TRABAJOS FIN DE GRADO

Universidad de Jaén

*Escuela Politécnica Superior de Jaén*

# ANÁLISIS DE LA HERRAMIENTA UNITY 3D PARA REALIZAR MODELOS DIGITALES EN AUTOMATIZACIÓN

Autor: Pedro Casado Cruz

Grado: Ingeniería Electrónica Industrial

Director: Prof. Dña. ELISABET ESTEVEZ ESTEVEZ

Departamento del director: Ingeniería Electrónica y Automática

Fecha: 18/06/2024

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Dña ELISABET ESTEVEZ ESTEVEZ , tutora del Proyecto Fin de Carrera titulado: ANÁLISIS DE LA HERRAMIENTA UNITY 3D PARA REALIZAR MODELOS DIGITALES EN AUTOMATIZACIÓN, que presenta PEDRO CASADO CRUZ, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Junio de 2024

El alumno:

Firmado por CASADO CRUZ PEDRO - 53915880Q el dï¿½a 18/06/2024 con un certificado emitido por AC FNMT Usuarios

PEDRO CASADO CRUZ

Los tutores:

ESTEVEZ ESTEVEZ ELISABET - 72578245C
Firmado digitalmente por ESTEVEZ ESTEVEZ ELISABET - 72578245C
Fecha: 2024.06.18 11:41:16 +02'00'

SANCHEZ GARCIA ALEJANDRO - 77321466C
Firmado digitalmente por SANCHEZ GARCIA ALEJANDRO - 77321466C
Fecha: 2024.06.18 12:55:34 +02'00'

ELISABET ESTEVEZ ESTEVEZ

ALEJANDRO SÁNCHEZ GARCÍA

# Índice

# Índice de Figuras

# 1. Resumen

El presente proyecto se enfoca en el análisis de un software de simulación de plantas industriales con capacidad de conexión con controladores industriales, lo que permite al programador interactuar con elementos de alto grado de interactividad y realismo.

La ejecución de este proyecto se basa en Unity 3D, una plataforma ampliamente reconocida en el ámbito del diseño de videojuegos, que es robusta y flexible para la creación de entornos virtuales interactivos. Su versatilidad se adapta perfectamente al propósito de crear Modelos Digitales para la Automatización, donde se enfoca en la reproducción fidedigna de una planta industrial.

Además, este proyecto explora la comunicación en tiempo real con sistemas reales mediante el protocolo de comunicación Modbus. Cuando la comunicación es unidireccional, se aborda el concepto de Sombra Digital. Por otro lado, cuando la comunicación es bidireccional, se discute el concepto de Gemelo Digital.

## 2. Introducción

Los motores gráficos, originalmente diseñados para el desarrollo de videojuegos, han experimentado una notable evolución en su estructura y funcionalidad. En las etapas iniciales del desarrollo de videojuegos, la falta de claridad en la diferenciación de los componentes resultó en una programación homogénea y poco estructurada. La aparición de juegos como Doom [1] en la década de los 90 marcó un hito al introducir una arquitectura que permitía la reutilización eficiente de componentes, dando origen al término “Game-Engine” [2].

Esta evolución ha llevado a los motores gráficos a trascender su función inicial, impactando no solo en el entretenimiento puro, sino también en aplicaciones para otros sectores, como la formación y entretenimiento a través de Serious Games o aplicaciones de Realidad Virtual enfocadas al sector industrial. Así, los entornos virtuales se han convertido en herramientas cruciales para la verificación del comportamiento de instalaciones, minimizando riesgos y detectando posibles errores sin afectar a componentes reales [3].

En este contexto, la simulación precisa de fuerzas en objetos es esencial. Dichas fuerzas que surgen en los objetos físicos de la instalación se realizan a través de lo que se conoce como motor físico, sistema de software que implementa una simulación aproximada de ciertos sistemas físicos, incluyendo dinámica de sólidos rígidos, detección de colisiones y dinámica de fluidos en tiempo real. Implementar un motor físico no es tarea sencilla, siendo un campo de investigación continuo para mejorar la fidelidad al comportamiento real de los objetos.

Explorando esta evolución y adaptación de los motores gráficos, surge el concepto de Gemelo Digital. Este enfoque implica establecer una comunicación en tiempo real bidireccional entre el sistema físico real y su correspondiente modelo virtual. Los Gemelos Digitales, al incorporar motores gráficos avanzados y motores físicos precisos, se fundan como herramientas valiosas en la optimización y validación de procesos industriales, brindando un enfoque innovador para la ingeniería y automatización.

## 2.1. Motivación

El principal incentivo de este proyecto ha sido la exploración de las capacidades de Unity 3D como herramienta para la creación de Gemelos Digitales, aprovechando su enfoque innovador en la simulación 3D para la optimización y análisis de sistemas industriales.

El Gemelo Digital es un concepto emergente en la ingeniería y la automatización industrial, que se fundamenta en la creación de réplicas virtuales precisas de sistemas físicos y procesos en tiempo real. Esta duplicación digital representa un objeto o sistema del mundo real en un entorno virtual, permitiendo su simulación y monitoreo. La clave del Gemelo Digital reside en la sincronización continua entre el objeto físico y su versión digital, lo que habilita la retroalimentación bidireccional y la actualización en tiempo real.

Por otro lado, la posibilidad de diseñar un entorno de uso gratuito y accesible para todos ha sido un factor adicional en la motivación de este proyecto.

## 2.2. Objetivo

Se llevará a cabo una evaluación para determinar la efectividad y viabilidad de Unity 3D como herramienta para crear Gemelos Digitales en el ámbito de la automatización industrial.

En este proyecto, el enfoque se centrará en analizar la capacidad de Unity 3D para recrear con precisión una planta real, explorando cómo replicar acciones y la relación entre la planta física y su representación digital.

El objetivo principal es desarrollar un proyecto final completamente funcional, donde todos los elementos estén interconectados de manera que la planta digital se asemeje lo más posible a la planta real. Esto permitirá su uso en investigación, enseñanza y otros ámbitos.

Además, se busca brindar apoyo en el manejo de Unity 3D, asegurando que los pasos a seguir estén claros y ayuden a reducir la dificultad inicial, que es uno de los principales desafíos en este proceso.

## 2.3. Estructura

Este documento está estructurado en varias secciones.

En primer lugar, se presenta una introducción a Unity 3D, que abarca su proceso de instalación, interfaz, el lenguaje de programación utilizado y el uso del programa de manera básica.

Posteriormente, se aborda el tema del modelo digital, donde se exploran los elementos relacionados con la automatización en Unity y como integrarlos.

A continuación, se introduce el concepto de Sombra Digital, donde se profundiza en los elementos mencionados en otras secciones, pero añade la comunicación Modbus y cómo estos elementos reciben y procesan la información correspondiente.

Concluyendo, se abordará el Gemelo Digital, donde se incorporará la bidireccionalidad en la comunicación, representando así la fase final del proyecto.

Además, se incluye un anexo detallado sobre Modbus, que abarca unos conceptos básicos y cómo funciona en Unity, entre otras cosas.

# 3. Unity 3D

La simulación 3D en el ámbito de la automatización industrial ha experimentado un crecimiento significativo, siendo crucial para la optimización de procesos y la reducción de costos. Por esta razón, han surgido varios entornos de simulación 3D diseñados para abordar los desafíos de replicar con precisión entornos industriales complejos.

En este proyecto se empleará Unity 3D, un motor de juego multiplataforma ampliamente reconocido en la industria del desarrollo de videojuegos y aplicaciones interactivas. Sus capacidades para establecer conexiones Modbus, su interfaz gráfica intuitiva y, sobre todo, su disponibilidad gratuita en su forma básica son características destacables a considerar. Además, la idea de poder obtener una simulación de una planta industrial sencilla en conexión con la planta real, partiendo de un entorno originalmente asociado a los videojuegos, resulta especialmente atractiva.

En esta sección, se presentarán los pasos esenciales para descargar e instalar Unity de manera sencilla. También se abordará el proceso de creación de un proyecto desde cero y se explorará la interfaz de Unity para lograr una mejor comprensión de las herramientas necesarias. Prosiguiendo con una introducción al lenguaje de programación utilizado y con unos principios esenciales para el inicio de un proyecto. Esto se hace con el objetivo de suavizar la curva de aprendizaje y facilitar la familiarización con el entorno de desarrollo. También se detallarán los componentes más importantes para lograr una representación realista y funcional de los objetos. Se explorará la función que desempeña el ‘Canvas’ en nuestro proyecto y cómo se crea. También se verá los diferentes elementos propios de Unity que pertenecen al Canvas, como los botones.

## 3.1. Descarga e Instalación de Unity 3D

Inicialmente, se accederá al sitio web oficial de Unity para efectuar la descarga, disponible en el enlace: https://unity.com/es [4]. Mientras se descarga la versión de Unity Hub para Windows, se procederá a la creación de una cuenta de Unity ID, necesaria para el registro mediante la dirección de correo electrónico correspondiente, tal como se muestra en la Figura 1.

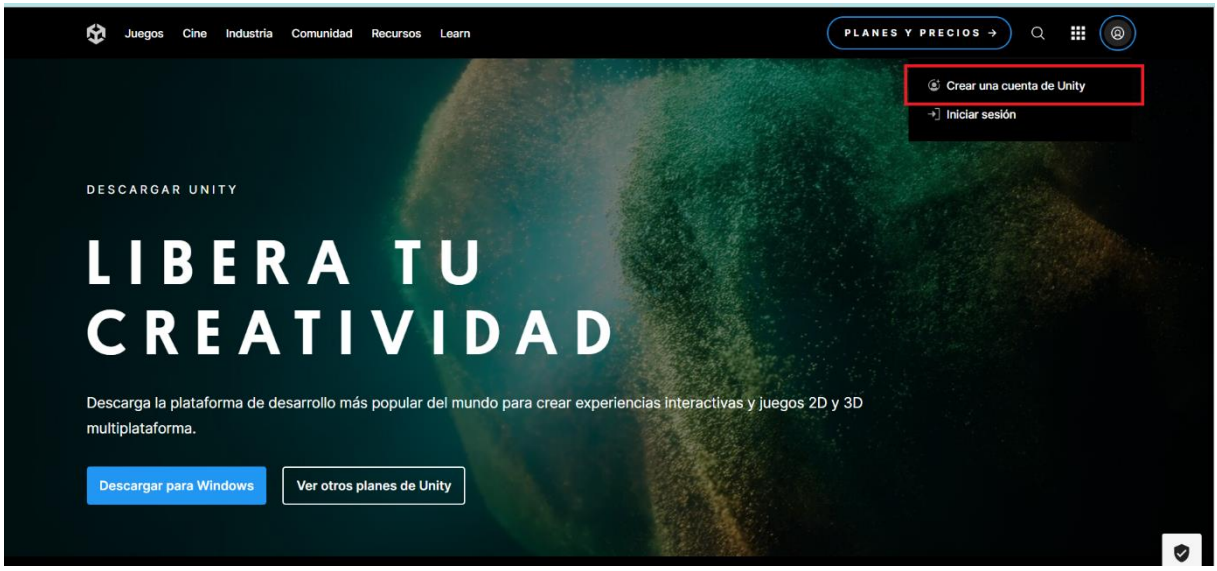



**Figura 1 Creación de la cuenta Unity**

Una vez completado el registro y descargado el archivo, se ejecutará para llevar a cabo la instalación. Durante este proceso, se abrirá un instalador que requerirá simplemente seleccionar la ubicación de la carpeta donde se instalará Unity Hub, desde la cual se accederá a los proyectos creados.

Tras finalizar la instalación y al iniciar Unity Hub, se deberá instalar las versiones de Unity necesarias para cada proyecto individual. La ruta de acceso pata llevar a cabo este proceso se muestra en la Figura 2.

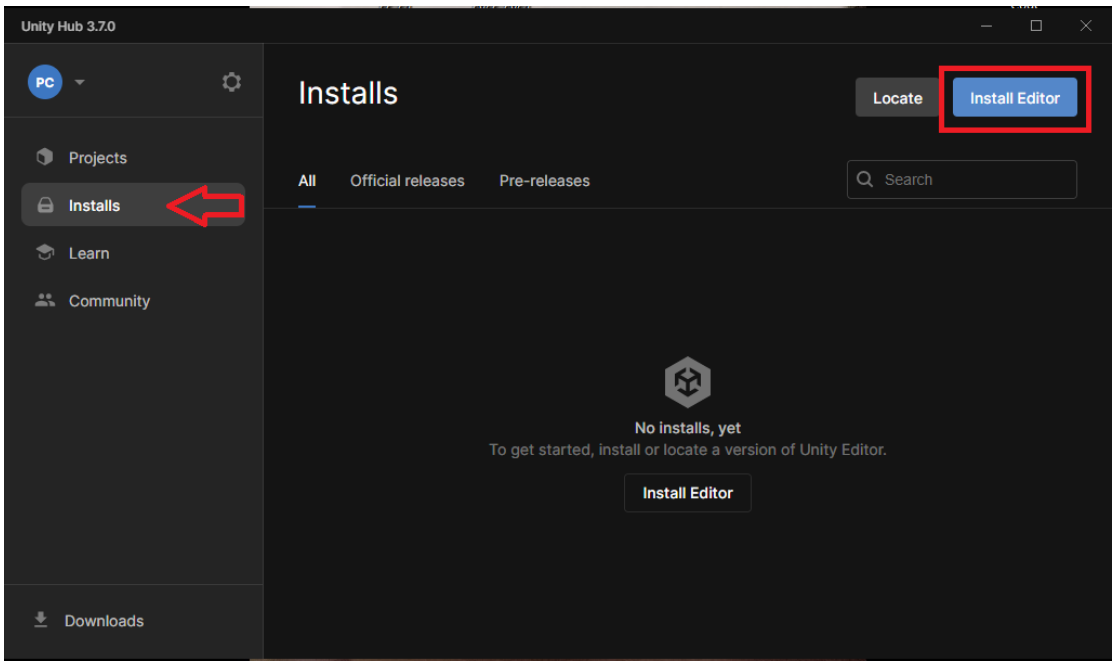



**Figura 2 Acceso para instalar las versiones**

Al seleccionar “Install Editor”, se desplegará una lista de versiones (Figura 3), entre las cuales se elegirá la versión recomendada que posea la etiqueta LTS, conocida por ser más estable y recibir un mayor respaldo del equipo de Unity, garantizando así una mayor seguridad. Para este proyecto se utilizó la versión 2022.3.10f1 LTS. Sin embargo, es posible consultar todas las versiones disponibles para su descarga en la página oficial de Unity.

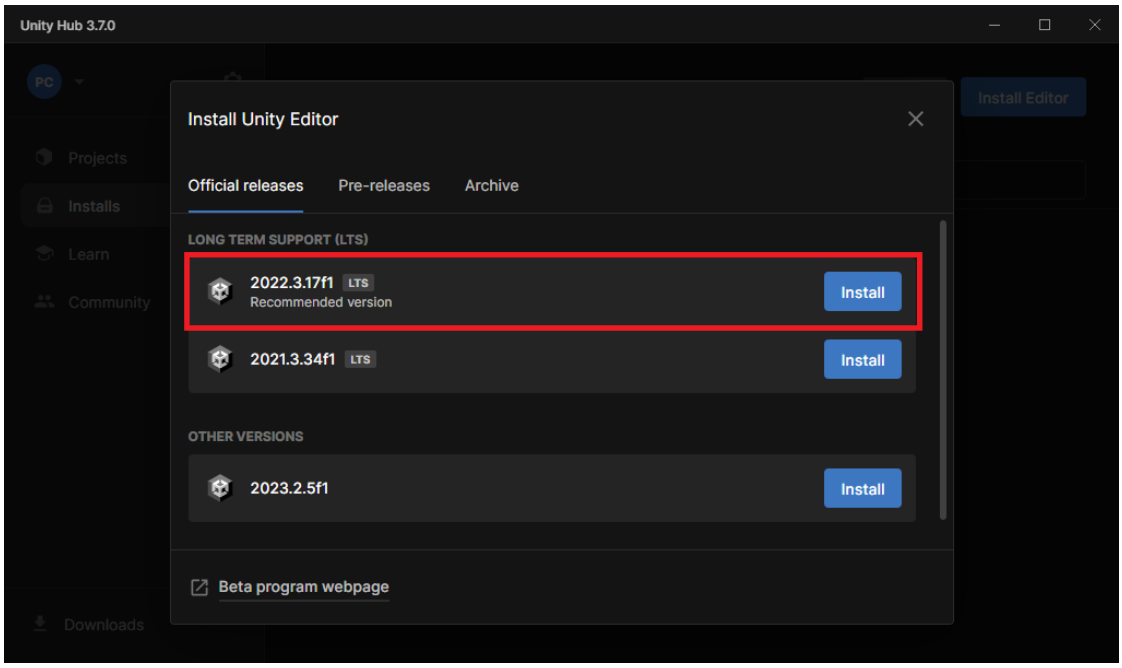



**Figura 3 Versión de Unity**

## 3.2. Creación de un nuevo Proyecto

Una vez que Unity esté instalado en el sistema, el siguiente paso es iniciar un nuevo proyecto. En el Unity Hub, navega hacia la sección de “Projects” y selecciona la opción “New Project“. Al hacer clic en esta opción, se abrirá una ventana con una variedad de plantillas disponibles, tal como se muestra en la Figura 4. Entre las opciones disponibles, se encuentran recursos de aprendizaje en la sección “Learning”, que incluyen cursos y tutoriales predefinidos diseñados para familiarizarte con el motor de Unity.

Para este proyecto en particular, se debe elegir la plantilla “3D Core”. Esta plantilla proporciona un proyecto 3D básico que utiliza el renderizado integrado de Unity. Una vez seleccionada la plantilla, se debe asegurar elegir la versión del editor

de Unity que se ha instalado previamente. Luego, se asigna un nombre al proyecto y se especifica la ubicación donde se desean guardar los archivos del proyecto.

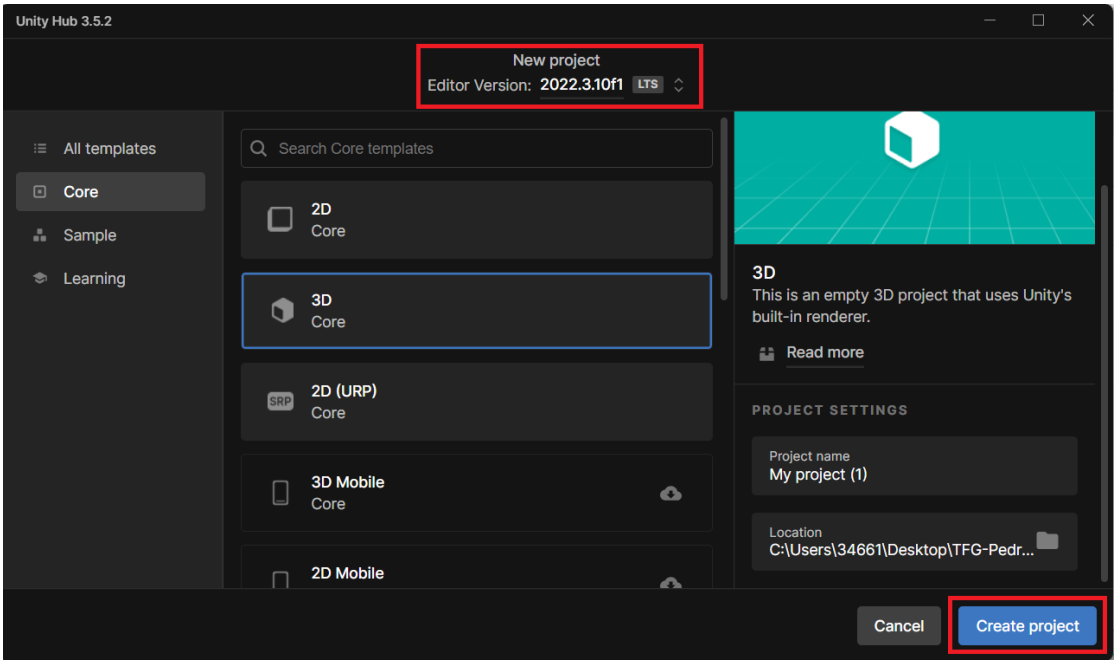



**Figura 4 Ventana de Creación del Proyecto**

## 3.3. Interfaz del Editor

La Interfaz de Unity [5] es totalmente personalizable según el gusto del usuario, pero prácticamente se divide en 5 vistas principales como se muestra en la Figura 5:

1.Jerarquía: La jerarquía revela la estructura de cómo los objetos están agrupados el uno al otro. La ventana de jerarquía es una representación de texto jerárquico de cada objeto en la escena. Cada elemento en la escena tiene una entrada en la jerarquía, por lo que las dos ventanas están inherentemente vinculadas.

2.Vista de Escena: La vista de escena proporciona una navegación visual y permite la edición de la escena actual. Esta vista puede mostrar una perspectiva 2D o 3D, según el tipo de proyecto en el que se esté trabajando. También se puede alternar a la Vista de Juego, donde se muestra el renderizado generado por la(s) cámara(s) en su juego.

3.Inspector: Esta herramienta permite la visualización y edición de todas las propiedades del objeto actualmente seleccionado. Ya que diferentes objetos tienen diferentes propiedades, el layout (diseño) y contenido de la ventana del inspector variarán en consecuencia.

4.Ventana del Proyecto: Muestra los activos (assets) de su librería que están disponibles para ser utilizados en el proyecto. Los activos importados en todo proyecto aparecen en esta vista facilitando así su gestión y uso.

5.Consola: Muestra errores, advertencias y otros mensajes generados por Unity. Para ayudar con la depuración (debugging), también puede mostrar sus propios mensajes en la Consola utilizando las funciones Debug.Log, Debug.LogWarning y Debug.LogError.

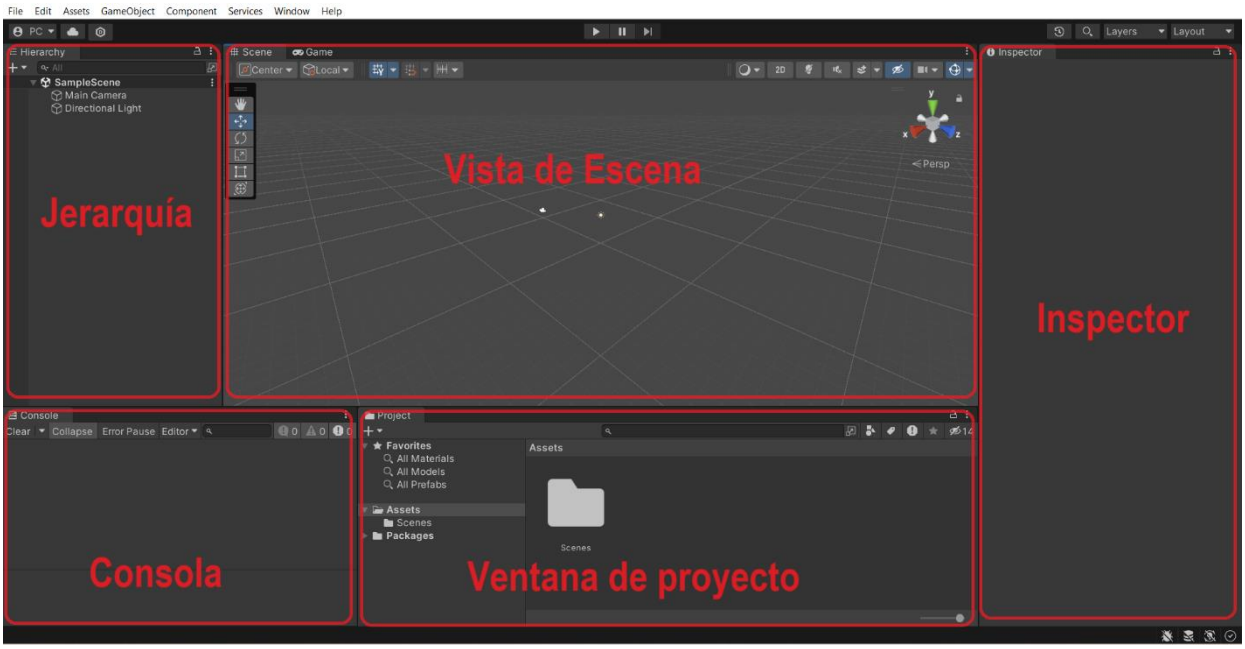



**Figura 5 Partes de la Interfaz de Unity**

## 3.4. Lenguaje de Programación

C# (C Sharp) se destaca como el lenguaje de programación principal utilizado en Unity, desarrollado por Microsoft para su plataforma .NET. A diferencia de otros lenguajes en .NET, C# está específicamente adaptado para esta plataforma, eliminando elementos heredados innecesarios y simplificando el proceso de programación.

La sintaxis y estructura de C# comparten similitudes con C++, lo que facilita la migración de códigos y el aprendizaje para desarrolladores familiarizados con estos lenguajes. Aunque también se asocia con la sencillez y alta productividad de Visual Basis, C# se distingue por integrar lo mejor de lenguajes como Java, incorporando modificaciones para optimizar el desarrollo de componentes.

En conclusión, C# emerge como un lenguaje de programación que incorpora los aspectos positivos y ventajas de otros lenguajes existentes como Visual Basic, Java o C++, y los fusiona en uno solo.

## 3.5. Principios Esenciales para Iniciar el Proyecto

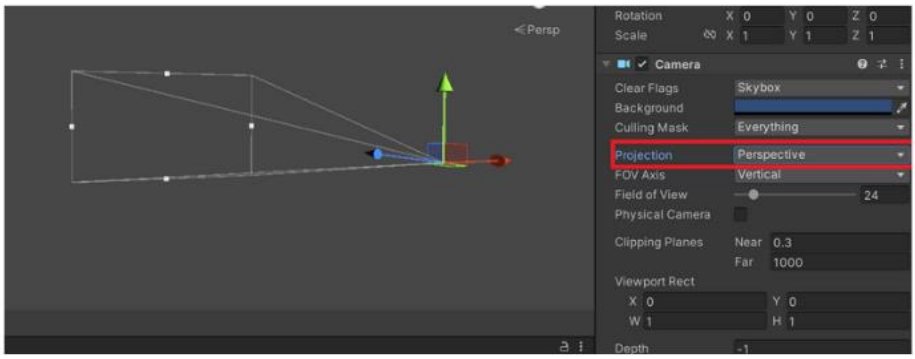
En esta sección se abordan algunas de las acciones fundamentales pero simples, como la Cámara principal del Unity, la creación de objetos y la elaboración de scripts, así como el proceso de depuración del código.

### 3.5.1. Aspectos de la ‘Main Camera’

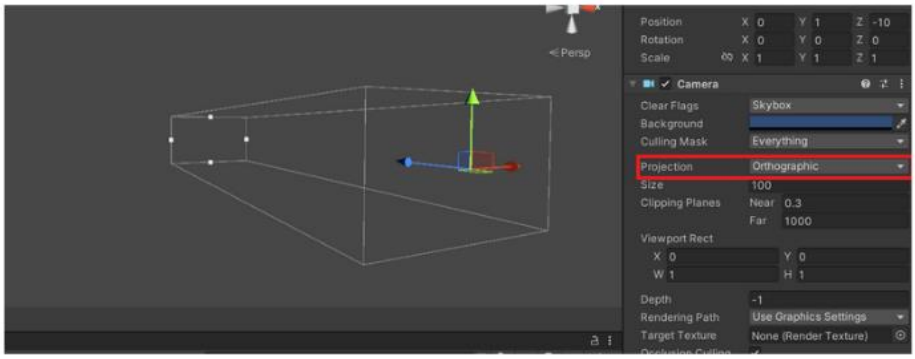
La Main Camera en Unity 3D es un componente crucial en la creación de entornos virtuales. Sirve como un elemento central que facilita la visualización de la escena desde la perspectiva del usuario.

A continuación, se presentarán algunas de las características más importantes:

- Perspectiva y proyección: La Main Camera puede ser configurada para mostrar la escena en perspectiva o en proyección ortográfica, adaptándose así a diferentes requisitos de representación visual en entornos industriales. Véase en la Figura 6.




a)




b)

**Figura 6 Proyección de la Main Camera. a) Perspective y b) Orthograpfic**

- Control de visualización: Permite ajustar parámetros como el campo de visión (FOV), determinando así el ángulo de visión de la cámara y qué parte de la escena es visible para el usuario.
- Renderizado: Permite configurar opciones de renderizado como el nivel de calidad, resolución y efectos visuales, asegurando un rendimiento óptimo en la representación de detalles en procesos industriales simulados.

### 3.5.2. Creación de un Objeto

Para iniciar la creación de un nuevo objeto, se debe situar sobre la ventana de Jerarquía, hacer clic derecho y se abrirá una ventana con las opciones disponibles para añadir. Dentro de la sección 3D Object, se visualizan los elementos básicos como el cubo, la esfera, la cápsula, el cilindro, un plano o un ‘quad’ (Figura 7). Este último se asemeja al plano, pero sus bordes solo son una unidad de largos y la superficie es orientada en el plano XY del espacio de coordenadas local.

En Unity, también es posible importar objetos previamente modelados desde fuentes externas, ya sea creados por el usuario o encontrados en internet. Esto se abordará más en profundidad en el apartado de Objetos Dinámicos.

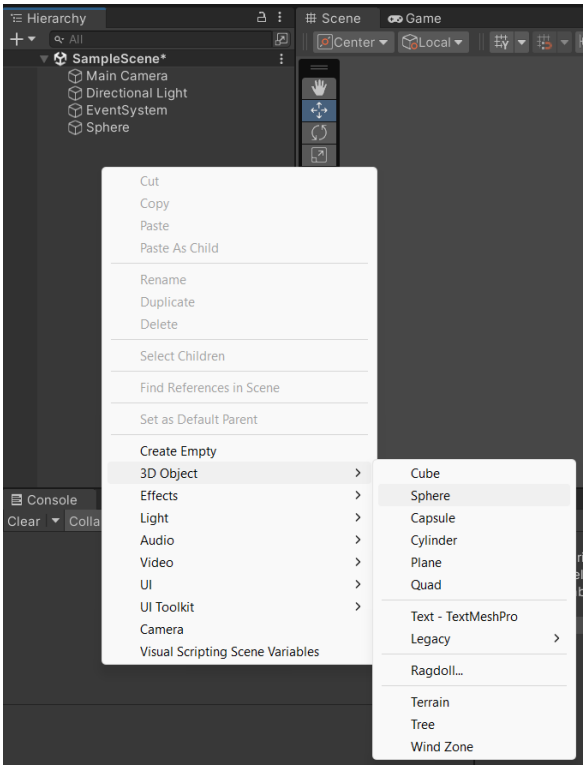
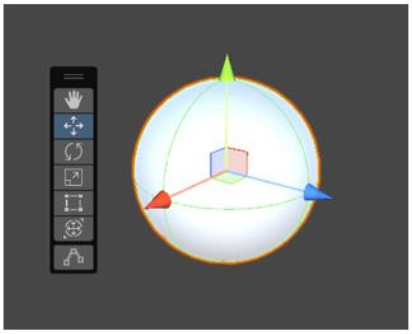
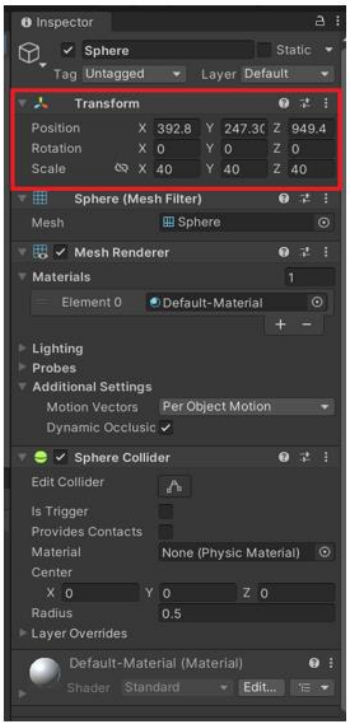



**Figura 7 Creación de un Objeto**

Una vez seleccionado el tipo de objeto deseado, en la ventana del Inspector se encuentran las opciones para ajustar su Posición, Rotación y Escala, como se muestra en la pare B de la Figura 8. Mientras que en la parte A de la misma figura, nos muestra la posibilidad de modificar estas propiedades visualmente mediante las flechas que aparecen sobre el objeto.



A: Esfera con flechas de modificación




B: Inspector de la esfera

**Figura 8 Parametrización del concepto de esfera. A y B**

Para cambiar la apariencia del objeto, es necesario crear y aplicar un nuevo material. Para ello, se debe situar sobre la ventana de proyecto, hacer clic derecho y seleccionar la opción Create, luego Material, tal y como muestra la Figura 9.

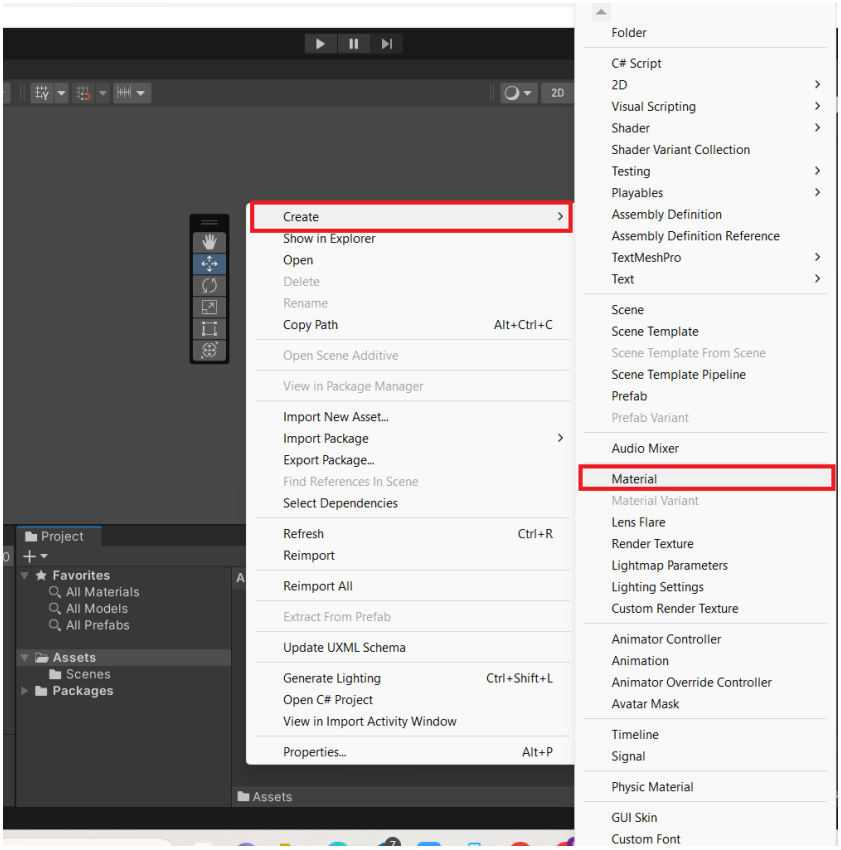
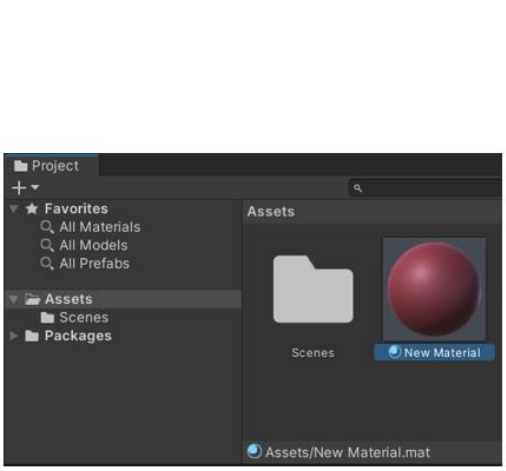



**Figura 9 Creación del Material del Objeto**

Una vez creado el material, se pueden ajustar sus características, como color, brillo, rugosidad, añadirle efectos de renderizado, etc., como se ilustra en la parte B de la Figura 10. Una vez definido el material, basta con arrastrarlo desde la ventana de proyecto, como se muestra en la parte A de la Figura 10, hacia el objeto deseado para aplicarlo.




A: Ubicación del Material creado


B: Características del material en el Inspector

**Figura 10 Ubicación y características del material creado. A y B**

### 3.5.3. Creación del Comportamiento de un objeto

Una vez creado el objeto, es necesario agregarle un comportamiento específico que dependerá de su función en la aplicación o juego como, por ejemplo, mover una caja horizontalmente o desplazarla hacia la izquierda al presionar un botón. Para ello, es preciso crear un Script y programarlo según la función que se requiera.

Para iniciar la creación del script, se debe ubicar en la ventana de proyecto, hacer clic derecho, seleccionar la opción “Create” y luego C# Script, como se ilustra en la Figura 11.

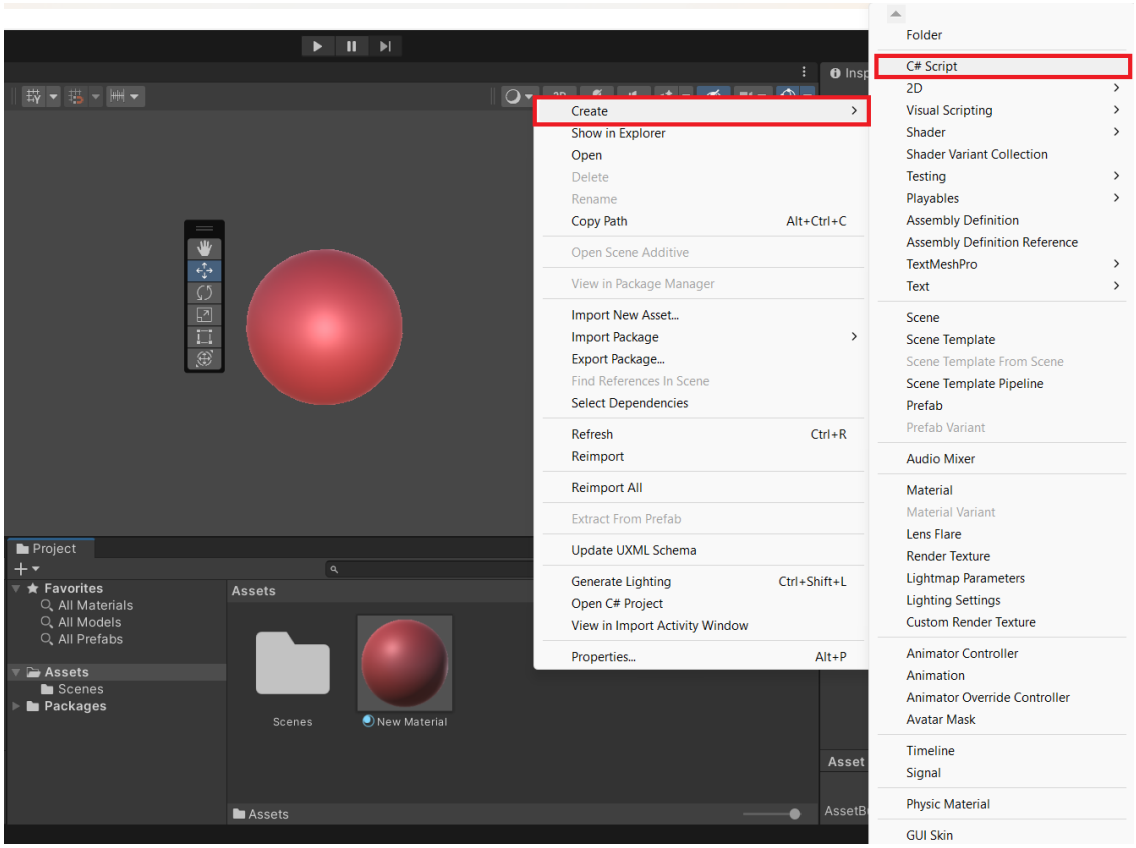



**Figura 11 Creación de un Script**

Se generará un script base vacío (Figura 12) que incluye algunos términos que necesitan ser explicados previamente.

En la parte superior del código se encuentran los “using statements”, que son directivas de compilación que permiten utilizar tipos definidos en un espacio de nombres sin especificar el espacio de nombres completo en ese tipo. Uno de los más importante es el ‘UnityEngine’, que proporciona acceso a la API de Unity, incluyendo clases y funciones que permiten interactuar con el motor de Unity y los elementos fundamentales del juego. Entre estos elementos, el ‘GameObject’ es clave, ya que constituye la unidad básica de todos los objetos en escena. Los GameObjects son fundamentales para la lógica y la interactividad en un juego desarrollado con Unity.

En la Figura 12, se muestra un script denominado ‘Ejemplo’, el cual va a partir de la clase ‘MonoBehaviour’. Esta última constituye una clase base proporcionada por Unity, diseñada para facilitar la elaboración de scripts que se vinculan a GameObjects en la escena. Estos scripts contienen lógica y comportamiento que define cómo interactúan los GameObjects con el entorno del juego y entre sí. Al

heredar de MonoBehaviour, los scripts obtienen acceso a una serie de métodos predefinidos que se ejecutan en diferentes momentos del ciclo de vida del GameObject.

En la misma Figura 12 se presenta una estructura básica que incluye métodos fundamentales para el funcionamiento de cualquier script en Unity. Por un lado, se encuentra el método 'void Start() {}'. Este método se invoca una vez al inicio, antes del primer frame de actualización. Su propósito es inicializar variables u objetos necesarios para el funcionamiento del script, como configurar referencias a otros objetos en la escena o establecer valores iniciales.

Por otro lado, está el método 'void Update() {}'. Este método se ejecuta una vez por cada frame de la ejecución del sistema. Se utiliza para actualizar el estado de los objetos en función del tiempo transcurrido desde el último frame. Aquí es donde se inserta la lógica del sistema que necesita ser ejecutada continuamente, como el movimiento de objetos, la detección de colisiones, entre otras operaciones relevantes.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
        Script de Unity | 1 referencia
5   public class Ejemplo : MonoBehaviour
6   {
7       // Start is called before the first frame update
            Mensaje de Unity | 0 referencias
8       void Start()
9       {
10
11      }
12
13      // Update is called once per frame
            Mensaje de Unity | 0 referencias
14      void Update()
15      {
16
17      }
18  }
```

**Figura 12 Script Ejemplo**

Para asignar el Script al objeto deseado en Unity, primero se selecciona el objeto en la ventana de jerarquía para acceder a su inspector. Luego, se arrastra el

código desde la ventana de proyecto hacia el inspector, como se muestra en la Figura 13.

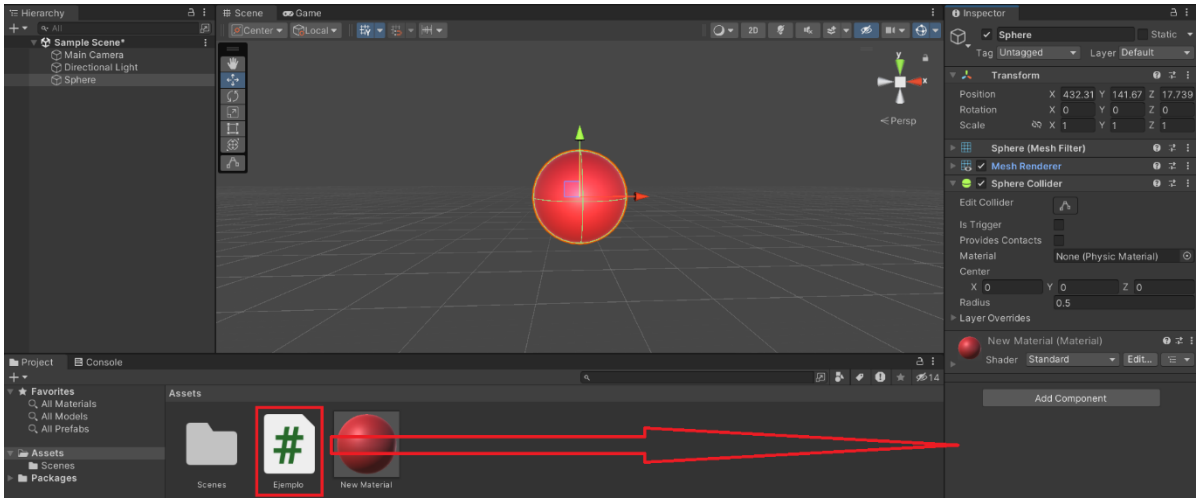



**Figura 13 Añadir Script a Objeto**

La depuración del código es un aspecto crucial en el desarrollo de software. Ya que implica la identificación y corrección de errores. Una de las técnicas más comunes es la inserción estratégica de mensajes de impresión, por ejemplo, mediante Debug.Log. Sin embargo, la mejor forma de depurar es mediante puntos de ruptura, que permiten examinar el comportamiento del programa en tiempo de ejecución para solucionar fallos y mejorar la eficiencia.

Para establecer un punto de ruptura y depurar el código en busca de fallos, se debe utilizar Microsoft Visual Studio. En la línea de código deseada, se puede asignar un punto de ruptura haciendo clic en la parte izquierda, donde aparece la opción de un punto rojo. Luego, para asociar el código a Unity y examinarlo en tiempo de ejecución, se selecciona la opción Asociar a Unity en Visual Studio y se permite la depuración en Unity. (Figura 14)

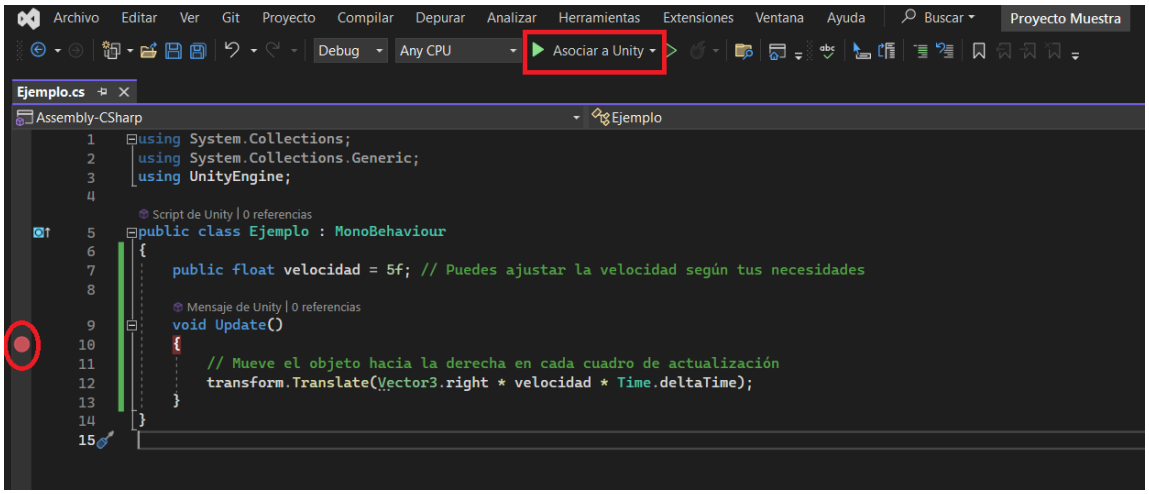



**Figura 14 Punto de Ruptura y Asociación al Unity**

En la Figura 14, se muestra el script de la Figura 12 con unas modificaciones añadidas. Dentro de este script, se ha declarado una variable para almacenar la velocidad, a la cual se le asigna el valor '5f', un número decimal de punto flotante. Luego, en el método Update, se encuentra una línea de código que mueve el objeto en la dirección derecha de su sistema de coordenadas locales, utilizando la velocidad determinada, y garantizando un movimiento suave y consistente mediante la utilización de 'Time.deltaTime'.

## 3.6. Cómo modelar un objeto en Unity

En Unity, uno de los aspectos fundamentales es la creación y manipulación de objetos dentro del entorno virtual. Para lograr una representación realista y funcional de estos objetos, es importante comprender y aplicar adecuadamente los componentes físicos proporcionados por Unity.

*Rigidbody*, *Collider* y *Trigger* surgen como componentes clave para dotar a los objetos de comportamientos físicos y de interacción con su entorno. Estos componentes, que se explicarán a continuación, permiten simular aspectos como la gravedad, la detección de colisiones y la activación de eventos.

- **Rigidbody**

El Rigidbody es un componente que simula el comportamiento físico de un objeto en Unity. Al añadir un Rigidbody a un objeto, éste se vuelve dinámico y se ve

afectado por las leyes físicas del entorno, como la gravedad, la fricción y las fuerzas externas.

Para asignar un Rigidbody a un objeto, se debe seleccionar dicho objeto y acceder a la ventana de Inspector. Posteriormente, se procede a hacer clic en “Add Component” y buscar la opción de “Rigidbody”.

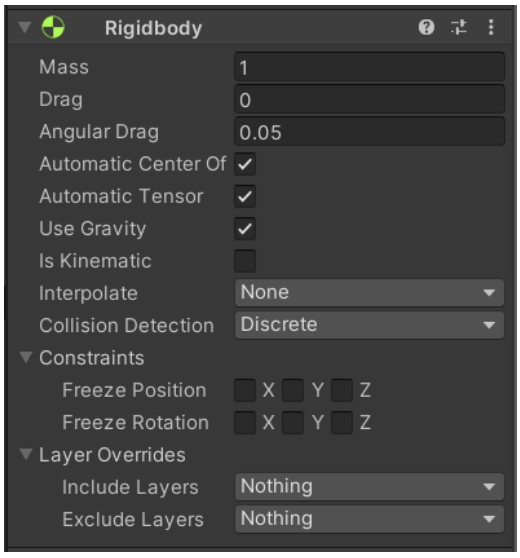



**Figura 15 Propiedades Rigidbody**

En la Figura 15, se presenta la ventana de propiedades que se despliega al agregar el Rigidbody a un objeto. En dicha ventana, se encuentran diversas opciones que permiten configurar el comportamiento físico del objeto de manera detallada. En primer lugar, se dispone de la opción para establecer la masa del objeto, expresada en kilogramos. A continuación, se encuentran los parámetros relacionados con la resistencia al aire, tanto en términos de movimiento (Drag) como de influencia en el torque del objeto (Angular Drag). Además, se proporcionan casillas para utilizar un centro de masa y un tensor del objeto ya calculado automáticamente.

Asimismo, se presenta una casilla destinada a controlar la influencia de la gravedad sobre el objeto, así como otra que, al activarse, restringe la manipulación del objeto exclusivamente a través de código, excluyendo la interacción con el motor de física de Unity (Is Kinematic). Se incluyen opciones adicionales para la interpolación y el tipo de detección de colisiones deseadas.

La sección denominada 'Constraints' ofrece la posibilidad de establecer restricciones específicas de movimiento para el Rigidbody, tanto en términos de rotación como de traslación. Por último, se brinda la opción para agregar o eliminar capas asociadas al Rigidbody.

- **Collider**

El Collider es un componente que delimita el área física de un objeto en Unity. Se puede conceptualizar como una malla invisible que envuelte al objeto, permitiéndole interactuar físicamente con el entorno virtual. Su función primordial radica en la detección de colisiones con otros objetos presentes en el entorno.

En Unity, se dispone de diversos tipos de colliders que pueden ser implementados en los objetos según las necesidades del proyecto. Entre los más usuales se encuentran los colliders con formas primitivas como el “Box Collider”, que presenta una forma de caja, el “Sphere Collider”, que adopta una forma esférica y el “Capsule Collider”, que se asemeja a una cápsula.

Para asignar un Box Collider a un objeto en Unity, se sigue un proceso similar al asignar un Rigidbody, pero se buscará el collider deseado.

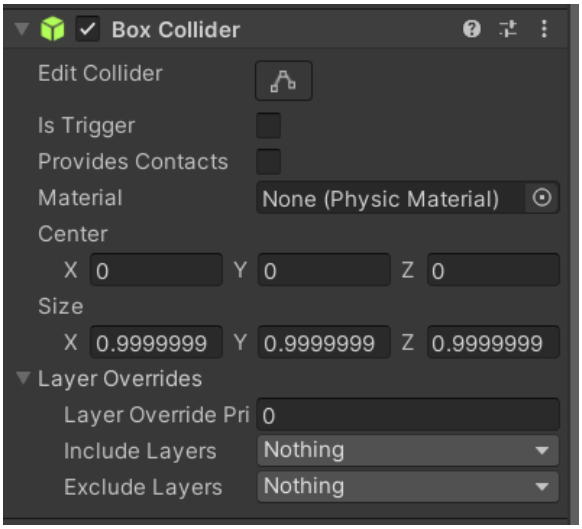



**Figura 16 Propiedades Box Collider**

La Figura 16 muestra la ventana de propiedades asociada a un Box Collider. En esta ventana se encuentra la casilla de Trigger, cuya función será detallada

posteriormente, así como una casilla para controlar los eventos generados por las colisiones (Provides Contacts).

Además, se ofrece la opción "Material", que permite asignar un material previamente configurado al Collider. Por ejemplo, se puede seleccionar el material "Acero", otorgándole al Collider las propiedades asociadas a dicho material. También se incluyen opciones para ajustar el centro y el tamaño de la malla del Collider, permitiendo una personalización precisa de su geometría.

Por último, se proporciona la opción para agregar o eliminar capas asociadas al Box Collider.

- **Trigger**

El Trigger, por otro lado, se presenta como un tipo especial de Collider que no provoca una respuesta física directa al entrar en contacto con otro objeto. En su lugar, su función radica en activar eventos al ingresar o salir otro objeto de su área. Este mecanismo facilita la creación de áreas de activación, zonas de detección y otros efectos basados en desencadenadores.

Un Collider configurado como Trigger carece de las propiedades de un objeto sólido y siempre permitirá que otros objetos lo atraviesen. Cuando un Collider ingresa al espacio físico de un Trigger, este invoca una función conocida como “OnTriggerEnter()”, y cuando abandona el espacio físico de un Trigger, se invoca la función “OnTriggerExit()” en el objeto que está configurado como Trigger.

Para activar esta funcionalidad, simplemente se debe seleccionar la opción “Is Trigger” dentro de las propiedades del Collider correspondiente.

## 3.7. Canvas

El Canvas en Unity desempeña un papel fundamental en la construcción y organización de la Interfaz de Usuario (UI, por sus siglas en inglés). Actúa como el lienzo principal donde se colocan y manipulan todos los elementos de la interfaz, como textos, botones, toggles e imágenes. Sirve como el contenedor visual que determina la disposición de estos elementos en la pantalla de juego.

En dicho proyecto, el Canvas tiene una importancia significativa, ya que albergará varios botones que activarán distintos modos de uso, así como permitirá la modificación de características clave. Para su creación, se accede a la ventana de Jerarquía y se hace clic derecho, lo que abre un menú contextual. Dentro de este menú, se selecciona la opción UI para abrir otra ventana y, finamente, se elige Canvas, tal y como se muestra en la Figura 17.

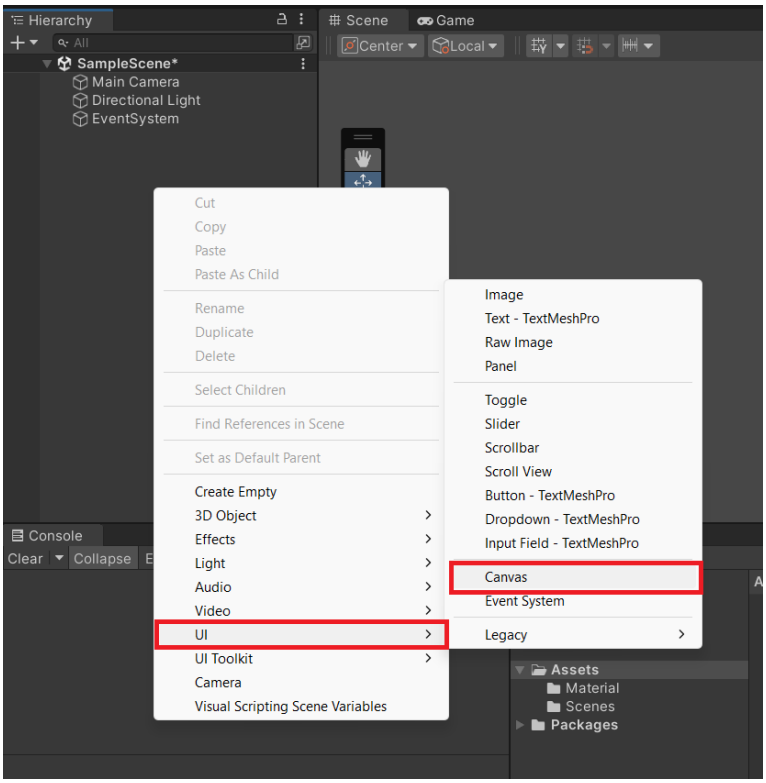



**Figura 17 Creación del Canvas**

Un ajuste que se debe realizar al Canvas es cambiar su modo de escala para que sea uniforme con respecto a la pantalla. Esta modificación se debe hacer al principio, ya que el Canvas viene con un modo de escala predeterminado diferente. Para realizar este cambio, tal y como se ilustra en la Figura 18, se selecciona el objecto Canvas y se accede al inspector. Luego, se busca la sección de Canvas Scaler y se modifica la opción de UI Scale Mode seleccionando “Scale With Screen Size”.

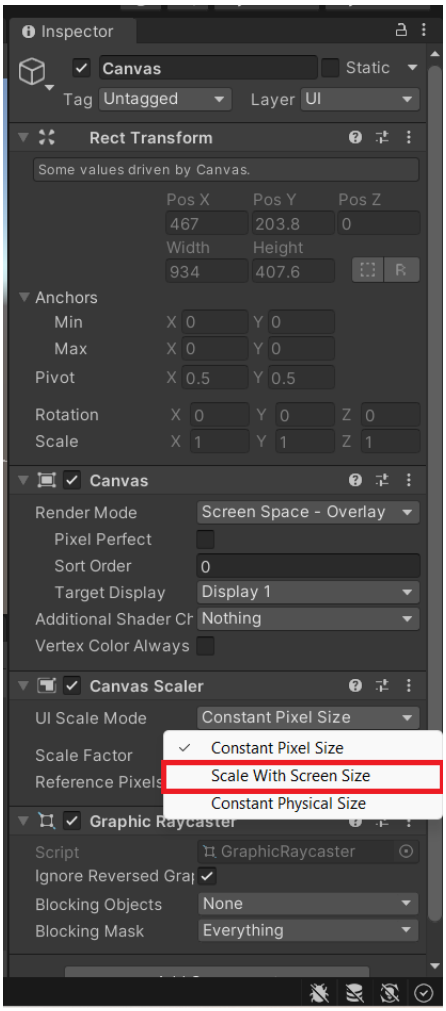



**Figura 18 Cambio modo de escala del Canvas**

Para añadir diferentes elementos al Canvas, simplemente se repiten los pasos de la Figura 17, pero en la ventana final se selecciona el tipo de elemento deseado. Se pueden añadir botones, textos modificables, controles deslizantes y una variedad de otros elementos de interés.

### 3.7.1. Elementos Propios del UI

Aquí se presentarán los elementos más destacados del UI que se encuentran dentro del Canvas. Todos los elementos que se explicarán a continuación están ubicados en la misma ventana a la que se accede para crear el Canvas.

- **Text – TextMeshPro (TMP)**

Este elemento de UI se utiliza principalmente para añadir texto. Al crear este tipo de texto, aparecerá la opción de importar TMP Essentials, la cual debe seleccionarse (Figura 19). TextMeshPro es un componente de Unity que ofrece

funcionalidades avanzadas de texto en comparación con el componente de texto estándar de Unity, ya que brinda una mayor flexibilidad y control sobre la apariencia y el comportamiento del texto. Para utilizar este elemento, se necesita agregar un “using statements”, que es ‘using TMPro’. Y para declararlo, se utiliza una variable pública de tipo TMP_Text.

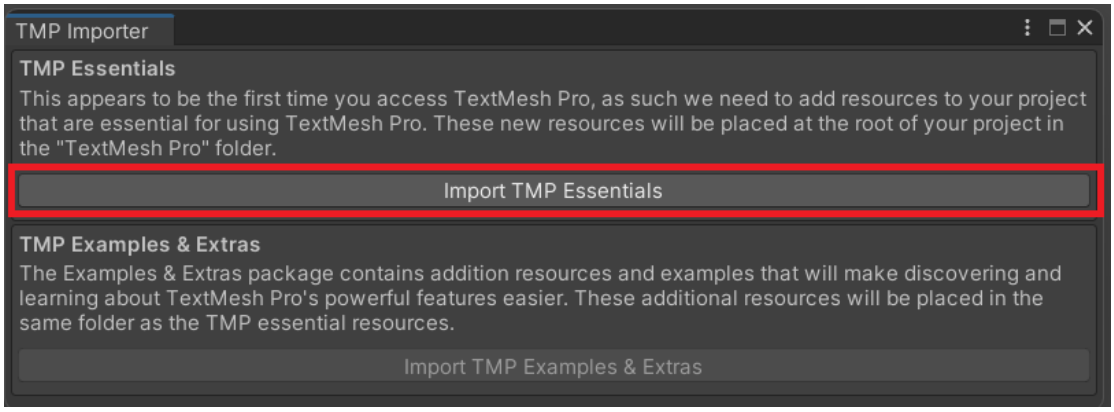



**Figura 19 TextMesh Pro**

- **Panel**

El panel en Unity es un componente importante dentro de la interfaz de usuario (UI) que se utiliza para organizar y agrupar otros elementos de UI de manera coherente y visualmente atractiva, siendo así su función principal, proporcionar una estructura visual para organizar los elementos de la pantalla, permitiendo ajustar el diseño, aplicar estilos visuales consistentes y agregar funcionalidades interactivas. Esto facilita la creación de interfaces de usuario claras, mejorando la experiencia del usuario al interactuar con la aplicación.

- **Button**

Este elemento permite a los usuarios interactuar con el juego o la aplicación al hacer clic en él. Su funcionalidad básica es activar una acción o evento cuando se presiona. Para otorgarle funcionalidad al botón, como se ilustra en la Figura 20, disponemos de un recuadro denominado 'On Click()', donde al seleccionar el botón de adición ('+') se añade una funcionalidad. Aquí se debe especificar un objeto que contenga un script asociado con la función deseada para el botón, luego se selecciona dicha función que se activará al hacer clic en el botón. Se detallará más en el video del apartado Caso de uso: Manipulación de un vástago de doble efecto.

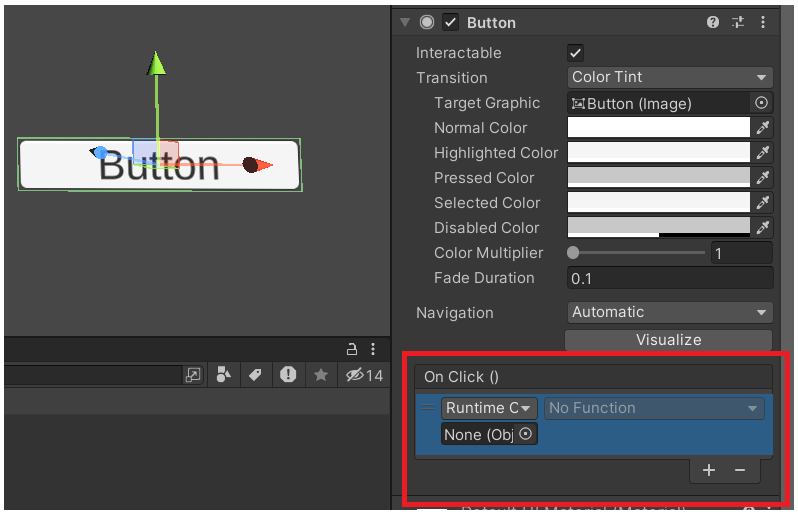



**Figura 20 On Click() de Button**

En el script se empleará una variable pública de tipo Button, la cual puede ser accedida y modificada desde el editor de Unity. Dentro del botón, se encuentra el objeto 'Text (TMP)' que puede ser modificado para cambiar el texto mostrado.

- **Toggle**

Es un elemento que funciona como un interruptor con dos estados: activado y desactivado. Su objetivo principal es permitir al usuario alternar entre dos opciones, como encender o apagar una función, activar o desactivar una configuración, entre otros. Los métodos más comunes utilizados con un Toggle:

- isOn: Este método devuelve un valor booleano que indica si el Toggle está actualmente activado (true) o desactivado (false). Puede ser útil para verificar el estado actual del Toggle en un script y realizar acciones basadas en ese estado.
- OnValueChanged: Este método se invoca automáticamente cuando el estado del Toggle cambia, es decir, cuando el usuario hace clic en él para activarlo o desactivarlo. Para otorgarle la funcionalidad al método es necesario seleccionar donde se encuentra como en el ejemplo del botón, utilizando el recuadro marcado en la Figura 21.

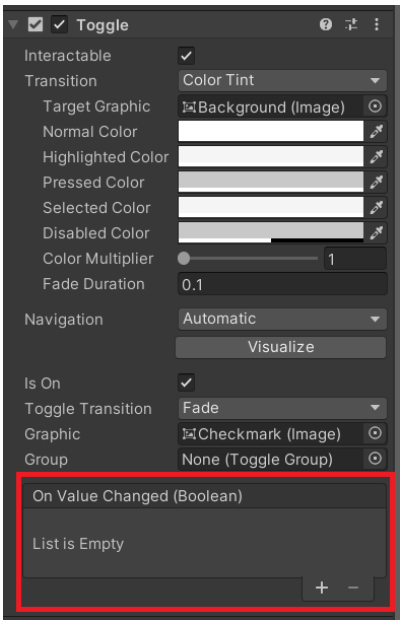



**Figura 21 Inspector del Toggle**

Para declararlo en un script, se utiliza una variable pública de tipo Toggle. Dentro del Toggle, se encuentran características modificables como el 'Background' y el 'Label'.

- **DropDown**

Proporciona a los usuarios una lista desplegable de opciones entre las cuales pueden elegir. Su función básica es permitir a los usuarios seleccionar una opción de la lista desplegable, lo que activa un evento cuando se realiza una selección. Se declara con una variable pública de tipo Dropdown. Dentro de este elemento, se pueden encontrar características como el 'Label', 'Arrow' y otras más avanzadas.

- **Input Field (TMP)**

Es un elemento que permite a los usuarios ingresar y editar texto. Es una versión mejorada del Input Field estándar de Unity que utiliza TMP para proporcionar un texto más legible y con mayor calidad visual. Para utilizar este elemento, se necesita agregar un "using statements", que es 'using TMPro'. Y para declararlo, se utiliza una variable pública de tipo TMP_InputField.

# 4. Modelo Digital

Al emplear Unity para la creación de Modelos Digitales, se debe aprovechar su capacidad gráfica y las diversas funciones que ofrece esta herramienta. Entre estas funciones se encuentran la capacidad de crear elementos desde cero, importar elementos preexistentes y una variedad de otras características.

En este apartado se detallará un flujograma general de cómo utilizar Unity para poder definir un modelo digital. Seguidamente, se presentará un ejemplo de objeto discreto, otro de objeto dinámico y se explicará un caso de uso donde se manipulará un vástago de doble efecto. Para concluir, se ofrecerá un QR donde se podrá visualizar un video del Modelo digital en Unity, donde se mostrará el montaje desde cero de un cilindro que se accionará mediante botones.

## 4.1. Metodología

La creación de un modelo digital en Unity implica varias etapas, desde la conceptualización hasta la implementación en la plataforma. Este flujograma proporciona una guía básica sobre cómo utilizar Unity para definir un modelo digital.

1) *Conceptualización del modelo digital:*
    - Identificar el propósito y los requisitos del modelo digital.
    - Definir las características clave y la funcionalidad requerida.
2) *Creación de activos:*
    - Diseñar o adquirir los activos necesarios para el modelo digital, como modelos 3D, texturas, y sonidos si fuese necesario.
    - Importar los activos a Unity y organizarlos en sus carpetas correspondientes.
3) *Configuración de escenas:*
    - Crear una nueva escena en Unity para el modelo digital.
    - Disponer los activos en la escena de acuerdo con el diseño previamente establecido.
4) *Configuración de cámaras y luces:*
    - Ajustar la posición, rotación y configuración de las cámaras para obtener la perspectiva deseada del modelo.

5) *Implementación de elementos de interfaz de usuario (UI):*
    - Diseñar y agregar elementos de UI, utilizando botones, toggles, sliders, etc...
    - Configurar la funcionalidad de estos elementos para interactuar con el modelo digital.
6) *Implementación de interactividad:*
    - Agregar scripts y lógica de juego para permitir la interacción del usuario con el modelo digital.
7) *Pruebas y ajustes:*
    - Probar el modelo digital en el editor de Unity para verificar su funcionamiento y apariencia.
    - Realizar ajustes según sea necesario para corregir posibles errores.
8) *Compilación y distribución:*
    - Compilar el modelo digital para la plataforma de destino, ya sea PC, móvil o web.

## 4.2. Objetos Discretos

La estructura de este apartado seguirá los pasos descritos en la Metodología, excluyendo los pasos 4 y 8, los cuales no son necesarios en esta ocasión debido a que es un proyecto menor. No se requiere un ajuste extenso de cámaras y luces, ni tampoco la etapa de compilación y distribución.

### 4.2.1. Conceptualización

Un objeto discreto se refiere a un componente o elemento que puede cambiar de posición o estado de manera instantánea en respuesta a ciertas condiciones o acciones predefinidas. En este apartado se procederá al análisis de un elemento básico y a la misma vez importante en la automatización, como es el LED.

### 4.2.2. Creación de activos

Para su creación en Unity, existen varias formas disponibles. Por ejemplo, es posible generar un modelo 3D utilizando un software de modelado como Blender, Maya o 3ds Max, y luego importarlo a Unity. Otra opción es acceder a la Asset Store de Unity, la cual ofrece una amplia variedad de activos precreados, si bien, esta

última alternativa implica un coste. La opción más directa consiste en utilizar las primitivas 3D básicas, como esferas o cilindros, para dar forma por ejemplo a un LED y, a continuación, ajustar sus propiedades como color, tamaño y otros atributos para que se asemejen a un LED.

### 4.2.3. Configuración de la escena e Implementación de elementos UI

Para este caso específico, se creará una esfera, como se explicó anteriormente, seguido de la creación de un Toggle, el cual será responsable de activar y desactivar el LED.

Se sugiere, al manipular el Canvas y sus elementos, tener la ventana del juego abierta en paralelo a la de la escena, dado que será necesario posicionar la Cámara Principal enfocando la escena que se desea proyectar, mientras que, en el Canvas, se ajustan las ubicaciones de los elementos mediante la ventana del juego. Véase en la Figura 22.

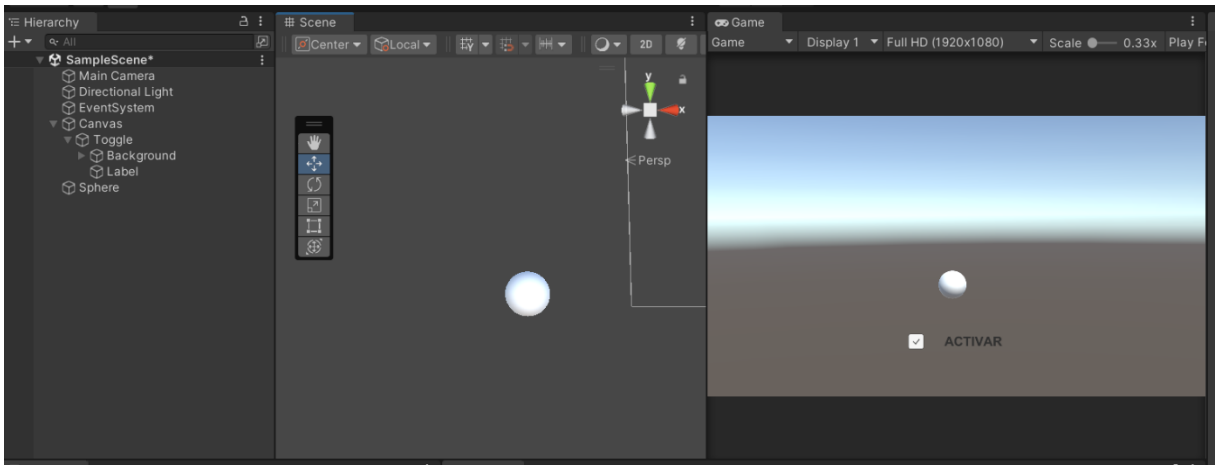



**Figura 22 Posicionamiento LED**

### 4.2.4. Implementación de interactividad

Una vez colocados todos los elementos, se desarrollará un Script llamado ‘LED’ que heredará de la clase ‘MonoBehaviour’, para controlar la funcionalidad del LED, el cual adquirirá un tono rojo claro cuando esté inactivo y rojo fuerte cuando se active. En la Figura 23, se puede observar cómo se va a organizar el Script, lo cual servirá como punto de partida para su posterior explicación.

```
using UnityEngine;

Script de Unity | 0 referencias
public class LED : MonoBehaviour
{
    //DECLARACIÓN DE VARIABLES

    //MÉTODO Start()

    //MÉTODO CambiarColor()
}
```

**Figura 23 Organización Script LED**

La Figura 24 muestra las variables empleadas en el script, donde, las variables públicas de tipo Toggle y GameObject representan el interruptor utilizado para cambiar el color y el objeto esférico en sí, respectivamente. Se espera que estas variables sean asignadas al componente correspondiente. Por otro lado, la variable privada de tipo Renderer se utilizará para acceder al componente "Renderer" de la esfera, el cual es responsable de las propiedades visuales del objeto, como el color.

Las dos últimas líneas de código definen variables de tipo Color, empleadas para representar colores. La primera, "colorRojoFuerte", se inicializa con un tono de rojo intenso, mientras que la segunda variable, "colorRojoClaro", se inicializa con un tono de rojo más suave. Cada color se define mediante valores para los componentes rojo, verde y azul (RGB), junto con un parámetro de opacidad. Estas variables pueden ser utilizadas posteriormente en el programa para asignar colores específicos a elementos visuales.

```
public Toggle toggle;
public GameObject esfera;

private Renderer esferaRenderer;

private Color colorRojoFuerte = new Color(1f, 0f, 0f, 1f); // Rojo fuerte
private Color colorRojoClaro = new Color(1f, 0.5f, 0.5f, 1f); // Rojo claro
```

**Figura 24 Variables Script LED**

En el método Start, ilustrado en la Figura 25, se obtendrá el componente "Renderer" del GameObject de la esfera y se le asignará el color rojo claro inicialmente. Además, se añadirá una línea de código que agrega un listener (un listener es una función que se ejecutará cuando ocurra un evento específico) al evento onValueChanged del toggle. En este caso, se está añadiendo una función anónima que llamará al método CambiarColor() cuando el valor del toggle cambie, es decir, cuando el usuario lo active o desactive.

```
void Start()
{
    // Obtener el componente Renderer de la esfera
    esferaRenderer = esfera.GetComponent<Renderer>();

    esferaRenderer.material.color = colorRojoClaro;

    // Agregar un listener al toggle para detectar cambios
    toggle.onValueChanged.AddListener(delegate { CambiarColor(); });
}
```

**Figura 25 Método 'Start' del Script LED**

Finalmente, se escribirá el método CambiarColor(), el cual evaluará el estado del toggle para determinar qué color debe tener la esfera. Si el toggle está activado (toggle.isOn es verdadero), se asignará el color rojo fuerte al material de la esfera. Si el toggle está desactivado, se asignará el color rojo claro. Véase en la Figura 26.

```
void CambiarColor()
{
    // Cambiar el color de la esfera dependiendo del estado del toggle
    if (toggle.isOn)
    {
        esferaRenderer.material.color = colorRojoFuerte;
    }
    else
    {
        esferaRenderer.material.color = colorRojoClaro;
    }
}
```

**Figura 26 Método 'CambiarColor' del Script LED**

Volviendo a Unity, es necesario asignar el script a un objeto para que funcione. Se puede optar por asignarlo directamente al inspector de la esfera; sin embargo, se prefiere crear un objeto vacío. Esto resultará beneficioso para futuras acciones, ya que facilita la organización y separación de responsabilidades, entre otros aspectos.

Para ello, situados en la ventana de Jerarquía, se hará clic derecho, apareciendo la Figura 7, y se seleccionará "Create Empty", al cual se le dará un nombre apropiado, por ejemplo, LED. Con este objeto seleccionado, se arrastrará el script hacia la ventana de Inspector de LED, donde, al soltar el script, aparecerá la ubicación para asignar los componentes Toggle y Esfera. Se arrastrarán los componentes "Toggle" y "Sphere" desde la ventana de Jerarquía a sus ubicaciones correspondientes, tal y como se muestra en la Figura 27.

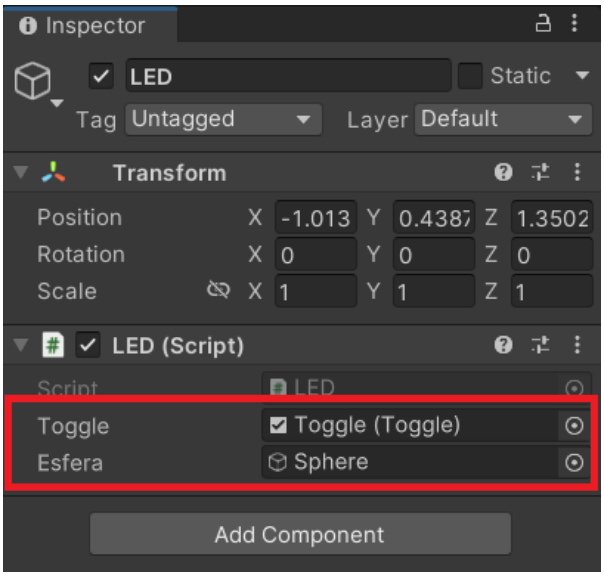



**Figura 27 Componentes Externos del Script LED**

### 4.2.5. Pruebas y ajustes

Una vez completado el último paso, se procederá a ejecutar el juego en Unity y se podrá interactuar con el LED. El funcionamiento de este componente se ilustra en la Figura 28, donde en la parte A se aprecia que al activar el toggle, el LED adquiere un color rojo intenso. Por otro lado, en la parte B, al desactivar el toggle, el LED se mantiene en un color rojo más suave.

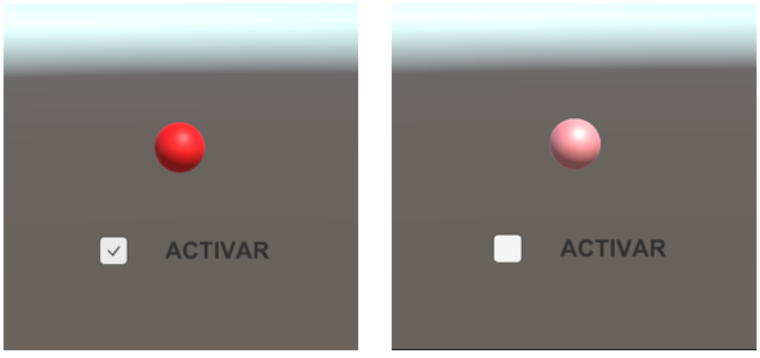



**Figura 28 LEDs en funcionamiento**

Para embellecer el LED, con el propósito de utilizarlo en el Modelo Digital, se ha decidido importar un modelo prefabricado. En primer lugar, es necesario descargar el modelo que se importará. Existen varias plataformas web donde estos

modelos están disponibles, algunas gratuitas y otras de pago. Para esta demostración, se utilizará una fuente gratuita, la cual se puede acceder a través del siguiente enlace: https://www.traceparts.com/es. Se seleccionará el modelo de LED que mejor se adapte a los requerimientos del proyecto. En este caso particular, se ha elegido el modelo de la Figura 29. La importación a Unity será explicada en Objetos Dinámicos.

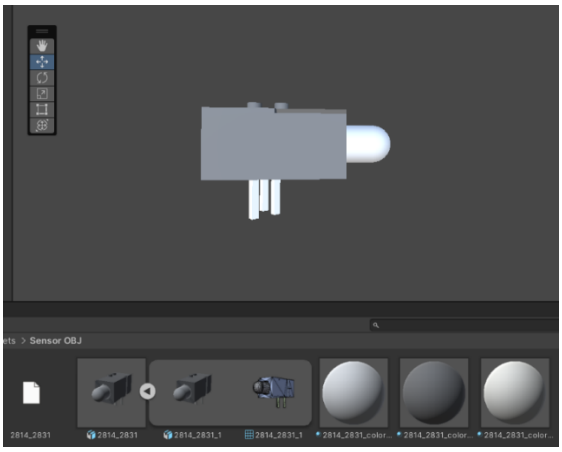



**Figura 29 Sensor OBJ**

## 4.3. Objetos Dinámicos

La estructura de este apartado seguirá los pasos descritos en la Metodología, excluyendo los pasos 4, 5 y 8, los cuales no son necesarios en esta ocasión. No se requiere un ajuste extenso de cámaras y luces, ni implementar ningún elemento UI, ni tampoco la etapa de compilación y distribución.

### 4.3.1. Conceptualización

En esta instancia, se procederá a la importación de un modelo 3D y se explicará cómo dar dinamismo a objetos. En este caso lo haré a través de un cilindro neumático, al cual se le aplicará la funcionalidad de movimiento al vástago sin depender de factores externos.

### 4.3.2. Creación de activos

En proyectos más simples, resulta altamente recomendable el uso de las primitivas 3D básicas para implementar cualquier elemento deseado. Sin embargo, en proyectos de mayor complejidad, especialmente en el ámbito de la

automatización, es frecuente optar por la utilización de modelos 3D preexistentes, los cuales se importan directamente a Unity para su posterior manipulación.

En primer lugar, es necesario descargar el modelo que se importará por lo que se entrará en https://www.traceparts.com/es y se elegirá el modelo deseado.

La importación del modelo a Unity es un proceso sencillo. Una vez completada la descarga, se encontrará una carpeta comprimida que contiene el contenido del modelo. Esta carpeta debe ser descomprimida y luego arrastrada a la ventana de Proyecto de Unity. Esto generará una carpeta que contendrá el modelo importado. Al abrir esta carpeta, se visualizará el modelo junto con un desglose de todos sus elementos constituyentes, incluyendo materiales y mallas.

#### 4.3.3. Configuración de la escena

Para utilizarlo simplemente se arrastra el modelo a la ventana de Jerarquía o directamente a la Escena, como se ilustra en la Figura 30.

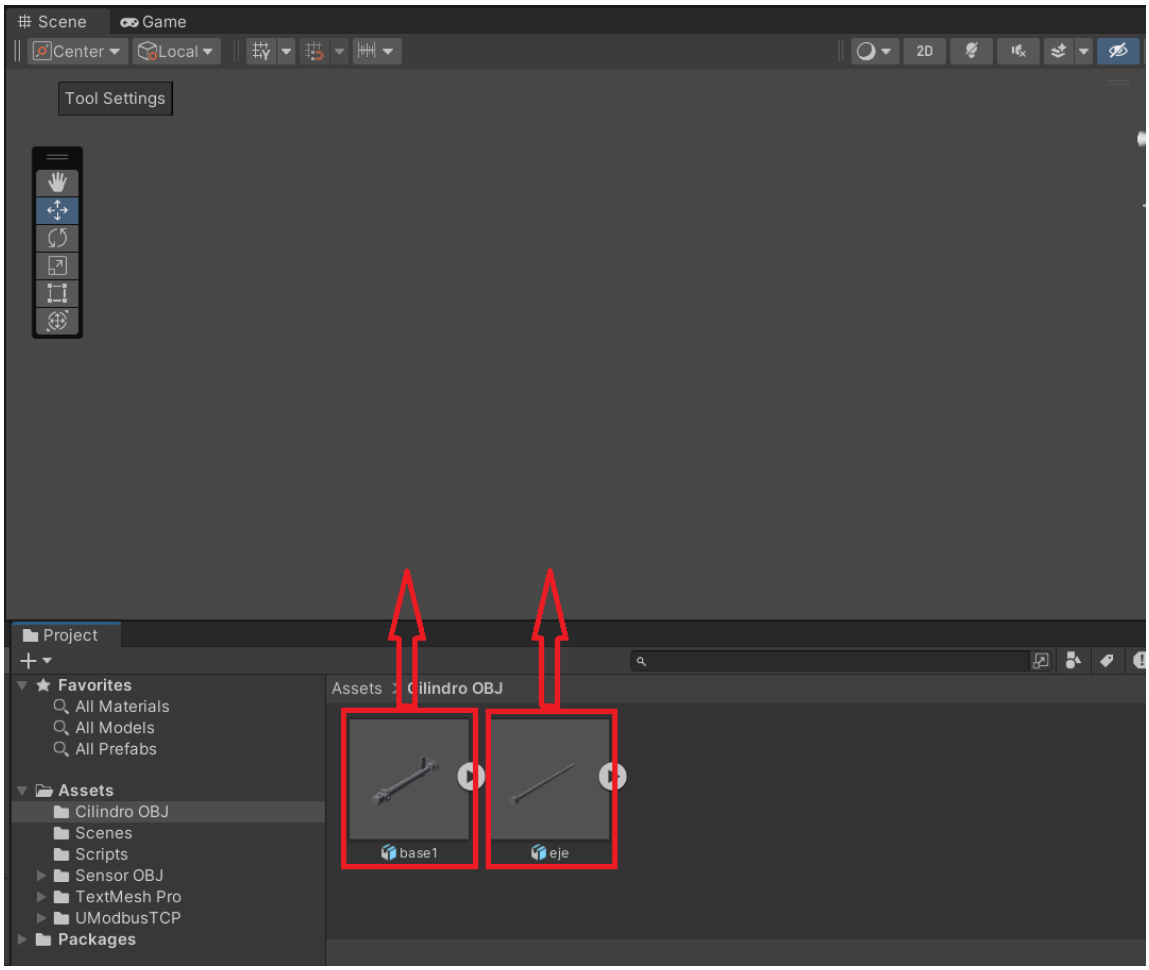



**Figura 30 Añadir el Modelado 3D**

Una vez que el cilindro esté presente en la escena, será necesario ajustar sus proporciones para que sea visible correctamente en la cámara. Además, en la ventana de Jerarquía, se recomienda modificar los nombres de los elementos para una mejor claridad.

### 4.3.4. Implementación de interactividad

Para aplicar la funcionalidad de movimiento, se llevará a cabo la creación de un script llamado ‘MovimientoCilindro’, el cual heredará de ‘MonoBehaviour’. Este script será responsable de programar el movimiento y su organización se realizará de acuerdo con la estructura mostrada en la Figura 31. Además, se creará un objeto vacío en la ventana de Jerarquía para asignarle el script del movimiento, siguiendo el ejemplo del apartado Objetos .

```
public class MovimientoCilindro : MonoBehaviour
{
    // DECLARACIÓN DE VARIABLES

    // MÉTODO Start()

    // MÉTODO Update()
}
```

**Figura 31 Organización Script MovimientoCilindro**

Para la declaración de variables (Figura 32) se empleará una variable pública para la velocidad de movimiento del vástago, la cual podrá ser modificada desde el Inspector. También se necesitarán variables para almacenar la posición inicial y la posición actual del vástago.

Para evitar que el vástago salga del cuerpo del cilindro, se establecerá una distancia máxima de recorrido, la cual estará determinada por las características del modelo 3D. Finalmente, se utilizará una variable pública de tipo GameObject para representar el vástago al que se desea otorgar movimiento y al que se deberá asignar el componente "Vástago" del cilindro.

```
public float speed_actual = 5f;

public Vector3 new_pos;
private Vector3 startingPosition;
private float dis_max = 40f;

public GameObject vastago;
```

**Figura 32 Variables del Script MovimientoCilindro**

En la Figura 33 se mostrará tanto el método Update como el método Start, donde este último solo inicializará la posición inicial del vástago al iniciar el script.

En el método Update, se calculará la nueva posición del objeto "vástago" sumando un desplazamiento horizontal basado en la velocidad actual y el tiempo transcurrido desde el último frame (Frame o fotograma se refiere a la imagen concreta dentro de una sucesión de imágenes en movimiento).

Se utilizará la función Mathf.Clamp() para limitar la posición horizontal del "vástago". Finalmente, se asignará la nueva posición al objeto "vástago", lo que resultará en el movimiento físico del vástago en la escena. Si se alcanza el límite de

movimiento horizontal establecido, el movimiento se detendrá y la velocidad se actualizará a cero.

```
void Start()
{
    startingPosition = vastago.transform.position;
}

Mensaje de Unity | 0 referencias
void Update()
{
    new_pos = vastago.transform.position + (Vector3.right * speed_actual * Time.deltaTime);

    // Mathf.Clamp es una función que se utiliza para limitar un valor dentro de un rango específico.
    // Mathf.Clamp(valor, valorMínimo, valorMáximo)
    float clampedX = Mathf.Clamp(new_pos.x, 0 , startingPosition.x + dis_max);
    new_pos.x = clampedX;

    vastago.transform.position = new_pos;

    if (clampedX != new_pos.x)
    {
        speed_actual = 0;
    }

}
```

**Figura 33 Método 'Start' y 'Update' del Script MovimientoCilindro**

De vuelta en Unity, será necesario arrastrar el componente "Vástago" del cilindro desde la ventana de Jerarquía hasta su ubicación en el Inspector del objeto vacío. (Figura 34)

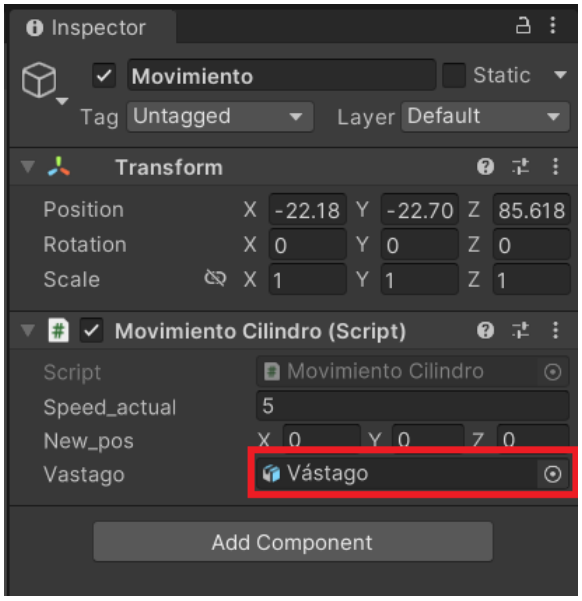



**Figura 34 Componente Externo del Script MovimientoCilindro**

### 4.3.5. Pruebas y ajustes

Finalmente, al iniciar la reproducción en Unity, se podrá observar cómo el vástago experimenta un movimiento horizontal automático y se detiene únicamente cuando alcanza el límite mecánico del cilindro como se muestra en la Figura 35.

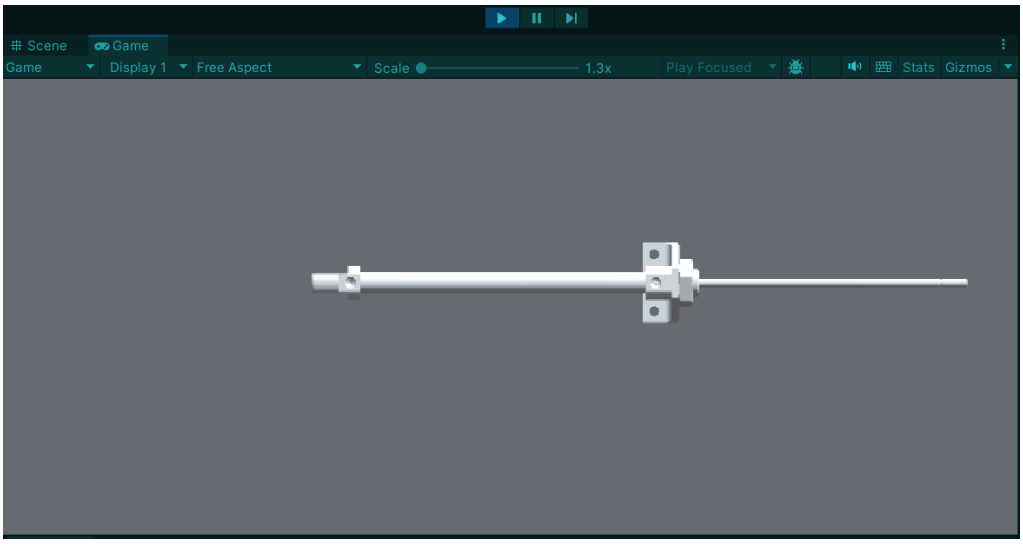



**Figura 35 Cilindro totalmente extendido**

## 4.4. Caso de uso: Manipulación de un vástago de doble efecto

La estructura de este apartado seguirá los pasos descritos en la Metodología, excluyendo los pasos 4 y 8, los cuales no son necesarios en esta ocasión. No se requiere un ajuste extenso de cámaras y luces, ni tampoco la etapa de compilación y distribución.

### 4.4.1. Conceptualización

Para este caso de uso se implementarán los sensores al cilindro, donde se emplearán conceptos previamente explicados. El procedimiento para su funcionamiento constará de dos cubos, los cuales representarán los dos sensores del cilindro, junto con el vástago del cilindro.

### 4.4.2. Creación de activos y configuración de la escena

Para la parte visual de sensor se aplicará el modelo de la Figura 29. Para la parte magnética, se ha hecho uso del elemento cubo, al que se le aplicará un Rigibody y un Box Collider configurado como Trigger, como ya se explicó en el apartado de Cómo modelar un objeto en Unity. La Figura 36 presenta las vistas con la caracterización de los elementos que forman un sensor magnético.

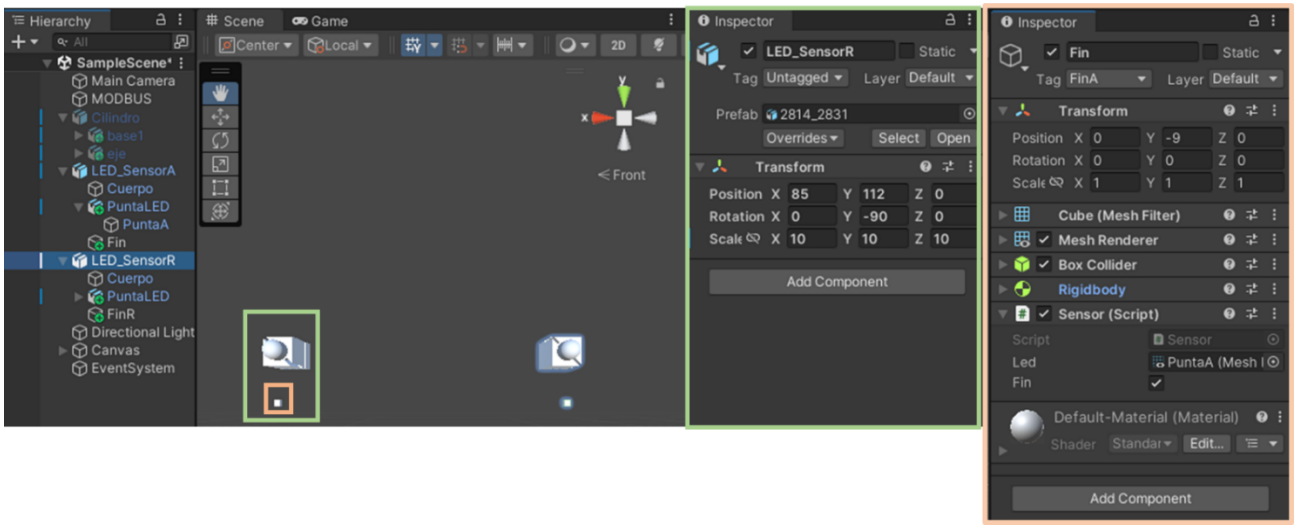



**Figura 36 Creación de sensores magnéticos**

Una vez configurados los dos Trigger, será necesario ajustar su tamaño y posicionarlos en los lugares donde se desee que el vástago se detenga, asegurándose de que se encuentren en el camino con él. Un ejemplo de posicionamiento, sin mostrar la parte visual del sensor, se muestra en la Figura 37.

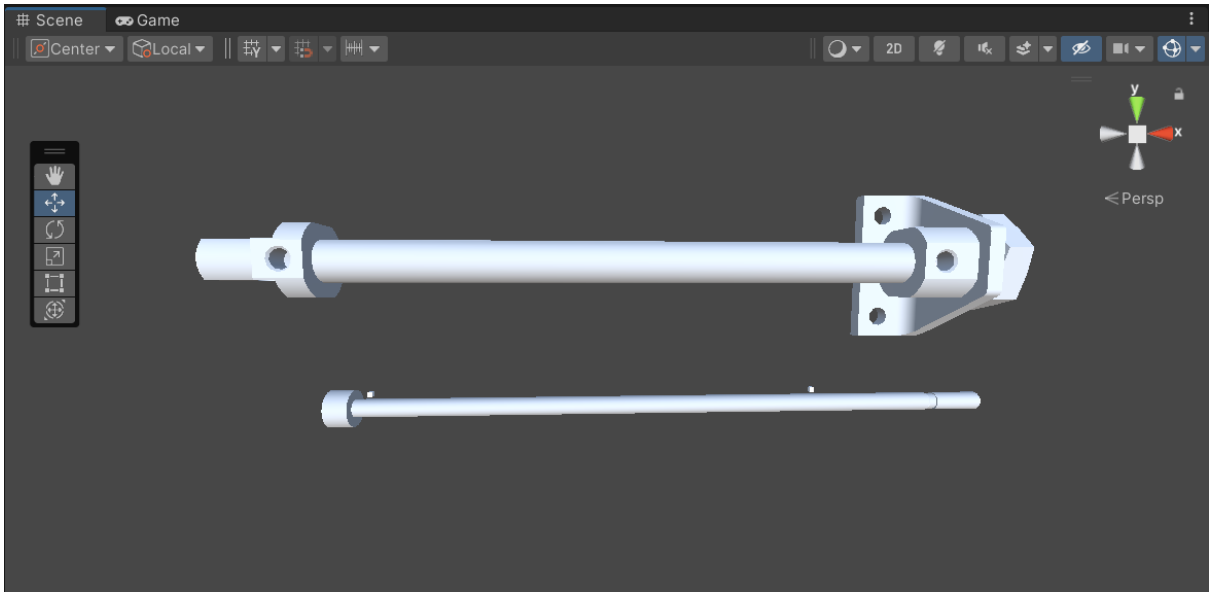



**Figura 37 Posicionamiento de los Cubos Sensores**

Para el componente vástago del cilindro, se aplicará un Rigidbody. En la pestaña de Constraints, se seleccionarán todos los ejes excepto el eje Z de la posición, de manera que el vástago tenga libertad de movimiento solo en ese eje. También se le aplicará un Box Collider, abarcando todo el vástago como área del collider. Posteriormente, será necesario ajustar esta área para que solo englobe el pistón. Se puede apreciar cómo la Figura 38 ilustra la aparición de la malla asociada

al Box Collider al aplicarlo, y en la Figura 39, cómo esta malla ha sido ajustada para cubrir el área deseada.

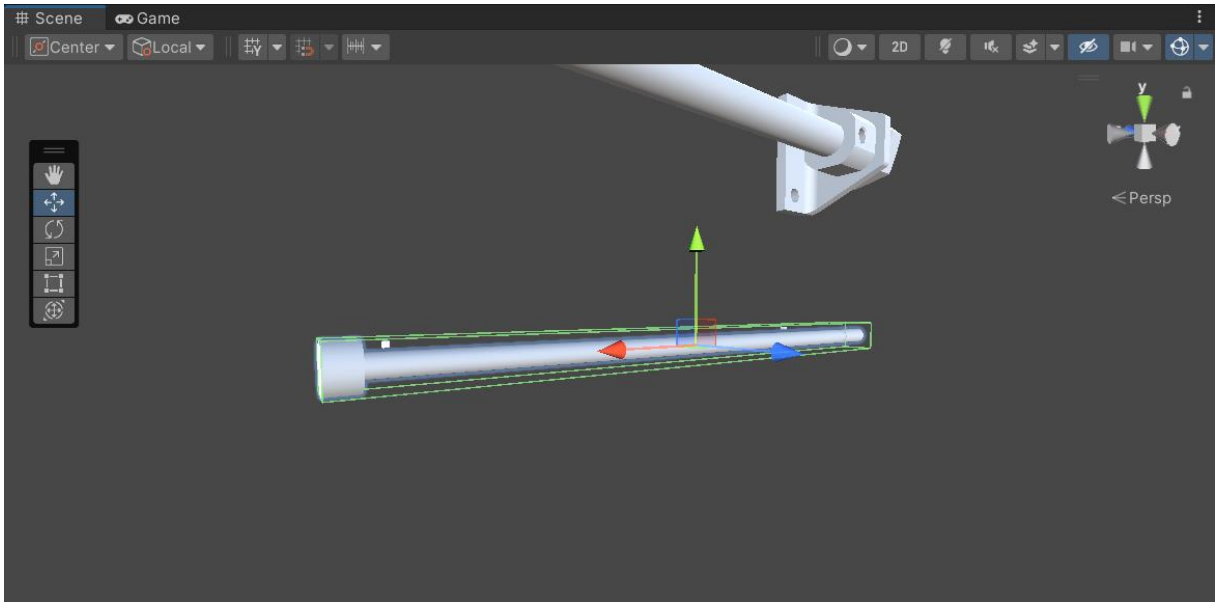



**Figura 38 Box Collider del Vástago sin ajustar**

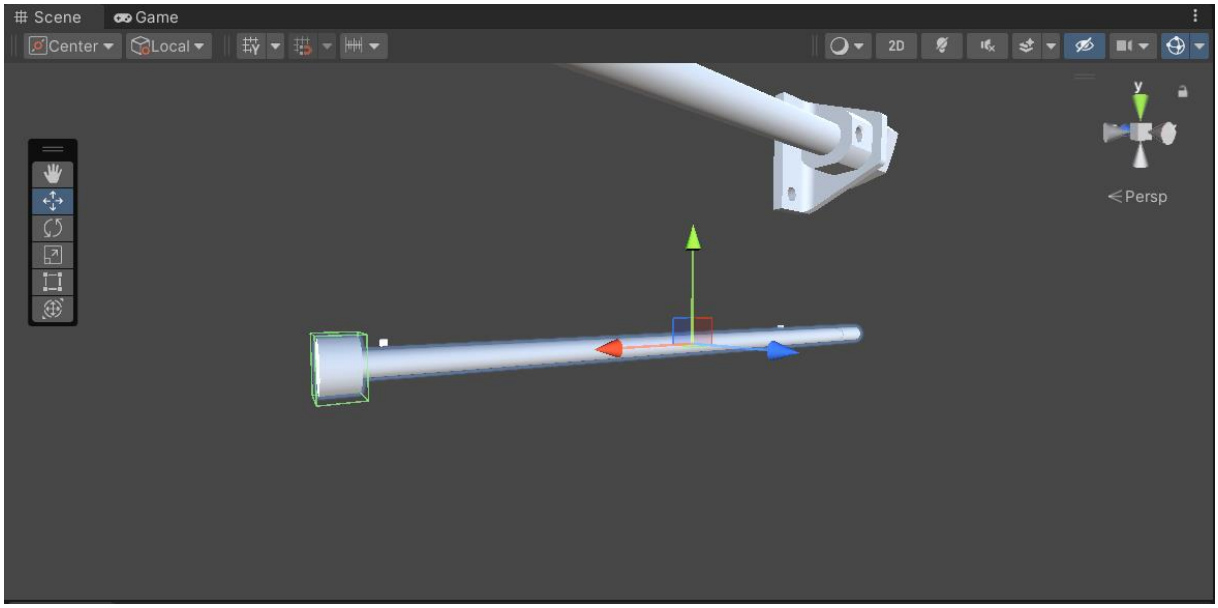



**Figura 39 Box Collider del Vástago ajustado**

El funcionamiento de estos sensores se basa en la colisión del pistón, el cual tiene un Box Collider, y los diferentes cubos que actúan como Triggers. Cuando el pistón colisiona con uno de los cubos, se invoca la función mencionada en el

apartado de Cómo modelar un objeto en Unity, donde se establece una variable como verdadera, la cual se encarga de detener el movimiento del vástago.

### 4.4.3. Implementación de interactividad

Una vez implementados estos sensores se explicarán los scripts necesarios para la manipulación del vástago.

- **Funcionalidad cilindro**

Se creará un script llamado ‘Cilindro’ que heredará de ‘MonoBehaviour’, y que se encargará del movimiento del vástago dependiendo de unos botones de activación. Este script se asociará al objeto donde se encuentre el cilindro entero. En la Figura 40 se puede observar la estructura que seguirá dicho script.

```
public class Cilindro : MonoBehaviour
{
    //DECLARACIÓN DE VARIABLES

    //MÉTODO Start()

    //MÉTODO Update()

    //MÉTODO SetMode()

}
```

**Figura 40 Organización Script Cilindro**

Antes de comenzar a explicar las variables del Script mostradas en la Figura 41, es importante mencionar que '[HideInInspector]' se emplea en el código con el fin de evitar que las variables situadas debajo aparezcan en la ventana de Inspector, con el fin de una mejor organización.

Para la declaración de variables, se asemeja a la Figura 32, donde habría que añadir una variable para almacenar la velocidad máxima con la que se moverá el vástago y una distancia mínima para tener el movimiento del vástago limitado en ambas direcciones.

```
public float speed_max = 50;
[HideInInspector]
public float speed_actual;
[HideInInspector]
public Vector3 new_pos;
private Vector3 startingPosition;
public float dis_min = 0.0f;
public float dis_max = 100;

public GameObject vastago;
```

**Figura 41 Variables Script Cilindro**

El método Start se mantendrá sin cambios según la Figura 33, mientras que en el método Update se realizarán unas modificaciones (Figura 42) para permitir su utilización tanto en movimientos horizontales como verticales.

Inicialmente, se obtiene la rotación actual del objeto usando 'vastago.transform.rotation.eulerAngles', lo cual devuelve los ángulos de rotación en grados en relación con los ejes X, Y y Z. Se evalúa la rotación en el eje Z del vástago mediante 'Mathf.Approximately(rotation.z, 0)' y 'Mathf.Approximately(rotation.z, 90)'. Estas condiciones verifican si el objeto está en posición horizontal (0 grados) o vertical (90 grados).

Cuando el objeto está en posición horizontal, se activa la primera rama del condicional 'if'. Aquí, se calcula la nueva posición con el movimiento hacia la derecha en el eje X. Luego, se aplica 'Mathf.Clamp' a la coordenada X de 'new_pos' para asegurar que el objeto no se desplace fuera del rango definido por 'startingPosition.x - dis_min' (límite inferior) y 'startingPosition.x + dis_max' (límite superior).

En el caso de que el objeto esté en posición vertical, se activa la segunda rama del condicional 'else if'. En este caso se mueve hacia abajo en el eje Y. De manera similar, se aplica 'Mathf.Clamp' a la coordenada Y de 'new_pos' para garantizar que el objeto no se desplace fuera del rango definido por 'startingPosition.x - dis_max' (límite inferior) y 'startingPosition.x + dis_min' (límite superior).

Finalmente, la posición del objeto vastago se actualiza con 'vastago.transform.position = new_pos'.

```
void Start()
{
    startingPosition = vastago.transform.position;
}

Mensaje de Unity | 1 referencia
public void Update()
{
    Vector3 rotation = vastago.transform.rotation.eulerAngles;

    if (Mathf.Approximately(rotation.z, 0))
    {
        // Movimiento horizontal, mueve hacia la derecha en el eje X
        new_pos = vastago.transform.position + (Vector3.right * speed_actual * Time.deltaTime);

        float clampedX = Mathf.Clamp(new_pos.x, startingPosition.x - dis_min, startingPosition.x + dis_max);
        if (new_pos.x != clampedX)
        {
            new_pos.x = clampedX;
            speed_actual = 0;
        }

        vastago.transform.position = new_pos;
    }
    else if (Mathf.Approximately(rotation.z, 90))
    {
        // Movimiento vertical, mueve hacia abajo en el eje Y
        new_pos = vastago.transform.position + (Vector3.down * speed_actual * Time.deltaTime);

        float clampedY = Mathf.Clamp(new_pos.y, startingPosition.y - dis_max, startingPosition.y + dis_min);
        if (new_pos.y != clampedY)
        {
            new_pos.y = clampedY;
            speed_actual = 0;
        }

        vastago.transform.position = new_pos;
    }
}
```

**Figura 42 Método ‘Start’ y ‘Update’ del Script Cilindro**

El método SetMode que se muestra en la Figura 43 define una función llamada SetMode que toma un parámetro de tipo string llamado ‘mode’. Dentro de esta función, se utiliza una estructura de control switch para determinar qué acción tomar en función del valor del parámetro ‘mode’. Si ‘mode’ es igual a "Avanzar", la variable ‘speed_actual’ se establece en el valor máximo de velocidad (‘speed_max’). Si ‘mode’ es igual a "Retroceder", ‘speed_actual’ se establece en el valor negativo del máximo de velocidad (- ‘speed_max’). Si ‘mode’ es igual a "Parar", ‘speed_actual’ se establece en 0. En caso de que ‘mode’ no coincida con ninguno de estos casos, ‘speed_actual’ también se establece en 0.

```
public void SetMode(string mode)
{

    switch (mode)
    {
        case "Avanzar":
            speed_actual = speed_max;
            break;

        case "Retroceder":
            speed_actual = -speed_max;
            break;

        case "Parar":
            speed_actual = 0;
            break;

        default:
            speed_actual = 0;
            break;
    }
}
```

**Figura 43 Método ‘SetMode’ del Script Cilindro**

### 4.4.4. Implementación de elementos UI

Para habilitar el movimiento, se incluirán tres botones: uno para "Avanzar", otro para "Retroceder" y el último para "Parar" el vástago. Después de agregar los botones con sus respectivos nombres, se les asignará la función correspondiente según se detalla en la sección "Button" de Elementos Propios del UI en la sección “Button”. En este caso, cada botón se asociará con el objeto "Cilindro", que contiene el script 'Cilindro'. Se seleccionará la función SetMode del script y, dado que esta función requiere un parámetro de tipo string, se establecerá la palabra correspondiente para cada botón. Como se muestra en la Figura 44, al hacer clic en el botón, activará el caso "Avanzar" dentro de la función SetMode.

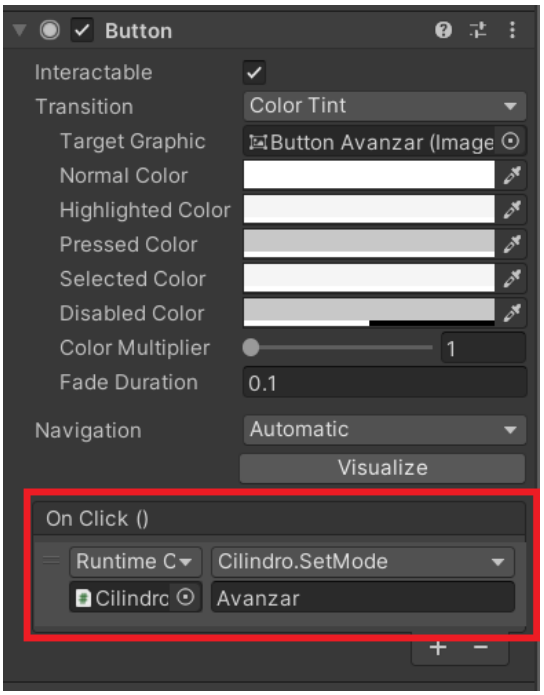



**Figura 44 Inspector del botón Avanzar**

- **Funcionalidad Sensor**

Se creará un script llamado ‘Sensor’ que heredará de ‘MonoBehaviour’, el cual se encargará de modificar el LED dependiendo de los Colliders. Este script se asociará a los dos objetos que ejercen de Trigger para el sensor. En la Figura 45 se muestra la organización en la que se basará el script.

```
public class Sensor : MonoBehaviour
{
    //DECLARACIÓN DE VARIABLES

    //MÉTODO Update()

    //MÉTODO OnTriggerExit()

    //MÉTODO OnTriggerEnter()

}
```

**Figura 45 Organización del Script Sensor**

Para este script, simplemente se necesitarán declarar las variables que definen los colores necesarios, la variable privada de tipo Renderer para acceder al LED y por último una variable booleana llamada ‘Fin’ que marcará el estado del LED. Véase en la Figura 46.

```
public Renderer led;
[HideInInspector]
public Color colorRojoFuerte = new Color(2.0f, 0.0f, 0.0f);  // Rojo intenso
[HideInInspector]
public Color colorRojoFlojo = new Color(0.4f, 0.0f, 0.0f);  // Rojo menos intenso

public bool Fin = false;
```

**Figura 46 Variables del Script Sensor**

La Figura 47, nos muestra tres métodos. En el método Update(), solo se necesita que esté constantemente actualizando el color del LED según la variable booleana 'Fin'. Si "Fin" es true, el color del material se establece en "colorRojoFuerte"; de lo contrario, se establece en "colorRojoFlojo".

El siguiente método OnTriggerExit() se llama cuando otro objeto sale del área de colisión asociada con este objeto. Dentro de este método, la variable booleana "Fin" se establece en false, lo que indica que el objeto ya no está en el área de colisión. Por último, el método OnTriggerEnter() se llama cuando otro objeto entra en el área de colisión asociada con este objeto. Dentro de este método, la variable booleana "Fin" se establece en true, indicando que el objeto está ahora dentro del área de colisión.

```
public void Update()
{
    led.material.color = Fin ? colorRojoFuerte : colorRojoFlojo;
}

Mensaje de Unity | 1 referencia
public void OnTriggerExit(Collider other)
{
    Fin = false;
}
Mensaje de Unity | 1 referencia
public void OnTriggerEnter(Collider other)
{
    Fin = true;
}
```

**Figura 47 Método 'Update', 'OnTriggerExit' y 'OnTriggerEnter' del Script Sensor**

### 4.4.5. Pruebas y ajustes

A continuación, se ofrecerá un código QR, con el cuál se accederá a un video que muestra la creación desde cero de este caso de uso, y su funcionamiento:

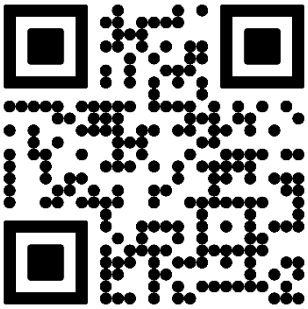


**Figura 48 QR - Modelo Digital**

# 5. Sombra Digital

Una Sombra Digital consiste en una representación digital e interacción con el sistema físico. Ambos están interconectados, estableciendo así una comunicación unidireccional entre ellos. Específicamente, se llevará a cabo una lectura de datos desde la planta real hacia la Sombra Digital.

En este punto, se abordará la elección del protocolo de comunicación que se usará, la implementación de los datos de las entradas digitales y de las bobinas que nos llegan mediante ese protocolo, previamente elegido, y finalizará con el caso de uso del Modelo Digital pero con el protocolo de comunicación elegido y con la adición del cálculo de la velocidad media real, lo cual se podrá observar en el código QR adjunto al final. Este contendrá un vídeo donde se mostrará más en detalle el funcionamiento de este caso.

## 5.1. Elección del protocolo de comunicación

A la hora de establecer la comunicación, existen varios protocolos de comunicación y métodos para conseguirlo. A continuación, se presentan algunos de ellos:

- **UDP (User Datagram Protocol)**

Es un protocolo de comunicación que ofrece una comunicación no orientada a la conexión y sin garantía de entrega de datos. Se utiliza en aplicaciones donde la velocidad es prioritaria sobre la integridad de los datos. En Unity, UPD podría ser útil para la trasmisión de datos de baja prioridad o para aplicaciones donde la latencia es crítica.

- **OPC UA (Open Platform Communications Unified Architecture)**

Es un estándar de comunicación interoperable diseñado específicamente para entornos industriales. OPC UA permite la comunicación entre dispositivos de diferentes fabricantes de manera segura y confiable. Proporciona funcionalidades avanzadas como seguridad, descubrimiento de servicios y semántica de datos, lo que hace ideal para integraciones complejas entre Unity y sistemas de control industrial.

- **MODBUS**

Es un protocolo de comunicación de propósito general ampliamente utilizado en la automatización industrial. Proporciona una comunicación sencilla y confiable entre dispositivos, con un enfoque en la lectura y escritura de datos en tiempo real.

Tras evaluar diversos protocolos de comunicación, se ha determinado que MODBUS es la opción más adecuada para establecer la comunicación por varias razones, detalladas en el Apéndice A: Modbus.

## 5.2. Implementación de la lectura de una entrada digital

Este apartado se centrará de nuevo en el LED, cuyo funcionamiento se ha examinado anteriormente en el apartado Objetos Discretos. En esta ocasión, se mejorará su rendimiento mediante la adición de una comunicación que permita recibir una señal de “Discrete Input”, la cual indicará el estado del LED. La dirección de este input se especificará mediante un byte y un bit, como se explica en Simulador PLC, siendo completamente ajustables según las necesidades.

A continuación, se presentarán los dos scripts requeridos para su correcto funcionamiento. Uno de ellos, el encargado de la conexión Modbus, se explicará tanto para que pueda recibir una entrada digital como para que reciba el estado de una bobina, de cara a los siguientes apartados.

### 5.2.1. Conexión con Modbus

En este apartado se creará un Script llamado ‘Modbus’, el cual heredará del driver ' UModbusTCP’. Este script se organizará según se muestra en la Figura 49 y se encargará de gestionar la comunicación Modbus entre la aplicación y dispositivos externos. Este script será agregado a un objeto vacío creado para tal propósito, permitiendo la asignación de los elementos necesarios. En este caso, se debe asignar únicamente el Toggle de la conexión.

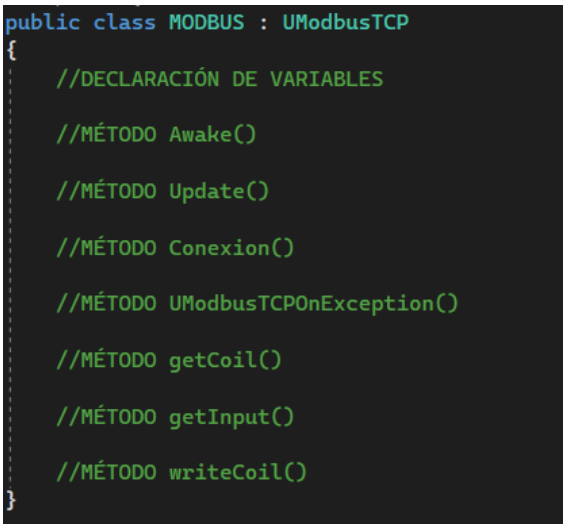



**Figura 49 Organización del Script MODBUS**

En la Figura 50, la primera pareja de variables, ‘updateInterval’ y ‘timeSinceLastUpdate’, se utilizan para gestionar la frecuencia de actualización de ciertas operaciones dentro del programa. ‘updateInterval’ representa el intervalo de tiempo, en segundos, entre cada actualización, al que se le asigna el valor de 0.5, mientras que ‘timeSinceLastUpdate’ registra el tiempo transcurrido desde la última actualización.

Luego, las variables ‘m_oUModbusTCP’ y ‘m_oUModbusTCPException’ están relacionadas con la comunicación con dispositivos Modbus TCP. ‘m_oUModbusTCP’ que se explica en el apartado Conexión, mientras que ‘m_oUModbusTCPException’ se emplea para manejar excepciones relacionadas con la comunicación Modbus TCP.

La variable ‘ValoresInputs’ es un array de bytes que almacena los valores de las entradas discretas leídas. Mientras que la variable ‘ValoresCoils’ es un array de bytes que almacena los valores del estado de las bobinas.

Las variables iniCoils, numCoils, iniInput, numInput, representan respectivamente la dirección inicial de las bobinas, el número de bobinas a leer, la dirección inicial de las entradas digitales y el número de entradas a leer.

Posteriormente, las variables sIp y usPort representan la dirección IP del dispositivo Modbus TCP y el puerto de comunicación utilizado para la conexión, el cual normalmente se utilizará el puerto 502.

Finalmente, el objeto conexion es un elemento de interfaz de usuario (UI) ya explicado en el apartado Elementos Propios del UI.

```
public float updateInterval = 0.5f; // Intervalo de actualización en segundos
[HideInInspector]
public float timeSinceLastUpdate = 0f;

//UModbusTCP
UModbusTCP m_oUModbusTCP;
UModbusTCP.ExceptionData m_oUModbusTCPException;

[HideInInspector]
public byte[] ValoresInputs;
[HideInInspector]
public byte[] ValoresCoils;

public ushort iniCoils = 0;
public ushort numCoils = 16;
public ushort iniInput = 0;
public ushort numInput = 16;

public string sIp;
public ushort usPort = Convert.ToUInt16("502");

public Toggle conexion;
```

**Figura 50 Variables del Script MODBUS**

La Figura 51 muestra el método Awake(), el cual se activa durante la inicialización del script. En este caso específico, el método comienza asignando null a las variables ‘m_oUModbusTCP’ y ‘m_oUModbusTCPException’, para asegurar de que estas variables se inicialicen correctamente.

Luego, se asigna a ‘m_oUModbusTCP’ la instancia actual de ‘UModbusTCP’ mediante Instance. También se inicializa el array ValoresCoils y ValoresInputs con el tamaño especificado por numCoils y numInputs, respectivamente.

```
void Awake()
{

    //UModbusTCP
    m_oUModbusTCP = null;
    m_oUModbusTCPException = null;

    m_oUModbusTCP = Instance;

    ValoresCoils = new byte[numCoils];
    ValoresInputs = new byte[numInput];

}
```

**Figura 51 Método 'Awake' del Script MODBUS**

El método Update() ilustrado en la Figura 52, se emplea para supervisar el tiempo transcurrido desde la última actualización y llevar a cabo la actualización periódica de los valores de entrada. En caso de que el Toggle de la conexión se encuentre activo, se intenta establecer la conexión mediante una llamada a su función correspondiente. Por otro lado, si el Toggle no está activo, se procede a desconectar la conexión.

```
public void Update()
{

    timeSinceLastUpdate += Time.deltaTime;

    if (timeSinceLastUpdate >= updateInterval)
    {
        timeSinceLastUpdate = 0f;
        if (conexion.isOn)
        {
            Conexion();
        }
        else
        {
            Disconnect();
        }

    }
}
```

**Figura 52 Método 'Update' del Script MODBUS**

La Figura 53 muestra el método encargado de establecer la conexión MODBUS y de leer los valores de entrada correspondientes. En caso de no estar conectado, procede a intentar establecer la conexión. Posteriormente, configura un controlador de excepciones para gestionar posibles es de conexión y finalmente, se procede a la lectura de datos ya explicada en el apartado Lectura de Datos. Este método será llamado de forma controlada en el método Update().

```
public void Conexion()
{

    if (!m_oUModbusTCP.connected)
    {
        m_oUModbusTCP.Connect(sIp, usPort);
    }

    //Exception callback
    if (m_oUModbusTCPException != null)
    {
        m_oUModbusTCP.OnException -= m_oUModbusTCPException;
    }
    m_oUModbusTCPException = new UModbusTCP.ExceptionData(UModbusTCPOnException);
    m_oUModbusTCP.OnException += m_oUModbusTCPException;


    //Aqui se obtiene la cadena bytes para las Discrete Inputs
    ValoresInputs = new byte[numInput];
    m_oUModbusTCP.ReadDiscreteInputs(1, 1, iniInput, numInput, ref ValoresInputs);

    //Aqui se obtiene la cadena bytes para las Coils
    ValoresCoils = new byte[numCoils];
    m_oUModbusTCP.ReadCoils(1, 1, iniCoils, numCoils, ref ValoresCoils);

}
```

**Figura 53 Método 'Conexion' del Script MODBUS**

Se puede observar en la Figura 54, el método que se invoca cuando se genera una excepción durante la comunicación MODBUS. En tal situación, se limita a imprimir un mensaje de excepción en la consola de depuración de Unity.

```
void UModbusTCPOnException(ushort _oID, byte _oUnit, byte _oFunction, byte _oException)
{
    Debug.Log("Exception: " + _oException);
}
```

**Figura 54 Método 'UModbusTCPOnException' del Script MODBUS**

En la Figura 55 se observa los métodos encargados de recuperar el estado de un bit específico dentro de un byte en los arrays de bytes ValoresCoils y ValoresInputs. Cada método toma dos parámetros: b, que representa el índice en el array, y bit, que indica el número de bit dentro del byte seleccionado.

Dentro de cada método, se realiza un desplazamiento de bits hacia la derecha en el valor obtenido de la posición b en el array correspondiente. Luego, se aplica una operación AND a nivel de bits con 1 para aislar el bit deseado y descartar los otros bits. Si el bit deseado es 1, el resultado será 1; si es 0, el resultado será 0.

Finalmente, se verifica si el resultado de la operación AND no es igual a 0. Si este es el caso, significa que el bit seleccionado está activado, por lo que se devuelve true. De lo contrario, se devuelve false, indicando que el bit está desactivado.

```
public bool getCoil(byte b, byte bit)
{

    return (((ValoresCoils[b] >> bit) & 1) != 0);
}
3 referencias
public bool getInput(byte b, byte bit)
{
    return (((ValoresInputs[b] >> bit) & 1) != 0);
}
```

**Figura 55 Método 'getCoil' y 'getInput' del Script MODBUS**

Por último, en la Figura 56 se encuentra el método encargado para la escritura de datos, necesaria para el apartado de Gemelo Digital. Este método toma tres parámetros: b, que representa el índice en el array, bit, que indica el número de bit dentro del byte seleccionado y un booleano estado, que indica el valor a escribir en la coil.

El primer paso dentro del método es calcular la dirección de la coil. Esta dirección se determina mediante la fórmula (b * 8 + bit), que convierte los bytes proporcionados en una dirección única de tipo ushort. Posteriormente, se define un array de bytes llamado bResult, con una longitud de 12, que se utilizará para almacenar el resultado de la operación MODBUS.

Finalmente, se invoca el método WriteSingleCoils del objeto m_oUModbusTCP, que es responsable de ejecutar la escritura en la coil.

```
public void writeCoil(byte b, byte bit, bool estado)
{
    ushort coilAddress = (ushort)(b * 8 + bit);
    byte[] bResult = new byte[12];
    m_oUModbusTCP.WriteSingleCoils(1, 1, coilAddress, estado, ref bResult);
}
```

**Figura 56 Método 'writeCoil' del Script MODBUS**

### 5.2.2. Funcionalidad del Sensor

Se desarrollará el Script llamado ‘Sensor_Modbus’ que heredará de la clase ‘Sensor’, vista en el apartado de Caso de uso: Manipulación de un vástago de doble efecto. Este script es destinado al funcionamiento del sensor que dependerá de una entrada digital. Este script será agregado al objeto de cada final de carrera que se añada, permitiendo la asignación de los elementos necesarios. En este caso, se debe asignar únicamente la malla del LED que se encuentra en su interior. Para

comenzar se mostrará en la Figura 57 la estructura que tendrá este script, para posteriormente explicarla.

```
public class Sensor_Modbus : Sensor
{
    //DECLARACIÓN DE VARIABLES

    //MÉTODO Awake()

    //MÉTODO Update()
}
```

**Figura 57 Organización del Script Sensor_Modbus**

La Figura 58 muestra la declaración de dos variables de tipo Byte para representar tanto el byte como el bit que indicarán la posición del LED en los registros.

Posteriormente, se declara una variable llamada 'modbusScript' del tipo MODBUS. Esta variable se utiliza para hacer referencia al script MODBUS visto anteriormente.

Al heredar del script 'Sensor', tiene total acceso a las variables declaradas y métodos de dicho script.

```
    public byte Sensor_byte = 0;
    public byte Sensor_bit = 0;

    private MODBUS modbusScript;
```

**Figura 58 Variables del Script Sensor_Modbus**

Para el método Awake(), ilustrado en la Figura 59, se inicializa la variable 'modbusScript' utilizando 'FindObjectOfType<MODBUS>()'. Esta función busca y devuelve una instancia del componente MODBUS en la escena, para poder así acceder al script 'MODBUS'.

```
void Awake()
{
    modbusScript = FindObjectOfType<MODBUS>();
}
```

**Figura 59 Método 'Awake' del Script Sensor_Modbus**

Por último el método Update (Figura 60) verifica si la referencia modbusScript, que hace referencia al objeto de la clase MODBUS, no es nula. Esta validación se realiza para asegurar la correcta inicialización del objeto modbusScript en el método Awake. Posteriormente, en caso de que modbusScript no sea nulo, se invoca al método getInput del objeto modbusScript, suministrándole como argumentos los valores de Sensor_byte y Sensor_bit. El resultado de esta llamada es asignado a la variable Fin, perteneciente al script 'Sensor'. Finalmente, se invoca al método Update de la clase base (Sensor) mediante base.Update(), garantizando así que el comportamiento por defecto del método Update de la clase base sea ejecutado de manera adecuada.

```
public new void Update()
{

    if (modbusScript != null)
    {
        Fin = modbusScript.getInput(Sensor_byte, Sensor_bit);
    }
    base.Update();
}
```

**Figura 60 Método 'Update' del Script Sensor_Modbus**

## 5.3. Implementación de la lectura de una bobina

Este apartado abordará de nuevo el cilindro, cuyo funcionamiento se ha examinado anteriormente en el apartado Objetos Dinámicos. En esta ocasión, se creará un script para que reciba el estado de una bobina y según este estado dependa si el vástago se mueve o no se mueve.

Para este ejemplo, al necesitar una comunicación y recibir datos, necesitaremos el script trabajado en el apartado Conexión con Modbus. Sin embargo, en este caso, en lugar de recibir 'Discrete Inputs', se recibirán 'Coils', por lo que se utilizará el método 'getCoil'.

A continuación, se explicará el segundo script necesario para su correcto funcionamiento.

### 5.3.1. Funcionalidad del Cilindro

Se creará un script llamado 'Cilindro_Modbus' que heredará de la clase 'Cilindro', vista en el apartado de Caso de uso: Manipulación de un vástago de doble

efecto, y que tendrá la estructura ilustrada en la Figura 61, la cual se explicará a continuación.

```
public class Cilindro_Modbus : Cilindro
{
    //DECLARACIÓN DE VARIABLES

    //MÉTODO Awake()

    //MÉTODO Update()

}
```

**Figura 61 Organización del Script Cilindro_Modbus**

Para la declaración de variables (Figura 62) primero se definen variables de tipo byte que van a representar la dirección de la bobina que va a controlar tanto el movimiento cuando se retrae (‘cilRe_byte’ y ‘cilRe_bit’) como el movimiento de expansión (‘cilEx_byte’ y ‘cilEx_bit’).

Por otro lado, se declarará la variable llamada ‘modbusScript’ del tipo MODBUS, como en el apartado anterior.

Al heredar del script ‘Cilindro’, tiene total acceso a las variables declaradas y métodos de dicho script.

```
    public byte cilEx_byte = 0;
    public byte cilEx_bit = 0;
    public byte cilRe_byte = 0;
    public byte cilRe_bit = 1;

    private MODBUS modbusScript;
```

**Figura 62 Variables del Script Cilindro_Modbus**

En el método Awake() (Figura 63) se aplica lo mismo que en el apartado Funcionalidad del .

```
void Awake()
{
    modbusScript = FindObjectOfType<MODBUS>();

}
```

**Figura 63 Método 'Awake' del Script Cilindro_Modbus**

El método Update que se observa en la Figura 64 es el encargado de gestionar el movimiento del cilindro en función de los estados de las bobinas recibidos. Inicia utilizando el método 'getCoil' de la clase MODBUS para determinar si la bobina encargada de la retracción está activa. En caso afirmativo, ejecuta la función SetMode("Retroceder") de la clase Cilindro.

A continuación, examina si la bobina de expansión está activa, y si es así, procede a llamar a la función SetMode("Avanzar") de la clase Cilindro. Posteriormente, verifica si ninguna de las bobinas está activa y si la posición del cilindro, representada por new_pos.x, se encuentra dentro de los límites definidos por dis_min y dis_max. Si no se encuentra dentro de los límites, se invoca SetMode("Parar"), lo que pararía el cilindro.

Finalmente, se llama al método Update de la clase base (Cilindro). Esto garantiza el movimiento del cilindro según la velocidad establecida previamente mediante el método SetMode.

```
new void Update()
{

    if (modbusScript.getCoil(cilRe_byte, cilRe_bit))
    {
        SetMode("Retroceder");
    }
    else if (modbusScript.getCoil(cilEx_byte, cilEx_bit))
    {
        SetMode("Avanzar");
    }
    else if (new_pos.x >= dis_max || new_pos.x <= dis_min)
    {
        SetMode("Parar");
    }
    base.Update();

}
```

**Figura 64 Método 'Update' del script Cilindro_Modbus**

## 5.4. Caso de uso: Manipulación de un vástago de doble efecto mediante Modbus

En esta implementación, se integrarán los tres scripts previamente desarrollados: 'MODBUS', 'Sensor_Modbus' y 'Cilindro_Modbus'. Además, se creará un nuevo script encargado de calcular el tiempo promedio del cilindro en la planta real y también se calculará la velocidad para asociarla al cilindro digital con el objetivo de mejorar su similitud con el comportamiento real. Además, este script

servirá como punto de partida para el desarrollo del script definitivo en el siguiente apartado (Gemelo Digital).

### 5.4.1. Cálculo del Tiempo Promedio real

Para calcular el tiempo promedio que tarda el cilindro real en realizar un movimiento, se desarrollará un script denominado 'Supervisor_Tiempo', que heredará de la clase 'MonoBehaviour', y será agregado a un objeto vacío en la Jerarquía para poder seleccionarlo y añadir los componentes necesarios. Este script se encargará de, al activarse un toggle, permitir que el cilindro real realice movimientos mientras el cilindro digital permanece detenido. Posteriormente, se calculará el tiempo transcurrido en cada movimiento y se contabilizará el número total de movimientos realizados. Esto permitirá obtener el tiempo promedio de desplazamiento del cilindro real y posteriormente poder calcular la velocidad promedio. En la Figura 65 se muestra la estructura que seguirá este script.

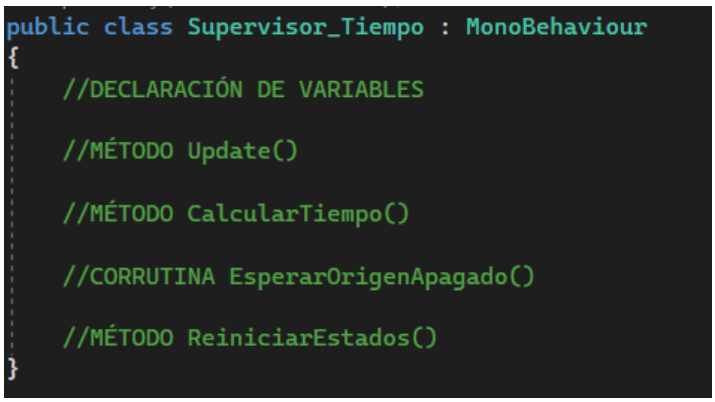



**Figura 65 Organización del Script Supervisor_Tiempo**

Como se muestra en la Figura 66, se definen variables públicas de los tipos MODBUS, Cilindro_Modbus y dos instancias de Sensor_Modbus, que servirán para interactuar con los componentes MODBUS y los sensores asociados al cilindro respectivamente.

Continuamos con la declaración de dos variables de tipo TMP_Text, diseñadas para mostrar tanto el tiempo como el número de movimientos del cilindro. Además, Se incluye también una variable pública de tipo Toggle, que se utiliza para activar el cálculo del tiempo.

Seguidamente, se inicializan variables públicas para controlar el estado de los sensores y para almacenar tiempos y conteos relevantes para el cálculo del tiempo, así como una variable para guardar el tiempo promedio calculado.

Por último, se inicializan una variable para almacenar la carrera del cilindro real, y otra para almacenar la velocidad promedio calculada.

```
public MODBUS modbusScript;
public Cilindro_Modbus cilindroModbus;
public Sensor_Modbus Sensor_origen;
public Sensor_Modbus Sensor_destino;

public TMP_Text tiempoText;
public TMP_Text movimientosText;
public Toggle calcularTiempo;

private bool origenActivado = false;
private bool destinoActivado = false;
private float tiempoInicioOrigen;
private float tiempoInicioDestino;
private int intentos = 0;
private float sumaTiempos = 0f;
private float tiempoPromedio;

private float carreraCilindro = 100f; //mm
private float VelocidadPromedio;
```

**Figura 66 Variables del Script Supervisor_Tiempo**

En el método Update de la Figura 67, primero, se verifica si el toggle calcularTiempo está activado. En caso afirmativo, se procede a llamar a la función CalcularTiempo().

En caso contrario, es decir, si el interruptor está desactivado, se asigna la velocidad promedio calculada a la variable de la velocidad máxima del script 'Cilindro_Modbus', y se llama a la función ReiniciarEstados().

```
void Update()
{
    if (calcularTiempo.isOn)
    {
        CalcularTiempo();
    }
    else
    {
        cilindroModbus.speed_max = VelocidadPromedio;
        ReiniciarEstados();

    }
}
```

**Figura 67 Método 'Update' del Script Supervisor_Tiempo**

En la Figura 68 se encuentra el método CalcularTiempo(), el cual comienza con la lectura de los estados actuales de los sensores de origen y destino. Estos sensores son monitoreados continuamente para determinar si están activados o desactivados. Después, se comprueba el estado del sensor de origen. Si este sensor no estaba activado previamente pero ahora está encendido, se inicia una corrutina denominada ‘EsperarOrigenApagado()’.

Posteriormente, se monitorea el estado del sensor de destino. Si este sensor no estaba activado previamente, el sensor de origen está activado, y el sensor de destino ahora está encendido, se registra el tiempo de activación del sensor de destino y se marca como activado.

Cuando ambos estados están activados, se procede al cálculo del tiempo transcurrido entre la activación de los sensores. Se desactivan las variables de activación de los sensores y se calcula la diferencia absoluta entre los tiempos de activación del sensor de origen y del sensor de destino. Este tiempo entre sensores se acumula en una variable de suma total de tiempos y se incrementa un contador de intentos, que representa el número total de movimientos del cilindro.

Con los datos recopilados, se calcula el tiempo promedio de los movimientos del cilindro. Este valor se va a mostrar en un componente de texto junto con el número de movimientos realizados por el cilindro. Además, se calcula la velocidad promedio del cilindro basada en la ‘carreraCilindro’ y el tiempo promedio calculado.

```
void CalcularTiempo()
{
    bool origenEstadoActual = modbusScript.getInput(Sensor_origen.Sensor_byte, Sensor_origen.Sensor_bit);
    bool destinoEstadoActual = modbusScript.getInput(Sensor_destino.Sensor_byte, Sensor_destino.Sensor_bit);

    if (!origenActivado && origenEstadoActual)
    {
        StartCoroutine(EsperarOrigenApagado());
    }
    else if (!destinoActivado && origenActivado && destinoEstadoActual)
    {
        tiempoInicioDestino = Time.time;
        destinoActivado = true;
    }

    if (origenActivado && destinoActivado)
    {
        origenActivado = false;
        destinoActivado = false;

        float tiempoEntreSensores = Mathf.Abs(tiempoInicioOrigen - tiempoInicioDestino);
        sumaTiempos += tiempoEntreSensores;
        intentos++;

        tiempoPromedio = (sumaTiempos / intentos);
        tiempoText.text = tiempoPromedio.ToString() + " s";
        movimientosText.text = "El cilindro real lleva " + intentos.ToString() + " movimiento(s) de " + Sensor_origen + " a " + Sensor_destino;
        VelocidadPromedio = (carreraCilindro / tiempoPromedio);
    }
}
```

**Figura 68 Método ‘CalcularTiempo’ del Script Supervisor_Tiempo**

El método EsperarOrigenApagado, mostrado en la Figura 69, se implementa como una corrutina, lo que permite que se ejecute de manera asíncrona y no bloquee el flujo principal del programa. Las corrutinas se definen en Unity utilizando la interfaz ‘IEnumerator’ y se inician con el método ‘StartCoroutine’.

Dentro de este método se encuentra un bucle while, que verifica continuamente el estado del sensor de origen. Mientras el sensor permanezca encendido, la instrucción yield return null indica al motor de Unity que la corrutina debe suspenderse y continuar en el siguiente frame, antes de volver a comprobar el estado del sensor. Una vez que el sensor se apaga, se registra el tiempo actual en la variable tiempoInicioOrigen. Finalmente, se marca origenActivado como verdadero.

```
IEnumerator EsperarOrigenApagado()
{

    while (modbusScript.getInput(Sensor_origen.Sensor_byte, Sensor_origen.Sensor_bit))
    {
        yield return null;
    }

    tiempoInicioOrigen = Time.time;
    origenActivado = true;
}
```

**Figura 69 Método ‘EsperarOrigenApagado’ del Script Supervisor_Tiempo**

Por último, se ha implementado un método para restablecer el estado de varias variables importantes en el sistema de medición y control del cilindro. Véase en la Figura 70.

```
void ReiniciarEstados()
{
    origenActivado = false;
    destinoActivado = false;
    intentos = 0;
    sumaTiempos = 0f;
}
```

**Figura 70 Método 'ReiniciarEstados' del Script Supervisor_Tiempo**

Para mostrar el cálculo del tiempo promedio en Unity, se agregará un Panel a la escena, ubicado en una esquina superior, perteneciente a los Elementos Propios del UI. En este Panel se añadirá un toggle para activar el cálculo del tiempo, junto con tres TextMeshPro, por cada dirección de movimiento, para visualizar los resultados. Estos elementos se pueden observar en los recuadros rojos de la Figura 71. Uno de los TextMeshPro se utilizará para indicar la 'Tiempo Promedio (Dirección del movimiento)', definiéndose esta dirección con el nombre de los sensores, mientras que otro se ubicará a su derecha para mostrar el valor calculado del tiempo. Justo debajo se colocará un tercer TextMeshPro para mostrar el número de movimientos realizados por el cilindro para dicho cálculo.

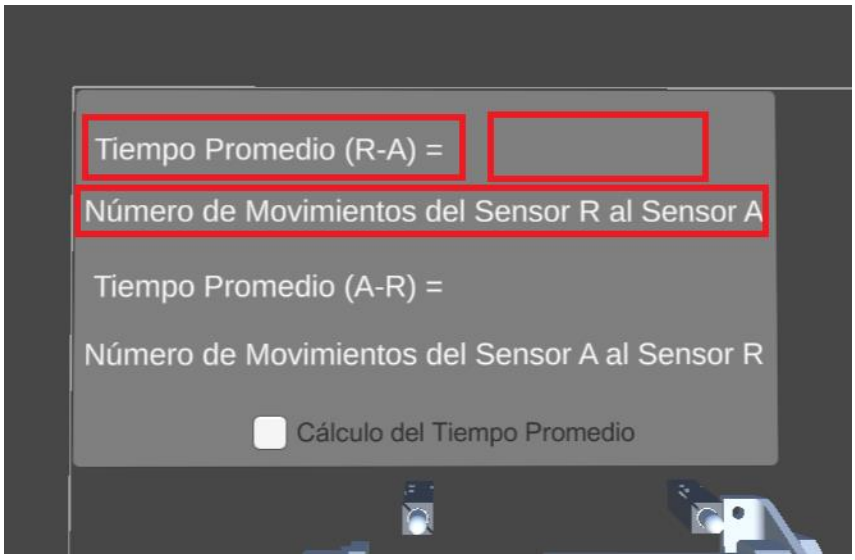



**Figura 71 Panel de Velocidad Media**

Para concluir, se proporcionará un código QR que permitirá acceder a una página donde se mostrarán dos videos, uno con la implementación del panel en Unity y otro con el funcionamiento, donde se podrá observar tanto el Unity, el PLC y el cilindro real:

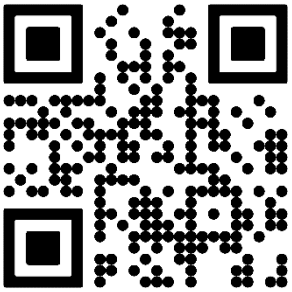


**Figura 72 QR - Sombra Digital**

# 6. Gemelo Digital

La diferencia fundamental entre el Gemelo Digital y la Sombra Digital radica en el establecimiento de una comunicación bidireccional entre la planta física y su representación virtual. Mientras que en la Sombra Digital se realiza una lectura de entradas digitales mediante el método "Read Discrete Inputs" y la lectura de bobinas con "Read Coils", en el Gemelo Digital se continúan utilizando estas funciones, pero además se implementa la función "Write Single Coils" para lograr la bidireccionalidad deseada. Esta última función se analiza con mayor detalle en el Apéndice A: Modbus.

En este apartado se presenta la implementación de un diagnóstico de fallos, complementado con un código QR al final. Este QR proporciona acceso a un video completo que detalla el funcionamiento del proyecto en su fase final.

## 6.1. Diagnóstico de fallos

Para tener un Gemelo Digital coherente es necesario que tenga una mínima inteligencia artificial en algún módulo, el cual puede estar implementado en cualquier lugar, en mi caso estará implementado en un Script que tendrá como finalidad hacer un diagnóstico de fallos.

El sistema operará basándose en el cálculo del tiempo de desplazamiento del vástago, tal como se describe en el apartado Cálculo del Tiempo Promedio real. Una vez obtenido el tiempo necesario para un ciclo de movimiento se almacenará en una variable de referencia. Este tiempo servirá como base para el diagnóstico de fallos. En caso de que el tiempo de un ciclo de desplazamiento del vástago supere este tiempo de referencia multiplicado por un margen, el cual es totalmente modificable, saltará una alarma roja que indicará que se ha detectado un fallo. Mientras no se detecten fallos, se mostrará una alarma de color verde que indicará un funcionamiento adecuado.

Para implementar el proceso descrito previamente, se creará un script denominado ‘Sensor_Tiempo’, que heredará de la clase ‘Supervisor_Tiempo’. Este script será agregado a un objeto vacío en la Jerarquía de Unity, lo que permitirá seleccionarlo y añadir los componentes necesarios.

La estructura que seguirá este script se visualiza en la Figura 73.

```
public class Sensor_Tiempo : Supervisor_Tiempo
{
    //DECLARACIÓN DE VARIABLES

    //MÉTODO Start()

    //MÉTODO Update()

    //MÉTODO CalcularTiempo()

    //MÉTODO DiagnosticarFallos()

    //CORRUTINA EsperarOrigenApagado()

    //CORRUTINA DetecciónFallos()

    //MÉTODO ReiniciarSistema()

    //MÉTODO MostrarAlarmaVerde()

    //MÉTODO MostrarAlarmaRoja()

    //MÉTODO InicializarAlarmas()
}
```

**Figura 73 Organización del Script Sensor_Tiempo**

Al heredar del script descrito en el apartado Cálculo del Tiempo Promedio real, tiene total acceso a las variables públicas, por lo que en la Figura 74 aparecen las variables que hay que añadir nuevas.

En primer lugar, se declara una variable para almacenar el margen de tiempo para el movimiento del cilindro. Esta variable es crucial para la lógica de detección de fallos, permitiendo determinar si el cilindro está operando fuera de los parámetros esperados.

A continuación, se declara una variable de tipo ‘TMP_Text’ que se utilizará para mostrar si funciona correctamente o si se ha detectado un fallo.

Se añaden las variables de tipo ‘GameObject’ que representarán los objetos de las alarmas visuales. Además, se declaran variables de tipo ‘bool’ para el control del código.

Por último, se especifican las variables de tipo ‘byte’ que indican la dirección del byte y el bit en el protocolo MODBUS, utilizados para la respuesta del cilindro.

```
private float Margen = 1.6f; // el cilindro tardaría más del 60% del tiempo que debería

public TMP_Text falloText;

public GameObject alarmaVerde;
public GameObject alarmaRoja;
private bool falloDetectado = false;
private bool TiempoCalculado = false;

public byte CoilRespuesta_byte = 0;
public byte CoilRespuesta_bit = 6;
```

**Figura 74 Variables del Script Sensor_Tiempo**

El método Start de la Figura 75, establece las condiciones iniciales necesarias para el correcto funcionamiento del sistema de diagnóstico de fallos. En primer lugar, llama al método 'InicializarAlarmas()', cuyo propósito es configurar las alarmas visuales que indican el estado del cilindro. Seguidamente, se invoca el método 'MostrarAlarmaVerde()', que asegura que la alarma verde, indicando un estado sin fallos, sea visible desde el inicio.

En el método Update de la misma figura, se verifica primero si el toggle 'calcularTiempo' está activado. Si es así, se llama al método 'CalcularTiempo()', encargado de calcular el tiempo de movimiento del cilindro entre los sensores. En el caso de que 'calcularTiempo' no esté activado y no se haya detectado ningún fallo ('falloDetectado' es false), se establece la velocidad máxima del cilindro ('cilindroModbus.speed_max') igual a la velocidad promedio calculada en el método 'CalcularTiempo()'. Posteriormente, se llama al método DiagnosticarFallos, que analizará el comportamiento del cilindro para detectar posibles fallos.

```
private void Start()
{
    InicializarAlarmas();
    MostrarAlarmaVerde();
}
Mensaje de Unity | 0 referencias
void Update()
{
    if (calcularTiempo.isOn)
    {
        CalcularTiempo();
    }
    else if (!falloDetectado)
    {
        cilindroModbus.speed_max = VelocidadPromedio;
        DiagnosticarFallos();
    }
}
```

**Figura 75 Método 'Start' y 'Update' del Script Sensor_Tiempo**

El método ‘CalcularTiempo()’, mostrado en la Figura 76, permanece igual que el explicado en el apartado Cálculo del Tiempo Promedio real, solamente habría que actualizar la variable ‘TiempoCalculado’ a true.

```
void CalcularTiempo()
{
    bool origenEstadoActual = modbusScript.getInput(Sensor_origen.Sensor_byte, Sensor_origen.Sensor_bit);
    bool destinoEstadoActual = modbusScript.getInput(Sensor_destino.Sensor_byte, Sensor_destino.Sensor_bit);

    if (!origenActivado && origenEstadoActual)
    {
        StartCoroutine(EsperarOrigenApagado());
    }
    else if (!destinoActivado && origenActivado && destinoEstadoActual)
    {
        tiempoInicioDestino = Time.time;
        destinoActivado = true;
    }

    if (origenActivado && destinoActivado)
    {
        origenActivado = false;
        destinoActivado = false;

        float tiempoEntreSensores = Mathf.Abs(tiempoInicioOrigen - tiempoInicioDestino);
        sumaTiempos += tiempoEntreSensores;
        intentos++;

        tiempoPromedio = (sumaTiempos / intentos);
        tiempoText.text = tiempoPromedio.ToString() + " s";
        movimientosText.text = "El cilindro real lleva " + intentos.ToString() + " movimiento(s) de " + Sensor_origen + " a " + Sensor_destino;
        VelocidadPromedio = carreraCilindro / tiempoPromedio;

        TiempoCalculado = true;

    }
}
```

**Figura 76 Método ‘CalcularTiempo’ del Script Sensor_Tiempo**

Una vez que el toggle para el cálculo del tiempo está desactivado y no hay ningún fallo detectado, se llamará al método de la Figura 77, que comenzará con la lectura del estado actual del sensor de origen, que estará monitoreado continuamente para determinar si está activado o desactivado. Después, se comprueba el estado del sensor. Si este sensor no estaba activado previamente pero ahora está encendido, y el tiempo ya fue calculado previamente, se inicia la corrutina ‘EsperarOrigenApagado()’.

Finalmente, si tanto el sensor de origen como el sensor de destino están activados (origenActivado && destinoActivado), se resetean sus estados a desactivados, estableciendo las variables ‘origenActivado’ y ‘destinoActivado’ a false. Este reset garantiza que el ciclo de detección de fallos puede comenzar de nuevo cuando se cumplan las condiciones adecuadas.

```
void DiagnosticarFallos()
{
    bool origenEstadoActual = modbusScript.getInput(Sensor_origen.Sensor_byte, Sensor_origen.Sensor_bit);
    if (!origenActivado && origenEstadoActual && TiempoCalculado)
    {
        StartCoroutine(EsperarOrigenApagado());
    }
    if (origenActivado && destinoActivado)
    {
        origenActivado = false;
        destinoActivado = false;
    }
}
```

**Figura 77 Método 'DiagnosticarFallos' del Script Sensor_Tiempo**

La corrutina 'EsperarOrigenApagado()' realiza la misma función que la explicada en el apartado Cálculo del Tiempo Promedio real, pero comprobando si el toggle del cálculo del tiempo está desactivado, si es así, iniciaría la corrutina 'DetecciónFallos()', la cual también se encuentra en la Figura 78.

Dentro de esta corrutina, se utiliza un bucle while que se ejecuta mientras el sensor de destino no esté activado, lo cual se verifica mediante la función 'getInput' del script MODBUS. Mientras el bucle está en ejecución, la variable 'destinoActivado' se establece en true, indicando que el proceso de detección de fallos está en marcha.

Durante cada iteración del bucle, se calcula el tiempo transcurrido desde que se inició el monitoreo, restando el tiempo actual al tiempo de inicio registrado. Este tiempo actual se compara con el tiempo promedio calculado previamente multiplicado por el margen predefinido. Si el tiempo transcurrido excede este valor calculado, el cilindro está tardando más de lo esperado en llegar a su destino, lo que detecta un posible fallo.

En caso de detectar un fallo, el texto de 'falloText' se actualiza para reflejar que se ha detectado el fallo. La variable 'falloDetectado' se establece en true y, además, se llama al método 'MostrarAlarmaRoja' para activar la alarma visual correspondiente, y se accede a la función 'writeCoil' del script MODBUS para actualizar una coil a true, indicando el fallo. Para finalizar la corrutina se utiliza la instrucción 'yield break'.

Si no se detecta ningún fallo, la corrutina sigue su ejecución y el bucle while se repite con la instrucción 'yield return null'.

```
IEnumerator EsperarOrigenApagado()
{
    while (modbusScript.getInput(Sensor_origen.Sensor_byte, Sensor_origen.Sensor_bit))
    {
        yield return null;
    }

    tiempoInicioOrigen = Time.time;
    origenActivado = true;

    if (!calcularTiempo.isOn)
    {
        StartCoroutine(DetecciónFallos());
    }
}
2 referencias
IEnumerator DetecciónFallos()
{
    while (!modbusScript.getInput(Sensor_destino.Sensor_byte, Sensor_destino.Sensor_bit))
    {
        destinoActivado = true;
        float tiempoActual = Time.time - tiempoInicioOrigen;
        if (tiempoActual > tiempoPromedio * Margen)
        {
            falloText.text = "Fallo detectado: El cilindro sufre algún problema.";
            falloDetectado = true;
            MostrarAlarmaRoja();
            modbusScript.writeCoil(CoilRespuesta_byte, CoilRespuesta_bit, true);
            yield break; // Detener la corrutina
        }
        yield return null;
    }
}
```

**Figura 78 Corrutinas ‘EsperarOrigenApagado’ y ‘DetecciónFallos’ del Script Sensor_Tiempo**

En la Figura 79, se puede observar el método encargado de restablecer el estado de varias variables importantes en el sistema de medición y control del cilindro. Llama al método ‘MostrarAlarmaVerde’ para activar la alarma visual correspondiente y, por último, accede a la función ‘writeCoil’ del script MODBUS para actualizar la coil a false, indicando así que no hay fallos.

En Unity, habrá un botón que estará configurado para que, cada vez que se pulse, se acceda a este método.

```
public void ReiniciarSistema()
{
    MostrarAlarmaVerde();
    origenActivado = false;
    falloDetectado = false;
    sumaTiempos = 0f;
    intentos = 0;
    modbusScript.writeCoil(CoilRespuesta_byte, CoilRespuesta_bit, false);
}
```

**Figura 79 Método ‘ReiniciarSistema’ del Script Sensor_Tiempo**

Finalmente, la Figura 80 muestra los métodos encargados de actualizar las alarmas visuales. Para lograr esto, utiliza el método SetActive(true) en el objeto

‘alarmaVerde’ o ‘alarmaRoja’, lo que hace que la alarma sea visible al usuario. Mientras que con el método SetActive(false), volvería estas alarmas invisibles.

En ‘MostrarAlarmaVerde’, se hace visible la alarma verde e invisible la alarma roja, actualizando también el texto de la variable ‘falloText’ para reflejar el estado correcto del sistema.

En ‘MostrarAlarmaRoja’, se hace visible la alarma roja e invisible la alarma verde, además, se detiene la ejecución de la coroutine ‘DetecciónFallos’ mediante la llamada a ‘StopCoroutine’, para así evitar que el proceso de detección de fallos continúe ejecutándose en segundo plano.

Por último, el método ‘InicializarAlarmas’, restablece la alarma roja y verde invisibles.

```
void MostrarAlarmaVerde()
{
    alarmaVerde.SetActive(true);
    alarmaRoja.SetActive(false);
    falloText.text = "Todo va correctamente";
}
1 referencia
void MostrarAlarmaRoja()
{
    alarmaVerde.SetActive(false);
    alarmaRoja.SetActive(true);
    StopCoroutine(DetecciónFallos());
}
1 referencia
void InicializarAlarmas()
{
    alarmaVerde.SetActive(false);
    alarmaRoja.SetActive(false);
}
```

**Figura 80 Métodos ‘MostrarAlarmaVerde’, ‘MostrarAlarmaRoja’ y ‘InicializarAlarmas’ del Script Sensor_Tiempo**

Para visualizar la detección de fallos en Unity, se implementará un panel en la parte superior derecha de la interfaz. En este panel, se añadirán una alarma verde y una alarma roja, cuyos modelos se habrán descargado previamente, tal como se explica en el apartado Creación de activos. Adicionalmente, se incluirá un TextMeshPro que se utilizará para indicar si el sistema está funcionando correctamente o si se ha detectado algún fallo. Finalmente, se añadirá un botón de

reinicio, el cual se configurará de acuerdo con lo explicado en el apartado Elementos Propios del UI. Al hacer clic en este botón, se invocará el método ‘ReiniciarSistema’.

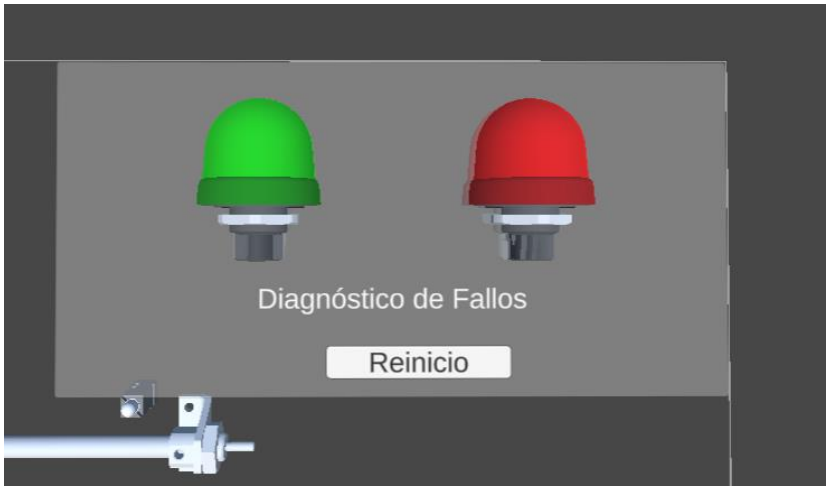



**Figura 81 Panel del Diagnóstico de fallos**

Para concluir este apartado, se proporcionará un código QR que dirigirá a una serie de vídeos. Uno de estos vídeos presentará la integración y uso del panel de diagnóstico de fallos, mientras que el otro ofrecerá una visualización del proyecto en acción. Este último mostrará el entorno de desarrollo en Unity, el PLC y el comportamiento del cilindro real.

**Figura 82 QR - Gemelo Digital**

# 7. Conclusión

Después de abordar los diversos aspectos de este proyecto, se puede realizar una síntesis de los objetivos alcanzados.

Unity 3D ha demostrado ser una herramienta poderosa, efectiva y viable para la creación de Gemelos Digitales en el ámbito de la automatización industrial. Su potencia gráfica es un punto destacado en la recreación precisa de una planta real, permitiendo tanto la creación desde cero de cualquier modelo como la importación de modelos ya existentes, lo que aligera el proceso.

La elección del protocolo de comunicación MODBUS como medio de comunicación ha sido exitosa, ya que ofrece librerías suficientes para su integración con Unity y con el PLC.

Este proyecto se ha desarrollado a una pequeña escala con el objetivo de reducir la dificultad inicial en el manejo de Unity 3D, sirviendo como base para la investigación y enseñanza de esta plataforma. Sin embargo, se recomienda encarecidamente el uso de Unity 3D para el desarrollo de proyectos similares a una escala mayor, ya que puede ser de gran ayuda a nivel industrial.

La capacidad de Unity 3D para integrarse con sistemas de automatización industrial y su flexibilidad en el diseño de modelos lo posicionan como una herramienta valiosa para futuras aplicaciones en este campo.

# 8. Bibliografía

[1] «Doom (videojuego de 1993),» 10 Marzo 2024. [En línea]. Available: https://es.wikipedia.org/wiki/Doom_(videojuego_de_1993).

[2] G. B. M. S. G. L. R. S. M. D. González Muñoz C., «TecsMedia: Análisis Motores gráficos y su aplicación en la industria,» 2015. [En línea]. Available: https://www.aragon.es/documents/20127/674325/Estado%20del%20arte%20GameEngines%20y%20su%20impacto%20en%20la%20industria.pdf/db827568-09ef-e931-01e8-d293b9fca834.

[3] J. Gómez, «Contribuciones al desarrollo de plataforma estandarizada para la simulación de sistemas de automatización,» 2020. [En línea]. Available: https://idus.us.es/bitstream/handle/11441/102849/TFG-3002-GOMEZ%20JIMENEZ.pdf?sequence=1&isAllowed=y.

[4] Unity Documentation, «Manual de Unity,» 2016. [En línea]. Available: https://docs.unity3d.com/es/530/Manual/InstallingUnity.html.

[5] Unity Documentation, «Manual de Unity,» 2016. [En línea]. Available: https://docs.unity3d.com/es/530/Manual/LearningtheInterface.html.

[6] Aula 21, «Modbus: Qué es y cómo funciona,» [En línea]. Available: https://www.cursosaula21.com/modbus-que-es-y-como-funciona/.

[7] Wikipedia, La enciclopedia libre, «Modbus,» 14 Enero 2024. [En línea]. Available: https://es.wikipedia.org/wiki/Modbus.

# 9. Apéndice A: Modbus

En este punto se abordará lo relacionado con Modbus. Se comenzará con una introducción que explicará los conceptos básicos, los tipos y cómo funcionan. Luego, se analizará Modbus dentro de Unity, donde se explicarán los métodos de lectura y escritura, así como lo necesario para establecer la conexión Modbus-Unity y el simulador de PLC utilizado.

## 9.1. Introducción. Conceptos Básicos

Modbus es un protocolo de comunicación abierto, utilizado para transmitir información a través de redes en serie entre dispositivos electrónicos, basado en la arquitectura maestro/esclavo (RTU) o cliente/servidor (TCP/IP). Convertido en un protocolo de comunicaciones estándar de facto en la industria, es el que goza de mayor disponibilidad para la conexión de dispositivos electrónicos industriales.

Las principales razones por las cuales el uso de Modbus en el entorno industrial se ha impuesto por encima de otros protocolos de comunicaciones son:

- Se diseñó teniendo en cuenta su uso para aplicaciones industriales
- Es público y gratuito
- Es fácil de implementar y requiere poco desarrollo
- Maneja bloques de datos sin suponer restricciones

Este sistema de comunicación se usa generalmente para transmitir señales de los dispositivos de instrumentación y control a un controlador principal o a un sistema de recolección de datos (SCADA). Permite el control de una red de dispositivos, por ejemplo, un sistema de medida de temperatura y humedad, y comunicar los resultados a un ordenador.

Ahora se procederá a explorar algunas de las diferentes versiones del protocolo Modbus.

### 9.1.1. Versiones del Protocolo

Hay muchas variantes de protocolos Modbus, existen versiones del protocolo Modbus para el puerto serie, para Ethernet, y otros protocolos que soportan el

conjunto de protocolos TCP/IP de Internet. A continuación, se mostrarán las tres versiones más comunes [6] [7]:

- **Modbus RTU**

El protocolo Modbus RTU es un medio de comunicación que permite el intercambio de datos entre controladores lógicos programables (PLC) y ordenadores (PC) mediante conexiones en serie. Esta versión se destaca por su codificación binaria y su robusta verificación de errores CRC, lo que la convierte en la implementación más común en aplicaciones industriales y de producción automatizada. Utiliza una transmisión de datos serial simple a través de tecnología UART, con velocidades de baudios que van desde 1200 a 115200 bits por segundo, aunque la mayoría de los dispositivos admiten hasta 38400 bits por segundo.

En una red Modbus RTU, existe un maestro y uno o más esclavos, cada uno con una dirección de dispositivo de 8 bits. Los mensajes del maestro incluyen la dirección del esclavo al que se dirigen. Los esclavos solo responden si reconocen su dirección y dentro de un tiempo determinado; de lo contrario, se considera un error de “no respuesta”. Este protocolo es preferido en entornos donde se utilizan dispositivos Modbus similares, se comprende bien el protocolo Modbus y se busca un estándar de instalación confiable y eficiente.

- **Modbus TCP**

Modbus TCP fue desarrollado para capitalizar las infraestructuras LAN existentes y ampliar la capacidad de conexión en redes. A través de la encapsulación de los bloques de datos de solicitud y respuesta del Modbus RTU en un bloque TCP, este sistema se transmite a través de Ethernet estándar.

A diferencia de Modbus RTU, la dirección IP adquiere mayor relevancia en Modbus TCP, aunque el número de unidades sigue siendo importante y su interpretación varía según la aplicación. El puerto predeterminado es 502, aunque puede ser reasignado según sea necesario.

Esta versión sigue el modelo de referencia de RED OSI, definiendo las capas de presentación y aplicación. Modbus TCP redefine los roles de maestro y esclavo,

donde los esclavos actúan como servidores y los maestros como clientes, permitiendo múltiples clientes para un servidor.

Modbus TCP es comúnmente utilizado en PLC, sistemas SCADA y dispositivos como sensores y actuadores, proporcionando una sólida base para la automatización industrial y el control de procesos.

- **Modbus Plus**

Modbus Plus es un protocolo de red diseñador para proporcionar comunicación de alta velocidad entre pares, utilizando un esquema de token bus, lo que significa que los dispositivos tienen acceso al medio de comunicación a través de un token que pasa de un dispositivo a otro. A diferencia del Modbus estándar, Modbus Plus es un sistema completo que incluye un medio de comunicación predefinido y la implementación de la capa física OSI, lo que lo convierte en una solución integral para aplicaciones industriales.

Este sistema LAN está específicamente diseñado para su uso en entornos de control industrial, permitiendo a los dispositivos en red intercambiar mensajes para el control y la supervisión de procesos en diferentes ubicaciones de una planta industrial. Emplea un mecanismo de control de acceso con paso controlado, lo que garantiza un funcionamiento determinista, aunque su velocidad puede variar según las condiciones.

La capa de enlace de datos de Modbus Plus se basa en el protocolo multipunto HDLC, que utiliza un mecanismo para el control de acceso con paso controlado por señales. Utiliza una implementación física RS485 y funciona sobre cables de par trenzado blindado, lo que permite la transmisión sincronizada de datos a una velocidad de hasta 1Mbps, ofreciendo un rendimiento mejorado en comparación con Modbus Serial.

## 9.2. Modbus en Unity

De entre las diversas versiones consideradas anteriormente, se ha optado por emplear la versión Modbus TCP para la comunicación en el proyecto.

Se empleó un driver de Modbus TCP como punto de partida para el desarrollo de la comunicación. A partir de este recurso, se llevaron a cabo una serie de modificaciones y adaptaciones para adecuarlo a los requisitos específicos de mi proyecto.

Para llevar a cabo su implementación en Unity, se debe iniciar descargando el recurso desde el enlace proporcionado: https://github.com/nodlag/umodbustcp. Tras la descarga, se obtendrá una carpeta comprimida que deberá ser descomprimida. Posteriormente, la carpeta descomprimida se arrastrará dentro de la sección “Assets” en la ventana de Proyecto de Unity.

A continuación, se presentarán tanto los diversos roles que se definen en Modbus TCP como una explicación detallada sobre la conexión, lectura de datos y escritura de datos. Además, se abordarán las funciones necesarias para su correcto funcionamiento.

### 9.2.1. Roles en Modbus

Los roles en Modbus TCP definen las funciones y responsabilidades de los dispositivos involucrados en la comunicación, estableciendo una estructura clara para la transmisión de datos y comandos. En este apartado, exploraremos los dos roles principales en la comunicación Modbus TCP: el servidor y el cliente.

El PLC, al encontrarse directamente conectado a la planta real y ser responsable de controlar sus operaciones, asume la función de servidor. Actúa como el dispositivo que suministra datos y acepta comandos a través del protocolo Modbus TCP, desempeñando un papel central en la gestión y supervisión de los procesos en la planta real.

Por otra parte, el ordenador con Unity, al interactuar con el PLC para obtener datos y enviar comandos, cumple la función de cliente. Utiliza el protocolo Modbus TCP para enviar solicitudes al PLC, como la obtención de datos de sensores o el envío de instrucciones de control. El ordenador con Unity puede ser visto como el punto de acceso desde el cual se monitorean y controlan las operaciones de la planta a través del PLC.

### 9.2.2. Conexión

La conexión entre Unity y el protocolo de comunicación Modbus TCP implica configurar una conexión TCP/IP entre Unity y el dispositivo designado como servidor Modbus. En esta configuración, se establecen los parámetros correspondientes, incluyendo la dirección IP del dispositivo Modbus y el puerto TCP utilizado para la comunicación. Un ejemplo de uso de la función para establecer conexión que se usará en el Script destinado a gestionar la comunicación Modbus, es la mostrada en la Figura 84.

Por una parte, tenemos 'm_oUModbusTCP', un objeto que se empleará repetidamente y que sirve para establecer la conexión con un dispositivo Modbus TCP. Este objeto contiene las funciones necesarias para administrar la comunicación con el dispositivo, como conectar, leer y escribir datos. En la parte A de la Figura 83 nos muestra cómo se declara la variable, mientras que en la parte B nos muestra cómo se inicializaría la variable para que funcione correctamente. Esta parte B irá en un método que se explicará más adelante en el apartado Conexión con Modbus.


**Figura 83 Declaración (a) e Inicialización (b) de la variable m_oUModbusTCP**

Por otro lado, 'sIp' representa la dirección IP del dispositivo y 'usPort' el puerto de comunicación, permitiendo así la interacción con el dispositivo a través de la red.

```
m_oUModbusTCP.Connect(sIp, usPort);
```

**Figura 84 Función Connect**

### 9.2.3. Lectura de Datos

Una vez establecida la conexión, Unity puede enviar solicitudes de lectura al dispositivo Modbus, detallando los registros de datos específicos que se desean obtener. Estas solicitudes incluyen información vital, como la identificación del

dispositivo y los registros de datos específicos que se quieren acceder. Tras recibir la solicitud, el dispositivo Modbus la procesa y envía una respuesta a Unity. Esta respuesta contiene los datos solicitados, que pueden incluir información detallada, como valores de sensores o estados de dispositivos.

La lectura del valor de las salidas se realiza mediante la función ‘ReadCoils’. Esta función crea un mensaje ModbusTCP de lectura que solicita al dispositivo PLC los valores de un conjunto específico de bobinas. Una vez que se recibe la respuesta del PLC, los valores de las bobinas se almacenan en un array de bytes para su posterior procesamiento en la aplicación Unity3D.

En la Figura 85 se presenta un ejemplo de aplicación de la función 'ReadCoils'. Esta función requiere varios parámetros para operar de manera adecuada. En primer lugar, se utiliza un identificador interno para gestionar la comunicación con el PLC, el cual se establece como '1' en el ejemplo de la figura. El segundo parámetro indica la dirección del dispositivo esclavo al que se enviará la solicitud, también configurado como '1' en este caso. A continuación, se especifica la dirección de memoria del PLC desde donde se iniciará la lectura de las bobinas, indicada por el parámetro 'iniCoils'. El cuarto parámetro, denominado 'numCoils', define la cantidad de bobinas que se leerán a partir de la dirección especificada en 'iniCoils'. Por último, el parámetro 'ref ValoresCoils' corresponde a un array de bytes que almacenará el resultado de la función una vez completada la operación de lectura de las bobinas. Este parámetro se pasa por referencia para permitir que la función modifique directamente los datos del array.

```
m_oUModbusTCP.ReadCoils(1, 1, iniCoils, numCoils, ref ValoresCoils);
```

**Figura 85 Ejemplo de uso de la Función 'ReadCoils'**

La lectura de las entradas del PLC se realiza mediante la función ‘ReadDiscreteInputs’ mostrada en la Figura 86. Al igual que con los valores de las salidas, se crea un mensaje ModbusTCP de lectura para solicitar al dispositivo PLC los valores de un conjunto específico de entradas. Los parámetros empleados son los mismos a los de la función ‘ReadCoils’, variando únicamente en el nombre asignado a cada uno de ellos.

```
m_oUModbusTCP.ReadDiscreteInputs(1, 1, iniInput, numInput, ref ValoresInputs);
```

**Figura 86 Ejemplo de uso de la Función 'ReadDiscreteInputs'**

### 9.2.4. Escritura de Datos

Además de la lectura, Unity tiene la capacidad de enviar solicitudes de escritura al dispositivo Modbus, permitiendo la modificación de registros de datos específicos. En estas solicitudes, Unity especifica los registros a modificar y los nuevos valores que se desean asignar. Una vez reciba la solicitud de escritura, el dispositivo Modbus la procesa y envía una confirmación a Unity, indicando que la operación se ha llevado a cabo con éxito.

Esto se logra mediante la creación de un mensaje ModbusTCP de escritura que contiene la información necesaria para identificar el dispositivo PLC, la dirección de inicio del registro de salida y el valor del estado que se desea escribir.

En la Figura 87 se presenta un ejemplo de aplicación de la función 'WriteSingleCoils'. Esta función requiere varios parámetros para operar de manera adecuada. En primer lugar, se utiliza un identificador interno para gestionar la comunicación con el PLC, el cual se establece como '1' en el ejemplo de la figura. El segundo parámetro indica la dirección del dispositivo esclavo al que se enviará la solicitud, también configurado como '1' en este caso. El tercer parámetro, denominado ‘coilAddress’, es la dirección del registro en el PLC donde se escribirán los valores. La dirección se calcula en base a la dirección inicial y a la posición del bit específico dentro del registro. A continuación, se define el estado de la coil que se quiere escribir en el PLC, indicada por el parámetro ‘estado’. El último parámetro, indicado con ‘ref bResult’, es un array de bytes que se utiliza para almacenar el resultado de la operación de escritura. Se pasa por referencia para que la función pueda modificar directamente los datos del array.

```
m_oUModbusTCP.WriteSingleCoils(1, 1, coilAddress, estado, ref bResult);
```

**Figura 87 Ejemplo de uso de la Función 'WriteSingleCoils'**

## 9.3. Simulador PLC

EasyModbusServerSimulator es una aplicación de simulación de servidor Modbus que proporciona un entorno de desarrollo y pruebas para sistemas de automatización industrial que utilizan el protocolo Modbus. Esta aplicación permite emular un servidor ModbusTCP o ModbusRTU, lo que permite a los desarrolladores probar y depurar sus aplicaciones Modbus sin necesidad de tener acceso a un PLC o dispositivo de campo real.

Una vez que se dispone del PLC real, simplemente sería necesario cambiar la Ip del ordenador donde se está utilizando el simulador, por la Ip del PLC.

Enlace de descarga: https://sourceforge.net/projects/easymodbustcpserver/

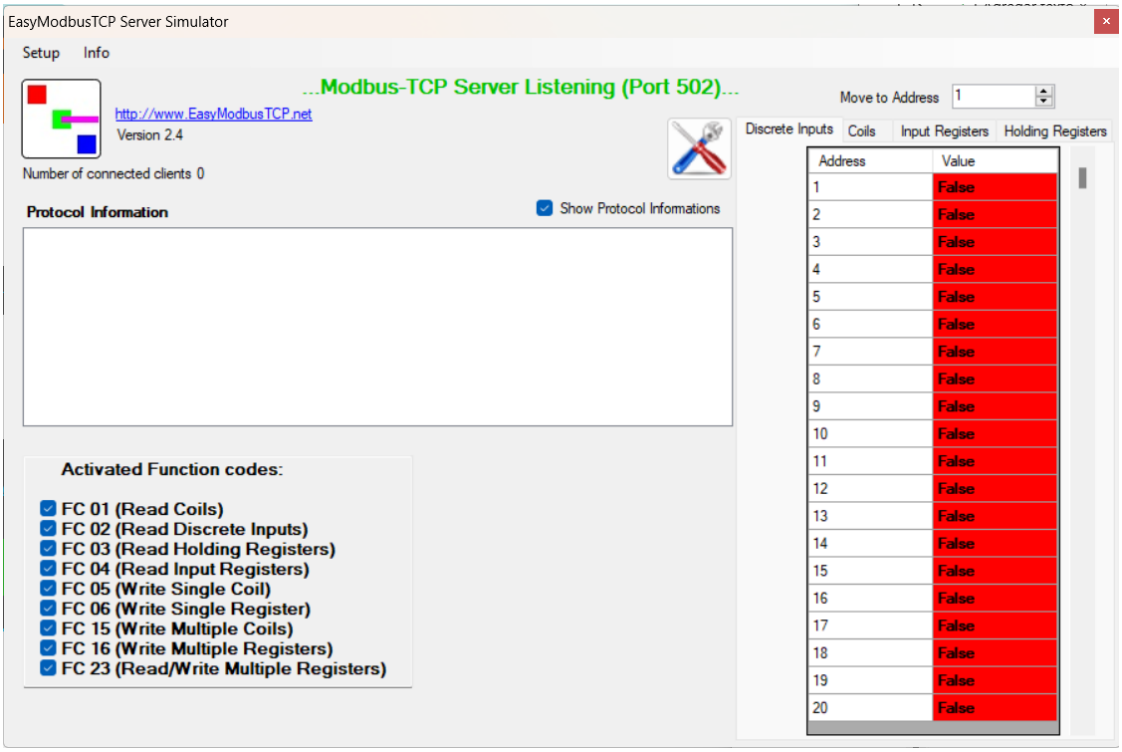



**Figura 88 Aplicación EasyModbusServerSimulator**

Ahora se llevará a cabo un análisis detallado de la Figura 88, explicando las partes de la aplicación y su composición:

- En primer lugar, se dirigirá la atención a la pestaña de Setup, ubicada en la parte superior izquierda, donde se encuentra la ventana de propiedades, ilustrada en la Figura 89. Aquí, se tienen opciones para

modificar las propiedades del Modbus RTU, aunque no serán relevantes. En la sección ModbusProperties, se podrá seleccionar la versión específica de Modbus con la cual se trabajará, siendo en este caso Modbus TCP, y elegir el puerto, que actúa como el canal de comunicación para la transferencia de datos. Se mantendrá el puerto predeterminado 502.

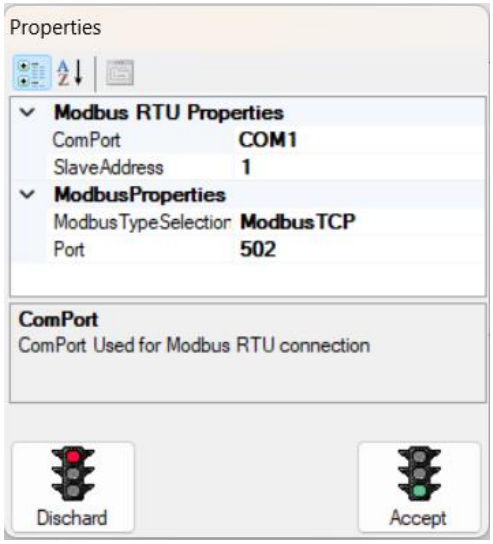



**Figura 89 Propiedades del Simulador**

- En la sección central izquierda de la aplicación se observa un recuadro blanco, el cual constituye la información del protocolo. Una vez se haya establecido la conexión, este recuadro mostrará detalles sobre las funciones en curso, indicando los valores y momentos específicos de su ejecución, tal y como se muestra en la Figura 90.

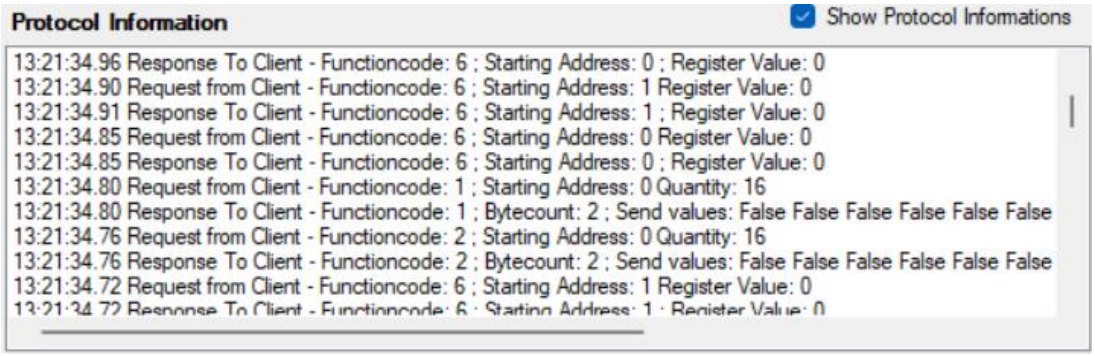



**Figura 90 Información del protocolo Simulador**

- En la sección de la derecha se encuentra un recuadro vertical compuesto por cuatro pestañas distintas, cada una destinada a un tipo específico de datos y funcionalidades.

  Cada pestaña contiene una tabla de dos columnas, donde la primera columna indica la dirección correspondiente y la segunda columna muestra el valor asociado a esa dirección.

  En este entorno, las direcciones están estructuradas por bytes y bits. Cada dirección corresponde a un bit específico, y se organiza en forma de bytes, donde un byte está compuesto por 8 bits. La numeración de los bits dentro de un byte comienza desde el 0 hasta el 7. Por tanto, en una dirección dada, el bit 0 corresponderá al primer bit del byte, el bit 1 al segundo, y así sucesivamente hasta el bit 7.

  Las pestañas de la Figura 91 se distribuyen de la siguiente manera:

  - **Discrete Inputs (Entradas Discretas):** Esta pestaña aloja una tabla destinada a valores booleanos que representan los estados de las entradas discretas del PLC. Algunos ejemplos serían cosas como interruptores de luz, botones e interruptores de proximidad.

  - **Coils (Bobinas):** Aquí se presenta otra tabla que también maneja valores booleanos. Estos valores están asociados a las bobinas del PLC y reflejan su estado de activación y desactivación.

  - **Input Registers (Registros de Entrada):** La tercera pestaña está reservada para valores numéricos que solo pueden ser leídos. Los registros de entrada almacenan datos numéricos provenientes de dispositivos externos o sensores conectados al PLC.

  - **Holding Registers (Registros de Retención):** La última pestaña también trata con valores numéricos, pero estos pueden ser leídos y modificados, y está dedicada a los registros de retención.

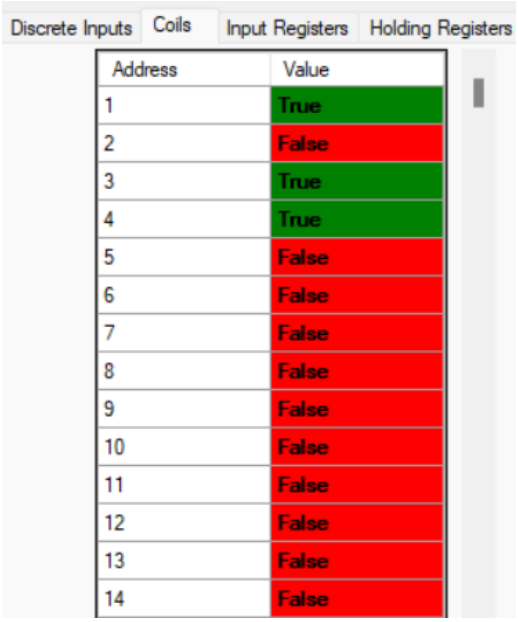



**Figura 91 Pestaña Coils del Simulador**