GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS INDUSTRIALES DE GRANDES DIMENSIONES

Autor: Mario Serrano Rodríguez

Director: José Antonio Rodríguez Mondéjar

Madrid

Agosto de 2019

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

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

El autor D. Mario Serrano Rodríguez

DECLARA ser el titular de los derechos de propiedad intelectual de la obra: APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS INDUSTRIALES DE GRANDES DIMENSIONES, 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 27 de agosto de 2019

ACEPTA

Fdo Mario Serrano Rodríguez

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

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

Aplicación de la realidad virtual a la programación de robots industriales de grandes dimensiones

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2018/2019 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Mario Serrano Rodríguez Fecha: 26/ 08/ 2019

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: José Antonio Rodríguez Mondéjar Fecha: 26/ 08/ 2019

APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS INDUSTRIALES DE GRANDES DIMENSIONES

Autor: Serrano Rodríguez, Mario.

Director: Rodríguez Mondéjar, José Antonio.

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

Introducción.

Este proyecto consiste en facilitar la programación de un brazo robótico industrial de grandes dimensiones, el IRB 8700 diseñado por ABB. Para ello, se desarrollará un programa haciendo uso de la realidad virtual para que el usuario interaccione con el brazo robótico, siendo capaz de visualizar y mover el robot de manera intuitiva. También se adaptará el programa para que los problemas que conllevan las grandes dimensiones del brazo robótico sean solventados.

Estado de la técnica.

Actualmente, la realidad virtual no sólo es usada en el mundo del entretenimiento, sino que también tiene diversos usos en el mundo profesional.

Entre ellos destaca el uso de realidad virtual en medicina para la enseñanza de los estudiantes, sobre todo de anatomía. También se está experimentando con el entrenamiento quirúrgico en entornos de realidad virtual.

Figura 1: Estudiante aprendiendo anatomía usando realidad virtual

Otro ámbito profesional que se ayuda de esta herramienta es del entrenamiento de prevención de riesgos laborales en entornos más peligrosos, ya que la realidad virtual ayuda a recrear las situaciones de mayor peligro sin suponer un riesgo para el usuario.

En cuanto a la robótica de grandes dimensiones, ésta tiene un papel especialmente remarcable en la industria automovilística. Esto es debido a que esta industria es destinada a un público masivo, por lo tanto, para competir con los precios del mercado,

los fabricantes necesitan tener un alto grado de automatización en sus cadenas de producción. Para ello, los brazos robóticos son comúnmente usados, y debido al gran peso de algunos de los componentes del coche, estos robots necesitan ser de grandes dimensiones.

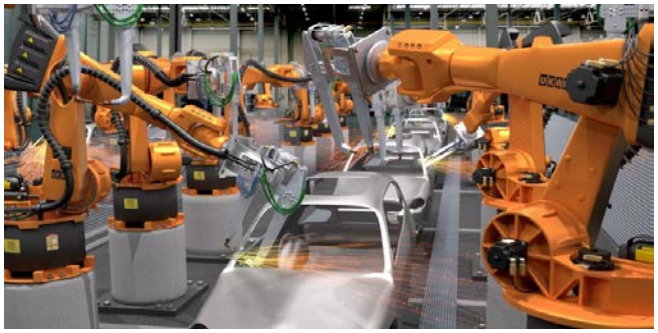

Figura 2: Cadena de producción automatizada

En cuanto a la realidad virtual aplicada a la robótica, podemos encontrar distintos proyectos y programas que ayudan en el manejo y programación de un brazo robótico a través de un entorno de realidad virtual.

En este caso, este proyecto se centrará en el desarrollado por Carlos Álvarez Vereterra. Este programa consiste en un entorno de realidad virtual en el que el usuario puede mover el objetivo de un robot (este robot es el IRB 120, de un tamaño comedido) con nuestro mando, y en cuando definimos un nuevo objetivo, el robot se desplaza hasta alcanzarlo. También, el usuario puede guardar varios puntos en un camino para que el robot lo ejecute cuando se desee. Esta aplicación tiene otras funcionalidades, pero las que se han mencionado son en las que este proyecto se centrará.

Objetivo del proyecto

El objetivo de este proyecto consiste en permitir la programación y simulación de un brazo robótico de grandes dimensiones, concretamente en IRB 8700, diseñado y producido por ABB. Para ello, el proyecto se basará en el programa ya desarrollado por Carlos Álvarez Vereterra, en el que se implantará el modelo del IRB 8700 para que desempeñe las mismas funciones que el IRB 120 ya implementado. También se trabajará en una herramienta tipo pinza para darle un mayor realismo al modelo virtual, y se hará que la misma sea capaz de coger y depositar objetos. Por último, se adaptará el programa para que el manejo de un robot de grandes dimensiones sea más fácil implementando un zoom que el usuario puede manejar con los botones del mando del set de realidad virtual.

Metodología

El zoom fue el primer elemento en ser implementado. La mayor dificultad fue que Unity no permite al usuario cambiar el campo de visión, y este método es el más popular para hacer zoom en videojuegos. Como método alternativo, se desarrolló un script para cambiar la escala del jugador. Aumentando la escala, los movimientos del usuario tienen mayor recorrido en la escena, abarcan un área mayor y, por lo tanto, se puede alcanzar rápidamente cualquier punto dentro del rango de alcance del robot. Disminuyendo la escala, el usuario puede moverse con mayor precisión y centrarse en un área específica.

Después, se importó el modelo del IRB 8700 a los entornos de Unity y RobotStudio. Importarlo a RobotStudio fue muy sencillo porque este programa ya tiene una base de datos con varios modelos de robots, incluyendo el IRB 8700.

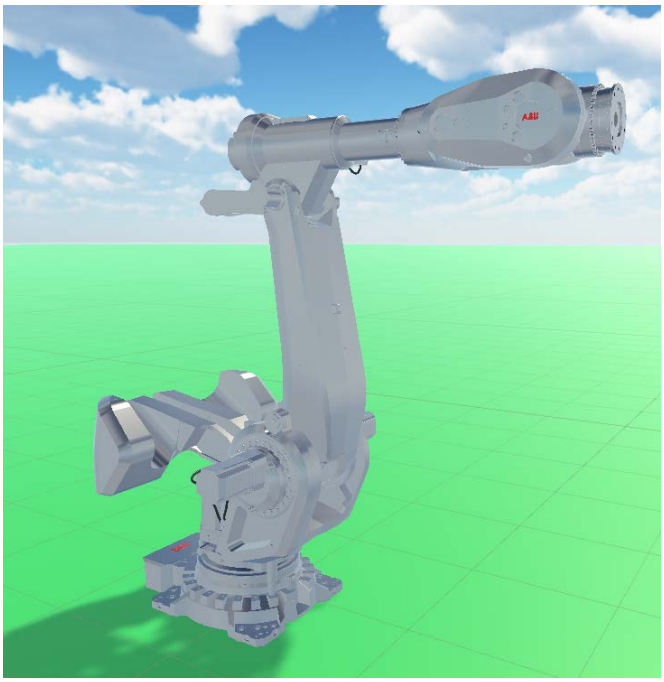

Figura 3: IRB 8700 importado completamente a Unity

Finalmente, se pasó a implementar la pinza en el programa. Para ello, se descargó el modelo 3D de la página web de un fabricante de sistemas de agarre (Schunk), y después se crearon las garras manualmente en Unity. Después se implementó el comportamiento de la pinza (primero el control de apertura y cierre y luego los algoritmos necesarios para hacer que la pinza sea capaz de agarrar y soltar objetos).

Resultados y conclusiones

El resultado final es un modelo de un robot completamente funcional, que se mueve adecuadamente cuando el usuario se lo ordena, y que es capaz de agarrar objetos con su herramienta de tipo pinza.

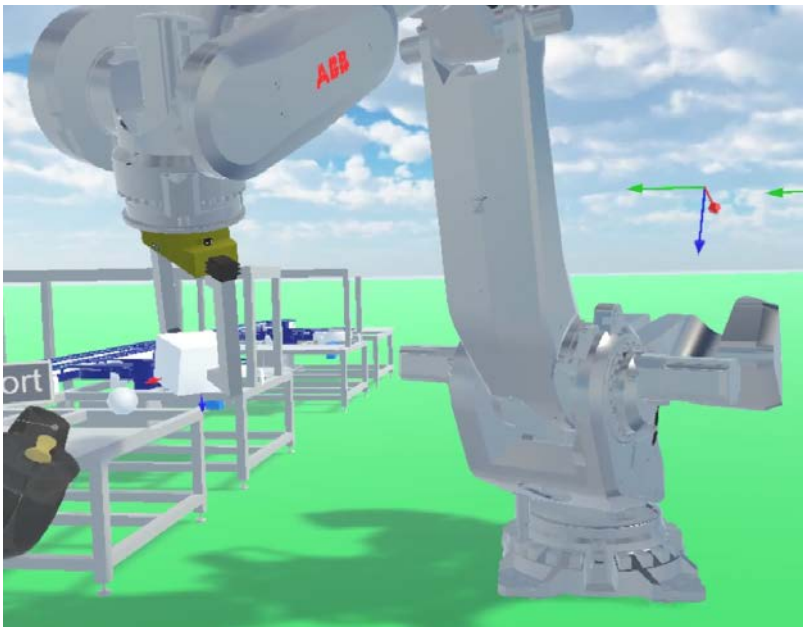

Figura 4: IRB 8700 agarrando un cubo con su pinza

Gracias a esto, el usuario puede interactuar con otro tipo de robot distinto del IRB 120 (y con unas dimensiones muy distintas), haciendo la experiencia de la interacción con la robótica a través de la realidad virtual mucho más rica y variada.

Además, la pinza ayuda al modelo del robot a ser más realista, y el zoom implementado hace la experiencia de usuario de programar el robot más rápida y sencilla.

USE OF VIRTUAL REALITY IN PROGRAMMING LARGE SIZED ROBOTS

Author: Serrano Rodríguez, Mario.

Director: Rodríguez Mondéjar, José Antonio.

Collaborating entity: ICAI – Universidad Pontificia Comillas.

PROJECT SUMMARY

Introduction

This project consist of making the programing of a large sized robot (the IRB 8700, designed by ABB) easier. For that, using virtual reality environments, a program will be developed so that the user can interact with the robotic arm, being able to visualize and move the robot intuitively. Also, the program will be adapted so that the problems involved with the large size of the robot can be solved.

State of the art

Nowadays, virtual reality is not only used for entertainment, but it has also many uses among the professional scope.

Among them, we can highlight the use of virtual reality in medicine, specifically for teaching the students in different aspects like anatomy. Furthermore, experimentally, some virtual reality-based surgical training programs are being carried out.

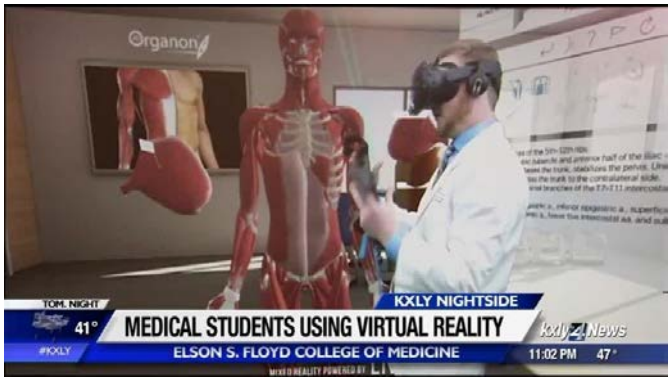

Figure 1: Student learning anatomy thanks to VR

Another professional scope that uses this tool is the training of staff in risks prevention in some especially dangerous environments, as virtual reality helps simulating situations of greater risk without making the user taking any risk.

About large sized robots, they are mostly used in the car industry. This is because this industry is consumed by a massive amount of people, so, in other to compete against the prices of the competence, manufacturers need to have their production lines highly

automatized. For that, robotic arms are commonly used, and due to the high weight of some components, these robots need to have a large size.

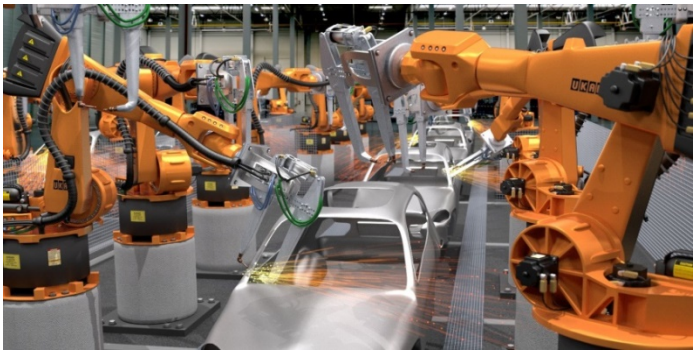

Figure 2: Automatized car production line

About virtual reality used in robotics, we can find several project and programs that help programming and move a robotic arm with the use of a virtual reality interface.

In this case, this project will be focused in the one developed by Carlos Álvarez Vereterra. This program consists of a virtual reality environment developed in Unity in which the user is able to move the target of a robot (this robot is the IRB 120, which is small sized) with our VR controller, and as soon as that is made, the robot then moves to reach that target. Also, the user can save several points into a path and then make the robot execute that path. For calculating the robot's cinematics, another program called RobotStudio is used. This program communicates continuously with the VR environment in order to make the robot in Unity able to move.

Other functionalities can be found in this project, but the ones mentioned are the ones this project is focused in.

Project objective

This project objective consists of making the programming and simulation of a large sized robotic arm possible. For that, this project will be based in the program already developed by Carlos Álvarez Vereterra, in which the IRB 8700 robotic arm will be introduced so that it can perform the same actions than the IRB 120 can already perform as it has previously been implemented by Carlos Álvarez Vereterra. Also, a gripper tool will be added to this robot so it is more realistic, and it will be programmed so that it can grab and release objects in the virtual reality environment. Lastly, the program will be adapted so that working with a large sized robot is easier and handier. For that, a zoom that the user can change by pressing different buttons of the VR controller will be developed.

Methodology

The zoom was the first feature to be implemented. The main difficulty was that Unity doesn't allow the user to change the field of view, which is the most popular method for zooming in videogames. As an alternative method, a script for changing the player's scale was developed. By increasing the user's scale, the user is able to move more widely and quickly reach any point reachable by the large sized robot. By decreasing the user's scale, the user is able to move more precise and focus in a specific area.

After, the model of the IRB 8700 was imported to the Unity's and RobotStudio environment. Importing it to RobotStudio is an easy task because this program already has a database with many robot models, including the IRB 8700.

About importing it to Unity, the 3D files downloaded from ABB's webpage needed to be converted to a compatible format. After that, the robot needed to have a behavior assigned. For that, a similar script to the one that gives behavior to the IRB 120 was adapted for the IRB 8700.

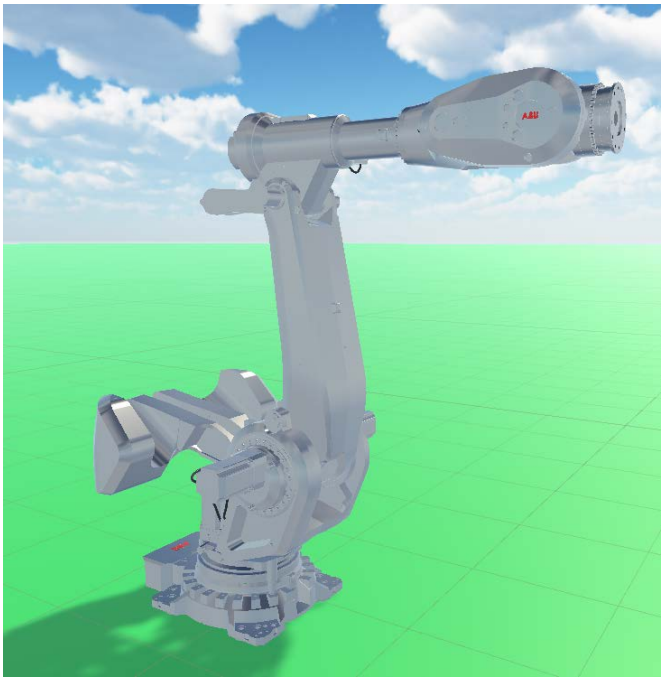

Figure 3: IRB 8700 fully imported in Unity

Finally, the gripper was implemented in the program. For that, a 3D model was downloaded from a gripping systems manufacturer (Schunk) and then the fingers were created manually in Unity. Then, the behavior of the gripper was implemented. Firstly, the control for opening and closing the fingers was implemented, and then, the algorithms necessary to make the gripper able to grab objects were developed.

Results and conclusion

The final result is a fully functional robot model that moves properly when the users interacts with it and is able to grab objects with its gripper.

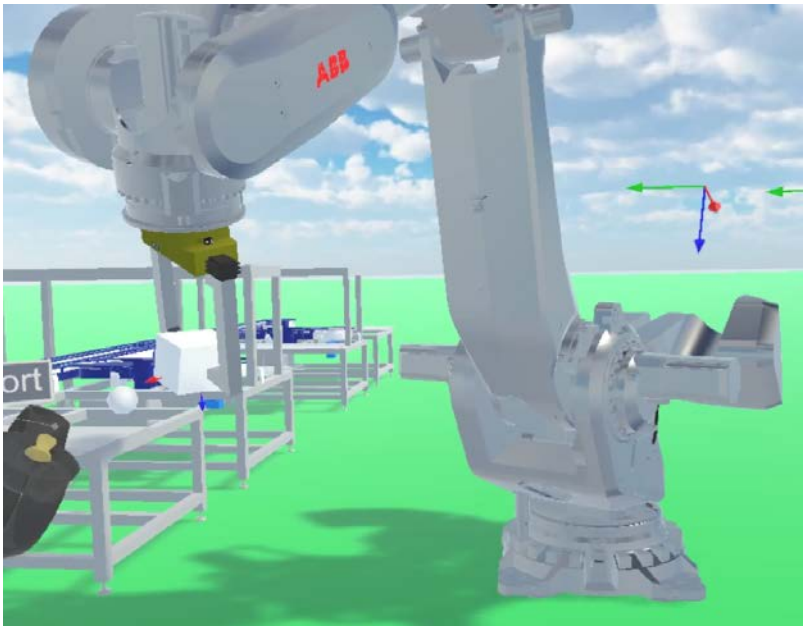

Figure 4: IRB 8700 grabing a cube with its gripper

Thanks to this, the user can interact with another type of robot than the IRB 120 that has a very different size, making the experience of VR interaction with robotics richer.

Also, the gripper helps the robot model to be more realistic, and the zoom implemented makes the user experience of programming the robot quicker and easier.

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE LA REALIDAD VIRTUAL A LA PROGRAMACIÓN DE ROBOTS INDUSTRIALES DE GRANDES DIMENSIONES

Autor: Mario Serrano Rodríguez

Director: José Antonio Rodríguez Mondéjar

Madrid

Agosto de 2019

Índice general

Memoria 4
1. Introducción 5
2. Estado del arte 6
2.1. Usos actuales de la realidad virtual 6
2.2. La robótica de grandes dimensiones 8
2.3. Realidad virtual aplicada a la robótica 10
3. Objetivo del proyecto 12
4. Puesta a punto para el proyecto 13
4.1. Aprendizaje de conceptos de robótica y RobotStudio 13
4.2. Aprendizaje de cómo programar en Unity 15
4.3. Aprendizaje del funcionamiento del programa ya desarrollado 16
5. Implementación de un zoom 19
5.1. Dificultades para implementar un zoom en realidad virtual 19
5.2. Alternativas e implementación 20
6. Modelado del robot ABB IRB 8700 22
6.1. Modelo en RobotStudio 22
6.2. Importación del modelo 3D en Unity 24
6.3. Comportamiento del robot en Unity 25
7. Modelado de una herramienta tipo “pinza” 28
7.1. Importación del modelo 3D en Unity 28
7.2. Comportamiento de la pinza en Unity 29
7.3. Modelado de la herramienta en RobotStudio 33
8. Resultados y conclusiones 35
Bibliografía 36
Presupuesto 38
1. Recursos 39
2. Precios unitarios 39
3. Presupuesto general 39
Código fuente 40
1. CameraZoom.cs 41
2. IRB8700.cs 43
3. Gripper.cs 45
4. ViveController.cs 47

Índice de figuras

Figura 1: Estudiante aprendiendo anatomía gracias a la VR 6
Figura 2: Estudiante usando un simulador quirúrgico de VR 7
Figura 3: Predicción de la producción de sets de realidad virtual 7
Figura 4: Cadena de producción de automóviles con robots 8
Figura 5: Comparativa de tamaño entre una persona y el IRB 8700 9
Figura 6: IRB 8700 levantando y transportando la carrocería de un coche 9
Figura 7: Banco de trabajo de soldadura en la VR Robotics Simulator 10
Figura 8: IRB 120 y minifábrica. Captura del entorno virtual del programa de Carlos Álvarez Vereterra 11
Figura 9: Fotografía de un FlexPendant, idéntico al utilizado durante la práctica 13
Figura 10: Captura del FlexPendant con las instrucciones 14
Figura 11: Entorno de programación de RobotStudio, en este caso listo para el movimiento lineal 15
Figura 12: Captura de pantalla de uno de los vídeos del curso en Udemy 16
Figura 13: Administrador de paquetes de Unity 17
Figura 14: Error en Unity al intentar cambiar el campo de visión 19
Figura 15: Creación de un nuevo proyecto en RobotStudio 22
Figura 16: Estación de trabajo en RobotStudio con el IRB 8700 23
Figura 17: Modelo del IRB 8700 en el entorno virtual de Unity 24
Figura 18: IRB 8700 y su jerarquía 25
Figura 19: Pinza de Schunk sin garras, a partir de la cual se genera el archivo 3D 28
Figura 20: Pinza completa integrada en el IRB 8700 29
Figura 21: IRB 8700 sujetando un objeto (cubo blanco) a través de las pinzas 32
Figura 22: Herramienta posicionada en el IRB 8700 en RobotStudio 33

Índice de códigos

Código 1: *Communications.cs* – Fragmento que envía un punto fijo erróneo 18
Código 2: *cameraZoom.cs* – Implementación del zoom y corrección del desplazamiento de la cámara 21
Código 3: *IRB8700.cs* - Cambio en la rotación de los ejes a partir de las variables a1-a6 26
Código 4: *Communications.cs* - Función *MoveRobot* 27
Código 5: IRB8700.cs - Fragmento del código que varía la posición de ambas garras a partir de la variable *grip* 29
Código 6: *ViveController.cs* - Cambio de la variable *grip* del robot según se mueva horizontalmente el joystick 30
Código 7: *gripper.cs* – Funciones *OnCollisionStay* y *OnCollisionExit* 31
Código 8: *ViveController.cs* – Control del cierre de las garras modificado 32

Memoria

1. Introducción

Este proyecto consiste en facilitar la programación de un brazo robótico industrial de grandes dimensiones, el IRB 8700 diseñado por ABB. Para ello, se desarrollará un programa haciendo uso de la realidad virtual para que el usuario interaccione con el brazo robótico, siendo capaz de visualizar y mover el robot de manera intuitiva. También se adaptará el programa para que los problemas que conllevan las grandes dimensiones del brazo robótico sean solventados.

2. Estado del arte

2.1. Usos actuales de la realidad virtual

Actualmente, el uso de la realidad virtual está muy lejos de reducirse únicamente al entretenimiento y videojuegos. A pesar de lo que la mayoría de los usuarios piensan, la realidad virtual tiene un papel mucho más amplio en nuestra sociedad de hoy en día, y, sobre todo, un gran potencial para ser de mayor utilidad en muchas de las áreas profesionales de nuestro entorno.

A día de hoy, los usos de la realidad virtual en el ámbito profesional son muy diversos. En primer lugar, me gustaría destacar su utilidad en el ámbito de la medicina, ya que esta herramienta le es muy útil sobre todo a los estudiantes para poder aprender anatomía de forma mucho más gráfica que con un libro y mucho más rápida y menos engorrosa que una práctica con un cuerpo real [1]. Así, gracias esto, el estudiante ya no necesita de la supervisión de un profesional y puede interactuar con los órganos y tejidos sin ninguna preocupación.

Figura 1: Estudiante aprendiendo anatomía gracias a la VR

Además del estudio de la anatomía, también se ha comenzado a experimentar con el entrenamiento quirúrgico en entornos de realidad virtual. Así, el médico puede formarse para realizar operaciones en un futuro sin la necesidad de un cuerpo físico. Las ventajas que ofrece la VR son muy similares a las anteriores mencionadas.

Figura 2: Estudiante usando un simulador quirúrgico de VR

Otro uso en el ámbito profesional de la VR es el de la enseñanza en colegios. La realidad virtual resulta una herramienta muy atractiva para los usuarios de corta edad, y ésta puede servir para mejorar la atención de los niños a la hora de enseñar cualquier concepto de distintas materias, además de proporcionar un espacio inmersivo y entretenido con muchas posibilidades para la interacción y donde las distracciones tienden a reducirse debido a esto.

Otras aplicaciones destacables son el entrenamiento de bomberos [2] y la formación en prevención de riesgos laborales para ciertos empleos (como, por ejemplo, la construcción), ya que se pueden simular situaciones adversas sin poner en riesgo al trabajador para poder así poner en prácticas la distintas medidas de seguridad enseñadas durante los periodos de formación [3].

Cuando la realidad virtual empezó a ser desarrollada por fabricantes, la primera sensación podría ser que su uso iba a limitarse al entretenimiento. Como puede verse, esta visión ha cambiado mucho con el tiempo y el trabajo de distintas aplicaciones por parte de diversos desarrolladores.

No sólo han aumentado los usos profesionales de la VR, sino que también las cifras de popularidad de distintos sets de realidad virtual van aumentando año tras año [4].

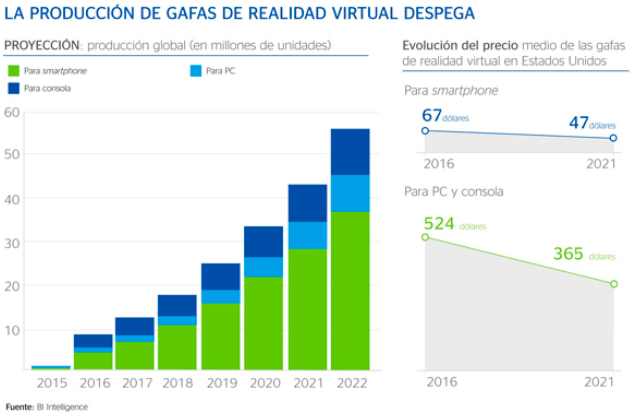

Figura 3: Predicción de la producción de sets de realidad virtual

Este aumento en las cifras nos hace pensar en un futuro donde las posibilidades que nos brinda esta herramienta se multipliquen gracias a que habrá más gente desarrollando distintas aplicaciones y programas, a la vez que aumente la demanda de soluciones a problemas a través de la realidad virtual.

2.2. La robótica de grandes dimensiones

Una vez la situación actual de uno de los dos principales componentes de mi proyecto, voy a pasar a analizar la del segundo: la robótica de grandes dimensiones.

Actualmente, los brazos robóticos tienen una amplia gama de usos, entre los cuales destaca considerablemente las aplicaciones en procesos industriales para automatizar todo tipo de tareas (soldadura, pintura, fresado, desplazamiento de objetos… etc.). Gracias a esto, es posible automatizar muchos procesos de producción industriales.

Aparte de tener un mayor alcance y, por lo tanto, una mayor área de trabajo, la principal utilidad de los brazos robóticos de grandes dimensiones es que son capaces de levantar mayor peso que los de pequeñas dimensiones.

La industria que más se ayuda de este tipo de robots es la automovilística. Hoy día, el coche se ha convertido en un bien de consumo masivo, y la mayoría de los fabricantes tienen cadenas de producción automatizadas para poder competir con los precios de la competencia. Los brazos robóticos de grandes dimensiones tienen un papel crucial en esto, ya que muchos de los componentes de los automóviles son grandes, pesados y de una sola pieza que solamente pueden ser levantados por este tipo de brazos robóticos [5].

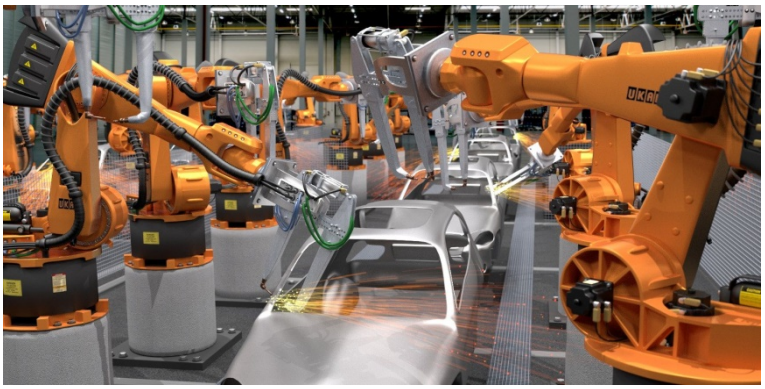

Figura 4: Cadena de producción de automóviles con robots

De entre toda la oferta de robots de grandes dimensiones, destaca el IRB 8700, fabricado por ABB. A partir de ahora, hablaremos únicamente de la versión IRB 8700-

800/3.50 de este modelo, ya que es la que se va a utilizar a lo largo de todo el proyecto.

Actualmente, es uno de los robots con mayor capacidad de carga, siendo capaz de levantar 1000 kg con la muñeca hacia abajo y 800 kg en cualquier posición [6], y mide aproximadamente 3 metros de alto.

Figura 5: Comparativa de tamaño entre una persona y el IRB 8700

Las principales ventajas de este modelo es que puede operar a una velocidad un 25% más rápida que otros modelos con una carga máxima y dimensiones similares. Además, cuenta con una alta fiabilidad y un diseño sencillo. Su radio de acción (que podría denominarse como su alcance) es de 3.5 metros [7]. No es tan amplio como otros modelos de grandes dimensiones, ya que para tener una carga máxima tan elevada, es necesario contener el radio de acción. La aplicación más común para este modelo dentro de la industria automovilística es, como puede intuirse, la de levantar y desplazar los componentes más pesados del automóvil [8].

Figura 6: IRB 8700 levantando y transportando la carrocería de un coche

Uno de los principales problemas de este modelo y su alta capacidad de carga, es que el entrenamiento del personal de manipulación del robot puede resultar peligroso. Además, como cualquier brazo robótico, su programación y manejo a través del FlexPendant puede resultar engorrosa y poco intuitiva en las primeras tomas de contacto.

Estos problemas son principalmente los que queremos solventar a la hora de desarrollar un simulador a través de realidad virtual enfocado en el entrenamiento de operarios, ya que no estaremos manejando un robot real, por lo que el peligro disminuye considerablemente, y será todo más intuitivo, visual e interactivo.

2.3. Realidad virtual aplicada a la robótica

Como he dicho anteriormente, los simuladores de manejo de brazos robóticos hacen el aprendizaje del manejo de estos más intuitivo y seguro. Estas ventajas son las que han llevado a diversos desarrolladores a hacer diversas aplicaciones, sobre todo en los últimos años.

De entre las que podemos encontrar en internet, la que más destaca es *VR Robotics Simulator,* desarrollada por MindRend Technologies [9]. En esta aplicación podemos construir un banco de trabajo con un brazo robótico y simular tanto el movimiento del robot como distintos procesos que se encargue de automatizar (soldadura con arco, levantamiento y desplazamiento de objetos…). Esta aplicación se encuentra a la venta en *Steam* y es compatible con *Oculus Rift* y *HTC Vive.*

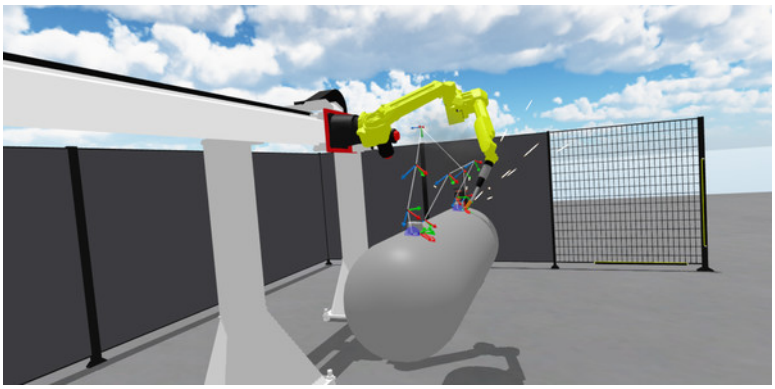

Figura 7: Banco de trabajo de soldadura en la VR Robotics Simulator

También ha sido desarrollada otra aplicación distinta por investigadores del MIT [10]. Esta aplicación tiene un objetivo y una manera de interactuar con el robot distintos que la anterior mencionada.

El objetivo principal es el de manejar un brazo robótico real a distancia. Mientras que el *VR Robotics Simulator* se reduce a ser un simulador, como su propio nombre indica, con esta aplicación se maneja un brazo físico en tiempo real. Además, el manejo de éste se hace como si los brazos del usuario fueran los propios brazos del robot, por lo que se tiene un control más "real" del robot, aunque este método puede resultar algo menos preciso.

Por último, cabe destacar el programa en Unity y RobotStudio desarrollado por Carlos Álvarez Vereterra [11]. Este programa permite simular el comportamiento de un brazo robótico IRB 120. Además, cuenta con un programa de operarios interactivo integrado en la minifábrica ICAI. Al igual que el *VR Robotics Simulator,* este programa es compatible con *Oculus Rift* y *HTC Vive,* y permite al usuario cambiar la posición del brazo con la ayuda del mando, así como programar un camino a recorrer pasando por distintos puntos alcanzables y un programa interactivo de entrenamiento de operarios.

Explicado en resumidas cuentas, el programa consiste en un entorno virtual programado en Unity, en el cuál se produce toda la interacción con el usuario. Dicho entorno virtual se comunica con RobotStudio para llevar a cabo los cálculos de la cinemática del brazo robótico. RobotStudio recibe las coordenadas del objetivo elegido por el usuario a través de los mandos de realidad virtual, y RobotStudio se encarga de calcular los ángulos de las articulaciones necesarios para dicho objetivo. El entorno virtual en Unity también se comunica con PLCSIM para simular el comportamiento de los sensores y actuadores virtuales.

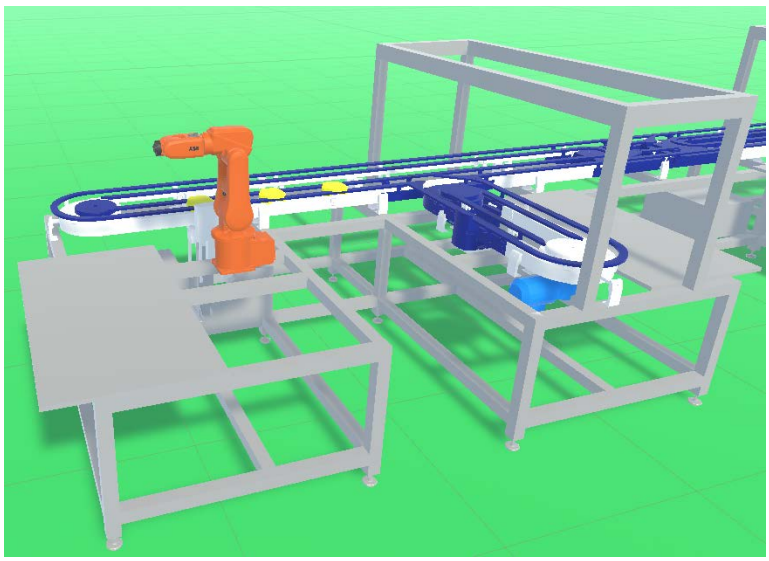

Figura 8: IRB 120 y minifábrica. Captura del entorno virtual del programa de Carlos Álvarez Vereterra

Como puede verse, este programa también ofrece un amplio abanico de posibilidades y mejoras de cara al futuro, ya que Unity es un entorno muy versátil y extendido.

3. Objetivo del proyecto

El objetivo de este proyecto consiste en permitir la programación y simulación de un brazo robótico de grandes dimensiones, concretamente en IRB 8700, diseñado y producido por ABB. Para ello, el proyecto se basará en el programa ya desarrollado por Carlos Álvarez Vereterra, en el que se implantará el modelo del IRB 8700 para que desempeñe las mismas funciones que el IRB 120 ya implementado. También se trabajará en una herramienta tipo pinza para darle un mayor realismo al modelo virtual, y se hará que la misma sea capaz de coger y depositar objetos. Por último, se adaptará el programa para que el manejo de un robot de grandes dimensiones sea más fácil implementando un zoom que el usuario puede manejar con los botones del mando del set de realidad virtual.

Así, esta aplicación pasará a estar más completa, ya que el usuario no solo podrá familiarizarse con la programación y el movimiento de robots de pequeñas dimensiones, sino también de grandes dimensiones y así ser consciente de las diferencias entre unos y otros de manera visual e intuitiva.

4. Puesta a punto para el proyecto

Antes de comenzar a desarrollar el proyecto, fue necesario adquirir una serie de conocimientos de los que no se disponían, ya que se desconocían conceptos básicos de robótica, cómo se mueve un brazo robótico y cómo se programa, así como el manejo de los programas Unity y RobotStudio.

4.1. Aprendizaje de conceptos de robótica y RobotStudio

Estos conocimientos son normalmente adquiridos a través de la asignatura de Automatización y Robótica Industrial, sin embargo, ya que esta asignatura no ha sido cursada aún, la familiarización con el brazo robótico y RobotStudio ha tenido que llevarse a cabo de manera independiente con la ayuda del director del proyecto.

La manera más eficaz de conseguir esto en el menor tiempo posible ha sido a través de la realización de dos sesiones de prácticas pertenecientes a la asignatura mencionada.

La primera práctica se centra en el manejo y programación del robot IRB 120 a través del FlexPendant, que es un panel de control interactivo con pantalla táctil y diversos botones.

Figura 9: Fotografía de un FlexPendant, idéntico al utilizado durante la práctica

Así, en esta práctica se aprendió a realizar diversas acciones básicas de control del brazo robótico, las cuales son las siguientes:

- Movimiento del robot según sus ejes con distintos grados de precisión: consiste en mover el robot cambiando los ángulos de las articulaciones.
- Movimiento lineal del robot: consiste en mover el objetivo del robot (la punta de la herramienta) de manera lineal en los 3 ejes cartesianos del espacio.

- Programación de recorrido en el espacio: se definen 3 puntos por los cuales el objetivo del robot tiene que pasar, definiendo así un recorrido. Para ello se debe usar el modo programación del FlexPendant y definir ahí las tres instrucciones del movimiento, como muestra la siguiente figura:

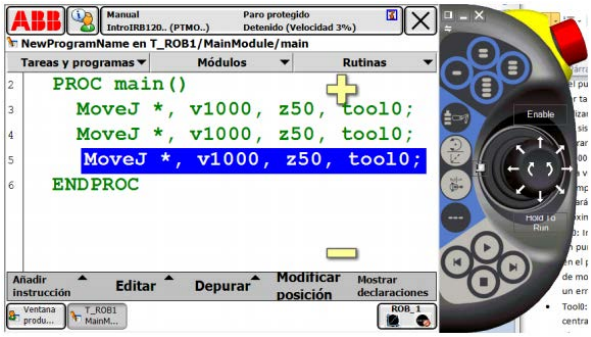

Figura 10: Captura del FlexPendant con las instrucciones

- Recorrido con retardo: punto análogo al anterior, pero definiendo los puntos a partir del punto anterior con un desplazamiento dado.
- Uso de funciones (como por ejemplo, recorrer un cuadrado introduciendo sus puntos) y manejo de la pinza para agarrar y soltar objetos.

La segunda práctica se centra más en el manejo de RobotStudio. En ella se aprendió a crear un proyecto e importar el modelo del brazo robótico deseado, la utilidad de cada una de las pestañas del programa, añadir una herramienta al brazo, desplazar el robot a través de articulaciones o linealmente, y simular una estación ya creada con ayuda de RAPID, el lenguaje de programación de RobotStudio.

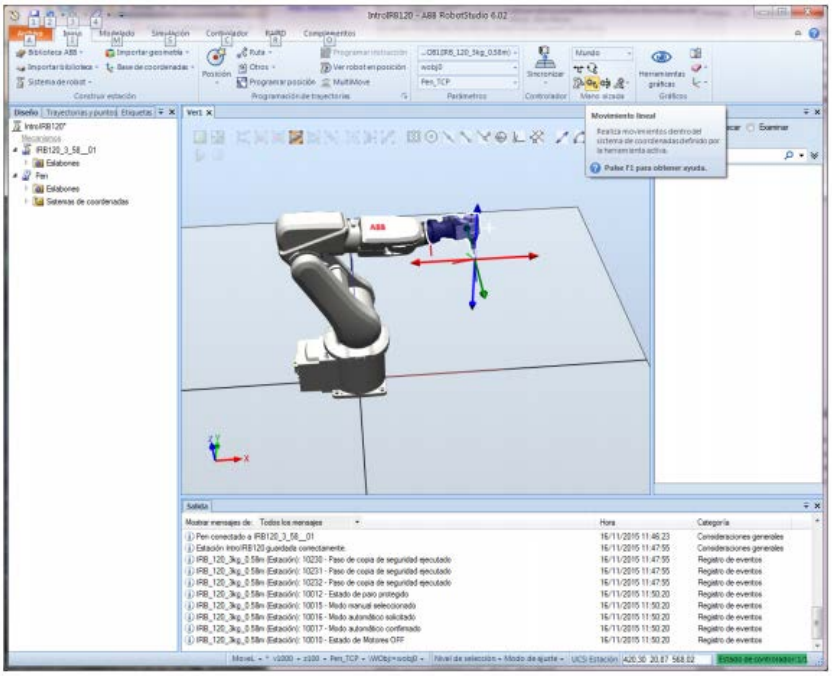

Figura 11: Entorno de programación de RobotStudio, en este caso listo para el movimiento lineal

En esta práctica también se aprendió a definir una herramienta en el robot real a través del FlexPendant.

4.2. Aprendizaje de cómo programar en Unity

Unity es un motor de videojuego que sirve como herramienta para crear escenas de juego de manera sencilla pero también de manera muy potente y eficiente. Este programa fue el elegido por Carlos Álvarez Vereterra para crear el entorno virtual de su programa, y, por lo tanto, es el que se usó también en este proyecto para adaptar dicho programa a los robots de grandes dimensiones.

Antes de comenzar a programar en este entorno, fue necesario realizar un curso online presente en la plataforma Udemy. El curso consistía en diversos videos de tipo tutorial en los que se explicaba desde cero cómo programar un videojuego en un entorno de realidad virtual [12]. La suma total de la duración de los vídeos fue de 5 horas, pero la duración real de la realización del curso fue significativamente mayor, ya que para aprender bien lo que se estaba explicando, fue necesario ir paso a paso haciendo las mismas acciones que se realizaban en el tutorial.

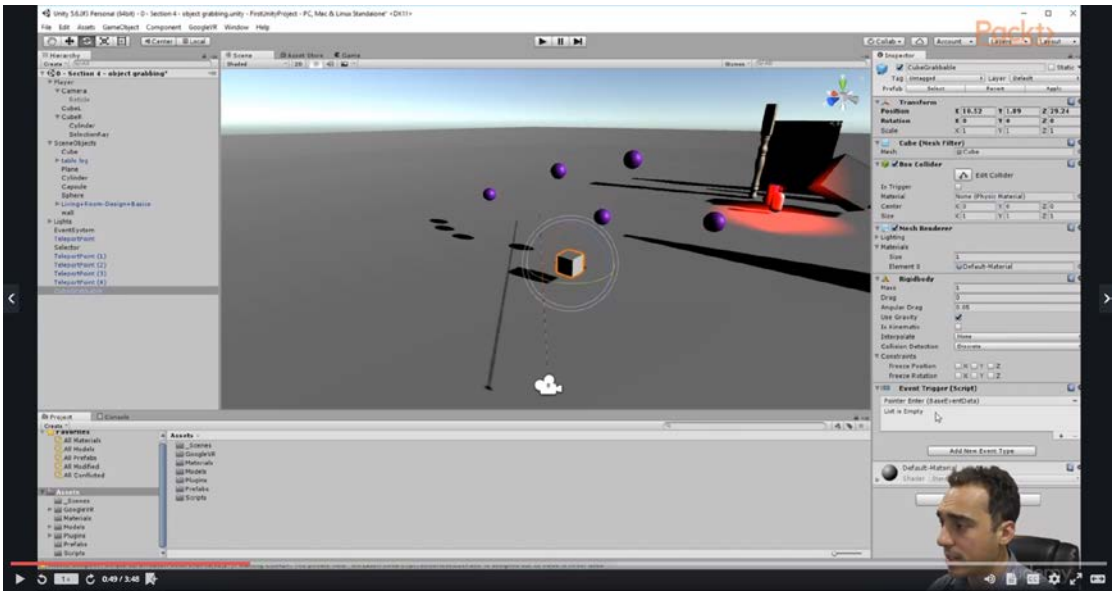

Figura 12: Captura de pantalla de uno de los vídeos del curso en Udemy

Gracias a este curso se adquirieron los siguientes conocimientos, necesarios para poder programar con soltura en esta herramienta:

- Familiarización con la interfaz de usuario y la estructura y arquitectura de Unity.
- Aprendizaje de cómo crear un script de comportamiento, en qué casos hace falta, y cómo programarlo en C#.
- Aprendizaje de qué son los objetos en Unity, sus propiedades y cómo añadirlos a una escena.
- Aprendizaje de cómo implementar una interacción básica con un set de realidad virtual como Oculus Rift.
- Implementación del movimiento de distintos objetos presentes en la escena o del propio jugador.
- Implementación de interacción entre el jugador y los distintos elementos de la escena (agarrar objetos, teletransportarse o incluso programar un videojuego interactivo muy sencillo con un objetivo a alcanzar).

4.3. Aprendizaje del funcionamiento del programa ya desarrollado

Una vez siendo capaz de entender cómo funciona un brazo robótico y cómo programar en Unity, fue necesario entender cómo funcionaba el programa de Carlos Álvarez Vereterra para basarnos en él e implantar las funciones deseadas.

Para ello, se hizo un estudio a fondo de la memoria de su proyecto [11] para entender el entorno en Unity, los scripts de Unity y los scripts de RobotStudio.

Hubo dificultades destacables en entender cómo funciona la comunicación entre ambos programas, ya que se desconocía el funcionamiento de los protocolos TCP/IP y UDP/IP, los cuales se utilizaron para dos hilos de comunicaciones de distinta información pero carácter similar.

Una vez comprendido el funcionamiento del programa, se pasó a ejecutarlo y comprobar que todo funcionaba correctamente. Sin embargo, hubo diversos problemas que retrasaron el comienzo de desarrollar nuevo contenido para la aplicación.

El primero fue un fallo con el plug-in de interacción con el set de realidad virtual, ya que este no funcionaba correctamente y no era posible establecer una conexión entre el entorno virtual de Unity y nuestro set de realidad virtual Oculus Rift. Este plug-in es el de *OpenVR,* compatible tanto con Oculus Rift como con HTC Vive, y se encontraba en su versión 1.0.

En primer lugar se decidió actualizarlo a su versión 2.0, pero esta solución no fue posible ya que todo el programa y todos los scripts ya desarrollados estaban basados en la versión 1.0 del plug-in, y adaptarlos todos a la nueva versión sería un proceso demasiado engorroso.

Fue entonces cuando se consultó al autor de la aplicación ya desarrollada para ver cómo solucionar este problema. Tras un tiempo inspeccionando el programa, se encontró el causante del problema, y es que aunque el plug-in estaba importado en los ficheros del proyecto en Unity, era necesario también instalar el paquete encargado del funcionamiento del plug-in. Esto se hizo a través del administrador de paquetes, y una vez instalado, la interacción con el set Oculus Rift comenzó a funcionar correctamente.

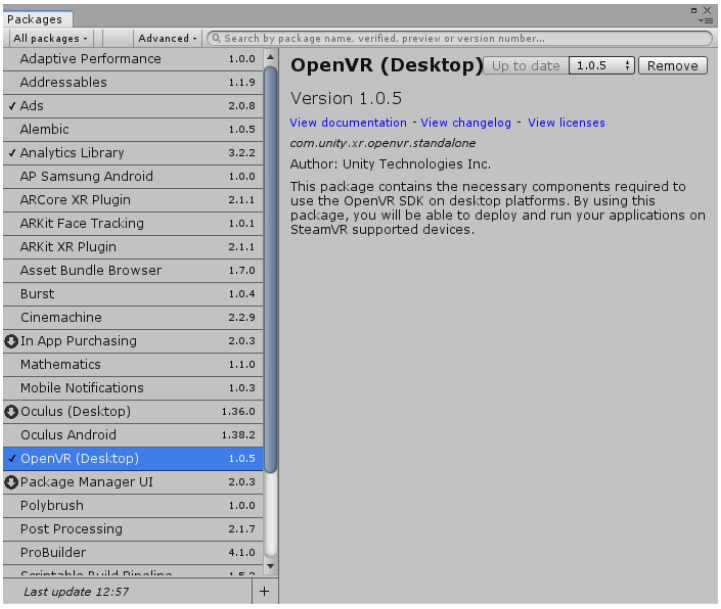

Figura 13: Administrador de paquetes de Unity

El siguiente problema llegó al intentar mover el brazo robótico, ya que fallaba la comunicación entre Unity y RobotStudio, la cual se encarga de los cálculos necesarios para el movimiento del brazo robótico tras seleccionar un nuevo objetivo al que queremos que vaya el robot.

Primero se inspeccionó el script de RAPID en RobotStudio y se vio que todo el código era correcto. Al hacer la prueba de mover el robot, la ventana de comandos daba el error de que el objetivo seleccionado no era alcanzable, por lo tanto, pudo suponerse que el problema estaba en el script de comunicaciones del programa en Unity.

Al inspeccionar dicho script, pudo verse que el error estaba en la línea que enviaba la información a RobotStudio, ya que en vez de enviarse el punto definido por el usuario, se enviaba un punto con las coordenadas escritas a mano en dicho código, el cual no era alcanzable y, aunque lo fuese, no permitiría moverse el robot a la posición deseada.

```
void CommsRobotStudio()
{
    while (!abort)
    {
        if (flag)
        {
            flag = false;
            Debug.Log("beforeclienttcp");
            clientRobotStudio = new
             TcpClient(ipRobotStudioTCP,portRobotStudioTCP);
            streamRobotStudio = clientRobotStudio.GetStream();
            Debug.Log("afterclienttcp");
            streamRobotStudio.ReadTimeout = 2000;
            streamRobotStudio.WriteTimeout = 2000;
            float testf = 203.593f - 50 * i;

            dataSend = System.Text.Encoding.ASCII.GetBytes(testf.ToString() +
            ";-242.103;431.301;0.658873;0.409562;0.569494;-0.271703;-
            1;0;0;1");
            //dataSend = System.Text.Encoding.ASCII.GetBytes(message);
```

Código 1: *Communications.cs* – Fragmento que envía un punto fijo erróneo

Como puede verse, la última línea de esa parte del script está comentada, y la línea de código superior envía a RobotStudio un punto introducido manualmente. Visto el error, se borró esta línea y se hizo válida la última línea que estaba comentada.

Una vez cambiado esto, la comunicación entre Unity y RobotStudio comenzó a funcionar correctamente, y con ella el programa completo desarrollado por Carlos Álvarez Vereterra.

5. Implementación de un zoom

La primera nueva funcionalidad introducida en la aplicación es un zoom que permite al usuario desenvolverse con soltura en el entorno de realidad virtual con un robot de grandes dimensiones. Con el robot IRB 120 este zoom puede no ser tan necesario, puesto que su tamaño es comedido y fácilmente accedemos a toda la región del espacio alcanzable por el brazo.

Sin embargo, el IRB 8700, que mide 3 metros de alto y tiene un alcance de 3,5 metros, requiere de esta herramienta para que el usuario sea capaz de moverlo por todo su espacio alcanzable sin muchas dificultades.

5.1. Dificultades para implementar un zoom en realidad virtual

En el mundo de los videojuegos, en la mayoría encontramos distintas clases de zooms, pero la más extendida sin duda es la de reducir y aumentar el campo de visión del jugador. Este método es el que se usa, por ejemplo, en los juegos de tipo *shooter* en primera persona, cuando se apunta por la mirilla del arma.

En Unity, variar el campo de visión de la cámara del jugador es muy sencillo a través de un script [13]. Sin embargo, cuando se intentó implementar este método a través de un script en nuestro programa, apareció el siguiente error:

Figura 14: Error en Unity al intentar cambiar el campo de visión

Este error dice que no se puede alterar el campo de visión de una cámara cuando está activada la realidad virtual. Esto es debido a que el campo de visión de la cámara virtual en Unity debe coincidir con el campo de visión que tienen realmente las gafas del set de realidad virtual. Por lo tanto, los cambios en el campo de visión cuando se usa la realidad virtual pueden producir fuertes mareos en el usuario, por lo que Unity impide que este campo pueda ser alterado mientras el programa esté funcionando [14].

5.2. Alternativas e implementación

Ya que no se pudo implantar el zoom a través de modificar el campo de visión, se pasó a buscar alternativas que permitieran conseguir el resultado deseado.

En un primer intento, se hizo una prueba que consistía en poder modificar la escala de todos los componentes de la escena (minifábrica y robots) cuando el usuario lo deseara.

El resultado no fue el esperado, ya que al variar la escala el programa empezó a dar problemas y a no funcionar correctamente. Esto es porque al aumentar la escala de los objetos, éstos chocaban unos entre otros al conservar la posición de su centro de gravedad. Esto hacía que el motor físico de Unity diera problemas y los objetos pasaran a tener un comportamiento extraño.

Por lo tanto, se pasó a intentar modificar la escala del propio jugador según se pulsaran los botones correspondientes en el mando.

Así, cuando el jugador ha aumentado su escala, se encuentra con un robot más pequeño, el cual puede mover rápidamente de un lado para otro. Y cuando disminuye su escala, el jugador encuentra al robot muy grande y puede mover el objetivo con precisión pero con un alcance reducido.

Para ello, se implementó un script dentro del objeto correspondiente al mando izquierdo del jugador (ya que el derecho está reservado para el control del robot). En dicho script, llamado *cameraZoom.cs*, se cambia la escala del jugador según éste presione el joystick del mando hacia arriba o hacia abajo. Dentro del comando *Update()*, el cual se ejecuta en cada fotograma del programa, para que el script detecte si el jugador está pulsando el joystick verticalmente, se usa el comando *GetAxis().y,* el cual devuelve un valor entre 1 y -1 (valores positivos para joystick hacia arriba y negativos hacia abajo). Si se detecta una pulsación, la escala se reduce o aumenta correspondientemente a su velocidad correspondiente (*zoomSpeed*) siempre y cuando no vaya a salirse de los rangos establecidos (*minScale* y *maxScale*).

Una vez reducida o aumentada la escala (recordemos que todo esto ocurre en cada fotograma), se calcula el desplazamiento que habría entre la posición inicial de la cámara (cuya altura se corresponde con la cabeza del jugador) y la final después de aumentar o disminuir la escala. Este desplazamiento entonces se le resta a la posición final de la cámara, haciendo así que la posición de la cámara permanezca constante aunque aumentemos o disminuyamos la escala del jugador y evitando mareos a éste. Por ejemplo, si no hiciéramos esto y aumentásemos la escala del jugador, la cabeza ascendería y viceversa, y eso es lo que se ha evitado gracias a este método.

Todo esto puede verse reflejado y comentado en el siguiente fragmento de código:

```
void Update()
{

    posIni = camera.gameObject.transform.position; //get camera's initial position

    //zoomIn
    if (hand.controller.GetAxis().y >= 0.5)   //joystick up detected
    {

        if (scale >= minScale) //don't go below minimum
        {
            scale -= Time.deltaTime * zoomSpeed;  //reduce scale
        }
        player.gameObject.transform.localScale = new Vector3(scale, scale, scale); //change the actual
                                                                                     player's scale

        posFin = camera.gameObject.transform.position;        //get final position
        displacement = posFin - posIni;                       //calculate displacement between final and
                                                                initial position
        player.gameObject.transform.position -= displacement; //substract displacement so we stay in the
                                                                same position

    }


    //zoomOut
    if (hand.controller.GetAxis().y <= -0.5)
    {
        if (scale <= maxScale)
        {
            scale += Time.deltaTime * zoomSpeed;
        }
        player.gameObject.transform.localScale = new Vector3(scale, scale, scale);

        posFin = camera.gameObject.transform.position;
        displacement = posFin - posIni;
        player.gameObject.transform.position -= displacement;


    }
}
```

Código 2: *cameraZoom.cs* – Implementación del zoom y corrección del desplazamiento de la cámara

6. Modelado del robot ABB IRB 8700

6.1. Modelo en RobotStudio

El modelado del robot IRB 8700 en RobotStudio fue muy sencillo, ya que este programa es desarrollado por ABB y tiene una base de datos con los modelos de todos los robots, incluido el que se quiere implementar.

Para ello, es necesario crear un nuevo proyecto. A continuación, hay que seleccionar *Solución con estación y controlador virtual*, y en el desplegable *Modelo del robot* seleccionar el modelo deseado, tal y como se muestra en la siguiente figura:

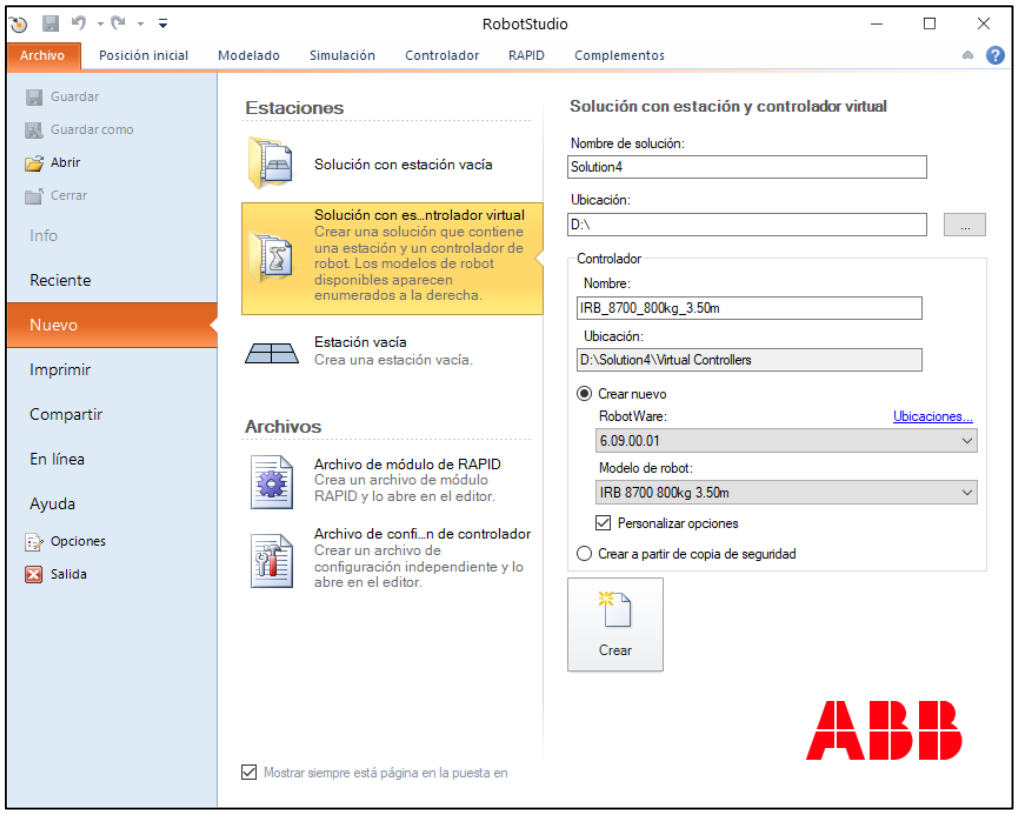

Figura 15: Creación de un nuevo proyecto en RobotStudio

Una vez creado el proyecto, podemos visualizar el brazo robótico en nuestra estación de trabajo simulada.

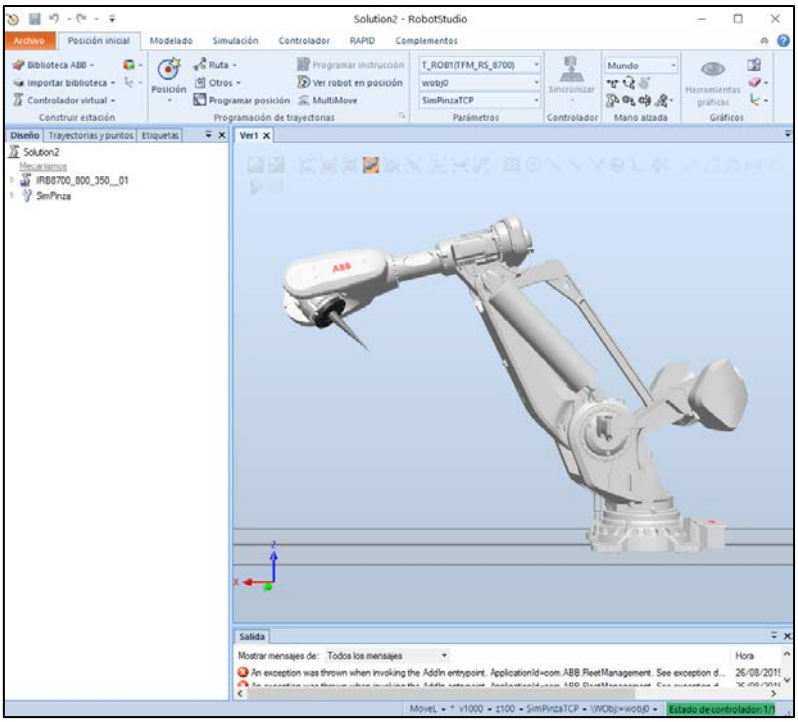

Figura 16: Estación de trabajo en RobotStudio con el IRB 8700

Ya solo queda asignarle el código en RAPID que definirá su comportamiento. El código que se implementará será el mismo usado en el programa de RobotStudio desarrollado previamente a este proyecto [11]. Dicho código funciona para cualquier tipo de brazo robótico, y se compone de dos tareas simultáneas:

- Tarea principal: basada en el protocolo de comunicación TCP/IP, se encarga de recibir las coordenadas del nuevo objetivo del robot cuando el usuario las defina en el entorno virtual de Unity. Hace los cálculos necesarios y, siempre que el objetivo sea alcanzable por el robot, envía los valores de los ángulos de las articulaciones necesarios para alcanzarlo. Si el punto no es alcanzable, también envía a Unity la información que dice que no lo es. Por último, esta tarea también se encarga de recibir y guardar los puntos que el usuario determine para la ejecución de un camino y hacer que el robot simulado ejecute dicho camino cuando el usuario lo desee.

- Tarea Secundaria: basada en el protocolo de comunicación UDP/IP, se encarga de enviar a Unity los ángulos de las articulaciones del robot simulado siempre que este esté ejecutando un camino.

6.2. Importación del modelo 3D en Unity

Para importar la geometría 3D del IRB 8700, es necesario descargar el archivo que contiene la misma de la página web de ABB [7]. Sin embargo, a diferencia del IRB 120, la importación a Unity no es tan sencilla, ya que en la página web de ABB, los formatos disponibles para descargar el IRB 120 incluyen el formato *Google Sketchup*, el cual es directamente compatible con Unity y su importación se hace de manera inmediata.

Sin embargo, ninguno de los formatos disponibles para descargar el modelo del IRB 8700 son directamente importables en Unity, por lo que fue necesario descargar el modelo 3D en otro formato y convertirlo a alguno compatible en Unity.

Primeramente, se intentó descargar el modelo en formato IGES y pasarlo al anteriormente mencionado *Google Sketchup*. Para ello se utilizó el programa de escritorio de *Google Sketchup*, para poder así importar en el modelo en formato IGES y exportarlo en formato *Sketchup*.

Sin embargo, al importar el archivo obtenido en Unity, se observa que la geometría tiene errores, y en distintos ángulos se ve material que falta y aristas mal definidas, por lo que esta solución no fue válida debido a la pésima presentación del formato obtenido.

Como solución alternativa, se usó el programa CAD Exchanger. Este programa permitió convertir los archivos en formato IGES a formato FBX, directamente compatible con Unity. La conversión tuvo que ser realizada pieza a pieza, y una vez importadas todas a nuestra escena en Unity, el resultado sí que fue satisfactorio.

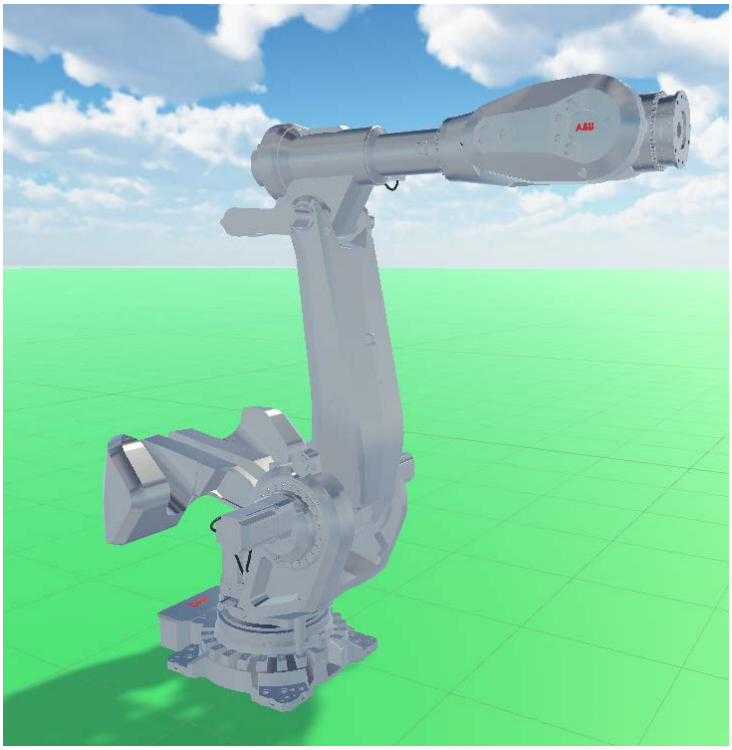

Figura 17: Modelo del IRB 8700 en el entorno virtual de Unity

Como puede apreciarse, esta vez la geometría no tenía ningún error y la escala era correcta. El modelo carecía de color por lo que se decidió dotarlo de un material metalizado. También se optó por el rojo para las serigrafías de ABB y el negro para los cables.

6.3. Comportamiento del robot en Unity

Para que el robot funcionara correctamente, fue necesario dotarlo de un script en Unity para que éste se moviera de la manera deseada (análoga a la ya implementada en los robots IRB 120).

Para ello, el primer paso fue agrupar las distintas piezas que componen el robot en un árbol de jerarquía análogo al ya implementado en el IRB 120. Así, se crean objetos vacíos para cada uno de los 6 ejes que tienen asignados como hijos la pieza de dicho eje. Finalmente, cada eje contiene como hijo al siguiente, empezando por el primero y llegando hasta el sexto.

Como resultado de esto, si se cambia la rotación de un eje, el resto de ejes contenidos en éste se mueven también solidarios a él.

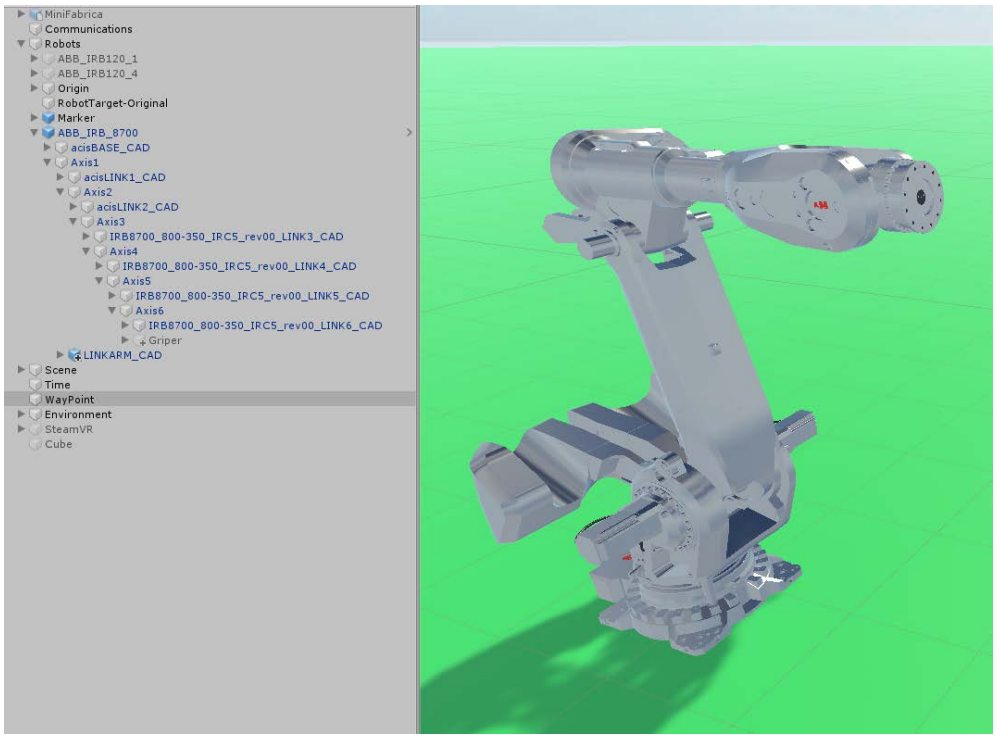

Figura 18: IRB 8700 y su jerarquía

Por último, queda definir el script que implementará el comportamiento del robot. De nuevo, se ha optado por basarse en el script del IRB 120, el cual cuenta con 6 variables (*a1, a2, a3, a4, a5, a6),* que se corresponden con los ángulos de las articulaciones del robot. El script se encarga de cambiar la rotación de los ejes convenientemente a partir de los valores de las 6 variables mencionadas.

```
void Update()
{
    // Linear interpolate to angle
    tAxis1.localRotation = Quaternion.Lerp(tAxis1.localRotation,
                           Quaternion.Euler(0, 0, -a1), 5 * Time.deltaTime);
    tAxis2.localRotation = Quaternion.Lerp(tAxis2.localRotation,
                           Quaternion.Euler(0, -a2, 0), 5 * Time.deltaTime);
    tAxis3.localRotation = Quaternion.Lerp(tAxis3.localRotation,
                           Quaternion.Euler(0, -a3+a2, 0), 5 * Time.deltaTime);
    tAxis4.localRotation = Quaternion.Lerp(tAxis4.localRotation,
                           Quaternion.Euler(a4, 0, 0), 5 * Time.deltaTime);
    tAxis5.localRotation = Quaternion.Lerp(tAxis5.localRotation,
                           Quaternion.Euler(0, -a5, 0), 5 * Time.deltaTime);
    tAxis6.localRotation = Quaternion.Lerp(tAxis6.localRotation,
                           Quaternion.Euler(a6, 0, 0), 5 * Time.deltaTime);
}
```

Código 3: *IRB8700.cs* - Cambio en la rotación de los ejes a partir de las variables a1-a6

A diferencia que el IRB 120, en el IRB 8700 la rotación del eje 3 funciona de manera distinta. Cuando se cambia únicamente el ángulo del eje 2, el ángulo del eje 3 se modifica para mantener constante la orientación del eje 3 (esto es una peculiaridad de este modelo). Para corregir esto y que el robot se mueva de manera correcta, en vez de definir la orientación del eje 3 únicamente a partir de *a3*, se define como la resta de las variables *a2* y *a3*. Todo esto se ve reflejado en el fragmento de código que se muestra anteriormente (código 3).

Los valores de los ángulos de las articulaciones (*a1*-a6) serán modificados por el script de comunicaciones cuando sea pertinente (es decir, cuando se esté ejecutando un camino o se haya cambiado la posición del objetivo por el usuario). Dicha modificación se implementa a través de la función *MoveRobot* que será llamada por el script cuando sea necesario.

```
void MoveRobot(IRB8700 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");

}
```

Código 4: *Communications.cs* - Función *MoveRobot*

Una vez importado el robot a RobotStudio y Unity y asignado su comportamiento, el usuario ya dispone de un robot completamente funcional capaz de moverse igual que el IRB 120 desarrollado previamente.

El usuario puede ahora modificar la posición y la orientación del objetivo del robot, así como guardar varios puntos para posteriormente hacer que el robot siga un camino pasando por todos ellos.

7. Modelado de una herramienta tipo “pinza”

Para dar más realismo al robot, se decidió dotarlo de una herramienta de tipo pinza en su punta, así como asignarle un comportamiento para que sea capaz de coger y depositar objetos.

7.1. Importación del modelo 3D en Unity

El primer paso fue importar la geometría 3D de una pinza paralela adecuada para el brazo robótico y su tamaño. Para ello se buscó un modelo en del fabricante Schunk, cuya especialidad son los sistemas de agarre para brazos robóticos.

Se optó por una pinza paralela debido a la sencillez en su diseño y comportamiento, y de entre todas las pinzas paralela se escogió el modelo con mayor fuerza de cierre [15] para que se correspondiera con las dimensiones del IRB 8700.

Esta vez, la página web sí que disponía de formatos directamente compatibles con Unity, por lo que no hubo ningún problema.

Sin embargo, el modelo 3D de la herramienta consistía únicamente en el mecanismo que porta las garras, sin incluir las propias garras.

Figura 19: Pinza de Schunk sin garras, a partir de la cual se genera el archivo 3D

Para obtener una pinza completamente funcional capaz de agarrar y soltar objetos, fue necesario buscar una manera de obtener un modelo 3D de unas garras compatibles, pero en la página web del fabricante no se ofrece ningún modelo 3D de estos elementos.

Por lo tanto, la opción más conveniente fue generar la geometría de una garra a mano en Unity de la manera más sencilla posible. Para ello se usó el plug-in ProBuilder, el cual es una herramienta muy potente para generar geometrías 3D, pero también permite diseñar geometrías sencillas rápidamente sin necesidad de profundizar mucho en el funcionamiento del programa.

La geometría de la garra finalmente diseñada puede parecer sencilla, pero cumple a la perfección con la función requerida y no tuvo que desperdiciarse mucho tiempo en su diseño. Finalmente, la geometría de la garra fue duplicada para disponer así de garra izquierda y derecha, y ambas se integraron con la geometría del mecanismo de la pinza. La pinza completa fue situada en el sexto eje del robot dentro de su jerarquía, haciéndola solidaria al movimiento del eje.

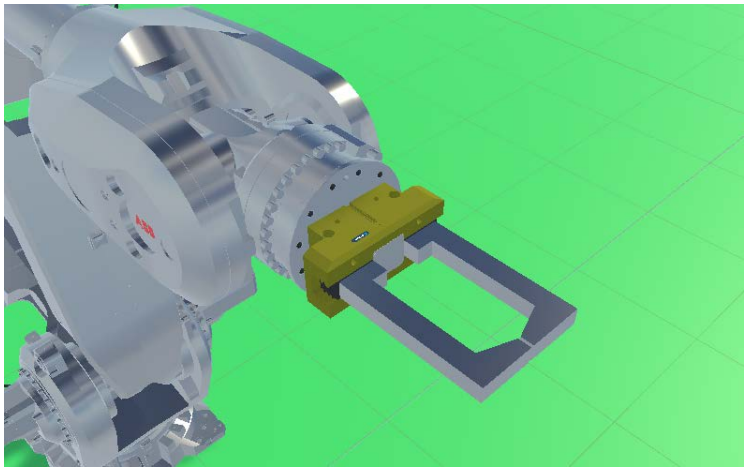

Figura 20: Pinza completa integrada en el IRB 8700

7.2. Comportamiento de la pinza en Unity

Para que la pinza sea capaz de coger y soltar objetos según se lo indique el usuario, fue necesario añadir una serie de scripts que definen el comportamiento de este objeto en Unity.

Para empezar, era necesario que las garras de la pinza pudieran abrirse y cerrarse según el usuario se lo indique a través del mando de la realidad virtual. Para ello se modificó primeramente el script de comportamiento del robot, añadiendo una nueva variable llamada *grip*, la cual varía entre 0 y 140. Esta variable asigna directamente la posición de cada una de las garras. El 0 se corresponde con las garras cerradas, y 140 con las garras abiertas al máximo (para evitar tener que hacer conversiones, se visualizó manualmente el valor de la posición de la garra completamente abierta, que se correspondía aproximadamente con 140, por lo tanto se forzó a que la variable *grip* se moviera entre 0 y 140).

```
void Update()
{
    leftFinger.localPosition = new Vector3(grip, 0, 0);
    rightFinger.localPosition = new Vector3(-grip, 0, 0);
}
```

Código 5: IRB8700.cs - Fragmento del código que varía la posición de ambas garras a partir de la variable *grip*

Como puede observarse en el código anterior, a una de las garras se le asigna el valor de *grip* con signo positivo y a otra con signo negativo, esto es para que las pinzas se muevan en direcciones opuestas como ocurre en la realidad.

Para que el usuario pueda controlar la apertura y el cierre de las pinzas, esto se hace a través del joystick del mando encargado de controlar el movimiento del robot. El código implementado se integra por lo tanto en el script *ViveController.cs*, junto a todos los comandos implementados anteriormente por Carlos Álvarez Vereterra.

De manera análoga al control del zoom mencionado anteriormente, se vuelve a utilizar la función *GetAxis()*, esta vez en el eje horizontal del joystick. El código se encarga de variar la variable *grip* del objeto IRB8700 y de que ésta varíe únicamente entre 0 y 140.

```
if (hand.controller.GetAxis().x >= 0.5)
{
    if (robot.grip < 140)
    {
        robot.grip += Time.deltaTime * grip_speed;
    }
}

if (hand.controller.GetAxis().x <= -0.5  )
{
    if (robot.grip > 0)
    {
        robot.grip -= Time.deltaTime * grip_speed;
    }
}
```

Código 6: *ViveController.cs* - Cambio de la variable *grip* del robot según se mueva horizontalmente el joystick

Una vez queda completamente implementado el control de la apertura y cierre de la pinza, falta definir el comportamiento de las garras para que sean capaces de agarrar y soltar un objeto.

Para ello, a cada una de las garras se le ha asignado el script *gripper.cs.* Este script se ayuda de los coliders y colisiones de Unity [16], que sirven para sacar partido a las simulaciones de la física de los objetos. Así, el script hace uso de las funciones *OnCollisionStay* [17] y *OnCollisionExit* [18], las cuales se ejecutan únicamente cuando el objeto al que pertenece el script está en contacto con otro o deja de estarlo respectivamente . Además, se ha impuesto que el código de estas funciones sólo se ejecute cuando el objeto contiene una etiqueta que lo define como agarrable (la cual será *grab*) para que no haya confusiones y nuestras pinzas solo interactúen con los objetos que definamos previamente como agarrables.

Este script contiene una variable booleana clave, la cual indica si la garra está en contacto o no con un objeto agarrable en tiempo real (esta variable es *collided*).

El funcionamiento consiste en que, si ambas pinzas están en contacto con el objeto (es decir, estamos dentro de la función *OnCollisionStay*), este pasará a tener su posición emparentada con una de las pinzas (que definiremos con la etiqueta *parent*). Para ello se ha utilizado la función *SetParent.* Además, cuando ambas pinzas estén en contacto con el objeto, la gravedad dejará de hacer efecto sobre éste a través de la función *UseGravity,* gracias a eso el podrá ser levantado del suelo.

Así, gracias a emparentar la posición y orientación del objeto con una de las pinzas y a ignorar la gravedad, dará la sensación de que las pinzas están agarrando el objeto.

Dentro de la función *OnCollisionStay()* también se comprueba continuamente si la otra garra sigue estando en contacto con el objeto, para ello se comprueba la variable *collided* de la otra garra. Así, si el contacto se pierde con alguna de las dos garras, se fuerza a que el objeto deje de estar emparentado con la garra y la gravedad vuelva a actuar sobre él, haciendo que el objeto se suelte y caiga directamente al suelo o donde se deposite.

Para que todo funcione correctamente, también se debe usar la función *OnCollisionExit*, que hará que la variable *collided* pase a *False* indicando así que el objeto ha dejado de estar en contacto con la pinza y se active el código que permite soltarlo explicado en el párrafo anterior.

```
private void OnCollisionStay(Collision collision)
{
    if (collision.transform.tag == "grab")
    {
        collided = true;


        if (other_gripper.collided && this.transform.tag == "parent")
        {
            (collision.transform).SetParent(this.gameObject.transform);
            if (collision.transform.GetComponent<Rigidbody>())
            {
                collision.transform.GetComponent<Rigidbody>().useGravity = false;
            }
        }

        if (!other_gripper.collided && this.transform.tag == "parent")
        {
            (collision.transform).SetParent(null);
            if (collision.transform.GetComponent<Rigidbody>())
            {
                collision.transform.GetComponent<Rigidbody>().useGravity = true;
            }
        }
    }

}

private void OnCollisionExit(Collision collision)
{
    if (collision.transform.tag == "grab")
    {
        collided = false;
    }
}
```

Código 7: *gripper.cs* – Funciones *OnCollisionStay* y *OnCollisionExit*

Por último, hay que asegurarse de que las garras no sean capaces de cerrarse más cuando ambas entran en contacto con el objeto. Si esto no se hiciera, las garras atravesarían el objeto provocando errores en la física del entorno virtual.

Para ello, se ha modificado ligeramente el fragmento del código de *ViveController.cs* que se encarga del control de la apertura y cierre de las garras a través del joystick del mando.

```
if (hand.controller.GetAxis().x <= -0.5  )
{
    if (robot.grip > 0 && (!(right_gripper.collided) ||!(left_gripper.collided)))
    {
        robot.grip -= Time.deltaTime * grip_speed;
    }
}
```

Código 8: *ViveController.cs* – Control del cierre de las garras modificado

Con esta ligera modificación, la variable *gripper* no podrá reducirse si ambas garras están en contacto con el objeto (para ello se analiza la variable *collided* de cada una de las garras).

Una vez todo el código ha sido implementado en los objetos correspondientes del programa, las pinzas no solo son capaces de abrirse y cerrarse a voluntad del usuario, sino que también simulan agarrar un objeto si ambas entran en contacto con uno y soltarlo si mientras se agarra el objeto las garras se abren, simulando así el comportamiento de unas pinzas paralelas reales.

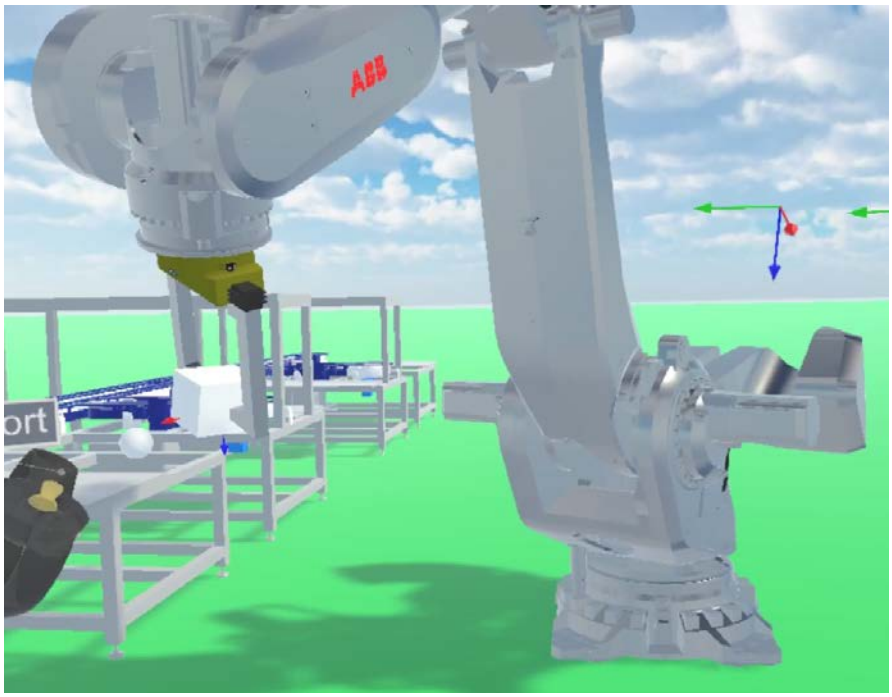

Figura 21: IRB 8700 sujetando un objeto (cubo blanco) a través de las pinzas

7.3. Modelado de la herramienta en RobotStudio

Para un correcto manejo del robot, se desea que, en el entorno virtual en Unity, cuando se marque un objetivo, la parte del robot que éste lleve al objetivo sea la punta de las pinzas (el final de las pinzas), y no la base de éstas (que es la terminación del eje 6 del robot sin herramienta).

Para que en el cálculo del movimiento del robot se tenga en cuenta este desplazamiento, hay que modificar la posición del objetivo del robot en RobotStudio por medio del modelado de una herramienta.

A priori puede parecer que lo que debe hacerse es modelar la herramienta completamente en RobotStudio, con toda la geometría de las piezas. Sin embargo, se ha optado por una solución mucho más rápida y práctica, que consiste en usar una geometría lo más sencilla posible que desplace el punto objetivo una distancia igual a la medida del largo de la pinza en Unity. Así, el punto objetivo del robot en RobotStudio quedará desplazado del final del eje 6 una distancia igual al largo de la pinza, cumpliendo con el objetivo deseado.

Después de medir la longitud de la pinza en Unity, se ha creado una geometría sencilla (un cono) en RobotStudio con una altura idéntica.

Para conseguir darle un carácter de herramienta a la geometría, se han seguido los pasos del tutorial creado por Juan Carlos Martín Castillo [19], que consiste en pulsar el botón Crear Herramienta en la pestaña de Modelado y seguir los pasos correspondientes para convertir el cono creado en una herramienta.

Después de hacer esto, podemos posicionar nuestra herramienta en el robot, quedando de la siguiente manera:

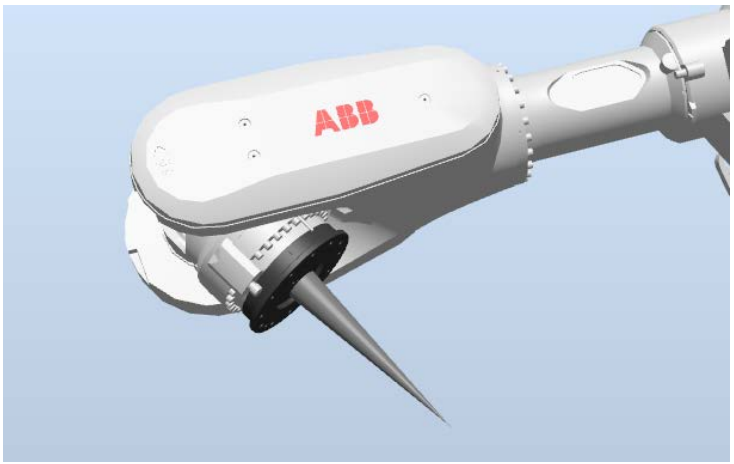

Figura 22: Simulación de herramienta posicionada en el IRB 8700 en RobotStudio

Por último, sólo falta modificar el código en RAPID para que al mover el robot y hacer los cálculos se utilice la nueva herramienta creada, para ello, cada vez que se llame a una función que mueva el robot o calcule los ángulos de las articulaciones, se sustituirá *tool0* que es la herramienta correspondiente a no tener ninguna herramienta, por el nombre de la nueva herramienta personalizada.

Gracias a esto, ahora en Unity cuando el usuario selecciona un objetivo, el robot trata de alcanzarlo con la punta de la pinza, y no con el final del eje 6, concluyendo con la integración de una herramienta de trabajo completamente funcional para el IRB 8700.

8. Resultados y conclusiones

Tras todas las mejoras implementadas en la aplicación desarrollada por Carlos Álvarez Vereterra, el resultado es que el usuario puede manejar un modelo del IRB 8700 con todas las funciones anteriormente implementadas en el robot IRB 120.

Además, se ha incluido una herramienta de tipo pinza que no solamente dota al robot de realismo estético, sino que esta es completamente funcional y con ella se pueden agarrar y soltar objetos del entorno virtual y, por lo tanto, el comportamiento del robot se acerca aún más a uno del mundo real.

También se ha implementado el zoom que hace que el manejo del robot de grandes dimensiones sea posible, ya que con este método de manejo el usuario debe ser capaz de alcanzar cualquier punto que quiera fijar como objetivo del robot.

Como ya se ha mencionado, la realidad virtual hace mucho más rápido e intuitivo mover el brazo robótico y poder manejarlo de manera remota sin tener que estar cerca del brazo real. Esto hace que este proceso sea mucho más seguro, especialmente si estamos hablando de robots capaces de mover cargas muy pesadas.

Hacer que el usuario pueda experimentar con distintos tamaños de robots al manejarlos a través de la realidad virtual, dotarlos de elementos que los hagan más reales y funcionales, e incluir características que hagan más cómoda la experiencia de usuario nos acerca un poco más a un futuro donde la realidad virtual sea una herramienta muy común cuando se hable de programación de robots industriales

Bibliografía

1. *Realidad Virtual.* Wikipedia (URL: https://es.wikipedia.org/wiki/Realidad_virtual). 26/08/2019
2. *Aplicaciones de la Realidad Virtual en formación y educación.* Editeca (URL: https://editeca.com/realidad-virtual-formacion-educacion/). 05/11/2018
3. *Virtualware desarrolla simulador de Realidad Virtual para Formación en PRL.* Virtualware, YouTube (URL: https://www.youtube.com/watch?v=9bep1Wmb6pk). 26/10/2017
4. *Tendencias 2017: La realidad virtual, a la caza de su momento 'Pokémon Go'.* BBVA. (URL: https://www.bbva.com/es/tendencias-2017-realidad-virtual-caza-momento-pokemon-go/). 21/12/2016
5. *El Robot Industrial de Ensamblaje Automotriz.* Ángel Aranguren, Motor Y Racing. (URL: https://www.motoryracing.com/coches/noticias/el-robot-industrial-de-ensamblaje-automotriz/). 18/07/2018
6. *TOP LARGEST ROBOTIC ARMS EVER BUILT.* MEE Services. (URL: https://www.meee-services.com/top-largest-robotic-arms-ever-built/). 08/11/2018
7. *IRB 8700.* ABB Robotics. (URL: https://new.abb.com/products/robotics/es/robots-industriales/irb-8700). 27/08/2019
8. *ABB Robotics IRB 8700 Launch at CIIF 2015.* ABB Robotics, YouTube. (URL: https://www.youtube.com/watch?v=D8rE__LVcH4&t=44s). 14/12/2015
9. *VR Robotics Simulator,* MindRend Technologies (URL: http://vrrobotsim.com/). 27/08/2019
10. *Using virtual reality to operate robots remotely.* Haptical. (URL: https://haptic.al/using-virtual-reality-to-operate-robots-remotely-ce7766eea342). 14/11/2018
11. Carlos Álvarez Vereterra. *Uso de la realidad virtual para el entrenamiento del personal de operación y mantenimiento: aplicación a la minifábrica ICAI*. Trabajo fin de máster. ICAI, junio de 2018.
12. *Building your First VR Experience with Unity,* Udemy (URL: https://www.udemy.com/course/building-your-first-vr-experience-with-unity/). Marzo de 2018
13. *Camera.fieldOfView.* Unity Documentation. (URL: https://docs.unity3d.com/ScriptReference/Camera-fieldOfView.html). 27/08/19
14. *Camera FOV (Field of View) not changeable at all.* Github (URL: https://github.com/googlevr/gvr-unity-sdk/issues/938). 16/07/2018
15. *PGN-plus.* Schunk (URL: https://schunk.com/es_es/sistemas-de-agarre/series/pgn-plus/). 27/08/19

16. *Colliders.* Unity Documentation. (URL: https://docs.unity3d.com/Manual/CollidersOverview.html). 27/08/19
17. *MonoBehaviour.OnCollisionStay(Collision).* Unity Documentation (URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnCollisionStay.html). 27/08/19
18. *MonoBehaviour.OnCollisionExit(Collision).* Unity Documentation (URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnCollisionExit.html). 27/08/19
19. *RobotStudio: Actividad 4 (Configuración básica de la herramienta).* Juan Carlos Martín Castillo (URL: https://www.youtube.com/watch?v=_jLw5q5rANg). 10/01/2015

Presupuesto

1. Recursos

Componente	Cantidad (unidades)
Ordenador compatible con realidad virtual	1
Tarjeta gráfica compatible con realidad virtual	1
Oculus Rift	1

2. Precios unitarios

Componente	Precio (€)
Ordenador compatible con realidad virtual	1600
Tarjeta gráfica compatible con realidad virtual	130
Oculus Rift	190

3. Presupuesto general

Componente	Coste (€)
Ordenador compatible con realidad virtual	1600
Tarjeta gráfica compatible con realidad virtual	130
Oculus Rift	190
TOTAL	**1.920**

Código fuente

1. CameraZoom.cs

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

public class cameraZoom : MonoBehaviour

{

    private Hand hand;
    public float scale = 1f;
    public float zoomSpeed = 0.1f;
    public float minScale = 0.001f;
    public float maxScale = 5;
    public GameObject player;
    public GameObject camera;
    private Vector3 posIni, posFin, displacement;


    void Start()
    {

    }

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

        if (hand.controller == null)
            return;

        posIni = camera.gameObject.transform.position;

        //zoomIn
        if (Input.GetKey(KeyCode.C) || hand.controller.GetAxis().y >= 0.5)
        {

            if (scale >= minScale)
            {
                scale -= Time.deltaTime * zoomSpeed;
            }
            player.gameObject.transform.localScale = new Vector3(scale, scale,
                                                                 scale);

            posFin = camera.gameObject.transform.position;
            displacement = posFin - posIni;
            player.gameObject.transform.position -= displacement;

        }

        //zoomOut
        if (Input.GetKey(KeyCode.X) || hand.controller.GetAxis().y <= -0.5)
        {
            if (scale <= maxScale)
            {
```

```
                scale += Time.deltaTime * zoomSpeed;
            }
            player.gameObject.transform.localScale = new Vector3(scale, scale,
                                                                 scale);

            posFin = camera.gameObject.transform.position;
            displacement = posFin - posIni;
            player.gameObject.transform.position -= displacement;

        }

    }

}
```

2. IRB8700.cs

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

public class IRB8700 : MonoBehaviour {

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


    public float grip;
    public GameObject leftFinger;
    public GameObject rightFinger;


    // 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;


    private Transform left, right;


    // 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>();
        left = leftFinger.GetComponent<Transform>();
        right = rightFinger.GetComponent<Transform>();


    }

    // Update is called once per frame
    void Update()
    {
        // Linear interpolate to angle
        tAxis1.localRotation = Quaternion.Lerp(tAxis1.localRotation,
Quaternion.Euler(0, 0, -a1), 5 * Time.deltaTime);
```

```
        tAxis2.localRotation = Quaternion.Lerp(tAxis2.localRotation,
Quaternion.Euler(0, -a2, 0), 5 * Time.deltaTime);
        tAxis3.localRotation = Quaternion.Lerp(tAxis3.localRotation,
Quaternion.Euler(0, -a3+a2, 0), 5 * Time.deltaTime);
        tAxis4.localRotation = Quaternion.Lerp(tAxis4.localRotation,
Quaternion.Euler(a4, 0, 0), 5 * Time.deltaTime);
        tAxis5.localRotation = Quaternion.Lerp(tAxis5.localRotation,
Quaternion.Euler(0, -a5, 0), 5 * Time.deltaTime);
        tAxis6.localRotation = Quaternion.Lerp(tAxis6.localRotation,
Quaternion.Euler(a6, 0, 0), 5 * Time.deltaTime);


        left.localPosition = new Vector3(grip, 0, 0);
        right.localPosition = new Vector3(-grip, 0, 0);

    }
}
```

3. Gripper.cs

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

public class gripper : MonoBehaviour
{

    public bool collided;
    public GameObject other_gripper;
    private gripper other;


    // Start is called before the first frame update
    void Start()
    {
        other = other_gripper.GetComponent<gripper>();
    }

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

    }

    private void OnCollisionStay(Collision collision)
    {
        if (collision.transform.tag == "grab")
        {
            collided = true;



            if (other.collided && this.transform.tag == "parent")
            {
                (collision.transform).SetParent(this.gameObject.transform);
                if (collision.transform.GetComponent<Rigidbody>())
                {
                    collision.transform.GetComponent<Rigidbody>().useGravity =
                    false;
                }
            }



            if (!other.collided && this.transform.tag == "parent")
            {
                (collision.transform).SetParent(null);
                if (collision.transform.GetComponent<Rigidbody>())
                {
                    collision.transform.GetComponent<Rigidbody>().useGravity =
                    true;
                }
            }
        }



    }
```

```
    private void OnCollisionExit(Collision collision)
    {
        if (collision.transform.tag == "grab")
        {
            collided = false;
        }
    }
}
```

4. ViveController.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;


    public GameObject robot_go;
    private IRB8700 robot;
    public int grip_speed = 5;
    public GameObject leftFingerTip;
    public GameObject rightFingerTip;
    private gripper right_gripper, left_gripper;

    // 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;

    robot = robot_go.GetComponent<IRB8700>();

    right_gripper = rightFingerTip.GetComponent<gripper>();
    left_gripper = leftFingerTip.GetComponent<gripper>();

}


// 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;

//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);

    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);
    }


    //Execute path
    if (hand.controller.GetPressDown(SteamVR_Controller.ButtonMask.Touchpad))
    {
        Debug.Log("Success");
        SendMessage(res, savedRotationEuler, 0, false, true);
    }


    // Grip control

    if (hand.controller.GetAxis().x >= 0.5)
    {
        if (robot.grip < 140)
        {
            robot.grip += Time.deltaTime * grip_speed;
        }
    }

    if (hand.controller.GetAxis().x <= -0.5  )
    {
        if (robot.grip > 0 && (!(right_gripper.collided) ||
          !(left_gripper.collided)))
            //stop if both fingers collide with the object
        {
            robot.grip -= Time.deltaTime * grip_speed;
        }
    }

}


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;
    }
}
```