ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA INDUSTRIAL

USO DE LA REALIDAD VIRTUAL PARA EL ENTRENAMIENTO DEL PERSONAL DE OPERACIÓN Y MANTENIMIENTO: APLICACIÓN A LA MINIFÁBRICA ICAI

Autor: Carlos Álvarez Vereterra
Director: José Antonio Rodríguez Mondéjar

Madrid
Junio 2018

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. Carlos Álvarez Vereterra DECLARA ser el titular de los derechos de propiedad intelectual de la obra: **Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI**, 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 19 de Junio de 2018

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

Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 17/18 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.: Carlos Álvarez Vereterra Fecha: 17/06/2018

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: José Antonio Rodríguez Mondéjar Fecha: 17/06/2018

USO DE LA REALIDAD VIRTUAL PARA EL ENTRENAMIENTO DEL PERSONAL DE OPERACIÓN Y MANTENIMIENTO: APLICACIÓN A LA MINIFÁBRICA ICAI

Autor: Álvarez Vereterra, Carlos.

Director: Rodríguez Mondéjar, José Antonio.

Entidad Colaborativa: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Introducción

Planteamiento del problema y estado de la técnica

En la actualidad existen diversas empresas que comercializan soluciones para entrenamiento de operarios mediante tecnología de realidad virtual. Estos servicios sirven para entrenar a operarios en tareas como la operación de paneles de control, tareas de mantenimiento preventivo, que tendrán que realizar mientras la planta esté en funcionamiento; además de tareas de mantenimiento correctivo, en caso de que se produzca alguna avería. La principal ventaja de estos sistemas es que no es necesario tener la planta construida, por lo que se pueden realizar las tareas de entrenamiento en paralelo a las tareas de construcción. Esto implica no invertir tiempo adicional para que los operarios aprendan a manejar el sistema, una vez finalizada la construcción de la planta.

La realidad virtual tiene dos principales motivaciones para integrarse en el campo de la robótica industrial. En primer lugar, puede hacer el proceso de programación mucho más intuitivo, si se realiza con controladores localizados en el espacio que imiten el movimiento de la mano del usuario. En segundo lugar, tiene la ventaja de la seguridad. Los robots industriales actuales pueden moverse a velocidades elevadas, como por ejemplo el ABB IRB 120, que, dependiendo de la articulación, puede moverse desde 250º/s hasta 600º/s. Esto hace que la presencia del operario en el volumen de trabajo del robot suponga un peligro para la seguridad, que se elimina con la simulación en realidad virtual.

Objeto del proyecto

El objetivo de este proyecto es que la solución de software desarrollada sea capaz de emplear la tecnología de realidad virtual para el entrenamiento de operarios de plantas industriales, además de la programación de un manipulador robótico.

Se simulará la minifábrica ICAI en un entorno de realidad virtual, se desarrollará un módulo de entrenamiento de operarios para tareas de operación y mantenimiento. Además, se desarrollará otro módulo de programación del robot ABB IRB 120 con controles en realidad virtual, para ver si es factible y aporta alguna ventaja con respecto a los métodos de control y programación del robot tradicionales.

En ambos módulos, el sistema debe de ser sencillo de utilizar, para que el usuario pueda centrarse en el proceso de aprendizaje, y no en el manejo del sistema de realidad virtual.

Metodología

Para la ejecución de este proyecto se ha decidido seguir la estrategia de dividir el problema en varios subsistemas. Las partes en las que se ha dividido el proyecto son:

Simulación de la planta en un entorno de realidad virtual

Utilizando el motor gráfico Unity 5, se desarrollará una simulación de la minifábrica ICAI compatible con realidad virtual, en la que se pueda visualizar la planta funcionando, controlar a través de un pupitre de mando (Figura 1), y entrenar al operario para la realización de tareas de mantenimiento.

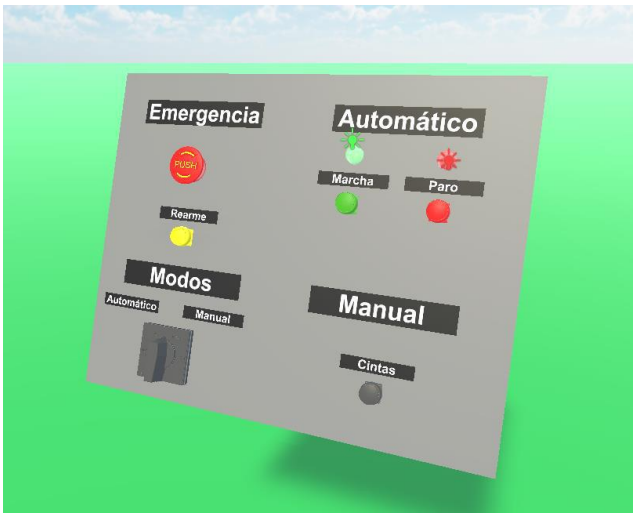


Figura 1: Pupitre de control de la planta virtual

Además, también se importará un modelo 3D del manipulador robótico ABB IRB 120 y se desarrollará una solución que permita, en primer lugar, comprobar la alcanzabilidad de puntos marcados con el mando inalámbrico del sistema de realidad virtual (en este caso el Oculus Rift), como se muestra en la Figura 2; y en segundo lugar, guardar esos puntos en un camino o *path* para que el robot lo pueda ejecutar posteriormente. Los cálculos de la cinemática inversa y los ángulos de las articulaciones se harán utilizando el software RobotStudio, por lo que habrá que diseñar un sistema de comunicaciones (basado en TCP/IP y UDP/IP) que permita el intercambio de información entre ambos programas.

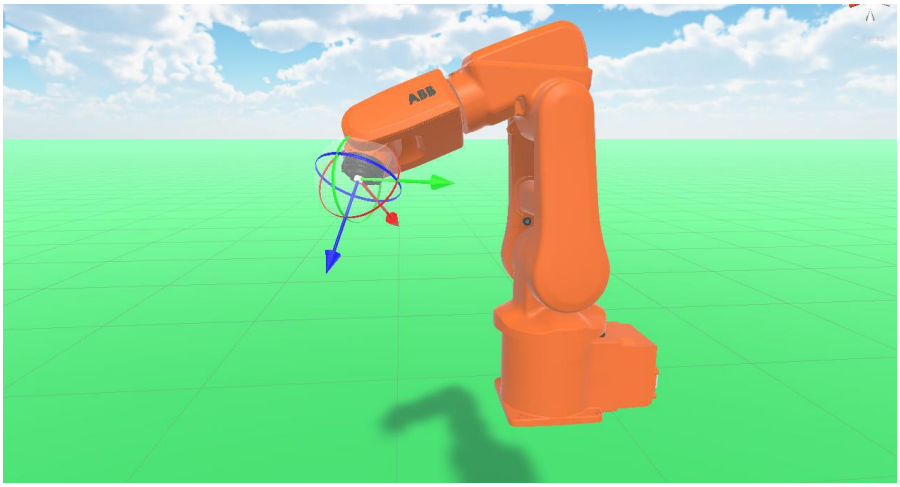


Figura 2: Comprobación de la alcanzabilidad de un punto, visualizado en Unity 5

Cálculo y programación del robot con RobotStudio

Utilizando el software RobotStudio, se desarrollará un programa que permita comprobar la alcanzabilidad de un objetivo, enviado desde Unity, y si es posible, RobotStudio enviará de vuelta los ángulos de las articulaciones del robot. Además, este programa tiene que permitir crear un *path* de forma dinámica, es decir, que el usuario pueda guardar puntos objetivo desde Unity, y posteriormente ejecutar dicho *path*.

Módulo de control utilizando un PLC virtual

Utilizando el software PLCSIM Advanced, desarrollado por Siemens, se simulará un autómata programable, que será el encargado de gestionar la planta virtual. Dentro de esta parte, se distinguen dos secciones:

- Programación del PLC. Se desarrollarán los programas de control necesarios para gestionar los distintos actuadores de la planta, como las cintas transportadoras.

- Interfaz con el PLC virtual. Se desarrollará un programa que permita captura las entradas de la planta, como pueden ser los pulsadores del pupitre de control de la planta (Figura 1), escribirlas en las entradas del PLC virtual utilizando la API de PLCSIM Advanced, y transferir las salidas del PLC virtual a la planta virtual de Unity.

Resultados y conclusiones

En cuanto al programa de entrenamiento de operarios, se ha conseguido desarrollar un tutorial que muestre información al operario dependiendo de la operación que tiene que realizar, se simula la interacción con elementos como válvulas y herramientas, y se hace énfasis en la secuencia que tiene que realizar el usuario, por lo que se consideran cumplidos los objetivos del proyecto con respecto a esta área.

Además, se ha conseguido simular la minifábrica ICAI, controlada por un autómata programable, y la programación podría ser portada a los PLCs físicos que controlan la planta, con lo que se consigue una interoperabilidad óptima, ya que no es necesario desarrollar código específico para la simulación.

Los resultados obtenidos en el módulo de programación del robot son muy positivos. No solo se ha demostrado la viabilidad del uso de la tecnología de realidad virtual para la programación de robots industriales, sino que tras la experiencia de usuarios que nunca habían utilizado un sistema de realidad virtual o nunca habían programado un robot industrial, resulta extremadamente intuitivo, comparado con los sistemas tradicionales, como una controladora *FlexPendant*, que requiere horas de entrenamiento para ser manejada con soltura.

La flexibilidad que proporciona el poder indicarle el objetivo al robot con la mano en un entorno 3D, como la capacidad de cambiar de punto de vista simplemente moviendo la cabeza o agachándose, proporciona un nivel de control y facilidad de uso incomparables, con un tiempo de adaptación o entrenamiento mínimo.

La conclusión es que la tecnología de realidad virtual para la programación de robots industriales no solo es una posibilidad anecdótica, sino que probablemente en el futuro sea una de las formas estándar de interactuar con los manipuladores robóticos.

USE OF VIRTUAL REALITY TECHNOLOGY FOR OPERATOR TRAINING (OPERATION AND MAINTENANCE) APPLIED TO ICAI'S MINI-FACTORY.

Author: Álvarez Vereterra, Carlos.

Director: Rodríguez Mondéjar, José Antonio.

Collaborating Entity: ICAI – Universidad Pontificia Comillas

PROJECT SUMMARY

Introduction

State of the art

Currently there are several companies that offer solutions for operator training using virtual reality technology. These services aim to train operators in tasks such as the operation of control panels, preventive maintenance tasks, which they will have to carry out while the plant is in operation; in addition to corrective maintenance tasks, in case of any failure. The main advantage of these systems is that it is not necessary to have the plant built, so that the training tasks can be carried out at the same time of the construction of the plant. This implies not investing additional time for the operators to learn how to operate the system, once the construction of the plant is completed.

Virtual reality has two main motivations to be integrated in the field of industrial robotics. First, it can make the programming process much more intuitive, if it is done with controllers located in 3D space that mimic the movement of the user's hand. Second, it has the advantage of security. Today's industrial robots can move at high speeds, such as the ABB IRB 120, which, depending on the articulation, can move from 250 º/s to 600 º/s. This means that the presence of the operator in the work volume of the robot supposes a danger for his safety, that is eliminated with the simulation in virtual reality.

Project objectives

The objective of this project is that the developed software solution is able to use virtual reality technology for the training of operators of industrial plants as well as programming a robotic manipulator.

The ICAI mini-factory will be simulated in a virtual reality environment, with a training module for operators for operation and maintenance tasks. In addition, another module for programming the ABB IRB 120 robot with virtual reality controls will be developed to see if it is feasible and provides advantages when compared to traditional control and programming methods of the robot.

In both modules, the system must be simple to use, so that the user can focus on the learning process, and not on the management of the virtual reality system.

Methodology

For the execution of this project it has been decided to follow the strategy of dividing the problem into several subsystems. The parts in which the project has been divided are:

Simulation of the plant in a virtual reality environment

Using the Unity 5 graphics engine, a simulation of the ICAI mini-factory compatible with virtual reality will be developed, in which the operating plant can be visualized, controlled through a control desk (Figure 1), and train the operator for the realization of maintenance tasks.

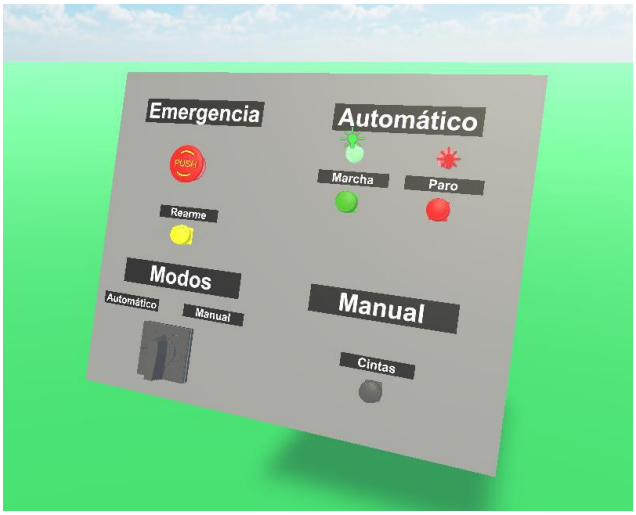


Figure 1: Control panel of the virtual plant

In addition, a 3D model of the robotic manipulator ABB IRB 120 will also be imported and a solution will be developed that allows checking the reachability of points marked with the wireless controller of the virtual reality system (in this case the Oculus Rift), as shown in Figure 1; and save those points in a path so that the robot can execute it later. The calculations of the inverse kinematics and joint angles will be made using the RobotStudio software, so we will have to design a communication system (based on TCP / IP and UDP / IP) that allows the exchange of information between both programs.

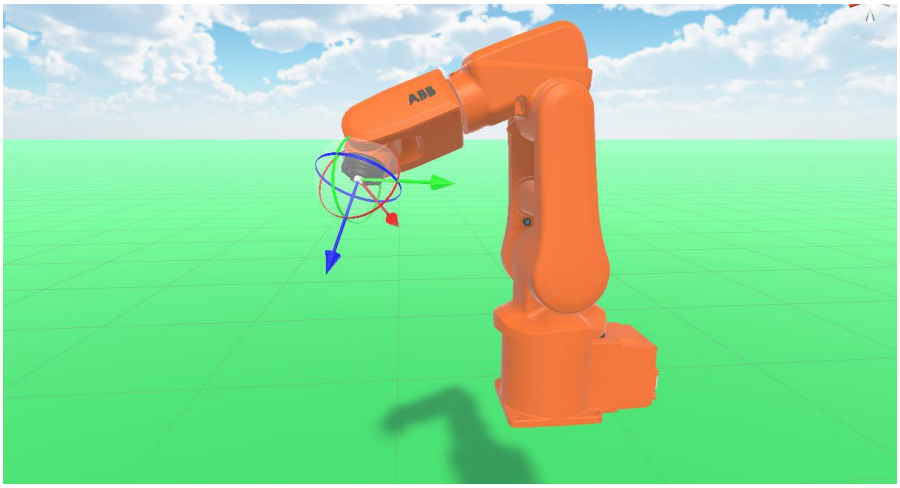


Figure 2: Robot target reachability check in Unity 5

Calculation and programming of the robot with RobotStudio

Using the RobotStudio software, a program will be developed to verify the reachability of a target, sent from Unity, and if possible, RobotStudio will send back the angles of the robot's joints. In addition, this program must allow a path to be created dynamically, that is, the user will be able to save target points from Unity, and then execute that path.

Control module using a virtual PLC

Using the PLCSIM Advanced software, developed by Siemens, a programmable automaton will be simulated, which will be in charge of managing the virtual plant. Within this part, two sections are distinguished:

- PLC programming. The control programs necessary to manage the different actuators of the plant, such as the conveyor belts, will be developed.
- Interface with the virtual PLC. A program will be developed to capture the inputs of the plant, such as the buttons of the control desk of the plant (Figure 1), write them in the virtual PLC inputs using the PLCSIM Advanced API, and transfer the outputs of the PLC virtual to the Unity virtual plant.

Results and conclusions

Regarding the operator training program, a tutorial has been developed that shows information to the operator depending on the operation he must perform, simulates the interaction with elements such as valves and tools, and emphasizes the sequence that must be performed by the user, so that the objectives of the project with respect to this area are considered fulfilled.

In addition, it has been possible to simulate the ICAI mini-factory, controlled by a programmable automaton, and the programming could be carried out to the physical PLCs that control the plant, which implies optimum interoperability, since it is not necessary to develop specific code for the simulation.

The results obtained in the robot programming module are outstanding. Not only has the viability of using virtual reality technology for the programming of industrial robots has been proven, but after observing the experience of users who had never used a virtual reality system or had never programmed an industrial robot, it is extremely intuitive, compared to traditional systems, such as a *FlexPendant* controller, which requires hours of training to be handled proficiently.

The flexibility provided by being able to point the objective at the robot by hand in a 3D environment, as well as the ability to change points of view simply by moving the head or ducking down, provides an incomparable level of control and ease of use, with minimal time of adaptation or minimum training.

The conclusion is that virtual reality technology for the programming of industrial robots is not only an anecdotal possibility, but probably in the future will be one of the standard ways of interacting with robotic manipulators.

Universidad Pontificia Comillas
ICAI

Trabajo fin de máster

Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI

Carlos Álvarez Vereterra

Director de proyecto:
Dr. José A. Rodríguez Mondéjar

Índice general

I. Memoria **10**

1. Introducción 12
2. Estado del arte 14
   - 2.1. Realidad virtual 14
   - 2.2. Entrenamiento de operarios con tecnología de realidad virtual . 16
   - 2.3. Realidad Virtual aplicada a robots industriales 18
3. Objetivo del proyecto 20
4. Arquitectura del proyecto 22
5. Modelado de la planta en Unity 5 24
   - 5.1. Introducción a Unity 24
   - 5.2. Importación de modelos 3D 26
     - 5.2.1. Modelo 3D del robot ABB IRB 120 26
     - 5.2.2. Modelo 3D de la minifábrica ICAI 26
   - 5.3. Cintas transportadoras 33
     - 5.3.1. Cintas transportadoras rectas 33
     - 5.3.2. Cintas transportadoras curvas 36
   - 5.4. Comportamiento del robot 38
     - 5.4.1. Cinemática directa 39
     - 5.4.2. Implementación 40
   - 5.5. Preparación de la escena para RV 41
     - 5.5.1. SteamVR 41
   - 5.6. Comunicaciones 42
     - 5.6.1. Hilo de comunicaciones con RobotStudio TCP/IP . . . 44
     - 5.6.2. Hilo de comunicaciones con RobotStudio UDP/IP . . . 48
     - 5.6.3. Hilo de comunicaciones con interfaz PLCSIM TCP/IP 50
   - 5.7. Sistemas de coordenadas 50
     - 5.7.1. Sistema de coordenadas “Mundo” 51

5.7.2. Sistema de coordenadas “Herramienta” 52
5.8. Preparación de datos para RobotStudio 53
5.9. Objeto objetivo del robot 57
5.9.1. Gizmo de traslación 58
5.9.2. Gizmo de rotación 60
5.9.3. Resultado final 62
5.10. Controles de realidad virtual 64
5.10.1. Controles de programación del robot en RV 64
5.10.2. Controles de movimiento en RV 68
5.10.3. Pupitre de control de la planta 70
5.11. Programa de entrenamiento de operarios 70
6. ABB IRB 120 76
6.1. Introducción al manipulador 76
6.2. Cinemática inversa 77
6.3. Introducción a RobotStudio 79
6.4. Alcanzabilidad de un punto 81
6.5. Configuración de la simulación de RobotStudio 82
6.6. Tarea principal 84
6.7. Tarea secundaria 90
7. PLC Virtual 92
7.1. Programación del PLC 92
7.1.1. Introducción 92
7.1.2. Grafcet de emergencia 92
7.1.3. Grafcet de gestión de modos 94
7.1.4. Grafcet de cintas transportadoras 95
7.2. Desarrollo interfaz PLCSIM 96
8. Instalación del Oculus Rift 98
8.1. Hardware 98
8.2. Calibración del Oculus Rift 99
9. Resultados y conclusiones 102

II. Presupuesto **108**

1. Mediciones . . . 109
   1.1. Componentes . . . 109
   1.2. Equipo y herramientas . . . 109
   1.3. Software . . . 109
   1.4. Ingeniería . . . 110
2. Precios unitarios . . . 111
   2.1. Componentes . . . 111
   2.2. Equipo y herramientas . . . 111
   2.3. Software . . . 111
   2.4. Ingeniería . . . 112
3. Sumas parciales . . . 113
   3.1. Componentes . . . 113
   3.2. Equipo y herramientas . . . 113
   3.3. Software . . . 113
   3.4. Ingeniería . . . 114
4. Presupuesto general . . . 115

III. Código Fuente **116**

1. Unity . . . 117
   1.1. IRB120.cs . . . 117
   1.2. VRController.cs . . . 118
   1.3. ControllerTip.cs . . . 123
   1.4. ToolCoordSystem.cs . . . 125
   1.5. WorldCoordSystem.cs . . . 127
   1.6. ControllerHints.cs . . . 129
   1.7. Communications.cs . . . 129
   1.8. StraightConveyorBelt.cs . . . 133
   1.9. CurvedConveyorBelt.cs . . . 134
   1.10. AirValve.cs . . . 135

1.11. Tutorial.cs 136
2. Rapid 138
2.1. Tarea principal 138
2.2. Tarea secundaria 142
3. Interfaz PLCSIM - Unity 143
3.1. AutoVR.cs 143
3.2. PLCInstance 145

Índice de figuras

1. Televisor 3D con tecnología de gafas activas 14
2. Oculus Rift Developer Kit 1 15
3. Oculus Rift Consumer Version 1 16
4. Solución EON Idesk, basada en monitor 3D 17
5. Sistema de EON basada en tecnología CAVE 17
6. Proyecto IVRE para la programación de robots industriales 18
7. Proyecto HapticVive para la simulación de respuesta háptica 19
8. Arquitectura de la solución desarrollada 22
9. Interfaz de Unity 5 24
10. Opciones de importación del ABB IRB120 en Unity 26
11. Modelo 3D de la minifábrica en AutoCAD 27
12. Menú exportar en FBX desde AutoCAD 27
13. Ventana *Profiler* de Unity 28
14. Rendimiento de Unity con el modelo exportado desde AutoCAD 29
15. Modelo 3D de la minifábrica en 3DS Max 30
16. Cambio de unidades en el menú *Units Setup* 30
17. Cambio de unidades en el menú *System Units Setup* 31
18. Opciones del menú *FBX Export* 31
19. Opciones de importación de la minifábrica en Unity 32
20. Rendimiento de Unity con el modelo exportado desde 3DS Max 32
21. Comparativa de rendimiento de Unity 33
22. Primera implementación de cinta transportadora con el método *AddForce()* 34
23. Cinta transportadora en estado inicial 34
24. Cinta transportadora, teleportada hacia atrás 35
25. Cinta transportadora, movida con el método *MovePosition()* 35
26. Cinta transportadora curva en estado inicial 36

27. Cinta transportadora curva tras aplicar la rotación negativa de θ^o . . . 37
28. Cinta transportadora curva en estado final 38
29. Colocación de ejes en eslabones con el criterio de Denavit y Hartenberg [28] 39
30. Jerarquía del robot 40
31. Comprobación de SDK's cargados 42
32. Diagrama de flujo del ciclo de vida de un script en Unity 43
33. Comparativa del sistema de coordenadas de Unity y RobotStudio . . . 50
34. Diferencias entre sistemas de coordenadas zurdos y diestros 50
35. Sistema de coordenadas Mundo 52
36. Parentesco del sistema de coordenadas "Herramienta" con respecto al mando 52
37. Sistema de coordenadas "Herramienta", solidario al mando 53
38. *GameObject* Origen colocado en la base del modelo del IRB 120 54
39. Rotación representada por la matriz R 55
40. Gizmo de traslación 58
41. Gizmo de rotación 61
42. ABB IRB 120 una vez alcanzado el objetivo 62
43. Leyenda de controles del Oculus Rift [52] 65
44. Marcador de punto de ruta guardado 67
45. Textos de ayuda para enseñar controles 68
46. Suelo teleportable de la planta 69
47. Resultado final del arco de teleportación 69
48. Pupitre de control de las cintas transportadoras de la planta 70
49. Representación de válvula de aire comprimido 72
50. Textos de ayuda para orientar al operario 73
51. Máquina de estados del tutorial 73
52. Resolución, precisión y repetibilidad [59] 76
53. Ejes del brazo robótico ABB IRB 120 77
54. Representación visual del objetivo del método del gradiente 78
55. Ventana *RAPID* de RobotStudio 80

56. Ventana *Simulation* de RobotStudio 80
57. Valor de cf_i para cada cuadrante 82
58. Configuración del *Run Mode* en *Continuous* 83
59. Configuración de la opción *PC Interface* para habilitar las comunicaciones *socket* 83
60. Configuración de la opción Multitasking para poder ejecutar dos tareas de forma simultánea 84
61. Grafcet de emegencia 93
62. Llamada a grafcet de emegencia en OB1 93
63. Grafcet de gestión de modo automático manual 94
64. Llamada a grafcet de gestión de modo automático manual en OB1 . . . 95
65. Grafcet de cintas transportadoras 95
66. Llamada a grafcet de cintas transportadoras en OB1 96
67. Hardware del Oculus Rift 98
68. Pantalla *Set Floor Position in VR* 99
69. Pantalla *Initiate Sensor Tracking* 99
70. Pantalla *Room Orientation* 100
71. Pantalla *Set your boundaries* 100
72. Usuario programando el robot en realidad virtual 102

Índice de tablas

1. Rendimiento gráfico con el modelo exportado de AutoCAD 29
2. Rendimiento gráfico con el modelo exportado de 3DS Max 32
3. Capacidad de carga del robot IRB 120 76
4. Especificaciones mecánicas de las articulaciones del robot IRB 120 . . . 77
5. Estructura de variables de tipo *robtarget* 81
6. Diferencia de rendimiento entre USB 2.0 y 3.0 98

1. Mediciones de componentes 109
2. Mediciones de equipo y herramientas 109
3. Mediciones de software 109
4. Mediciones de ingeniería 110
5. Precios unitarios de componentes 111
6. Precios unitarios de equipo y herramientas 111
7. Precios unitarios de software 111
8. Precios unitarios de ingeniería 112
9. Suma parcial de componentes 113
10. Suma parcial de equipo y herramientas 113
11. Suma parcial de software 113
12. Suma parcial de ingeniería 114
13. Presupuesto general del proyecto 115

Índice de códigos

1. *StraightConveyor.cs* - Implementación de la cinta transportadora recta 36
2. *CurvedConveyor.cs* - Implementación de la cinta transportadora curva . 38
3. *IRB120.cs* - Script de movimiento del robot 41
4. Pséudocódigo de métodos de comunicación síncronos 42
5. *Communications.cs* - Inicialización de hilo de comunicación 44
6. *Communications.cs* - Forma correcta de cerrar los hilos al parar la simulación 45
7. *Communications.cs* - Configuración de la conexión 46
8. *Communications.cs* - Envío de datos a RobotStudio 47
9. *Communications.cs* - Recepción de datos a RobotStudio 48
10. *Communications.cs* - Inicialización de hilo de comunicación 49
11. *WorldCoordinateSystem.cs* - Sistema de coordenadas "Mundo" 51
12. *ToolCoordinateSystem.cs* - Sistema de coordenadas "Herramienta" . . . 52
13. *VRController.cs* - Cálculo de los ángulos de herramienta 55
14. *VRController.cs* - Preparación de datos para RobotStudio 56
15. *ControllerTip.cs* - Inicialización de la manipulación del gizmo de traslación 59
16. *ViveController.cs* - Cálculo de los ángulos de herramienta 60
17. *ControllerTip.cs* - Inicialización de la manipulación del gizmo de rotacion 61
18. *ControllerTip.cs* - Manipulación del gizmo de rotación 62
19. *VRController.cs* - Recálculo del objetivo tras modificarlo 63
20. *VRController.cs* - Función *GizmoToAngles()* 63
21. *VRController.cs* - Función *CorrectAngle()* 64
22. Ejemplo de detección de botones 65
23. *VRController.cs* - Mover objetivo del robot 65
24. *VRController.cs* - Comprobar alcanzabilidad del objetivo del robot . . 66
25. *VRController.cs* - Guardar punto en ruta del robot 67
26. *VRController.cs* - Ejecutar ruta guardada del robot 67
27. *AirValve.cs* - Válvula de aire comprimido 72
28. *Tutorial.cs* - Máquina de estados del tutorial 73
29. *MainModule.mod* - Inicialización comunicación *socket* 84
30. *MainModule.mod* - Cálculo de alcanzabilidad 85
31. *MainModule.mod* - Función StringToTarget() 86
32. *MainModule.mod* - Función JointTargetToString() 87
33. *MainModule.mod* - Gestión de errores de RAPID 88
34. *MainModule.mod* - Añadir un punto al *path* del robot 88
35. *MainModule.mod* - Ejecución del *path* guardado 89
36. *SecondaryModule.mod* - Muestreo de los ángulos durante el movimiento 90

Memoria

1. Introducción

Este proyecto pretende diseñar un sistema basado en realidad virtual para entrenar un operario en el uso o en el mantenimiento de una planta. Por ejemplo, el operario podrá programar un robot utilizando únicamente el sistema de realidad virtual. Además, en el caso del entrenamiento para un cambio de herramienta, el operario mediante las gafas de realidad virtual podrá desplazarse a lo largo de la planta, localizando los componentes que debe accionar, en el orden adecuado, hasta completar la operación. El sistema será aplicado a la minifábrica ICAI.

2. Estado del arte

2.1. Realidad virtual

La tecnología de diseño asistido por ordenador o CAD (Computer Aided Design) nace a mediados de la década de los 60, y desde entonces ha revolucionado múltiples disciplinas, como la ingeniería, el diseño o la arquitectura. El software CAD ha evolucionado de forma considerable desde sus inicios. Lo que antes eran simples representaciones en dos dimensiones de algún objeto o proyecto, hoy en día son modelos en 3D fotorrealistas, animados e interactivos.

A pesar de los grandes avances que ha sufrido esta disciplina, la forma de experimentar los modelos y entornos en 3D no ha cambiado prácticamente en los últimos 30 años. Lo más habitual es interactuar con ellos utilizando periféricos como un teclado y un ratón, y visualizarlos a través de una pantalla o un monitor. A pesar de que la tecnología de pantallas y monitores ha avanzado en mucho en la última década, en aspectos como la resolución o la fidelidad del color [1], tienen una limitación inherente: no dejan de ser representaciones de un objeto tridimensional en un plano.

Para superar esta limitación, en la última década los televisores 3D se establecieron como un nicho del mercado audiovisual. El principio de funcionamiento de estos televisores es el efecto estereoscópico, que consiste en proyectar dos imágenes distintas tomadas desde distintos puntos de vista, simulando la separación que existe entre los ojos del usuario, y así creando la ilusión de objetos tridimensionales. La implementación más popular de esta tecnología fue la basada en gafas activas (ver Figura 1). Si un televisor estándar proyecta una imagen 60 veces por segundo, un televisor en 3D funciona a 120Hz, proyectando las dos imágenes distintas de forma alterna. De este modo, las gafas activas se sincronizan con la tasa de refresco del televisor y bloquean la visión de un ojo en cada ciclo. Así, se consigue que el usuario perciba una imagen distinta por ojo, creando la ilusión del 3D.

Figura 1: Televisor 3D con tecnología de gafas activas

Sin embargo, esta tecnología ha experimentado un declive en popularidad pronunciado, hasta el punto de que las principales empresas audiovisuales han dejado de emitir contenido en este formato, y los fabricantes de televisores han retirado estos productos de sus catálogos [2]. Aun teniendo en cuenta el fracaso de esta tecnología, ésta era un paso adelante para experimentar contenido en 3D real.

La tecnología de realidad virtual nace de la necesidad de experimentar entornos virtuales de una forma más inmersiva, además de facilitar la interacción con ellos. Aunque la tecnología de realidad virtual ya existía en los años 90, ésta no estaba lo suficientemente avanzada como para ofrecer una buena experiencia al usuario. Además, el coste de esta tecnología era demasiado alto para el consumidor medio, por lo que quedaba reservada para aplicaciones militares e investigación en universidades. Un ejemplo de esta tecnología con un coste elevado son los sistemas CAVE.

Los sistemas CAVE (CAVE Assisted Virtual Enviroment, o entorno asistido de realidad virtual) consisten en un espacio en forma de cubo. En las paredes, suelo y techo del entorno CAVE se proyectan imágenes estereoscópicas que, gracias al uso de gafas activas, se perciben en 3D [3]. El principal inconveniente de esta tecnología es su elevado coste, ya que requieren un espacio dedicado para este uso, la instalación de un proyector por superficie, además de un sistema de cámaras infrarrojas en caso de que se quiera hacer "tracking" o rastrear la posición y orientación de herramientas, para poder interactuar con el entorno virtual. Por este motivo, la realidad virtual no estaba al alcance del mercado de consumo.

Esto no cambiaría hasta Julio del año 2012, cuando el emprendedor californiano Palmer Luckey funda la empresa Oculus VR. En esta fecha Oculus lanza una campaña de crowdfunding para financiar el desarrollo de su producto, el "Oculus Rift Developer Kit 1"(ver Figura 2), unas gafas de realidad virtual o HMD (Head-mounted Display).

Figura 2: Oculus Rift Developer Kit 1

Como su nombre indica, estas gafas de realidad virtual estaban orientadas a desarrolladores, y se caracterizaron por tener una pantalla de 7 pulgadas, con una resolución de 640x800 píxeles por ojo, que el usuario veía a través de dos lentes. La combinación del efecto estereoscópico con las lentes proporcionaba una experiencia en 3D mucho más inmersiva que la de sus principales competidores. Además, era capaz de determinar la orientación de la cabeza del usuario gracias a una IMU (siglas en

inglés de Unidad de Medición Inercial), que obtiene esta información combinando el uso de acelerómetros y giróscopos.

Con un precio de salida de $300, la campaña de financiación fue todo un éxito, recaudando aproximadamente 2,4 millones de dólares, un 975 % del objetivo inicial de $250.000 [4]. Dos años después, en el año 2014, Oculus VR fue adquirida por Facebook por un importe estimado de 2.000 millones de dólares [5]. En un informe redactado por Greenlight Insights, se estima que los ingresos de toda la industria de Realidad Virtual alcancen 7.200 millones de dólares [6].

La última versión del Oculus Rift, la Consumer Version 1 (ver Figura 3), posee la capacidad de determinar la localización de las gafas en el espacio, no solo su orientación. Esto lo hace a través de cámaras infrarrojas que localizan diodos infrarrojos instalados en el dispositivo, además de la IMU. Al contrario que versiones anteriores, tiene dos mandos inalámbricos que también son rastreados por las cámaras, permitiendo una interacción con el entorno muy natural, además de una sensación de inmersión sin precedentes.

Figura 3: Oculus Rift Consumer Version 1

2.2. Entrenamiento de operarios con tecnología de realidad virtual

En la actualidad existen diversas empresas que comercializan soluciones para entrenamiento de operarios mediante tecnología de realidad virtual. Estos servicios sirven para entrenar a operarios en tareas como la operación de paneles de control, tareas de mantenimiento preventivo, que tendrán que realizar mientras la planta esté en funcionamiento; además de tareas de mantenimiento correctivo, en caso de que se produzca alguna avería. La principal ventaja de estos sistemas es que no es necesario tener la planta construida, por lo que se pueden realizar las tareas de entrenamiento en paralelo a las tareas de construcción. Esto implica no invertir tiempo adicional para que los operarios aprendan a manejar el sistema, una vez finalizada la construcción de la planta.

La mayoría de estas soluciones implementan monitores en 3D con gafas activas o la tecnología CAVE, ambas descritas en el apartado anterior. Por ejemplo, la empresa

multinacional norteamericana EON Reality comercializa dos productos distintos para el entrenamiento de operarios [7].

EON Idesk, una solución basada en un monitor 3D con gafas activas (ver Figura 4). Para poder interactuar con la simulación virtual, el usuario sujetará una herramienta con forma de lápiz. Esta herramienta tiene instalados varios diodos infrarrojos, cuya luz es capturada por varias cámaras infrarrojas. Con esta información, el sistema es capaz de calcular la posición y orientación del útil, permitiendo al usuario interactuar de forma natural con el objeto virtual.

Figura 4: Solución EON Idesk, basada en monitor 3D

EON Immersive 3D Training Enviroment (ver Figura 5), desarrollado en colaboración con ExxonMobil Research Qatar, consiste de un sistema CAVE con un mando que utiliza la misma tecnología que en el caso de Idesk, pero que permite un rango de movimientos mayor.

Figura 5: Sistema de EON basada en tecnología CAVE

EON Reality no es la única compañía que ofrece estos sistemas. Empresas como Siemens, Schneider Electric o Immersion tienen productos similares para el entrenamiento de operarios en realidad virtual.

2.3. Realidad Virtual aplicada a robots industriales

La realidad virtual tiene dos principales motivaciones para integrarse en el campo de la robótica industrial. En primer lugar, puede hacer el proceso de programación mucho más intuitivo, si se realiza con controladores localizados en el espacio que imiten el movimiento de la mano del usuario. En segundo lugar, tiene la ventaja de la seguridad. Los robots industriales actuales pueden moverse a velocidades elevadas, como por ejemplo el ABB IRB 120, que dependiendo de la articulación, puede moverse desde 250º/s hasta 600º/s [8]. Esto hace que la presencia del operario en el volumen de trabajo del robot suponga un peligro para la seguridad, que se elimina con la simulación en realidad virtual.

La multinacional suiza ABB, uno de los principales fabricantes de robots industriales, incluye soporte para realidad virtual en Robot Studio [9], su software de simulación y programación de robots y plantas industriales. En concreto, el usuario es capaz de visualizar la simulación del proyecto utilizando unas gafas Oculus Rift CV1. Esto permite que el usuario pueda detectar ángulos que puedan resultar peligrosos, puntos singulares, movimientos irrealizables, etc. Además, el usuario puede moverse por la planta virtual utilizando el joystick de un mando de consola de videojuegos.

En relación a este campo, existen proyectos como IVRE (Immersive Virtual Robotics Environment o entorno virtual inmersivo de robótica), desarrollado por el Laboratory for Computational Sensing and Robotics de la universidad Johns Hopkins [10]. Este proyecto utiliza un Oculus Rift DK1 para visualizar el modelo del robot en 3D, y unos controladores Razer Hydra para hacer el "tracking" de la mano del operario. Con este controlador, el usuario es capaz de mover la herramienta del robot virtual con su mano.

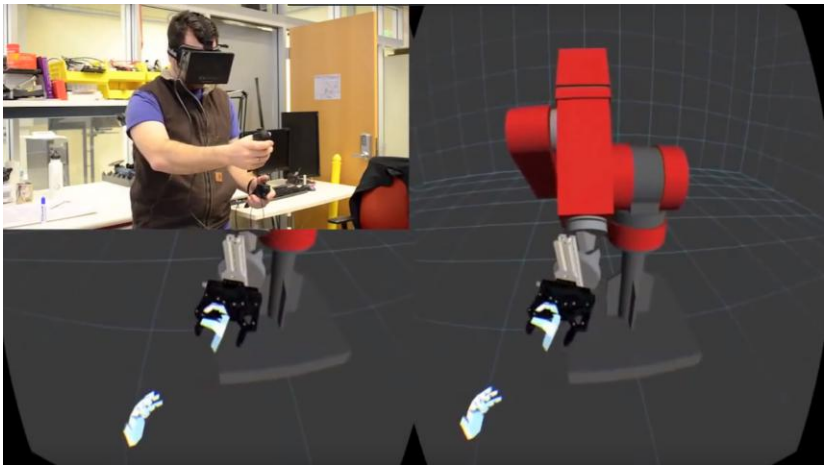


Figura 6: Proyecto IVRE para la programación de robots industriales

Otro proyecto relacionado es desarrollado por la empresa Suncom Garage [11]. Este proyecto combina el robot colaborativo YuMi de ABB con el mando inalámbrico del sistema de realidad virtual HTC Vive. De forma análoga a los controladores del Oculus Rift CV1, el sistema conoce la posición y orientación de la mano del controlador del

HTC Vive en todo momento, y el robot YuMi replicará los movimientos del operario. Esta aplicación puede ser interesante para implementar telepresencia en labores de riesgo extremo, como la desactivación de bombas o gestión de residuos nucleares.

El proyecto HapticVive, desarrollado por la universidad de Queen's en Belfast, da respuesta al principal problema de la realidad virtual, la falta de respuesta háptica [12]. La respuesta háptica consiste en la realimentación de fuerza que sentimos al tocar un objeto. Este proyecto utiliza un robot colaborativo Baxter, desarrollado por la empresa Rethink Robotics, para simular la resistencia que ofrece una caja al ser empujada, como se puede apreciar en la Figura 7.

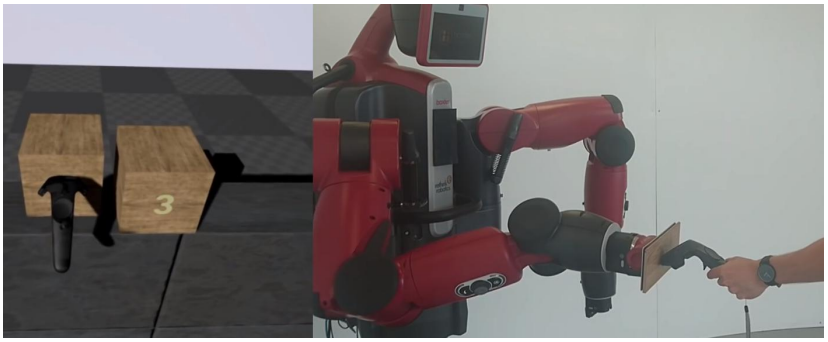


Figura 7: Proyecto HapticVive para la simulación de respuesta háptica

3. Objetivo del proyecto

El objetivo de este proyecto es que la solución de software desarrollada sea capaz de emplear la tecnología de realidad virtual para el entrenamiento de operarios de plantas industriales, además de la programación de un manipulador robótico. Se simulará la minifábrica ICAI en un entorno de realidad virtual, se desarrollará un módulo de entrenamiento de operarios para tareas de operación y mantenimiento. Además, se desarrollará otro módulo de programación del robot ABB IRB 120 con controles en realidad virtual, para ver si es factible y aporta alguna ventaja con respecto a los métodos de control y programación del robot tradicionales. En ambos módulos, el sistema debe de ser sencillo de utilizar, para que el usuario pueda centrarse en el proceso de aprendizaje, y no en el manejo del sistema de realidad virtual.

4. Arquitectura del proyecto

Aunque se desarrollarán en detalle en cada apartado, es necesario hacer una breve introducción de cómo está estructurado el proyecto, compuesto de tres partes principales (Figura 9):

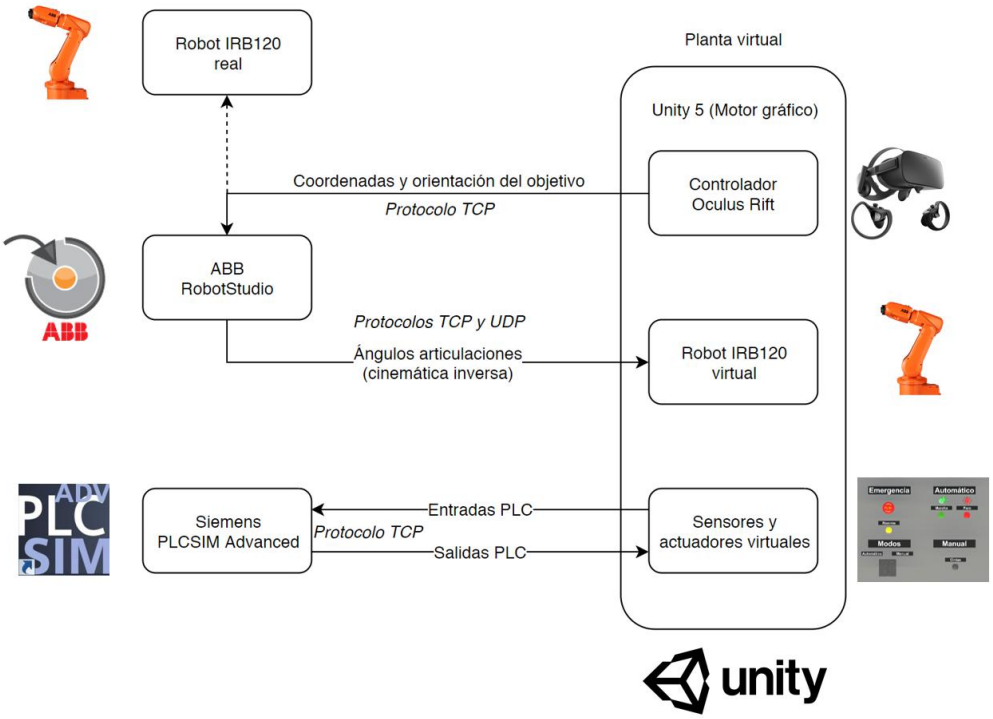


Figura 8: Arquitectura de la solución desarrollada

- Simulación de la planta en realidad virtual, desarrollada con Unity 5. Este será el entorno 3D que el usuario visualizará a través de las gafas de realidad virtual. Los actuadores principales de esta planta serán las cintas transportadoras de palés y los manipuladores robóticos. Aunque su comportamiento esta programado en Unity, es preciso que sean controlados por un PLC virtual y por RobotStudio, respectivamente. Por ello, esta simulación de comunicará con PLCSIM Advanced por protocolo TCP y con RobotStudio por los protocolos TCP y UDP.
- Simulación del robot, ejecutada con ABB RobotStudio. Será la encargada de resolver la cinemática inversa del robot, y se comunica con Unity por protocolo TCP y UDP. Además, el código ejecutado en RobotStudio también puede ser ejecutado en una controladora de un robot físico.
- Simulación del PLC, implementada con Siemens PLCSIM Advanced. Unity le transmite las entradas de la planta al PLC virtual, y éste devuelve las salidas de acuerdo a su programación, utilizando el protocolo TCP.

5. Modelado de la planta en Unity 5

5.1. Introducción a Unity

Unity es un motor gráfico de videojuegos, que permite desarrollar aplicaciones 3D y 2D para diversas plataformas, como ordenadores, consolas, o *smartphones*. Es uno de los motores gráficos más accesibles, debido a su flujo de trabajo *drag and drop*, como a su capacidad de *scripting* en el lenguaje C# **??**. Es ésta capacidad de *scripting* lo que lo hace tan flexible, ya que permite, por ejemplo, comunicarse con otras aplicaciones a través de protocolos de red, ventaja que se aprovechara en este proyecto.

Figura 9: Interfaz de Unity 5

La interfaz de Unity (Figura 9) se compone de:

- Barra de herramientas. Utilizando el botón con el icono del *play*, se puede arrancar la simulación de Unity.
- Ventana de proyecto. En esta ventana se encuentran todos los componentes del proyecto, organizados por carpetas. Entre estos elementos, se pueden encontrar: *scripts* de comportamiento, modelos 3D, texturas, sonidos, etc.
- Vista de escena. La vista de escena permite visualizar, moverse y editar la escena de la simulación.
- Ventana jerarquía. En esta ventana se encuentran todos los objetos que se muestran en la vista de escena, además de su jerarquía. Por ejemplo, existe un objeto vacío llamado "Robots", que a su vez tiene 4 hijos, uno por cada objeto *ABB IRB 120* (en la escena hay 4 robots). A su vez, los objetos *ABB IRB 120* tendrán como hijos todos los componentes del modelo 3D.

- Ventana del inspector. En esta ventana se muestran todos los "componentes" de cada objeto mostrado en la vista de escena. Estos componentes tienen distintas funciones. Por ejemplo, todos los objetos tienen un componente *Transform* o transformada, que almacena la posición del objeto, su rotación y su escala. Si se quisiese que un objeto particular se comportase como un sólido rígido, basta con añadir un componente de tipo *Rigidbody* al objeto en esta ventana. Además, si se le quiere asignar un *script* de comportamiento, tambien basta con arrastrarlo desde la ventana de proyecto a la ventana del inspector del objeto.

El flujo de trabajo de Unity es el siguiente:

- Arrastrar los objetos desde la ventana de proyecto hasta la vista de escena.
- Ajustar la jerarquía de los objetos en la ventana de jerarquía, teniendo en cuenta las posiciones relativas de los objetos.
- Asignarle componentes y *scripts* de comportamiento a los objetos en la ventana del inspector.
- Darle al botón con el icono de *play* en la barra de herramientas, para arrancar la simulación.

5.2. Importación de modelos 3D

5.2.1. Modelo 3D del robot ABB IRB 120

La importación del modelo 3D del robot ABB IRB 120 es bastante sencilla. ABB proporciona el modelo 3D en su página web [13], en varios formatos. Utilizando el formato *Google Sketchup*, Unity es capaz de reconocer el modelo 3D directamente, sin necesidad de convertirlo a otro formato.

El único problema encontrado con ese modelo es la escala. En la ventada 2D del editor de Unity, es muy complicado tener una sensación de escala realista. Al visualizar el modelo en 3D utilizando el Oculus Rift por primera vez, éste medía más de 100 metros de alto. En las opciones de importación del modelo de Unity (ver Figura 10), es necesario hacer la conversión de pulgadas a metros, haciendo los siguientes cambios:

- Unit conversion: 1 Meters = 0,03937008.
- Scale factor: $0,001$.
- Use file Scale: activado.

Una vez hechas estas correcciones, el modelo ya estará a escala 1:1, lo cual es fácil de comprobar utilizando un HMD de realidad virtual como el Oculus Rift.

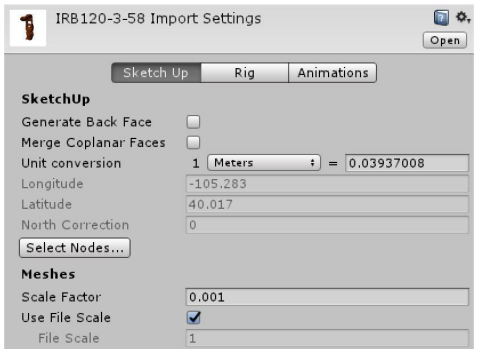


Figura 10: Opciones de importación del ABB IRB120 en Unity

5.2.2. Modelo 3D de la minifábrica ICAI

El diseño del modelo 3D de la minifábrica ICAI está fuera del alcance de este proyecto. Afortunadamente, gracias a alumnos de años anteriores, se dispone de una recreación exacta desarrollada en el software AutoCAD (Figura 11).

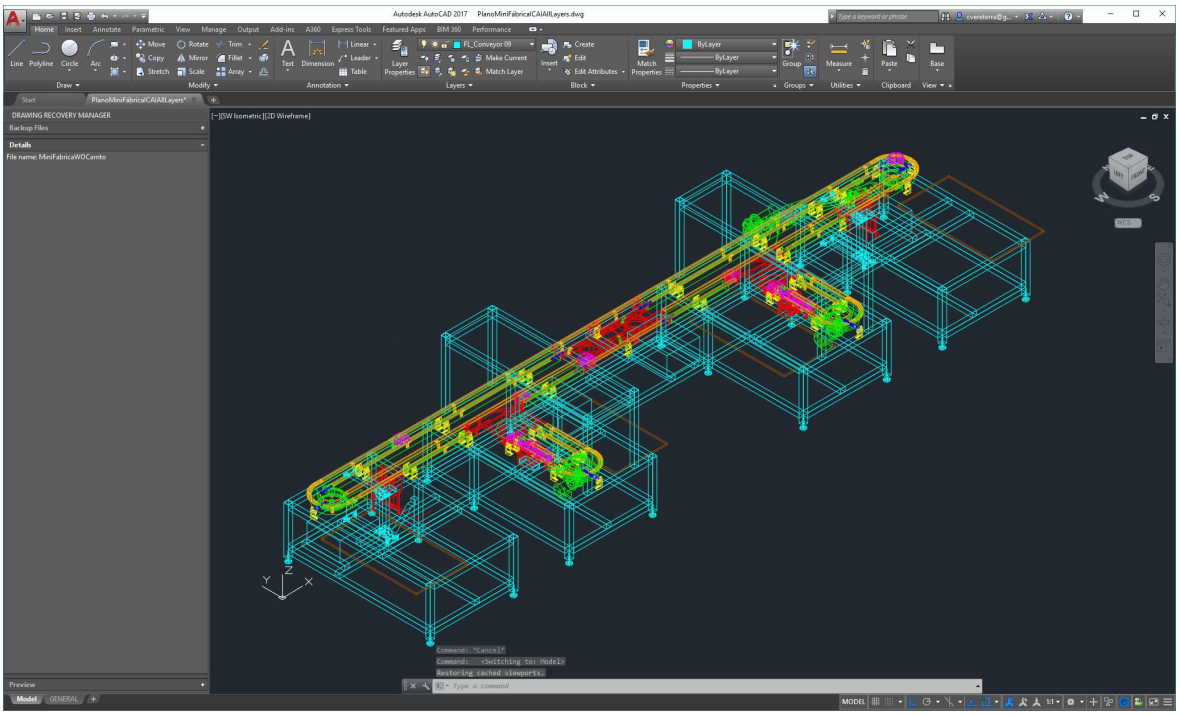


Figura 11: Modelo 3D de la minifábrica en AutoCAD

AutoCAD utiliza un formato de archivo con la extensión '.dwg' (del inglés DraWinG, o plano) para almacenar sus archivos en 3D [14]. Este formato, desarrollado por Autodesk, no es compatible con Unity, como se puede comprobar en la documentación oficial [15]. Esto implica que no se podrá importar directamente, sino que será necesario convertirlo a otra extensión. Entre los formatos compatibles, se encuentra '.fbx', un formato desarrollado también por Autodesk. AutoCAD permite exportar directamente a este formato, así que esta fue la primera opción. El menú de exportar a FBX se muestra en la Figura 12.

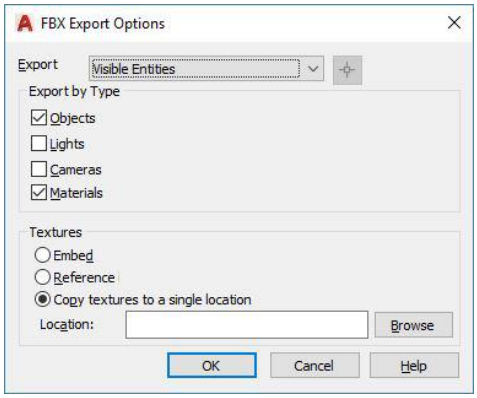


Figura 12: Menú exportar en FBX desde AutoCAD

Una vez exportado, basta con mover el archivo a la carpeta Assets del proyecto, y arrastrarlo a la ventana del Editor o al menú de Jerarquía de Unity. Sin embargo,

al colocar el modelo de la minifábrica exportado en '.fbx' en la escena y visualizarlo a través de un HMD de realidad virtual, el usuario experimentaba una sensación de mareo, impidiendo que pudiese utilizar el sistema de realidad virtual más de unos segundos.

La principal causa de mareos y náuseas en RV es un *framerate* o tasa de fotogramas demasiado baja. Este efecto está suficientemente documentado, y se produce al haber una discordancia entre los estímulos visuales y los estímulos del sistema vestibular, localizado en el oído interno, que se encarga del control del equilibrio [16]. Unity dispone de una herramienta llamada *Profiler*, y su función es la de optimizar la aplicación que se está desarrollando. Esta herramienta muestra cuanto tiempo es invertido en realizar distintas tareas del juego [17]. Permite analizar el rendimiento de la CPU, GPU, memoria, tiempo de renderizado, etc.

Al abrir esta ventana, el motor gráfico empezará a capturar información sobre el rendimiento, que luego podrá ser exportada para posterior procesamiento. Tras unos segundos capturando datos, los resultados son evidentes. En la Figura 13 se aprecia que la mayor parte del tiempo de computación se dedica al renderizado de la escena (color verde).

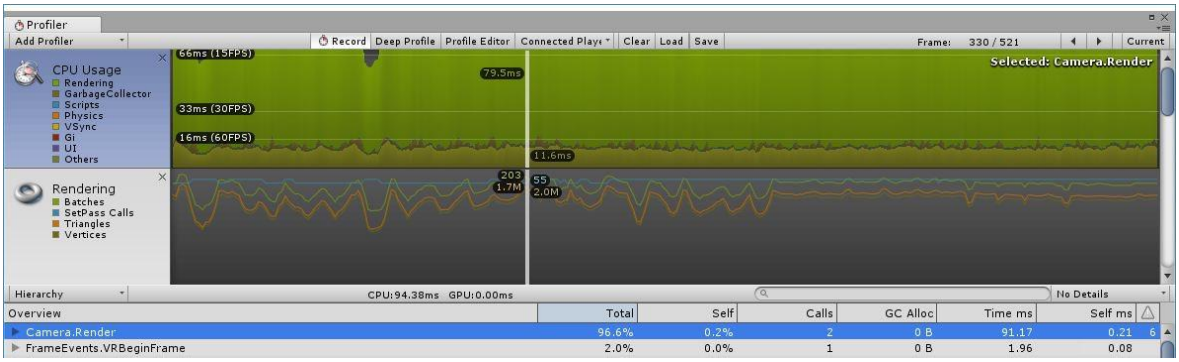


Figura 13: Ventana *Profiler* de Unity

La siguiente prueba consistió en capturar datos en dos estados distintos, mirando hacia el modelo 3D de la minifábrica, y no haciéndolo. El motivo de realizar esta prueba es que los motores gráficos no renderizan aquellos objetos que no están en el campo de visión de la cámara, para ahorrar recursos computacionales. De esta forma, si existe una diferencia notable entre la tasa de fotogramas al mirar al modelo y al no hacerlo, se podrá concluir que el modelo 3D es la causa del problema.

En la Figura 14 y la Tabla 1 se muestran los resultados, que son concluyentes. Cuando el modelo no se encuentra a la vista, la media de fotogramas por segundo es de 142,06, con una desviación típica de 4,86 fps. Al rotar la cámara (el HMD de realidad virtual) hacia el modelo (en la gráfica ocurre aproximadamente en el fotograma número 1000), los fps bajan hasta los 16, menos de la quinta parte requerida para proporcionar una experiencia de RV satisfactoria.

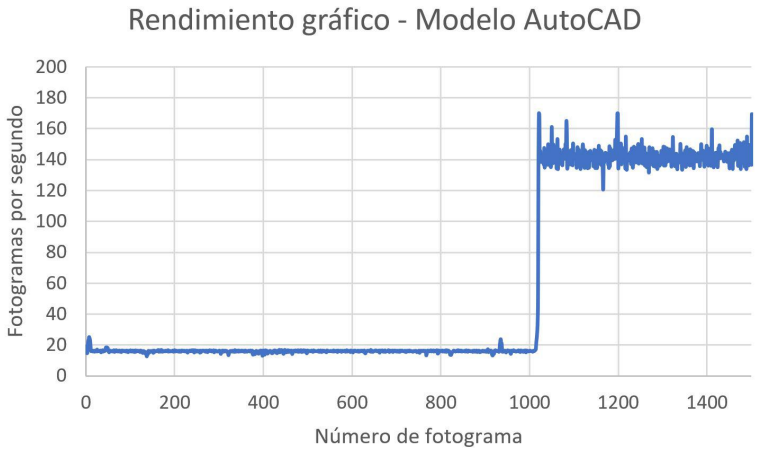


Figura 14: Rendimiento de Unity con el modelo exportado desde AutoCAD

	Minifábrica visible	Minifábrica no visible
Media	16,00 fps	142,06 fps
Desviación típica	0,87 fps	4,86 fps

Tabla 1: Rendimiento gráfico con el modelo exportado de AutoCAD

De esta forma, se concluye que el modelo exportado a FBX desde AutoCAD no es apto para esta aplicación, donde la fluidez es crucial, por lo que es necesario encontrar una alternativa.

Este problema de rendimiento se debe a que AutoCAD no es un software de modelado 3D para videojuegos, sino un software CAD para aplicaciones de diseño industrial donde la precisión y la fidelidad son cruciales, sobre todo en campos como el prototipado rápido o impresión 3D.

Una alternativa de Autodesk, la misma compañía que desarrolla AutoCAD, es utilizar 3DS Max, un programa especializado en modelado y animación para videojuegos [18].

3DS Max no es un programa gratuito, sino que tiene un modelo de negocio de suscripción [19]. Sin embargo, para esta aplicación se ha utilizado la versión de prueba gratuita válida durante 30 días. Al estar desarrollado por Autodesk, es compatible con archivos '.dwg', por lo que no será realizar ninguna conversión para abrirlo, como se muestra en la Figura 15.

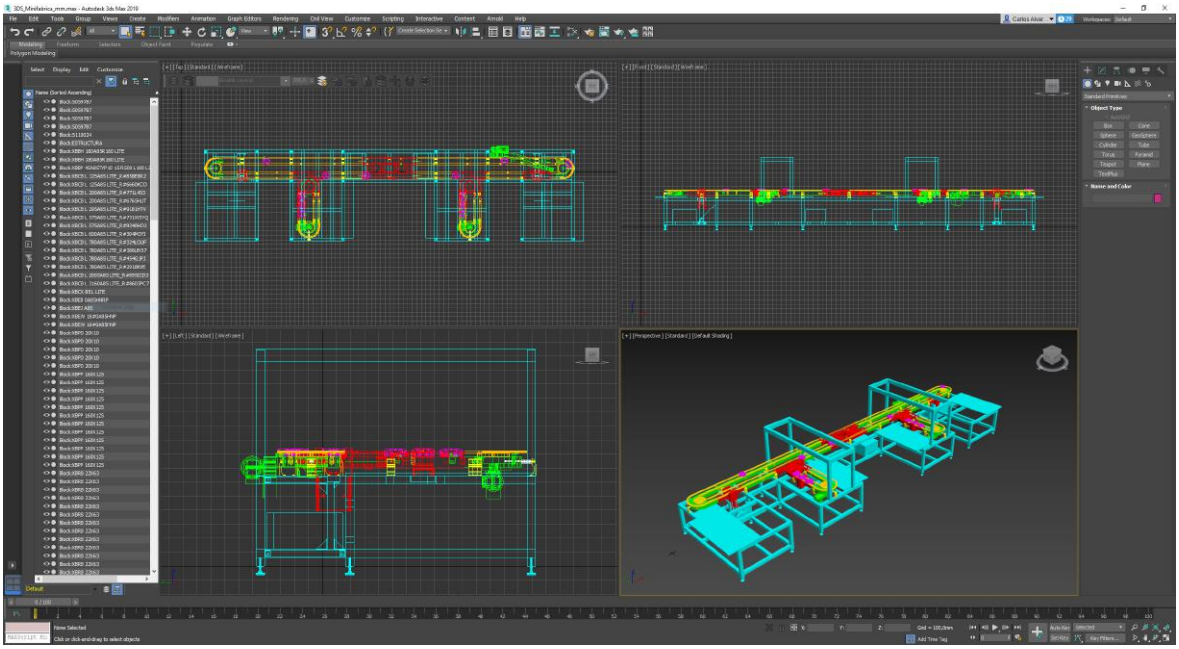


Figura 15: Modelo 3D de la minifábrica en 3DS Max

El programa utiliza el sistema de unidades imperial por defecto, por lo que no interpretará todas las dimensiones en mm, sino que lo hará en pies y en pulgadas. Si se exportase el modelo sin corregir este error, las dimensiones del modelo serían incorrectas. Para ello, es necesario cambiar al sistema métrico decimal en el menú *Units Setup*, utilizando la unidad de mm (ver Figura 16).

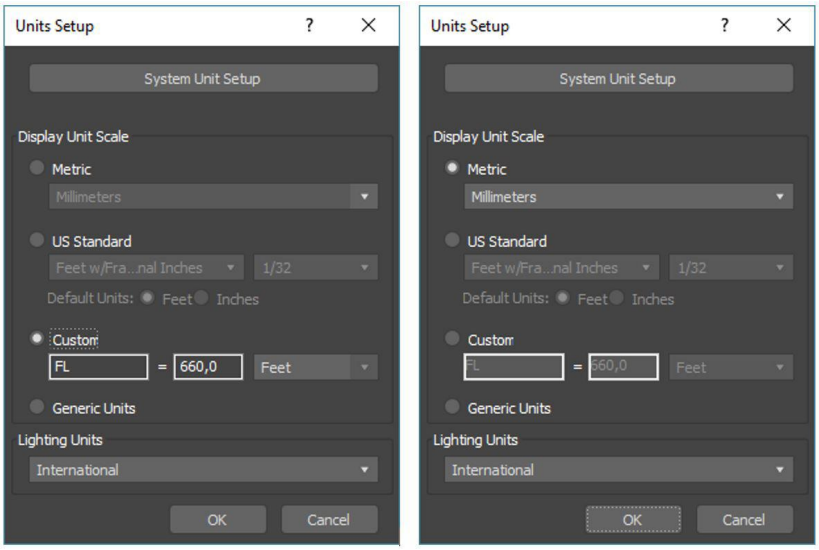


Figura 16: Cambio de unidades en el menú *Units Setup*

No basta con cambiar el sistema de unidades en el menú *Units Setup*, sino que también es necesario cambiarlo en el menú *System Units Setup* (Figura 17).

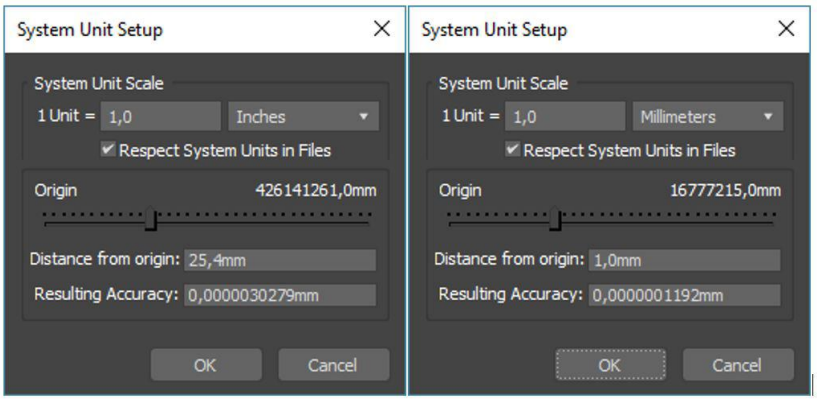


Figura 17: Cambio de unidades en el menú *System Units Setup*

Una vez realizados estos cambios, ya se podrá exportar el modelo con las medidas correctas. Para ello, basta con ir al menú *FBX Export*, y comprobar por última vez que las unidades de la escena están en milímetros, como se puede apreciar en la Figura 18. Es importante dejar desactivada la opción de escala automática, ya que si no, las medidas no serán las correctas al importar el modelo a Unity.

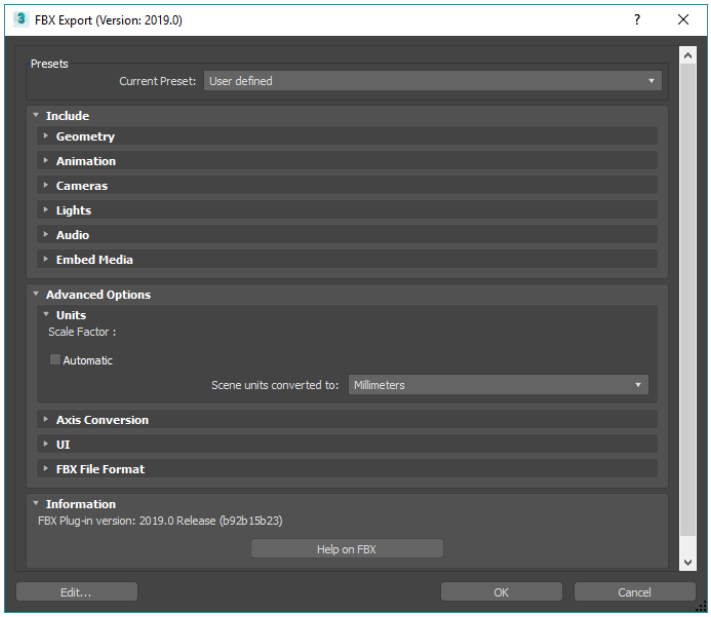


Figura 18: Opciones del menú *FBX Export*

Una vez importado el archivo '.fbx' resultante en la escena de Unity, es importante activar la opción *Use File Scale*, para que el modelo esté a escala 1:1 (Figura 19). También ayuda activar la opción *Optimize Mesh*, que optimiza el mallado del modelo para reducir los tiempos de acceso de la GPU [20].

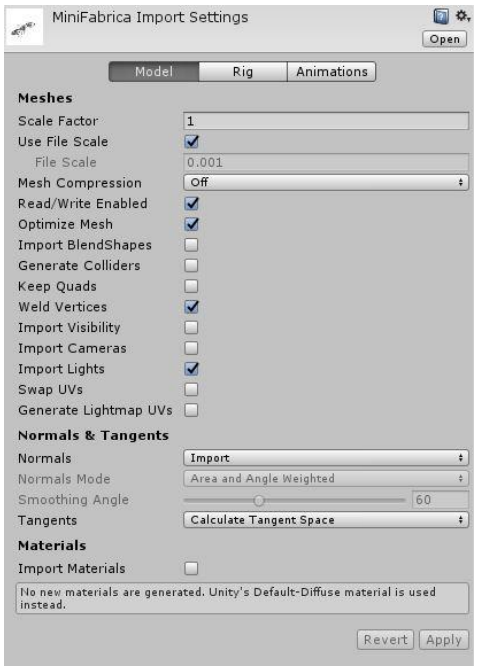


Figura 19: Opciones de importación de la minifábrica en Unity

En la Figura 20 y en la Tabla 2 se muestran los resultados de la herramienta *Profiler* tras exportar el modelo desde 3DS Max.

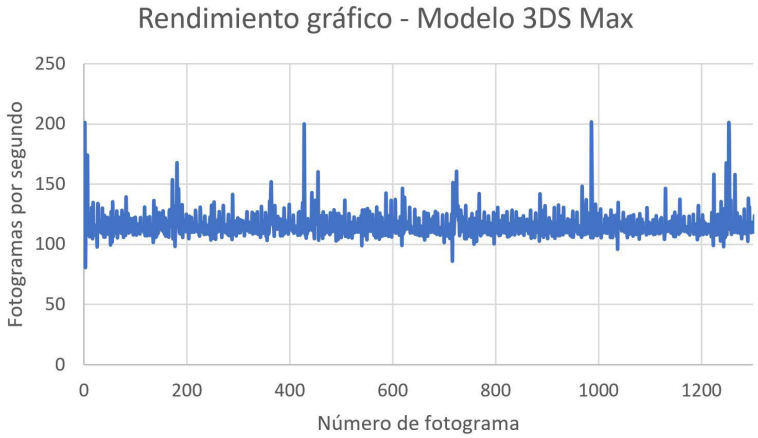


Figura 20: Rendimiento de Unity con el modelo exportado desde 3DS Max

	Minifábrica visible
Media	115,80 fps
Desviación típica	9,87 fps

Tabla 2: Rendimiento gráfico con el modelo exportado de 3DS Max

Los resultados son muy positivos, ya que se ha pasado de una tasa de fotogramas de 16 fps, a unos 115,8 fps de media, lo que supone un aumento de rendimiento de más del 600 %. Además, estamos por encima de la tasa de fotogramas recomendada de 90 fps, como se puede comprobar en la comparativa de la Figura 21.

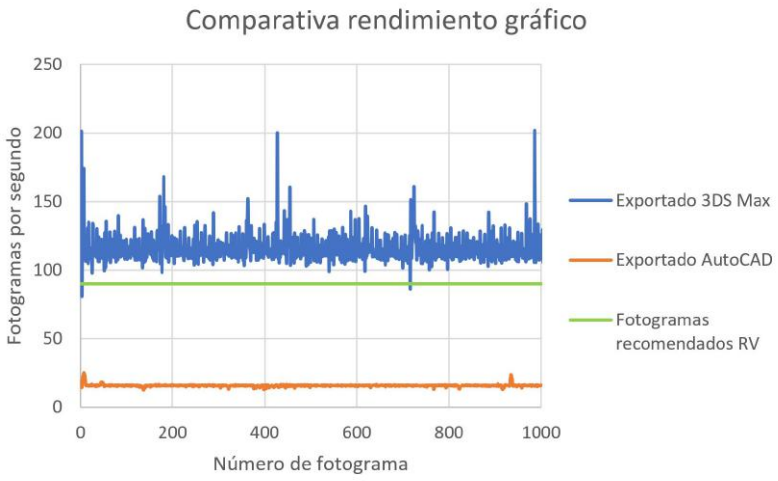


Figura 21: Comparativa de rendimiento de Unity

5.3. Cintas transportadoras

5.3.1. Cintas transportadoras rectas

Para modelar las cintas transportadoras de la minifábrica, se utilizarán prismas rectangulares (compuestos a base de un cubo tridimensional con la escala cambiada) con el ancho y largo del recorrido de la cinta. A pesar de que el modelo 3D de la minifábrica está importado, es importante destacar que sus componentes no tienen un componente de tipo *Box Collider* asociado. Los componentes de tipo *Box Collider* se encargan de comunicarle al motor de físicas que calcule las colisiones con otros objetos sólidos [21]. Si el modelo de la planta no tiene *Colliders* asociados, no se calcularán las colisiones con los distintos elementos estructurales de planta. Esto sería demasiado caro computacionalmente hablando, complicaría innecesariamente la simulación y podría tener un impacto negativo en la tasa de fotogramas por segundo. Por lo tanto, sólo se incluirá componentes de tipo *Box Collider* a las bases de las cintas.

Para modelar los palés, basta con añadirles un componente *Box Collider*, como a la base de las cintas, y un componente *Rigidbody*, para indicarle al motor de físicas que el objeto, con su *collider*, es un sólido rígido [22]. Además, este componente permite aplicar fuerzas al objeto, ya sean externas, la gravedad, etc. para permitir que se muevan de forma realista.

La primera solución adoptada para implementar las cintas transportadoras consiste en añadir una fuerza al objeto que se trata de mover, en su centro de gravedad, cuando

éste entra en contacto con la cinta, como se muestra en la Figura 22. Esto se puede implementar utilizando la función *AddForce()*, de la clase *Rigidbody*, que tiene como argumento un vector que represente la dirección, el sentido y la magnitud de la fuerza [23].

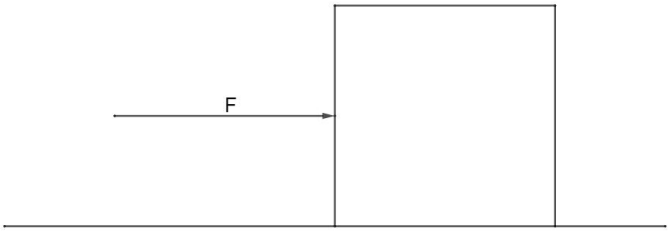


Figura 22: Primera implementación de cinta transportadora con el método *AddForce()*

Los resultados de esta primera solución no fueron muy buenos. Unity no modela el rozamiento de forma automática, por lo que al aplicar una fuerza a un objeto que está en contacto con otro, se modela como una colisión cada fotograma, con lo cual el objeto no se mueve de forma suave, sino que da "tirones" y "saltos" a lo largo de su recorrido.

Finalmente, el método utilizado resulta algo menos intuitivo, pero el resultado es un movimiento suave a lo largo de la cinta, movido por el motor de físicas y fácil de integrar y encadenar. Este método consiste en aplicar dos transformaciones consecutivas. En primer lugar, se parte de la situación descrita en la Figura 23, donde están representados dos sistemas de coordenadas, el de la cinta transportadora y el del palé.

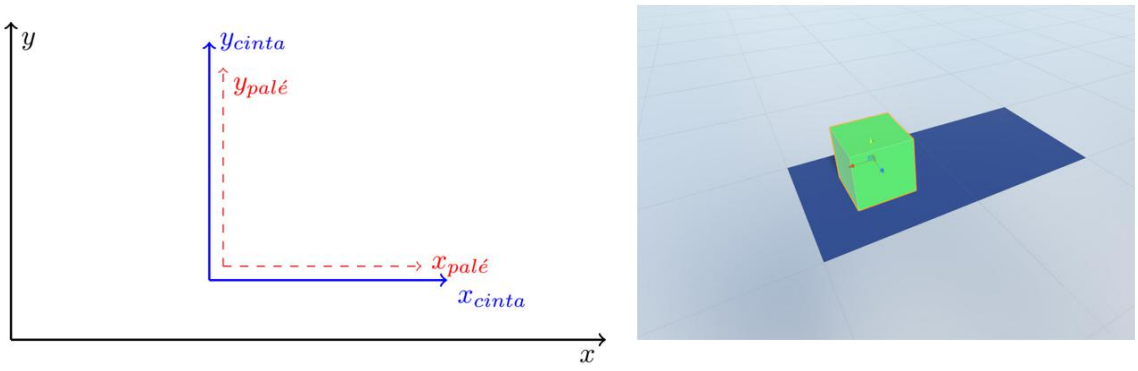


Figura 23: Cinta transportadora en estado inicial

La primera transformación consiste en teleportar la cinta transportadora hacia atrás una distancia Δd, en el eje x:

$$posición'_{cinta} = posición_{cinta} - \Delta d_x$$

Al haber modificado la posición de la cinta transportadora directamente, la posición del palé no se ve afectada, como se puede comprobar en la Figura 24.

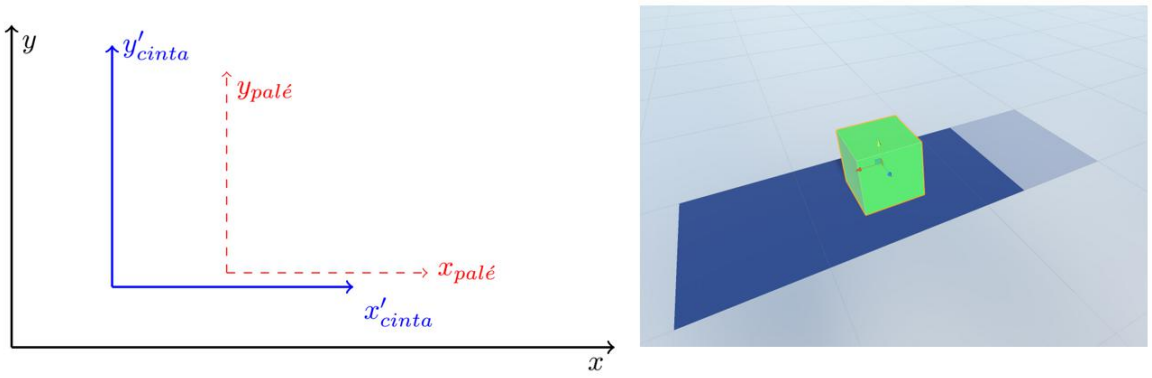


Figura 24: Cinta transportadora, teleportada hacia atrás

Por último, se aplica la función *MovePosition()*. Esta función tiene como argumento un objeto de tipo *Vector3*, que representa la posición final tras la transformación. Como es lógico, este movimiento se aplica en la dirección contraria a la transformación anterior, es decir, hacia delante.

La ventaja de este método con respecto a modificar el componente *Transform* directamente, es que si es llamados en cada *FixedUpdate()*, será renderizado como un movimiento continuo, que además afecta a todos los componentes del tipo *Rigidbody* que estén interactuando con el la cinta [24](en este caso, el palé), como se aprecia en la Figura 25.

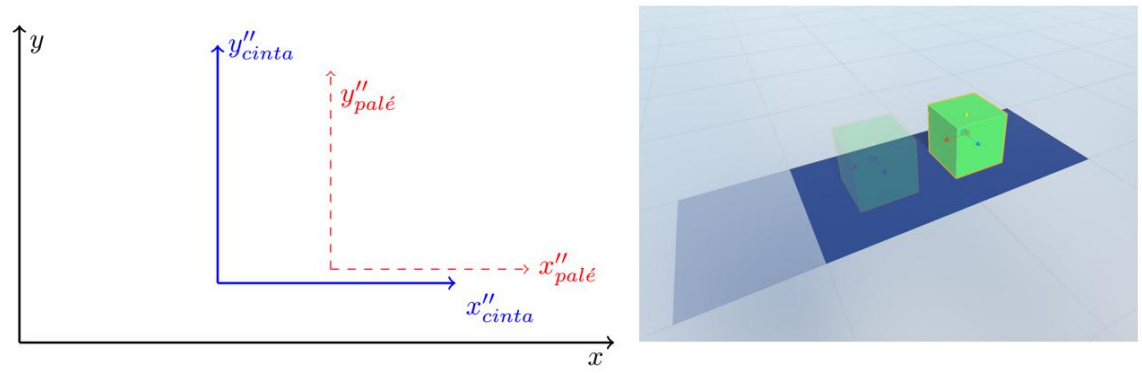


Figura 25: Cinta transportadora, movida con el método *MovePosition()*

La solución implementada se puede apreciar en el Código 1. Como aclaración, todas aquellas funciones que interactúen con el motor de físicas deben de ser ejecutadas en el bucle *FixedUpdate()*, y no en *Update()* como la mayoría de funciones. Esto se debe a que *FixedUpdate()* se ejecuta un número fijo de veces por segundo, mientras que *Update()* se ejecuta cada vez que se renderiza un nuevo fotograma, y el tiempo entre fotogramas es variable e inversamente proporcional al número de fotogramas por segundo.

```
public float speed = 2f; // Movement speed
void FixedUpdate()
{
    if (active)
    {
      //Get belt Rigidbody component
        Rigidbody rigidbody = GetComponent<Rigidbody>();
        //Teleport belt backwards
        rigidbody.position -= transform.forward * speed * Time.deltaTime;
        //Move belt forwards
        rigidbody.MovePosition(rigidbody.position + transform.forward * speed *
            Time.deltaTime);
    }
}
```

Código 1: *StraightConveyor.cs* - Implementación de la cinta transportadora recta

5.3.2. Cintas transportadoras curvas

Para modelar los tramos curvos de la cinta transportadora, en primer lugar es necesario definir una rotación alrededor de un eje. Aunque las rotaciones suelen ser más intuitivas al ser representadas en ángulos de Euler (un ángulo de rotación por eje), las rotaciones en Unity se definen como cuaterniones. Esto se debe a que los ángulos de Euler, para ciertos valores, presenta discontinuidad en la transformación (*Gimbal lock*). Los cuaterniones son representaciones más abstractas, pero más robustas que los ángulos de Euler [25].

Para esta aplicación, se utilizará el método *AngleAxis(float angle, Vector3 axis)*, de la clase *Quaternion*, que devolverá un cuaternión representando un giro de θ^o alrededor del eje, definido como un vector de tres componentes [26].

En la Figura 26 se representa un giro de θ^o, alrededor del eje y. Será necesario definir dos cuaterniones, uno para el giro en el sentido positivo que marca el eje, y otro para el sentido negativo, como se aprecia a continuación.

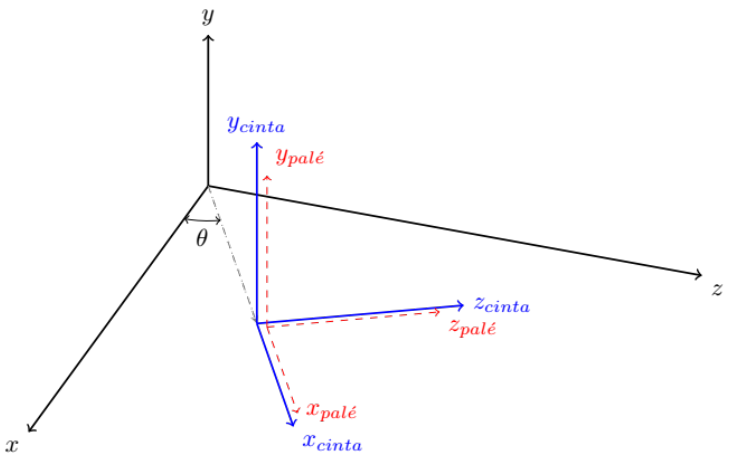


Figura 26: Cinta transportadora curva en estado inicial

En primer lugar, se aplica la rotación del cuaternión de sentido negativo representadas por las siguientes ecuaciones:

$$posición'_{cinta} = posición_{centro} + q_{inversa} \cdot (posición_{cinta} - posición_{centro})$$

$$rotación'_{cinta} = rotación_{cinta} \cdot q_{inversa}$$

En la Figura 27 se aprecia que al aplicar esta transformación se modifican tanto la posición como la orientación de la cinta, representada por el sistema de coordenadas $(x'_{cinta}, y'_{cinta}, z'_{cinta})$. Sin embargo, al modificar la posición y la orientación de la cinta directamente, accediendo a su componente *Transform*, el palé no se ve afectado $(x_{palé}, y_{palé}, z_{palé})$.

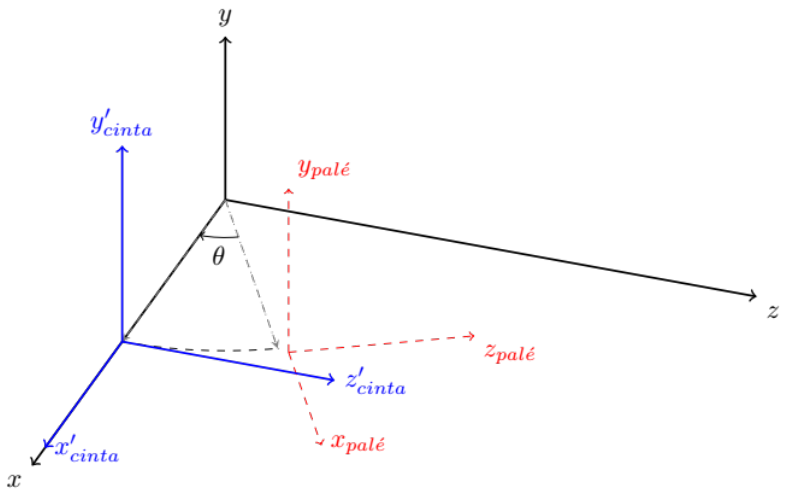


Figura 27: Cinta transportadora curva tras aplicar la rotación negativa de θº

En el siguiente paso, es necesario devolver la cinta transportadora a su posición y orientación original, aplicando la rotación positiva de θº, renderizando de tal modo que parezca que el palé tiene un movimiento suave y continuo. Para ello, se utilizará el método *MovePosition(Vector3 position)* para modificar la posición (como en las cintas rectas), y el método *MoveRotation(Quaternion rotation)* para cambiar la orientación, cómo se describe a continuación:

$$posición''_{cinta} = MovePosition(posición_{centro} + q \cdot (posición'_{cinta} - posición_{centro})$$

$$rotación''_{cinta} = MoveRotation(q \cdot rotación_{cinta})$$

De forma análoga al método *MovePosition()*, la ventaja de llamar a *MoveRotation()* [27] en el bucle *FixedUpdate()* es que el resultado final será renderizado como un movimiento suave y continuo, afectando al palé, como se aprecia en la Figura 28.

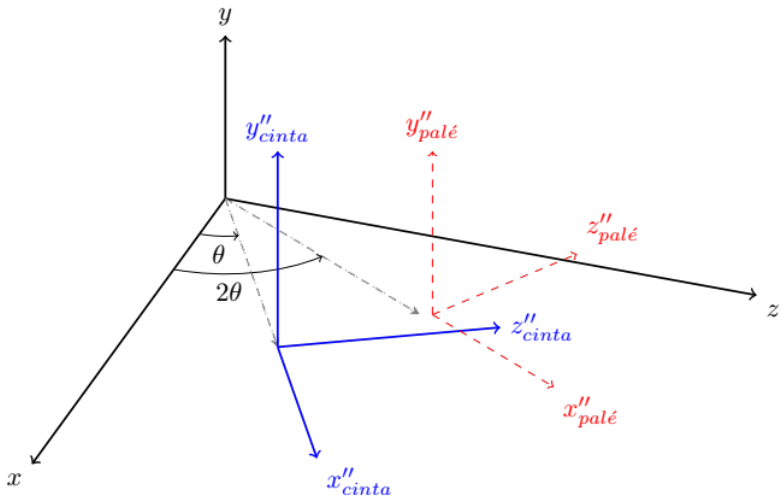


Figura 28: Cinta transportadora curva en estado final

El resultado final es que la cinta transportadora ha vuelto a su posición y orientación original (como si no se hubiese movido), y el palé ha girado un ángulo θ^o alrededor del centro del giro, representado por el eje y, de forma suave y continua. En el Código 2 se muestra el script de la implementación de la cinta transportadora curva.

```
public float speed = 2f; // Rotation speed
public Vector3 axis = new Vector3(0, 1, 0); //y axis
void FixedUpdate()
{
  if (active)
  {
    // +theta rotation
    Quaternion q = Quaternion.AngleAxis(Time.deltaTime*speed, axis);
    // -theta rotation
    Quaternion qinverse = Quaternion.AngleAxis(-Time.deltaTime*speed, axis);
    Rigidbody rb = this.GetComponent<Rigidbody>();
    // Modify position
    rb.position = (center.transform.position + qinverse * (rb.transform.position -
        center.transform.position));
    rb.MovePosition(center.transform.position + q * (rb.position -
        center.transform.position));
    // Modify rotation
    rb.rotation = rb.transform.rotation * qinverse;
    rb.MoveRotation(rb.rotation * q);
  }
}
```

Código 2: *CurvedConveyor.cs* - Implementación de la cinta transportadora curva

5.4. Comportamiento del robot

La posición de la herramienta del robot ABB IRB 120 quedará determinada por los ángulos de sus articulaciones. Por lo tanto, para calcular la posición de una articulación,

habrá que tener en cuenta el ángulo de las anteriores, e ir aplicando transformaciones hasta obtener el resultado deseado. Este problema se conoce como cinemática directa.

5.4.1. Cinemática directa

Para resolver el problema de la cinemática directa, Denavit y Hartenberg idearon un sistema que utiliza álgebra matricial. Es necesario establecer un sistema de coordenadas en cada articulación, solidario con cada eslabón. Además, estos sistemas de coordenadas tienen que cumplir las siguientes tres reglas [28]:

1. "El eje z_{i-1} yace a lo largo del eje de la articulación."
2. "El eje x_i es normal al eje z_{i-1} y se debe cortar con él."
3. "El eje y_i completa el sistema de coordenadas dextrógiro según se requiera."

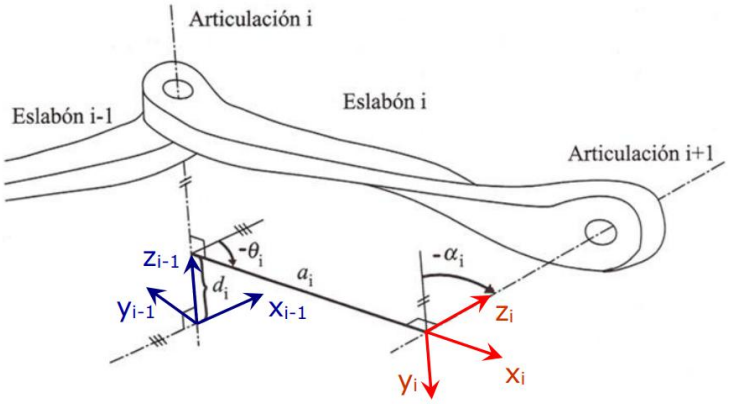


Figura 29: Colocación de ejes en eslabones con el criterio de Denavit y Hartenberg [28]

Una vez colocados los ejes de cada eslabón siguiendo las tres reglas anteriormente descritas, es necesario calcular los siguientes parámetros [28]:

- θ_i: "ángulo de la articulación del eje x_{i-1} al eje x_i respecto del eje z_{i-1}".
- d_i: "distancia desde el origen del sistema de coordenadas (i-1)-ésimo hasta la intersección del eje z_{i-1} con el eje x_i a lo largo del eje z_{i-1}".
- a_i: "distancia desde la intersección del eje z_{i-1} con el eje x_i hasta el origen del sistema i-ésimo a lo largo del eje x_i".
- α_i: "ángulo de separación del eje z_{i-1} al eje z_i respecto del eje x_i".

Una vez definidos estos parámetros, ya es posible calcular la matriz de paso entre articulaciones, desde la articulación $i-1$ hasta i, como se muestra a continuación [28]:

$$ {}^{i-1}A_i = \begin{bmatrix} cos\theta_i & -cos\alpha_i sen\theta_i & sen\alpha_i sen\theta_i & a_i cos\theta_i \\ sen\theta_i & cos\alpha_i cos\theta_i & -sen\alpha_i cos\theta_i & a_i sen\theta_i \\ 0 & sen\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

5.4.2. Implementación

Unity proporciona una herramienta que permite implementar y resolver el problema de la cinemática directa sin tener que programar el método de Denavit Hartenberg, y es mediante el árbol de jerarquía. Es necesario crear *GameObjects* vacíos (llamados $Axis_i$ en la Figura 30), en la posición de cada eje, que actuarán como los sistemas de coordenadas de cada eslabón. A estos objetos les asigna como hijos los eslabones sucesivos. De esta forma, cuando se modifique la rotación de cada eslabón, se hará alrededor de estos sistemas de coordenadas, de forma local, y los eslabones sucesivos se moverán acorde a dicha rotación.

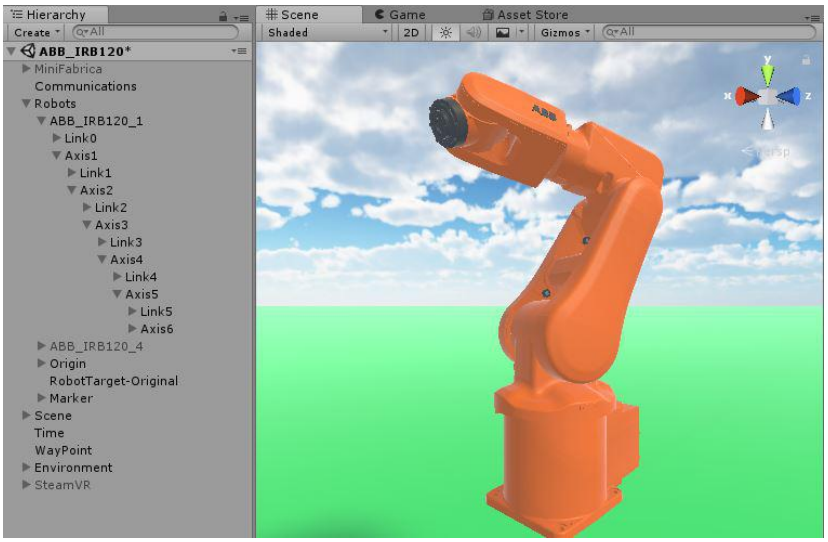


Figura 30: Jerarquía del robot

En el Código 3 se buscan los *GameObjects* creados y se almacena su componente *Transform*, que contiene su posición y rotación. En el bucle *Update()* se modifica la rotación local de cada eje, utilizando el método *Lerp()*, de la clase *Quaternion*. Esta función tiene los siguientes argumentos:

- *Quaternion* a. En este caso la rotación local del eje.
- *Quaternion* b. Representación del ángulo del eje convertida a cuaternión.
- *float* t. Factor de velocidad multiplicado por el tiempo que ha pasado desde el fotograma anterior [29] (*Time.deltaTime*).

La función *Lerp()* interpola de forma lineal entre la orientación representada por el cuaternión "a" y la rotación representada por el cuaternión "b", en el intervalo "t", y posteriormente normaliza el resultado [30].

```
// Use this for initialization
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>();

}

// Update is called once per frame
void Update()
{
    // Linear interpolate to angle
    tAxis1.localRotation = Quaternion.Lerp(tAxis1.localRotation, Quaternion.Euler(0,
        -a1, 0), speed * Time.deltaTime);
    tAxis2.localRotation = Quaternion.Lerp(tAxis2.localRotation, Quaternion.Euler(0,
        0, -a2), speed * Time.deltaTime);
    tAxis3.localRotation = Quaternion.Lerp(tAxis3.localRotation, Quaternion.Euler(0,
        0, -a3), speed * Time.deltaTime);
    tAxis4.localRotation = Quaternion.Lerp(tAxis4.localRotation,
        Quaternion.Euler(-a4, 0, 0), speed * Time.deltaTime);
    tAxis5.localRotation = Quaternion.Lerp(tAxis5.localRotation, Quaternion.Euler(0,
        0, -a5), speed * Time.deltaTime);
    tAxis6.localRotation = Quaternion.Lerp(tAxis6.localRotation,
        Quaternion.Euler(-a6, 0, 0), speed * Time.deltaTime);
}
```

Código 3: *IRB120.cs* - Script de movimiento del robot

5.5. Preparación de la escena para RV

5.5.1. SteamVR

Para hacer que la simulación de Unity sea compatible con realidad virtual, existen dos alternativas disponibles:

- *Oculus SDK*. SDK propietario de Oculus, y no es compatible con sistemas que no sean de esta compañía.
- *SteamVR*. Plugin de Valve basado en OpenVR, compatible con la mayoría de soluciones de realidad virtual.

Para este proyecto se ha decidido implementar usando *SteamVR*, ya que no se ha utilizado un Oculus Rift exclusivamente, sino que en la fase de desarrollo el proyectista disponía de un HTC Vive. Para utilizar *SteamVR* basta con importar el *plugin* desde la *Asset Store* de Unity, donde se encuentra disponible de forma gratuita.

Tras haber importado el *plugin*, solo es necesario arrastrar al editor o al árbol jerárquico el *prefab* u objeto *Player*, localizado en la carpeta *SteamVR/InteractionSystem/Core/Prefabs/Player.prefab*

Una vez conectado el Oculus Rift al PC como se detalla en el apartado 8.1, es importante comprobar que no hay dos SDK's de realidad virtual cargados a la vez, dado que entran en conflicto y hacen que la cámara no siga la posición de la cabeza. En la Figura 31 se muestra cómo debería estar configurado, con la opción *Virtual Reality Supported* activada, y solo con OpenVR cargado. Si además de OpenVR aparece Oculus SDK, este último debe de ser eliminado de la lista.

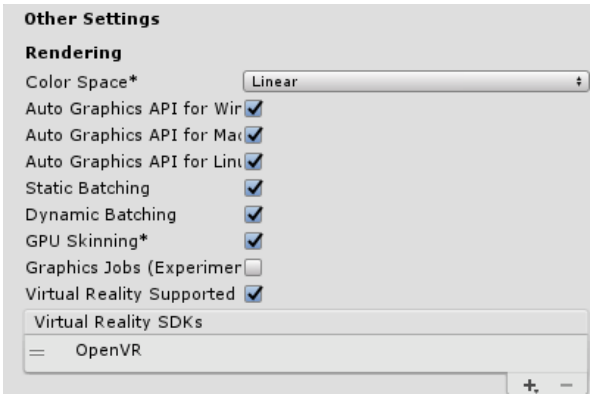


Figura 31: Comprobación de SDK's cargados

5.6. Comunicaciones

Como se ha visto en el esquema de comunicaciones de la Figura 8, la solución desarrollada tendrá que comunicarse con RobotStudio, para comprobar la alcanzabilidad de un punto, guardarlo en una ruta, hacer que ejecute una ruta y recibir los ángulos del robot mientras se está moviendo en la simulación o en el robot real.

Además, también tendrá que comunicarse con PLCSIM, tanto para enviar datos que actúen como entradas del PLC virtual (botones del pupitre de mando, por ejemplo), como para recibir las salidas del PLC (activar los motores que hacen actuar a las cintas).

Haciendo pruebas con las primeras implementaciones de comunicaciones TCP/IP, se encontró el primer problema. La mayoría de implementaciones de comunicaciones son métodos síncronos, es decir, que el *program pointer* del programa no avanzará hasta la siguiente línea de código hasta que reciba algún dato, como se muestra en el siguiente pseudocódigo:

```
// Update is called once per frame
void Update () {
  Debug.Log("Will read from network stream in the next line");
  stream.Read();
```

```
  Debug.Log("This line won't be executed until some data is received");\\
}
```

Código 4: Pséudocódigo de métodos de comunicación síncronos

Este código supone un problema si se incluye en la función *Update()* de Unity, ya que si se observa el flujo de ejecución del motor gráfico (ver Figura 32), este no será capaz de dibujar el siguiente fotograma hasta que haya recibido un dato.

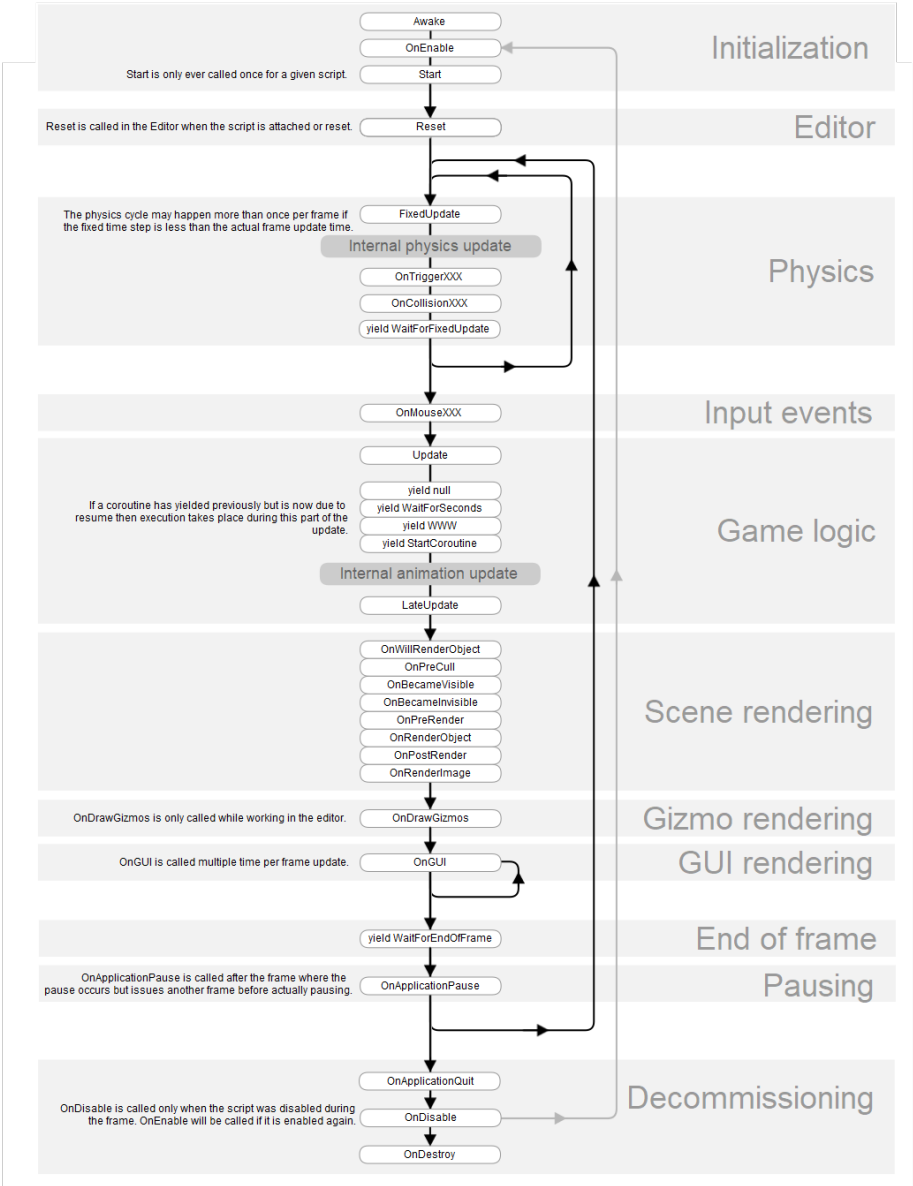


Figura 32: Diagrama de flujo del ciclo de vida de un script en Unity

La solución a este problema es la utilización de hilos o *Threads*. La utilización de hilos permite ejecutar varias tareas a la vez [31], en este caso las tareas de comunicaciones, sin afectar al hilo principal, que se encarga de la lógica y el renderizado de la imagen de la simulación.

Antes de explicar de qué tareas se encarga cada hilo, es necesario aclarar las diferencias entre los protocolos TCP/IP y UDP/IP [32]. El protocolo TCP (*Transmission Control Protocol*), es un protocolo orientado a conexiones, y es útil en aplicaciones donde se necesita fiabilidad (este protocolo incluye confirmaciones de recepción), a costa de aumentar ligeramente el tiempo de transmisión. El protocolo UDP (*Universal Datagram Protocol*) es un protocolo sin conexiones, y se utiliza para aplicaciones dónde el tiempo de transmisión es una prioridad, y no tanto la fiabilidad o la recepción del 100 % de los paquetes enviados.

En cuanto al diseño de las comunicaciones, se han decidido usar tres hilos distintos, que se explicarán en los siguientes apartados.

- Hilo de comunicaciones con RobotStudio - Protocolo TCP/IP.
- Hilo de comunicaciones con RobotStudio - Protocolo UDP/IP.
- Hilo de comunicaciones con interfaz PLCSIM - Protocolo TCP/IP.

5.6.1. Hilo de comunicaciones con RobotStudio TCP/IP

Este hilo se encargará de mandar las coordenadas, orientación de la herramienta y configuración del robot a RobotStudio, y recibirá los ángulos resultantes si el punto es alcanzable. Además, también se utilizará para guardar puntos en *Paths* del robot, y para dar la órden de ejecutar el *Path* guardado.

La justificación para utilizar el protocolo TCP es que estas operaciones no son continuas, sino que se realizan con un tiempo considerable entre ellas. Por ello, se prioriza la fiabilidad en la conexión, al recibir confirmación de recepción, y el tiempo no es relevante.

En el Código 5 se muestra cómo se inicializa el hilo de comunicaciones. En primer lugar, es necesario incluir tres referencias: *System.Net* y *System.Net.Sockets* para las comunicaciones, y *System.Threading* para poder crear hilos separados. Para utilizar un hilo, simplemente basta con declarar un objeto de tipo *Thread*. Para implementar las comunicaciones TCP/IP, se necesitan dos objetos, uno de tipo *TcpClient* y otro de tipo *NetworkStream*. Para inicializar el hilo, basta con crearlo introduciendo como argumento de entrada el nombre de la función que contiene la lógica del hilo.

```
//Dependencies for communications and threading
using System.Net;
using System.Net.Sockets;
using System.Threading;

public class Communications : MonoBehaviour {
```

```
  //Bool to abort communications
  private bool abort = false;

  //Declare thread
  Thread threadRobotStudio = null;
  //Declare TcpClient and NetworkStream
  TcpClient clientRobotStudio;
  NetworkStream streamRobotStudio;

  // Use this for initialization
  void Start(){
    // Create and start RobotStudio communications thread
        threadRobotStudio = new Thread(CommsRobotStudio);
        threadRobotStudio.Start();
  }

  void CommsRobotStudio()
    {
        //Prevent thread locking
        while (!abort)
        {
          //Here goes the code for the communications thread
          //...
          //...
        }
        return;//Prevent thread locking
        Debug.Log("Aborted network thread");
    }
}
```

Código 5: *Communications.cs* - Inicialización de hilo de comunicación

En el mismo código se puede ver que hay dos comentarios que hacen referencia a un suceso denominado *Thread locking*, o bloqueo del hilo. Este problema consiste en que, al crear y ejecutar un hilo, si no se cierra de forma correcta, la siguiente vez que se ejecute la simulación, el programa se quedará colgado, ya que la CPU estará intentando crear un hilo nuevo que ya existe y se sigue ejecutando, aunque la simulación esté parada. Este problema se debe a que, al principio, los hilos se intentaban cerrar utilizando el método *Thread.Abort()*. Sin embargo, este método solo hace que se active una excepción del tipo *ThreadAbortException*, que en algunos casos cierra el hilo, pero no cuando se utilizan funciones síncronas de comunicaciones [33]. Al no ser una forma predecible de cerrar un hilo, se ha introducido una variable de tipo *bool* llamada *abort*, que una vez que se activa, hace que se salga del hilo, como se aprecia en el Código 6. También es importante no olvidar la sentencia *return*, ya que si no el hilo no se cerrará nunca.

```
//Activated if user stops Unity
void OnApplicationQuit()
{
  abort = true;
  //testThread.Abort(); //Not predictable

  //If the client or the stream are not null yet, close them
  if (clientRobotStudio != null || streamRobotStudio != null)
  {
    streamRobotStudio.Close();
    Debug.Log("Closed stream");
    clientRobotStudio.Close();
    Debug.Log("Closed Client");
  }
```

```
}
```

Código 6: *Communications.cs* - Forma correcta de cerrar los hilos al parar la simulación

El código de red (ver Código 7) se activa cuando hay un mensaje a enviar. Esto ocurre cuando el usuario mueve el objetivo del robot, o cambia su rotación. Cuando esto pasa, se activa la variable global *flag*, y en la variable global *message* se introduce el mensaje que contiene los datos que necesita RobotStudio para comprobar la alcanzabilidad de un punto. En primer lugar, al entrar al código se resetea la variable *flag*, y se procede a configurar la conexión TCP. Para crear un objeto *TcpClient* se necesitan dos argumentos [34], una variable de tipo *string* con la dirección IP de la máquina que está ejecutando el código de servidor TCP, además del puerto en el que tiene habilitado el servidor.

Posteriormente, se configuran las propiedades del objeto del tipo *NetworkStream*, en concreto *ReadTimeout* y *Writetimeout*. Estas propiedades establecen el tiempo máximo que se intentará leer o escribir, en este caso 2000 ms o 2 s. Si se sobrepasa, el código avanzará a la siguiente línea.

Por último, es necesario convertir la variable *message* (de tipo *string*) a la variable *dataSend*, de tipo *byte[]*, utilizando el método *System.Text.Encoding.ASCII.GetBytes()* [35].

```
//If communications flag is raised
if (flag)
{
  //Reset flag
  flag = false;
  //Create TcpClient in specific IP and port (must be the one of the RobotStudio
      socket server)
  clientRobotStudio = new TcpClient("192.168.1.135", 5000);
  //Get the NetworkStream from the TcpClient
  streamRobotStudio = clientRobotStudio.GetStream();

  //Set read and write timeouts to 2 seconds
  streamRobotStudio.ReadTimeout = 2000;
  streamRobotStudio.WriteTimeout = 2000;

  //Format message to send to bytes[]
  dataSend = System.Text.Encoding.ASCII.GetBytes(message);

  //Here goes the code for sending data to RobotSudio
  //...
  //...

  //Here goes the code for receiving data from RobotSudio
  //...
  //...
}
```

Código 7: *Communications.cs* - Configuración de la conexión

En el Código 8 se muestra la parte del script que se encarga de enviar datos a RobotStudio. Es importante utilizar la sentencia *try*{}-*catch()*{}, dado que si por alguna razón falla la comunicación y salta una excepción, si no está esta sentencia el

programa dará un error de tipo *Unhandled Exception* [36]. Con esta sentencia podemos cerrar las comunicaciones de forma controlada (los objetos de tipo *NetworkStream* y *TcpClient* con el método *Close()* [37][38]).

Para enviar los datos, se utiliza el método *Write()*, del objeto de tipo *NetworkStream*. Este método tiene 3 entradas [39]:

- *byte[] buffer*: buffer de datos a enviar, en este caso la variable *dataSend*.
- *int32 offset*: posición inicial del buffer desde la que se quiere enviar, en este caso la inicial (0).
- *int32 size*: longitud de los datos a enviar, o la longitud de la varible dataSend.

```
try
{
    streamRobotStudio.Write(dataSend, 0, dataSend.Length);
    Debug.Log("[U5] Sent message: " + System.Text.Encoding.ASCII.GetString(dataSend));
}
catch (System.IO.IOException e)
{
    Debug.Log("[IO] Error sending message to RobotStudio");
  //Close communications
    streamRobotStudio.Close();
    clientRobotStudio.Close();
}
```

Código 8: *Communications.cs* - Envío de datos a RobotStudio

De la misma forma que el código de escritura, el de recepción (ver Código 9) también tiene que ir acompañado de una sentencia *try*{}-*catch()*{}, para controlar las excepciones. En primer lugar, se inicializa la variable de tipo *byte[] dataReceive*, con caracteres "X".

Para la recepción de los datos, se utiliza el método *Read()*, también perteneciente al objeto de tipo *NetworkStream*. De forma similar al método *Write()*, esta función tiene 3 argumentos:

- *byte[] buffer*: buffer de datos a recibir, en este caso la variable *dataReceive*.
- *int32 offset*: posición inicial del buffer desde la que se quiere recibir, en este caso la inicial (0).
- *int32 size*: longitud de los datos a recibir, en este caso limitada a 20.

A continuación se comprueba si los datos recibidos contienen la cadena "error", utilizando el método *Contains()* [40]. En este caso, esto significa que RobotStudio ha concluido que el punto, en esa configuración, no es alcanzable. Para tener información adicional, se pone la variable *reachable* a *false*, y la variable *receivedResponse* a *true*.

Si el punto es alcanzable, basta con utilizar el método *MoveRobot*, de la clase del robot (script *IRB120.cs*), utilizando como argumento los *bytes* recibidos convertidos a una variable de tipo *string*. Por último, se configuran las variables *reachable* y *receivedResponse* a *true*.

```
try
{
    //We have to initialize the receiving buffer
    dataReceive = System.Text.Encoding.ASCII.GetBytes("XXXXXXXXXXXXXXXXXXXXXX");
    //Read from the NetworkStream and save data to dataReceive
    streamRobotStudio.Read(dataReceive, 0, 20);

  //If the receive message contains "error"
    if (System.Text.Encoding.ASCII.GetString(dataReceive).Contains("error"))
    {
        Debug.Log("[RS]: Error - Point not reachable");
        //Set flags
        reachable = false;
        receivedResponse = true;
    }
    else
    {
        Debug.Log("[RS] Received message: " +
            System.Text.Encoding.ASCII.GetString(dataReceive));
    //Move the robot to the received coordinates
        robot.MoveRobot(System.Text.Encoding.ASCII.GetString(dataReceive));
        //Set flags
        reachable = true;
        receivedResponse = true;
    }
    //Close communications
    streamRobotStudio.Close();
    clientRobotStudio.Close();
}
catch(System.IO.IOException e)
{
    Debug.Log("[IO] Error receiving message from RobotStudio> " + e.ToString());
  //Close communications
    streamRobotStudio.Close();
    clientRobotStudio.Close();
}
```

Código 9: *Communications.cs* - Recepción de datos a RobotStudio

5.6.2. Hilo de comunicaciones con RobotStudio UDP/IP

Este hilo se encargará de recibir los ángulos del robot (enviados por RobotStudio) mientras está realizando un *Path* que se haya programado, formado por varios puntos objetivo que se hayan guardado.

La justificación para utilizar el protocolo UDP es que la posición de las articulaciones del robot se muestrearán cada 10 ms, por lo que la rapidez en la conexión es importante, y no tanto si se pierde una medida. Lo que se busca en esta parte es visualizar el movimiento del robot en la simulación de Unity, y no recrearlo con una precisión extrema, ya que no se dispone de información de los ángulos entre medida y medida.

Como se puede apreciar en el Código 10, el hilo de comunicaciones UDP es considerablemente más sencillo que el de comunicaciones TCP, ya que solo tiene que recibir datos de RobotStudio.

De forma análoga al apartado anterior, se declara e inicializa el hilo de comunicaciones de la misma manera. La diferencia es que ahora tendremos que declarar un objeto de tipo *UdpClient*, en lugar de un *TcpClient*. Para las comunicaciones UDP no es necesario utilizar un objeto del tipo *NetworkStream*, como sí lo era en TCP. El objeto *UdpClient* tiene como argumento del constructor el número de puerto al que se quiere asociar este cliente, en este caso el 7000.

```
//Dependencies for communications and threading
using System.Net;
using System.Net.Sockets;
using System.Threading;

public class Communications : MonoBehaviour {
  //Bool to abort communications
  private bool abort = false;

  //Declare thread
  Thread threadRobotStudioUDP = null;
  //Declare UdpClient
  UdpClient clientRobotStudioUDP;

  // Use this for initialization
  void Start(){
    // Create and start RobotStudio communications thread
        threadRobotStudioUDP = new Thread(CommsRobotStudioUDP);
        threadRobotStudioUDP.Start();
  }

  void CommsRobotStudioUDP()
  {
    //Create an UdpClient object listening to port 7000
      clientRobotStudioUDP = new UdpClient(7000);
      //Specify any IP address and listening to port 7000
      IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, 7000);
      while (!abort)
      {
          try
          {
            //If there is something in receiving buffer
              if(!abort & clientRobotStudioUDP.Available > 0)
              {
                  byte[] angles;
                  angles = clientRobotStudioUDP.Receive(ref remoteEP);
                  robot.MoveRobot(robot,
                      System.Text.Encoding.ASCII.GetString(angles));
              }
          }
          catch(System.IO.IOException ex)
          {
              Debug.Log("UDP IOEx");
          }
      }
    return;
    Debug.Log("Aborted UDP network thread");
  }
}
```

Código 10: *Communications.cs* - Inicialización de hilo de comunicación

5.6.3. Hilo de comunicaciones con interfaz PLCSIM TCP/IP

El hilo de comunicaciones con PLCSIM es prácticamente idéntido al desarrollado en el apartado 5.6.1, por lo que no se volverá a explicar y se adjunta en el Anexo 1.7.

5.7. Sistemas de coordenadas

La solución desarrollada en este proyecto comparte información de posición y rotación entre Unity y RobotStudio. Sin embargo, estos dos programas no utilizan el mismo sistema de coordenadas, como se aprecia en la Figura 33.

Figura 33: Comparativa del sistema de coordenadas de Unity y RobotStudio

Si se escogen dos ejes al azar, se realiza el producto vectorial de esos dos ejes en cada sistema de coordenadas, se comprueba que el vector resultante tiene la misma dirección, pero sentido contrario. Esto sucede porque Unity emplea un sistema de coordenadas "zurdo", mientras que RobotStudio utiliza un sistema de coordenadas "diestro". Esta denominación varía según qué mano se utilice para calcular el producto vectorial, como se puede observar en la Figura 34.

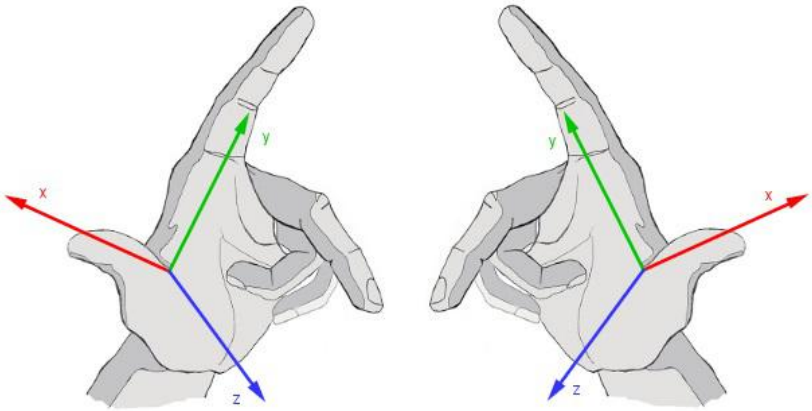


Figura 34: Diferencias entre sistemas de coordenadas zurdos y diestros

Para simplificar los cálculos, se convertirán las coordenadas y las orientaciones en Unity al sistema "diestro" de RobotStudio, dado que la flexibilidad de los scripts de C# hace más fácil esta conversión.

5.7.1. Sistema de coordenadas "Mundo"

El sistema de coordenadas "Mundo" tendrá su origen en el centro del mando de realidad virtual, con lo que se moverá con él. Sin embargo, la rotación del mando será completamente independiente de la de este sistema de coordenadas, que se mantendrá fija. Este sistema de coordenadas no estará emparentado con el mando, por lo que será necesario actualizar su posición manualmente en la función Update(), como se puede apreciar en el Código 11.

```
// Update is called once per frame
void Update () {
   //Update coordinate system position
    this.transform.position = handTransform.position;

   //Calculate coordinate system vectors
    worldX = sphereX.position - this.transform.position;
    worldY = sphereY.position - this.transform.position;
    worldZ = sphereZ.position - this.transform.position;

   //Normalize said vectors
    worldX = Vector3.Normalize(worldX);
    worldY = Vector3.Normalize(worldY);
    worldZ = Vector3.Normalize(worldZ);

    //Set points of line renderers
    lrWorldX.SetPosition(0, this.transform.position);
    lrWorldX.SetPosition(1, this.transform.position + new Vector3(.2f, 0f, 0f));

    lrWorldY.SetPosition(0, this.transform.position);
    lrWorldY.SetPosition(1, this.transform.position + new Vector3(0f, 0f, .2f));

    lrWorldZ.SetPosition(0, this.transform.position);
    lrWorldZ.SetPosition(1, this.transform.position + new Vector3(0f, .2f, 0f));
}
```

Código 11: *WorldCoordinateSystem.cs* - Sistema de coordenadas "Mundo"

Este sistema de coordenadas se compone de tres esferas, cada una marcando el final de cada eje. A estas tres esferas se les ha añadido un componente de tipo LineRenderer. Este componente tiene como entrada una matriz de dos o más puntos, y dibuja una línea entre esos dos puntos. Esta línea no está compuesta por un píxel de ancho (como puede ser la función DrawLine de la clase Debug [41], aunque esta solo funciona en la pantalla editor), sino es un polígono que siempre estará apuntando a la cámara, en este caso el HMD de realidad virtual [42]. El objetivo de este componente es dibujar una línea desde el origen del sistema de coordenadas (centro del mando) hasta las tres esferas. Para diferenciarlos, se les ha aplicado un material a cada una, con los tres colores de los ejes (rojo, verde y azul).

En el script *WorldCoordinateSystem.cs* (Código 11) se almacenarán los tres vectores normalizados del sistema de coordenadas, para facilitar los cálculos de rotación. El resultado final se puede ver en la Figura 35.

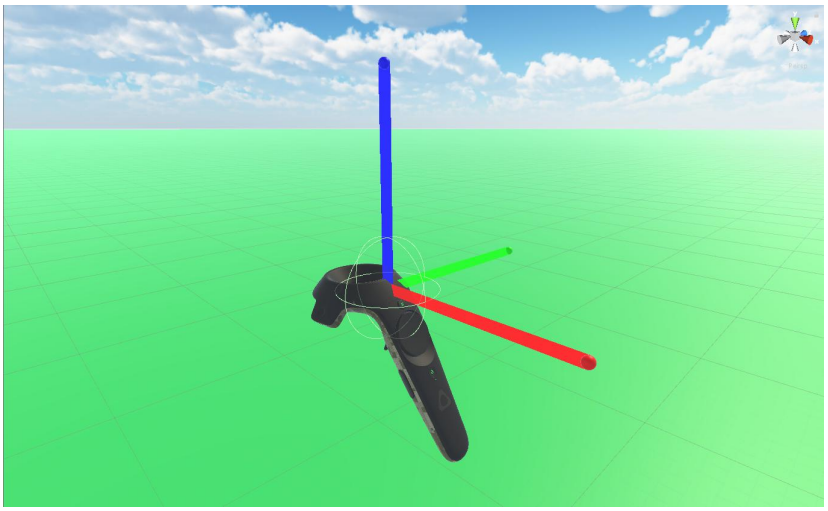


Figura 35: Sistema de coordenadas Mundo

5.7.2. Sistema de coordenadas "Herramienta"

El sistema de coordenadas "Herramienta", de forma idéntica al sistema "Mundo", tendrá su origen en el centro del mando de realidad virtual. La diferencia es que este sistema estará emparentado con el mando (ver Figura 36), con lo cual heredará su posición y orientación de forma automática, y no será necesario actualizarlo en la función Update(), como se hacía en el Código 11.

Figura 36: Parentesco del sistema de coordenadas "Herramienta" con respecto al mando

Es importante que al configurar las posiciones de los componentes de tipo LineRenderer, se utilicen las coordenadas locales de las esferas, localPosition en lugar de position (ver Código 12).

```
// Update is called once per frame
void Update () {
    //Calculate coordinate system vectos
    toolX = sphereX.position - transform.position;
    toolY = sphereY.position - transform.position;
    toolZ = sphereZ.position - transform.position;
    //Normalize said vectors
    toolX = Vector3.Normalize(toolX);
    toolY = Vector3.Normalize(toolY);
    toolZ = Vector3.Normalize(toolZ);

    //Set points of line renderers
    lrToolX.SetPosition(0, sphereX.localPosition);
```

```
        lrToolX.SetPosition(1, sphereX.localPosition + new Vector3(0f, 0f, -.1f));

        lrToolY.SetPosition(0, sphereY.localPosition);
        lrToolY.SetPosition(1, sphereY.localPosition + new Vector3(-.1f, 0f, 0f));

        lrToolZ.SetPosition(0, sphereZ.localPosition);
        lrToolZ.SetPosition(1, sphereZ.localPosition + new Vector3(0f, -.1f, 0f));
    }
```

Código 12: *ToolCoordinateSystem.cs* - Sistema de coordenadas "Herramienta"

También se almacenarán los tres vectores normalizados que componen este sistema de coordenadas, para simplificar los cálculos más adelante. En la Figura 37 se puede ver este sistema de coordenadas aplicado al mando, aunque la mayoría del eje X (rojo), quede oculto por la geometría del mismo.

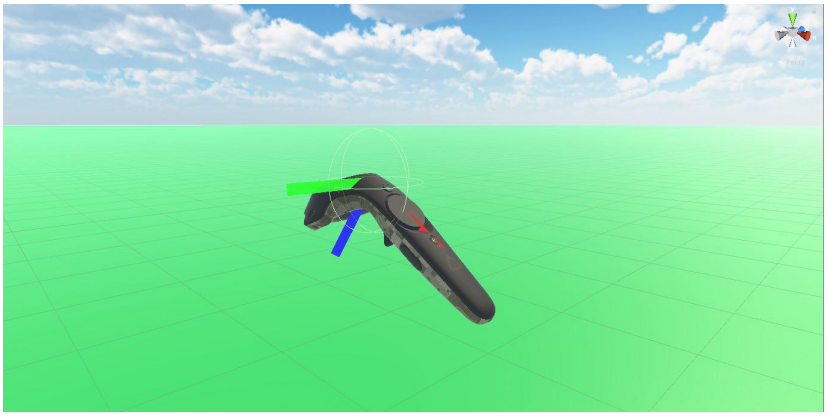


Figura 37: Sistema de coordenadas "Herramienta", solidario al mando

5.8. Preparación de datos para RobotStudio

Para que la simulación de RobotStudio compruebe si un punto es alcanzable o no, necesita una serie de datos. Esta estructura de datos está explicada en profundidad en la sección 6.4, pero se puede resumir en:

- Coordenadas finales de la herramienta.
- Orientación final de la herramienta.
- Configuración de los ejes del robot.

Las coordenadas finales de la herramienta se corresponderán con las coordenadas del mando, aunque con dos pequeñas transformaciones. En primer lugar, es necesario enviar las coordenadas del mando, pero expresadas en el sistema de coordenadas del robot, y no en el sistema de coordenadas del mundo de Unity. Para ello, se coloca un GameObject vacío en la base del robot (ver Figura 38).

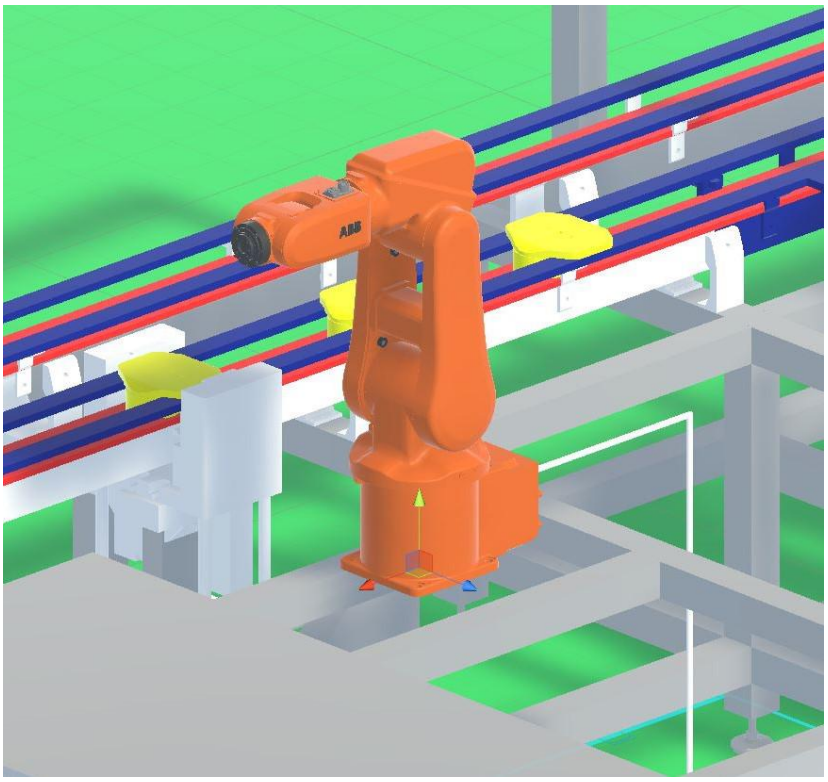


Figura 38: *GameObject* Origen colocado en la base del modelo del IRB 120

La posición de la herramienta se calcula haciendo la resta del vector de las coordenadas del mando y el vector de la posición del GameObject Origen, como se muestra en la siguiente ecuación:

$$\vec{p}_{herramienta} = \vec{p}_{mando} - \vec{p}_{origen}$$

También es necesario adaptar estas coordenadas al sistema de coordenadas de RobotStudio, como se detalla en la sección 5.7. Para ello, basta con intercambiar la coordenada "y" por la coordenada "z":

$$\begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = \begin{bmatrix} p_x \\ p_z \\ p_y \end{bmatrix}$$

La orientación de la herramienta es algo más complicada. En un primer momento, se intentó enviar directamente el cuaternión que representa la rotación del mando, rotado 180º en su eje longitudinal, ya que en aplicaciones de *pick&place* la herramienta suele estar mirando hacia el plano del suelo. Esto supuso varios problemas, al no estar representados en el mismo sistema de coordenadas.

La solución que se consideró óptima fue calcular la matriz de rotación R entre la base del sistema de coordenadas "mundo" y la base del sistema de coordenadas

"herramienta" (ver Figura 39). Los componentes de esta matriz de rotación R se calculan haciendo el producto escalar de los distintos elementos de la base [43], como se muestra a continuación:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} \vec{v}_x^m \cdot \vec{v}_x^h & \vec{v}_x^m \cdot \vec{v}_y^h & \vec{v}_x^m \cdot \vec{v}_z^h \\ \vec{v}_y^m \cdot \vec{v}_x^h & \vec{v}_y^m \cdot \vec{v}_y^h & \vec{v}_y^m \cdot \vec{v}_z^h \\ \vec{v}_z^m \cdot \vec{v}_x^h & \vec{v}_z^m \cdot \vec{v}_y^h & \vec{v}_z^m \cdot \vec{v}_z^h \end{bmatrix}$$

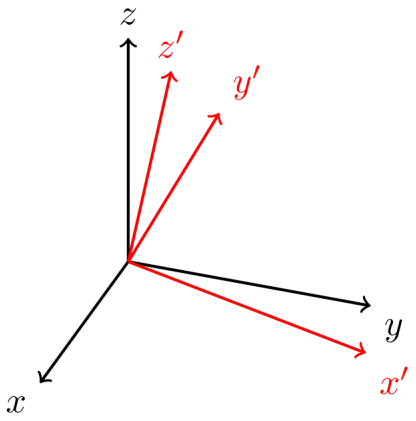


Figura 39: Rotación representada por la matriz R

Una vez calculada la matriz R de rotación, obtener los ángulos de Euler de la herramienta es muy sencillo [44]:

$$\theta_x = atan2(r_{32}, r_{33})$$
$$\theta_y = atan2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2})$$
$$\theta_z = atan2(r_{21}, r_{11})$$

Aunque la base del mundo no varíe, es necesario actualizar la base de la herramienta ya que su rotación cambia con la del mando, y este último está en constante movimiento. Por ello, se coloca en la función Update(), para que se recalcule cada fotograma (ver Código 13).

```
// Update is called once per frame
void Update(){
    //World base
        Vector3 vWorldX = worldCoord.worldX;
        Vector3 vWorldY = worldCoord.worldY;
        Vector3 vWorldZ = worldCoord.worldZ;

        //Tool base
        Vector3 vToolX = toolCoord.toolX;
        Vector3 vToolY = toolCoord.toolY;
        Vector3 VToolZ = toolCoord.toolZ;
```

```
        //Rotation matrix R
        float r11 = Vector3.Dot(vWorldX, vToolX);
        float r12 = Vector3.Dot(vWorldX, vToolY);
        float r13 = Vector3.Dot(vWorldX, VToolZ);
        float r21 = Vector3.Dot(vWorldY, vToolX);
        float r22 = Vector3.Dot(vWorldY, vToolY);
        float r23 = Vector3.Dot(vWorldY, VToolZ);
        float r31 = Vector3.Dot(vWorldZ, vToolX);
        float r32 = Vector3.Dot(vWorldZ, vToolY);
        float r33 = Vector3.Dot(vWorldZ, VToolZ);

        //Angles in degrees
        float thetax = Mathf.Atan2(r32, r33) * 180 / Mathf.PI;
        float thetay = Mathf.Atan2(-r31, Mathf.Sqrt(r32^2 + r33^2)) * 180 / Mathf.PI;
        float thetaz = Mathf.Atan2(r21, r11) * 180 / Mathf.PI;
}
```

Código 13: *VRController.cs* - Cálculo de los ángulos de herramienta

Finalmente, en el Código 14 se detalla la función *SendMessage()*, encargada de crear la cadena o *string* que se enviará a RobotStudio. Esta función tiene las siguientes entradas:

- *Vector3 position*: coordenadas de la herramienta, en mm.
- *Vector3 rotation*: vector $(\theta_x, \theta_y, \theta_z)$, calculado en el Código 13.
- *int* cf_x: configuración de cf_x, descrito con más detalle en el apartado 6.4, por defecto siempre 0.
- *bool savePoint*: indica a RobotStudio que guarde el punto calculado anteriormente y lo añada al *Path*.
- *bool executePath*: indica a RobotStudio que ejecute el *Path* guardado.

De esta función cabe destacar que el cálculo de cf_i está detallado en el apartado 6.4, pero hace referencia al cuadrante en el que se encuentra la articulación i. Además, dependiendo de si se quiere guardar un punto o ejecutar un *path*, se añadirán uno o dos caracteres "X" al inicio de la cadena a enviar. Se concatenan toda las cadenas, con el número de decimales indicado, y se activa el *flag* de envío.

```
void SendMessage(Vector3 position, Vector3 rotation, int cfx, bool savePoint, bool
    executePath)
{
    //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;

    //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;

    //Message control
    if (savePoint & !executePath)
        comms.message = "X";    //Save point into robot path
    else if (executePath & !savePoint)
        comms.message = "XX";   //Execute robot path
    else
        comms.message = "";     //Calculate joint angles

    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") + ";";
    //Debug.Log(comms.message);
    comms.flag = true;
}
```

Código 14: *VRController.cs* - Preparación de datos para RobotStudio

5.9. Objeto objetivo del robot

Desde la simulación en realidad virtual, es necesario encontrar una forma de colocar e interactuar con el objetivo del robot para indicarle la posición y orientación de la herramienta deseada. Para eso, se ha creado un objeto "objetivo", compuesto por dos objetos:

- Gizmo de traslación. Encargado de definir y modificar la posición de la herramienta.

- Gizmo de rotación. Encargado de modificar la orientación de la herramienta.

Traducida del inglés, la palabra gizmo significa literalmente "aparato" [45]. En terminología de programas CAD y desarrolo 3D, un gizmo suele hacer referencia al sistema de ejes que permiten mover y rotar un modelo en tres dimensiones, de ahí que se haya adoptado esta palabra.

5.9.1. Gizmo de traslación

Como se ha definido con anterioridad, el gizmo de traslación es el encargado de indicar al usuario a qué coordenadas va a ir la herramienta del robot. Este objeto se compone de un pequeño cubo en su centro, y tres modelos 3D de una flecha, cada una coloreada con el color del eje al que corresponde (Figura 40). Para evitar las confusiones, se ha decidido adoptar la paleta de colores asignada de forma idéntica a cómo asigna RobotStudio los ejes a la herramienta, descrita en el apartado 5.7.

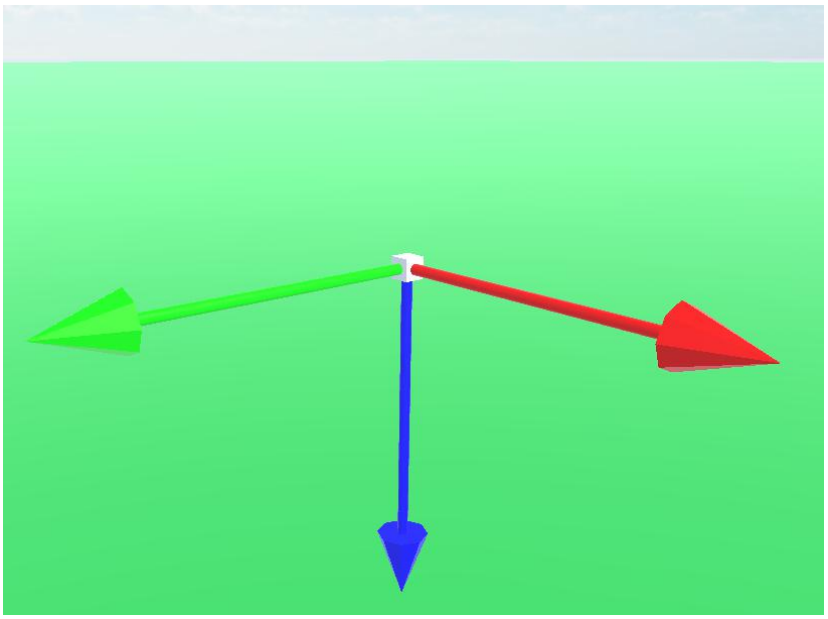


Figura 40: Gizmo de traslación

En primer lugar, es necesario asignar un *trigger* al mando, que permita interactuar con objetos al tocarlos o entrar dentro de su *Collider*. Para ello, se ha colocado una esfera con su *Sphere Collider* con la opción *isTrigger* activada. Esta opción hace que se ignoren las colisiones, pero que salten eventos de tipo *trigger* [46]. Los más utilizados son:

- *OnTriggerEnter*: Activado la primera vez que un *trigger* entra en contacto con un *collider* [47].
- *OnTriggerStay*: Activado cada fotograma mientras el *trigger* esté en contacto con un *collider* [48].

- *OnTriggerExit*: Activado cuando el *trigger* deja de estar en contacto con un *collider* [49].

En el Código 15 se muestra las acciones llevadas a cabo cuando la esfera *trigger* solidaria al mando está en contacto con los ejes. En primer lugar, se comprueba que el objeto que está tocando dicha esfera es un eje. Esto se ha implementado asignando a cada eje una etiqueta o *tag*, y se compara si el *tag* del collider se corresponde con el de un eje.

En caso afirmativo, se comprueba que no se está manipulando el gizmo aún (variable booleana *manipulating*), y que el usuario presiona el gatillo del mando, botón utilizado para interactuar con los ejes. Si se cumplen estas condiciones, se pone la variable *manipulating* a *true*, que actúa como *flag*. Además, se guarda el tag del collider en la variable *savedTag*, y se guardan las posiciones del mando como del gizmo.

```
void OnTriggerStay(Collider coll)
{
    if (coll.tag == "x_axis" || coll.tag == "y_axis" || coll.tag == "z_axis")
    {
        if (!manipulating &
            hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Trigger))
        {
          //Save collider tag
            savedTag = coll.tag;
            //Set manipulating flag to true
            manipulating = true;
            //Save positions
            savedGizmoPosition = translationGizmo.transform.position;
            savedHandPosition = hand.transform.position;
        }
    }
}
```

Código 15: *ControllerTip.cs* - Inicialización de la manipulación del gizmo de traslación

La lógica de movimiento del gizmo se muestra en el Código 16. Si el *flag manipulating* se ha puesto a *true* en el código anterior, se selecciona el eje dependiendo del *tag* del *collider*. Por ejemplo, si el *tag* es *"x_axis"*, se seleccionará el eje de movimiento correspondiente al eje x almacenado en el componente *TransformGizmo*. Este eje se asigna al eje del sistema de coordenadas herramienta (apartado 5.7.2) en el Código 24.

Para obtener el comportamiento esperado, no solo es necesario calcular el movimiento, sino el *offset*, que no es más que el vector que va desde el centro del gizmo hasta la mano. El movimiento se calcula como la posición actual de la esfera que actúa de cursor menos las coordenadas guardadas de la posición del gizmo. Para que el movimiento sea sólo en el eje que hemos seleccionado, se utiliza el método *Project()*, de la clase *Vector3*. Este método permite proyectar el vector de movimiento en el eje seleccionado [50].

Además, se ha añadido una variable de tipo *float* llamada *precision*. Si se aumenta esta variable, se consigue que movimientos grandes del mando se traduzcan en

movimientos pequeños en el gizmo, aumentando la precisión y mejorando la capacidad de ajuste fino.

```
// Update is called once per frame
void Update () {
    if (manipulating &
        hand.controller.GetPress(SteamVR_Controller.ButtonMask.Trigger))
    {
        //Linear movement
        if(savedTag == "x_axis" || savedTag == y_axis" || savedTag == "z_axis")
        {
            Vector3 movement = Vector3.zero;
            Vector3 offset = Vector3.zero;
            Vector3 axis = Vector3.zero;

        //Select axis
            if (savedTag == "x_axis")
                axis = transformGizmo.x_axis;
            else if (savedTag == "y_axis")
                axis = transformGizmo.y_axis;
            else if (savedTag == "z_axis")
                axis = transformGizmo.z_axis;

          //Calculate movement by projecting into momement axis
            movement = Vector3.Project(transform.position - savedGizmoPosition, axis);
            //Calculate offset by projecting into movement axis
            offset = Vector3.Project(savedHandPosition - savedGizmoPosition, axis);
            //Update translation gizmo position
            translationGizmo.transform.position = savedGizmoPosition + (movement -
                offset) / precision;
        }
    }
}
```

Código 16: *ViveController.cs* - Cálculo de los ángulos de herramienta

5.9.2. Gizmo de rotación

El gizmo de rotación permitirá al usuario modificar la orientación de la herramienta una vez se haya aproximado a su objetivo. Este objeto se compone de tres toroides (ver Figura 41), cada uno con el color del eje al que corresponde.

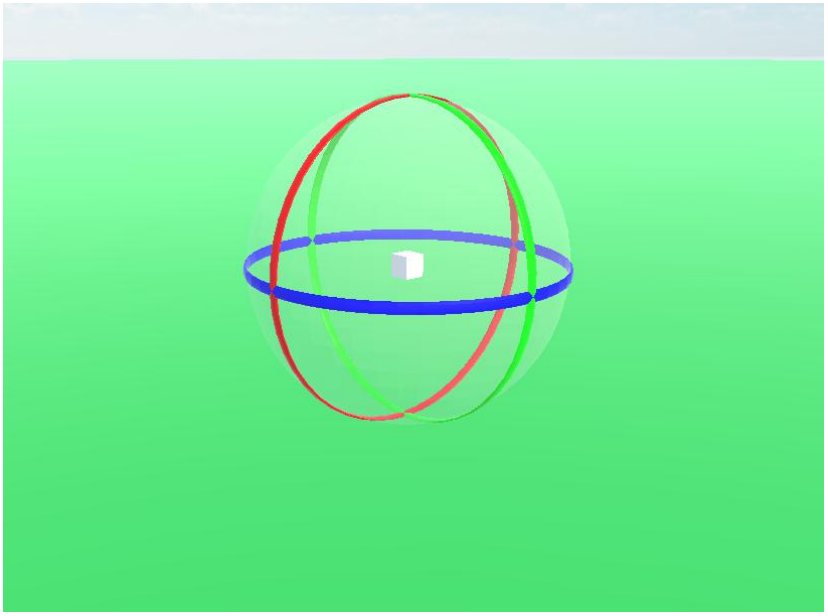


Figura 41: Gizmo de rotación

De forma análoga al gizmo de traslación, es necesario guardar cierta información una vez el usuario aproxima el mando a cada toroide y activa el gatillo del mando (Código 17). La única diferencia es que se necesita cierta información adicional, como la rotación original del gizmo, los ejes de rotación, y el vector que va desde el origen del gizmo hasta el mando.

```
void OnTriggerStay(Collider coll)
{
    if (coll.tag == "x_torus" || coll.tag == "y_torus" || coll.tag == "z_torus")
    {
        if (!manipulating &
            hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Trigger))
        {
            savedTag = coll.tag;
            manipulating = true;
            //Save positions
            savedGizmoPosition = translationGizmo.transform.position;
            savedHandPosition = hand.transform.position;
            //Save rotation
            savedGizmoRotation = rotationGizmo.transform.rotation;
            //Save rotation axis
            if (savedTag == "x_torus")
                savedAxis = Vector3.forward;
            else if (savedTag == "y_torus")
                savedAxis = Vector3.right;
            else if (savedTag == "z_torus")
                savedAxis = Vector3.down;
            //Save gizmo to controller vector
            savedGizmoToController = Vector3.ProjectOnPlane(savedHandPosition -
                savedGizmoPosition, savedAxis);
        }
    }
}
```

Código 17: *ControllerTip.cs* - Inicialización de la manipulación del gizmo de rotacion

En el Código 18 se muestra la lógica de rotación del gizmo. En primer lugar, es necesario recalcular el vector del que va desde el origen del gizmo hasta el mando. Además, este vector se proyecta en el plano definido por el vector del eje de rotación como vector normal, utilizando el método *ProjectOnPlane()*, de la clase *Vector3* [51].

```
//Rotational movement
if (savedTag == "x_torus" || savedTag == "y_torus" || savedTag == "z_torus")
{
    float angle = 0f;
    //Project gizmo to controller gizmo in plane defined by rot axis as normal vector
    Vector3 gizmoToController = Vector3.ProjectOnPlane(this.transform.position -
        savedGizmoPosition, savedAxis);
    //Calculate angle from the original gizmo to controller vector, to the current
        one, along the saved axis
    angle = Vector3.SignedAngle(savedGizmoToController, gizmoToController, savedAxis);

    //Angle on z axis goes backwards
    if (savedTag == "z_torus")
        savedAxis = -savedAxis;

    //Update rotation
    rotationGizmo.transform.rotation = savedGizmoRotation *
        Quaternion.AngleAxis(angle, -savedAxis);
    //Info for further recalculation
    vc.rotationGizmoRecalc = Quaternion.AngleAxis(angle, -savedAxis);
    vc.rotationDone = true;
}
```

Código 18: *ControllerTip.cs* - Manipulación del gizmo de rotación

5.9.3. Resultado final

Combinando los dos objetos descritos (Figura 42), se consigue no solo indicar la posición y orientación en la que se encuentra la herramienta, sino que también se puede modificar de forma sencilla e intuitiva utilizando los mandos.

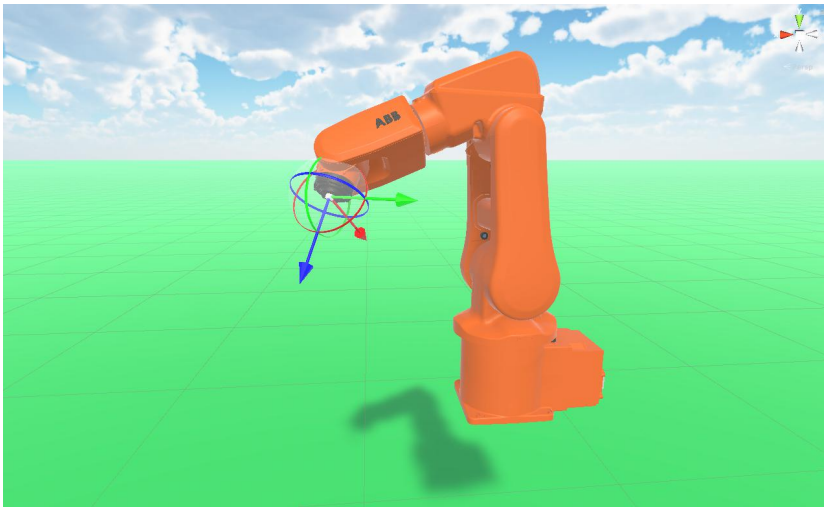


Figura 42: ABB IRB 120 una vez alcanzado el objetivo

Como se comprobará en el siguiente apartado (5.10.1), se manda el mensaje de cálculo cada vez que se mueve el objetivo del robot de forma solidaria al mando. Sin embargo, si el usuario utiliza alguno de los dos gizmos para modificar la posición o rotación de la herramienta, es necesario recomprobar si el robot puede llegar con esta nueva configuración.

Este cálculo se muestra en el Código 19. En primer lugar, se vuelve a calcular la posición relativa del gizmo de traslación con respecto al origen del sistema de coordenadas del robot, en mm. Si se ha cambiado la rotación, es necesario aplicar la misma rotación al sistema de coordenadas de la herramienta guardado, multiplicando los dos cuaterniones. Posteriormente, se envía el mensaje a RobotStudio y se actualiza la rotación guardada. Finalmente, se guardan los nuevos ejes.

```
//If user modified gizmo
if (recalculateGizmo)
{
    //Reset flag
    recalculateGizmo = false;
    //Relative position in mm
    relativePos = (translationGizmo.transform.position - origin.transform.position) *
        1000;

    //If rotation gizmo modified
    if (rotationDone)
    {
        //Reset flag
        rotationDone = false;
        //Apply rotation to saved orientation from the tool
        Quaternion rot = Quaternion.Euler(savedRotationEuler);
        rot = rot * rotationGizmoRecalc;
        //Recalculate target
        SendMessage(relativePos, GizmoToAngles(rotationGizmo.transform.rotation) , 0,
            false, false);
        //Update saved rotation
        savedRotationEuler = GizmoToAngles(rotationGizmo.transform.rotation);
    }
    else
        //Recalculate target
        SendMessage(relativePos, savedRotationEuler, 0, false, false);
    //Save axis for gizmo rotation
    savedXAxis = target.GetComponent<TransformGizmo>().x_axis;
    savedYAxis = target.GetComponent<TransformGizmo>().y_axis;
    savedZAxis = target.GetComponent<TransformGizmo>().z_axis;
}
```

Código 19: *VRController.cs* - Recálculo del objetivo tras modificarlo

La función *GizmoToAngles()*, del Código 19, se detalla en el Código 20. Esta función aplica la misma transformación que aplicando la matriz R, pero sin tener que recalcularla al haber cambiado la rotación. Esta transformación consiste en:

$$\begin{bmatrix}\theta'_x\\ \theta'_y\\ \theta'_z\end{bmatrix}^{RobotStudio} = \begin{bmatrix}-180^o + \theta_z\\ 360^o - \theta_x\\ 90^o - (\theta_y - 180^o)\end{bmatrix}^{Unity} ; (|\theta_x|, |\theta_y|, |\theta_z|)^{Unity} < 180^o$$

```
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;
}
```

Código 20: *VRController.cs* - Función *GizmoToAngles()*

Por último, la función *CorrectAngle()* (Código 21) tiene por argumento un ángulo, y se devuelve el valor limitado de dicho ángulo entre [-180º, 180º].

```
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 21: *VRController.cs* - Función *CorrectAngle()*

5.10. Controles de realidad virtual

5.10.1. Controles de programación del robot en RV

Para implementar los controles de programación del robot, es necesario leer las entradas de los botones de los dos mandos del sistema de realidad virtual, para luego mapearlos a distintas funciones. Afortunadamente, la API de *SteamVR* proporciona una manera sencilla de interceptar estas entradas, lanzando un evento cada vez que se pulsa algún botón, ya sea digital o analógico.

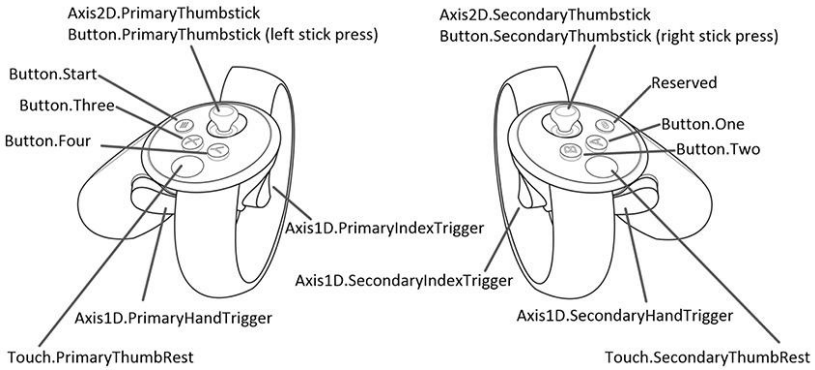


Figura 43: Leyenda de controles del Oculus Rift [52]

Existen varios métodos para capturar eventos del mando, asociados a un objeto de tipo *controller*, accesible a través de la clase *Hand* [53]. Los mas interesantes son los siguientes:

- *GetPressDown*: Detecta el flanco de subida la pulsar un botón.
- *GetPress*: Detecta toda la duración de la pulsación del botón.
- *GetPressUp*: Detecta el flanco de bajada la dejar de pulsar el botón.

Estas funciones tienen como argumento de entrada un objeto de tipo *ButtonMask*, perteneciente a la clase *SteamVR_Controller*. Así, si se quiere detectar pulsaciones de gatillo y touchpad, se haría como se muestra en el Código 22.

```
// Update is called once per frame
void Update () {
  //Detect Trigger button while it's pressed
  if (hand.controller.GetPress(SteamVR_Controller.ButtonMask.Trigger)
    Debug.Log("Trigger pressed");
  //Detect Grip button on first press
  if (hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Grip)
    Debug.Log("Grip pressed once");
  //Detect Touchpad button on release
  if (hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Touchpad)
    Debug.Log("Touchpad released");
}
```

Código 22: Ejemplo de detección de botones

En primer lugar, se quiere mover el objetivo del robot, dándole las coordenadas y orientación del mando, y enviar un mensaje a RobotStudio para que compruebe si esas coordenadas, con esa rotación y en una configuración de ejes determinada (se expandirá en el apartado 6.4) es alcanzable por el robot. Para ello, se programará que mientras el usuario mantenga pulsado el botón del grip del mando

```
// Update is called once per frame
```

```
void Update () {
  //Move robot target gizmo while pressing grip
    if (hand.controller.GetPress(SteamVR_Controller.ButtonMask.Grip))
    {
      //Save hand coordinates
        savedPosition = hand.transform.position;
        //Save calculated tool angles with Rotation matrix
        savedRotationEuler = new Vector3(angleX, angleY, angleZ);

    //Modify translation gizmo position to move target
        translationGizmo.transform.position = hand.transform.position;
        //Modify rotation gizmo rotation to reorient target
        rotationGizmo.transform.rotation = hand.transform.rotation;
    }
}
```

Código 23: *VRController.cs* - Mover objetivo del robot

Una vez movido el objetivo del robot, es necesario calcular si ese nuevo objetivo es alcanzable, por lo que será necesario comunicarse con RobotStudio. En el Código 24 se muestra como al dejar de pulsar el botón del *grip* del mando, se envía un mensaje con la posición relativa al sistema de coordenadas del robot, y la orientación del sistema de coordenadas herramienta, solidario al mando. Por último, se guardan los ejes de la herramienta por si se vuelve a modificar la posición o la orientación desde los gizmos.

```
// Update is called once per frame
void Update () {
  //Check robot target reachability on grip release
    if (hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Grip))
    {
      //Send message to RobotStudio to check reachability
        SendMessage(relativePos, savedRotationEuler, 0, false, false);
        //Assign tool axii to transform gizmo
        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;
    }
}
```

Código 24: *VRController.cs* - Comprobar alcanzabilidad del objetivo del robot

La siguiente opción es guardar un punto calculado en la ruta o *path* del robot. En el Código 25 se detecta una pulsación del botón de *Aplicación* del mando, y se envía un mensaje con la función *SendMessage()*, pasando como argumento el booleano de *savePoint* como verdadero. Además, utilizando el método *Instantiate()*, se instanciará un objeto compuesto por un sistema de coordenadas formado por tres vectores [54], para indicar la posición del punto guardado, como se muestra en la Figura 44.

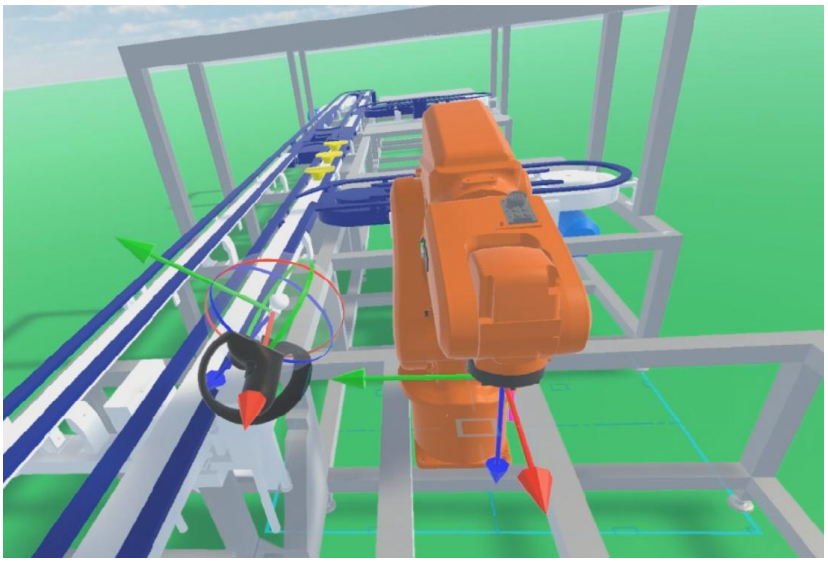


Figura 44: Marcador de punto de ruta guardado

```
// Update is called once per frame
void Update () {
  //Save point to path on application menu button
    if (hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.ApplicationMenu))
    {
      //Send message to RobotStudio to save point to path
        SendMessage(relativePos, savedRotationEuler, 0, true, false);
        //Spawn a marker at the translationGizmo position with rotationGizmo
            orientation
        Instantiate(marker, translationGizmo.transform.position,
            rotationGizmo.transform.rotation);
    }
}
```

Código 25: *VRController.cs* - Guardar punto en ruta del robot

Por último, en el Código 26 está la lógica necesaria para enviar la orden de ejecutar un *path*, que consiste en utilizar la función *SendMessage()*, con el booleano executePath a *true*.

```
// Update is called once per frame
void Update () {
  //Execute path on touchpad press
    if (hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Touchpad))
    {
      //Send message to RobotStudio to execute path
        SendMessage(relativePos, savedsRotationEuler, 0, false, true);
    }
}
```

Código 26: *VRController.cs* - Ejecutar ruta guardada del robot

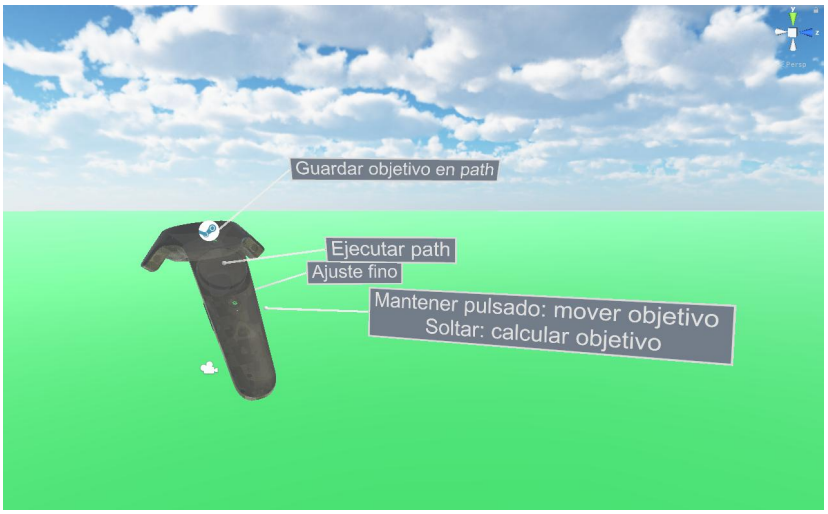


Figura 45: Textos de ayuda para enseñar controles

5.10.2. Controles de movimiento en RV

El movimiento en una simulación de realidad virtual plantea un desafío. El usuario dispone de un área de trabajo limitada (normalmente menor a 3 m^2), calibrada como se describe en el apartado 8.2. Como la minifábrica ICAI tiene una planta con una superficie bastante superior, es necesario implementar un sistema que permita al operario moverse libremente por la planta. Normalmente, en el diseño de aplicaciones de realidad virtual están disponibles dos alternativas:

- Movimiento mediante *joystick* analógico. Similar a los videojuegos tradicionales, consiste en mover un *joystick* en la dirección en la que se quiere mover.
- Movimiento mediante teletransporte. El usuario apunta a donde quiera ir, acciona un botón, y es teleportado instantáneamente a esa localización.

El problema de la primera opción en realidad virtual es que es habitual que produzca mareos. Similar al efecto de una tasa de fotogramas reducida (descrita en el apartado 5.2.2), existe una discordancia entre los estímulos visuales, que indican que hay movimiento, y los estímulos del oído interno, que detectan que se está en una posición estacionaria. Es posible que este efecto vaya desapareciendo para usuarios con varias horas de experiencia en realidad virtual, el fenómeno conocido como *VR legs* [55]. Sin embargo, el objetivo de este proyecto es que lo utilicen personas de todos los niveles de experiencia, desde una persona que nunca ha utilizado un sistema de realidad virtual, hasta un veterano. Por lo tanto, esta opción queda descartada.

La segunda opción tiene la ventaja de que no hay ningún estímulo visual de que el usuario se está moviendo, simplemente aparece en la nueva ubicación instantáneamente. Por lo tanto, se ha decidido emplear este método. Además, el *plugin* de *SteamVR* dispone de scripts que permiten la implementación de su sistema de forma rápida y sencilla.

En primer lugar, es necesario arrastrar el *prefab Teleporting*, ubicado en la carpeta *SteamVR/InteractionSystem/Teleport/Prefabs/Teleporting.prefab*. Una vez hecho esto, es necesario crear un *GameObject* vacío llamado *"Environment"*, y en él ubicar todos los planos que se quieran hacer teleportables. Para terminar, sólo hace falta arrastrar el script *"TeleportArea.cs"* a estos planos.

En la Figura 46 se puede apreciar la superficie que tiene la propiedad de ser teleportable en la escena de Unity, resaltada en amarillo. No se ha ampliado este área para evitar que el usuario se pueda alejar demasiado de la planta por accidente, minimizando la posibilidad de que se pierda.

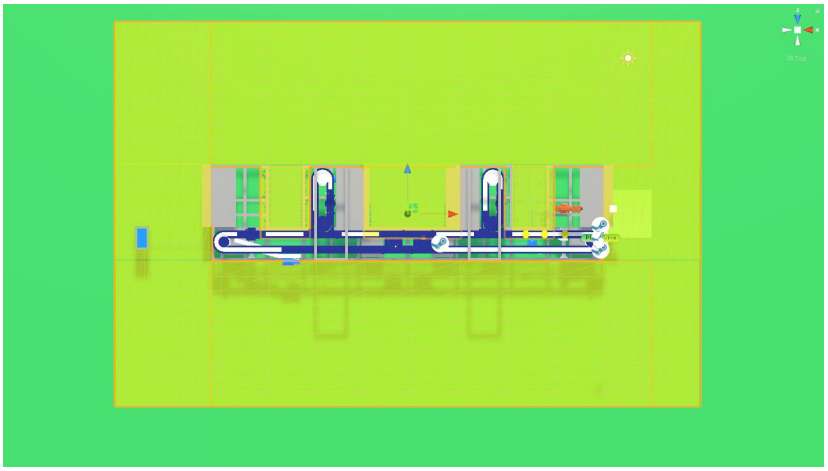


Figura 46: Suelo teleportable de la planta

Si se han realizado los pasos anteriores correctamente, al pulsar el *trackpad* del mando (o el *joystick* en el caso del Oculus Rift), debería de aparecer un arco compuesto por una línea discontinua verde, marcando con una circuferencia el objetivo, como ilustra la Figura 47. Además, si el usuario estuviese apuntando a un área no accesible, este arco cambiaría a color rojo.

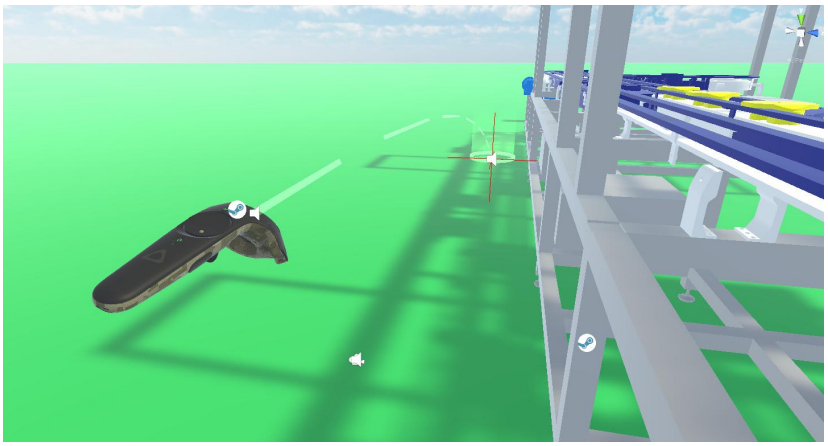


Figura 47: Resultado final del arco de teleportación

5.10.3. Pupitre de control de la planta

Para poder controlar las cintas transportadoras de la planta, se ha simulado en Unity un pupitre de control totalmente funcional. Este pupitre de control consta de los siguientes elementos:

- Seta de emergencia. Utilizada para hacer una parada en casos excepcionales o de peligro.
- Pulsador de rearme. Necesario para salir del estado de emergencia.
- Conmutador de modos. Utilizado para conmutar entre modo automático y manual
- Pulsadores de marcha y paro. Para poner en marcha o parar el sistema.
- LEDs de marcha y paro. LEDs indicadores del estado de la planta
- Pulsador de avance manual de las cintas.

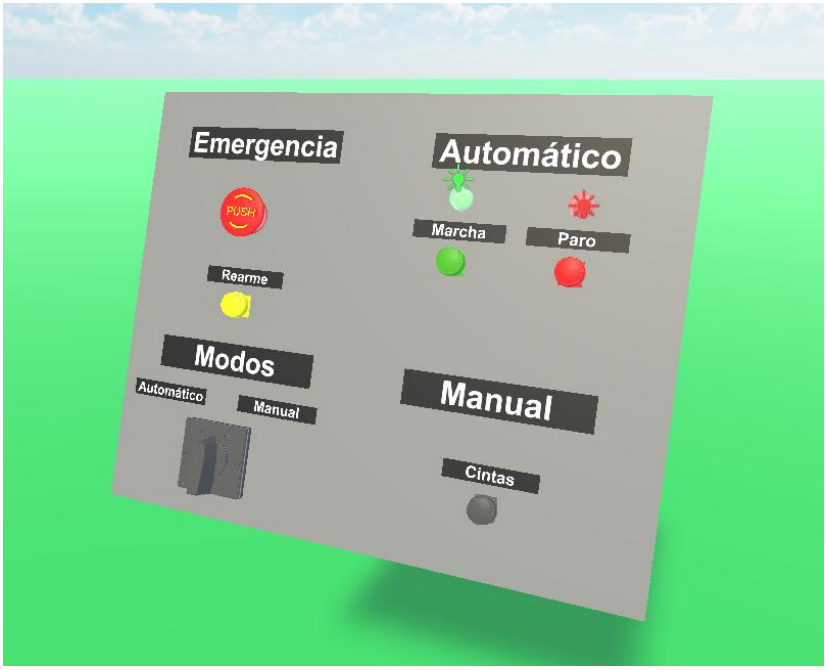


Figura 48: Pupitre de control de las cintas transportadoras de la planta

5.11. Programa de entrenamiento de operarios

El segundo objetivo del proyecto es desarrollar un programa de entrenamiento de operarios interactivo. Poder entrenar a operarios utilizando tecnología de realidad virtual tiene varias ventajas:

- Poder entrenar a los operarios sin que la planta esté construida. De esta forma, las tareas de entrenamiento y puesta en marcha se pueden realizar en paralelo. Esto permite poder empezar a producir desde que se realice la entrega del proyecto de una fábrica, por ejemplo.
- Telepresencia. Con el auge de la industria 4.0 y los avances en ciberseguridad, los sistemas SCADA tienden a pasar de ser una solución local, a una solución conectada, pudiendo ser operada remotamente. La tecnología de realidad virtual podría permitir a un ingeniero visualizar el estado de un proceso productivo con los datos del sistema SCADA, sin estar presencialmente en la fábrica.
- Facilidad de aprendizaje. La tecnología de realidad virtual permite convertir tareas cotidianas en experiencias interesantes, al convertirlas en juegos. Existen estudios que vinculan este enfoque didáctico con una mayor tasa de efectividad de aprendizaje [56]. Además, la tecnología de realidad virtual ofrece posibilidades prácticamente ilimitadas a la hora de mostrar información. En lugar de que el operario tenga que consultar un manual, la información podría estar al alcance del usuario, flotando al lado de cada componente, por ejemplo.

Una de las tareas más comunes de mantenimiento y operación de la minifábrica ICAI es el cambio de herramientas del robot. Este proceso tiene una secuencia de operaciones determinada, y consiste en:

1. Comprobar que los brazos robóticos no solo no están ejecutando ningún programa, sino que también están apagados.
2. Comprobar que la válvula de aire comprimido está cerrada.
3. Cambiar herramienta.
4. Abrir la válvula de aire comprimido.
5. Encender el brazo robótico.

En primer lugar, se necesita modelar la válvula de aire comprimido. La solución adoptada se puede ver en la Figura 49. La válvula diseñada, tiene dos posiciones, abierta y cerrada, ambas señalizadas por un componente *Text Mesh*, que permite mostrar textos en 3D y orientarlos como si fueran cualquier otro *GameObject*.

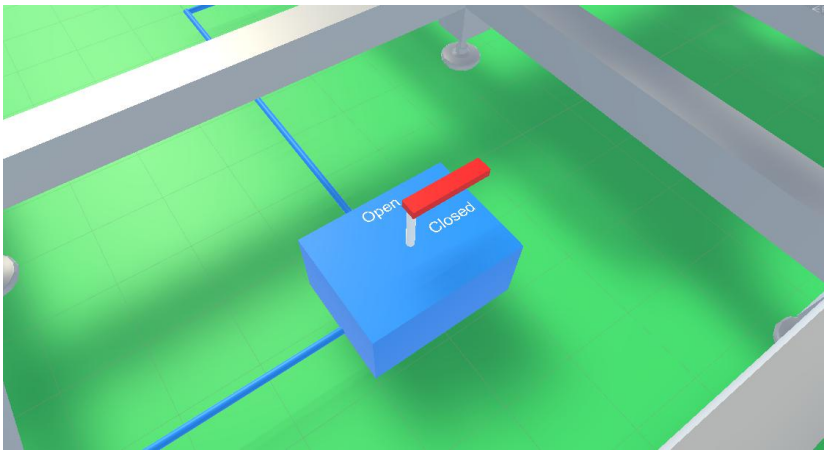


Figura 49: Representación de válvula de aire comprimido

El Código 27 muestra la lógica de este componente, que se modifica cambiando el valor de la variable booleana *closed*. Cuando se activa, se rota alrededor del eje vertical blanco (componente *rod*) 90 grados, en el sentido que corresponda. Esto se lleva a cabo utilizando el método *RotateAround()*, de la clase *Transform* [57]. Las variables booleanas *rotateOpenDone* y *rotateCloseDone* sirven para que el código se ejecute solo una vez al modificar la variable *closed*, y no cada fotograma.

```
// Update is called once per frame
void Update() {
  //If valve is open and has not been closed
    if(!closed & !rotateCloseDone)
    {
        rotateOpenDone = false;
        //Rotate valve 90 deg
        valve.RotateAround(rod.position, rod.up, 90);
        rotateCloseDone = true;
    }
    //If valve is closed and has not been opened
    else if (closed & !rotateOpenDone)
    {
        rotateCloseDone = false;
        //Rotate valve -90 deg
        valve.RotateAround(rod.position, rod.up, -90);
        rotateOpenDone = true;
    }
}
```

Código 27: *AirValve.cs* - Válvula de aire comprimido

Para indicar al usuario la operación que tiene que llevar a cabo en cada momento, se ha colocado un componente *Text Canvas*, de la librería de *SteamVR*. Este componente tiene un texto que se puede cambiar de forma programática, según se vayan cumpliendo objetivos.

Figura 50: Textos de ayuda para orientar al operario

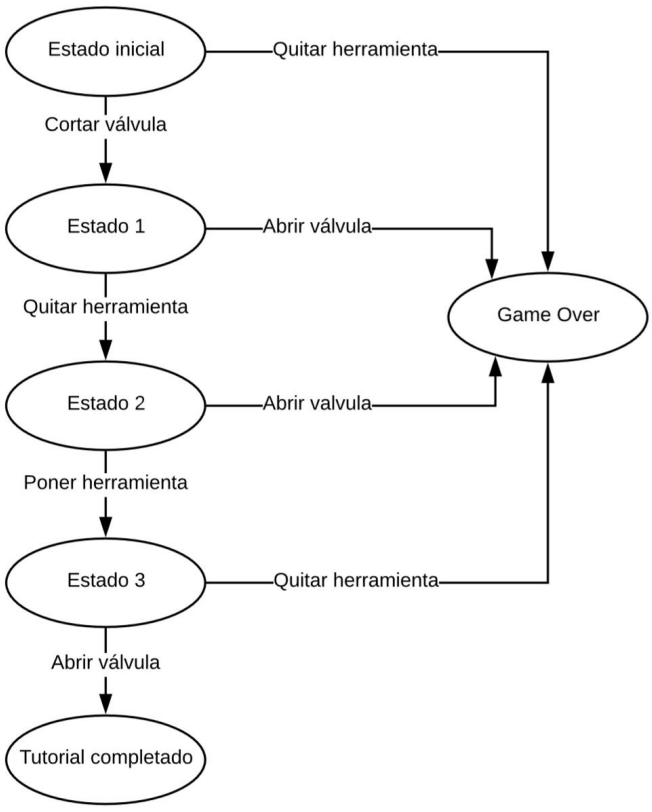


Figura 51: Máquina de estados del tutorial

En el Código 28 se observa la implementación de la máquina de estados de la Figura 51.

```
// Update is called once per frame
void Update () {
    //Edge detection
    //Valve
    valvePrev = valveCurr;
```

```
    valveCurr = valve.closed;
    //Tool
    toolPrev = toolCurr;
    toolCurr = robotTip.hasTool;


    //Initial state
    if (state == 0)
    {
        //Actions of this state
        tutorialText = "Locate the compressed air valve and close";

        //Transitions
        if (valveCurr & !valvePrev)
            state = 1;  //Remove tool state
        else if (toolCurr != toolPrev)
            state = 99; //Game Over
    }
    //Remove tool state
    else if (state == 1)
    {
        //Actions of this state
        tutorialText = "Remove the existing robot tool";

        //Transitions
        if (!toolCurr & toolPrev)
            state = 2; //Install tool state
        else if (valveCurr != valvePrev)
            state = 99; //Game over
    }
    //Install tool state
    else if (state == 2)
    {
        //Actions of this state
        tutorialText = "Install the new robot tool";

        //Transitions
        if (toolCurr & !toolPrev)
            state = 2;
        else if (valveCurr != valvePrev)
            state = 99; //Game over

    }
    //Tutorial completed game
    else if (state == 90)
    {
        //Actions of this state
        tutorialText = "Congratulations! You just became employee of the month!";
    }
    //Game over state
    else if(state == 99)
    {
        //Actions of this state
        tutorialText = "Maybe next time you'll get it right :(";
    }
}
```

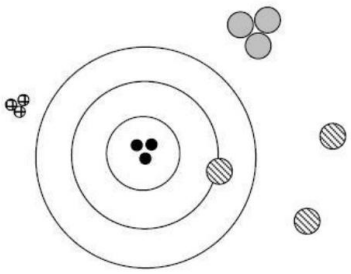
Código 28: *Tutorial.cs* - Máquina de estados del tutorial

6. ABB IRB 120

6.1. Introducción al manipulador

El robot del que se dispone en la minifábrica de ICAI es el ABB IRB 120. El IRB 120 es el manipulador robótico más compacto del catálogo de ABB, lo que lo hace ideal para aplicaciones del tipo de recogida y recolocación (denominadas *pick and place*) de piezas en una línea de producción industrual.

Además, es un robot con una repetibilidad nominal de 10 micras [58], lo que supone que, para objetivo programado, con unas coordenadas determinadas, el robot no se desviará más de una esfera con un radio de 5 micras (Figura 52). Estas desviaciones, aunque pequeñas y para algunas aplicaciones, despreciables, se deben a factores electromecánicos como puede ser el rozamiento de las articulaciones.



	Resolución	Precisión	Repetibilidad
●	Alta	Alta	Alta
⊕	Alta	Baja	Alta
○	Baja	Baja	Alta
◎	Baja	Baja	Baja

Figura 52: Resolución, precisión y repetibilidad [59]

Este modelo de robot industrial tiene 6 grados de libertad, pesa 25 kg en total, y puede soportar una carga en la muñeca de 3 kg en la muñeca con un alcance de 580 mm. La capacidad de carga al estar fijo al suelo, con respecto al centro de coordenadas del robot se muestra en la Tabla 3 [60].

Acción	**Carga en operación**	**Carga máx. (Parada emergencia)**
Fuerza xy	±250 N	±515 N
Fuerza z	-265 ±200 N	-265 ±365 N
Par xy	±195 Nm	±400 Nm
Par z	±85 Nm	±155 Nm

Tabla 3: Capacidad de carga del robot IRB 120

Las articulaciones del IRB 120 tienen distintas restricciones de movimiento. En la tabla 4 [60], se muestra el rango de movimiento articular y la velocidad angular máxima del robot para cada eje.

Eje	Rango de movimiento	Velocidad angular máxima
1	[+165º -165º]	250 º/s
2	[+110º -110º]	250 º/s
3	[+70º -110º]	250 º/s
4	[+160º -160º]	320 º/s
5	[+120 -120º]	320 º/s
6	[+400º -400º]	420 º/s

Tabla 4: Especificaciones mecánicas de las articulaciones del robot IRB 120

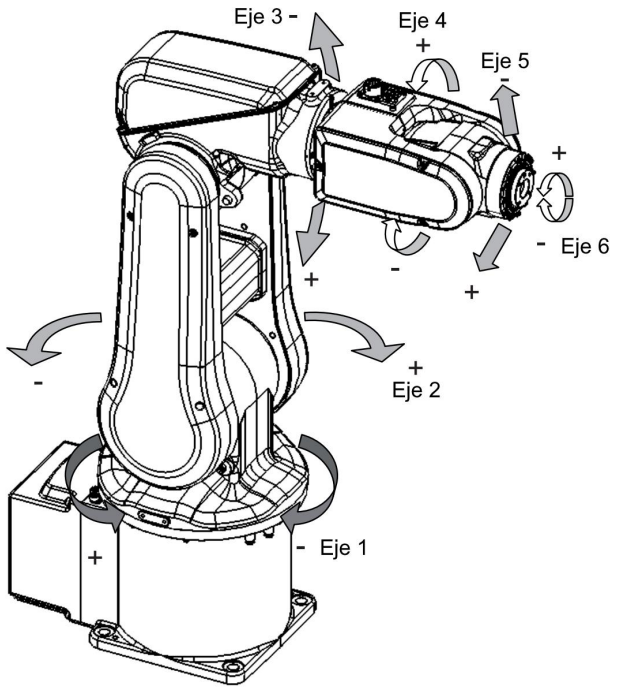


Figura 53: Ejes del brazo robótico ABB IRB 120

6.2. Cinemática inversa

El objetivo de esta parte del proyecto es poder programar el ABB IRB 120 directamente desde la planta virtual, sin utilizar una controladora del tipo *FlexPendant*. El operario estará utilizado unas gafas de realidad virtual, además de los mandos inalámbricos. El principio de funcionamiento se basará en marcar el objetivo al robot con los mandos inalámbricos en el espacio. Éstos mandos inalámbricos representarán la posición y rotación final de la herramienta, y el robot tendrá que adoptar ciertos ángulos para llegar a las coordenadas del mando, con su orientación. Este es el clásico problema de cinemática inversa.

Mientras que la cinemática directa es una técnica que permite el cálculo de la posición final de la herramienta del robot, partiendo de los ángulos de sus articulaciones,

la cinemática inversa permite realizar el cálculo contrario, obtener los ángulos de las articulaciones para que el robot llegue a unas coordenadas con una orientación concreta.

Existen distintas técnicas para resolver el problema de la cinemática inversa. Una forma es utilizando técnicas de métodos numéricos como la optimización mediante el método del gradiente.

En un principio, se consideró utilizar la implementación del método del gradiente, propuesta por Alan Zucconi [61]. En este método, se parte de un punto inicial (punto A en la Figura 54) en la curva del error que se quiere minimizar, y se quiere llegar a un punto que minimice este error (punto B).

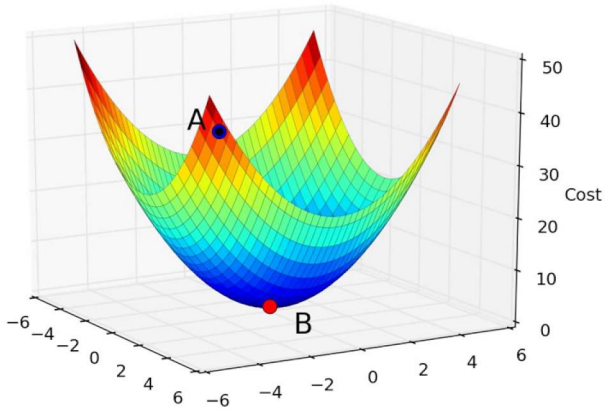


Figura 54: Representación visual del objetivo del método del gradiente

Para calcular la forma más rápida de llegar a este mínimo, basta con calcular la derivada de la curva, y ver qué pendiente es más pronunciada, que será la dirección que minimice el error de forma más rápida posible. Matemáticamente, esto se traduce en el cálculo de las derivadas parciales de la superficie.

$$f'_{\alpha_0}(\alpha_0, \alpha_1, \alpha_2) = \lim_{\Delta x \to 0} \frac{f(\alpha_0 + \Delta_x, \alpha_1, \alpha_2) - f(\alpha_0, \alpha_1, \alpha_2)}{\Delta x}$$

$$f'_{\alpha_1}(\alpha_0, \alpha_1, \alpha_2) = \lim_{\Delta y \to 0} \frac{f(\alpha_0, \alpha_1 + \Delta_y, \alpha_2) - f(\alpha_0, \alpha_1, \alpha_2)}{\Delta y}$$

$$f'_{\alpha_2}(\alpha_0, \alpha_1, \alpha_2) = \lim_{\Delta z \to 0} \frac{f(\alpha_0, \alpha_1, \alpha_2 + \Delta z) - f(\alpha_0, \alpha_1, \alpha_2)}{\Delta z}$$

Si se utilizan incrementos suficientemente pequeños de $\Delta x, \Delta y$ y Δz, es posible aproximar cada derivada parcial con su gradiente correspondiente.

$$\bigtriangledown f_{\alpha_0}(\alpha_0, \alpha_1, \alpha_2) = \lim_{\Delta x \to 0} \frac{f(\alpha_0 + \Delta_x, \alpha_1, \alpha_2) - f(\alpha_0, \alpha_1, \alpha_2)}{\Delta x}$$

$$\bigtriangledown f_{\alpha_1}(\alpha_0, \alpha_1, \alpha_2) = \lim_{\Delta y \to 0} \frac{f(\alpha_0, \alpha_1 + \Delta_y, \alpha_2) - f(\alpha_0, \alpha_1, \alpha_2)}{\Delta y}$$

$$\bigtriangledown f_{\alpha_2}(\alpha_0, \alpha_1, \alpha_2) = \lim_{\Delta z \to 0} \frac{f(\alpha_0, \alpha_1, \alpha_2 + \Delta z) - f(\alpha_0, \alpha_1, \alpha_2)}{\Delta z}$$

El resultado se almacena en un vector que está compuesto de los tres gradientes calculados.

$$\bigtriangledown f(\alpha_0, \alpha_1, \alpha_2) = [\bigtriangledown f_{\alpha_0}(\alpha_0, \alpha_1, \alpha_2), \bigtriangledown f_{\alpha_1}(\alpha_0, \alpha_1, \alpha_2), \bigtriangledown f_{\alpha_2}(\alpha_0, \alpha_1, \alpha_2)]$$

Tras realizar pruebas con esta implementación del método del gradiente, se comprobó que los resultados tienden a converger hacia una solución, pero sufre pequeñas oscilaciones según se acerca a las coordenadas de destino. Además, el robot no siempre alcanza el objetivo con sus articulaciones en ángulos dentro del rango de funcionamiento, y algunas soluciones resultan en configuraciones articulares inusuales. Otro inconveniente de utilizar esta implementación es que la forma de calcular el movimiento del robot puede que resulte en puntos singulares, y es muy probable que el camino resultante no se parezca demasiado al que realizaría el robot programado de la forma tradicional.

Con estos resultados, se decidió utilizar *RobotStudio* para el cálculo de la cinemática inversa. De esta forma, estarían integrados tanto el cálculo de los ángulos de las articulaciones como la programación del IRB 120.

6.3. Introducción a RobotStudio

RobotStudio es un software de programación de robots industriales desarrollado por ABB. La ventaja es que este software emula a las controladoras de los robots físicos, con lo cual no hace falta programar dos versiones de cada proceso (una para la simulación, y otra para el programa de producción). Este punto es muy importante, ya que se permitiría ejecutar el código desarrollado en este proyecto en un robot real. Con respecto al código, tanto los robots físicos como las simulaciones de RobotStudio se programan en el lenguaje RAPID, un lenguaje de alto nivel creado en 1994 [62].

La interfaz de RobotStudio es muy sencilla. La dos ventanas principales en las que se van a trabajar son:

- Ventana *RAPID* (Figura 55). En esta ventana se escribirá el código RAPID que se desea ejecutar en la simulación.
- Ventana *Simulation* (Figura 56). En esta ventana, utilizando el botón con el icono de *play*, se inicia la simulación.

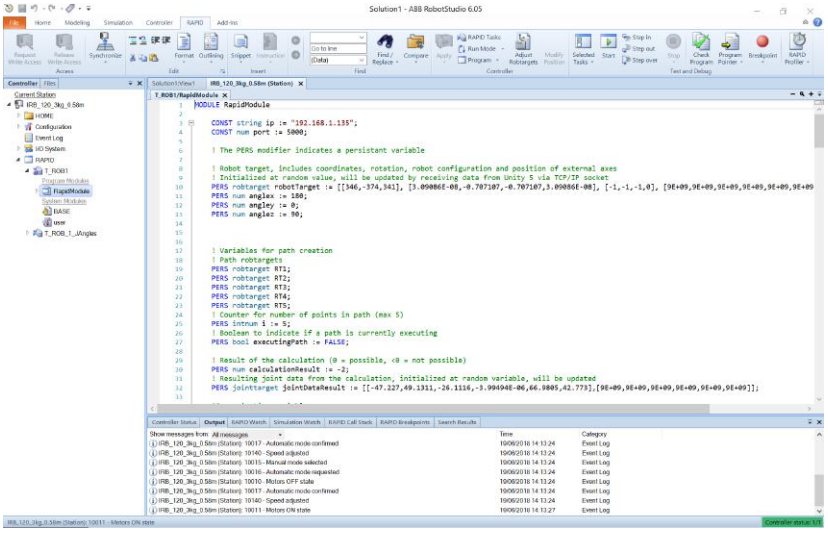


Figura 55: Ventana *RAPID* de RobotStudio

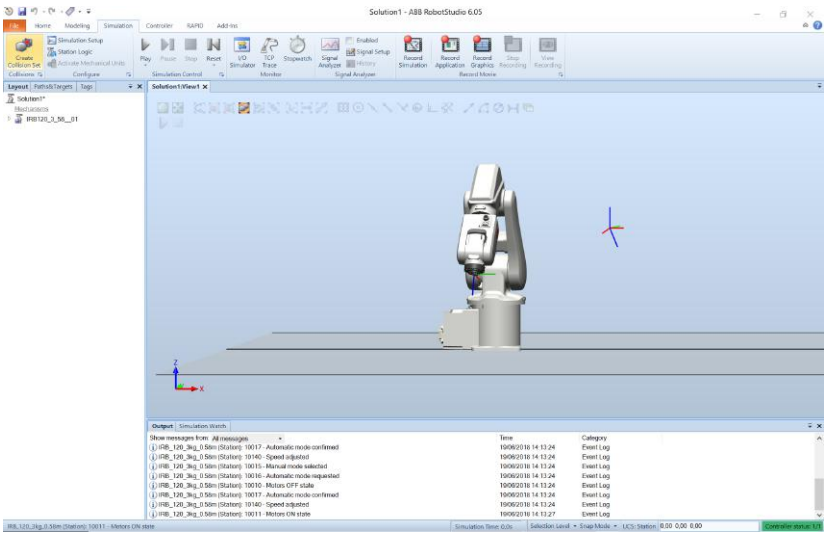


Figura 56: Ventana *Simulation* de RobotStudio

El flujo de trabajo de RobotStudio es el siguiente:

- Programar el robot en la ventana RAPID, en varias tareas si es necesario (como se desarrollará en el apartado 6.5).
- Ejecutar la simulación con el botón de *play* en la ventana Simulación.
- En caso de que salte un error de comunicaciones, por ejemplo, es necesario rearrancar la simulación.

6.4. Alcanzabilidad de un punto

Antes de calcular los ángulos de las articulaciones del robot para cierto punto, es necesario determinar si ese objetivo, con sus coordenadas y orientación, es alcanzable por el IRB 120. En el lenguaje RAPID, los puntos objetivo del robot se almacenan en variables de tipo *robtarget*, que sirven para definir la posición del robot y sus ejes externos [63]. Como se puede apreciar en la Tabla 5, las variables de tipo *robtarget* se componen de:

	Definición	**Tipo dato**	**Descripción**
trans	Traslación	pos	Coordenadas del objetivo [mm]
rot	Rotación	orient	Cuaternión de orientación
robconf	Configuración del robot	confdata	Configuración ejes
extax	Ejes externos	extjoint	Posición ejes externos

Tabla 5: Estructura de variables de tipo *robtarget*

Para la traslación, basta con enviar las coordenadas del mando inalámbrico del Oculus Rift de Unity a RobotStudio. Estas coordenadas no pueden estar expresadas en el sistema de coordenadas de Unity 5, ya que el modelo 3D del robot no se encuentra en dicho origen. Para solventar este problema, basta con crear un GameObject vacío en Unity y colocarlo en la base del robot, para que actúe como origen, como ya se ha explicado en la sección 5.8.

Sin embargo, no basta con hacer esta transformación. Esto se debe a que RobotStudio y Unity utilizan sistemas de coordenadas diferentes. Concretando más, el sistema de coordenadas de RobotStudio es un sistema "diestro", mientras que le sistema de coordenadas de Unity es un sistema de coordenadas "zurdo". Las diferencias entre estos sistemas de coordenadas se explican en detalle en la sección 5.7.

Debido a esta diferencia de sistemas de coordenadas, enviar los datos de la orientación de la herramienta no es tan trivial como la posición. Las rotaciones, tanto en Unity como en RobotStudio, se almacenan como cuaterniones. Por ello, se decidió enviar los ángulos de Euler de Unity a RobotStudio, traducidos como se detalla en la sección 5.8, y crear el cuaternión desde RobotStudio a partir de esos ángulos, como se explicará más adelante.

En cuanto a la configuración del robot, las variables de tipo *confdata* para el IRB 120 tienen 3 componentes [64]:

- cf_1 Cuadrante del eje 1.
- cf_4 Cuadrante del eje 4.
- cf_6 Cuadrante del eje 6.

Estos cuadrantes se numeran de una forma particular. Si el giro es horario, el primer cuadrante será el -1, el segundo cuadrante el -2, etc. Por el contrario, si el giro

es antihorario, el primer cuadrante será el 0, el segundo el 1, etc. Esto se muestra con claridad en la Figura 57.

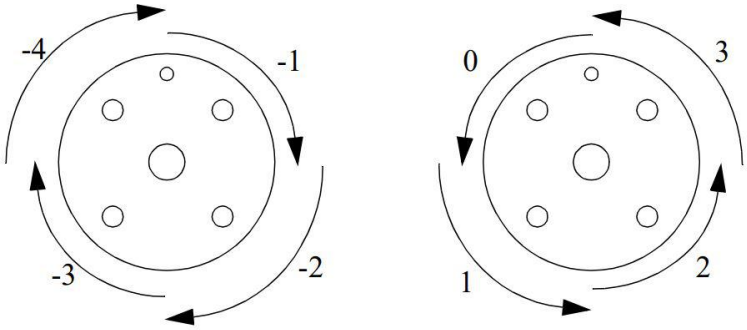


Figura 57: Valor de cf_i para cada cuadrante

Como el IRB 120 no tiene ejes adicionales, todos los valores de los ejes externos toman valores de 9^9, con lo que RobotStudio los ignorará [65].

6.5. Configuración de la simulación de RobotStudio

Para que la simulación se ejecute correctamente, es necesario hacer unos ajustes previos en RobotStudio. El primer parámetro que hay que modificar es el modo de ejecución de los programas. Por defecto, están configurados para que solo se ejecuten un ciclo, pero tal y como se han diseñado los programas, es necesario cambiar el modo de ejecución a modo contínuo. Este parámetro de configuración se cambia en la pestaña $Simulation \rightarrow Configure \rightarrow Simulation\ Setup \rightarrow Run\ mode \rightarrow Continuous$, como se muestra en la Figura 58.

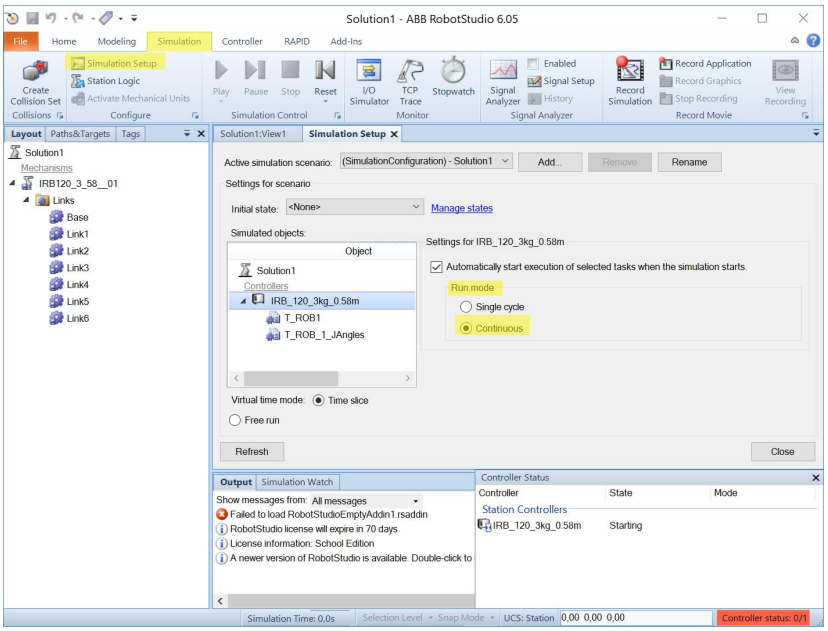


Figura 58: Configuración del *Run Mode* en *Continuous*

Además, para poder dotar al programa de la capacidad de comunicarse utilizando *sockets* TCP y UDP, es necesario habilitar la opción *PC Interface*. Esta opción se encuentra en la pestaña *Controller* → *Virtual Controller* → *Change Options* → *Communication* → *PC Interface* (ver Figura 59).

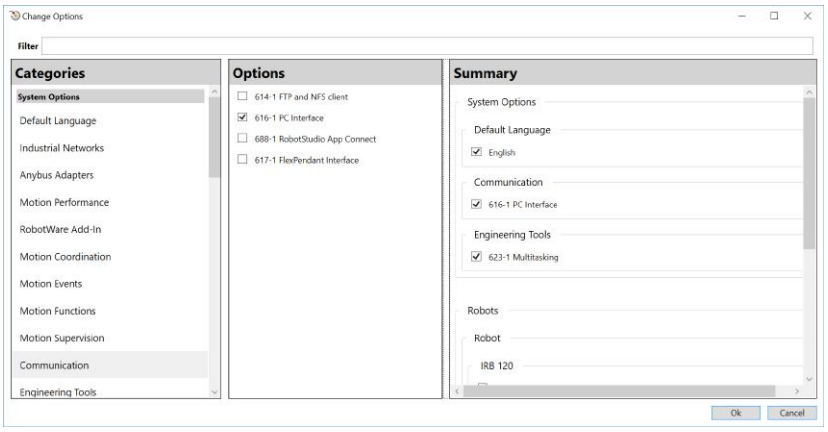


Figura 59: Configuración de la opción *PC Interface* para habilitar las comunicaciones *socket*

Como se explicará en detalle más adelante, no basta utilizar solo una tarea para el robot, sino que se necesitan como mínimo dos. Esto se debe a que durante el movimiento del robot en la simulación, habrá que enviar los ángulos del robot a Unity, y esta será la función de la tarea secundaria. RobotStudio no soporta por defecto la multitarea, sino que hay que habilitarla en las opciones. Esta configuración se encuentra en el menú *Controller* → *Virtual Controller* → *Change Options* → *Engineering tools* → *Multitasking* (Figura 60).

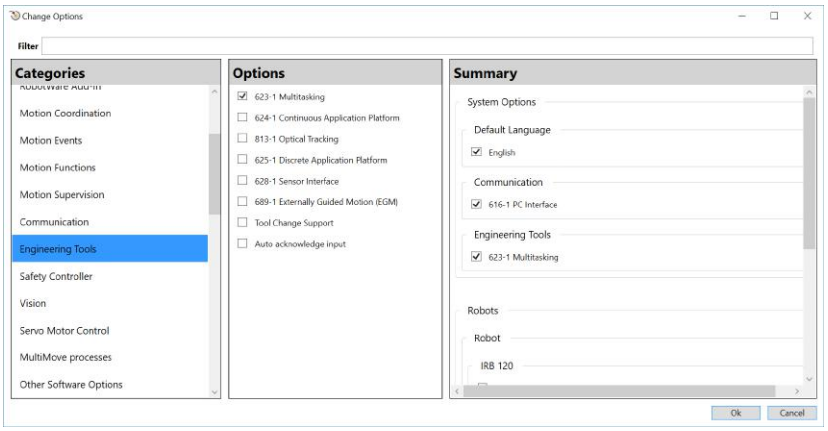


Figura 60: Configuración de la opción Multitasking para poder ejecutar dos tareas de forma simultánea

Tras activar la opción *Multitasking*, es necesario reiniciar la controladora virtual. Para ello, es necesario entrar en la pestaña *Controller tools* → *Restart* → *Shutdown* y posteriormente realizar un *Warmstart*, en el menú *Controller tools* → *Restart* → *Warmstart*.

6.6. Tarea principal

La tarea principal del programa desarrollado en RAPID es la encargada de recibir las coordenadas de la herramienta, su orientación, y la configuración de los ejes descritos en la sección 6.4. Además, como veremos a continuación, también podrá guardar puntos en un *Path*, además de simular ese *path*.

En primer lugar, se declara una variable de tipo *errnum* (*errorNumber*) para almacenar el código de error resultante, en caso de que RobotStudio detecte alguna excepción.

Además, se crea un socket denominado *server* con la instrucción *SocketCreate* [66], y se vincula a una dirección IP y un puerto concretos con la sentencia *SocketBind* [67]. Para que el servidor pueda ser detectar los mensajes de Unity, es necesario inicializarlo con la instrucción *SocketListen* [68], para que pueda recibir conexiones entrantes, y con la sentencia *SocketAccept* se aceptan las conexiones [69], vinculándolas al objeto *client*.

Como los datos se envían desde Unity como *byte[]*, será necesario indicárselo a RobotStudio al ejecutar la instrucción *SocketReceive* [70], utilizando el modificador $\backslash RawData := data$, que almacena el mensaje en la variable *data*. Una vez almacenada, para facilitar su procesamiento, se convierte esta variable *data* a una variable de tipo *string* (*message*) con la instrucción *UnpackRawBytes* [71], e indicando el número de caracteres con el modificador $\backslash ASCII := 70$

```
! Declare variable to store error number
VAR errnum errorNumber;
```

```
!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:=70;
```

Código 29: *MainModule.mod* - Inicialización comunicación *socket*

Una vez se tiene el mensaje en formato *string*, ahora es necesario procesarlo para que el programa sepa como debe proceder. En función de los primeros caracteres del mensaje, se pueden dar tres situaciones distintas:

1. Que el mensaje no tenga ningún carácter "X" al principio de la cadena. Esto significa que Unity está pidiendo que simplemente se comprueben alcanzabilidad del objetivo enviado en el mensaje.
2. Que el mensaje tenga solo un carácter "X" al principio de la cadena. Unity hace una petición para guardar el último punto calculado correcto en el *path*.
3. Que el mensaje tenga dos carácteres "X" al principio de la cadena. Esta es la orden para ejecutar el *path* guardado.

Para hacer esta distinción, se utiliza la sentencia *StrPart* (ver Código 30), para comprobar desde el carácter con el índice 1, hasta el carácter con ese mismo índice, con lo que efectivamente estamos comprobando el primer carácter de la cadena. Si es distinto de "X", el código estará en el primer caso.

Si es así, se llama a la función *StringToTarget* (ver Código 31), que se ha diseñado para convertir el mensaje recibido en una variable de tipo *robtarget*. Esta función localiza el índice en el que empieza cada dato, separado por el carácter ";". Luego se seleccionan los caracteres del dato con la función *StrPart*, se convierten a un valor numérico con *StrToVal* [72] y se completan los distintos componentes de la variable *robtarget*. Como única particularidad, el cuaternión de rotación se construye con los ángulos de Euler, utilizando la función *OrientZYX* [73].

```
!
    !  "message" - Calculate joint angles for robtarget
    ! "Xmessage" - Add target to path
    !"XXmessage" - Execute path

    ! Calculate joint angles
    IF StrPart(message,1,1) <> "X" THEN
        !Convert message string to robtaget object
        robotTarget := StringToTarget(message);
        !Reset calculation result variable
```

```
        calculationResult := 0;

        ! Calculate the joint angles given the robot target, and store errors in
            errorNumber
        jointDataResult := CalcJointT(robotTarget, tool0, \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
```

Código 30: *MainModule.mod* - Cálculo de alcanzabilidad

Una vez convertido el mensaje a una variable de tipo *robtarget*, se calculan los ángulos de las articulaciones para ese objetivo, y se almacenan en la variable *jointDataResult*, de tipo *jointtarget*, que almacena los ángulos de Euler [74]. Si el resultado del cálculo es satisfactorio (indicado por la variable *calculationResult*, escrita en la parte de *Error Handling*), se envían los ángulos utilizando la instrucción *SocketSend* [75], convertidos a una string por la función *JointTargetToString*. Por último, se cierra la conexión socket con la instrucción *SocketClose* [76], aplicadas a los objetos *client* y *server*.

```
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
```

Código 31: *MainModule.mod* - Función StringToTarget()

La función *JointTargetToString* (ver Código 32) simplemente concatena a una cadena existente el dato que se le pase.

```
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
```

Código 32: *MainModule.mod* - Función JointTargetToString()

El Código 33 muestra la gestión de errores y excepciones de RAPID. Si el error es que el objetivo fuerza los límites de las articulaciones, se escribe un "-1" en *calculationResult*. Si el error es que el objetivo está fuera del alcance del robot, se escribe un "-2" en *calculationResult*. En ambos casos, se envía la cadena "error" y se cierra la comunicación *socket*, además de saltar los warnings que puedan saltar para no detener la simulación en RobotStudio, utilizando las sentencias *SkipWarn* y *TRYNEXT*. Si se produce algún error con las comunicaciones por *socket*, se escribe también un valor de error en *calculationResult*, y se tratan de cerrar las

comunicaciones de forma controlada.

```
    ! 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
```

Código 33: *MainModule.mod* - Gestión de errores de RAPID

En caso de que el usuario haya requerido desde Unity guardar el punto anteriormente calculado a un *path*, se ejecuta el Código 34. La base de esta parte del código son las variables persistentes i, un contador, y RT_i, que representan el *robottarget* de cada punto de la ruta. Cada vez que se añade un punto al path, se modifica RT_{i+1} y se incrementa i.

```
!! Add previous point to target
ELSEIF StrPart(message,2,1) <> "X" THEN
    TEST i
        CASE 0:
            RT1 := robotTarget;
            i := i + 1; !Increase counter
        CASE 1:
            RT2 := robotTarget;
            i := i + 1; !Increase counter
        CASE 2:
            RT3 := robotTarget;
            i := i + 1; !Increase counter
        CASE 3:
            RT4 := robotTarget;
            i := i + 1; !Increase counter
        CASE 4:
            RT5 := robotTarget;
            i := i + 1; !Increase counter
        DEFAULT:
            i := i;
    ENDTEST
    !Close client and server sockets
    SocketClose client;
    SocketClose server;
```

Código 34: *MainModule.mod* - Añadir un punto al *path* del robot

En caso de que el usuario haya requerido ejecutar el *path* guardado hasta el momento, se ejecuta el Código 35. Este código primero activa el *flag executingPath*, utilizado por la segunda tarea para mandar a Unity los ángulos del robot durante el movimiento por UDP. Además, utilizando la sentencia *ConfJ \Off*, se especifica que si el robot no puede alcanzar el objetivo en una configuración, que pruebe con las demás posibles [77]. Una vez hecho esto, se hacen los movimientos mediante la instrucción *MoveJ*, que mueve el robot utilizando las articulaciones, sin reparar en si el recorrido es lineal [78]. Al terminar, se resetea el *flag executingPath* y se cierran las comunicaciones.

```
!! Add previous point to target
ELSEIF StrPart(message,2,1) = "X" THEN
    !Raise executingPath flag
    executingPath := TRUE;
    !Ignore current robot configuration during linear movement
    ConfJ \Off;
    TEST i
        CASE 1:
            MoveJ RT1, v50, z5, tool0;

        CASE 2:
            MoveJ RT1, v50, z5, tool0;
            MoveJ RT2, v50, z5, tool0;

        CASE 3:
            MoveJ RT1, v50, z5, tool0;
            MoveJ RT2, v50, z5, tool0;
            MoveJ RT3, v50, z5, tool0;

        CASE 4:
            MoveJ RT1, v50, z5, tool0;
            MoveJ RT2, v50, z5, tool0;
            MoveJ RT3, v50, z5, tool0;
            MoveJ RT4, v50, z5, tool0;

        CASE 5:
            MoveJ RT1, v50, z5, tool0;
            MoveJ RT2, v50, z5, tool0;
            MoveJ RT3, v50, z5, tool0;
            MoveJ RT4, v50, z5, tool0;
            MoveJ RT5, v50, z5, tool0;
    ENDTEST
    !Reset executingPath flag
    executingPath := FALSE;
    WaitTime 5;
    SocketSend client,\str:="error";
    SocketClose client;
    SocketClose server;
```

Código 35: *MainModule.mod* - Ejecución del *path* guardado

6.7. Tarea secundaria

Como una tarea de RAPID no puede avanzar su *program pointer* mientras está ejecutando una instrucción de movimiento como *MoveJ*, por ejemplo, es necesario hacer una tarea secundaria para enviar los ángulos del robot a la simulación de Unity mientras el robot se está moviendo en RobotStudio.

Esta tarea (Código 36), simplemente crea un *socket* UDP, y cuando la tarea principal activa el booleano *executingPath*, el robot muestrea sus ángulos de articulaciones actuales con la instrucción *CJointT()* y se envían por el *socket* UDP a Unity, tras haberlos convertidos a una cadena de caracteres con la función *JointTargetToString()*, explicada anteriormente en el Código 32.

```
PROC main()
    SocketCreate udpSocket \UDP;
    WHILE TRUE DO
        IF executingPath THEN
            jointangles := CJointT();
            SocketSendTo udpSocket,"192.168.1.143",7000 \Str :=
                jointTargetToString(jointangles);
        ENDIF

        WaitTime 0.05;
    ENDWHILE
ENDPROC
```

Código 36: *SecondaryModule.mod* - Muestreo de los ángulos durante el movimiento

7. PLC Virtual

7.1. Programación del PLC

7.1.1. Introducción

La programación del PLC virtual será idéntica a la de un PLC físico del tipo S7-1500, utilizando el software TIA Portal V14. El lenguaje de programación elegido será *Graph*, debido a su sencillez y parecido con los diagramas de grafcet, utilizados en automatización industrial para describir procesos de forma secuencial [79].

El modelo de ejecución de un PLC se basa en una tarea cíclica, denominada *OB1*. Dentro de esta tarea, se podrán hacer llamadas a distintas funciones. En este caso, todas las funciones se programarán en *FB's* o *Function Blocks*, ya que los bloques de función tienen la ventaja de que pueden ser instanciados en el *OB1* cuantas veces sea necesario. Para que cada instancia sea independiente, cada vez que se instancia un *FB* se creará automáticamente un *DB*, o *Data Block*, que almacena los datos y el estado de ejecución de cada instancia por separado.

7.1.2. Grafcet de emergencia

El grafcet de emergencia (Figura 61) tiene cuatro etapas:

1. *gsNormal*. Modo de operación normal. Al entrar a esta etapa se pondrán a 1 las variables de memoria *mgModoAuto* y *mgModoManual*, que serán las posiciones de memoria que indiquen en qué modo de operación se encuentra el programa. Posteriormente, serán sobreescritas por el grafcet que gestiona el modo manual y automático. Para salir de esta etapa, se tiene que activar la seta de emergencia. Al salir de esta etapa, se resetean estas dos posiciones de memoria.

2. *gsEmergencia*. Modo de emergencia, todas las salidas del PLC se ponen a 0 mientras se encuentre en esta etapa. Para salir de esta etapa, se tiene que desactivar la seta de emergencia.

3. *gsRearme1*. Primer modo de rearme. Todas las salidas siguen a 0, y no se saldrá de esta etapa hasta que se pulse el botón de rearme.

4. *gsRearme2*. Segundo modo de rearme. Todas las salidas se mantienen a 0, y no se saldrá de esta etapa hasta que no se ponga el conmutador en modo manual.

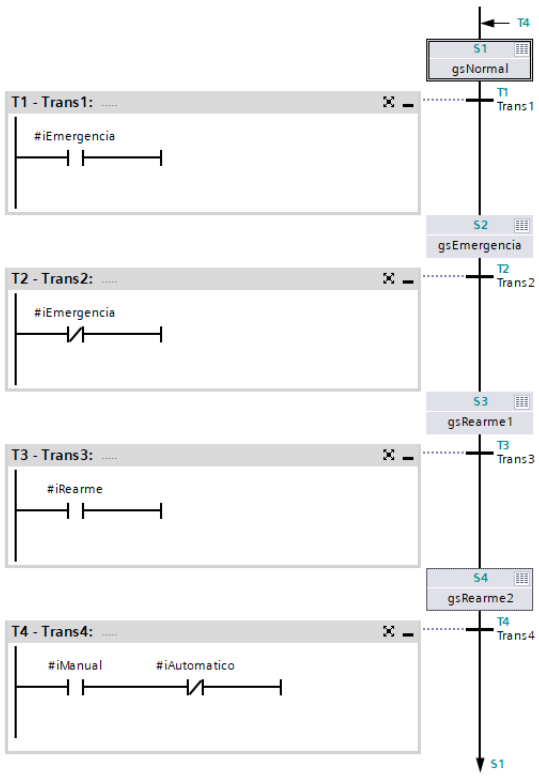


Figura 61: Grafcet de emegencia

La llamada a esta función en el OB1 se muestra en la Figura 62.

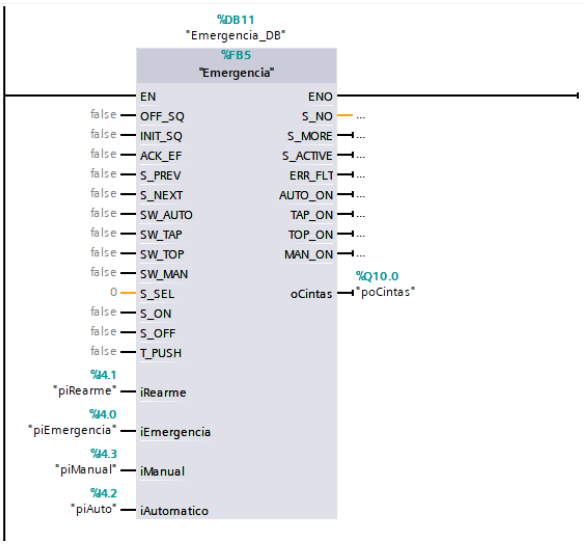


Figura 62: Llamada a grafcet de emegencia en OB1

7.1.3. Grafcet de gestión de modos

El grafcet de gestión de modos automático y manual (Figura 63) tiene dos etapas:

1. *gsNoAuto*. Modo manual. Al entrar a esta etapa se pone a 0 *mgModoAuto* y se pone a 1 *mgModoManual*. Para salir de esta etapa, el conmutador se tiene que poner en modo automático.

2. *gsAuto*. Modo automático. Al entrar en esta etapa se pone a 0 *mgModoManual*, y se pone a 1 *mgModoAuto*. Para salir de esta etapa el conmutador tiene que dejar de estar en modo automático.

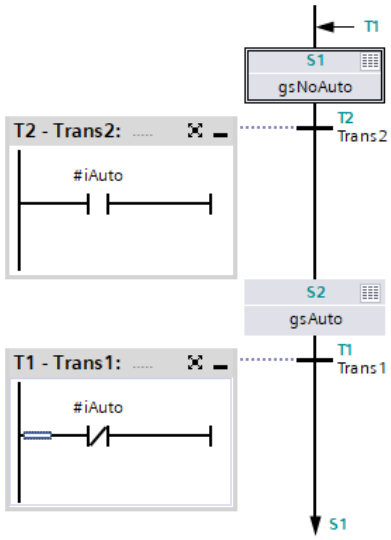


Figura 63: Grafcet de gestión de modo automático manual

La llamada de este gracfet se muestra en la Figura 64, y su activación está condicionada a que el programa se encuentre en modo automático o manual (*mgModoAuto* OR *mgModoManual*).

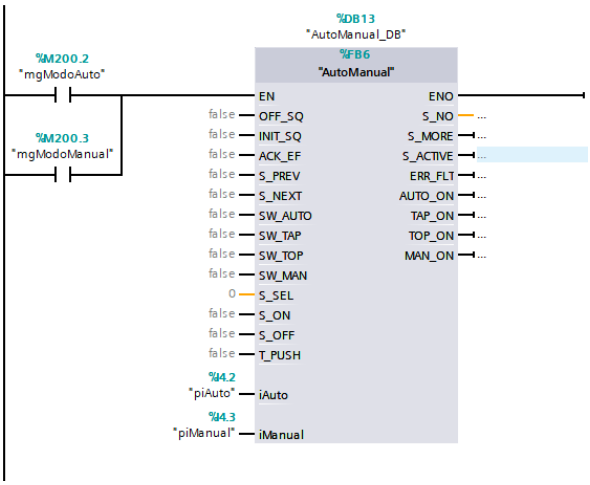


Figura 64: Llamada a grafcet de gestión de modo automático manual en OB1

7.1.4. Grafcet de cintas transportadoras

El grafcet de gestión las cintas transportadoras (Figura 65) tiene dos etapas:

1. *gsParada*. Motores de las cintas parados. Al estar en esta etapa se pone a 0 la salida digital que activa los motores. Para salir de esta etapa, el PLC tiene que recibir la órden de marcha desde el pupitre de control.

2. *gsMarcha*. Motores de las cintas en marcha. Al estar en esta etapa se pone a 1 la salida digital que activa los motores. Para salir de esta etapa el conmutador tiene que dejar de recibir la orden de marcha.

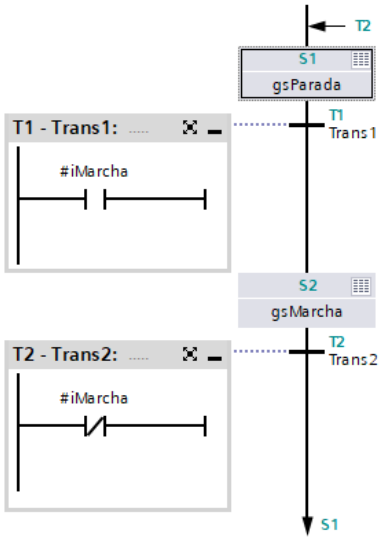


Figura 65: Grafcet de cintas transportadoras

La llamada de este gracfet se muestra en la Figura 66, y es necesario introducir un *delay* de 10 ms para que la gestión de grafcets se haga de forma jerárquica, y pueda arrancar sin problemas. Este *delay* tiene que ir tanto a la entrada *EN* o *Enable*, como a la entrada *INIT_SQ* o *Initiate sequence*.

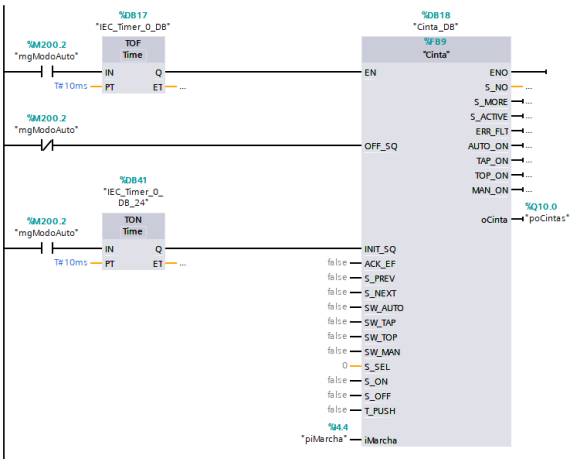


Figura 66: Llamada a grafcet de cintas transportadoras en OB1

7.2. Desarrollo interfaz PLCSIM

Una vez programado el PLC, es necesario desarrollar un programa que haga de interfaz entre PLCSIM y Unity, para escribir las entradas que envía Unity y asignarlas a las entradas del PLC virtual, y viceversa. El proyectista ya había desarrollado tal aplicación para otro proyecto, aunque con una pequeña diferencia de diseño. La primera versión se programó con el protocolo de comunicación de *Pipes*. Este protocolo, exclusivo de ordenadores con sistema operativo Windows, sirve para poder comunicar procesos. Sin embargo, esta elección de diseño tiene un gran inconveniente, que la simulación de PLCSIM y Unity tienen que estar corriendo en el mismo ordenador. Esto limita la flexibilidad del sistema, por lo que para este proyecto se ha reimplementado utilizando comunicación por *socket* TCP/IP, de forma análoga al código desarrollado en el apartado 5.6.1.

El código se puede revisar en el Anexo 3, pero explicado de forma sencilla, su funcionamiento es el siguiente:

1. Utilizando la API de PLCSIM Advanced, se registra una instancia de un PLC virtual de tipo S7-1500, si no está corriendo ya.

2. Se inicia el hilo o *Thread* de comunicaciones, que envía y recibe datos de Unity.

3. En el método *instance_OnEndOfCycle()*, ejecutado al final de cada ciclo del PLC virtual, se envían las salidas del PLC virtual a Unity, y se actualizan las entradas del PLC con el mensaje recibido de Unity.

8. Instalación del Oculus Rift

8.1. Hardware

El Oculus Rift (ver Figura 67) es un sistema de realidad virtual que se compone de los siguientes elementos:

- HMD de realidad virtual
- Cámaras para hacer el seguimiento
- Mandos inalámbricos

Figura 67: Hardware del Oculus Rift

La instalación del hardware del Oculus Rift es bastante sencilla. Los dos sensores tienen que ir conectados al PC por un puerto USB 3.0, y el HMD, además de tener que ir conectado a otro puerto USB 3.0, también necesita conectarse por HDMI al ordenador. A la hora de instalarlo, es importante comprobar las especificaciones de los puertos. Si los puertos USB son 2.0, se producirán problemas debido a la menor tasa de transferencia de datos [80] (ver Tabla6).

Tasa transferencia	**USB 2.0**	**USB 3.0**
Teórica	480 Mb/s	5 Gb/s
Real	40 MB/s	300 MB/s

Tabla 6: Diferencia de rendimiento entre USB 2.0 y 3.0

Además, el Oculus Rift requiere mucha potencia gráfica, por lo que tiene que estar conectados a la tarjeta gráfica dedicada, ya que algunos ordenadores también dispone de un puerto HDMI en la placa base, una tarjeta gráfica integrada en el chip de la CPU con una potencia muy reducida.

8.2. Calibración del Oculus Rift

Una vez conectado todo el hardware, es necesario calibrar los sensores. Esta operación solo se tiene que realizar una vez, a no ser que las cámaras se muevan o se cambien de orientación. En este caso, es necesario repetir el proceso descrito a continuación.

En primer lugar, hay que especificar la altura del usuario del Oculus Rift, para que el software sepa donde calibrar la posición del suelo. Esto se realiza en la pantalla de la Figura 68.

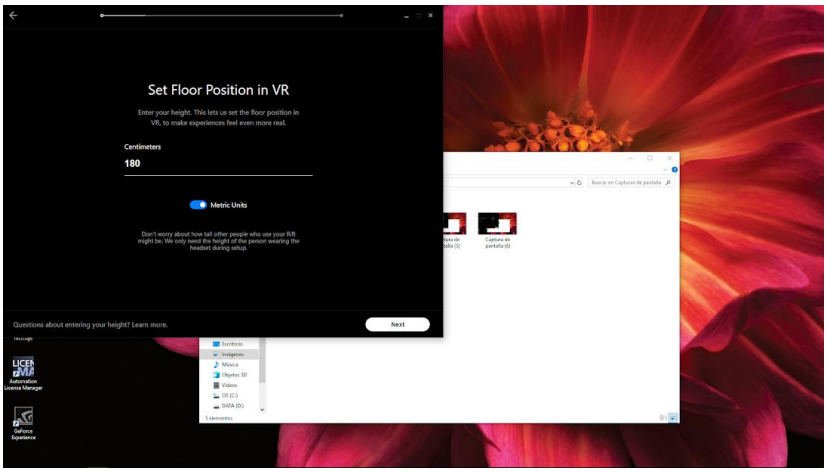


Figura 68: Pantalla *Set Floor Position in VR*

Antes de empezar la calibración del área de trabajo disponible, el usuario tiene que colocar los dos mandos inalámbricos a la vista de los sensores, y mantener pulsado el gatillo de su mano dominante durante unos segundos para poder empezar el proceso (Figura 69).

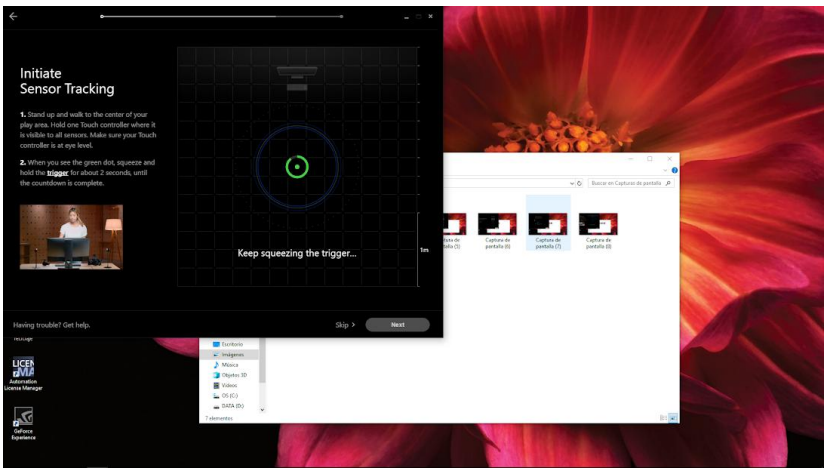


Figura 69: Pantalla *Initiate Sensor Tracking*

En el siguiente paso se configura la orientación de la habitación. Para ello, se mueve el mando inalámbrico de izquierda a derecha hasta que la posición de los sensores es correcta relativa a la posición del icono del monitor, y se pulsa el gatillo para confirmar (Figura 70).

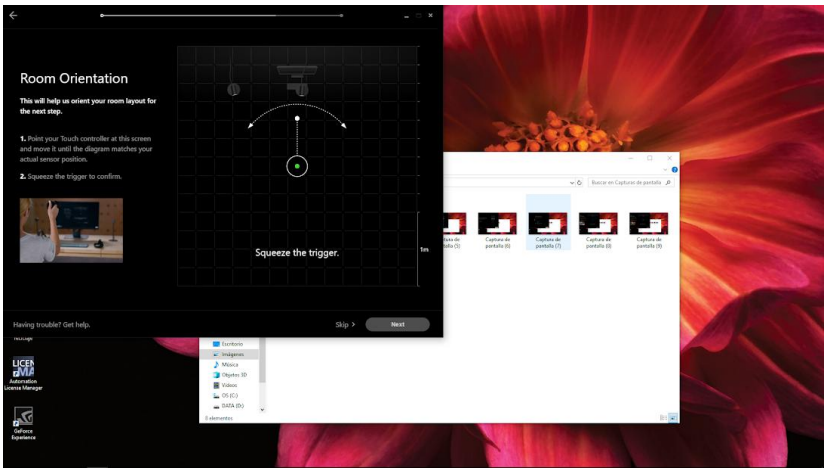


Figura 70: Pantalla *Room Orientation*

Por último, el usuario tiene que delimitar su área disponible utilizando el mando inalámbrico, y el *software* de Oculus distribuirá la superficie de forma óptima (ver Figura 71). Una vez configurado, cuando el usuario se acerque a los límites de la zona de trabajo, se mostrará una rejilla de neón para indicar al usuario que se está alejando de la zona en la que puede mover los mandos de forma segura.

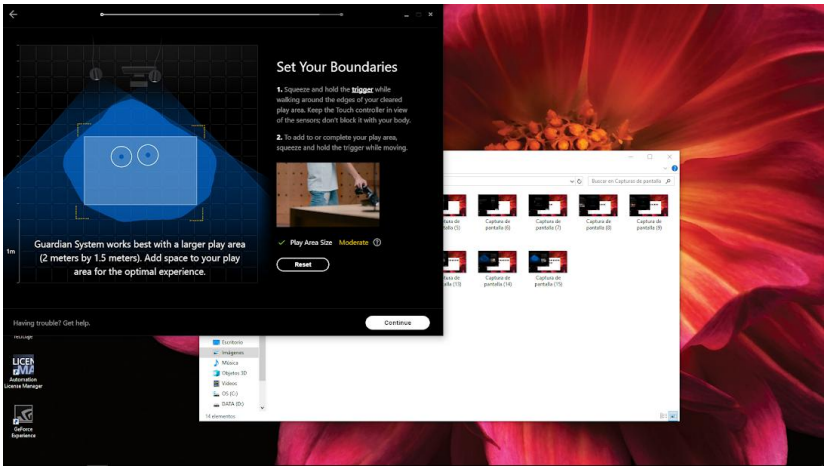


Figura 71: Pantalla *Set your boundaries*

Una vez realizados todos estos pasos, el Oculus Rift estará calibrado para la posición de los sensores y ya se podrá utilizar, siempre y cuando estos sensores no se muevan.

9. Resultados y conclusiones

En cuanto al programa de entrenamiento de operarios, se ha conseguido desarrollar un tutorial que muestre información al operario dependiendo de la operación que tiene que realizar, se simula la interacción con elementos como válvulas y herramientas, y se hace énfasis en la secuencia que tiene que realizar el usuario, por lo que se consideran cumplidos los objetivos del proyecto con respecto a esta área. Además, se ha conseguido simular la minifábrica ICAI, controlada por un autómata programable, y la programación podría ser portada a los PLCs físicos que controlan la planta, con lo que se consigue una interoperabilidad óptima, ya que no es necesario desarrollar código específico para la simulación.

Los resultados obtenidos en el módulo de programación del robot son muy positivos. No solo se ha demostrado la viabilidad del uso de la tecnología de realidad virtual para la programación de robots industriales, sino que tras la experiencia de usuarios que nunca habían utilizado un sistema de realidad virtual o nunca habían programado un robot industrial, resulta extremadamente intuitivo, comparado con los sistemas tradicionales, como una controladora FlexPendant, que requiere horas de entrenamiento para ser manejada con soltura. La flexibilidad que proporciona el poder indicarle el objetivo al robot con la mano en un entorno 3D, como la capacidad de cambiar de punto de vista simplemente moviendo la cabeza o agachándose, proporciona un nivel de control y facilidad de uso incomparables, con un tiempo de adaptación o entrenamiento mínimo.

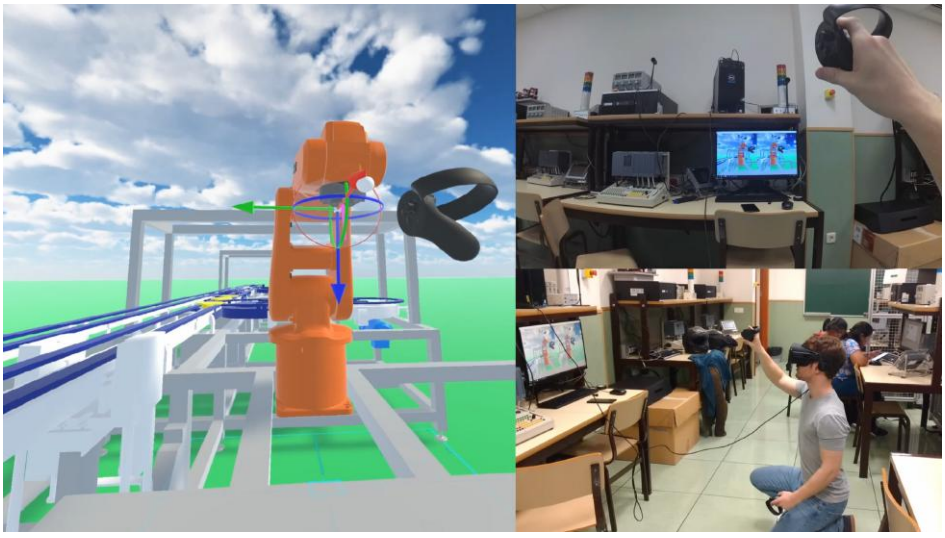


Figura 72: Usuario programando el robot en realidad virtual

La conclusión es que la tecnología de realidad virtual para la programación de robots industriales no solo es una posibilidad anecdótica, sino que probablemente en el futuro sea una de las formas estándar de interactuar con los manipuladores robóticos.

Bibliografía

[1] *Display Resolution*. Wikipedia. URL: https://en.wikipedia.org/wiki/Display_resolution (pág. 14).

[2] *3D TV is Dead*. Business Insider. 2017. URL: http://www.businessinsider.com/3d-tv-is-dead-2017-1 (pág. 15).

[3] C. Cruz, D. Sandin, T. DeFant, R. Kenyon y J. Hart. «The CAVE: audio visual experience automatic virtual environment». En: *Communications of the ACM* (1992) (pág. 15).

[4] *Oculus Rift: Step Into the Game*. Kickstarter. URL: https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game (pág. 16).

[5] *Facebook compra Oculus. La red social paga 1.450 millones de euros por la firma que desarrolla gafas de realidad virtual*. El País. URL: https://elpais.com/tecnologia/2014/03/26/actualidad/1395796446_034242.html (pág. 16).

[6] *Virtual Reality Industry Report: Spring 2017*. Greenlight Insights (pág. 16).

[7] EON Reality. URL: https://www.eonreality.com/ (pág. 17).

[8] *IRB 120 Industrial Robot Datasheet*. ABB (pág. 18).

[9] *RobotStudio 6.03 Release Notes*. ABB (pág. 18).

[10] *IVRE – An Immersive Virtual Robotics Environment*. Computational Interaction Robotics Laboratory. Johns Hopkins University. URL: https://cirl.lcsr.jhu.edu/research/human-machine-collaborative-systems/ivre/ (pág. 18).

[11] *YuMi meets HTC Vive*. Sumcom Garage, Youtube. URL: https://www.youtube.com/watch?v=16xArRG3ZuI (pág. 18).

[12] *Haptic Vive*. Queen's University, Belfast. URL: https://motherboard.vice.com/en_us/article/z43yp4/VR-robot-feedback (pág. 19).

[13] *IRB 120 CAD Models*. ABB. URL: https://new.abb.com/products/robotics/industrial-robots/irb-120/irb-120-cad (pág. 26).

[14] *DWG*. Wikipedia. URL: https://es.wikipedia.org/wiki/DWG (pág. 27).

[15] *3D Formats*. Unity Documentation. URL: https://docs.unity3d.com/Manual/3D-formats.html (pág. 27).

[16] James Reason. «Motion sickness». En: *Academic Press* (1975) (pág. 28).

[17] *Profiler Overview*. Unity Documentation. URL: https://docs.unity3d.com/Manual/Profiler.html (pág. 28).

[18] *3DS Max Overview*. Autodesk. URL: https://www.autodesk.es/products/3ds-max/overview (pág. 29).

[19] *3DS Max Suscripción*. Autodesk. URL: https://www.autodesk.es/products/3ds-max/subscribe?plc=3DSMAX (pág. 29).

[20] *Optimize Mesh*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/MeshUtility.Optimize.html (pág. 31).

[21] *Box Collider*. Unity Documentation. URL: https://docs.unity3d.com/Manual/class-BoxCollider.html (pág. 33).

[22] *Rigidbody*. Unity Documentation. URL: https://docs.unity3d.com/Manual/class-Rigidbody.html (pág. 33).

[23] *Rigidbody.AddForce*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html (pág. 34).

[24] *Rigidbody.MovePosition*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Rigidbody.MovePosition.html (pág. 35).

[25] *Kinematics of moving frames*. 2.017J Design of Electromechanical Robotic Systems - MIT OpenCourseWare, 2009 (pág. 36).

[26] *Quaternion.AngleAxis*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Quaternion.AngleAxis.html (pág. 36).

[27] *Rigidbody.MoveRotation*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Rigidbody.MoveRotation.html (pág. 37).

[28] José Porras Galán y Silvia Fernandez Villamarín. *Mecánica de un Robot*. ICAI (págs. 39, 40).

[29] *Time.deltaTime*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Time-deltaTime.html (pág. 40).

[30] *Quaternion.Lerp*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Quaternion.Lerp.html (pág. 41).

[31] *Threading (C#)*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/ (pág. 44).

[32] *TCP vs UDP*. Diffen. URL: https://es.diffen.com/tecnologia/TCP-vs-UDP (pág. 44).

[33] *C# Thread Termination and Thread.Abort()*. Stack Overflow. URL: https://stackoverflow.com/questions/2251964/c-sharp-thread-termination-and-thread-abort (pág. 45).

[34] *Clase TcpClient*. Microsoft. URL: https://msdn.microsoft.com/es-es/library/system.net.sockets.tcpclient(v=vs.110).aspx (pág. 46).

[35] *GetBytes() Method*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/system.text.asciiencoding.getbytes(v=vs.110).aspx (pág. 46).

[36] *Try-catch*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch (pág. 47).

[37] *NetworkStream.Close() Method*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/system.net.sockets.networkstream.close(v=vs.110).aspx (pág. 47).

[38] *TcpClient.Close() Method*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient.close(v=vs.110).aspx (pág. 47).

[39] *NetworkStream.Write() Method*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/system.net.sockets.networkstream.write(v=vs.110).aspx (pág. 47).

[40] *String.Contains() Method*. Microsoft. URL: https://msdn.microsoft.com/es-es/library/dy85x1sa(v=vs.110).aspx (pág. 47).

[41] *Debug.DrawLine*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Debug.DrawLine.html (pág. 51).

[42] *Line Renderer*. Unity Documentation. URL: https://docs.unity3d.com/Manual/class-LineRenderer.html (pág. 51).

[43] *Converting between coordinate systems*. Meshola. URL: http://www.meshola.com/Articles/converting-between-coordinate-systems (pág. 55).

[44] Gregory G. Slabaugh. *Computing Euler angles from a rotation matrix*. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.371.6578&rep=rep1&type=pdf (pág. 55).

[45] *Gizmo - Traducción*. Oxford Dictionaries. URL: https://es.oxforddictionaries.com/traducir/ingles-espanol/gizmo (pág. 58).

[46] *SphereCollider*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/SphereCollider.html (pág. 58).

[47] *Collider.OnTriggerEnter(Collider)*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html (pág. 58).

[48] *Collider.OnTriggerStay(Collider)*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Collider.OnTriggerStay.html (pág. 58).

[49] *Collider.OnTriggerExit(Collider)*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Collider.OnTriggerExit.html (pág. 59).

[50] *Vector3.Project*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Vector3.Project.html (pág. 59).

[51] *Vector3.ProjectOnPlane*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Vector3.ProjectOnPlane.html (pág. 62).

[52] *Input for OpenVR controllers*. Unity Documentation. URL: https://docs.unity3d.com/Manual/OpenVRControllers.html (pág. 65).

[53] Valve. *SteamVR Documentation* (pág. 65).

[54] *Object.Instantiate*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Object.Instantiate.html (pág. 66).

[55] Jeff W. Murray. *Building Virtual Reality with Unity and SteamVR*. 2017 (pág. 68).

[56] Juho Hamari, Jonna Koivisto y Harri Sarsa. *Does Gamification Work? — A Literature Review of Empirical Studies on Gamification*. 2014 (pág. 71).

[57] *Transform.RotateAround*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Transform.RotateAround.html (pág. 72).

[58] ABB. *Product Specifications - IRB 120* (pág. 76).

[59] José Antonio Rodríguez Mondéjar. *Robótica Industrial*. ICAI (pág. 76).

[60] ABB. *Product manual - IRB 120* (pág. 76).

[61] Alan Zucconi. *An Introduction to Gradient Descent*. URL: https://alanzucconi.com/2017/04/10/gradient-descent/ (pág. 78).

[62] *RobotStudio*. ABB. URL: https://new.abb.com/products/robotics/es/robotstudio (pág. 79).

[63] ABB Documentation. *Robtarget - Position data*. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc545.html (pág. 81).

[64] ABB Documentation. *Confdata - Robot configuration data*. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc498.html (pág. 81).

[65] ABB Documentation. *Extjoint - Position of external joints*. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc515.html (pág. 82).

[66] *SocketCreate - Create a new socket*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc207.html (pág. 84).

[67] *SocketBind - Bind a socket to my IP-address and port*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc204.html (pág. 84).

[68] *SocketListen - Listen for incoming connections*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc208.html (pág. 84).

[69] *SocketAccept - Accept an incoming connection*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc203.html (pág. 84).

[70] *SocketReceive - Receive data from remote computer*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc209.html (pág. 84).

[71] *UnpackRawBytes - Unpack data from rawbytes data*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc275.html (pág. 84).

[72] *StrToVal - Converts a string to a value*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc464.html (pág. 85).

[73] *OrientZYX - Builds an orient from euler angles*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc275.html (pág. 85).

[74] *jointtarget - Joint position data*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc522.html (pág. 86).

[75] *SocketSend - Send data to remote computer*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc211.html (pág. 86).

[76] *SocketClose - Close a socket*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc205.html (pág. 86).

[77] *ConfJ - Controls the configuration during joint movement*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc28.html (pág. 89).

[78] *MoveJ - Moves the robot by joint movement*. ABB Documentation. URL: http://developercenter.robotstudio.com/BlobProxy/manuals/RapidIFDTechRefManual/doc117.html (pág. 89).

[79] *Grafcet*. Wikipedia. URL: https://es.wikipedia.org/wiki/GRAFCET (pág. 92).

[80] *What is USB 3.1?* Kingston. URL: https://www.kingston.com/en/usb/usb_30 (pág. 98).

Presupuesto

1. Mediciones

1.1. Componentes

Componente	Cantidad [uds.]
Oculus Rift	1
Ordenador compatible con RV	1

Tabla 1: Mediciones de componentes

1.2. Equipo y herramientas

Componente	Cantidad [uds.]
Ordenador Personal	1

Tabla 2: Mediciones de equipo y herramientas

1.3. Software

Programa	Cantidad [uds.]
Unity 5	1
RobotStudio	1
Siemens TIA Portal	1
Siemens PLCSIM Advanced	1
MikTeX	1
Texmaker	1
AutoCAD	1
3DS Max	1

Tabla 3: Mediciones de software

1.4. Ingeniería

Tarea	**Horas**
Anteproyecto	75
Diseño	25
Programación	250
Documentación	50

Tabla 4: Mediciones de ingeniería

2. Precios unitarios

2.1. Componentes

Componente	**Precio unitario [€/ud.]**
Oculus Rift	449
Ordenador compatible con RV	1.500

Tabla 5: Precios unitarios de componentes

2.2. Equipo y herramientas

Componente	**Precio unitario [€/ud.]**
Ordenador Personal	500

Tabla 6: Precios unitarios de equipo y herramientas

2.3. Software

Programa	**Precio unitario [€/ud.]**
Unity 5	Licencia gratuita
RobotStudio	Licencia universitaria
Siemens TIA Portal	Licencia universitaria
Siemens PLCSIM Advanced	Licencia universitaria
MikTeX	Licencia gratuita
Texmaker	Licencia gratuita
AutoCAD	Prueba gratuita
3DS Max	Prueba gratuita

Tabla 7: Precios unitarios de software

2.4. Ingeniería

Tarea	**Precio [€/hora]**
Anteproyecto	20
Diseño	25
Programación	30
Documentación	20

Tabla 8: Precios unitarios de ingeniería

3. Sumas parciales

3.1. Componentes

Componente	Cantidad [uds.]	Precio unitario [€/ud.]	Total [€]
Oculus Rift	1	449	449
Ordenador compatible con RV	1	1.500	1.500
		TOTAL	1.949

Tabla 9: Suma parcial de componentes

3.2. Equipo y herramientas

Componente	Cantidad [uds.]	Precio unitario [€/ud.]	Total [€]
Ordenador Personal	1	500	500
		TOTAL	500

Tabla 10: Suma parcial de equipo y herramientas

3.3. Software

Programa	Cantidad [uds.]	Precio unitario [€/ud.]	Total [€]
Unity 5	1	Licencia gratuita	0
RobotStudio	1	Licencia universitaria	0
Siemens TIA Portal	1	Licencia universitaria	0
Siemens PLCSIM Advanced	1	Licencia universitaria	0
MikTeX	1	Licencia gratuita	0
Texmaker	1	Licencia gratuita	0
AutoCAD	1	Prueba gratuita	0
3DS Max	1	Prueba gratuita	0
		TOTAL	0

Tabla 11: Suma parcial de software

3.4. Ingeniería

Tarea	**Horas**	**Precio [€/hora]**	**Total [€]**
Anteproyecto	75	20	1.500
Diseño	25	25	625
Programación	250	30	7.500
Documentación	50	20	1.000
		TOTAL	10.625

Tabla 12: Suma parcial de ingeniería

4. Presupuesto general

Concepto	**Coste [€]**
Componentes	1.949
Equipos y herramientas	500
Software	0
Ingeniería	10.625
TOTAL	13.074

Tabla 13: Presupuesto general del proyecto

Código Fuente

1. Unity

1.1. IRB120.cs

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

public class IRB120 : MonoBehaviour {

    // Target angles
    public float a1;
    public float a2;
    public float a3;
    public float a4;
    public float a5;
    public float a6;

    // Angular movement speed
    public int speed = 5;

    // Robot axii
    private Transform tAxis1;
    private Transform tAxis2;
    private Transform tAxis3;
    private Transform tAxis4;
    private Transform tAxis5;
    private Transform tAxis6;


    // Use this for initialization
    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>();

    }

    // Update is called once per frame
    void Update()
    {
        // Linear interpolate to angle
        tAxis1.localRotation = Quaternion.Lerp(tAxis1.localRotation,
            Quaternion.Euler(0, -a1, 0), speed * Time.deltaTime);
        tAxis2.localRotation = Quaternion.Lerp(tAxis2.localRotation,
            Quaternion.Euler(0, 0, -a2), speed * Time.deltaTime);
        tAxis3.localRotation = Quaternion.Lerp(tAxis3.localRotation,
            Quaternion.Euler(0, 0, -a3), speed * Time.deltaTime);
        tAxis4.localRotation = Quaternion.Lerp(tAxis4.localRotation,
            Quaternion.Euler(-a4, 0, 0), speed * Time.deltaTime);
        tAxis5.localRotation = Quaternion.Lerp(tAxis5.localRotation,
            Quaternion.Euler(0, 0, -a5), speed * Time.deltaTime);
        tAxis6.localRotation = Quaternion.Lerp(tAxis6.localRotation,
            Quaternion.Euler(-a6, 0, 0), speed * Time.deltaTime);

    }
}
```

1.2. VRController.cs

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

[RequireComponent(typeof(SteamVR_TrackedObject))]
public class ViveController : MonoBehaviour
{
    public GameObject target;
    public GameObject origin;
    private Communications comms;
    private Hand hand;



    private LineRenderer lrWX;
    private LineRenderer lrWY;
    private LineRenderer lrWZ;
    private LineRenderer lrTX;
    private LineRenderer lrTY;
    private LineRenderer lrTZ;

    private Vector3 savedPosition;
    private Vector3 savedRotationEuler;

    private WorldCoodSystem worldCoord;
    private ToolCoordSystem toolCoord;

    public float angleX;
    public float angleY;
    public float angleZ;

    //save axis for gizmo rotation
    public Vector3 savedXAxis;
    public Vector3 savedYAxis;
    public Vector3 savedZAxis;


    public Quaternion rotationGizmoRecalc;
    public bool rotationDone;

    public bool recalculateGizmo;

    private GameObject marker;

    // Use this for initialization
    void Start()
    {
        target = GameObject.Find("RobotTarget");
        origin = GameObject.Find("Origin");

        comms = GameObject.Find("Communications").GetComponent<Communications>();

        worldCoord = GameObject.Find("World").GetComponent<WorldCoodSystem>();
        toolCoord = GameObject.Find("Tool").GetComponent<ToolCoordSystem>();
        recalculateGizmo = false;
        marker = GameObject.Find("Marker");

        rotationGizmoRecalc = Quaternion.identity;

    }

    void Awake()
    {
```

```
}

// Update is called once per frame
void Update()
{

    //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);

    //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;



    Vector3 res =
        (target.transform.parent.Find("TranslationGizmo").transform.position -
        origin.transform.position) * 1000;

    if (hand == null)
    {
        hand = this.GetComponent<Hand>();
        Debug.Log(hand.name);
    }

    if (hand.controller == null)
        return;


    //Debug.Log(SteamVR_Controller.ButtonMask.Axis3);

    //Move robot target gizmo while pressing grip
    if (hand.controller.GetPress(SteamVR_Controller.ButtonMask.Grip))
    {
        savedPosition = hand.transform.position;
        savedRotationEuler = new Vector3(angleX, angleY, angleZ);


        if (hand.controller.GetPress(SteamVR_Controller.ButtonMask.Touchpad))
        {
            savedRotationEuler = new Vector3(180, 0, SnapAngle(angleZ, 45));
            Vector3 correctedRot = new Vector3(0,
                SnapAngle(hand.transform.rotation.eulerAngles.y, 45), 0);
            target.transform.parent.Find("TranslationGizmo").transform.position =
                hand.transform.position;
```

```
            target.transform.parent.Find("RotationGizmo").transform.rotation =
                Quaternion.Euler(correctedRot);

            //Check robot target reachability on grip release
            if (hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Grip))
            {
                SendMessage(res, savedRotationEuler, 0, false, false);
                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;

            }

        }
        else
        {
            target.transform.parent.Find("TranslationGizmo").transform.position =
                hand.transform.position;
            target.transform.parent.Find("RotationGizmo").transform.rotation =
                hand.transform.rotation;
        }

    }

    //Check robot target reachability on grip release
    if (hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Grip))
    {
        SendMessage(res, savedRotationEuler, 0, false, false);
        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;

    }

    if (recalculateGizmo)
    {
        recalculateGizmo = false;
        res =
            (target.transform.parent.Find("TranslationGizmo").transform.position
            - origin.transform.position) * 1000;

        if (rotationDone)
        {
            rotationDone = false;
            Quaternion rot;
            rot = Quaternion.Euler(savedRotationEuler);
            rot = rot * rotationGizmoRecalc;

            SendMessage(res,GizmoToAngles(target.transform.parent.Find(
                "RotationGizmo").transform.rotation) , 0, false, false);
            savedRotationEuler = GizmoToAngles(target.transform.parent.Find(
                "RotationGizmo").transform.rotation);
        }
        else
            SendMessage(res, savedRotationEuler, 0, false, false);
        //save axis for gizmo rotation
        savedXAxis = target.GetComponent<TransformGizmo>().x_axis;
        savedYAxis = target.GetComponent<TransformGizmo>().y_axis;
        savedZAxis = target.GetComponent<TransformGizmo>().z_axis;
    }
```

```
        //Save point to
        if
            (hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.ApplicationMenu))
        {
            SendMessage(res, savedRotationEuler, 0, true, false);
            Instantiate(marker,
                target.transform.parent.Find("TranslationGizmo").transform.position,
                target.transform.parent.Find("RotationGizmo").transform.rotation);
        }

        if (hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Touchpad))
        {
            SendMessage(res, savedRotationEuler, 0, false, true);
        }









    }


    void SendMessage(Vector3 position, Vector3 rotation, int cfx, bool savePoint,
        bool executePath)
    {
        //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;

        //Calculate cf1 according to its quadrant in horizontal plane
        if (x >= 0 && y <= 0)
        {

            cf1 = -1;
        }
        else if (x >= 0 && y > 0)
        {
            //ok
            cf1 = 0;

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

        }
        else if (x < 0 && y > 0)
        {
            //ok
            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;

    //Message control
    if (savePoint & !executePath)
        comms.message = "X";    //Save point into robot path
    else if (executePath & !savePoint)
        comms.message = "XX";   //Execute robot path
    else
        comms.message = "";     //Calculate joint angles

    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") + ";";
    //Debug.Log(comms.message);
    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;
    }
}
```

1.3. ControllerTip.cs

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

public class ControllerTip : MonoBehaviour {


    public int precision;

    private Hand hand;

    private bool manipulating;
    private Vector3 savedGizmoPosition;
    private Vector3 savedHandPosition;

    private Quaternion savedGizmoRotation;
    private Vector3 savedVectorX;
    private Vector3 savedVectorY;
    private Vector3 savedVectorZ;
    private Vector3 savedGizmoToController;
    private Vector3 savedAxis;

    private VRController vc;

    private string savedTag;

    private TransformGizmo tg;


    private GameObject translationGizmo;
    private GameObject rotationGizmo;



    // Use this for initialization
    void Start () {
        manipulating = false;
        hand = transform.GetComponentInParent<Hand>();
        tg = GameObject.Find("RobotTarget").GetComponent<TransformGizmo>();

        precision = 1;
        vc = hand.GetComponent<VRController>();


        translationGizmo = GameObject.Find("TranslationGizmo");
        rotationGizmo = GameObject.Find("RotationGizmo");
    }

  // Update is called once per frame
  void Update () {
        if (manipulating &
            hand.controller.GetPress(SteamVR_Controller.ButtonMask.Trigger))
        {
            //Linear movement
```

```
            if(savedTag == "x_axis" || savedTag == "y_axis" || savedTag == "z_axis")
            {
                Vector3 movement = Vector3.zero;
                Vector3 offset = Vector3.zero;
                Vector3 axis = Vector3.zero;

                if (savedTag == "x_axis")
                    axis = tg.transform.forward;
                else if (savedTag == "y_axis")
                    axis = tg.transform.right;
                else if (savedTag == "z_axis")
                    axis = tg.transform.up;

                movement = Vector3.Project(this.transform.position -
                    savedGizmoPosition, axis);
                offset = Vector3.Project(savedHandPosition - savedGizmoPosition,
                    axis);
                translationGizmo.transform.position = savedGizmoPosition + (movement
                    - offset) / precision;
            }
            //Rotational movement
            if (savedTag == "x_torus" || savedTag == "y_torus" || savedTag ==
                "z_torus")
            {

                float angle = 0f;
                Vector3 gizmoToController =
                    Vector3.ProjectOnPlane(this.transform.position -
                    savedGizmoPosition, savedAxis);
                angle = Vector3.SignedAngle(savedGizmoToController,
                    gizmoToController, savedAxis);
                if (savedTag == "z_torus")
                    savedAxis = -savedAxis;
                rotationGizmo.transform.rotation = savedGizmoRotation *
                    Quaternion.AngleAxis(angle, -savedAxis);

                vc.rotationGizmoRecalc = Quaternion.AngleAxis(angle, -savedAxis);
                vc.rotationDone = true;
            }

        }

        if (manipulating &
            hand.controller.GetPressUp(SteamVR_Controller.ButtonMask.Trigger))
        {
            //Raise recalculate flag
            vc.recalculateGizmo = true;

            //Reset everything
            manipulating = false;
            savedTag = "";
            savedGizmoPosition = Vector3.zero;
            savedHandPosition = Vector3.zero;
            savedGizmoRotation = Quaternion.identity;
        }
    }

    void OnTriggerStay(Collider coll)
    {

        if (coll.tag == "valve" &
            hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Trigger))
        {
            Debug.Log("vaaaaaaalve");
            coll.GetComponentInParent<AirValve>().closed =
                !coll.GetComponentInParent<AirValve>().closed;
        }

        if (coll.tag == "x_axis" || coll.tag == "y_axis" || coll.tag == "z_axis" ||
            coll.tag == "x_torus" || coll.tag == "y_torus" || coll.tag == "z_torus")
```

```
        {
            if (!manipulating &
                hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Trigger))
            {
                savedTag = coll.tag;
                manipulating = true;
                //Save positions
                savedGizmoPosition = translationGizmo.transform.position;
                savedHandPosition = hand.transform.position;
                //Save rotation
                savedGizmoRotation = rotationGizmo.transform.rotation;

                if (savedTag == "x_torus")
                    savedAxis = Vector3.forward;
                else if (savedTag == "y_torus")
                    savedAxis = Vector3.right;
                else if (savedTag == "z_torus")
                    savedAxis = Vector3.down;

                savedGizmoToController = Vector3.ProjectOnPlane(savedHandPosition -
                    savedGizmoPosition, savedAxis);

            }
        }
    }
}
```

1.4. ToolCoordSystem.cs

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

public class ToolCoordSystem : MonoBehaviour {

    public Transform handTransform;

    public Transform toolXTransform;
    public Transform toolYTransform;
    public Transform toolZTransform;

    private LineRenderer lrToolX;
    private LineRenderer lrToolY;
    private LineRenderer lrToolZ;

    public bool lrEnabled = false;
    private int lrSizeOffset = 100;

    public Vector3 toolX;
    public Vector3 toolY;
    public Vector3 toolZ;



    // Use this for initialization
    void Start () {
        handTransform = GameObject.Find("Hand2").transform;

        toolXTransform = this.transform.Find("ToolX");
        toolYTransform = this.transform.Find("ToolY");
        toolZTransform = this.transform.Find("ToolZ");

        lrToolX = this.transform.Find("ToolX").GetComponent<LineRenderer>();
        lrToolY = this.transform.Find("ToolY").GetComponent<LineRenderer>();
```

```
        lrToolZ = this.transform.Find("ToolZ").GetComponent<LineRenderer>();

        this.transform.Find("ToolX").GetComponent<MeshRenderer>().enabled = false;
        this.transform.Find("ToolY").GetComponent<MeshRenderer>().enabled = false;
        this.transform.Find("ToolZ").GetComponent<MeshRenderer>().enabled = false;

        lrToolX.useWorldSpace = false;
        lrToolY.useWorldSpace = false;
        lrToolZ.useWorldSpace = false;

        lrToolX.positionCount = 2;
        lrToolY.positionCount = 2;
        lrToolZ.positionCount = 2;

        lrToolX.startWidth = 0.01f;
        lrToolY.startWidth = 0.01f;
        lrToolZ.startWidth = 0.01f;

        lrToolX.endWidth = 0.01f;
        lrToolY.endWidth = 0.01f;
        lrToolZ.endWidth = 0.01f;

    }

    // Update is called once per frame
    void Update () {
        //Calculate coordinate system vectos
        toolX = toolXTransform.position - transform.position;
        toolY = toolYTransform.position - transform.position;
        toolZ = toolZTransform.position - transform.position;
        //Normalize said vectors
        toolX = Vector3.Normalize(toolX);
        toolY = Vector3.Normalize(toolY);
        toolZ = Vector3.Normalize(toolZ);


        if (lrEnabled)
        {
            if (!lrToolX.enabled || !lrToolY.enabled || !lrToolZ.enabled)
            {
                lrToolX.enabled = true;
                lrToolY.enabled = true;
                lrToolZ.enabled = true;

                this.transform.Find("ToolX").GetComponent<MeshRenderer>().enabled =
                    true;
                this.transform.Find("ToolY").GetComponent<MeshRenderer>().enabled =
                    true;
                this.transform.Find("ToolZ").GetComponent<MeshRenderer>().enabled =
                    true;

            }


            lrToolX.SetPosition(0, toolXTransform.localPosition * -lrSizeOffset);
            lrToolX.SetPosition(1, toolXTransform.localPosition * -lrSizeOffset + new
                Vector3(0f, 0f, -.1f) * lrSizeOffset);

            lrToolY.SetPosition(0, toolYTransform.localPosition * -lrSizeOffset);
            lrToolY.SetPosition(1, toolYTransform.localPosition * -lrSizeOffset + new
                Vector3(-.1f, 0f, 0f) * lrSizeOffset);

            lrToolZ.SetPosition(0, toolZTransform.localPosition * -lrSizeOffset);
            lrToolZ.SetPosition(1, toolZTransform.localPosition * -lrSizeOffset + new
                Vector3(0f, -.1f, 0f) * lrSizeOffset);

        }
        else if(lrToolX.enabled || lrToolY.enabled || lrToolZ.enabled)
        {
            lrToolX.enabled = false;
```

```
            lrToolY.enabled = false;
            lrToolZ.enabled = false;
            this.transform.Find("ToolX").GetComponent<MeshRenderer>().enabled = false;
            this.transform.Find("ToolY").GetComponent<MeshRenderer>().enabled = false;
            this.transform.Find("ToolZ").GetComponent<MeshRenderer>().enabled = false;

        }
    }
}
```

1.5. WorldCoordSystem.cs

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

public class WorldCoodSystem : MonoBehaviour {

    public Transform handTransform;

    private Transform worldXTransform;
    private Transform worldYTransform;
    private Transform worldZTransform;


    private LineRenderer lrWorldX;
    private LineRenderer lrWorldY;
    private LineRenderer lrWorldZ;


    public Vector3 worldX;
    public Vector3 worldY;
    public Vector3 worldZ;

    public bool lrEnabled = false;


    // Use this for initialization
    void Start () {
        handTransform = GameObject.Find("Hand2").transform;

        worldXTransform = this.transform.Find("WorldX");
        worldYTransform = this.transform.Find("WorldY");
        worldZTransform = this.transform.Find("WorldZ");


        lrWorldX = this.transform.Find("WorldX").GetComponent<LineRenderer>();
        lrWorldY = this.transform.Find("WorldY").GetComponent<LineRenderer>();
        lrWorldZ = this.transform.Find("WorldZ").GetComponent<LineRenderer>();


        this.transform.Find("WorldX").GetComponent<MeshRenderer>().enabled = false;
        this.transform.Find("WorldY").GetComponent<MeshRenderer>().enabled = false;
        this.transform.Find("WorldZ").GetComponent<MeshRenderer>().enabled = false;

        lrWorldX.positionCount = 2;
        lrWorldY.positionCount = 2;
        lrWorldZ.positionCount = 2;

        lrWorldX.startWidth = 0.01f;
        lrWorldY.startWidth = 0.01f;
        lrWorldZ.startWidth = 0.01f;

        lrWorldX.endWidth = 0.01f;
```

```
        lrWorldY.endWidth = 0.01f;
        lrWorldZ.endWidth = 0.01f;


    }

    // Update is called once per frame
    void Update () {
        this.transform.position = handTransform.position;


        worldX = worldXTransform.position - this.transform.position;
        worldY = worldYTransform.position - this.transform.position;
        worldZ = worldZTransform.position - this.transform.position;


        worldX = Vector3.Normalize(worldX);
        worldY = Vector3.Normalize(worldY);
        worldZ = Vector3.Normalize(worldZ);



        if (lrEnabled)
        {
            if (!lrWorldX.enabled || !lrWorldY.enabled || !lrWorldZ.enabled)
            {
                lrWorldX.enabled = true;
                lrWorldY.enabled = true;
                lrWorldZ.enabled = true;

                this.transform.Find("WorldX").GetComponent<MeshRenderer>().enabled =
                    true;
                this.transform.Find("WorldY").GetComponent<MeshRenderer>().enabled =
                    true;
                this.transform.Find("WorldZ").GetComponent<MeshRenderer>().enabled =
                    true;

            }
            lrWorldX.SetPosition(0, this.transform.position);
            lrWorldX.SetPosition(1, this.transform.position + new Vector3(.2f, 0f,
                0f));

            lrWorldY.SetPosition(0, this.transform.position);
            lrWorldY.SetPosition(1, this.transform.position + new Vector3(0f, 0f,
                .2f));

            lrWorldZ.SetPosition(0, this.transform.position);
            lrWorldZ.SetPosition(1, this.transform.position + new Vector3(0f, .2f,
                0f));


        }
        else if (lrWorldX.enabled || lrWorldY.enabled || lrWorldZ.enabled)
        {
            lrWorldX.enabled = false;
            lrWorldY.enabled = false;
            lrWorldZ.enabled = false;
            this.transform.Find("WorldX").GetComponent<MeshRenderer>().enabled =
                false;
            this.transform.Find("WorldY").GetComponent<MeshRenderer>().enabled =
                false;
            this.transform.Find("WorldZ").GetComponent<MeshRenderer>().enabled =
                false;

        }

    }
}
```

1.6. ControllerHints.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;
using Valve.VR.InteractionSystem;
using Valve.VR;
public class ControllerHints : MonoBehaviour {


    Player player;
    Coroutine hintCoroutine;
  // Use this for initialization
  void Start () {

        player = Player.instance;
    }

  // Update is called once per frame
  void Update () {
        Hand hand = GetComponent<Hand>();
        ControllerButtonHints.ShowTextHint(hand,
            EVRButtonId.k_EButton_SteamVR_Trigger, "Ajuste fino");
        ControllerButtonHints.ShowTextHint(hand, EVRButtonId.k_EButton_Grip,
            "Mantener pulsado: mover objetivo \n Soltar: calcular objetivo");
        ControllerButtonHints.ShowTextHint(hand,
            EVRButtonId.k_EButton_ApplicationMenu, "Guardar objetivo en path");
        ControllerButtonHints.ShowTextHint(hand,
            EVRButtonId.k_EButton_SteamVR_Touchpad, "Ejecutar path");
    }
}
```

1.7. Communications.cs

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

public class Communications : MonoBehaviour {


    //public Connector conn = new Connector();
    byte[] dataSend;
    byte[] dataReceive;
    public bool flag = false;
    public string message;

    public bool reachable = false;
    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;
// Use this for initialization
NetworkStream streamRobotStudio;
NetworkStream streamPLCSIM;

IRB120 robot;
void Start () {
    // Find robot IRB120 script
    robot = GameObject.Find("ABB_IRB120_1").GetComponent<IRB120>();
    // Create and start RobotStudio communications thread
    threadRobotStudio = new Thread(CommsRobotStudio);
    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();


}

// Update is called once per frame
void Update () {

}

void CommsRobotStudioUDP()
{
    clientRobotStudioUDP = new UdpClient(7000);
    IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, 7000);
    while (!abort)
    {
        try
        {
            if (!abort & clientRobotStudioUDP.Available > 0)
            {
                byte[] angles;
                angles = clientRobotStudioUDP.Receive(ref remoteEP);
                MoveRobot(robot, System.Text.Encoding.ASCII.GetString(angles));
            }


        }
        catch (System.IO.IOException ex)
        {
            Debug.Log("UDP IOEx");
        }

    }
    return;
    Debug.Log("Aborted network thread");
}

void CommsPLCSIM()
{
    while (!abort)
    {
        clientPLCSIM = new TcpClient("127.0.0.1", 6000);
        streamPLCSIM = clientPLCSIM.GetStream();
        streamPLCSIM.ReadTimeout = 2000;
        streamPLCSIM.WriteTimeout = 2000;

        byte[] inputsPLCSIM = System.Text.Encoding.ASCII.GetBytes("1");
        try
```

```
            {
                streamPLCSIM.Write(inputsPLCSIM, 0, inputsPLCSIM.Length);
                Debug.Log("[U5] Sent message: " +
                    System.Text.Encoding.ASCII.GetString(inputsPLCSIM));
            }
            catch (System.IO.IOException e)
            {
                Debug.Log("[IO] Error sending message to PLCSIM");

                streamPLCSIM.Close();
                //stream.Dispose();
                clientPLCSIM.Close();
            }


            try
            {
                byte[] outputsPLCSIM;
                // We have to initialize the receiving buffer
                outputsPLCSIM =
                    System.Text.Encoding.ASCII.GetBytes("XXXXXXXXXXXXXXXXXXXXXX");
                streamPLCSIM.Read(outputsPLCSIM, 0, 20);
                Debug.Log("Received Message: " +
                    System.Text.Encoding.ASCII.GetString(outputsPLCSIM));
                if (System.Text.Encoding.ASCII.GetString(outputsPLCSIM).Contains(
                    "error"))
                {

                }
                else
                {
                    Debug.Log("[PLCSIM] Received message: " +
                        System.Text.Encoding.ASCII.GetString(outputsPLCSIM));

                }

            }
            catch (System.IO.IOException e)
            {
                Debug.Log("[IO] Error receiving message from PLCSIM");
                Debug.Log(e.ToString());
                streamPLCSIM.Close();
                //stream.Dispose();
                clientPLCSIM.Close();
            }
        }
        return;
        Debug.Log("Aborted network thread");

    }

    void CommsRobotStudio()
    {
        while (!abort)
        {
            if (flag)
            {
                flag = false;
                clientRobotStudio = new TcpClient("192.168.1.135", 5000);
                streamRobotStudio = clientRobotStudio.GetStream();

                streamRobotStudio.ReadTimeout = 2000;
                streamRobotStudio.WriteTimeout = 2000;
                dataSend = System.Text.Encoding.ASCII.GetBytes(message);
                try
                {
                    streamRobotStudio.Write(dataSend, 0, dataSend.Length);
                    Debug.Log("[U5] Sent message: " +
                        System.Text.Encoding.ASCII.GetString(dataSend));
                }
```

```
                catch (System.IO.IOException e)
                {
                    Debug.Log("[IO] Error sending message to RobotStudio");

                    streamRobotStudio.Close();
                    //stream.Dispose();
                    clientRobotStudio.Close();
                }


                try
                {
                    // We have to initialize the receiving buffer
                    dataReceive =
                        System.Text.Encoding.ASCII.GetBytes("XXXXXXXXXXXXXXXXXXXXXX");
                    streamRobotStudio.Read(dataReceive, 0, 20);

                    if (System.Text.Encoding.ASCII.GetString(dataReceive).Contains(
                        "error"))
                    {
                        Debug.Log("[RS]: Error - Point not reachable");
                        reachable = false;
                        receivedResponse = true;
                    }
                    else
                    {
                        Debug.Log("[RS] Received message: " +
                            System.Text.Encoding.ASCII.GetString(dataReceive));

                        MoveRobot(robot,
                            System.Text.Encoding.ASCII.GetString(dataReceive));
                        reachable = true;
                        receivedResponse = true;
                    }



                    streamRobotStudio.Close();
                    //stream.Dispose();
                    clientRobotStudio.Close();
                }
                catch(System.IO.IOException e)
                {
                    Debug.Log("[IO] Error receiving message from RobotStudio> " +
                        e.ToString());

                    streamRobotStudio.Close();
                    //stream.Dispose();
                    clientRobotStudio.Close();
                }

            }
        }
        return;
        Debug.Log("Aborted network thread");

    }

    void MoveRobot(IRB120 robot, string angles)
    {
        int a1, a2, a3, a4, a5, a6;
        string[] result;
        angles = angles.Replace("X", string.Empty);

        result = angles.Split(';');
        a1 = int.Parse(result[0]);
        a2 = int.Parse(result[1]);
        a3 = int.Parse(result[2]);
        a4 = int.Parse(result[3]);
```

```
        a5 = int.Parse(result[4]);
        a6 = int.Parse(result[5]);
        robot.a1 = a1;
        robot.a2 = a2;
        robot.a3 = a3;
        robot.a4 = a4;
        robot.a5 = a5;
        robot.a6 = a6;
        Debug.Log("[U5] Robot moved to coordinates");

    }


    void OnApplicationQuit()
    {
        abort = true;
        //testThread.Abort(); //Not predictable

        if (clientRobotStudio != null || streamRobotStudio != null)
        {
            streamRobotStudio.Close();
            Debug.Log("Closed stream");
            clientRobotStudio.Close();
            Debug.Log("Closed Client");

        }


    }
}
```

1.8. StraightConveyorBelt.cs

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

public class StraightConveyorBelt : MonoBehaviour {

    public bool active = true;
    public float speed = 0.25f;

    public string realignAxis;

    //public float reorientationSpeed;
    public Vector3 reorientationVector;



    // Use this for initialization
    void Start()
    {

    }


    // Update is called once per frame
    void Update()
    {

    }

    void FixedUpdate()
    {
```

```
        if (active)
        {
            Rigidbody rigidbody = GetComponent<Rigidbody>();
            rigidbody.position -= transform.forward * speed * Time.deltaTime;
            rigidbody.MovePosition(rigidbody.position + transform.forward * speed *
                Time.deltaTime);
        }


    }

    void OnTriggerStay(Collider coll)
    {
        if (coll.tag == "Pallet")
        {
            Vector3 targetPosition = coll.transform.position;
            Quaternion targetOrientation = coll.transform.rotation;

            if (realignAxis == "z")
                targetPosition.z = GetComponentInParent<Transform>().position.z;
            else if (realignAxis =="x")
                targetPosition.x = GetComponentInParent<Transform>().position.x;


            coll.GetComponent<Rigidbody>().position =
                Vector3.Lerp(coll.transform.position, targetPosition,speed * 100 *
                Time.deltaTime);

            targetOrientation.SetLookRotation(reorientationVector);
            coll.GetComponent<Rigidbody>().rotation =
                Quaternion.Lerp(coll.transform.rotation, targetOrientation, 10*speed
                * Time.deltaTime);

        }
    }
}
```

1.9. CurvedConveyorBelt.cs

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

public class CurvedConveyorBelt : MonoBehaviour {


    public bool active = true;
    public float speed = 2f;

    public Vector3 axis = new Vector3(0, 1, 0);
    public Transform center;
    public float angle = 0;

    // Use this for initialization
    void Start()
    {

    }

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

```
    }

    void FixedUpdate()
    {

        if (active)
        {

            Quaternion q = Quaternion.AngleAxis(Time.deltaTime*speed, axis);
            Quaternion qinverse = Quaternion.AngleAxis(-Time.deltaTime*speed, axis);

            Rigidbody rb = this.GetComponent<Rigidbody>();

            rb.position = (center.transform.position + qinverse *
                (rb.transform.position - center.transform.position));
            rb.MovePosition(center.transform.position + q * (rb.position -
                center.transform.position));

            rb.rotation = rb.transform.rotation * qinverse;
            rb.MoveRotation(rb.rotation * q);

        }
    }
}
```

1.10. AirValve.cs

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

public class AirValve : MonoBehaviour {

    public Transform valve;
    public Transform rod;

    private Transform openTransform;

    public bool closed = true;

    private bool rotateOpenDone = false;
    private bool rotateCloseDone = false;


  // Use this for initialization
  void Start () {
        valve = this.transform.Find("Valve");
        rod = this.transform.Find("Rod");
        closed = true;
  }

    // Update is called once per frame
    void Update() {


        if(!closed & !rotateCloseDone)
        {
            rotateOpenDone = false;
            valve.RotateAround(rod.position, rod.up, 90);
            rotateCloseDone = true;
        }
        else if (closed & !rotateOpenDone)
        {
```

```
            rotateCloseDone = false;
            valve.RotateAround(rod.position, rod.up, -90);
            rotateOpenDone = true;
        }

  }
}
```

1.11. Tutorial.cs

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

public class Tutorial : MonoBehaviour {


    private int state;

    private bool valvePrev;
    private bool valveCurr;

    private bool toolPrev;
    private bool toolCurr;

    private AirValve valve;
    private RobotTip robotTip;

  // Use this for initialization
  void Start () {
        state = 0;
        valve = GameObject.Find("AirValve").GetComponent<AirValve>();
        robotTip = GameObject.Find("RobotTip").GetComponent<RobotTip>();

    }

void Update () {
        //Edge detection
        //Valve
        valvePrev = valveCurr;
        valveCurr = valve.closed;
        //Tool
        toolPrev = toolCurr;
        toolCurr = robotTip.hasTool;


        //Initial state
        if (state == 0)
        {
            //Actions of this state
            tutorialText = "Locate the compressed air valve and close";

            //Transitions
            if (valveCurr & !valvePrev)
                state = 1;  //Remove tool state
            else if (toolCurr != toolPrev)
                state = 99; //Game Over
        }
        //Remove tool state
        else if (state == 1)
        {
            //Actions of this state
            tutorialText = "Remove the existing robot tool";
```

```
            //Transitions
            if (!toolCurr & toolPrev)
                state = 2; //Install tool state
            else if (valveCurr != valvePrev)
                state = 99; //Game over
        }
        //Install tool state
        else if (state == 2)
        {
            //Actions of this state
            tutorialText = "Install the new robot tool";

            //Transitions
            if (toolCurr & !toolPrev)
                state = 2;
            else if (valveCurr != valvePrev)
                state = 99; //Game over

        }
        //Tutorial completed game
        else if (state == 90)
        {
            //Actions of this state
            tutorialText = "Congratulations! You just became employee of the month!";
        }
        //Game over state
        else if(state == 99)
        {
            //Actions of this state
            tutorialText = "Maybe next time you'll get it right :(";
        }
}
```

2. Rapid

2.1. Tarea principal

```
MODULE MainModule

    CONST string ip := "192.168.1.135";
    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 5 via
        TCP/IP socket
    PERS robtarget robotTarget := [[205,-185,368],
        [0.181652,-0.956802,0.219421,0.0581909], [-1,-1,-1,0],
        [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    PERS num anglex := -160.845;
    PERS num angley := 11.0153;
    PERS num anglez := -23.9682;



    ! Variables for path creation
    ! Path robtargets
    PERS robtarget RT1;
    PERS robtarget RT2;
    PERS robtarget RT3;
    PERS robtarget RT4;
    PERS robtarget RT5;
    ! Counter for number of points in path (max 5)
    PERS intnum i := 0;
    ! 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 := 0;
    ! Resulting joint data from the calculation, initialized at random variable, will
        be updated
    PERS jointtarget jointDataResult :=
        [[-45.6272,2.69252,34.1251,14.5937,71.0694,-206.731],
        [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;

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

        !VAR robtarget p1;
        !p1 := cRobt(\Tool:=tool0, \WObj:=wobj0);

        !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:=70;

!
!  "message" - Calculate joint angles for robtarget
! "Xmessage" - Add target to path
!"XXmessage" - Execute path

! Calculate joint angles
IF StrPart(message,1,1) <> "X" THEN
    !Convert message string to robtaget object
    robotTarget := StringToTarget(message);
    !Reset calculation result variable
    calculationResult := 0;

    ! Calculate the joint angles given the robot target, and store errors in
        errorNumber
    jointDataResult := CalcJointT(robotTarget, tool0, \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

!! Add previous point to target
ELSEIF StrPart(message,2,1) <> "X" THEN
    TEST i
        CASE 0:
            RT1 := robotTarget;
            i := i + 1; !Increase counter
        CASE 1:
            RT2 := robotTarget;
            i := i + 1; !Increase counter
        CASE 2:
            RT3 := robotTarget;
            i := i + 1; !Increase counter
        CASE 3:
            RT4 := robotTarget;
            i := i + 1; !Increase counter
        CASE 4:
            RT5 := robotTarget;
            i := i + 1; !Increase counter
        DEFAULT:
            i := i;
    ENDTEST
    SocketSend client,\str:="error";
    !Close client and server sockets
    SocketClose client;
    SocketClose server;

! Execute path
ELSEIF StrPart(message,2,1) = "X" THEN
    !Raise executingPath flag
    executingPath := TRUE;
    !Ignore current robot configuration during linear movement
    ConfL \Off;
    TEST i
        CASE 1:
            MoveL RT1, v50, z5, tool0;

        CASE 2:
            MoveL RT1, v50, z5, tool0;
            MoveL RT2, v50, z5, tool0;
```

```
            CASE 3:
                MoveL RT1, v50, z5, tool0;
                MoveL RT2, v50, z5, tool0;
                MoveL RT3, v50, z5, tool0;

            CASE 4:
                MoveL RT1, v50, z5, tool0;
                MoveL RT2, v50, z5, tool0;
                MoveL RT3, v50, z5, tool0;
                MoveL RT4, v50, z5, tool0;

            CASE 5:
                MoveL RT1, v50, z5, tool0;
                MoveL RT2, v50, z5, tool0;
                MoveL RT3, v50, z5, tool0;
                MoveL RT4, v50, z5, tool0;
                MoveL RT5, v50, z5, tool0;
        ENDTEST
        !Reset executingPath flag
        executingPath := FALSE;
        WaitTime 5;
        SocketSend client,\str:="error";
        SocketClose client;
        SocketClose server;
    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 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
```

2.2. Tarea secundaria

```
MODULE SecondaryModule
    PERS jointtarget jointangles;
    !Variable from the main task
    PERS bool executingPath;

    !Declaration variables
    VAR socketdev udpSocket;
    VAR socketdev clientUDP;
    VAR string message;
    VAR rawbytes data;


    PROC main()

        SocketCreate udpSocket \UDP;

        WHILE TRUE DO
            IF executingPath THEN
                jointangles := CJointT();
                SocketSendTo udpSocket,"192.168.1.143",7000 \Str :=
                    jointTargetToString(jointangles);
            ENDIF

            WaitTime 0.05;
        ENDWHILE
    ENDPROC



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

ENDMODULE
```

3. Interfaz PLCSIM - Unity

3.1. AutoVR.cs

```
public class autoVR
{
    // Socket communication thread
    private Thread threadServer;
    private Thread threadSend;
    private Thread threadReceive;

    static bool abortComms;

    static void Main(string[] args)
    {
        // Instance
        autoVR autoVR = new autoVR();
        // autoVR logo
        autoVR.TitleWindow();

        // Initialize abort flag
        abortComms = false;

        // Start server thread
        autoVR.threadServer = new Thread(autoVR.Server);
        autoVR.threadServer.Start();

        //// Create the PLCInstance: virtualController
        PLCInstance virtualController = null;
        // Name of the instance
        string instanceName = "autoVR";

        try
        {
            // Register new PLCInstance
            virtualController = new PLCInstance(instanceName, false);
            autoVR.LogAuto("Instance Registered");
            // Simulate S7-1500 PLC
            virtualController.instance.CPUType = ECPUType.CPU1500_Unspecified;
        }
        catch (SimulationRuntimeException simRuntimeEx)
        {
            autoVR.LogAuto("The instance already exists");
            // Link with existing PLCInstance
            virtualController = new PLCInstance(instanceName, true);
            autoVR.LogAuto("Instance assigned");
        }

        try
        {
            // Power on PLCInstance
            virtualController.PowerOnPLCInstance();
        }
        catch (SimulationRuntimeException simRuntimeEx2)
        {
            // Instance was already on
            autoVR.LogAuto("Instance was already on");
        }

        try
        {
            // Reset PLCInstance if it was running
            virtualController.StopPLCInstance();
        }
        catch (SimulationRuntimeException sre)
```

```
            {
                // Instance was not running
            }
            // Run PLCInstance
            virtualController.RunPLCInstance();
            autoVR.LogAuto("Instance running");

            // Update IO tags: IO.0, IO.1...
            virtualController.instance.UpdateTagList(ETagListDetails.IO);

        }

        // Server thread
        public void Server()
        {
            // Server
            TcpListener serverSocket = new TcpListener(6000);
            serverSocket.Start();
            LogAuto("Server Started");



            while (!abortComms)
            {
                try
                {

                    // Client
                    TcpClient clientSocket = default(TcpClient);
                    clientSocket = serverSocket.AcceptTcpClient();
                    LogAuto("Client accepted");

                    NetworkStream networkStream = clientSocket.GetStream();
                    byte[] bytesFromUnity = new byte[4];
                    networkStream.Read(bytesFromUnity, 0, 4);
                    string dataFromUnity =
                        System.Text.Encoding.ASCII.GetString(bytesFromUnity);


                    // Read data
                    GlobalVars.receiveString = dataFromUnity;
                    LogReceiver(GlobalVars.receiveString);
                    // Coordination flag
                    GlobalVars.receiveFlag = true;


                    Byte[] bytesToUnity =
                        Encoding.ASCII.GetBytes(GlobalVars.sendString);
                    networkStream.Write(bytesToUnity, 0, bytesToUnity.Length);
                    LogSender("Sent message to Unity: " + GlobalVars.sendString);
                    networkStream.Flush();

                }
                catch (Exception ex)
                {
                    LogAuto(ex.ToString());
                }
                //Has to be lower than timeout in Unity Script
                Thread.Sleep(500);
            }

            //clientSocket.Close();
            serverSocket.Stop();
            return;
        }
    }
```

3.2. PLCInstance

```
void instance_OnEndOfCycle(IInstance in_Sender, ERuntimeErrorCode
    in_ErrorCode,
    DateTime in_DateTime, long in_CycleTime_ns, uint in_CycleCount)
{
    //Console.WriteLine("End of Cycle");
    try
    {
        // Sending data to Unity (outputs)
        string q00 = autoVR.ConvertToString(instance.ReadBool("DO_Cintas"));


        GlobalVars.sendString = q00;
        //Console.WriteLine(GlobalVars.sendString);

        // Turn on flag for sender thread
        GlobalVars.sendFlag = true;

        // Read virtual plant data
        if (GlobalVars.receiveFlag)
        {

            bool i00 = true;
            //// Read inputs
            bool i00 = autoVR.ConvertToBool(GlobalVars.receiveString[0]);
            bool i40 = autoVR.ConvertToBool(GlobalVars.receiveString[1]);
            bool i41 = autoVR.ConvertToBool(GlobalVars.receiveString[2]);
            bool i42 = autoVR.ConvertToBool(GlobalVars.receiveString[3]);
            bool i43 = autoVR.ConvertToBool(GlobalVars.receiveString[4]);
            bool i44 = autoVR.ConvertToBool(GlobalVars.receiveString[5]);
            bool i45 = autoVR.ConvertToBool(GlobalVars.receiveString[6]);
            bool i46 = autoVR.ConvertToBool(GlobalVars.receiveString[7]);
            bool i47 = autoVR.ConvertToBool(GlobalVars.receiveString[8]);



            // Write inputs to I0.0, I0.1...
            instance.WriteBool("DI_Go_Belt", i00);
            instance.WriteBool("piEmergencia", i40);
            instance.WriteBool("piRearme", i41);
            instance.WriteBool("piAuto", i42);
            instance.WriteBool("piManual", i43);
            instance.WriteBool("piMarcha", i44);
            instance.WriteBool("piParo", i45);
            instance.WriteBool("piCintas", i47);


            //Console.WriteLine(i1.ToString()+i2.ToString()+i3.ToString());

            // Reset flag
            GlobalVars.receiveFlag = false;
        }

    }
    catch (Exception ex)
    {

    }
}
```