

Trabajo Fin de Máster

Ingeniería Industrial

Desarrollo del Gemelo Digital de Alimentador de Bandejas e Integración en Entorno de Realidad Virtual

Autor: Miguel Ángel Ridaol Olivar

Tutor: Juan Manuel Escaño González

Cotutor: Javier Gómez Jiménez

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022



Trabajo Fin de Máster
Ingeniería Industrial

Desarrollo del Gemelo Digital de Alimentador de Bandejas e Integración en Entorno de Realidad Virtual

Autor:

Miguel Ángel Ridaol Olivar

Tutor:

Juan Manuel Escaño González

Profesor titular

Cotutor:

Javier Gómez Jiménez

Dpto. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Máster: Desarrollo del gemelo digital de alimentador de bandejas e integración en entorno de realidad virtual

Autor: Miguel Ángel Ridaol Olivar
Tutor: Juan Manuel Escaño González
Cotutor: Javier Gómez Jiménez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2022

El Secretario del Tribunal

Agradecimientos

A mi familia y amigos por el apoyo mostrado durante mi etapa de formación universitaria.

Por supuesto, agradecer también a mis tutores Juan Manuel Escaño y Javier Gómez Jiménez, por todo el tiempo dedicado y su orientación a la hora de hacer este proyecto.

Y al equipo del proyecto DENiM, por dejarme ser parte del trabajo.

Miguel Ángel Ridao Olivar

Alumno del Máster en Ingeniería Industrial

Sevilla, 2022

Resumen

En el presente trabajo se tiene como objetivo principal la elaboración del gemelo digital del alimentador de bandejas presentes en una célula de fabricación flexible, la cual se encuentra ubicada en los laboratorios del departamento de Ingeniería de Automática y Sistemas de la Universidad de Sevilla. Además, dicho gemelo digital se integra dentro de un sistema de realidad virtual, el cual se complementa con un sistema SCADA que permite la visualización de datos.

Abstract

The main objective of this project is the development of the digital twin of the box feeder in a flexible manufacturing cell, which is located in the laboratories of the Department of Automation and Systems Engineering of the University of Seville. Furthermore, this digital twin is integrated into a virtual reality system, which is complemented with a SCADA system that allows data visualisation.

Índice

Agradecimientos	vii
Resumen	ix
Abstract	xi
Índice	xiii
Índice de Figuras	xv
1 Introducción	19
2 Programas empleados	23
2.1 Autodesk 3DS Max	23
2.1.1. Interfaz de 3DS Max	24
2.2.2. Integración de 3DS Max con Unity 3D	27
2.2 Unity 3D	28
2.2.1 Unity Hub	29
2.2.2. Interfaz de Unity	30
2.3. Visual Studio Code	35
2.3.1. Integración de VS Code con Unity 3D	36
3 Descripción de la planta	39
3.1 Célula de fabricación flexible	39
3.2 Alimentador de bandejas	41
4 Desarrollo del gemelo digital	43
4.1 Elaboración de los modelos 3D	43
4.1.1 Estructura del alimentador de bandejas	43
4.1.2 Retenedores de bandejas	45

4.1.3 Sensor de proximidad tipo barrera	45
4.1.4 Cuadro de mandos	46
4.2 Desarrollo del comportamiento físico de los elementos	48
4.2.1 Sensores	51
4.2.2 Actuadores	53
4.2.3 Botones del cuadro de mando	54
4.2.4 Indicadores led	56
5 Sistema SCADA	59
5.1 Desarrollo del sistema SCADA	60
5.1.1 Implementación de ventana de gráficos	60
5.1.2 Representación de curvas gráficas	61
5.1.3 Botones para la interacción con la ventana gráfica.	64
6 Sistema de realidad virtual	67
6.1 Dispositivo Oculus Quest 2	67
6.1.1 Características técnicas	67
6.1.2 Tipos de conexiones	69
6.2. Creación de proyectos RV en Unity 3D	70
6.2.1 Visualización de los mandos	70
6.2.2 Desplazamientos	71
6.2.3 Giros	72
6.2.4 Teletransporte	73
6.2.5 Interactuación con objetos	77
7 Conclusiones y trabajos futuros	81
8 Anexo I	83
8.1 Configuración del dispositivo Oculus Quest 2	83
8.2 Instalación del paquete XR Toolkit en Unity 3D	83
8.3. Exportación de Unity 3D a Oculus	86
8.3.1 Exportación standard	87
8.3.2 Exportación con Oculus Link	88
9 Anexo II	89
10 Anexo III	91
Referencias	117

Índice de Figuras

Figura 1.1 - Gemelos digitales.	20
Figura 1.2 - Objetivos del proyecto DENiM.	21
Figura 2.1 - Interfaz de trabajo de 3DS Max.	24
Figura 2.2 - Barra de herramientas de 3DS MAX.	24
Figura 2.3 - Explorador de escena de 3DS MAX.	25
Figura 2.4 - Ventana de visualización de 3DS MAX.	26
Figura 2.5 - Panel de comandos de 3DS MAX.	26
Figura 2.6 - Controles de barra de estado de 3DS MAX.	27
Figura 2.7 - Exportar archivos en formato FBX.	27
Figura 2.8 - Interfaz de Unity Hub	29
Figura 2.9 - Ventana Add Modules de Unity Hub.	30
Figura 2.10 - Interfaz de trabajo de Unity.	30
Figura 2.11 - Barra de Herramientas.	31
Figura 2.12 - Gizmos Rotate, Scale y Translate.	31
Figura 2.13 - Ventana Hierarchy.	32
Figura 2.14 - Ventana Scene.	32
Figura 2.15 - Ventana Game.	33
Figura 2.16 - Ventana Stats.	33
Figura 2.17 - Ventana Instructor.	34
Figura 2.18 - Ventana Project.	35
Figura 2.19 - Ventana Console.	35
Figura 2.20 - Interfaz de Visual Studio Code.	36
Figura 2.21 - Estructura general de un Script.	37
Figura 3.1 - Esquema de la célula flexible de fabricación.	40
Figura 3.2 - Comparativa entre el modelo 3D y la célula de fabricación flexible real.	40
Figura 3.3 - Esquema del alimentador de bandejas.	41
Figura 3.4 - Proceso de giro para el guardado de cajas.	42
Figura 4.1 - Comparativa entre el modelo 3D y el real del alimentador de bandejas.	43
Figura 4.2 - Comparativa entre el modelo 3D y pistón de elevación real.	44

Figura 4.3 - Comparativa entre el modelo 3D y el cilindro de giro real.	44
Figura 4.4 - Comparativa entre el modelo 3D y el retenedor de bandejas real.	45
Figura 4.5 - Comparativa entre el modelo 3D y el sensor de proximidad tipo barrera real.	46
Figura 4.6 - Comparativa entre el modelo 3D y el cuadro de mandos real.	46
Figura 4.7 - Panel de control para la interacción con el alimentador de bandejas	47
Figura 4.8 - Componente Rigidbody.	48
Figura 4.9 - Ejemplos de colliders empleados.	49
Figura 4.10 - Componente Mesh Collider.	49
Figura 4.11 - Componente Physic material.	50
Figura 4.12 - Sensor inductivo.	51
Figura 4.13 - Sensor de barrido.	53
Figura 4.14 - Componente canvas.	55
Figura 4.15 - Componente Button.	55
Figura 4.16 - Asignación de objetos a variables tipo GameObject	57
Figura 5.1 - Ventana del sistema SCADA.	60
Figura 5.2 - Ajustes automáticos de los valores del eje Y.	61
Figura 5.3 - Ajuste automático sin dependencia del número de datos mostrado.	62
Figura 5.4 - Comparativa de distintas configuraciones visuales de la gráfica de barras.	63
Figura 5.5 - Comparativa de distintas configuraciones visuales de la gráfica de líneas	64
Figura 6.1 - Dispositivo Oculus Quest 2.	68
Figura 6.2 - Mandos Oculus Quest 2.	68
Figura 6.3 - Visualización de los mandos en la escena.	71
Figura 6.4 - Continuos Move Provider.	72
Figura 6.5 - Snap Turn Provider.	73
Figura 6.6 - Configuración de la forma del Ray Interactor.	74
Figura 6.7 - Configuración visual del Ray Interactor.	74
Figura 6.8 - Visualización de los XR Ray Interactor.	75
Figura 6.9 - Configuración de la activación de los rayos.	75
Figura 6.10 - Configuración de los eventos para la activación de los rayos.	76
Figura 6.11 - Activación del modo teletransporte.	76
Figura 6.12 - Configuración de la vibración de los mandos.	77
Figura 6.13 - Interacciones con el sistema UI.	78
Figura 6.14 - Simultaneidad de teletransporte y objeto agarrado.	78
Figura 6.15 - Mejora de la visualización al agarrar objetos.	79
Figura 6.16 - Agarrar objetos en VR.	79
Figura 6.17 - Sistema de información de componentes.	80
Figura 6.18 - Configuración elemento XR Simple Interactable.	80
Figura 8.1 - Activación de XR Plug-in Management.	84
Figura 8.2 - Activación de paquetes en previsión.	84
Figura 8.3 - Instalación del paquete XR Plugin Management.	85
Figura 8.4 - Activación del paquete de acciones para los mandos.	85
Figura 8.5 - Configuración del Preset Manager.	86
Figura 8.6 - Componentes del elemento XR Rig.	86
Figura 8.7 - Exportar archivos de forma standard.	87
Figura 8.8 - Exportar archivos mediante Oculus Link.	88

1 INTRODUCCIÓN

Hoy en día, donde la información tiene un gran valor para el desarrollo y el avance en muchos aspectos, el tener la capacidad de gestionar y emplear gran cantidad de información acerca de un entorno determinado es de gran importancia, ya que hace que este se vuelva más controlable y predecible, gracias a lo cual se podrá crear modelos que recreen su funcionamiento fielmente. Es a partir de esta idea, donde surge el concepto de *gemelo digital*.

Antes de nada, es importante saber que es un gemelo digital, aunque dentro de la industria no es un concepto que esté totalmente definido, ya que se le ha dado multitud de definiciones. La primera definición práctica del gemelo digital la da Herranz en [1], como "contraparte virtual e informatizada de un sistema físico que puede utilizarse para simularlo con diversos fines". El gemelo digital consiste en una representación digital de un objeto, proceso o servicio físico. Y gracias a estas replicas digitales se pueden realizar simulaciones antes de que estas sean creadas físicamente o implementen cambios en objetos reales.

Como ya se ha mencionado para poder crear los gemelos digitales es necesario contar con gran cantidad de información sobre los objetos físicos. Esta información hoy en día es fácil de obtener gracias a que muchos de estos objetos cuentan con numerosos sensores encargados de recopilar datos en tiempo real sobre su estado, condiciones de trabajo o posición. Con toda esta información, se crea un programa informático el cual los usa para recrear simulaciones que pueden predecir cómo funcionará un producto o procesos.

Los gemelos digitales se emplean para dos cuestiones principales. Por un lado, para la realización de prototipos digitales antes de crear el producto final, de esta forma se sabe cómo será realmente y cuál será su comportamiento. Y, por otro lado, se emplean para realizar pruebas en distintos escenarios de uso sin tener que hacerlo con el modelo real, evitando todos los inconvenientes que ello conllevaría [1].

Importantes grupos industriales están apostando por la introducción de los gemelos digitales en su actividad. Por ejemplo, General Electric ha desarrollado modelos digitales para motores de aviación, parques eólicos, plataformas petrolíferas "off-shore", etc. General Electric también es socio fundador de "Digital Twins Consortium", cuyo objetivo es acelerar el uso de los Gemelos Digitales mediante demostraciones reales del valor de esta tecnología. Importantes empresas

como Johnson Controls, Microsoft o Autodesk. Rolls-Royce también está trabajando en gemelos digitales, en concreto para sus motores de aviación y con el objetivo de aumentar su eficiencia.

A nivel nacional también existen importantes empresas que se han involucrado en el desarrollo de gemelos digitales. Por ejemplo, Sacyr está creando gemelos digitales para optimizar la gestión de sus plantas industriales de agua. Muchas empresas de desarrollo de productos para la industria, tales como Siemens, Schneider Electric o Sothis a nivel nacional, ofrecen en su cartera productos basados en el desarrollo de gemelos digitales.

Otro aspecto importante que compone este trabajo es la realidad virtual, a partir de ahora RV. Y es que este nuevo tipo de tecnología tiene una gran cantidad de utilidades en distintos campos de la vida, como son la educación, formación, ocio, cultura e industria. Y es en este último campo, la industria, donde se centra este trabajo.



Figura 1.1 - Gemelos digitales [25].

En la industria, la cual cada día se encuentra más digitalizada, y donde se busca ahorrar costes y mejorar la seguridad de sus trabajadores, en la realidad virtual se ha encontrado un gran aliado. Ya que, gracias a ella, se pueden evitar elaborar costosas y lentas maquetas físicas de carrocerías de coches o aviones, entre otras cosas. Y en su lugar se puede elaborar un sistema CAD que incorpore todas las dimensiones y características, mediante el cual se genera una "maqueta" de RV, la cual, permite al usuario interactuar con ella y observar todos los detalles que necesite. Además, todas las modificaciones necesarias se podrían hacer de forma rápida dentro del gemelo digital [2]. En cuanto a la seguridad, el uso de esta tecnología principalmente permite formar a los trabajadores en un ambiente muy similar a donde desempeñarán su trabajo, pero eliminando los posibles peligros.

Actualmente, cada vez más empresas están integrando la realidad virtual en aplicaciones, campañas de marketing, etc. Por ejemplo, VELUX, una empresa especializada en techo y tragaluces lanzó una aplicación para poder visualizar los beneficios de sus servicios en el propio hogar del cliente. La empresa Volvo la cual usa esta tecnología para evaluar prototipos, diseños y tecnologías de seguridad activa, mediante una conducción real a la que se le añaden elementos virtuales que actúan como reales tanto para conductor como para el vehículo [29].

Este trabajo se encuentra dentro de las tareas del proyecto europeo Digital intelligence for collaborative ENergy management in Manufacturing (DENiM) el cual persigue, la gestión de la energía dentro del sector manufacturero, uno de los sectores con el mayor consumo energético

del planeta. Por ello, para hacer una gestión eficiente de la energía que se consume es clave para garantizar la competitividad y sostenibilidad durante la transición energética mundial. Con este objetivo el proyecto DENiM desarrolla “una serie de herramientas integradas para el suministro de servicios digitales avanzados, esto incluye conectividad segura en la periferia y con el Internet de las Cosas (IdC), analítica de datos, gemelos digitales, modelización de energía y automatización”. Con esta tecnología se quiere conseguir tener la capacidad de supervisar y optimizar de forma automática el uso de energía, al mismo tiempo que se informa al usuario acerca del impacto tanto ambiental como económico de cada una de las decisiones que se toman a lo largo del proceso de fabricación. El proyecto se va a ensayar en cinco plantas reales de varios países, entre los que se incluye Irlanda, Italia, España o Eslovenia [3].

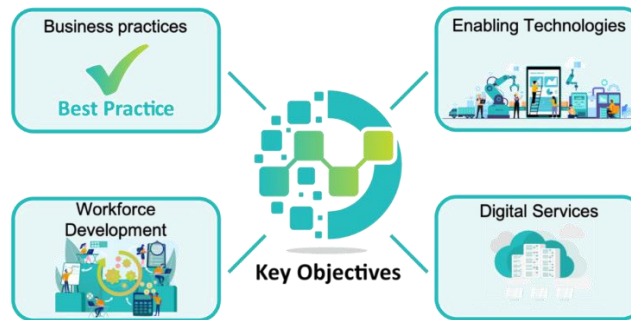


Figura 1.2 - Objetivos del proyecto DENiM [26].

El objetivo principal de este trabajo es la creación de un gemelo digital de un alimentador de bandejas situado dentro de una célula de fabricación flexible, además, se elaborará un sistema de realidad virtual de la planta completa. Para ello, la estructura de este trabajo se compone de varias partes diferenciadas. En primer lugar, se explicarán cada uno de los programas informáticos que han sido necesarios para la elaboración del proyecto. En segundo lugar, se hará una explicación de la planta que se ha modelado y seguidamente se desarrollará todo el proceso seguido para la creación del gemelo digital de la misma. En tercer lugar, se expondrá el sistema SCADA realizado, así como las principales características que lo componen. Y en último lugar, se explicará cómo se ha creado el sistema de realidad virtual mediante el uso del dispositivo Oculus Quest 2.

2 PROGRAMAS EMPLEADOS

En este apartado, se explican cada uno de los programas informáticos que han sido utilizados para la elaboración del proyecto. Entre estos programas se encuentran Autodesk 3DS Max para el diseño gráfico por ordenador, Unity 3D como motor gráfico, y Visual Studio Code como editor de código.

2.1 Autodesk 3DS Max

En este proyecto, el primer paso a realizar es la elaboración de los modelos 3D de todos los elementos que se quieren representar. Para ello, es necesario contar con un programa que permita crear estos modelos gráficos 3D. Actualmente, existen numerosas alternativas como Cinema4D, Autodesk Maya o ZBrush. Sin embargo, en este trabajo se optó por el programa 3DS Max debido al gran potencial que ofrece.

El software 3DS Max es un programa de gráficos por ordenador utilizado para crear modelos 3D, animaciones e imágenes digitales. Es un software de modelado, animación y renderizado en 3D construido y desarrollado para la visualización de juegos y diseños. A menudo, esta herramienta es empleada para la creación de personajes, edificios y las propiedades físicas de líquidos como agua entre otras cosas [8].

Como uno de los paquetes 3D más utilizados en el mundo, 3DS Max es una parte integral de muchos estudios profesionales y constituye una parte importante de su línea de producción de juegos y películas. 3DS Max se utiliza en la industria de los videojuegos para crear modelos de personajes en 3D, activos de juegos y animaciones. También, es popular para anuncios de televisión y efectos especiales de cine, 3DS Max se utiliza a menudo para generar gráficos para su uso junto con el trabajo de acción en vivo.

3DS Max encaja en el proceso de animación en casi todas las etapas. Desde el modelado y rigging hasta la iluminación y el renderizado, este programa facilita la creación de animaciones de calidad profesional de forma más fácil y sencilla. Las industrias inmobiliaria y arquitectónica utilizan 3DS Max para generar imágenes fotorrealistas de edificios en la fase de diseño. 3DS Max utiliza el modelado poligonal que es una técnica común en el diseño de juegos.

Para este proyecto, este software ha sido empleado para realizar todo lo relacionado con el modelado 3D de los elementos que componen el apilador de cajas, como son la estructura principal del apilador, los retenedores y el cuadro de control. Una vez ya han sido diseñados, se exportarán al programa Unity tal y como se explicarán en el apartado 2.2.

2.1.1. Interfaz de 3DS Max

El software 3DS Max, cuenta con una gran cantidad de herramientas y funciones con las que poder realizar gran variedad de modelos 3D virtuales, animaciones, renderizados, como se ha mencionado anteriormente. Es por ello, que en este documento no se van a explicar todas y cada una de dichas funciones. El alcance del documento es explicar las herramientas que han sido necesarias para el desarrollo del proyecto, y, por tanto, únicamente se describirán de forma breve las herramientas que se centran en la creación de los modelos en 3D.

Como se puede ver en la figura 2.1, la interfaz de trabajo de 3DS MAX se divide en cinco sectores principales. En primer lugar, encontramos la barra de herramientas (A); a la derecha se encuentra el explorador de escenas (B); en el centro de la pantalla está la ventana de visualización (C); a la izquierda de la pantalla se encuentra el panel de comandos (D); y en último lugar, en la parte inferior de la pantalla aparece los controles de barra de estado (E).

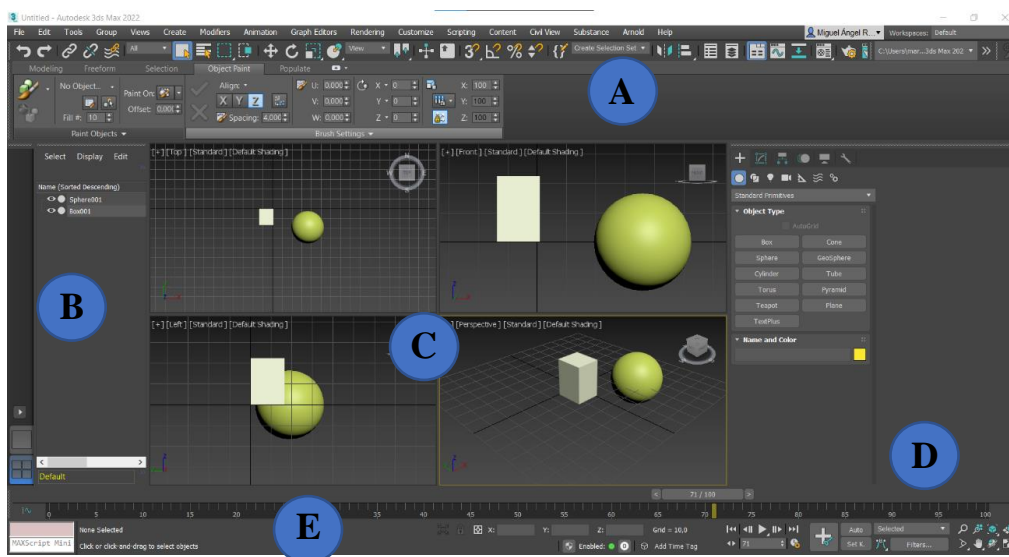


Figura 2.1 - Interfaz de trabajo de 3DS Max.

Barra de herramientas

Se encuentra en la parte superior de la pantalla y, por un lado, proporciona comandos como Deshacer/Rehacer y gestión de archivos, entre otros. Además, de una lista desplegable para cambiar entre las diferentes interfaces de trabajo.

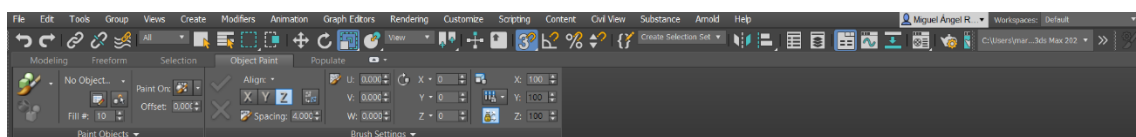



Figura 2.2 - Barra de herramientas de 3DS MAX.

También, por otro lado, se encuentran también muchos de los comandos principales que se utilizan en 3ds Max. De entre todas ellas, destaca la representada con el siguiente icono . Esta herramienta actúa como un asistente a la hora de dibujar el modelo 3D, ya que permite seleccionar vértices, líneas, puntos medios, caras, entre otros, lo que facilita al usuario a tener una mayor precisión durante el proceso.

Explorador de escenas

La función de esta ventana es la visualización, poder ordenar, filtrar, agrupar y seleccionar objetos, así como cambiar el nombre, eliminar, ocultar, y congelar objetos. Es una ventana que representa la jerarquía de todos y cada uno de los elementos presentes en la escena. De igual modo, permite realizar un diseño dividido en varias capas.

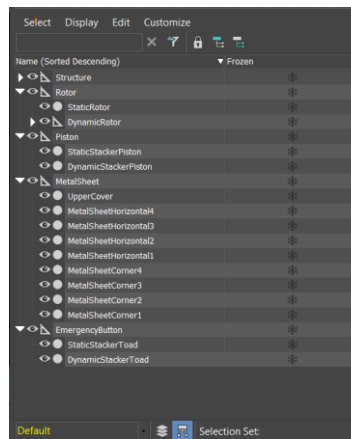


Figura 2.3 - Explorador de escena de 3DS MAX.

Ventana de visualización

La ventana de visualización es el espacio donde se muestra el modelo 3D. Permite ver varias vistas a la vez o solo una única vista de este, según las preferencias del usuario, tal y como se muestra en la figura 2.4, donde aparecen el alzado, perfil, planta y perspectiva del modelo. Desde esta ventana se puede obtener una vista preliminar de iluminación, sombras, la profundidad de campo y otros efectos.

3DS MAX cuenta con gizmos que permiten al usuario visualizar los ejes de coordenadas, y así poder hacer las acciones de mover, rotar y escalar con mayor facilidad. De igual modo, cuenta con la opción de activar un grill para facilitar el diseño.

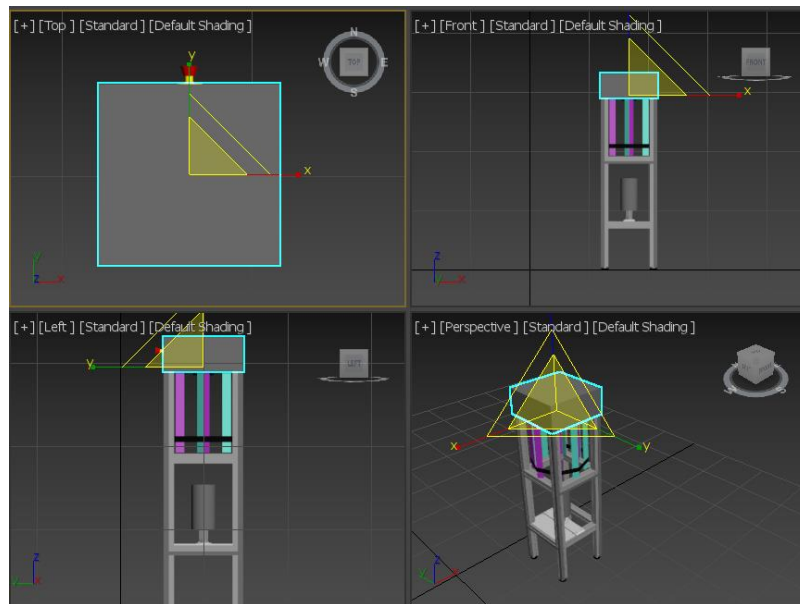


Figura 2.4 - Ventana de visualización de 3DS MAX.

Panel de comandos

Mediante el panel de comandos, se proporciona acceso a las herramientas necesarias para la creación y modificación de la geometría del modelo. Entre estas herramientas destacan geometrías básicas en 3D y 2D, como cubos, esferas, rectángulos; herramientas para unir elementos; cambios en la malla, etc. Con todo ello, es posible crear geometrías tan complejas como el usuario necesite.

Además, permite el acceso a las funciones más comunes a la hora de diseñar, que igualmente el usuario puede editar a su gusto. Así como, a la creación de los materiales y texturas necesarias para la escena.

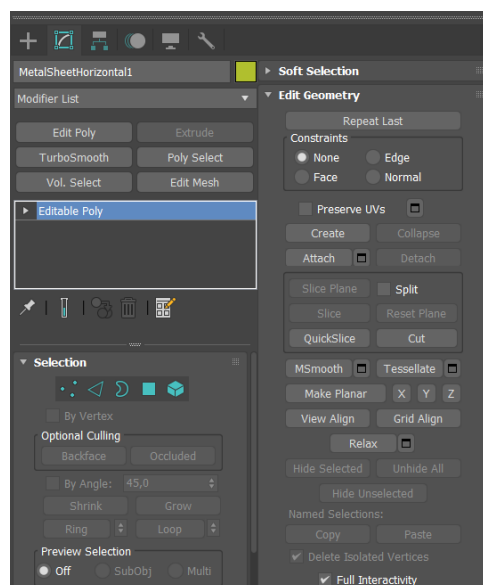


Figura 2.5 - Panel de comandos de 3DS MAX.

Controles de barra de estado

En cuanto a los controles de barra de estado, permite tener acceso a las coordenadas globales o locales de cada elemento que componen la escena, pudiéndose modificar numéricamente desde esta barra. Además, cuentan con las herramientas necesarias para la creación de animaciones y el renderizado.

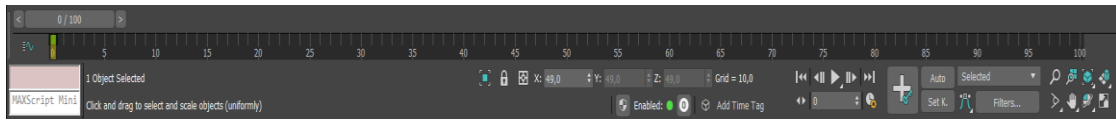


Figura 2.6 - Controles de barra de estado de 3DS MAX.

2.2.2. Integración de 3DS Max con Unity 3D

Una vez los modelos 3D estén completamente diseñado, es momento de exportarlo al programa Unity 3D, donde se pasará a programar el comportamiento de cada uno de los modelos. Para ello, en este apartado, se explicará cuáles son los pasos necesarios para llevarlo a cabo.

En este caso, el tipo de archivo que se emplean tienen un formato FBX. Los archivos FBX son uno de los más usados a la hora de exportar dibujos 2D o 3D, ya que mantiene intacto la jerarquía del modelo, pudiendo diferenciar cada uno de los elementos que lo componen en cualquier programa al que se exporten, de igual manera ocurre con los materiales asignados a cada uno de dichos elementos. [13]

El proceso para exportar archivos en formato FBX es bastante sencillo. Tal y como se muestra en la figura xx lo único que se debe hacer es hacer clic en *Export*, lo que abrirá una nueva ventana donde se guardará el archivo con un nombre identificativo y seleccionando FBX en el tipo de formato en el que se guardará.

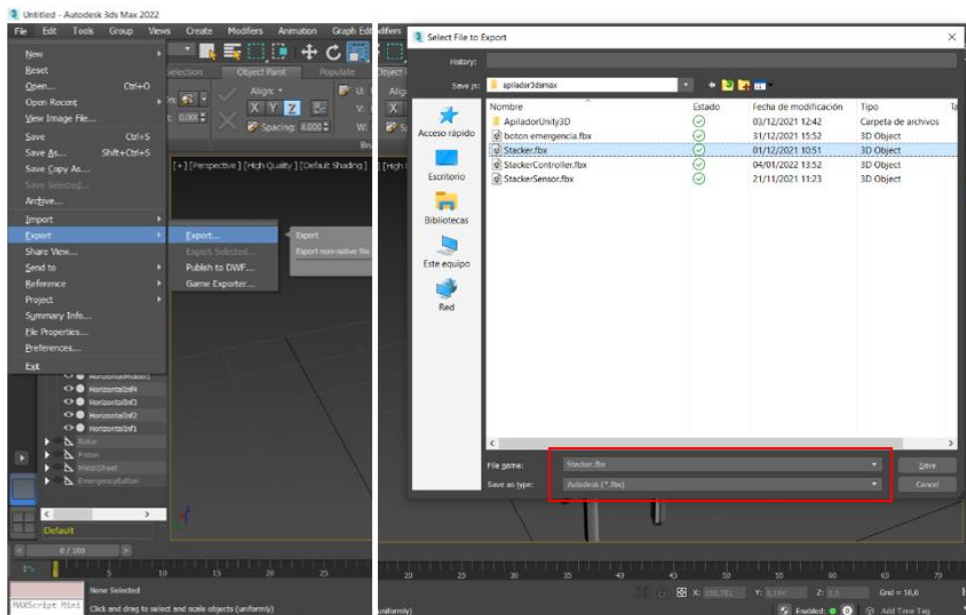


Figura 2.7 - Exportar archivos en formato FBX.

Una vez se tiene guardado el archivo, el último paso es agregar dicho archivo en la carpeta *Project* de Unity, descrita en el apartado 2.2. En este momento el proyecto en Unity, ya cuenta con el archivo del modelo 3D. Por lo que ahora solo se tendrá que arrastrar el archivo en la escena correspondiente, a través de la pestaña *Hierarchy*.

El principal problema que presenta esta exportación de archivos es la pérdida de las texturas de los elementos. El problema procede de que las texturas es una serie de parámetros que se entienden de manera distinta tanto en Unity como en 3DS Max y otros softwares. Esto conlleva a una pérdida de realismo al verse incrementada su apariencia y comportamiento con la luz. Ante este problema existen varias soluciones, como por ejemplo el empleo de texturas PBR (*Physical Based Rendering*). Sin embargo, esto conlleva a que los proyectos en Unity 3D sean más pesados y una pérdida de rendimiento del mismo. Es por ello, que se decidió no incluir las texturas, ya que la apariencia de los elementos era bastante buena y para este trabajo no se requería que los modelos fuesen hiperrealistas.

Además, otro problema que surge al exportar los archivos desde 3DS Max a Unity 3D, viene relacionado con la orientación y dirección de los ejes de cada uno de los objetos importados, ya que estos pueden sufrir modificaciones y una vez definidos el programa Unity no permite modificarlos. Aunque existen alternativas para poder obtener la dirección y sentidos de los ejes, tal y como se explica en [21] donde se crea un nuevo `GameObject` vacío, como padre del objeto del que se desea modificar sus ejes.

2.2 Unity 3D

El segundo paso de este proyecto consiste en dotar de comportamientos físicos a los objetos, realizar el renderizado, crear animaciones, etc. Por ello, es necesario utilizar un motor gráfico. En el mercado existen numerosos editores de texto como Unreal Engine o CryENGINE, pero para este trabajo se optó por motor gráfico Unity 3D.

Lo primero que se debe saber es que Unity es un motor gráfico que es muy empleado en la industria del desarrollo de videojuegos. La principal ventaja que ofrece Unity es facilitar y mejorar el desarrollo de videojuegos por medio de una plataforma que evita que sus usuarios tengan que crear y mantener su propia plataforma de desarrollo [4]. Además, cuenta con una gran comunidad de usuarios, lo que permite poder acceder a una gran cantidad de documentación, foros y comunidades donde resolver problemas y buscar ideas para desarrollar los proyectos.

Unity es una plataforma de desarrollo, que es muy empleada en la industria de los videojuegos, ya que incorpora funcionalidades tales como motor gráfico para renderizar gráficos, motor físico para simular las leyes de la física, etc. [5]. Por otra parte, otra característica por la que es tan interesante es la centralización de todo lo necesario para el desarrollo de videojuegos, ya que permite crearlos para distintas plataformas como videoconsolas, PC, dispositivos móviles, etc. Todo esto se consigue mediante un editor visual y programación vía scripting basada en lenguaje C#.

Además de los motores y animaciones que incluye el software de Unity, incorpora grandes herramientas que permiten trabajar de forma remota, con inteligencia artificial o soporte para la implementación de la realidad virtual, la cual tiene un papel importante en este trabajo.

En los siguientes apartados, se van a describir la interfaz del programa Unity. Se hará una breve descripción de los elementos empleados para este proyecto tanto del menú de Unity Hub, como de la ventana de trabajo de Unity.

2.2.1 Unity Hub

Esta aplicación de escritorio de Unity permite al usuario poder descargar y gestionar de forma ordenada tanto todas las versiones de Unity que se tengan instaladas, como cada uno de los proyectos que se han realizado. De esta manera, el usuario del programa puede acceder a toda esta información de una manera fácil y ordenada. Además de esto, la aplicación también permite la gestión de licencias, unirse a la comunidad, aprender por medio de tutoriales, y la creación de nuevos proyectos.

Para crear un nuevo proyecto, lo único que se deberá hacer es pulsar el botón *New* situado arriba a la derecha, donde se ofrecen varias opciones para los proyectos, como formato 3D, 2D, para dispositivos móviles, realidad aumentada, etc. De la misma manera podremos añadir proyectos que han sido realizados en otros ordenadores, para lo que simplemente hay que pulsar *Add* y seleccionar el archivo deseado.

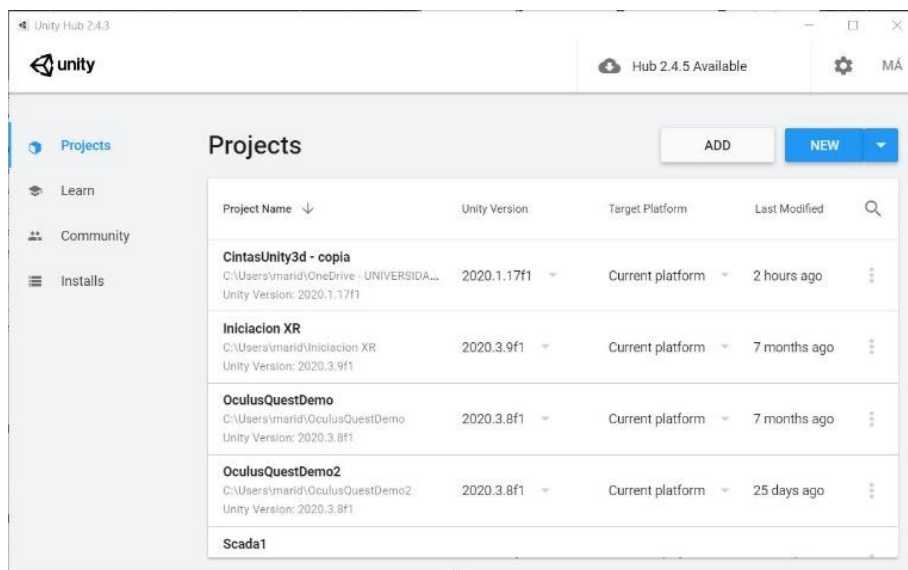


Figura 2.8 - Interfaz de Unity Hub

En la figura 2.8, se puede ver la interfaz de Unity Hub, donde en la parte izquierda se localizan cuatro pestañas. La primera de ellas es *Projects*, donde se encuentran cada uno de los proyectos realizados en Unity. En segundo lugar, tenemos *Learn* y *Comunidad*, en las cuales se encuentran tutoriales y proyectos para poder iniciarse en el uso de este programa o seguir aprendiendo, al igual que un foro donde plantear y resolver dudas acerca de los proyectos en Unity.

En último lugar, encontramos la pestaña *Installs*, mediante la cual se tiene acceso a todas las versiones descargadas de Unity. Además, a cada una de estas versiones se le podrán instalar los módulos que sean necesarios para los proyectos. En el caso particular de este trabajo, donde se va a implementar un sistema de realidad virtual, es necesario añadir dos módulos. Como se puede observar en la figura 2.9, los módulos necesarios para la realidad virtual son los correspondientes para el soporte de dispositivos Android, ya que las gafas Oculus Quest 2 llevan instaladas este sistema operativo.

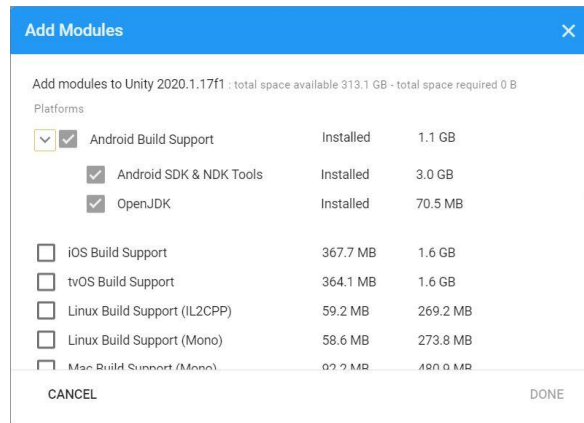


Figura 2.9 - Ventana Add Modules de Unity Hub.

2.2.2. Interfaz de Unity

Una vez creado el nuevo proyecto, aparecerá la interfaz de Unity, como se ve en la figura 2.10, donde se realizará todo el proceso de desarrollo. Antes de comenzar a trabajar con esta herramienta, lo primero que se debe de conocer es la disposición de los distintos elementos que aparecen y la función que cada uno de estos desempeñan.

Si se comienza por la parte superior de la ventana, encontramos la barra de herramientas (A) donde aparecen una serie de opciones muy utilizadas durante el desarrollo de los proyectos. Debajo, se encuentra una serie de ventanas que son *Hierarchy* (B) donde aparecerán todos los objetos que se creen para el proyecto, con su jerarquía dentro de la escena. A su lado, aparece la ventana *Scene* (C) en la que se puede visualizar, editar y navegar por la simulación. Después, encontramos la ventana *Game* (D) donde se ofrece una visión del juego final. A la izquierda, se encuentra la ventana *Inspector* (E) donde se puede ver todas las características y elementos que componen cada objeto. Por último, en la parte inferior, encontramos la pestaña *Project* (F) donde aparecen todos los archivos que son necesarios para el proyecto, y la ventana *Console* (G) en la que aparecen mensajes de errores, advertencias, etc.

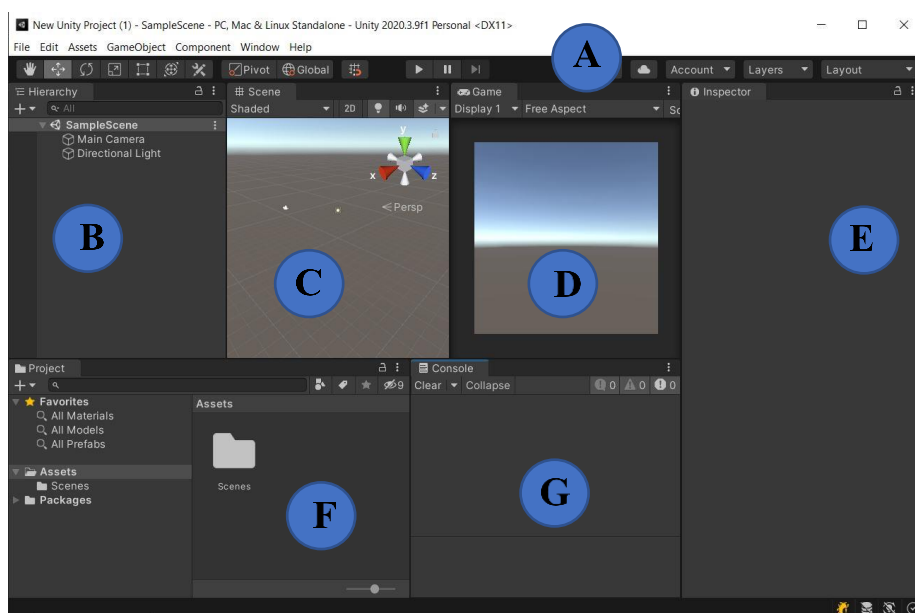


Figura 2.10 - Interfaz de trabajo de Unity.

A pesar de que la configuración de la figura 2.10 es la que viene por defecto, Unity permite hacer modificaciones para que se adapte al usuario y le sea más fácil el uso del programa. Las modificaciones consisten en la reubicación de cada una de las ventanas, así como la unificación de dos ventanas en un mismo espacio, de manera que aparezcan como pestañas y se puedan ir alternando según sean las necesidades de cada momento. A continuación, se explica con más detalle cada una de las ventanas previamente citadas.

Barra de Herramientas

En la parte superior de la interfaz de trabajo, se encuentra la barra de herramientas. En primer lugar, encontramos una serie de pestañas situadas en la parte superior izquierda. Estas pestañas, como *File* o *Edit* permiten configurar varios aspectos relacionado con el proyecto. En el caso particular de este trabajo, estas pestañas serán importante a la hora de configurar la realidad virtual.



Figura 2.11 - Barra de Herramientas.

Además, se encuentran otra serie de funciones que permiten manipular los objetos, muy usadas durante la elaboración de proyectos en Unity. Estas funciones además de moverse por la escena son las de trasladar, rotar y escalar un objeto, incluso la posibilidad de hacer estas tres acciones a la vez. Estas acciones disponen de accesos rápidos mediante las teclas *W*, *E* y *R* respectivamente.

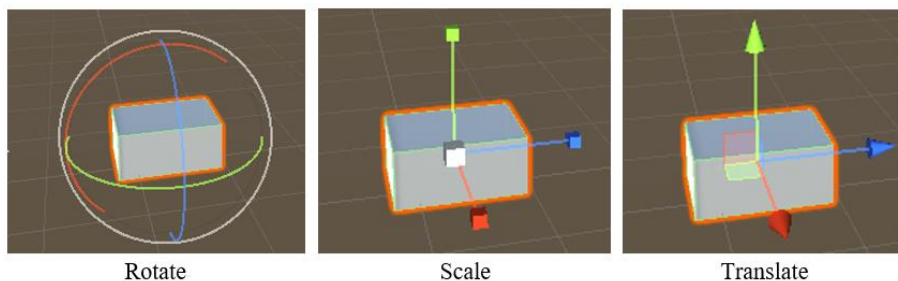


Figura 2.12 - Gizmos Rotate, Scale y Translate.

En la parte central, encontramos los botones *Play*, *Pause* y *Step* necesarios para la reproducción de la escena creada. Y en la parte derecha, se encuentran una serie de herramientas, en las que destaca la herramienta “*Collaborate*”, que permite sincronizar y compartir proyectos para poder facilitar el trabajo en grupo sin importar la ubicación de las personas que lo componen.

Ventana Hierarchy

La ventana *Hierarchy*, como se ve en la figura 2.13, se muestran todos los *GameObjects*, es decir, todos los elementos presentes en la escena desde personajes y objetos coleccionables hasta luces, cámaras y efectos especiales [6] que componen la escena del proyecto.

Además, la lista de *GameObjects* muestra la jerarquía de la escena. Mostrando la relación que existe entre cada uno de ellos, apareciendo el concepto de “parentesco”. Este concepto hace

referencia a la creación de un *GameObject* llamado “padre” y crear otros *GameObjects* llamados “hijos”, los cuales se encontrarán agrupados dentro del padre. Esto lo que implica es que cada uno de los hijos heredará todas las características del padre, además de poder tener más características propias.

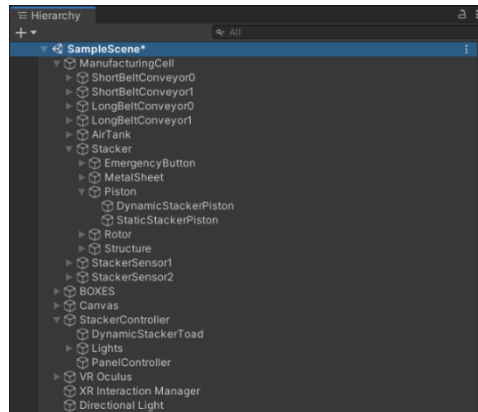


Figura 2.13 - Ventana Hierarchy.

Por último, al seleccionar cualquier *GameObject* dentro de esta ventana, implicará que este quedará remarcado en la escena, destacando de los demás, y que, en la ventana *Inspector* aparecerán todas sus características.

Ventana Scene

Esta ventana permite al usuario visualizar y editar la escena. En ella, se puede seleccionar cualquiera de los objetos creados y editar cada una de sus características. Del mismo modo que sucede cuando se selecciona un objeto en la ventana *Hierarchy*, cuando se hace en esta ventana, se obtienen todas sus características en la ventana *Instructor*.

Al igual que en 3DS Max, otro elemento importante presente en esta ventana es el *gizmo*, que se encuentra situado en la parte superior de ventana, tal y como se puede ver en la figura 2.14. Esta herramienta proporciona al usuario la orientación de la vista actual dentro de la escena, además de poder cambiarla. La orientación en la escena viene dada por los ejes de coordenadas cartesianas X, Y y Z. Además, haciendo clic con el botón derecho sobre el *gizmo*, permite elegir una de las vistas predeterminadas, tales como vista superior, laterales, inferior, alzado, etc.

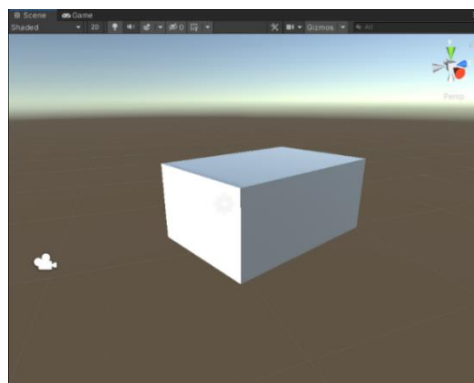


Figura 2.14 - Ventana Scene.

Para poder moverse por la escena existen varias alternativas. Por un lado, se puede realizar las flechas del teclado para movernos hacia delante, hacia atrás y hacia los lados, según la dirección en la que se esté mirando. Por otro lado, si se utiliza el ratón, se puede hacer zoom usando la rueda, moverse libremente al dejar pulsada la rueda y mover el ratón, o recorrer en primera persona la escena manteniendo pulsado el botón derecho.

Ventana Game

Esta ventana muestra al usuario cual será la visión final que se tendrá del juego, ya que es la visión que ofrece cada una de las cámaras que han sido configuradas para la escena. En el caso de contar con más de una cámara, se puede acceder a cada una de ella mediante la pestaña *Display*, situada en la parte superior, donde podremos seleccionar la deseada.

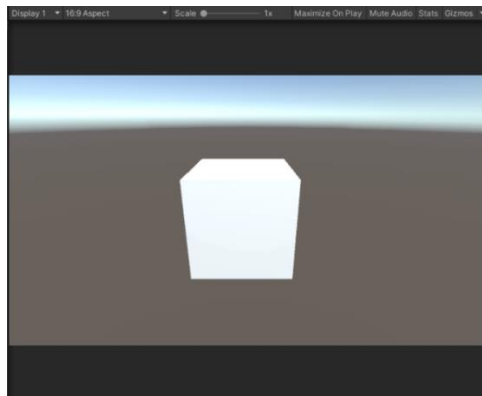


Figura 2.15 - Ventana Game.

Siguiendo en la parte superior, junto a la pestaña *Display*, aparece una pestaña que permite ajustar la imagen al monitor donde vaya a ser visualizado. Seguidamente, se encuentra *Scale* que permite acercarse o alejarse para ver más o menos detalles presentes en la pantalla de visualización del juego, con el objetivo de adaptarse a dispositivos con distintas resoluciones.

También, se cuenta con la opción *Maximize on Play* que activa la función de ventana completa cuando se active el modo de reproducción. De igual modo, *Mute Audio* permite activar o desactivar el audio de la escena cuando se encuentre activado el modo reproducción.

Si se pulsa *Stats*, se desplegará una pequeña ventana, tal y como se muestra en la figura 2.16, donde se mostrarán estadísticas relacionadas con la reproducción de la escena. Estas características son correspondientes al audio y gráficos en tiempo real durante la reproducción.

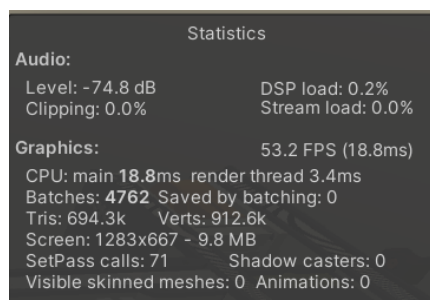


Figura 2.16 - Ventana Stats.

Ventana Instructor

En la ventana *Inspector*, como se puede ver en la figura 2.17, permite ver y editar todas las características del *GameObject* que se ha seleccionado. En primer lugar, en la parte superior, aparece el nombre, así como alguna etiqueta que le haya sido asignada o asignarle a una capa concreta, y también, permite desactivar y activar el *GameObject* dentro de la escena.

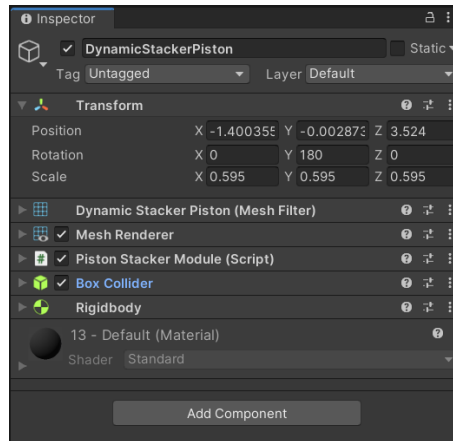


Figura 2.17 - Ventana Instructor.

Las características que pueden aparecer son muy variadas y dependerá en parte del tipo de *GameObject* que se seleccione. Sin embargo, una de las características principales y que suelen tener todos los *GameObjects* es la pestaña *Transform*, donde podemos ajustar de manera precisa la posición, rotación y escala en cada uno de los ejes. Entre las características que se pueden añadir, destacan como scripts, propiedades físicas, materiales, ajustes de cámara e iluminación, etc.

Cuando se quiera añadir un nuevo componente al objeto, como puede ser alguno de los recientemente citados, existen dos alternativas. En primer lugar, habría que arrastrar la característica deseada dentro de esta ventana. Y, en segundo lugar, sería seleccionar la opción *Add component* situada en la parte inferior de la venta y añadir el componente deseado.

Ventana Project

En esta venta, figura 2.18, se podrán encontrar todos los archivos que se emplean en el proyecto, archivos tales como modelos 3D/2D, materiales, scripts, etc. Estos archivos se ordenan mediante carpetas y temática, del mismo modo que lo hace el explorador de archivos de un ordenador. Cuando seleccionamos una de las carpetas en la parte izquierda, en la ventana derecha, aparecen todos los archivos que se encuentran dentro de esta carpeta. Estos archivos pueden ser arrastrados y añadidos a los *GameObjects* de la escena.

Para que sea más cómodo, Unity incorpora, entre otras cosas, una barra de búsqueda o la posibilidad de aumentar o disminuir el tamaño del icono de los archivos, según las necesidades del usuario.

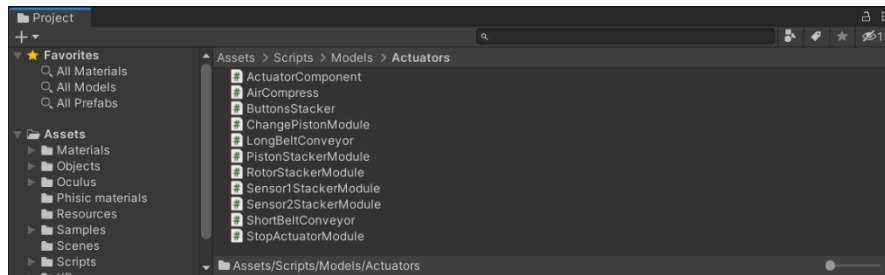


Figura 2.18 - Ventana Project.

Ventana Console

Esta ventana, como se ve en la figura 2.19, sirve para mostrar mensajes relevantes durante la reproducción. Dentro de los mensajes se pueden distinguir tres variantes. Por un lado, encontramos los mensajes de error (señalados por medio de un octógono rojo); los errores de advertencia (señalados mediante un triángulo amarillo); y, por último, los mensajes generados (señalados con un bocadillo blanco) que muestra información que el programador desea. Una característica que hace a esta ventana muy útil es que haciendo clic en el mensaje te muestra el *GameObject* afectado, además de una breve descripción del problema.

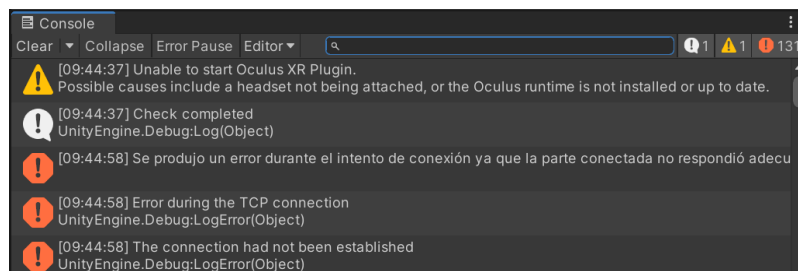


Figura 2.19 - Ventana Console.

En la parte superior encontramos varias pestañas con distintas funciones. De estas, destacamos la pestaña *Clear* que borra todos los mensajes de la consola para dejarla vacía. También, está la pestaña *Collapse* que elimina los mensajes repetidos; la pestaña *Error Pause* que, al activarla, en caso de aparecer un error durante la simulación esta se detiene. La pestaña *Editor* permite, que, al conectarse a un puerto remoto, se muestren el registro de Unity *Player* local en lugar del registro de compilación remota. La barra de búsqueda permite buscar entre los mensajes con una palabra específica. Y, por último, en la parte superior derecha, están los tres que muestran el número de mensajes, advertencias y errores detectados respectivamente y si se pulsan, actúa como filtro para mostrar solamente el tipo de mensaje elegido.

2.3. Visual Studio Code

En este proyecto, como se cometerá más adelante, para poder recrear los movimientos y comportamientos de los objetos que intervienen es necesario configurarlos mediante el uso de programación. Para lo que es necesario disponer de un editor de texto. En el mercado existen numerosos editores de texto como Notepad++, Atom, Vim o TextMate. Sin embargo, en este trabajo se optó por el programa Visual Studio Code.

Visual Studio Code (VS Code) es un editor de texto gratuito de código abierto diseñado por Microsoft que se encuentra disponible para Windows, Linux y macOS. Esta herramienta permite editar, soporte para la depuración entre otros, y a su vez es compatible con múltiples lenguajes de programación, tales como C++, C#, Visual Basic.NET, Java y Python entre otros [7]. Por defecto, Unity como herramienta Visual Studio Code, sin embargo, se podría elegir otro editor en la opción *External Tools* dentro de las preferencias de Unity.

Entre las principales ventajas que proporciona este programa, destaca en primer lugar, la multicompatibilidad con respecto a todos los lenguajes. También, dispone de múltiples complementos que permiten trabajar mejor y más rápido, permitiendo, entre otras capacidades, la integración con el control de versiones. Y, por último, destaca la optimización del propio programa.

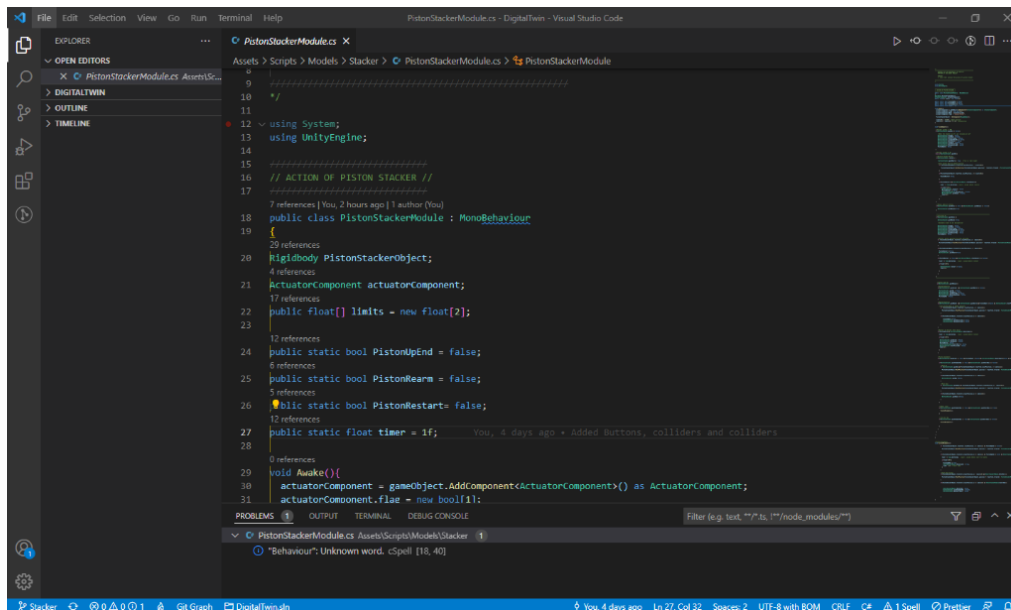


Figura 2.20 - Interfaz de Visual Studio Code.

En la figura 2.20, se muestra la interfaz de trabajo de Visual Studio Code. Como se puede ver es una interfaz muy sencilla, donde a la derecha vemos un explorador de archivos donde están todos los *scripts*, en la parte central, la ventana donde se crea y edita el código. Y en la parte inferior, encontraríamos la ventana donde se muestran distintos mensajes de aviso, de igual forma que funciona la consola de la interfaz de Unity. También, da la posibilidad de añadir extensiones que permitan nuevas funciones dentro del programa.

2.3.1. Integración de VS Code con Unity 3D

El comportamiento de cada uno de los *GameObjects* es controlado por medio de *scripts*. En el caso de Unity, el lenguaje empleado para realizar estos *scripts* es C#, un lenguaje orientado a objetos que cuenta con una sintaxis sencilla, muy parecida a Java o C++ lo que facilita la tarea al desarrollador [11].

Para poder asignar un script a un *GameObject*, debemos añadirlo en la ventana *Instructor*, y para ello, existen dos alternativas. Por un lado, se puede añadir como un elemento más haciendo clic en el botón situado en la parte inferior *Add Component* y seleccionar la opción *New script*. Por

otro lado, si se tiene ya creado el *script* y guardado en las carpetas del proyecto, se puede arrastrar desde la ventana *Project* hasta la ventana *Instructor* del *GameObject*.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Robot : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10     }
11
12     // Update is called once per frame
13     void Update()
14     {
15     }
16
17
18 }
```

Figura 2.21 - Estructura general de un Script.

Una vez creado el *script* y adjuntado al *GameObject* correspondiente, el siguiente paso será elaborar las líneas de código que lo compondrán. Cuando se hace clic en el icono del *script*, se abre directamente el programa VS Code y aparecerá la estructura del *script* que se puede ver en la figura 2.21. En primer lugar, en el parte superior precedido por el comando *using* encontramos las librerías empleadas para ese *script*, que podrán variar de uno a otro. A continuación, encontramos la clase base *MonoBehaviour* de la que deriva cada *script* de Unity, usándola para controlar mejor el acceso a los elementos, de forma que no lo tenga que hacer el propio usuario, la cual deberá tener el nombre del *script*. [12].

Por defecto, el *script* creará dos funciones principales, la función *Start* y *Update*. La función *Start* solo se llamará una vez y será antes de que la simulación comience, por lo que abarcará todo lo relacionado con la inicialización. Por otro lado, la función *Update* se ejecutará continuamente de forma cíclica, es especialmente útil para poder implementar acciones en respuesta a entradas. Aunque estas son las funciones más usadas, existen una variedad de funciones que tienen variaciones como *Awake* o *FixedUpdate*.

Otro aspecto importante son las variables, y es que encontramos dos tipos de variables a la hora de programar, públicas y privadas. Por un lado, variables públicas son las que pueden llegar a ser modificadas en cualquier momento a través la ventana *Inspector* y podrán ser llamadas en otros *scripts*. Y las variables privadas solo podrán ser modificadas y llamadas dentro del mismo *script*.

Una vez explicado los programas utilizados para poder desarrollar el gemelo digital, en el siguiente capítulo, se explicará cuál es el funcionamiento del servidor de bandejas, el proceso de modelación 3D y su integración en Unity 3D.

3 DESCRIPCIÓN DE LA PLANTA

Lo primero que se debe saber es que el alimentador de bandejas se encuentra instalado dentro de una célula de fabricación flexible, donde su función es gestionar el número de bandejas que están presentes en cada instante del proceso.

Hay que recalcar que este trabajo forma parte de un proyecto mayor cuyo objetivo es la realización de un gemelo digital de la planta completa. Es por ello, que este trabajo parte de otros previos donde se elaboró el sistema de cintas encargado del transporte de las bandejas [14][15]. Además, en trabajos futuros, se deberá incluir los elementos restantes que están presentes durante en la planta.

3.1 Célula de fabricación flexible

La célula de fabricación flexible está compuesta por un circuito formado por cuatro cintas transportadoras, dispuestas de manera que forman un cuadrado, que permiten trasladar bandejas que a su vez pueden portar pallets con determinadas piezas, de modo que éstas puedan ser tratadas por otras máquinas en distintos puntos de trabajo alrededor del circuito. Para poder conseguir esto, además cuenta con cuatro pistones de cambio, situados en cada una de las esquinas con los que se llevan a cabo los cambios de una cinta a otra. También, se encuentran otros dispositivos adicionales, como son los pistones de paro, los sensores inductivos o los sensores de fuerza, elementos que son necesarios para un correcto comportamiento de la planta [14].

Los tratamientos de las piezas en los puntos de trabajo pueden ser muy variados y abarcan desde operaciones de ensamblaje, clasificación, posicionado, hasta la gestión del control de calidad de las operaciones previas. En la siguiente imagen, se puede observar un esquema en el que se muestra la disposición de todos los elementos presentes en la planta.

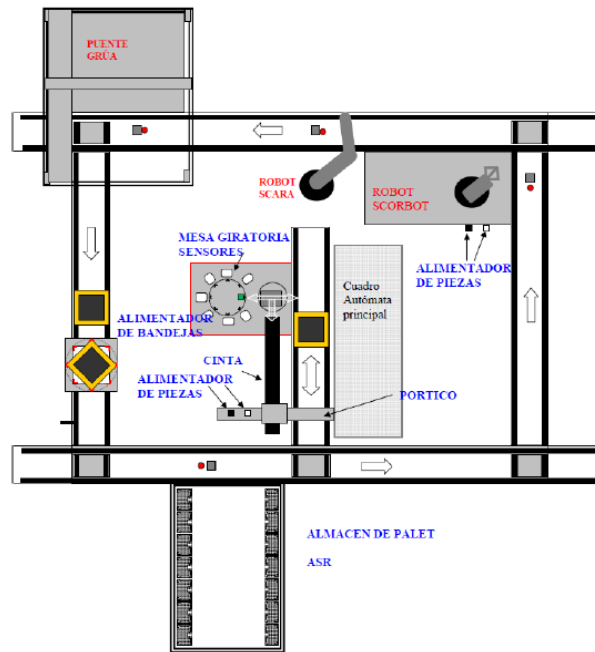


Figura 3.1 - Esquema de la célula flexible de fabricación [22].

El calificativo de célula flexible hace referencia a la capacidad que tiene la planta de adaptarse a distintas demandas de producción en función de las necesidades. De forma que no se pretende tener una programación como un único ciclo repetitivo y fijo, sino como un conjunto de programas modulares que serán gestionados de forma coordinada para la consecución de los objetivos perseguidos en un determinado momento.

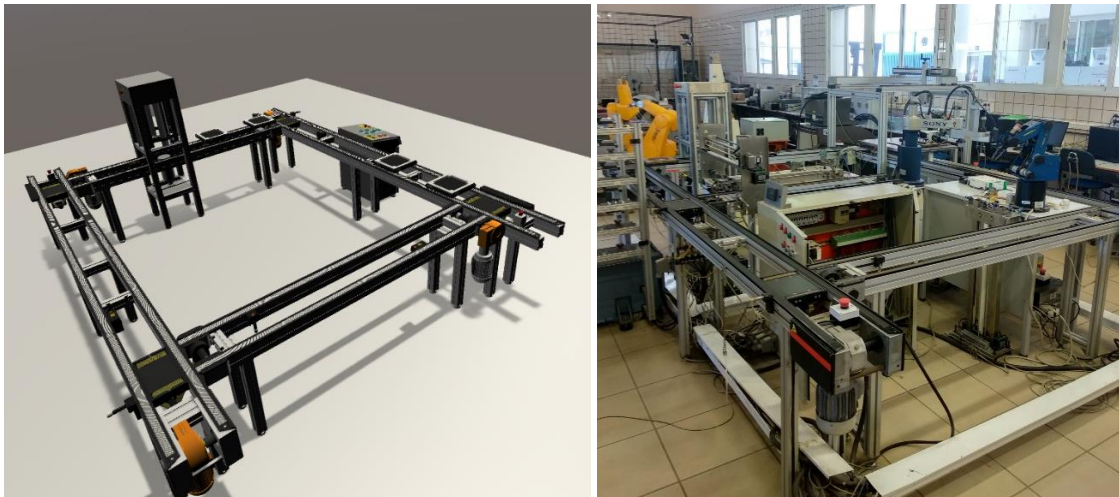


Figura 3.2 - Comparativa entre el modelo 3D y la célula de fabricación flexible real.

En la figura 3.2, a la izquierda, se puede apreciar los elementos que ya han sido modelados de la planta, que como se comentó anteriormente, son las cintas transportadoras y el alimentador de bandejas. También, se puede apreciar la gran similitud que presentan todos los elementos, con la idea de asemejar los más posible el gemelo digital a la realidad.

3.2 Alimentador de bandejas

Una vez descrito el contexto de trabajo, se pasa a explicar el objeto de este proyecto, el desarrollo del gemelo digital del alimentado de bandejas. En la figura 3.3, se puede ver un esquema que representa todos los elementos con los que cuenta este alimentador.

Entre los elementos, por un lado, se cuenta con una estructura principal, donde se sitúan tanto el pistón de elevación como el rotor de giro, que se encargan de todo el proceso de subir/bajar y almacenar las bandejas. Por otro lado, se disponen de unos elementos adicionales, como son sensores, indicadores led, retenedores para posicionar de forma correcta las cajas, un sensor de barrido que detecte si las bandejas se encuentran vacías o portan algún elemento, y un puesto de mando donde controlar todo el proceso, que son necesarios para que todo el proceso se desarrolle de manera exitosa. Todos estos elementos vienen nombrados y representados en la siguiente figura.

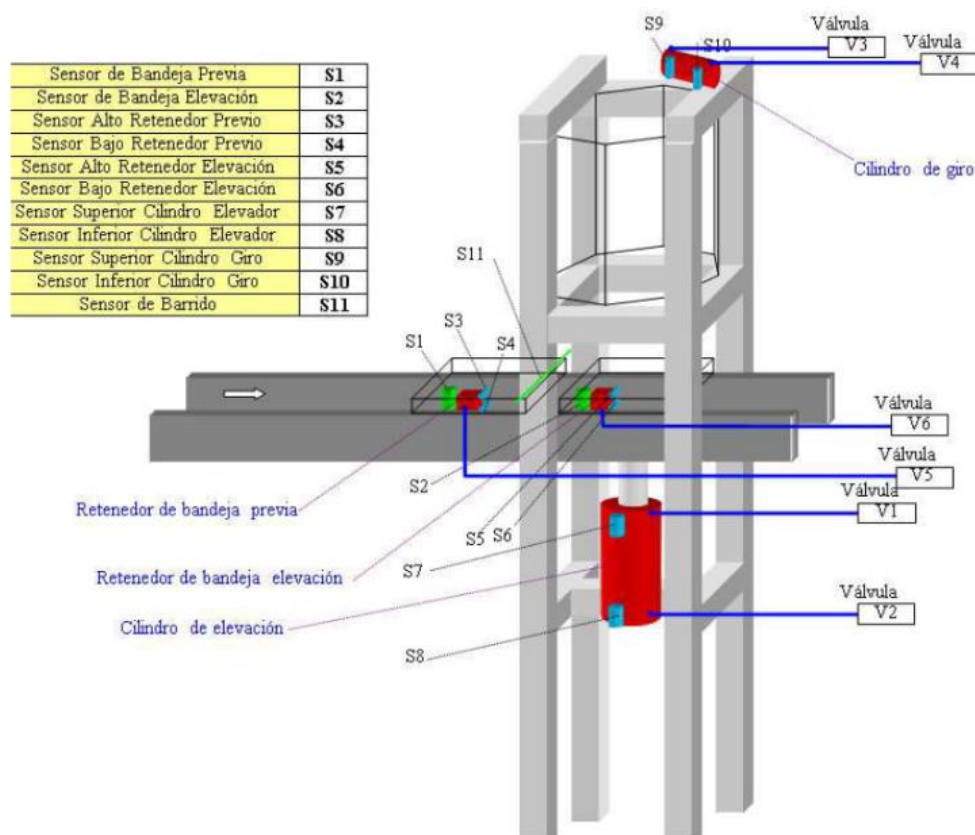


Figura 3.3 - Esquema del alimentador de bandejas [22].

La función principal del apilador es la de depositar y recolectar las bandejas que circulan por las cintas, según sean las necesidades que demande el sistema en cada instante. Para conseguir almacenar las bandejas, cuenta con un mecanismo principal sencillo. Este mecanismo consta de un pistón que elevará la bandeja hasta la altura de la estructura donde se guardarán; y a dicha altura se encuentra un cilindro de giro que hará rotar las cajas 45 grados. La finalidad de este giro es que al final del giro la bandeja quede soportada sobre los listones horizontales que conforman parte de la estructura, tal y como vemos en la figura 3.4. En esta figura vemos a la derecha la caja aún sin girar y a la izquierda una vez está ya ha sido girada.

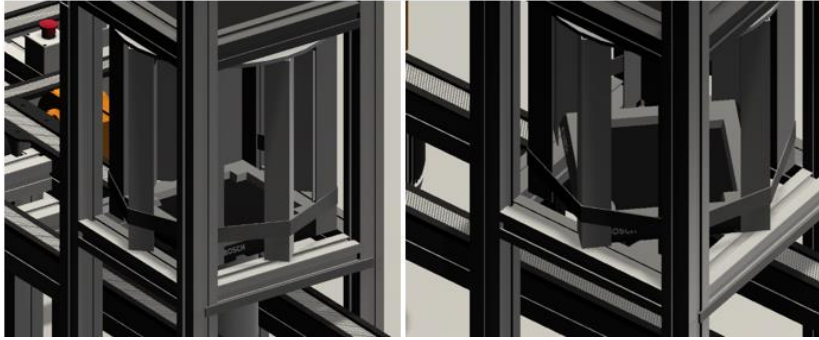


Figura 3.4 - Proceso de giro para el guardado de cajas.

En los siguientes apartados, se mostrarán los modelos 3D realizados, con ayuda del programa 3DS MAX, y se explicarán las funciones de cada uno de los elementos del sistema, recientemente nombrados. Además, se verá cómo ha sido el proceso de integración en el programa Unity 3D, dónde se dota al sistema de su comportamiento físico que tiene en la realidad, así como todas las acciones que se pueden realizar desde el cuadro de mandos.

4 DESARROLLO DEL GEMELO DIGITAL

Como ya se ha comentado con anterioridad, el primer paso es la recreación en formato digital de todos y cada uno de los elementos presentes en el proceso. Este será el punto de partida a través del cual se construirá todo el gemelo digital del sistema de apilación de cajas.

4.1 Elaboración de los modelos 3D

4.1.1 Estructura del alimentador de bandejas

La estructura del alimentador puede considerarse como el elemento principal del sistema de alimentación de bandejas de la planta y se encuentra conformada por tres elementos diferenciados. Todo el sistema se puede observar en la figura 4.1.



Figura 4.1 - Comparativa entre el modelo 3D y el real del alimentador de bandejas.

En primer lugar, encontramos los elementos estructurales, que además de su función estructural, servirán para almacenar las cajas como se indicó anteriormente. Esta estructura está compuesta por cuatro pilares, situados en las esquinas y tres conjuntos de barras horizontales, situados a distintas alturas del apilador, que le aporta rigidez a la estructura. Además, cada pilar cuenta con una chapa metálica, situada en la arista exterior y a lo largo del pilar con el fin de proteger la estructura. También, en la parte superior, cuenta con una cubierta metálica con el fin de almacenar dentro todos los componentes necesarios para el funcionamiento del apilador, dotándolo de una mejor apariencia visual y mayor protección.

En segundo lugar, se encuentra el pistón encargado de elevar y descender las bandejas, situado en la parte inferior de la estructura. El pistón, se encuentra apoyado sobre una plataforma metálica, que está atornillada al conjunto de barras horizontales que se encuentran en la parte inferior. Para que el pistón pueda elevarse, necesita de aire comprimido, por lo que se encuentra conectado a la red de aire comprimido de la planta.

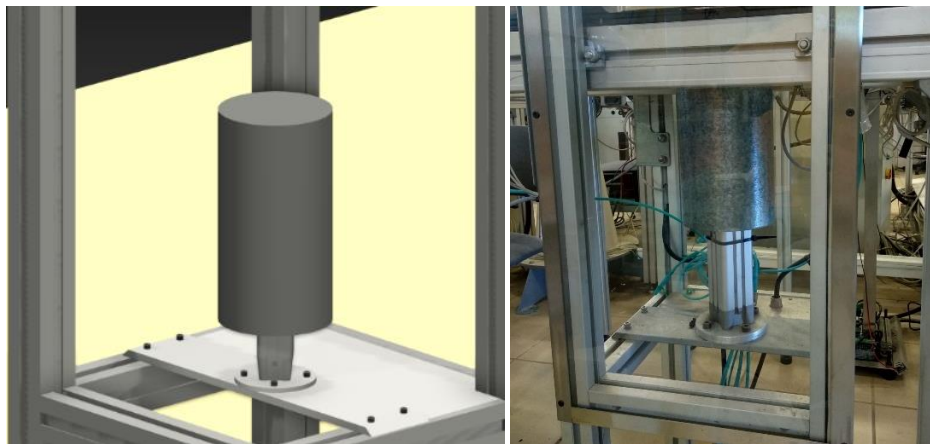


Figura 4.2 - Comparativa entre el modelo 3D y pistón de elevación real.

Y, por último, está el cilindro de giro, encargado de hacer girar a las cajas y que se encuentra situado en la parte superior. El cilindro de giro está compuesto por un motor eléctrico, que se encarga de hacer girar un disco. A este disco se le han fijado ocho listones en forma de L, que se encuentran dispuesto en los vértices de un octógono imaginario, con el fin de que el proceso descrito anteriormente se pueda llevar a cabo.



Figura 4.3 - Comparativa entre el modelo 3D y el cilindro de giro real.

4.1.2 Retenedores de bandejas

Otro elemento presente son los retenedores de bandejas, que se pueden ver en la figura 4.4. El sistema de funcionamiento de este dispositivo consiste en un pequeño cilindro metálico, situado en la parte central del soporte, con un grado de libertad, el cual le permite elevar y descender su posición. Para ello, cuenta con un sistema de aire comprimido que le permite realizar dichos movimientos, por lo que al igual que el pistón se encuentran conectados a la red de aire comprimido de la planta. Además, cuenta con un sensor inductivo, montado en un soporte adjunto al retenedor, que detecta una pequeña chapa metálica situada en la parte inferior de las bandejas. Este sensor, se emplea para poder automatizar el proceso de servir y almacenar bandejas.

En la planta, se disponen de dos retenedores como se puede ver en la figura 3.3. Uno de ellos se encuentra a la altura de la estructura principal, con el objetivo de que las bandejas se encuentran en la posición precisa para que, al ser elevada por el pistón, esta no golpee la estructura y se pueda almacenar de manera correcta en la parte superior del apilador. El segundo retenedor, está situado de igual forma sobre la cinta, pero en una posición anterior a la del apilador, con el objetivo de hacer parar a la bandeja posterior mientras el apilador se encuentra en funcionamiento.

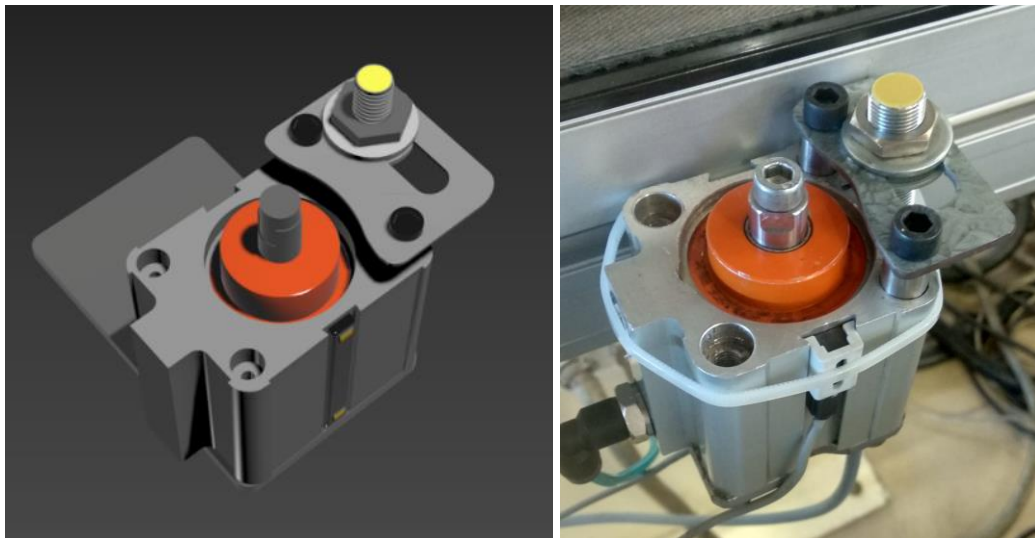


Figura 4.4 - Comparativa entre el modelo 3D y el retenedor de bandejas real.

4.1.3 Sensor de proximidad tipo barrera

El siguiente elemento que está presente es el sensor de proximidad tipo barrera, que se encuentra situado justo antes del pistón elevador. La función principal de este sensor es la de evitar que se almacenen cajas que no se encuentren vacías, debido a los problemas que esto podría ocasionar durante el proceso de almacenamiento.

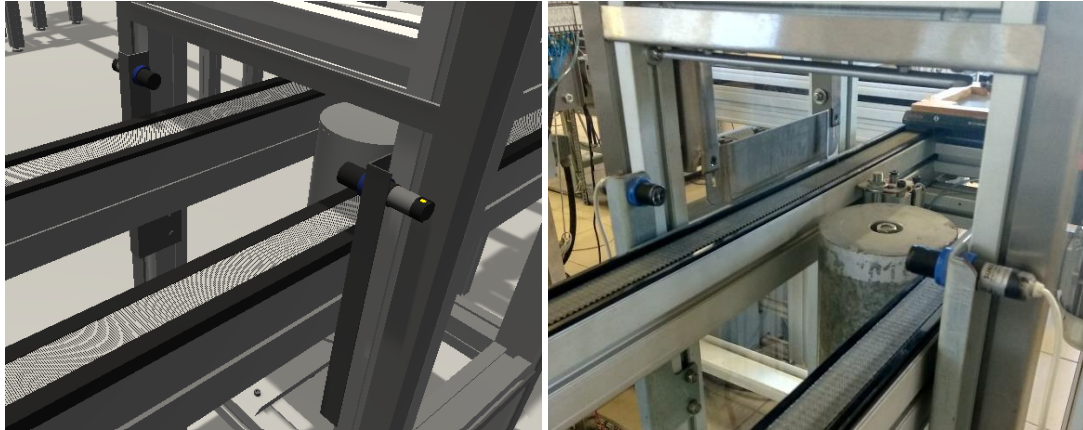


Figura 4.5 - Comparativa entre el modelo 3D y el sensor de proximidad tipo barrera real.

En la figura anterior, se puede ver que el sensor está compuesto por un emisor y un receptor, situadas a ambos lados de la cinta, las cuales se encuentran sujetadas por dos piezas metálicas que se unen a la estructura del alimentador de bandejas. Este sensor, al igual que los sensores inductivos de los retenedores, se emplea para la automatización del proceso.

4.1.4 Cuadro de mandos

El cuarto elemento que componen al sistema es el cuadro de mandos, cuyo objetivo es poder actuar sobre el sistema de alimentación de cajas. Este cuadro de mando está compuesto por una serie de botones, como se puede ver en la figura 4.7, mediante los cuales se realizan una serie de acciones sobre el alimentador de bandejas, que se pasan a explicar a continuación. Además, todos los botones cuentan con un led a su lado, que se enciende para indicar la posición en la que se encuentran cada uno de los elementos móviles. Por su parte, en la parte superior derecha encontramos una serie de indicadores leds, que indican al usuario que los requisitos de seguridad, presión del aire, fuente de alimentación y movimiento de la cinta se cumplen, y, por tanto, todo se encuentra en orden para su funcionamiento. En el caso de que alguno de estos requisitos no se cumpla, el sistema no podrá ponerse en funcionamiento.



Figura 4.6 - Comparativa entre el modelo 3D y el cuadro de mandos real.

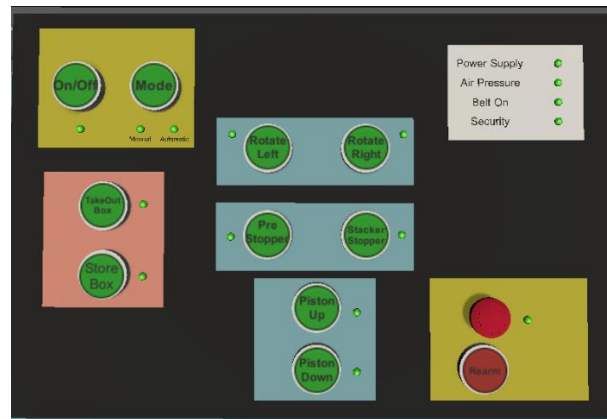


Figura 4.7 - Panel de control para la interacción con el alimentador de bandejas

En primer lugar, en la parte superior derecha, rodeado por una pegatina de color amarillo, se encuentra el botón *On/Off*, que se encarga de encender y apagar el puesto de mando. Justo a su lado, aparece el botón *Mode*, que selecciona el modo de uso del sistema entre un modo manual, que permite al usuario controlarlo mediante el uso de los botones, y un modo automático que automatiza el proceso de servir y almacenar bandejas. En el modelo, para pasar de modo manual a automático, se ha configurado un protocolo para que todos los actuadores vuelvan a su posición inicial, en el que previamente, el pistón y los sensores deben estar en su posición más baja, y el rotor debe estar en una de sus posiciones predeterminadas.

Debajo de los botones anteriores, situados en la parte de color anaranjada, se encuentran los botones *TakeOut Box* y *Store Box*, que solo se encontrarán operativo mientras el sistema se encuentre en modo automático. Ambos botones tienen funciones parecidas, ya que el procedimiento de almacenado y servir bandejas es similar. Este procedimiento está descrito por una serie de acciones combinadas que se describe en la máquina de estados que se muestra en el anexo II.

En la parte central superior, se encuentran los botones *Rotate Right* y *Rotate Left*. Estos botones sirven para el rotar el rotor 45 grados a la derecha o a la izquierda respectivamente. Además, cuentan con un led, que indican el final de carrera en cada uno de los sentidos de giro, ya que el cilindro de giro no puede girar más de 45 grados en cada una de las direcciones.

También, en la parte central, pero situados justo debajo de los botones anteriores, se encuentran los botones *Stacker Stopper* y *Pre Stopper*. El primero de ellos, activa el retenedor de bandejas situado justo en la posición donde debe para la caja para ser almacenada. Por otro lado, el segundo, activa el retenedor que se encarga de parar las bandejas posteriores cuando el apilador está en funcionamiento, situado a una cierta distancia del apilador.

En la parte inferior central, se encuentran los botones *Piston Up* y *Piston Down*. Estos son los botones encargados de subir y bajar el pistón elevador de una forma manual. Cabe recalcar que una vez pulsado uno de estos dos botones, la acción no concluirá hasta que el pistón llegue a su posición final. Esto mismo ocurre con el resto de los botones.

Y, por último, en la esquina inferior izquierda, están la seta de alarma y el botón *Rearm*, que son los encargados de parar el proceso en cuanto se detecte una emergencia. El proceso de parada de emergencia del sistema se activa cuando se pulsa la seta de emergencia, ya sea la situada en el cuadro de mando o la situada en la parte superior del alimentador. Una vez la seta ha sido pulsada, todos los actuadores del sistema se detienen al instante, y volverán en el caso de los pistones a su posición inferior y en el del cilindro de giro en una de sus posiciones predeterminadas. Una vez se desactiva la seta de emergencia y ha sido pulsado el botón "*Rearm*", se encenderá de nuevo la luz que afirma que el sistema vuelve a cumplir el requisito de seguridad.

4.2 Desarrollo del comportamiento físico de los elementos

Una vez se tiene ya recreado digitalmente el modelo del apilador se pasa a la fase de dotarlo de comportamientos físicos, con el fin de que obtenga un comportamiento lo más parecido a la realidad posible. Todo este proceso se lleva a cabo mediante el motor gráfico Unity 3D, el cual se encarga del cálculo de efectos cinemáticos y dinámicos básicos.

Este motor gráfico, proporciona una gran cantidad de herramientas que ayudan a modelar cualquier comportamiento físico de los objetos, sin importar la complejidad de estos. Sin embargo, los principales efectos físicos que se debe tener en cuenta para modelar la física de un elemento son la acción de la gravedad, interacciones con otros objetos y las colisiones entre objetos.

También, para entender cómo funciona el motor gráfico, es importante conocer el concepto de elemento físico. Y es que, elemento físico es la parte del objeto donde los efectos físicos e interacciones con otros objetos son relevantes para el análisis del comportamiento del objeto, por lo que será la única parte del objeto que tendrá los efectos físicos, siendo los objetos restantes elementos meramente visuales. Esta diferenciación, se debe a que hay objetos en los cuales, sí es muy importante sus cualidades físicas para el desarrollo del gemelo digital, mientras que hay otras, que añadirles dichas cualidades no aportan una relevancia para su desarrollo.

En Unity 3D, los efectos físicos solo afectan a los objetos que sean sólidos rígidos. Para esto se acude directamente a la ventana *Inspector* de cada uno de ellos, haciendo clic en el botón *Add Component* y seleccionando el componente deseado. Los componentes que ayudan a modelar los efectos físicos citados anteriormente son principalmente tres, *Rigidbody*, *Colliders* y *Physic material*. A continuación, se pasará a describir las características de cada uno de ellos.

El componente *Rigidbody*, es el encargado de permitir que los objetos actúen bajo el control de la física. Un objeto con este componente puede recibir fuerzas para hacer que se muevan en una manera realista, además, es necesario para que se vea influenciado por el efecto de la gravedad. Este componente, también, cuenta con una serie de propiedades que permite tener una configuración más precisa del efecto físico deseado. Entre estas propiedades, destacan las siguientes: *Mass*, masa del objeto; *Drag*, que determina la resistencia al aire que afecta el objeto cuando se mueva con fuerzas; *Use Gravity*, si está activado, el objeto es afectado por la gravedad; *Is Kinematic*, el objeto no es afectado por el motor de física y solo se puede manipular por su Transform, es decir, es controlado por medio de programación.

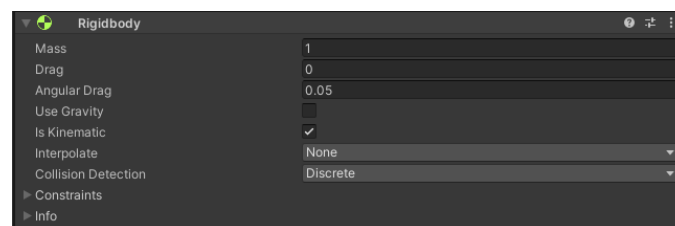


Figura 4.8 - Componente Rigidbody.

Por otro lado, se encuentra el componente *colliders*, que es el borde físico de un objeto, no es un elemento visual. Por tanto, determinará el volumen que tendrá en cuenta el motor físico para las colisiones, es decir, para la interacción entre dos objetos físicos. Un collider, no necesita tener la misma forma exacta que la geometría del objeto y, de hecho, una aproximación a menudo es más eficiente e indistinguible en el juego. Es por esta razón que existen varios tipos de colliders, por un lado, existen collider de geometrías básicas, que consumen menor cantidad de recursos, como

un cubo o una esfera y, por otro lado, existe el *Mesh Collider* que es aquel que tiene la misma geometría que el objeto, y por tanto el que más recursos consume.

En este trabajo se han usado tanto colliders con geometrías básicas como *colliders* que recrean la geometría del objeto. A modo de ejemplo, tenemos dos objetos que representan a cada uno de los dos casos que se acaban de citar. Por un lado, el pistón, emplea un *Mesh Collider* debido a que para que la interacción entre este y las cajas fuese correcta, solo permitía que el collider se adaptase a la geometría del pistón. Por otro lado, tenemos las barras horizontales, donde reposan las cajas cuando están almacenadas, las cuales emplean un *Box Collider* ya que esta simplificación no afecta al modo en el que la caja apoya sobre las barras. En la siguiente figura se puede ver estos dos casos, donde la silueta coloreada de verde representa la geometría del collider.

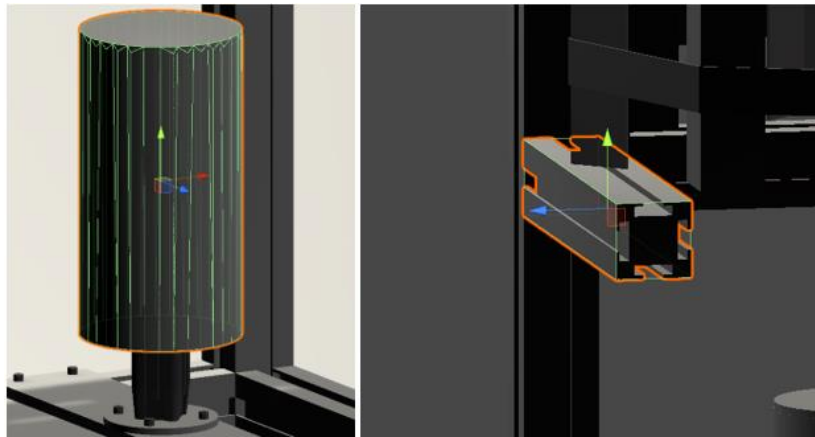


Figura 4.9 - Ejemplos de colliders empleados.

Los *colliders* también cuentan con una serie de propiedades que ayudan a su correcto funcionamiento, como se ve en la figura 4.10, entre ellas destaca la propiedad *material* que permite asociar el objeto a un *Physic material* determinado.

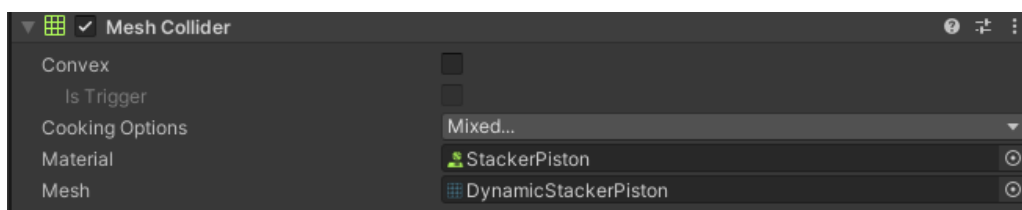


Figura 4.10 - Componente Mesh Collider.

El componente *Physic material*, permite ajustar el comportamiento que tendrá un objeto con el resto de los objetos de su entorno. Entre las propiedades que se pueden ajustar están coeficiente de rozamiento dinámico y estático, y el grado de rebote. También, permite definir como se combinan los valores de fricción o rebote de dos objetos que interactúan entre sí. Se pueden dar tres formas de combinación, que se tome el valor mínimo, máximo o medio de entre los objetos que interactúan.

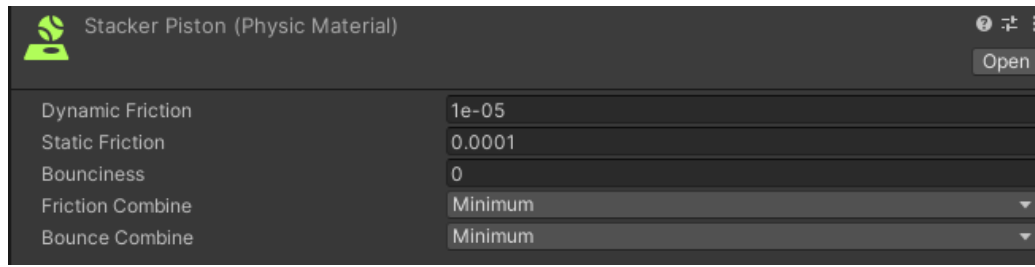


Figura 4.11 - Componente Physic material.

Otro aspecto importante a la hora de recrear comportamientos físicos de los elementos es configurar como estos se comportarán. Para ello, estos movimientos serán configurados por medios de *scripts*, programados en lenguaje C#, que se añadirán a cada elemento al que se quiera dotar de movimiento.

De forma general, los *scripts* cuentan con una estructura general que está compuesta por una clase del tipo *MonoBehaviour*, que consiste en una estructura de datos que combina estados y acciones en una sola unidad. Además, dentro de esta clase, existen dos funciones principales que se repiten en cada *script*. Por un lado, se cuenta con una función, principalmente *Awake()* o *Start()*, la cual es llamada de forma automática justo al iniciar el *script*, y es donde se inicializan las variables y se declaran las referencias entre los *scripts*. Por otro lado, se encuentra el segundo tipo de función, que se ejecuta a determinados intervalos de tiempo de forma periódica y que se emplea para actualizar información del objeto, principalmente *FixedUpdate()* o *Update()*.

En este proyecto, se empleó por un lado la función *Awake()*, la cual es muy parecida a *Start()*, cuya principal diferencia se encuentra en el momento en que es ejecutada, siendo la primera ejecutada con anterioridad a la segunda. Por otro lado, se empleó la función *FixedUpdate()* en vez de *Update()* debido a que esta última se ejecuta en cada frame, mientras que la primera, se ejecutará siempre en un intervalo fijo, por lo que se consigue una ejecución de la escena más homogénea, al no depender de los *fps* (*frames per seconds*), cuyo valor es bastante cambiante. Una vez, explicada la estructura general, el siguiente paso será explicar el funcionamiento de cada uno de los *scripts* de forma individual.

Para la elaboración del modelo del alimentador de bandejas, se han distinguido cuatro elementos principales, que se definen a continuación:

- Sensores.
 - Sensor inductivo.
 - Sensor de barrido.
- Actuadores
 - Pistón elevador.
 - Cilindro rotor.
 - Pistón retenedor.
- Botones del cuadro de mando.
 - Pulsador.
 - Interruptor.
 - Selector.
- Indicadores leds.

4.2.1 Sensores

Los sensores son los elementos encargados de recoger distinta información sobre eventos que suceden en la planta. Dicha información será enviada al sistema de control, para que decida las órdenes sobre los actuadores en cada instante de tiempo. Para poder leer el valor de todos los sensores presentes en el proceso se ha creado una clase, llamada *SensorComponent*, que se muestra a continuación

```
public class SensorComponent : MonoBehaviour
{
    public string Name;
    public int Number;
    public int PLC;
    public bool[] flag
    {
        get {return flagValue;}
        set{
            flagValue = value;
            if (AutomationParameters.internalAutomation){
                InternalAutomation.AutomationProcess(ref Automation.Memory);
                InternalAutomation2.StackerAutomationProcess(ref Automation.Memory);
            }
        }
    }
    private bool[] flagValue;
}
```

Para poder clasificar los sensores, esta clase emplea tres campos. En primer lugar, *Name* con el que se indica el nombre del sensor; *Number* para indicar el número del sensor; y, por último, *Flag* en la que se almacena el valor del sensor mediante una variable booleana (*true* cuando está activo y *false* cuando se encuentra desactivado).

En el caso del alimentador de bandejas, hemos visto que se cuenta con dos tipos de sensores. Por un lado, un sensor inductivo, presente en cada uno de los retenedores. Y, por otro lado, un sensor de barrido situado justo antes del pistón elevador.

Sensor inductivo

Este sensor, detectará las chapas de metal presentes en la parte inferior de las bandejas. Para simular este comportamiento, se ha optado por hacer el siguiente modelo, cuyo código se muestra en el anexo III.

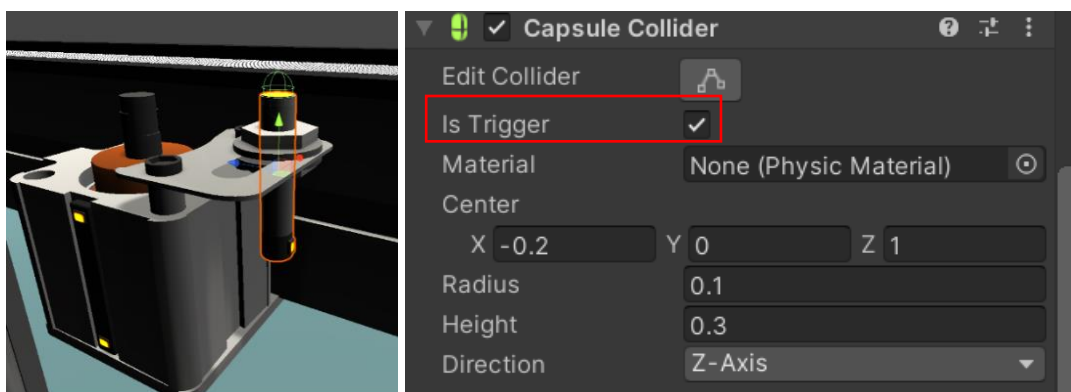


Figura 4.12 - Sensor inductivo.

En primer lugar, a los objetos metálicos se les asignará una etiqueta llamada *metal*, con el fin de diferenciar los elementos que son metálicos de los que no. Para añadir esta etiqueta, se seleccionará el objeto deseado, y en la parte superior de la ventana *Inspector* en el apartado *Tag*, se seleccionará la etiqueta deseada, que previamente ha tenido que ser definida. El siguiente paso, será añadir un *collider* en el extremo del sensor inductivo, tal y como se muestra en la figura 4.12. Con esta posición del *collider*, cuando una bandeja pase por ese punto, la chapa metálica atravesará el *collider*. Cabe recalcar, que este *collider*, tiene la función de detectar cuando un objeto entra y sale del mismo y realizar una llamada a una función, y no la de recrear interacciones físicas con otros objetos. Para poder tener esta cualidad se deberá de marcar la opción *Is Trigger*, como se ve en la figura 4.12.

Para conseguir el comportamiento deseado se ha elaborado el siguiente *script*, llamado *InductiveSensor*, que se adjuntará al *GameObject* del sensor inductivo. El *script* inicialmente define el elemento como un sensor, para poder guardarlo en la memoria de forma ordenada. Luego, se han empleado dos funciones, una para cuando el elemento entre dentro del *collider* (*OnTriggerEnter*) y otra para cuando este salga (*OnTriggerExit*). De esta forma el valor del sensor tendrá valor *true* cuando el elemento entre y *false* cuando el elemento salga.

```
public class InductiveSensor : MonoBehaviour
{
    // DECLARATION OF GLOBAL VARIABLES
    SensorComponent sensorComponent;

    void Awake()
    {
        sensorComponent = gameObject.AddComponent<SensorComponent>() as
SensorComponent;
        sensorComponent.flag = new bool[1];
        sensorComponent.Name = transform.name;
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "metal")
            sensorComponent.flag = new bool[] {true};
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "metal")
            sensorComponent.flag = new bool[] {false};
    }
}
```

Sensor de proximidad tipo barrera

El modelado del sensor de barrido es muy similar al del sensor inductivo. La única diferencia entre ambos se encuentra en la condición necesaria para activarlo. Mientras que, para el sensor inductivo, es necesario que el objeto tenga la etiqueta *metal*, para el sensor de barrido el elemento puede ser cualquiera, independientemente de la etiqueta que tenga asignada. La clase que define al sensor de barrido se llama *ObjectsSensor* y se muestra en el anexo III.

En la siguiente figura, se puede ver como el *collider* tipo *Trigger* recrea el láser que une a emisor y receptor, que, al ser interceptado por un objeto opaco, hace activar al sensor. De esta forma se consigue imitar de una forma adecuada la funcionalidad de este sensor.

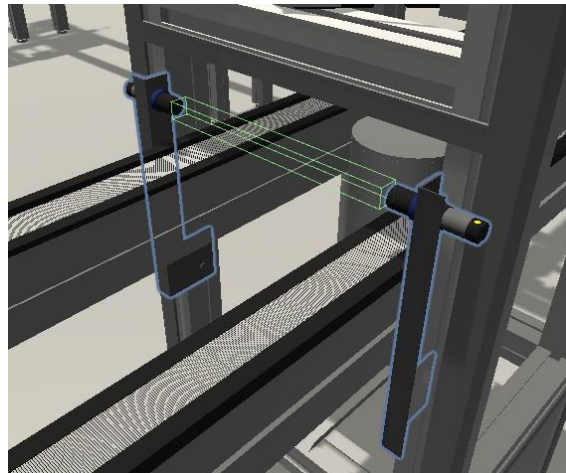


Figura 4.13 - Sensor de barrido.

4.2.2 Actuadores

Los actuadores son los elementos presentes en la planta que ejecutan alguno de los movimientos necesarios durante el proceso de almacenaje de las bandejas. Los actuadores ejecutarán su movimiento en función del estado del sensor o los sensores correspondientes. Para poder llevar el control de todos los actuadores presentes, se ha elaborado una clase, *ActuatorComponent*, donde se definen varios aspectos para su posterior identificación.

Con esta clase se consigue facilitar el direccionamiento y la unificación de todos los actuadores presentes. Para ello, se utilizan cuatro campos diferentes: *Name*, para el nombre del actuador; *Number* para el número del actuador; *Type* clasifica el actuador dentro de cada uno de los grandes grupos de elementos de la planta; y *Flag* donde se almacena el valor del actuador mediante una variable booleana (true cuando está activo y false cuando se encuentra desactivado). A continuación, se muestra el contenido de dicha clase.

```
public class ActuatorComponent : MonoBehaviour
{
    public string Name;
    public int Number;
    public string Type;
    public bool[] flag;
}
```

Una vez definido el componente general del actuador, se pasa a explicar cuáles son los elementos que se han modelado como actuadores. En el caso del alimentador de bandejas los actuadores son, como ya se ha comentado previamente, el pistón elevador, cilindro rotor y los pistones retenedores.

Pistón elevador

Para el pistón elevador, la clase encargada de modelar su movimiento se llama *PistonStackerModule*, la cual se muestra dentro del anexo III. En esta clase, dentro de la función *Awake* se describen los campos necesarios para el actuador, y también, se define el sensor que indica la posición, superior o inferior, en la que se encuentra el pistón. Estos sensores servirán para informar mediante luces en el cuadro de mando en que posición se encuentra el pistón. Dentro de la misma función, se definen los límites superior e inferior en los que se puede encontrar en pistón.

En la función *FixedUpdate()*, se describen los movimientos de subida y bajada del pistón. Para que se lleve a cabo uno de estos movimientos, debe de cumplir dos requisitos: que el actuador correspondiente esté activado y que la posición actual del pistón no sea igual o superior al límite al que se dirige.

Por último, para recrear el movimiento, se accederá a la variable vertical de la posición del objeto y se aumentará/disminuirá según suba o baje respectivamente. Para poder ajustar la velocidad del pistón se ha creado la variable *SpeedUpDown*, que indicará el valor de incremento o decremento de la posición por cada instante de tiempo, cuyo valor viene definido en la clase *ModuleSupport*, que se muestra en el anexo III.

Cilindro rotor

Para conseguir modelar el cilindro rotor, la clase creada es *RotorStackerModule*, de estructura similar a la empleada para el pistón elevador. Este se diferencia respecto del anterior en el tipo de movimiento que realiza, que consiste en una rotación en vez de una traslación. Para poder realizar la traslación, esta vez, se actuará sobre el giro del eje vertical del objeto. También cuenta con el parámetro *SpeedTurn* para poder controlar la velocidad con la que gira, cuyo valor viene definido en la clase *ModuleSupport*, que se muestra en el anexo III.

Retenedores de bandejas

Los retenedores de bandejas que se encuentran en la cinta están controlados por la clase *StopperStackerModule*, que se muestra en el anexo III. Esta clase realiza los movimientos de subida y bajada de los pistones de los retenedores de la misma forma que se hace para el pistón elevador. La única diferencia existente, aparece debido a la presencia de dos leds indicadores de posición situados en el mismo retenedor. Para poder conseguir este comportamiento de los leds, se creará una variable que haga referencia al *GameObject* del led, y mediante la función *SetActive* se mostrará u ocultará en la escena.

4.2.3 Botones del cuadro de mando

Otro aspecto importante es diseñar como se van a comportar los botones del cuadro de mando y que ordenes van a enviar para activar a los actuadores correspondientes. Para poder realizar esto, el programa Unity 3D, ofrece una gran cantidad de herramientas, dentro del campo de la interfaz del usuario (UI).

En Unity 3D, el componente principal de la UI es el Canvas, un área donde todos los elementos que componen la UI deben de situarse. El *canvas* se trata de un *GameObject*, al cual se le añade un componente *canvas* y todos los elementos UI deben de ser hijos de este *canvas*. En la escena el área *canvas* viene representada mediante un rectángulo de color blanco, esto facilita poder posicionar de forma correcta todos los elementos UI dentro del mismo.

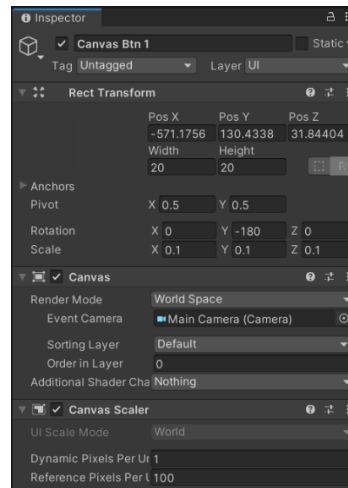


Figura 4.14 - Componente canvas.

El ajuste principal es el *Render Mode* el cual se utiliza para elegir el tipo de renderizado que empleará el *canvas*. Por un lado, tenemos el modo *Screen Space - Overlay* que coloca los elementos de la UI en la pantalla renderizada sobre la escena, si la pantalla se redimensiona o cambia de resolución, el lienzo cambiará automáticamente de tamaño para adaptarse a ella. Por otro lado, se encuentra *Screen Space - Camera*, similar al anterior, pero en este modo de renderizado el canvas se coloca a una distancia determinada frente a una cámara especificada. Los elementos de la UI se renderizan con esta cámara, lo que significa que la configuración de la cámara afecta a la apariencia de la UI. Y, por último, *World Space* que implica que el lienzo se comportará como cualquier otro objeto de la escena. El tamaño del lienzo puede establecerse manualmente utilizando su *Rect Transform*, y los elementos de la UI se renderizarán delante o detrás de otros objetos de la escena basándose en la colocación 3D. Esto es útil para los UI que están destinados a ser una parte del mundo. Por lo tanto, para el caso de los botones, como se ve en la figura 4.14, la opción elegida ha sido la última explicada, *World Space*.

Una vez se tiene elegido el tipo de *canvas*, y en este caso, también el tamaño y la ubicación del lienzo, el siguiente paso es introducir los botones mediante el elemento de Unity, *Button*. A través de este componente, el programa nos permite configurar múltiples aspectos visuales de cada uno de los botones. Entre estos aspectos podemos elegir el color, tonalidad, etc., que tendrá el botón en múltiples situaciones como pueden ser en una situación normal, cuando se pulsa el botón, cuando este se encuentra seleccionado y otras más como se ve en la figura 4.15.

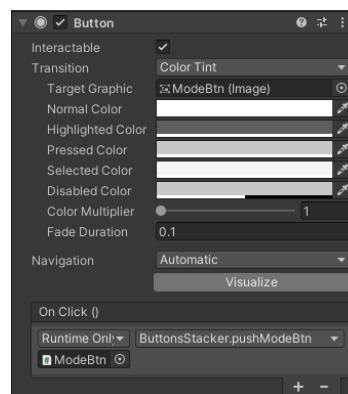


Figura 4.15 - Componente Button.

Otro ajuste que se permite realizar es configurar los eventos que tendrán lugar cuando se presione el botón, que se realiza en la pestaña *On Click*, como se puede ver en la figura 4.15. Para configurar los eventos, en esta pestaña se deberá, por un lado, añadir el *GameObject* que contiene el *script* desde el que se llamará al evento, y posteriormente, elegir la función del *script* que quiere que se ejecute cada vez que se pulse el botón. Al igual que para los sensores y los actuadores, para poder controlar todos y cada uno de los botones se han creado la clase *BottonComponent*, que se muestra en el anexo III. En esta clase, se definen los parámetros *Name* con el que se indica el nombre del botón; *Number* para indicar el número del botón; y, por último, *Flag* con la que se almacena el valor del botón mediante una variable booleana (*true* cuando está pulsado y *false* cuando no se encuentra pulsado).

También, se han diferenciados varios tipos de botones, por lo que cada uno de ellos irá controlado mediante una clase diferente. Los diferentes botones empleados en el proyecto han sido un pulsador, definido mediante la clase *PusherButton*; un selector, cuya clase es *SelectorButton*; y un interruptor, cuya clase es *SwitchButton*. En el anexo III se muestran las clases creadas para cada uno de los tipos de botones.

4.2.4 Indicadores led

Otro aspecto a tener en cuenta es el sistema de indicadores led, que indica al usuario la posición de cada uno de los elementos e información sobre el modo de funcionamiento, y el estado de los requisitos previos necesario para el funcionamiento del sistema de alimentación de bandejas. Para conseguir el efecto de encendido y apagado de los leds, se optó por crear un *GameObject* independiente para cada uno, con el fin de poder ocultar (led apagado) y mostrarlos (led encendido). Para poder ocultar y mostrar *GameObject* la función que se ha utilizado es *GameObject.SetActive()*. A continuación, se muestran algunos ejemplos de la estructura de programación de los indicadores leds. El código que contiene la totalidad de todas las luces que aparecen en el gemelo digital se llama *LightsStacker*, que se encuentra en el anexo III.

```
//Check security Light
if(!RearmBtn[0] || AlarmBtn1[0] || AlarmBtn2[0] || !OnOffBtn[0])
{
    CheckSecurityLight.SetActive(false);
}
if(RearmBtn[0] && !AlarmBtn1[0] && !AlarmBtn2[0] && OnOffBtn[0])
{
    CheckSecurityLight.SetActive(true);
}
//Piston Up Light
if(StackerPistonUp && OnOffBtn[0])
{
    UpLight.SetActive(true);
    DownLight.SetActive(false);
    PistonUpBtn[0] = false;
}
else if(!OnOffBtn[0])
{
    UpLight.SetActive(false);
    PistonUpBtn[0] = false;
    PistonDownBtn[0] = false;
}
```

Se puede observar que, en ambos ejemplos, la estructura es idéntica y lo que cambia son las condiciones del bucle *If* necesarias para activar/desactivar los *GameObjects*. Además, se puede apreciar que previamente se ha definido una variable del tipo *GameObject*. Esta variable, se emplea para asignar el objeto 3D presente en la escena, a la variable que se activa/desactiva en el *script*. En la siguiente imagen vemos como se ha realizado esta asignación.

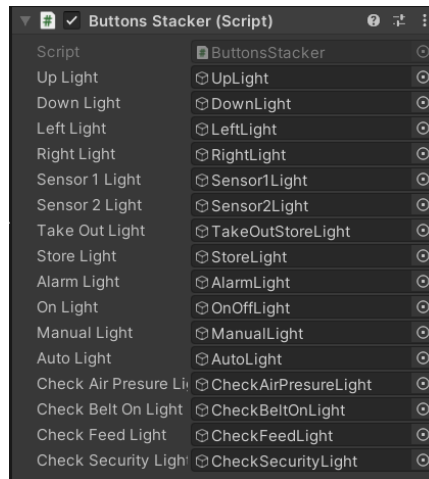


Figura 4.16 - Asignación de objetos a variables tipo *GameObject*.

Sin embargo, este proceso de asignación puede ser bastante tedioso y, además, se debe realizar siempre que se cree un nuevo botón. Es por ello por lo que se empleó otro método, mediante el cual se realiza esta asignación por medios de comandos dentro del *script*. De esta forma, mediante la función *Find* se podrá buscar cualquier *GameObject* presente en la escena, como se muestran en los siguientes ejemplos.

```
DownLight = GameObject.Find("StackerController/Lights/DownLight").gameObject;
LeftLight = GameObject.Find("StackerController/Lights/LeftLight").gameObject;
RightLight = GameObject.Find("StackerController/Lights/RightLight").gameObject;
```

Una vez realizado todo este proceso, se tendrán configurado el comportamiento de todos los botones y el de cada uno de sus leds indicadores. Es de gran importancia que estos elementos se encuentren bien modelados, ya que serán la herramienta mediante la cual el usuario pueda interactuar con el alimentador de bandejas.

5 SISTEMA SCADA

Otro aspecto importante de cualquier gemelo digital es poder visualizar datos del modelo, ya sean datos en tiempo real o un histórico de datos. Es por ello que resulta de gran utilidad el disponer de un sistema que permita al usuario, no solo actuar con el gemelo digital, sino que además permita ver que efectos tienen las acciones que realice sobre este. Además, cuando se trata de crear un entorno de realidad virtual, es necesario que la representación de los datos se haga dentro de la escena, por lo que para este trabajo se recurrió a realizar un sistema SCADA propio en Unity 3D. En este apartado, se explicará el sistema de representación de datos que se ha creado y cuál ha sido el procedimiento para su desarrollo.

Un sistema SCADA se define como “una herramienta para la automatización y el control industrial utilizada en los procesos productivos que puede controlar, supervisar, recopilar, analizar datos, además de generar informes mediante una aplicación informática. Su principal función es la de evaluar los datos con el propósito de subsanar posibles errores” [24]. En la actualidad, estos sistemas han cobrado un papel fundamental dentro de las plantas industriales, ya que permiten mejorar la eficiencia, comunicar problemas de forma rápida para disminuir el tiempo de parada o inactividad y tomar decisiones más inteligentes.

En el caso concreto de este proyecto, el sistema SCADA consistirá en una ventana de representación de curvas gráficas. Mediante el diseño de esta ventana, se quiere conseguir un entorno donde poder mostrar valores tomados en distintos instantes de tiempo. Entre las características principales destacan el ajuste automático de los valores representados en los ejes, apariencia visual de los elementos, diferentes tipos de gráficas, etc. Su finalidad es poder conectar, en trabajos futuros, el gemelo digital a una base de datos, de la cual poder leer los valores almacenados y ser representados gráficamente. Además, como se verá en el siguiente apartado, se incorporarán ventanas informativas a distintos objetos mediante los que mostrar mensajes informativos, datos o la propia ventana de representación de curvas gráficas.

5.1 Desarrollo del sistema SCADA

El sistema SCADA creado para este proyecto consta de una ventana en la que se muestran la gráfica. Los tipos de graficas creado son de tipo barras y de tipo líneas, pudiéndose alternar entre una y otra. Además de mostrar las gráficas, la ventana cuenta con una serie de botones mediante los cuales el usuario puede modificar hasta cierto punto la apariencia visual de las gráficas. Con estos botones, se podrá modificar aspectos como el tipo de gráfica, como ya se ha comentado anteriormente, e incrementar o disminuir la cantidad de valores que se muestran tanto al inicio como al final de la gráfica. También, para facilitar la lectura de las gráficas, estas llevan incorporadas una cuadrícula y las etiquetas en cada uno de los ejes para indicar los valores representados. En la siguiente imagen, se puede ver la apariencia de la venta del sistema SCADA.

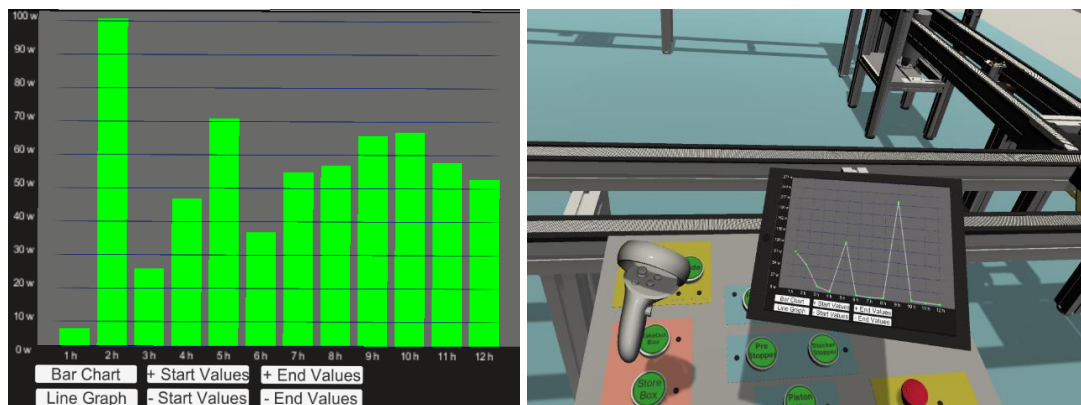


Figura 5.1 - Ventana del sistema SCADA.

Una vez, comentado de manera general la estructura del sistema SCADA, a continuación, se pasará a explicar en más detalle cada uno de los elementos que lo componen.

5.1.1 Implementación de ventana de gráficos

Lo primero que se ha creado, ha sido la ventana que contendrá tanto las gráficas como los botones. En este caso, dicha ventana se recreó de manera que pareciera la pantalla de un dispositivo electrónico de tipo tableta.

El primer elemento que compone la ventana es un *GameObject* tipo *canvas*, debido a que será una ventana con elementos UI, igual que los botones. Este *canvas*, debido a que irá asociado a un objeto que puede moverse libremente por la escena, habrá que configurarlo de manera que su *Render Mode* sea del tipo *World Space*. Como hijo de este elemento *canvas*, se añadirá un panel (de color negro), de manera que actúe como un fondo que contenga a los botones y a las gráficas. El siguiente elemento, hijo del anterior, será otro panel que contiene únicamente las gráficas (de color gris) y que contendrá un nuevo elemento *canvas* que debe tener su origen en la esquina inferior izquierda para que, al representar la gráfica, este sea su origen. Para concluir, se añadirán por un lado los elementos para las etiquetas de los ejes, los cuales deben contener un componente *Text* para configurar los diferentes aspectos de las etiquetas, y, por otro lado, los elementos para formar la cuadrícula con un componente imagen, que dará el aspecto visual de una línea. Estos elementos solo se crearán uno de cada tipo para cada eje, ya que el resto serán clonaciones de este, cuya posición será calculada de manera automática mediante código.

5.1.2 Representación de curvas gráficas

Una vez se tiene elaborada la ventana que contendrá a las gráficas, es momento de implementar dichas gráficas, para lo que se ha creado la clase llamada *window_graph*, la cual se muestra en el anexo III. Dicha clase será adjuntada al *GameObject* que contiene todo el sistema SCADA.

Para la representación de las gráficas, la clase *window_graph*, cuenta con tres funciones principales. La primera función es *ShowGraph* que independientemente del tipo de grafica que sea, se encarga de encuadrar bien dicha gráfica y añadir elementos como cuadrículas o los ejes de coordenadas con sus respectivas etiquetas. Para ello, en primer lugar, la función necesita una serie de entradas que son la lista de valores a representar, indicar el tipo de gráfica, el valor inicial y final en el eje X, y los nombres de las etiquetas para ambos ejes, tal y como se muestra a continuación.

```
ShowGraph(List<int> valueList, IGraphVisual graphVisual, int minVisAmount, int
maxVisAmount, getAxisLabelX, getAxisLabelY);
```

Otro aspecto importante es que independientemente del valor de los datos y el número de ellos, estos siempre se encuentren bien encuadrados dentro de la ventana donde se representan sin que haya valores fuera de esta. Para ello, lo primero será conocer las dimensiones de la ventana, para lo que se usaron los siguientes comandos.

```
// Set boundaries
float graphWidth = GraphContainer.sizeDelta.x;
float graphHeight = GraphContainer.sizeDelta.y;
```

Una vez conocido las dimensiones de la ventana, se cuadrarán las gráficas en ambos ejes. En cuanto al eje Y, lo primero que se debe hacer es encontrar el valor máximo y mínimo. Seguidamente, se calculará la diferencia entre ambos valores para que, de esta forma, entre la parte superior de la gráfica y el valor máximo siempre haya una distancia, que en este caso se será del 20% de la diferencia entre el valor máximo y mínimo, para que de esta manera no se observe una gráfica muy comprimida dentro de la ventana. En las siguientes figuras podemos ver una comparativa donde se puede observar lo comentado.

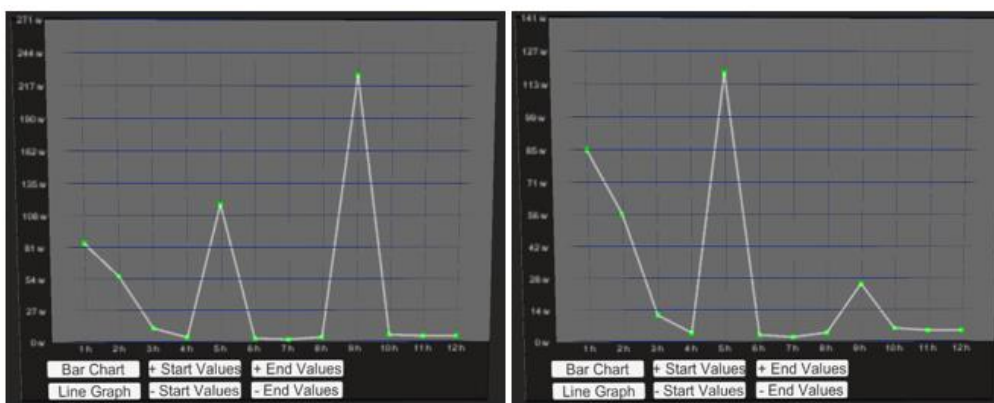


Figura 5.2 - Ajustes automáticos de los valores del eje Y.

El mismo problema aparece cuando varía el número de datos que se muestran en el eje X. Para conseguir esto, se calculará la distancia que separarán los datos entre sí. Una vez obtenida dicha distancia, la posición de cada dato será dicha distancia por el número que ocupa en la lista de datos. Además, se le añadirá una distancia al inicio y al final para que quede una gráfica más cuidada visualmente. Las siguiente líneas muestran como se ha realizado lo anterior.

```
float xSize = graphWidth / ((maxVisAmount - minVisAmount) + 1);
float xPosition = xSize + xIndex * xSize;
```

Una vez, los datos están bien encuadrados dentro de la venta, es momento de posicionar la cuadrícula y los valores en los ejes. En el caso del eje X, el número de valores y de líneas verticales de la cuadrícula que se mostrarán dependerá únicamente del número de datos que se muestren en cada momento. En cuanto al eje Y, este número dependerá únicamente del criterio del programador. Hay que tener en cuenta, que representar un número excesivo de valores hará que estos se solapen y su lectura no sea cómoda para el usuario.

Para representar los valores en los ejes, el procedimiento es parecido para ambos ejes. Donde se proporcionará información sobre como activar el *GameObject*, la posición en la que debe encontrarse o el texto que debe mostrarse. A continuación, se muestra la forma realizada para el eje Y. Una diferencia que aparece en el eje Y, es que, a la hora de representar los valores, estos serán el resultado de dividir el rango de valores entre el número de divisiones que se deseen realizar. Además, se redondearán para evitar que se muestren decimales, simplificando de esta manera la visualización de los datos. Esto se puede ver en la figura 5.2.

```
getAxisLabelY = (float _f) => Mathf.RoundToInt(_f) + " w";
float normValue = i * 1f / sepCount;
RectTransform labelY = Instantiate(labelTemplateY);
labelY.transform.SetParent(GraphContainer);
labelY.gameObject.SetActive(true);
labelY.anchoredPosition = new Vector2(-0.6f, normValue * graphHeight);
labelY.GetComponent<Text>().text = getAxisLabelY(yMin + (normValue * (yMax - yMin)));
gameObjectList.Add(labelY.gameObject);
```

En cuanto a la cuadrícula que facilita la lectura de la gráfica, se compone de líneas horizontales y verticales que se configuran de manera independiente pero muy similar. A continuación, se muestran las líneas de código necesarios para su configuración el eje X.

```
RectTransform dashX = Instantiate(dashTemplateX);
dashX.SetParent(GraphContainer, false);
dashX.gameObject.SetActive(true);
dashX.anchoredPosition = new Vector2(xPosition - 9.3f, -6.2f);
gameObjectList.Add(dashX.gameObject);
```

En la siguiente figura, vemos la apariencia que tiene las gráficas una vez se cambian el número de datos que se representan y el número de divisiones horizontales que se realizan. Se puede observar que, a pesar de la variación de datos representados y divisiones horizontales realizadas, la gráfica sigue manteniendo la misma proporción dentro de la ventana.

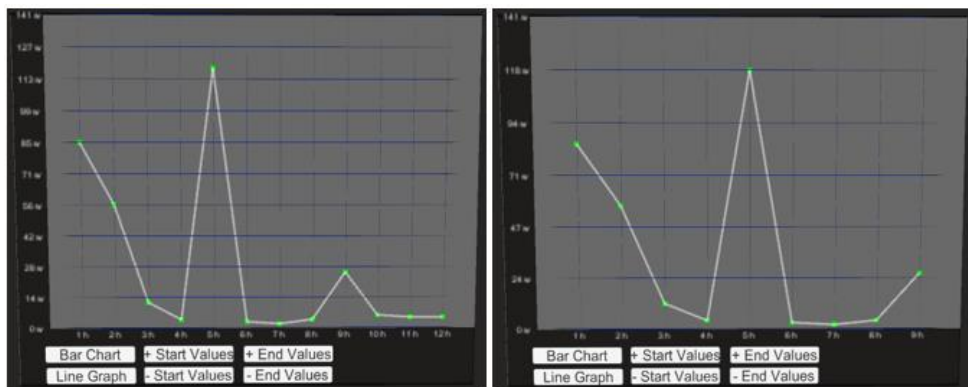


Figura 5.3 - Ajuste automático sin dependencia del número de datos mostrado.

En segundo lugar, se encuentra la función *BarChartVisual* que se encarga de dibujar la gráfica de barras. Para poder crear las barras, será necesario configurar una serie de características de forma que cada una de las barras quede bien definida geoméricamente. Estas características son el lugar donde se localiza la base, tanto en el eje X como en el eje Y; la anchura de cada barra, que deberá permitir tener un pequeño espacio entre barras; y su color.

```
gameObject.GetComponent<Image>().color = barColor;
rectTransform.anchoredPosition = new Vector2(graphPos.x, 0f);
rectTransform.sizeDelta = new Vector2(barWidth * barWidthMultiplier,
graphPos.y);
rectTransform.anchorMin = new Vector2(0, 0);
rectTransform.pivot = new Vector2(.5f, 0f);
```

En la siguiente figura se puede observar una comparativa entre varias situaciones dadas al variar estos parámetros.

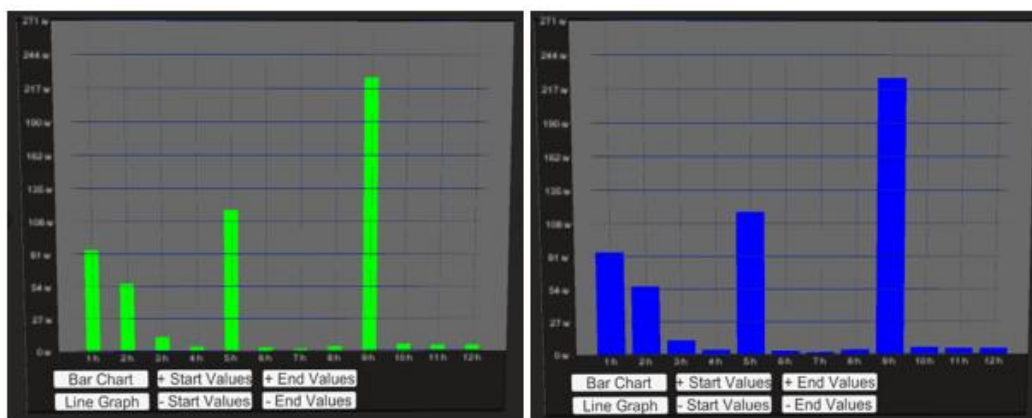


Figura 5.4 - Comparativa de distintas configuraciones visuales de la gráfica de barras.

Y en último lugar, se encuentra la función *LineChartVisual* que se encarga de representar la gráfica de líneas. En este caso habrá que configurar geoméricamente los puntos, igual que se hizo anteriormente, pero, además, habrá que configurar la conexión entre puntos.

En el caso de los puntos, únicamente habrá que configurar la posición que ocuparán dentro de la gráfica, el color; y sus dimensiones (ancho y alto). Para las conexiones, se optó por una unión lineal, para la que había que definir características como su color; la distancia (para lo que se calcula la distancia entre dos puntos contiguos) y la dirección que debía tener. En las siguientes líneas, se puede ver mediante que comandos se llevó a cabo.

```
gameObject.GetComponent<Image>().color = dotConnectionColor; RectTransform
Vector2 dir = (dotPosB - dotPosA).normalized;
float distance = Vector2.Distance(dotPosA, dotPosB);
rectTransform.sizeDelta = new Vector2(distance, 0.1f);
rectTransform.anchoredPosition = dotPosA + dir * distance * 0.5f;
rectTransform.localEulerAngles=new Vector3(0,0, GetAngleFromVectorFloat(dir));
```

A modo de poder apreciar como pueden ser configurada las gráficas de líneas, se muestra la siguiente figura. En ella, se pueden observar cómo varían los colores y geometrías de la gráfica.

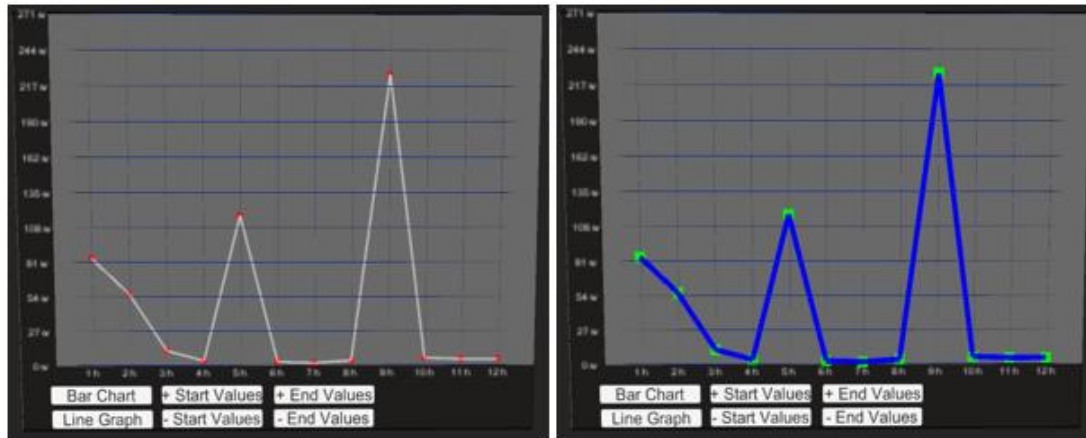


Figura 5.5 - Comparativa de distintas configuraciones visuales de la gráfica de líneas

Por último, para evitar que, al cambiar la gráfica, ya sea al variar el número de datos mostrados o el tipo de gráfica, esta se superponga a la anterior es necesario crear una función que elimine la anterior. Para ello, se ha empleado la función *CleanGraph*, la cual se muestra a continuación.

```
private void CleanGraph()
{
    foreach (GameObject gameObject in gameObjectList)
    {
        Destroy(gameObject);
    }
    gameObjectList.Clear();
}
```

5.1.3 Botones para la interacción con la ventana gráfica.

El último paso, es la elaboración de los botones. El proceso de configuración de estos botones es exactamente igual que el empleado para los botones del panel de control del alimentador de bandejas explicado en el capítulo anterior. Es decir, son botones cuya forma y color pueden ser modificados, al igual que el texto que contienen, y al pulsar dichos botones se ejecuta una función que previamente ha sido programada.

Como se ve en la figura 5.5, el sistema SCADA cuenta con seis botones. En primer lugar, el botón *Bar Chart*, cambiará el tipo de gráfica cambiará a gráfica de barras. Por el contrario, al presionar el botón *Line Graph* el tipo de gráfica se cambiará a gráfica de líneas. Por otro lado, se cuenta con los botones *+ Start Values* y *- Start Values* los cuales sirven para poder aumentar y disminuir respectivamente el número de valores que se muestran en la gráfica en la parte izquierda de la misma. Por último, se tienen los botones *+ End Values* y *- End Values*, mediante los cuales se aumenta y disminuye respectivamente los valores que se muestran en la gráfica en la parte derecha de esta.

Para conseguir estas funciones para los botones es necesario modelar el comportamiento mediante programación. Por ejemplo, para elegir el tipo de gráfica, se utiliza la función *SetGraph*. Esta función comprueba si la variable *isBarChart* esta activada o no, esta se activa al pulsar el botón *Bar Chart* y se desactiva al pulsar *Line Graph*. En función de ello, se llama a la función *ShowGraph* con unas entradas u otras.


```

public void SetGraph(bool isBarChart)
{
    if (isBarChart)
    {
        this.isBarChart = isBarChart;
        ShowGraph(this.valueList, barChartVisual, this.minVisAmount,
this.maxVisAmount, this.GetAxisLabelX, this.GetAxisLabelY);
    } else
    {
        this.isBarChart = isBarChart;
        ShowGraph(this.valueList, lineGraphVisual, this.minVisAmount,
this.maxVisAmount, this.GetAxisLabelX, this.GetAxisLabelY);
    }
}

```

Por otro lado, para los botones encargados de aumentar y disminuir el número de valores que se muestran, las funciones que los controlan son *IncreaseStartValue*, *DecreaseStartValue*, *IncreaseEndValue* y *DecreaseEndValue*. En función del botón que se pulse, lo que se hará será aumentar o disminuir los valores iniciales o finales, teniendo en cuenta que nunca se podrán mostrar más valores de los que se tienen y que al menos siempre se mostrará un valor.

```

public void IncreaseStartValue ()
{
    if (minVisAmount > 0)
    {
        minVisAmount--;
        SetGraph(isBarChart);
    }
}
public void DecreaseStartValue ()
{
    if (minVisAmount <= maxVisAmount)
    {
        minVisAmount++;
        SetGraph(isBarChart);
    }
}

```

En el código anterior, se muestra la función que controla el botón para incrementar y decrementa el número de valores mostrados en la parte inicial de la gráfica. En este caso particular de *IncreaseStartValue*, cuando se pulsa el botón y el número de datos mostrados es mayor a cero, se deja de mostrar el dato situado más a la izquierda. En el caso de querer disminuir el número de datos mostrados, *DecreaseStartValue*, el procedimiento es el mismo. La única diferencia es la condición de entrada que se comprueba que el número mínimo de datos mostrados no sea mayor al número de datos con los que se cuenta. Para los botones que aumentan y disminuyen los datos mostrados en la parte derecha de la gráfica, el procedimiento es el mismo.

6 SISTEMA DE REALIDAD VIRTUAL

La realidad virtual (RV) es una de tecnologías que, en los últimos años ha ido aumentando su nivel de relevancia dentro de la industria debido a las grandes ventajas que ofrece. Y es que este tipo de tecnología permite resolver problemas e incidencias, mejorar la optimización de procesos de trabajos, el mantenimiento de industrias y sus montajes. Pero una de las grandes ventajas que proporciona la RV en este ámbito es el refuerzo a la formación y el entrenamiento de los trabajadores. En este sentido, la realidad virtual permite poner al operario en una situación determinada sin necesidad de exponerlo a ningún tipo de riesgo asociado a su trabajo, lo que se traduce en un aprendizaje en un entorno 100% seguro. Esta tecnología, que muchos creen que solo es útil en el mundo de los videojuegos, es un verdadero avance para que todo el motor industrial se modernice y alcance cotas más altas de excelencia. En definitiva, la tecnología de la realidad virtual es un empuje importante para la digitalización que ayudará de forma inmediata a la reducción de tiempos y costes mejorando muchos aspectos de la producción.

Dentro del mercado de dispositivos de realidad virtual existen una variedad de marcas tales como HP, HTC o Sony, entre otras, que ofrecen una gran cantidad de dispositivos con diferencias características. Sin embargo, para la realización de este proyecto se ha optado por emplear el dispositivo Oculus Quest 2 desarrollado por la marca Oculus.

6.1 Dispositivo Oculus Quest 2

6.1.1 Características técnicas

Las gafas de realidad virtual Oculus Quest 2 se tratan de la segunda generación de los dispositivos Oculus Quest. Estos dispositivos, se lanzaron al mercado con la novedad de que no requerían de cables que sirvieran de conexión para utilizar los gráficos del ordenador. A pesar de que no llega a tener la misma calidad de imagen que los dispositivos que son controlados por ordenador, consigue llegar al nivel requerido por la mayoría de los juegos ya que por lo general estos no requieren un nivel de procesamiento realmente alto. En la siguiente imagen se puede observar la apariencia de este dispositivo.



Figura 6.1 - Dispositivo Oculus Quest 2.

Las gafas a nivel técnico cuentan con una pantalla OLED con una resolución de 1920 x 1832 píxeles por ojo. Cuenta con un procesador Qualcomm Snapdragon XR2 con una frecuencia de refresco máxima de 90 Hz, que se encuentra limitada a 72 Hz por defecto, una memoria RAM de 6GB y una memoria de almacenamiento que varía desde los 64 a los 256 GB. Esto junto a que cuentan con un sonido posicional cinematográfico en 3D, permite al usuario ver los juegos y videos de 360° de una forma envolvente. Por otro lado, las gafas cuentan con cuatro cámaras de posición que permiten al dispositivo saber en cada instante de tiempo donde se encuentra el usuario e indicarle que no se sale de la zona de seguridad (zona diseñada por el usuario donde el juego puede desarrollarse de forma segura). En cuanto a la conectividad, las gafas cuentan con un puerto USB tipo C para su conexión con el ordenador, y, además, una conectividad inalámbrica con Wifi 6 y Bluetooth 5.1.

Además de las gafas, también cuentan con unos mandos con los que moverse por los menús e interactuar con el entorno virtual. Son mandos muy ligeros que facilitan la movilidad y cuentan con un arco para proteger la mano de posibles golpes que se produzcan durante su uso. Otro punto positivo con el que cuentan es la forma ergonómica de estos y la ubicación de los botones, que además cuentan con botones que se pueden diferenciar unos de otros al tacto, lo que facilita su reconocimiento mientras se usan las gafas [17]. Antes de comenzar como se ha configurado el sistema de realidad virtual, es importante conocer como son los mandos y el nombre y posición de sus botones, los cuales se muestran en la siguiente figura [23].

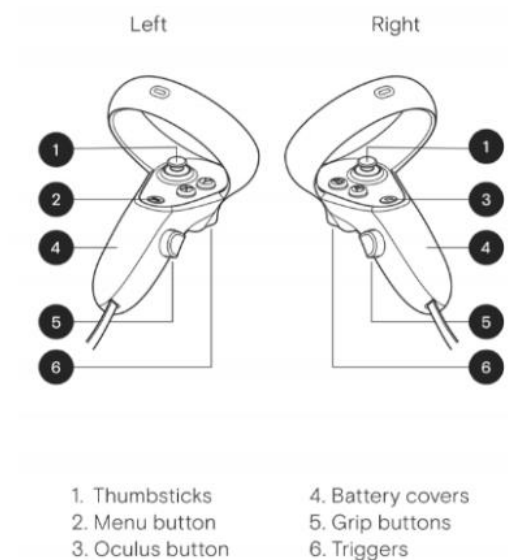


Figura 6.2 - Mandos Oculus Quest 2.

6.1.2 Tipos de conexiones

A pesar de que estas gafas no necesitan cables para su uso, Oculus da la posibilidad de mejorar la calidad de imagen. Es por ello por lo que se ofrecen dos alternativas de conexión, Oculus Link y Air Link. En este apartado se explicarán en qué consisten cada una de estas alternativas.

Oculus Link

En primer lugar, Oculus Link es, la conexión más extendida de las dos. Esta conexión se basa en conectar un cable, alimentado por hilos de fibra óptica a un ordenador para poder utilizar todo el potencial de su tarjeta gráfica. En definitiva, se basa en convertir el dispositivo inalámbrico en uno convencional con cable. Para poder utilizar Oculus Link, el primer paso será conectar el cable a las gafas, seguidamente, habrá que abrir la aplicación para PC de Oculus y comprobar que la conexión se ha realizado de forma exitosa. Para mejorar el rendimiento y la calidad se pueden realizar unos ajustes que variarán en función de las características de cada ordenador.

El primer parámetro que se debe configurar es la tasa de refresco. Este parámetro permitirá al usuario reducir la latencia y hará el entorno virtual más sólido con movimientos más suaves, lo que hará reducir los mareos durante su uso, uno de los mayores inconvenientes que tienen las gafas de realidad virtual. Desde la aplicación, se puede elegir entre una tasa de refresco de 70, 80 y 90 Hz, siendo recomendable usar la mayor tasa de refresco, siempre que el ordenador al que se conecte lo permita.

El siguiente parámetro es el *supersampling*, que se encarga de ajustar la resolución de la imagen mostrada. Al aumentar este parámetro se consigue que la imagen se vea cada vez más nítida. Sin embargo, una mala elección puede causar que el rendimiento se vea afectando, por lo que es necesario ajustarlo en función de las características del PC al que se conecte.

El último parámetro es el *bitrate*, que hace referencia a la cantidad de información que reproduce por segundo, a mayor cantidad de información, mayor calidad de vídeo. Este parámetro puede modificarse a través de la herramienta *Oculus Debug Tools*. En esta herramienta, habrá que cambiar y ajustar el parámetro *Encode Bitrate*, entre 0 y 288 Mbps. El aumento de este parámetro, como ya se dijo aumenta la calidad de la imagen, pero al mismo tiempo aumenta la latencia, por lo que si no se emplea una tarjeta gráfica adecuada la imagen tendrá poca fluidez. En definitiva, la elección de este y los otros dos parámetros es una decisión de compromiso donde se tendrá que elegir una configuración donde calidad de imagen y rendimiento se equilibren [18].

Oculus Air Link

En segundo lugar, la alternativa más reciente, es Oculus Air Link. Es una conexión que hoy en día se encuentra todavía en una fase experimental, por lo que aún no se trata de una conexión perfecta, que busca mantener la idea de un dispositivo inalámbrico, pero con la calidad de imagen de uno conectado por cable.

Air Link es una tecnología que busca la comunicación entre el hardware del ordenador y el software de las Oculus Quest 2, la misma idea que presentaba Oculus Link. Sin embargo, la diferencia con Oculus Link es que esta se realiza de forma inalámbrica mediante una conexión Wifi, lo que supone una mayor comodidad a la hora de su uso y un ahorro a la hora de comprar los accesorios necesarios. Es importante saber que para que la experiencia sea lo mejor posible, la conexión Wifi debe de ser de 5Ghz, y, además, se recomienda que el router que proporciona la conexión Wifi no tenga muchos dispositivos conectados, incluso que las gafas de realidad virtual sea el único dispositivo conectado.

Para poder acceder a esta función, en primer lugar, se deberá activar la función *Reiniciar ajustes de Operaciones experimentales*, además de activar la función de Oculus Air Link en el PC que se vaya a utilizar. Una vez realizado este paso, desde el menú se podrán realizar una serie de ajustes para mejorar la experiencia, como el *bitrate*, donde se podrá elegir entre un *bitrate* fijo o uno dinámico que se ajuste a las condiciones de la red [19].

6.2. Creación de proyectos RV en Unity 3D

Una vez se ha comprendido la tecnología y se ha familiarizado con los dispositivos de RV, el siguiente paso es conseguir conectar el software empleado para la creación del modelo virtual de la célula flexible de fabricación y el software encargado de dar soporte a la realidad virtual.

Para poder conectar los dispositivos de Oculus y crear aplicaciones dentro del programa Unity 3D, es necesario seguir un procedimiento de configuración de este y añadir una serie de herramientas. Para poder realizar este procedimiento existen varias alternativas entre las que se encuentran paquetes para RV creados por Oculus, los cuales se pueden descargar desde la misma tienda de Unity 3D; paquetes creados por agentes externos a Unity 3D y Oculus, como el paquete Talia; y paquetes creados por Unity 3D. En este trabajo, y tras analizar estas tres alternativas, se optó por emplear el paquete diseñado por Unity 3D, llamado XR Toolkit.

El paquete XR Toolkit consiste en un sistema interacción de alto nivel basado en componentes que permite a los usuarios implementar la interactividad en aplicaciones de RV y realidad aumentada, AR, sin la necesidad de programar estas interacciones. Este paquete incluye una serie de componentes que se pueden adjuntar a los distintos *GameObjects* presentes en la escena para dotarlos de propiedades interactivas tales como teletransporte, movimientos, interacción con objetos, etc. Generalmente estos sistemas necesitaban ser programados por el usuario o descargados de librerías externas. Ahora Unity los integra en su software con el objetivo que el desarrollo de una aplicación XR sea más rápido y estable.

Para poder disponer de esta herramienta, es necesario realizar una serie de ajustes y añadir una serie de elementos. El procedimiento seguido en este trabajo para llevar a cabo esta configuración se describe paso a paso en el anexo I. Una vez realizada la instalación de los paquetes necesarios y su configuración, es el momento de configurar las interacciones que podrá hacer el usuario cuando utilice las gafas de realidad virtual.

6.2.1 Visualización de los mandos

La visualización de los mandos a efectos técnicos no aporta una gran utilidad, en el sentido que su incorporación es meramente visual y no tiene repercusión en la interacción con el entorno virtual. Sin embargo, el poder visualizar los mandos dentro del entorno virtual, resulta de una gran utilidad para el usuario, sobre todo si este es algo inexperto.



Figura 6.3 - Visualización de los mandos en la escena.

Además de los mandos, otro elemento que es importante visualizar son los rayos, llamados *XR Ray Interactor*, los cuales emergen de los mandos y actúan como una prolongación de estos, haciendo la misma función que puede tener un puntero laser, tal y como se puede observar más adelante en la figura 6.8. Mediante estos rayos, el usuario podrá saber a qué punto exacto está apuntando con cada uno de sus mandos en cada momento. Esto resulta de gran importancia cuando se pretende seleccionar un botón específico, un lugar concreto donde teletransportarse, etc. En definitiva, hace que la experiencia de realidad virtual sea mucho más sencilla y fluida.

6.2.2 Desplazamientos

El primer modo de interacción del usuario con la escena es el desplazamiento. Cuando se habla en realidad virtual de desplazamientos hay que saber que existen varias formas de moverse por la escena. La primera forma es moverse de manera real, es decir, el usuario se mueve en la realidad y sus pasos se ven reflejados en el entorno virtual. Esta forma de moverse es la que más sensación de realismo da y la que menos sensación de mareo produce. Sin embargo, esta forma de movimiento es la menos utilizada, ya que generalmente, no se cuenta con grandes espacios libres de obstáculos donde moverse libremente y de forma segura.

Por este motivo, debido a las limitaciones que se han comentado, aparecen dos alternativas para recrear el movimiento en el entorno virtual. La primera alternativa son los conocidos como movimientos continuos. Este tipo de movimientos son controlados por los mandos, mediante los joysticks, asemejándose al tipo de movimiento que tiene un videojuego tradicional. El principal inconveniente es que, en escenas de grandes dimensiones, este movimiento puede ser algo ineficiente y lento, ya que se necesitará una gran cantidad de tiempo y espacio para recorrer grandes distancias. Además, la sensación de mareo con este tipo de movimientos es alta cuando su uso es prolongado.

Para añadir a la escena este tipo de desplazamiento, el procedimiento es el siguiente: se añadirá en el objeto *XR Rig* y en la ventana "Inspector" se añadirán el componente *Continuos Move Provider (action-based)*. Dentro de este componente, como se puede ver en la siguiente figura, se puede configurar tanto la velocidad de movimiento en el apartado *Move Speed*; como asignar el botón encargado de activar la acción y que mando lo realiza.

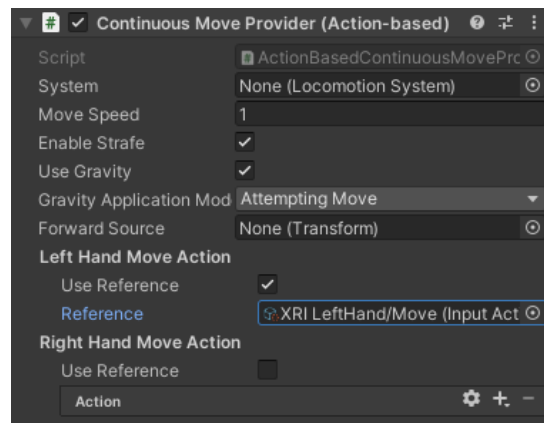


Figura 6.4 - Continuos Move Provider.

En definitiva, y tras realizar pruebas en cada una de las formas de movimientos, se ha concluido que debido a las ventajas y desventajas que presentan cada una, la mejor decisión es hacer una combinación de las dos. Con ello se consigue que, para distancias cortas, y cuando el entorno de uso lo permita, el usuario se podrá mover por la escena tal y como se mueve en la vida real, y que se complementará con los movimientos continuos cuando sean desplazamientos largos o existan impedimentos en el entorno real.

6.2.3 Giros

Otro tipo de movimientos que el usuario necesita realizar son los giros. Al igual que pasaba con los desplazamientos, se disponen de varias alternativas para llevarlos a cabo. La primera forma es hacer giros de manera real, es decir, el usuario hace el giro en la realidad y esto se refleja en el entorno virtual. Al contrario que pasaba en los desplazamientos, el giro real si es de gran utilidad, ya que da una alta sensación de realismo, no causa grandes sensaciones de mareo y, además, no requiere de grandes espacios sin obstáculos. Es por ello, que los videojuegos normalmente solo usan esta forma de realizar los giros.

A pesar de ello, también existen tipos de giros que el usuario puede controlar por medio de los mandos, más concretamente mediante los *joysticks*. El primero, se llama *Continuos Turn*, que consiste en hacer giros continuos, donde el giro de la cámara irá en la dirección en la que se mueva el *joystick*. Sin embargo, la experiencia que se tuvo al realizar este trabajo es que es un tipo de giro que no resulta cómodo, ya que la sensación de mareo es elevada y no aporta ninguna ventaja respecto al giro realizado de forma real. Además, debido a que los mandos cuentan con un número limitado de botones, se decidió no implementar.

El segundo, es el *Snap Turn*, que consiste en hacer giros inmediatos de un cierto ángulo predefinido, que irá en un sentido u otro dependiendo del sentido en el que se mueva el *joystick*. Este tipo de giros, no son nada realistas, pero en ciertas ocasiones, puede ser de gran utilidad para hacer cambios rápidos. También, es importante definir el ángulo que se girará, ya que ángulos tanto pequeños como grandes, causan mareos al usuario.

Para implementar una de estas dos formas de giro, el procedimiento es el siguiente. Dentro del objeto *XR Rig*, añadir el componente *Continuos turn Provider (action-based)* para giros continuos y *Snap turn Provider (action-based)* para giros discontinuos.

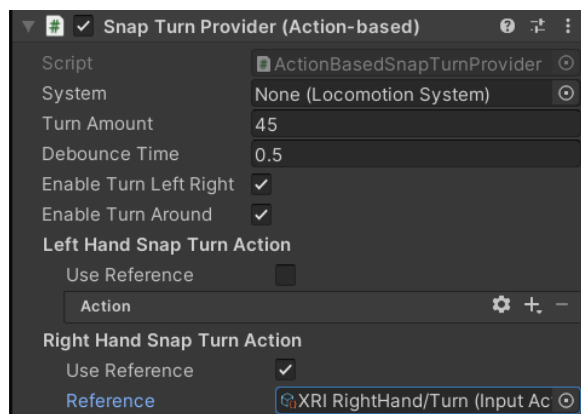


Figura 6.5 - Snap Turn Provider.

Como se ve en la figura 6.5, el componente *Snap turn Provider (action-based)* cuenta con varios parámetros para ajustar. Por un lado, está *Turn Amount* que son los grados que girará la cámara cada vez. También, se debe asignar la acción a uno de los mandos y en concreto a uno de sus botones, de igual manera que se hace en el componente *Continuos Move Provider*.

6.2.4 Teletransporte

La otra alternativa para moverse en realidad virtual, y que en este proyecto completará a las dos mencionadas previamente, es el teletransporte. Esta forma de moverse consiste en saltar desde un punto a otro de la escena de forma inmediata. Con este se consigue dos ventajas principales, por un lado, tiempos cortos para los desplazamientos, ya sean cortos o largos, y una disminución de la sensación de mareo por parte del usuario. Sí es cierto, que esta forma de moverse, para distancias muy cortas puede resultar incomodo ya que realizar saltos inmediatos de un punto a otro de forma repetitiva, provoca una mala sensación al usuario.

Para implementar el teletransporte en la escena, lo primero es añadir a la escena el componente que determina la zona donde el teletransporte se encuentra activo. Para este componente existen dos posibilidades. Por un lado, está el *Teleportation Area* permite el teletransporte a la ubicación señalada por el usuario dentro del área delimitada. Por otro lado, está el *Teleportation Anchor* el cual consiste en un destino que transporta al usuario a una posición y/o rotación específica predeterminada. Para añadir cualquiera de estos elementos: se añade al objeto *XR Rig* el componente *Teleportation Provider*, luego clic con el botón izquierdo en la ventana *Hierarchy*, seleccionar *XR* y elegir una de las dos opciones anteriores.

Una vez se han definido los espacios donde se puede realizar el teletransporte, el siguiente paso es crear la herramienta mediante la cual se podrá llevar a cabo esta acción. Para ello, se ha empleado el elemento *Ray Interactor*, que funciona a modo de puntero laser y permite elegir de forma precisa el lugar al cual teletransportarse. Este elemento, permite ser ajustado mediante varios parámetros para que su manejo sea lo más cómodo posible.

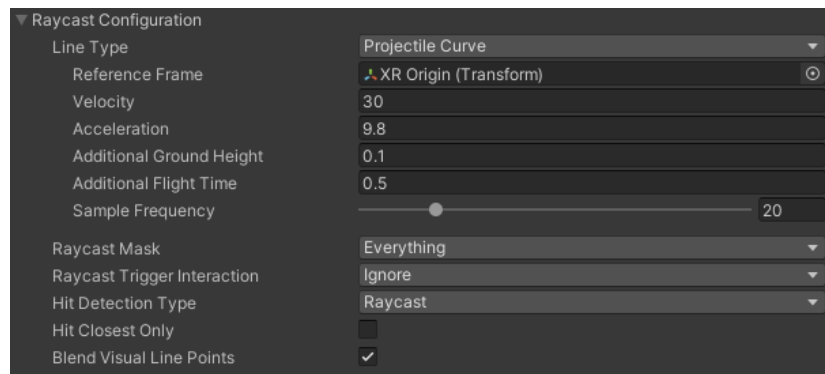


Figura 6.6 - Configuración de la forma del Ray Interactor.

Como vemos en la figura 6.6, en el elemento *XR Ray Interactor*, se puede configurar el tipo de línea, eligiendo entre curva o recta. En el caso del teletransporte se empleó una línea curva para facilitar su uso, cuya curvatura fue editada a través de los parámetros presentes en la figura anterior.

De igual modo, se puede configurar el aspecto visual de los rayos. Como se puede ver en la figura 6.7, donde se pueden configurar aspectos como, el grosor del rayo, su longitud y el color de este. En el caso del color, es una configuración de gran importancia ya que ayuda al usuario a detectar las zonas en las que es posible usar la herramienta de teletransporte. Esto se consigue mediante los *Valid Color Gradient* y *Invalid Color Gradient*, los cuales permitirá mostrar el rayo con un color u otro dependiendo de si es una forma donde se permite el teletransporte o no.

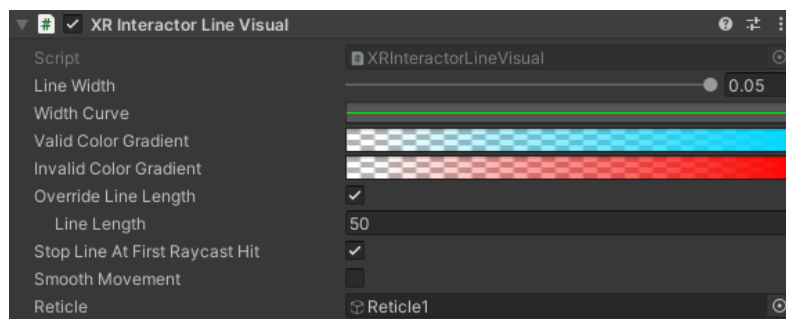


Figura 6.7 - Configuración visual del Ray Interactor.

Como se ve en la figura anterior, aparece un campo llamado *Reticle*. Este elemento consiste en un *GameObject*, que aparece al final del rayo, ayudando al usuario a detectar el punto al cual apunta en cada momento, esto es de gran utilidad cuando se quiere apuntar a objetivos lejanos. En la siguiente figura podemos ver cómo está implementado en el proyecto, donde se aprecia que dentro de la zona gris (*Teleportation Area*) el rayo se visualiza con un color azul y un *Reticle* en su punta, que indica que en ese punto el teletransporte está permitido. Sin embargo, fuera de esta zona el rayo tiene un color rojo y no cuenta con el elemento *Reticle*, indicando que ese punto el teletransporte no está permitido.

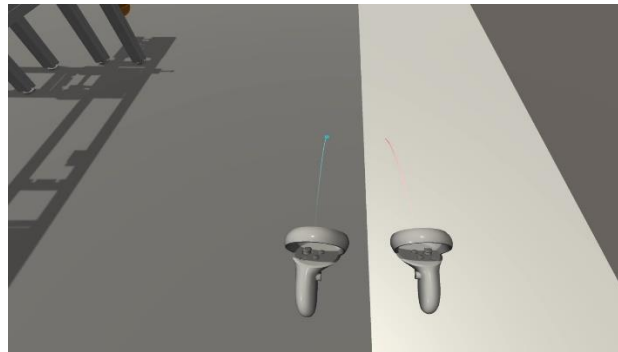


Figura 6.8 - Visualización de los XR Ray Interactor.

Con todo esto, ya es posible realizar el teletransporte de una forma correcta y fácil para el usuario. Sin embargo, existe un inconveniente, y es que con esta configuración los rayos aparecerán todo el tiempo independientemente de si se desea o no teletransportarse, lo que resulta molesto para el usuario. Para evitar esto, se configuró de tal manera que los rayos solo serían visibles mientras se tenga pulsado un botón, el cual puede ser elegido libremente. Una vez pulsado dicho botón, los rayos serían visibles y por consiguiente el teletransporte estaría activado. Para conseguir esto, se creó la clase *ControllerLocomotion*, la cual se puede ver su estructura en el anexo III.

Como se ve en la figura 6.9, al crear la clase anterior y adjuntarla al elemento *XR Rig*, aparecen una serie de opciones que habrá que completar. Por un lado, se debe adjuntar el elemento *Ray Interactor* tanto del mando derecho como del izquierdo. Posteriormente, en el apartado *Teleport Activation Button* se elegirá uno de los botones de los mandos con el cual se harán visibles los rayos y activará el teletransporte. Y, por último, mediante la opción “*Activation Threshold*” se podrá crear una breve espera de tiempo entre el momento que se pulsa el botón y el momento en el que se hacen visibles los rayos.

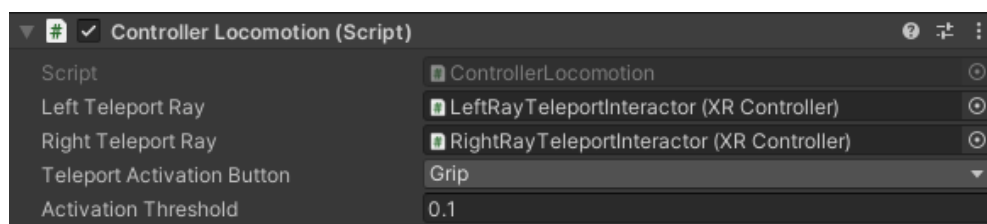


Figura 6.9 - Configuración de la activación de los rayos.

Una vez realizada la configuración anterior, se debe asignar al *GameObject* deseado, que en este caso serán los rayos encargados del teletransporte. Para ello, en los objetos *Ray interactor* dentro del componente *XR Ray Interactor*, se emplearán las opciones que aparecen en el apartado *Interactor Events*. Estas opciones permiten crear distintas configuraciones para varios eventos, en este caso, como se ve en la siguiente figura, será cuando se pulse y deje de pulsar el botón elegido, donde se activará tanto el *Reticle* como el teletransporte cuando se pulse y se desactivarán cuando se deje de pulsar.

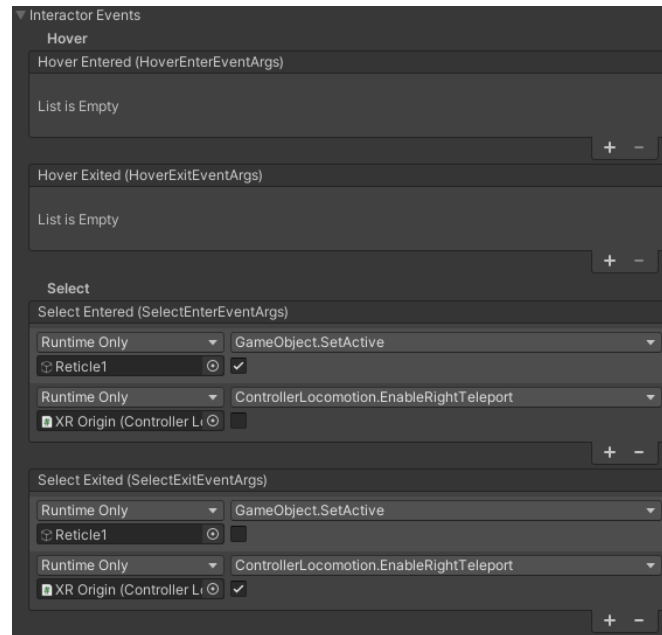


Figura 6.10 - Configuración de los eventos para la activación de los rayos.

Una vez realizado estos pasos, se obtiene el efecto deseado tal y como se ve en la siguiente figura. En el caso del proyecto, para activar el modo teletransporte es necesario mantener pulsado el botón *Grip* y para teletransportarse es necesario pulsar el botón *Trigger*.

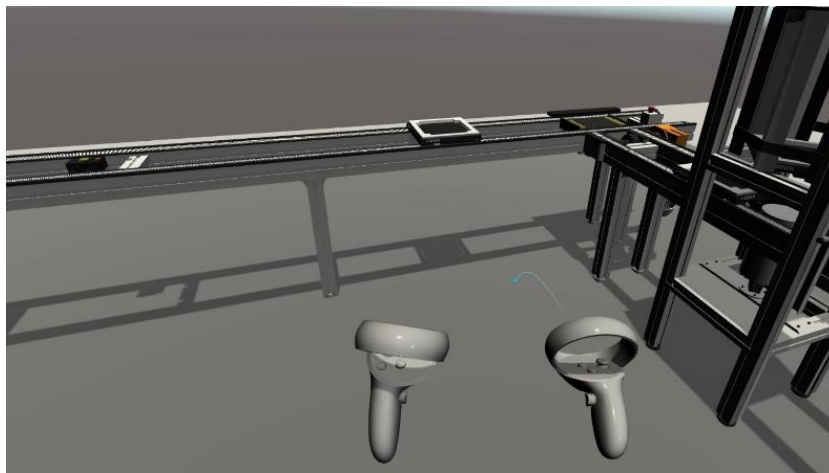


Figura 6.11 - Activación del modo teletransporte.

A parte de configurar estos eventos, Unity permite configurar las vibraciones de los mandos cuando se realizan ciertas acciones. Esto se realiza dentro del elemento *XR Ray Interactor*, en el apartado *Haptic Events*. Como se puede ver en la siguiente figura, en este caso se añadieron vibraciones cuando se seleccionaba un objeto y cuando el rayo lo seleccionaba. Se ve que se puede configurar tanto la intensidad como el tiempo de vibración.

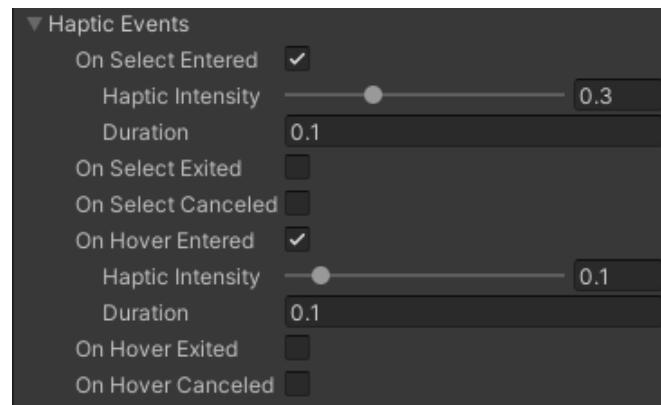


Figura 6.12 - Configuración de la vibración de los mandos.

6.2.5 Interactuación con objetos

A parte de poder moverse por la escena, también es importante poder interactuar con los objetos presentes en ella. Existen muchas formas de interactuar, pero este trabajo se ha centrado en dos formas concretas: poder manipular objetos y pulsar botones. Mediante estas dos acciones, el usuario podrá controlar el alimentador de bandejas mediante los botones del puesto de mandos y coger las bandejas, de igual forma que lo haría en la vida real.

Por un lado, para poder interactuar con los botones no habrá que realizar ninguna modificación a la ya realizada en el apartado 5.2.3. Y, por otro lado, para poder coger objetos en la escena se deberá añadir a cada uno de los objetos el elemento *XR Grab Interactable*.

Pero para que la interacción con los objetos sea más sencilla se creará un nuevo rayo. La configuración del rayo se hará siguiendo los pasos descritos anteriormente, pero con algunas pequeñas modificaciones para adaptarlo a su uso.

En primer lugar, no se añadirá ningún *Reticle* puesto que para pulsar los botones o coger objetos, el usuario no se encontrará a una gran distancia de ellos como puede darse en el caso del teletransporte. También, la forma del rayo será una línea recta de una longitud reducida.

Del mismo modo que ocurría con los rayos del teletransporte, el hecho de que el rayo este siempre visible es algo incómodo para el usuario. Por ello, se ocultará mientras no se requiera su presencia, pero de forma diferente a como se hizo anteriormente. En este caso, en el objeto *Ray Interactor* en el componente *XR Ray Line Visual* para el campo *Valid Color Gradient* se elegirá un color, pero para el campo *Invalid Color Gradient* se elegirá un color totalmente transparente. Además, dentro del componente *XR Ray Interactor* en el apartado *Raycast Mask* elegiremos dos etiquetas, *UI* que asignaremos a los botones y *Grab* que asignaremos a los objetos que se puedan coger. De esta forma se consigue que el rayo esté solo activo cuando se detecte alguno de estos elementos con lo que se pueden interactuar. En la siguiente imagen, podemos ver como en el mando derecho aparece el rayo, ya que está apuntando a uno de los botones.

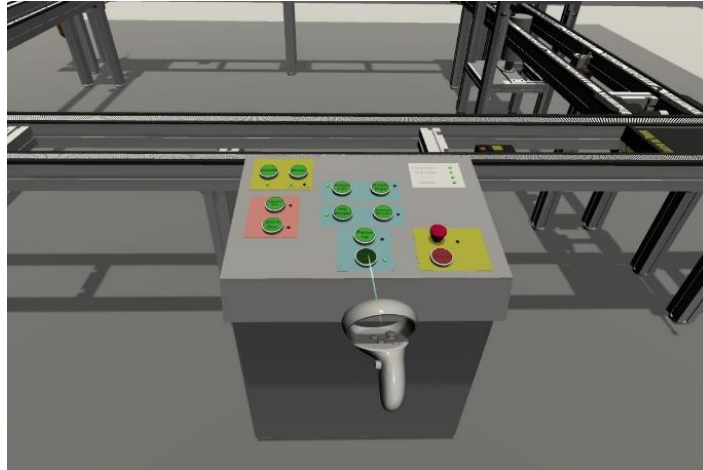


Figura 6.13 - Interacciones con el sistema UI.

Además, hay que tener en cuenta dos aspectos que aparecen cuando se manipula un objeto y se tiene configurado el teletransporte y cuando se cuenta con la visualización de los mandos. El primer aspecto es que cuando se coge un objeto se hace con el mismo botón que con el que se activa el teletransporte, lo que lleva a que en aparezca el rayo del teletransporte en un momento en el que no se requiere. Para evitarlo, hay dos opciones o bien se eligen un botón distinto para cada acción, o bien, como se ha realizado en este proyecto, se crea una condición extra que evite que se pueda activar el teletransporte con el mando con el cual se esté agarrando en ese momento un objeto. Esto se configura en el componente *ControllerLocomotion*. En la siguiente figura, se ve como al agarrar un objeto se muestra de forma simultánea el rayo del teletransporte.

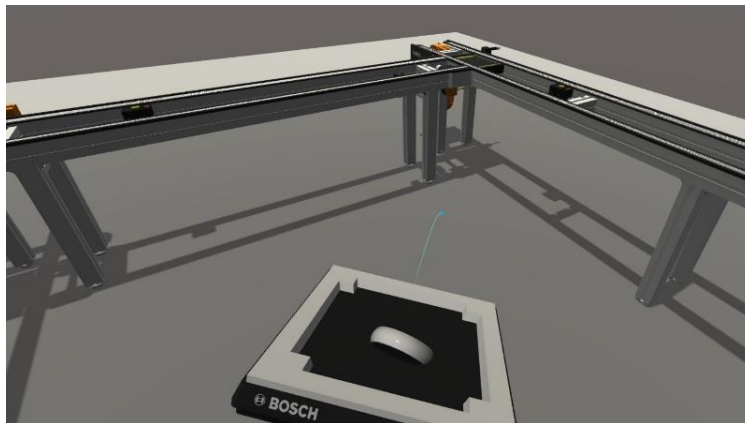


Figura 6.14 - Simultaneidad de teletransporte y objeto agarrado.

El segundo aspecto hace referencia a que cuando se tiene un objeto agarrado y se muestran los mandos, estos dos objetos se fusionan entre si dando un aspecto visual poco real. Para evitar este efecto, en este proyecto se optó por ocultar el mando cuando se agarraba un objeto. Para conseguir esto, iremos al *GameObjects* de cada uno de los mandos (*RightHand Controller* y *LeftHand Controller*) y en el componente *XR Direct Interactor* y se activará la opción *Hide Controller On Select*. En la siguiente imagen, se observa que en la mano derecha del usuario únicamente se muestra el objeto que ha sido agarrado, mientras que en la mano izquierda se muestran tanto el objeto como el mando, donde ambos se unen de forma irreal.

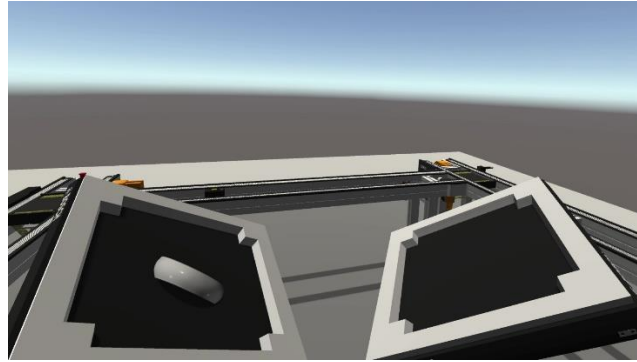


Figura 6.15 - Mejora de la visualización al agarrar objetos.

En el caso concreto de algunos objetos en los que es necesario para su utilización que estos estén en una dirección y posición correcta, se deberán seguir los siguientes pasos. En primer lugar, en el elemento que se desee permitir coger, se creará un *GameObject* vacío, llamado *Pivot*, donde solo aparecerán los ejes de coordenadas. El segundo paso será colocar tanto el origen de coordenadas como los ejes en la posición deseada, teniendo en cuenta que el eje Z es el eje longitudinal del mando, y el eje Y es el eje transversal del mando. Por último, adjuntaremos el elemento creado y se adjuntará dentro del componente *XR Grab Interactable* en el apartado *Attach Transform*.

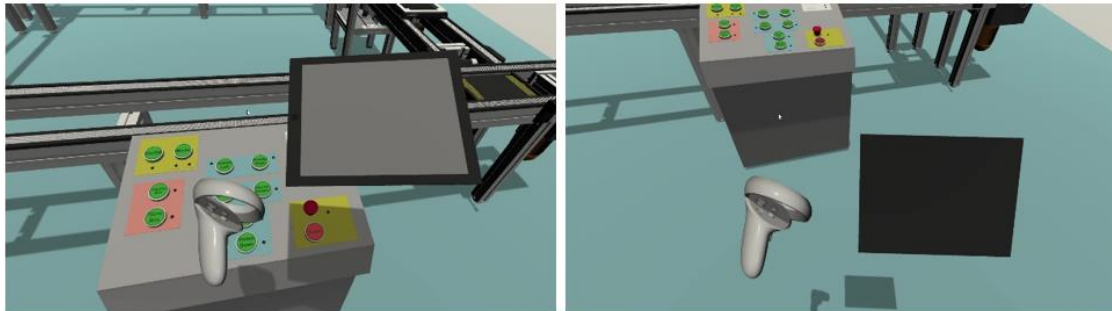


Figura 6.16 - Agarrar objetos en VR.

En último lugar, se ha diseñado una última forma de interactuar con ciertos objetos. Esta forma de interacción tiene como objetivo principal dar al usuario información dentro de la escena de ciertos objetos. El mecanismo de funcionamiento de este sistema que se describe a continuación.

En primer lugar, el usuario deberá accionar el rayo que activa esta opción mediante el botón correspondiente (en este caso se optó por el botón *Primary Button*). Una vez accionado, cuando se detecte un objeto configurado para mostrar información, el rayo cambiará de color, el objeto cambiará a un color verde y aparecerá el panel con la información.

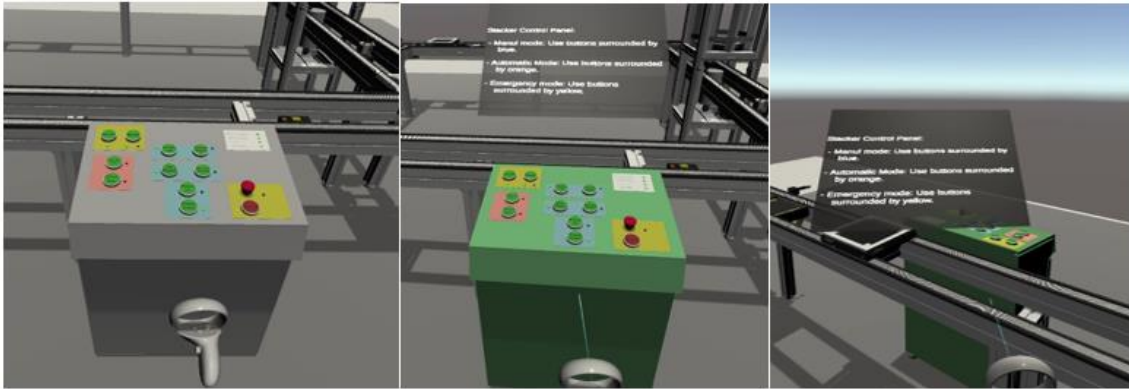


Figura 6.17 - Sistema de información de componentes.

Para conseguir este efecto, lo primero será configurar los rayos, que en este caso se siguió el mismo procedimiento que se utilizó para los rayos del teletransporte, explicados anteriormente. Posteriormente, se creará un *GameObject* vacío como hijo del objeto principal del que se quiere mostrar información. Es importante que este *GameObject* vacío tenga los ejes en la misma dirección y sentido que la cámara. Seguidamente, como hijo de este, se creará un elemento *canvas* que a su vez contenga el plano y el texto a mostrar. A continuación, se deberá añadir el elemento *XR Simple Interactable* al objeto principal y configurarlo, como se muestra en la figura 6.18. Con esto se consigue que al apuntar al objeto este cambie de color y aparezca el panel de información.

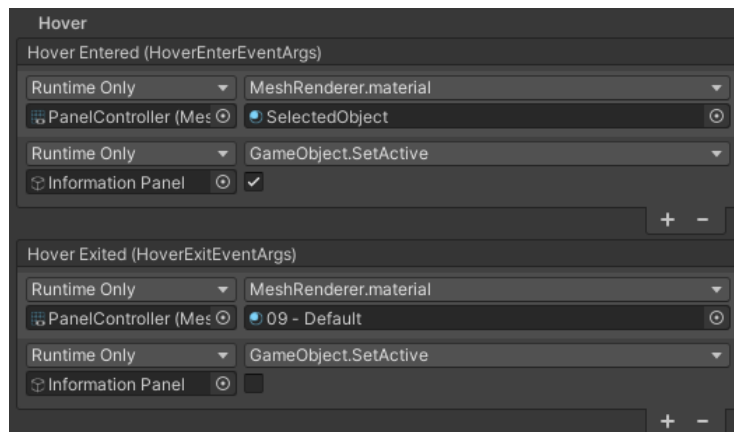


Figura 6.18 - Configuración elemento "XR Simple Interactable".

Para poder conseguir que el panel de información se oriente siempre en la dirección del usuario se ha desarrollado una clase llamada *FollowCamera*, la cual se muestra en el anexo III, que fue adjuntado al elemento *GameObject* del panel.

7 CONCLUSIONES Y TRABAJOS FUTUROS

En este trabajo se ha continuado con la creación de un gemelo digital de una planta de fabricación flexible que se encuentra presente en los laboratorios de la Escuela Técnica Superior de Ingeniería de Sevilla. Para ello, se ha empleado softwares de gran capacidad como son 3DS Max y Unity 3D, cuyas posibilidades y calidad que ofrecen al usuario a la hora de crear nuevos modelos en 3D y dotar a estos de comportamientos físicos realistas, son destacables. Para la creación del gemelo digital no se ha requerido únicamente de conocimientos de modelado 3D, sino que, además, para la elaboración de los comportamientos físicos de los objetos es necesario conocimientos en el uso de la programación en lenguaje C#.

Los principales problemas que ha surgido durante la elaboración de este trabajo han sido dos. Por un lado, durante la elaboración de los modelos 3D, se encontraron multitud de problemas a la hora de exportar las texturas al programa Unity 3D. Finalmente, debido a que supondría un aumento del tamaño de los archivos y puesto que no supondría una gran mejora para el objetivo final del trabajo, se optó por no incluir las texturas de los objetos y dotarlos únicamente con un color similar al que tienen en la realidad. Por estos mismos motivos, hubo ciertos aspectos de la geometría de los objetos que se simplificó, como pueden ser tornillos, cristales, etc.

Por otro lado, a la hora de modelar los comportamientos físicos también existieron algunos inconvenientes. Estos inconvenientes se debieron a que aparecían ciertos comportamientos irreales cuando se daban ciertas interacciones entre objetos. Para poder corregir esto, se tuvieron que ajustar parámetros como la geometría de los *colliders*, ajustes en la masa o coeficientes de rozamientos y rebote.

Por último, se abarca una tecnología que está en pleno auge, como es la realidad virtual, mediante la cual se consigue que la experiencia al utilizar el gemelo digital sea más realista y envolvente. Pero del mismo modo que ocurrió con el modelado 3D, también surgieron problemas durante el proceso. Problemas como ajustes de escala, alturas de las cámaras o la propia configuración entre Unity 3D y el dispositivo de Oculus entre otros. Problemas que han podido ser solventados.

Pero a pesar de que se ha llegado a un punto donde la calidad de este sistema es bastante buena, hay aspectos que se podrían mejorar o implementar. Entre los aspectos a mejorar, destaca la cantidad de memoria que ocupa los archivos de Unity 3D, que impide que se pueda utilizar las

gafas de realidad virtual sin la necesidad de un cable. Para solventarlo podría o bien emplearse el sistema de Oculus link inalámbrico o bien disminuir el tamaño y los recursos necesarios de los archivos de Unity 3D. A modo de mejora, se podría implementar el *Hand Tracking*, el cual permite interactuar con el gemelo digital sin necesidad de mandos, o incluir sonidos del ambiente que doten de mayor realismo a la experiencia.

En línea con la creación de un gemelo digital completo de la planta, todavía quedan por modelar varios elementos que se encuentran presentes en esta, como son brazos robóticos, almacenes de bandejas o mesas de trabajo. Además, para poder sacar más partido al gemelo digital, será necesario crear un sistema de comunicaciones que permita conectar la planta virtual y la real con lo que se pueda una conexión entre ambas plantas. De esta manera se podrá optar entre un modo donde al efectuar acciones en la planta real, estas se vean recreadas al mismo tiempo en la planta virtual; y un segundo modo que funcione de forma inversa. Además, sería necesario la creación de una base de datos que permita representar los datos recogidos de la planta en las ventanas gráficas elaboradas. A su vez, debido a que la realidad virtual se encuentra en continuo avance, sería interesante progresar en los problemas y limitaciones mencionados anteriormente.

8 ANEXO I

En este segundo anexo, se explicará todo el procedimiento necesario para realizar la conexión del programa Unity 3D y el dispositivo Oculus Quest 2, y poder realizar la visualización del gemelo digital realizado en las gafas de realidad virtual [20].

8.1 Configuración del dispositivo Oculus Quest 2

El primer paso a seguir para poder desarrollar las aplicaciones para las gafas Oculus Quest 2 es la creación de una cuenta Oculus, con la que podamos acceder a la aplicación para móviles *Oculus*, la cual se encuentra disponible en todas las plataformas de descarga de aplicaciones, de manera gratuita. Esta aplicación permitirá al usuario gestionar su dispositivo VR, del que podrá tener información de aspectos como nivel de batería, aplicaciones instaladas, conexiones, etc. Pero, sin embargo, la función más relevante para este trabajo es la activación del *Modo desarrollador*, que consiste en una configuración que permite acceder a los comandos de *Android Device Bridge* (ADB) mediante una conexión USB [16]. Este modo es el que permite a los usuarios poder cargar las aplicaciones que han creado en el dispositivo. En el caso particular de este trabajo, permitirá cargar las escenas creadas en el programa Unity 3D. Para poder activar este modo, los pasos a seguir son los siguientes: estar dado de alta como desarrollador en Oculus, dentro de la aplicación para móviles, hacer clic en *Más opciones de configuración* y seguidamente en *Modo desarrollador*. Por último, conectar las gafas al PC mediante USB y aceptar los permisos. Con todo esto, el dispositivo ya estaría listo para cargar la aplicación desarrollada por el usuario.

8.2 Instalación del paquete XR Toolkit en Unity 3D

Una vez se tienen configuradas las gafas, es momento de configurar el programa Unity para que pueda desarrollar las aplicaciones de realidad virtual. En este apartado, se explicará paso a paso el procedimiento a seguir para llevar a cabo aplicaciones de realidad virtual en Unity 3D.

El primer paso, será activar el paquete encargado de la gestión del sistema XR. Para ello, hay que acudir al menú *Project Settings* mediante la pestaña *Edit*. Una vez dentro del menú, entraremos

en el campo de *XR Plug-in Management*, habrá que instalar el complemento y se hará clic en *Initiate XR on Startup* y seguidamente en la opción *Oculus*, tal y como se puede ver en la figura 8.1.

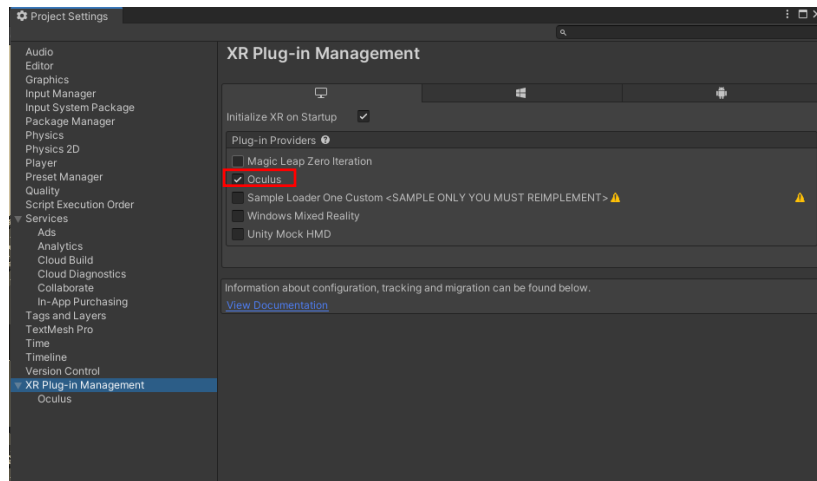


Figura 8.1 - Activación de XR Plug-in Management.

El segundo paso consiste en activar la opción que permite utilizar en el proyecto de Unity 3D paquetes que se encuentra en previsualización. Para realizar este paso, de nuevo se acudiría al menú *Project Settings*, en el campo *Package Manager* y habrá que activar la opción *Enable Preview Packages* tal y como se ve en la siguiente figura.

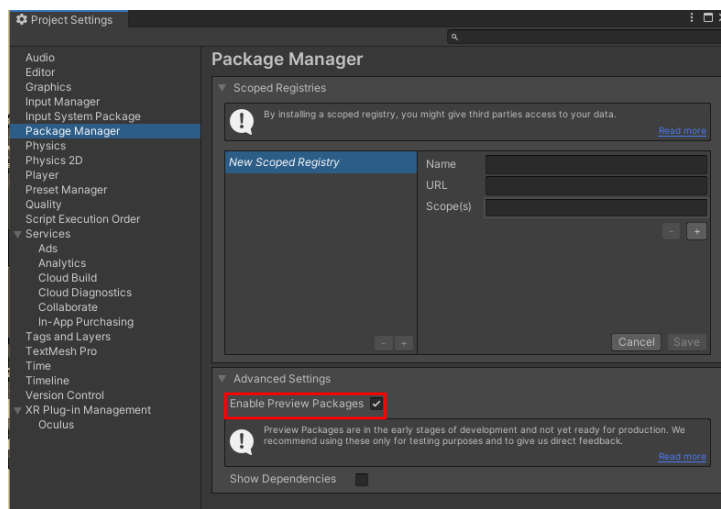


Figura 8.2 - Activación de paquetes en previsualización.

Una vez activados los paquetes en previsualización, es momento de instalar el paquete que se encargará de la gestión de los complementos de XR. Para activarlo, bastará con ir al menú *Package Manager* mediante la pestaña *Window*. Una vez dentro del menú, habrá que habilitar la opción *Unity Registry* en la parte superior de la ventana. A continuación, se instalará el paquete *XR Plugin Management* y *XR interaction Toolkit*, y se importarán los *Samples* que se encuentran disponibles.

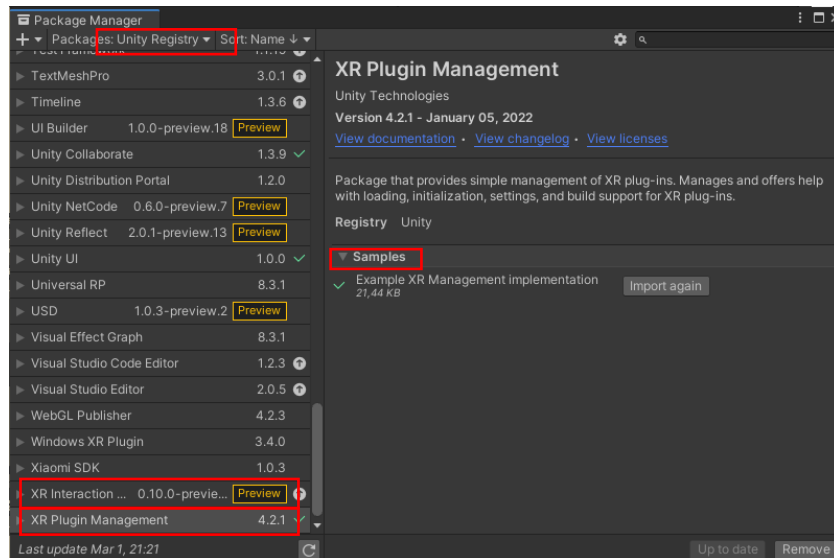


Figura 8.3 - Instalación del paquete "XR Plugin Management".

El siguiente paso será cargar todas las acciones que podrán realizar los mandos, las cuales se pueden ver en la figura 8.4. Para ello, en la ventana *Project* habrá que hacer clic en las siguientes carpetas: *Assets > Samples > XR Interaction Toolkit > 2.0.0 > Starter Assets*. En esta carpeta hay que buscar los archivos *XRI Deafault Left Controller* y *XRI Deafault Right Controller* las cuales contienen las acciones del mando izquierdo y derecho respectivamente. Y en cada uno de estos elementos se hará clic en el botón *Add to ActionBasedController default*, que se encuentra situado en la parte superior de la ventana *Inspector*.

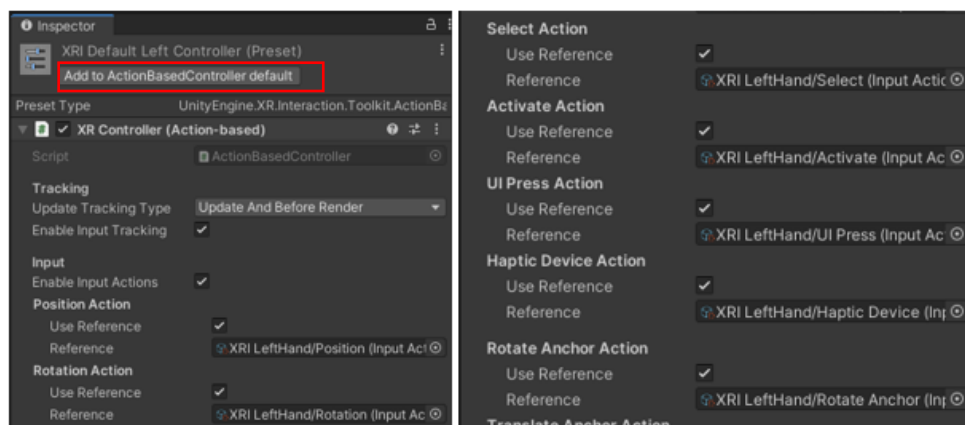


Figura 8.4 - Activación del paquete de acciones para los mandos.

A continuación, en el menú *Project Settings*, en el apartado *Preset Manager* se añadirán dos elementos mediante el botón *Add Default Preset*. Se añadirá un elemento para el mando derecho y otro para el mando izquierdo. A cada elemento en la columna *Preset* se añadirá el componente correspondiente a cada uno de los mandos.

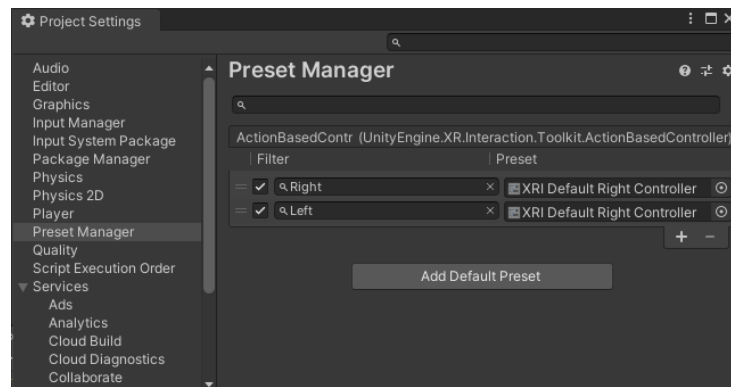


Figura 8.5 - Configuración del Preset Manager.

En último lugar, en la ventana *Hierarchy* se añadirá el elemento *XR Rig* al que se le añadirá el componente *Input Action Manager* en su ventana *Inspector*. En este componente en el apartado *Action Assets*, se modificará el apartado *Size* y se le asignará un valor de 1, después se añadirá el archivo *XRI Default Input Actions* que se encuentra en la carpeta *Starter Assets* que se describió previamente como acceder a ella.

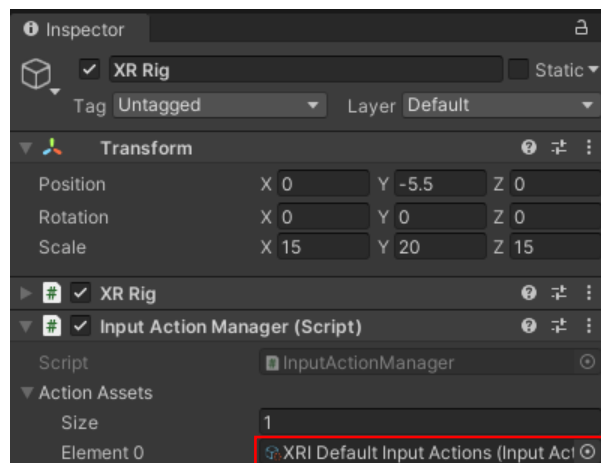


Figura 8.6 - Componentes del elemento XR Rig.

Una vez completado este último paso, el proyecto en Unity 3D se encuentra ya configurado para que se puedan desarrollar las aplicaciones de realidad virtual. Los siguientes pasos a seguir serán añadir elementos y acciones que permitan al usuario interactuar con los elementos que aparecen en la escena y poder moverse en ella. Estos pasos, se explican en el apartado 6.2.

8.3. Exportación de Unity 3D a Oculus

Cuando la escena ya ha sido creada completamente, el siguiente paso es la exportación de los archivos que la componen a las gafas de realidad virtual. Como se ve en el capítulo 6, existen varias formas de conexiones de las gafas, por lo que a la hora de cargar la escena creada en Unity 3D, existen diferentes procesos de exportación en función de que tipo de conexión que se vaya a utilizar.

8.3.1 Exportación standard

En el caso de elegir utilizar la conexión tipo standard, es decir, cuando se utilizan las gafas como un dispositivo inalámbrico, para poder hacer la exportación del proyecto, se abrirá el menú *Build Settings*, al cual se accede mediante la pestaña *File*. Dentro de este menú, se seleccionará la escena que se quiere cargar en las gafas por medio del botón *Add Open Scenes* situado en la parte superior de la ventana. Después de esto en el cuadro superior deberá aparecer el nombre de la escena elegida. Luego, hay que elegir la plataforma del dispositivo al que se exportará la escena, en este caso, los dispositivos de Oculus cuentan con sistema operativo Android. También, cuando las gafas se encuentren conectadas al ordenador y sean detectadas por este, ya que se requiere aceptar una serie de permisos previos, que se pueden aceptar desde las mismas gafas, en el apartado *Run Device* aparecerá el nombre del dispositivo de Oculus, lo que será señal que las gafas se encuentran listas para la exportación de la escena.

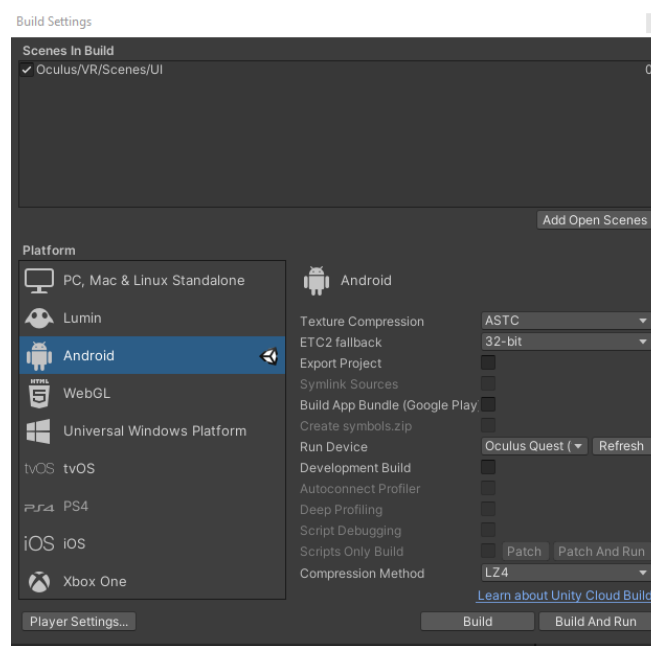


Figura 8.7 - Exportar archivos de forma standard.

Por último, se hará clic en el botón *Build and Run*, situado en la esquina inferior derecha, para cargar la escena en las gafas. Cuando se hace clic en el botón *Build and Run*, se guardará el archivo *apk* generado dentro de la carpeta del proyecto y se empezará a subir los archivos a las gafas (puede llevar unos minutos). Una vez cargado el archivo, ya se pueden desconectar las gafas del ordenador y ver la escena a través de las gafas.

Una vez el archivo con la escena se encuentra almacenado en las gafas de realidad virtual, ya se podrá acceder a él, en cualquier momento, desde las mismas. Para poder acceder a ella, el procedimiento a seguir es el siguiente. Primero, en la barra inferior (si no aparece, seleccionar el botón *Oculus*), se seleccionará la opción *Aplicaciones*. Luego, En la parte superior derecha, desplegaremos la pestaña y seleccionaremos la opción *Orígenes desconocidos*. En este punto, se podrá acceder a todas las escenas que se hayan exportados a las gafas de realidad virtual.

8.3.2 Exportación con Oculus Link

Otra opción, como ya se ha comentado, es usar las gafas conectadas a un PC para aprovechar la capacidad de este. Para poder exportar los archivos de la escena creada mediante Oculus Link, se debe realizar un procedimiento diferente al descrito en el apartado anterior, para poder exportar la escena. El primer paso será elegir la plataforma, en este caso, se elegirá la opción *PC, Mac & Linux Standalone*, ya que la escena se ejecutará a través del ordenador. Para poder cambiar de un tipo de plataforma a otra, lo único que se debe hacer es elegir la nueva plataforma y hacer clic en el botón *Switch Platform*, situado en la parte inferior derecha del menú.

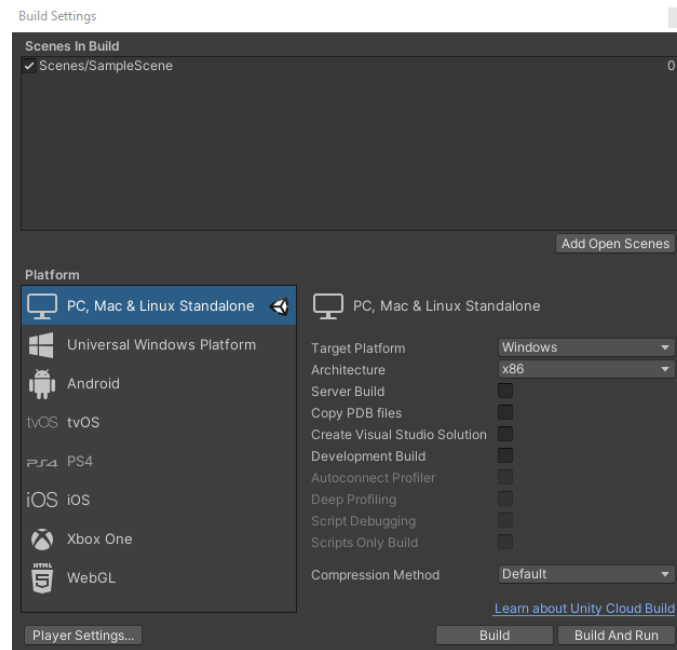


Figura 8.8 - Exportar archivos mediante Oculus Link.

Una vez elegida la plataforma, se añadirá la escena deseada, tal y como se explicó anteriormente. El último paso, será conectar las gafas al ordenador mediante el cable de Oculus Link y hacer clic en el botón *Build And Run*. Esta operación, habrá que repetirla cada vez que se quiera reproducir la escena en las gafas, ya que esta no se almacena en las gafas, sino que ejecuta desde el ordenador.

9 ANEXO II

En el siguiente anexo se representará, de forma simplificada, los diagramas de estado que han sido implementados para el control de las funciones del apilador de bandejas. En ellos, para su simplificación, no se incluirán los botones de emergencia.

En primer lugar, se muestra el diagrama de estado general del sistema de alimentación de bandejas, donde se muestran todas las acciones que se pueden realizar. Y, en segundo lugar, se muestran de forma detallada los diagramas de estados de los procesos automáticos de almacenaje y extracción de bandejas.

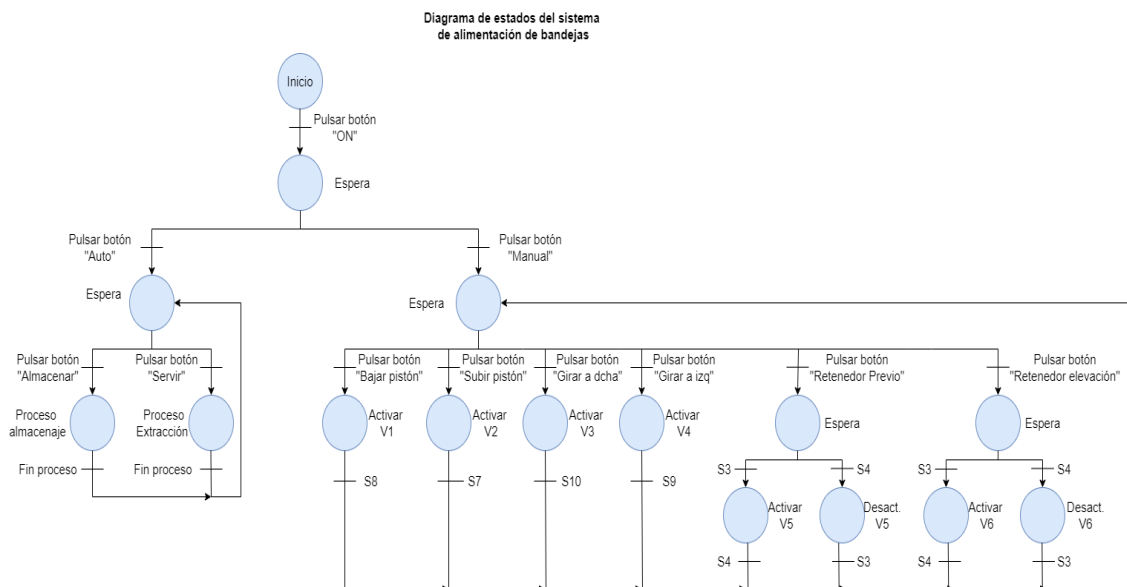


Diagrama de estado del proceso de almacenaje de bandejas

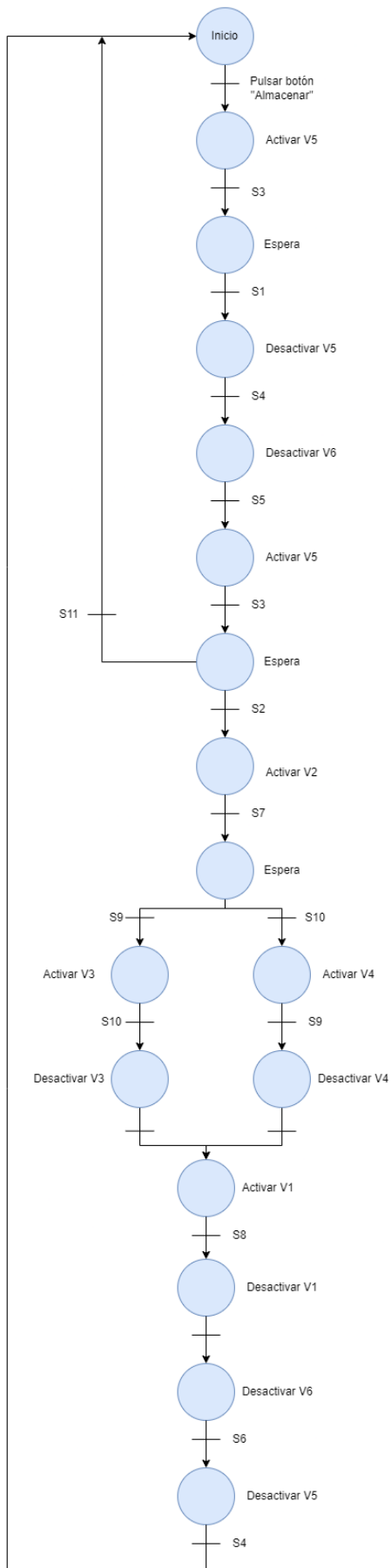
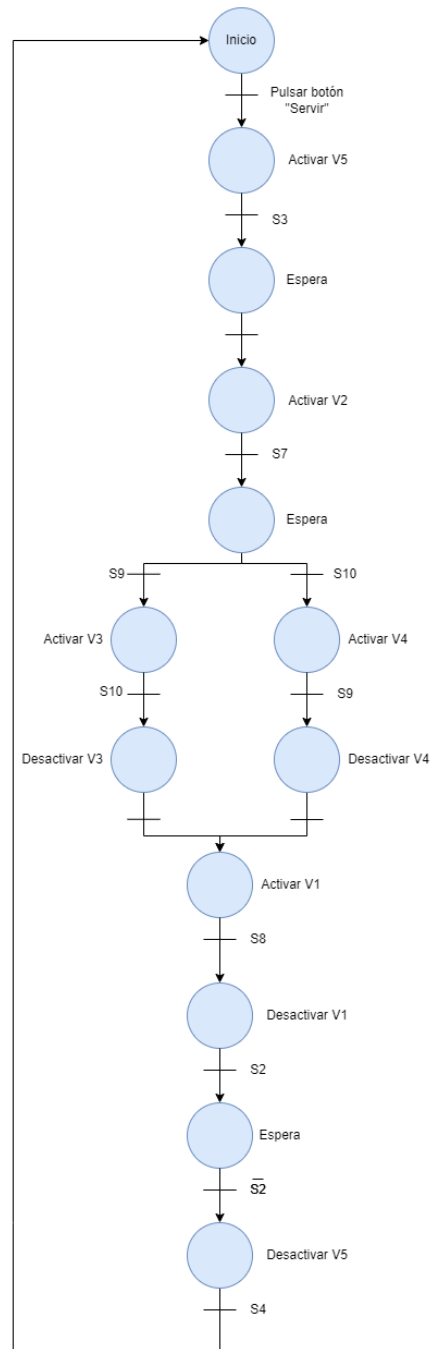


Diagrama de estado del proceso de extracción de bandejas



10 ANEXO III

En este anexo, se mostrarán todos los scripts empleados a lo largo del desarrollo del gemelo digital y para la integración en el entorno de realidad virtual.

Script 1. Sensor Component.

```
using System;
using UnityEngine;
////////////////////////////////////
// VARIABLES FOR THE SENSOR //
////////////////////////////////////
public class SensorComponent : MonoBehaviour
{
    public string Name;
    public int Number;
    public int PLC;

    // When change the sensor value, it launches any functions
    public bool[] flag
    {
        get {return flagValue; }
        set
        {
            flagValue = value;

            if (AutomationParameters.internalAutomation)
            {
                InternalAutomation.AutomationProcess(ref Automation.Memory);
                InternalAutomation2.StackerAutomationProcess(ref
Automation.Memory);
            }
        }
    }
    private bool[] flagValue;
}
```

Script 2. Sensor inductivo.

```

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

////////////////////////////////////
// PARAMETER FOR ALL MODELS //
////////////////////////////////////
public class InductiveSensor : MonoBehaviour
{
    // DECLARATION OF GLOBAL VARIABLES
    SensorComponent sensorComponent;

    void Awake()
    {
        sensorComponent = gameObject.AddComponent<SensorComponent>() as
SensorComponent;
        sensorComponent.flag = new bool[1];
        sensorComponent.Name = transform.name;
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "metal")
            sensorComponent.flag = new bool[]{true};
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "metal")
            sensorComponent.flag = new bool[]{false};
    }
}

```

Script 3. Sensor de barrido.

```

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

public class ObjectsSensor : MonoBehaviour
{
    // DECLARATION OF GLOBAL VARIABLES
    SensorComponent sensorComponent;
    GameObject ObjectSensorStackerLight; // object stacker sensor ON
GameObject

    void Awake()
    {
        sensorComponent = gameObject.AddComponent<SensorComponent>() as
SensorComponent;
        sensorComponent.flag = new bool[1];
        sensorComponent.Name = transform.name;

        ObjectSensorStackerLight =
GameObject.Find("Stacker/ObjectStackerSensor/ObjectStackerSensorLeft/LightSens
or").gameObject;
    }

    private void OnTriggerEnter(Collider other)
    {
        sensorComponent.flag[0] = true;
    }
    private void OnTriggerExit(Collider other)

```

```

    {
        sensorComponent.flag[0] = false;
    }

    void Update()
    {
        if(sensorComponent.flag[0])
        {
            ObjectSensorStackerLight.SetActive(true);
        }
        else
        {
            ObjectSensorStackerLight.SetActive(false);
        }
    }
}

```

Script 4. Actuator Component

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
////////////////////////////////////
// VARIABLES FOR THE ACTUATOR //
////////////////////////////////////
public class ActuatorComponent : MonoBehaviour
{
    public string Name;
    public int Number;
    public string Type;
    public bool[] flag;
}

```

Script 5. Pistón elevador.

```

using System;
using UnityEngine;

////////////////////////////////////
// ACTION OF PISTON STACKER //
////////////////////////////////////
public class PistonStackerModule : MonoBehaviour
{
    Rigidbody PistonStackerObject;
    ActuatorComponent actuatorComponent;
    SensorComponent sensorComponent;
    public static float SpeedRate =1f;

    void Awake()
    {
        actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as
        ActuatorComponent;
        actuatorComponent.flag = new bool[2];
        actuatorComponent.Name = transform.name;
        actuatorComponent.Type = "StackerPiston";

        sensorComponent = gameObject.AddComponent<SensorComponent>() as
        SensorComponent;
        sensorComponent.flag = new bool[2];
        sensorComponent.Name = transform.name;
    }
}

```

```

    PistonStackerObject = GetComponent<Rigidbody>();
}
void FixedUpdate() {

    //Move Up
    if (actuatorComponent.flag[0] &&
PistonStackerObject.transform.localPosition.z <
PistonStackerModuleModel.Parameters.Limits[1])
    {
        PistonStackerObject.MovePosition(PistonStackerObject.position -
transform.forward * PistonStackerModuleModel.Parameters.SpeedUpDown*SpeedRate
* Time.fixedDeltaTime);
    }
    else if ( PistonStackerObject.transform.localPosition.z >=
PistonStackerModuleModel.Parameters.Limits[1])
    {
        sensorComponent.flag[0] = true;
        sensorComponent.flag[1] = false;
    }

    //Move Down
    if (actuatorComponent.flag[1] &&
PistonStackerObject.transform.localPosition.z >
PistonStackerModuleModel.Parameters.Limits[0])
    {
        PistonStackerObject.MovePosition(PistonStackerObject.position +
transform.forward * PistonStackerModuleModel.Parameters.SpeedUpDown*SpeedRate
* Time.fixedDeltaTime);
    }
    else if( PistonStackerObject.transform.localPosition.z <=
PistonStackerModuleModel.Parameters.Limits[0])
    {
        sensorComponent.flag[0] = false;
        sensorComponent.flag[1] = true;
    }
}
}
}

```

Script 6. Cilindro rotor.

```

using System;
using UnityEngine;
////////////////////////////////////
// ACTION OF ROTOR STACKER //
////////////////////////////////////
public class RotorStackerModule : MonoBehaviour
{
    Rigidbody RotorStackerObject;
    ActuatorComponent actuatorComponent;
    SensorComponent sensorComponent;
void Awake()
{
    actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as
ActuatorComponent;
    actuatorComponent.flag = new bool[2];
    actuatorComponent.Name = transform.name;
    actuatorComponent.Type = "StackerRotor";

    sensorComponent = gameObject.AddComponent<SensorComponent>() as
SensorComponent;
    sensorComponent.flag = new bool[2];
    sensorComponent.Name = transform.name;
    RotorStackerObject = GetComponent<Rigidbody>();
}
}

```

```

void FixedUpdate()
{
    //Move Right
    if (actuatorComponent.flag[1] &&
RotorStackerObject.transform.localEulerAngles.z >
RotorStackerModuleModel.Parameters.Limits[0])
    {
        transform.RotateAround(transform.position, Vector3.down,
RotorStackerModuleModel.Parameters.SpeedTurn*Time.fixedDeltaTime);
    }
    else if (RotorStackerObject.transform.localEulerAngles.z <=
RotorStackerModuleModel.Parameters.Limits[0])
    {
        sensorComponent.flag[0] = false;
        sensorComponent.flag[1] = true;
    }
    //Move Left
    if (actuatorComponent.flag[0] &&
RotorStackerObject.transform.localEulerAngles.z <
RotorStackerModuleModel.Parameters.Limits[1])
    {
        transform.RotateAround(transform.position, Vector3.up,
RotorStackerModuleModel.Parameters.SpeedTurn*Time.fixedDeltaTime);
    }
    else if (RotorStackerObject.transform.localEulerAngles.z >=
RotorStackerModuleModel.Parameters.Limits[1])
    {
        sensorComponent.flag[1] = false;
        sensorComponent.flag[0] = true;
    }
}
}

```

Script 7. Cilindro retenedor.

```

using System;
using UnityEngine;
////////////////////////////////////
// ACTION OF SENSOR1 STACKER //
////////////////////////////////////
public class Stopper1StackerActuatorModule : MonoBehaviour
{
    static Rigidbody Sensor1StackerObject;
    ActuatorComponent actuatorComponent;
    SensorComponent sensorComponent;
    void Awake ()
    {
        actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as
ActuatorComponent;
        actuatorComponent.flag = new bool[1];
        actuatorComponent.Name = transform.name;
        actuatorComponent.Type = "Sensor1Stacker";

        sensorComponent = gameObject.AddComponent<SensorComponent>() as
SensorComponent;
        sensorComponent.flag = new bool[2];
        sensorComponent.Name = transform.name;

        Sensor1StackerObject = GetComponent<Rigidbody>();
    }

    void FixedUpdate ()
    {

```

```

        if (actuatorComponent.flag[0] &&
Sensor1StackerObject.transform.localPosition.z <
SensorsStackerModuleModel.Parameters.Limits[1])
        {
            Sensor1StackerObject.MovePosition(Sensor1StackerObject.position -
transform.forward * SensorsStackerModuleModel.Parameters.SpeedUpDown *
Time.fixedDeltaTime);
        }
        else if(Sensor1StackerObject.transform.localPosition.z >=
SensorsStackerModuleModel.Parameters.Limits[1])
        {
            sensorComponent.flag[0] = true;
            sensorComponent.flag[1] = false;
        }
        if (!actuatorComponent.flag[0] &&
Sensor1StackerObject.transform.localPosition.z >
SensorsStackerModuleModel.Parameters.Limits[0])
        {
            Sensor1StackerObject.MovePosition(Sensor1StackerObject.position +
transform.forward * SensorsStackerModuleModel.Parameters.SpeedUpDown *
Time.fixedDeltaTime);
        }
        else if(Sensor1StackerObject.transform.localPosition.z <=
SensorsStackerModuleModel.Parameters.Limits[0])
        {
            sensorComponent.flag[0] = false;
            sensorComponent.flag[1] = true;
        }
    }
}

```

Script 8. Button Component.

```

using System;
using UnityEngine;

////////////////////////////////////
// VARIABLES FOR THE BUTTON //
////////////////////////////////////
public class ButtonComponent : MonoBehaviour
{
    public string Name;
    public int Number;

    // When change the sensor value, it launches any functions
    public bool[] flag
    {
        get { return flagValue; }
        set
        {
            flagValue = value;
        }
    }
    private bool[] flagValue;
}

```


Script 9. Botón interruptor.

```

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

////////////////////////////////////
// PARAMETER FOR ALL BUTTONS MODELS //
////////////////////////////////////
public class switchButton : MonoBehaviour
{
    // DECLARATION OF GLOBAL VARIABLES
    ButtonComponent buttonComponent;
    void Awake()
    {
        buttonComponent = gameObject.AddComponent<ButtonComponent>() as
ButtonComponent;
        buttonComponent.flag = new bool[13];
        buttonComponent.Name = transform.name;
    }
    public void switchbutton()
    {
        if(!buttonComponent.flag[0])
        {
            buttonComponent.flag[0] = true;
        }
        else if(buttonComponent.flag[0])
        {
            buttonComponent.flag[0] = false;
        }
    }
}

```

Script 10. Botón selector.

```

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

////////////////////////////////////
// PARAMETER FOR ALL BUTTONS MODELS //
////////////////////////////////////
public class SelectorButton : MonoBehaviour
{
    // DECLARATION OF GLOBAL VARIABLES
    ButtonComponent buttonComponent;
    void Awake()
    {
        buttonComponent = gameObject.AddComponent<ButtonComponent>() as
ButtonComponent;
        buttonComponent.flag = new bool[13];
        buttonComponent.Name = transform.name;

        buttonComponent.flag[0] = true; // manual mode On
        buttonComponent.flag[1] = false; //auto mode Off
    }
    public void selectorButton()
    {
        if(buttonComponent.flag[0] && !buttonComponent.flag[1])
        {
            buttonComponent.flag[0] = false;
            buttonComponent.flag[1] = true;
        }
        else if(!buttonComponent.flag[0] && buttonComponent.flag[1])

```

```

        {
            buttonComponent.flag[0] = true;
            buttonComponent.flag[1] = false;
        }
    }
}

```

Script 11. Botón pulsador.

```

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

////////////////////////////////////
// PARAMETER FOR ALL BUTTONS MODELS //
////////////////////////////////////
public class PusherButton : MonoBehaviour
{
    // DECLARATION OF GLOBAL VARIABLES
    ButtonComponent buttonComponent;
    void Awake()
    {
        buttonComponent = gameObject.AddComponent<ButtonComponent>() as
ButtonComponent;
        buttonComponent.flag = new bool[13];
        buttonComponent.Name = transform.name;
    }

    public void pusherbutton()
    {
        buttonComponent.flag = new bool[] {true};
    }
}

```

Script 12. Botones del alimentador de bandejas.

```

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

////////////////////////////////////
// ACTION OF BUTTONS STACKER //
////////////////////////////////////

public class ButtonsStacker : MonoBehaviour
{
    ButtonComponent buttonComponent;

    void Awake()
    {
        buttonComponent = gameObject.AddComponent<ButtonComponent>() as
ButtonComponent;
        buttonComponent.flag = new bool[13];
        buttonComponent.Name = transform.name;

        UpLight = GameObject.Find("StackerController/Lights/UpLight").gameObject;
        DownLight =
GameObject.Find("StackerController/Lights/DownLight").gameObject;
        LeftLight =
GameObject.Find("StackerController/Lights/LeftLight").gameObject;
    }
}

```

```

    RightLight =
GameObject.Find("StackerController/Lights/RightLight").gameObject;
    Sensor1Light =
GameObject.Find("StackerController/Lights/StackerStopper1Light").gameObject;
    Sensor2Light =
GameObject.Find("StackerController/Lights/StackerStopper2Light").gameObject;
    TakeOutLight =
GameObject.Find("StackerController/Lights/TakeOutLight").gameObject;
    StoreLight =
GameObject.Find("StackerController/Lights/StoreLight").gameObject;
    AlarmLight =
GameObject.Find("StackerController/Lights/AlarmLight").gameObject;
    OnLight =
GameObject.Find("StackerController/Lights/OnOffLight").gameObject;
    ManualLight =
GameObject.Find("StackerController/Lights/ManualLight").gameObject;
    AutoLight =
GameObject.Find("StackerController/Lights/AutoLight").gameObject;
    CheckAirPresureLight =
GameObject.Find("StackerController/Lights/CheckAirPresureLight").gameObject;
    CheckBeltOnLight =
GameObject.Find("StackerController/Lights/CheckBeltOnLight").gameObject;
    CheckElectricFeedLight =
GameObject.Find("StackerController/Lights/CheckFeedLight").gameObject;
    CheckSecurityLight =
GameObject.Find("StackerController/Lights/CheckSecurityLight").gameObject;
}

/// Buttons lights
GameObject UpLight; //Piston Up Light GameObject
GameObject DownLight; //Piston Down Light GameObject
GameObject LeftLight; //Rotor Left Light GameObject
GameObject RightLight; //Rotor Right Light GameObject
GameObject Sensor1Light; //Previous Sensor Light GameObject
GameObject Sensor2Light; //Back Sensor Light GameObject
GameObject TakeOutLight; //TakeOut Light GameObject
GameObject StoreLight; //Store Light GameObject
GameObject AlarmLight; //Alarm Light GameObject
GameObject OnLight; //On Light GameObject
GameObject ManualLight; //Manual Mode Light GameObject
GameObject AutoLight; //Auto Mode Light GameObject
GameObject CheckAirPresureLight; //Check Air pressure GameObject
GameObject CheckBeltOnLight; //Check belt on GameObject
GameObject CheckElectricFeedLight; //Check power supply GameObject
GameObject CheckSecurityLight; //Check security GameObject

void Update()
{
    //Stacker sensors
    bool StackerSensor1Up =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper1").flag[0]; //true sensor1 up ( S3 )
    bool StackerSensor1Down =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper1").flag[1]; //true sensor1 down ( S4 )
    bool StackerSensor2Up =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper2").flag[0]; //true sensor2 up ( S5 )
    bool StackerSensor2Down =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper2").flag[1]; //true sensor2 down ( S6 )
    bool StackerPistonUp =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerPiston").flag[0]; //true piston at up position( S7 )
    bool StackerPistonDown =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerPiston").flag[1]; //true piston at down position( S8 )
}

```

```

        bool StackerRotorRight =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicRotor").flag[0]; //true rotor at right position ( S9 )
        bool StackerRotorLeft =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicRotor").flag[1]; //true rotor at left position ( S10 )

        //Stacker Buttons
        bool[] OnOffBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"OnOffBtn").flag;
        bool[] RotorLeftBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"RotorLeftBtn").flag;
        bool[] RotorRightBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"RotorRightBtn").flag;
        bool[] PistonUpBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"PistonUpBtn").flag;
        bool[] PistonDownBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"PistonDownBtn").flag;
        bool[] RearmBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"RearmBtn").flag;
        bool[] ModeBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"ModeBtn").flag;
        bool[] StackerSensor1Btn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"StackerStopper1Btn").flag;
        bool[] StackerSensor2Btn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"StackerStopper2Btn").flag;
        bool[] AlarmBtn1 =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"AlarmBtn1").flag;
        bool[] AlarmBtn2 =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"AlarmBtn2").flag;
        bool[] TakeOutBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"TakeOutBtn").flag;
        bool[] StoreBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"StoreBtn").flag;

        //Air tank sensor
        bool AirTankCapacity =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[4].Sensors,
"AirTank").flag[0]; //minimal air tank value

//Check Air presue Light
        if(OnOffBtn[0] && AirTankCapacity)
        {
            CheckAirPresureLight.SetActive(true);
        }
        else if(!OnOffBtn[0] || !AirTankCapacity)
        {
            CheckAirPresureLight.SetActive(false);
        }

//Check belt on Light
        if(OnOffBtn[0])
        {
            CheckBeltOnLight.SetActive(true);
        }

```

```

        if(!OnOffBtn[0])
        {
            CheckBeltOnLight.SetActive(false);
        }
//Check power supply Light
        if(OnOffBtn[0])
        {
            CheckElectricFeedLight.SetActive(true);
        }
        if(!OnOffBtn[0]/*&& AirTank.Capacity >=
AirTankModel.Parameters.MinCapacity*/)
        {
            CheckElectricFeedLight.SetActive(false);
        }
//Check security Light
        if(!RearmBtn[0] || AlarmBtn1[0] || AlarmBtn2[0] || !OnOffBtn[0])
        {
            CheckSecurityLight.SetActive(false);
        }
        if(RearmBtn[0] && !AlarmBtn1[0] && !AlarmBtn2[0] && OnOffBtn[0])
        {
            CheckSecurityLight.SetActive(true);
        }

//Piston Up Light
        if(StackerPistonUp && OnOffBtn[0])
        {
            UpLight.SetActive(true);
            DownLight.SetActive(false);
            PistonUpBtn[0] = false;
        }
        else if(!OnOffBtn[0])
        {
            UpLight.SetActive(false);
            PistonUpBtn[0] = false;
            PistonDownBtn[0] = false;
        }

//Piston Down Light
        if(StackerPistonDown && OnOffBtn[0])
        {
            DownLight.SetActive(true);
            UpLight.SetActive(false);
            PistonDownBtn[0] = false;
        }
        else if(!OnOffBtn[0])
        {
            DownLight.SetActive(false);
            PistonUpBtn[0] = false;
            PistonDownBtn[0] = false;
        }

//Rotor Right Light
        if(StackerRotorRight && OnOffBtn[0])
        {
            RightLight.SetActive(true);
            LeftLight.SetActive(false);
            RotorRightBtn[0] = false;
        }
        else if (!OnOffBtn[0])
        {
            RightLight.SetActive(false);
            RotorRightBtn[0] = false;
            RotorLeftBtn[0] = false;
        }
        /*if(!StackerRotorRight || !OnOffBtn[0])
        {

```

```
    RightLight.SetActive(false);
  }*/

//Rotor Left Light
if(StackerRotorLeft && OnOffBtn[0])
{
    RightLight.SetActive(false);
    LeftLight.SetActive(true);
    RotorLeftBtn[0] = false;
}
else if (!OnOffBtn[0])
{
    LeftLight.SetActive(false);
    RotorRightBtn[0] = false;
    RotorLeftBtn[0] = false;
}
/*if(!StackerRotorLeft || !OnOffBtn[0])
{
    LeftLight.SetActive(false);
}*/

//Previous Sensor 1 Light
if(StackerSensor1Up && OnOffBtn[0])
{
    Sensor1Light.SetActive(true);
}
if(StackerSensor1Down || !OnOffBtn[0])
{
    Sensor1Light.SetActive(false);
}

//Previous Sensor 2 Light
if(StackerSensor2Up && OnOffBtn[0])
{
    Sensor2Light.SetActive(true);
}
if(StackerSensor2Down || !OnOffBtn[0])
{
    Sensor2Light.SetActive(false);
}

//Store Light
if(StoreBtn[0])
{
    StoreLight.SetActive(true);
}
if(!StoreBtn[0])
{
    StoreLight.SetActive(false);
}

//Take Out Light
if(TakeOutBtn[0])
{
    TakeOutLight.SetActive(true);
}
if(!TakeOutBtn[0])
{
    TakeOutLight.SetActive(false);
}

//Alarm Light
if(AlarmBtn1[0] || AlarmBtn2[0])
{
    AlarmLight.SetActive(true);
}
if(!AlarmBtn1[0] && !AlarmBtn2[0])
```

```

    {
        AlarmLight.SetActive(false);
    }

    //On-Off Light
    if (OnOffBtn[0])
    {
        OnLight.SetActive(true);
    }
    if (!OnOffBtn[0])
    {
        OnLight.SetActive(false);
    }

    //Manual Mode Light
    if (ModeBtn[0] && OnOffBtn[0])
    {
        ManualLight.SetActive(true);
    }
    if (!ModeBtn[0] || !OnOffBtn[0])
    {
        ManualLight.SetActive(false);
    }

    //Auto Mode Light
    if (ModeBtn[1] && OnOffBtn[0])
    {
        AutoLight.SetActive(true);
    }
    if (!ModeBtn[1] || !OnOffBtn[0])
    {
        AutoLight.SetActive(false);
    }
}
}

```

Script 13. Luces del alimentador de bandejas.

```

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

////////////////////////////////////
// LIGHTS OF STACKER //
////////////////////////////////////

public class LightsStacker : MonoBehaviour
{
    ButtonComponent buttonComponent;

    void Awake ()
    {
        buttonComponent = gameObject.AddComponent<ButtonComponent>() as
        ButtonComponent;
        buttonComponent.flag = new bool[13];
        buttonComponent.Name = transform.name;

        UpLight = GameObject.Find("StackerController/Lights/UpLight").gameObject;
        DownLight =
        GameObject.Find("StackerController/Lights/DownLight").gameObject;
        LeftLight =
        GameObject.Find("StackerController/Lights/LeftLight").gameObject;
        RightLight =
        GameObject.Find("StackerController/Lights/RightLight").gameObject;
    }
}

```

```

    Sensor1Light =
GameObject.Find("StackerController/Lights/StackerStopper1Light").gameObject;
    Sensor2Light =
GameObject.Find("StackerController/Lights/StackerStopper2Light").gameObject;
    TakeOutLight =
GameObject.Find("StackerController/Lights/TakeOutLight").gameObject;
    StoreLight =
GameObject.Find("StackerController/Lights/StoreLight").gameObject;
    AlarmLight =
GameObject.Find("StackerController/Lights/AlarmLight").gameObject;
    OnLight =
GameObject.Find("StackerController/Lights/OnOffLight").gameObject;
    ManualLight =
GameObject.Find("StackerController/Lights/ManualLight").gameObject;
    AutoLight =
GameObject.Find("StackerController/Lights/AutoLight").gameObject;
    CheckAirPressureLight =
GameObject.Find("StackerController/Lights/CheckAirPressureLight").gameObject;
    CheckBeltOnLight =
GameObject.Find("StackerController/Lights/CheckBeltOnLight").gameObject;
    CheckElectricFeedLight =
GameObject.Find("StackerController/Lights/CheckFeedLight").gameObject;
    CheckSecurityLight =
GameObject.Find("StackerController/Lights/CheckSecurityLight").gameObject;
    UpStackerSensor2Light =
GameObject.Find("StackerStopper2/Lights/LightUp").gameObject;
    DownStackerSensor2Light =
GameObject.Find("StackerStopper2/Lights/LightDown").gameObject;
    InductiveSensorStacker2Light =
GameObject.Find("StackerStopper2/Lights/LightSensor").gameObject;
    UpStackerSensor1Light =
GameObject.Find("StackerStopper1/Lights/LightUp").gameObject;
    DownStackerSensor1Light =
GameObject.Find("StackerStopper1/Lights/LightDown").gameObject;
    InductiveSensorStacker1Light =
GameObject.Find("StackerStopper1/Lights/LightSensor").gameObject;
}

/// stacker lights
GameObject UpLight; //Piston Up Light GameObject
GameObject DownLight; //Piston Down Light GameObject
GameObject LeftLight; //Rotor Left Light GameObject
GameObject RightLight; //Rotor Right Light GameObject
GameObject Sensor1Light; //Previous Sensor Light GameObject
GameObject Sensor2Light; //Back Sensor Light GameObject
GameObject TakeOutLight; //TakeOut Light GameObject
GameObject StoreLight; //Store Light GameObject
GameObject AlarmLight; //Alarm Light GameObject
GameObject OnLight; //On Light GameObject
GameObject ManualLight; //Manual Mode Light GameObject
GameObject AutoLight; //Auto Mode Light GameObject
GameObject CheckAirPressureLight; //Check Air pressure GameObject
GameObject CheckBeltOnLight; //Check belt on GameObject
GameObject CheckElectricFeedLight; //Check power supply GameObject
GameObject CheckSecurityLight; //Check security GameObject
GameObject UpStackerSensor2Light; // Stacker sensor 2 Up Light GameObject
GameObject DownStackerSensor2Light; // Stacker sensor 2 Down Light
GameObject
GameObject InductiveSensorStacker2Light; // inductive stacker sensor
2 GameObject
GameObject UpStackerSensor1Light; // Stacker sensor 1 Up Light GameObject
GameObject DownStackerSensor1Light; // Stacker sensor 1 Down Light
GameObject
GameObject InductiveSensorStacker1Light; // inductive stacker sensor
1 GameObject

void Update()
{

```



```

//Stacker sensors
bool InductiveSensorStacker1 =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"InductiveSensorStacker1").flag[0]; //true box in the stacker sensor 1 ( S1 )
    bool InductiveSensorStacker2 =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"InductiveSensorStacker2").flag[0]; //true box in the stacker sensor 2 ( S2 )
    bool StackerSensor1Up =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper1").flag[0]; //true sensor1 up ( S3 )
    bool StackerSensor1Down =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper1").flag[1]; //true sensor1 down ( S4 )
    bool StackerSensor2Up =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper2").flag[0]; //true sensor2 up ( S5 )
    bool StackerSensor2Down =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerStopper2").flag[1]; //true sensor2 down ( S6 )
    bool StackerPistonUp =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerPiston").flag[0]; //true piston at up position( S7 )
    bool StackerPistonDown =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicStackerPiston").flag[1]; //true piston at down position( S8 )
    bool StackerRotorRight =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicRotor").flag[0]; //true rotor at right position ( S9 )
    bool StackerRotorLeft =
AutomationFunctions.ReturnSensorComponentByName(Automation.Memory[5].Sensors,
"DynamicRotor").flag[1]; //true rotor at left position ( S10 )

//Stacker Buttons
bool[] OnOffBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"OnOffBtn").flag;
    bool[] RotorLeftBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"RotorLeftBtn").flag;
    bool[] RotorRightBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"RotorRightBtn").flag;
    bool[] PistonUpBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"PistonUpBtn").flag;
    bool[] PistonDownBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"PistonDownBtn").flag;
    bool[] RearmBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"RearmBtn").flag;
    bool[] ModeBtn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"ModeBtn").flag;
    bool[] StackerSensor1Btn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"StackerStopper1Btn").flag;
    bool[] StackerSensor2Btn =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"StackerStopper2Btn").flag;
    bool[] AlarmBtn1 =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"AlarmBtn1").flag;
    bool[] AlarmBtn2 =
AutomationFunctions.ReturnButtonComponentByName(Automation.Memory[6].Buttons,
"AlarmBtn2").flag;

```

```

        bool[] TakeOutBtn =
AutomationFunctions.ReturnButtonComponentByName (Automation.Memory[6].Buttons,
"TakeOutBtn").flag;
        bool[] StoreBtn =
AutomationFunctions.ReturnButtonComponentByName (Automation.Memory[6].Buttons,
"StoreBtn").flag;

        //Air tank sensor
        bool AirTankCapacity =
AutomationFunctions.ReturnSensorComponentByName (Automation.Memory[4].Sensors,
"AirTank").flag[0]; //minimal air tank value
        //Check Air pressue Light
        if (OnOffBtn[0] && AirTankCapacity)
        {
            CheckAirPresureLight.SetActive (true);
        }
        else if (!OnOffBtn[0] || !AirTankCapacity)
        {
            CheckAirPresureLight.SetActive (false);
        }

        //Check belt on Light
        if (OnOffBtn[0])
        {
            CheckBeltOnLight.SetActive (true);
        }
        if (!OnOffBtn[0])
        {
            CheckBeltOnLight.SetActive (false);
        }

        //Check power supply Light
        if (OnOffBtn[0])
        {
            CheckElectricFeedLight.SetActive (true);
        }
        if (!OnOffBtn[0] /*&& AirTank.Capacity >=
AirTankModel.Parameters.MinCapacity*/)
        {
            CheckElectricFeedLight.SetActive (false);
        }

        //Check security Light
        if (!RearmBtn[0] || AlarmBtn1[0] || AlarmBtn2[0] || !OnOffBtn[0])
        {
            CheckSecurityLight.SetActive (false);
        }
        if (RearmBtn[0] && !AlarmBtn1[0] && !AlarmBtn2[0] && OnOffBtn[0])
        {
            CheckSecurityLight.SetActive (true);
        }

        //Piston Up Light
        if (StackerPistonUp && OnOffBtn[0])
        {
            UpLight.SetActive (true);
            DownLight.SetActive (false);
            PistonUpBtn[0] = false;
        }
        else if (!OnOffBtn[0])
        {
            UpLight.SetActive (false);
            PistonUpBtn[0] = false;
            PistonDownBtn[0] = false;
        }

        //Piston Down Light
        if (StackerPistonDown && OnOffBtn[0])
        {
            DownLight.SetActive (true);
            UpLight.SetActive (false);
            PistonDownBtn[0] = false;
        }

```

```

    }
    else if(!OnOffBtn[0])
    {
        DownLight.SetActive(false);
        PistonUpBtn[0] = false;
        PistonDownBtn[0] = false;
    }
    //Rotor Right Light
    if(StackerRotorRight && OnOffBtn[0])
    {
        RightLight.SetActive(true);
        LeftLight.SetActive(false);
        RotorRightBtn[0] = false;
    }
    else if (!OnOffBtn[0])
    {
        RightLight.SetActive(false);
        RotorRightBtn[0] = false;
        RotorLeftBtn[0] = false;
    }
    //Rotor Left Light
    if(StackerRotorLeft && OnOffBtn[0])
    {
        RightLight.SetActive(false);
        LeftLight.SetActive(true);
        RotorLeftBtn[0] = false;
    }
    else if (!OnOffBtn[0])
    {
        LeftLight.SetActive(false);
        RotorRightBtn[0] = false;
        RotorLeftBtn[0] = false;
    }
    //Previous stopper 1 Light
    if(StackerSensor1Up && OnOffBtn[0])
    {
        Sensor1Light.SetActive(true);
        UpStackerSensor1Light.SetActive(true);
        DownStackerSensor1Light.SetActive(false);
    }
    if(StackerSensor1Down || !OnOffBtn[0])
    {
        Sensor1Light.SetActive(false);
        UpStackerSensor1Light.SetActive(false);
        DownStackerSensor1Light.SetActive(true);
    }
    //inductive sensor stopper1 light
    if(InductiveSensorStacker1)
    {
        InductiveSensorStacker1Light.SetActive(true);
    }
    else
    {
        InductiveSensorStacker1Light.SetActive(false);
    }
    //Previous stopper 2 Light
    if(StackerSensor2Up && OnOffBtn[0])
    {
        Sensor2Light.SetActive(true);
        UpStackerSensor2Light.SetActive(true);
        DownStackerSensor2Light.SetActive(false);
    }
    if(StackerSensor2Down || !OnOffBtn[0])
    {
        Sensor2Light.SetActive(false);
        UpStackerSensor2Light.SetActive(false);
        DownStackerSensor2Light.SetActive(true);
    }
}

```

```
//inductive sensor stopper2 light
if(InductiveSensorStacker2)
{
    InductiveSensorStacker2Light.SetActive(true);
}
else
{
    InductiveSensorStacker2Light.SetActive(false);
}
//Store Light
if(StoreBtn[0])
{
    StoreLight.SetActive(true);
}
if(!StoreBtn[0])
{
    StoreLight.SetActive(false);
}
//Take Out Light
if(TakeOutBtn[0])
{
    TakeOutLight.SetActive(true);
}
if(!TakeOutBtn[0])
{
    TakeOutLight.SetActive(false);
}
//Alarm Light
if(AlarmBtn1[0] || AlarmBtn2[0])
{
    AlarmLight.SetActive(true);
}
if(!AlarmBtn1[0] && !AlarmBtn2[0])
{
    AlarmLight.SetActive(false);
}
//On-Off Light
if(OnOffBtn[0])
{
    OnLight.SetActive(true);
}
if(!OnOffBtn[0])
{
    OnLight.SetActive(false);
}
//Manual Mode Light
if(ModeBtn[0] && OnOffBtn[0])
{
    ManualLight.SetActive(true);
}
if(!ModeBtn[0] || !OnOffBtn[0])
{
    ManualLight.SetActive(false);
}
//Auto Mode Light
if(ModeBtn[1] && OnOffBtn[0])
{
    AutoLight.SetActive(true);
}
if(!ModeBtn[1] || !OnOffBtn[0])
{
    AutoLight.SetActive(false);
}
}
}
```

Script 14. Controller Locomotion.

```

using System.Collections;
using UnityEngine;
using System.Collections.Generic;
using UnityEngine.XR.Interaction.Toolkit;

public class ControllerLocomotion : MonoBehaviour
{
    public XRController leftTeleportRay;
    public XRController rightTeleportRay;
    public InputHelpers.Button teleportActivationButton;
    public float activationThreshold = 0.1f;
    public bool EnableLeftTeleport { get; set; } = true;
    public bool EnableRightTeleport { get; set; } = true;

    void Start() {
        InitializeTeleportRay(leftTeleportRay);
        InitializeTeleportRay(rightTeleportRay);
    }

    void InitializeTeleportRay(XRController teleportRay) {
        if(!teleportRay) { return; }

        teleportRay.gameObject.SetActive(false);
    }

    void Update()
    {
        if(leftTeleportRay)
        {
            leftTeleportRay.gameObject.SetActive(EnableLeftTeleport &&
            CheckIfActivated(leftTeleportRay));
        }
        if(rightTeleportRay)
        {
            rightTeleportRay.gameObject.SetActive(EnableRightTeleport &&
            CheckIfActivated(rightTeleportRay));
        }
    }

    public bool CheckIfActivated(XRController controller)
    {
        InputHelpers.IsPressed(controller.inputDevice,
        teleportActivationButton, out bool isActivated, activationThreshold);
        return isActivated;
    }
}

```

Script 15. Seguidor de cámara VR.

```

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

public class FollowCamera : MonoBehaviour
{
    [SerializeField] private Transform target;
    void Update()
    {
        transform.LookAt(target);
    }
}

```

Script 16. Information Controller.

```

using System.Collections;
using UnityEngine;
using System.Collections.Generic;
using UnityEngine.XR.Interaction.Toolkit;

public class InformationController : MonoBehaviour
{
    public XRController leftTeleportRay;
    public XRController rightTeleportRay;
    public InputHelpers.Button teleportActivationButton;
    public float activationThreshold = 0.1f;
    public bool EnableLeftTeleport { get; set; } = true;
    public bool EnableRightTeleport { get; set; } = true;
    void Start() {
        InitializeTeleportRay(leftTeleportRay);
        InitializeTeleportRay(rightTeleportRay);
    }

    void InitializeTeleportRay(XRController teleportRay) {
        if(!teleportRay) { return; }

        teleportRay.gameObject.SetActive(false);
    }

    void Update()
    {
        if(leftTeleportRay)
        {
            leftTeleportRay.gameObject.SetActive(EnableLeftTeleport &&
            CheckIfActivated(leftTeleportRay));
        }
        if(rightTeleportRay)
        {
            rightTeleportRay.gameObject.SetActive(EnableRightTeleport &&
            CheckIfActivated(rightTeleportRay));
        }
    }
    public bool CheckIfActivated(XRController controller)
    {
        InputHelpers.IsPressed(controller.inputDevice,
        teleportActivationButton, out bool isActivated, activationThreshold);
        return isActivated;
    }
}

```

Script 17. Gráficas Sistema SCADA.

```

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

public class Window_graph : MonoBehaviour
{
    private bool isBarChart = true;
    [SerializeField] private Sprite dotSprite;
    private RectTransform GraphContainer;
    private RectTransform labelTemplateX;
    private RectTransform labelTemplateY;
    private RectTransform dashTemplateX;
    private RectTransform dashTemplateY;
    private List<GameObject> gameObjectList;
}

```

```

// Cached values
private List<int> valueList;
private IGraphVisual lineGraphVisual;
private IGraphVisual barChartVisual;
private int minVisAmount;
private int maxVisAmount;
private Func<int, string> getAxisLabelX;
private Func<float, string> getAxisLabelY;

public void Awake()
{
    GraphContainer =
transform.Find("GraphContainer").GetComponent<RectTransform>();
    labelTemplateX =
GraphContainer.Find("labelTemplateX").GetComponent<RectTransform>();
    labelTemplateY =
GraphContainer.Find("labelTemplateY").GetComponent<RectTransform>();
    dashTemplateX =
GraphContainer.Find("dashTemplateX").GetComponent<RectTransform>();
    dashTemplateY =
GraphContainer.Find("dashTemplateY").GetComponent<RectTransform>();

    gameObjectList = new List<GameObject>();
    getAxisLabelX = (int _i) => (_i + 1) + " h";
    getAxisLabelY = (float _f) => Mathf.RoundToInt(_f) + " w";

    valueList = new List<int>() { 5, 98, 23, 44, 68, 34, 52, 54, 63, 64,
55, 50 };

    lineGraphVisual = new LineGraphVisual(GraphContainer, dotSprite,
Color.green, new Color(1, 1, 1, 0.5f));
    barChartVisual = new BarChartVisual(GraphContainer, Color.green,
0.8f);

    ShowGraph(this.valueList, lineGraphVisual, 0, valueList.Count,
this.getAxisLabelX, this.getAxisLabelY);
}

//Buttons functions
public void SetGraph(bool isBarChart)
{
    if (isBarChart)
    {
        this.isBarChart = isBarChart;
        ShowGraph(this.valueList, barChartVisual, this.minVisAmount,
this.maxVisAmount, this.getAxisLabelX, this.getAxisLabelY);
    } else
    {
        this.isBarChart = isBarChart;
        ShowGraph(this.valueList, lineGraphVisual, this.minVisAmount,
this.maxVisAmount, this.getAxisLabelX, this.getAxisLabelY);
    }
}

public void IncreaseStartValue ()
{
    if (minVisAmount > 0)
    {
        minVisAmount--;
        SetGraph(isBarChart);
    }
}

public void DecreaseStartValue ()
{
    if (minVisAmount <= maxVisAmount)
    {

```

```

        minVisAmount++;
        SetGraph(isBarChart);
    }
}

public void DecreaseEndValue ()
{
    if (maxVisAmount > minVisAmount + 1)
    {
        maxVisAmount--;
        SetGraph(isBarChart);
    }
}

public void IncreaseEndValue ()
{
    if (maxVisAmount < valueList.Count)
    {
        maxVisAmount++;
        SetGraph(isBarChart);
    }
}

private void CleanGraph()
{
    // Clean up old graph before drawing new one
    foreach (GameObject gameObject in gameObjectList)
    {
        Destroy(gameObject);
    }
    gameObjectList.Clear();
}

private void ShowGraph(List<int> valueList, IGraphVisual graphVisual, int
minVisAmount = 0, int maxVisAmount = 5, Func<int, string> getAxisLabelX =
null, Func<float, string> getAxisLabelY = null)
{
    this.valueList = valueList;
    this.GetAxisLabelX = getAxisLabelX;
    this.GetAxisLabelY = getAxisLabelY;

    // Set default label values
    if (getAxisLabelX == null)
    {
        getAxisLabelX = delegate (int _i) { return _i.ToString(); };
    }
    if (getAxisLabelY == null)
    {
        getAxisLabelY = delegate (float _f) { return
Mathf.RoundToInt(_f).ToString(); };
    }

    // if value is invalid display whole graph
    if (minVisAmount < 0)
    {
        minVisAmount = 0;
    } else if (minVisAmount >= maxVisAmount)
    {
        minVisAmount = maxVisAmount - 1;
    }
    if (maxVisAmount <= 0 || maxVisAmount > valueList.Count)
    {
        maxVisAmount = valueList.Count;
    } else if (maxVisAmount < minVisAmount)
    {
        maxVisAmount = minVisAmount + 1;
    }
    this.minVisAmount = minVisAmount;
}

```



```

this.maxVisAmount = maxVisAmount;

CleanGraph();

// Set boundaries
float graphWidth = GraphContainer.sizeDelta.x;
float graphHeight = GraphContainer.sizeDelta.y;
float yMax = valueList[0];
float yMin = valueList[0];

// Cycle through all our values and find the highest value
for (int i = minVisAmount; i < maxVisAmount; i++)
{
    int value = valueList[i];
    if (value > yMax)
    {
        yMax = value;
    }
    if (value < yMin)
    {
        yMin = value;
    }
}

float yDiff = yMax - yMin;
if (yMax - yMin <= 0)
{
    yDiff = 5f;
}
if (yMax + yDiff * 0.2f <= 100f) // Scale the graph without letting it
go above 100%
{
    yMax += yDiff * 0.2f;
} else
{
    yMax = 100f;
}
if (yMin - yDiff * 0.2f >= 0f) // Scale the graph without letting it
go below 0%
{
    yMin -= yDiff * 0.2f;
} else
{
    yMin = 0f;
}

// Plot the horizontal points and label them
float xSize = graphWidth / ((maxVisAmount - minVisAmount) + 1);
int xIndex = 0;

for (int i = minVisAmount; i < maxVisAmount; i++) {
    float xPosition = xSize + xIndex * xSize;
    float yPosition = ((valueList[i] - yMin) / (yMax - yMin)) *
graphHeight; // Normalized Y position

    gameObjectList.AddRange(graphVisual.AddGraphVisual(new
Vector2(xPosition, yPosition), xSize));

    RectTransform labelX = Instantiate(labelTemplateX);
    labelX.transform.SetParent(GraphContainer);
    labelX.gameObject.SetActive(true);
    labelX.anchoredPosition = new Vector3(xPosition, -7f, 15f);
    labelX.GetComponent<Text>().text = getAxisLabelX(i);
    gameObjectList.Add(labelX.gameObject);

    RectTransform dashX = Instantiate(dashTemplateX);
    dashX.SetParent(GraphContainer, false);
    dashX.gameObject.SetActive(true);

```

```

        dashX.anchoredPosition = new Vector2(xPosition, -6.2f);
        gameObjectList.Add(dashX.gameObject);

        xIndex++;
    }

    // Connect the points and create the vertical labels
    int sepCount = 10;
    for (int i = 0; i <= sepCount; i++)
    {
        float normValue = i * 1f / sepCount;

        RectTransform labelY = Instantiate(labelTemplateY);
        labelY.transform.SetParent(GraphContainer);
        labelY.gameObject.SetActive(true);
        labelY.anchoredPosition = new Vector2(-9f, normValue *
graphHeight);
        labelY.GetComponent<Text>().text = getAxisLabelY(yMin + (normValue
* (yMax - yMin)));
        gameObjectList.Add(labelY.gameObject);

        RectTransform dashY = Instantiate(dashTemplateY);
        dashY.SetParent(GraphContainer, false);
        dashY.gameObject.SetActive(true);
        dashY.anchoredPosition = new Vector2(-9.5f, normValue *
graphHeight);
        gameObjectList.Add(dashY.gameObject);
    }
}

private interface IGraphVisual
{
    List<GameObject> AddGraphVisual(Vector2 graphPos, float
graphPosWidth);
}

private class BarChartVisual : IGraphVisual
{
    private RectTransform graphContainer;
    private Color barColor;
    private float barWidthMultiplier;

    public BarChartVisual(RectTransform graphContainer, Color barColor,
float barWidthMultiplier)
    {
        this.graphContainer = graphContainer;
        this.barColor = barColor;
        this.barWidthMultiplier = barWidthMultiplier;
    }

    public List<GameObject> AddGraphVisual (Vector2 graphPos, float
graphPosWidth)
    {
        GameObject barGameObject = CreateBar(graphPos, graphPosWidth);

        return new List<GameObject>() { barGameObject };
    }

    private GameObject CreateBar(Vector2 graphPos, float barWidth)
    {
        GameObject gameObject = new GameObject("bar", typeof(Image));
        gameObject.transform.SetParent(graphContainer, false);
        gameObject.GetComponent<Image>().color = barColor;
        RectTransform rectTransform =
gameObject.GetComponent<RectTransform>();
        rectTransform.anchoredPosition = new Vector2(graphPos.x, 0f);
        rectTransform.sizeDelta = new Vector2(barWidth *
barWidthMultiplier, graphPos.y);
    }
}

```

```

        rectTransform.anchorMax = new Vector2(0, 0);
        rectTransform.anchorMin = new Vector2(0, 0);
        rectTransform.pivot = new Vector2(.5f, 0f);

        return gameObject;
    }
}

private class LineGraphVisual : IGraphVisual
{
    private RectTransform graphContainer;
    private Sprite dotSprite;
    private GameObject lastDotGameObject;
    private Color dotColor;
    private Color dotConnectionColor;

    public LineGraphVisual (RectTransform graphContainer, Sprite
dotSprite, Color dotColor, Color dotConnectionColor)
    {
        this.graphContainer = graphContainer;
        this.dotSprite = dotSprite;
        this.dotColor = dotColor;
        this.dotConnectionColor = dotConnectionColor;
        lastDotGameObject = null;
    }

    public List<GameObject> AddGraphVisual(Vector2 graphPos, float
graphPosWidth)
    {
        List<GameObject> gameObjectList = new List<GameObject>();
        GameObject dotGameObject = CreateDot(graphPos);
        gameObjectList.Add(dotGameObject);

        if (lastDotGameObject != null &&
dotGameObject.GetComponent<RectTransform>().anchoredPosition.x >
lastDotGameObject.GetComponent<RectTransform>().anchoredPosition.x)
        {
            GameObject dotConnection =
CreateDotConnection(lastDotGameObject.GetComponent<RectTransform>().anchoredPo
sition, dotGameObject.GetComponent<RectTransform>().anchoredPosition);
            gameObjectList.Add(dotConnection);
        }
        lastDotGameObject = dotGameObject;

        return gameObjectList;
    }

    private GameObject CreateDot(Vector2 anchorPos)
    {
        GameObject gameObject = new GameObject("dot", typeof(Image));
        gameObject.transform.SetParent(graphContainer, false);
        gameObject.GetComponent<Image>().sprite = dotSprite;
        gameObject.GetComponent<Image>().color = dotColor;
        RectTransform rectTransform =
gameObject.GetComponent<RectTransform>();
        rectTransform.anchoredPosition = anchorPos;
        rectTransform.sizeDelta = new Vector2(0.2f, 0.2f);
        rectTransform.anchorMax = new Vector2(0, 0);
        rectTransform.anchorMin = new Vector2(0, 0);

        return gameObject;
    }

    private GameObject CreateDotConnection(Vector2 dotPosA, Vector2
dotPosB)
    {
        GameObject gameObject = new GameObject("dotConnection",
typeof(Image));
        gameObject.transform.SetParent(graphContainer, false);

```

```
        gameObject.GetComponent<Image>().color = dotConnectionColor;
        RectTransform rectTransform =
gameObject.GetComponent<RectTransform>();
        Vector2 dir = (dotPosB - dotPosA).normalized;
        float distance = Vector2.Distance(dotPosA, dotPosB);

        rectTransform.anchorMin = new Vector2(0, 0);
        rectTransform.anchorMax = new Vector2(0, 0);
        rectTransform.sizeDelta = new Vector2(distance, 0.1f);
        rectTransform.anchoredPosition = dotPosA + dir * distance * 0.5f;
        rectTransform.localEulerAngles = new Vector3(0, 0,
GetAngleFromVectorFloat(dir));

        return gameObject;
    }
}

public static float GetAngleFromVectorFloat(Vector3 dir)
{
    dir = dir.normalized;
    float n = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
    if (n < 0) n += 360;

    return n;
}
}
```

REFERENCIAS

- [1] Herranz, Arantxa. Digital twins: qué son, para qué sirven y cuáles son los beneficios y problemas de los gemelos digitales. Xataka. 26 de mayo de 2021. Disponible On-line: <https://www.xataka.com/pro/digital-twins-que-sirven-cuales-beneficios-problemas-gemelos-digitales>
- [2] Earls, Alan. Por qué las empresas no deberían ignorar la Realidad Virtual. DealerWorld. 25 de abril de 2022. Disponible On-line: <https://www.dealerworld.es/tendencias/por-que-las-empresas-no-deberian-ignorar-la-realidad-virtual>
- [3] Proyecto DENiM. Comision europea, Cordis, 2020. Disponible On-line: <https://cordis.europa.eu/project/id/958339/es>
- [4] Erosa García, David. ¿Qué es Unity? OpenWebinars. 1º de junio de 2019. Disponible On-line: <https://openwebinars.net/blog/que-es-unity/>
- [5] Asensio, Iván. Qué es Unity y para qué sirve. *MasterD*. Disponible On-line: <https://www.mas-terd.es/blog/que-es-unity-3d-tutorial>
- [6] GameObjects. Unity Documentation. 1 de agosto de 2017. Disponible On-line: <https://docs.unity3d.com/es/2018.4/Manual/GameObjects.html>
- [7] Zohair Mustafeez, Anusheh. What is Visual Studios Code? Educative. Disponible On-line: <https://www.educative.io/edpresso/what-is-visual-studio-code>
- [8] Qué es 3D Studio Max y para qué sirve. La Educación en la Era Digital. 1 de julio de 2019. Disponible On-line: <https://ayto-torrijos.com/herramientas/que-es-3d-studio-max-y-para-que-sirve/>
- [9] Creating and using scripts. Unity Documentation. 19 de marzo del 2018. Disponible On-line: <https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html>
- [10] Visual Studio Code ¿Qué es? y ¿Qué no es? ITpro. 22 de agosto de 2016. Disponible On-line: <https://blogs.itpro.es/eduardocloud/2016/08/22/visual-studio-code-que-es-y-que-no>

- [11] C# (C Sharp). Lenguajes de programación. Disponible On-line: <https://lenguajesdeprogramacion.net/c-sharp/>
- [12] ¿Qué es MonoBehaviour en Unity 3D? ForoAyuda. Disponible On-line: <https://foroayuda.es/que-es-monobehaviour-en-unity3d/#:~:text=se%20describe%20com%3A,MonoBehaviour%20es%20la%20clase%20base%20de%20la%20que%20deriva%20cada,tenga%20que%20hacerlo%20usted%20mismo>
- [13] Importar objetos desde 3D Studio Max. Unity Documentation. 5 de junio de 2020. Disponible On-line: <https://docs.unity3d.com/es/530/Manual/HOWTO-ImportObjectMax.html>
- [14] Gómez Jiménez, Juan. Trabajo Fin de Master, Universidad de Sevilla, 2021. “Diseño de gemelo digital del sistema de transporte de célula de fabricación flexible”.
- [15] Gómez Jiménez, Javi. Trabajo Fin de Master, Universidad de Sevilla, 2021. “Metodología para el desarrollo de gemelos digitales. Aplicación a célula de fabricación flexible”.
- [16] Usar el modo desarrollador. Oculus Support. Disponible On-line: https://business.Oculus.com/support/1310318635799580/?locale=es_ES
- [17] García, José. Oculus Quest 2, análisis: una de las mejores (y asequibles) opciones para iniciarse en la realidad virtual. Xataka. 17 de octubre de 2020. Disponible On-line: <https://www.xataka.com/analisis/oculus-quest-2-analisis-caracteristicas-precio-especificaciones>
- [18] Alejandro VR. Tutorial de Oculus Link para mejorar Oculus Quest 2. AVR. 20 de noviembre de 2020. Disponible On-line: <https://alehandorvr.com/tutorial-oculus-link-en-oculus-quest-2>
- [19] Alejandro VR. Air Link conecta tus Quest 2 con tu PC sin cable. AVR. 27 de abril de 2021. Disponible On-line: <https://alehandorvr.com/air-link-la-forma-inalambrica-de-usar-oculus-quest-2>
- [20] Buckley, Daniel. Unity XR Interaction Toolkit Tutorials – Complete Guide. GameDev Academy. 5 de enero de 2022. Disponible On-line: <https://gamedevacademy.org/unity-xr-interaction-toolkit-tutorial/>
- [21] ¿Cómo arreglo la rotación de un modelo importado? Oculus Support. Disponible On-line: <https://docs.unity3d.com/es/530/Manual/HOWTO-FixZAxisIsUp.html>
- [22] Castaño, Fernando. Informe Técnico de Automatización, Universidad de Sevilla, 2019. “Alimentador de bandejas automatizado en célula de fabricación flexible”
- [23] Como capturar tu pantalla en realidad virtual. Las cosas del espacio van despacio. 11 de mayo de 2021. Disponible On-line: <https://stormseeker.xyz/como-capturar-tu-pantalla-en-realidad-virtual/>
- [24] Qué es un Sistema SCADA, para qué sirve y cómo funciona. Aula 21. Disponible On-line: <https://www.cursosaula21.com/que-es-un-sistema-scada/>
- [25] King, Ryan. An introduction to digital twins. Rowse. 13 de diciembre de 2019. Disponible On-line: <https://www.rowse.co.uk/blog/post/an-introduction-to-digital-twins>
- [26] DENiM Challenges. DENiM official website. Disponible On-line: <https://denim-fof.eu/project/challenges-2/>
- [27] 5 empresas que están aplicando ya la realidad virtual en su estrategia. Tecnología para los negocios. Disponible On-line: <https://ticnegocios.camaravalencia.com/servicios/tendencias/5-empresas-que-estan-aplicando-ya-la-realidad-virtual-en-su-estrategia/>