "Y" se puede iniciar el proceso de finalización del programa.

7.4.1.5 Métodos utilizados por la interfaz

En esta sección se explican aquellos métodos que se vinculan a la interfaz para que el usuario pueda ejecutarlos cuando crea conveniente. Aquellos que conlleven mandar un mensaje a *RobotStudio* activarán la variable flag "canSend".

```
public void Connect()
{
    try
    {
        clientRobotStudioTCP.Connect(ip, pRobotStudioTCP);
        connected = true;
    }
    catch (System.Net.Sockets.SocketException e)
    {
        Debug.Log("ERROR 013: " + e.ToString());
    }
}
```

Figura 118 - Communications. Método Connect.

El método "Connect" que aparece en la *Figura 118* inicia la conexión con *RobotStudio*.

```
    public void Sync() // Function that uses the interface to sync with
RobotStudio
    {
        if(!connected)
        {
            Debug.Log("ERROR 014: Not connected to Robotstudio");
            return;
        }

        if (!sim & !sync)
        {
            // Flag to send data
            sendSync = true;
            canSend = true;
        }
    }
```

Figura 119 - Communications. Método Sync.

El método "Sync" que aparece en la *Figura 119* permite al usuario activar la variable flag "sendSync" para comenzar el proceso de sincronización. Pero antes de ello se comprueba si el sistema está conectado a *RobotStudio*.

```
    public void Sim()
    {
        if (!sync)
        {
            upPlay = !upPlay;
            canSend = true;
        }
    }
```

Figura 120 - Communications. Método Sim.

El método "Sim" que aparece en la *Figura 120* activa la variable flag "upPlay" para iniciar o pausar la simulación. Esta acción solo se produce si el programa no se está sincronizando.

```
    public void StopSim()
    {
        if (sim)
        {
            stopSim = true;
            canSend = true;
        }
    }
```

Figura 121 - Communications. Método StopSim.

El método "StopSim" que aparece en la *Figura 121* activa la variable flag "stopSim", pero solo si la simulación se está ejecutando.

```
public void End()
{
    if (!sync && !sim)
    {
        endProg = true;
        canSend = true;
    }
}
```

Figura 122 - Communications. Método End.

El método "End" que aparece en la *Figura 122* activa la variable flag "endProg", pero solo si el programa no está simulando ni se está sincronizando con *RobotStudio*.

```
public void Loop()
{
    upLoop = !upLoop;
    if(upLoop)
        loopImage.color = Color.green;
    else
        loopImage.color = Color.red;
}
```

Figura 123 - Communications. Método Loop.

El método "Loop" que aparece en la *Figura 123* cambia el estado de la variable "upLoop" que indica si la simulación se encuentra en bucle o no.

7.4.1.6 Métodos para el funcionamiento interno de la clase

```
private void InitComms()
{
    // Initialize all the boolean variables as expected
    // Behaviour Variables
    abort = false;
    sendSync = false;
    upPlay = false;
    upLoop = false;
    stopSim = false;
    endProg = false;

    // Communication Variables
    connected = false;
    canSend = false;
    canEnd = false;

    // Simulation Variables
    sim = false;
    inPlay = false;

    // Synchronization Variables
    sync = false;
    robTargetSent = false;
    instructionsSent = false;
    initialPositionSent = false;
    inLoop = false;
}
```

Figura 124 - Communications. Método InitComms.

El método "InitComms" de la *Figura 124* sirve para inicializar todas las variables booleanas usadas en esta clase.

```
private void InitSync()
{
    robTargetSent = false;
    instructionsSent = false;
    initialPositionSent = false;
}
```

Figura 125 - Communications. Método InitSync.

El método "InitSync" de la *Figura 125* reinicia las variables de sincronización.

```
    private float[] AnglesStrToFloat(string a) // 28.692;-20.938;17.530;-
11.391;34.141;13.159
    {
        string[] result;
        result = a.Split(';');

        float[] jAngles = new float[result.Length];
        for (int i = 0; i < result.Length; i++)
        {
            jAngles[i] = -float.Parse(result[i], NumberStyles.Float,
CultureInfo.InvariantCulture);
        }

        return jAngles;
    }
```

Figura 126 - Communications. Método AnglesStrToFloat.

El método "AnglesStrToFloat" de la *Figura 126* convierte la cadena de caracteres recibida desde *RobotStudio* en un vector de floats para poder ser usados por la rutina principal.

Es importante tener en cuenta la configuración de los datos representados en caracteres porque si no, no se convierten correctamente en datos numéricos para el sistema.

```
    private string AddStrToMsg(string s) { return (s + ";"); }

    private string AddNumToMsg(float n) { return (n.ToString("0.##",
CultureInfo.InvariantCulture) + ";"); }
```

Figura 127 - Communications. Métodos de conversión a cadena de caracteres.

Los métodos de la *Figura 127* sirven para añadir datos básicos a los mensajes de salida siguiendo el protocolo de comunicaciones definido en la sección *5.2*.

```
    private string RobTargetToStr(RobTarget r)
    {
        Vector3 pos = r.GetRSPosition();
        Vector3 ang = r.GetRSRotation();

        string msgRT = "T[";

        for (int i = 0; i < 3; i++)
        {
            msgRT += AddNumToMsg(pos[i]);
        }

        for (int i = 0; i < 3; i++)
        {
            msgRT += AddNumToMsg(ang[i]);
        }

        for (int i = 0; i < 4; i++)
        {
            msgRT += AddNumToMsg(r.GetRSConfig(i));
        }

        msgRT += "]";

        return msgRT;
    }
```

Figura 128 - Communications. Método RobTargetToStr.

El método "RobTargetToStr" de la *Figura 128* convierte una variable de la clase *RobTarget* en una cadena de caracteres que se enviará a *RobotStudio*. El mensaje resultante tiene la siguiente apariencia: *T[num;num;num;num;num;num;num;num;num;num]*.

Los tres primeros elementos corresponden con la posición, los tres siguientes con la rotación y los cuatro últimos con la configuración del *RobTarget*.

```
    private string InstructionToStr(Instruction i)
    {
        string msgI = "M[";

        msgI += AddStrToMsg(i.GetMoveType());
        msgI += AddNumToMsg(i.GetSpeed());
        msgI += AddNumToMsg(i.GetPrecision());

        msgI += "]";

        return msgI;
    }
```

Figura 129 - Communications. Método InstructionToStr.

El método "InstructionToStr" de la *Figura 129* convierte la información de una variable *Instruction* en una cadena de caracteres. El mensaje resultante quedaría tal que así:

M[letter;num;num]. El mensaje almacena por orden: tipo de movimiento, velocidad y precisión.

```
    private string JointTargetToStr(GenericRobot r)
    {
        string msgJ = "A[";

        for (int i = 0; i< r.GetNumJoints();i++)
        {
            msgJ += (-r.joints[i].GetAngle()).ToString("0.##",
CultureInfo.InvariantCulture);

            if (i == r.GetNumJoints()-1)
                msgJ += "]";
            else
                msgJ += ",";
        }

        return msgJ;
    }
```

Figura 130 - Communications. Método JointTargetToStr.

El método "JointTargetToStr" de la *Figura 130* sirve para convertir un vector con los ángulos de las articulaciones del robot activo en una cadena de caracteres que *RobotStudio* pueda procesar después. Por ello además de usar de configuración la cultura de referencia invariante, que corresponde al inglés, se ha usado "0.##" como entrada para el método "ToString", de esta forma el número convertido tendrá dos decimales como máximo y cuando los tenga el símbolo que lo marcará será un punto.

7.5 INTERACCIÓN CON EL USUARIO

Para dar al usuario libertad por el entorno del programa y poder usar las funcionalidades que se han programado, se han creado dos bloques de programa diferentes:

- Controlador de usuario en primera persona.
- Interfaz de usuario superpuesta a la escena.

En esta sección se explicarán ambos bloques.

7.5.1 OBJETO DEL USUARIO EN PRIMERA PERSONA

Este objeto este compuesto por una cápsula y un objeto de cámara. La jerarquía se encuentra en la *Figura 131*.

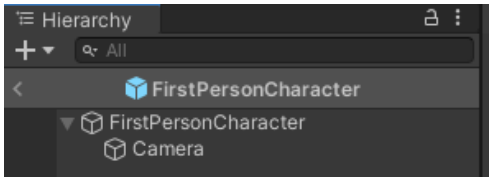

Figura 131 - Jerarquía del objeto del usuario.

Con esto se ha dotado al usuario de apariencia visual básica y se muestra en la *Figura 132*.

Figura 132 - Objeto del usuario.

Los componentes que se le han añadido al objeto se muestran en la *Figura 133.*

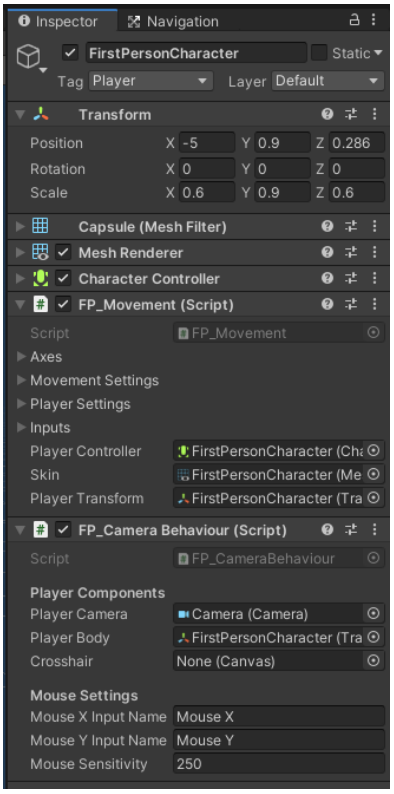

Figura 133 - Componentes añadidos al objeto del usuario.

La clase "CharacterController" [**22**] viene de serie en la librería de *Unity* y facilita al programador la integración de un usuario en el programa. El único inconveniente de usar este componente es que no se podrá definir al usuario como *RigidBody*. Se ha hecho así porque el componente "CharacterController" tiene un sistema de movimiento avanzado que le permite subir y bajar escaleras y limitar el ángulo máximo que puede tener una rampa para poder subirla.

Por otro lado las clases *FP_CameraBehaviour* y *FP_Movement* se han creado específicamente para el programa y se explican en las secciones a continuación.

7.5.1.1 Movimiento de la cámara

El movimiento de la cámara está controlado por la clase *FP_CameraBehaviour*.

```
    // Functional components
    [Header("Player Components")]
    public Camera playerCamera;
    public Transform playerBody;
    public Canvas crosshair;

    // Functional variables
    [Header("Mouse Settings")]
    [SerializeField] private string mouseXInputName = "Mouse X";
    [SerializeField] private string mouseYInputName = "Mouse Y";
    [SerializeField] private float mouseSensitivity = 250f;
    private float pitch;
```

Figura 134 - FP_CameraBehaviour. Variables.

Para el funcionamiento de la clase se han definido un objeto cámara, una variable tipo *Transform* que almacena la posición del usuario y un elemento de interfaz para usar como puntero.

Para la configuración del ratón se configuran las entradas predefinidas de *Unity* y un valor que sirve como modificador de la sensibilidad del ratón.

```
    private void LockCursor()
    {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
        crosshair.enabled = true;
    }
```

Figura 135 - FP_CameraBehaviour. Método LockCursor.

El método “LockCursor” sirve para ocultar el cursor a la vez que se muestra una especie de mirilla que se superpone a la escena como la interfaz.

```
    private void UnlockLockCursor()
    {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
        crosshair.enabled = false;
    }
```

Figura 136 - FP_CameraBehaviour. Método UnlockCursor.

El método "UnLockCursor" realiza la operación contraria al método "LockCursor" permitiendo al usuario usar el ratón dentro del programa para poder interactuar con la interfaz.

```
    private void CameraRotation()
    {
        float mouseX = Input.GetAxis(mouseXInputName) * mouseSensitivity *
Time.deltaTime;
        float mouseY = Input.GetAxis(mouseYInputName) * mouseSensitivity *
Time.deltaTime;

        pitch -= mouseY;

        pitch = Mathf.Clamp(pitch, -90f, 90f);

        playerCamera.transform.localRotation = Quaternion.Euler(pitch, 0f, 0f);
        playerBody.transform.Rotate(Vector3.up * mouseX);
    }
```

Figura 137 - FP_CameraBehaviour. Método CameraRotation.

El método "CameraRotation" es el que permite a la cámara rotar en la escena según el movimiento del ratón. El método limita el movimiento de la cámara en el eje vertical a ±90 grados evitando que se invierta.

```
    // Update is called once per frame
    void Update()
    {
        if (Input.GetKey(KeyCode.Mouse1) && !crosshair.enabled)
            LockCursor();
        else if (!Input.GetKey(KeyCode.Mouse1) && crosshair.enabled)
            UnlockLockCursor();

        if (crosshair.enabled)
            CameraRotation();
    }
```

Figura 138 - FP_CameraBehaviour. Método Update.

En el método *Update* de esta clase es donde se llama a las funciones definidas y explicadas anteriormente. Se llamará al método "LockCursor" al pulsarse el botón derecho del ratón y al método "UnlockCursor" al soltarlo. También se ha diseñado para que el método "CameraRotation" solo se ejecute cuando la mirilla este activa. De esta forma el usuario no rotará la cámara cuando trate de interactuar con la interfaz.

7.5.1.2 Movimiento del usuario por la escena

El movimiento del usuario está controlado por la clase *FP_Movement.*

Para una mejor organización se han creado pequeñas subclases que almacenan los diferentes parámetros. Se han diseñado dos tipos de movimientos diferentes para un mayor abanico de posibilidades.

- El primer modo de movimiento se asemeja al de una persona real, con movimiento a ras de suelo, saltos, colisiones, subida y bajada de escaleras y rampas y gravedad.
- El segundo modo de movimiento proporciona una mayor libertad al usuario. Movimiento en los tres ejes del espacio, ausencia de gravedad y colisiones. Se ha denominado modo fantasma para referirse a él.

Ambos modos cuentan con dos niveles de velocidad, pero el segundo puede regularla a conveniencia con la rueda del ratón.

```
[Serializable]
public class Axes
{
    [Tooltip("X axis")] public string abscissaInputName = "X";
    [Tooltip("Y axis")] public string ordinateInputName = "Y";
    [Tooltip("Z axis")] public string applicateInputName = "Z";
}
```

Figura 139 - FP_Movement. Ejes del entorno.

La clase “Axes” definida en la *Figura 139* contiene el nombre de los elementos de entrada para el desplazamiento definidos en la sección 7.1.3. Las variables “X” y “Z” en ambos modos de movimiento para poder detectar si el usuario pulsa “W”, “A”, “S” o “D”, que son las teclas convencionales para moverse por entornos 3D. En modo fantasma el usuario podrá usar el espacio para subir verticalmente y “Left Control” para bajar.

```
[Serializable]
public class MovementSettings
{
    [Header("Speed Settings")]
    public float walkSpeed = 5;
    public float runSpeed = 10;
    public float ghostSpeed = 8;
    [Tooltip("Pow factor for ghost mode")] public float spdBoost;

    [Tooltip("Real speed applied to the player")] public float speed;
    [Tooltip("Vertical velocity of the player")] public float yVelocity;
    [Tooltip("Direction the player is moving to")] public Vector3 movement;

    [Header("PhysicSettings")]
    public float gravity = -9.8f;
    public LayerMask groundLayer;
}
```

Figura 140 - FP_Movement. MovementSettings.

La clase "MovementSettings" definida en la *Figura 140* contiene las variables relacionadas con el movimiento del usuario.

- Diferentes niveles de velocidad.
- Modificador de velocidad.
- Fuerza de la gravedad.
- *Layer* reconocida como suelo.

Además de otras variables que informan de la dirección de movimiento del objeto.

```
[Serializable]
public class PlayerSettings
{
    public bool ghostmode = false;
    public bool isGrounded = false;
    public float height = 0.9f;
    public float jumpHeight = 1.25f;
    public float groundSphereCastDistance = 1;
}
```

Figura 141 - FP_Movement. PlayerSettings.

La clase "PlayerSettings" de la *Figura 141* almacena las variables referentes al estado del objeto del usuario. Con estas sabemos si el objeto usa un tipo de movimiento u otro, si se encuentra en el suelo o volando, la altura del objeto y la altura máxima de salto.

```
[Serializable]
public class InputSettings
{
    [Header("KeyBindings")]
    public KeyCode jumpKey = KeyCode.Space;
    public KeyCode modeKey = KeyCode.LeftAlt;
    public KeyCode runKey = KeyCode.LeftShift;

    [Header("Behaviour variables")]
    public float xAxis;
    public float yAxis;
    public float zAxis;
    public float yScroll;
    public bool run;
    public bool jump;
    public bool changeMode;
}
```

Figura 142 - FP_Movement. InputSettings.

La clase "InputSettings" de la *Figura 142* tiene definidas las entradas por teclado que podrá usar el usuario, así como variables de comportamiento que sirven para gestionar los distintos parámetros. Tal y como está configurado, el usuario podrá cambiar de modo al pulsar "Alt", saltar al pulsar espacio y correr al pulsar "Left Shift".

```
// Creating parameter variables
public Axes axes = new Axes();
public MovementSettings movementSettings = new MovementSettings();
public PlayerSettings playerSettings = new PlayerSettings();
public InputSettings inputs = new InputSettings();
public CharacterController playerController;
public MeshRenderer skin;
public Transform playerTransform;
public RaycastHit hitInfo;
```

Figura 143 - FP_Movement. Variables.

En la *Figura 143* se definen las variables que usará el programa. Además de las clases que se han definido antes, se van a usar:

- Una variable que representa la malla visual del objeto.
- Una variable que almacena la posición del usuario.
- Una variable que se usa para guardar la información obtenida al realizar un *RayCast.*

```
// Awake method is called once at the start
void Awake()
{
    playerController = GetComponent<CharacterController>();
    playerTransform = GetComponent<Transform>();
    skin = GetComponent<MeshRenderer>();
}
```

Figura 144 - FP_Movement. Método Awake.

Se usa el método *Awake* para inicializar las variables que corresponden a los componentes del objeto.

```
// Update is called once per frame
void Update()
{
    Inputs();
    GameModeSelector();

    if (playerSettings.ghostmode)
        GhostMove();
    else
    {
        CheckisGrounded();
        CalcMoveDirection();
        Jump();
        ApplyGravity();
        Move();
    }
}
```

Figura 145 - FP_Movement. Método Update.

En el método *Update* se ejecutan por orden los métodos definidos más adelante. Lo primero que se hace es actualizar las variables de entrada y cambiar de modo de movimiento si así se ha solicitado con los métodos “Inputs” y “GameModeSelector”. Después, dependiendo de si se encuentra en modo fantasma o no, se ejecutan distintos métodos.

En modo fantasma se ejecuta únicamente el método “GhostMove” mientras que si se encuentra en el modo normal se ejecutan por orden los métodos “CheckisGrounded”, “CalcMoveDirection”, “Jump”, “ApplyGravity” y “Move”. Se hace de esta manera para asegurar su correcto funcionamiento y que los métodos no se pisen entre ellos.

```
    // Update the inputs of the player
    private void Inputs()
    {
        // Axes input
        inputs.xAxis = Input.GetAxisRaw(axes.abscissaInputName);
        inputs.yAxis = Input.GetAxisRaw(axes.ordinateInputName);
        inputs.zAxis = Input.GetAxisRaw(axes.applicateInputName);

        // Keyboard input
        inputs.run = Input.GetKey(inputs.runKey);
        inputs.jump = Input.GetKeyDown(inputs.jumpKey);
        inputs.changeMode = Input.GetKeyDown(inputs.modeKey);

        // Mouse input
        inputs.yScroll = Input.mouseScrollDelta.y;
    }
```

Figura 146 - FP_Movement. Método Inputs.

El método "Inputs" que aparece en la *Figura 146* se encarga de actualizar las variables de entrada en cada fotograma.

```
    private void GameMode()
    {
        if (playerSettings.ghostmode)
        {
            playerSettings.ghostmode = false;
            playerController.enabled = true;
            skin.enabled = true;
        }
        else
        {
            playerSettings.ghostmode = true;
            playerController.enabled = false;
            movementSettings.yVelocity = 0;
            skin.enabled = false;
        }
    }
```

Figura 147 - FP_Movement. Método GameMode.

El método "GameMode" de la *Figura 147* se encarga de cambiar los parámetros que definen al usuario para cambiar de modo fantasma a normal. En modo fantasma se desactiva el componente "CharacterController" para desactivar las colisiones, se desactiva la malla de renderizado visual y se cancela el movimiento vertical que tuviera antes el usuario. Estos cambios se invierten al pasar a modo normal.

```
    private void GameModeSelector()
    {
        if (inputs.changeMode)
            GameMode();

        if (!playerSettings.ghostmode)
        {
            if (inputs.jump && !playerSettings.isGrounded)
                GameMode();
        }
    }
```

Figura 148 - FP_Movement. Método GameModeSelector.

El método "GameModeSelector" se usa para gestionar la llamada al método "GameMode" según las variables de entrada. Se ha configurado para que el usuario pueda cambiar a modo fantasma pulsando "Alt" como se dijo antes y también pulsando espacio en el aire.

```
    // Check if the player is grounded method
    private void CheckisGrounded()
    {
        // Sphere cast position
        Vector3 spherePosition = playerTransform.position + new Vector3(0, -
playerSettings.height + 0.3f, 0);

        playerSettings.isGrounded = Physics.SphereCast(spherePosition, 0.3f,
Vector3.down, out hitInfo, playerSettings.groundSphereCastDistance,
movementSettings.groundLayer);
    }
```

Figura 149 - FP_Movement. Método CheckisGrounded.

El método "CheckisGrounded" es usado para detectar si el usuario se encuentra sobre el suelo usando un método parecido al *RayCast* pero más fiable porque en vez de usar una recta usa una esfera para detectar colisiones.

```
    // Calculate move direction
    private void CalcMoveDirection()
    {
        if (playerSettings.isGrounded)
        {
            // Raycast
            if (Physics.Raycast(playerTransform.position, Vector3.down, out
hitInfo, Mathf.Infinity))
            {
                // Real movement direction
                Vector3 forward = Vector3.Cross(hitInfo.normal,
playerTransform.right);
                Vector3 right = Vector3.Cross(playerTransform.forward,
hitInfo.normal);

                // Saving movement direction
                movementSettings.movement = (-forward * inputs.zAxis + -right *
inputs.xAxis);
            }
            else
                movementSettings.movement = playerTransform.forward *
inputs.zAxis + playerTransform.right * inputs.xAxis;
        }
        else
            movementSettings.movement = playerTransform.forward * inputs.zAxis +
playerTransform.right * inputs.xAxis;
    }
```

Figura 150 - FP_Movement. Método CalcMoveDirection.

El método "CalcMoveDirection" se usa en el modo normal para calcular la dirección en la que se debe mover el usuario al pulsar alguna de las teclas "wasd". Se ha hecho así para que pueda moverse sin problemas por rampas.

```
    // Jump method
    private void Jump()
    {
        if (inputs.jump)
        {
            if (playerSettings.isGrounded)
            {
                movementSettings.yVelocity +=
Mathf.Sqrt(playerSettings.jumpHeight * 2f * -movementSettings.gravity);
                playerSettings.isGrounded = false;
            }
        }
    }
```

Figura 151 - FP_Movement. Método Jump.

El método "Jump" de la *Figura 151* permite al usuario saltar una altura concreta definida en sus parámetros aplicándole una velocidad vertical al objeto del usuario.

```
    // Apply gravity method
    private void ApplyGravity()
    {
        if (playerSettings.isGrounded && movementSettings.yVelocity < 0)
            movementSettings.yVelocity = movementSettings.gravity *
Time.deltaTime;
        else
            movementSettings.yVelocity += movementSettings.gravity *
Time.deltaTime;
    }
```

Figura 152 - FP_Movement. Método ApplyGravity.

El método "ApplyGravity" aplica gravedad al objeto del usuario.

```
    // Defines the normal movement of the player
    private void Move()
    {
        // Speed modifier
        if (Input.GetKey(inputs.runKey))
            movementSettings.speed = movementSettings.runSpeed;
        else
            movementSettings.speed = movementSettings.walkSpeed;

        // Ceiling collide detection
        if ((playerController.collisionFlags & CollisionFlags.Above) != 0)
            movementSettings.yVelocity = 0;

        // Simple Move
        playerController.Move((movementSettings.movement.normalized *
movementSettings.speed + Vector3.up * movementSettings.yVelocity) *
Time.deltaTime);
    }
```

Figura 153 - FP_Movement. Método Move.

El método "Move" define el movimiento del usuario en modo normal en función de todos los parámetros calculados anteriormente en el código y usando el método "Move" definido en la clase "CharacterController".

```
    // Ghost move
    private void GhostMove()
    {
        // Ghost movement
        movementSettings.movement = playerTransform.forward * inputs.zAxis +
playerTransform.up * inputs.yAxis + playerTransform.right * inputs.xAxis;

        // Speed modifier (exponential based)
        movementSettings.spdBoost += Input.mouseScrollDelta.y * 0.1f;
        movementSettings.spdBoost = Mathf.Clamp(movementSettings.spdBoost, -10,
2);

        movementSettings.speed = movementSettings.ghostSpeed * Mathf.Pow(2.0f,
movementSettings.spdBoost);

        // Simple Move
        playerController.transform.position +=
movementSettings.movement.normalized * movementSettings.speed * Time.deltaTime;
    }
```

Figura 154 - FP_Movement. Método GhostMove.

El método "GhostMove" define el movimiento del usuario en modo fantasma en función de todos los parámetros calculados anteriormente en el código actualizando la posición del objeto en cada fotograma.

7.5.2 INTERFAZ

Para desarrollar la interfaz del sistema se han usado principalmente los elementos incluidos en la librería de *Unity* [**23**]. Algunos de estos elementos son:

- *Canvas.* Es el elemento más importante para una interfaz puesto que es el que se encarga de renderizarlo. También necesitarán un componente "Graphic Raycast" en caso de ser interactuables.
- "Button". Es un objeto que se comportan como un botón. La interacción con este objeto genera un evento [**24**].
- "Dropdown". Es un objeto con el que el usuario puede seleccionar una opción entre un listado preconfigurado. La interacción con este objeto generará un evento al seleccionar una de las opciones [**25**].
- "Slider". Es un objeto que permite regular su valor al desplazarlo por la pantalla [**26**].

Todos los menús desarrollados se encuentran dentro del objeto "UI". Se puede ver la jerarquía de la interfaz en la *Figura 155*.

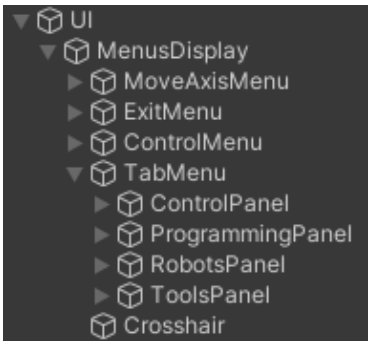

Figura 155 - Jerarquía de la interfaz.

Antes de explicar los diferentes menús creados en detalle es importante dejar claro algunos conceptos que se explicarán en las dos secciones a continuación y hacen referencia al sistema de eventos y a cómo optimizar una interfaz.

7.5.2.1 Optimización de una interfaz

Como se ha explicado, todos los objetos de la interfaz deben estar contener o ser hijos de un objeto con un componente *Canvas.* El problema radica en que si la jerarquía de objetos se vuelve muy compleja y el número de objetos crece se puede llegar a afectar significativamente al rendimiento.

- Un objeto con un componente *Canvas* se actualizará si el objeto que lo tiene o alguno de sus hijos cambia. Una forma de solucionar esto es añadiendo más componentes tipo *Canvas* en los hijos. Al hacerlo solo se actualizará la zona afectada sin afectar al resto de la interfaz
- En vez de activar o desactivar el objeto que contiene el componente *Canvas* para hacer desaparecer la interfaz de la pantalla es mejor desactivar únicamente el componente *Canvas.*
- Desactivar el componente "Graphic Raycast" de todos los objetos de tipo texto. Como no son interactuables no lo necesitan, pero consume recursos.
- Desactivar "Pixel Perfect" para los *Canvas* en elementos movibles de interfaz.

7.5.2.2 Eventos

Normalmente se usaría únicamente el sistema de eventos prediseñado de *Unity*, que está basado en eventos tipo "OnClick", "OnValueChange", etc.

Los objetos tipo botón por ejemplo tienen eventos tipo "OnClick" lo que implica que al pulsarlos crearan un evento y llamará a los métodos que se le hayan configurado en el inspector.

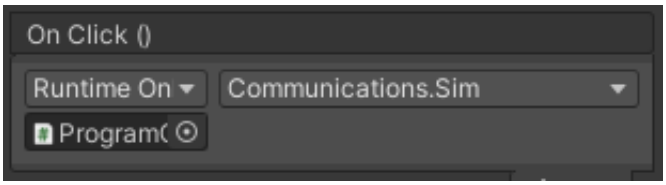

Figura 156 - Ejemplo de configuración de eventos.

En la *Figura 156* se muestra un ejemplo de configuración que corresponde al botón que "Play" del *Menú de control*. Cuando se pulse este botón se llamará al método "Sim" del código *Communications*.

Sin embargo, esto solo nos sirve si los elementos interactuables ya se encuentran en escena. Si queremos crearlos a través de código tenemos que crear un sistema de eventos que aprovecha el sistema de *Unity* pero le da una vuelta de tuerca. [**27**]

Los eventos de .NET o Csharp se basan en el modelo de delegado. El modelo de delegado sigue el patrón de diseño del observador [**28**], que permite que un suscriptor se registre con un proveedor y reciba notificaciones de él. El emisor de un evento inserta una notificación de que se ha producido un evento, y un receptor de eventos recibe la notificación y define una respuesta a la misma. [**29**]

Los receptores tienen que subscribirse al evento al iniciarse su código y desubscribirse al finalizar su ejecución. Esto se hace suscribiéndose durante el método *Start* y desubscribiendose durante el método *OnDestroy* tal y como se verá más adelante al definirse diferentes clases.

7.5.2.3 Menú de salida

La parte funcional del menú de salida consta únicamente de dos botones. El botón "Exit" llamará al método "End" de la clase *Communications* y el botón "Connect" llamará al método "Connect" de la misma clase. Ambos métodos se han configurado como se muestra en el ejemplo de la *Figura 156.*

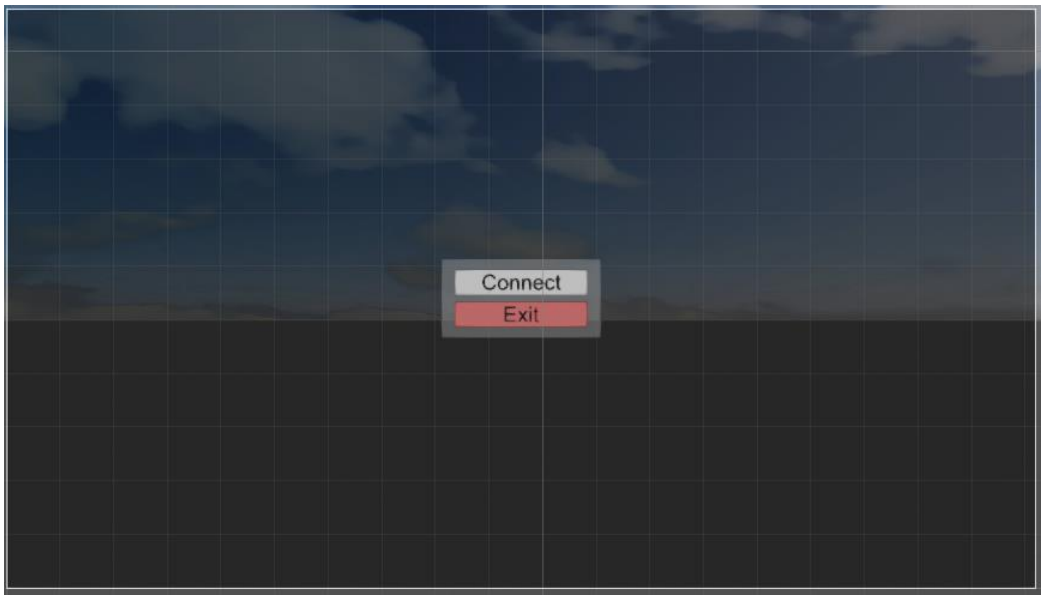

Figura 157 - Menú de salida.

7.5.2.3.1 Clase ExitMenu

El menú de salida usa para su funcionamiento la clase *ExitMenu.*

```
// Start is called before the first frame update
void Start()
{
    exitMenu.enabled = false;
}

// Update is called once per frame
void Update()
{
    // Exit Sample
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if (exitMenu.enabled)
            exitMenu.enabled = false;
        else
            exitMenu.enabled = true;
    }
}
```

Figura 158 - ExitMenu. Métodos de Unity.

Al iniciarse el programa se desactiva el *Canvas* del menú de salida. En cada fotograma se comprueba si el usuario pulsa "Escape" porque es el botón que se ha configurado para activar y desactivar este menú.

```
// Quit function
public static void Quit()
{
    Application.Quit();
    #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
    #endif
}
```

Figura 159 - ExitMenu. Método Quit.

El método estático "Quit" se ha definido en esta clase por tratarse del menú de salida, pero al ser estático se podría haber definido en otra clase. Es el método que se encarga de finalizar el programa definitivamente.

7.5.2.4 Menú de movimiento de ejes

Este menú es el que usará el usuario para mover las diferentes articulaciones del brazo robótico. En la *Figura 160* se muestra un ejemplo de este menú al usarse el *ABB IRB120*

Figura 160 - Menú de movimiento de ejes.

Los componentes empleados para alinear los elementos son "Content Size Fitter" y "Vertical Layout Group" [**20**].

Uno de los elementos más importantes de este menú es el *Prefab* que se muestra en la *Figura 161*.

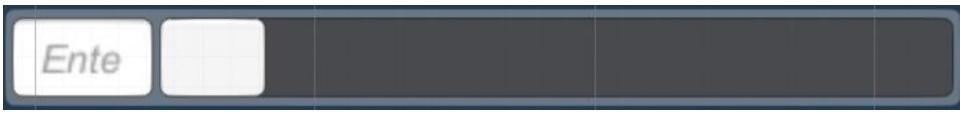

Figura 161 - Prefab de articulación.

A este elemento se le ha añadido el componente *CJointDisplay* y permite actualizar al ángulo de las articulaciones del robot a través del "Slider" o introduciéndolo por texto. En caso de introducirlo por texto habrá que usar la coma como símbolo decimal.

7.5.2.4.1 Clase CJointDisplay

La clase *CJointDisplay* representa visualmente una articulación del robot. El problema que se presentó al crear esta clase es la gestión de los ángulos máximos y mínimos (y sus signos) para representar la misma funcionalidad que en *RobotStudio* a pesar de tener un sistema de coordenadas diferente.

```
    // Public variables
    public CJoint joint;

    // Display variables
    public Text minAngleDisplay;
    public Text maxAngleDisplay;
    public InputField inputField;

    public Slider slider;

    // Delegate functionality
    public delegate void CjointDisplayDelegate(CJoint joint);
    public static event CjointDisplayDelegate OnValueChange;
```

Figura 162 - CJointDisplay. Variables.

Las variables que contiene esta clase son:

- Una variable de la clase *CJoint* que es la que tiene que mostrar en pantalla.
- Elementos de texto para mostrar los ángulos máximo y mínimo
- Un elemento de entrada de texto y un "Slider" para cambiar el ángulo de la articulación.
- También se ha creado el evento "OnValueChange" al que se llamará cada vez que cambie el valor de la articulación desde la interfaz.

```
    // Initializer method
    public void Prime(CJoint joint)
    {
        this.joint = joint;

        // The angle that is shown to the user is fixed to have the same momevent
that it will have in RobotStudio
        // MinMax angles
        minAngleDisplay.text = (-joint.maxAngle).ToString("F2");
        maxAngleDisplay.text = (-joint.minAngle).ToString("F2");

        slider.minValue = -joint.maxAngle;
        slider.maxValue = -joint.minAngle;

        // Real angle
        slider.value = -joint.angle;
        if (inputField != null)
            inputField.text = (-joint.angle).ToString("F2");
    }
```

Figura 163 - CJointDisplay. Método Prime.

El método "Prime" de la *Figura 163* inicializa el *Prefab* que se muestra en la *Figura 161*.

Como se explica en la figura, se ha ajustado la inicialización para que para el usuario el robot se mueva en el mismo sentido en el que lo haría en *RobotStudio*.

```
    public void ChangeAngle(float value)
    {
        if (inputField != null)
            inputField.text = value.ToString("F2");

        joint.SetAngle(-value);

        if (OnValueChange != null)
            OnValueChange.Invoke(joint);
    }

    public void SetAngle()
    {
        joint.SetAngle(-float.Parse(inputField.text));
        slider.value = -joint.GetAngle();
        inputField.text = (-joint.GetAngle()).ToString("F2");

        if (OnValueChange != null)
            OnValueChange.Invoke(joint);
    }
```

Figura 164 - CJointDisplay. Métodos ChangeAngle y SetAngle.

El método "ChangeAngle" es usado por el "Slider" mientras que el método "SetAngle" es usado por el elemento de entrada de texto. Se han usado métodos distintos porque necesitan un tratamiento diferente, pero ambos afectan al ángulo de la articulación de la misma manera.

Al final de ambos se encuentra la gestión del evento que proporciona a aquellos elementos subscritos la variable *CJoint* que almacena.

7.5.2.4.2 Clase MoveAxisMenu

La clase *MoveAxisMenu* se encuentra en el objeto principal del programa "ProgramObject".

```
// List of robots
public List<CJoint> joints = new List<CJoint>();

// Display
public MoveAxisMenuDisplay moveAxisMenuDisplay;
public bool needChange;
public string displayName;
```

Figura 165 - MoveAxisMenu. Variables.

Las variables que contiene esta clase son:

- Una lista de las articulaciones del robot.
- Una variable de la clase que gestiona la sección visual de ésta. *MoveAxisMenuDisplay*.
- Una variable booleana que es usada para hacerle saber a la rutina principal que se ha producido un cambio en alguna de las articulaciones y hace falta mover el robot.

```
void Start()
{
    CJointDisplay.OnValueChange += HandleOnValueChange;
    moveAxisMenuDisplay.canvas.enabled = false;
}

void OnDestroy()
{
    CJointDisplay.OnValueChange -= HandleOnValueChange;
}

// Event Method
void HandleOnValueChange(CJoint joint)
{
    if (joints.Contains(joint))
    {
        needChange = true;
    }
}
```

Figura 166 - MoveAxisMenu. Gestión de Eventos.

Esta clase se subscribe a los eventos producidos en la clase *CJointDisplay* e invoca el método "HandleOnValueChange" que activa la variable flag "needChange". También desactiva el

Canvas que gestiona porque no es un menú que se encuentre en pantalla al inicio del programa.

```
// Initialize Methods
// Used when a new robot is selected
public void Prime(List<CJoint> joints, string displayName)
{
    if (!moveAxisMenuDisplay.canvas.enabled)
        moveAxisMenuDisplay.canvas.enabled = true;

    this.displayName = displayName;
    this.joints = joints;
    moveAxisMenuDisplay.Prime(this);
}
```

Figura 167 - MoveAxisMenu. Método Prime.

El método "Prime" de la *Figura 167* activa el *Canvas*, actualiza sus variables internas e inicializa *MoveAxisMenuDisplay*.

```
// Used when the display robot suffers a change in their angles
public void UpdateValues(List<CJoint> joints, string displayName)
{
    this.displayName = displayName;
    this.joints = joints;
    moveAxisMenuDisplay.UpdateValues(this);
}
```

Figura 168 - MoveAxisMenu. Método UpdateValues.

El método "UpdateValues" es usado desde la rutina principal para actualizar las articulaciones que almacena esta clase durante la simulación.

7.5.2.4.3 Clase MoveAxisMenuDisplay

La clase *MoveAxisMenuDisplay* gestiona la representación de la clase *MoveAxisMenu* en la interfaz. La razón por la que se han creado clases distintas para la gestión de datos y la gestión de la interfaz es por la libertad que aporta. Al separar estas clases las simplificamos y definimos su funcionalidad con mayor exactitud.

```
// Public variables
    public Text displayName;
    public Transform targetTransform;
    public CJointDisplay CJointDisplayPrefab;

    public List<CJointDisplay> cJointDisplays;

    public MoveAxisMenu moveAxisMenu;
    public Canvas canvas;
```

Figura 169 - MoveAxisMenuDisplay. Variables.

Las variables que contiene esta clase son:

- Una variable tipo *Transform* para almacenar la posición del objeto al que se añadirán los distintos *Prefab* de la clase *CJointDisplay*.
- Una lista de los *CJointDisplay* que se han creado para poder actualizar sus valores más adelante.
- El *Prefab* de la clase *CJointDisplay* que se asigna desde el inspector.
- El *Canvas* del menú y una variable de la clase que gestiona.

```
    void Awake()
    {
        CJointDisplayPrefab = Resources.Load<CJointDisplay>("ABB
Library/Robots/UIPrefab/JointBar");
        canvas = GetComponent<Canvas>();
    }
```

Figura 170 - MoveAxisMenuDisplay. Método Awake.

En el método *Awake* se carga la información del *Prefab* e inicializa la variable del *Canvas*.

```
    // Initilizer functions
    public void Prime(MoveAxisMenu moveAxisMenu)
    {
        displayName.text = moveAxisMenu.displayName;
        for (int a = 1; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.moveAxisMenu = moveAxisMenu;
        foreach (CJoint joint in this.moveAxisMenu.joints)
        {
            cJointDisplays.Add((CJointDisplay)Instantiate(CJointDisplayPrefab));
            cJointDisplays[cJointDisplays.Count -
1].transform.SetParent(targetTransform, false);
            cJointDisplays[cJointDisplays.Count - 1].Prime(joint);
        }
    }
```

Figura 171 - MoveAxisMenuDisplay. Método Prime.

El método "Prime" de esta clase inicializa los elementos de texto del menú, destruye todos los *Prefab* de articulación que tuviera almacenados (si los tuviera) y crea todos de nuevo con la información que se le ha pasado.

```
public void UpdateValues(MoveAxisMenu moveAxisMenu)
{
    if(displayName != null)
        displayName.text = moveAxisMenu.displayName;

    if (moveAxisMenu != null)
        this.moveAxisMenu = moveAxisMenu;

    foreach (CJoint joint in this.moveAxisMenu.joints)
        cJointDisplays[this.moveAxisMenu.joints.IndexOf(joint)].Prime(joint);
}
```

Figura 172 - MoveAxisMenuDisplay. Método UpdateValues.

El método "UpdateValues" en vez de destruir todos los *Prefab*, únicamente actualiza sus valores, lo cual es significativamente mejor para el rendimiento durante la simulación.

7.5.2.5 Menú de control

El menú de control que aparece en la *Figura 173* se usa principalmente para gestionar algunos de los métodos definidos en la clase *Communications*. Este menú da control al usuario sobre el estado del programa:

- El círculo verde de la izquierda pasa a ser rojo durante la sincronización tras pulsar el botón "Sync" que llama al método con el mismo nombre de la clase *Communications*.
- El botón "Play" pasa a mostrar "Pause" al iniciarse la simulación y llama al método "Sim" de la clase *Communications*.
- El botón "Stop" detiene la simulación llamando al método "stopSim" de la clase *Communications*.
- El botón "Loop" cambia el modo de ejecución llamando al método con el mismo nombre de la clase *Communications*. Si se encuentra en bucle el círculo de la derecha pasa a verde.

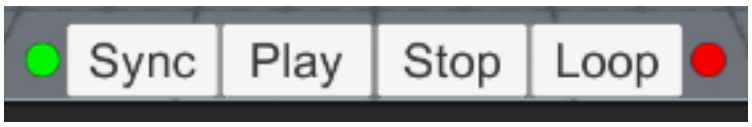

Figura 173 - Menú de control.

7.5.2.6 Panel de control

El panel de control este compuesto por tres botones que activan el panel de su nombre desactivando el resto mediante sus respectivos *Canvas*.

- El botón "Robots" activa el panel de robots.
- El botón "Tools" activa el panel de herramientas.
- El botón "Prog." Activa el panel de programación.

Figura 174 - Panel de control.

7.5.2.6.1 Clase ControlPanel

La clase *ControlPanel* tiene definidos los métodos que gestionan el comportamiento del panel de control.

- El método "ProgramPanel" está asignado al botón "Prog.".
- El método "RobotPanel" está asignado al botón "Robots".
- El método "ToolPanel" está asignado al botón "Tools".

```
public void ProgramPanel()
{
    programmingPanel.enabled = true;
    robotsPanel.enabled = false;
    toolsPanel.enabled = false;
}

public void RobotPanel()
{
    programmingPanel.enabled = false;
    robotsPanel.enabled = true;
    toolsPanel.enabled = false;
}

public void ToolPanel()
{
    programmingPanel.enabled = false;
    robotsPanel.enabled = false;
    toolsPanel.enabled = true;
}
```

Figura 175 - ControlPanel. Métodos.

7.5.2.7 Panel de robots

El panel de robots se encarga de mostrar al usuario la librería de robots disponibles y la lista de robots creados tal y como se muestra en la *Figura 176*.

Figura 176 - Panel de robots.

La sección inferior nos muestrea los tipos de robots disponibles. Tras seleccionar alguno el usuario podrá añadirlo a la lista al pulsar el botón "Add", que llama al método "AddRobot" de la clase *MainProgram*.

En caso de querer eliminar alguno se selecciona de la lista que aparece en la sección superior y se pulsa el botón "Delete" que llama al método "DeleteRobot" de la clase *MainProgram.*

En el bloque de texto se muestra el robot activo.

7.5.2.7.1 Clase RobotUIDisplay

El *Prefab* de la *Figura 177* contiene un componente de la clase *RobotUIDisplay*.

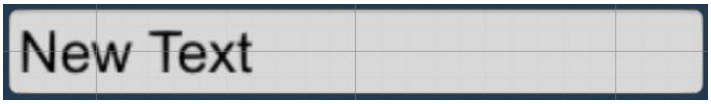

Figura 177 - RobotUIDisplay Prefab.

Este componente es en realidad un botón modificado y muestra el nombre del robot que almacena.

```
    // Text element of the scene
    public Text textName;

    // Internal robot
    public GenericRobot robot;

    // Delegate functionality
    public delegate void RobotUIDisplayDelegate(GenericRobot robot);
    public static event RobotUIDisplayDelegate OnClick;
```

Figura 178 - RobotUIDisplay. Variables.

Las variables que componen esta clase son:

- El objeto tipo texto a usar.
- El robot del que se pretende mostrar información.
- Un evento para que otras clases sepan si el botón ha sido pulsado.

```
// Initializer method
public void Prime(GenericRobot robot)
{
    this.robot = robot;
    if (textName != null)
        textName.text = robot.displayName;
}

// UI Elements method
public void Click()
{
    if (OnClick != null)
        OnClick.Invoke(robot);
}
```

Figura 179 - RobotUIDisplay. Métodos.

Además del método "Prime" que se usa para inicializar el texto a mostrar, nos encontramos con el método "Click" que se debe configurar en la ventana del inspector del *Prefab* como en el ejemplo de la *Figura 156* y sirve para activar el evento cuando se pulse éste.

7.5.2.7.2 Clase RobotScrollListDisplay

La sección superior del panel de robots, que se ve en la *Figura 176* tiene un componente de la clase *RobotScrollListDisplay*.

```
// Public variables
public Text actRobotShowSlot;
public Transform targetTransform;
public RobotUIDisplay robotUIDisplayPrefab;

public RobotList robotList;
```

Figura 180 - RobotScrollListDisplay. Variables.

Las variables que contiene esta clase son:

- Un elemento de tipo texto para mostrar el robot activo.
- Una variable tipo *Transform* para almacenar el objeto con el que vincular los *Prefab*.
- El *Prefab* del elemento mostrado en la *Figura 177*.
- La lista de robots a mostrar en pantalla.

```
    void Awake()
    {
        robotUIDisplayPrefab = Resources.Load<RobotUIDisplay>("ABB
Library/Robots/UIPrefab/RobotDisplaySlot");
    }

    // Start/Destroy
    void Start()
    {
        RobotList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        RobotList.OnChanged -= HandleOnChanged;
    }
```

Figura 181 - RobotScrollListDisplay. Métodos de Unity.

En el método *Awake* se cargan los *Prefab* para optimizar los recursos. En los otros métodos la clase se subscribe y desubscribe de los eventos producidos en la clase *RobotList*.

```
    // HandleOnClick
    void HandleOnChanged(RobotList robotList)
    {
        if (this.robotList == robotList)
            Prime(robotList);
    }
```

Figura 182 - RobotScrollListDisplay. Método HandleOnChanged.

El método "HandleOnChanged" es llamado con el evento de la clase *RobotList*. Al hacerlo se llama al método "Prime" definido a continuación.

```
    // Initilizer method
    public void Prime(RobotList robotList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.robotList = robotList;
        foreach (GenericRobot robot in this.robotList.robots)
        {
            RobotUIDisplay display =
(RobotUIDisplay)Instantiate(robotUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(robot);
        }
    }
```

Figura 183 - RobotScrollListDisplay. Método Prime.

El método "Prime" de la clase *RobotScrollListDisplay* destruye todos los *Prefab* vinculados a la variable "targetTransform" y los vuelve a crear con la información actualizada.

```
// Functional methods
public void ShowRobot(int aR)
{
    string displayName = "Nothing";
    if (robotList.robots[aR] != null)
        displayName = robotList.robots[aR].displayName;

    actRobotShowSlot.text = displayName;
}

public void EraseShowRobot()
{
    actRobotShowSlot.text = "";
}
```

Figura 184 - RobotScrollListDisplay. Métodos ShowRobot y EraseShowRobot.

Los métodos "ShowRobot" y "EraseShowRobot" actualizan la información mostrada en la variable tipo texto de la clase con la información del robot activo.

7.5.2.7.3 Clase RobotLibraryIcon

La clase *RobotLibraryIcon* contiene los datos de un tipo de robot que se deben mostrar al usuario para poder identificarlo. Normalmente esto incluye un nombre y un "Sprite" que es básicamente una imagen.

7.5.2.7.4 Clase RobotLibraryIconDisplay

El elemento de la *Figura 185* contiene la clase *RobotLibraryIconDisplay*.

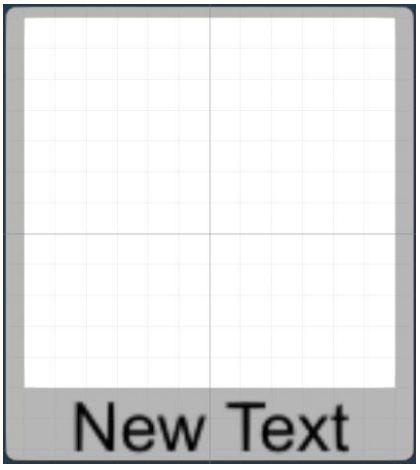

Figura 185 - Prefab del icono de un robot.

Este elemento se mostrará en la sección inferior del panel de robots y formará parte de la parte visual de la librería de robots.

```
// Visual variables
    public Text textName;
    public Image image;

    // Internal variable
    public RobotLibraryIcon robot;

    // Event management
    public delegate void RobotLibraryIconDisplayDelegate(RobotLibraryIcon
robot);
    public static event RobotLibraryIconDisplayDelegate onClick;
```

Figura 186 - RobotLibraryIconDisplay. Variables.

Las variables que contiene esta clase son:

- Un objeto de tipo texto.
- Un objeto de tipo imagen.
- La variable de la que se debe mostrar la información.
- Un evento de tipo “OnClick” que se activa al pulsar el objeto.

```
    // Start is called before the first frame update
    void Start()
    {
        if (robot != null)
            Prime(robot);
    }

    // Initializer Method
    public void Prime(RobotLibraryIcon robot)
    {
        this.robot = robot;
        if (textName != null)
            textName.text = robot.displayName;
        if (image != null)
            image.sprite = robot.sprite;
    }
```

Figura 187 - RobotLibraryIconDisplay. Métodos Start y Prime.

El método *Start* llama al método “Prime” al iniciar el programa. El método “Prime” inicializa la información del *Prefab* acorde a la variable que se le ha pasado.

```
    // UI Method
    public void Click()
    {
        string displayName = "Nothing";
        if (robot != null)
            displayName = robot.displayName;

        if (onClick != null)
            onClick.Invoke(robot);
    }
```

Figura 188 - RobotLibraryIconDisplay. Método Click.

El método "Click" se debe configurar en la ventana del inspector del *Prefab* como en el ejemplo de la *Figura 156* y sirve para activar el evento cuando se pulse el icono.

7.5.2.7.5 Clase RobotLibraryDisplay

La clase *RobotLibraryDisplay* corresponde a la sección inferior del panel de robots y muestra los iconos de los diferentes tipos de robot. Su funcionamiento es parecido al de la clase *RobotScrollListDisplay*.

```
// Public variables
    public Transform targetTransform;
    public RobotLibraryIconDisplay robotLibraryIconDisplayPrefab;

    public RobotLibrary robotLibrary;
```

Figura 189 - RobotLibraryDisplay. Variables.

Las variables de esta clase son:

- Una variable tipo *Transform* para almacenar el objeto con el que vincular los *Prefab*.
- El *Prefab* del elemento mostrado en la *Figura 185*.
- La variable de la que se debe mostrar la información.

```
    void Awake()
    {
        robotLibraryIconDisplayPrefab =
Resources.Load<RobotLibraryIconDisplay>("ABB
Library/Robots/UIPrefab/RobotLibraryIconDisplaySlot");
    }
```

Figura 190 - RobotLibraryDisplay. Método Awake.

En el método *Awake* se cargan los *Prefab* para optimizar los recursos.

```
// Initilizer method
    public void Prime(RobotLibrary robotLibrary)
    {
        this.robotLibrary = robotLibrary;
        foreach (RobotLibraryIcon robot in this.robotLibrary.robotIcons)
        {
            RobotLibraryIconDisplay display =
(RobotLibraryIconDisplay)Instantiate(robotLibraryIconDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(robot);
        }
    }
```

Figura 191 - RobotLibraryDisplay. Método Prime.

El método "Prime" inicializa todos los iconos presentes en la librería.

7.5.2.8 Panel de herramientas

El panel de herramientas se encarga de mostrar al usuario la librería de herramientas disponibles y la lista de herramientas creadas tal y como se muestra en la *Figura 192*.

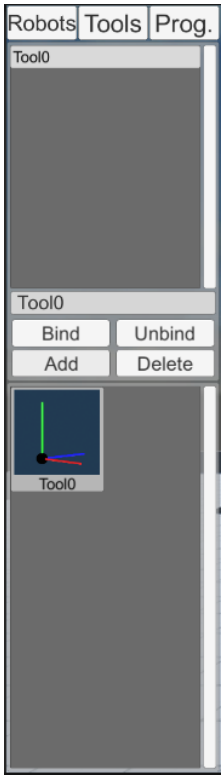

Figura 192 - Panel de herramientas.

La sección inferior nos muestrea los tipos de herramientas disponibles. Tras seleccionar alguno el usuario podrá añadirlo a la lista al pulsar el botón "Add", que llama al método "AddTool" de la clase *MainProgram*.

En caso de querer eliminar alguno se selecciona de la lista que aparece en la sección superior y se pulsa el botón "Delete" que llama al método "DeleteTool" de la clase *MainProgram.*

En este panel también aparecen los botones "Bind" y "UnBind" que llaman a los métodos "BindTool" y "UnBindTool" de la clase *MainProgram. Estos métodos vinculan o desvinculan la herramienta activa del robot activo.*

En el bloque de texto se muestra la herramienta activa.

7.5.2.8.1 Clase ToolUIDisplay

El *Prefab* de la contiene un componente de la clase *ToolUIDisplay*.

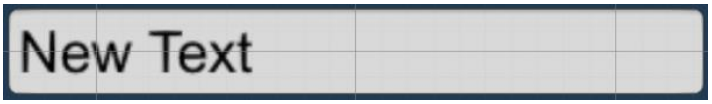

Figura 193 - ToolUIDisplay Prefab.

Este componente es en realidad un botón modificado y muestra el nombre de la herramienta que almacena.

```
    // Text element of the scene
    public Text textName;

    // Internal robot
    public Tool tool;

    // Delegate functionality
    public delegate void ToolUIDisplayDelegate(Tool tool);
    public static event ToolUIDisplayDelegate OnClick;
```

Figura 194 - ToolUIDisplay. Variables.

Las variables que componen esta clase son:

- El objeto tipo texto a usar.
- La herramienta de la que se pretende mostrar información.

- Un evento para que otras clases sepan si el botón ha sido pulsado.

```
// Initializer method
public void Prime(Tool tool)
{
    this.tool = tool;
    if (textName != null)
        textName.text = tool.displayName;
}

// UI Elements method
public void Click()
{
    if (OnClick != null)
        OnClick.Invoke(tool);
}
```

Figura 195 - ToolUIDisplay. Métodos Prime y Click.

Además del método "Prime" que se usa para inicializar el texto a mostrar, nos encontramos con el método "Click" que se debe configurar en la ventana del inspector del *Prefab* como en el ejemplo de la *Figura 156* y sirve para activar el evento cuando se pulse el elemento.

7.5.2.8.2 Clase ToolScrollListDisplay

La sección superior del panel de herramientas tiene un componente de la clase *ToolScrollListDisplay*.

```
// Public variables
public Text actToolShowSlot;
public Transform targetTransform;
public ToolUIDisplay toolUIDisplayPrefab;

public ToolList toolList;
```

Figura 196 - ToolScrollListDisplay. Variables.

Las variables que contiene esta clase son:

- Un elemento de tipo texto para mostrar la herramienta activa.
- Una variable tipo *Transform* para almacenar el objeto con el que vincular los *Prefab*.
- El *Prefab* del elemento mostrado en la Figura 193.
- La lista de herramientas a mostrar en pantalla.

```
    void Awake()
    {
        toolUIDisplayPrefab = Resources.Load<ToolUIDisplay>("ABB
Library/RobotStudioObjects/Tools/UIPrefab/ToolDisplaySlot");
    }

    // Start/Destroy
    void Start()
    {
        ToolList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        ToolList.OnChanged -= HandleOnChanged;
    }
```

Figura 197 - ToolScrollListDisplay. Métodos de Unity.

En el método *Awake* se cargan los *Prefab* para optimizar los recursos. En los otros métodos la clase se subscribe y desubscribe de los eventos producidos en la clase *ToolList.*

```
    // HandleOnClick
    void HandleOnChanged(ToolList toolList)
    {
        if (this.toolList == toolList)
        {
            Prime(toolList);
        }
    }
```

Figura 198 - ToolScrollListDisplay. Método HandleOnClick.

El método “HandleOnChanged” es llamado con el evento de la clase *ToolList.* Al hacerlo se llama al método “Prime” definido a continuación.

```
    // Initilizer function
    public void Prime(ToolList toolList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.toolList = toolList;
        foreach (Tool tool in this.toolList.tools)
        {
            ToolUIDisplay display =
(ToolUIDisplay)Instantiate(toolUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(tool);
        }
    }
```

Figura 199 - ToolScrollListDisplay. Método Prime.

El método "Prime" de la clase *ToolScrollListDisplay* destruye todos los *Prefab* vinculados a la variable "targetTransform" y los vuelve a crear con la información actualizada.

```
// Functional methods
public void ShowTool(int aT)
{
    string displayName = "Nothing";
    if (toolList.tools[aT] != null)
        displayName = toolList.tools[aT].displayName;

    actToolShowSlot.text = displayName;
}

public void EraseShowTool()
{
    actToolShowSlot.text = "";
}
```

Figura 200 - ToolScrollListDisplay. Métodos ShowTool y EraseShowTool

Los métodos "ShowTool" y "EraseShowTool" actualizan la información mostrada en la variable tipo texto de la clase con la información de la herramienta activa.

7.5.2.8.3 Clase ToolLibraryIcon

La clase *ToolLibraryIcon* contiene los datos de un tipo de herramienta que se deben mostrar al usuario para poder identificarla. Normalmente esto incluye un nombre y un "Sprite", que es básicamente una imagen.

7.5.2.8.4 Clase ToolLibraryIconDisplay

El elemento de la *Figura 201* contiene la clase *ToolLibraryIconDisplay*.

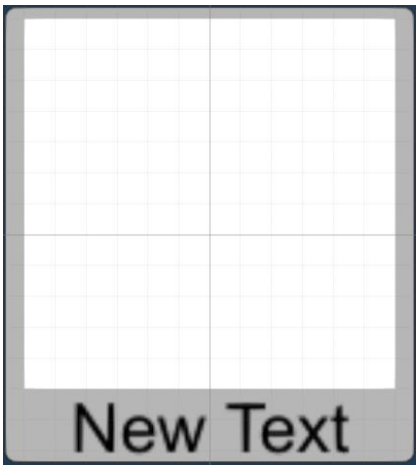

Figura 201 - Prefab del icono de una herramienta

Este elemento se mostrará en la sección inferior del panel de herramientas y formará parte de la parte visual de la librería de herramientas.

```
// Visual variables
public Text textName;
public Image image;

// Internal variable
public ToolLibraryIcon tool;

// Event management
public delegate void ToolLibraryIconDisplayDelegate(ToolLibraryIcon tool);
public static event ToolLibraryIconDisplayDelegate onClick;
```

Figura 202 - ToolLibraryIconDisplay. Variables.

Las variables que contiene esta clase son:

- Un objeto de tipo texto.
- Un objeto de tipo imagen.
- La variable de la que se debe mostrar la información.
- Un evento de tipo “OnClick” que se activa al pulsar el objeto.

```
// Start is called before the first frame update
void Start()
{
    if (tool != null)
        Prime(tool);
}

// Initializer method
public void Prime(ToolLibraryIcon tool)
{
    this.tool = tool;
    if (textName != null)
        textName.text = tool.displayName;
    if (image != null)
        image.sprite = tool.sprite;
}
```

Figura 203 - ToolLibraryIconDisplay. Métodos Start y Prime.

El método *Start* llama al método “Prime” al iniciar el programa. El método “Prime” inicializa la información del *Prefab* acorde a la variable que se le ha pasado.

```
    // UI method
    public void Click()
    {
        string displayName = "Nothing";
        if (tool != null)
            displayName = tool.displayName;

        if (onClick != null)
            onClick.Invoke(tool);
    }
```

Figura 204 - ToolLibraryIconDisplay. Método Click.

El método "Click" se debe configurar en la ventana del inspector del *Prefab* como en el ejemplo de la *Figura 156* y sirve para activar el evento cuando se pulse el icono.

7.5.2.8.5 Clase ToolLibraryDisplay

La clase *ToolLibraryDisplay* corresponde a la sección inferior del panel de herramientas y muestra los iconos de los diferentes tipos de herramientas. Su funcionamiento es parecido al de la clase *ToolScrollListDisplay*.

```
    // Public variables
    public Transform targetTransform;
    public ToolLibraryIconDisplay toolLibraryIconDisplayPrefab;

    public ToolLibrary toolLibrary;
```

Figura 205 - ToolLibraryDisplay. Variables.

Las variables de esta clase son:

- Una variable tipo *Transform* para almacenar el objeto con el que vincular los *Prefab*.
- El *Prefab* del elemento mostrado en la *Figura 201*.
- La variable de la que se debe mostrar la información.

```
    void Awake()
    {
        toolLibraryIconDisplayPrefab =
Resources.Load<ToolLibraryIconDisplay>("ABB
Library/RobotStudioObjects/Tools/UIPrefab/ToolLibraryIconDisplaySlot");
    }
```

Figura 206 - ToolLibraryDisplay. Método Awake.

En el método *Awake* se cargan los *Prefab* para optimizar los recursos.

```
    // Initilizer method
    public void Prime(ToolLibrary toolLibrary)
    {
        this.toolLibrary = toolLibrary;
        foreach (ToolLibraryIcon tool in this.toolLibrary.toolIcons)
        {
            ToolLibraryIconDisplay display =
(ToolLibraryIconDisplay)Instantiate(toolLibraryIconDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(tool);
        }
    }
```

Figura 207 - ToolLibraryDisplay. Método Prime.

El método "Prime" inicializa todos los iconos presentes en la librería.

7.5.2.9 Panel de programación

El panel de programación se encarga de mostrar al usuario la lista de *RobTarget* creados y la lista de instrucciones creadas tal y como se muestra en la *Figura 208.*

Figura 208 - Panel de programación.

La sección superior del panel permite al usuario añadir y eliminar *RobTarget* con los botones "Add" y "Delete" que llaman a los métodos "AddTarget" y "DeleteTarget" de la clase *MainProgram* respetivamente. Al seleccionar alguno de los objetivos creados el robot se desplazará hasta la posición que tenía a la hora de crearlo.

La sección inferior corresponde a la creación de instrucciones. Los botones "Add" y "Delete" llaman a los métodos "AddInstruction" y "DeleteInstrucion"de la clase *MainProgram* respectivamente.

Para crear las instrucciones el usuario tendrá que seleccionar entras las opciones que se muestras en los objetos tipo "Dropdown".

Los tres "Dropdown" de la izquierda corresponden con el tipo de movimiento, velocidad y precisión de la instrucción.

Tipo de movimiento	Precisión	Velocidad
MoveL	fine	v5
MoveJ	z0	v10
MoveC	z1	v20
	z5	v30
	z10	v40
	z15	v50
	z20	v60
	z30	v80
	z40	v100
	z50	v150
	z60	v200
	z80	v300
	z100	v400
	z150	v500
	z200	v600
		v800
		v1000
		v1500
		v2000
		v2500

Tabla 1 - Opciones de instrucción disponible.

En la *Tabla 1* se muestra las opciones disponibles en los "Dropdown". Las columnas de velocidad y precisión ofrecen las opciones predefinidas en *RobotStudio*. La columna de tipo de movimiento nos ofrece los dos tipos de movimientos básicos *MoveL* y *MoveJ*. También se ha configurado el movimiento avanzado *MoveC*.

Los "Dropdown" de la derecha corresponden a los *RobTarget* que se usarán en la instrucción y tiene disponibles todas las variables de este tipo que se hayan creado durante la ejecución.

En caso de usarse el comando *MoveC* se usarán ambos y habrá que comprobar que la instrucción a realizar se puede ejecutar.

7.5.2.9.1 Clase RobTargetUIDisplay

La clase *RobTargetUIDisplay* gestiona el *Prefab* del elemento que se muestra en la parte superior del panel de control y aparece en la *Figura 209*.

Figura 209 - RobTargetUIDisplay Prefab.

Las variables que componen esta clase son:

- El objeto tipo texto a usar.
- El *RobTarget* del que se pretende mostrar información.
- Un evento para que otras clases sepan si el botón ha sido pulsado.

```
// Text element of the scene
public Text textName;

// Internal robTarget
public RobTarget target;

// Delegate functionality
public delegate void RobTargetUIDisplayDelegate(RobTarget target);
public static event RobTargetUIDisplayDelegate OnClick;
```

Figura 210 - RobTargetUIDisplay. Variables.

Además del método "Prime" que se usa para inicializar el texto a mostrar, nos encontramos con el método "Click" que se debe configurar en la ventana del inspector del *Prefab* como en el ejemplo de la *Figura 156* y sirve para activar el evento cuando se pulse el elemento.

```
// Initializer method
public void Prime(RobTarget target)
{
    this.target = target;
    if (textName != null)
        textName.text = target.displayName;
}

// UI Elements method
public void Click()
{
    if (OnClick != null)
        OnClick.Invoke(target);
}
```

Figura 211 - RobTargetUIDisplay. Métodos Prime y Click.

7.5.2.9.2 Clase RobTargetScrollListDisplay

La clase *RobTargetScrollListDisplay* funciona de forma similar a las clases *ToolScrollListDisplay* y *RobotScrollListDisplay*.

```
// Public variables
public Transform targetTransform;
public RobTargetUIDisplay robTargetUIDisplayPrefab;

public Text actRobTargetShowSlot;

public RobTargetList TargetList;
```

Figura 212 - RobTargetScrollListDisplay. Variables.

Las variables que contiene esta clase son:

- Un elemento de tipo texto para mostrar el *RobTarget* activo.
- Una variable tipo *Transform* para almacenar el objeto con el que vincular los *Prefab*.
- El *Prefab* del elemento mostrado en la *Figura 209*.
- La lista de *RobTarget* a mostrar en pantalla.

```
    void Awake()
    {
        robTargetUIDisplayPrefab = Resources.Load<RobTargetUIDisplay>("ABB
Library/RobotStudioObjects/RobTargets/UIPrefab/RobTargetDisplaySlot");
    }

    // Start/Destroy
    void Start()
    {
        RobTargetList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        RobTargetList.OnChanged -= HandleOnChanged;
    }
```

Figura 213 - RobTargetScrollListDisplay. Métodos de Unity.

En el método *Awake* se cargan los *Prefab* para optimizar los recursos. En los otros métodos la clase se subscribe y desubscribe de los eventos producidos en la clase *RobTargetList*.

```
    // HandleOnChange
    void HandleOnChanged(RobTargetList targetList)
    {
        if (this.TargetList == targetList)
            Prime(targetList);
    }
```

Figura 214 - RobTargetScrollListDisplay. Método HandleOnChange.

El método "HandleOnChange" es llamado con el evento de la clase *RobTargetList*. Al hacerlo se llama al método "Prime" definido a continuación.

```
    // Initilizer function
    public void Prime(RobTargetList robTargetList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.TargetList = robTargetList;
        foreach (RobTarget target in this.TargetList.targets)
        {
            RobTargetUIDisplay display =
(RobTargetUIDisplay)Instantiate(robTargetUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(target);
        }
    }
```

Figura 215 - RobTargetScrollListDisplay. Método Prime.

El método "Prime" de la clase *RobTargetScrollListDisplay* destruye todos los *Prefab* vinculados a la variable "targetTransform" y los vuelve a crear con la información actualizada.

```
// Functional methods
public void ShowTarget(int aT)
{
    string displayName = "Nothing";
    if (TargetList.targets[aT] != null)
        displayName = TargetList.targets[aT].displayName;

    actRobTargetShowSlot.text = displayName;
}

public void EraseShowTarget()
{
    actRobTargetShowSlot.text = "";
}
```

Figura 216 - RobTargetScrollListDisplay. Métodos ShowTarget y EraseShowTarget.

Los métodos "ShowTarget" y "EraseShowTarget" actualizan la información mostrada en la variable tipo texto de la clase con la información del *RobTarget* activo.

7.5.2.9.3 Clase InstructionUIDisplay

La clase *InstuctionUIDisplay* gestiona el *Prefab* del elemento que se muestra en la parte inferior del panel de programación y aparece en la *Figura 217*.

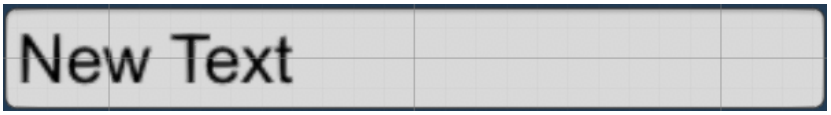

Figura 217 - InstructionUIDisplay Prefab.

Las variables que componen esta clase son:

- El objeto tipo texto a usar.
- La instrucción de la que se pretende mostrar información.
- Un evento para que otras clases sepan si el botón ha sido pulsado.

```
// Text element of the scene
public Text textName;

// Internal robTarget
public Instruction order;

// Delegate functionality
public delegate void InstructionUIDisplayDelegate(Instruction order);
public static event InstructionUIDisplayDelegate OnClick;
```

Figura 218 - InstructionUIDisplay. Variables.

Además del método "Prime" que se usa para inicializar el texto a mostrar, nos encontramos con el método "Click" que se debe configurar en la ventana del inspector del *Prefab* como en el ejemplo de la *Figura 156* y sirve para activar el evento cuando se pulse el elemento.

```
// Initializer method
public void Prime(Instruction order)
{
    this.order = order;
    if (textName != null)
        textName.text = order.displayName;
}

// UI Elements method
public void Click()
{
    if (OnClick != null)
        OnClick.Invoke(order);
}
```

Figura 219 - InstructionUIDisplay. Métodos Prime y Click.

7.5.2.9.4 Clase InstructionScrollListDisplay

La clase *InstructionScrollListDisplay* es similar a las clases *RobTargetScrollListDisplay*, *RobotScrollListDisplay* y *ToolScrollListDisplay*.

```
// Public variables
public Text actInstructionShowSlot;
public Transform targetTransform;
public InstructionUIDisplay instructionUIDisplayPrefab;

public InstructionList instructionList;
```

Figura 220 - InstructionScrollListDisplay. Variables.

Las variables que contiene esta clase son:

- Un elemento de tipo texto para mostrar la instrucción activa.
- Una variable tipo *Transform* para almacenar el objeto con el que vincular los *Prefab*.

- El *Prefab* del elemento mostrado en la *Figura 217*.
- La lista de instruccionesRobTarget a mostrar en pantalla.

```
    void Awake()
    {
        instructionUIDisplayPrefab = Resources.Load<InstructionUIDisplay>("ABB
Library/RobotStudioObjects/Instructions/InstructionDisplaySlot");
    }

    // Start/Destroy
    void Start()
    {
        InstructionList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        InstructionList.OnChanged -= HandleOnChanged;
    }
```

Figura 221 - InstructionScrollListDisplay. Métodos de Unity.

En el método *Awake* se cargan los *Prefab* para optimizar los recursos. En los otros métodos la clase se subscribe y desubscribe de los eventos producidos en la clase *InstructionList.*

```
    // HandleOnClick
    void HandleOnChanged(InstructionList instructionList)
    {
        if (this.instructionList == instructionList)
            Prime(instructionList);
    }
```

Figura 222 - InstructionScrollListDisplay. Método HandleOnChange.

El método "HandleOnChange" es llamado con el evento de la clase *InstructionList.* Al hacerlo se llama al método "Prime" definido a continuación.

```
    // Initilizer function
    public void Prime(InstructionList instructionList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.instructionList = instructionList;
        foreach (Instruction order in this.instructionList.orders)
        {
            InstructionUIDisplay display =
(InstructionUIDisplay)Instantiate(instructionUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(order);
        }
    }
```

Figura 223 - InstructionScrollListDisplay. Método Prime.

El método "Prime" de la clase *InstructionScrollListDisplay* destruye todos los *Prefab* vinculados a la variable "targetTransform" y los vuelve a crear con la información actualizada.

```
// Functional methods
public void ShowInstruction(int aI)
{
    string displayName = "Nothing";
    if (instructionList.orders[aI] != null)
        displayName = instructionList.orders[aI].displayName;

    actInstructionShowSlot.text = displayName;
}

public void EraseShowInstruction()
{
    actInstructionShowSlot.text = "";
}
```

Figura 224 - InstructionScrollListDisplay. Métodos ShowInstruction y EraseShowInstruction.

Los métodos "ShowInstruction" y "EraseShowInstruction" actualizan la información mostrada en la variable tipo texto de la clase con la información de la herramienta activa.

7.5.2.9.5 Clase InstructionsOptionsMenu

La clase *InstructionOptionsmenu* gestiona los "Dropdown" de la sección inferior del panel de programación.

```
// Public variables
public MoveTypeDropdown moveType;
public SpeedDropdown speed;
public PrecisionDropdown precision;
public RobTargetDropdown target_1;
public RobTargetDropdown target_2;

public RobTargetList targetList;
```

Figura 225 - InstructionsOptionsMenu. Variables.

Esta clase contiene una variable de cada clase de "Dropdown" que se ha creado. Excepto de la clase *RobTargetDropdown* de la que tiene dos.

También tiene una variable tipo *RobTargetList* para poder actualizar las opciones mostradas al usuario.

```
    // Start is called before the first frame update
    void Start()
    {
        targetList = gameObject.GetComponent(typeof(RobTargetList)) as
RobTargetList;
        MoveTypeDropdown.OnClick += HandleOnClick;

        if (GetActMoveType() != 2)
        {
            target_2.Prime(targetList);
            target_2.dropDown.gameObject.SetActive(false);
        }
    }

    void OnDestroy()
    {
        MoveTypeDropdown.OnClick += HandleOnClick;
    }
```

Figura 226 - InstructionsOptionsMenu. Métodos de Unity.

En los métodos propios de Unity de esta clase, además de subscribirse y desubscribirse de los eventos de la clase *MoveTypeDropdown*, se desactiva el segundo elemento *RobTargetDropdown* por estar seleccionado el tipo de movimiento *MoveC*.

```
    // Functional Methods
    // Event Method
    void HandleOnClick(int t)
    {
        if (GetActMoveType() == 2)
        {
            target_2.dropDown.gameObject.SetActive(true);
            target_2.Prime(targetList);
        }
        else
            target_2.dropDown.gameObject.SetActive(false);
    }
```

Figura 227 - InstructionsOptionsMenu. Método HandleOnClick.

El método "HandleOnClick" de esta clase comprueba si se ha seleccionado el tipo de movimiento *MoveC* y en caso de ser así activa el segundo elemento *RobTargetDropdown.*

```
    // Get Functions
    public int GetActMoveType() { return moveType.dropDown.value; }
    public int GetActSpeed() { return speed.dropDown.value; }
    public int GetActPrecision() { return precision.dropDown.value; }
    public int GetActTarget1() { return target_1.dropDown.value; }
    public int GetActTarget2() { return target_2.dropDown.value; }
```

Figura 228 - InstructionsOptionsMenu. Métodos Get.

Los métodos "Get" como de costumbre nos permiten acceder a los valores de las variables internas. En este caso en concreto, nos devuelven el valor guardado en los objetos tipo "Dropdown", que corresponde con la posición en la lista de la opción seleccionada.

7.5.2.9.6 Clase MoveTypeDropdown

La clase *MoveTypeDropdown* gestiona el elemento "Dropdown" que permite seleccionar el tipo de movimiento. A diferencia de las otras clases de este tipo, esta clase cuenta con un evento.

```
    // Public variables
    public List<string> options = new List<string>();

    // Internal dropDown
    public Dropdown dropDown;

    // Delegate
    public delegate void MoveTypeDropDownDelegate(int t);
    public static event MoveTypeDropDownDelegate OnClick;
```

Figura 229 - MoveTypeDropdown. Variables.

La clase cuenta con: una lista de variables tipo "string" que corresponde con las opciones mostradas al usuario, la variable tipo "Dropdown" a gestionar y un evento "OnClick" que se activa cuando el usuario seleccione alguna opción.

```
    // Start is called before the first frame update
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();

        // Add the different options
        options.Add("MoveL");
        options.Add("MoveJ");
        options.Add("MoveC");

        // Adds the options to the dropdown options
        foreach (var option in options)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text = option });
        }
        dropDown.RefreshShownValue();
    }
```

Figura 230 - MoveTypeDropdown. Método Start.

El método *Start* inicializa el "Dropdown" borrando las opciones que tuviera anteriormente y almacenando unas nuevas, que corresponden con los tipos de movimientos disponibles.

```
    public void Click()
    {
        if (OnClick != null)
            OnClick.Invoke(dropDown.value);
    }
```

Figura 231 - MoveTypeDropdown. Método Click.

El método "Click" se debe configurar en el inspector en el elemento "Dropdown" correspondiente.

7.5.2.9.7 Clase PrecisionDropdown

La clase *PrecisionDropdown* gestiona el elemento "Dropdown" que permite seleccionar el nivel de precisión.

```
    public List<string> options = new List<string>();

    // Internal dropDown
    public Dropdown dropDown;
```

Figura 232 - PrecisionDropdown. Variables.

La clase almacena una lista de variables tipo "string" que corresponde a las opciones mostradas en escena y el objeto tipo "Dropdown" que gestiona.

```
    // Start is called before the first frame update
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();

        // Add the different options
        options.Add("fine");
        options.Add("z0");
        options.Add("z1");
        options.Add("z5");
        options.Add("z10");
        options.Add("z15");
        options.Add("z20");
        options.Add("z30");
        options.Add("z40");
        options.Add("z50");
        options.Add("z60");
        options.Add("z80");
        options.Add("z100");
        options.Add("z150");
        options.Add("z200");

        // Adds the options to the dropdown options
        foreach (var option in options)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text = option });
        }
        dropDown.RefreshShownValue();
    }
```

Figura 233 - PrecisionDropdown. Método Start.

El método *Start* inicializa el "Dropdown" borrando las opciones que tuviera anteriormente para crearle unas nuevas que corresponden con las que están disponibles.

7.5.2.9.8 Clase SpeedDropdown

La clase *SpeedDropdown* gestiona el elemento "Dropdown" que permite seleccionar el nivel de velocidad.

```
    public List<string> options = new List<string>();

    // Internal dropDown
    public Dropdown dropDown;
```

Figura 234 - SpeedDropdown. Variables.

La clase almacena una lista de variables tipo "string" que corresponde a las opciones mostradas en escena y el objeto tipo "Dropdown" que gestiona.

```
    // Start is called before the first frame update
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();

        // Add the different options
        options.Add("v5");
        options.Add("v10");
        options.Add("v20");
        options.Add("v30");
        options.Add("v40");
        options.Add("v50");
        options.Add("v60");
        options.Add("v80");
        options.Add("v100");
        options.Add("v150");
        options.Add("v200");
        options.Add("v300");
        options.Add("v400");
        options.Add("v500");
        options.Add("v600");
        options.Add("v800");
        options.Add("v1000");
        options.Add("v1500");
        options.Add("v2000");
        options.Add("v2500");

        // Adds the options to the dropdown options
        foreach (var option in options)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text = option });
        }
        dropDown.RefreshShownValue();
    }
```

Figura 235 - SpeedDropdown. Método Start.

El método *Start* inicializa el "Dropdown" borrando las opciones que tuviera anteriormente para crearle unas nuevas que corresponden con las que están disponibles.

7.5.2.9.9 Clase RobTargetDropdown

La clase *RobTargetDropdown* gestiona el elemento "Dropdown" que permite seleccionar los *RobTarget* creados para uso en la generación de instrucciones.

```
    // Public variables
    public List<string> options = new List<string>();

    public RobTargetList targetList;

    // Internal dropDown
    public Dropdown dropDown;
```

Figura 236 - RobTargetDropdown. Variables.

La clase almacena una lista de variables tipo "string" que corresponde a las opciones mostradas en escena, el objeto tipo "Dropdown" que gestiona y la lista de *RobTarget* creados.

```
    // Start/Destroy
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();
        RobTargetList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        RobTargetList.OnChanged -= HandleOnChanged;
    }
```

Figura 237 - RobTargetDropdown. Métodos de Unity.

El método *Start* inicializa el "Dropdown" borrando las opciones que tuviera anteriormente y subscribiéndose a los eventos producidos en la clase *RobTargetList*.

```
    // HandleOnChange
    void HandleOnChanged(RobTargetList targetList)
    {
        if (this.targetList == targetList)
            Prime(targetList);
    }
```

Figura 238 - RobTargetDropdown. Método HandleOnChange.

El método "HandleOnChange" llama al método "Prime" que se explica a continuación al producirse un evento en la clase *RobTargetList*.

```
    // Initializer method
    public void Prime(RobTargetList robTargetList)
    {
        dropDown.ClearOptions();

        this.targetList = robTargetList;
        foreach (RobTarget target in this.targetList.targets)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text =
target.displayName });
        }
        dropDown.RefreshShownValue();
    }
```

Figura 239 - RobtArgetDropdown. Método Prime.

El método "Prime" de esta clase se ha creado para actualizas las opciones disponibles para el usuario cada vez que se crea o destruye un *RobTarget*.

Capítulo 8. PRESUPUESTO

SUMAS PARCIALES

En esta sección se exponen todos elementos que han sido necesarios para la realización del proyecto teniendo en cuenta cantidades y costes unitarios.

Hardware	Cantidad (uds.)	Coste unitario (€/ud.)	Coste total (€)
Ordenador compatible con la realidad virtual	1	1 500	1 500
Oculus Rift S	1	449	449
		Total:	1 949

Tabla 2 - Suma parcial de hardware

En la *Tabla 2* se hace una estimación aproximada del coste del hardware necesario para la realización del proyecto. El coste del Hardware de realidad virtual se encuentra en la página web del vendedor [**30**].

Para el desarrollo del proyecto no ha sido necesario acceder al brazo robótico, ni es necesario su uso para aprovechar el programa diseñado, por lo que no se ha añadido. En caso de añadirlo no cambiaría el precio puesto que lo aporta la universidad.

Software	Cantidad (uds.)	Coste unitario (€/ud.)	Coste total (€)
RobotStudio	1	1 500	1 500
Unity	1	Gratuito	0
Autodesk Fusion 360	1	Gratuito	0
		Total:	1 500

Tabla 3 - Suma parcial de software

En la *Tabla* 3 aparece reflejado los costes por el software empleado. Puesto que se han usado licencias gratuitas para la mayor parte, solo representa un coste la propia aplicación,

RobotStudio. El coste de la licencia es anual pero el proyecto se ha desarrollado en menos de un año.

	Cantidad (horas)	Coste unitario (€/h)	Coste total (€)
Trabajo	540	20	10 800
		Total:	10 800

Tabla 4 - Suma parcial de Ingeniería

PRESUPUESTO GENERAL

En esta sección aparece reflejado el coste total de las secciones principales de gastos. El total sube a 14.249 euros.

	Coste (€)
Hardware	1 949
Software	1 500
Trabajo	10 800
Total:	14 249

Tabla 5 - Presupuesto general desglosado

Capítulo 9. OBJETIVOS DE DESARROLLO SOSTENIBLE

*Figura 240 - Objetivos de desarrollo sostenible [**17**].*

La Asamblea General de la ONU adoptó para el año 2030 los objetivos planteados en la *Figura 240.*

El proyecto desarrollado está enfocado en el desarrollo tecnológico para la industria como se explica en el Capítulo 4. , lo cual lo relaciona directamente con el punto 9 de la lista.

Capítulo 10. ANÁLISIS DE RESULTADOS

10.1 FUNCIONAMIENTO DEL PROGRAMA OBTENIDO

Se ha desarrollado un sistema que funciona con la interacción de dos programas independientes que usan lenguajes de programación distintos.

RobotStudio con lenguaje de programación RAPID y *Unity* con Csharp.

El usuario puede programar en *Unity* el sistema que desee y una vez termine sincronizar la información con *RobotStudio*. Una vez está la información necesaria sincronizada el usuario puede iniciar la simulación y controlarla desde *Unity*. Todo esto se hace a través de una interfaz de usuario desarrollada desde cero y que busca asemejarse a la de *RobotStudio* para facilitar su aprendizaje.

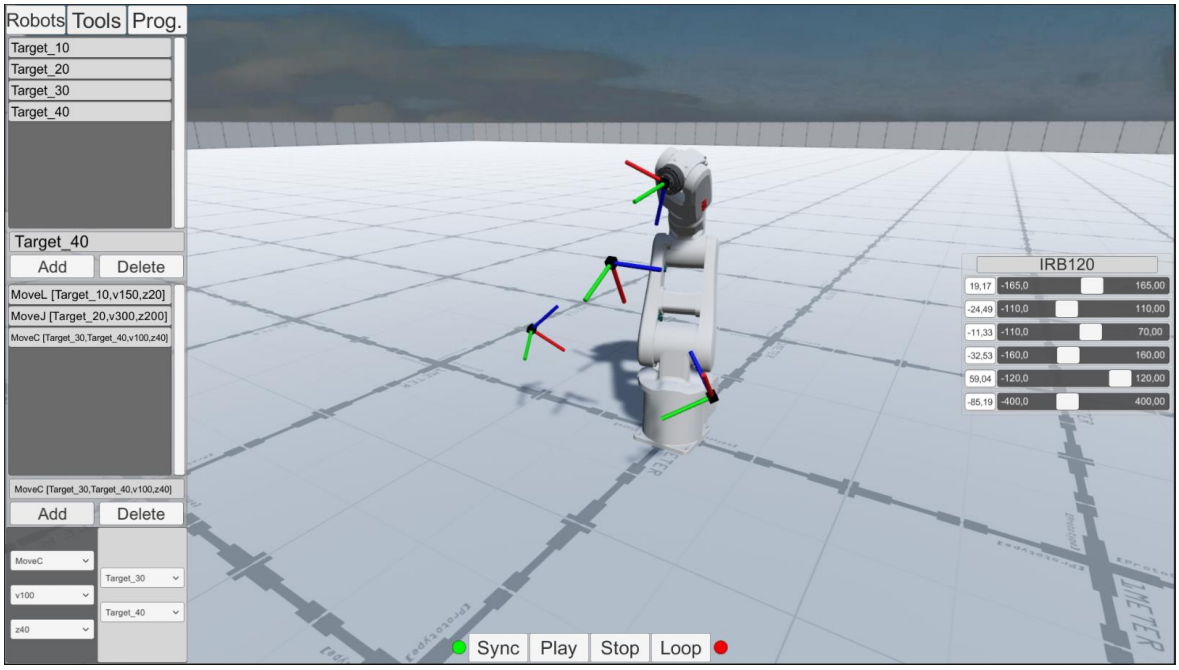

Figura 241 - Visual de la estación desarrollada.

La información que se sincroniza entre *RobotStudio* y *Unity* es:

- Puntos definidos en el espacio
- Comandos de movimiento a elegir entre tres tipos:
  - *MoveL*
  - *MoveJ*
  - *MoveC*
- Posición del robot al iniciar la sincronización.
- Si el programa se quiere realizar en bucle indefinidamente o ejecutarse una única vez.

Una vez iniciada la simulación, el usuario puede pausarla o detenerla en cualquier momento, pero el robot no dejará de moverse hasta no terminar de ejecutar la última acción. Esto se debe a como está programado *RobotStudio* y no tiene fácil solución.

Cuando se desee salir del programa, el usuario puede abrir el *Menú de salida* y darle a "Exit", que finalizará la ejecución de *RobotStudio* y Unity de forma controlada.

10.2 LIMITACIONES DEL PROGRAMA

Durante el desarrollo del programa se han encontrado y superado una gran variedad de obstáculos. El más difícil ha sido lidiar con el lenguaje de programación RAPID. Su limitación a la hora de usar variables y poder compartirlas entre las diferentes tareas y módulos ha dificultado la programación. Además, aunque *RobotStudio* añade la posibilidad de manejar una gran cantidad de errores que se pueden producir durante la simulación de alguno de sus procesos, no te permite crearlos para aquellos que no lo tienen.

Al usar la instrucción *MoveC*, en caso de haberse programado mal los puntos de la instrucción y no poder ejecutarse correctamente detendrán la ejecución del código de *RobotStudio* de forma repentina y sin poder notificárselo al programa de *Unity*.

Para evitar esto hay que tener en cuenta los puntos a continuación:

- El punto inicial y los dos puntos definidos en la instrucción tienen que definir una trayectoria circular
- El punto inicial tiene que estar a una distancia mayor de 0.1mm del primer punto de la instrucción.
- El punto inicial tiene que estar a una distancia mayor de 0.1mm del segundo punto de la instrucción.
- El primer punto de la instrucción tiene que estar a una distancia mayor de 0.1mm del segundo punto de la instrucción.
- El giro circular debe ser menor de 240 grados.

Además, se ha creado una lista con los errores que se pueden producir en la ejecución y que se han podido manejar en el código en el *ANEXO III: Lista de errores*

Capítulo 11. CONCLUSIONES Y TRABAJOS FUTUROS

11.1 CONCLUSIONES

Como reflexión final del proyecto es menester hablar de los objetivos cumplidos y las observaciones relevantes sobre las tecnologías empleadas.

Uno de los principales objetivos que se tuvo en mente para las especificaciones del programa fue resolver limitaciones presentes en el programa anterior, que se detenía después de cada movimiento sin tener en cuenta la precisión que se le había configurado. Este se ha logrado superar y se ha obtenido un programa base que tiene un gran margen de mejora al tener unos cimientos sólidos.

Se ha podido comprobar que la realidad virtual tiene mucho que aportar en el mundo de la industria para agilizar la creación y simulación de procesos. Un aspecto en el que programas como el desarrollado en este proyecto puede ser especialmente relevante es en el diseño de una fábrica y su ensayo para verificar el rendimiento y correcto funcionamiento sin haberla tenido que construir físicamente.

Es por esto por lo que según mejore la tecnología es muy probable que la realidad virtual o aumentada aparezcan en más sectores tecnológicos aportando sistemas mucho más intuitivos y realistas.

11.2 TRABAJOS FUTUROS

Algunos de los próximos objetivos para tener en cuenta para futuras versiones del programa serían:

- Creación de módulo de realidad virtual, puesto que por la cuarentena no se tuvo acceso al hardware y no pudo desarrollarse. Concretamente:
    - Interfaz sujeta a los mandos de realidad virtual con facilidad para cambiar de menú de opciones.
    - Permitir agarrar con los mandos diferentes partes del robot o el propio robot y modificar su posición y ángulo de giro.
    - Sistema de movimiento que permite cambiar entre un usuario realista y una especie de modo fantasma que dé la posibilidad de moverse libremente por la planta para poder enfocar la simulación desde diferentes ángulos o tener mayor precisión a la hora de colocar el brazo robótico.
- Añadir un módulo que permita recrear estaciones de trabajo o añadir modelos 3D de plantas ya creadas.
- Añadir más herramientas para una mayor variedad de funcionamiento. Esto implica tener en cuenta sus posibles interacciones con el entorno.
- Si es posible, añadir un nuevo tipo de movimiento a la hora de posicionar el robot además de colocar los ejes por separado. Este sería por ejemplo el que requiere de *RobotStudio* para moverse alrededor del *TCP* de la herramienta.

Capítulo 12. REFERENCIAS

[1] Epic Games, Inc. (2020) Unreal Engine Website. [Online]. https://www.unrealengine.com/en-US/

[2] Unity Technologies. (2020) Unity Main Page. [Online]. https://unity.com/es

[3] ABB Asea Brown Boveri Ltd. (2020) ABB RobotStudio Website. [Online]. https://new.abb.com/products/robotics/es/robotstudio

[4] Francisco Javier Domínguez Sánchez-Girón, "APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS," ICAI, Universidad Pontificia de Comillas, Madrid, Trabajo fin de grado 2019.

[5] Fundación Wikimedia, Inc. (2019, Noviembre) Wikipedia. [Online]. https://es.wikipedia.org/wiki/Tarjeta_perforada

[6] Teseo. (2017, Marzo) Teseo. [Online]. https://teseo.es/noticias/aplicaciones-y-usos-de-la-realidad-virtual/

[7] Tecnología para los negocios. (2019, Abril) ticnegocios. [Online]. https://ticnegocios.camaravalencia.com/servicios/tendencias/aplicaciones-de-la-realidad-virtual-en-pymes/

[8] Wikipedia Inc. (2020) Transmission Control Protocol. [Online]. https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[9] Wikipedia Inc. (2020) User Datagram Protocol. [Online]. https://en.wikipedia.org/wiki/User_Datagram_Protocol

[10] ABB Asea Brown Boveri Ltd. (2019, Sep.) ABB IRB120 Product Library Website. [Online]. https://library.e.abb.com/public/7139d7f4f2cb4d0da9b7fac6541e91d1/3HAC035960%20PS%20IRB%20120-en.pdf

[11] Autodesk Inc. (2020) Autodesk Fusion 360 Product Website. [Online]. https://www.autodesk.com/products/fusion-360/overview

[12] Rachel Gordon. (2017, October) MIT News. [Online]. http://news.mit.edu/2017/mit-csail-new-system-teleoperating-robots-virtual-reality-1009

[13] Andrzej Burghardt et al. (2020, Enero) MDPI. [Online]. https://www.mdpi.com/2076-3417/10/2/486/htm

[14] S.K. Ong, A.W.W. Yew, N.K. Thanigaivel, and A.Y.C. Nee. (2019, June) ScienceDirect. [Online]. https://www.sciencedirect.com/science/article/pii/S0736584519300250#!

[15] RobotWorx a SCOTT company. RobotWorx. [Online]. https://www.robots.com/blogs/abb-offers-vr-integration-for-robot-programming

[16] Microsoft. (2020) System.Globalization Namespace. [Online]. https://docs.microsoft.com/es-es/dotnet/api/system.globalization?view=netcore-3.1

[17] Microsoft. (2020) System.Net Namespace. [Online]. https://docs.microsoft.com/es-es/dotnet/api/system.net?view=netcore-3.1

[18] Microsoft. (2020) System.Net.Socket. [Online]. https://docs.microsoft.com/es-es/dotnet/api/system.net.sockets?view=netcore-3.1

[19] Microsoft. (2020) System.Threading. [Online]. https://docs.microsoft.com/es-es/dotnet/api/system.threading?view=netcore-3.1

[20] Unity Technologies. (2020) Unity Scripting Reference. [Online]. https://docs.unity3d.com/ScriptReference/index.html

[21] Microsoft. (2020) try-catch. [Online]. https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/try-catch

[22] Unity Technologies. (2020) Scripting API CharacterController. [Online]. https://docs.unity3d.com/ScriptReference/CharacterController.html

[23] Unity Technologies. (2020) Unity User Interface. [Online]. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html

[24] Unity Technologies. (2020) Unity Button. [Online]. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-Button.html

[25] Unity Technologies. (2020) Unity Dropdown. [Online]. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-Dropdown.html

[26] Unity Technologies. (2020) Unity Slider. [Online]. https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-Slider.html

[27] Craig Perko. (2015, June) Youtube. UI Lists. [Online]. https://www.youtube.com/watch?v=GpPWbM_6DVY&list=PLW2i42bgplOkhM1iuOecfGyQWd_ymKap5&index=1

[28] Microsoft. (2017, Mar.).Net Modelo de diseño del observador. [Online]. https://docs.microsoft.com/es-es/dotnet/standard/events/observer-design-pattern

[29] Microsoft. (2017, Mar.).Net Controlar y provocar eventos. [Online]. https://docs.microsoft.com/es-es/dotnet/standard/events/

[30] Facebook Technologies, LLC. (2020, Sep.) Oculus Rift S. [Online]. https://www.oculus.com/rift-s/

[31] Naciones Unidas. (2020, Sep.) Objetivos de desarrollo sostenible. [Online]. https://www.un.org/sustainabledevelopment/es/2015/09/la-asamblea-general-adopta-la-agenda-2030-para-el-desarrollo-sostenible/

[32] ABB. (2017, Oct.) Technical Reference Manual. [Online]. https://library.e.abb.com/public/b227fcd260204c4dbeb8a58f8002fe64/Rapid_instructions.pdf?x-sign=f79v/883X1nHGc8fqH+WAJ2F30y/M6TZfYUuPuQpP+jeMBygouyGg+WSj8A9Otry#page=1741&zoom=100,200,154

[33] ABB Asea Brown Boveri Ltd. (2020) ABB Main Website. [Online]. https://new.abb.com/es

[34] Carlos Álvarez Vereterra, "USO DE LA REALIDAD VIRTUAL PARA EL ENTRENAMIENTO DEL PERSONAL DE OPERACIÓN Y MANTENIMIENTO: APLICACIÓN A LA MINI-FABRICA ICAI," ICAI, Universidad Pontificia Comillas, Madrid, Trabajo fin de máster.

[35] Mario Serrano Rodríguez, "APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS INDUSTRIALES DE GRANDES DIMENSIONES," ICAI, Universisdad Pontificia Comillas, Madrid, Trabajo fin de grado 2019.

[36] C00pala. (2017, May) Youtube. Unity UI - Scroll Menu. [Online]. https://www.youtube.com/watch?v=ZI6DwJtjlBA&t=1s

[37] Microsfot. (2020) Explorador de API.Net. [Online]. https://docs.microsoft.com/es-es/dotnet/api/?view=netcore-3.1

ANEXO I: CONCEPTOS BÁSICOS DE UNITY

En esta sección se explican variables y términos propios del entorno de Unity que ayudarán a la hora de comprender y replicar el trabajo realizado. [**20**]

OBJECT

Clase base de todos los objetos contenidos en Unity. Cualquier variable pública que derive de la clase Object se mostrará en el inspector de Unity.

DESTROY

Método que elimina un objeto.

INSTANTIATE

Clona el objeto original y crea una copia de éste. Se usa normalmente con *Prefab*.

COMPONENT

Clase base para todo aquello adherido a un *GameObject* en el inspector de Unity.

BEHAVIOUR

Clase que deriva de *Component*. Es un *Component* pero puede activarse y desactivarse.

MONOBEHAVIOUR

Es la clase base de la que casi todos los códigos de Unity derivan. Si se desarrolla en C# hay que poner explícitamente que deriva de esta clase. Algunos de los métodos más relevantes que contiene esta clase son: *Awake*, *Start*, *Update*, *FixedUpdate*, *OnDestroy* e *Invoke*.

AWAKE

Awake se llama cuando la instancia del código se está cargando.

Se llama a Awake cuando un *GameObject* activo que contiene el código se inicializa cuando se carga una escena, o cuando un *GameObject* previamente inactivo se pone en activo o después de que se inicialice un *GameObject* creado con *Instantiate*. Awake se utiliza para inicializar las variables o estados antes de que se inicie la aplicación.

START

Start se llama cuando el código es activado, justo antes que cualquier llamada a el método *Update*.

Como *Awake*, Start es llamado una vez en toda la vida del código. Sin embargo, *Awake* es llamado cuando el objeto del código es inicializado se haya activado o no el objeto. Por tanto, no tienen por qué llamarse en el mismo fotograma.

UPDATE

Si el *MonoBehaviour* está activo se llama una vez por fotograma.

FIXEDUPDATE

Método independiente de la tasa de fotogramas y que se emplea para el cálculo de físicas de Unity. Habitualmente se llama a este método cincuenta veces por segundo.

ONDESTROY

Método llamado al destruirse el objeto correspondiente o finalizar el programa o escena.

INVOKE

Método que permite llamar a un método concreto en un tiempo determinado.

VECTOR3

Variable que sirve para representar puntos y direcciones en tres dimensiones

QUATERNION

Son una forma de representar rotaciones con un vector de dimensión 4 normalizado. No sufre de bloqueo de Cardán o "Gimbal lock" y es compacto. Unity usa internamente Quaternions para representar todas las rotaciones.

GAMEOBJECT

Clase base de todas las entidades de Unity. Deriva de la clase *Object*.
Sirve para obtener la información de cualquier entidad en la escena activa u otra escena. Algunas de las propiedades y/o componentes relevantes de un *GameObject* son *Layer* y *Transform*.

GETCOMPONENT

Método que permite obtener un componente concreto que este adherido a un *GameObject*.

LAYER

Capa en la que se encuentra el *GameObject*. Sirve para controlar aquello que se renderiza y si es afectado por un *RayCast*.

RENDERER

Clase que permite que cualquier objeto se muestre en pantalla. Es necesario para renderizar objetos, mallas o partículas. Si se desactiva el objeto se vuelve invisible. También permite cambiar el material que tiene asignado un objeto modificando así su apariencia.

TRANSFORM

Clase que deriva de la clase *Component*. Permite modificar la posición, rotación y escala de un objeto. Todos los objetos en una escena tienen un componente *Transform* adherido.

Se puede aprovechar la jerarquía de objetos de Unity para vincular la posición, rotación y escala de los objetos. De esta manera un objeto puede tener una posición global, respecto al eje de coordenadas del mundo y una posición local respecto al *GameObject* padre.

Algunas de sus propiedades más relevantes son: *Position*, *EulerAngles* y *Rotation* con sus respectivas versiones locales, así como *Forward*, *Right* y *Up* que son vectores normalizados de los ejes del componente *Transform* en el eje de coordenadas del mundo.

POSITION

Posición de un objeto en los ejes de coordenadas del mundo.

LOCALPOSITION

Posición de un objeto respecto a su padre.

EULERANGLES

Rotación del objeto expresado en ángulos de Euler y en grados.

LOCALEULERANGLES

Rotación de un objeto expresado en ángulos de Euler y en grados respecto a su padre.

ROTATION

Un *Quaternion* que almacena la rotación de un objeto respecto a los ejes de coordenadas del mundo.

LOCALROTATION

Un *Quaternion* que almacena la rotación de un objeto respecto a su padre.

PARENT

Padre del *Transform*.

FORWARD

Devuelve un vector normalizado que representa el eje azul del *Transform* en los ejes de coordenadas del mundo.

RIGHT

Devuelve un vector normalizado que representa el eje rojo del *Transform* en los ejes de coordenadas del mundo.

UP

Devuelve un vector normalizado que representa el eje verde del *Transform* en los ejes de coordenadas del mundo.

SETPARENT

Método que establece el padre del *Transform*.

PREFAB

El sistema de prefabs de Unity permite almacenar *GameObject* junto con su configuración y componentes para poder reusarlos en la escena.

COLLIDER

Malla invisible que puede tener una forma geométrica simple o compleja y permite detectar colisiones entre diferentes objetos. No tiene por qué coincidir con los límites del *Renderer* adjunto al objeto. Todo *RigidBody* requiere de un *Collider* simple para poder funcionar correctamente.

RIGIDBODY

Clase base de todos los objetos ligados a físicas. Es la forma más sencilla de aplicar masa, rozamiento, inercia y colisiones a un objeto de forma realista.

En esta sección se explican variables y términos propios del entorno de Unity que ayudarán a la hora de comprender y replicar el trabajo realizado.

RAYCAST

Método propio del sistema de físicas de Unity que permite crear un rayo con un origen y longitud concretas pudiendo detectar el *Collider* de *GameObject* concretos dependiendo del *Layer* en el que se encuentran. Existen variaciones de este método usando esferas o cubos en vez de líneas.

CANVAS

Elemento básico de renderizado. Todos los componentes de una interfaz deben tener un canvas como componente o ser hijo de un *GameObject* con uno.

ANEXO II: CONCEPTOS BÁSICOS DE RAPID

En esta sección se explican variables y términos propios del entorno de *RobotStudio* y el lenguaje de programación RAPID que ayudarán a la hora de comprender y replicar el trabajo realizado. [**32**]

DEFINICIÓN DE VARIABLES

En el lenguaje RAPID existen tres formas de definir las variables:

CONSTANTES

Representan datos de un valor fijo a los que no se puede reasignar un nuevo valor. Se declaran precedidos del término *CONS*

VARIABLES

Representan datos a los que se les puede asignar nuevos valores durante la ejecución del programa. Se declaran precedidos del término *VAR*

PERSISTENTES

Representan variables que al cambiar su valor durante la ejecución también se modifica su valor de inicialización. De esta manera se conservan los datos con cada nueva ejecución. Se declaran precedidos del término *PERS*

MODULE

Elemento del lenguaje de programación RAPID que sirve para fragmentar el código en archivos que pueden almacenar variables y funciones. Cada *Task* debe tener al menos un módulo, ya sea de sistema o de programa.

TASK

Una tarea se puede considerar como un hilo de ejecución de programa. Teniendo varias tareas se pueden estar ejecutando varios procesos a la vez. Cada robot activo tiene su propia tarea. Se pueden añadir más tareas sin estar ligadas a robots y suelen ser útiles a la hora de tener sistemas que dependen de un sistema comunicaciones o requieren de una subrutina para ejecutarse correctamente.

ROBTARGET

Variable que define la posición del robot y de sus ejes externos

SPEEDDATA

Variable que especifica la velocidad a la que se moverán los ejes externos del robot.

ZONEDATA

Variable que especifica como debe terminarse en una posición. Sirve para definir la precisión mínima que se busca a la hora de acercarse a un punto objetivo.

JOINTTARGET

Variable que almacena el ángulo de giro de cada eje del robot.

CONFDATA

Variable que define la configuración de los ejes del robot. Es un entero que tiene valor 0 cuando el ángulo del eje del robot va de 0 a 90 grados, 1 de 90 a 180 grados, etc.

Si el ángulo es menor que 0 toma valores negativos, -1 de 0 a -90, -2 de -90 a -180, etc.

MOVEL

Comando de *RobotStudio* que permite mover el robot con un desplazamiento lineal al especificar el punto objetivo, la velocidad, la precisión y la herramienta a utilizar.

MOVEJ

Comando de *RobotStudio* similar a *MoveL* pero que en vez de producir un desplazamiento lineal realizará el movimiento que requiera de menos carga computacional para alcanzar el punto objetivo.

MOVEC

Comando de *RobotStudio* similar a *MoveL* pero más complejo ya que requiere de dos puntos al definirse y necesita que estos y el punto inicial cumplan unos requisitos concretos para ejecutarse sin errores.

TCP

Tool Center Point. Punto de la herramienta que definen la posición y orientación del robot.

ANEXO III: LISTA DE ERRORES

Lista de los errores que se pueden producir en la ejecución del programa:

ERROR 001: LIMITE DEL ROBOT

La posición es alcanzable, pero se el robot se encuentra fuera de rango en alguno de sus ejes.

ERROR 002: ROBOT FUERA DE RANGO

La posición se encuentra fuera del rango de trabajo del robot.

ERROR 003: NÚMERO DE PUNTOS Y/O MOVIMIENTOS MÁXIMOS SUPERADO

Se han programado en Unity un número mayor de puntos o instrucciones del que se ha permitido a RobotStudio tener

ERROR 004: NO FUE POSIBLE ALCANZAR LA POSICIÓN INICIAL DEL ROBOT

Al sincronizar la información de Unity con RobotStudio no se pudo alcanzar la posición del robot en Unity en el entorno de RobotStudio.

ERROR 005: OPCIÓN DE SINCRONIZACIÓN ERRÓNEA

Error producir en RobotStudio durante la sincronización si el mensaje recibido no corresponde con ninguna de las opciones programadas.

ERROR 006: OPCIÓN DE SIMULACIÓN ERRÓNEA

Error producir en RobotStudio durante la simulación si el mensaje recibido no corresponde con ninguna de las opciones programadas.

ERROR 007: OPCIÓN GENÉRICA ERRÓNEA

Error producir en RobotStudio si el mensaje recibido no corresponde con ninguna de las opciones programadas en la selección inicial de opciones.

ERROR 008: NO SE HA PODIDO RECIBIR ÁNGULOS PARA LOS EJES DEL ROBOT

Error producido en el protocolo UDP desarrollado en Unity si no se ha recibido mensaje desde RobotStudio cuando se ha iniciado la simulación.

ERROR 009: NO HAY PUNTOS CREADOS

Error producido en la sincronización con RobotStudio si no hay puntos que sincronizar.

ERROR 010: NO HAY INSTRUCCIONES PROGRAMADAS

Error producido en la sincronización con RobotStudio si no hay instrucciones que sincronizar.

ERROR 011: NO SE HA PODIDO ENVIAR UN MENSAJE A ROBOTSTUDIO

Error producido en el protocolo TCP desarrollado en Unity si por algún motivo no se ha podido enviar mensaje a RobotStudio.

ERROR 012: NO SE HA RECIBIDO MENSAJE DE ROBOTSTUDIO

Error producido en el protocolo TCP desarrollado en Unity si por algún motivo no se ha podido recibir mensaje de RobotStudio.

ERROR 013: NO HA SIDO POSIBLE CONECTARSE CON ROBOTSTUDIO

Error al intentar conectarse con RobotStudio, normalmente producido porque el programa de RobotStudio no ha sido iniciado.

ERROR 014: SIN CONEXIÓN CON ROBOTSTUDIO

Error producido al intentar iniciar la sincronización sin estar conectado el programa a RobotStudio.

ERROR 015: NO HAY ROBOTS CREADOS

Error al intentar añadir un *Robtarget* o eliminar un robot cuando no se ha añadido ninguno a la escena.

ERROR 016: NO HAY PUNTOS QUE ELIMINAR

Error producido al intentar eliminar un *Robtarget* cuando no se ha creado ninguno antes.

ERROR 017: NO HAY ROBOT SELECCIONADO

Error producido si se trata de añadir un tipo de robot a la escena sin haber seleccionado ninguno de la librería antes.

ERROR 018: NO HAY HERRAMIENTA SELECCIONADA

Error producido si se trata de añadir un tipo de herramienta a la escena sin haber seleccionado ninguno de la librería antes.

ERROR 019: NO HAY HERRAMIENTAS CREADAS

Aviso del sistema que se produce al tratar de eliminar, vincular o desvincular herramientas sin haberse creado una antes.

ERROR 020: NO HAY INSTRUCCIONES QUE ELIMINAR

Error producido si se trata de eliminar una instrucción sin haberse creado una antes.

ERROR 021: EL ROBOT TIENE MENOS EJES

Error producido si al tratar de colocar un ángulo concreto de un robot se excede el número máximo de ejes.

ERROR 022: TIPO DE MOVIMIENTO ERRÓNEO

Error causado al crear una instrucción si se selecciona un tipo de movimiento que no está programado.

ERROR 023: NIVEL DE VELOCIDAD INEXISTENTE

Error causado al crear una instrucción si se selecciona un nivel de velocidad que no está programado

ANEXO IV: CÓDIGO FUENTE DE ROBOTSTUDIO

MAINMODULE

```
MODULE MainModule
    !***********************************************************
    !
    ! Module:  MainModule
    !
    ! Descripction:
    !   Module in charge of the execution of the simulation
    !
    ! Author: Pablo Giménez Suárez
    !
    ! Versión: 1.0
    !
    !***********************************************************


    !***********************************************************
    !
    ! MainModule main
    !
    !***********************************************************
    PERS robtarget tTarget_1:=[[0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0,0,0]];
    PERS robtarget tTarget_2:=[[0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0,0,0]];
    PERS speeddata spd:=[0,0,0,0];
    PERS zonedata zn:=[FALSE,0,0,0,0,0,0];

    PROC main()

        ! With confJ off the robot will go to the nearest possible conf
        ConfJ\Off;
        ConfL\Off;
        CornerPathWarning(FALSE);

        WHILE (NOT abort) DO

            IF setPos THEN
                MoveAbsJ actTargetPos,v500,fine,tool0;
                WaitRob\InPos;
                setPos:=FALSE;
            ENDIf

            WHILE actSim DO

                WHILE (NOT holdSim) DO
                    canPause:=FALSE;
```

```
                tTarget_1:=targets{pPoints};
                IF pPoints<nPoints THEN
                    tTarget_2:=targets{pPoints+1};
                ENDIF
                spd:=speed{pMoves};
                zn:=zone{pMoves};

                TEST moveType{pMoves}

                CASE 0:
                    MoveL tTarget_1,spd,zn,tool0;
                    pPoints:=pPoints+1;
                    pMoves:=pMoves+1;

                CASE 1:
                    MoveJ tTarget_1,spd,zn,tool0;
                    pPoints:=pPoints+1;
                    pMoves:=pMoves+1;

                CASE 2:
                    MoveC tTarget_1,tTarget_2,spd,zn,tool0;
                    pPoints:=pPoints+2;
                    pMoves:=pMoves+1;

                DEFAULT:

                ENDTEST

                IF ((actLoop) AND (pMoves>nMoves)) THEN
                    pMoves:=1;
                    pPoints:=1;
                ELSEIF (pMoves>nMoves) THEN
                    holdSim:=TRUE;
                    actSim:=FALSE;
                ENDIF

            ENDWHILE
            WaitRob\ZeroSpeed;
            canPause:=TRUE;

        ENDWHILE

    ENDWHILE

    ! Error handling
ERROR
    IF ERRNO=ERR_ROBLIMIT THEN
        SkipWarn;
        errorNum:=1;
        holdSim:=TRUE;
        actSim:=FALSE;
        TRYNEXT;
```

```
        ELSEIF ERRNO=ERR_OUTSIDE_REACH THEN
            SkipWarn;
            errorNum:=2;
            holdSim:=TRUE;
            actSim:=FALSE;
            TRYNEXT;

        ENDIF

    ENDPROC

ENDMODULE
```

TCPMODULE

```
MODULE TCPModule
    !***********************************************************
    !
    ! Module:  TCPModule
    !
    ! Descripction:
    !   Module in charge of the main communications between Unity and RobotStudio
    !
    ! Author: Pablo Giménez Suárez
    !
    ! Version: 1.0
    !
    !***********************************************************

    ! Local Communication Variables
    VAR string key;
    VAR string k;

    VAR socketdev serverTCP;
    VAR socketdev clientTCP;

    CONST num portTCP:=5000;
    CONST string ip:="127.0.0.1";

    PERS string msgTCP;

    ! Variables to take specific numbers from the msg str
    PERS num tempV1;
    PERS num tempV2;
    PERS bool ok;
    PERS string tString;

    !***********************************************************
```

```
    !
    ! TCP Communication main
    !
    !*********************************************************
    PROC main_2()
        abort:=FALSE;
        ! TRUE if the program must end

        actSim:=FALSE;
        ! TRUE if simulation is active

        holdSim:=FALSE;
        ! TRUE if the simulation must be paused

        actSync:=FALSE;
        ! TRUE if sincronization is active

        actLoop:=FALSE;
        ! TRUE if there are loops

        setPos:=FALSE;

        ! SERVER INITIALIZATION
        ! Create server socket and bind it to ip address and port
        SocketCreate serverTCP;
        SocketBind serverTCP,ip,portTCP;
        SocketListen serverTCP;

        ! Accept incoming connections from server socket into client socket
        SocketAccept serverTCP,clientTCP,\Time:=WAIT_MAX;


        WHILE (NOT abort) DO
            msgTCP:=ReceiveMsgTCP();

            IF (StrLen(msgTCP)>0) THEN
                k:=StrPart(msgTCP,1,1);

                TEST k

                CASE "S":
                    ! Checks that the number of points and moves are less than
the limit  S;num;num; S;0;0;
                    tempV1:=StrFind(msgTCP,1,";");
                    tempV2:=StrFind(msgTCP,tempV1+1,";");
                    ok:=StrToVal(StrPart(msgTCP,tempV1+1,tempV2-tempV1-
1),nPoints);

                    tempV1:=tempV2;
                    tempV2:=StrFind(msgTCP,tempV1+1,";");
                    ok:=StrToVal(StrPart(msgTCP,tempV1+1,tempV2-tempV1-
1),nMoves);
```

```
                    pPoints:=1;
                    pMoves:=1;
                    ! Pointers to go through the vectors

                    IF nPoints<=maxNumOfPoints AND nMoves<=maxNumOfMoves THEN
                        ! Synchronize the robtargets and/or paths
                        actSync:=TRUE;

                        socketSend clientTCP\str:="R";
                        ! Ready to synchronize
                    ELSE
                        actSync:=FALSE;
                        socketSend clientTCP\str:="ERROR 003: Too many
points/moves. (max:" + numToStr(maxNumOfPoints,2) + "/" +
numToStr(maxNumofMoves,2) + ")";
                    ENDIF

                    WHILE actSync DO
                        msgTCP:=ReceiveMsgTCP();
                        IF (StrLen(msgTCP)>0) THEN
                            key:=StrPart(msgTCP,1,1);

                            TEST key

                            CASE "T":
                                ! Adds robtargets in order. T[num;num;num;
num;num;num;  num;num;num;num;]
                                tempV1:=StrFind(msgTCP,1,"[");
                                tempV2:=StrFind(msgTCP,1,"]");


targets{pPoints}:=StrToTarget(StrPart(msgTCP,tempV1+1,tempV2-tempV1-1));
                                pPoints:=pPoints+1;
                                SocketSend clientTCP\str:="C";

                            CASE "M":
                                ! Adds movement type, speeddata and precision
M[letter;num;num;]
                                tempV1:=StrFind(msgTCP,1,"[");
                                tempV2:=StrFind(msgTCP,1,"]");

                                ok:=StrToInstr(StrPart(msgTCP,tempV1+1,tempV2-
tempV1-1));

                                pMoves:=pMoves+1;
                                SocketSend clientTCP\str:="C";

                            CASE "L":
                                ! Tells if the program is in loop
                                actLoop:=NOT actLoop;
                                SocketSend clientTCP\str:="C";

                            CASE "A":
```

```
                                ! Sets the initial position of the robot
A[num,num,num,num,num,num]
                                tempV1:=StrFind(msgTCP,1,"[");
                                tempV2:=StrFind(msgTCP,1,"]");

                                ! This section should be modified in order to
scalate depending on the number of axis of each robot
                                tString:="["+StrPart(msgTCP,tempV1,tempV2-
tempV1+1)+",[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]]";

                                ok:=StrToVal(tString,actTargetPos);

                                IF ok THEN
                                    setPos:=TRUE;
                                    WaitUntil(NOT setPos),\MaxTime:=WAIT_MAX;
                                    SocketSend clientTCP\str:="C";
                                ELSE
                                    SocketSend clientTCP\str:="ERROR 004:
Couldn't set the initial position of the robot";
                                ENDIF

                            CASE "S":
                                ! Ends the sychronization
                                actSync:=FALSE;

                            CASE "N":
                                ! Nothing received

                            DEFAULT:
                                SocketSend clientTCP\str:="ERROR 005: Wrong
synchronization choice";

                            ENDTEST
                        ENDIF

                    ENDWHILE

                CASE "P":
                    ! Simulation
                    pPoints:=1;
                    pMoves:=1;
                    holdSim:=FALSE;
                    actSim:=TRUE;
                    SocketSend clientTCP\str:="S";

                    WHILE actSim DO
                        msgTCP:=ReceiveMsgTCP();
                        IF StrLen(msgTCP)>0 THEN
                            key:=StrPart(msgTCP,1,1);

                            TEST key

                            CASE "P":
```

```
                                ! Pauses/replay the simulation
                                holdSim:=NOT holdSim;
                                WaitUntil(canPause),\MaxTime:=WAIT_MAX;
                                SocketSend clientTCP\str:="P";


                            CASE "S":
                                ! Stops de simulation
                                holdSim:=TRUE;
                                WaitUntil(canPause),\MaxTime:=WAIT_MAX;
                                actSim:=FALSE;

                            CASE "N":
                                ! Nothing received

                            DEFAULT:
                                SocketSend clientTCP\str:=("ERROR 006: Wrong
simulation choice: " + key);

                            ENDTEST
                        ENDIF

                    ENDWHILE

                    ! Message to confirm the end of the simulation
                    SocketSend clientTCP\str:="E";

                CASE "E":
                    ! Ends the program
                    holdSim:=TRUE;
                    actSim:=FALSE;
                    abort:=TRUE;

                CASE "N":
                    ! Nothing received

                DEFAULT:
                    ! Wrong option
                    SocketSend clientTCP\str:="ERROR 007: Wrong generic option:"
+ k;

                ENDTEST
            ENDIF


            TEST errorNum

            CASE 1:
                SocketSend clientTCP\str:="ERROR 001: Robot limit. The position
is reachable, but outside joint/s limits.";
                errorNum:=0;

            CASE 2:
```

```
                SocketSend clientTCP\str:="ERROR 002: Robot out of reach. The
position is outside the working range.";
                errorNum:=0;

            ENDTEST

        ENDWHILE

        SocketSend clientTCP\str:="Y";
        SocketClose serverTCP;
        SocketClose clientTCP;
        stop;

    ENDPROC

    FUNC string ReceiveMsgTCP()
        VAR string tMsg;
        VAR rawbytes rawData;

        ! Receive robotTarget from Unity in string format
        SocketReceive clientTCP,\RawData:=rawData\Time:=2;
        UnpackRawBytes rawData,1,tMsg,\ASCII:=RawBytesLen(rawData);

        RETURN tMsg;

        ! Error handler
    ERROR
        IF ERRNO=ERR_FNCNORET THEN
            skipWarn;
            tMsg:="N";
            RETURN tMsg;

        ELSEIF ERRNO=ERR_SOCK_TIMEOUT THEN
            SkipWarn;
            TRYNEXT;

        ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
            SkipWarn;
            holdSim:=TRUE;
            actSim:=FALSE;
            abort:=TRUE;
            TRYNEXT;
        ENDIF
    ENDFUNC


    ! [FJD]
    FUNC string jointTargetToString(jointtarget target)
        VAR string tempString;
        tempString:=NumToStr(target.robax.rax_1,0);
        ConcatenateString tempString,target.robax.rax_2;
        ConcatenateString tempString,target.robax.rax_3;
        ConcatenateString tempString,target.robax.rax_4;
```

```
    ConcatenateString tempString,target.robax.rax_5;
    ConcatenateString tempString,target.robax.rax_6;
    RETURN tempString;
ENDFUNC

! [FJD]
PROC ConcatenateString(INOUT string inString,num value)
    inString:=inString+";"+NumToStr(value,0);
ENDPROC

! Function that returns moveType
FUNC num GetMove(string data)
    VAR num type;

    TEST data
    CASE "L":
        type:=0;

    CASE "J":
        type:=1;

    CASE "C":
        type:=2;
    DEFAULT:

    ENDTEST

    RETURN type;
ENDFUNC

! Function that returns speeddata variable type for movement commands
FUNC speeddata GetSpeed(string data_3)
    VAR speeddata tempSpeed;

    TEST data_3
    CASE "0":
        tempSpeed:=v5;

    CASE "1":
        tempSpeed:=v10;

    CASE "2":
        tempSpeed:=v20;

    CASE "3":
        tempSpeed:=v30;

    CASE "4":
        tempSpeed:=v40;

    CASE "5":
        tempSpeed:=v50;
```

```
        CASE "6":
            tempSpeed:=v60;

        CASE "7":
            tempSpeed:=v80;

        CASE "8":
            tempSpeed:=v100;

        CASE "9":
            tempSpeed:=v150;

        CASE "10":
            tempSpeed:=v200;

        CASE "11":
            tempSpeed:=v300;

        CASE "12":
            tempSpeed:=v400;

        CASE "13":
            tempSpeed:=v500;

        CASE "14":
            tempSpeed:=v600;

        CASE "15":
            tempSpeed:=v800;

        CASE "16":
            tempSpeed:=v1000;

        CASE "17":
            tempSpeed:=v1500;

        CASE "18":
            tempSpeed:=v2000;

        CASE "19":
            tempSpeed:=v2500;

        DEFAULT:

        ENDTEST


        RETURN tempSpeed;
    ENDFUNC

    ! Function that returns zonedata variable type for movement commands
    FUNC zonedata GetPrecision(string data_2)
        VAR zonedata tempPrec;
```

```
    TEST data_2
    CASE "0":
        tempPrec:=fine;

    CASE "1":
        tempPrec:=z0;

    CASE "2":
        tempPrec:=z1;

    CASE "3":
        tempPrec:=z5;

    CASE "4":
        tempPrec:=z10;

    CASE "5":
        tempPrec:=z15;

    CASE "6":
        tempPrec:=z20;

    CASE "7":
        tempPrec:=z30;

    CASE "8":
        tempPrec:=z40;

    CASE "9":
        tempPrec:=z50;

    CASE "10":
        tempPrec:=z60;

    CASE "11":
        tempPrec:=z80;

    CASE "12":
        tempPrec:=z100;

    CASE "13":
        tempPrec:=z150;

    CASE "14":
        tempPrec:=z200;

    DEFAULT:

    ENDTEST

    RETURN tempPrec;
ENDFUNC
```

```
FUNC bool StrToInstr(string data_1)
    !M[letter;num;num;] M[L;13;9;]
    VAR num pMove;
    VAR num pSpeed;
    VAR num pPrec;

    ! Position index
    pMove:=StrFind(data_1,1,";");
    pSpeed:=StrFind(data_1,pMove+1,";");
    pPrec:=StrFind(data_1,pSpeed+1,";");

    moveType{pMoves}:=GetMove(StrPart(data_1,1,pMove-1));
    speed{pMoves}:=GetSpeed(StrPart(data_1,pMove+1,pSpeed-pMove-1));
    zone{pMoves}:=GetPrecision(StrPart(data_1,pSpeed+1,pPrec-pSpeed-1));
    ok:=TRUE;
    RETURN ok;
ENDFUNC

! Modification of [FJD] code
FUNC robtarget StrToTarget(string data)
    VAR robtarget tempTarget;
    VAR bool bResult;

    VAR num pX;
    VAR num pY;
    VAR num pZ;

    VAR num tempRX;
    VAR num pRX;
    VAR num tempRY;
    VAR num pRY;
    VAR num tempRZ;
    VAR num pRZ;

    VAR num pcf1;
    VAR num pcf4;
    VAR num pcf6;
    VAR num pcfx;

    ! Position index
    pX:=StrFind(data,1,";");
    pY:=StrFind(data,pX+1,";");
    pZ:=StrFind(data,pY+1,";");

    pRX:=StrFind(data,pZ+1,";");
    pRY:=StrFind(data,pRX+1,";");
    pRZ:=StrFind(data,pRY+1,";");

    pcf1:=StrFind(data,pRZ+1,";");
    pcf4:=StrFind(data,pcf1+1,";");
    pcf6:=StrFind(data,pcf4+1,";");
    pcfx:=StrFind(data,pcf6+1,";");
```

```
        ! Position data
        bResult:=StrToVal(StrPart(data,1,pX-1),tempTarget.trans.x);
        bResult:=StrToVal(StrPart(data,pX+1,pY-pX-1),tempTarget.trans.y);
        bResult:=StrToVal(StrPart(data,pY+1,pZ-pY-1),tempTarget.trans.z);

        ! Orientation data
        bResult:=StrToVal(StrPart(data,pZ+1,pRX-pZ-1),tempRX);
        bResult:=StrToVal(StrPart(data,pRX+1,pRY-pRX-1),tempRY);
        bResult:=StrToVal(StrPart(data,pRY+1,pRZ-pRY-1),tempRZ);
        tempTarget.rot:=OrientZYX(tempRZ,tempRY,tempRX);

        ! Configuration data
        bResult:=StrToVal(StrPart(data,pRZ+1,pcf1-pRZ-1),tempTarget.robconf.cf1);
        bResult:=StrToVal(StrPart(data,pcf1+1,pcf4-pcf1-
1),tempTarget.robconf.cf4);
        bResult:=StrToVal(StrPart(data,pcf4+1,pcf6-pcf4-
1),tempTarget.robconf.cf6);
        bResult:=StrToVal(StrPart(data,pcf6+1,pcfx-pcf6-
1),tempTarget.robconf.cfx);

        ! External axii data
        tempTarget.extax.eax_a:=9E+09;
        tempTarget.extax.eax_b:=9E+09;
        tempTarget.extax.eax_c:=9E+09;
        tempTarget.extax.eax_d:=9E+09;
        tempTarget.extax.eax_e:=9E+09;
        tempTarget.extax.eax_f:=9E+09;

        RETURN tempTarget;
    ENDFUNC

ENDMODULE
```

UDPMODULE

```
MODULE UDPModule
    !***********************************************************
    !
    ! Module:  UPDModule
    !
    ! Description:
    !   Module based on Carlos Álvarez Vereterra UDP communication code
    !
    ! Author: Pablo Giménez Suárez
    !
    ! Version: 1.0
    !
    !***********************************************************
```

```
    ! Temporal variables for sending data
    PERS jointtarget jointangles:=[[0,0,0,0,0,0],[0,0,0,0,0,0]];

    ! Local Communication Variables
    VAR socketdev serverUDP;

    CONST num portUDP:=6000;
    CONST string ip:="127.0.0.1";

    !***********************************************************
    !
    ! UDP Communication main
    !
    !***********************************************************
    PROC main_3()
        SocketCreate serverUDP\UDP;

        WHILE (NOT abort) DO
            IF actSim THEN
                jointangles:=CJointT();
                SocketSendTo
serverUDP,ip,portUDP\Str:=jointTargetToString(jointangles);
            ENDIF

            WaitTime 0.005;
        ENDWHILE
        SocketClose serverUDP;
    ENDPROC

    FUNC string jointTargetToString(jointtarget target)
        VAR string tempString;
        tempString:=NumToStr(target.robax.rax_1,3);
        ConcatenateString tempString,target.robax.rax_2;
        ConcatenateString tempString,target.robax.rax_3;
        ConcatenateString tempString,target.robax.rax_4;
        ConcatenateString tempString,target.robax.rax_5;
        ConcatenateString tempString,target.robax.rax_6;
        RETURN tempString;
    ENDFUNC

    ! [FJD]
    PROC ConcatenateString(INOUT string inString,num value)
        inString:=inString+";"+NumToStr(value,3);
    ENDPROC

ENDMODULE
```

GENERICVARIABLES

```
MODULE GenericVariables
    !***********************************************************
    !
    ! Module:  GenericVariables
    !
    ! Description:
    !   Shared module between all the task to work with the same variables
    !
    ! Author: Pablo Giménez Suárez
    !
    ! Version: 1.0
    !
    !***********************************************************

    ! VARIABLE PARAMETERS FOR VECTORS
    CONST num maxNumOfPoints:=100;
    PERS num nPoints;
    PERS num pPoints;
    ! Number of points used

    CONST num maxNumOfMoves:=100;
    PERS num nMoves;
    PERS num pMoves;
    ! Number of moves (J,L or C) used

    VAR num errorNum:= 0;

    ! RUNTIME VARIABLES
    PERS bool abort:=FALSE;
    ! TRUE if the program must end

    PERS bool actSim:=FALSE;
    ! TRUE if simulation is active

    PERS bool holdSim:=FALSE;
    ! TRUE if the simulation must be paused

    PERS bool actSync:=FALSE;
    ! TRUE if sincronization is active

    PERS bool actLoop:=FALSE;
    ! TRUE if there are loops

    PERS bool setPos:= FALSE;
    ! TRUE if we are going to set the instant position of the robot

    PERS bool canPause := FALSE;
    ! TRUE if the simulation can be paused

ENDMODULE
```

ARRAYMODULE

```
MODULE ArrayModule
    !***********************************************************
    !
    ! Module:  ArrayModule
    !
    ! Description:
    !   Module that stores all the PERS data vectors to make robot movements
    !
    ! Author: Pablo Giménez Suárez
    !
    ! Version: 1.0
    !
    !***********************************************************

    ! DATA ARRAYS
    PERS speeddata speed{maxNumOfMoves};
    ! Speed mode for each movement

    PERS zonedata zone{maxNumOfMoves};
    ! Precision mode for each movement

    PERS num moveType{maxNumOfMoves};
    ! Stores the 3 type of movements

    PERS robtarget targets{maxNumOfPoints};
    ! Number of targets

    PERS jointtarget actTargetPos;
ENDMODULE
```

ANEXO IV: CÓDIGO FUENTE DE UNITY

FP_CAMERABEHAVIOUR

```
//////////////////////////////////////////////////////////////
//
// Class: FP_CameraBehaviour
// Description: Camera rotation
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
//////////////////////////////////////////////////////////////

using System;
using UnityEngine;

[RequireComponent(typeof(Camera))]
public class FP_CameraBehaviour : MonoBehaviour
{
    // Functional components
    [Header("Player Components")]
    public Camera playerCamera;
    public Transform playerBody;
    public Canvas crosshair;

    // Functional variables
    [Header("Mouse Settings")]
    [SerializeField] private string mouseXInputName = "Mouse X";
    [SerializeField] private string mouseYInputName = "Mouse Y";
    [SerializeField] private float mouseSensitivity = 250f;
    private float pitch;

    // Private functions
    private void LockCursor()
    {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
        crosshair.enabled = true;
    }

    private void UnlockLockCursor()
    {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
        crosshair.enabled = false;
    }

    private void CameraRotation()
    {
        float mouseX = Input.GetAxis(mouseXInputName) * mouseSensitivity *
Time.deltaTime;
        float mouseY = Input.GetAxis(mouseYInputName) * mouseSensitivity *
Time.deltaTime;
```

```
        pitch -= mouseY;

        pitch = Mathf.Clamp(pitch, -90f, 90f);

        playerCamera.transform.localRotation = Quaternion.Euler(pitch, 0f, 0f);
        playerBody.transform.Rotate(Vector3.up * mouseX);
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKey(KeyCode.Mouse1) && !crosshair.enabled)
            LockCursor();
        else if (!Input.GetKey(KeyCode.Mouse1) && crosshair.enabled)
            UnlockLockCursor();

        if (crosshair.enabled)
            CameraRotation();
    }
}
```

FP_MOVEMENT

```
////////////////////////////////////////////////////////////////
//
// Class: FP_Movement
// Description: Player movement behaviour
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
////////////////////////////////////////////////////////////////
using System;
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(CharacterController))]
public class FP_Movement : MonoBehaviour
{
    // Properties
    [Serializable]
    public class Axes
    {
        [Tooltip("X axis")] public string abscissaInputName = "X";
        [Tooltip("Y axis")] public string ordinateInputName = "Y";
        [Tooltip("Z axis")] public string applicateInputName = "Z";
    }

    [Serializable]
    public class MovementSettings
    {
        [Header("Speed Settings")]
        public float walkSpeed = 5;
        public float runSpeed = 10;
        public float ghostSpeed = 8;
```

```
        [Tooltip("Pow factor for ghost mode")] public float spdBoost;

        [Tooltip("Real speed applied to the player")] public float speed;
        [Tooltip("Vertical velocity of the player")] public float yVelocity;
        [Tooltip("Direction the player is moving to")] public Vector3 movement;

        [Header("PhysicSettings")]
        public float gravity = -9.8f;
        public LayerMask groundLayer;
    }

    [Serializable]
    public class PlayerSettings
    {
        public bool ghostmode = false;
        public bool isGrounded = false;
        public float height = 0.9f;
        public float jumpHeight = 1.25f;
        public float groundSphereCastDistance = 1;
    }

    [Serializable]
    public class InputSettings
    {
        [Header("KeyBindings")]
        public KeyCode jumpKey = KeyCode.Space;
        public KeyCode modeKey = KeyCode.LeftAlt;
        public KeyCode runKey = KeyCode.LeftShift;

        [Header("Behaviour variables")]
        public float xAxis;
        public float yAxis;
        public float zAxis;
        public float yScroll;
        public bool run;
        public bool jump;
        public bool changeMode;
    }

    // Creating parameter variables
    public Axes axes = new Axes();
    public MovementSettings movementSettings = new MovementSettings();
    public PlayerSettings playerSettings = new PlayerSettings();
    public InputSettings inputs = new InputSettings();
    public CharacterController playerController;
    public MeshRenderer skin;
    public Transform playerTransform;
    public RaycastHit hitInfo;

    #region Unity functions

    // Awake method is called once at the start
    void Awake()
    {
        playerController = GetComponent<CharacterController>();
        playerTransform = GetComponent<Transform>();
        skin = GetComponent<MeshRenderer>();
    }

    // Update is called once per frame
    void Update()
```

```
    {
        Inputs();
        GameModeSelector();

        if (playerSettings.ghostmode)
            GhostMove();
        else
        {
            CheckisGrounded();
            CalcMoveDirection();
            Jump();
            ApplyGravity();
            Move();
        }
    }

    #endregion

    #region Private functions

    // Update the inputs of the player
    private void Inputs()
    {
        // Axes input
        inputs.xAxis = Input.GetAxisRaw(axes.abscissaInputName);
        inputs.yAxis = Input.GetAxisRaw(axes.ordinateInputName);
        inputs.zAxis = Input.GetAxisRaw(axes.applicateInputName);

        // Keyboard input
        inputs.run = Input.GetKey(inputs.runKey);
        inputs.jump = Input.GetKeyDown(inputs.jumpKey);
        inputs.changeMode = Input.GetKeyDown(inputs.modeKey);

        // Mouse input
        inputs.yScroll = Input.mouseScrollDelta.y;
    }

    // Gamemode method
    private void GameMode()
    {
        if (playerSettings.ghostmode)
        {
            playerSettings.ghostmode = false;
            playerController.enabled = true;
            skin.enabled = true;
        }
        else
        {
            playerSettings.ghostmode = true;
            playerController.enabled = false;
            movementSettings.yVelocity = 0;
            skin.enabled = false;
        }
    }

    private void GameModeSelector()
    {
        if (inputs.changeMode)
            GameMode();

        if (!playerSettings.ghostmode)
```

```
        {
            if (inputs.jump && !playerSettings.isGrounded)
                GameMode();
        }
    }

    // Check if the player is grounded method
    private void CheckisGrounded()
    {
        // Sphere cast position
        Vector3 spherePosition = playerTransform.position + new Vector3(0, -
playerSettings.height + 0.3f, 0);

        playerSettings.isGrounded = Physics.SphereCast(spherePosition, 0.3f,
Vector3.down, out hitInfo, playerSettings.groundSphereCastDistance,
movementSettings.groundLayer);
    }

    // Calculate move direction
    private void CalcMoveDirection()
    {
        if (playerSettings.isGrounded)
        {
            // Raycast
            if (Physics.Raycast(playerTransform.position, Vector3.down, out
hitInfo, Mathf.Infinity))
            {
                // Real movement direction
                Vector3 forward = Vector3.Cross(hitInfo.normal,
playerTransform.right);
                Vector3 right = Vector3.Cross(playerTransform.forward,
hitInfo.normal);

                // Saving movement direction
                movementSettings.movement = (-forward * inputs.zAxis + -right *
inputs.xAxis);
            }
            else
                movementSettings.movement = playerTransform.forward *
inputs.zAxis + playerTransform.right * inputs.xAxis;
        }
        else
            movementSettings.movement = playerTransform.forward * inputs.zAxis +
playerTransform.right * inputs.xAxis;
    }

    // Jump method
    private void Jump()
    {
        if (inputs.jump)
        {
            if (playerSettings.isGrounded)
            {
                movementSettings.yVelocity +=
Mathf.Sqrt(playerSettings.jumpHeight * 2f * -movementSettings.gravity);
                playerSettings.isGrounded = false;
            }
        }
    }

    // Apply gravity method
```

```
    private void ApplyGravity()
    {
        if (playerSettings.isGrounded && movementSettings.yVelocity < 0)
            movementSettings.yVelocity = movementSettings.gravity *
Time.deltaTime;
        else
            movementSettings.yVelocity += movementSettings.gravity *
Time.deltaTime;
    }

    // Defines the normal movement of the player
    private void Move()
    {
        // Speed modifier
        if (Input.GetKey(inputs.runKey))
            movementSettings.speed = movementSettings.runSpeed;
        else
            movementSettings.speed = movementSettings.walkSpeed;

        // Ceiling collide detection
        if ((playerController.collisionFlags & CollisionFlags.Above) != 0)
            movementSettings.yVelocity = 0;

        // Simple Move
        playerController.Move((movementSettings.movement.normalized *
movementSettings.speed + Vector3.up * movementSettings.yVelocity) *
Time.deltaTime);
    }

    // Ghost move
    private void GhostMove()
    {
        // Ghost movement
        movementSettings.movement = playerTransform.forward * inputs.zAxis +
playerTransform.up * inputs.yAxis + playerTransform.right * inputs.xAxis;

        // Speed modifier (exponential based)
        movementSettings.spdBoost += Input.mouseScrollDelta.y * 0.1f;
        movementSettings.spdBoost = Mathf.Clamp(movementSettings.spdBoost, -10,
2);

        movementSettings.speed = movementSettings.ghostSpeed * Mathf.Pow(2.0f,
movementSettings.spdBoost);

        // Simple Move
        playerController.transform.position +=
movementSettings.movement.normalized * movementSettings.speed * Time.deltaTime;
    }

    #endregion
}
```

MAINPROGRAM

```
//////////////////////////////////////////////////////////////
//
```

```
// Class: MainProgram
// Description: Program behaviour
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class MainProgram : MonoBehaviour
{
    // Program Variables
    public Communications comms;
    public RobTargetList targetList;
    public RobotList robotList;
    public RobotLibrary robotLibrary;
    public MoveAxisMenu moveAxisMenu;
    public ToolList toolList;
    public ToolLibrary toolLibrary;
    public InstructionList instructionList;
    public InstructionOptionsMenu instructionOptions;

    public float lerpTime = 0.05f;

    #region UI Element Methods
    // Robtarget methods
    public void AddTarget()
    {
        RobTarget t;
        string name;
        int id;
        if (robotList.robots.Count > 0)
        {
            t = (RobTarget)Instantiate(targetList.robtargetPrefab);

            if (targetList.targets.Count > 0)
            {
                id = targetList.targets[targetList.targets.Count - 1].GetId() +
1;
                name = "Target_" + (id + 1).ToString() + "0";
            }
            else
            {
                name = "Target_10";
                id = 0;
            }

            t.Prime(id, robotList.robots[robotList.actRobot].GetRobType(),
robotList.robots[robotList.actRobot].joints,
toolList.tools[toolList.actTool].GetTcp(), name);

            targetList.Add(t);
        }
        else
        {
            Debug.Log("ERROR 015: There are no robots");
        }
    }

    public void DeleteTarget()
    {
```

```
        if (robotList.robots.Count > 0)
        {
            RobTarget t;

            if (targetList.targets.Count > 0)
            {
                t = targetList.targets[targetList.actTarget];
                targetList.Delete(t);

                Destroy(t.gameObject);
            }
            else
                Debug.Log("ERROR 016: There are no targets to remove");
        }
    }

    // Robot methods
    public void AddRobot()
    {
        GenericRobot t;
        string name;
        switch (robotLibrary.actRobType)
        {
            case 0:
                name = "IRB120";
                t = (IRB120)Instantiate(robotLibrary.robotsPrefab[0]);

                if (robotList.robots.Count > 0)
                {
                    t.SetId(robotList.robots[robotList.robots.Count - 1].GetId()
+ 1);
                    t.SetName(name + "_" +
(robotList.robots[robotList.robots.Count - 1].GetId() + 1).ToString());
                }

                robotList.Add(t);
                break;

            default:
                Debug.Log("ERROR 017: No robot selected");
                break;
        }
    }

    public void DeleteRobot()
    {
        if (robotList.robots.Count > 0)
        {
            GenericRobot t = robotList.robots[robotList.actRobot];
            robotList.Delete(t);
            Destroy(t.gameObject);
        }
        else
            Debug.Log("ERROR 015: There are no robots");
    }

    // Tool methods
    public void AddTool()
    {
        Tool t;
        string name;
```

```
        switch (toolLibrary.actToolType)
        {
            case 0:
                name = "Tool0";
                t = (Tool0)Instantiate(toolLibrary.toolsPrefab[0]);

                if (toolList.tools.Count > 0)
                {
                    t.SetId(toolList.tools[toolList.tools.Count - 1].GetId() +
1);
                    t.SetName(name + "_" + (toolList.tools[toolList.tools.Count -
1].GetId() + 1).ToString());
                }

                toolList.Add(t);;
                break;

            default:
                Debug.Log("ERROR 018: No tool selected");
                break;
        }
    }

    public void DeleteTool()
    {
        if (toolList.tools.Count > 0)
        {
            Tool t = toolList.tools[toolList.actTool];
            toolList.Delete(t);
            Destroy(t.gameObject);
        }
        else
            Debug.Log("ERROR 019: There are no tools");
    }

    public void BindTool()
    {
        if (toolList.tools.Count > 0 && robotList.robots.Count > 0)

toolList.tools[toolList.actTool].BindTool(robotList.robots[robotList.actRobot].to
olReference);
        else
            Debug.Log("ERROR 019: There are no tools");
    }

    public void UnBindTool()
    {
        if (toolList.tools.Count > 0 && robotList.robots.Count > 0)
            toolList.tools[toolList.actTool].UnBindTool();
        else
            Debug.Log("ERROR 019: There are no tools");
    }

    // Instruction methods
    public void AddInstruction()
    {
        RobTarget[] targets;
        Instruction t;
        string name = "MoveL";

        if (targetList.targets.Count > 0)
```

```
        {
            if (instructionOptions.GetActMoveType() == 2)
            {
                name = "MoveC";
                targets = new RobTarget[] {
targetList.targets[instructionOptions.GetActTarget1()],
targetList.targets[instructionOptions.GetActTarget2()] };
            }
            else
            {
                if (instructionOptions.GetActMoveType() != 0)
                    name = "MoveJ";
                targets = new RobTarget[] {
targetList.targets[instructionOptions.GetActTarget1()] };
            }

            t = new Instruction(name, targets,
instructionOptions.GetActMoveType(), instructionOptions.GetActSpeed(),
instructionOptions.GetActPrecision());

            instructionList.Add(t);
        }
    }

    public void DeleteInstruction()
    {
        Instruction t;
        if (instructionList.orders.Count > 0)
        {
            t = instructionList.orders[instructionList.actInstr];
            instructionList.Delete(t);
        }
        else
            Debug.Log("ERROR 020: There are no Instructions to remove");
    }
    #endregion

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        // Initializing components
        comms = gameObject.GetComponent(typeof(Communications)) as
Communications;
        targetList = gameObject.GetComponent(typeof(RobTargetList)) as
RobTargetList;
        robotList = gameObject.GetComponent(typeof(RobotList)) as RobotList;
        robotLibrary = gameObject.GetComponent(typeof(RobotLibrary)) as
RobotLibrary;
        moveAxisMenu = gameObject.GetComponent(typeof(MoveAxisMenu)) as
MoveAxisMenu;
        toolList = gameObject.GetComponent(typeof(ToolList)) as ToolList;
        toolLibrary = gameObject.GetComponent(typeof(ToolLibrary)) as
ToolLibrary;
        instructionList = gameObject.GetComponent(typeof(InstructionList)) as
InstructionList;
        instructionOptions =
gameObject.GetComponent(typeof(InstructionOptionsMenu)) as
InstructionOptionsMenu;

        Application.targetFrameRate = 300;
```

```
    }

    // Update is called once per frame
    void Update()
    {
        // Movement of the active robot with RobotStudio angles received with UDP
communications
        if(comms.sim && (comms.rAngles.Length ==
robotList.robots[robotList.actRobot].GetNumJoints()))
        {
            float[] tAngles = new
float[robotList.robots[robotList.actRobot].GetNumJoints()];

            for (int i = 0; i <
robotList.robots[robotList.actRobot].GetNumJoints(); i++)
                tAngles[i] =
Mathf.Lerp(robotList.robots[robotList.actRobot].GetJoint(i).GetAngle(),
comms.rAngles[i], Time.deltaTime/0.005f);

            if (comms.inPlay)
            {
                robotList.robots[robotList.actRobot].AxesMovement(tAngles);

moveAxisMenu.UpdateValues(robotList.robots[robotList.actRobot].GetJoints(),
robotList.robots[robotList.actRobot].displayName);
            }
        }

        // Specific angle selected with input fields
        if(moveAxisMenu.needChange)
        {
            for (int i = 0; i <
robotList.robots[robotList.actRobot].GetNumJoints(); i++)
            {

robotList.robots[robotList.actRobot].MoveAxis(moveAxisMenu.joints[i].GetAngle(),
i);
            }

            moveAxisMenu.needChange = false;
        }

        // Move the robot to the selected robtarget
        if(targetList.placeRobot && !comms.sim)
        {

robotList.robots[robotList.actRobot].AxesMovement(targetList.targets[targetList.a
ctTarget].angles);
            moveAxisMenu.Prime(robotList.robots[robotList.actRobot].joints,
robotList.robots[robotList.actRobot].displayName);

            targetList.placeRobot = false;
        }
    }
    #endregion
}
```

COMMUNICATIONS

```
//////////////////////////////////////////////////////////////
//
// Class: Communications
// Description: Communications module. TCP and UDP protocols.
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
//////////////////////////////////////////////////////////////
using System.Globalization;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using UnityEngine;
using UnityEngine.UI;

public class Communications : MonoBehaviour
{
    #region Variables
    // Behaviour Variables
    [Header("Behaviour Variables")]
    [Tooltip("Boolean that tells if the program is running or not")] public bool
abort;
    [Tooltip("Flag to start the synchronization with RobotStudio")] public bool
sendSync;
    [Tooltip("Bollean that tells if the simulation state must be updated")]
public bool upPlay;
    [Tooltip("Bollean that tells if the simulation must be in loop")] public bool
upLoop;
    [Tooltip("Flag to start a break in the simulation")] public bool stopSim;
    [Tooltip("Flag to start the end of the program")] public bool endProg;

        // Number of points and instructions
        private int nPoints;
        private int nMoves;

        // Position through the list of points and moves
        private int pPoints;
        private int pMoves;


    [Header("Communication Variables")]
    private static string ip = "127.0.0.1";
    NetworkStream streamRobotStudio = default;
    [Tooltip("Boolean that tells if the program is connected to RobotStudio")]
public bool connected;
    [Tooltip("Boolean that tells if a message can be sent")] public bool canSend;
    [Tooltip("Boolean that tells if the program can end after robotstudio has
been stopped")] public bool canEnd;
    public string inMsg;
    public string outMsg;
    public float[] rAngles = null;

        // TCP variables
        private int pRobotStudioTCP = 5000;
        Thread threadRobotStudioTCPReceive = null;
        Thread threadRobotStudioTCPSend = null;
```

```
        TcpClient clientRobotStudioTCP = new TcpClient();

        // UDP variables
        private int pRobotStudioUDP = 6000;
        Thread threadRobotStudioUDP = null;
        UdpClient clientRobotStudioUDP;


    [Header("Simulation Variables")]
    public RobotList robotList;
    public InstructionList instructionList;
    [Tooltip("State of the simulation")] public bool sim;
    [Tooltip("State of the simulation play in RobotStudio")] public bool inPlay;


    [Header("Syncrhonization Variables")]
    [Tooltip("State of the synchronization with RobotStudio")] public bool sync;
    public bool robTargetSent;
    public bool instructionsSent;
    public bool initialPositionSent;
    [Tooltip("State of the simulation is in loop or not")] public bool inLoop;


    // UI Elements
    public Text playButtonText;
    public Image syncImage;
    public Image loopImage;
    public Image connectImage;
    #endregion

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        // Initialize compontents
        robotList = gameObject.GetComponent(typeof(RobotList)) as RobotList;
        instructionList = gameObject.GetComponent(typeof(InstructionList)) as
InstructionList;

        InitComms();

        // Create and start RobotStudio Receive TCP communications thread
        threadRobotStudioTCPReceive = new Thread(CommsRobotStudioTCPReceive);
        threadRobotStudioTCPReceive.Start();

        // Create and start RobotStudio Send TCP communications thread
        threadRobotStudioTCPSend = new Thread(CommsRobotStudioTCPSend);
        threadRobotStudioTCPSend.Start();

        // Create and start RobotStudio UDP communications thread
        threadRobotStudioUDP = new Thread(CommsRobotStudioUDP);
        threadRobotStudioUDP.Start();
    }

    void Update()
    {
        // Most of this tasks are here because they depends on the secondary
threads and uses functiones only available in the main thread
        if(sync)
            syncImage.color = Color.red;
        else
```

```
            syncImage.color = Color.green;

        if (connected)
            connectImage.color = Color.green;
        else
            connectImage.color = Color.white;

        if(!inPlay)
            playButtonText.text = "Play";
        else
            playButtonText.text = "Pause";

        if(canEnd)
            ExitMenu.Quit();
    }

    // Activated if user stops Unity
    void OnApplicationQuit()
    {
        abort = true;
        // If the client or the stream are not null yet , close them
        if ((clientRobotStudioTCP != null || streamRobotStudio != null) &&
connected)
        {
            connected = false;
            streamRobotStudio.Close();
            clientRobotStudioTCP.Close();
        }
    }
    #endregion

    #region UDP Communications
    // UDP Comms
    void CommsRobotStudioUDP() // receives the real robot's joint angles from
RobotStudio
    {
        clientRobotStudioUDP = new UdpClient(pRobotStudioUDP);
        IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, pRobotStudioUDP);
        while (!abort)
        {
            try
            {
                if (sim & clientRobotStudioUDP.Available > 0)
                {
                    byte[] angles;
                    angles = clientRobotStudioUDP.Receive(ref remoteEP);

                    if (angles != null)
                        rAngles =
AnglesStrToFloat(System.Text.Encoding.ASCII.GetString(angles));
                }
            }
            catch (System.IO.IOException ex)
            {
                if(sim)
                    Debug.Log("ERROR 008: Unable to receive joint angles.  " +
ex.ToString());
            }
        }
        return;
    }
```

```
    #endregion

    #region TCP Communications
    // TCP Comms
    void CommsRobotStudioTCPSend()
    {
        while(!abort)
        {
            if (connected)
            {
                try // Sending data
                {
                    if (canSend)
                    {
                        if (sync) // Syncrhonize robtargets and instructions
                        {
                            if (!robTargetSent)
                            {
                                if (instructionList.targets.Count > 0)
                                {
                                    // Update outMsg to confirm that a robtarget
is going to be sent
                                    outMsg =
RobTargetToStr(instructionList.targets[pPoints]);
                                    pPoints++;

                                    if (pPoints == nPoints)
                                        robTargetSent = true;

                                    Debug.Log("Sending Point: " + outMsg);
                                }
                                else
                                {
                                    Debug.Log("ERROR 009: There are no points
created");
                                    robTargetSent = true;
                                    canSend = true;
                                    outMsg = null;
                                }
                            }
                            else if (!instructionsSent)
                            {
                                if (instructionList.orders.Count > 0)
                                {
                                    // Update outMsg to confirm that a
instruction is going to be sent
                                    outMsg =
InstructionToStr(instructionList.orders[pMoves]);
                                    pMoves++;

                                    if (pMoves == nMoves)
                                        instructionsSent = true;

                                    Debug.Log("Sending Instruction: " + outMsg);
                                }
                                else
                                {
                                    Debug.Log("ERROR 010: There are no orders");
                                    instructionsSent = true;
                                    canSend = true;
                                    outMsg = null;
```

```
                                }
                            }
                            else if (upLoop != inLoop)
                            {
                                inLoop = upLoop;
                                outMsg = "L";
                            }
                            else if (!initialPositionSent)
                            {
                                outMsg =
JointTargetToStr(robotList.robots[robotList.actRobot]);
                                initialPositionSent = true;
                            }
                            else
                            {
                                sync = false;
                                outMsg = "S";
                            }
                        }

                        if (sendSync)
                        {
                            sendSync = false;

                            // Data from the lists
                            pPoints = 0;
                            pMoves = 0;
                            nPoints = instructionList.targets.Count;
                            nMoves = instructionList.orders.Count;

                            outMsg = AddStrToMsg("S") + AddNumToMsg(nPoints) +
AddNumToMsg(nMoves);
                        }

                        if (stopSim)
                        {
                            stopSim = false;
                            outMsg = "S";
                        }

                        if (endProg)
                        {
                            endProg = false;
                            outMsg = "E";
                        }

                        if (upPlay)
                        {
                            upPlay = false;
                            outMsg = "P";
                        }

                        // Send Message
                        if (outMsg != null)
                        {
                            Debug.Log("Data Sent: " + outMsg);
                            byte[] outStream =
System.Text.Encoding.ASCII.GetBytes(outMsg);
                            streamRobotStudio.Write(outStream, 0,
outStream.Length);
                            streamRobotStudio.Flush();
```

```
                            canSend = false;
                        }
                    }
                }

                catch (System.IO.IOException e)
                {
                    Debug.Log("ERROR 011: Can't send message to RobotStudio. " +
e.ToString());
                    connected = false;
                }

            }
        }
    }

    void CommsRobotStudioTCPReceive()
    {
        while (!abort)
        {
            if (connected)
            {
                streamRobotStudio = clientRobotStudioTCP.GetStream();

                try // Try receiving
                {
                    var buffsize = clientRobotStudioTCP.ReceiveBufferSize;
                    byte[] inStream = new byte[buffsize];

                    streamRobotStudio.Read(inStream, 0, buffsize);
                    inMsg = System.Text.Encoding.ASCII.GetString(inStream);

                    Debug.Log("Data Received: " + inMsg);
                    if (!inMsg.Contains("ERROR"))
                        inMsg = inMsg[0].ToString();

                    switch (inMsg)
                    {
                        case "R": // Starts syncronization
                            InitSync();
                            sync = true;
                            canSend = true;
                            break;

                        case "C": // Confirms that it can send again
                            canSend = true;
                            break;

                        case "S": // Starts the simulation
                            sim = true;
                            inPlay = true;
                            break;

                        case "P": // Play/Pauses the simulation
                            inPlay = !inPlay;
                            break;

                        case "E": // Ends the simulation
                            sim = false;
                            inPlay = false;
```

```
                            break;

                        case "Y":
                            Debug.Log("Finalizing...");
                            canEnd = true;
                            break;

                        default:
                            Debug.Log(inMsg);
                            break;
                    }

                    Thread.Sleep(100);
                }

                catch (System.IO.IOException e)
                {
                    Debug.Log("ERROR 012: Not receiving message from RobotStudio.
" + e.ToString());
                    connected = false;
                }
            }
        }
    }
    #endregion

    #region Methods for UI Elements
    public void Connect()
    {
        try
        {
            clientRobotStudioTCP.Connect(ip, pRobotStudioTCP);
            connected = true;
        }
        catch (System.Net.Sockets.SocketException e)
        {
            Debug.Log("ERROR 013: " + e.ToString());
        }
    }

    public void Sync() // Function that uses the interface to sync with
RobotStudio
    {
        if(!connected)
        {
            Debug.Log("ERROR 014: Not connected to Robotstudio");
            return;
        }

        if (!sim & !sync)
        {
            // Flag to send data
            sendSync = true;
            canSend = true;
        }
    }

    public void Sim()
    {
        if (!sync)
        {
```

```
            upPlay = !upPlay;
            canSend = true;
        }
    }

    public void StopSim()
    {
        if (sim)
        {
            stopSim = true;
            canSend = true;
        }
    }

    public void End()
    {
        if (!sync && !sim)
        {
            endProg = true;
            canSend = true;
        }
    }

    public void Loop()
    {
        upLoop = !upLoop;
        if(upLoop)
            loopImage.color = Color.green;
        else
            loopImage.color = Color.red;
    }
    #endregion

    #region Private Functional Methods
    private void InitComms()
    {
        // Initialize all the boolean variables as expected
        // Behaviour Variables
        abort = false;
        sendSync = false;
        upPlay = false;
        upLoop = false;
        stopSim = false;
        endProg = false;

        // Communication Variables
        connected = false;
        canSend = false;
        canEnd = false;

        // Simulation Variables
        sim = false;
        inPlay = false;

        // Synchronization Variables
        sync = false;
        robTargetSent = false;
        instructionsSent = false;
        initialPositionSent = false;
        inLoop = false;
}
```

```
    private void InitSync()
    {
        robTargetSent = false;
        instructionsSent = false;
        initialPositionSent = false;
    }

    private float[] AnglesStrToFloat(string a) // 28.692;-20.938;17.530;-
11.391;34.141;13.159
    {
        string[] result;
        result = a.Split(';');

        float[] jAngles = new float[result.Length];
        for (int i = 0; i < result.Length; i++)
        {
            jAngles[i] = -float.Parse(result[i], NumberStyles.Float,
CultureInfo.InvariantCulture);
        }

        return jAngles;
    }

    private string AddStrToMsg(string s) { return (s + ";"); }

    private string AddNumToMsg(float n) { return (n.ToString("0.##",
CultureInfo.InvariantCulture) + ";"); }

    private string RobTargetToStr(RobTarget r)
    {
        Vector3 pos = r.GetRSPosition();
        Vector3 ang = r.GetRSRotation();

        string msgRT = "T[";

        for (int i = 0; i < 3; i++)
        {
            msgRT += AddNumToMsg(pos[i]);
        }

        for (int i = 0; i < 3; i++)
        {
            msgRT += AddNumToMsg(ang[i]);
        }

        for (int i = 0; i < 4; i++)
        {
            msgRT += AddNumToMsg(r.GetRSConfig(i));
        }

        msgRT += "]";

        return msgRT;
    }

    private string InstructionToStr(Instruction i)
    {
        string msgI = "M[";

        msgI += AddStrToMsg(i.GetMoveType());
```

```
        msgI += AddNumToMsg(i.GetSpeed());
        msgI += AddNumToMsg(i.GetPrecision());

        msgI += "]";

        return msgI;
    }

    private string JointTargetToStr(GenericRobot r)
    {
        string msgJ = "A[";

        for (int i = 0; i< r.GetNumJoints();i++)
        {
            msgJ += (-r.joints[i].GetAngle()).ToString("0.##",
CultureInfo.InvariantCulture);

            if (i == r.GetNumJoints()-1)
                msgJ += "]";
            else
                msgJ += ",";
        }

        return msgJ;
    }
    #endregion
}
```

CONTROLPANEL

```
//////////////////////////////////////////////////////////////
//
// Class: ControlPanel
// Description: Class that manages the different main panels of the interface
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class ControlPanel : MonoBehaviour
{
    // Objects of the different canvas panels
    public Canvas robotsPanel;
    public Canvas programmingPanel;
    public Canvas toolsPanel;

    // Public behaviour menus
    public void ProgramPanel()
    {
        programmingPanel.enabled = true;
        robotsPanel.enabled = false;
        toolsPanel.enabled = false;
    }
```

```
    public void RobotPanel()
    {
        programmingPanel.enabled = false;
        robotsPanel.enabled = true;
        toolsPanel.enabled = false;
    }

    public void ToolPanel()
    {
        programmingPanel.enabled = false;
        robotsPanel.enabled = false;
        toolsPanel.enabled = true;
    }
}
```

EXITMENU

```
//////////////////////////////////////////////////////////////
//
// Class: ExitMenu
// Description: Behaviour of the Exit menu
//
// Author: Pablo Giménez Suárez
// Last Review: 07/06/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class ExitMenu : MonoBehaviour
{
    // UI variables
    public Canvas exitMenu;

    // Start is called before the first frame update
    void Start()
    {
        exitMenu.enabled = false;
    }

    // Update is called once per frame
    void Update()
    {
        // Exit Sample
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (exitMenu.enabled)
                exitMenu.enabled = false;
            else
                exitMenu.enabled = true;
        }
    }

    // Quit function
    public static void Quit()
    {
        Application.Quit();
```

```
        #if UNITY_EDITOR
            UnityEditor.EditorApplication.isPlaying = false;
        #endif
    }
}
```

MOVEAXISMENU

```
//////////////////////////////////////////////////////////////
//
// Class: MoveAxisMenu
// Description: Class that moves the different angles of the robot
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;

public class MoveAxisMenu : MonoBehaviour
{
    // List of robots
    public List<CJoint> joints = new List<CJoint>();

    // Display
    public MoveAxisMenuDisplay moveAxisMenuDisplay;
    public bool needChange;
    public string displayName;

    #region Unity Methods
    void Start()
    {
        CJointDisplay.OnValueChange += HandleOnValueChange;
        moveAxisMenuDisplay.canvas.enabled = false;
    }

    void OnDestroy()
    {
        CJointDisplay.OnValueChange -= HandleOnValueChange;
    }
    #endregion

    #region Functional Methods
    // Event Method
    void HandleOnValueChange(CJoint joint)
    {
        if (joints.Contains(joint))
        {
            needChange = true;
        }
    }

    // Initialize Methods
    // Used when a new robot is selected
    public void Prime(List<CJoint> joints, string displayName)
```

```
    {
        if (!moveAxisMenuDisplay.canvas.enabled)
            moveAxisMenuDisplay.canvas.enabled = true;

        this.displayName = displayName;
        this.joints = joints;
        moveAxisMenuDisplay.Prime(this);
    }

    // Used when the display robot suffer a change in their angles
    public void UpdateValues(List<CJoint> joints, string displayName)
    {
        this.displayName = displayName;
        this.joints = joints;
        moveAxisMenuDisplay.UpdateValues(this);
    }
    #endregion
}
```

INSTRUCTIONOPTIONSMENU

```
//////////////////////////////////////////////////////////////
//
// Class: InstructionOptionMenu
// Description: Class that manages the behaviour of the instructions options menu
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class InstructionOptionsMenu : MonoBehaviour
{
    // Public variables
    public MoveTypeDropdown moveType;
    public SpeedDropdown speed;
    public PrecisionDropdown precision;
    public RobTargetDropdown target_1;
    public RobTargetDropdown target_2;

    public RobTargetList targetList;

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        targetList = gameObject.GetComponent(typeof(RobTargetList)) as
RobTargetList;
        MoveTypeDropdown.OnClick += HandleOnClick;

        if (GetActMoveType() != 2)
        {
            target_2.Prime(targetList);
            target_2.dropDown.gameObject.SetActive(false);
        }
```

```
    }

    void OnDestroy()
    {
        MoveTypeDropdown.OnClick += HandleOnClick;
    }
    #endregion

    // Functional Methods
    // Event Method
    void HandleOnClick(int t)
    {
        if (GetActMoveType() == 2)
        {
            target_2.dropDown.gameObject.SetActive(true);
            target_2.Prime(targetList);
        }
        else
            target_2.dropDown.gameObject.SetActive(false);
    }

    // Get Functions
    public int GetActMoveType() { return moveType.dropDown.value; }
    public int GetActSpeed() { return speed.dropDown.value; }
    public int GetActPrecision() { return precision.dropDown.value; }
    public int GetActTarget1() { return target_1.dropDown.value; }
    public int GetActTarget2() { return target_2.dropDown.value; }

}
```

INSTRUCTIONLIST

```
//////////////////////////////////////////////////////////////
//
// Class: InstructionList
// Description: Class that manages the list of instructions that the system has
//
// Author: Pablo Giménez Suárez
// Last Review: 03/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;

public class InstructionList : MonoBehaviour
{
    // List of instructions
    public List<Instruction> orders = new List<Instruction>();

    // List of the targets used in the instructions in order
    public List<RobTarget> targets = new List<RobTarget>();

    // Selected Instruction
    public int actInstr;
```

```
    // List Display
    public InstructionScrollListDisplay scrollListDisplay;

    // Delegate
    public delegate void InstructionListDelegate(InstructionList
InstructionList);
    public static event InstructionListDelegate OnChanged;


    #region Unity Methods
    // Event management
    void Start()
    {
        InstructionUIDisplay.OnClick += HandleOnClick;
    }

    void OnDestroy()
    {
        InstructionUIDisplay.OnClick -= HandleOnClick;
    }

    #endregion

    void HandleOnClick(Instruction order)
    {
        if (orders.Contains(order))
        {
            actInstr = orders.IndexOf(order);
            scrollListDisplay.ShowInstruction(actInstr);
        }
    }

    // Public functions
    public void Add(Instruction order)
    {
        // Debug functionality
        if (order == null)
            return;

        orders.Add(order);

        if (order.GetMoveType() == "C")
        {
            targets.Add(order.targets[0]);
            targets.Add(order.targets[1]);
        }
        else
            targets.Add(order.targets[0]);

        if (OnChanged != null)
            OnChanged.Invoke(this);

        actInstr = orders.Count - 1;
        scrollListDisplay.ShowInstruction(actInstr);
    }

    public void Delete(Instruction order)
    {
        // Debug functionality
        if (order == null)
            return;
```

```
        if (!orders.Contains(order))
            return;

        orders.Remove(order);

        if (order.GetMoveType() == "C")
        {
            targets.Remove(order.targets[0]);
            targets.Remove(order.targets[1]);
        }
        else
            targets.Remove(order.targets[0]);

        if (OnChanged != null)
            OnChanged.Invoke(this);

        if (orders.Count > 0)
        {
            actInstr = orders.Count - 1;
            scrollListDisplay.ShowInstruction(actInstr);
        }
        else
            scrollListDisplay.EraseShowInstruction();
    }
}
```

ROBOTLIST

```
//////////////////////////////////////////////////////////////
//
// Class: RobotList
// Description: Class that manages the list of robots that has been created
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;

public class RobotList : MonoBehaviour
{
    // List of robots
    public List<GenericRobot> robots = new List<GenericRobot>();
    public int actRobot;

    // Display
    public RobotScrollListDisplay scrollListDisplay;
    public MoveAxisMenu moveAxisMenu;

    // Delegate
    public delegate void RobotListDelegate(RobotList robotList);
    public static event RobotListDelegate OnChanged;

    #region Unity Methods
    void Start()
    {
        RobotUIDisplay.OnClick += HandleOnClick;
```

```
    }

    void OnDestroy()
    {
        RobotUIDisplay.OnClick -= HandleOnClick;
    }
    #endregion

    // Functional Methods
    void HandleOnClick(GenericRobot robot)
    {
        if (robots.Contains(robot))
        {
            actRobot = robots.IndexOf(robot);
            scrollListDisplay.ShowRobot(actRobot);
            moveAxisMenu.Prime(robot.joints, robot.displayName);
        }
    }

    public void Add(GenericRobot robot)
    {
        // Debug functionality
        if (robot == null)
            return;

        robots.Add(robot);
        if (OnChanged != null)
            OnChanged.Invoke(this);

        actRobot = robots.Count - 1;
        moveAxisMenu.Prime(robot.joints, robot.displayName);
        scrollListDisplay.ShowRobot(actRobot);
    }

    public void Delete(GenericRobot robot)
    {
        // Debug functionality
        if (robot == null)
            return;
        if (!robots.Contains(robot))
            return;

        robots.Remove(robot);
        if (OnChanged != null)
            OnChanged.Invoke(this);

        if (robots.Count > 0)
        {
            actRobot = robots.Count - 1;
            moveAxisMenu.Prime(robot.joints, robot.displayName);
            scrollListDisplay.ShowRobot(actRobot);
        }
        else
        {
            moveAxisMenu.moveAxisMenuDisplay.canvas.enabled = false;
            scrollListDisplay.EraseShowRobot();
        }
    }
}
```

ROBTARGETLIST

```
//////////////////////////////////////////////////////////////
//
// Class: RobTargetList
// Description: Class that manages the list of robtargets that the system has
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;

public class RobTargetList : MonoBehaviour
{
    // List of targets
    public List<RobTarget> targets = new List<RobTarget>();
    public int actTarget;

    // Extra functional variables
    public bool placeRobot;

    // Display
    public RobTargetScrollListDisplay ScrollListDisplay;

    // Delegate
    public delegate void RobTargetListDelegate(RobTargetList robTargetList);
    public static event RobTargetListDelegate OnChanged;

    // Prefab
    public RobTarget robtargetPrefab;

    #region Unity Methods
    void Start()
    {
        RobTargetUIDisplay.OnClick += HandleonClick;
    }

    void OnDestroy()
    {
        RobTargetUIDisplay.OnClick -= HandleonClick;
    }
    #endregion

    void HandleonClick(RobTarget target)
    {
        if (targets.Contains(target))
        {
            actTarget = targets.IndexOf(target);
            ScrollListDisplay.ShowTarget(actTarget);

            placeRobot = true;
        }
    }

    // Functional Methods
    public void Add(RobTarget target)
    {
```

```
        // Debug functionality
        if (target == null)
            return;

        targets.Add(target);
        if (OnChanged != null)
            OnChanged.Invoke(this);

        actTarget = targets.Count - 1;
        ScrollListDisplay.ShowTarget(actTarget);
    }

    public void Delete(RobTarget target)
    {
        // Debug functionality
        if (target == null)
            return;
        if (!targets.Contains(target))
            return;

        targets.Remove(target);
        if (OnChanged != null)
            OnChanged.Invoke(this);

        if (targets.Count > 0)
        {
            actTarget = targets.Count - 1;
            ScrollListDisplay.ShowTarget(actTarget);
        }
        else
            ScrollListDisplay.EraseShowTarget();
    }
}
```

TOOLLIST

```
//////////////////////////////////////////////////////////////
//
// Class: ToolList
// Description: Class that manages the list of tools that has been created
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ToolList : MonoBehaviour
{
    // List of tools
    public List<Tool> tools = new List<Tool>();
    public int actTool;

    // Display
```

```
    public ToolScrollListDisplay ScrollListDisplay;

    // Delegate
    public delegate void ToolScrollListDelegate(ToolList toolList);
    public static event ToolScrollListDelegate OnChanged;

    #region Unity Methods
    void Start()
    {
        ToolUIDisplay.OnClick += HandleOnClick;
    }

    void OnDestroy()
    {
        ToolUIDisplay.OnClick -= HandleOnClick;
    }
    #endregion

    // Functional Methods
    void HandleOnClick(Tool tool)
    {
        if (tools.Contains(tool))
        {
            actTool = tools.IndexOf(tool);
            ScrollListDisplay.ShowTool(actTool);
        }
    }

    public void Add(Tool tool)
    {
        // Debug functionality
        if (tool == null)
            return;

        tools.Add(tool);
        if (OnChanged != null)
            OnChanged.Invoke(this);

        actTool = tools.Count - 1;
        ScrollListDisplay.ShowTool(actTool);
    }

    public void Delete(Tool tool)
    {
        if (tool == null)
            return;
        if (!tools.Contains(tool))
            return;

        tools.Remove(tool);
        if (OnChanged != null)
            OnChanged.Invoke(this);

        if (tools.Count > 0)
        {
            actTool = tools.Count - 1;
            ScrollListDisplay.ShowTool(actTool);
        }
        else
            ScrollListDisplay.EraseShowTool();
    }
```

```
}
```

ROBOTLIBRARY

```
//////////////////////////////////////////////////////////////
//
// Class: RobotLibrary
// Description: Class that defines de behaviour of the library of robots
available
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;

public class RobotLibrary : MonoBehaviour
{
    // List of the different robots in the program
    [Header("Robot library data")]
    public List<RobotLibraryIcon> robotIcons = new List<RobotLibraryIcon>();
    public List<GenericRobot> robotsPrefab = new List<GenericRobot>();

    [Header("Robot types")]
    public int actRobType;

    // Visual variables
    [Header("Library Display")]
    public RobotLibraryDisplay robotLibraryDisplay;

    #region Unity Methods
    void Awake()
    {
        // RobotIcons
        robotIcons[0] = Resources.Load<RobotLibraryIcon>("ABB
Library/Robots/UIIconPrefab/Icon_IRB120");

        // robotsPrefabs
        robotsPrefab[0] = Resources.Load<GenericRobot>("ABB
Library/Robots/ScenePrefab/IRB120/IRB120");
    }

    // Event management
    void Start()
    {
        robotLibraryDisplay.Prime(this);
        RobotLibraryIconDisplay.onClick += HandleonClick;
    }

    void OnDestroy()
    {
        RobotLibraryIconDisplay.onClick -= HandleonClick;
    }
    #endregion
```

```
    void HandleonClick(RobotLibraryIcon robot)
    {
        if (robotIcons.Contains(robot))
            actRobType = robot.robType;
    }
}
```

TOOLLIBRARY

```
//////////////////////////////////////////////////////////////
//
// Class: ToolLibrary
// Description: Class that defines de behaviour of the library of tools available
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;

public class ToolLibrary : MonoBehaviour
{
    // List of the different robots in the program
    [Header("Tool library data")]
    public List<ToolLibraryIcon> toolIcons = new List<ToolLibraryIcon>();
    public List<Tool> toolsPrefab = new List<Tool>();

    public int actToolType;

    // Display
    [Header("Library Display")]
    public ToolLibraryDisplay toolLibraryDisplay;

    #region Unity Methods
    void Start()
    {
        toolLibraryDisplay.Prime(this);
        ToolLibraryIconDisplay.onClick += HandleonClick;
    }

    void OnDestroy()
    {
        ToolLibraryIconDisplay.onClick -= HandleonClick;
    }
    #endregion

    void HandleonClick(ToolLibraryIcon tool)
    {
        if (toolIcons.Contains(tool))
            actToolType = tool.toolType;
    }
}
```

TOOLSCROLLLISTDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: ToolScrollListDisplay
// Description: Class that shows the list of tools that has been created
//
// Author: Pablo Giménez Suárez
// Last Review: 07/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class ToolScrollListDisplay : MonoBehaviour
{
    // Public variables
    public Text actToolShowSlot;
    public Transform targetTransform;
    public ToolUIDisplay toolUIDisplayPrefab;

    public ToolList toolList;

    #region Unity Methods
    void Awake()
    {
        toolUIDisplayPrefab = Resources.Load<ToolUIDisplay>("ABB
Library/RobotStudioObjects/Tools/UIPrefab/ToolDisplaySlot");
    }

    // Start/Destroy
    void Start()
    {
        ToolList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        ToolList.OnChanged -= HandleOnChanged;
    }
    #endregion

    // HandleOnClick
    void HandleOnChanged(ToolList toolList)
    {
        if (this.toolList == toolList)
        {
            Prime(toolList);
        }
    }

    // Initilizer function
    public void Prime(ToolList toolList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.toolList = toolList;
        foreach (Tool tool in this.toolList.tools)
```

```
        {
            ToolUIDisplay display =
(ToolUIDisplay)Instantiate(toolUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(tool);
        }
    }

    // Functional methods
    public void ShowTool(int aT)
    {
        string displayName = "Nothing";
        if (toolList.tools[aT] != null)
            displayName = toolList.tools[aT].displayName;

        actToolShowSlot.text = displayName;
    }

    public void EraseShowTool()
    {
        actToolShowSlot.text = "";
    }
}
```

ROBTARGETSCROLLLISTDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: RobTargetScrollListDisplay
// Description: Class that shows the list of robtargets that has been created
//
// Author: Pablo Giménez Suárez
// Last Review: 07/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class RobTargetScrollListDisplay : MonoBehaviour
{
    // Public variables
    public Transform targetTransform;
    public RobTargetUIDisplay robTargetUIDisplayPrefab;

    public Text actRobTargetShowSlot;

    public RobTargetList TargetList;

    #region Unity Methods
    void Awake()
    {
        robTargetUIDisplayPrefab = Resources.Load<RobTargetUIDisplay>("ABB
Library/RobotStudioObjects/RobTargets/UIPrefab/RobTargetDisplaySlot");
    }

    // Start/Destroy
```

```
    void Start()
    {
        RobTargetList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        RobTargetList.OnChanged -= HandleOnChanged;
    }
    #endregion

    // HandleOnChange
    void HandleOnChanged(RobTargetList targetList)
    {
        if (this.TargetList == targetList)
            Prime(targetList);
    }

    // Initilizer function
    public void Prime(RobTargetList robTargetList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.TargetList = robTargetList;
        foreach (RobTarget target in this.TargetList.targets)
        {
            RobTargetUIDisplay display =
(RobTargetUIDisplay)Instantiate(robTargetUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(target);
        }
    }

    // Functional methods
    public void ShowTarget(int aT)
    {
        string displayName = "Nothing";
        if (TargetList.targets[aT] != null)
            displayName = TargetList.targets[aT].displayName;

        actRobTargetShowSlot.text = displayName;
    }

    public void EraseShowTarget()
    {
        actRobTargetShowSlot.text = "";
    }
}
```

ROBOTSCROLLLISTDISPLAY

```
//////////////////////////////////////////////////////////////////
//
// Class: RobotScrollListDisplay
// Description: Class that shows the list of robots that has been created
```

```
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
////////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class RobotScrollListDisplay : MonoBehaviour
{
    // Public variables
    public Text actRobotShowSlot;
    public Transform targetTransform;
    public RobotUIDisplay robotUIDisplayPrefab;

    public RobotList robotList;

    #region Unity Methods
    void Awake()
    {
        robotUIDisplayPrefab = Resources.Load<RobotUIDisplay>("ABB
Library/Robots/UIPrefab/RobotDisplaySlot");
    }

    // Start/Destroy
    void Start()
    {
        RobotList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        RobotList.OnChanged -= HandleOnChanged;
    }
    #endregion

    // HandleOnClick
    void HandleOnChanged(RobotList robotList)
    {
        if (this.robotList == robotList)
            Prime(robotList);
    }

    // Initilizer method
    public void Prime(RobotList robotList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.robotList = robotList;
        foreach (GenericRobot robot in this.robotList.robots)
        {
            RobotUIDisplay display =
(RobotUIDisplay)Instantiate(robotUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(robot);
        }
    }

    // Functional methods
    public void ShowRobot(int aR)
```

```
    {
        string displayName = "Nothing";
        if (robotList.robots[aR] != null)
            displayName = robotList.robots[aR].displayName;

        actRobotShowSlot.text = displayName;
    }

    public void EraseShowRobot()
    {
        actRobotShowSlot.text = "";
    }
}
```

INSTRUCTIONSCROLLLISTDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: InstructionScrollListDisplay
// Description: Class that shows the list of instructions that has been created
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class InstructionScrollListDisplay : MonoBehaviour
{
    // Public variables
    public Text actInstructionShowSlot;
    public Transform targetTransform;
    public InstructionUIDisplay instructionUIDisplayPrefab;

    public InstructionList instructionList;

    #region Unity Methods
    void Awake()
    {
        instructionUIDisplayPrefab = Resources.Load<InstructionUIDisplay>("ABB
Library/RobotStudioObjects/Instructions/InstructionDisplaySlot");
    }

    // Start/Destroy
    void Start()
    {
        InstructionList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        InstructionList.OnChanged -= HandleOnChanged;
    }
    #endregion
```

```
    // HandleOnClick
    void HandleOnChanged(InstructionList instructionList)
    {
        if (this.instructionList == instructionList)
            Prime(instructionList);
    }

    // Initilizer function
    public void Prime(InstructionList instructionList)
    {
        for (int a = 0; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.instructionList = instructionList;
        foreach (Instruction order in this.instructionList.orders)
        {
            InstructionUIDisplay display =
(InstructionUIDisplay)Instantiate(instructionUIDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(order);
        }
    }

    // Functional methods
    public void ShowInstruction(int aI)
    {
        string displayName = "Nothing";
        if (instructionList.orders[aI] != null)
            displayName = instructionList.orders[aI].displayName;

        actInstructionShowSlot.text = displayName;
    }

    public void EraseShowInstruction()
    {
        actInstructionShowSlot.text = "";
    }
}
```

MOVETYPEDROPDOWN

```
//////////////////////////////////////////////////////////////
//
// Class: MoveTypeDropdown
// Description: Class that manage the MoveType dropdown behaviour
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class MoveTypeDropdown : MonoBehaviour
{
```

```
    // Public variables
    public List<string> options = new List<string>();

    // Internal dropDown
    public Dropdown dropDown;

    // Delegate
    public delegate void MoveTypeDropDownDelegate(int t);
    public static event MoveTypeDropDownDelegate OnClick;

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();

        // Add the different options
        options.Add("MoveL");
        options.Add("MoveJ");
        options.Add("MoveC");

        // Adds the options to the dropdown options
        foreach (var option in options)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text = option });
        }
        dropDown.RefreshShownValue();
    }
    #endregion

    public void Click()
    {
        if (OnClick != null)
            OnClick.Invoke(dropDown.value);
    }
}
```

PRECISIONDROPDOWN

```
//////////////////////////////////////////////////////////////
//
// Class: PrecisionDropdown
// Description: Class that manage the Precision dropdown behaviour
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PrecisionDropdown : MonoBehaviour
{
    public List<string> options = new List<string>();
```

```
    // Internal dropDown
    public Dropdown dropDown;

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();

        // Add the different options
        options.Add("fine");
        options.Add("z0");
        options.Add("z1");
        options.Add("z5");
        options.Add("z10");
        options.Add("z15");
        options.Add("z20");
        options.Add("z30");
        options.Add("z40");
        options.Add("z50");
        options.Add("z60");
        options.Add("z80");
        options.Add("z100");
        options.Add("z150");
        options.Add("z200");

        // Adds the options to the dropdown options
        foreach (var option in options)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text = option });
        }
        dropDown.RefreshShownValue();
    }
    #endregion
}
```

SPEEDDROPDOWN

```
//////////////////////////////////////////////////////////////
//
// Class: SpeedDropdown
// Description: Class that manage the Speed dropdown behaviour
//
// Author: Pablo Giménez Suárez
// Last Review: 03/07/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;


public class SpeedDropdown : MonoBehaviour
{
```

```
    public List<string> options = new List<string>();

    // Internal dropDown
    public Dropdown dropDown;

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();

        // Add the different options
        options.Add("v5");
        options.Add("v10");
        options.Add("v20");
        options.Add("v30");
        options.Add("v40");
        options.Add("v50");
        options.Add("v60");
        options.Add("v80");
        options.Add("v100");
        options.Add("v150");
        options.Add("v200");
        options.Add("v300");
        options.Add("v400");
        options.Add("v500");
        options.Add("v600");
        options.Add("v800");
        options.Add("v1000");
        options.Add("v1500");
        options.Add("v2000");
        options.Add("v2500");

        // Adds the options to the dropdown options
        foreach (var option in options)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text = option });
        }
        dropDown.RefreshShownValue();
    }
    #endregion
}
```

ROBTARGETDROPDOWN

```
//////////////////////////////////////////////////////////////
//
// Class: RobTargetDropdown
// Description: Class that manage the Robtarget dropdown behaviour
//
// Author: Pablo Giménez Suárez
// Last Review: 08/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
```

```
public class RobTargetDropdown : MonoBehaviour
{
    // Public variables
    public List<string> options = new List<string>();

    public RobTargetList targetList;

    // Internal dropDown
    public Dropdown dropDown;

    #region Unity Methods
    // Start/Destroy
    void Start()
    {
        dropDown = transform.GetComponent<Dropdown>();
        dropDown.ClearOptions();
        RobTargetList.OnChanged += HandleOnChanged;
    }

    void OnDestroy()
    {
        RobTargetList.OnChanged -= HandleOnChanged;
    }
    #endregion

    // HandleOnChange
    void HandleOnChanged(RobTargetList targetList)
    {
        if (this.targetList == targetList)
            Prime(targetList);
    }

    // Initializer method
    public void Prime(RobTargetList robTargetList)
    {
        dropDown.ClearOptions();

        this.targetList = robTargetList;
        foreach (RobTarget target in this.targetList.targets)
        {
            dropDown.options.Add(new Dropdown.OptionData() { text =
target.displayName });
        }
        dropDown.RefreshShownValue();
    }
}
```

MOVEAXISMENUDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: MoveAxisMenuDisplay
// Description: Display class of the move axis menu
//
// Author: Pablo Giménez Suárez
```

```
// Last Review: 07/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class MoveAxisMenuDisplay : MonoBehaviour
{
    // Public variables
    public Text displayName;
    public Transform targetTransform;
    public CJointDisplay CJointDisplayPrefab;

    public List<CJointDisplay> cJointDisplays;

    public MoveAxisMenu moveAxisMenu;
    public Canvas canvas;

    #region  Unity Methods
    void Awake()
    {
        CJointDisplayPrefab = Resources.Load<CJointDisplay>("ABB
Library/Robots/UIPrefab/JointBar");
        canvas = GetComponent<Canvas>();
    }
    #endregion

    // Initilizer functions
    public void Prime(MoveAxisMenu moveAxisMenu)
    {
        displayName.text = moveAxisMenu.displayName;
        for (int a = 1; a < targetTransform.childCount; a++)
            Destroy(targetTransform.GetChild(a).gameObject);

        this.moveAxisMenu = moveAxisMenu;
        foreach (CJoint joint in this.moveAxisMenu.joints)
        {
            cJointDisplays.Add((CJointDisplay)Instantiate(CJointDisplayPrefab));
            cJointDisplays[cJointDisplays.Count -
1].transform.SetParent(targetTransform, false);
            cJointDisplays[cJointDisplays.Count - 1].Prime(joint);
        }
    }

    public void UpdateValues(MoveAxisMenu moveAxisMenu)
    {
        if(displayName != null)
            displayName.text = moveAxisMenu.displayName;

        if (moveAxisMenu != null)
            this.moveAxisMenu = moveAxisMenu;

        foreach (CJoint joint in this.moveAxisMenu.joints)
            cJointDisplays[this.moveAxisMenu.joints.IndexOf(joint)].Prime(joint);
    }
}
```

ROBOTLIBRARYDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: RobotLibraryDisplay
// Description: Class that shows the library of robots available
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class RobotLibraryDisplay : MonoBehaviour
{
    // Public variables
    public Transform targetTransform;
    public RobotLibraryIconDisplay robotLibraryIconDisplayPrefab;

    public RobotLibrary robotLibrary;

    #region Unity Methods
    void Awake()
    {
        robotLibraryIconDisplayPrefab =
Resources.Load<RobotLibraryIconDisplay>("ABB
Library/Robots/UIPrefab/RobotLibraryIconDisplaySlot");
    }
    #endregion

    // Initilizer method
    public void Prime(RobotLibrary robotLibrary)
    {
        this.robotLibrary = robotLibrary;
        foreach (RobotLibraryIcon robot in this.robotLibrary.robotIcons)
        {
            RobotLibraryIconDisplay display =
(RobotLibraryIconDisplay)Instantiate(robotLibraryIconDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(robot);
        }
    }
}
```

ROBOTLIBRARYICON

```
//////////////////////////////////////////////////////////////
//
// Class: RobotLibraryIcon
// Description: Class that stores the data of the prefabRobots
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
```

```
public class RobotLibraryIcon : MonoBehaviour
{
    public string displayName;
    public Sprite sprite;
    public int robType;
}
```

ROBOTLIBRARYICONDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: RobotLibraryIconDisplay
// Description: Class that shows the data of the different robots
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class RobotLibraryIconDisplay : MonoBehaviour
{
    // Visual variables
    public Text textName;
    public Image image;

    // Internal variable
    public RobotLibraryIcon robot;

    // Event management
    public delegate void RobotLibraryIconDisplayDelegate(RobotLibraryIcon
robot);
    public static event RobotLibraryIconDisplayDelegate onClick;

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        if (robot != null)
            Prime(robot);
    }
    #endregion

    // Initializer Method
    public void Prime(RobotLibraryIcon robot)
    {
        this.robot = robot;
        if (textName != null)
            textName.text = robot.displayName;
```

```
        if (image != null)
            image.sprite = robot.sprite;
    }

    // UI Method
    public void Click()
    {
        string displayName = "Nothing";
        if (robot != null)
            displayName = robot.displayName;

        if (onClick != null)
            onClick.Invoke(robot);
    }
}
```

TOOLLIBRARYDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: ToolLibraryDisplay
// Description: Class that shows the library of tool available
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class ToolLibraryDisplay : MonoBehaviour
{
    // Public variables
    public Transform targetTransform;
    public ToolLibraryIconDisplay toolLibraryIconDisplayPrefab;

    public ToolLibrary toolLibrary;

    #region Unity Methods
    void Awake()
    {
        toolLibraryIconDisplayPrefab =
Resources.Load<ToolLibraryIconDisplay>("ABB
Library/RobotStudioObjects/Tools/UIPrefab/ToolLibraryIconDisplaySlot");
    }
    #endregion

    // Initilizer method
    public void Prime(ToolLibrary toolLibrary)
    {
        this.toolLibrary = toolLibrary;
        foreach (ToolLibraryIcon tool in this.toolLibrary.toolIcons)
```

```
        {
            ToolLibraryIconDisplay display =
(ToolLibraryIconDisplay)Instantiate(toolLibraryIconDisplayPrefab);
            display.transform.SetParent(targetTransform, false);
            display.Prime(tool);
        }
    }
}
```

TOOLLIBRARYICON

```
//////////////////////////////////////////////////////////////
//
// Class: ToolLibraryIcon
// Description: Class that stores the icon display data of each tool
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class ToolLibraryIcon : MonoBehaviour
{
    public string displayName;
    public Sprite sprite;
    public int toolType;
}
```

TOOLLIBRARYICONDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: ToolLibraryIconDisplay
// Description: Class that shows the icon data of the different tools
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class ToolLibraryIconDisplay : MonoBehaviour
{
    // Visual variables
    public Text textName;
    public Image image;

    // Internal variable
    public ToolLibraryIcon tool;
```

```
    // Event management
    public delegate void ToolLibraryIconDisplayDelegate(ToolLibraryIcon tool);
    public static event ToolLibraryIconDisplayDelegate onClick;

    #region Unity Methods
    // Start is called before the first frame update
    void Start()
    {
        if (tool != null)
            Prime(tool);
    }
    #endregion

    // Initializer method
    public void Prime(ToolLibraryIcon tool)
    {
        this.tool = tool;
        if (textName != null)
            textName.text = tool.displayName;
        if (image != null)
            image.sprite = tool.sprite;
    }

    // UI method
    public void Click()
    {
        string displayName = "Nothing";
        if (tool != null)
            displayName = tool.displayName;

        if (onClick != null)
            onClick.Invoke(tool);
    }
}
```

CJOINT

```
//////////////////////////////////////////////////////////////
//
// Class: CJoint
// Description: Class that defines the behaviour of a robot joints
//
// Author: Pablo Giménez Suárez
// Last Review: 07/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class CJoint : MonoBehaviour
{
    // Public variables
    // Each joint has rotation angle
    public float angle;

    // Each joint has a max and min angle that can rotate
```

```
    public float maxAngle;
    public float minAngle;

    // Methods
    public float GetAngle() { return angle; }
    public void SetAngle(float a)
    {
        angle = a;
        // Clamps the angle of the joint
        if (angle > maxAngle)
            angle = maxAngle;
        else if (angle < minAngle)
            angle = minAngle;
    }
}
```

CJOINTDISPLAY

```
//////////////////////////////////////////////////////////////////
//
// Class: CJointDisplay
// Description: Class that shows and manage the data of a joint
//
// Author: Pablo Giménez Suárez
// Last Review: 05/08/2020
//
//////////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class CJointDisplay : MonoBehaviour
{
    // Public variables
    public CJoint joint;

    // Display variables
    public Text minAngleDisplay;
    public Text maxAngleDisplay;
    public InputField inputField;

    public Slider slider;

    // Delegate functionality
    public delegate void CjointDisplayDelegate(CJoint joint);
    public static event CjointDisplayDelegate OnValueChange;

    // Initializer method
    public void Prime(CJoint joint)
    {
        this.joint = joint;

        // The angle that is shown to the user is fixed to have the same momevent
that it will have in RobotStudio
        // MinMax angles
        minAngleDisplay.text = (-joint.maxAngle).ToString("F2");
        maxAngleDisplay.text = (-joint.minAngle).ToString("F2");
```

```
        slider.minValue = -joint.maxAngle;
        slider.maxValue = -joint.minAngle;

        // Real angle
        slider.value = -joint.angle;
        if (inputField != null)
            inputField.text = (-joint.angle).ToString("F2");
    }

    #region Methods for UI Elements
    public void ChangeAngle(float value)
    {
        if (inputField != null)
            inputField.text = value.ToString("F2");

        joint.SetAngle(-value);

        if (OnValueChange != null)
            OnValueChange.Invoke(joint);
    }

    public void SetAngle()
    {
        joint.SetAngle(-float.Parse(inputField.text));
        slider.value = -joint.GetAngle();
        inputField.text = (-joint.GetAngle()).ToString("F2");

        if (OnValueChange != null)
            OnValueChange.Invoke(joint);
    }
    #endregion
}
```

GENERICROBOT

```
//////////////////////////////////////////////////////////////
//
// Class: GenericRobot
// Description: Abstract class that unifies all the robots added to the program
//
// Author: Pablo Giménez Suárez
// Last Review: 07/08/2020
//
//////////////////////////////////////////////////////////////
using System.Collections.Generic;
using UnityEngine;

public abstract class GenericRobot : MonoBehaviour
{
    // Private parameters of Robots
    // Robot identifier
    [Header("Robot Identity")]
    public string displayName;
    public float robType;
    public int id;
```

```
    [Header("Joints Parameters")]
    // Number of joints
    public int nJoints;

    // Vector of Joints
    public List<CJoint> joints;

    // Joints transform
    public Transform[] tJoints;

    [Header("Relevant Transforms")]
    // Robot transforms
    public Transform robotPivot;
    public Transform toolReference;

    // Public methods
    // Get functions
    public int GetId() { return id; }
    public float GetRobType() { return robType; }
    public int GetNumJoints() { return nJoints; }
    public List<CJoint> GetJoints() { return joints;  }
    public CJoint GetJoint(int n)
    {
        if (n < nJoints)
            return joints[n];
        else
            return null;
    }

    // Set functions
    public void SetId(int i) { id = i; }
    public void SetName(string s)
    {
        if (displayName != null)
            displayName = s;
    }

    virtual public void MoveAxis(float newAngle, int n) { }
    virtual public void AxesMovement(float[] newAngles) { }
    virtual public void SetRobotPosition(Vector3 newPosition) { }
    virtual public void SetRobotRotation(Vector3 newRotation) { }
}
```

IRB120

```
//////////////////////////////////////////////////////////////
//
// Class: IRB120
// Description: Robot IRB 120 ABB Library
//
// Author: Pablo Giménez Suárez
// Last Review: 26/06/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
```

```
public class IRB120 : GenericRobot
{
    public override void MoveAxis(float newAngle, int n) // Moves an axis of the
robot
    {
        joints[n].SetAngle(newAngle);
        switch (n)
        {
            case 0:
                tJoints[0].localEulerAngles = new Vector3(0,
joints[0].GetAngle(), 0);
                break;

            case 1:
                tJoints[1].localEulerAngles = new Vector3(0, 0,
joints[1].GetAngle());
                break;

            case 2:
                tJoints[2].localEulerAngles = new Vector3(0, 0,
joints[2].GetAngle());
                break;

            case 3:
                tJoints[3].localEulerAngles = new Vector3(joints[3].GetAngle(),
0, 0);
                break;

            case 4:
                tJoints[4].localEulerAngles = new Vector3(0, 0,
joints[4].GetAngle());
                break;

            case 5:
                tJoints[5].localEulerAngles = new Vector3(joints[5].GetAngle(),
0, 0);
                break;

            default:
                Debug.LogError("Error 021: The robot has less joints");
                break;
        }
    }

    override public void AxesMovement(float[] newAngles) // Moves all the robot
axis
    {
        for (int i = 0; i < nJoints; i++)
        {
            joints[i].SetAngle(newAngles[i]);
        }

        tJoints[0].localEulerAngles = new Vector3(0, joints[0].GetAngle(), 0);
        tJoints[1].localEulerAngles = new Vector3(0, 0, joints[1].GetAngle());
        tJoints[2].localEulerAngles = new Vector3(0, 0, joints[2].GetAngle());
        tJoints[3].localEulerAngles = new Vector3(joints[3].GetAngle(), 0, 0);
        tJoints[4].localEulerAngles = new Vector3(0, 0, joints[4].GetAngle());
        tJoints[5].localEulerAngles = new Vector3(joints[5].GetAngle(), 0, 0);
    }
```

```
    override public void SetRobotPosition(Vector3 newPosition) // Set the robot
position
    {
        robotPivot.position = newPosition;
    }
    override public void SetRobotRotation(Vector3 newRotation) // Set the robot
rotation
    {
        robotPivot.eulerAngles = newRotation;
    }
}
```

ROBOTUIDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: RobotUIDisplay
// Description: Class that shows the data of a robot
//
// Author: Pablo Giménez Suárez
// Last Review: 07/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class RobotUIDisplay : MonoBehaviour
{
    // Text element of the scene
    public Text textName;

    // Internal robot
    public GenericRobot robot;

    // Delegate functionality
    public delegate void RobotUIDisplayDelegate(GenericRobot robot);
    public static event RobotUIDisplayDelegate OnClick;

    // Initializer method
    public void Prime(GenericRobot robot)
    {
        this.robot = robot;
        if (textName != null)
            textName.text = robot.displayName;
    }

    // UI Elements method
    public void Click()
    {
        if (OnClick != null)
            OnClick.Invoke(robot);
    }
}
```

INSTRUCTION

```
//////////////////////////////////////////////////////////////
//
// Class: Instruction
// Description: Class that contains the information needed to create a
instruction in Robotstudio
//
// Author: Pablo Giménez Suárez
// Last Review: 08/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public class Instruction
{
    // Variables for instruction
    public string displayName;
    private string moveType;
    private string displaySpeed;
    private string displayZone;
    private int speeddata;
    private int zonedata;
    public RobTarget[] targets;

    // Constructor
    public Instruction(string n, RobTarget[] r, int m, int s, int z)
    {
        targets = r;

        // MoveType selector
        switch (m)
        {
            case 0:
                moveType = "L";
                break;

            case 1:
                moveType = "J";
                break;

            case 2:
                moveType = "C";
                break;

            default:
                Debug.Log("ERROR 022: Wrong movetype");
                break;
        }

        if (s >= 0 && s <= 19)
        {
            speeddata = s;
            // Speed display switch
            switch (speeddata)
            {
                case 0:
                    displaySpeed = "v5";
                    break;
```

```
            case 1:
                displaySpeed = "v10";
                break;

            case 2:
                displaySpeed = "v20";
                break;

            case 3:
                displaySpeed = "v30";
                break;

            case 4:
                displaySpeed = "v40";
                break;

            case 5:
                displaySpeed = "v50";
                break;

            case 6:
                displaySpeed = "v60";
                break;

            case 7:
                displaySpeed = "v80";
                break;

            case 8:
                displaySpeed = "v100";
                break;

            case 9:
                displaySpeed = "v150";
                break;

            case 10:
                displaySpeed = "v200";
                break;

            case 11:
                displaySpeed = "v300";
                break;

            case 12:
                displaySpeed = "v400";
                break;

            case 13:
                displaySpeed = "v500";
                break;

            case 14:
                displaySpeed = "v600";
                break;

            case 15:
                displaySpeed = "v800";
                break;
```

```
                case 16:
                    displaySpeed = "v1000";
                    break;

                case 17:
                    displaySpeed = "v1500";
                    break;

                case 18:
                    displaySpeed = "v2000";
                    break;

                case 19:
                    displaySpeed = "v2500";
                    break;

                default:
                    Debug.Log("ERROR 023: That speed doesn't exist");
                    break;
            }
        }
        else
        {
            Debug.Log("ERROR 023: That speed doesn't exist");
            return;
        }

        if (z >= 0 && z <= 14)
        {
            zonedata = z;
            // Precision display switch
            switch (zonedata)
            {
                case 0:
                    displayZone = "fine";
                    break;

                case 1:
                    displayZone = "z0";
                    break;

                case 2:
                    displayZone = "z1";
                    break;

                case 3:
                    displayZone = "z5";
                    break;

                case 4:
                    displayZone = "z10";
                    break;

                case 5:
                    displayZone = "z15";
                    break;

                case 6:
                    displayZone = "z20";
                    break;
```

```
                case 7:
                    displayZone = "z30";
                    break;

                case 8:
                    displayZone = "z40";
                    break;

                case 9:
                    displayZone = "z50";
                    break;

                case 10:
                    displayZone = "z60";
                    break;

                case 11:
                    displayZone = "z80";
                    break;

                case 12:
                    displayZone = "z100";
                    break;

                case 13:
                    displayZone = "z150";
                    break;

                case 14:
                    displayZone = "z200";
                    break;

                default:
                    Debug.Log("ERROR 024: That precision doesn't exist");
                    break;
            }
        }
        else
        {
            Debug.Log("ERROR 024: That precision doesn't exist");
            return;
        }

        if (m == 2)
        {
            displayName = n + " [" + targets[0].displayName + "," +
targets[1].displayName + "," + displaySpeed + "," + displayZone + "]";
        }
        else
        {
            displayName = n + " [" + targets[0].displayName + "," + displaySpeed
+ "," + displayZone + "]";
        }

    }

    // Get Functions
    public int GetSpeed() { return speeddata; }
    public int GetPrecision() { return zonedata; }
    public string GetMoveType() { return moveType; }
}
```

INSTUCTIONUIDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: InstructionUIDisplay
// Description: Class that shows the data of a Instruction
//
// Author: Pablo Giménez Suárez
// Last Review: 03/07/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class InstructionUIDisplay : MonoBehaviour
{
    // Text element of the scene
    public Text textName;

    // Internal robTarget
    public Instruction order;

    // Delegate functionality
    public delegate void InstructionUIDisplayDelegate(Instruction order);
    public static event InstructionUIDisplayDelegate OnClick;

    // Initializer method
    public void Prime(Instruction order)
    {
        this.order = order;
        if (textName != null)
            textName.text = order.displayName;
    }

    // UI Elements method
    public void Click()
    {
        if (OnClick != null)
            OnClick.Invoke(order);
    }
}
```

ROBTARGET

```
//////////////////////////////////////////////////////////////
//
// Class: RobTarget
// Description: Class that contains the information needed to create a robTarget
in Robotstudio
//
```

```
// Author: Pablo Giménez Suárez
// Last Review: 26/06/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using System.Collections.Generic;

public class RobTarget : MonoBehaviour
{
    // Name
    public string displayName;
    private int id;

    // Object
    public Transform target;

    // Transform data
    private Vector3 robotStudioPosition = Vector3.zero;
    private Vector3 robotStudioRotation = Vector3.zero;

    // Robot config data
    private int[] cf = new int[] { 0, 0, 0, 0 };

    // Joint data
    public float[] angles;

    // Constructor
    public void Prime(int d, float robType, List<CJoint> joints, Transform t,
string name)
    {
        // Target name
        displayName = name;

        // Identifier
        id = d;

        angles = new float[joints.Count];
        // Joints
        for(int i = 0; i < joints.Count; i++)
        {
            angles[i] = joints[i].angle;
        }


        // World position
        target.position = t.position;
        target.rotation = t.rotation;

        // robTarget position
        robotStudioPosition.x = t.position.x * 1000;
        robotStudioPosition.y = t.position.z * 1000;
        robotStudioPosition.z = t.position.y * 1000;

        // robTarget global rotation with quaternions
        robotStudioRotation.x = -t.eulerAngles.x;
        robotStudioRotation.y = -t.eulerAngles.z;
        robotStudioRotation.z = -t.eulerAngles.y;

        // Robot configuration data
        if (robType == 0)
        {
```

```
            // cf1
            cf[0] = RobConf(-joints[0].GetAngle());

            // cf4
            cf[1] = RobConf(-joints[3].GetAngle());

            // cf6
            cf[2] = RobConf(-joints[5].GetAngle());

            // cfx
            cf[3] = 0; // cfx is not used in IRB120 robots
        }
    }

    // Rob conf selector
    private int RobConf(float a)
    {
        if (a >= 0)
        {
            return (int)(a/90f);
        }
        else
        {
            return (int)((a/90f) - 1);
        }
    }

    // Get functions
    public string GetName() { return displayName;  }
    public int GetId() { return id; }
    public Vector3 GetRSPosition() { return robotStudioPosition; }
    public Vector3 GetRSRotation() { return robotStudioRotation; }
    public int GetRSConfig(int n) { return cf[n]; }
    public Transform GetRobTargetTransform() { return target; }

}
```

ROBTARGETUIDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: RobTargetUIDisplay
// Description: Class that shows the data of a robtarget
//
// Author: Pablo Giménez Suárez
// Last Review: 08/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class RobTargetUIDisplay : MonoBehaviour
{
    // Text element of the scene
    public Text textName;
```

```
    // Internal robTarget
    public RobTarget target;

    // Delegate functionality
    public delegate void RobTargetUIDisplayDelegate(RobTarget target);
    public static event RobTargetUIDisplayDelegate OnClick;

    // Initializer method
    public void Prime(RobTarget target)
    {
        this.target = target;
        if (textName != null)
            textName.text = target.displayName;
    }

    // UI Elements method
    public void Click()
    {
        if (OnClick != null)
            OnClick.Invoke(target);
    }
}
```

TOOL

```
//////////////////////////////////////////////////////////////
//
// Class: Tool
// Description: Abstract Class of all the tools that can be added
//
// Author: Pablo Giménez Suárez
// Last Review: 26/06/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;

public abstract class Tool : MonoBehaviour
{
    // Identifying variables
    public string displayName;
    public int id;
    public int toolType;

    // Boolean that verifies if the robot holds the tool
    [SerializeField]
    protected bool robHold;

    // Transform of the TCP of the tool
    [SerializeField]
    protected Transform tcp;

    // Transform of the wrist of the tool
    public Transform wrist;
    private Vector3 pastPosition;
    private Quaternion pastRotation;
```

```
    // Get functions
    public int GetId() { return id; }
    public Transform GetTcp() { return tcp; } // Returns the tcp of the tool
    public bool GetToolState() { return robHold; } // Returns the robHold state

    // Public functions
    public void BindTool(Transform newParent) // Binds the tool to the robot
    {
        pastPosition = wrist.position;
        pastRotation = wrist.rotation;

        robHold = true;
        wrist.position = newParent.position;
        wrist.rotation = newParent.rotation;
        wrist.SetParent(newParent);
    }

    public void UnBindTool() // Unbinds the tool from the robot
    {
        robHold = false;
        wrist.SetParent(null);
        wrist.position = pastPosition;
        wrist.rotation = pastRotation;
    }


    // Set functions
    public void SetId(int n) { id = n; }
    public void SetName(string s) { displayName = s; }
    public void SetTcp(Vector3 pos, Quaternion rot) // Set the tcp position of
the tool
    {
        tcp.position = pos;
        tcp.rotation = rot;
    }
    public void SetWrist(Vector3 pos, Quaternion rot) // Public function to move
the tool
    {
        wrist.position = pos;
        wrist.rotation = rot;
    }
}
```

TOOL0

```
//////////////////////////////////////////////////////////////
//
// Class: Tool0
// Description: Default Tool of RobotStudio
//
// Author: Pablo Giménez Suárez
// Last Review: 08/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
```

```
public class Tool0 : Tool // Tool0 has the same functionality as a generic tool
{ }
```

TOOLUIDISPLAY

```
//////////////////////////////////////////////////////////////
//
// Class: ToolUIDisplay
// Description: Class that shows the data of a tool
//
// Author: Pablo Giménez Suárez
// Last Review: 08/08/2020
//
//////////////////////////////////////////////////////////////
using UnityEngine;
using UnityEngine.UI;

public class ToolUIDisplay : MonoBehaviour
{
    // Text element of the scene
    public Text textName;

    // Internal robot
    public Tool tool;

    // Delegate functionality
    public delegate void ToolUIDisplayDelegate(Tool tool);
    public static event ToolUIDisplayDelegate OnClick;

    // Initializer method
    public void Prime(Tool tool)
    {
        this.tool = tool;
        if (textName != null)
            textName.text = tool.displayName;
    }

    // UI Elements method
    public void Click()
    {
        if (OnClick != null)
            OnClick.Invoke(tool);
    }
}
```