COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS

Autor: Francisco Javier Domínguez Sánchez-Girón

Director: José Antonio Rodríguez Mondéjar

Madrid

Agosto de 2019

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. Francisco Javier Domínguez Sánchez-Girón DECLARA ser el titular de los derechos de propiedad intelectual de la obra: APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente).*

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e

intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a .26….. de ……agosto…….... de 2019…

ACEPTA

Fdo……

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

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

APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2018-2019 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.: Francisco Javier Domínguez Sánchez-Girón Fecha: 26 / 08 / 19

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: José Antonio Rodríguez Mondéjar Fecha: 27 / 8 / 2019

mondejar@comillas.edu

Firmado digitalmente por mondejar@comillas.edu
Nombre de reconocimiento (DN): cn=mondejar@comillas.edu
Fecha: 2019.08.27 13:27:23 +02'00'

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS

Autor: Francisco Javier Domínguez Sánchez-Girón

Director: José Antonio Rodríguez Mondéjar

Madrid

Agosto de 2019

APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS

Autor: Domínguez Sánchez-Girón, Francisco Javier.

Director: Rodríguez Mondéjar, José Antonio.

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

Este proyecto consiste en el desarrollo de una aplicación de ordenador que permita la programación de robots industriales mediante realidad virtual, aprovechando al máximo las ventajas que ésta supone.

Aunque ya existen diversos prototipos y proyectos de desarrollo de la realidad virtual aplicada al manejo y programación de robots industriales con distintos enfoques, hasta el momento no se ha aprovechado por completo el potencial de esta tecnología. En caso de hacerse, la realidad virtual podría convertirse en el método estándar en programación de robots por permitir una programación remota y libre de muchas de las constricciones que conlleva la programación in situ.

Este proyecto tiene por objeto el desarrollo de una aplicación eficaz y fiable para programar los brazos robóticos industriales IRB120 de ABB presentes en el Laboratorio de Sistemas Digitales 1 de ICAI. Se pretende que la aplicación permita un nivel de utilización similar al del dispositivo estándar de ABB para operación del robot llamado *FlexPendant*. El proyecto parte de una aplicación desarrollada como Trabajo Fin de Máster por Carlos Álvarez Vereterra en 2018 cuya única funcionalidad consiste en marcar con un mando de realidad virtual hasta cinco puntos en el espacio para que el robot los recorra. La aplicación carece de interfaz y el robot de herramienta, por lo que ambos elementos han sido implementados como parte de este proyecto.

La aplicación se ha desarrollado en Unity, software que controla la realidad virtual, y RobotStudio, software que controla el robot. Respecto a la programación, todas las instrucciones para el robot se crean y almacenan en Unity y sólo se requiere a RobotStudio para comprobar la viabilidad de algunas de ellas o ejecutarlas. La aplicación emplea el complemento SteamVR para controlar la realidad virtual en Unity, ya que es compatible con prácticamente todos los sistemas de realidad virtual y permite programarlos de forma genérica.

Las instrucciones para el robot se corresponden con acciones simples que el robot puede realizar. En Unity se ha creado una clase llamada *Action* que representa el conjunto de estas acciones. Dicha clase es de tipo *abstract*, lo que implica que, al ser creada, una entidad *Action*, debe ser especificada como una de las siguientes subclases:

- Wait. Representa una espera durante un tiempo determinado.
- GoTo. Representa la acción de llevar la herramienta del robot a un punto en el espacio.

- MoveAxis. Representa la acción de alcanzar una combinación determinada de ángulos en los ejes del robot.
- ActDeact. Representa la acción de activar o desactivar la herramienta.

La programación del robot se basa en el uso de estas clases y de la propiedad de polimorfismo, que permite que se llame a una función de una clase y que se ejecute de distinta forma según el tipo de subclase del que se trate, pero sin necesidad de comprobarlo. Por tanto, cada *Action* cuenta con su propia definición de una función llamada *Execute*, la cual realiza las operaciones y comunicaciones necesarias para la correcta ejecución de la acción que dicha *Action* representa.

El usuario puede comprobar la viabilidad de las *Actions* antes de crearlas, realizándose la comunicación correspondiente, para posteriormente añadirlas secuencialmente a una lista de *Actions* que puede ejecutarse secuencialmente y en bucle a modo de programa. Durante el proceso de comprobación, el robot virtual se mueve independientemente del real para que el usuario pueda visualizar lo que está programando antes de ejecutarlo. Durante la ejecución, el robot virtual imita el movimiento del real para que el usuario pueda ver en tiempo real cómo actúa el robot real. El proceso de comprobación, creación y ejecución de *Actions* se lleva a cabo a través de una interfaz de usuario sencilla, intuitiva, fácil de navegar y que el usuario puede activar o desactivar en cualquier momento.

Las comunicaciones relativas a la comprobación y ejecución de instrucciones se realizan mediante el protocolo TCP/IP, siempre iniciándose desde Unity y concluyéndose cuando RobotStudio le proporciona la respuesta. Dicha respuesta puede, por un lado, incluir la información necesaria para la creación de la *Action*, como en el caso de la comprobación de una *GoTo*, ya que se requieren los ángulos en los ejes para realizar el movimiento de muestra, o bien simplemente consistir en un aviso que indica la correcta ejecución de la instrucción enviada. Cuando se ejecuta una instrucción de movimiento, RobotStudio envía de forma continua mediante el protocolo UDP/IP los valores de los ángulos del robot real a Unity, para que el virtual pueda imitar el movimiento.

La aplicación final cumple con las funciones básicas buscadas, pero es incapaz de generar trayectorias fluidas debido a que las instrucciones de movimiento no se ejecutan de forma consecutiva en RobotStudio. Esto implica que el robot pasa por todas las posiciones que se le indiquen siempre con total precisión y parándose brevemente al alcanzarlos, sea o no necesario. Por otro lado, la interacción con la interfaz no funciona exactamente como se esperaba, lo cual afecta a la experiencia del usuario, pero no al correcto funcionamiento de la aplicación.

Aunque la aplicación final está lejos de ofrecer las funcionalidades que ofrece el *FlexPendant*, sienta las bases para que se pueda desarrollar un producto completo y profesional sin requerir más de un proyecto de la magnitud y presupuesto del proyecto desarrollado en este documento.

APPLICATION OF VIRTUAL REALITY TO ROBOT PROGRAMMING

Author: Domínguez Sánchez-Girón, Francisco Javier.

Director: Rodríguez Mondéjar, José Antonio.

Collaboring Institution: ICAI – Universidad Pontificia Comillas.

PROJECT SUMMARY

This project consists in the development of a computer application that allows the programming of industrial robots through virtual reality, taking full advantage of its use.

Although there are already several prototypes and projects developing the application of virtual reality to the operation and programming of industrial robots with different approaches, so far, the potential of this technology has not been fully exploited. Was this to be the case, virtual reality could become the standard method in robot programming by allowing remote programming and free of most of the constraints that on-site programming involves.

The purpose of this project is to develop an efficient and reliable application to program the ABB IRB120 industrial robotic arms present at the ICAI Digital Systems Laboratory 1. The application is intended to allow a level of utilization similar to that of the standard ABB device for robot operation called *FlexPendant*. The project is based on an application developed as Master's Thesis by Carlos Álvarez Vereterra in 2018. That application's only functionality consists in marking, through the use of a virtual reality remote control, up to five points in space for the robot take its tool to. The application lacks an interface and the robot lacks its tool, so both elements have been implemented as part of this project.

The application has been developed in Unity, software that controls virtual reality, and RobotStudio, software that controls the robot. Regarding programming the robot, all instructions given to the robot are created and stored in Unity and RobotStudio is only required to verify the viability of some of them or execute them. The application uses the SteamVR plug-in to control virtual reality in Unity, since it is compatible with practically all virtual reality systems and allows programming them generically.

The instructions for the robot correspond to the simplest actions it is able to perform. In Unity, a class called *Action* that represents the whole of these actions has been created. Said class is of *abstract* type, which implies that every time an *Action* entity is created, it must be specified as one of the following subclasses:

- *Wait*. Represents waiting for a certain period of time.
- *GoTo*. Represents the action of taking the tool to a point in space.
- *MoveAxis*. Represents the action of reaching a certain combination of angles in the axes of the robot.
- *ActDeact*. Represents the action of activating or deactivating the tool.

Robot programming is based on the use of these classes and the polymorphism property, which allows calling a function within a class, with it having different effects

depending on the type of subclass, but without the need to check the corresponding one. Therefore, each *Action* has its own definition of a function called *Execute*, which performs the operations and communications required for the correct execution of the command that said *Action* represents.

The user can check the viability of the *Actions* before creating them, carrying out the corresponding communication, to then add them sequentially to a list of *Actions* that can be executed sequentially and looped as a program. During the verification process, the virtual robot moves independently of the real one so that the user can visualize what is being programmed before executing it. During execution, the virtual robot imitates the movement of the real one so that the user can see in real time how the real robot acts. The process of checking, creating and executing *Actions* is carried out through a simple, intuitive, easy-to-navigate user interface that the user can activate or deactivate at any time.

Communications regarding the verification and execution of instructions are made using the TCP/IP protocol, starting in Unity and ending when RobotStudio provides a response. Said response can either include the information necessary for the creation of the *Action*, as in the case of the verification of a *GoTo*, since the angles on the axes are required to perform the sample movement, or simply consist of a warning that indicates the correct Execution of the instruction sent. When a movement instruction is executed, RobotStudio continuously sends the angle values of the real robot to Unity using the UDP/IP protocol, so that the virtual one can imitate the movement.

The final application fulfills the basic functions as intended but is unable to generate smooth robot paths because of the movement instructions not being executed consecutively in RobotStudio. This means the robot always reaches the given positions with absolute precision and briefly stopping afterwards, even when it is not necessary. Moreover, the interaction between the user and the interface does not work exactly as designed, affecting the user experience, but not the adequate operation of the application.

Although the final application is far from offering the features offered by the *FlexPendant*, it lays the foundation for a complete and professional product to be developed without requiring more than one project of the magnitude and budget of the one described in this document.

Índice de la memoria

Índice de figuras

DOCUMENTO Nº1, MEMORIA

Capítulo 1 Introducción

Este proyecto consiste en el desarrollo de una aplicación de ordenador que explote las ventajas que ofrece la realidad virtual en la programación de robots industriales. En este documento se explica el trabajo realizado y el funcionamiento de la aplicación.

1.1 Objetivo

El objetivo de este proyecto es el desarrollo de una aplicación basada en realidad virtual que permita programar de forma eficaz y fiable los brazos robóticos industriales IRB120 de ABB presentes en el Laboratorio de Sistemas Digitales 1 de ICAI.

La aplicación debe ser precisa y fiable, contar con una interfaz simple e intuitiva y permitir programar como mínimo las acciones más comunes para el robot, como llevar la herramienta a un punto y activarla o desactivarla. Los menús de la interfaz deben presentar la información útil para el operario de una forma visual y fácilmente comprensible. Debería permitir una utilización similar a la que permite el dispositivo estándar de ABB para operación del robot llamado *FlexPendant*.

1.2 Estado del arte

Aunque ya existen diversos prototipos y proyectos de desarrollo de la realidad virtual aplicada al manejo y programación de robots industriales con distintos enfoques, hasta el momento no se ha aprovechado por completo el potencial de esta tecnología. En caso de hacerse, la realidad virtual podría convertirse en el método estándar en programación de robots por permitir una programación remota y libre de muchas de las constricciones que conlleva la programación in situ. Entre las principales iniciativas destacan:

- Trabajo de la *Slovak University of Technology in Bratislava* en el cual se analizan los beneficios de implementar realidad virtual para la programación de robots industriales [1].
- Funcionalidades de realidad virtual desarrolladas por ABB e integradas en su software de control de robots RobotStudio. Permiten al usuario experimentar una simulación altamente precisa a través de realidad virtual, pero no permiten programar los robots [2].
- Prototipo desarrollado por el laboratorio CSAIL del MIT para uso con Oculus Rift y parcialmente financiado por Boeing. Se trata de una aplicación que permite operar mediante realidad virtual un brazo robótico industrial Baxter para realizar acciones sencillas [3].

1.3 Aplicación base

El proyecto parte de una aplicación desarrollada como Trabajo Fin de Máster por Carlos Álvarez Vereterra en 2018. Dicha aplicación consta de una única funcionalidad respecto a los robots, que consiste en marcar con un mando de realidad virtual hasta cinco puntos en el espacio para que el robot los recorra. La aplicación carece de interfaz y el robot de herramienta, por lo que ambos elementos han sido implementados como parte de este proyecto [4].

Figura 1: Aplicación base.

1.4 Formación

Para poder empezar a trabajar en el proyecto me fue necesario aprender a utilizar tanto Unity como RobotStudio, ya que no contaba con ningún conocimiento previo. En el caso de Unity aprendí principalmente mediante un curso online de la página web Udemy enfocado al desarrollo de aplicaciones de realidad virtual y certificado por la empresa desarrolladora de Unity [5], pero además he recurrido con mucha frecuencia a los manuales de referencia de Unity [6] [7]. En el caso de RobotStudio, he utilizado documentación proporcionada por el director del proyecto, diversos vídeos colgados en la web [8] y la documentación de referencia de ABB [9] [10].

1.5 Principio de funcionamiento

La aplicación se ha desarrollado en Unity, software que controla la realidad virtual, y RobotStudio, software que controla el robot. En Unity se ha programado en C# y en RobotStudio en RAPID. Respecto a la programación del robot, todas las instrucciones se crean y almacenan en Unity y sólo se requiere a RobotStudio para comprobar la viabilidad de algunas de ellas o ejecutarlas. La aplicación emplea el complemento SteamVR para controlar

la realidad virtual en Unity, ya que es compatible con prácticamente todos los sistemas de realidad virtual y permite programarlos de forma genérica.

Las instrucciones para el robot se corresponden con las acciones más simples que el robot puede realizar. En Unity se ha creado una clase llamada *Action* que representa el conjunto de estas acciones. Dicha clase es de tipo *abstract*, lo que implica que, al ser creada, una entidad *Action*, debe ser especificada como una de las siguientes subclases:

- Wait. Representa una espera durante un tiempo determinado.
- GoTo. Representa la acción de llevar la herramienta del robot a un punto en el espacio.
- MoveAxis. Representa la acción de alcanzar una combinación determinada de ángulos en los ejes del robot.
- ActDeact. Representa la acción de activar o desactivar la herramienta.

La programación del robot se basa en el uso de estas clases y de la propiedad de polimorfismo, que permite que se llame a una función de una clase y que se ejecute de distinta forma según el tipo de subclase del que se trate, pero sin necesidad de comprobarlo. Por tanto, cada *Action* cuenta con su propia definición de una función llamada *Execute*, la cual realiza las operaciones y comunicaciones necesarias para la correcta ejecución de la acción que dicha *Action* representa.

El usuario puede comprobar la viabilidad de las *Actions* antes de crearlas, realizándose la comunicación correspondiente, para posteriormente añadirlas secuencialmente a una lista de *Actions* que puede ejecutarse secuencialmente y en bucle a modo de programa. Durante el proceso de comprobación, el robot virtual se mueve independientemente del real para que el usuario pueda visualizar lo que está programando antes de ejecutarlo. Durante la ejecución, el robot virtual imita el movimiento del real para que el usuario pueda ver en tiempo real cómo actúa el robot real. El proceso de comprobación, creación y ejecución de *Actions* se lleva a cabo a través de una interfaz de usuario sencilla, intuitiva, fácil de navegar y que el usuario puede activar o desactivar en cualquier momento.

Las comunicaciones relativas a la comprobación y ejecución de instrucciones se realizan mediante el protocolo TCP/IP, siempre iniciándose desde Unity y concluyéndose cuando RobotStudio le proporciona la respuesta. Dicha respuesta puede, por un lado, incluir la información necesaria para la creación de la *Action*, como en el caso de la comprobación de una *GoTo*, ya que se requieren los ángulos en los ejes para realizar el movimiento de muestra, o bien simplemente consistir en un aviso que indica la correcta ejecución de la instrucción enviada. Cuando se ejecuta una instrucción de movimiento, RobotStudio envía de forma continua mediante el protocolo UDP/IP los valores de los ángulos del robot real a Unity, para que el virtual pueda imitar el movimiento.

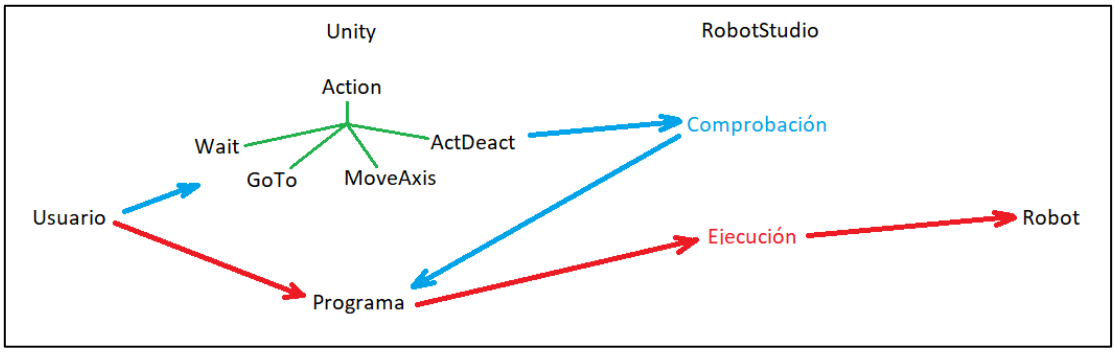

Figura 2: Esquema de funcionamiento de la nueva aplicación.

Capítulo 2 Desarrollo de la realidad virtual mediante Unity

Unity es el software empleado para el desarrollo de la parte principal de la aplicación: el entorno y la interfaz de realidad virtual y los algoritmos de programación.

2.1 Usuario virtual

El usuario virtual es el objeto controlado por el usuario y a través del cual percibe la realidad virtual.

2.1.1 Objeto del usuario

Los principales elementos que componen el objeto del usuario virtual se explican a continuación y su jerarquía se muestra en la figura 3:

- *PlayerOculus*. Un objeto a modo de cuerpo que además engloba el resto de los componentes. Este objeto carece de lo necesario para poder colisionar con otros objetos y por lo tanto el usuario es técnicamente incorpóreo. Esto permite que el usuario pueda atravesar cualquier objeto para colocarse donde le plazca. Este objeto contiene el código *User*, que controla la parte que afecta directamente al usuario de lo desarrollado en este proyecto.
- *VRCamera*. Las gafas de realidad virtual, que se mueven exactamente como las reales y contienen un componente de tipo Camera propio de Unity, la visión del usuario.
- *Hand1* y *Hand2*. Los mandos de realidad virtual, que se mueven exactamente como los reales y cuentan con componentes *Hand*, *MonoBehaviour* propio de SteamVR y que se utiliza para detectar y manejar la interacción del usuario con los mandos en el mundo real. Respecto a las funcionalidades de la interfaz, el derecho se utiliza para interactuar con los menús de la interfaz, por lo que lleva un objeto con el *MonoBehaviour MenuPointer*, y el izquierdo para señalar puntos que el robot tiene que alcanzar, por lo que lleva el *MonoBehaviour ViveController*, ambos códigos se explican en el apartado 2.4.

Figura 3: Jerarquía del objeto del usuario.

2.1.2 Código *User*

El código *User* es el código encargado de las funciones que afectan directamente al usuario de la aplicación. Se trata de un objeto *MonoBehaviour* que permite activar o desactivar la interfaz, mover y rotar artificialmente al usuario y teletransportarlo a su posición inicial.

Las variables que emplea son las siguientes:

- Como variables públicas:
  - *leftHand* y *rightHand*. Variables del tipo *Hand*, cada una se refiere a uno de los dos mandos, el izquierdo o el derecho.
  - *user*. Objeto de Unity que se corresponde con el cuerpo del usuario.
  - *cam*. Objeto de tipo *Camera* que se corresponde con la visión del usuario.
  - *Menu*. El objeto que alberga la interfaz.
- Como variables privadas:
  - *camObj*. El objeto que contiene el elemento *cam*, es decir, la versión virtual de las gafas de realidad virtual.
  - *UserTransform*. Variable auxiliar del tipo T*ransform* que se utilizará para acceder fácilmente al componente *Transform* del usuario.
  - *menuIsActive*. Variable booleana que indica si la interfaz está activada (es visible) o no.
  - *speedRot* y *speedMov*. Valores de velocidad de rotación y traslación del usuario respectivamente. Sus valores han sido determinados empíricamente.
  - *menuMan*. El elemento *MenuManager*, el código que controla la interfaz y que se explica más adelante en el documento.

```
public Hand leftHand;
public Hand rightHand;
public GameObject Menu; // menu interface object
public Camera cam; // VR camera
public GameObject user; // user object
public MenuPointer pointer;

private GameObject camObj; // camera object
private Transform userTransform;
private bool menuIsActive = false; // indicates whether the menu is active
(true) or not (false)
private float speedRot = 0.1f; // speed of forced user rotation
private float speedMov = 0.003f; // speed of forced user translation
private Vector3 initPos; // initital position
private Quaternion initRot; // initial rotation
private MenuManager menuMan;
```

Figura 4: Variables del código *User*.

En la función *Update* se llama a las funciones principales y se fija la altura a la que se puede encontrar el usuario, de manera que siempre permanezca a la misma. El valor de altura utilizado ha sido determinado empíricamente.

```
private void Update() // Update is called once per frame
{
    ManageMenu();
    Move();
    Rotate();
    if (user.transform.position.y != -0.5f) // -0.5 as position in y axis has
    been empirically determined to be most comfortable for using the
    application
    {
        user.transform.position = new Vector3(user.transform.position.x, -
        0.5f, user.transform.position.z);
    }
}
```

Figura 5: *Update* del código *User*.

La función *Move* permite al usuario desplazarse artificialmente hacia delante o hacia detrás en función de la posición vertical del *joystick* izquierdo y hacia la derecha o hacia la izquierda en función de su posición horizontal, ambos movimientos tomando como referencia la visión del propio usuario. Primero se crea un vector para cada tipo de movimiento, uno se igualará a la dirección frontal (*forward*) de la visión del usuario y el otro a la dirección derecha (*right*), ambos son multiplicados por la velocidad de movimiento y por el *Input* correspondiente del *joystick*. El *Input* del *joystick* es un vector cuyo primer elemento es su posición horizontal y el segundo, la vertical. Si el *joystick* está en reposo, ambos elementos toman el valor 0, a la derecha o arriba, el elemento correspondiente toma el valor 1, a la izquierda o abajo, -1, y entre medias, todos los valores proporcionales con una cierta resolución. Finalmente se utiliza el método *Translate* en la dirección y con la magnitud de la suma de los dos vectores mencionados y en coordenadas absolutas.

```
private void Move() // move the user according to controller input
{
    Vector3 speedF = cam.transform.forward * speedMov *
    leftHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1];
    // left controller's vertical axis
    Vector3 speedR = cam.transform.right * speedMov *
    leftHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[0];
    // left controller's horizontal axis
    transform.Translate(speedF + speedR,Space.World);
}
```

Figura 6: Función *Move* de *User*.

Dado que esta función desplaza al usuario en función de su visión, el usuario podría subir o bajar considerablemente respecto al suelo o simplemente no ser capaz de quedarse a una altura adecuada para manejar la aplicación. Sin embargo, esta posibilidad es anulada mediante la limitación de altura ya mencionada e incluida en su *Update*. Esto hace que el usuario solo pueda desplazarse artificialmente en su plano horizontal, de manera que, si no mira en una dirección exactamente paralela a este, no podrá moverse a plena velocidad. No obstante, este código es el más simple posible y esta no es una funcionalidad importante, sólo ha sido implementada para que el operario pueda trabajar desde la posición que prefiera sin tener que alcanzarla en el mundo real.

La función *Rotate* emplea el mismo proceso que *Move*, pero utilizando la posición horizontal del joystick derecho y el método *Rotate* en coordenadas relativas, de manera que el usuario rote respecto a su eje Y propio, es decir, respecto a un eje perpendicular al suelo sobre el que se encuentre.

```
private void Rotate() // rotate the user according to controller input
{
    float yaw = speedRot *
    rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[0];
    // right controller's horizontal axis
    camObj.transform.Rotate(new Vector3(0, yaw, 0),Space.Self);
}
```

Figura 7: Función *Rotate* de *User*.

Cabe destacar que los valores de velocidades de rotación y traslación son extremadamente bajos para paliar los posibles mareos que producen este tipo de funcionalidades en usuarios de realidad virtual. Sin embargo, también se ha demostrado que las personas pueden adaptarse y dejar de padecer estos mareos. En última instancia, queda a elección del usuario el hacer uso o no de esta característica.

La función *Go0* asigna los valores iniciales a la posición y a la rotación del usuario, por si este quisiese restablecerlas, y desactiva la interfaz, ya que esta función sólo puede llamarse a través de un botón de la interfaz y no es lógico dejarla visible en un lugar en el que el usuario ya no está.

```
    public void Go0() // teleport user to initial position and with same rotation
    values
    {
        transform.position = initPos;
        transform.rotation = initRot;
        menuMan.ManageMenus(false, true); // deactivate the interface visibility
        and reset the menu visibility order
    }
```

Figura 8: Función *Go0* de *User*.

La función *ManageMenu* se encarga de activar y desactivar la interfaz cuando el usuario pulsa el botón *Grip* derecho (botón en la empuñadora del mando). Si la interfaz pasa a estar activa, se le asigna la posición necesaria para que aparezca delante del usuario y su misma rotación, ya que por cómo está creada, de esta manera queda orientada hacia el usuario. A continuación, llama a la función *NotVisible* del *MenuPointer*. Por último, la interfaz se activa y desactiva llamando a la función *ManageMenus* del *MenuManager*, pasándole el indicador *menuIsActive* como argumento. Estas dos últimas funciones mencionadas se explican más adelante en el documento.

```
public void ManageMenu()  // activate or deactivate the menu interface according
to grip button input
    {
        userTransform = camObj.transform; // the position and rotation of the
        user (the VR camera) are updated
        if (rightHand == null) // make sure the right controller has been
        correctly identified
        {
            rightHand =
            this.transform.GetChild(0).GetChild(2).GetComponent<Hand>();
        }
        else
        {
            if
(rightHand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Grip))
            {
                menuIsActive = !menuIsActive;
                if (menuIsActive) // if the menu is becoming active, its
                transform is set so that it appears where the user is looking at
                and facing the user
                {
                    Menu.transform.position = userTransform.position +
                    userTransform.forward * 3.5f;
                    Menu.transform.rotation = userTransform.rotation;
                }
                pointer.NotVisible(); // tell the MenuPointer (see script)
                menuMan.ManageMenus(menuIsActive, false); // tell the MenuManager
                that its interface visibility has changed
            }
        }
    }
```

Figura 9: Función *ManageMenu* de *User*.

2.2 Robot virtual

En este apartado se explica el funcionamiento del robot virtual, que estaba ya implementado en la aplicación base. Sin embargo, este no contaba con la herramienta que posee el robot real, cuya implementación también se desarrolla en este apartado.

El funcionamiento del robot virtual se basa en que cada una de sus articulaciones es hija de la anterior, tal y como se muestra en la figura 10, de manera que la rotación de cada una se aplica a todas las posteriores. Los objetos *Link* son las piezas del robot y los objetos *Axis*, los ejes sobre los que puede rotar.

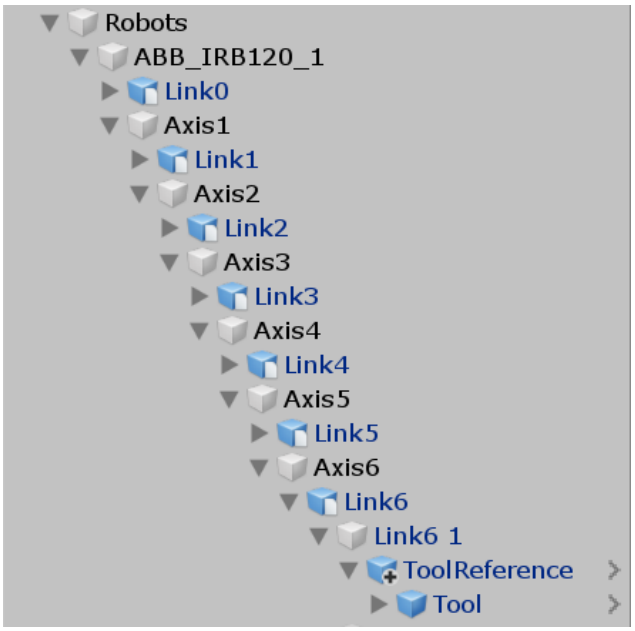

Figura 10: Jerarquía del objeto del robot virtual.

2.2.1 Código *IRB120*

Este código maneja el funcionamiento del robot virtual y ya estaba implementado, pero ha sido ligeramente modificado conforme a los requisitos de la nueva aplicación.

Todos los ejes se manejan en el código como variables *tAxis* y las rotaciones se realizan de forma fluida gracias al método *Quaternion.Lerp* empleado en la función *Update*, la cual no ha sido modificada.

```
    void Update() // Update is called once per frame
    {
        // linear interpolate to angle
        tAxis1.localRotation = Quaternion.Lerp(tAxis1.localRotation,
        Quaternion.Euler(0, -axisAngles[0], 0), 5 * Time.deltaTime);
        tAxis2.localRotation = Quaternion.Lerp(tAxis2.localRotation,
        Quaternion.Euler(0, 0, -axisAngles[1]), 5 * Time.deltaTime);
        tAxis3.localRotation = Quaternion.Lerp(tAxis3.localRotation,
        Quaternion.Euler(0, 0, -axisAngles[2]), 5 * Time.deltaTime);
        tAxis4.localRotation = Quaternion.Lerp(tAxis4.localRotation,
        Quaternion.Euler(-axisAngles[3], 0, 0), 5 * Time.deltaTime);
        tAxis5.localRotation = Quaternion.Lerp(tAxis5.localRotation,
        Quaternion.Euler(0, 0, -axisAngles[4]), 5 * Time.deltaTime);
        tAxis6.localRotation = Quaternion.Lerp(tAxis6.localRotation,
        Quaternion.Euler(-axisAngles[5], 0, 0), 5 * Time.deltaTime);
    }
}
```

Figura 11: *Update* de *IRB120*.

Los ángulos en cada eje estaban definidos como 6 variables independientes. Estas variables han sido sustituidas por un vector de 6 elementos para mayor comodidad en el manejo de la información.

Por otro lado, la pinza ha sido añadida como objeto hijo de la última articulación y se han añadido dos variables al código del robot que permiten controlarla:

- *tool*. Variable booleana que indica el estado de la herramienta: *true* si está activada, en este caso, abierta al tratarse de una pinza, y *false* si está desactivada o cerrada.
- *actTool*. Variable entera que indica si la herramienta está en reposo, en cuyo caso toma el valor 1, o realizando una instrucción (abriéndose o cerrándose), tomando el valor 0. También podría tomar otros valores que identificaran fases intermedias en caso de ser necesario para otras herramientas.

Estas variables residen en el código del robot a pesar de no ser usadas en él para que el control de la herramienta pueda hacerse de forma genérica, comunicándose con el robot independientemente del tipo de herramienta. Las modificaciones del código mencionadas se muestran en la figura 12:

```
    public float[] axisAngles = new float[6]; // the virtual robot's joint angles
    [FJD: replaced 6 variables with a size 6 array]

    // virtual tool indicators [FJD]
    public bool tool = false; // tool state: true = open gripper, false = closed
    gripper
    public int actTool = 0; // tool readiness indicator: 0 = not ready, 1 = ready
```

Figura 12: Modificaciones del código *IRB120*.

2.2.2 Objeto de la herramienta

Los modelos 3D de todas las piezas de la pinza, excepto de la placa interfaz, que es la pieza que conecta el conjunto con el robot, fueron proporcionados por separado por el director del proyecto. La placa interfaz fue modelada en SolidEdge cumpliendo únicamente con las

características esenciales para que el comportamiento de la pinza virtual no se viese afectado. Dichas características resultaron ser únicamente el grosor de la placa y la geometría de la unión. Posteriormente se ensamblaron virtualmente todas las piezas en SolidEdge para conformar la herramienta completa. En la figura 14 puede verse el conjunto completo.

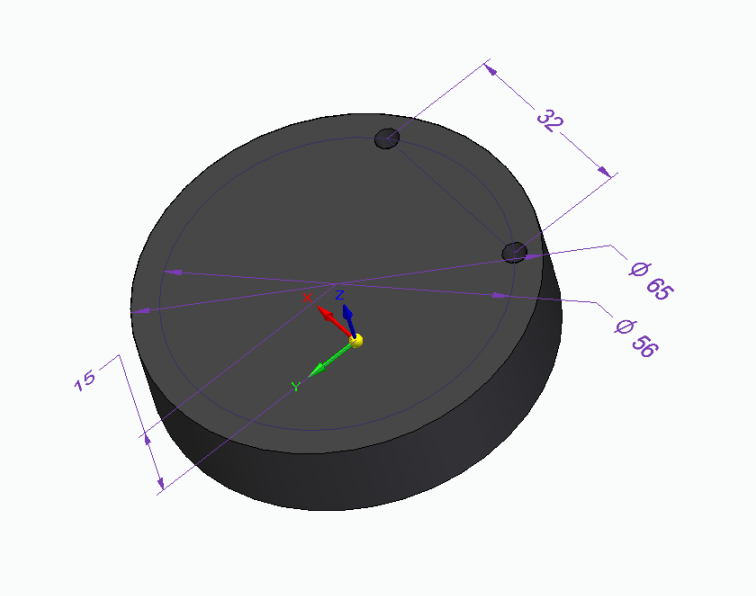

Figura 13: Modelo 3D sustitutivo de la placa interfaz en SolidEdge.

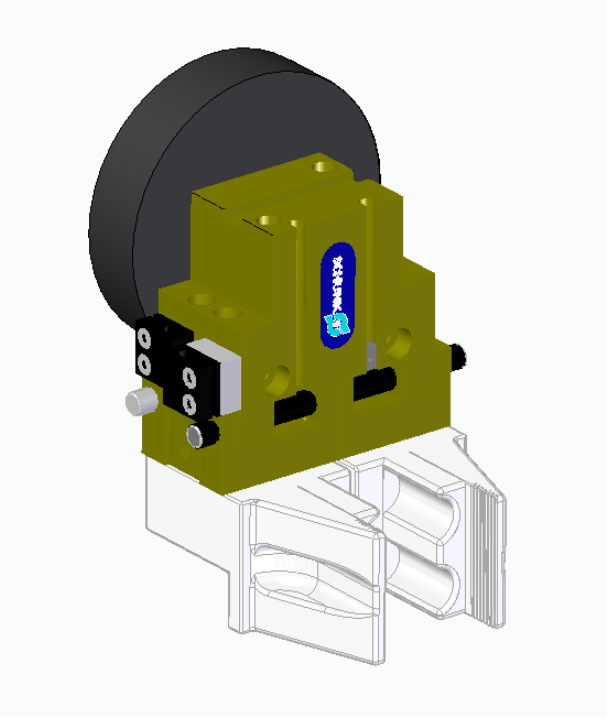

Figura 14: Conjunto de la herramienta completo en SolidEdge. De arriba abajo: placa interfaz, mecanismo Schunk y dedos de la pinza.

Tras procesar el archivo de ensamblaje de SolidEdge mediante Sketchup para obtener un modelo compatible con Unity, se importó la pinza en el programa y se procedió a colocarla cuidadosa y manualmente a falta de un método mejor en Unity. Pero, dado que Unity establece por defecto como punto de pivote de los objetos su centro de gravedad, era prácticamente imposible la correcta colocación de la pinza, ya que el punto de referencia que se necesitaba para la unión de la pinza al robot era el centro de la superficie de unión de la placa interfaz. Cambiando el ajuste de pivote de *Center* a *Pivot* se obtenía como pivote un punto arbitrario de la pieza principal del conjunto *Schunk*, probablemente debido al hecho de que se había utilizado dicha pieza como base para ensamblaje en SolidEdge.

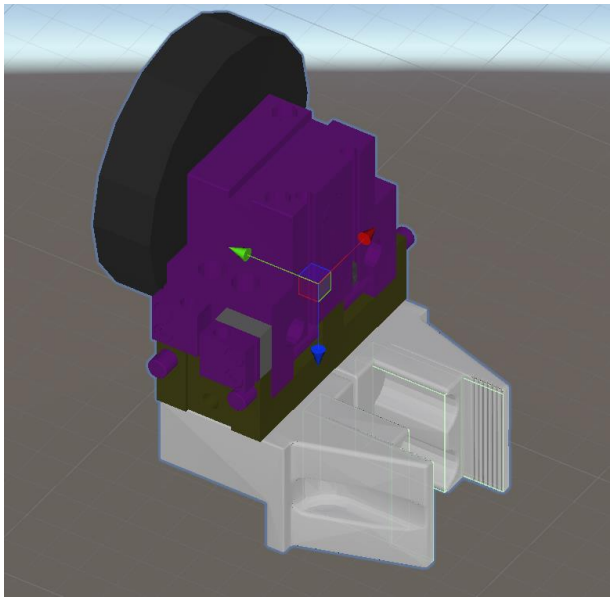

Figura 15: Punto de pivote de la pinza en Unity por defecto.

Para solucionar este problema, hubo que crear un objeto vacío, colocarlo en las coordenadas conocidas del punto requerido respecto al pivote ya obtenido y establecer la pinza como objeto hijo de ese objeto. Manteniendo el ajuste *Pivot* activo, el nuevo punto de pivote de la pinza es el del objeto padre, que, al ser un objeto vacío, tiene como pivote su propia posición, es decir, el punto que se necesitaba.

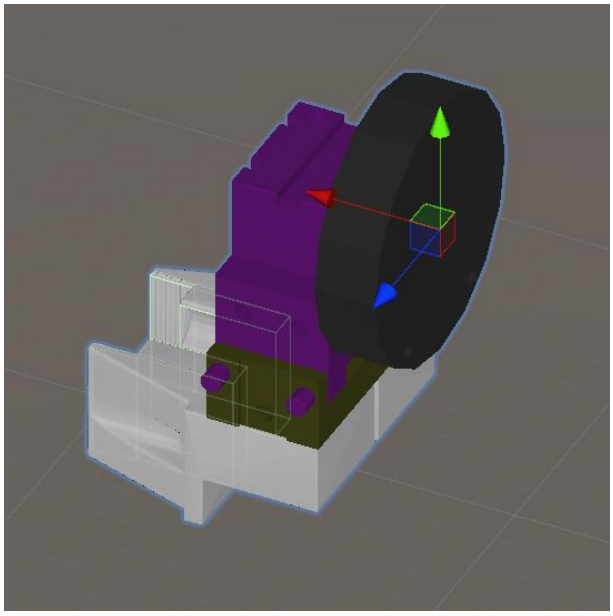

Figura 16: Punto de unión de la pinza con el robot en Unity.

Además, se estableció la pinza como objeto hijo de la sexta y última articulación del robot. Las partes del mecanismo Schunk que en la pinza real mueven los dedos se establecieron como objetos hijo de los dedos de la pinza, ya que en la pinza virtual son los dedos los que se mueven mediante código.

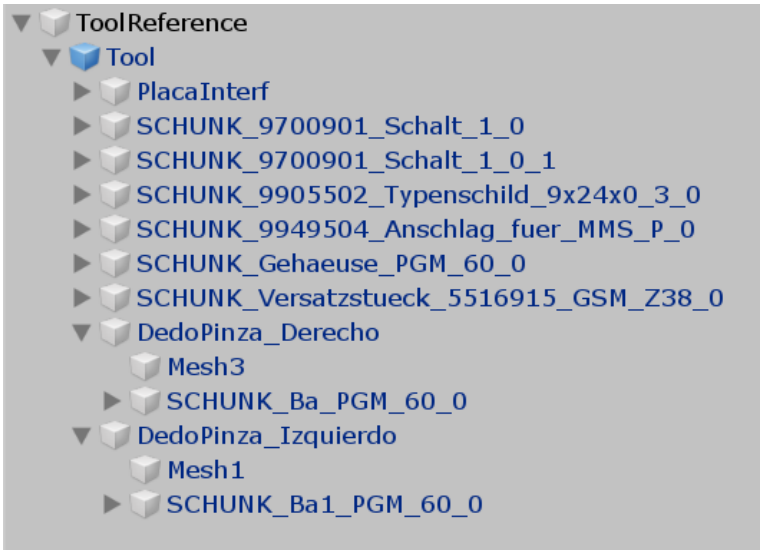

Figura 17: Jerarquía del objeto de la herramienta.

Por último, de la pinza sólo contienen *MeshColliders* con *Rigidbody* los dedos, es decir, el resto de la pinza es incorpórea y no interviene en la física de la simulación. Pero, además, las superficies interiores de los dedos cuentan con *BoxColliders Trigger* para detectar cuando un objeto esté siendo agarrado, ya que el agarre de objetos se va a simular por completo mediante código para evitar el azar inherente a la simulación física. Estos *Colliders* pueden verse en la figura 18, están delimitados por líneas verdes y sobresalen mínimamente respecto a los dedos de la pinza.

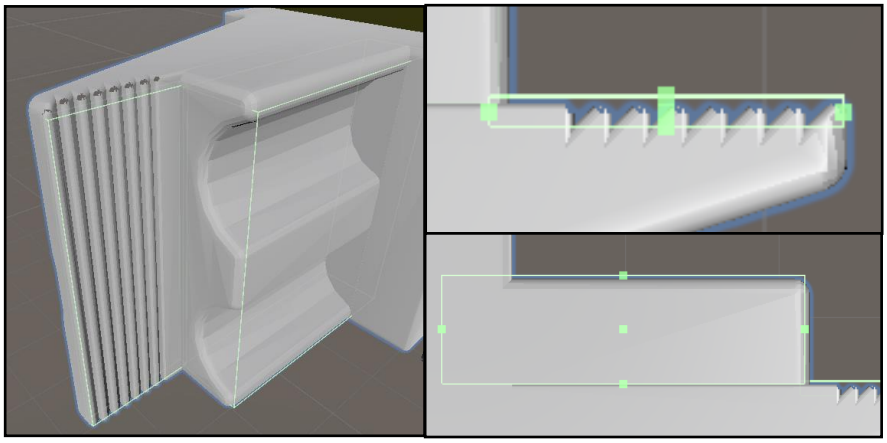

Figura 18: *BoxColliders Trigger* de los dedos de la pinza.

2.2.2.1 Código *ToolFinger*

El código *ToolFinger* es el *MonoBehaviour* asociado a los dedos de la pinza virtual y que se encarga de simular su comportamiento y el agarre de objetos.

Las variables que emplea son las siguientes:

- Como variables públicas:
  - *robot*. Robot virtual al que pertenece la pinza.
  - *x_open* y *x_closed*. Coordenadas en el eje X propio de la pinza en las que se sitúa el dedo en cuestión cuando la pinza está abierta y cerrada respectivamente. Sus valores han sido medidos manualmente en el entorno de Unity.
  - *speed*. Velocidad de movimiento de los dedos de la pinza. Valor determinado empíricamente.
- Como variables privadas:
  - *vel_open*. Velocidad de apertura de cada dedo de la pinza (tienen distinto signo).
  - *tmov*. Velocidad usada para mover los dedos de la pinza.
  - *rb*. Componente *Rigidbody* del dedo.
  - *stat_act* y *stat_ant*. Estados actual y anterior (variable auxiliar para la detección de flancos) requeridos para la pinza.
  - *Right* y *Left*. Indicadores booleanos del tipo de dedo.
  - *movingObject*. Objeto que ha sido agarrado por la pinza.
  - *rbant*. Estado anterior del indicador *actTool* del robot (variable auxiliar para la detección de flancos).
  - *already*. Indicador booleano de la situación en la que se le requiere a la pinza alcanzar el estado en el que se encuentra.

```
public IRB120 robot; // the virtual robot the gripper tool is attached to
public float x_open; // the position in local x axis in which the gripper
fingers are considered to be open
public float x_closed = 0; // the position in local x axis in which the
gripper fingers are considered to be closed
public const float speed = 1; // the gripper fingers' speed of movement

private float vel_open = 0; // the speed of movement for each finger
private float tmov; // the used speed for each finger
private Rigidbody rb; // the finger's Rigidbody component
private bool stat_act = false; // current tool or finger state
private bool stat_ant = false; // last tool or finger state
private bool Right = false; // boolean right finger identifier
private bool Left = false; // boolean left finger identifier
private GameObject movingObject = null; // grabbed object
private int rbant = 0; // last robot's tool readiness indicator state
private bool already = false; // indicates if the tool has been required to
reach the state in which it already is (true)
```

Figura 19: Variables de *ToolFinger*.

En la función *Start* se determina el tipo de dedo, izquierdo o derecho, y por tanto el signo de la velocidad de apertura. Dado que el eje X propio de la pinza crece en la dirección en la que se encuentra el dedo izquierdo, la posición x_*open* de este será mayor que x_*closed* y viceversa para el derecho. Además, la *vel_open* será positiva para el izquierdo y negativa para el derecho. De esta forma, el código identifica perfectamente el dedo al que ha sido asignado.

```
private void Start() // Start is called before the first frame update
{
    // since the local x axis grows in the direction of the left finger:
    if (x_open > x_closed)
    {
        Left = true; // left finger is identified
        vel_open = speed; // left finger opens with positive speed, closes
        with negative
    } else if (x_open < x_closed)
    {
        Right = true; // right finger is identified
        vel_open = -speed; // right finger opens with negative speed, closes
        with positive
    }
}
```

Figura 20: *Start* de *ToolFinger*.

En la función *Update* se siguen los pasos siguientes:

1. Se comprueba el estado en el que debería estar la herramienta según el valor del *tool* del *robot*.
    a. Si acaba de cambiar de abierta a cerrada, *tmov* toma *vel_open* como valor negativo, ya que el dedo ha de cerrarse.
    b. Si acaba de cambiar de cerrada a abierta, *tmov* toma el valor vel_open
        i. Si algún objeto estaba agarrado por la pinza, el objeto es liberado. Este proceso se explica con detalle a continuación.
2. Si el estado de la pinza no ha cambiado, pero sí lo ha hecho el indicador *actTool*, se activa el indicador *already*.
3. Por último, llama a las funciones que controlan el movimiento del dedo.

```
private void Update() // Update is called once per frame
    {
1       stat_act = robot.tool; // the finger state is updated with virtual tool
        state
        if (stat_act ^ stat_ant) // if the finger state has changed:
        {
a           if (!stat_act) // if the finger state has just changed to false
            (close)
            {
                tmov = -vel_open; // closing
            }
b           else // if the finger state has just changed to true (open)
            {
                tmov = vel_open; // opening
i               if (movingObject != null && movingObject.transform.parent !=
                null) // if an object had been grabbed:
                {
                    movingObject.GetComponent<Rigidbody>().isKinematic = false;
                    // make the object affected by external forces again
                    movingObject.transform.parent = null; // make the object's
                    Transform independent of the tool's
                }
            }
        }
2       else if (rbant != robot.actTool && robot.actTool == 0) // if the finger
        state has not changed, the robot's tool readiness indicator has changed
        from 1 to 0 (means the tool has been required to act in some way, see
        IRB120):
        {
            already = true; // the corresponding indicator is set to true
            Debug.Log("[TF] Tool already ready");
        }
        rbant = robot.actTool; // last tool readiness state is updated
        stat_ant = stat_act; // last tool state is updated
3       StopMoving(); // stops the finger if necessary
        MoveFinger(); // moves the finger
    }
```

Figura 21: *Update* de *ToolFinger*.

La función *OnTriggerEnter* es propia de Unity y se llama cuando un objeto entra dentro del *Collider Trigger* del dedo. Como el *Collider* está situado en la superficie interior de la pinza, significa que un objeto está siendo agarrado. Para simularlo, se establece a la pinza como objeto padre del objeto agarrado y se hace al objeto inmune a todo tipo de fuerzas mediante la opción *isKinematic* del componente *Rigidbody*. Por último, se indica que la pinza ha alcanzado un punto de reposo mediante el indicador *actTool.* Para liberar el objeto se revierten las instrucciones descritas.

```
private void OnTriggerEnter(Collider other) // gets called whenever an object
enters the finger trigger Colliders (placed on its inner surfaces)
{
        if (!stat_act) // if the gripper is closing:
        {
            Debug.Log("[TF] Object grabbed");
            tmov = 0; // stop moving the finger
            other.transform.SetParent(this.transform.parent); // make the grabbed
            object move and rotate with the tool
            movingObject = other.gameObject; // set movingObject to grabbed
            object
            movingObject.GetComponent<Rigidbody>().isKinematic = true; // make
            movingObject not be affected by external forces (like the gripper) to
            ensure it stays in place
            robot.actTool = 1; // tool is ready
        }
}
```

Figura 22: *OnTriggerEnter* de *ToolFinger*.

La función *StopMoving* detiene el movimiento del dedo cuando es necesario. Si ha alcanzado x_*closed* o x_*open* en la situación correspondiente o si ya estaba en esas posiciones y se le ha requerido alcanzarlas, situación identificada mediante el indicador *already*. Además de detener el movimiento, reinicia los indicadores y asegura que la posición del dedo sea la correcta. La función *MoveFinger* mueve el dedo a velocidad *tmov* en metros por segundo en su dirección X propia.

```
private void StopMoving() // stops the movement of the finger if it's reached
a stoping state
{
    if ((Left && transform.localPosition.x <= x_closed || // if the finger
        (left or right) has reached x_closed and...
        Right && transform.localPosition.x >= x_closed) &&
        (tmov == -vel_open || already && !stat_act)) // ...and it was closing
        or already there and closed:
    {
        tmov = 0; // stop moving
        robot.actTool = 1; // tool is ready
        transform.localPosition = new Vector3(x_closed,
        transform.localPosition.y, transform.localPosition.z); // finger is
        positioned in x_closed to ensure proper functioning
        Debug.Log("[TF] Closed tool");
        already = false; // already indicator is reset
    }
    else if ((Left && transform.localPosition.x >= x_open || // if the finger
        (left or right) has reached x_open and...
        Right && transform.localPosition.x <= x_open) &&
        (tmov == vel_open || already && stat_act)) // ...and it was opening
        or already there and openened:
    {
        tmov = 0; // stop moving
        robot.actTool = 1; // tool is ready
        transform.localPosition = new Vector3(x_open,
        transform.localPosition.y, transform.localPosition.z); // finger is
        positioned in x_open to ensure proper functioning
        Debug.Log("[TF] Opened tool");
        already = false; // already indicator is reset
    }
}
```

Figura 23: Función *StopMoving* de *ToolFinger*.

```
private void MoveFinger() // move the finger
{
    this.gameObject.transform.Translate(tmov * Vector3.right *
    Time.deltaTime, Space.Self); // teleports the finger each frame at a
    speed of tmov m/s in its positive x direction
}
```

Figura 24: Función *MoveFinger* de *ToolFinger*.

2.3 Interfaz

En este apartado se explica lo relativo a la interfaz de usuario. La interfaz es el objeto que contiene los menús que permiten al usuario acceder a las distintas funcionalidades de la aplicación y los programas de instrucciones para los robots.

2.3.1 Interacción

En Unity existen dos métodos principales de detección de eventos en los que un objeto apunta a otro, esos métodos son:

- Mediante *Graphic Raycaster*. Se crea una línea instantáneamente que es capaz de colisionar con elementos de interfaz y proporciona acceso directamente a los eventos propios de este tipo de interacción, como *OnPointerEnter* u *OnPointerExit* (cuando se empieza o se deja de señalar al elemento). Es el método más eficiente para interacciones con interfaz porque está diseñado para ello.
- Mediante *Physics Raycaster*. Se crea una línea instantáneamente que es capaz de colisionar con *Colliders*. Esas colisiones pueden filtrarse, detectarse y caracterizarse mediante código. De cara a su implementación para interacción con una interfaz, es un método poco eficiente porque requiere que todos los elementos contengan *Colliders* y porque no aprovecha lo que Unity proporciona ya hecho.

Sin embargo, para poder acceder a información como el punto exacto a donde se está apuntando, el primer método resulta extremadamente complicado al tratarse de realidad virtual, ya que requiere reprogramar parte de lo que ya está programado en Unity, que está pensado principalmente para interacciones con el puntero de un ratón. Probando la primera versión de la aplicación se determinó que era esencial que el usuario pudiese ver el lugar al que apuntaba, y tras diversos intentos fallidos de implementarlo mediante el método óptimo, se optó finalmente por el otro método por ser mucho más sencillo. El código desechado fue elaborado según una serie de vídeos didácticos y ha sido incluido en la carpeta del proyecto por si se siguiese desarrollando la aplicación en futuros trabajos [11].

2.3.2 Objeto de la interfaz

El objeto de la interfaz se llama *UserInterface* y engloba todo lo relativo a ella. Para crear el objeto de la interfaz y todos sus menús se han utilizado objetos de Unity destinados a la creación de este tipo de elementos. Los objetos utilizados son:

- Un objeto de tipo *Canvas*, que actúa como objeto base, siendo el objeto padre de los demás y sirviendo como plano en el espacio en el que se colocan el resto de los elementos.
- Objetos de tipo *Button*, que son botones que permiten activar diversos procesos.
- Objetos de tipo *Toggle*, que son casillas seleccionables que activan y desactivan una variable booleana interna a la que se puede acceder y vincular diversos procesos.
- Objetos de tipo *Dropdown*, que son menús de opciones desplegables a cuyo índice de posición de la opción seleccionada se pueden vincular diversos procesos.

La vinculación de los elementos con los códigos de la interfaz se configura en el *Inspector*, creando uno o más eventos que al activarse pueden llamar a las funciones públicas de los códigos del objeto que se dé como argumento o incluso cambiar parámetros intrínsecos al objeto, como por ejemplo su visibilidad. En el caso de los botones los eventos son de tipo *OnClick*, por lo que se activan cuando son pulsados, y en el caso de los *Toggles* y *Dropdowns* los eventos son de tipo *OnValueChanged*, por lo que se activan cuando se cambia el estado

del elemento. *OnValueChanged* permite además entregarle a la función a la que llame el parámetro del elemento en cuestión si esta función requiere como argumento único una variable del mismo tipo, es decir, para el *Toggle* permite entregar el booleano asociado y para el *Dropdown*, el entero correspondiente a la posición de la opción escogida.

En la figura 25 se muestra como ejemplo de la configuración de los eventos *OnClick* la del botón *GoToButton* del menú de adición de acciones *AddMenu*. Los eventos son los siguientes:

1. Activa el menú *GoToMenu*.
2. Desactiva el menú *AddMenu* (menú en el que se encuentra).
3. Llama a la función *creatingGoTo* del código *ProgramManager* para activar la funcionalidad correspondiente.
4. Desactiva el objeto *ProgramDisplay*, que muestra el programa actual.
5. Llama a la función *UpdateActual* del código *MenuManager* dándole como argumento el objeto *GoToMenu*.

En la figura 26 se muestra como ejemplo de la configuración de los eventos *OnValueChanged* la del *Dropdown VelocityDropdown* del menú de creación de acciones de tipo *GoTo GoToMenu*. En el evento se llama a la función *UpdateVelocity* del código *ProgramManager* pasando como argumento el entero correspondiente a la posición de la opción seleccionada, al haber 3 opciones: 0, 1 o 2.

Los códigos mencionados se explican más adelante en este capítulo.

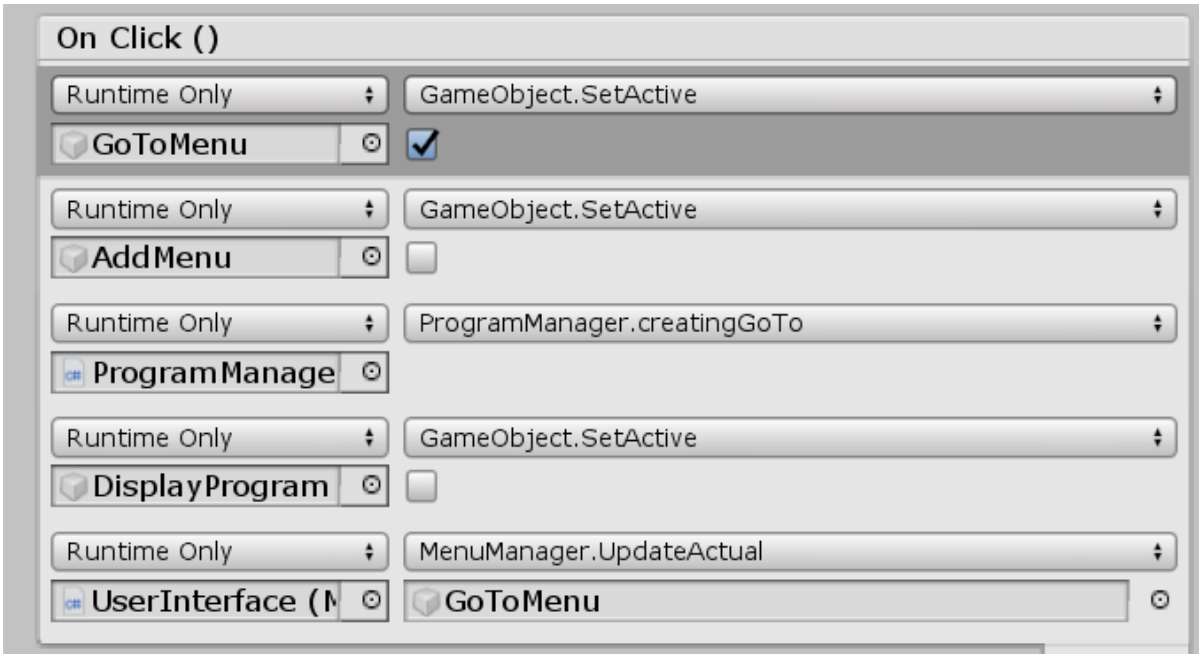

Figura 25: Ejemplo de configuración de eventos *OnClick* del botón *GoToButton*.

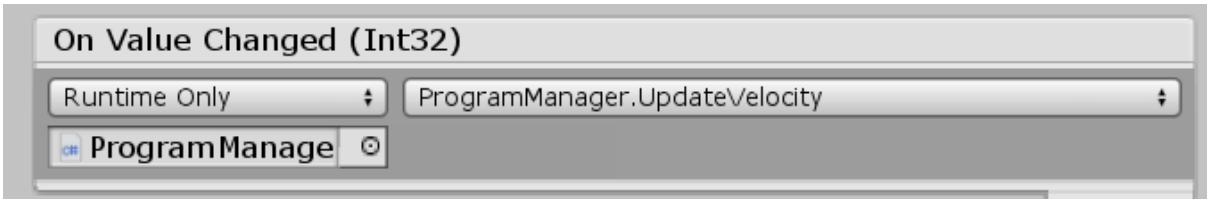

Figura 26: Ejemplo de configuración de eventos *OnValueChanged* del *Dropdown VelocityDropdown*.

Por defecto los elementos de interfaz de Unity son mucho más grandes de lo que se necesita en este proyecto, por lo que al darles un tamaño adecuado quedan con una calidad visual considerablemente baja. Para que tanto la imagen como el texto de los elementos pueda verse con claridad, se ha optado por incrementar los tamaños de ambos y luego establecer la escala de los elementos a un valor que haga que tengan el tamaño requerido. De esta forma no se aprecian los píxeles en los bordes ni el texto de los elementos. Por otro lado, para simplificar el código y a la hora de activar la interfaz poder limitarse a darle el valor de rotación del usuario, se comprobó y dio a la interfaz una rotación sobre su eje Y propio de 90 grados.

Dentro del objeto *UserInterface*, todos los elementos visibles de la interfaz se encuentran agrupados como hijos de un objeto llamado *Menus*, que es el objeto que se activa y desactiva, mientras que lo referente a los programas se encuentra agrupado en otro objeto llamado *Programs*, que es invisible, pero se mantiene siempre activo para que sus códigos puedan mantenerse en ejecución. La jerarquía de la interfaz se muestra en la figura 27. Cada menú está formado por un objeto padre invisible que contiene diversas combinaciones de los elementos de interfaz mencionados. En la figura 28 se muestra como ejemplo de jerarquía la del menú de creación de acciones de tipo *Wait*. La transición de un menú a otro se realiza mediante eventos *OnClick* de botones que activan el siguiente menú y desactivan el que les contiene.

Figura 27: Jerarquía del *UserInterface*.

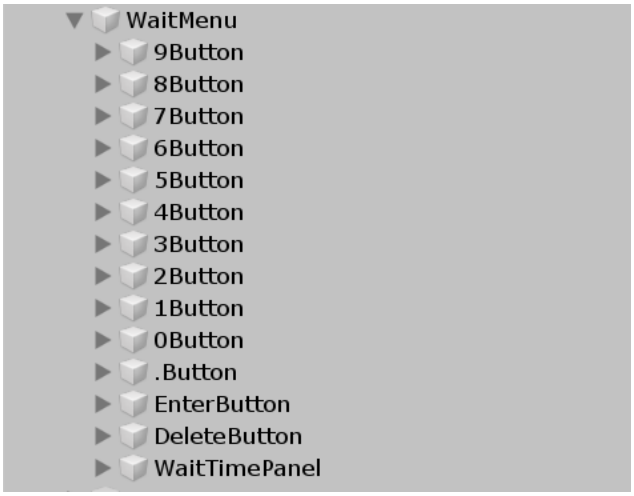

Figura 28: Jerarquía del menú *WaitMenu*.

La interfaz cuenta con diversos menús de los cuales no todos tienen su funcionalidad implementada. Los menús más importantes son los que se corresponden a la creación de acciones. Algunos de estos menús se muestran en las siguientes figuras, las ligeramente apreciables líneas verdes delimitan los *Colliders* de cada elemento:

Figura 29: Aspecto del menú de creación de acciones de tipo *Wait WaitMenu*.

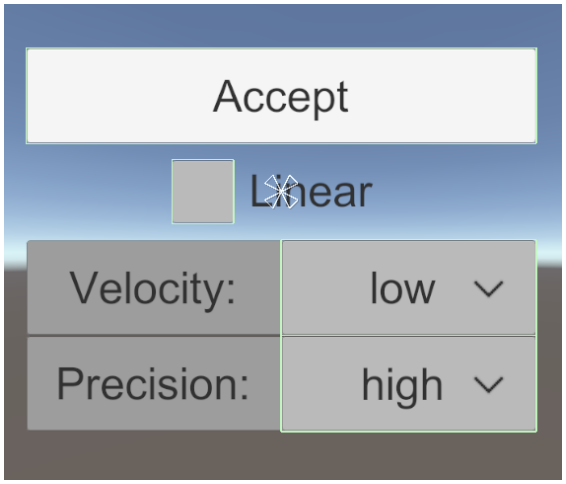

Figura 30: Aspecto del menú de creación de acciones de tipo *GoTo GoToMenu*. En orden de arriba abajo: un *Button*, un *Toggle* y dos *Dropdowns*.

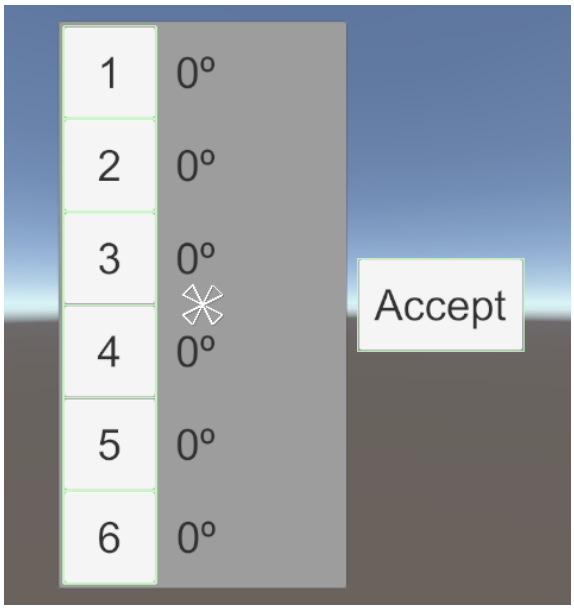

Figura 31: Aspecto del menú de creación de acciones de tipo *MoveAxis MoveAxisMenu*.

2.3.3 Código *MenuPointer*

Este código va asociado a un objeto hijo del mando con el que se controla la interfaz, en este caso el derecho, y permite generar las interacciones necesarias para ello. No va asociado directamente al mando para mayor claridad a la hora de visualizar sus numerosos componentes.

Las variables que emplea son:

- *dot*. Única variable pública, se refiere al objeto utilizado para indicar el lugar del menú al que el usuario apunta con el mando. Se ha escogido una esfera negra de pequeñas dimensiones.
- *curr* y *last*. Objetos actual y anterior a los que apunta el usuario.

En la función *Update* se realizan los siguientes pasos:

1. En primer lugar, se genera el *Physics Raycast*, utilizando como máscara la necesaria para que sólo se detecten colisiones con la interfaz, en este caso la interfaz está en la capa número 5.
2. Si se detecta algo:
    a. Se asigna el objeto detectado a *curr*.
    b. Si previamente no se apuntaba a nada, se ejecuta el evento *OnPointerEnter* del objeto apuntado actualmente.
    c. Si previamente se apuntaba a un objeto distinto al actual, se ejecuta además el evento *OnPointerExit* del objeto anterior.
    d. Si *dot* no está activo, se activa para que sea visible.
    e. Se asigna la posición de la colisión o *RaycastHit* al objeto *dot* para que el usuario visualice claramente a dónde está apuntando.
3. Si no se apunta a nada:
    a. Se asigna el objeto *null* a *curr*.
    b. Si previamente se apuntaba a algo, se ejecuta el evento *OnPointerExit* de ese objeto.
    c. Si *dot* estaba activo, se desactiva para que no sea visible.
4. Por último, se actualiza *last* para poder repetir el proceso.

```
void Update() // Update is called once per frame
    {
        RaycastHit hit;
1       Physics.Raycast(transform.position, transform.forward, out hit, 100, 1 <<
        5); // a Ray is cast from the object's position in it's forward
        direction, info is stored in hit and 5 is the UI's layer
2       if (hit.collider != null) // if something is being pointed
        {
a           curr = hit.collider.gameObject; // update current object
b           if (last == null) // if nothing was being pointed just before, only
            execute OnPointerEnter of current object
            {
                ExecuteEvents.Execute(curr, new
                 PointerEventData(EventSystem.current),
                 ExecuteEvents.pointerEnterHandler);
            }
c           else if (curr != last) // if something different was being pointed
            just before, also execute OnPointerExit of last object
            {
                ExecuteEvents.Execute(last, new
                 PointerEventData(EventSystem.current),
                 ExecuteEvents.pointerExitHandler);
                ExecuteEvents.Execute(curr, new
                 PointerEventData(EventSystem.current),
                 ExecuteEvents.pointerEnterHandler);
            }
d           if (!dot.activeSelf) // if it's the first consecutive time the object
            is being pointed, make dot visible
            {
                dot.SetActive(true);
            }
e           dot.transform.position = this.transform.position + hit.distance *
            this.transform.forward; // dot is always where the Ray collides with
            the object
        }
3       else // if nothing's being pointed
        {
a           curr = null;
b           if (curr != last) // if something was being pointed just before,
            execute OnPointerExit of last object
            {
                ExecuteEvents.Execute(last, new
                 PointerEventData(EventSystem.current),
                 ExecuteEvents.pointerExitHandler);
            }
c           if (dot.activeSelf) // if dot was visible, make dot invisible
            {
                dot.SetActive(false);
            }
        }
4       last = curr; // update last object
    }
```

Figura 32: *Update* de *MenuPointer*.

La función *NotVisible* ejecuta el evento *OnPointerExit* de *curr* y le asigna el objeto *null* en caso de que *curr* sea otro objeto. Esta función es llamada desde el código *User* cuando el usuario desactiva el menú y su función es impedir que un elemento de la interfaz se quede en el estado "apuntado" mientras esta está desactivada. Los procesos mencionados a los que se ha denominado eventos se explican en el siguiente apartado.

```
public void NotVisible()
{
    if (curr != null)
    {
        ExecuteEvents.Execute(curr, new
         PointerEventData(EventSystem.current),
         ExecuteEvents.pointerExitHandler);
        curr = null;
    }
}
```

Figura 33: Función *NotVisible* de *MenuPointer*.

2.3.4 Código *UI_Element*

Este código va a asociado a todos los elementos de interfaz y permite manejar las interacciones del usuario con ellos. Además de ser *MonoBehaviour*, este código es de tipo *IPointerEnterHandler*, *IPointerExitHandler*, *IPointerDownHandler* e *IPointerUpHandler*, lo que le da permiso para controlar los eventos de interfaz de entrada y salida del puntero y pulsado y soltado respectivamente. Los eventos pueden entenderse como funciones que se llaman automáticamente cuando ocurre algo. El código no es también declarado como *IPointerClickHandler* porque los eventos de tipo *OnClick* ya están definidos en el *Inspector*.

Las variables que emplea son las siguientes:

- Como variables públicas:
  - *normalColor*, *highlightColor* y *clickedColor*. Son los colores que toma el elemento en función de sus tres posibles estados: normal, apuntado y activado respectivamente.
  - *canToggle*. Booleano que determina la capacidad del elemento, en caso de ser un botón, de permanecer activado tras ser presionado una vez y requerir una segunda vez para ser desactivado.
  - *pointed* y *clicked*. Indicadores booleanos de los estados apuntado y activado respectivamente.
- Como variables privadas:
  - Los componentes necesarios del elemento:
    - *image*. El recuadro que contiene el texto y puede tomar distintos colores.
    - *coll*. El *Collider* necesario para la interacción.
    - *rTrans*. El componente que contiene la información sobre el tamaño y forma del elemento.
  - *menu*. Componente *MenuManager* del objeto interfaz que contiene al elemento.
  - *hand*. El componente *Hand* del mando con en el que se interactúa.

```
public Color32 normalColor = Color.grey;
public Color32 highlightColor = Color.white;
public Color32 clickedColor = Color.cyan;
public bool canToggle = false; // allows the button to stay pressed when
clicked until next click
public bool pointed = false; // indicates whether the button is being pointed
by the user
public bool clicked = false; // indicates whether the button is pressed

private Image image = null; // the visible image of the button
private BoxCollider coll; // a Collider component for the interaction with
buttons to be done through PhysicsRaycast
private RectTransform rTrans; // rectangle transform of the button
private MenuManager menu; // MenuManager attached to this object's root
parent (the interface object)
private Hand hand; // the controller that interacts with the menu
```

Figura 34: Variables de *UI_Element*.

En la función *Awake* se inicializan las variables que lo requieren, obteniéndose el objeto de la interfaz como el objeto padre primigenio que exista en la jerarquía mediante el método *root*.

```
private void Awake() // called before first frame update
{
    image = this.gameObject.GetComponent<Image>();
    menu = this.transform.root.GetComponent<MenuManager>();
    hand = menu.rightHand;
    if (image != null)
    {
        image.color = normalColor;
    }
}
```

Figura 35: *Awake* de *UI_Element*.

En la función *Update* se ejecutan los eventos de pulsado o soltado en función de los indicadores *pointed* y *clicked* y de si el usuario pulsa el botón *Trigger* derecho (botón con forma de gatillo del mando). Para que el elemento pueda pulsarse tiene que estar apuntado y no pulsado y para que pueda soltarse tiene que estar apuntado y pulsado.

```
private void Update() // called every frame update
{
    if (pointed &&
    hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Trigger))
    // if element is pointed and trigger button is pressed, a click is
    processed
    {
        ExecutePress();
    }
    if (clicked &&
    (hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Trigger)))
    // if clicked and trigger button is released, a release is processed
    {
        ExecuteRelease();
    }
}
```

Figura 36: *Update* de *UI_Element*.

En las funciones *OnPointerEnter* y *OnPointerExit*, se actualizan el estado de pointed y, si el elemento no está pulsado, el color, ya que, si el elemento está pulsado, ha de mantener el color que lo indica. En la función *OnPointerDown* si el elemento no está pulsado, se actualizan el color y el estado de *clicked* coherentemente, y si sí lo está, significa que *canToggle* está activado y, por lo tanto, se desactiva *clicked* y se actualiza el color al color que indica que está siendo apuntado, ya que se trata de la segunda pulsación y no puede no estarlo. En *OnPointerUp* se actualizan el color y el estado de *clicked*, pero sólo si *canToggle* está desactivado, ya que, si no lo está, la única forma de desactivar el elemento es pulsándolo por segunda vez.

```
public void OnPointerEnter(PointerEventData eventData) // called when the
button gets pointed
{
    if (image != null && !clicked)
    {
        image.color = highlightColor;
    }
    pointed = true;
}
public void OnPointerExit(PointerEventData eventData) // called when the
button stops being pointed
{
    if (image != null && !clicked)
    {
        image.color = normalColor;
    }
    pointed = false;
}
public void OnPointerDown(PointerEventData eventData) // called when the
button gets clicked
{
    if (!clicked)
    {
        if (image != null)
        {
            image.color = clickedColor;
        }
        clicked = true;
    }
    else // if the button can be toggled, its release is processed when
    clicked again
    {
        if (image != null)
        {
            image.color = highlightColor;
        }
        clicked = false;
    }
}
public void OnPointerUp(PointerEventData eventData) // called when the button
gets unclicked
{
    if (!canToggle) // if the button can't be toggled, its release is
    processed when it gets unclicked
    {
        if (image != null)
        {
            if (pointed)
            {
                image.color = highlightColor;
            }
            else
            {
                image.color = normalColor;
            }
        }
        clicked = false;
    }
}
```

Figura 37: Código de los eventos de interfaz de *UI_Element*.

La función *setCollider* asigna al tamaño de *coll* el mismo valor que el del tamaño del elemento mediante el método *sizeDelta* de *rTrans*, siempre y cuando el elemento no sea de tipo *Toggle*, en cuyo caso el tamaño se determina en el *Inspector*, ya que sólo hay dos instancias, ni de tipo *Item*, que se corresponde con una de las opciones de un *Dropdown*, en cuyo caso se establece un tamaño que ha sido medido en el *Inspector*, pero que es necesario establecer mediante código porque las opciones de los *Dropdown* se generan cuando la aplicación está siendo ejecutada. Mediante las funciones *OnEnable* y *OnValidate*, la función *setCollider* es llamada cada que vez que el objeto es reactivado o tras ser modificado en el *Inspector*.

```
private void OnEnable() // called when the button's gameobject state changes
from inactive to active (visible)
{
    setCollider();
}

private void OnValidate() // called after the button is modified in Unity
{
    setCollider();
}

private void setCollider() // sets the right size for the button's collider
{
    rTrans = GetComponent<RectTransform>();
    coll = GetComponent<BoxCollider>();
    if (coll == null)
    {
        coll = gameObject.AddComponent<BoxCollider>();
    }
    if (!this.gameObject.name.Contains("Toggle"))
    {
        if (this.gameObject.name.Contains("Item"))
        {
            coll.size = new Vector3(320, 80, 0);
        }
        else
        {
            coll.size = rTrans.sizeDelta;
        }
    }
}
```

Figura 38: Funciones de configuración de *Collider* de *UI_Element.*

La función *ExecutePress* ejecuta los eventos *OnPointerDown* y *OnPointerClick* a través del *EventSystem* de la aplicación, objeto de Unity que maneja la detección de eventos. El evento *OnPointerClick* hace referencia a los eventos *OnClick* creados en el *Inspector*. La función *ExecuteRelease* ejecuta el evento *OnPointerUp*.

```
public void ExecutePress()
{
    ExecuteEvents.Execute(this.transform.gameObject, new
    PointerEventData(EventSystem.current),
    ExecuteEvents.pointerClickHandler); // calls the default OnPointerClick
    handler for the events created in the Inspector window to get executed
    ExecuteEvents.Execute(this.transform.gameObject, new
    PointerEventData(EventSystem.current), ExecuteEvents.pointerDownHandler);
    // calls the custom OnPointerDown handler
}

private void ExecuteRelease()
{
    ExecuteEvents.Execute(this.transform.gameObject, new
    PointerEventData(EventSystem.current), ExecuteEvents.pointerUpHandler);
    // calls the custom OnPointerUp handler
}
```

Figura 39: Funciones de ejecución de eventos de interfaz de *UI_Element*.

2.3.5 Código *MenuManager*

Este código va asociado al objeto de la interfaz y se encarga de manejar todo lo referente a los menús que contiene. Está principalmente constituido por varias funciones públicas a las que se accede desde los eventos de la interfaz.

Las variables que emplea son:

- Como variables públicas:
  - *prgMan*, *rightHand* y *Menus*. Código *ProgramManager*, *Hand* que controla la interfaz y el objeto que alberga los elementos visibles de la interfaz respectivamente.
  - *timeObject*. Objeto que muestra la fecha, la hora y el tiempo de uso de la aplicación.
  - *minHeight*. Altura mínima a la que puede encontrarse la interfaz para que ningún menú atraviese el suelo del entorno de la aplicación. Su valor ha sido determinado visualmente y en base al menú más restrictivo, el de mayor altura.
- Como variables privadas:
  - *menuChildren* y *numChildren*. La lista de objetos que son hijos del objeto *Menus* y su número de elementos respectivamente.
  - *actualMenu* y *lastMenu*. Menú activo actual y menú activo anterior respectivamente.
  - *heritLine*. Lista de enteros en la que se almacena la posición dentro de *menuChildren* del menú anterior correspondiente a cada menú.

```
    public ProgramManager prgMan; // the menu's ProgramManager
    public Hand rightHand;
    public GameObject Menus; // interface object
    public GameObject timeObject; // time-displaying object
    public float minHeight; // minimum height the menu can stay completely
    visible at

    private List<GameObject> menuChildren = new List<GameObject>(); // the
    children of the interface object are the different menus
    private int numChildren; // number of menus
    private GameObject actualMenu; // current active menu
    private GameObject lastMenu; // current active menu's previous menu
    private List<int> heritLine = new List<int>(); // list in which the position
    inside menuChildren of the corresponding previous menu to each menu is stored
    // each of its positions refer to the positions of menuChildren
```

Figura 40: Variables de *MenuManager*.

En la función *Awake* se inicializan las variables que lo requieren. En *heritLine* se introduce un -1 por cada objeto hijo de *Menus*. Se inicializa *actualMenu* asignándole el hijo de posición 0 de *Menus*, que es el primer menú.

```
    private void Awake()
    {
        numChildren = Menus.transform.childCount;
        for (int i = 0; i < numChildren; i++) // initialisation for menuChildren
        and heritLine
        {
            menuChildren.Add(Menus.transform.GetChild(i).gameObject);
            heritLine[i] = -1;
        }
        actualMenu = menuChildren[0];
    }
```

Figura 41: *Awake* de *MenuManager*.

En la función *Update* se llama a la función *GoBack* si se pulsa el botón *ApplicationMenu* del mando (botón "B" en Oculus Touch) y si no está activado el indicador *angleChange* del *ProgramManager*, ya que significa que el botón está siendo usado para otra cosa. Además, se llama a las funciones *KeepVisibleHeight* y *PreventZRot*.

```
    private void Update() // Update is called once per frame
    {
        if
(rightHand.controller.GetPressDown(SteamVR_Controller.ButtonMask.ApplicationMenu))
// if button ApplicationMenu (B in Oculus Touch) is pressed, the user goes back
in the menus
        {
            if (!prgMan.angleChange) // the same button is used for another
            functionality (see ProgramManager)
            {
                GoBack();
            }
        }
        KeepVisibleHeight();
        PreventZRot();
    }
```

Figura 42: *Update* de *MenuManager*.

La función *ManageMenus* activa o desactiva *Menus* según el argumento booleano *m* y, si lo activa y el argumento booleano *reset* está activado, activa además el primer menú y desactiva los demás. El parámetro *reset* se utiliza cuando se usa una funcionalidad de la interfaz tras la cual no tiene sentido mantenerla activa ni en el mismo menú, como teletransportar al usuario a su posición inicial (función *Go0* de *User*) o en la función *PlaceTime*, la cual se explica en este apartado.

```
public void ManageMenus(bool m, bool reset) // sets the interface visibility
to boolean m
{
    if (reset & m)  // if boolean reset is true, resets the menu visibility
    order
    {
        menuChildren[0].SetActive(m); // the first menu is activated to be
        visible when the interface visibility is re-activated
        for (int i = 1; i < numChildren; i++)
        {
            menuChildren[i].SetActive(!m); // deactivates the rest
        }
    }
    Menus.SetActive(m);
}
```

Figura 43: Función *ManageMenus* de *MenuManager*.

La función *UpdateActual* es llamada desde cada botón que lleva de un menú a otro y dando como argumento *newActual* el menú al que lleva. La función actualiza entonces los objetos *actualMenu* y *lastMenu* y finalmente almacena la posición en *menuChildren* del nuevo *lastMenu* en la posición de *heritLine* correspondiente a la posición del nuevo *actualMenu* en *menuChildren*, pero sólo si el valor de *heritLine* en esa posición es negativo, para que la operación se realice sólo una vez. De esta manera, cada vez que se produce una transición entre menús, se guarda la información de entre qué menús se ha producido, obteniéndose en *heritLine* una lista que permite obtener el *lastMenu* de cualquier menú.

```
public void UpdateActual(GameObject menu) // updates the actualMenu and
lastMenu, called when the user goes from one menu to another
{
    lastMenu = actualMenu;
    actualMenu = menu;
    int i;
    i = actualMenu.transform.GetSiblingIndex();
    if (heritLine[i] < 0) // the information is stored in heritLine for later
    use, each new actualMenu gets its corresponding previous (lastMenu)
    stored
    { // the information is only stored once thanks to heritLine having been
    initialised with -1 in every position
        heritLine[i] = lastMenu.transform.GetSiblingIndex();
    }
}
```

Figura 44: Función *UpdateActual* de *MenuManager*.

La función *FindPrevLast* devuelve el *lastMenu* correspondiente al *actualMenu* que se le dé como argumento según la información almacenada en *heritLine*.

```
    private void FindPrevLast(GameObject newActual) // find the current menu's
    previous menu based on information previously stored
    {
        lastMenu =
(menuChildren[heritLine[newActual.transform.GetSiblingIndex()]]);
    }
```

Figura 45: Función *FindPrevLast* de *MenuManager*.

La función *GoBack* desactiva *actualMenu*, activa *lastMenu* y actualiza los objetos a los que se refieren, obteniendo el nuevo *lastMenu* mediante *FindPrevLast*. Además, llama a la función *notCreatingGoTo* del *ProgramManager*, la cual se explica más adelante en este capítulo.

```
    public void GoBack() // deactivates the current active menu and activates its
    corresponding previous menu
    {
        if (actualMenu.transform.GetSiblingIndex() != 0 && Menus.activeSelf)
        // if the current menu is not the first one and the interface is visible:
        {
            actualMenu.SetActive(false);
            lastMenu.SetActive(true);
            actualMenu = lastMenu;
            FindPrevLast(actualMenu); // the new previous menu is set
            prgMan.notCreatingGoTo(); // tell the ProgramManager that the user is
            not creating a GoTo Action in case he/she was
        }
    }
```

Figura 46: Función *GoBack* de *MenuManager*.

La función *PlaceTime* se llama desde la interfaz y crea y coloca el objeto *TimeDisplay* donde ésta se encuentre, tras buscar y borrar cualquier otra instancia del objeto si la hay. Además, desactiva el menú mediante *ManageMenus* con el parámetro *reset* activo.

```
    public void PlaceTime() // instantiates the time-displaying object
    {
        GameObject find = GameObject.FindWithTag("TimeDisplay"); // if another
        instance of the object is found somewhere else, it gets deleted
        if (find != null)
        {
            Destroy(find);
        }
        Instantiate(timeObject, transform.position, transform.rotation); // the
        objects gets created where the menu is currently at and with its rotation
        ManageMenus(false, true); // deactivate the menu interface visibility and
        reset the menu visibility order
    }
```

Figura 47: Función *PlaceTime* de *MenuManager*.

La función *ExitApplication* se llama desde la interfaz y permitirá cerrar la aplicación cuando ésta esté terminada y haya sido exportada de Unity como un archivo ejecutable.

```
    public void ExitApplication() // shuts the application
    {
        Application.Quit();
    }
```

Figura 48: Función *ExitApplication* de *MenuManager*.

La función *KeepVisibleHeight* impide que la interfaz se coloque por debajo de la altura mínima que asegura que ningún menú atraviese el suelo del entorno. La función *PreventZRot* impide que la interfaz pueda rotar en torno a su eje Z propio, es decir, al eje perpendicular a su superficie, ya que esta rotación no aporta nada al usuario.

```
    private void KeepVisibleHeight() // keep the menu at a visible height
    {
        if (transform.position.y < minHeight) // if the menu is placed below the
        minimum height, gets instantly placed at the minimum height
        {
            this.gameObject.transform.position =
new Vector3(transform.position.x, minHeight, transform.position.z);
        }
    }
    private void PreventZRot() // prevent the menu from rotating perpendicularly
    with respect to the user's vision
    {
        if (transform.localEulerAngles.z != 0) // if the menu is placed with a
        certain value of the aforementioned rotation, it gets set to 0
        {
            transform.localEulerAngles =
new Vector3(transform.localEulerAngles.x, transform.localEulerAngles.y, 0);
        }
    }
```

Figura 49: Funciones *KeepVisibleHeight* y *PreventZRot* de *MenuManager*.

2.3.6 Código *TimeTextDisp*

Código encargado del funcionamiento del objeto *TimeDisplay*. La función *GetDate* obtiene y muestra la fecha y la hora del sistema en el que se esté ejecutando la aplicación a través del *Text dateTime*, que se refiere a un componente *Text* del objeto. La función *GetTime* calcula y muestra el tiempo que la aplicación lleva en ejecución en horas, minutos y segundos a través del *Text timeRunning*, que también se corresponde con un *Text* de *TimeDisplay*.

```
    private void GetDate() // sets dateTime string component to current date and
    time
    {
        DateTime localDate = DateTime.Now;
        CultureInfo culture = new CultureInfo("es-ES");
        dateTime.text = localDate.ToString(culture);
    }

    private void GetTime() // sets timeRunning string component to the time
    passed since the application started running
    {
        float timePassed = Time.time;
        int seconds = (int)(timePassed % 60);
        int minutes = (int)(timePassed / 60) % 60;
        int hours = (int)(timePassed / 3600) % 60;
        timeRunning.text = "Time using application:" + System.Environment.NewLine
        + string.Format("{0:00}:{1:00}:{2:00}", hours, minutes, seconds);
    }
```

Figura 50: Funciones *GetDate* y *GetTime* de *TimeTextDisp*.

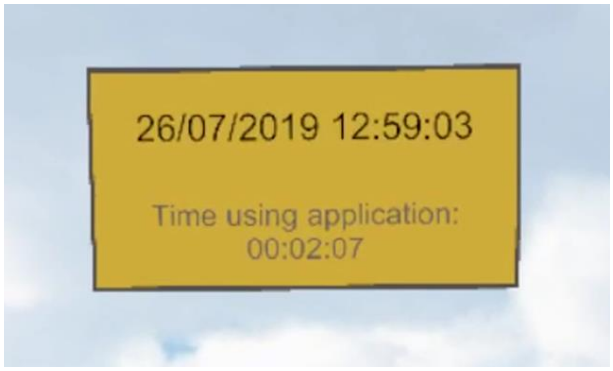

Figura 51: Objeto resultado de *TimeTextDisp*.

2.4 Programación

La programación del robot se basa en el uso de clases y de sus propiedades de herencia y polimorfismo, las cuales permiten llamar a una función de una clase que se ejecute de distinta forma según el tipo de clase hija del que se trate, pero sin necesidad de comprobarlo.

2.4.1 Código *Action*

En este código se definen la clase *Action*, que se corresponde con el concepto abstracto de una acción genérica realizable por un robot, y sus clases hijas, correspondientes cada una con un tipo concreto de acción.

2.4.1.1 Clase abstracta *Action*

La clase *Action* se define como clase abstracta, ya que no tiene sentido que se pueda implementar una acción sin concretar sus características. Por tanto, en ella únicamente se declaran las funciones pertenecientes a la clase. Estas funciones son:

- La función *Print*, que devuelve una cadena de caracteres con la información relevante sobre la acción.
- La función *Execute r*ecibe como argumentos el código encargado de las comunicaciones, el robot al que se aplica la acción y *flanc*, una variable que indica con el valor 1 que la acción se está ejecutando por primera vez consecutiva y con 0, que no. La función devuelve un 1 si la acción ha terminado de ejecutarse y un 0 si no.

```
public abstract class Action // abstract class that represents an action done by a
robot
{
    public abstract string Print(); // returns string with relevant information
    about the action
    public abstract int Execute(int flanc, IRB120 obj, Communications comms);
    // Execute method executes tha action and tells the Program to continue by
    returning 1 and not 0 (see Program code) when the action has been executed
    // flanc variable determines if a certain action is being executed for the first
    consecutive time (1) or not (0)
}
```

Figura 52: Definición de la clase abstracta *Action*.

2.4.1.2 Clase *Wait*

Esta clase representa una espera de un cierto intervalo de tiempo por parte del robot. El parámetro de la clase es la variable *time*, el tiempo de espera en segundos, argumento requerido en la creación de *Waits*.

En su función *Execute*:

1. Si *flanc* vale 1, se almacena el tiempo actual en la variable estática *t* y se devuelve 0.
2. Si *flanc* vale 0: si el tiempo transcurrido desde *t* es menor que *time*, se devuelve 0, si no, se devuelve 1 para que el programa continúe.

```
    public override int Execute(int flanc, IRB120 robot, Communications comms)
    // override of Execute method
    {
        if (flanc != 0) // if it's the first consecutive time the action is being
        executed, the time is stored in t
        {
            t = Time.time;
            return 0;
        }
        else if (Time.time - t < time) // t is used as the reference time, if the
        time passed is smaller than parameter time, Execute returns 0
        {
            return 0;
        }
        else // if the time passed is gretaer than or equal to time, Execute
        returns 1 so the Program can continue executing
        {
            Debug.Log("[W] Waited " + time + " seconds.");
            return 1;
        }
    }
```

Figura 53: *Execute* de las *Action* de tipo *Wait*.

2.4.1.3 Clase *GoTo*

Esta clase representa la acción de alcanzar un punto con la herramienta por parte del robot. Su parámetro es una variable de tipo *pointData* llamada *pd*. La estructura *pointData* contiene la información necesaria para definir un punto a alcanzar por el robot y consta de los siguientes campos:

- *target*. Cadena de caracteres con la información necesaria para que RobotStudio pueda definir un *robtarget* para el robot real.
- *number*. El número ordinal del punto dentro del programa al cual pertenece la acción *GoTo* a la que está asociado.
- *mode.* Su valor "J" indica movimiento no lineal y su valor "L" indica movimiento lineal (instrucciones *MoveJ* y *MoveL* en RobotStudio respectivamente).
- *velocity*. Puede tomar los valores "1", "2" o "3", que indican valores bajo, medio y alto de velocidad de movimiento respectivamente (constantes *speeddata* en RobotStudio).
- *precision.* Puede tomar los valores "1", "2" o "3", que indican valores bajo, medio y alto de precisión de movimiento respectivamente (constantes *zonedata* en RobotStudio).

En su función *Execute*:

1. Si *flanc* vale 1, se envía el mensaje adecuado a RobotStudio a través de *Communications* y se pone su variable *executed*, indicador de ejecución, a 0, *Communications* indicará el fin de la ejecución asignándole el valor 1. Se devuelve 0.
2. Si *flanc* vale 0, cuando *executed* vale 1 se devuelve 1 para que el programa continúe.

```
public override int Execute(int flanc, IRB120 robot, Communications comms)
// override of Execute method
{
    if (flanc != 0) // if it's the first consecutive time the action is being
    executed, RobotStudio is told to execute a Move instruction
    {
        SendGoTo(pd, comms);
    }
    else if (comms.executed == 1) // if Communications indicates the action
    has been executed, a 1 is returned
    {
        Debug.Log("[GT] Gone to point " + pd.number);
        return 1;
    }
    return 0; // 0 is returned in every other case
}
private void SendGoTo(pointData pd1, Communications comms)
{
    pd1.target = pd1.target.Replace("XC", string.Empty); // the message sent
    is coherent with what RobotStudio expects to receive: only "X" at the
    beginning means execution of a movement to a point
    comms.message = "X" + pd1.velocity + pd1.precision + pd1.mode +
    pd1.target;
    comms.executed = 0; // Communications' execution indicator is set to 0,
    will indicate execution is finished with value 1
    comms.flag = true; // Communications' flag is activated for the
    communication to be done
}
```

Figura 54: *Execute* de las *Action* de tipo *GoTo*.

2.4.1.4 Clase *MoveAxis*

Esta clase representa la acción de alcanzar una cierta combinación de ángulos en sus articulaciones por parte del robot. Como parámetros requiere un vector con los ángulos en cuestión llamado *angles* y otros dos parámetros de velocidad y precisión que funcionan como los de la acción *GoTo*. Su función *Execute* sigue un proceso análogo al de la de *GoTo*.

```
public override int Execute(int flanc, IRB120 robot, Communications comms)
// override of Execute method
{
    if (flanc != 0) // if it's the first consecutive time the action is being
    executed,  RobotStudio is told to execute a MoveAbsJ instruction (reach a
    certain angle combination)
    {
        SendMovAx(angles, comms, robot);
    }
    else if (comms.executed == 1) // if Communications indicates the action
    has been executed, a 1 is returned
    {
        Debug.Log("[MA] Executed axis move");
        return 1;
    }
    return 0; // 0 is returned in every other case
}
private void SendMovAx(float[] angles, Communications comms, IRB120 robot)
{
    // the message sent is coherent with what RobotStudio expects to receive:
    only "A" at the beginning means execution of a movement to reach a
    certain angle combination
    comms.message = "A" + vel + precision + "[" + angles[0] + "," + angles[1]
    + "," + angles[2] + "," + angles[3] + "," + angles[4] + "," + angles[5] +
    "]";
    comms.executed = 0; // Communications' execution indicator is set to 0,
    will indicate execution is finished with value 1
    comms.flag = true; // Communications' flag is activated for the
    communication to be done
}
```

Figura 55: *Execute* de las *Action* de tipo *MoveAxis*.

2.4.1.5 Clase *ActDeact*

Esta clase representa la acción de activar o desactivar la herramienta del robot, en este caso abrir o cerrar la pinza. Requiere como único parámetro el booleano llamado *tool*, que representa el estado de la herramienta buscado: *true* para abierta y *false* para cerrada.

En su función *Execute*:

1. Si *flanc* vale 1, se envía el mensaje adecuado a RobotStudio a través de *Communications* y se pone su variable *executed*, indicador de ejecución, a 0. *Communications* indicará el fin de la ejecución asignándole el valor 1 al indicador. Se devuelve 0.
2. Si *flanc* vale 0:
    a. Cuando *executed* vale 1, es decir, la herramienta real ya está actuando, se asigna el valor 2 a *executed* y se manda actuar a la herramienta virtual asignando el valor 0 al indicador *actTool* y pasándole como estado requerido para la herramienta (variable *tool* de *IRB120*) el parámetro *tool* de la acción. Se devuelve 0.
    b. Si *executed* vale 2, cuando *actTool* vale 1, que significa que le herramienta virtual ha cumplido con la orden asignada, se devuelve 1 para que el programa continúe.

```
    public override int Execute(int flanc, IRB120 robot, Communications comms)
    // override of Execute method
    {
1       if (flanc != 0) // if it's the first consecutive time the action is being
        executed, RobotStudio is told to activate or deactivate the tool's
        signals
        {
            // the message sent is coherent with what RobotStudio expects to
            receive:
            comms.message = "Y"; // "Y" means tool-related instruction
            if (tool)
            {
                comms.message += "T"; // "T" means activating (opening) the tool
            }
            else
            {
                comms.message += "F"; // "F" means deactivating (closing) the
                tool
            }
            robot.actTool = 1; // virtual robot's tool readiness indicator is set
            to 1 (virtual tool is ready in its current state) to ensure the
            process will be done correctly
            comms.executed = 0; // Communications' execution indicator is set to
            0, will indicate execution is finished with value 1
            comms.flag = true; // Communications' flag is activated for the
            communication to be done
            return 0;
        }
2       else
        {
a           if (comms.executed == 1) // if Communications indicates the real
            action has started being executed, the indicator is put in an
            auxiliary state and the virtual tool is told to act
            {
                comms.executed = 2; // auxiliary state
                robot.actTool = 0; // the tool is not ready
                robot.tool = tool; // the virtual robot is told the new tool
                state
            }
b           else if (comms.executed == 2) // if Communications' execution
            indicator is in auxiliary state:
            {
                if (robot.actTool == 1) // if the tool is ready
                {
                    if (tool)
                    {
                        Debug.Log("[AD] Activated tool.");
                    }
                    else
                    {
                        Debug.Log("[AD] Deactivated tool.");
                    }
                    return 1; // 1 is returned for the Program to continue
                }
            }
            return 0; // 0 is returned is every other case
        }
    }
```

Figura 56: *Execute* de las *Action* de tipo *ActDeact.*

2.4.2 Código *Program*

Este código se encarga de albergar y controlar las secuencias de instrucciones para los robots y va asociado a los objetos destinados a mantenerlos activos dentro de la interfaz.

Las variables que emplea son las siguientes:

- *iAct* e i*Ant*. Únicas variables privadas. La primera es el índice de posición de la *Action* siendo ejecutada y la segunda, una variable auxiliar que permite la detección de cambios en el valor de *iAct*.
- *actions.* Lista de *Actions* que conforman la secuencia de instrucciones para el robot.
- *exe*. Booleano que determina si la secuencia ha de ejecutarse o no.
- *loop*. Booleano que determina si la ejecución de la secuencia debe realizarse en bucle o no.
- *Robot* y *comms*. El robot al que se aplica el programa y el código que maneja las comunicaciones.

```
public List<Action> actions = new List<Action>(); // Action list to be
executed, the content of the Program
public bool exe = false; // boolean that determines if the Program must run
(true) or not (false)
public bool loop = false; // boolean that determines if the Program must run
continuously (true) or just once (false)
public int ngots = 0; // number of GoTo Actions (used by ProgramManager)
public IRB120 Robot; // virtual robot to which the program applies
[HideInInspector]
public Communications comms; // script that handles communications with
RobotStudio

private int iAct = 0; // index for action in the list currently in execution
private int iAnt = -1; // auxiliar index for detection of changes of value
(flancs) in iAct
```

Figura 57: Variables de *Program*.

En la función *Update* se llama a la función *Execute*. La función *Execute* efectúa la ejecución sólo si se activa *exe* y sigue los siguientes pasos:

1. Se obtiene el valor de *inc* como el resultado de ejecutar la *Action* actual, entregando como *flanc* el resultado de la operación *XOR* entre *iAct* e *iAnt*, que sólo vale 1 si tienen distinto valor. De esta manera, *flanc* sólo vale 1 si es la primera vez consecutiva que se ejecuta la *Action*. *iAnt* se inicializa a -1 para que se genere correctamente el primer *flanc*.
2. Se incrementa el índice *iAct* en un valor de *inc*. De esta manera, sólo se avanza en el programa cuando *inc* vale 1, es decir cuando el *Execute* de la Action indica que la ejecución real de la acción ha terminado.
3. Si *inc* vale 0, es decir, el programa todavía no avanza, se actualiza el valor de *iAnt* para que el valor enviado como *flanc* sea 0 y cumpla con su función.
4. Si *iAct* alcanza el valor del número de elementos en *actions*, significa que la última *Action* ya ha sido ejecutada, ya que el índice empieza en la posición 0. Entonces, se reinicia *iAct* y si *loop* no está activo, se desactiva *exe* y se reinicia *iAnt* para que pueda repetirse el proceso cuando el usuario lo requiera.

```
    void Update() // Update is called once per frame
    {
        Execute();
    }

    void Execute() // executes each action in the order they were added if
    boolean exe is true
    {
        int inc; // the value the index iAct will be incremented each time an
        Action is executed
        if (exe)
        {
1           inc = actions[iAct].Execute(iAct^iAnt,Robot,comms); // inc will be 0
            if the real action has not been executed yet and 1 if it has
2           iAct += inc; // iAct^iAnt (the Action flanc, see Action) will only be
            1 if they have different values: only when iAct's value has changed
3           if (inc == 0) // if the program will proceed to the next action (inc
            = 1) iAnt must stay the same for the flanc to be sent to the next
            action correctly
            {
                iAnt = iAct;
            }
4           if (iAct == actions.Count) // if last action has been executed:
            {
                iAct = 0; // index is reset
                if (!loop) // if the program was meant to execute just once:
                {
                    exe = false; // stop executing
                    iAnt = -1; // for the next flanc to be sent correctly
                }
            }
        }
    }
```

Figura 58: *Update* de *Program*.

La función *Print* devuelve una cadena de caracteres que contiene las informaciones de *Print* de todas las *Action*. Si el programa estuviese vacío, devuelve una frase que lo indica.

```
    public string Print() // returns a string containing each action and its
    relevant information
    {
        string progString = ""; // initialise the string
        for (int i = 0; i < actions.Count; i++)
        {
            progString += actions[i].Print() + System.Environment.NewLine; // add
            each action
        }
        if (progString == "") // if string is empty:
        {
            return ("Program is empty...");
        }
        return progString;
    }
```

Figura 59: Función *Print* de *Program*.

2.4.3 Código *ProgramManager*

Este código va asociado al objeto *ProgramManager* de la interfaz y se encarga de manejar todo lo relativo a los programas para los robots. Su funcionamiento se basa en el uso de funciones públicas llamadas a través de los elementos de la interfaz.

La función *SetProgram* es llamada desde los botones de programa de la interfaz y actualiza el programa que se va a modificar.

```
public void SetProgram(GameObject prgrm) // sets the Program with which the
user is currently working
{
    prog = prgrm.GetComponent<Program>();
    prog.comms = comms; // sets the Communications object of the Program
}
```

Figura 60: Función *SetProgram* de *ProgramManager*.

La función *AddAction* agrega la *Action* que se dé como argumento a la lista *actions* del programa con el que se está trabajando.

```
private void AddAction(Action act) // adds an Action to the Program
{
    prog.actions.Add(act); // the Action is added through list's Add method
}
```

Figura 61: Función *AddAction* de *ProgramManager*.

La función *RemoveAction* elimina la *Action* que se encuentre en la posición especificada de *actions* y decrementa en 1 el valor del contador *ngots* si se trata de una *Action GoTo*. No se ha podido implementar la llamada de esta función. La función *UpdateProgText* se explica más adelante en este apartado.

```
public void RemoveAction(int index) // removes the Action in the position
index of the Program's Action list // not fully implemented yet
{
    if (prog.actions[index].Print().Contains("point")) // if the Action is
    GoTo:
    {
        prog.ngots--; // number of GoTos gets decremented by one
    }
    prog.actions.RemoveAt(index); // the Action is deleted through list's
    RemoveAt method
    UpdateProgText();
}
```

Figura 62: Función *RemoveAction* de *ProgramManager*.

La función *EmptyProgram* vacía la lista *actions* y reinicia el contador *ngots*.

```
public void EmptyProgram() // empties the Program's content
{
    prog.actions.Clear(); // the list of actions is cleared through list's
    Clear method
    prog.ngots = 0; // the number of GoTo Actions is reset
    UpdateProgText();
}
```

Figura 63: Función *EmptyProgram* de *ProgramManager*.

La función *AddingAction* permite añadir *Actions* de tipo *Wait* o *ActDeact* en función la *string adding* que recibe como argumento. Si *adding* es "w", se procede a añadir una *Wait*: su parámetro se encuentra en formato *string* en la variable *waitString*, por lo que se convierte a *float* mediante el método *Parse* antes de crearla y añadirla. Además, se vacía *waitString* y se reestablece el valor de *waitText* a su valor por defecto, el uso de estas dos variables se detalla más adelante en este apartado. La función se llama con "w" como argumento desde el botón de *WaitMenu* cuyo texto es el símbolo habitual de la tecla "*Enter*" en un teclado de ordenador. Si *adding* es "t", se procede a crear y añadir una *ActDeact* usando como parámetro el booleano *tool*. Dado que no tiene sentido que el estado requerido para la herramienta sea el mismo en el que ya está, *tool* cambia automáticamente cada vez que se va a añadir una *ActDeact*. Esto no impide que se pueda dar la situación descrita si el programa se ejecuta en bucle, razón por la cual el código *ToolFinger* está preparado para afrontarla.

```
public void AddingAction(string adding) // adds Wait or ActDeact Actions
depending on adding string argument, different buttons give different
arguments
{
    if (adding == "w") // "w" means Wait
    {
        float t;
        t = float.Parse(waitString); // parameter t for new Wait's time is
        created
        waitString = "";
        waitText.text = "Enter time..."; // string component of Text for
        displaying wait time is reset
        Action act = new Wait(t);
        AddAction(act); // new Wait is added
        UpdateProgText();
        Debug.Log("[PM] Added wait of: " + t + " sec.");
    }
    else if (adding == "t") // "t" means ActDeact
    {
        tool = !tool; // the tool state boolean's value is changed
        Action act = new ActDeact(tool);
        AddAction(act); // new ActDeact is added
        UpdateProgText();
        Debug.Log("[PM] Added tool action: " + tool + ".");
    }
}
```

Figura 64: Función *AddingAction* de *ProgramManager*.

La función *ExecuteProgram* cambia el estado del booleano *exe* del programa en uso y activa o desactiva la colisión entre las capas 8 y 10, la de la herramienta y la de los objetos que el robot puede agarrar, en función de si se ejecuta o no el programa respectivamente.

```
public void ExecuteProgram() // changes the Program's execution boolean state
{
    prog.exe = !prog.exe;
    if (prog.exe) // if the program is now executing
    {
        Debug.Log("[PM] " + this.gameObject.name + " is executing.");
        Physics.IgnoreLayerCollision(8, 10, false); // collision between Tool
        and Grababble layers is enabled
    }
    else
    {
        Debug.Log("[PM] " + this.gameObject.name + " is NOT executing.");
        Physics.IgnoreLayerCollision(8, 10, true); // collision between Tool
        and Grababble layers is disabled
    }
}
```

Figura 65: Función *ExecuteProgram* de *ProgramManager*.

La función *creatingGoTo* activa el indicador de que se está creando una acción *GoTo* del código *ViveController*, *settingPoint*. La función *notCreatingGoTo* desactiva *settingPoint*. Este indicador activa o desactiva la funcionalidad que proporciona el código *ViveController*, que controla el proceso de señalar puntos para crear *GoTos* y se explica más adelante en el documento.

```
public void creatingGoTo() // changes the settingPoint indicator accordingly
for the correspondent functionalities to be enabled
{
    if (!vc.settingPoint) vc.settingPoint = true;
}

public void notCreatingGoTo() // sets the settingPoint indicator to false for
the corresponding functionalities to be disabled
{
    if (vc.settingPoint) vc.settingPoint = false;
}
```

Figura 66: Funciones *creatingGoTo* y *notCreatingGoTo* de *ProgramManager*.

La función *finishedGoTo* se llama desde el botón *Accept* del menú de creación de *GoTos* y finaliza dicho proceso. Incrementa *ngots* del *Program* en uso y crea y añade una *Action* de tipo *GoTo* a partir de un *pointData* creado con estas variables:

- *savedMessage* de *Communications* como *target*. Tras señalarse un punto con el mando mediante *ViveController*, se realiza una comunicación con RobotStudio en la que se comprueba la viabilidad de dicho punto. Si se recibe el visto bueno de RobotStudio, los datos del punto se guardan en *savedMessage* y son utilizados para crear el *GoTo* si el usuario finalemnete lo decide.
- *ngots* como *number*.

- *mod* como *mode*. Esta es la variable que determina si se requiere que el movimiento empleado por el robot para alcanzar el punto debe ser lineal o no. Su valor se modifica en función de lo que seleccione el usuario en un elemento de tipo *Toggle* de la interfaz llamada *LinearToggle*.
- *vel* como *velocity* y *prec* como *precision*. Ambas variables se refieren a parámetros del movimiento y sus valores son modificados de forma análoga a la del de *mod*, pero mediante elementos *Dropdown* que permiten escoger entre 3 opciones: baja, media y alta.

```
public void finishedGoTo() // creates the new GoTo Action and ends the
process
{
    prog.ngots++; // updates number of GoTo Actions in the Program, this
    number also represents the ordinal number of the GoTo Action in the
    program
    pointData pd = new pointData // pointData for the GoTo Action
    {
        target = comms.savedMessage, // the target is the last message sent
        to RobotStudio for reachability checking
        number = prog.ngots,
        mode = mod,
        velocity = vel,
        precision = prec
    };
    Action act = new GoTo(pd); // the new GoTo Action is given its position
    number and obtained from Communications
    AddAction(act);
    UpdateProgText();
    notCreatingGoTo(); // ends process
}
```

Figura 67: Función *finishedGoTo* de *ProgramManager*.

La función *WaitStringAdd* añade la *string* que recibe como argumento a *waitString* y actualiza *waitText* para que muestre el contenido de *waitString*. Esta función se llama desde cada botón del teclado virtual de *WaitMenu*. Cada botón proporciona como argumento el carácter que representa. La función *DeleteWString* elimina el último carácter de *waitString* y actualiza *waitText*. Esta función es llamada desde el botón de *WaitMenu* cuyo texto es el símbolo habitual de la tecla "Eliminar" en un teclado de ordenador.

```
public void WaitStringAdd(string s) // adds an element to the string
component of the Text displaying the waiting time of the Wait Action being
created
{
    waitString += s;
    waitText.text = waitString;
}

public void DeleteWString() // deletes last element in the string component
of the Text displaying the waiting time of the Wait Action being created
{
    waitString = waitString.Remove(waitString.Length-1);
    waitText.text = waitString;
}
```

Figura 68: Funciones *WaitStringAdd* y *DeleteWString* de *ProgramManager*.

Las funciones *UpdateMode*, *UpdateVelocity* y *UpdatePrecision* actualizan los valores de *mod*, *vel* y *prec* respectivamente desde sus respectivos elementos de interfaz.

```
public void UpdateMode(bool b)  // updates the mode to be stored in the next
GoTo Action created
{
    mode = b;
    if (mode)
    {
        mod = "L";
    }
    else
    {
        mod = "J";
    }
}
public void UpdateVelocity(int i) // updates the velocity to be stored in the
next instruction of movement's Action created
{
    vel = (i+1).ToString();
}
public void UpdatePrecision(int i) // updates the precision to be stored in
the next instruction of movement's Action created
{
    prec = (i+1).ToString();
}
```

Figura 69: Funciones *UpdateMode*, *UpdateVelocity* y *UpdatePrecision* de *ProgramManager*.

La función *ConfCollisionDetection* es la única llamada desde la función *Start* del código y configura la detección de colisiones entre todas las capas de la siguiente forma:

- Desactiva las colisiones entre la capa de la herramienta (8) y el resto de capas con las que es posible que interactúe: la de los objetos que el robot puede agarrar (10), ya que sólo interesa que pueda interactuar con esta capa cuando se esté ejecutando un programa, la de la interfaz (5), la de la estructura que conforma la fábrica (9) y la capa por defecto (0), en la que se encuentra todo lo que no se encuentre en otra capa.
- Desactiva las colisiones entre la capa del suelo (11) y la de la interfaz (5), la de la estructura (9) y la capa por defecto (0), ya que no es necesario.
- Desactiva las colisiones entre la capa de los objetos que el robot puede agarrar (10) y las capas de la interfaz (5) y por defecto (0).

```
private void ConfCollisionDetection() // disables the collision detection
between certain layers
{
    // layer 8 = Tool: the virtual tool object is in this layer
    // layer 11 = Floor: the whole floor of the application environment is in
    this layer
    // layer 10 = Grabbable: the objects that are grabbable by the virtual
    robot are in this layer
    // layer 5 = UI: the user interface is in this layer
    // layer 9 = Structure: the structure of the virtual fatory is in this
    layer
    // layer 0 = Default: most other objects are in this layer, the player,
    for example
    Physics.IgnoreLayerCollision(8, 8, true);
    Physics.IgnoreLayerCollision(8, 10, true);
    Physics.IgnoreLayerCollision(8, 5, true);
    Physics.IgnoreLayerCollision(8, 0, true);
    Physics.IgnoreLayerCollision(8, 9, true);
    Physics.IgnoreLayerCollision(11, 5, true);
    Physics.IgnoreLayerCollision(11, 0, true);
    Physics.IgnoreLayerCollision(11, 9, true);
    Physics.IgnoreLayerCollision(10, 0, true);
    Physics.IgnoreLayerCollision(10, 5, true);
}
```

Figura 70: Función *ConfCollisionDetection* de *ProgramManager*.

La función *UpdateProgText* asigna al contenido *string* del texto *progText* la información relevante del *Program* en uso mediante su función *Print*. El *Text progText* es un texto que se muestra en los menús de la interfaz relacionados con la modificación del programa para que el usuario pueda ver lo que está creando.

```
public void UpdateProgText() // updates the string component of the Text that
displays a Program's content
{
    progText.text = prog.Print();
}
```

Figura 71: Función *UpdateProgText* de *ProgramManager*.

La función *ChangeLoop* asigna al booleano *loop* del *Program* que se está usando el valor del booleano asociado a un elemento de tipo *Toggle* que aparece junto al botón de ejecutar del menú *ProgramOptions*, desde el cual se pueden ejecutar, editar o vaciar programas.

```
public void ChangeLoop(bool b) // changes the Program loop boolean to value b
{
    prog.loop = b;
}
```

Figura 72: Función *ChangeLoop* de *ProgramManager*.

La función *ModAngButton* permite al código identificar el botón del *MoveAxisMenu*, menú que permite crear acciones *MoveAxis*, que se ha pulsado y la función *ModAngle* permite identificar el eje cuyo ángulo se quiere modificar. Ambas funciones se llaman desde los botones de *MoveAxisMenu*, cada botón tiene un eje asociado cuyo número es mostrado por su texto.

```
public void ModAngButton(UI_Element bttn) // allows this script to identify
the button pressed (called from the button)
{
    button = bttn;
}
public void ModAngle(int ax) // allows this script to identify the
corresponding axis to the button pressed (called from the button)
{
    axis = ax;
}
```

Figura 73: Funciones *ModAngButton* y *ModAngle* de *ProgramManager*.

La función *ChangingAngle* es la única llamada desde la *Update* del código y se encarga de manejar el menú de creación de *MoveAxis*. Sigue los siguientes pasos:

1. Si el texto que muestra algún ángulo de eje del robot está activo, significa que todos lo están, ya que forman parte del mismo menú y por lo tanto se activa la actualización de sus valores mediante *DispAngles*.
2. Se asigna el estado de activación del botón con el que se esté trabajando a *butact*
3. Si está pulsado:
    a. Si no se estaba ya modificando el ángulo en cuestión, se actualiza el estado de *angleChange*, indicador de dicho proceso, y se almacena el valor original del ángulo antes de que sea modificado como valor de referencia en *tempAngle*.
    b. Si se estaba modificando el ángulo, es decir, *angleChange* está activo:
        i. Si se pulsa el botón *ApplicationMenu* derecho (también usado para la función *GoBack* de la interfaz) se reestablece el valor del ángulo y se cancela el proceso mediante *CancelAngleChange*, activando el indicador de cancelación *cancel*.
        ii. Si no, se permite modificar el valor del ángulo utilizando la posición vertical del *joystick* derecho siempre y cuando el ángulo del robot no alcance en valor absoluto el valor del ángulo de referencia incrementado en 180 o, si lo ha hecho, el valor de posición del *joystick* tenga el signo necesario para decrementar en valor absoluto el ángulo del robot.
4. Si el botón acaba de pasar a estar desactivado y no se ha cancelado el proceso, se comprueba la viabilidad de la configuración de ángulos actual mediante *CheckAxes*.
5. Finalmente, se actualiza el valor de *butant*, variable que permite detectar los flancos de bajada en el estado de activación del robot.

```
    private void ChangingAngle() // allows the user to change tha angles of the
    robot
    {
        bool cancel = false;
1       if (textAngle1.IsActive()) // if any of the texts for the display of the
         angles is visible by the user, the values keep updating
        {
            DispAngles();
        }
        if (button != null) // if a button has been identified:
        {
2           butact = button.clicked;
3           if (butact) // if the button is clicked:
            {
a               if (!angleChange) // if the user wasn't already changing an angle
                   value:
                {
                    angleChange = true; // indicator is set to true
                    tempAngle = robot.axisAngles[axis]; // reference angle is
                        initialised with current value
                }
b               else
                {
i                   if
(rightHand.controller.GetPressDown(SteamVR_Controller.ButtonMask.ApplicationMenu))
// if ApplicationMenu button (B in Oculus Touch) is pressed:
                    {
                        robot.axisAngles[axis] = tempAngle; // the angle is reset
                            to its original value
                        CancelAngleChange(); // the process is cancelled
                        cancel = true;
                    }
ii                  else if
(Mathf.Abs(robot.axisAngles[axis]) < Mathf.Abs(tempAngle) + 180 ||
// rotating the virtual robot more than +180 or -180 from the reference angle is redundant and
therefore not allowed
robot.axisAngles[axis] >= tempAngle + 180 &&
rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1] < 0 ||
// if the virtual robot has been rotated +180 from the reference angle, only negative rotation
is allowed
robot.axisAngles[axis] <= tempAngle - 180 &&
rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1] > 0)
// if the virtual robot has been rotated -180 from the reference angle, only positive rotation
is allowed
                    {
                        robot.axisAngles[axis] +=
rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1];
// angle value is incremented or decremented based on the vertical position of the right
joystick
                    }
                }
            }
4           else if (butact ^ butant && !cancel) // if the button has just been
             unclicked (clicked a second time)
            {
                CheckAxes(); // the current angle configuration's reachability is
                   checked
            }
5           butant = butact; // the auxiliary button variable is updated to allow
             detection of changes in button's state
        }
    }
```

Figura 74: Función *ChangingAngle* de *ProgramManager*.

La función *DispAngles* actualiza el valor de los textos *textAngle*, colocados en el menú *MoveAxisMenu*, para que muestren en tiempo real los valores de los ángulos del robot en cada uno de sus ejes.

```
private void DispAngles() // updates the values of the Texts that display the
virtual robot's angles
{
    textAngle1.text = (robot.axisAngles[0]).ToString() + "º";
    textAngle2.text = (robot.axisAngles[1]).ToString() + "º";
    textAngle3.text = (robot.axisAngles[2]).ToString() + "º";
    textAngle4.text = (robot.axisAngles[3]).ToString() + "º";
    textAngle5.text = (robot.axisAngles[4]).ToString() + "º";
    textAngle6.text = (robot.axisAngles[5]).ToString() + "º";
}
```

Figura 75: Función *DispAngles* de *ProgramManager*.

La función *CheckAxes* crea el mensaje necesario para que RobotStudio realice la comprobación de viabilidad de la configuración de ángulos actual y activa el *flag* de *Communications* para que se realice la comunicación.

```
private void CheckAxes() // sends the current angle configuration of the
virtual robot to RobotStudio for the reachability to be checked
{
    // the meessage sent is coherent with what RobotStudio expects: "AC"
    means checking the reachability of a certain angle configuration
    comms.message = "AC" + "[" + robot.axisAngles[0] + "," +
    robot.axisAngles[1] + "," + robot.axisAngles[2] + "," +
    robot.axisAngles[3] + "," + robot.axisAngles[4] + "," +
    robot.axisAngles[5] + "]";
    comms.flag = true;
}
```

Figura 76: Función *CheckAxes* de *ProgramManager*.

La función *CancelAngleChange* llama a la función *ExecutePress* del botón para generar la segunda pulsación que necesita para desactivarse y desactiva el indicador *angleChange*.

```
private void CancelAngleChange() // resets the axis button state and the
boolean indicator angleChange is set to false
{
    button.ExecutePress(); // executes a second press in the button that was
    pressed (it's a button that can be toggled)
    angleChange = false;
}
```

Figura 77: Función *CancelAngleChange* de *ProgramManager*.

La función *AcceptedAngleChange* desactiva *angleChange* y crea y añade una *Action* de tipo *MoveAxis* mediante los valores actuales de los ángulos del robot y “2” (valor medio) como parámetro de velocidad y “3” (valor alto) como parámetro de precisión. Se ha escogido no dar a elegir el valor de estos parámetros por ser la acción *MoveAxis* poco habitual en la programación de los robots.

```
public void AcceptedAngleChange() // a new MoveAxis Action is added to the
Program, velocity and precision are provisionally set to medium value ("2"),
the ability to choose should be eventually implemented
{
    angleChange = false;
    Action act = new MoveAxis(robot.axisAngles,"2","3");
    AddAction(act);
    UpdateProgText();
    Debug.Log("[PM] Added movement of axes.");
}
```

Figura 78: Función *AcceptedAngleChange* de *ProgramManager*.

2.4.4 Código *ViveController*

Este código ya era parte de la aplicación original, pero ha sido ligeramente modificado para ajustarse a su nueva función dentro de la aplicación.

Se han añadido dos variables:

- *settingPoint*. Booleano que activa o desctiva la funcionalidad del código.
- *hidden*. Posición inicial del *Gizmo*, objeto que representa la posición y rotación del punto marcado, a la que este es teletransportado cuando *settingPoint* no está activo. Su valor es inicializado en la *Awake* del código.

Además, se han eliminado ciertas partes de manera que el código funcione de la siguiente forma:

1. Si *settingPoint* está activado:
    a. Si se pulsa el *Grip* izquierdo se teletransporta continuamente el *Gizmo* a la posición del mando virtual.
    b. Si se suelta el *Grip*, se realiza una comunicación con RobotStudio que comprueba la viabilidad del punto en el que se encuentre el *Gizmo*. El mensaje mandado empieza por la clave “XC” que necesita RobotStudio para que se realice el proceso correctamente.
    c. Si se modifican la rotación o posición del *Gizmo* utilizando el *Trigger* izquierdo, se repite el proceso de comprobación.
2. Si *settingPoint* no está activado, se asigna la posición *hidden* al *Gizmo*.

El código puede encontrarse en el Anexo III.

2.4.5 Código *Communications*

Este código ya era parte de la aplicación original, pero también ha sido ligeramente modificado para cumplir mejor con su cometido en la nueva aplicación. Sólo se ha modificado la comunicación TCP/IP. Los cambios realizados se explican a continuación:

- Se ha añadido una primera comunicación TCP/IP que se produce al iniciarse la aplicación y en la que Unity envía el mensaje "I" a RobotStudio para que éste active la comunicación UDP/IP, de manera que la aplicación se inicie con el robot virtual y el robot real en la misma posición. Esta comunicación se realiza según el booleano *init*, que se inicializa como *true* y pasa a *false* al realizarse la comunicación.
- Se ha añadido una variable de tipo *string* llamada *savedMessage* que almacena cada mensaje que se envía a RobotStudio para su uso posterior en caso de recibir el visto bueno.
- Se ha añadido una variable de tipo entero llamada *executed* que toma distintos valores según se haya ejecutado la acción enviada. Para cambiar su valor se busca la palabra "*success*" en el mensaje recibido. El valor 0 indica que la acción no se ha ejecutado todavía y el valor 1, que sí, aunque en el caso de la ejecución de *ActDeacts* se utiliza un estado auxiliar con valor 2.

El código puede encontrarse en el Anexo III y los procesos de comunicación mencionados se explican en el Capítulo 3.

2.4.6 Código *TestObjects*

Este código va asociado a los objetos de prueba que el robot puede agarrar y hace que se teletransporten a su posición inicial brevemente después de tocar el suelo. Este código ha sido implementado para facilitar la comprobación de la fiabilidad de la aplicación en la ejecución de programas en bucle y puede encontrarse en el Anexo III.

Capítulo 3 Control del robot mediante RobotStudio

El proyecto de RobotStudio cuenta con dos módulos o códigos que se ejecutan simultáneamente e interactúan con Unity mediante distintos canales de comunicación:

- *MainModule*. Este código se encarga de interactuar con el módulo de comunicación TCP/IP de Unity, recibiendo y enviando mensajes referentes a la comprobación y ejecución de acciones por parte del robot. El protocolo TCP/IP es lento, pero fiable. Este código ha sido modificado respecto a su versión original para su correcto funcionamiento dentro de la nueva aplicación.
- *SecondaryModule*. Este código se encarga de interactuar con el módulo de comunicación UDP/IP de Unity, transmitiendo en tiempo real la configuración de los ángulos del robot real para que el robot virtual pueda imitarla. El protocolo UDP/IP es rápido, pero menos fiable. Este código realiza su función según el estado de un booleano llamado *executingPath*, el cual es modificable por *MainModule*. Este código no ha sido modificado.

3.1 Código *MainModule*

Este código está basado en el código con la misma función del trabajo original. Se encarga de recibir, responder y ejecutar instrucciones en función de los mensajes recibidos de Unity, interactuando únicamente a través del módulo TCP/IP de la comunicación.

En primer lugar, se configura la comunicación y se obtiene el mensaje recibido en formato *string* mediante la función *UnpackRawBytes*, esta parte no ha sido modificada respecto al trabajo original.

A continuación, se procede a analizar y actuar en función del mensaje recibido:

Si se recibe "I", se trata de la inicialización de la aplicación. Por lo tanto, se desactiva la herramienta mediante el booleano *toolAct* y la función *UpdateToolState*, que se explica más adelante en este apartado, ya que no es lógico comenzar con la herramienta activada, y se activa el módulo UDP/IP mediante el booleano *executingPath*, para que el robot virtual adquiera exactamente la misma configuración de ángulos que el real. Este proceso se realiza durante 5 segundos para asegurar su correcta consecución.

```
IF StrPart(message,1,1) = "I" THEN
    toolAct := FALSE;
    UpdateToolState;
    executingPath:=TRUE;
    WaitTime 5;
    executingPath:=FALSE;
    SocketSend client,\str:="initialised";
    SocketClose client;
    SocketClose server;
```

Figura 79: Código del proceso de inicialización en RobotStudio.

Si se recibe un mensaje que empiece por "XC", se trata de la comprobación de viabilidad de un punto en el espacio, es decir, de una acción *GoTo*, y se siguen los siguientes pasos:

1. Se obtiene el *robtarget* correspondiente al punto mediante la función *StringToTarget* ya incluida en el trabajo original.
2. Se comprueba la viabilidad del punto mediante la función propia de RobotStudio *CalcJointT*, que calcula la combinación de ángulos necesaria para alcanzar el *robtarget* que se le dé como argumento. Durante este cálculo, se puede producir un error de viabilidad, cambiándose el valor de la variable *calculationResult* a -1 o -2 respectivamente y concluyéndose la comunicación. En caso de que *calculationResult* no haya tomado estos valores, se envían los ángulos a Unity. La parte del código que controla los errores mencionados puede encontrarse en el Anexo III.

```
! GoTo
ELSEIF StrPart(message,1,1) = "X" THEN
    ! Check reachability for GoTo
    IF StrPart(message,2,1) = "C" THEN
        ! Convert message string to robtaget object
        robotTarget :=
        StringToTarget(StrPart(message,3,StrLen(message)-2));
        ! Reset calculation result variable
        calculationResult := 0;
        ! Calculate the joint angles given the robot target, and
        store errors in errorNumber
        jointDataResult := CalcJointT(robotTarget, PinzaToolTCP,
        \WObj:=wobj0);
        ! If there were no errors
        IF calculationResult <> -1 AND calculationResult <> -2 THEN
            ! Send resulting robot configuration to Unity
            SocketSend client,
            \str:=jointTargetToString(jointDataResult);
            ! Close both server and client sockets
            SocketClose client;
            SocketClose server;
        ENDIF
```

Figura 80: Código de la comprobación de una acción *GoTo* en RobotStudio.

Si se recibe un mensaje que empiece por "X", se trata de la ejecución de una acción *GoTo* y se psiguen los siguientes pasos:

1. Se extraen los parámetros necesarios del mensaje de manera coherente a como fue construido en Unity mediante las funciones *GetSpeed* y *GetPrecision*.
2. Se activa el módulo UDP/IP mediante *executingPath* para que el robot virtual imite el movimiento.
3. Se ejecuta la instrucción *Move* correspondiente al modo seleccionado.
4. Se espera a que el robot esté parado para continuar con el programa mediante la instrucción *WaitRob* con el parámetro *InPos*. Aunque esta instrucción es necesaria para que se generen correctamente las trayectorias y no se produzcan errores durante la ejecución, también es la principal limitación de la aplicación, ya que no permite que se encadenen instrucciones de movimiento de forma fluida.

```
            ! Execute GoTo
            ELSE
                ! Get Move instruction parameters
                vel := StrPart(message,2,1);
                spd := GetSpeed(vel);
                prec := StrPart(message,3,1);
                zn := GetPrecision(prec);
                mode := StrPart(message,4,1);
                robotTarget :=
                StringToTarget(StrPart(message,5,StrLen(message)-4));
                executingPath := TRUE;
                ! Ignore current robot configuration during linear movement
                ConfL \Off;
                ! Execute Move instruction
                IF mode = "L" THEN
                    MoveL robotTarget, spd, zn, PinzaToolTCP;
                ELSE
                    MoveJ robotTarget, spd, zn, PinzaToolTCP;
                ENDIF
                ! Wait until finished moving
                WaitRob \InPos;
                executingPath := FALSE;
                SocketSend client,\str:="success";
                SocketClose client;
                SocketClose server;
            ENDIF
```

Figura 81: Código de la ejecución de una acción *GoTo* en RobotStudio.

```
FUNC speeddata GetSpeed(string value)
    VAR speeddata tempSpeed;
    IF value = "3" THEN
                tempSpeed := v2000;
            ELSEIF value = "2" THEN
                tempSpeed := v400;
            ELSE
                tempSpeed := v50;
            ENDIF
    RETURN tempSpeed;
ENDFUNC
FUNC zonedata GetPrecision(string value)
    VAR zonedata tempPrec;
        IF value = "1" THEN
                tempPrec := fine;
            ELSEIF value = "2" THEN
                tempPrec := z50;
            ELSE
                tempPrec := z200;
            ENDIF
    RETURN tempPrec;
ENDFUNC
```

Figura 82: Funciones *GetSpeed* y *GetPrecision* de *MainModule*.

Si se recibe un mensaje que empiece por "AC", se trata de la comprobación de viabilidad de una combinación de ángulos, es decir, de una acción *MoveAxis*. Si el mensaje empieza sólo por "A" se trata de la ejecución de una *MoveAxis*. Ambos procesos son análogos a los de comprobación y ejecución de una acción *GoTo*, pero con los siguientes matices:

- El *jointtarget* correspondiente a los ángulos recibidos se crea con una configuración por defecto añadiendo la *string* "[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]", ya que simplifica el proceso y este tipo de acción no se utiliza habitualmente en la programación de robots.
- La comprobación también se realiza aplicando la función *CalcJointT*, pero al *robtarget* equivalente a la combinación recibida, calculado mediante la función *CalcRobT*, que no permite la detección de errores necesaria para la comprobación. Aunque es posible que este método no sea el más eficiente computacionalmente, el código requerido es muy simple y este tipo de cálculo no requiere rapidez, ya que las comprobaciones sólo se realizan una vez.

La parte del código correspondiente puede verse en el Anexo III.

Si el mensaje recibido empieza por "Y", se trata de una instrucción referente a la herramienta. Si el siguiente carácter es "T", se trata de la activación de la herramienta, y si es "F", de desactivarla. El control de la herramienta se realiza mediante el booleano *toolAct*, que representa su estado, y la función *UpdateToolState*, que cambia su estado según *toolAct* accediendo a las señales internas del microcontrolador asociado al robot que lo controlan y sólo si es necesario.

```
            ! Activate or deactivate tool
            ELSEIF StrPart(message,1,1) = "Y" THEN
                IF StrPart(message,2,1) = "T" THEN
                    toolAct := TRUE;
                ELSEIF StrPart(message,2,1) = "F" THEN
                    toolAct := FALSE;
                ENDIF
                UpdateToolState;
                SocketSend client,\str:="success";
                SocketClose client;
                SocketClose server;
```

Figura 83: Código de control de la herramienta en RobotStudio.

```
PROC UpdateToolState()
    IF toolAct THEN
        IF pinzaOutput1 = 1 THEN
            SetDO pinzaOutput1, 0;
        ENDIF
        IF pinzaOutput0 = 0 THEN
            SetDO pinzaOutput0, 1;
        ENDIF
    ELSE
        IF pinzaOutput1 = 0 THEN
            SetDO pinzaOutput1, 1;
        ENDIF
        IF pinzaOutput0 = 1 THEN
            SetDO pinzaOutput0, 0;
        ENDIF
    ENDIF
ENDPROC
```

Figura 84: Función *UpdateToolState* de *MainModule*.

3.2 Creación de la herramienta

Tras haber creado el modelo 3D del conjunto de piezas completo que constituye la versión virtual de la herramienta, fue necesario exportarlo en formato *.sat* para poder importarlo en RobotStudio. Una vez en RobotStudio se siguió el siguiente procedimiento para crear una herramienta en base al modelo importado:

1. Se selecciona la opción Crear Herramienta de la pestaña de Modelado.
2. Se selecciona la pieza correspondiente al modelo de la herramienta. No se han introducido las coordenadas de su centro de gravedad ni sus momentos de inercia por desconocimiento e imposibilidad de medición, pero, al ser una herramienta ligera y pequeña, las consecuencias no serán apreciables.
3. Se introducen las coordenadas en ejes propios de la herramienta de su TCP, el punto del espacio sobre el que actúa la herramienta, es decir, el punto medio de la línea imaginaria que une los puntos medios de los extremos de los dedos de la pinza.
4. Se arrastra la herramienta creada al objeto del robot.
5. Se sincroniza el entorno de la estación de trabajo con el de programación para que los códigos reconozcan el nombre y las características de la nueva herramienta.

De esta manera, el robot de RobotStudio queda con una versión prácticamente idéntica de la herramienta real.

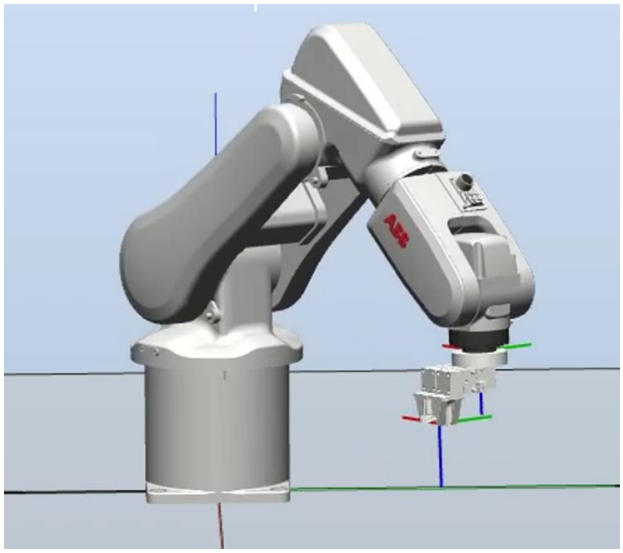

Figura 85: Robot de RobotStudio con la herramienta implementada.

Por último, para que la pinza real pueda abrirse y cerrarse, sólo hay que acceder a las señales internas que la controlan. Sin embargo, el ordenador en el que se encontraban instalados los softwares necesarios para trabajar con realidad virtual no tenía estas señales configuradas, por lo que, dado que su configuración escapa a los objetivos del proyecto, se optó por crear dos señales que emulan ser las señales reales. Estas señales se llaman *PinzaOutput0* y *PinzaOutput1*, la primera se corresponde con la señal real que está normalmente desactivada y la segunda, al contrario. Ambas señales se manejan en el código tal y como se haría con las reales, por lo que en el futuro sería suficiente con sustituir sus nombres por los de las señales reales en el *MainModule*.

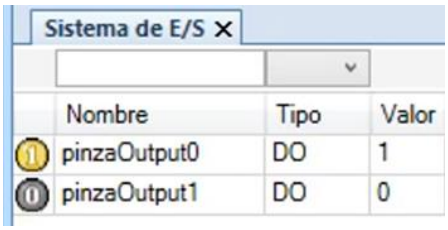

Figura 86: Señales creadas para simular la herramienta.

Capítulo 4 Resultados

En este capítulo se describe el resultado obtenido y se explican los problemas con los que ha quedado la aplicación final.

4.1 Funcionamiento

La aplicación final funciona prácticamente como debe y, aunque no ofrece la abundante variedad de funcionalidades que ofrece el dispositivo *FlexPendant*, permite programar el robot de forma básica tal y como se pretendía. Además, cuenta con funcionalidades extra como la posibilidad de ver la fecha, la hora y el tiempo que lleva la aplicación en uso o las relativas a la movilidad del usuario.

4.2 Problemas

El problema principal e intrínseco a como está diseñada la aplicación es la imposibilidad de crear trayectorias fluidas, siendo el parámetro *precision* de las *GoTo* y *MoveAxis* completamente inútil. Esto se debe a que *MainModule* no permite que se ejecuten instrucciones de movimiento seguidas, ya que, incluso si se pudiese prescindir del comando *WaitRob*, entre una instrucción de movimiento y otra necesariamente se concluye e inicia la comunicación y se recibe y analiza el mensaje que envía Unity.

Por otro lado, se ha detectado que las gafas de realidad virtual interfieren en la interacción entre el mando de realidad virtual y la interfaz. Esto se debe a que SteamVR incluye por defecto una funcionalidad que permite que las gafas de realidad virtual interactúen con elementos de interfaz de Unity. Esto provoca que los elementos de la interfaz cambien de color según estén siendo apuntados por las gafas, es decir, mirados por el usuario. Aunque la interferencia es meramente visual y no influye en el funcionamiento de la aplicación, es un efecto innecesario y molesto para el usuario. Para anular este efecto hay que reprogramar la detección de eventos de forma similar a la que se intentó para poder usar *Graphic Raycast*, por lo que no se ha podido solucionar este problema.

Por último, el robot sitúa la pinza incorrectamente cuando, una vez colocado el *Gizmo* por primera vez, éste es modificado de cierta forma. Esto ocurre aparentemente de manera aleatoria. Sin embargo, si se procede a pesar de haberse creado la acción *GoTo* estando la pinza mal colocada, la acción se realiza correctamente al ser ejecutada.

Capítulo 5 Futuras implementaciones

En primer lugar, sería fundamental completar la aplicación implementando los algoritmos necesarios para que el usuario no pueda avanzar en los procesos que requieren ciertas condiciones sin haberlas cumplido. Esto ocurre por ejemplo en la creación de *GoTos,* ya que se puede avanzar desde el menú *GoToMenu* sin haber indicado ningún punto en el espacio con el que crear el *GoTo* o sin haber recibido el visto bueno de RobotStudio.

Respecto al problema de la generación de trayectorias fluidas, una posible solución podría consistir en lo siguiente: en lugar de crear *Actions* tras comprobar su viabilidad, se podría enviar una cadena de caracteres que representase la *Action* en cuestión a RobotStudio. RobotStudio podría concatenar las cadenas de caracteres separadas en renglones hasta tener una representación en caracteres del programa que el usuario pretende crear. La ejecución del programa se podría realizar en un bucle *FOR* utilizando como límite el número de líneas en el programa y que ejecutaría distintas instrucciones de RobotStudio en función de los caracteres de cada línea. De esta manera sí se estarían ejecutando las instrucciones de movimiento de forma consecutiva. En caso de llevarse a cabo este cambio, habría que prescindir por completo del sistema de programación mediante *Actions*, pero se podría aprovechar la mayor parte de la aplicación.

En el caso, de la interferencia visual producida por las gafas de realidad virtual en la interfaz, reprogramar la detección de eventos para usar *Graphic Raycast* lo solucionaría y permitiría a la aplicación manejar la interacción con la interfaz de forma mucho más eficiente al mismo tiempo.

5.1 Conclusión

Dados los problemas que presenta y el trabajo que necesita la aplicación para convertirse en un producto completo, considero altamente viable la posibilidad de que la aplicación se pueda terminar en un proyecto de las mismas características que el desarrollado en este documento. Aunque por el momento parece inviable imitar a la perfección las funcionalidades del *FlexPendant*, la aplicación podría adquirir sus características más importantes.

Bibliografía

[1] Holubek, R., Roman, R. y Delgado Sobrino, D. R. (octubre de 2018). *Using Virtual Reality as a Support Tool for the Offline Robot Programming.* Slovak University of Technology in Bratislava. Obtenido de (última consulta 27 de agosto de 2019): *https://www.researchgate.net/publication/327905308_Using_Virtual_Reality_as_a_Support_Tool_for_the_Offline_Robot_Programming*

[2] RobotWorx. *ABB Offers VR Integration for Robot Programming.* Obtenido de (última consulta 27 de agosto de 2019): *https://www.robots.com/blogs/abb-offers-vr-integration-for-robot-programming*

[3] Gordon, R. (2017, 11 de octubre). *Teleoperating robots with virtual reality.* CSAIL, MIT. Obtenido de (última consulta 27 de agosto de 2019): *http://news.mit.edu/2017/mit-csail-new-system-teleoperating-robots-virtual-reality-1009*

[4] Álvarez Vereterra, C. (2018). *Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI.* Trabajo Fin de Máster, ICAI, Universidad Pontificia de Comillas.

[5] Packt Publishing (2018). *Building your First VR Experience with Unity*. En (última consulta 27 de agosto de 2019): *https://www.udemy.com/course/building-your-first-vr-experience-with-unity/*

[6] Unity. *Unity Scripting Reference*. En (última consulta 27 de agosto de 2019): *https://docs.unity3d.com/ScriptReference/*

[7] Unity. *Unity User Manual (2019.2).* En (última consulta 27 de agosto de 2019): *https://docs.unity3d.com/Manual/index.html*

[8] Martín Castillo, J. C. *Curso Robotstudio*. Videos didácticos en (última consulta 27 de agosto de 2019): *https://www.youtube.com/watch?v=oDnHaVMZjp4&list=PLVdvHpsfqw1bX194j7Slup6ri5se2668p*

[9] ABB. *Introduction to RAPID*. Obtenido de (última consulta 27 de agosto de 2019): *http://www.oamk.fi/~eeroko/Opetus/Tuotantoautomaatio/Robotiikka/Introduction_to_RAPID_3HAC029364-001_rev-_en.pdf*

[10] ABB. *Technical reference manual*. Obtenido de (última consulta 27 de agosto de 2019): *https://library.e.abb.com/public/688894b98123f87bc1257cc50044e809/Technical%20reference%20manual_RAPID_3HAC16581-1_revJ_en.pdf*

[11] VR with Andrew. *Vive Pointer (SteamVR 2.2).* Vídeos didácticos en (última consulta 27 de agosto de 2019): *https://www.youtube.com/watch?v=3mRI1hu9Y3w&list=PLmc6GPFDyfw85CcfwbB7ptNVJn5BSBaXz*

DOCUMENTO Nº2, PRESUPUESTO

SUMAS PARCIALES

En este apartado se exponen las cantidades y precios unitarios de todos los recursos requeridos en la realización del proyecto. Todos los costes son aproximados.

Equipo	Cantidad (uds.)	Coste unitario (€/ud.)	Coste total (€)
Ordenador compatible con realidad virtual	1	1.500	1.500
Oculus Rift	1	500	500
Ordenador personal	1	570	570
		Total:	2.570

Programas	Cantidad (uds.)	Coste unitario (€/ud.)	Coste total (€)
Unity	1	Licencia gratuita	0
RobotStudio	1	1.500	1.500
SolidEdge	1	Licencia gratuita	0
SketchUp	1	Licencia gratuita	0
		Total:	1.500

	Cantidad (horas)	Coste unitario (€/h)	Coste total (€)
Trabajo	300	20	6.000

PRESUPUESTO GENERAL

Por lo tanto, el presupuesto general es de **10.070 €.**

ANEXO I, CONCEPTOS DE BASE DE UNITY

MonoBehaviour

Clase base de la cual derivan todos los códigos de Unity.

Acceso de variables y funciones

El tipo de acceso de las variables y las funciones determina desde qué códigos o elementos pueden ser utilizadas:

- Público (*public*). Acceso no limitado.
- Protegido (*protected*). Acceso interno y desde clases que derivadas.
- Privado (*private*). Acceso propio.

Ventana Inspector

Ventana del entorno de Unity que muestra la información principal del objeto presente en el proyecto seleccionado y todos sus componentes asociados.

Relación padre-hijo de objetos

Cuando un objeto es hijo de otro, es solidario a él, compartiendo sus coordenadas absolutas cartesianas y ángulos de rotación y su escala.

Jerarquía de objetos

Ventana del entorno de Unity que muestra todos los objetos presentes en el proyecto ordenados según sus relaciones de parentesco.

Awake

Función propia de *MonoBehaviours* llamada antes de actualizar el primer fotograma.

Start

Función propia de *MonoBehaviours* llamada al actualizar el primer fotograma.

Update

Función propia de *MonoBehaviours* llamada al actualizar cada fotograma.

Translate

Traslada al objeto desde el que se llame en la dirección y distancia (módulo) que conformen el vector que se le dé por argumento. Se puede indicar si las coordenadas son absolutas o propias.

Rotate

Rota al objeto desde el que se llame según el vector de ángulos que se le dé por argumento. Se puede indicar si los ángulos son absolutos o propios.

Collider

Cuerpos invisibles que se asocian a objetos y pueden detectar colisiones o la presencia de otros objetos en su interior si son de tipo *Trigger*, en cuyo caso además no influyen en la simulación física.

Rigidbody

Componente que otorga a los objetos la condición de cuerpos rígidos, haciendo que participen en la simulación física.

null

Objeto vacío propio de Unity.

Text

Objeto de interfaz que muestra un texto cuyo valor alberga una variable que guarda como cadena de caracteres que se llama *text*.

ANEXO II, CONCEPTOS DE BASE DE ROBOTSTUDIO

speeddata

Información utilizada para determinar la velocidad de movimiento de los ejes del robot.

zoneddata

Información utilizada para determinar a cuánta proximidad debe encontrarse el robot de la posición en cuestión antes de iniciar la siguiente instrucción de movimiento.

robtarget

Información que representa la posición del robot.

jointtarget

Información que representa la posición de los ejes del robot.

TCP

Tool Center Point, el punto de una herramienta sobre el que el actúa.

ANEXO III, CÓDIGO FUENTE

Código *User*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Valve.VR.InteractionSystem;

// Author: Fco Javier Domínguez Sánchez-Girón

public class User : MonoBehaviour
{
    public Hand leftHand;
    public Hand rightHand;
    public GameObject Menu; // menu interface object
    public Camera cam; // VR camera
    public GameObject user; // user object
    public MenuPointer pointer;

    private GameObject camObj; // camera object
    private Transform userTransform;
    private bool menuIsActive = false; // indicates whether the menu is active (true) or not (false)
    private float speedRot = 0.1f; // speed of forced user rotation
    private float speedMov = 0.003f; // speed of forced user translation
    private Vector3 initPos; // initital position
    private Quaternion initRot; // initial rotation
    private MenuManager menuMan;

    private void Awake() // called before first frame update
    {
        initPos = this.gameObject.transform.position;
        initRot = this.gameObject.transform.rotation;
        camObj = cam.gameObject;
        menuMan = Menu.GetComponent<MenuManager>();
    }

    private void Update() // Update is called once per frame
    {
        ManageMenu();
        Move();
        Rotate();
        if (user.transform.position.y != -0.5f) // -0.5 as position in y axis has been empirically
determined to be most comfortable for using the application
        {
            user.transform.position = new Vector3(user.transform.position.x, -0.5f,
user.transform.position.z);
        }
    }

    private void Move() // move the user according to controller input
    {
        Vector3 speedF = cam.transform.forward * speedMov *
leftHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1]; // left controller's vertical
axis
        Vector3 speedR = cam.transform.right * speedMov *
leftHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[0]; // left controller's horizontal
axis
        transform.Translate(speedF + speedR,Space.World);
    }

    private void Rotate() // rotate the user according to controller input
    {
        float yaw = speedRot * rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[0];
// right controller's horizontal axis
        camObj.transform.Rotate(new Vector3(0, yaw, 0),Space.Self);
    }

    public void Go0() // teleport user to initial position and with same rotation values
    {
        transform.position = initPos;
        transform.rotation = initRot;
```

```
        menuMan.ManageMenus(false, true); // deactivate the interface visibility and reset the menu
visibility order
    }

    public void UserScale() // possible function to make the user smaller or bigger to increase or
decrease his/her precision when setting points for the robot
    {
        //
    }

    public void ManageMenu()  // activate or deactivate the menu interface according to grip button
input
    {
        userTransform = camObj.transform; // the position and rotation of the user (the VR camera) are
updated
        if (rightHand == null) // make sure the right controller has been correctly identified
        {
            rightHand = this.transform.GetChild(0).GetChild(2).GetComponent<Hand>();
        }
        else
        {
            if (rightHand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Grip))
            {
                menuIsActive = !menuIsActive;
                if (menuIsActive) // if the menu is becoming active, its transform is set so that it
appears where the user is looking at and facing the user
                {
                    Menu.transform.position = userTransform.position + userTransform.forward * 3.5f;
                    Menu.transform.rotation = userTransform.rotation;
                }
                pointer.NotVisible(); // tell the MenuPointer (see script)
                menuMan.ManageMenus(menuIsActive, false); // tell the MenuManager that its interface
visibility has changed
            }
        }
    }
}
```

Código *IRB120*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// the original author of this code is Carlos Álvarez Vereterra
// Fco Javier Domínguez Sánchez-Girón has removed certain parts or added the elements marked with
[FJD] in their comment

public class IRB120 : MonoBehaviour {

    public float[] axisAngles = new float[6]; // the virtual robot's joint angles [FJD: replaced 6
variables with a size 6 array]
    public int speed = 5; // angular movement speed

    // robot axes
    private Transform tAxis1;
    private Transform tAxis2;
    private Transform tAxis3;
    private Transform tAxis4;
    private Transform tAxis5;
    private Transform tAxis6;

    // virtual tool indicators [FJD]
    public bool tool = false; // tool state: true = open gripper, false = closed gripper
    public int actTool = 0; // tool readiness indicator: 0 = not ready, 1 = ready

    void Start()
    {
        // get each robot axis
        tAxis1 = GameObject.Find("Axis1").GetComponent<Transform>();
        tAxis2 = GameObject.Find("Axis2").GetComponent<Transform>();
        tAxis3 = GameObject.Find("Axis3").GetComponent<Transform>();
        tAxis4 = GameObject.Find("Axis4").GetComponent<Transform>();
        tAxis5 = GameObject.Find("Axis5").GetComponent<Transform>();
        tAxis6 = GameObject.Find("Axis6").GetComponent<Transform>();
        axisAngles[0] = 0;
        axisAngles[1] = 0;
        axisAngles[2] = 0;
        axisAngles[3] = 0;
        axisAngles[4] = 0;
        axisAngles[5] = 0;
    }

    void Update() // Update is called once per frame
    {
        // linear interpolate to angle
        tAxis1.localRotation = Quaternion.Lerp(tAxis1.localRotation, Quaternion.Euler(0, -
axisAngles[0], 0), 5 * Time.deltaTime);
        tAxis2.localRotation = Quaternion.Lerp(tAxis2.localRotation, Quaternion.Euler(0, 0, -
axisAngles[1]), 5 * Time.deltaTime);
        tAxis3.localRotation = Quaternion.Lerp(tAxis3.localRotation, Quaternion.Euler(0, 0, -
axisAngles[2]), 5 * Time.deltaTime);
        tAxis4.localRotation = Quaternion.Lerp(tAxis4.localRotation, Quaternion.Euler(-axisAngles[3],
0, 0), 5 * Time.deltaTime);
        tAxis5.localRotation = Quaternion.Lerp(tAxis5.localRotation, Quaternion.Euler(0, 0, -
axisAngles[4]), 5 * Time.deltaTime);
        tAxis6.localRotation = Quaternion.Lerp(tAxis6.localRotation, Quaternion.Euler(-axisAngles[5],
0, 0), 5 * Time.deltaTime);
    }
}
```

Código *ToolFinger*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Author: Fco Javier Domínguez Sánchez-Girón

public class ToolFinger : MonoBehaviour
{
    public IRB120 robot; // the virtual robot the gripper tool is attached to
    public float x_open; // the position in local x axis in which the gripper fingers are considered
to be open
    public float x_closed = 0; // the position in local x axis in which the gripper fingers are
considered to be closed
    public const float speed = 1; // the gripper fingers' speed of movement

    private float vel_open = 0; // the speed of movement for each finger
    private float tmov; // the used speed for each finger
    private Rigidbody rb; // the finger's Rigidbody component
    private bool stat_act = false; // current tool or finger state
    private bool stat_ant = false; // last tool or finger state
    private bool Right = false; // boolean right finger identifier
    private bool Left = false; // boolean left finger identifier
    private GameObject movingObject = null; // grabbed object
    private int rbant = 0; // last robot's tool readiness indicator state
    private bool already = false; // indicates if the tool has been required to reach the state in
which it already is (true)

    // activate tool = open gripper
    // deactivate tool = close gripper

    private void Awake() // Awake is called before Start
    {
        rb = GetComponent<Rigidbody>();
    }

    private void Start() // Start is called before the first frame update
    {
        // since the local x axis grows in the direction of the left finger:
        if (x_open > x_closed)
        {
            Left = true; // left finger is identified
            vel_open = speed; // left finger opens with positive speed, closes with negative
        } else if (x_open < x_closed)
        {
            Right = true; // right finger is identified
            vel_open = -speed; // right finger opens with negative speed, closes with positive
        }
    }

    private void Update() // Update is called once per frame
    {
        stat_act = robot.tool; // the finger state is updated with virtual tool state
        if (stat_act ^ stat_ant) // if the finger state has changed:
        {
            if (!stat_act) // if the finger state has just changed to false (close)
            {
                tmov = -vel_open; // closing
            }
            else // if the finger state has just changed to true (open)
            {
                tmov = vel_open; // opening
                if (movingObject != null && movingObject.transform.parent != null) // if an object had
been grabbed:
                {
                    movingObject.GetComponent<Rigidbody>().isKinematic = false; // make the object
affected by external forces again
                    movingObject.transform.parent = null; // make the object's Transform independent
of the tool's
```

```
                }
            }
        }
        else if (rbant != robot.actTool && robot.actTool == 0) // if the finger state has not changed,
the robot's tool readiness indicator has changed from 1 to 0 (means the tool has been required to act
in some way, see IRB120):
        {
            already = true; // the corresponding indicator is set to true
            Debug.Log("[TF] Tool already ready");
        }
        rbant = robot.actTool; // last tool readiness state is updated
        stat_ant = stat_act; // last tool state is updated
        StopMoving(); // stops the finger if necessary
        MoveFinger(); // moves the finger
    }

    private void OnTriggerEnter(Collider other) // gets called whenever an object enters the finger
trigger Colliders (placed on its inner surfaces)
    {
        if (!stat_act) // if the gripper is closing:
        {
            Debug.Log("[TF] Object grabbed");
            tmov = 0; // stop moving the finger
            other.transform.SetParent(this.transform.parent); // make the grabbed object move and
rotate with the tool
            movingObject = other.gameObject; // set movingObject to grabbed object
            movingObject.GetComponent<Rigidbody>().isKinematic = true; // make movingObject not be
affected by external forces (like the gripper) to ensure it stays in place
            robot.actTool = 1; // tool is ready
        }
    }

    private void StopMoving() // stops the movement of the finger if it's reached a stoping state
    {
        if ((Left && transform.localPosition.x <= x_closed || // if the finger (left or right) has
reached x_closed and...
            Right && transform.localPosition.x >= x_closed) &&
            (tmov == -vel_open || already && !stat_act)) // ...and it was closing or already there and
closed:
        {
            tmov = 0; // stop moving
            robot.actTool = 1; // tool is ready
            transform.localPosition = new Vector3(x_closed, transform.localPosition.y,
transform.localPosition.z); // finger is positioned in x_closed to ensure proper functioning
            Debug.Log("[TF] Closed tool");
            already = false; // already indicator is reset
        }
        else if ((Left && transform.localPosition.x >= x_open || // if the finger (left or right) has
reached x_open and...
            Right && transform.localPosition.x <= x_open) &&
            (tmov == vel_open || already && stat_act)) // ...and it was opening or already there and
openened:
        {
            tmov = 0; // stop moving
            robot.actTool = 1; // tool is ready
            transform.localPosition = new Vector3(x_open, transform.localPosition.y,
transform.localPosition.z); // finger is positioned in x_open to ensure proper functioning
            Debug.Log("[TF] Opened tool");
            already = false; // already indicator is reset
        }
    }

    private void MoveFinger() // move the finger
    {
        this.gameObject.transform.Translate(tmov * Vector3.right * Time.deltaTime, Space.Self); //
teleports the finger each frame at a speed of tmov m/s in its positive x direction
    }
}
```

Código *MenuPointer*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;

// Author: Fco Javier Domínguez Sánchez-Girón

public class MenuPointer : MonoBehaviour
{
    public GameObject dot; // dot prefab to display point in interface to which user is pointing

    private GameObject curr = null; // current pointed object
    private GameObject last = null; // last pointed object

    void Update() // Update is called once per frame
    {
        RaycastHit hit;
        Physics.Raycast(transform.position, transform.forward, out hit, 100, 1 << 5); // a Ray is cast
from the object's position in it's forward direction, info is stored in hit and 5 is the UI's layer
        if (hit.collider != null) // if something is being pointed
        {
            curr = hit.collider.gameObject; // update current object
            if (last == null) // if nothing was being pointed just before, only execute OnPointerEnter
of current object
            {
                ExecuteEvents.Execute(curr, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerEnterHandler);
            }
            else if (curr != last) // if something different was being pointed just before, also
execute OnPointerExit of last object
            {
                ExecuteEvents.Execute(last, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerExitHandler);
                ExecuteEvents.Execute(curr, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerEnterHandler);
            }
            if (!dot.activeSelf) // if it's the first consecutive time the object is being pointed,
make dot visible
            {
                dot.SetActive(true);
            }
            dot.transform.position = this.transform.position + hit.distance * this.transform.forward;
// dot is always where the Ray collides with the object
        }
        else // if nothing's being pointed
        {
            curr = null;
            if (curr != last) // if something was being pointed just before, execute OnPointerExit of
last object
            {
                ExecuteEvents.Execute(last, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerExitHandler);
            }
            if (dot.activeSelf) // if dot was visible, make dot invisible
            {
                dot.SetActive(false);
            }
        }
        last = curr; // update last object
    }

    public void NotVisible()
    {
        if (curr != null)
        {
            ExecuteEvents.Execute(curr, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerExitHandler);
            curr = null;
```

```
        }
    }
}
```

Código *UI_Element*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;
using Valve.VR.InteractionSystem;
using System;

// Author: Fco Javier Domínguez Sánchez-Girón

public class UI_Element : MonoBehaviour, IPointerEnterHandler, IPointerExitHandler,
IPointerDownHandler, IPointerUpHandler
{
    public Color32 normalColor = Color.grey;
    public Color32 highlightColor = Color.white;
    public Color32 clickedColor = Color.cyan;
    public bool canToggle = false; // allows the button to stay pressed when clicked until next click
    public bool pointed = false; // indicates whether the button is being pointed by the user
    public bool clicked = false; // indicates whether the button is pressed

    private Image image = null; // the visible image of the button
    private BoxCollider coll; // a Collider component for the interaction with buttons to be done
through PhysicsRaycast
    private RectTransform rTrans; // rectangle transform of the button
    private MenuManager menu; // MenuManager attached to this object's root parent (the interface
object)
    private Hand hand; // the controller that interacts with the menu

    private void Awake() // called before first frame update
    {
        image = this.gameObject.GetComponent<Image>();
        menu = this.transform.root.GetComponent<MenuManager>();
        hand = menu.rightHand;
        if (image != null)
        {
            image.color = normalColor;
        }
    }

    private void Update() // called every frame update
    {
        if (pointed && hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Trigger)) // if
element is pointed and trigger button is pressed, a click is processed
        {
            ExecutePress();
        }
        if (clicked && (hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Trigger))) // if
clicked and trigger button is released, a release is processed
        {
            ExecuteRelease();
        }
    }

    public void OnPointerEnter(PointerEventData eventData) // called when the button gets pointed
    {
        if (image != null && !clicked)
        {
            image.color = highlightColor;
        }
        pointed = true;
    }

    public void OnPointerExit(PointerEventData eventData) // called when the button stops being
pointed
    {
        if (image != null && !clicked)
        {
            image.color = normalColor;
```

```
        }
        pointed = false;
    }

    public void OnPointerDown(PointerEventData eventData) // called when the button gets clicked
    {
        if (!clicked)
        {
            if (image != null)
            {
                image.color = clickedColor;
            }
            clicked = true;
        }
        else // if the button can be toggled, its release is processed when clicked again
        {
            if (image != null)
            {
                image.color = highlightColor;
            }
            clicked = false;
        }
    }

    public void OnPointerUp(PointerEventData eventData) // called when the button gets unclicked
    {
        if (!canToggle) // if the button can't be toggled, its release is processed when it gets
unclicked
        {
            if (image != null)
            {
                if (pointed)
                {
                    image.color = highlightColor;
                }
                else
                {
                    image.color = normalColor;
                }
            }
            clicked = false;
        }
    }

    private void OnEnable() // called when the button's gameobject state changes from inactive to
active (visible)
    {
        setCollider();
    }

    private void OnValidate() // called after the button is modified in Unity
    {
        setCollider();
    }

    private void setCollider() // sets the right size for the button's collider
    {
        rTrans = GetComponent<RectTransform>();
        coll = GetComponent<BoxCollider>();
        if (coll == null)
        {
            coll = gameObject.AddComponent<BoxCollider>();
        }
        if (!this.gameObject.name.Contains("Toggle"))
        {
            if (this.gameObject.name.Contains("Item"))
            {
                coll.size = new Vector3(320, 80, 0);
            }
            else
            {
```

```
                coll.size = rTrans.sizeDelta;
            }
        }
    }

    public void ExecutePress()
    {
        ExecuteEvents.Execute(this.transform.gameObject, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerClickHandler); // calls the default OnPointerClick handler for the events created
in the Inspector window to get executed
        ExecuteEvents.Execute(this.transform.gameObject, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerDownHandler); // calls the custom OnPointerDown handler
    }

    private void ExecuteRelease()
    {
        ExecuteEvents.Execute(this.transform.gameObject, new PointerEventData(EventSystem.current),
ExecuteEvents.pointerUpHandler); // calls the custom OnPointerUp handler
    }
}
```

Código *MenuManager*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Valve.VR.InteractionSystem;

// Author: Fco Javier Domínguez Sánchez-Girón

public class MenuManager : MonoBehaviour
{
    public ProgramManager prgMan; // the menu's ProgramManager
    public Hand rightHand;
    public GameObject Menus; // interface object
    public GameObject timeObject; // time-displaying object
    public float minHeight; // minimum height the menu can stay completely visible at

    private List<GameObject> menuChildren = new List<GameObject>(); // the children of the interface
object are the different menus
    private int numChildren; // number of menus
    private GameObject actualMenu; // current active menu
    private GameObject lastMenu; // current active menu's previous menu
    private List<int> heritLine = new List<int>(); // list in which the position inside menuChildren
of the corresponding previous menu to each menu is stored
                                   // each of its positions refer to the positions of menuChildren
    private void Awake()
    {
        numChildren = Menus.transform.childCount;
        for (int i = 0; i < numChildren; i++) // initialisation for menuChildren and heritLine
        {
            menuChildren.Add(Menus.transform.GetChild(i).gameObject);
            heritLine[i] = -1;
        }
        actualMenu = menuChildren[0];
    }

    private void Update() // Update is called once per frame
    {
        if (rightHand.controller.GetPressDown(SteamVR_Controller.ButtonMask.ApplicationMenu)) // if
button ApplicationMenu (B in Oculus Touch) is pressed, the user goes back in the menus
        {
            if (!prgMan.angleChange) // the same button is used for another functionality (see
ProgramManager)
            {
                GoBack();
            }
        }
        KeepVisibleHeight();
        PreventZRot();
    }

    public void ManageMenus(bool m, bool reset) // sets the interface visibility to boolean m
    {
        if (reset & m)  // if boolean reset is true, resets the menu visibility order
        {
            menuChildren[0].SetActive(m); // the first menu is activated to be visible when the
interface visibility is re-activated
            for (int i = 1; i < numChildren; i++)
            {
                menuChildren[i].SetActive(!m); // deactivates the rest
            }
        }
        Menus.SetActive(m);
    }

    public void UpdateActual(GameObject menu) // updates the actualMenu and lastMenu, called when the
user goes from one menu to another
    {
        lastMenu = actualMenu;
        actualMenu = menu;
```

```
        int i;
        i = actualMenu.transform.GetSiblingIndex();
        if (heritLine[i] < 0) // the information is stored in heritLine for later use, each new
actualMenu gets its corresponding previous (lastMenu) stored
        {                       // the information is only stored once thanks to heritLine having been
initialised with -1 in every position
            heritLine[i] = lastMenu.transform.GetSiblingIndex();
        }
    }

    private void FindPrevLast(GameObject newActual) // find the current menu's previous menu based on
information previously stored
    {
        lastMenu = (menuChildren[heritLine[newActual.transform.GetSiblingIndex()]]);
    }

    public void GoBack() // deactivates the current active menu and activates its corresponding
previous menu
    {
        if (actualMenu.transform.GetSiblingIndex() != 0 && Menus.activeSelf) // if the current menu is
not the first one and the interface is visible:
        {
            actualMenu.SetActive(false);
            lastMenu.SetActive(true);
            actualMenu = lastMenu;
            FindPrevLast(actualMenu); // the new previous menu is set
            prgMan.notCreatingGoTo(); // tell the ProgramManager that the user is not creating a GoTo
Action in case he/she was
        }
    }

    public void PlaceTime() // instantiates the time-displaying object
    {
        GameObject find = GameObject.FindWithTag("TimeDisplay"); // if another instance of the object
is found somewhere else, it gets deleted
        if (find != null)
        {
            Destroy(find);
        }
        Instantiate(timeObject, transform.position, transform.rotation); // the objects gets created
where the menu is currently at and with its rotation
        ManageMenus(false, true); // deactivate the menu interface visibility and reset the menu
visibility order
    }

    public void ExitApplication() // shuts the application
    {
        Application.Quit();
    }

    private void KeepVisibleHeight() // keep the menu at a visible height
    {
        if (transform.position.y < minHeight) // if the menu is placed below the minimum height, gets
instantly placed at the minimum height
        {
            this.gameObject.transform.position = new Vector3(transform.position.x, minHeight,
transform.position.z);
        }
    }
    private void PreventZRot() // prevent the menu from rotating perpendicularly with respect to the
user's vision
    {
        if (transform.localEulerAngles.z != 0) // if the menu is placed with a certain value of the
aforementioned rotation, it gets set to 0
        {
            transform.localEulerAngles = new Vector3(transform.localEulerAngles.x,
transform.localEulerAngles.y, 0);
        }
    }
}
```

Código *TimeTextDisp*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System;
using System.Globalization;

// Author: Fco Javier Domínguez Sánchez-Girón

public class TimeTextDisp : MonoBehaviour
{
    public Text dateTime; // Text object for displaying current date and time
    public Text timeRunning; // Text object for displaying the time the application has been used

    private void Update() // Update is called once per frame
    {
        GetDate();
        GetTime();
    }

    private void GetDate() // sets dateTime string component to current date and time
    {
        DateTime localDate = DateTime.Now;
        CultureInfo culture = new CultureInfo("es-ES");
        dateTime.text = localDate.ToString(culture);
    }

    private void GetTime() // sets timeRunning string component to the time passed since the
application started running
    {
        float timePassed = Time.time;
        int seconds = (int)(timePassed % 60);
        int minutes = (int)(timePassed / 60) % 60;
        int hours = (int)(timePassed / 3600) % 60;
        timeRunning.text = "Time using application:" + System.Environment.NewLine +
string.Format("{0:00}:{1:00}:{2:00}", hours, minutes, seconds);
    }
}
```

Código *Action*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Author: Fco Javier Domínguez Sánchez-Girón

public abstract class Action // abstract class that represents an action done by a robot
{
    public abstract string Print(); // returns string with relevant information about the action
    public abstract int Execute(int flanc, IRB120 obj, Communications comms); // Execute method
executes tha action and tells the Program to continue by returning 1 and not 0 (see Program code) when
the action has been executed
                                // flanc variable determines if a certain action is being executed
for the first consecutive time (1) or not (0)
}

public class Wait : Action // Action child class, a wait of a certain time in seconds
{
    private float time; // time to be waited
    private static float t = 999; // auxiliary variable that stores the time value when

    public Wait(float tt) // Wait constructor
    {
        time = tt;
    }

    public override string Print() // override of Print method
    {
        return("Wait " + time + " seconds.");
    }

    public override int Execute(int flanc, IRB120 robot, Communications comms) // override of Execute
method
    {
        if (flanc != 0) // if it's the first consecutive time the action is being executed, the time
is stored in t
        {
            t = Time.time;
            return 0;
        }
        else if (Time.time - t < time) // t is used as the reference time, if the time passed is
smaller than parameter time, Execute returns 0
        {
            return 0;
        }
        else // if the time passed is gretaer than or equal to time, Execute returns 1 so the Program
can continue executing
        {
            Debug.Log("[W] Waited " + time + " seconds.");
            return 1;
        }
    }
}

public struct pointData // data required to define a point to reach by the robot
{
    public string target; // contains the information required to define a robtarget variable in
RobotStudio
    public int number; // the ordinal number of the point inside the Program
    public string mode; // "J" for MoveJ (non-linear) and "L" for MoveL (linear), RobotStudio
instructions for moving the robot to reach a point
    public string velocity; // 1, 2 or 3, each number corresponding to the low, medium and high
speeddata constants used in RobotStudio
    public string precision; // 1, 2 or 3, each number corresponding to the low, medium and high
zonedata constants used in RobotStudio
}

public class GoTo : Action // Action child class, the action of going to a point
```

```
{
    private pointData pd; // data of the point to reach

    public GoTo(pointData pd1) // GoTo constructor
    {
        pd = pd1;
    }

    public override string Print() // override of Print method
    {
       return("Go to point " + pd.number + " using " + "Move" + pd.mode + " " + pd.velocity + " " +
pd.precision);
    }

    public override int Execute(int flanc, IRB120 robot, Communications comms) // override of Execute
method
    {
        if (flanc != 0) // if it's the first consecutive time the action is being executed,
RobotStudio is told to execute a Move instruction
        {
            SendGoTo(pd, comms);
        }
        else if (comms.executed == 1) // if Communications indicates the action has been executed, a 1
is returned
        {
            Debug.Log("[GT] Gone to point " + pd.number);
            return 1;
        }
        return 0; // 0 is returned in every other case
    }
    private void SendGoTo(pointData pd1, Communications comms)
    {
        pd1.target = pd1.target.Replace("XC", string.Empty); // the message sent is coherent with what
RobotStudio expects to receive: only "X" at the beginning means execution of a movement to a point
        comms.message = "X" + pd1.velocity + pd1.precision + pd1.mode + pd1.target;
        comms.executed = 0; // Communications' execution indicator is set to 0, will indicate
execution is finished with value 1
        comms.flag = true; // Communications' flag is activated for the communication to be done
    }
}

public class MoveAxis : Action // Action child class, the action of setting the robot's angles to
certain values
{
    private float[] angles = new float[6]; // robot angles' values
    private string vel; // velocity of the movement (certain speeddata constants in RobotStudio)
    private string precision; // precision of the movement (certain zonedata constants in RobotStudio)

    public MoveAxis(float[] ang, string v, string prec) // MoveAxis constructor
    {
        for (int i = 0; i < 6; i++)
        {
            angles[i] = ang[i];
        }
        vel = v;
        precision = prec;
    }

    public override string Print() // override of Print method
    {
        return ("Set robot angles to: " + angles[0] + ", " + angles[1] + ", " + angles[2] + ", " +
angles[3] + ", " + angles[4] + " and " + angles[5]);
    }

    public override int Execute(int flanc, IRB120 robot, Communications comms) // override of Execute
method
    {
        if (flanc != 0) // if it's the first consecutive time the action is being executed,
RobotStudio is told to execute a MoveAbsJ instruction (reach a certain angle combination)
        {
            SendMovAx(angles, comms, robot);
```

```
        }
        else if (comms.executed == 1) // if Communications indicates the action has been executed, a 1
is returned
        {
            Debug.Log("[MA] Executed axis move");
            return 1;
        }
        return 0; // 0 is returned in every other case
    }
    private void SendMovAx(float[] angles, Communications comms, IRB120 robot)
    {
        // the message sent is coherent with what RobotStudio expects to receive: only "A" at the
beginning means execution of a movement to reach a certain angle combination
        comms.message = "A" + vel + precision + "[" + angles[0] + "," + angles[1] + "," + angles[2] +
"," + angles[3] + "," + angles[4] + "," + angles[5] + "]";
        comms.executed = 0; // Communications' execution indicator is set to 0, will indicate
execution is finished with value 1
        comms.flag = true; // Communications' flag is activated for the communication to be done
    }
}

public class ActDeact : Action // Action child class, the action of activating or deactivating the
robot tool, in this case, a 2-finger gripper
{
    private bool tool; // true for activating and false for deactivating

    public ActDeact(bool ad) // ActDeact constructor
    {
        tool = ad;
    }

    public override string Print() // override of Print method
    {
        if (tool)
        {
            return("Activate tool.");
        }
        else
        {
            return("Deactivate tool.");
        }
    }

    public override int Execute(int flanc, IRB120 robot, Communications comms) // override of Execute
method
    {
        if (flanc != 0) // if it's the first consecutive time the action is being executed,
RobotStudio is told to activate or deactivate the tool's signals
        {
            // the message sent is coherent with what RobotStudio expects to receive:
            comms.message = "Y"; // "Y" means tool-related instruction
            if (tool)
            {
                comms.message += "T"; // "T" means activating (opening) the tool
            }
            else
            {
                comms.message += "F"; // "F" means deactivating (closing) the tool
            }
            robot.actTool = 1; // virtual robot's tool readiness indicator is set to 1 (virtual tool
is ready in its current state) to ensure the process will be done correctly
            comms.executed = 0; // Communications' execution indicator is set to 0, will indicate
execution is finished with value 1
            comms.flag = true; // Communications' flag is activated for the communication to be done
            return 0;
        }
        else
        {
            if (comms.executed == 1) // if Communications indicates the real action has started being
executed, the indicator is put in an auxiliary state and the virtual tool is told to act
            {
```

```
                comms.executed = 2; // auxiliary state
                robot.actTool = 0; // the tool is not ready
                robot.tool = tool; // the virtual robot is told the new tool state
            }
            else if (comms.executed == 2) // if Communications' execution indicator is in auxiliary
state:
            {
                if (robot.actTool == 1) // if the tool is ready
                {
                    if (tool)
                    {
                        Debug.Log("[AD] Activated tool.");
                    }
                    else
                    {
                        Debug.Log("[AD] Deactivated tool.");
                    }
                    return 1; // 1 is returned for the Program to continue
                }
            }
            return 0; // 0 is returned is every other case
        }
    }
}
```

Código *Program*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Author: Fco Javier Domínguez Sánchez-Girón

public class Program : MonoBehaviour
{
    public List<Action> actions = new List<Action>(); // Action list to be executed, the content of
the Program
    public bool exe = false; // boolean that determines if the Program must run (true) or not (false)
    public bool loop = false; // boolean that determines if the Program must run continuously (true)
or just once (false)
    public int ngots = 0; // number of GoTo Actions (used by ProgramManager)
    public IRB120 Robot; // virtual robot to which the program applies
    [HideInInspector]
    public Communications comms; // script that handles communications with RobotStudio

    private int iAct = 0; // index for action in the list currently in execution
    private int iAnt = -1; // auxiliar index for detection of changes of value (flancs) in iAct

    void Update() // Update is called once per frame
    {
        Execute();
    }

    void Execute() // executes each action in the order they were added if boolean exe is true
    {
        int inc; // the value the index iAct will be incremented each time an Action is executed
        if (exe)
        {
            inc = actions[iAct].Execute(iAct^iAnt,Robot,comms); // inc will be 0 if the real action
has not been executed yet and 1 if it has
            iAct += inc;                                        // iAct^iAnt (the Action flanc, see
Action) will only be 1 if they have different values: only when iAct's value has changed
            if (inc == 0) // if the program will proceed to the next action (inc = 1) iAnt must stay
the same for the flanc to be sent to the next action correctly
            {
                iAnt = iAct;
            }
            if (iAct == actions.Count) // if last action has been executed:
            {
                iAct = 0; // index is reset
                if (!loop) // if the program was meant to execute just once:
                {
                    exe = false; // stop executing
                    iAnt = -1; // for the next flanc to be sent correctly
                }
            }
        }
    }

    public string Print() // returns a string containing each action and its relevant information
    {
        string progString = ""; // initialise the string
        for (int i = 0; i < actions.Count; i++)
        {
            progString += actions[i].Print() + System.Environment.NewLine; // add each action
        }
        if (progString == "") // if string is empty:
        {
            return ("Program is empty...");
        }
        return progString;
    }
}
```

Código *ProgramManager*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using Valve.VR.InteractionSystem;

// Author: Fco Javier Domínguez Sánchez-Girón

public class ProgramManager : MonoBehaviour
{

    public Communications comms; // the object that handles communication with RobotStudio
    public ViveController vc; // point-setting script
    public IRB120 robot; // the robot with the user is currently working with
    public Hand rightHand; // the controller the interface interacts with
    [HideInInspector]
    public Program prog; // the Program the user is currently working with
    public Text waitText; // displays the time parameter when creating a Wait Action
    public Text progText; // displays the Program content
    public bool angleChange = false; // indicator that the user is modifying an angle value
    // these Texts display each of the robot axis' angle value en degrees
    public Text textAngle1;
    public Text textAngle2;
    public Text textAngle3;
    public Text textAngle4;
    public Text textAngle5;
    public Text textAngle6;

    private bool butact = false; // current button state
    private bool butant = false; // last button state
    private string waitString = ""; // introduced in waitText for wait time display
    private bool tool = false; // tool state
    private bool mode = false; // indicator of GoTo mode (false = non-linear, true = linear)
    private string mod = "J";
    private string vel = "1";
    private string prec = "3";
    private int axis = -1; // the axis the user is currently modifying the angle of
    private float tempAngle = 0; // the reference angle from which the value is changed
    private UI_Element button; // the button that corresponds to the axis being modified

    private void Start() // Start is called before the first frame update
    {
        ConfCollisionDetection();
    }

    private void Update() // Update is called once per frame
    {
        ChangingAngle();
    }

    public void SetProgram(GameObject prgrm) // sets the Program with which the user is currently
working
    {
        prog = prgrm.GetComponent<Program>();
        prog.comms = comms; // sets the Communications object of the Program
    }

    private void AddAction(Action act) // adds an Action to the Program
    {
        prog.actions.Add(act); // the Action is added through list's Add method
    }

    public void RemoveAction(int index) // removes the Action in the position index of the Program's
Action list // not fully implemented yet
    {
        if (prog.actions[index].Print().Contains("point")) // if the Action is GoTo:
        {
            prog.ngots--; // number of GoTos gets decremented by one
```

```
        }
        prog.actions.RemoveAt(index); // the Action is deleted through list's RemoveAt method
        UpdateProgText();
    }

    public void EmptyProgram() // empties the Program's content
    {
        prog.actions.Clear(); // the list of actions is cleared through list's Clear method
        prog.ngots = 0; // the number of GoTo Actions is reset
        UpdateProgText();
    }

    public void AddingAction(string adding) // adds Wait or ActDeact Actions depending on adding
string argument, different buttons give different arguments
    {
        if (adding == "w") // "w" means Wait
        {
            float t;
            t = float.Parse(waitString); // parameter t for new Wait's time is created
            waitString = "";
            waitText.text = "Enter time..."; // string component of Text for displaying wait time is
reset
            Action act = new Wait(t);
            AddAction(act); // new Wait is added
            UpdateProgText();
            Debug.Log("[PM] Added wait of: " + t + " sec.");
        }
        else if (adding == "t") // "t" means ActDeact
        {
            tool = !tool; // the tool state boolean's value is changed
            Action act = new ActDeact(tool);
            AddAction(act); // new ActDeact is added
            UpdateProgText();
            Debug.Log("[PM] Added tool action: " + tool + ".");
        }
    }

    public void ExecuteProgram() // changes the Program's execution boolean state
    {
        prog.exe = !prog.exe;
        if (prog.exe) // if the program is now executing
        {
            Debug.Log("[PM] " + this.gameObject.name + " is executing.");
            Physics.IgnoreLayerCollision(8, 10, false); // collision between Tool and Grababble layers
is enabled
        }
        else
        {
            Debug.Log("[PM] " + this.gameObject.name + " is NOT executing.");
            Physics.IgnoreLayerCollision(8, 10, true); // collision between Tool and Grababble layers
is disabled
        }
    }

    public void creatingGoTo() // changes the settingPoint indicator accordingly for the correspondent
functionalities to be enabled
    {
        if (!vc.settingPoint) vc.settingPoint = true;
    }

    public void notCreatingGoTo() // sets the settingPoint indicator to false for the corresponding
functionalities to be disabled
    {
        if (vc.settingPoint) vc.settingPoint = false;
    }

    public void finishedGoTo() // creates the new GoTo Action and ends the process
    {
        prog.ngots++; // updates number of GoTo Actions in the Program, this number also represents
the ordinal number of the GoTo Action in the program
        pointData pd = new pointData // pointData for the GoTo Action
```

```
        {
            target = comms.savedMessage, // the target is the last message sent to RobotStudio for
reachability checking
            number = prog.ngots,
            mode = mod,
            velocity = vel,
            precision = prec
        };
        Action act = new GoTo(pd); // the new GoTo Action is given its position number and obtained
from Communications
        AddAction(act);
        UpdateProgText();
        notCreatingGoTo(); // ends process
    }

    public void WaitStringAdd(string s) // adds an element to the string component of the Text
displaying the waiting time of the Wait Action being created
    {
        waitString += s;
        waitText.text = waitString;
    }

    public void DeleteWString() // deletes last element in the string component of the Text displaying
the waiting time of the Wait Action being created
    {
        waitString = waitString.Remove(waitString.Length-1);
        waitText.text = waitString;
    }

    public void UpdateMode(bool b)  // updates the mode to be stored in the next GoTo Action created
    {
        mode = b;
        if (mode)
        {
            mod = "L";
        }
        else
        {
            mod = "J";
        }
    }
    public void UpdateVelocity(int i) // updates the velocity to be stored in the next instruction of
movement's Action created
    {
        vel = (i+1).ToString();
    }
    public void UpdatePrecision(int i) // updates the precision to be stored in the next instruction
of movement's Action created
    {
        prec = (i+1).ToString();
    }

    private void ConfCollisionDetection() // disables the collision detection between certain layers
    {
        // layer 8 = Tool: the virtual tool object is in this layer
        // layer 11 = Floor: the whole floor of the application environment is in this layer
        // layer 10 = Grabbable: the objects that are grabbable by the virtual robot are in this layer
        // layer 5 = UI: the user interface is in this layer
        // layer 9 = Structure: the structure of the virtual fatory is in this layer
        // layer 0 = Default: most other objects are in this layer, the player, for example
        Physics.IgnoreLayerCollision(8, 8, true);
        Physics.IgnoreLayerCollision(8, 10, true);
        Physics.IgnoreLayerCollision(8, 5, true);
        Physics.IgnoreLayerCollision(8, 0, true);
        Physics.IgnoreLayerCollision(8, 9, true);
        Physics.IgnoreLayerCollision(11, 5, true);
        Physics.IgnoreLayerCollision(11, 0, true);
        Physics.IgnoreLayerCollision(11, 9, true);
        Physics.IgnoreLayerCollision(10, 0, true);
        Physics.IgnoreLayerCollision(10, 5, true);
    }
```

```
    public void UpdateProgText() // updates the string component of the Text that displays a Program's
content
    {
        progText.text = prog.Print();
    }

    public void ChangeLoop(bool b) // changes the Program loop boolean to value b
    {
        prog.loop = b;
    }

    public void ModAngButton(UI_Element bttn) // allows this script to identify the button pressed
(called from the button)
    {
        button = bttn;
    }
    public void ModAngle(int ax) // allows this script to identify the corresponding axis to the
button pressed (called from the button)
    {
        axis = ax;
    }

    private void ChangingAngle() // allows the user to change tha angles of the robot
    {
        bool cancel = false;
        if (textAngle1.IsActive()) // if any of the texts for the display of the angles is visible by
the user, the values keep updating
        {
            DispAngles();
        }
        if (button != null) // if a button has been identified:
        {
            butact = button.clicked;
            if (butact) // if the button is clicked:
            {
                if (!angleChange) // if the user wasn't already changing an angle value:
                {
                    angleChange = true; // indicator is set to true
                    tempAngle = robot.axisAngles[axis]; // reference angle is initilised with current
value
                }
                else
                {
                    if
(rightHand.controller.GetPressDown(SteamVR_Controller.ButtonMask.ApplicationMenu)) // if
ApplicationMenu button (B in Oculus Touch) is pressed:
                    {
                        robot.axisAngles[axis] = tempAngle; // the angle is reset to its original
value
                        CancelAngleChange(); // the process is cancelled
                        cancel = true;
                    }
                    else if (Mathf.Abs(robot.axisAngles[axis]) < Mathf.Abs(tempAngle) + 180 || //
rotating the virtual robot more than +180 or -180 from the reference angle is redundant and therefore
not allowed
                        robot.axisAngles[axis] >= tempAngle + 180 &&
rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1] < 0 || // if the virtual robot
has been rotated +180 from the reference angle, only negative rotation is allowed
                        robot.axisAngles[axis] <= tempAngle - 180 &&
rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1] > 0) // if the virtual robot has
been rotated -180 from the reference angle, only positive rotation is allowed
                    {
                        robot.axisAngles[axis] +=
rightHand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_Axis0)[1]; // angle value is incremented
or decremented based on the vertical position of the right joystick
                    }
                }
            }
            else if (butact ^ butant && !cancel) // if the button has just been unclicked (clicked a
second time)
            {
```

```
                CheckAxes(); // the current angle configuration's reachability is checked
            }
            butant = butact; // the auxiliary button variable is updated to allow detection of changes
in button's state
        }
    }

    private void DispAngles() // updates the values of the Texts that display the virtual robot's
angles
    {
        textAngle1.text = (robot.axisAngles[0]).ToString() + "º";
        textAngle2.text = (robot.axisAngles[1]).ToString() + "º";
        textAngle3.text = (robot.axisAngles[2]).ToString() + "º";
        textAngle4.text = (robot.axisAngles[3]).ToString() + "º";
        textAngle5.text = (robot.axisAngles[4]).ToString() + "º";
        textAngle6.text = (robot.axisAngles[5]).ToString() + "º";
    }

    private void CheckAxes() // sends the current angle configuration of the virtual robot to
RobotStudio for the reachability to be checked
    {
        // the meessage sent is coherent with what RobotStudio expects: "AC" means checking the
reachability of a certain angle configuration
        comms.message = "AC" + "[" + robot.axisAngles[0] + "," + robot.axisAngles[1] + "," +
robot.axisAngles[2] + "," + robot.axisAngles[3] + "," + robot.axisAngles[4] + "," +
robot.axisAngles[5] + "]";
        comms.flag = true;
    }

    private void CancelAngleChange() // resets the axis button state and the boolean indicator
angleChange is set to false
    {
        button.ExecutePress(); // executes a second press in the button that was pressed (it's a
button that can be toggled)
        angleChange = false;
    }

    public void AcceptedAngleChange() // a new MoveAxis Action is added to the Program, velocity and
precision are provisionally set to medium value ("2"), the ability to choose should be eventually
implemented
    {
        angleChange = false;
        Action act = new MoveAxis(robot.axisAngles,"2","3");
        AddAction(act);
        UpdateProgText();
        Debug.Log("[PM] Added movement of axes.");
    }

    public void SendPause() // tell RobotStudio to pause any instruction it may be executing, not
fully implemented
    {
        comms.message = "pause";
        comms.flag = true;
    }

    public void SendResume() // tell RobotStudio to resume executing the instruction it had paused
executing, not fully implemented
    {
        comms.message = "resume";
        comms.flag = true;
    }
}
```

Código *ViveController*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Valve.VR.InteractionSystem;

// the original author of this code is Carlos Álvarez Vereterra
// Fco Javier Domínguez Sánchez-Girón has only removed certain parts or added the elements marked with
[FJD] in their comment

[RequireComponent(typeof(SteamVR_TrackedObject))]
public class ViveController : MonoBehaviour
{
    // variables for point-setting functionality:
    public GameObject target;
    public GameObject origin;
    public Communications comms;
    public GameObject marker;
    public WorldCoodSystem worldCoord;
    public ToolCoordSystem toolCoord;
    public GameObject transGizmo;
    public GameObject rotGizmo;
    private Hand hand;
    public float angleX;
    public float angleY;
    public float angleZ;
    public Vector3 savedXAxis;
    public Vector3 savedYAxis;
    public Vector3 savedZAxis;
    public Quaternion rotationGizmoRecalc;
    public bool rotationDone;
    public bool recalculateGizmo = false;
    private LineRenderer lrWX;
    private LineRenderer lrWY;
    private LineRenderer lrWZ;
    private LineRenderer lrTX;
    private LineRenderer lrTY;
    private LineRenderer lrTZ;
    private Vector3 savedPosition;
    private Vector3 savedRotationEuler;

    // additional variables: [FJD]
    public bool settingPoint = false; // enables point-setting functionality
    private Vector3 hidden; // point gizmo's initial position

    private void Awake() // called before first frame update
    {
        rotationGizmoRecalc = Quaternion.identity;
        hidden = transGizmo.transform.position;
    }

    private void Update() // Update is called once per frame
    {
        if (settingPoint) // if the user opens the point-setting interface: [FJD]
        {
            // world coordinates
            Vector3 vWorldX = worldCoord.worldX;
            Vector3 vWorldY = worldCoord.worldY;
            Vector3 vWorldZ = worldCoord.worldZ;

            // tool coordinates
            Vector3 vToolX = toolCoord.toolX;
            Vector3 vToolY = toolCoord.toolY;
            Vector3 VToolZ = toolCoord.toolZ;

            // rotation matrix R
            float r11 = (float)Vector3.Dot(vWorldX, vToolX);
            float r12 = (float)Vector3.Dot(vWorldX, vToolY);
            float r13 = (float)Vector3.Dot(vWorldX, VToolZ);
```

```
            float r21 = (float)Vector3.Dot(vWorldY, vToolX);
            float r22 = (float)Vector3.Dot(vWorldY, vToolY);
            float r23 = (float)Vector3.Dot(vWorldY, VToolZ);
            float r31 = (float)Vector3.Dot(vWorldZ, vToolX);
            float r32 = (float)Vector3.Dot(vWorldZ, vToolY);
            float r33 = (float)Vector3.Dot(vWorldZ, VToolZ);

            // resulting angles
            float thetax = Mathf.Atan2(r32, r33);
            float thetay = Mathf.Atan2(-r31, Mathf.Sqrt(r32 * r32 + r33 * r33));
            float thetaz = Mathf.Atan2(r21, r11);
            float angleX = thetax * 180 / Mathf.PI;
            float angleY = thetay * 180 / Mathf.PI;
            float angleZ = thetaz * 180 / Mathf.PI;

            // resulting coordinates
            Vector3 res = (transGizmo.transform.position - origin.transform.position) * 1000;

            if (hand == null)
            {
                hand = this.GetComponent<Hand>();
            }
            if (hand.controller == null)
                return;

            // move robot target gizmo while pressing grip
            if (hand.controller.GetPress(SteamVR_Controller.ButtonMask.Grip))
            {
                savedPosition = hand.transform.position; // save hand coordinates
                savedRotationEuler = new Vector3(-angleX, -angleY, angleZ); // save calculated tool
angles
                // gizmo's position and rotation are set to hand's:
                transGizmo.transform.position = hand.transform.position;
                rotGizmo.transform.rotation = hand.transform.rotation;
            }

            // check robot target reachability on grip release
            if (hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Grip))
            {
                SendMessageRS(res, savedRotationEuler); // tell RobotStudio to check reachability
                // transform gizmo is set to tool axes
                target.GetComponent<TransformGizmo>().x_axis = toolCoord.toolX;
                target.GetComponent<TransformGizmo>().y_axis = toolCoord.toolY;
                target.GetComponent<TransformGizmo>().z_axis = toolCoord.toolZ;

                // save axis for gizmo rotation
                savedXAxis = target.GetComponent<TransformGizmo>().x_axis;
                savedYAxis = target.GetComponent<TransformGizmo>().y_axis;
                savedZAxis = target.GetComponent<TransformGizmo>().z_axis;
            }

            // repeat process if gizmo's position or rotation are modified
            if (recalculateGizmo)
            {
                recalculateGizmo = false; // reset flag
                res = (transGizmo.transform.position - origin.transform.position) * 1000; // relative
position in mm
                if (rotationDone) // if rotation gizmo is modified:
                {
                    rotationDone = false; // reset flag
                    Quaternion rot = Quaternion.Euler(savedRotationEuler); // apply rotation to saved
tool orientation
                    rot = rot * rotationGizmoRecalc;
                    SendMessageRS(res, GizmoToAngles(rotGizmo.transform.rotation)); // recalculate
target
                    savedRotationEuler = GizmoToAngles(rotGizmo.transform.rotation); // update saved
rotation
                }
                else
                    SendMessageRS(res, savedRotationEuler); // recalculate target
                // save axis for gizmo rotation
```

```
            savedXAxis = target.GetComponent<TransformGizmo>().x_axis;
            savedYAxis = target.GetComponent<TransformGizmo>().y_axis;
            savedZAxis = target.GetComponent<TransformGizmo>().z_axis;
        }
    }
    else // hide the gizmo [FJD]
    {
        transGizmo.transform.position = hidden;
        rotGizmo.transform.position = hidden;
    }
}

void SendMessageRS(Vector3 position, Vector3 rotation)
{
    // position data
    float x = position.x;
    float y = position.z;
    float z = position.y;

    // rotation data
    float rx = rotation.x;
    float ry = rotation.y;
    float rz = rotation.z;

    // robot configuration data
    int cf1 = 0;
    int cf4 = 0;
    int cf6 = 0;
    int cfx = 0;

    // calculate cf1 according to its quadrant in horizontal plane
    if (x >= 0 && y <= 0)
    {
        cf1 = -1;
    }
    else if (x >= 0 && y > 0)
    {
        cf1 = 0;

    }
    else if (x < 0 && y <= 0)
    {
        cf1 = -2;

    }
    else if (x < 0 && y > 0)
    {
        cf1 = 1;
    }

    // calculate cf4 according to y rotation
    if (ry >= -90 && ry < 0)
    {
        cf4 = 0;
    }
    else if (ry >= -180 && ry < -90)
    {
        cf4 = 1;
    }
    else if (ry >= 0 && ry < 90)
    {
        cf4 = -1;
    }
    else if (ry >= 90 && ry < 180)
    {
        cf4 = -2;
    }

    // assume same configuration
    cf6 = cf4;
```

```
        // the message to be sent to RobotStudio is created:
        comms.message = "XC"; // this means calculation of joint angles (target reachability) for
RobotStudio [FJD]
        comms.message += x.ToString("0") + ";";
        comms.message += y.ToString("0") + ";";
        comms.message += z.ToString("0") + ";";
        comms.message += rx.ToString("0.00000") + ";";
        comms.message += ry.ToString("0.00000") + ";";
        comms.message += rz.ToString("0.00000") + ";";
        comms.message += cf1.ToString("0") + ";";
        comms.message += cf4.ToString("0") + ";";
        comms.message += cf6.ToString("0") + ";";
        comms.message += cfx.ToString("0") + ";";
        comms.flag = true;
    }

    Vector3 GizmoToAngles(Quaternion gizmoRot)
    {
        Vector3 result;
        result.x = -180 + CorrectAngle(gizmoRot.eulerAngles.z);
        result.y =  360 - CorrectAngle(gizmoRot.eulerAngles.x);
        result.z =   90 - CorrectAngle(gizmoRot.eulerAngles.y - 180);
        result.x = CorrectAngle(result.x);
        result.y = CorrectAngle(result.y);
        result.z = CorrectAngle(result.z);
        return result;
    }

    float SnapAngle(float angle, float deltaAngle)
    {
        float multiplier = Mathf.Round(angle / deltaAngle);
        return multiplier * deltaAngle;
    }

    float CorrectAngle(float angle)
    {
        float correctedAngle;
        if (angle > 180)
            correctedAngle = angle - 360;
        else if (angle < -180)
            correctedAngle = angle + 360;
        else
            correctedAngle = angle;
        return correctedAngle;
    }
}
```

Código *Communications*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using SharpConnect;
using System.Net;
using System.Net.Sockets;
using System.Threading;

// the original author of this code is Carlos Álvarez Vereterra
// Fco Javier Domínguez Sánchez-Girón has only removed certain parts or added the elements marked with
[FJD] in their comment

public class Communications : MonoBehaviour {

    // communication variables
    private static string ip = "127.0.0.1";
    public string ipRobotStudioTCP = ip;
    public string ipPLCSIM = ip;
    public int portRobotStudioTCP = 5000;
    public int portRobotStudioUDP = 7000;
    public int portPLCSIM = 6000;
    byte[] dataSend;
    byte[] dataReceive;
    public bool flag = false;
    public string message;
    public bool receivedResponse = false;
    private bool abort = false;
    Thread threadRobotStudio = null;
    Thread threadPLCSIM = null;
    Thread threadRobotStudioUDP = null;
    TcpClient clientRobotStudio;
    UdpClient clientRobotStudioUDP;
    TcpClient clientPLCSIM;
    int i = 0;
    NetworkStream streamRobotStudio;
    NetworkStream streamPLCSIM;
    public IRB120 robot;

    // additional variables: [FJD]
    public string savedMessage; // stores each message sent to RobotStudio (1 at a time) for later
possible creation of GoTo Action
    public int executed = 0; // indicates whether a sent instructions has been executed (0 = not, 1 =
yes)
    private bool init = true; // initialization's communication flag

    void Start () {
        // Create and start RobotStudio communications thread
        threadRobotStudio = new Thread(CommsRobotStudioTCP);
        threadRobotStudio.Start();
        //// Create and start PLCSIM communications thread
        //threadPLCSIM = new Thread(CommsPLCSIM);
        //threadPLCSIM.Start();
        // Create and start RobotStudio communications thread for Path execution
        threadRobotStudioUDP = new Thread(CommsRobotStudioUDP);
        threadRobotStudioUDP.Start();
}

    void CommsPLCSIM()
    {
        while (!abort)
        {
            clientPLCSIM = new TcpClient(ipPLCSIM, portPLCSIM);
            streamPLCSIM = clientPLCSIM.GetStream();
            streamPLCSIM.ReadTimeout = 2000;
            streamPLCSIM.WriteTimeout = 2000;
            float testf = 203.593f - 50 * i;
            byte[] inputsPLCSIM = System.Text.Encoding.ASCII.GetBytes("1");
            try
```

```
            {
                streamPLCSIM.Write(inputsPLCSIM, 0, inputsPLCSIM.Length);
                Debug.Log("[PLCSIM] Sent message: " +
System.Text.Encoding.ASCII.GetString(inputsPLCSIM));
            }
            catch (System.IO.IOException e)
            {
                Debug.Log("[PLCSIM] Error sending message to PLCSIM");
                streamPLCSIM.Close();
                clientPLCSIM.Close();
            }
            testf = testf - 10;
            try
            {
                byte[] outputsPLCSIM;
                // We have to initialize the receiving buffer
                outputsPLCSIM = System.Text.Encoding.ASCII.GetBytes("XXXXXXXXXXXXXXXXXXXXXX");
                streamPLCSIM.Read(outputsPLCSIM, 0, 20);
                Debug.Log("[PLCSIM] Received Message: " +
System.Text.Encoding.ASCII.GetString(outputsPLCSIM));
                if (!(System.Text.Encoding.ASCII.GetString(outputsPLCSIM).Contains("error")))
                {
                    Debug.Log("[PLCSIM] Received message: " +
System.Text.Encoding.ASCII.GetString(outputsPLCSIM));
                }
            }
            catch (System.IO.IOException e)
            {
                Debug.LogError("[PLCSIM] Error receiving message from PLCSIM");
                Debug.Log(e.ToString());
                streamPLCSIM.Close();
                clientPLCSIM.Close();
            }
        }
        return;
    }

    // continuous communication for moving the virtual robot like the real one
    void CommsRobotStudioUDP() // receives the real robot's joint angles from RobotStudio
    {
        clientRobotStudioUDP = new UdpClient(portRobotStudioUDP);
        IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, portRobotStudioUDP);
        while (!abort)
        {
            try
            {
                if(!abort & clientRobotStudioUDP.Available > 0)
                {
                    byte[] angles;
                    angles = clientRobotStudioUDP.Receive(ref remoteEP);
                    MoveRobot(robot, System.Text.Encoding.ASCII.GetString(angles)); // virtual robot's
joint configuration is updated according to real robot's
                }
            }
            catch(System.IO.IOException ex)
            {
                Debug.LogError("[UDP] Error > Unable to receive joint angles > " + ex.ToString());
            }
        }
        return;
    }

    // communication for sending information and receiving verification
    void CommsRobotStudioTCP()
    {
        string receivedMessage; // message from RobotStudio
        while (!abort)
        {
            if (init) // first communication, used to initialize the virtual robot by telling
RobotStudio to send current joint angles (through UDP) [FJD]
            {
```

```
                init = false; // initialization flag is reset
                Debug.Log("[TCP] beforeclientTCP");
                clientRobotStudio = new TcpClient(ipRobotStudioTCP, portRobotStudioTCP);
                streamRobotStudio = clientRobotStudio.GetStream();
                Debug.Log("[TCP] afterclientTCP");
                streamRobotStudio.ReadTimeout = 20000;
                streamRobotStudio.WriteTimeout = 20000;
                dataSend = System.Text.Encoding.ASCII.GetBytes("I"); // "I" means initialization for
RobotStudio
                try
                {
                    streamRobotStudio.Write(dataSend, 0, dataSend.Length);
                }
                catch (System.IO.IOException e)
                {
                    Debug.LogError("[TCP] Initialization error > Unable to send message to RobotStudio
> " + e.ToString());
                    streamRobotStudio.Close();
                    clientRobotStudio.Close();
                }
                try
                {
                    dataReceive = System.Text.Encoding.ASCII.GetBytes("XXXXXXXXXXXXXXXXXXXXXX"); //
we have to initialize the receiving buffer
                    streamRobotStudio.Read(dataReceive, 0, 20);
                    if (System.Text.Encoding.ASCII.GetString(dataReceive).Contains("error"))
                    {
                        Debug.LogError("[TCP]: Initialization error > Initialization not allowed by
RobotStudio");
                        receivedResponse = true;
                    }
                    else
                    {
                        executed = 1;
                        receivedMessage =
System.Text.Encoding.ASCII.GetString(dataReceive).Replace("X", string.Empty);
                        Debug.Log("[TCP] Received message: " + receivedMessage);
                    }
                    streamRobotStudio.Close();
                    clientRobotStudio.Close();
                }
                catch (System.IO.IOException e)
                {
                    Debug.LogError("[TCP] Initialization error > Unable to receive message from
RobotStudio > " + e.ToString());
                    streamRobotStudio.Close();
                    clientRobotStudio.Close();
                }
            }

            if (flag) // used for asking RobotStudio for target reachability, sending instructions and
receiving confirmation
            {
                flag = false; // flag is reset
                Debug.Log("[TCP] beforeclientTCP");
                clientRobotStudio = new TcpClient(ipRobotStudioTCP, portRobotStudioTCP);
                streamRobotStudio = clientRobotStudio.GetStream();
                Debug.Log("[TCP] afterclientTCP");
                streamRobotStudio.ReadTimeout =20000;
                streamRobotStudio.WriteTimeout = 20000;
                dataSend = System.Text.Encoding.ASCII.GetBytes(message);
                savedMessage = message;
                try
                {
                    streamRobotStudio.Write(dataSend, 0, dataSend.Length);
                    Debug.Log("[TCP] Sent message: " +
System.Text.Encoding.ASCII.GetString(dataSend));
                }
                catch (System.IO.IOException e)
                {
                    Debug.LogError("[TCP] Error sending message to RobotStudio > " + e.ToString());
```

```
                    streamRobotStudio.Close();
                    clientRobotStudio.Close();
                }
                try
                {
                    dataReceive = System.Text.Encoding.ASCII.GetBytes("XXXXXXXXXXXXXXXXXXXXXX"); //
we have to initialize the receiving buffer
                    streamRobotStudio.Read(dataReceive, 0, 20);
                    if (System.Text.Encoding.ASCII.GetString(dataReceive).Contains("error"))
                    {
                        Debug.LogError("[TCP] Error > Point not reachable");
                        receivedResponse = true;
                    }
                    else if (System.Text.Encoding.ASCII.GetString(dataReceive).Contains("success")) //
[FJD]
                    {
                        executed = 1; // execution flag is set to 1 to indicate execution has been
confirmed
                        receivedMessage =
System.Text.Encoding.ASCII.GetString(dataReceive).Replace("X", string.Empty);
                        Debug.Log("[TCP] Received message: " + receivedMessage);
                    }
                    else
                    {
                        receivedMessage = System.Text.Encoding.ASCII.GetString(dataReceive);
                        Debug.Log("[TCP] Received message: " + receivedMessage);
                        MoveRobot(robot, System.Text.Encoding.ASCII.GetString(dataReceive));
                        receivedResponse = true;
                    }
                    streamRobotStudio.Close();
                    clientRobotStudio.Close();
                }
                catch(System.IO.IOException e)
                {
                    Debug.LogError("[TCP] Error receiving message from RobotStudio > " +
e.ToString());
                    streamRobotStudio.Close();
                    clientRobotStudio.Close();
                }

            }
        }
        return;
    }

    public float[] MoveRobot(IRB120 robot, string angles) // moves the virtual robot
    {
        float a1, a2, a3, a4, a5, a6;
        string[] result;
        angles = angles.Replace("X", string.Empty);
        result = angles.Split(';');
        a1 = float.Parse(result[0]);
        a2 = float.Parse(result[1]);
        a3 = float.Parse(result[2]);
        a4 = float.Parse(result[3]);
        a5 = float.Parse(result[4]);
        a6 = float.Parse(result[5]);
        float[] ang = { a1, a2, a3, a4, a5, a6 };
        robot.axisAngles = ang; // virtual robot's joint angles are set to the new ones
        return ang;
    }

    void OnApplicationQuit()
    {
        abort = true;
        if (clientRobotStudio != null || streamRobotStudio != null)
        {
            streamRobotStudio.Close();
            Debug.Log("[COM] Closed stream");
            clientRobotStudio.Close();
            Debug.Log("[COM] Closed Client");
```

```
        }
    }
}
```

Código *TestObjects*

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TestObjects : MonoBehaviour
{
    private Vector3 initialPos;
    private Quaternion initialRot;
    private bool moved1 = false;
    private bool moved2 = false;
    private float t;

    // Start is called before the first frame update
    void Start()
    {
        initialPos = transform.position;
        initialRot = transform.rotation;
    }

    // Update is called once per frame
    void Update()
    {
        if (transform.position != initialPos && !moved1)
        {
            moved1 = true;
        }
        if (Time.time - t >= 2 && moved2)
        {
            transform.position = initialPos;
            transform.rotation = initialRot;
            moved1 = false;
            moved2 = false;
        }
    }
    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Floor" && moved1)
        {
            t = Time.time;
            moved2 = true;
        }
    }
}
```

Código *MainModule*

```
MODULE MainModule

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

    ! The PERS modifier indicates a persistant variable

    ! Robot target, includes coordinates, rotation, robot configuration and position of external axes
    ! Initialized at random value, will be updated by receiving data from Unity via TCP/IP socket
    PERS robtarget robotTarget := [[477,-4,50], [0.0160149,-0.0323332,-0.999134,-0.0207343], [-
1,0,0,0], [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    PERS num anglex := 177.684;
    PERS num angley := -1.91076;
    PERS num anglez := 176.332;
    PERS jointtarget angleTarget := [[101, 52, -9, 1, 45, 6],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

    PERS bool toolAct := TRUE;

    ! Boolean to indicate if a path is currently executing
    PERS bool executingPath := FALSE;

    ! Result of the calculation (0 = possible, <0 = not possible)
    PERS num calculationResult := -2;
    ! Resulting joint data from the calculation, initialized at random variable, will be updated
    PERS jointtarget jointDataResult := [[3.07759,51.8538,-15.7295,-
0.0531168,55.5042,7.42994],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

        ! Resulting robtarget data from the calculation, initialized at random variable, will be
updated
    PERS robtarget targetDataResult := [[-36.6396,481.733,149.975], [0.00529182,-
0.734676,0.678164,0.0177754], [1,0,0,0], [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

    ! Communication variables
    VAR socketdev server;
    VAR socketdev client;
    VAR string message;
    VAR rawbytes data;

        ! Indicates if the angles have been obtained from message
    VAR bool gotAngles := FALSE;

    PROC main()
        ! Declare variable to store error number
        VAR errnum errorNumber;

                ! Data received and converted to speeddata, zonedata and Move J or L used
        VAR string vel;
        VAR string prec;
        VAR string mode;

                ! String used to create jointtarget from message
        VAR string joint;

                ! speeddata and zonedata used
        VAR speeddata spd;
        VAR zonedata zn;

        ! Create server socket and bind it to ip address and port
        SocketCreate server;
        SocketBind server, ip, port;

        ! Listen and accept incoming connections from server socket into client socket
        SocketListen server;
        SocketAccept server, client;

        ! Receive robotTarget from Unity in rawbytes format
        SocketReceive client, \RawData:=data;
        ! Convert rawbytes to string for easier manipulation
```

```
        UnpackRawBytes data, 1, message, \ASCII:=80;

        IF StrLen(message) > 0 THEN
            ! Initialization
            IF StrPart(message,1,1) = "I" THEN
                toolAct := FALSE;
                UpdateToolState;
                executingPath:=TRUE;
                WaitTime 5;
                executingPath:=FALSE;
                SocketSend client,\str:="initialised";
                SocketClose client;
                SocketClose server;
            ! GoTo
            ELSEIF StrPart(message,1,1) = "X" THEN
                                ! Check reachability for GoTo
                IF StrPart(message,2,1) = "C" THEN
                    ! Convert message string to robtaget object
                    robotTarget := StringToTarget(StrPart(message,3,StrLen(message)-2));
                    ! Reset calculation result variable
                    calculationResult := 0;
                    ! Calculate the joint angles given the robot target, and store errors in
errorNumber
                    jointDataResult := CalcJointT(robotTarget, PinzaToolTCP, \WObj:=wobj0);
                    ! If there were no errors
                    IF calculationResult <> -1 AND calculationResult <> -2 THEN
                        ! Send resulting robot configuration to Unity
                        SocketSend client,\str:=jointTargetToString(jointDataResult);
                        ! Close both server and client sockets
                        SocketClose client;
                        SocketClose server;
                    ENDIF
                                ! Execute GoTo
                ELSE
                                    ! Get Move instruction parameters
                    vel := StrPart(message,2,1);
                    spd := GetSpeed(vel);
                    prec := StrPart(message,3,1);
                    zn := GetPrecision(prec);
                    mode := StrPart(message,4,1);
                    robotTarget := StringToTarget(StrPart(message,5,StrLen(message)-4));
                    executingPath := TRUE;
                    ! Ignore current robot configuration during linear movement
                    ConfL \Off;
                                    ! Execute Move instruction
                    IF mode = "L" THEN
                        MoveL robotTarget, spd, zn, PinzaToolTCP;
                    ELSE
                        MoveJ robotTarget, spd, zn, PinzaToolTCP;
                    ENDIF
                                    ! Wait until finished moving
                    WaitRob \InPos;
                    executingPath := FALSE;
                    SocketSend client,\str:="success";
                    SocketClose client;
                    SocketClose server;
                ENDIF
                        ! MoveAxis
            ELSEIF StrPart(message,1,1) = "A" THEN
                                ! Check reachability for MoveAxis
                IF StrPart(message,2,1) = "C" THEN
                                    ! Create target from received message
                    joint := "[" + StrPart(message,3,StrLen(message)-2) +
",[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]]";
                    gotAngles := StrToVal(joint,angleTarget);
                    calculationResult := 0;
                    targetDataResult := CalcRobT(angleTarget,PinzaToolTCP,\WObj:=wobj0);
                                    ! Check reachability
                    jointDataResult := CalcJointT(targetDataResult, PinzaToolTCP, \WObj:=wobj0);
                    IF calculationResult <> -1 AND calculationResult <> -2 THEN
                        ! Send resulting robot configuration to Unity
```

```
                    SocketSend client,\str:=jointTargetToString(jointDataResult);
                    SocketClose client;
                    SocketClose server;
                ENDIF
                            ! Execute MoveAxis
            ELSE
                                ! Get parameters
                vel := StrPart(message,2,1);
                spd := GetSpeed(vel);
                prec := StrPart(message,3,1);
                zn := GetPrecision(prec);
                joint := "[" + StrPart(message,4,StrLen(message)-3) +
",[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]]";
                gotAngles := StrToVal(joint,angleTarget);
                IF gotAngles AND joint <> "" THEN
                    executingPath := TRUE;
                    MoveAbsJ angleTarget, spd, zn, PinzaToolTCP;
                    WaitRob \InPos;
                    executingPath := FALSE;
                    SocketSend client,\str:="success";
                    SocketClose client;
                    SocketClose server;
                ELSE
                    SocketSend client,\str:="error";
                    SocketClose client;
                    SocketClose server;
                ENDIF
            ENDIF
        ! Activate or deactivate tool
        ELSEIF StrPart(message,1,1) = "Y" THEN
            IF StrPart(message,2,1) = "T" THEN
                toolAct := TRUE;
            ELSEIF StrPart(message,2,1) = "F" THEN
                toolAct := FALSE;
            ENDIF
            UpdateToolState;
            SocketSend client,\str:="success";
            SocketClose client;
            SocketClose server;
        ELSEIF message = "pause" THEN
            StopMove;
            SocketSend client,\str:="paused";
            SocketClose client;
            SocketClose server;
        ELSEIF message = "resume" THEN
            StartMove;
            SocketSend client,\str:="resumed";
            SocketClose client;
            SocketClose server;
        ENDIF
    ENDIF
    ! Error handling
    ERROR
        IF ERRNO = ERR_ROBLIMIT THEN
            SkipWarn;
            calculationResult := -1;
            !Send error
            SocketSend client,\str:="error";
            !Close communication
            SocketClose client;
            SocketClose server;
            TRYNEXT;
        ELSEIF ERRNO = ERR_OUTSIDE_REACH THEN
            SkipWarn;
            calculationResult := -2;
            !Close communication
            !Send error
            SocketSend client,\str:="error";
            SocketClose client;
            SocketClose server;
```

```
            TRYNEXT;
    ELSEIF ERRNO = ERR_SOCK_TIMEOUT OR ERRNO = ERR_SOCK_CLOSED THEN
        SkipWarn;
        calculationResult := -2;
        SocketClose client;
        SocketClose server;
        TRYNEXT;
        ENDIF
ENDPROC

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

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

FUNC speeddata GetSpeed(string value)
    VAR speeddata tempSpeed;
    IF value = "3" THEN
                tempSpeed := v2000;
            ELSEIF value = "2" THEN
                tempSpeed := v400;
            ELSE
                tempSpeed := v50;
            ENDIF
    RETURN tempSpeed;
ENDFUNC
FUNC zonedata GetPrecision(string value)
    VAR zonedata tempPrec;
        IF value = "1" THEN
                tempPrec := fine;
            ELSEIF value = "2" THEN
                tempPrec := z50;
            ELSE
                tempPrec := z200;
            ENDIF
    RETURN tempPrec;
ENDFUNC

PROC UpdateToolState()
    IF toolAct THEN
        IF pinzaOutput1 = 1 THEN
            SetDO pinzaOutput1, 0;
        ENDIF
        IF pinzaOutput0 = 0 THEN
            SetDO pinzaOutput0, 1;
        ENDIF
    ELSE
        IF pinzaOutput1 = 0 THEN
            SetDO pinzaOutput1, 1;
        ENDIF
        IF pinzaOutput0 = 1 THEN
            SetDO pinzaOutput0, 0;
        ENDIF
    ENDIF
ENDPROC

FUNC robtarget StringToTarget(string value)
    VAR robtarget tempTarget;
    VAR bool bResult;
    VAR num ax;
```

```
        VAR num ay;
        VAR num az;

        VAR num posX;
        VAR num posY;
        VAR num posZ;
        VAR num posAX;
        VAR num posAY;
        VAR num posAZ;
        VAR num poscf1;
        VAR num poscf4;
        VAR num poscf6;
        VAR num poscfx;

        ! Starting position index
        posX:=StrFind(value,1,";");
        posY:=StrFind(value,posX+1,";");
        posZ:=StrFind(value,posY+1,";");
        posAX:=StrFind(value,posZ+1,";");
        posAY:=StrFind(value,posAX+1,";");
        posAZ:=StrFind(value,posAY+1,";");
        poscf1:=StrFind(value,posAZ+1,";");
        poscf4:=StrFind(value,poscf1+1,";");
        poscf6:=StrFind(value,poscf4+1,";");
        poscfx:=StrFind(value,poscf6+1,";");

        ! Position data
        bResult:=StrToVal(StrPart(value,1,posX-1),tempTarget.trans.x);
        bResult:=StrToVal(StrPart(value,posX+1,posY-posX-1),tempTarget.trans.y);
        bResult:=StrToVal(StrPart(value,posY+1,posZ-posY-1),tempTarget.trans.z);

        ! Orientation data
        bResult:=StrToVal(StrPart(value,posZ+1,posAX-posZ-1),ax);
        bResult:=StrToVal(StrPart(value,posAX+1,posAY-posAX-1),ay);
        bResult:=StrToVal(StrPart(value,posAY+1,posAZ-posAY-1),az);

        tempTarget.rot := OrientZYX(az,ay,ax);
        anglex := ax;
        angley := ay;
        anglez := az;

        ! Configuration data
        bResult:=StrToVal(StrPart(value,posAZ+1,poscf1-posAZ-1),tempTarget.robconf.cf1);
        bResult:=StrToVal(StrPart(value,poscf1+1,poscf4-poscf1-1),tempTarget.robconf.cf4);
        bResult:=StrToVal(StrPart(value,poscf4+1,poscf6-poscf4-1),tempTarget.robconf.cf6);
        bResult:=StrToVal(StrPart(value,poscf6+1,poscfx-poscf6-1),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
```