UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Desarrollo de un entorno en Unity para la prueba y verificación de sensores

Trabajo Fin de Máster

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Arias Ronquillo, Christian

Tutor/a: Gracia Calandin, Luis Ignacio

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DESARROLLO DE UN ENTORNO EN UNITY PARA LA PRUEBA Y VERIFICACIÓN DE SENSORES

MÁSTER EN AUTOMÁTICA E INFORMÁTICA INDUSTRIAL

Autor: Christian Iván Arias Ronquillo

Tutor: Luis Ignacio Gracia Calandín

Septiembre, 2023

RESUMEN

Para el presente trabajo se desarrolló un entorno en Unity que interactúa con 10 sensores físicos que están conectados a una placa Arduino MEGA, la comunicación entre las plataformas se realiza mediante comunicación serial. El espacio virtual de Unity está dividido por zonas y en cada una de ellas se han colocado diferentes elementos que responden a los estímulos detectados por los sensores. Se tiene dos brazos robóticos cuyas articulaciones son controladas mediante un joystick y un encoder. Un robot móvil que esquiva obstáculos que son generados por un sensor de presencia. Una estación de mecanizado que elige entre dos piezas una de ellas se activa mediante un sensor de efecto hall que responde al detectar un campo magnético. Una estación de transporte de cristales que interactúan con un sensor de impacto que al activarse rompe uno de los cristales. Una serie de lámparas que se encienden con un sensor de micrófono al detectar cierto nivel de audio, una pantalla de registro del ritmo cardiaco y el nivel de conductividad de la piel que reciben información de un sensor de ritmo cardiaco y un GSR. Se colocaron aspersores de agua que se activan si el sensor de flama detecta una llama. Además, en la pantalla del usuario del sistema se lleva un registro de la temperatura. Por último, se tiene una pantalla de control general que refleja el valor de todos los sensores. Tanto el entorno de Unity como la lectura de los sensores se han programado con un enfoque de programación concurrente, para lo cual se utilizaron librerías como FreeRTOS de Arduino que permite separar el código por tareas, mientras que en Unity se utilizaron hilos en algunos scripts para la gestión de los datos.

Palabras Clave: Arduino, Unity, FreeRTOS, programación concurrente, sensores.

ABSTRACT

For this project, we've created a Unity environment that interacts with ten physical sensors connected to an Arduino MEGA board. These sensors communicate with Unity using serial communication. Within Unity, we've divided the virtual space into different zones, each containing various elements that react to signals from the sensors. We've incorporated two robotic arms whose movements are controlled using a joystick and an encoder. Additionally, there's a mobile robot that autonomously navigates and avoids obstacles, thanks to a presence sensor. A machining station has the capability to select between two pieces, activated by a Hall effect sensor that detects a magnetic field. We've integrated lamps that respond to sound levels detected by a microphone sensor and a display that records heart rate and skin conductivity data from corresponding sensors. There's also a crystal transport station that interacts with an impact sensor, breaking one of the crystals upon activation. As an added safety feature, we've included water sprayers that turn on if the flame sensor detects a fire. Furthermore, we've implemented a general system screen that logs temperature data and a control screen that provides real-time information from all the sensors. To efficiently manage all these tasks, we've adopted a concurrent programming approach. For Arduino, we've utilized libraries like FreeRTOS to organize the code into separate tasks. In Unity, we've employed threads in certain scripts to handle data management.

Key words: Arduino, Unity, FreeRTOS, concurrent programming, sensors

ÍNDICE

Índice de figuras

Índice de Tablas

1. Introducción

1.1. Motivación del TFM

Los avances tecnológicos han permitido recrear entornos que simulan la realidad, donde se pueden desarrollar pruebas de funcionamiento y operación de sistemas que benefician a los proyectos industriales, reduciendo los tiempos y el impacto económico de dichas implementaciones. Estos espacios se han podido realizar gracias a los motores gráficos que pueden simular de una manera precisa ciertas características físicas del mundo real. Además, cada vez es más común encontrar diferentes dispositivos que permiten a los usuarios interactuar con estos entornos. Si examinamos el funcionamiento de estos sistemas encontraremos que están compuestos por sensores que envían datos a una placa electrónica de adquisición y tratamiento para posteriormente ser digitalizados en el entorno.

Analizando los motores gráficos podemos encontrar a Unity que es un software de desarrollo de espacios virtuales que se ha adaptado para crear soluciones de visualización en 3D, al ser una herramienta versátil su uso está en aumento. Si nos centramos en la parte industrial o de negocios algunas de sus aplicaciones son:

- **Simulación y entrenamiento**: Con esta herramienta las empresas pueden crear entornos virtuales que simulan diversas situaciones laborales que permiten a los empleados practicar antes de exponerse a tareas reales.
- **Diseño y visualización**: Creación de modelos 3D y visualizaciones de productos y prototipos. Esto es útil para probar los conceptos y los diseños preliminares de productos antes de pasar a la producción, lo que

conlleva disminuir los tiempos de reingeniería y amenorar los costos de prototipado.

- **Realidad aumentada y realidad virtual**: Desarrollo de experiencias inmersivas donde el usuario no solo interactúa con el entorno mediante una pantalla, sino que es parte del espacio generando estímulos adicionales que pueden mejorar los procesos de aprendizaje y capacitación. También la implementación de los llamados "Gemelos digitales" que son elementos virtualizados que tienen todas las características físicas de un dispositivo en el mundo real y donde es posible implementar herramientas de mayor complejidad como sistemas de control.
- **Publicidad y marketing**: Creación de anuncios y experiencias publicitarias interactivas. Las empresas pueden crear experiencias en línea que permitan a los usuarios interactuar con los productos antes de comprarlos.
- **Automatización y robótica**: La robótica y la automatización es otro de los campos donde Unity tendría un gran impacto, las empresas pueden crear programas personalizados que permiten a los robots interactuar con su entorno y realizar tareas específicas de manera autónoma.

El creciente uso de Unity en múltiples entornos demuestra la importancia de comprender su funcionamiento con diferentes herramientas y dispositivos, en específico para la adquisición de datos con sensores, porque estos permiten capturar comportamientos del mundo físico que resultan necesarios para la simulación de los entornos virtuales.

Ahora, analizando el sistema electrónico que se necesita para adquirir y procesar los datos de los sensores, encontramos que se puede utilizar la placa Arduino, se trata de un dispositivo electrónico pensado para el prototipado, es de código abierto y permite la creación de soluciones personalizadas mediante su programación. Este dispositivo, además de ser uno de los más utilizados para proyectos electrónicos alrededor del mundo, cuenta con una comunidad significativa de programadores y diseñadores que aportan con documentación sobre su uso. Muchas empresas también fabrican módulos de sensores y actuadores que son compatibles con Arduino y que simplifican el diseño electrónico de los sistemas.

Adicionalmente, vamos a mencionar la importancia de los datos y su adquisición. Los sensores son dispositivos esenciales en los entornos que permiten la recopilación de datos que, si nos centramos en la industria, proporcionan información sobre la maquinaria, el medio ambiente, el rendimiento del proceso y otros aspectos clave.

Con el desarrollo del proyecto se probarán los siguientes sensores:

Sensores GSR: Sensor de conductividad eléctrica de la piel. Este sensor puede dar una medida de las respuestas del Sistema Nervioso Simpático humano, que está directamente involucrado en la regulación del comportamiento emocional en los humanos.

ky-039 sensor de latido del corazón: Es un sensor óptico de pulsos cardiacos que se coloca en el dedo y puede medir el ritmo cardiaco en tiempo real. Se utiliza en dispositivos de control y estabilidad de pacientes con afecciones cardiacas o para el control general del estado de salud de un paciente.

KY-031 sensor de impacto: Es un sensor de tipo digital que puede cambiar su estado lógico si este o una superficie sujeta a este son golpeados mediante un contacto físico. Su uso se centra en detectar situaciones de colisión o impacto.

KY-032 sensor de obstáculos: Sensor analógico óptico emisor receptor. Utilizado principalmente en la robótica móvil en sistemas de seguidores de línea o detección de obstáculos.

KY-024 sensor de campo magnético: Se activa cuando detecta un campo magnético proveniente de un imán natural o de algún material ferromagnético, sus principales partes son un potenciómetro para ajustar la sensibilidad y un sensor de Efecto Hall Lineal. Se puede utilizar para el realizar el cálculo de la velocidad de un mecanismo en rotación o como un interruptor de proximidad.

KY-038 sensor micrófono: Permite detectar cualquier tipo de sonido a través del micrófono de condensador. El sensor micrófono es útil para detectar ruidos o sonidos del entorno dependiendo su configuración.

KY-023 sensor joystick: Este elemento te permite controlar y manejar determinados aparatos electrónicos. Normalmente se utilizan para proyectos robóticos en el cual se necesitan para la movilidad analógica de las articulaciones de un brazo robótico. El Módulo Joystick, es más utilizado para proyectos de robótica y control de dispositivos RF (Radio Frecuencia).

KY-026 sensor de flama: Por medio de un LED receptor infrarrojo detecta longitudes de onda de llama en un rango de 760nm a 1100nm. Este sensor es útil para sistema de detección de incendios, como una medida de seguridad.

KY-015 sensor de temperatura y humedad: Es un sensor de temperatura y humedad de salida de señal digital, tiene un tamaño ultra compacto, es de bajo consumo de energía y tiene gran utilidad cuando se requiere detectar dos magnitudes al mismo tiempo. Los sensores de temperatura y humedad son los

más extendidos en el mercado y su uso va desde sistemas industriales hasta juguetes.

KY-040 encoder rotativo: Es un codificador incremental con dos salidas desfasadas que nos indicará la dirección en la que se está girando el eje. Es ampliamente utilizado en el control de motores paso a paso, servomotores, control de potenciómetros digitales, controles de máquinas industriales tales como los husillos de tornos y fresadoras CNC, brazos robóticos, controles de instrumentos electrónicos (diales).

La combinación de Unity, Arduino y sensores pueden proporcionar soluciones de visualización y control personalizadas y altamente efectivas para los entornos industriales, lo que puede resultar en mayor eficiencia, menor tiempo de inactividad y mejor calidad de los productos.

1.2. Objetivos del proyecto

1.2.1. Objetivo General

Desarrollar un entorno virtual en Unity que permita visualizar e interactuar con 10 sensores físicos.

1.2.2. Objetivos específicos

- Estudiar el desarrollo de entornos en Unity.
- Calibrar y probar el funcionamiento correcto de los sensores.
- Realizar la comunicación entre los sensores, los dispositivos de adquisición de datos y la interfaz de Unity.
- Programar e implementar un entorno virtual interactivo en Unity.
- Realizar las pruebas de funcionamiento del sistema implementado.

1.3. Alcance del proyecto

Para el desarrollo del presente trabajo, se empezará por la prueba y calibración de 10 sensores y su adquisición mediante la placa electrónica Arduino, se obtendrán los datos de los sensores, se observarán los tiempos de respuesta y muestreo, además se analizará el ruido para filtrar las señales correspondientes. Con los sensores probados, calibrados y en funcionamiento se realizará la conexión de Arduino con el ordenador mediante comunicación serial, y los datos adquiridos serán enviados a Unity para su visualización.

En la plataforma de Unity se programará un entorno virtual, donde el usuario pueda visualizar e interactuar con la información proporcionada por los sensores. Se va a implementar al menos una acción para el uso de los datos dentro de las pantallas de Unity, además se implementará una pantalla general dentro del entorno para la visualización en forma de histogramas de los datos.
Para la validación del sistema se llevará un registro del proceso de prueba para la calibración de los sensores y su posterior uso dentro de la plataforma Unity. De igual manera se probarán las actividades desarrolladas para la interacción del usuario con cada una de sus actividades y su correcto funcionamiento. Por último, se analizarán los datos obtenidos mediante los histogramas dentro de Unity.

1.4. Metodología

La metodología del modelo en V se usará como base para la implementación del proyecto con las modificaciones correspondientes al alcance y objetivos de este. El modelo en V se muestra en la siguiente figura:

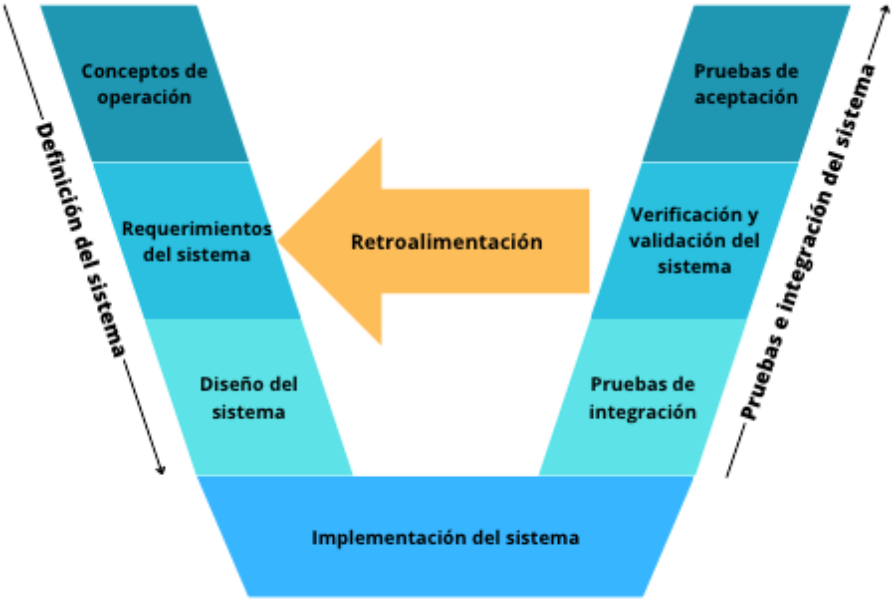

Figura 1. Metodología del TFM estructurada en modelo en V.

El modelo en V es una variación del modelo en cascada en donde se han añadido los principios de validación y verificación, de esta manera se tiene una relación directa entre las etapas de desarrollo y las etapas de pruebas. Tiene un amplio uso en proyectos de ingeniería y de la industria.

Los pasos principales de la metodología del modelo en V:

Conceptos de operación: En esta primera etapa, se define a grandes rasgos lo que debe hacer el sistema y como lo va a realizar.

Requerimientos del sistema: En esta etapa, se identifica y documenta lo que necesita el sistema para funcionar correctamente, esto se definen en colaboración con los clientes y las partes interesadas.

Diseño del sistema: En este apartado, se define la arquitectura del sistema, los módulos principales y las interfaces entre ellos, se lo realiza con base en los requisitos establecidos anteriormente.

Implementación: Luego del diseño se procede con la puesta en marcha del sistema, se ejecuta el código fuente del software y las interfaces físicas. Cada módulo se pone en funcionamiento de manera individual y se llevan a cabo pruebas unitarias.

Integración: En esta sección, los módulos individuales desarrollados anteriormente se combinan y se integran para formar el sistema completo. Se llevan a cabo pruebas de integración para garantizar que los módulos funcionen correctamente juntos y cumplan con los requisitos establecidos.

Verificación y validación: En esta etapa, se comprueba que el sistema cumple con todos los requisitos establecidos, además los usuarios y las partes interesadas verifican el sistema para asegurarse de que cumple con sus expectativas.

Pruebas de aceptación: Después de que la verificación y validación fueron exitosas, el sistema se despliega en el entorno de producción o pruebas finales. Se lleva a cabo la retroalimentación continua para corregir errores, realizar actualizaciones y agregar nuevas funcionalidades según sea necesario.

2. Estado del arte

2.1. Sensores

Estos dispositivos permiten identificar cambios físicos o químicos en el entorno, su función principal es convertir una magnitud en una señal eléctrica o digital, para que otros sistemas puedan interpretarla y utilizarla. Dependiendo de cómo estén construidos, los sensores tienen la capacidad de detectar y medir distintas magnitudes (Acedo Sánchez, 2006).

Estos dispositivos se componen de dos partes esenciales: el elemento sensor y el circuito de procesamiento. El primero se encarga de captar la magnitud y generar una señal eléctrica proporcional a la misma; por otro lado, el circuito de procesamiento amplifica, filtra y prepara la señal generada por el elemento sensor para que pueda ser interpretada y aprovechada por otros dispositivos o sistemas.

Los sensores son esenciales en la automatización de procesos y se los utiliza en diferentes industrias, tanto manufactureras como de otra índole, que incluyen áreas como la robótica, la conservación de energía y el control ambiental. Además, están presentes en dispositivos relacionados con la automoción, los electrodomésticos, la agricultura y la medicina. Se utilizan para recopilar datos, información que permite tomar decisiones y mejorar la eficiencia y seguridad de diversos sistemas y dispositivos (Pallás Areny, 2005).

2.1.1. Tipos de sensores

Existen varias formas de clasificar los sensores en base a sus características, pero únicamente analizaremos dos de ellas. La primera que los clasifica de acuerdo con el tipo de salida que tiene el sensor donde pueden ser analógicos o digitales y la segunda forma de clasificar los sensores es de acuerdo con el parámetro variable, que podría ser resistencia, capacidad, inductancia, tensión, carga o corriente (Lab-Volt, 2001).

En la primera clasificación podemos encontrar:
Sensor digital: Puede adoptar únicamente dos valores de salida; 1 (encendido) o 0 (apagado), los estados de un sensor digital son absolutos y únicos, y se usan para verificar estados de verdad o negación en un sistema automatizado, por ejemplo, una botella que debe detenerse para ser llenada puede cambiar la señal de un sensor digital de 0 a 1 para marcar su llegada al proceso de llenado.

Sensor analógico: Es un dispositivo que tiene como objetivo emitir señales en proporción al elemento o cantidad que se esté midiendo. Para que se realice este procedimiento es necesario que el instrumento reciba la alimentación desde un equipo de control eléctrico donde exista corriente continua. Algunos ejemplos son los sensores de temperatura, proximidad y presión (Pallás Areny, 2005).

En la siguiente tabla se muestra la segunda clasificación de sensores, para las magnitudes más frecuentes:

Tabla 1. Sensores y métodos de detección ordinarios para magnitudes más frecuentes. (Pallás Areny, 2005)

Sensores	Magnitudes								
	Posición Distancia Desplazamiento	**Velocidad**	**Aceleración Vibración**	**Temperatura**	**Presión**	**Caudal Flujo**	**Nivel**	**Fuerza**	**Humedad**
Resistivos	Potenciómetro Galgas Magnetoresistencia		Galgas + masa resorte	RTD Termistores	Potenciómetro + tubo de Bourdon	Anemómetro de hilo caliente Galgas + voladizo Termistores	Potenciómetro + flotador Termistores LDR	Galgas	Humistor
Capacitivos	Condensador diferencial				Condensador variable + diafragma		Condensador variable	Galgas capacitivas	
Inductivos y electromagnéticos	LVDT Corrientes de Foucault Resolver Inductosyn Efecto Hall	Ley de Faraday LVT Efecto Hall Corrientes de Foucault	LVDT + masa resorte		LVDT+diafragma Reluctancia variable + diafragma	LVDT + rotámetro Ley de Faraday	LVDT+flotador Corrientes de Foucault	Magneto elástico LVDT+célula carga	Dieléctrico variable
Generadores			Piezoeléctricos + masa resorte	Termopares Piroeléctricos	Piezoeléctricos			Piezoeléctricos	
Digitales	Codificación incremental y absoluta	Codificación incremental		Osciladores de cuarzo	Codificador+tubo Bourdon	Vórtices			SAW
Uniones p-n	Fotoeléctricos			Diodo Transistor Convertidores T/I			Fotoeléctricos		
Ultrasonidos	Reflexión	Efecto Doppler				Efecto Doppler Tiempo tránsito Vórtices	Reflexión Absorción		

2.1.2. Exactitud, fidelidad y sensibilidad

La exactitud, la fidelidad y la sensibilidad son conceptos clave para describir el rendimiento de los sensores. A continuación, se presenta una breve explicación de cada uno de ellos.

Exactitud: Se refiere a qué tan cerca está la medición realizada por el sensor del valor verdadero o establecido como referencia, puede expresarse como un porcentaje del error máximo permitido. Por ejemplo, si un sensor tiene una exactitud del 1%, significa que su medición puede tener un error máximo de $\pm$1% en comparación con el valor real.

Fidelidad: Está relacionada con la capacidad del sensor para proporcionar mediciones consistentes y repetibles en condiciones similares. Esto implica que al medir la misma magnitud varias veces con el mismo sensor, se espera que las mediciones sean cercanas entre sí. Para evaluar la fidelidad de un sensor, se pueden utilizar diferentes métodos, como realizar mediciones repetidas bajo

condiciones controladas y comparar las lecturas obtenidas. Se utilizan medidas estadísticas, como la desviación estándar o el coeficiente de variación, para cuantificar la variabilidad de las mediciones y determinar la consistencia del sensor.

Sensibilidad: Es la capacidad del sensor para detectar y responder a cambios en la magnitud que se está midiendo. Indica la relación entre la variación en la señal de salida del sensor y la variación correspondiente en la magnitud medida. Una mayor sensibilidad implica que el sensor puede detectar cambios más pequeños en la magnitud medida. Por ejemplo, un sensor de temperatura puede tener una sensibilidad de 0.1°C/mV, lo que significa que su señal de salida varía en 0.1 mV por cada °C de cambio en la temperatura.

Estas características pueden cambiar entre los diferentes modelos y fabricantes de sensores. Por lo tanto, es fundamental analizar estos aspectos antes de seleccionar un sensor para una aplicación específica (Pallás Areny, 2005).

2.1.3. Error de medición en los sensores

Existen diferentes tipos de errores que deben considerarse al evaluar la precisión de un sensor.

Error sistemático: También conocido como error determinístico o error de sesgo, es un error constante y predecible que se mantiene en todas las mediciones realizadas por el sensor. Puede originarse por desajustes en la calibración, errores en el diseño del sensor o interferencias externas. Este tipo de error puede corregirse mediante técnicas de calibración y ajuste.

Error aleatorio: También conocido como error de incertidumbre, es un error impredecible y fluctuante que varía de una medición a otra. Puede ser causado por fluctuaciones ambientales, ruido eléctrico, variaciones en las condiciones de medición, entre otros factores. Este tipo de error puede reducirse mediante técnicas de promediado, filtrado y estadísticas.

Error total: El error total de un sensor es la suma del error sistemático y el error aleatorio. Representa la máxima desviación posible entre la medición del sensor y el valor verdadero.

La especificación del error proporcionada por el fabricante del sensor es un parámetro clave para evaluar su rendimiento y determinar si cumple con los requisitos de precisión de la aplicación (Pallás Areny, 2005).

Para minimizar el error en los sensores, se deben seguir buenas prácticas de instalación, calibración y mantenimiento. Además, en algunos casos, es posible aplicar técnicas de compensación de error para ajustar y corregir las mediciones del sensor con base en información adicional disponible o modelos matemáticos. Esto puede mejorar la precisión y reducir el impacto de los errores sistemáticos (Acedo Sánchez, 2006).

2.1.4. Filtrado de la señal de un sensor

El proceso de filtrado de la señal de un sensor se refiere a aplicar técnicas o algoritmos para reducir o eliminar el ruido y las interferencias presentes en la señal captada. El objetivo principal es mejorar la calidad y confiabilidad de la señal, destacando la información relevante y eliminando componentes indeseables.

Es importante filtrar la señal del sensor debido a la posible contaminación por diferentes fuentes de ruido, como interferencias electromagnéticas, fluctuaciones ambientales o variaciones aleatorias. Estas interferencias pueden afectar la precisión y exactitud de las mediciones, dificultando la extracción de la información necesaria (Lab-Volt, 2001).

Existen diversas técnicas de filtrado que se pueden aplicar según las características particulares de la señal y las interferencias presentes. Algunas técnicas comunes incluyen filtros pasa bajos, filtros pasa altos, filtros pasa banda, filtros adaptativos y técnicas de promedio y suavizado.

La elección del tipo de filtro y su configuración depende de la aplicación y las características específicas de la señal e interferencias. A menudo se utilizan combinaciones de diferentes técnicas de filtrado para obtener los mejores resultados.

El filtrado de señales se realiza mediante dispositivos o algoritmos de procesamiento de señales digitales (DSP) o circuitos analógicos específicos, dependiendo de la naturaleza de la señal y los requisitos de la aplicación. El filtrado puede realizarse en tiempo real durante la adquisición de datos del sensor o aplicarse a datos previamente almacenados (Pallás Areny, 2005).

2.1.5. Calibración de un sensor

Es necesario ajustar y configurar el sensor para obtener mediciones precisas y confiables. Consiste en comparar las lecturas del sensor con un estándar de referencia conocido y realizar los ajustes necesarios para corregir cualquier

desviación o error en las mediciones. Existen variaciones en la respuesta y precisión de los sensores causadas por el envejecimiento, las condiciones ambientales, el desgaste y los posibles errores de fabricación.

La calibración se lleva a cabo en un entorno controlado utilizando equipos y estándares de calibración precisos y certificados. A continuación, se muestran los pasos generales involucrados. (Lab-Volt, 2001)

Selección del estándar de calibración: Se elige un estándar de referencia confiable y preciso que esté trazable a patrones nacionales o internacionales, y sea capaz de generar una señal o magnitud conocida y estable.

Preparación del equipo: Se verifica y prepara el equipo de calibración para asegurar su correcto funcionamiento y estabilidad, incluyendo conexiones, configuración de parámetros y estabilización de las condiciones ambientales.

Ajuste inicial: Se realiza un ajuste inicial utilizando los valores de calibración proporcionados por el fabricante como punto de partida, lo cual reduce la desviación inicial y asegura un estado adecuado para la calibración.

Comparación con el estándar: Se realizan mediciones simultáneas entre el sensor y el estándar de calibración en diferentes puntos de medición. Las lecturas del sensor se registran y se comparan con los valores conocidos del estándar de referencia.

Análisis y corrección: Se analizan las diferencias entre las lecturas del sensor y los valores del estándar de calibración. Se aplican correcciones o ajustes para minimizar los errores y desviaciones encontrados durante la comparación.

Documentación y certificación: Se registran los resultados de la calibración, incluyendo mediciones, correcciones y otros detalles relevantes. Se emite un certificado de calibración que muestra la trazabilidad, los valores corregidos y la fecha de la calibración.

Es importante destacar que la frecuencia de calibración puede variar según el tipo de sensor, su uso y las regulaciones o estándares aplicables en cada industria, algunos sensores pueden requerir calibración regularmente, mientras que otros pueden necesitarla ocasionalmente (Acedo Sánchez, 2006).

2.1.6. Proceso de adquisición y tratamiento de datos

El sensor es el componente encargado de detectar y convertir una magnitud física o química en una señal eléctrica. Existen diferentes tipos de sensores, como los de temperatura, presión, luz o movimiento, cada uno con su propio mecanismo para transformar la magnitud en una señal eléctrica. Por ejemplo, un sensor de temperatura puede utilizar un termistor para medir cambios en la resistencia eléctrica en función de la temperatura.

En algunos casos, se requiere un transductor entre el sensor y el circuito para adaptar la señal generada por el sensor a niveles o formatos específicos. Por ejemplo, si el sensor produce una señal analógica, un transductor puede convertirla en una señal digital o adaptar su rango de voltaje para que sea compatible con el circuito (Acedo Sánchez, 2006).

El circuito de acondicionamiento de señal recibe la señal del sensor y realiza diversas operaciones para prepararla para su procesamiento. Esto puede incluir amplificación, filtrado, adaptación de niveles de voltaje o conversión analógico-digital (ADC). El circuito de acondicionamiento de señal optimiza la señal, reduciendo el ruido, mejorando la sensibilidad y asegurando que sea adecuada para su procesamiento posterior.

La señal acondicionada se envía al microcontrolador o circuito de procesamiento, que contiene una unidad central de procesamiento (CPU), memoria y puertos de entrada/salida. El microcontrolador ejecuta un programa o algoritmo diseñado para procesar los datos del sensor. Puede realizar cálculos, tomar decisiones, enviar señales de salida y controlar otros dispositivos (Lab-Volt, 2001).

El programa o algoritmo que se ejecuta en el microcontrolador es esencial para interpretar y utilizar los datos del sensor. Puede ser desarrollado en lenguajes de programación como C o Arduino, y debe estar diseñado adecuadamente para cumplir con las tareas requeridas. Puede incluir cálculos, lógica de control, condiciones, bucles y otras instrucciones.

La salida del circuito de procesamiento puede variar según la aplicación. Puede ser una señal de control para otros dispositivos, una indicación visual, una acción mecánica u otra respuesta basada en los datos del sensor y el programa implementado. Por ejemplo, encender una luz o controlar un brazo robótico (Pallás Areny, 2005).

2.2. Arduino

Arduino es una plataforma de prototipado electrónico de código abierto que consta de hardware y software. Se utiliza para crear proyectos interactivos y sistemas autónomos. El corazón de Arduino es una placa de desarrollo con un microcontrolador, que es un chip programable que actúa como el cerebro del dispositivo.

La arquitectura más común en los microcontroladores Arduino es la arquitectura AVR, desarrollada por Atmel (ahora parte de Microchip Technology). Sin embargo, también existen placas Arduino basadas en otras arquitecturas, como ARM (Fitzgerald & Shiloh, 2012).

El microcontrolador en una placa Arduino que viene con un firmware preinstalado llamado "bootloader" que permite cargar el código del programa en el microcontrolador sin necesidad de un programador externo. Esto facilita el proceso de programación y hace que Arduino sea accesible para principiantes. Arduino utiliza un lenguaje de programación basado en C/C++, que se simplifica y se proporciona una biblioteca de funciones para facilitar la programación. Los usuarios pueden escribir su propio código en el entorno de desarrollo integrado (IDE) de Arduino, que es un software gratuito y multiplataforma.

El IDE de Arduino permite escribir y compilar el código, cargarlo en la placa Arduino a través de un cable USB y ejecutarlo en el microcontrolador. La placa Arduino puede interactuar con diferentes componentes electrónicos, como sensores, actuadores y módulos de comunicación, lo que permite una amplia gama de aplicaciones (Torrente Artero, 2013).

2.2.1. Arduino UNO

La placa Arduino Uno es un dispositivo de desarrollo popular y versátil que es adecuada para una amplia gama de proyectos de electrónica y programación. Ofrece capacidades básicas pero sólidas para la mayoría de las aplicaciones y es una excelente opción para principiantes y proyectos pequeños. Las características principales del Arduino Uno son las siguientes:

Microcontrolador: La placa Arduino Uno utiliza un microcontrolador ATmega328P basado en la arquitectura AVR de 8 bits, con una velocidad de reloj de 16 MHz.

Memoria: El ATmega328P tiene 32 KB de memoria flash, de los cuales 0.5 KB se utilizan para el bootloader. También cuenta con 2 KB de memoria SRAM y 1 KB de memoria EEPROM.

Pines de E/S Digitales: La Arduino Uno proporciona 14 pines de entrada/salida digitales, de los cuales 6 pueden utilizarse como salidas de modulación por ancho de pulso (PWM).

Pines de Entrada Analógica: La placa cuenta con 6 pines de entrada analógica de 10 bits que también pueden utilizarse como pines digitales.

Interfaces de Comunicación: Incluye una interfaz USB para la comunicación con el ordenador y la carga de código. Además, tiene un puerto serie UART (Universal Asynchronous Receiver-Transmitter) para la comunicación serial.

Alimentación: La Arduino Uno puede alimentarse a través de un cable USB conectado al ordenador o mediante una fuente de alimentación externa con una tensión recomendada de 7-12 V. También incluye un regulador de voltaje interno para generar tensiones de 5 V y 3.3 V para alimentar los componentes.

Memoria de Programa: Arduino Uno utiliza un bootloader que ocupa 0.5 KB de la memoria flash, dejando 31.5 KB disponibles para el programa del usuario.

Dimensiones: La placa Arduino Uno tiene dimensiones de aproximadamente 68.6 mm x 53.4 mm (Fitzgerald & Shiloh, 2012).

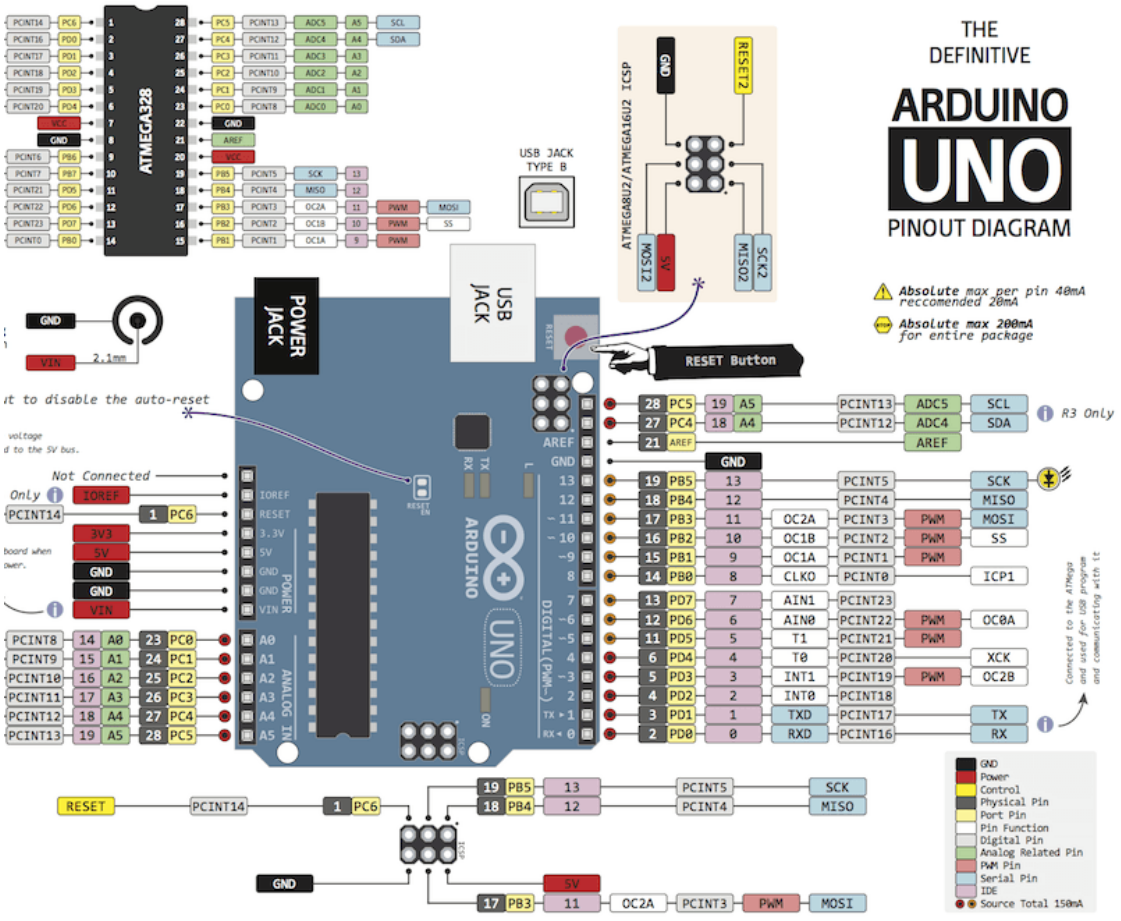

Figura 2. Detalle del hardware de la placa Arduino Uno

2.2.2. Arduino MEGA

La placa Arduino Mega es una versión avanzada y ampliada de Arduino Uno, especialmente diseñada para proyectos más complejos que necesitan mayor cantidad de pines de entrada/salida y mayor capacidad de procesamiento. A continuación, se presentan las características técnicas principales de la placa Arduino Mega.

Microcontrolador: La placa Arduino Mega utiliza un microcontrolador ATmega2560 basado en la arquitectura AVR de 8 bits, con una velocidad de reloj de 16 MHz.

Memoria: El ATmega2560 cuenta con una memoria flash de 256 KB, de los cuales 8 KB se reservan para el bootloader. Además, tiene 8 KB de memoria SRAM y 4 KB de memoria EEPROM.

Pines de entradas/salidas digitales: La placa Arduino Mega proporciona 54 pines de entrada/salida digitales, de los cuales 15 pueden utilizarse como salidas de modulación por ancho de pulso (PWM).

Pines de entrada analógica: Esta placa cuenta con 16 pines de entrada analógica de 10 bits que también pueden utilizarse como pines digitales.

Interfaces de comunicación: Incluye una interfaz USB para la comunicación con el ordenador y la carga del código. Además, dispone de cuatro puertos serie UART (Universal Asynchronous Receiver-Transmitter) y soporte para comunicación I2C (Inter-Integrated Circuit) y SPI (Serial Peripheral Interface).

Alimentación: La placa Arduino Mega puede ser alimentada de dos formas principales: a través de un cable USB conectado al ordenador o mediante una fuente de alimentación externa. Se recomienda una tensión de entrada de 7-12 V, y la placa incluye un regulador de voltaje interno para generar tensiones de 5 V y 3.3 V para alimentar los componentes.

Memoria de programa: Arduino Mega utiliza un bootloader que ocupa 8 KB de la memoria flash, dejando 248 KB disponibles para el programa del usuario.

Dimensiones físicas: La placa Arduino Mega tiene un tamaño aproximado de 101.6 mm x 53.4 mm (Fitzgerald & Shiloh, 2012).

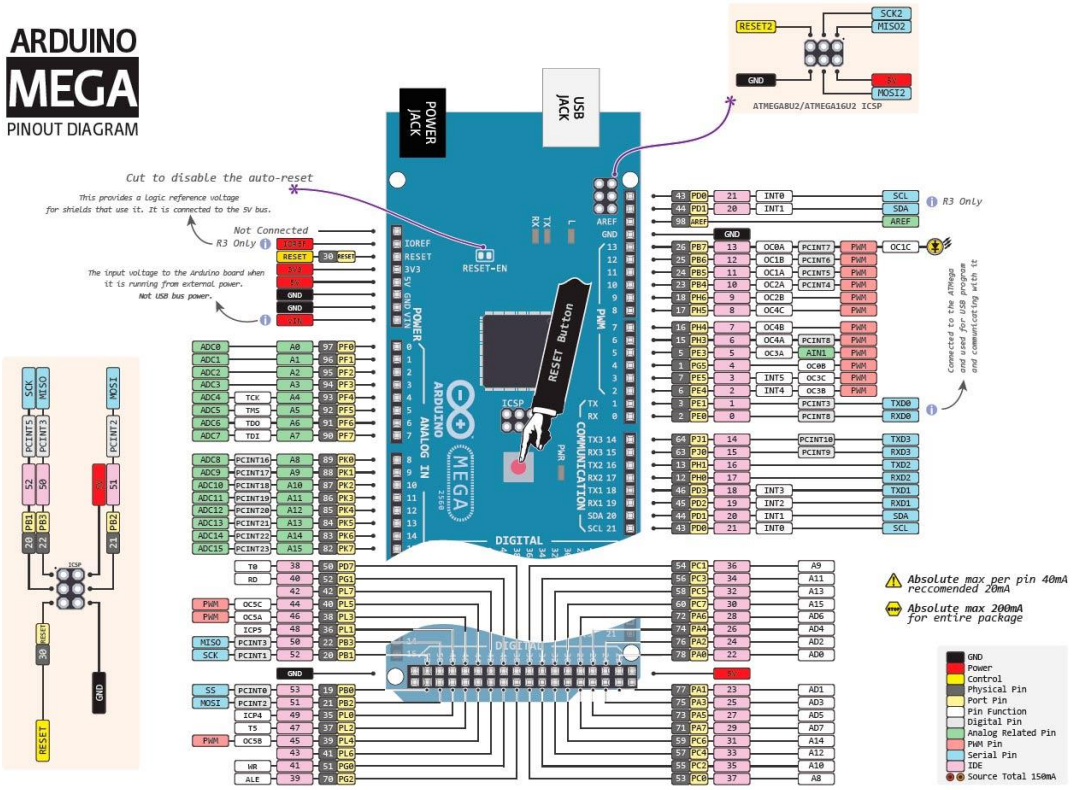

Figura 3. Detalle del hardware de la placa Arduino Mega

La placa Arduino Mega ofrece mayor capacidad de procesamiento, más pines de entrada/salida y mayor memoria en comparación con Arduino Uno. Esto la

convierte en una opción adecuada para proyectos más grandes y complejos que requieren mayores recursos.

2.2.3. Comunicación de las placas Arduino

Las placas Arduino incluyen diferentes medios de comunicación incorporados que les permiten interactuar con otros dispositivos. A continuación, se describen los medios de comunicación más comunes que se encuentran en la mayoría de las placas Arduino.

Comunicación Serial: Las placas Arduino cuentan con una interfaz de comunicación serial que permite enviar y recibir datos en formato serie. Utilizan el puerto UART (Universal Asynchronous Receiver-Transmitter) para esta comunicación. Esto facilita la conexión directa con otros dispositivos a través de cables UART o mediante adaptadores USB a serial.

Interfaz USB: Todas las placas Arduino incluyen un puerto USB integrado que se utiliza para la comunicación con una computadora. Este puerto permite cargar programas en la placa y también enviar y recibir datos entre la placa y la computadora. La comunicación a través del puerto USB se basa en el protocolo de comunicación serial.

Comunicación I2C: Muchas placas Arduino tienen soporte para el protocolo de comunicación I2C (Inter-Integrated Circuit). I2C es un bus de comunicación serie que permite la conexión de múltiples dispositivos mediante solo dos cables: uno para datos (SDA) y otro para el reloj (SCL). Esto simplifica la comunicación con sensores, pantallas y otros dispositivos compatibles con I2C.

Comunicación SPI: Algunas placas Arduino también incluyen soporte para el protocolo de comunicación SPI (Serial Peripheral Interface). SPI es un protocolo de comunicación en serie síncrono utilizado para la transferencia de datos entre dispositivos. Permite una comunicación rápida y bidireccional con una variedad de dispositivos, como pantallas LCD, tarjetas SD y sensores.

Cada protocolo de comunicación ofrece capacidades distintas y se pueden utilizar para interactuar con diversos dispositivos externos. Además de estos medios, también es posible utilizar otros protocolos y periféricos adicionales mediante el uso de librerías y placas de expansión (shields) compatibles con Arduino (Torrente Artero, 2013).

2.3. Comunicación Serial

La comunicación serial es un método utilizado para transferir datos secuencialmente entre dispositivos electrónicos. A diferencia de la comunicación paralela, donde se transmiten varios bits simultáneamente en diferentes líneas, la comunicación serial transfiere los bits uno por uno a través de una única línea de comunicación.

En la comunicación serial, existen dos componentes principales: el transmisor y el receptor. El transmisor toma los datos que se desean enviar y los divide en una secuencia de bits. Estos bits se envían de manera secuencial, uno tras otro, a través de un canal de comunicación hacia el receptor.

Existen varios protocolos de comunicación serial comunes, como RS-232, UART (Universal Asynchronous Receiver/Transmitter), SPI (Serial Peripheral Interface) e I2C (Inter-Integrated Circuit). Cada protocolo tiene sus propias características y especificaciones, pero todos siguen el mismo principio básico de transmitir datos de manera secuencial.

Durante la comunicación serial, los datos se envían en forma de tramas. Una trama es una secuencia estructurada de bits que contiene los datos a transmitir, así como información adicional, como bits de inicio y de parada. Estos bits de inicio y de parada ayudan al receptor a sincronizarse y reconocer el inicio y el final de cada trama.

Puede ocurrir de manera síncrona o asíncrona. En la comunicación síncrona, se utiliza una señal de reloj adicional para sincronizar el transmisor y el receptor. Esto garantiza que ambos dispositivos estén en la misma frecuencia y tiempo

para la transmisión y recepción de los datos. En la comunicación asíncrona, no se emplea una señal de reloj separada, en su lugar, se incluyen bits de inicio y de parada en cada trama para sincronizar los dispositivos.

La elección del protocolo de comunicación serial depende de las necesidades específicas de la aplicación y los dispositivos involucrados. Algunos protocolos son más adecuados para distancias cortas y velocidades de transmisión más bajas, mientras que otros son más apropiados para distancias más largas y velocidades más altas.

2.3.1. Funcionamiento de la comunicación serial

En este proceso, hay un dispositivo transmisor que toma los datos que se desean enviar y los convierte en una secuencia de bits. Estos bits se envían de manera secuencial a través de la línea de comunicación hacia el dispositivo receptor.
Para facilitar la sincronización y el reconocimiento del inicio y el final de cada secuencia de bits, se incluyen bits de inicio y de parada en cada trama de datos. El bit de inicio marca el inicio de la trama, mientras que el bit de parada señala su final.

La velocidad de transmisión, medida en baudios o bits por segundo, indica la cantidad de bits enviados por segundo. Es importante que el transmisor y el receptor tengan la misma velocidad de transmisión para asegurar una comunicación adecuada.

Existen diversos protocolos de comunicación serial, como RS-232, UART, SPI e I2C, que establecen las reglas y especificaciones para la transmisión y recepción

de los datos. Estos protocolos definen aspectos como la estructura de las tramas, los niveles de voltaje, la velocidad de transmisión y la sincronización.
La comunicación serial puede ser síncrona o asíncrona. En la comunicación síncrona, se utiliza una señal de reloj adicional para sincronizar el transmisor y el receptor, asegurando que ambos dispositivos estén en la misma frecuencia y tiempo para la transmisión y recepción de los datos. En la comunicación asíncrona, no se requiere una señal de reloj separada y se utilizan bits de inicio y de parada en cada trama para sincronizar los dispositivos.

Para la comunicación serial, se necesita un cable adecuado para conectar el transmisor y el receptor. Se utilizan cables con conectores específicos, como los conectores DB9 o DB25 en el caso de RS-232. Es importante que la configuración de los pines del cable sea compatible con los estándares y requisitos del protocolo de comunicación utilizado.

La distancia máxima y la velocidad de transmisión admitidas en una comunicación serial pueden variar según el protocolo y los dispositivos involucrados. Algunos protocolos son más adecuados para distancias cortas y velocidades de transmisión más bajas, mientras que otros son más apropiados para distancias más largas y velocidades más altas.

2.4. Entornos virtuales simulados

Los entornos virtuales simulados son representaciones digitales de entornos del mundo real que se crean con el propósito de llevar a cabo simulaciones y experimentos. Estos entornos virtuales se diseñan para reproducir características y comportamientos específicos de situaciones reales, como entornos físicos, sistemas complejos o fenómenos naturales.

Los entornos virtuales simulados se utilizan en diversos campos, como la investigación científica, la ingeniería, la medicina, la robótica, los videojuegos y la formación. Estas simulaciones permiten explorar escenarios hipotéticos, evaluar el rendimiento de sistemas, predecir resultados y adquirir conocimiento en un entorno seguro y controlado (Lidon, 2019).

Existen diversas herramientas y plataformas disponibles para crear entornos virtuales simulados. Algunas de ellas son:

Unity: Unity es un motor de desarrollo de videojuegos y una plataforma de simulación ampliamente utilizada. Permite crear entornos virtuales interactivos, agregar física y comportamientos realistas, y programar la lógica del entorno.

Unreal Engine: Unreal Engine es otro motor de desarrollo de videojuegos y simulación popular. Ofrece herramientas avanzadas para crear entornos virtuales de alta fidelidad, con gráficos realistas, efectos especiales y simulación precisa de la física.

MATLAB/Simulink: MATLAB y Simulink son herramientas de software utilizadas en diversos campos de la ciencia y la ingeniería. Permiten modelar y simular

sistemas complejos, desde sistemas físicos hasta sistemas de control y comunicaciones.

Gazebo: Gazebo es una plataforma de simulación de código abierto ampliamente utilizada en robótica. Proporciona un entorno virtual para simular robots, sensores, entornos físicos y escenarios de interacción.

2.5. Unity

Unity es una herramienta de desarrollo de videojuegos y una plataforma para crear contenido en 3D, realidad virtual (RV) y realidad aumentada (RA). Fue lanzado por Unity Technologies en 2005 y ha ganado mucha popularidad en la industria del desarrollo de juegos y aplicaciones interactivas (Lidon, 2019).

Se utiliza para crear una amplia variedad de experiencias interactivas, incluyendo juegos para PC, consolas, dispositivos móviles y web, así como aplicaciones de RV y RA, simuladores, visualización arquitectónica y aplicaciones de entrenamiento. Algunas de las características y ventajas clave de Unity son las siguientes:

Multiplataforma: Unity permite el desarrollo de juegos y aplicaciones que se ejecutan en diferentes plataformas, como Windows, macOS, Linux, iOS, Android, consolas de juegos y la web.

Motor gráfico en tiempo real: Unity cuenta con un motor gráfico en tiempo real potente que ofrece gráficos de alta calidad y la posibilidad de crear entornos interactivos inmersivos.

Editor intuitivo: Unity proporciona un editor visual y un flujo de trabajo intuitivo que facilita el diseño, desarrollo y ajuste de los componentes del juego. No se requiere conocimientos de programación para realizar ciertas tareas.

Lenguajes de programación: Unity admite varios lenguajes de programación, como C# y JavaScript, lo que permite a los desarrolladores implementar lógica de juego personalizada y funcionalidades avanzadas.

Asset Store: Unity cuenta con un Asset Store que ofrece una amplia variedad de recursos, como modelos 3D, texturas, scripts y herramientas, que se pueden utilizar en proyectos para acelerar el desarrollo y mejorar la calidad visual.

Comunidad activa: Unity cuenta con una gran comunidad de desarrolladores en línea que comparten conocimientos, tutoriales y soluciones a problemas comunes, lo que facilita el aprendizaje y la resolución de dificultades (Jiménez Gil, 2023).

Unity se utiliza ampliamente, tanto por estudios de desarrollo independientes como por grandes empresas de videojuegos. Permite crear juegos de diferentes géneros, desde casuales y móviles hasta títulos de alta calidad. Además, Unity ha ganado popularidad en la industria de la arquitectura y la visualización, ya que se puede utilizar para crear simulaciones y presentaciones interactivas de edificios y espacios. También se utiliza en aplicaciones de entrenamiento y simuladores en diversos sectores, como medicina, aviación y educación (Linowes, 2015).

2.5.1. Unity como herramienta de ingeniería

Unity es una herramienta utilizada en ingeniería en diversas aplicaciones para mejorar la visualización, la simulación y la formación en entornos virtuales. A continuación, te mencionaré algunas formas en las que Unity se utiliza en el campo de la ingeniería son:

Visualización de datos y modelos: Unity permite importar modelos CAD y visualizarlos en 3D de manera interactiva. Esto facilita la comprensión de los modelos y la identificación de posibles problemas o mejoras en ingeniería de

diseño y arquitectura. Además, Unity puede integrarse con datos en tiempo real, como mediciones y sensores, para visualizar información compleja de manera más accesible y comprensible.

Simulaciones y prototipos virtuales: Unity se utiliza para crear simulaciones y prototipos virtuales en ingeniería. Esto permite simular el comportamiento de objetos físicos y sistemas complejos antes de construirlos físicamente, lo que puede ahorrar tiempo y costos en la etapa de desarrollo. Por ejemplo, se puede simular el rendimiento de estructuras, la mecánica de materiales, el flujo de fluidos o el comportamiento electromagnético.

Formación y entrenamiento: Unity se utiliza para desarrollar aplicaciones de formación y entrenamiento en ingeniería. Los usuarios pueden interactuar con simulaciones y prácticas virtuales para aprender y adquirir habilidades técnicas. Esto es especialmente útil en campos como la ingeniería industrial, la ingeniería de fabricación y la ingeniería de procesos, donde se pueden simular y practicar diferentes escenarios y procedimientos.

Realidad Virtual (RV) y Realidad Aumentada (RA): Unity se emplea en la creación de aplicaciones de RV y RA para ingeniería. Estas tecnologías permiten a los ingenieros visualizar y manipular objetos y sistemas complejos en entornos virtuales o superponer información digital en el mundo real. Por ejemplo, se pueden crear aplicaciones de RA para asistencia en la reparación y mantenimiento de equipos, o aplicaciones de RV para la visualización inmersiva de plantas industriales.

Análisis de datos: Unity se puede combinar con herramientas de análisis de datos para visualizar y comprender grandes conjuntos de datos en ingeniería. Por ejemplo, se pueden visualizar datos geoespaciales, datos de sensores o resultados de simulaciones en tiempo real para tomar decisiones informadas y realizar análisis de rendimiento (Lidon, 2019).

3. Diseño y experimentación

3.1. Conceptos de operación

El sistema que se va a desarrollar cumplirá con las siguientes especificaciones:

- Se van a utilizar 10 sensores de diferentes tipos que miden diferentes magnitudes del entorno.
- La placa Arduino realizará el proceso de adquisición y procesamiento de datos que posteriormente se enviarán al computador para ser utilizados por el motor gráfico Unity.
- Se diseñará e implementará un entorno virtual donde los datos proporcionados por la placa Arduino permitirán al usuario interactuar con diferentes elementos digitales.
- Dentro del entorno se quiere generar un registro de la lectura de todos los sensores y elaborar histogramas para su visualización.

3.2. Requerimientos del sistema

Las características específicas de los dispositivos que se utilizarán para el proyecto, y las herramientas de software se describen a continuación.

3.2.1. Sensor Grove GSR

La sigla GSR corresponde a 'respuesta galvánica de la piel', y se utiliza para evaluar la conductividad eléctrica de la piel. Esta técnica es capaz de mostrar las emociones que experimentan las personas.

Figura 4. Sensor GSR

Tabla 2. Características Grove GSR sensor

Parámetro	Valor
Voltajes de operación	3.3/5 V
Sensibilidad	Ajustable vía potenciómetro
Señal de salida	Tensión, lectura analógica
Material del contacto con la piel	Níquel
Tipo de sensor	Conductivo

3.2.2. Sensor de latidos del corazón KY-039

El funcionamiento de este sensor se basa en el siguiente principio: hay un LED que emite luz desde un extremo a través del dedo del usuario, mientras que en el otro extremo del dedo se encuentra un fototransistor que recoge la luz transmitida. Cuando el pulso de la presión arterial atraviesa el dedo, la corriente de la base del fototransistor experimenta una pequeña alteración debido a la disminución de la cantidad de luz que puede pasar a través del dedo. Esto, a su vez, provoca una salida diferente en el puerto analógico.

Para asegurar una medición precisa con el sensor, es esencial mantenerlo alejado de cualquier luz externa no deseada, como la luz ambiental del entorno. Por esta razón, se sugiere utilizar algún tipo de protección para el fototransistor, ya que la señal del latido cardíaco es bastante tenue y la luz ambiente puede introducir un nivel significativo de interferencia.

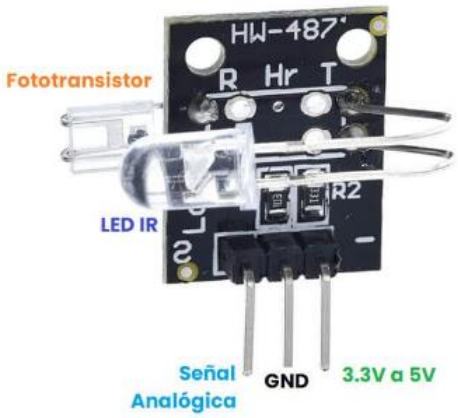

Figura 5. Sensor KY-039

Tabla 3. Características del sensor KY-039

Parámetro	Valor
Voltajes de operación	3.3/5 V
Señal de salida	Analógica
Tipo de sensor	Óptico

3.2.3. Sensor de impacto KY-031

El dispositivo viene preconfigurado con una resistencia pull-up de 10k ohmios. Esto implica que, cuando detecta un impacto, su funcionamiento se asemeja al de un interruptor o conmutador. El dispositivo consta de tres pines: uno de alimentación (VCC), otro de tierra (GND), y un tercero destinado a la señal de salida. En el momento en que se produce un contacto físico o un impacto, este último pin emite una señal lógica "1" de forma inmediata.

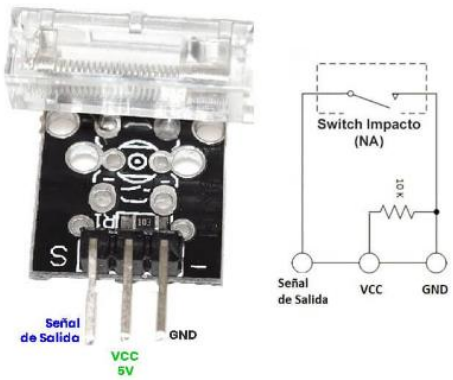

Figura 6. Sensor KY-031

Tabla 4. Características del sensor KY-031

Parámetro	Valor
Voltajes de operación	3.3/5 V
Corriente de operación	10 mA
Señal de salida	Digital 0 / 1 – (0 volts – 5 volts)
Tipo de sensor	Switch de contacto

3.2.4. Sensor de obstáculos KY-032

Este dispositivo está equipado con un par de sensores compuestos por un transmisor y un receptor de infrarrojos. El LED del transmisor emite una frecuencia específica de infrarrojos que el LED del receptor detecta. Una vez que la parte de la señal es identificada, activa el pin de "señal" en modo de encendido/apagado digital cuando se alcanza una distancia umbral específica. Posee dos potenciómetros, uno de ellos destinado a ajustar la distancia desde el objeto en el cual el sensor detecta y el otro encargado de modificar la frecuencia de infrarrojos emitida por el transmisor.

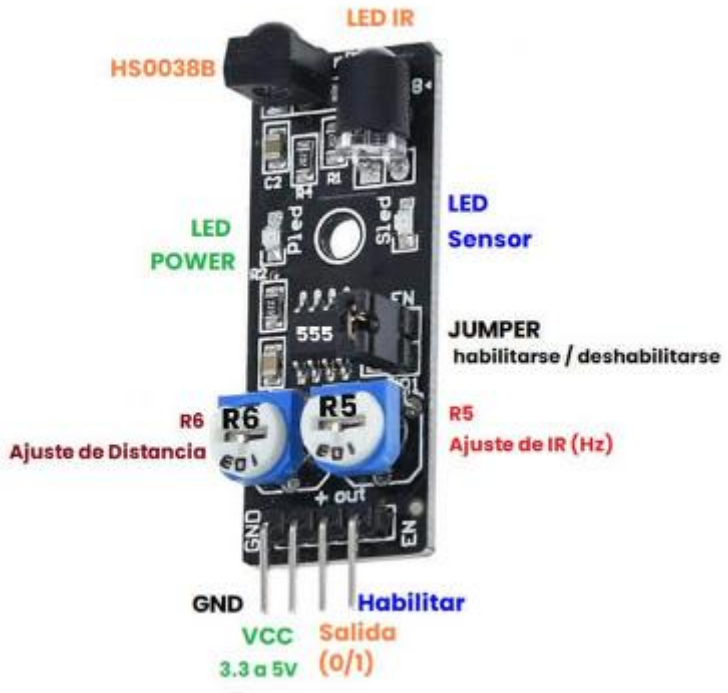

Figura 7. Sensor KY-032

Tabla 5. Características del sensor KY-032

Parámetro	Valor
Voltajes de operación	3.3/5 V
Corriente de operación	20 mA
Distancia de Detección	2 a 40cm
Ángulo efectivo de operación	35°
Tipo de sensor	Infrarrojo

3.2.5. Sensor de campo magnético KY-024

El efecto Hall, descubierto por Edwin Hall en 1879, se basa en su observación de que cuando un flujo de electrones atraviesa un conductor y se expone a un campo magnético, se genera un voltaje proporcional al producto de la fuerza del campo magnético y el flujo de electrones.

En el caso del módulo KY-024, se espera que la intensidad del punto 0 alcance aproximadamente 2.5 V, con una variación entre 0 V y 5 V en un rango de 1024 valores cuando se alimenta de manera con 5 V. A pesar de que este sensor exhibe una respuesta lineal, no resulta apropiado para medir con precisión la intensidad de un campo magnético debido a su baja exactitud.

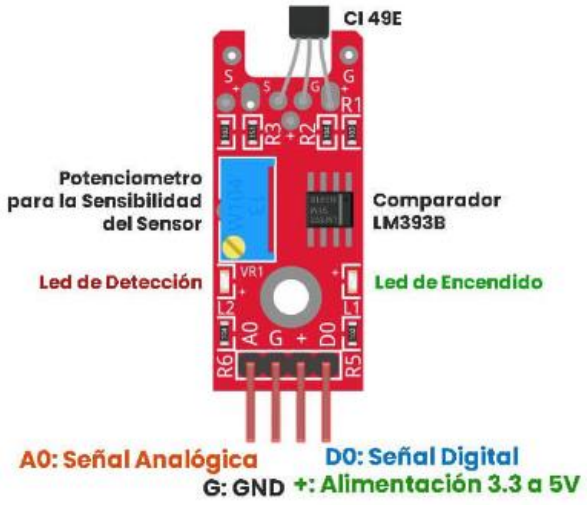

Figura 8. Sensor KY-024

Tabla 6. Características del sensor KY-024

Parámetro	Valor
Voltajes de operación	3.3/5 V
Corriente de operación	16 mA
Señal de salida	Analógica/Digital
Tipo de sensor	Efecto hall

3.2.6. Sensor micrófono KY-038

Este sensor se utiliza para detectar sonidos y funciona a través de un detector de ondas sonoras que las transforma y las envía como una señal eléctrica a un dispositivo receptor/codificador.

Este dispositivo genera dos tipos de señales, una digital y otra analógica, se debe seleccionar la más adecuada según la aplicación. Si se busca obtener el valor exacto del sensor, se puede utilizar la salida analógica para obtener los datos. Alternativamente, se puede emplear la salida digital, que se activa o desactiva según si el sensor detecta un nivel de sonido por encima del umbral de sensibilidad previamente configurado.

La placa de circuito del sensor KY-038 consta de tres componentes principales. La unidad del sensor, un amplificador y un comparador. Además, el sensor cuenta con un indicador LED que señala su estado de encendido, así como otro LED que indica la salida del comparador.

Figura 9. Sensor KY-038

Tabla 7. Características del sensor KY-038

Parámetro	Valor
Voltajes de operación	5 V
Umbral	Ajustable por potenciómetro
Señal de salida	Analógica/Digital
Tipo de sensor	Micrófono

3.2.7. Sensor Joystick KY-023

El módulo KY-023 proporciona un posicionamiento en ejes que se expresa a través de valores que oscilan entre 0 y 1023. Conforme movemos el joystick, estos valores varían, proporcionando las coordenadas de cada eje, y es posible imprimir este valor en el monitor serial del entorno de desarrollo Arduino.

La posición de reposo del joystick se encuentra en 512, con pequeñas variaciones debido a posibles imprecisiones en los resortes y el mecanismo. Al mover el joystick, notará que los valores cambian de 0 a 1023 según su posición.

Estado del Joystick en Reposo (centro):

Eje X = 512 (Tensión de salida: 2.5V)

Eje Y = 512 (Tensión de salida: 2.5V)

Joystick hacia arriba:

Eje X = 1023 (Tensión de salida: 5V)

Eje Y = 512 (Tensión de salida: 2.5V)

Joystick hacia abajo:

Eje X = 0 (Tensión de salida: 0V)

Eje Y = 512 (Tensión de salida: 2.5V)

Joystick hacia la derecha:

Eje X = 512 (Tensión de salida: 2.5V)

Eje Y = 1023 (Tensión de salida: 5V)

Joystick hacia la izquierda:

Eje X = 512 (Tensión de salida: 2.5V)

Eje Y = 0 (Tensión de salida: 0V)

Figura 10. Sensor KY-023

Tabla 8. Características del sensor KY-038

Parámetro	Valor
Voltajes de operación	3.3/5 V
Señal de salida	Analógica/Digital
Tipo de sensor	Potenciómetro

3.2.8. Sensor de flama KY-026

El sensor de llama KY-026 cuenta con un fotodiodo altamente sensible al espectro de luz emitido por una llama abierta, es capaz de detectar longitudes de onda que se encuentran en el rango de 760 nm a 1100 nm, que corresponde al espectro infrarrojo. Cuando se detecta la presencia de una llama, su salida digital cambia a estado de alto, ya la salida analógica proporciona una medición directa de la lectura.

Este dispositivo no debe entrar en contacto directo con una llama, ya que podría causar el derretimiento o la combustión del plástico protector, se recomienda mantener el sensor a una distancia segura y razonable de la fuente de la llama.

Figura 11. Sensor KY-026

Tabla 9. Características del sensor KY-026

Parámetro	Valor
Voltajes de operación	3.3/5 V
Corriente de operación	15mA
Sensibilidad	Ajustable por potenciómetro
Rango de detección	760nm a 1100nm
Ángulo de detección	60°
Señal de salida	Analógica/Digital
Tipo de sensor	Óptico

3.2.9. Sensor de temperatura y humedad KY-015

Este módulo sensor está diseñado para medir la temperatura y la humedad relativa en el entorno. También se lo conoce como DHT11, ya que incorpora este circuito en su interior. Este sensor ofrece mediciones de temperatura en grados Celsius y humedad relativa en porcentaje. Genera una señal digital que puede ser interpretada por microcontroladores como Arduino o Raspberry Pi para monitorear estas variables ambientales.

Se utiliza en proyectos de electrónica y automatización, especialmente en aplicaciones relacionadas con el control climático, la gestión del entorno y la supervisión en tiempo real de las condiciones ambientales. Para su uso generalmente se requerirá una biblioteca específica del sensor para las diferentes plataformas como Arduino, esto permitirá obtener mediciones precisas de temperatura y humedad.

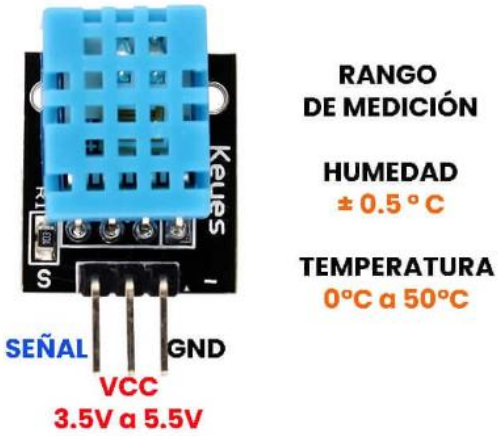

Figura 12. Sensor KY-015

Tabla 10. Características del sensor KY-015

Parámetro	Valor
Voltajes de operación	3.3/5 V
Rango de medición de humedad	20% a 90% RH (Error de medición de humedad: +-5%)
Resolución de medición de humedad	1% RH
Rango de medición de temperatura	0°C a 50°C (Error de medición de temperatura: +-2°)
Resolución de medición de temperatura	1°C
Señal de salida	Digital
Tipo de sensor	Capacitivo

3.2.10. Sensor encoder rotativo KY-040

Un encoder es un dispositivo electromecánico que traduce el movimiento mecánico en diversos tipos de señales eléctricas, como señales binarias digitales

o señales analógicas basadas en ondas y pulsos. Esencialmente, este dispositivo actúa como un intermediario entre un componente mecánico móvil y un controlador.

En estos dispositivos, la lectura se lleva a cabo mediante un disco en el que se encuentra la codificación necesaria para determinar con gran precisión la posición angular.

El funcionamiento de un codificador rotativo es bastante simple. Cuando se gira el eje incremental, se generan dos señales cuadradas conocidas como canal A (CLK) y canal B (DT). Si el pin común se conecta a tierra y se utilizan resistencias pull-up para los canales A y B, las señales A y B producirán pulsos desfasados entre sí en 90 grados, siguiendo esta secuencia:

GIRO	Tiempo	PINES	
		A(CLK)	B(DT)
Horario	t1	0	1
	t2	0	0
	t3	1	0
	t4	1	1
Antihorario	t1	1	0
	t2	0	0
	t3	0	1
	t4	1	1

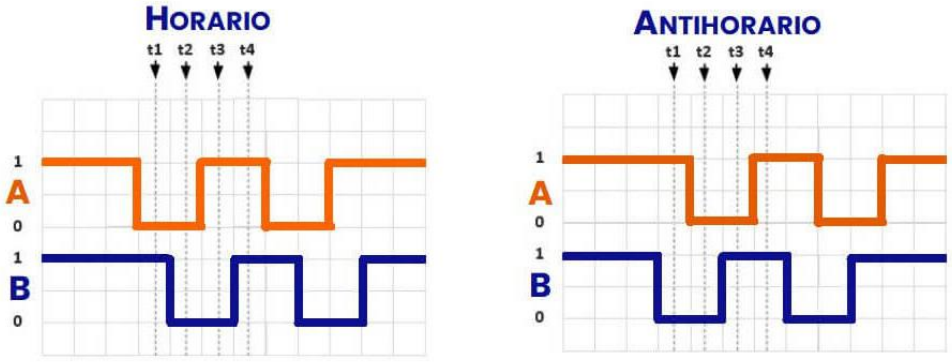

Figura 13. Código Grey de 2 bits para encoders

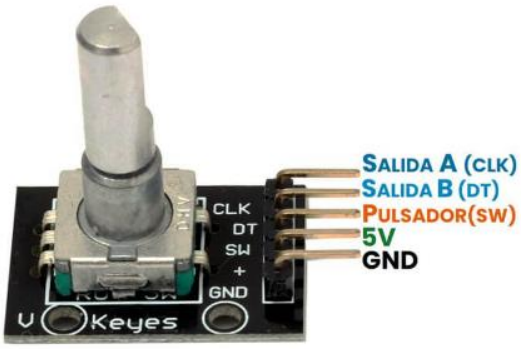

Figura 14. Sensor KY-040

Tabla 11. Características del sensor KY-038

Parámetro	Valor
Voltajes de operación	5 V
Corriente de operación	10 mA
Desfase entre señal A y B	90°
Rotación Angular	30°
Señal de salida	Digital
Tipo de sensor	Encoder

3.2.11. Arduino UNO

Se ha seleccionado a la placa Arduino UNO Rev.3 para la implementación del proyecto como tarjeta de adquisición y procesamiento de datos, esto debido a sus prestaciones y la abundante documentación que existe para su manejo y programación con los sensores antes mencionados. En total se tienen 5 señales analógicas y 8 señales digitales que traban con voltajes de 3.3 V o 5 V y con corrientes que varían entre los 10 mA a los 20 mA, y cuenta con comunicación serial. Las características de la placa Arduino UNO se detallan en la siguiente tabla:

Tabla 12. Características ARDUINO UNO Rev. 3

Microcontrolador	ATmega328
Voltaje de funcionamiento	5 V
Alimentación	7-12 V
Voltaje máximo de entrada	20 V
Pines digitales I/O	14 (6 con PWM)

Pines de entradas analógicas	6
Corriente DC por I/O Pin	40 mA
Corriente DC para pin 3.3V	50 mA
Memoria Flash	32 KB
SRAM	2 KB
EEPROM	1 KB
Velocidad de reloj	16 MHz

3.2.12. Características del ordenador para la implementación

El ordenador que se va a utilizar para el desarrollo del entorno cuenta con las siguientes características:

Tabla 13. Características del ordenador

Procesador	AMD Ryzen 7 2700X
Velocidad del procesador	3.7 GHz
Número de núcleos	8
Números de hilos	16
Memoria RAM	16 GB
Tarjeta de video	NVIDIA GeForce RTX 2060

3.2.13. Herramientas de software

Las herramientas de software que se necesitan para el desarrollo del entorno y la integración de los sensores con el mismo son las siguientes:

3.2.13.1. Unity

La herramienta elegida para el desarrollo del entorno es Unity, esto debido a que tiene mucha documentación sobre las herramientas de programación que se pueden utilizar para configurar los entornos y su facilidad de uso. También es importante mencionar que cuenta con librerías que simplifican la programación y que su interfaz permite mayor control de los recursos del sistema a desarrollar. La versión que se ha elegido para el desarrollo es la 2021.3.21f1, la misma que cuenta con todo el soporte necesario para la implementación del entorno. Esta versión es estable y al momento de ejecución del presente trabajo sigue

recibiendo soporte por Unity. Además, cuenta con compatibilidad de compilación de Windows y librerías de AI que serán usadas para el desarrollo del proyecto como NavMesh.

3.2.13.2. Arduino IDE

Para la programación de Arduino UNO se utilizará el Arduino IDE versión 2.2.1, compatible con Windows 64bits. Además de las librerías del fabricante de los sensores Adafruit, y la librería para la programación de tareas FreeRTOS que permite implementar concurrencia con limitaciones en la placa Arduino.

3.3. Diseño del sistema

Para el entorno se han seleccionado diversas tareas que simulan un espacio industrial o tecnológico con el uso de sensores. Estará dividido por zonas y actividades en las que el usuario podrá interactuar, las que se explican a continuación:

La cámara principal del usuario estará en primera persona y tendrá dos indicadores uno de temperatura y otro de humedad que permanecerán activos todo el tiempo y varían en función de la lectura del sensor KY-015.

En el techo del edificio del entorno se colocarán rociadores automáticos de agua que se activarán cuando se detecte una llama con el sensor KY-026.

Dentro de un espacio confinado se colocará el modelo 3D de un robot móvil que se desplazará en este espacio de forma circular, cuando el usuario ingrese a la zona se activará una tarea que permita la lectura del sensor KY-032. El usuario activará el sensor cuando coloque un objeto frente al emisor de este, esta acción resultará en la creación de un obstáculo frente al robot en el entorno virtual y este estará programado para esquivarlo.

En otro espacio se colocarán dos tipos de brazos robóticos que se activarán cuando el usuario entre a la zona determinada desencadenando una tarea de detección de dos sensores, para el primer tipo de robots se activará el sensor KY-023 y para el segundo tipo se activará el sensor KY-040. En ambos casos estos sensores controlarán el movimiento de las articulaciones de los brazos robóticos.

En el siguiente espacio se colocará un sistema que simule la clasificación de piezas, también se activará la tarea cuando el usuario acceda en la zona específica. La clasificación se hará con el sensor KY-024, que se activará cuando el usuario acerque un imán a este.

Para otro espacio se colocará una banda de transporte de piezas de cristal, cuando el usuario acceda en esta sección se podrá activar mediante un golpe el sensor KY-031 que causará que los cristales se rompan.

En una nueva sección se implementará una pantalla donde el usuario podrá observar una serie de videos con diferentes estímulos emocionales, y se activará la tarea para la detección del sensor GSR y KY-039, y así medir la conductividad de la piel y el ritmo cardiaco del usuario.

En el último espacio se colocarán una serie de luces que funcionarán con el sensor KY-038. Estando en la zona del sensor, este se activará y el usuario tendrá que generar un sonido que, dependiendo de la intensidad del sonido, se encenderá una cantidad de luces específicas. Mientras más intenso sea el sonido, más luces se encenderán.

En el entorno se colocará una pantalla donde el usuario podrá visualizar el estado de los 10 sensores utilizados en el proyecto, se mostrará un diagrama lineal con el comportamiento de cada uno y servirá como un centro de control.

3.3.1. Definición de estados del sistema

El sistema básicamente cuenta con tres estados definidos, en cada uno de ellos se ejecutarán diversas tareas. El primer estado se ha definido como “Movimiento Libre”, en el cual el usuario puede recorrer el entorno y acceder a cada una de las zonas de interacción; además, en este estado se realizará la medición continua de la temperatura, la humedad y del sensor de llama para activar los extintores del techo. El segundo estado tiene el nombre de “Panel de control”, este se activa cuando el usuario utilice el sistema de control central, donde se medirá la actividad de todos los sensores y se crearán diagramas lineales con el comportamiento de cada uno de ellos. Para el tercer estado que es “Zona de sensor” se producirá la interacción del usuario con los sensores correspondientes al espacio del entorno. En los siguientes diagramas se muestra el comportamiento descrito:

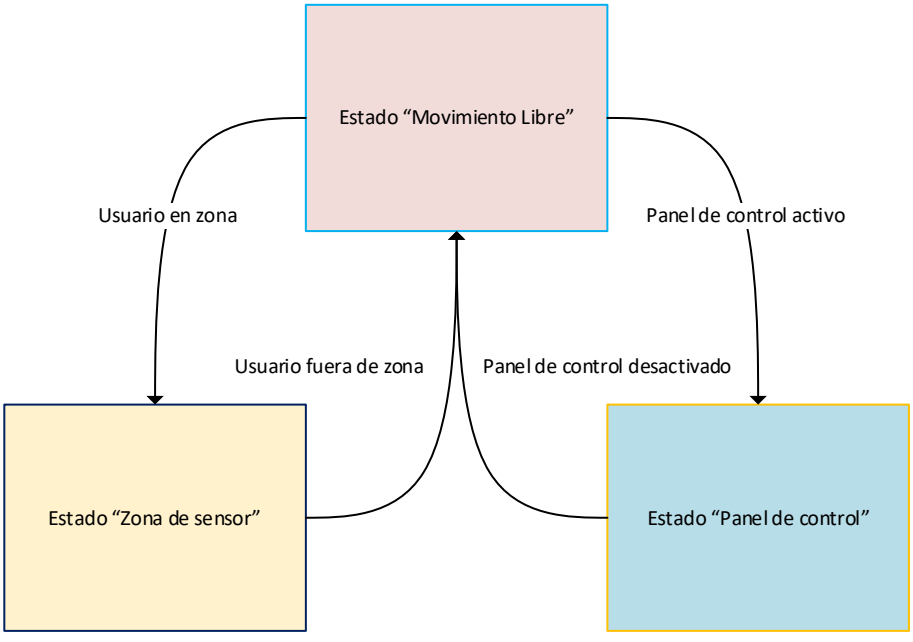

Figura 15. Diagrama de estados del sistema general

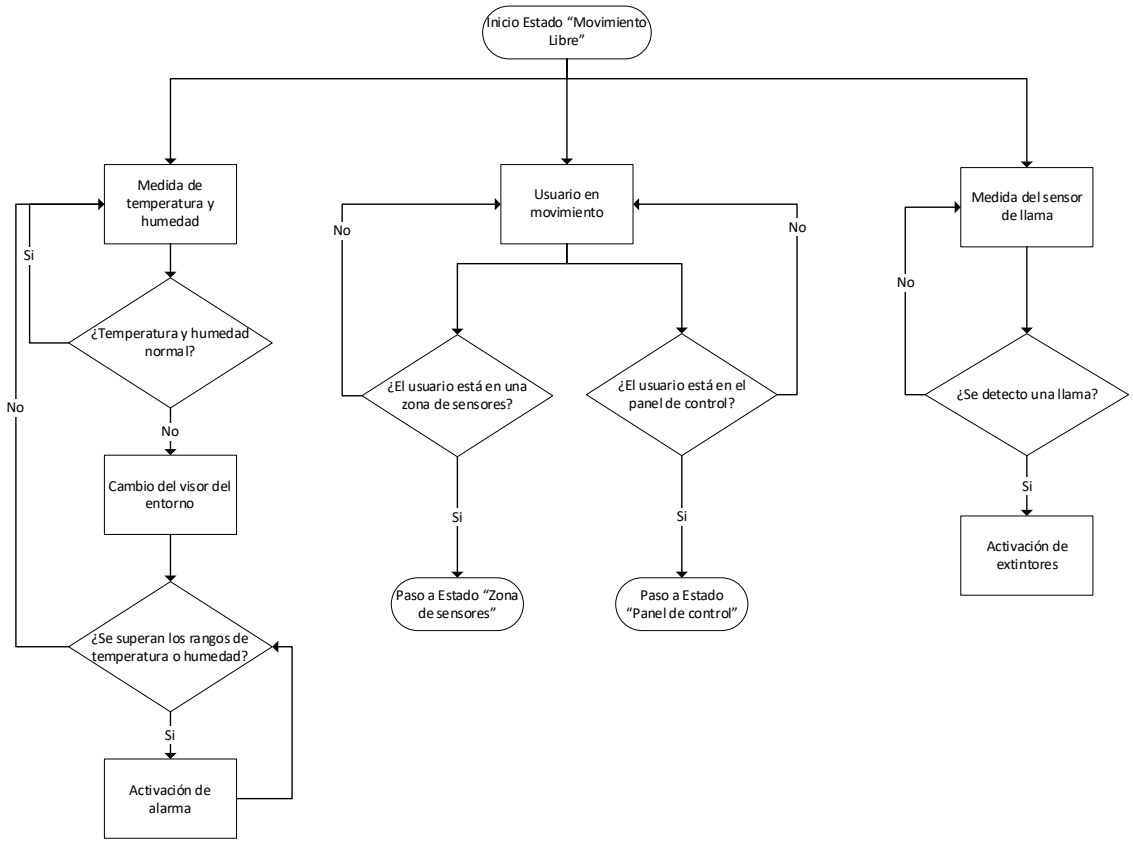

Figura 16. Diagrama del flujo del estado "Movimiento Libre"

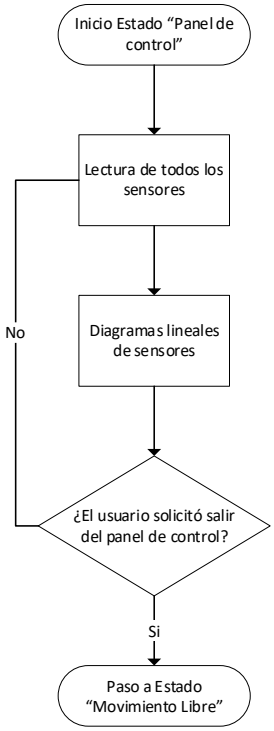

Figura 17. Diagrama de flujo del estado "Panel de control"

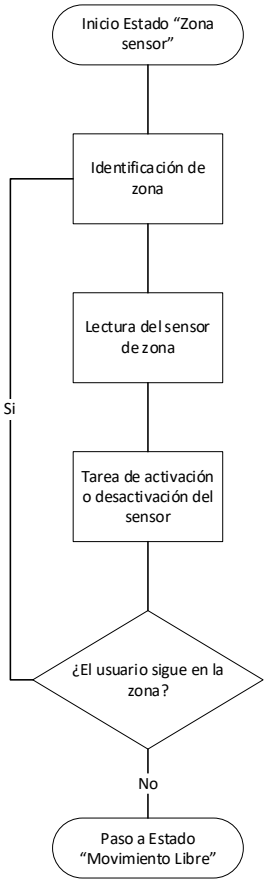

Figura 18. Diagrama de flujo del estado "Zona de sensor"

3.3.2. Esquema electrónico de conexión

Para la correcta adquisición de datos mediante la placa Arduino es necesario realizar una conexión adecuada de los sensores que se utilizarán en la aplicación, de esta manera también se ordena y definen los pines que se utilizarán para cada uno de ellos. El diagrama de conexionado se muestra en la siguiente figura:

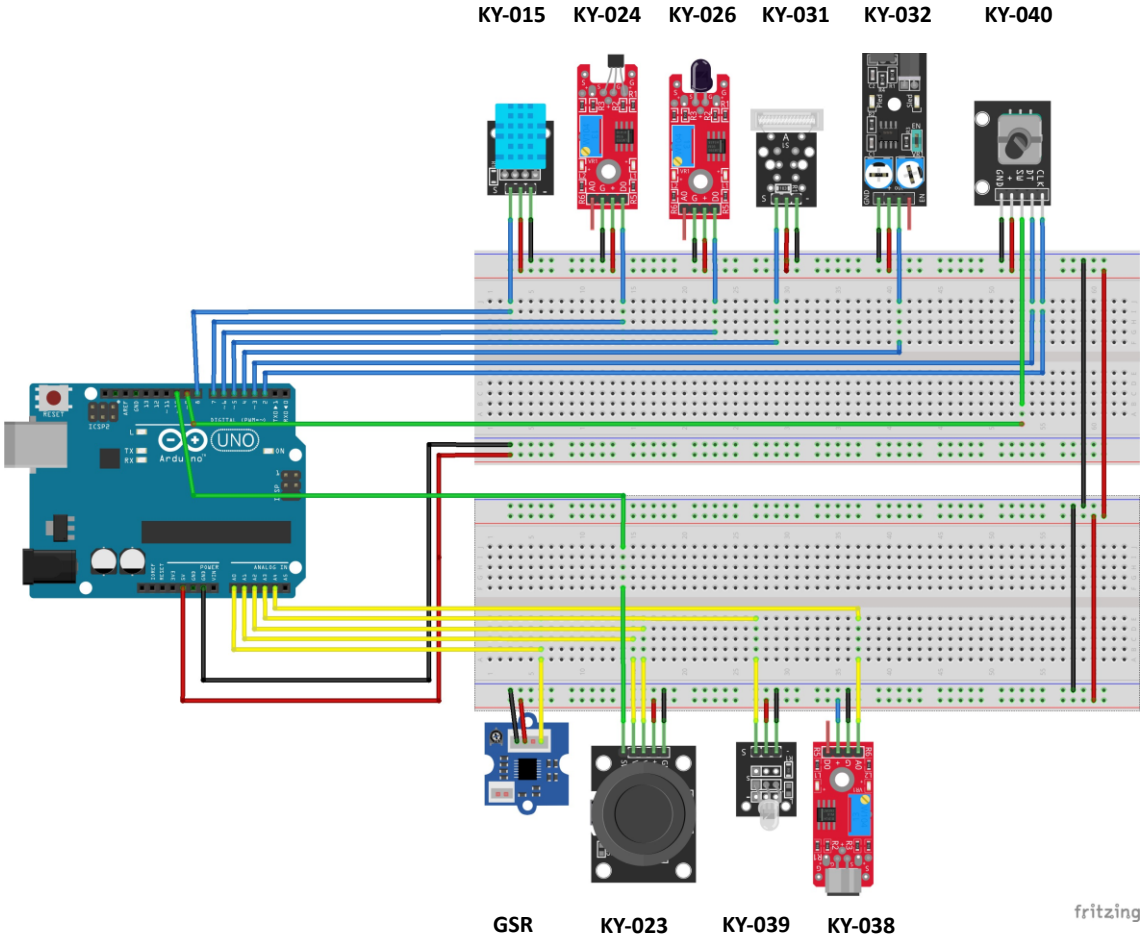

Figura 19. Distribución y conexionado de sensores en la placa Arduino

4. Implementación y pruebas del sistema

4.1. Prueba unitaria de sensores

Se realizaron las pruebas unitarias de sensores mediante los códigos específicos en Arduino para verificar el correcto funcionamiento de cada uno de ellos. Para los sensores analógicos y el sensor KY-015 se muestreó la señal con un periodo de 100 ms y se obtuvieron gráficas de las lecturas, se querían obtener al menos 5000 muestras para comprobar si existe un ruido significativo en las lecturas de cada sensor. Para los sensores digitales se realizaron pruebas de activación y desactivación de las mismas y se observó la cantidad de veces que el sistema respondía correctamente al estimulo.

4.1.1. Pruebas KY-015

Para la programación de este sensor se utilizó la librería DHT sensor library y el siguiente código:

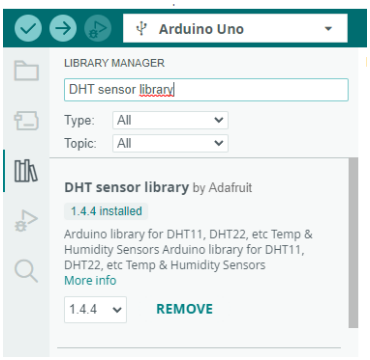

Figura 20. Librería DHT sensor Arduino

```
#include <DHT.h>

#define DHTPIN 8       // Define el pin al que está conectado el sensor
KY-015 (cambia esto si lo has conectado a otro pin)
#define DHTTYPE DHT11 // Indica el tipo de sensor que estás utilizando

DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600); // Inicia la comunicación serial
  dht.begin();         // Inicializa el sensor DHT
}
```

```
void loop() {
  // Lee la temperatura y la humedad del sensor
  float temperatura = dht.readTemperature();
  float humedad = dht.readHumidity();

  // Verifica si la lectura del sensor fue exitosa
  if (isnan(temperatura) || isnan(humedad)) {
    Serial.println("Error al leer el sensor DHT11");
    delay(2000);
    return;
  }

  // Imprime las lecturas en el monitor serial
  Serial.print ("Temperatura: ");
  Serial.println(temperatura);
  Serial.print ("Humedad: ");
  Serial.println (humedad);

  delay(100);  // Espera 100 milisegundos antes de realizar la próxima
lectura
}
```

Los resultados obtenidos se muestran en las siguientes gráficas:

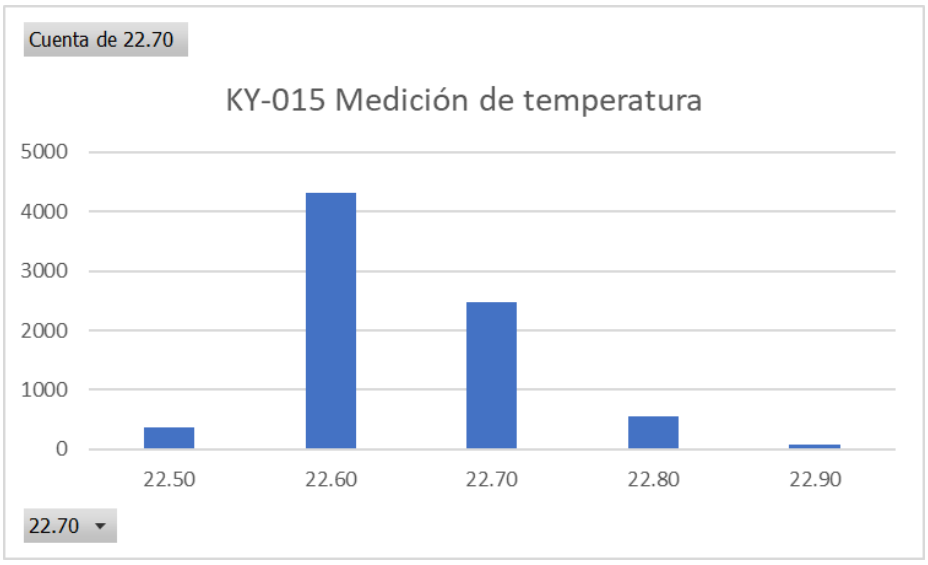

Figura 21. Mediciones de temperaturas en °C sensor KY-015

Para el caso del sensor de temperatura se tuvieron 7791 muestras de las cuales 4320 registraron una temperatura de 22.60°C y 2471 una temperatura de 22.70 °C, además se pueden ver variaciones que van desde los 22.50°C a los 22.90°C,

se podría decir que el error que tiene el sensor es de ±0.3°C, para esta aplicación se considera poco significativo este error y no se coloca ningún filtro adicional.

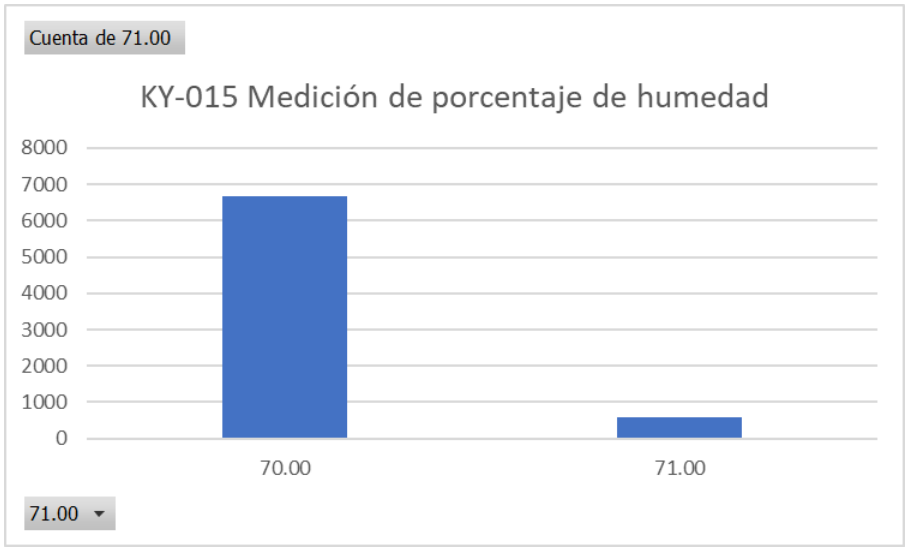

Figura 22. Mediciones de humedad en % sensor KY-015

Para la humedad se tomaron 7260 muestras de las cuales 6673 registraron un valor de 70% de humedad en el ambiente y 587 un valor de 71%. Se puede decir que el error del sensor para la medición de humedad es de ±1% de humedad, nuevamente este valor es poco significativo para esta aplicación.

4.1.2. Pruebas sensor GSR

Para las pruebas del sensor el fabricante sugiere que se establezca la sensibilidad del mismo hasta que se marque un valor de 512 en las lecturas, esto se lo puede realizar girando el potenciometró de la placa.

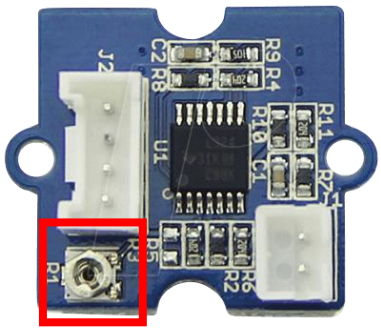

Figura 23. Potenciómetro para ajuste de sensibilidad sensor GSR

Para las pruebas se utilizó el siguiente código:

```
const int GSR=A0; //Se define el sensor GSR en el puerto analógico A0
int sensorValue=0; //Se inicializan las variables

void setup(){
  Serial.begin(9600); //Se establece la velocidad de transmisión de datos
en 9600 baudios
}

void loop(){

  sensorValue=analogRead(GSR); //Se lee la señal del sensor
  Serial.println(sensorValue); //Imprimer el valor del sensor en el
puerto serial
  delay(100); //Se espera un tiempo de 100ms para la siguiente medida
}
```

Los resultados de las lecturas del sensor fueron las siguientes:

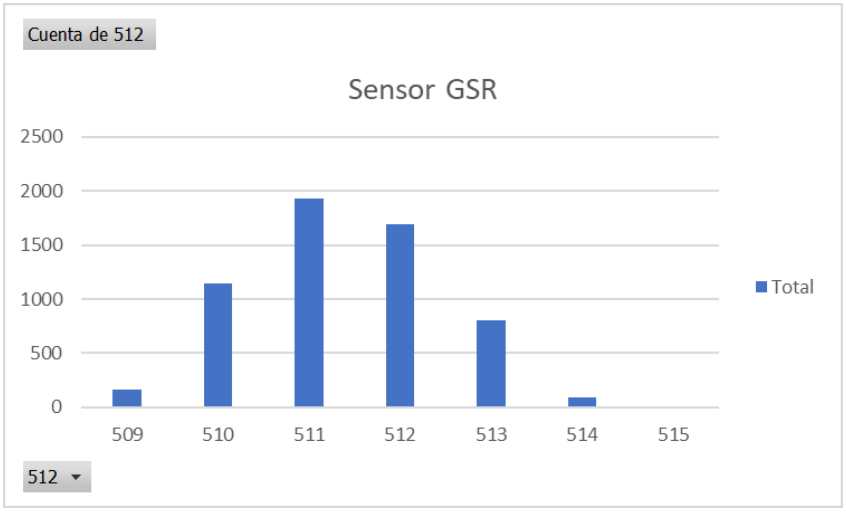

Figura 24. Mediciones del sensor GSR

Se obtuvieron 5818 muestras con el sensor y como se observa en la gráfica se tiene una distribución de datos similar para tres valores, la lectura de 511 se repitió 1930 veces mientras que para el valor de 512 se realizó un conteo de 1693 y para el valor de 510 se tuvieron 1144 datos, el conteo de valores es cercano para estas muestras lo que ocacionaría cierto ruido en las mediciones finales, por lo que se decide utilizár un filtro basico de promedio para mejorar el comportamiento. Se lo implementó con el siguiente código:

```
const int GSR=A0; //Se define el sensor GSR en el puerto analógico A0
int sensorValue=0; //Se inicializan las variables
int gsr_promedio=0;

void setup(){
  Serial.begin(9600); //Se establece la velocidad de transmisión de datos
en 9600 baudios
}

void loop(){

  long sum=0;
  for(int i=0;i<10;i++)           //Promedio de 10 valores muestreados
    {
      sensorValue=analogRead(GSR); //Se lee la señal del sensor
      sum += sensorValue;
      delay(10);
    }
  gsr_promedio = sum/10; //Cálculo del promedio
  Serial.println(gsr_promedio); //Imprime el valor del sensor en el
puerto serial

}
```

Los resultados obtenidos al aplicar el filtro fueron los siguientes:

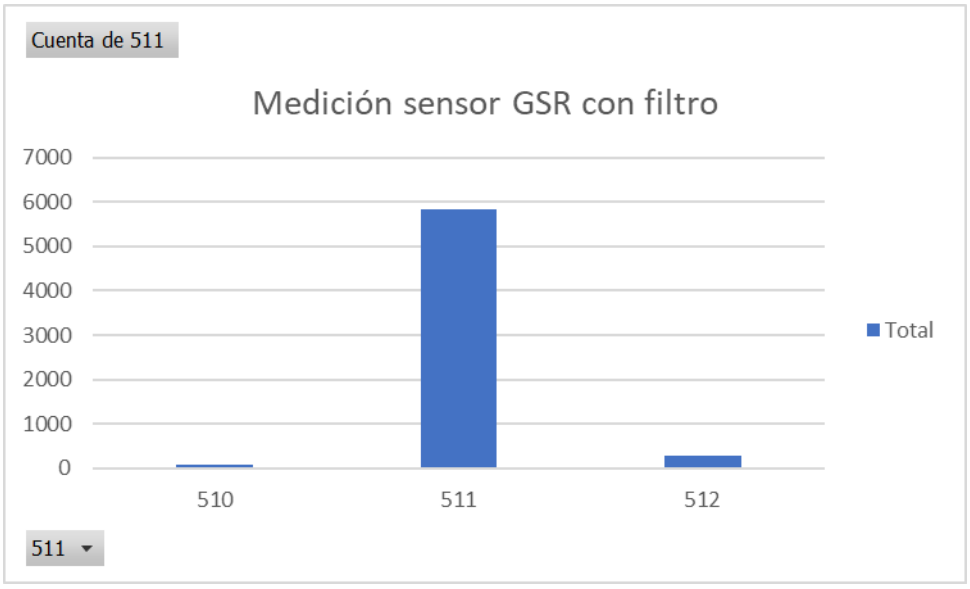

Figura 25. Mediciones del sensor GSR con filtro de promedio

En este caso se obtuvieron un total de 6183 muestras de las cuales 5830 dieron como resultado un valor de 511. Se puede observar que el filtro corrigió el problema de que las muestras estén dispersas entre varios valores.

4.1.3. Pruebas KY-039

Para las pruebas del sensor KY-039 se utilizó el mismo código que para el sensor GSR sin filtro, la única diferencia es la definición del puerto correspondiente que para este caso corresponde al pin A3. Se obtuvieron los siguientes resultados:

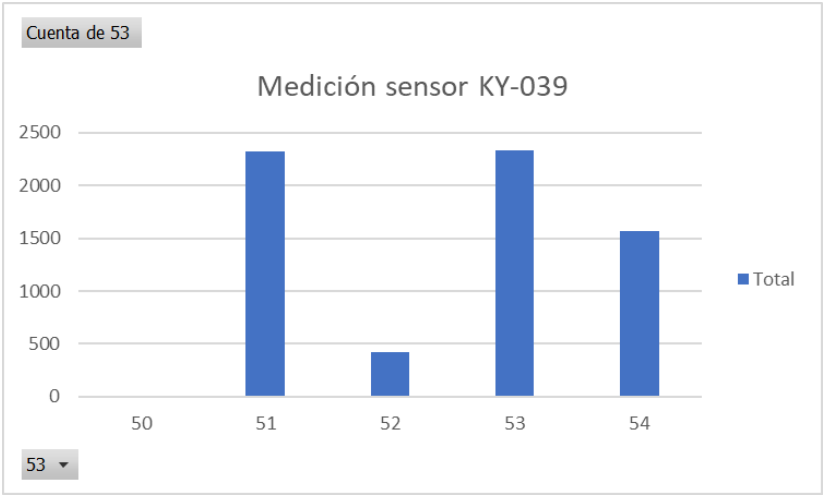

Figura 26. Mediciones del sensor KY-039

Al igual que para el sensor anterior vemos como los datos están dispersos entre varios valores. Para las mediciones se tomaron 6646 muestras y en los valores que más se repitieron fueron el 51, 53 y 54 con 2324, 2330 y 1566 conteos respectivamente. Por esta razón nuevamente se considera necesario aplicar un filtro de promedio. El código es igual al del filtro del sensor GSR, se obtuvieron los siguientes resultados:

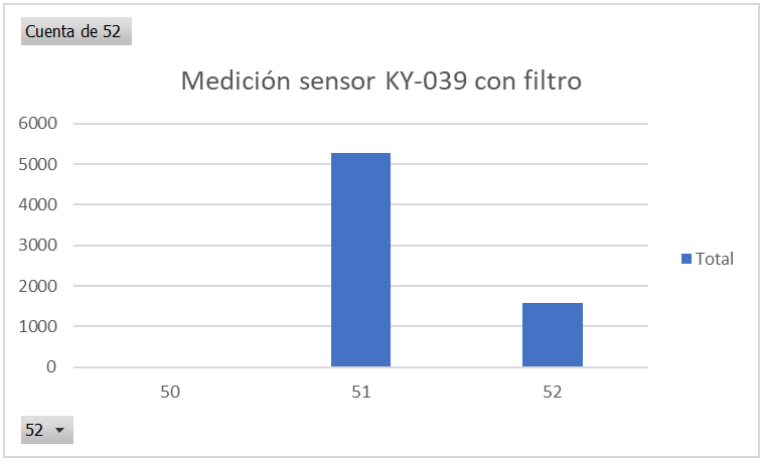

Figura 27. Mediciones del sensor KY-039 con filtro

4.1.4. Pruebas KY-038

Para las pruebas con este sensor se debe regular con el potenciómetro de la placa la sensibilidad hasta tener valores de medición entre 500 a 550. El código utilizado es igual al de los sensores anteriores con la diferencia de que este sensor se encuentra conectado en el pin A4 de la placa Arduino. Los resultados obtenidos se muestran en la siguiente figura:

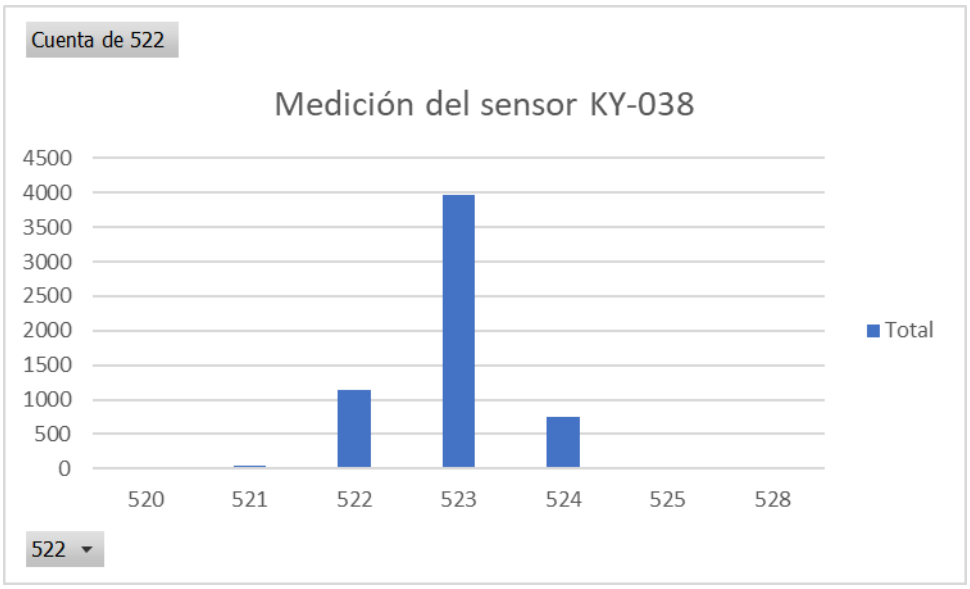

Figura 28. Mediciones del sensor KY-038

Para estas pruebas se tomaron 5927 muestras de las cuales 3972 dieron un resultado de 523. Si bien los resultados se encuentran dispersos entre los valores mostrados en la gráfica se decide no aplicar un filtro a este sensor debido a que se deben registrar sonidos que pueden ocurrir en un instante muy pequeño de tiempo, al aplicar un promedio estos cambios serían más difíciles de detectar.

4.1.5. Pruebas KY-023

Para el módulo del sensor Joystick KY-023 se utilizó el siguiente código para determinar el funcionamiento correcto del módulo:

```
const int KY23Y=A1; //Se define el sensor KY-023 en el puerto analógico
A1
const int KY23X=A2; //Se define el sensor KY-023 en el puerto analógico
A2
int sensorValueY=0; //Se inicializan las variables
int sensorValueX=0;

void setup(){
  Serial.begin(9600); //Se establece la velocidad de transmisión de datos
en 9600 baudios
}

void loop(){

  sensorValueY=analogRead(KY23Y); //Se lee la señal del sensor para el
eje Y
  sensorValueX=analogRead(KY23X); //Se lee la señal del sensor para el
eje X
  Serial.println(sensorValueY); //Imprime el valor del sensor en el
puerto serial
  Serial.println(sensorValueX); //Imprime el valor del sensor en el
puerto serial
  delay(100); //Se espera un tiempo de 100ms para la siguiente medida

}
```

Los resultados obtenidos fueron los siguientes:

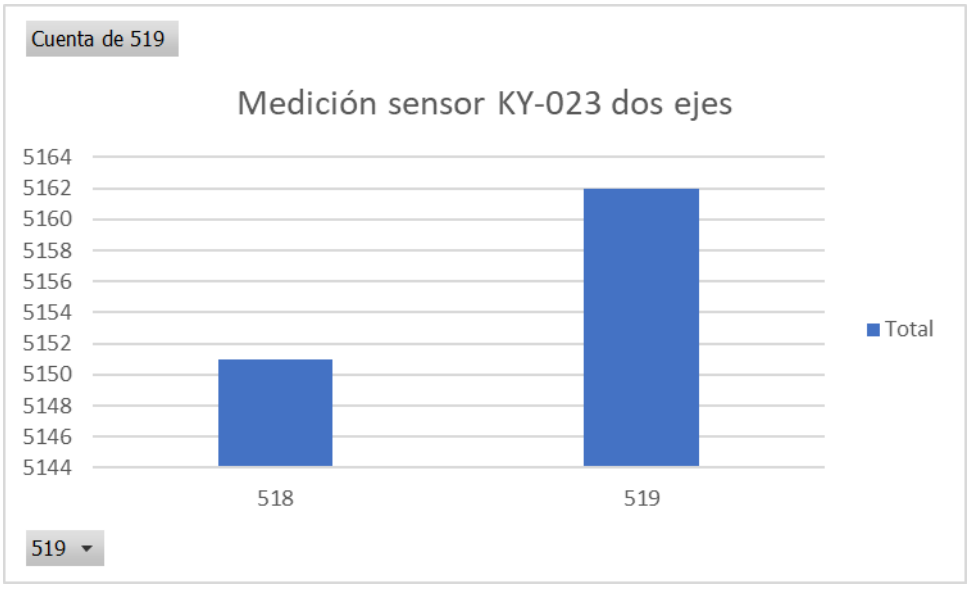

Figura 29. Mediciones del sensor KY-029 en dos ejes

Para este sensor se tomaron 10313 muestras, 5151 fueron del eje X y 5162 del eje Y. Con las pruebas podemos observar que no existen interferencias en las lecturas de los ejes del mismo, lo que comprueba el correcto funcionamiento del modulo KY-023.

4.1.6. Pruebas KY-024

Para probar este sensor se desarrolló el siguiente código de Arduino:

```
const int sensorPin = 7;  // Define el pin al que está conectado el
sensor KY-024
int lastSensorState = LOW; // Variable para rastrear el estado anterior

void setup() {
  pinMode(sensorPin, INPUT); // Configura el pin como entrada
  Serial.begin(9600); // Inicializa la comunicación serial
}

void loop() {
  int sensorState = digitalRead(sensorPin); // Lee el valor actual del
sensor (HIGH o LOW)

  // Compara el estado actual con el estado anterior
  if (sensorState != lastSensorState) {
    if (sensorState == LOW) {
      Serial.println("Campo magnetico detectado (de bajo a alto).");
    } else {
      Serial.println("Campo magnetico no detectado (de alto a bajo).");
    }
    lastSensorState = sensorState; // Actualiza el estado anterior
  }

  delay(100); // Pequeña pausa para estabilidad
}
```

Las pruebas consistieron en acercar un imán, la máxima distancia de detección para el sensor es de 1cm, aunque puede variar dependiendo del campo magnético utilizado, para determinar este valor de distancia se utilizaron 3 imanes diferentes que se muestran a continuación:

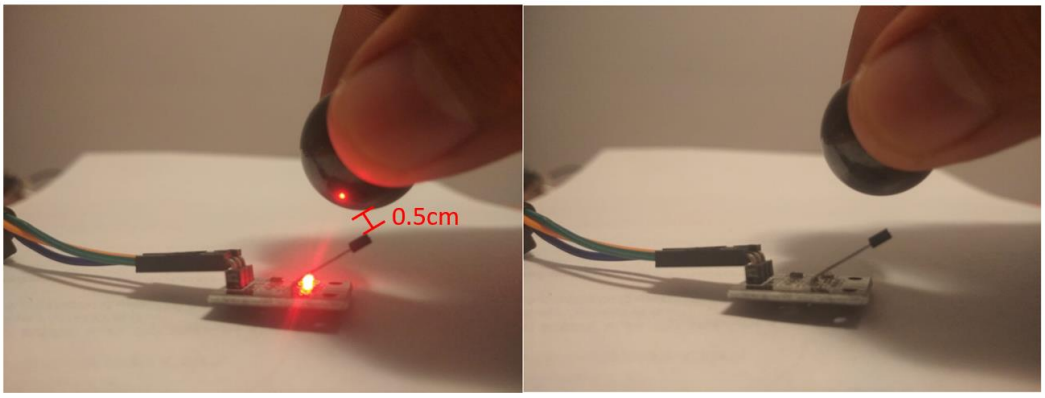

Figura 30. Primera prueba sensor hall con imán

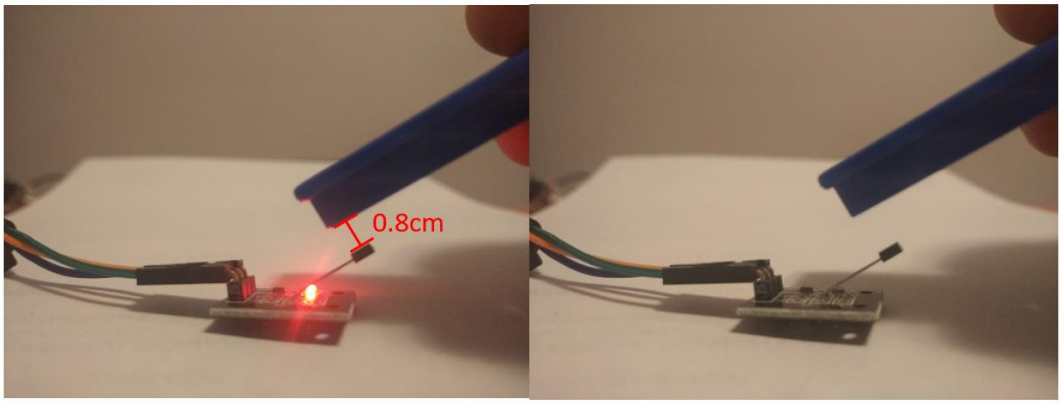

Figura 31. Segunda prueba sensor hall con imán

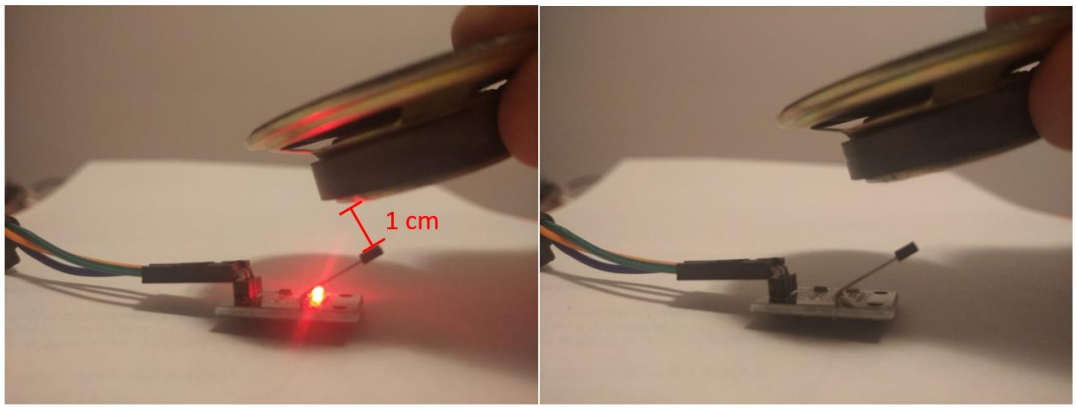

Figura 32. Tercera prueba sensor hall con imán

Para las pruebas se utilizó el tercer imán, el proceso fue acercar y alejar el imán 100 veces y ver la cantidad de aciertos, tambíen se realizó el proceso con diferentes velocidad de movimiento para determinar la sensibilidad del módulo, con ello se puede conocer el funcionamiento del sensor. Los resultados de las pruebas fueron los siguientes:

Tabla 14. Resultados pruebas con sensor KY-024

	Velocidad baja	Velocidad media	Velocidad alta
Número de aciertos	100	95	32
Número de errores	0	5	68

4.1.7. Pruebas KY-032

El código utilizado para probar este sensor es similar al utilizado para el KY-024, la diferencia el pin que se utilizó en la placa Arduino es el número 4 para entradas digitales. De igual manera se hicieron pruebas similares para determinar la distancia máxima de detección que se estableció en 8cm utilizando una pieza de 3.5x5.5x2 cm, recordando que el módulo KY-032 es un sensor emisor/receptor de luz infrarroja por lo que ciertos materiales que no reflejan luz no son detectados y ciertos materiales que reflejan mejor la luz son detectados con mayor facilidad y a mayor distancia. Para las pruebas se utilizaron diferentes objetos con variaciones en su tamaño y material, las pruebas se muestran en las siguientes figuras:

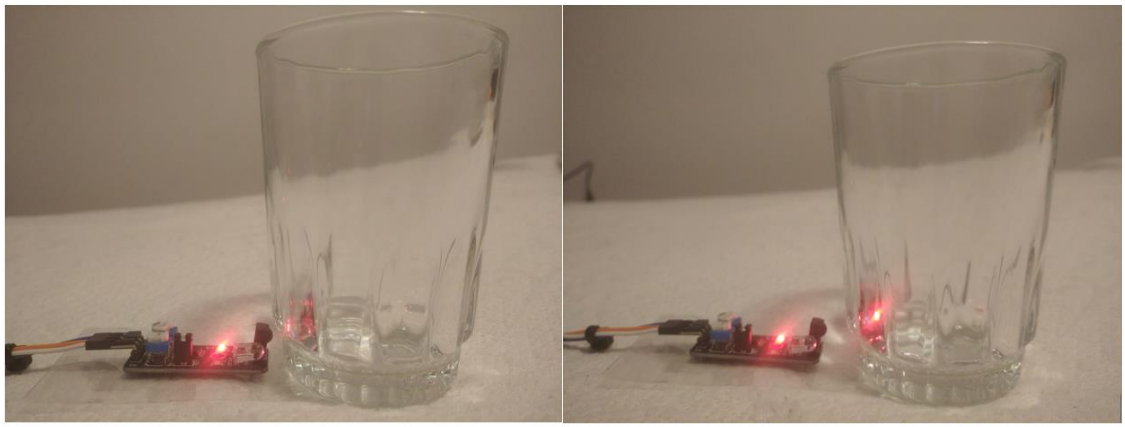

Figura 33. Primer objeto de pruebas KY-032

Con el cristal el sensor no funciona adecuadamente a pesar de que el objeto está muy cercano al sensor.

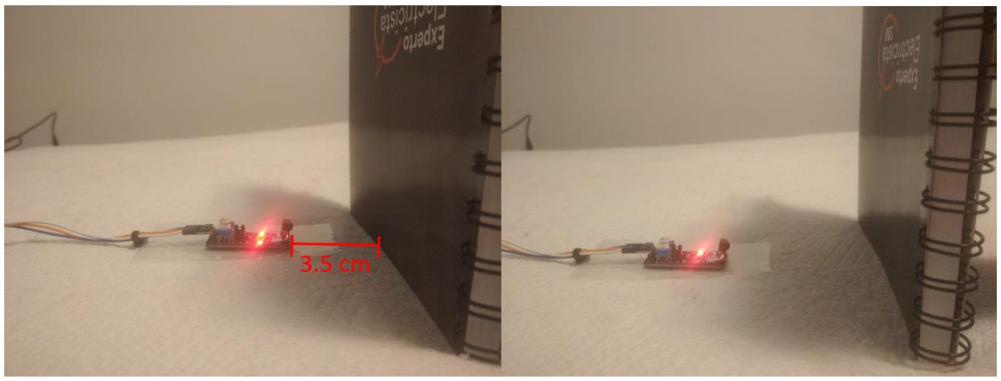

Figura 34. Segundo objeto de pruebas KY-032

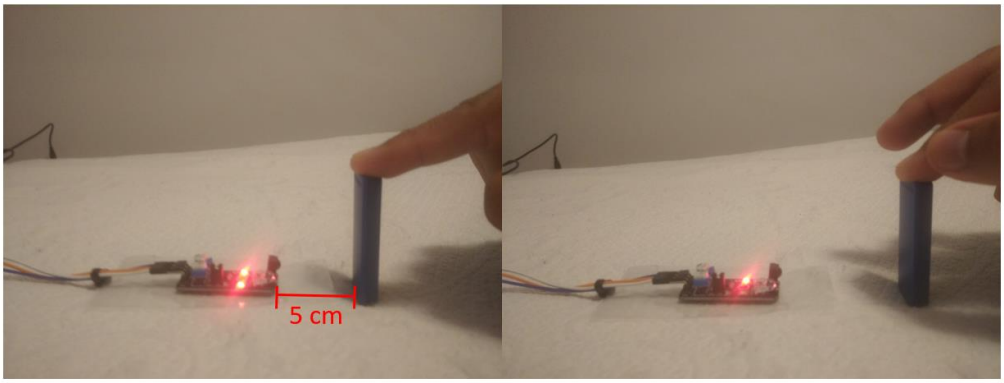

Figura 35. Tercer objeto de pruebas KY-032

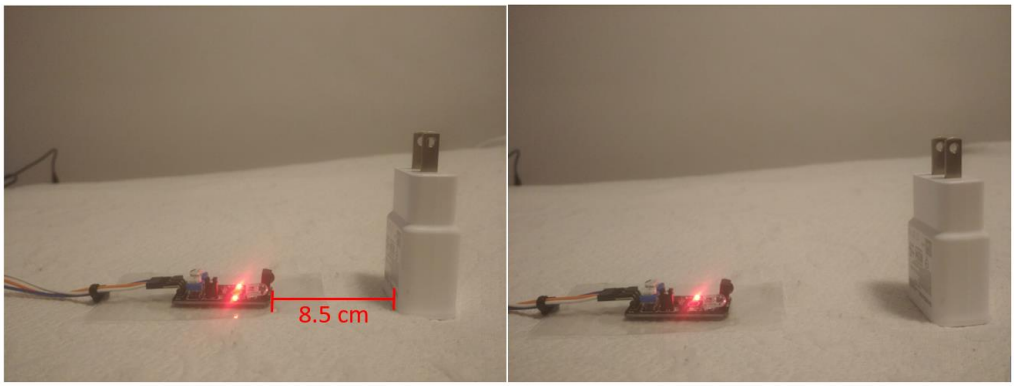

Figura 36. Cuarto objeto de pruebas KY-032

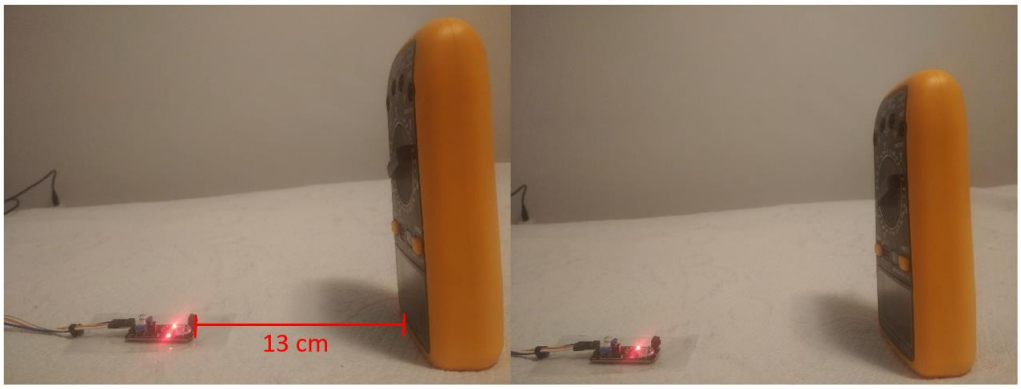

Figura 37. Quinto objeto de pruebas KY-032

Para las pruebas se realizó un proceso similar al del sensor KY-024 con el fin de determinar su correcto funcionamiento. Se utilizó solo el cuarto objeto para las pruebas, los resultados fueron los siguientes:

Tabla 15. Resultados pruebas con sensor KY-032

	Velocidad baja	Velocidad media	Velocidad alta
Número de aciertos	99	94	93
Número de errores	1	6	7

4.1.8. Pruebas KY-026

El código utilizado para probar este sensor es similar al utilizado para el KY-024, la diferencia es el pin que se utilizó en la placa Arduino, que es el número 6 para entradas digitales. Para este caso lo único que se midió fue la distancia de detección del sensor que fue de un máximo de 50 cm. Las pruebas se realizaron con la llama de una vela como se muestra en la siguiente figura:

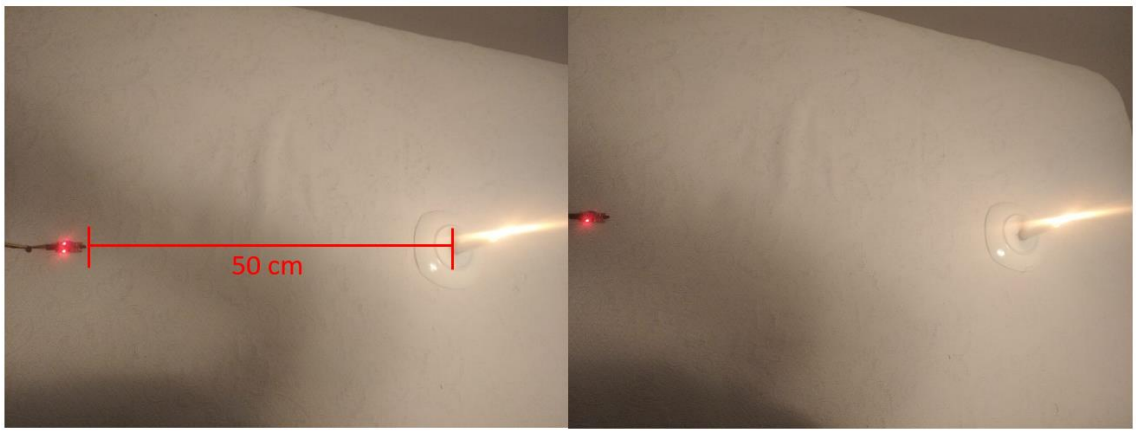

Figura 38. Prueba del sensor KY-026

Para este sensor solo se consideró necesario probar la detección de la llama y la cantidad de aciertos del sensor a una velocidad media de aparición del estimulo, debido a las caracteristicas físicas del fuego. Los resultados fueron los siguientes:

Tabla 16. Resultados pruebas con sensor KY-026

	Velocidad media
Número de aciertos	95
Número de errores	5

4.1.9. Pruebas KY-031

Para las pruebas de este sensor se utilizó el siguiente código:

```
const int sensorPin = 5;  // Define el pin al que está conectado el
sensor KY-031
int Led = 13;
int val;


void setup() {
  pinMode(sensorPin, INPUT); // Configura el pin como entrada
  pinMode(Led, OUTPUT); // Configuro el pin como salida
  Serial.begin(9600); // Inicializa la comunicación serial
}

void loop() {
  val = digitalRead(sensorPin);

  if(val == HIGH){
    Serial.println("Espera"); // Escritura en el puerto serial cuando el
sensor no está activo
    digitalWrite(Led, LOW);
  }else{
    Serial.println("Golpe detectado"); //Si se detecta la activación del
sensor se envía este mensaje al puerto serial
    digitalWrite(Led, HIGH);
  }
}
```

Posterior a ello se activo el sensor mediante golpes con diferentes fuerzas, para medir su respuesta. Los resultados obtenidos fueron los siguientes:

Tabla 17. Resultados pruebas con sensor KY-031

	Fuerza baja	Fuerza media	Fuerza alta
Número de aciertos	2	10	25
Número de errores	98	90	75

Al probar este sensor se puede observar que el impacto o golpe que debe recibir para activarse debe ser bastante fuerte, lo que resulta poco util para la aplicación deseada. Por esta razón se decide probar con otro sensor que realiza una función similar que es el sensor de impacto KY-002, la diferencia es su construcción interna que tiene un mecanismo de resorte para cerrar el circuito.

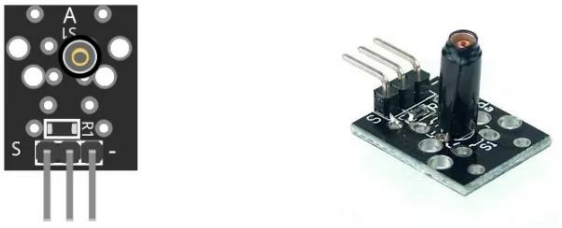

Figura 39. Sensor KY-002

El sensor se conectó en el mismo pin 5 de la placa Arduino y se utilizó el mismo código. Se realizaron las pruebas iguales que para el anterior sensor y los resultados fueron los siguientes:

Tabla 18. Resultados pruebas con sensor KY-002

	Fuerza baja	Fuerza media	Fuerza alta
Número de aciertos	74	85	92
Número de errores	26	15	8

Los resultados son mejores para este sensor, por lo que se decide cambiar el sensor KY-031 por el sensor KY-002.

4.1.10. Pruebas KY-040

Para el último sensor KY-040. El código utilizado se centró en determinar la posición del encoder rotativo y la dirección de su movimiento, además se debe recordar que el sensor cuenta con un pulsador que se programó para resetear el estado del encoder, se colocó un filtro para no detectar pulsaciones repetidas. El código se muestra a continuación:

```
#define salidaA 2
#define salidaB 3
#define boton 9
```

```
int contador = 0;
int estadoA;
int estadoPrevioA;
bool botonPresionado = false;
unsigned long debounceTiempo = 0;
unsigned long debounceDelay = 50; // Ajusta este valor según sea
necesario

void setup() {
  pinMode(salidaA, INPUT);
  pinMode(salidaB, INPUT);
  pinMode(boton, INPUT_PULLUP);

  Serial.begin(9600);

  estadoPrevioA = digitalRead(salidaA);
}

void loop() {
  estadoA = digitalRead(salidaA);

  if (estadoA != estadoPrevioA) {
    if (digitalRead(salidaB) != estadoA) {
      contador++;
    } else {
      contador--;
    }
    Serial.print("Posicion: ");
    Serial.println(contador);
  }

  estadoPrevioA = estadoA;

  unsigned long tiempoActual = millis();

  if (!digitalRead(boton)) {
    if (!botonPresionado && (tiempoActual - debounceTiempo) >
debounceDelay) {
      Serial.println("Boton pulsado: Contador a 0");
      contador = 0;
      botonPresionado = true;
    }
  } else {
    botonPresionado = false;
    debounceTiempo = tiempoActual;
  }
}
```

Los resultados se pueden apreciar mediante los datos del puerto serial:

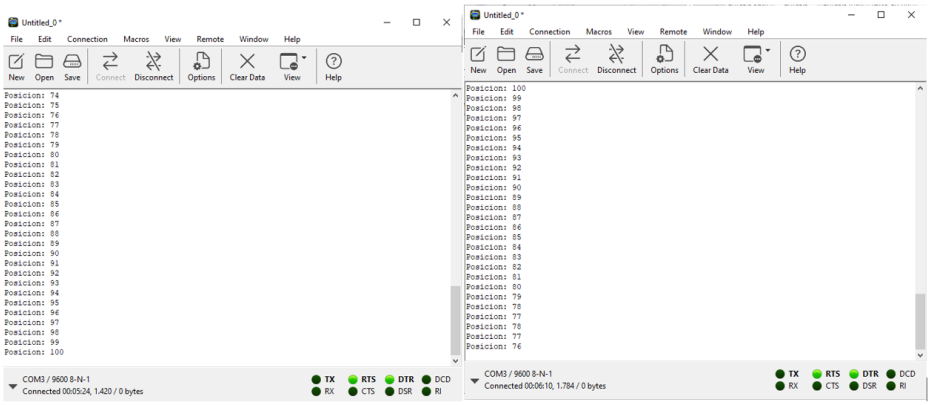

Figura 40. Resultados obtenidos del sensor KY-040

4.2. Implementación del entorno de Unity

El entorno desarrollado en Unity simula un espacio industrial donde se pueden encontrar diferentes áreas en las que el usuario podrá interactuar con los elementos mediante la activación de los sensores conectados al entorno. Para el desarrollo se utilizaron modelos gratis 3D de la plataforma Sketchfab, algunos fueron modificados dependiendo de las necesidades del sistema. Los modelos que tiene interacción directa con los sensores son los siguientes:

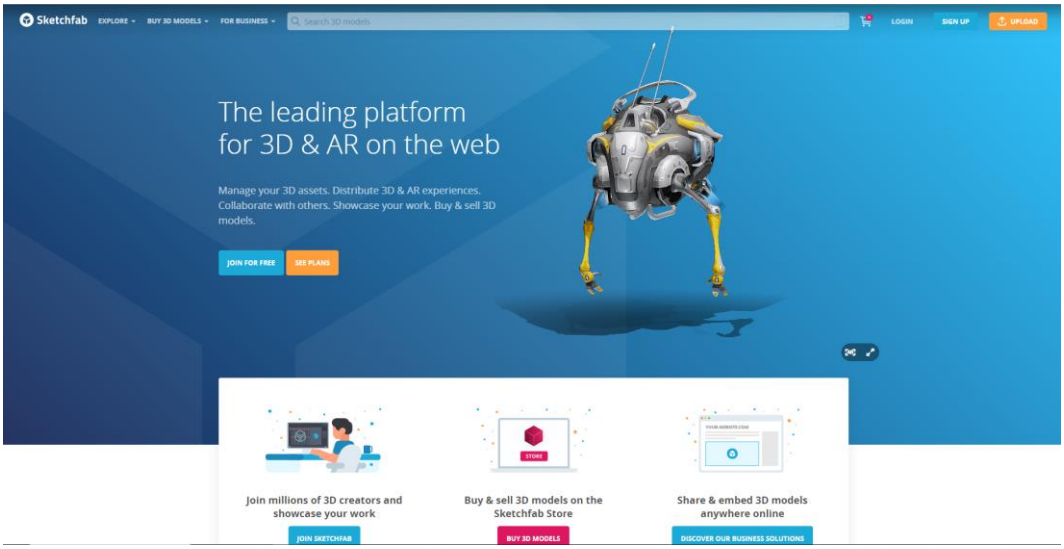

Figura 41. Plataforma de sketchfab

Figura 42. Modelo 3D brazo robótico invertido

En la Figura 42 se observa un brazo robótico invertido que interactúa con el sensor KY-040, cuando el encoder rota, también se mueven las articulaciones del robot; además, cuando se presiona el pulsador, se cambia la selección de la articulación.

Figura 43. Modelo 3D brazo robótico con herramienta

El sensor KY-023 controla el brazo robótico, el módulo joystick tiene dos ejes que permite mover dos articulaciones al mismo tiempo. Igual que el módulo anterior, también cuenta con un pulsador que controla el cambio entre articulaciones. Para el caso de los brazos robóticos es necesario mencionar que

ambos tienen 6 grados de libertad y sus modelos fueron modificados para organizar la jerarquía de los movimientos de sus articulaciones.

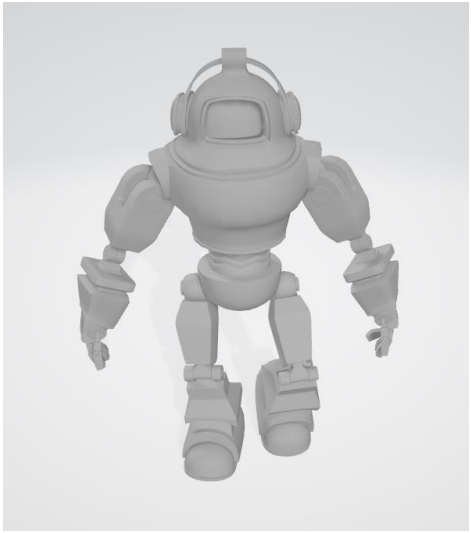

Figura 44. Modelo 3D robot móvil

El robot móvil está en un espacio confinado y su función es esquivar obstáculos, los obstáculos son creados de forma automática en el entorno mediante la activación del sensor KY-032. El robot está programado con la librería de NavMesh de Unity que modifica la trayectoria al detectar un objeto de colisión.

Figura 45. Modelo 3D luces activas

En el espacio existe un área donde se pueden encender una serie de luces mediante la lectura del sensor KY-038, dependiendo del nivel del sonido se

encienden las lámparas, entre más alto sea el sonido mayor la cantidad de luces activadas.

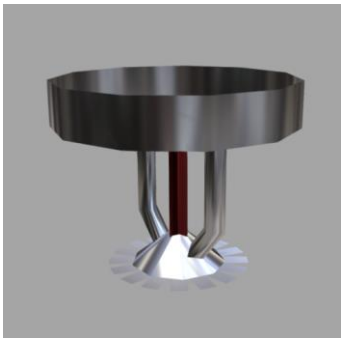

Figura 46. Modelo 3D rociador automático de agua

En el techo de la sección principal del entorno se encuentran los rociadores automáticos de agua que se activarán cuando el sensor KY-026 detecte una llama. En el espacio de Unity esto se ve reflejado con la programación del objeto que desencadena una animación cuando el sensor envíe la señal necesaria.

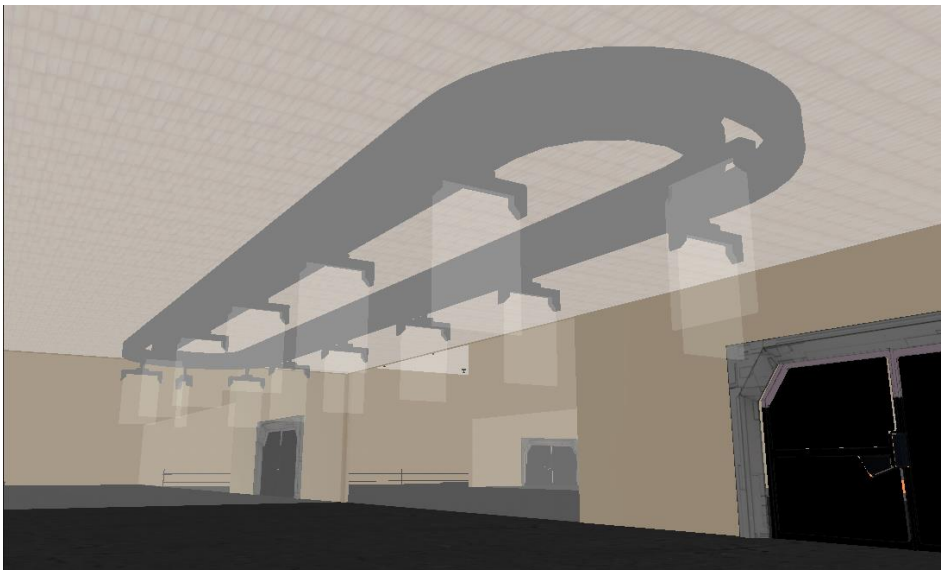

Figura 47. Modelo 3D de la cinta transportadora de vidrios en el entorno

El modelo de la Figura 47. Muestra el espacio donde se encuentra la banda transportadora de vidrios que interactúa con el sensor KY-002. En el entorno se

programó el sistema para que cuando se detecte un golpe del sensor uno de los vidrios active la animación de ruptura, los vidrios se irán rompiendo en orden, si todos los vidrios se rompen o al pasar un tiempo específico de la última ruptura, todos los vidrios se restaurarán a su estado inicial.

Figura 48. Modelo 3D estación de mecanizado

La estación de mecanizado cuenta con dos animaciones que se implementaron para diferentes piezas. La animación predeterminada muestra como el robot toma una de las piezas y la mueve a la máquina de mecanizado. Cuando el usuario acerque un imán al sensor KY-032, el sistema cambiará a la segunda animación donde se tomará la segunda pieza y se la colocará en la máquina de mecanizado correspondiente. El sistema está programado para que mientras exista una pieza dentro del proceso otra no pueda interferir en el mismo, el robot esperará hasta que termine el proceso para tomar una nueva pieza. Así mismo, se colocó una condición para no recibir señales adicionales del sensor mientras ya exista una pieza generada por el mismo.

Figura 49. Punto de vista del usuario en el entorno

El visor del usuario en Unity le permite observar el estado de la temperatura del entorno, Para ello se programó los elementos de texto para que varíen en función de las lecturas del sensor KY-015.

Figura 50. Modelo de pantalla 3D en entorno. Zona de videos con estímulos

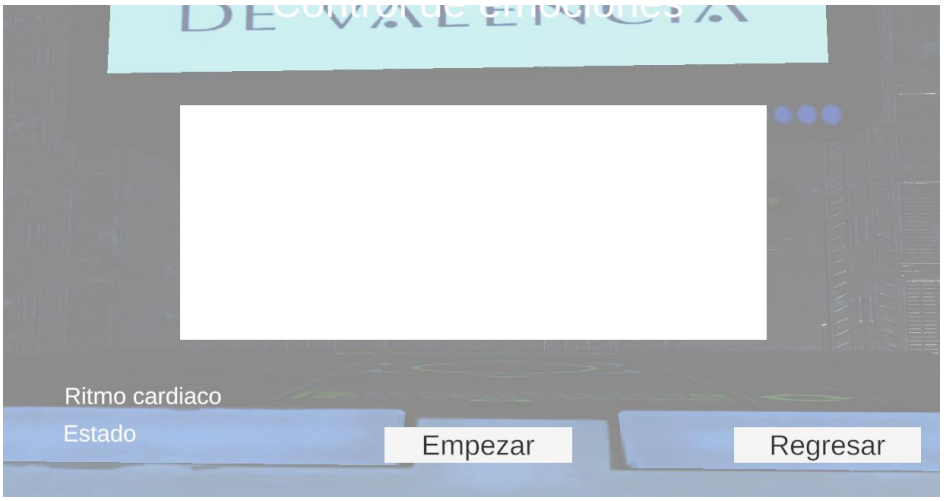

Figura 51. Pantalla de visualización de videos, medición del ritmo cardiaco y conductividad de la piel

En esta zona del entorno se tiene una pantalla que se puede activar presionando la tecla “e” y donde se mostrarán los datos de los sensores del ritmo cardiaco (ky-039) y GSR, adicionalmente se mostrará un video al usuario con el fin de variar estas lecturas.

Figura 52. Modelo de pantalla 3D en entorno. Estación de control.

En una zona diferente se tiene otra pantalla que sirve de estación de control, de igual manera se activa presionando la tecla “e” cerca de la misma y mostrará el estado de todos los sensores del entorno.

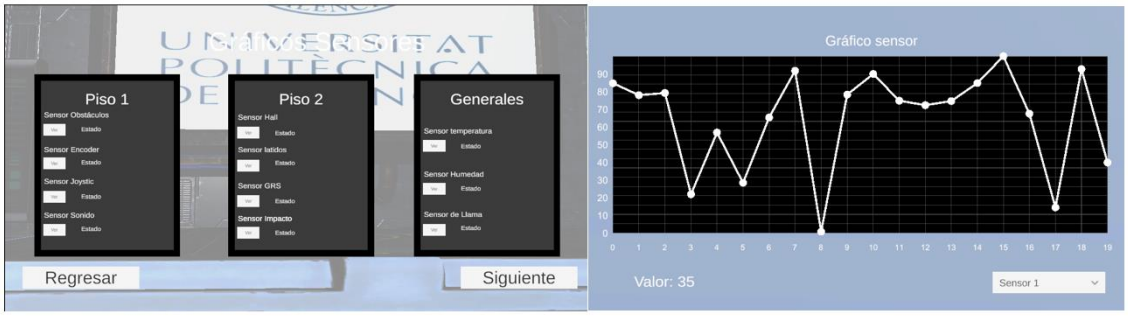

Figura 53. Pantalla de control general

4.3. Integración de Unity con Arduino

Fue necesario desarrollar un código de Arduino que integre todos los sensores antes probados y que envíe los datos mediante el puerto serial de manera que Unity pueda adquirir estos datos y procesarlos en las tareas especificas del entorno. Para esta aplicación, se necesitaba una adquisición de datos adecuada para el sistema, sobre todo por la cantidad de información que se quería manejar y enviar por el puerto serial, por esta razón se buscó la forma de implementar programación concurrente.

Arduino tiene una arquitectura basada en un microcontrolador y no cuenta por defecto con recursos para la programación concurrente pero existen librerías desarrolladas por la comunidad que permiten ejecutar tareas simultaneas. Estas librerías permiten simular y ejecutar "hilos" con tareas asignadas que se pueden ordenar y mejorar el rendimiento de un programa, aprovechando los recursos del sistema. Se debe recordar que el lenguaje que maneja arduino tiene bases de C por lo que la sintaxis de estas librerías son similares a las de threads. La librería especifica que se utilizó fue la de FreeRTOS.

Para las pruebas se utilizó el siguiente código que se ha resumido a las secciones principales por su extensión:

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>
#include <DHT.h>

#define DHTPIN 8
#define DHTTYPE DHT11

//Declaración de mutex
SemaphoreHandle_t serialMutex;
SemaphoreHandle_t dhtMutex;

//Definición de variables
int GSRValue = 0;
int joystickXValue = 0;
int joystickYValue = 0;

…… Resto de variable definidas

// Definición de pines analógicos
const int sensorGSR = A0;
…… Resto de pines usados

// Definición de pines digitales
const int encoderA = 2;
……Resto de pines digitales definidos

void setup() {
  Serial.begin(9600);

  // Crear un mutex para manejo del puerto serial
  serialMutex = xSemaphoreCreateMutex();
  dhtMutex = xSemaphoreCreateMutex();
  if (serialMutex != NULL) {
    Serial.println("Serial mutex created.");
  }
  if (dhtMutex != NULL) {
    Serial.println("Mutex for DHT created.");
  }

  // Configurar pines como entrada
  //Analógicos
  pinMode(sensorGSR, INPUT);
  ……Resto de configuración de pines
```

```
  // Crear tareas para leer cada uno de los sensores
  //Analógicos
  xTaskCreate(taskSensorGSR, "SensorGSRTask", 128, NULL, 1, NULL);
  ……Resto de creación de tareas para la lectura de sensores

}
```

La función xTaskCreate se utiliza para crear una tarea en FreeRTOS y se configura con un nombre, un tamaño de pila, una función que implementa la tarea, una prioridad y, opcionalmente, un puntero a datos adicionales y un puntero para almacenar el identificador de la tarea creada. Esta tarea se ejecutará de manera independiente en el sistema operativo en tiempo real. De la misma manera se utilizó la librería semphr que permite el uso de mutex y semáforos que sirven para bloquear y desbloquear el canal de comunicación y evitar en lo posible la perdida de datos. Se muestra a continuación el código para la definición de tareas para los sensores digitales y analógicos:

```
void loop() {
  // El loop no se utiliza en FreeRTOS
}

//////////////////////Funciones Sensores
analógicos//////////////////////////////
void taskSensorGSR(void *pvParameters) {
  (void)pvParameters;
  Serial.println("Lanzo GSR");
  while (1) {

    // Tomar el mutex para acceder al puerto serial
    if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
      // Leer valores del sensor
      GSRValue = analogRead(sensorGSR);
      Serial.print("GSR:");
      Serial.println(GSRValue);
      xSemaphoreGive(serialMutex); // Libero el mutex del puerto serial
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer sensor cada 10 ms

  }
}
…… Resto de tareas analógicas
```

```
//////////////////////Funciones Sensores
digitales//////////////////////////////

void taskKy026(void *pvParameters) {
  (void)pvParameters;
  Serial.println("Lanzo Ky026");
  while (1) {
    // Tomar el mutex para acceder al puerto serial
    if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
      // Leer el estado del sensor digital
      ky026Value = digitalRead(ky026);
      Serial.print("ky026: ");
      Serial.println(ky026Value);
      xSemaphoreGive(serialMutex);
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

……Resto de tareas digitales
```

La tarea de lectura del encoder tuvo que ser modificada para poder enviar los datos correctamente por el puerto serial sin bloquear el canal de comunicación con el mutex. Se lo hizo con el siguiente código:

```
void taskEncoder(void *pvParameters) {
  (void)pvParameters;
  Serial.println("Lanzo Encoder");

  while (1) {
    int estadoActualA = digitalRead(encoderA);
    if (estadoActualA != estadoAnteriorA) {
      if (digitalRead(encoderB) != estadoActualA) {
        // Rotación en sentido horario
        posicion++;
      } else {
        // Rotación en sentido antihorario
        posicion--;
      }
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        Serial.print("Posicion: ");
        Serial.println(posicion);
```

```
        xSemaphoreGive(serialMutex);
      }
    }
    estadoAnteriorA = estadoActualA;
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}
```

El sensor de temperatura y humedad KY-015 necesita su librería especial DHT con sus funciones específicas para leer y enviar los datos de temperatura y humedad como se muestra en el código de su tarea:

```
void taskKy015(void *pvParameters) {
  (void)pvParameters;
  Serial.println("Lanzo Ky015");
  while (1) {
    if (xSemaphoreTake(dhtMutex, portMAX_DELAY) == pdTRUE) {
      float humidity = dht.readHumidity();
      float temperature = dht.readTemperature();
      Serial.print("Humidity: ");
      Serial.print(humidity);
      Serial.print("%  Temperature: ");
      Serial.print(temperature);
      Serial.println("°C");
      xSemaphoreGive(dhtMutex);
    }
    vTaskDelay(1000 / portTICK_PERIOD_MS); // Leer cada 1 segundos
  }
}
```

Al probar este código con la placa Arduino UNO y revisar el puerto serial, se pudo observar que no se transmitía la información, por esta razón se decidió implementar el código por partes para detectar el error. Al realizar las pruebas se determinó que la placa Arduino UNO solo soportaba la lectura de 5 sensores de manera simultánea con la ejecución de tareas, cuando se agregaba un sensor adicional el sistema dejaba de transmitir la información. Por este motivo se decidió probar con la placa Arduino MEGA y examinar si el problema estaba relacionado con limitaciones del hardware. Al probar la integración del código

con la placa Arduino MEGA se tuvieron los siguientes resultados en el puerto serial:

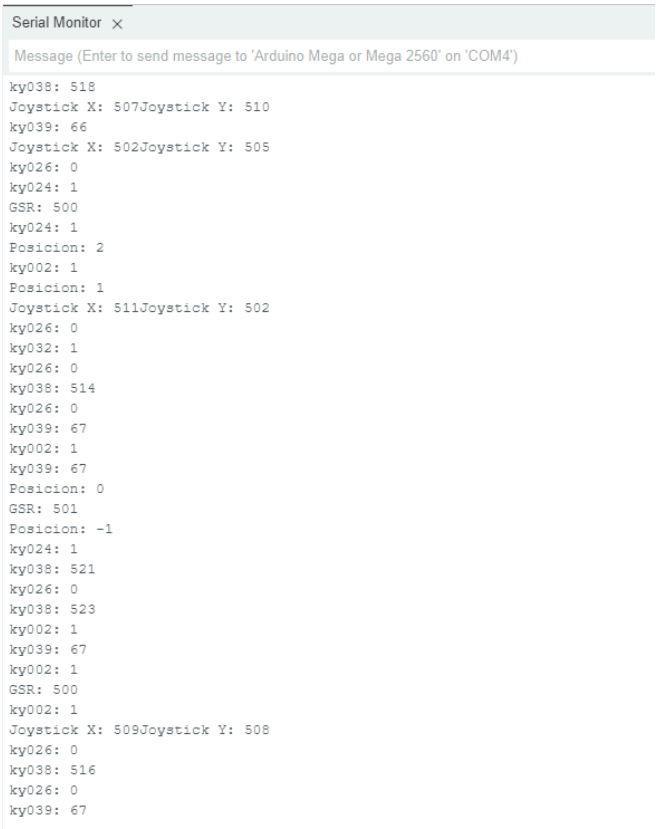

Figura 54. Lectura del puerto serial para el código de Arduino

En esta imagen podemos observar la lectura de 9 sensores. El dispositivo KY-015 no envía la señal de manera adecuada, esto debido a que es un sensor digital que envía información serializada al Arduino y cuya librería de control causa interferencias con la programación concurrente y con la ejecución de la tareas. Por este motivo se decide utilizar otro módulo sensor de temperatura que es el KY-013 que es un dispositivo que funciona en base a un termistor.

Con estas pruebas se decide reemplazar la placa Arduino UNO por la placa Arduino Mega y el sensor KY-015 por el KY-013, el primero por motivos de limitaciones en el hardware especificamente de la memoria SRAM, y el segundo por incompatibilidad con el sistema de tareas.

Tabla 19. Comparativa de características de Arduino UNO y de Arduino MEGA

PARÁMETRO	ARDUINO UNO	ARDUINO MEGA 2560
Microcontrolador	ATmega328	ATmega2560
Memoria FLASH	32 KB (0.5 KB reservados para el bootloader)	256 KB (8KB reservados para el bootloader)
Memoria SRAM	2 KB	8 KB
Memoria EEPROM	1 KB	4 KB
Entradas/Salidas Digitales	14	54
Salidas (PWM)	6	15
Entradas Analógicas	6	16

Figura 55. Sensor de temperatura KY-013

Los cambios que se realizaron en el código fueron agregar una tarea adicional para la lectura del sensor KY-013. Y el cálculo necesario para obtener el valor de la temperatura del termistor que se debe realizar mediante la ecuación de Steinhart-Hart. El código se muestra a continuación:

```
//Definición de variables
……
float R1 = 10000; // Valor de resistencia en el módulo
float logR2, R2, T;
float c1 = 0.001129148, c2 = 0.000234125, c3 = 0.0000000876741;
//coeficientes del termistor por steinhart-hart
……
// Definición de pines analógicos
……
const int ky013 = A5;
……
void taskKy013(void *pvParameters) {
  (void)pvParameters;
  while (1) {
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        ky013Value = analogRead(ky013);
        R2 = R1 / ((1023.0 / (float)ky013Value) - 1.0); //cálculo del
valor del termistor
        logR2 = log(R2);
```

```
        T = (1.0 / (c1 + (c2*logR2) + (c3*logR2*logR2*logR2))); //
Temperatura en Kelvin
        T = (T - 273.15)-5; //convertir Kelvin a Celcius más corrección
de temperatura -5 grados
        // Ahora, convierte la temperatura a un entero
        int temperaturaEntera = (int)T;
        Serial.print("Temperatura:");
        Serial.println(temperaturaEntera);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
}
```

Los resultados de la implementación del código fueron los siguientes:

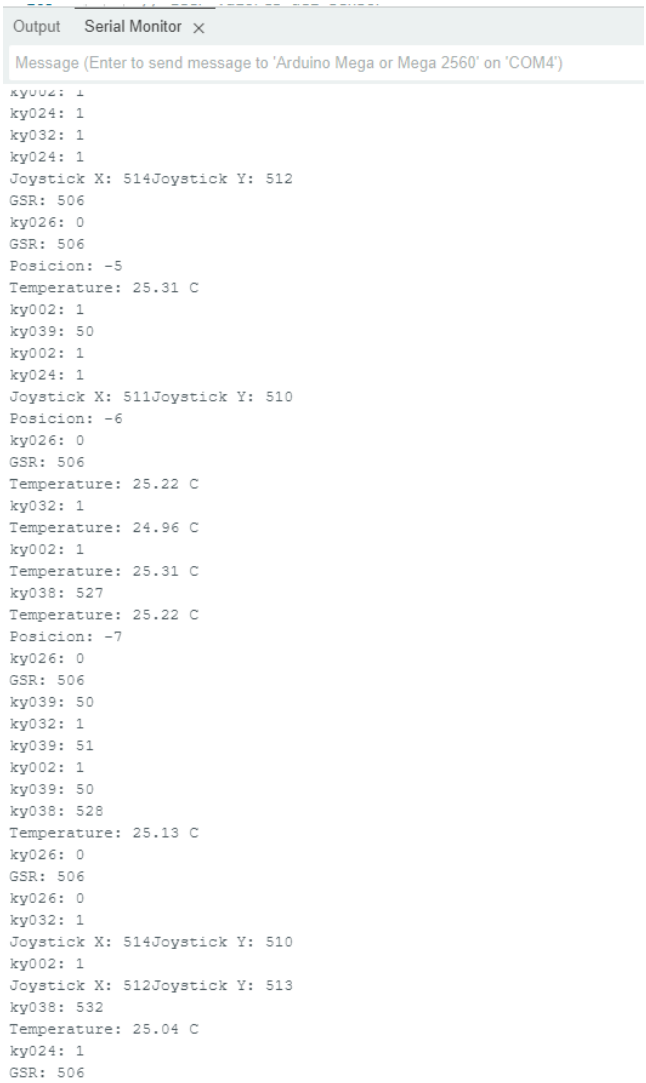

Figura 56. Resultados de la implementación del código modificado

Con las pruebas de este código podemos observar la lectura de los 10 sensores y del envío de datos por el puerto serial.

El siguiente paso fue probar que los datos lleguen correctamente a Unity mediante el puerto serial. Para ello es necesario configurar en "Edit->Project Settings->Player->Configuration->Api Compatibility Level->.NET Framework" esto para poder utilizar la librería necesaria para controlar el uso del puerto serial.

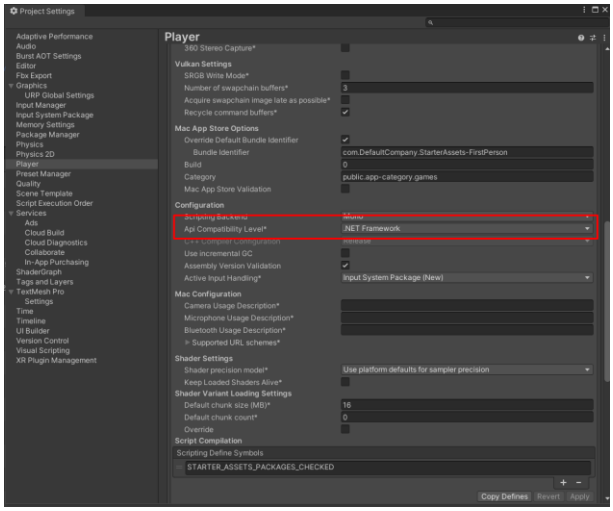

Figura 57. Configuración de Unity para el uso del puerto serial

En el entorno de Unity se creó un script que permita la lectura de los datos, y el código es el siguiente:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO.Ports; //Librería para el uso del puerto serial

public class ControlArduino : MonoBehaviour
{
    public string serialPortName = "COM4"; // Puerto serial Arduino Mega
    public int baudRate = 9600; // Velocidad de baudios de Arduino

    private SerialPort serialPort;

    private void Start()
    {
        //Se crea la comunicación con Unity
        serialPort = new SerialPort(serialPortName, baudRate);
        serialPort.Open();
        StartCoroutine(ReadData());
```

```
    }

    private IEnumerator ReadData()
    {
        while (true)
        {
            try
            {
                string data = serialPort.ReadLine();
                // Recepción de datos
                Debug.Log("Datos de Arduino: " + data);
            }
            catch (System.Exception e)
            {
                Debug.LogWarning("Error al leer los datos: " + e.Message);
            }
            yield return null;
        }
    }

    private void OnDestroy()
    {
        //Finaliza la comunicación y libera el recurso
        if (serialPort != null && serialPort.IsOpen)
        {
            serialPort.Close();
        }
    }
}
```

Los datos recibidos por el puerto serial son mostrados en la consola de Unity como se muestra en la siguiente figura:

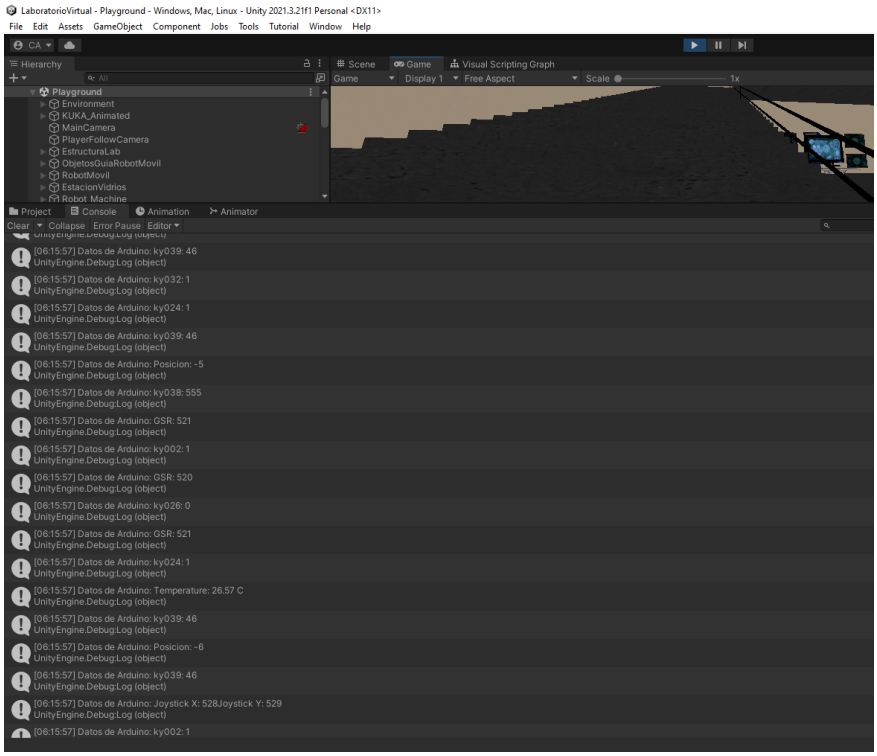

Figura 58. Consola de Unity con datos recibidos

Luego de haber comprobado que la comunicación existe y que se pueden enviar de manera adecuada los datos entre dispositivos, fue necesario ordenar el envío y recepción de datos. Se debe recalcar que no todos los sensores deben estar activos en todas las zonas del entorno y que la complejidad para leer y separar la información del puerto serial si se tienen todos los datos es muy alta. Por ello se implementó un sistema de solicitud de datos por parte de Unity, es decir que, el entorno enviará un mensaje a Arduino solicitando un dato puntual y Arduino enviará la información correspondiente. Para ello se creó una tarea en Arduino que pausa o reanuda la lectura de un sensor dependiendo del mensaje que se reciba por el puerto serial. En Unity se creó un script que maneja las conexiones, y tiene comunicación con el resto de objetos del entorno, cuando uno solicita un dato este gestiona la solicitud, recepción y relocalización del dato en el lugar correspondiente. A continuación, se muestra un extracto del código de Arduino para esta tarea:

```
void taskHandleCommands(void *pvParameters) {
  (void)pvParameters;
  Serial.println("Lanzo CommandHandlerTask");
  while (1) {
    if (Serial.available() > 0) {
      String command = Serial.readStringUntil('\n'); // Leer el comando
desde Unity
      command.trim(); // Eliminar espacios en blanco extras

      if (command == "gsrpausar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
          gsrPausado = true;
          xSemaphoreGive(serialMutex);
        }
      } else if (command == "gsrreanudar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
          gsrPausado = false;
          xSemaphoreGive(serialMutex);
        }
      }
…… Resto de sensores y estados
```

Para cada sensor y estado se implementó una condición que determina si un sensor debe enviar o no un dato a Unity. Con ello también se debió modificar las tareas de cada sensor agregando un condicional que lo active o desactive como se muestra en el siguiente código:

```
void taskKy024(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!ky024Pausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer el estado del sensor digital
        ky024Value = !digitalRead(ky024);
        Serial.print("ky024:");
        Serial.println(ky024Value);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}
```

En Unity fue necesario crear el siguiente código para el manejo de datos:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO.Ports; //Manejo del Puerto serial
using System.Threading; //librería para manejo de hilos

public class SerialManagerScript : MonoBehaviour
{
    //Variables para almacenar datos de los sensores
    private int GSRValue;
    private int JoystickXValue;
    private int JoystickYValue;
…… Resto de definición de variables

//Variables del puerto serial
public string serialPortName = "COM4"; // Nombre del puerto serial
public int baudRate = 9600; // Velocidad del puerto serial
private SerialPort serialPort;
private string receivedData;

//Hilo para leer el puerto serial
private Thread serialThread;
```

En la función Start se inicializa el hilo que maneja la comunicación serial, mientras que en OnDestroy se termina con la comunicación para liberar el recurso y de ser necesario otro dispositivo lo pueda a utilizar.

```
private void Start()
{
    //Inicializo el puerto serial
    OpenSerialPort();
    //Se lanza un hilo para la lectura de datos del puerto serial
    StartSerialThread();
    //Se inicializa Arduino con los datos necesarios
    SendDataToArduino("EstadoLpausar");
    SendDataToArduino("EstadoPCpausar");

……Resto de datos enviados a Arduino
    }


private void OnDestroy()
{
    CloseSerialPort();
}
private void OpenSerialPort()
{
    if (serialPort != null && serialPort.IsOpen)
    {
        Debug.LogWarning("El puerto serial se encuentra abierto.");
        return;
    }
    serialPort = new SerialPort(serialPortName, baudRate);
    try
    {
        serialPort.Open();
        Debug.Log("Puerto serial abierto.");
    }
    catch (System.Exception e)
    {
        Debug.LogError("Error al abrir el puerto: " + e.Message);
    }
}
```

La función ParseSensorData recibe el dato de Arduino y lo separa en partes identificando el elemento como una cadena de caracteres divididos por el símbolo ":", con ello se identifica el nombre del sensor y el dato que lleva puede ser almacenado en una variable de Unity. El código es el siguiente:

```
void ParseSensorData(string data)
 {
     string[] parts = data.Split(':');
     if (parts.Length == 2)
```

```
    {
        string sensorName = parts[0].Trim();
        string sensorValueStr = parts[1].Trim();

        if (int.TryParse(sensorValueStr, out int sensorValue))
        {
            Debug.Log("Sensor: " + sensorName + ", Valor: " + sensorValue);

            // Almacena el valor del sensor en la variable correspondiente
            switch (sensorName)
            {
                case "GSR":
                    GSRValue = sensorValue;
                    Debug.Log("GSR: " + GSRValue);
                    break;
                ……Resto de casos para guardar los datos del sensor

            }
        }
    }
}
```

También fue necesario crear una función GetSensorValue para que los objetos del entorno de Unity puedan acceder a la información y realizar sus tareas específicas. También se crearon las funciones para enviar datos por el puerto serial para Arduino y otra función para cerrarlo.

```
public int GetSensorValue(string data)
    {
        switch (data)
        {
            case "GSR":
                return GSRValue;
           ……Resto de casos para leer el dato del sensor
        }
        return 0;
    }

private void CloseSerialPort()
 {
     if (serialPort != null && serialPort.IsOpen)
     {
         serialPort.Close();
         Debug.Log("Puerto serial cerrado.");
     }
 }
 public void SendDataToArduino(string data)
 {
     if (serialPort != null && serialPort.IsOpen)
     {
         serialPort.WriteLine(data);
         Debug.Log(data);
     }
     else
     {
         Debug.LogWarning("El puerto serial no está abierto");
```

```
    }
}
```

Luego es necesario tener una función que seleccione el mensaje que se debe enviar al Arduino dependiendo del sensor solicitado, para ellos se implementó lo siguiente:

```
private void HandleObjectEvent(string message)
{
    switch (message)
    {
        case "MovimientoLibreEntrada":
            SendDataToArduino("EstadoLreanudar");
            break;
       ……Resto de casos para enviar mensajes a Arduino
    }
}
```

Por último, se hizo la función que debe ejecutar el hilo de lectura de datos del sistema recordando que leer datos desde Arduino es una operación bloqueante del sistema y por esta razón se utiliza un hilo. El hilo a su vez utiliza otra función que sirve para leer el dato del puerto serie.

```
// Inicia el hilo para la lectura del puerto serie
private void StartSerialThread()
{
    serialThread = new Thread(ReadSerialData);
    serialThread.IsBackground = true;
    serialThread.Start();
}

// Detiene el hilo de lectura del puerto serie
private void StopSerialThread()
{
    if (serialThread != null && serialThread.IsAlive)
    {
        serialThread.Interrupt();
        serialThread.Join();
    }
}

// Función que se ejecuta en el hilo aparte para leer el puerto serie
private void ReadSerialData()
{
    while (serialPort.IsOpen)
    {
        try
        {
```

```
                receivedData = serialPort.ReadLine();
                // Procesa los datos recibidos, por ejemplo, llamando a
ParseSensorData(receivedData);
                ParseSensorData(receivedData);
            }
            catch (System.Exception e)
            {
                Debug.LogWarning("Error al leer datos del puerto serie: " +
e.Message);
            }
        }
    }
}
```

En cuanto al entorno de Unity se crearon objetos vacios que contienen componentes de box colider que se encuentran en cada zona y que son los encargados de solicitar y recibir información por parte del manejador del puerto serial. Los resultados de probar el codigo en cada zona fueron los siguiente:

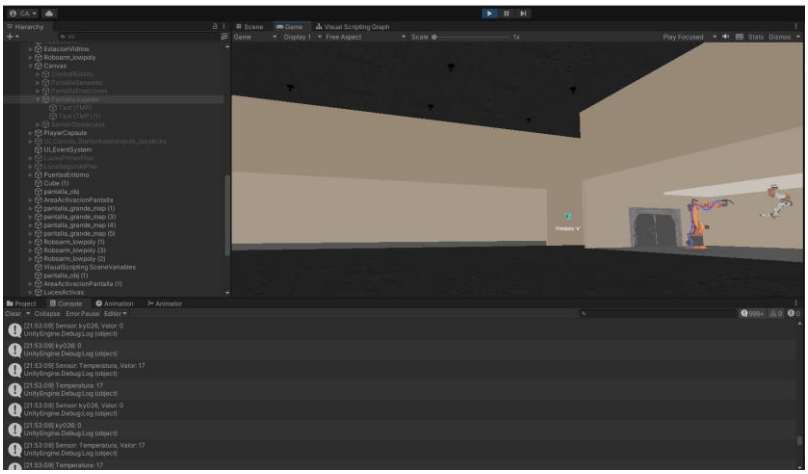

Figura 59. Zona de movimiento libre, datos sensor de Ky-013 (temperatura) y Ky-026 (flama)

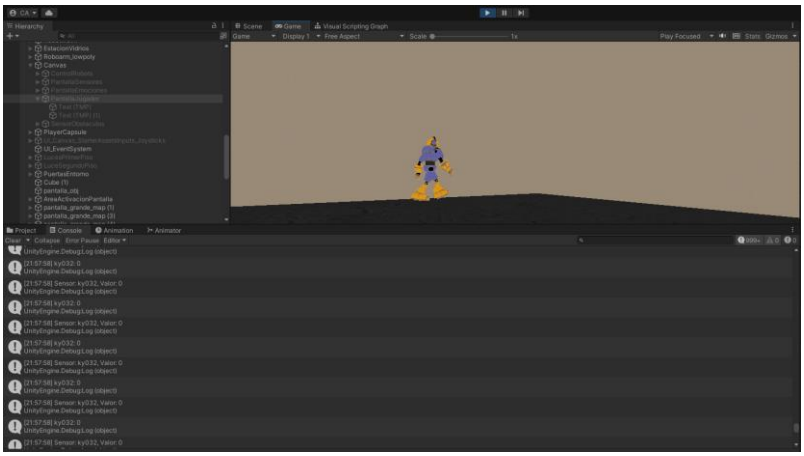

Figura 60. Zona de robot móvil, datos sensor Ky-032 (obstáculos)

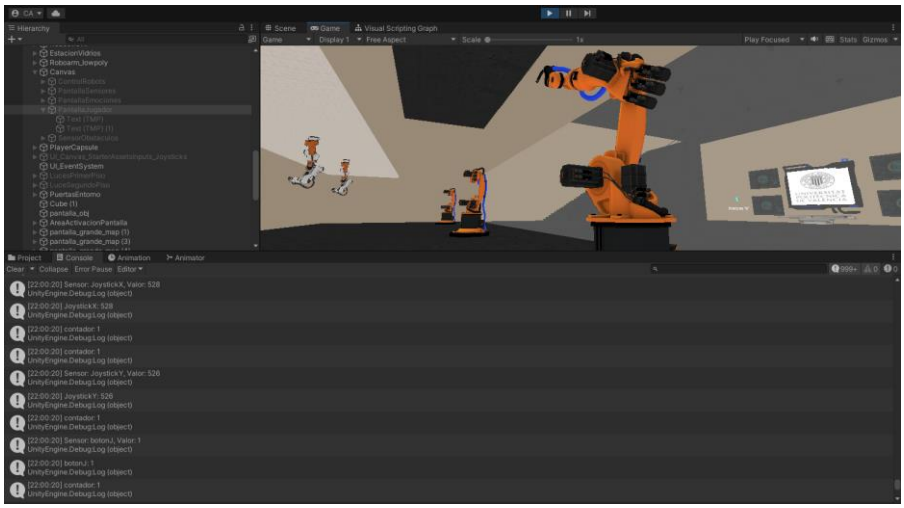

Figura 61. Zona brazo robótico 1, datos Joystick

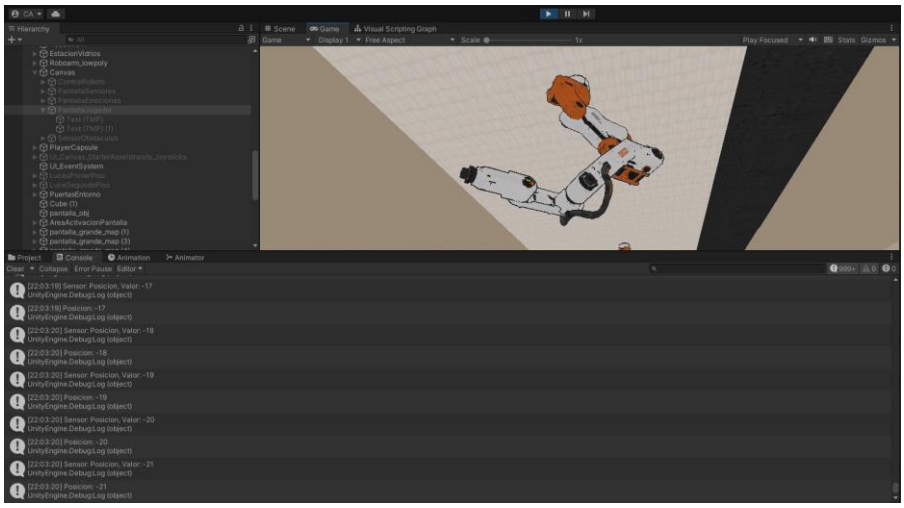

Figura 62. Zona brazo robótico 2, datos Encoder

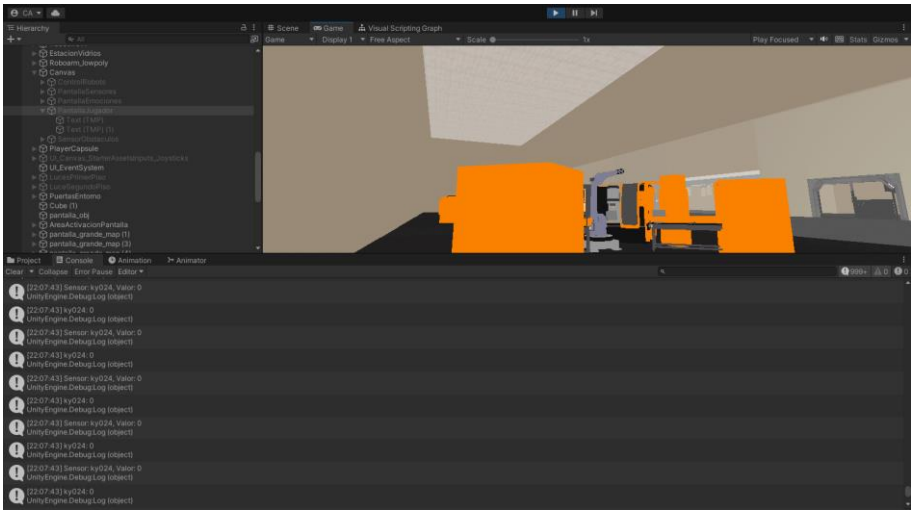

Figura 63. Zona de mecanizado, datos sensor Ky-024 (Efecto Hall)

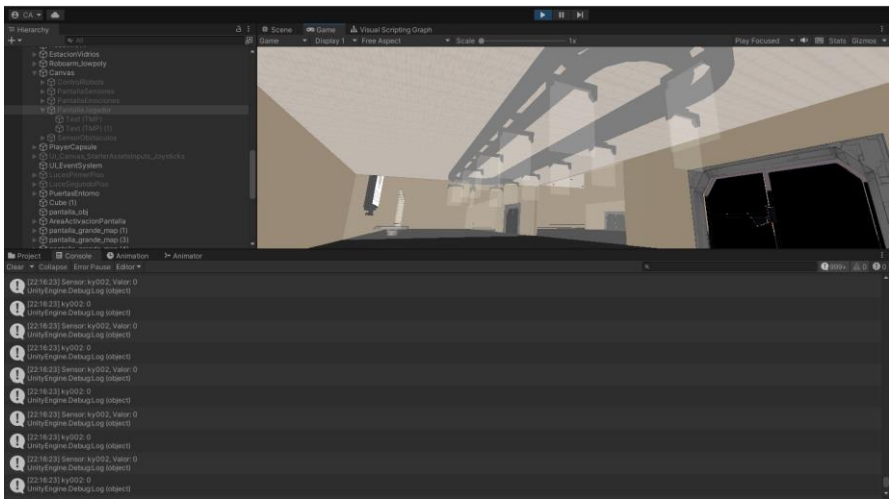

Figura 64. Zona de vidrios, datos sensor Ky-002(Impacto)

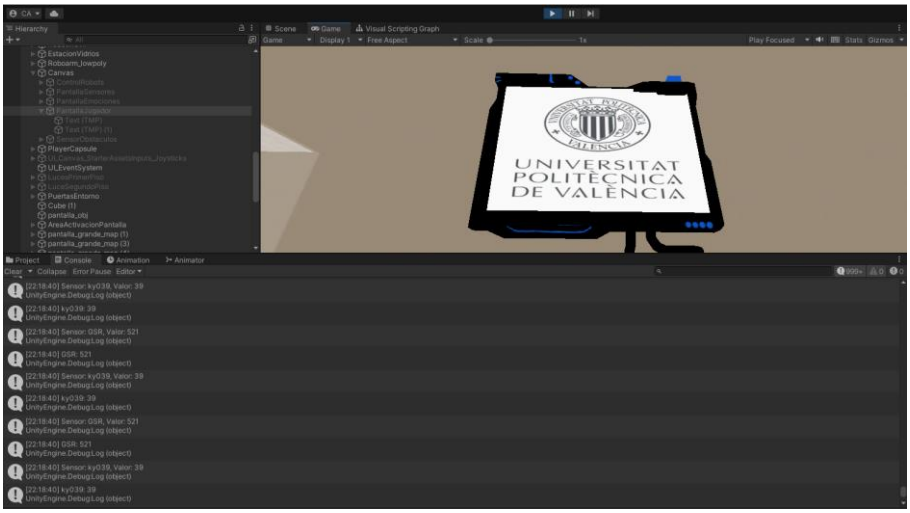

Figura 65. Zona de videos, datos sensores GSR y Ky-039(latidos)

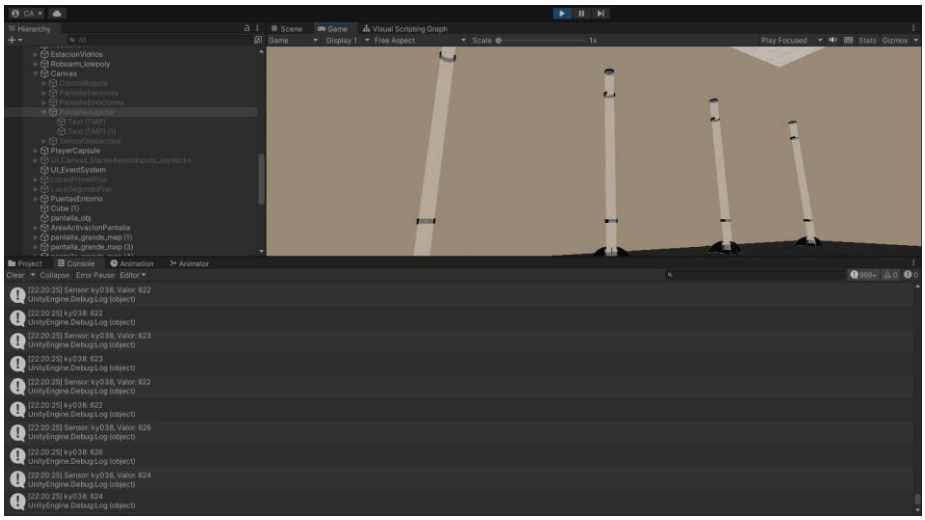

Figura 66. Zona de luces, datos sensor Ky-038

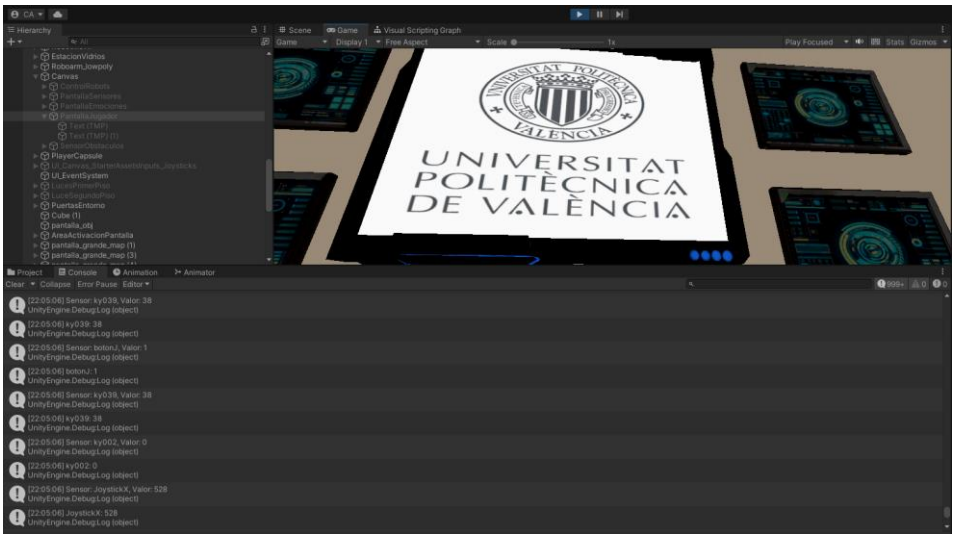

Figura 67. Zona de control, datos de todos los sensores

Como podemos ver en las imágenes se ha logrado seccionar de manera correcta la lectura de los sensores y separar la información. El código que se ha desarrollado funciona adecuadamente en el entorno. Por otra parte, cada uno de los objetos del entorno tienen scripts que manejan la información enviada por Arduino y que por su extensión se han colocado en el Anexo I del presente documento para su consulta.

5. Análisis de resultados

5.1. El entorno funcional

Para la versión final del entorno se colocaron guías para que el usuario pueda saber que hacer en cada una de las zonas. Al iniciar la experiencia se verá también un guía que le dirá al usuario como activar ciertos objetos del entorno.

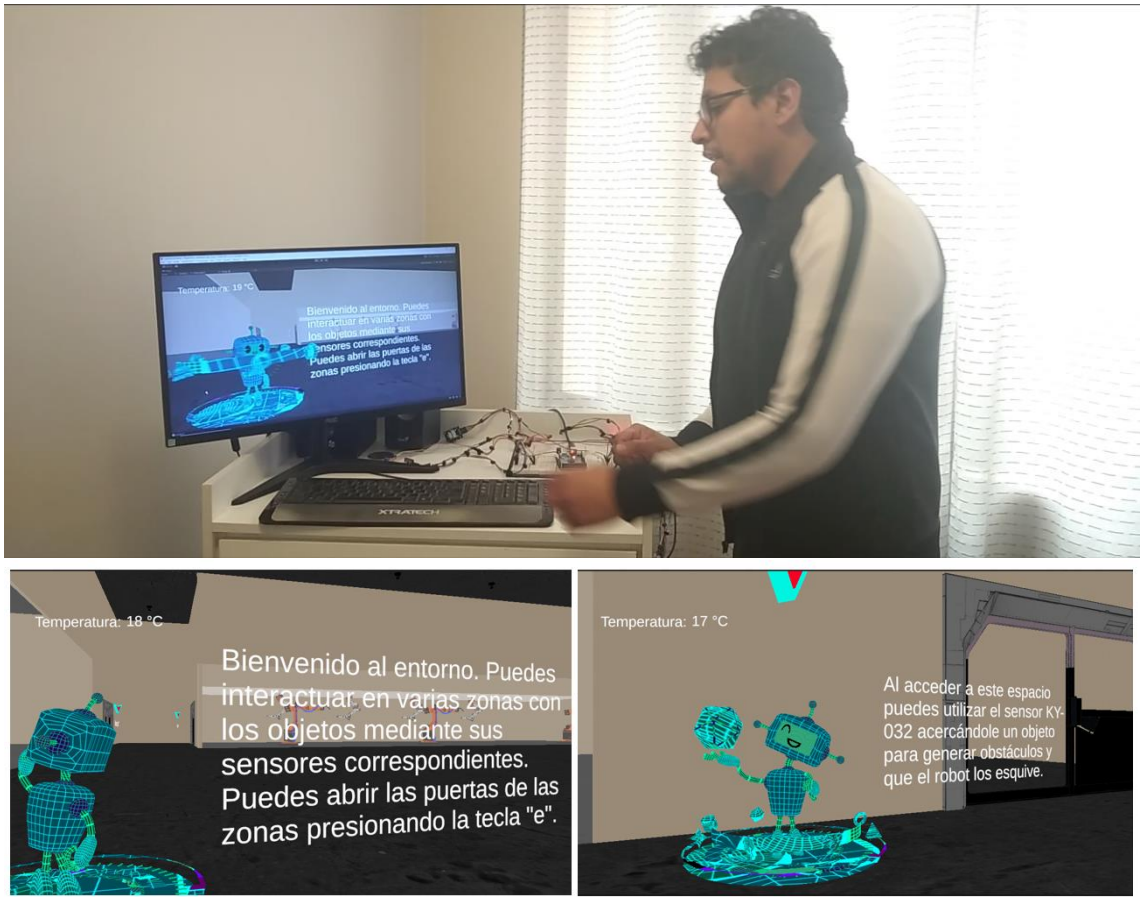

Figura 68. Elementos guía del entorno

Para el visor del usuario se registra el valor de la temperatura, para corregir datos que fluctuaban mucho y molestaban en la pantalla se colocó una condición para que se actualicen los datos del elemento cada 1s y esto mejoró el comportamiento. La medición de la temperatura tenía un error de 4 a 5 grados celsius por lo que se incluyó esta corrección en el codigo. Para el cambio de color de la camara de ususario su usó una transción suave, para que el cambio

no sé muy agresivo. Estas acciones mejoraron el funcionamiento de este elemento que recibe los datos del sensor KY-013, el registro de humedad se eliminó debido al cambio del sensor.

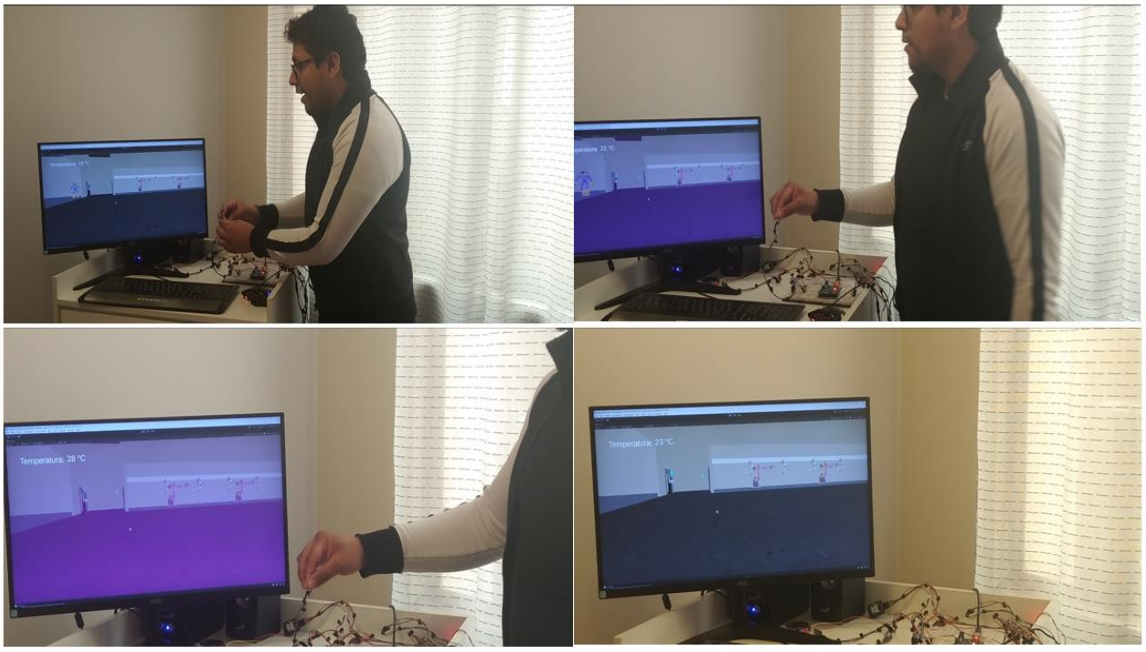

Figura 69. Pruebas del sistema, sensor de temperatura

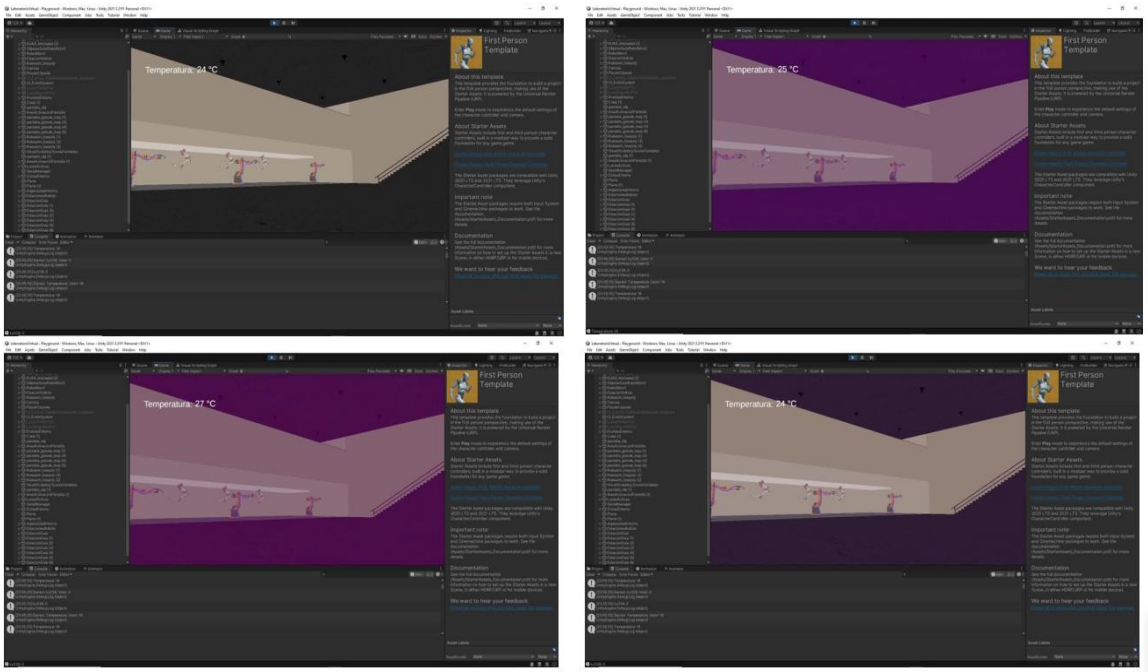

Figura 70. Registro del entorno con el sensor de temperatura

Para la generación de los obstáculos y que el robot pueda esquivarlos cuando aparecen en el entorno, se necesitaba un sistema que responda en tiempo real. Unity cuenta con la librería NavMesh que permite colocar un componente a un objeto, que hace posible navegar mediante el análisis del entorno que realiza el programa. En el robot móvil se colocó un componente de NavMesh Agent y se definieron 4 puntos en el entorno que marcan la trayectoria del robot en la zona. Para los obstáculos se colocó un objeto genérico de tipo cubo con un componente de NavMesh Obstacle y cada que se activa el sensor KY-032 se crea uno de estos objetos en el entorno, al finalizar la ejecución todos estos objetos se borran.

Figura 71. Pruebas del sistema, robot móvil

Figura 72. Registro del entorno con el robot móvil

Para el control de los brazos robóticos con el Joystick se tuvo que realizar un escalado de su medición a un rango de [-1,1] debido a que el sensor en su posición media entrega valores entre 515 a 520 y en sus extremos de 0 y de 1024, estos valores no servían para realizar el giro de las articulaciones. Otro de los problemas es que la función de Arduino enviaba los datos de manera ininterrumpida cuando se ingresa a la zona respectiva del robot. Esto causaba que el movimiento se acumule en cada interacción y no sea posible controlar la posición sino solo la velocidad de giro. Para arreglar este problema se realizó una interpolación inversa de valores entregados por el sensor, de esta manera se mapeó los datos entre el valor mínimo y máximo. Otro de los problemas fue identificar el eje de giro de cada una de las articulaciones, esto se resolvió probando el código del movimiento de forma individual.

El Joystick cuenta con un pulsador que tuvo que ser ajustado en el código de Unity, porque Arduino enviaba de forma indefinida los datos de este lo que no permitía usarlo de manera correcta. En este caso se quería usar el sensor para

cambiar las articulaciones que se estaban moviendo en ese momento, para solucionar este inconveniente se crearon dos variables en el código, una que almacena el estado actual del botón y otra que almacena el estado anterior, de esta manera solo se detecta el cambio de estado.

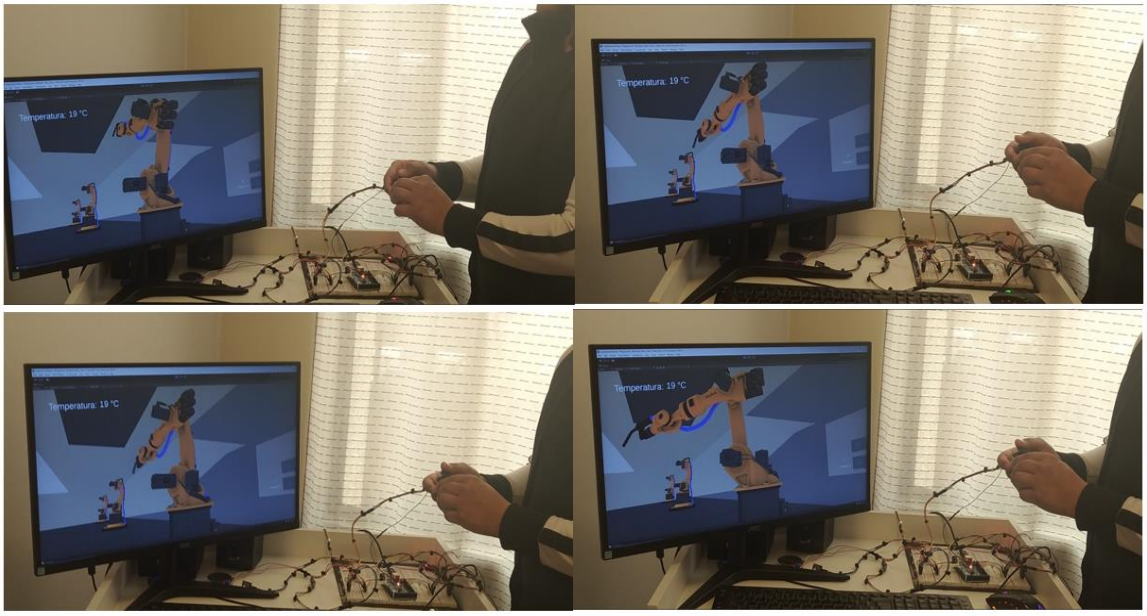

Figura 73. Pruebas del sistema, Joystick

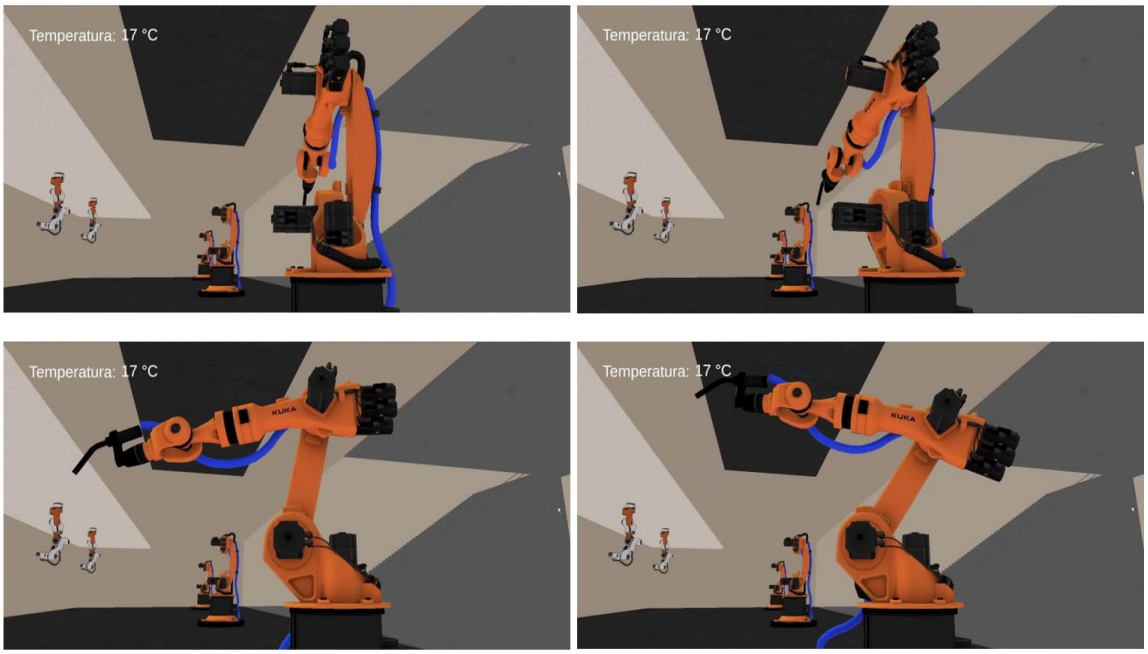

Figura 74. Brazos robóticos con el Joystick

Para el caso del encoder, Arduino solo enviaba datos al cambio de posición de este, lo que facilitó su programación en el entorno. El problema que surgió durante la implementación fue con el pulsador que tiene este módulo, que se programó para detectar cambios de alto a bajo y de bajo a alto. Con ello se pudieron controlar adecuadamente las articulaciones del robot.

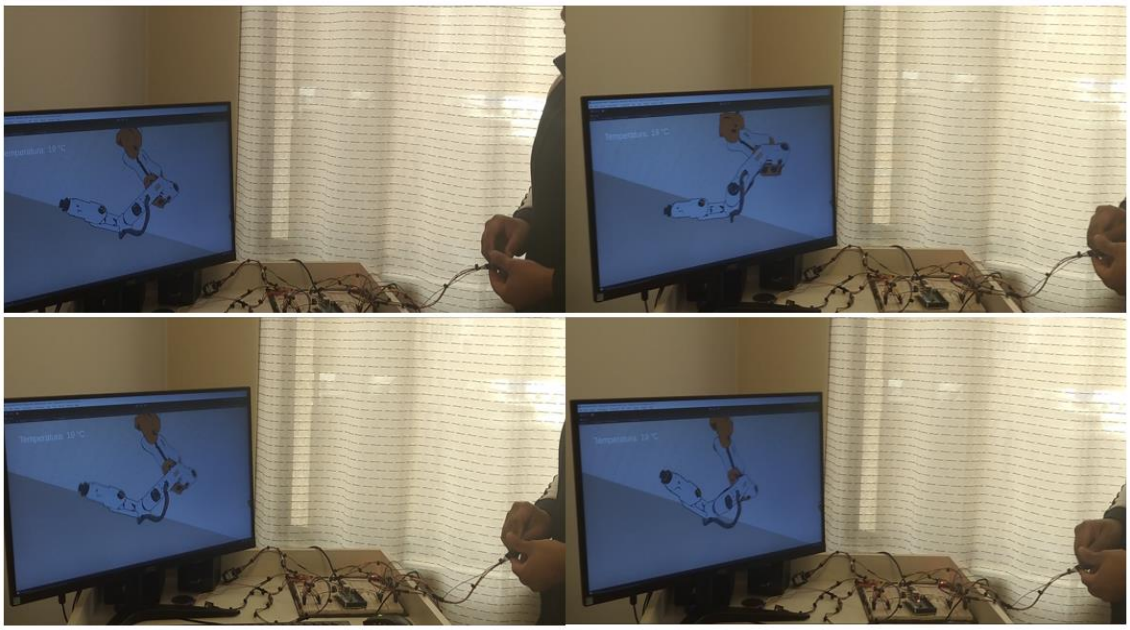

Figura 75. Pruebas del sistema, encoder

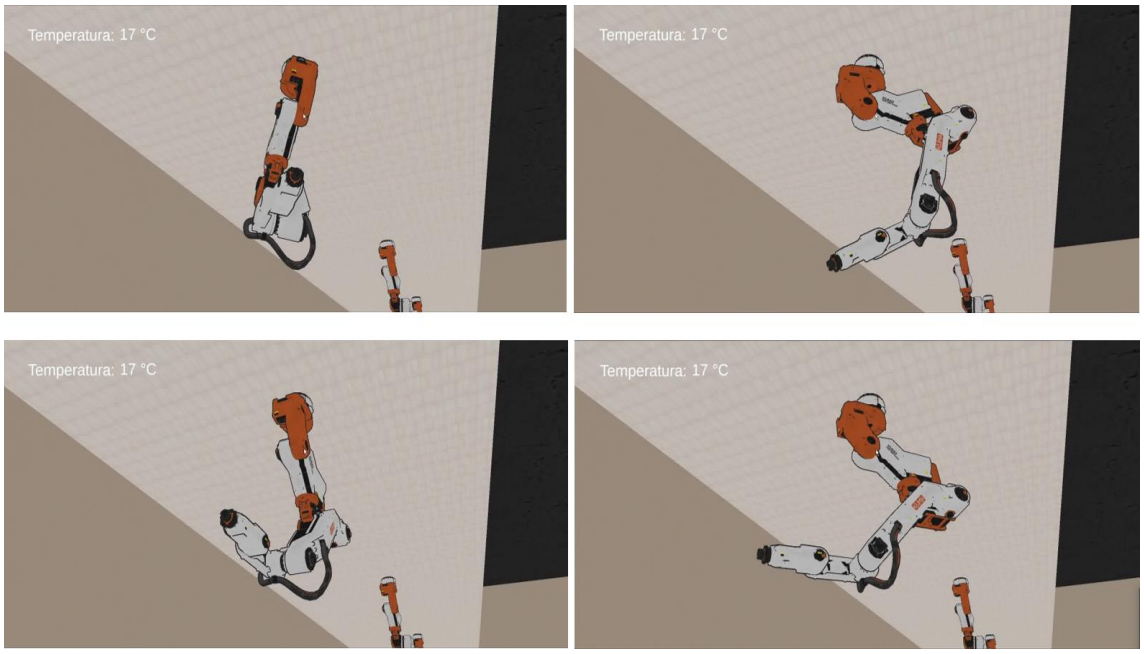

Figura 76. Entorno con sensor encoder

Para el sensor de flama y la activación del sistema contraincendios, fue necesario utilizar un sistema de partículas de Unity, que fue configurado para realizar su animación cuando el sensor KY-026 detecte una llama. No se presentaron problemas significativos para el control de este sensor.

Figura 77. Pruebas del sistema, sensor de flama

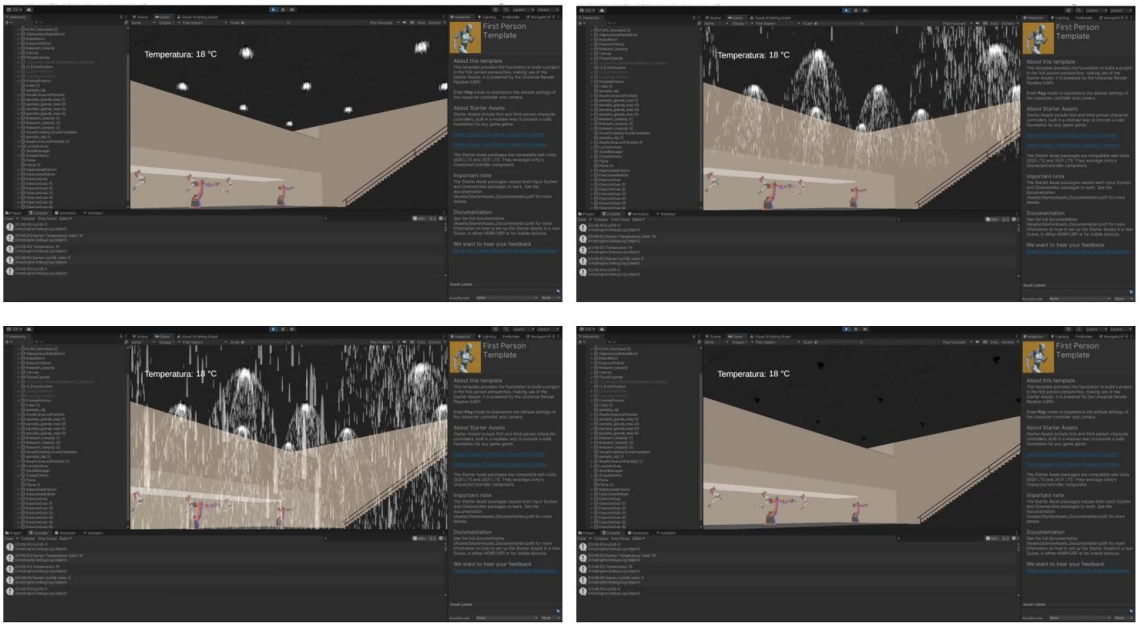

Figura 78. Registro del entorno con el sensor de flama activo

En la zona de mecanizado fue necesario manejar el sistema de animaciones mediante la programación de cada una de las actividades que realiza la estación. La activación del sensor KY-024 genera la animación de una pieza especifica, en el caso de no estar activado el sensor, el sistema realiza otra operación. El principal problema de esta sección del entorno fue programar cada una de las animaciones, y comprender el uso de los códigos y sintaxis para manejar el sistema de animaciones de Unity.

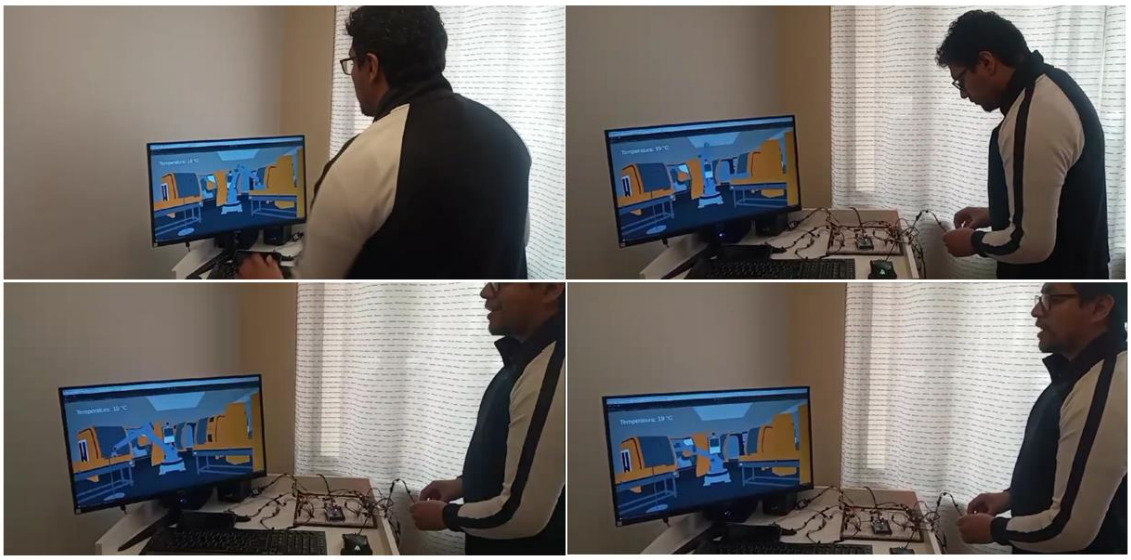

Figura 79. Pruebas del sistema, sensor de efecto hall

Figura 80. Registro del entorno en la zona de mecanizado

Para el sistema de soporte de los vidrios se utiliza las lecturas del sensor KY-002, que detecta golpes o vibraciones fuertes, al activarse causa que uno de los

vidrios se rompa. Para lograr este comportamiento se tienen dos objetos que están en el mismo lugar, el primero fue fraccionado en partes y el segundo esta completo. Cuando se activa el sensor el objeto entero desaparece y solo queda el que está en partes, este último reacciona a las físicas del entorno y cae al suelo. El problema que se enfrentó con este sistema fue que el sensor de impacto es poco sensible, no se detectan con facilidad sus cambios de estado además de que al estar sometido a impactos puede sufrir daños y dejar de funcionar correctamente.

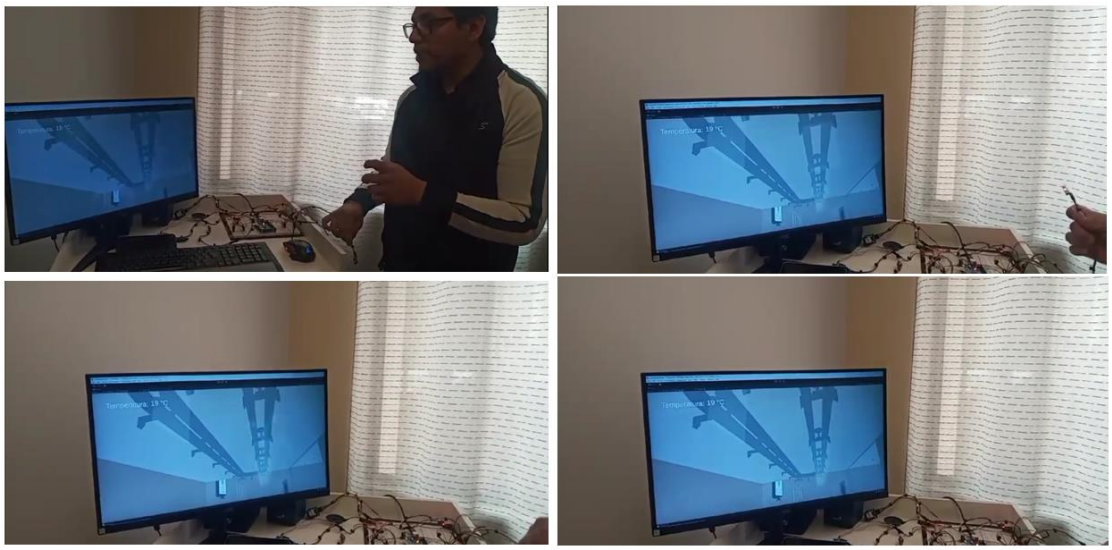

Figura 81. Pruebas del sistema, sensor de impacto

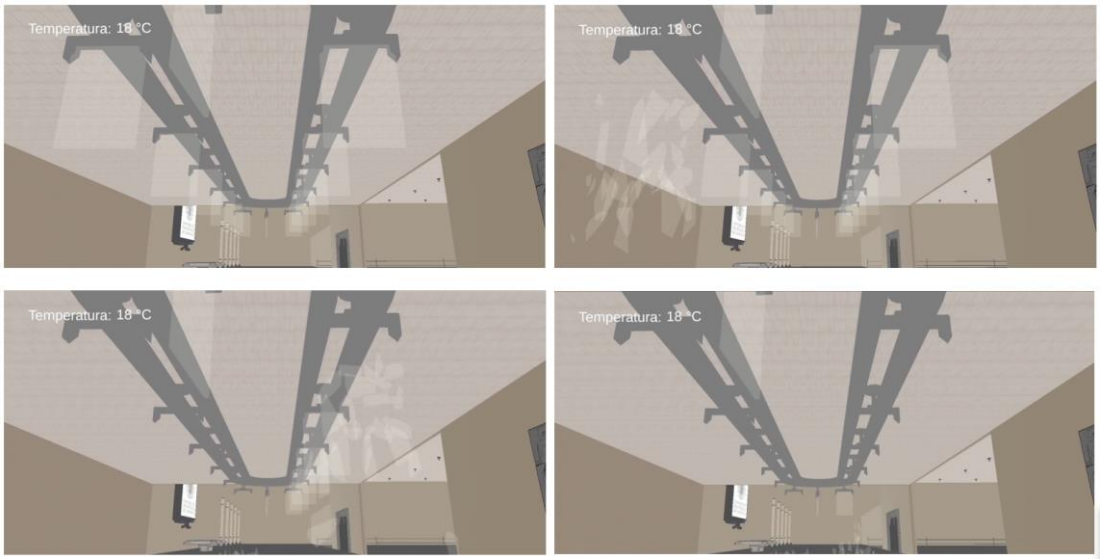

Figura 82. Sistema de soporte de vidrios

El sistema de luces activas que responden a estímulos auditivos registrados por el sensor KY-038, fueron programados para activar y desactivar un componente interno que varía dependiendo del nivel de la señal registrada. El principal problema es el nivel de sonido que debe ser detectado, el sonido tiene que ser estable en el tiempo, para mantener los niveles correspondientes y activar las luces. Esta condición es compleja, y normalmente las luces parpadean al recibir o registrar un estímulo. Para mejorar el comportamiento se podría colocar un filtro, pero en ese caso se perdería información de sonidos de corta duración.

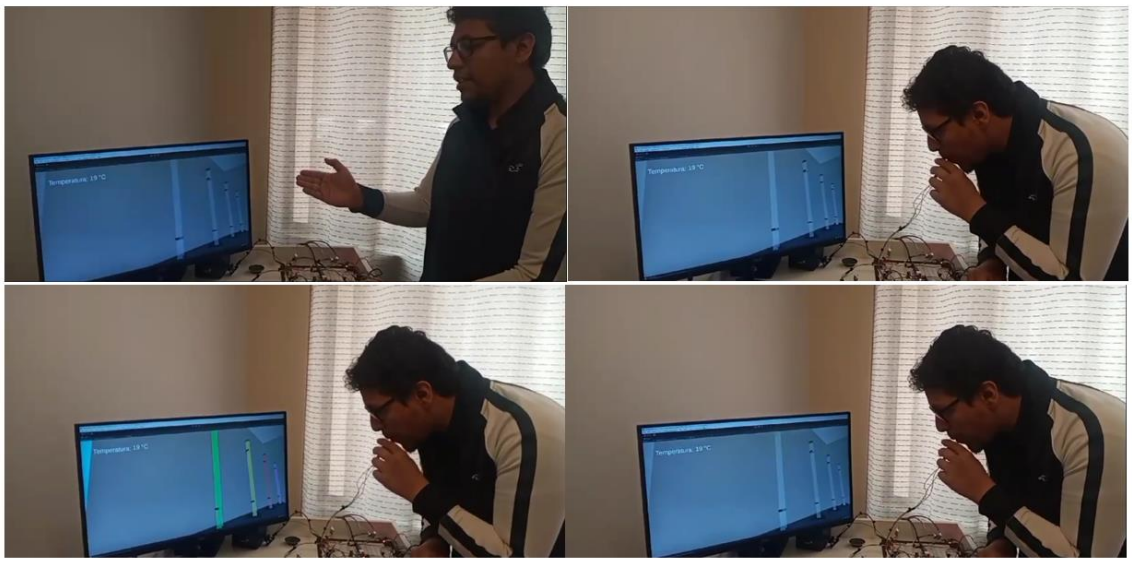

Figura 83. Pruebas del sistema, sensor de impacto

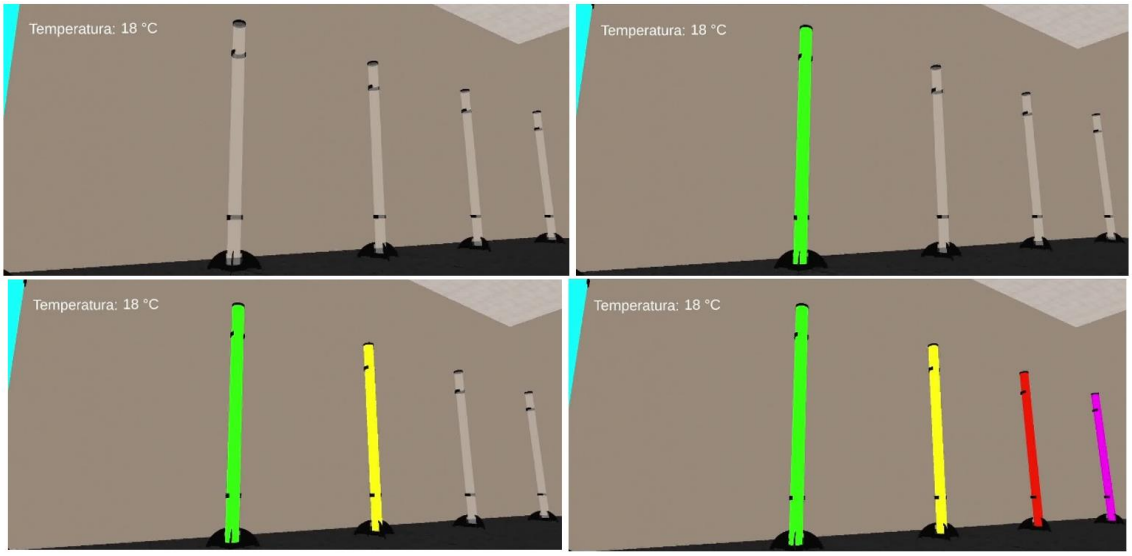

Figura 84. Sistema de luces activas

En la pantalla de registro de la respuesta galvánica de la piel con el sensor GSR y del ritmo cardiaco con el sensor KY-039, se colocó un video que muestra una serie de situaciones con diferentes cargas emocionales que busca estimular la variación en la lectura de los sensores. Se tienen dos registros numéricos en la pantalla para cada sensor, y botones para pausar y reanudar el video. En este caso el principal problema es la colocación de los sensores en los dedos del usuario y su calibración, esto debido a la diferencia de lecturas que se puede tener entre personas. Por este motivo se decidió mostrar los datos registrados por el sensor sin escalarse o modificarse. Para una calibración y transformación de medida es necesario realizar una mayor cantidad de pruebas con diferentes usuarios. La colocación de los sensores en los dedos en especial del sensor KY-039 es complicada y puede causar muchas variaciones en la medición.

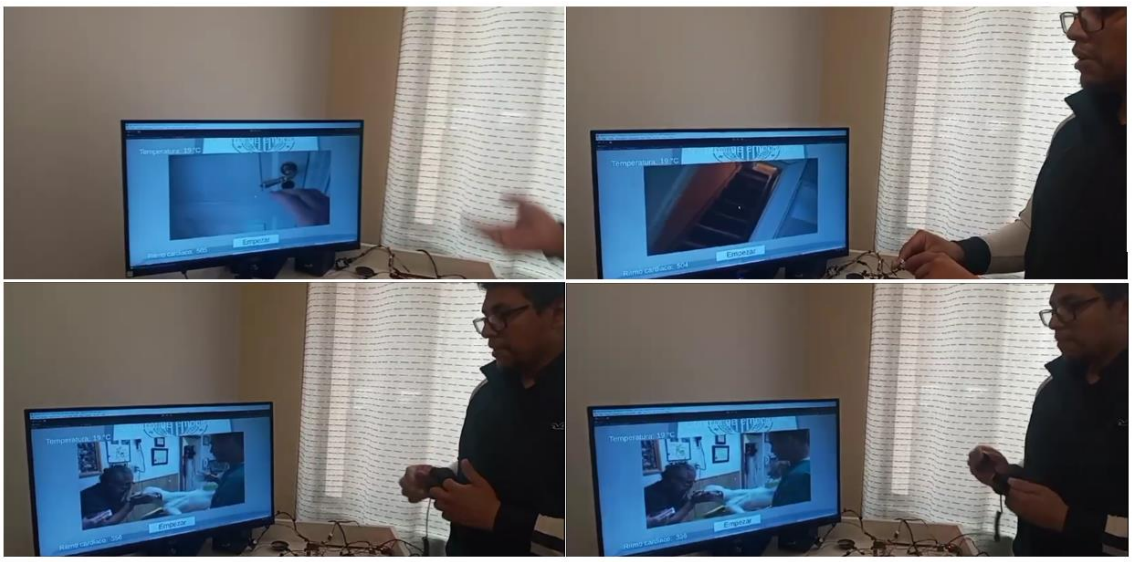

Figura 85. Pruebas del sistema, sensor GSR y de latidos de corazón

Por último, en el panel de control se colocaron registros numéricos que muestran el estado de todos los sensores lo cual es útil para ver como todos los sensores se encuentran en comunicación con Unity y con el entorno. El problema que se presentó en esta sección fue la programación del gráfico lineal de los

sensores, debido a que Unity no cuenta con ninguna herramienta para realizar esta tarea.

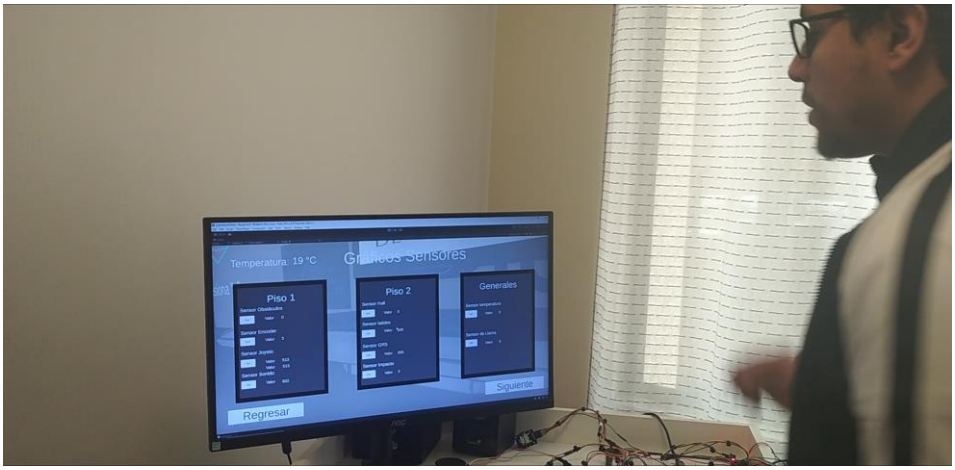

Figura 86. Pruebas del sistema, panel de control

5.2. Relación del trabajo con los Objetivos de Desarrollo Sostenible agenda 2030

El presente proyecto se enmarca en un contexto global de desarrollo sostenible, en línea con los Objetivos de Desarrollo Sostenible (ODS) establecidos por las Naciones Unidas. A través de la creación de un entorno tecnológico innovador en Unity que interactúa con sensores físicos y robótica, este trabajo se relaciona con varios ODS clave, contribuyendo a la promoción de la industria y la innovación, la mejora de la calidad educativa y la búsqueda de soluciones para comunidades más sostenibles. A continuación, se detallará cómo este proyecto específico se conecta con estos objetivos globales para abordar desafíos críticos y avanzar hacia un futuro más equitativo y sostenible.

El proyecto se relaciona con el ODS 4 – Educación de Calidad al incorporar elementos de aprendizaje y desarrollo de habilidades técnicas en su ejecución. En particular, ofrece oportunidades de aprendizaje relacionadas con la programación de robots y la interacción con sensores físicos. Al proporcionar

un entorno en Unity que interactúa con 10 sensores físicos y utiliza tecnologías como Arduino MEGA, el proyecto brinda una plataforma para que los usuarios adquieran conocimientos y habilidades en áreas relevantes para la tecnología y la informática. Los usuarios pueden aprender sobre la programación de sensores, la comunicación entre dispositivos y la gestión de sistemas tecnológicos complejos. Asimismo, al utilizar un enfoque de programación concurrente con tecnologías como FreeRTOS de Arduino y la gestión de hilos en Unity, el proyecto también ofrece oportunidades de aprendizaje avanzado en el campo de la informática.

El proyecto se relaciona con el ODS 9 - Industria, Innovación e Infraestructura al enfocarse en el desarrollo de un entorno tecnológico innovador. Este entorno utiliza 10 sensores físicos conectados a una placa Arduino MEGA y se comunica con una plataforma virtual en Unity. La infraestructura de comunicación serial entre las dos plataformas se ha diseñado con el propósito de mejorar la interacción entre el mundo físico y el virtual. Además, el proyecto incorpora elementos de programación concurrente, haciendo uso de tecnologías como FreeRTOS de Arduino y la gestión de hilos en Unity. Esto fomenta la eficiencia en la ejecución de tareas y contribuye a la optimización de la infraestructura tecnológica utilizada.

El proyecto se relaciona con el ODS 11 - Ciudades y Comunidades Sostenibles al abordar la creación de un entorno tecnológico que tiene el potencial de mejorar la calidad de vida en entornos urbanos y comunidades a través de la innovación tecnológica. Al desarrollar un entorno en Unity que interactúa con sensores físicos y utiliza tecnologías avanzadas como Arduino MEGA, el proyecto busca ofrecer soluciones tecnológicas que pueden aplicarse en entornos

urbanos para mejorar la eficiencia y la calidad de vida. Estas soluciones pueden incluir la gestión eficiente de recursos, la automatización de tareas y la creación de experiencias interactivas que pueden enriquecer la vida en la ciudad. Al mismo tiempo, al integrar sistemas de sensores y robótica en el entorno virtual, el proyecto promueve la exploración de soluciones innovadoras para desafíos urbanos, como la movilidad, la seguridad y la gestión de recursos.

El proyecto se relaciona con el ODS 17 - Alianzas para Lograr los Objetivos al incorporar múltiples componentes tecnológicos y colaborativos en su ejecución. Para empezar, el proyecto combina tecnologías de hardware como Arduino MEGA y sensores físicos con un entorno virtual en Unity, lo que demuestra la capacidad de integrar diversas plataformas y tecnologías para lograr un objetivo común. Esta colaboración entre diferentes sistemas tecnológicos es una manifestación de la cooperación y la alianza entre diferentes enfoques para resolver desafíos complejos. Además, el proyecto puede involucrar la colaboración entre profesionales de diferentes disciplinas, como la ingeniería, la informática y la programación, para diseñar y desarrollar soluciones tecnológicas avanzadas. Esta colaboración interdisciplinaria es fundamental para abordar problemas complejos y promover la innovación. En última instancia, al enfocarse en la interacción de múltiples elementos y tecnologías, el proyecto ejemplifica el espíritu del ODS 17, que destaca la importancia de las alianzas y la cooperación entre diversos actores, incluidos los sectores público y privado, para lograr objetivos sostenibles.

6. Conclusiones y trabajos futuros

6.1. Conclusiones

Se logró implementar con éxito un entorno completamente funcional en Unity que tiene conexión por comunicación serial con una placa Arduino MEGA que a su vez adquiere, procesa y envia datos de los sensores conectados en sus puertos digitales y analógicos.

Para el desarrollo del entorno fue necesario estudiar y compreder el funcionamiento de algunas herramientas de desarrollo. Entre las más importantes se encuentran el uso de threads o hilos en Unity además de sus librerías de NavMesh. En Arduino se destaca el uso de la librería FreeRTOS que permite con ciertas limitaciones el uso de computación concurrente.

Los hilos en Unity permitieron gestionar la comunicación del puerto serial, sin este enfoque el proceso hubiera quedado bloqueado siempre que se quisiera recibir un dato de Arduino. Es importante recordar que el envío y recepción de datos son actividades bloqueantes del sistema.

Separar las actividades de Arduino en tareas con el uso de la librería FreeRTOS simplificó la integración del sistema y permitió la correcta comunicación entre el software y el hardware. Sin este enfoque hubiera sido muy complejo gestionar el envío de datos por el puerto serial sobre todo por la cantidad de datos que se necesitaban enviar.

El uso de mutex para gestionar el envío de datos por parte de Arduino en el puerto serial evitó la perdida de datos y el bloqueo de la comunicación del

sistema, además de que garantiza que todos los sensores en algún momento pueden acceder correctamente al recurso para enviar la información.

El control del robot movil del entorno fue el unico dispositivo que utilizó una herramienta especial de Unity. El uso de NavMesh facilitó la implementación de este elemento, caso contrario debía programarse cada uno de los comportamientos del robot para esquivar los obstáculos y las animaciones necesarias para cada movimiento.

Se realizó la prueba y calibración de los sensores, garantizando que los datos recibidos por Unity sean adecuados para su uso. Se utilizó un protocolo de pruebas que permitió analizar los datos obetenidos. De los cuales se puede decir que el sensor analogico que tiene mayor cantidad de ruido es el KY-038 sensor de microfono seguido del sensor GSR. Para los sensores digitales el que tuvo la mayor cantidad de errores en sus lecturas fue el sensor KY-031 que fue reemplazado por el sensor KY-002. Otro dato que se puede destacar es que el sensor de efecto Hall KY-024 es el que peor rendimiento tuvo para detectar estimulos de alta velocidad.

6.2. Trabajos futuros

El presente trabajo puede aportar información para el desarrollo de entornos en realidad virtual o aumentada que permita capacitar o mejorar el aprendizaje de grupos de personas en diferentes áreas del conocimiento.

De igual manera podría utilizarse para fomentar proyectos donde se desee adquirir información de los usuarios dentro de un entorno virtual, por ejemplo,

en experiencias interactivas donde se quiera saber que tan estimulantes han sido para los diferentes usuarios

Así mismo la investigación serviría para implementar laboratorios con gemelos digitales que permitan recrear de una manera más precisa el comportamiento del entorno real dentro de un espacio virtual.

7. Bibliografía

Acedo Sánchez, J. (2006). *Instrumentación y control básico de procesos.* España: Díaz de Santos.

Bertran Albertí, E. (2006). *Proceso digital de señales. Fundamentos para comunicaciones y control .* Barcelona: Edicions de la Universitat Politècnica de Catalunya.

Corona Ramírez, L. G., Abarca Jiménez, G. S., & Mares Carreño, J. (2014). *Sensores y actuadores, aplicaciones con Arduino.* México D.F.: Grupo Editorial Patria, S.A. de C.V.

Craighead, J. D., Burke, J., & Murphy, R. R. (2007). Using the Unity Game Engine to Develop SARGE: A Case Study. *Computer*, 45-52.

Fitzgerald, S., & Shiloh, M. (2012). *Arduino libro de proyectos.* Torino: Arduio LLC.

Goilav, N., & Loi, G. (2016). *ARDUINO. Aprender a desarrollar para crear objetos inteligentes.* Barcelona: Ediciones ENI.

Jafri, R., Louzada Campos, R., Abid Ali, S., & Arabnia, H. R. (2017). Visual and Infrared Sensor Data-Based Obstacle Detection for the Visually Impaired Using the Google Project Tango Tablet Development Kit and the Unity Engine. *IEEE Access*, 443-454.

Jiménez Gil, L. I. (Junio de 2023). Introducción a Unity Engine. Prácticas de diseño interactivo. Valladolid: Universidad de Valladolid.

Juin-Ling, T., & Chia-Wei, C. (2018). Interaction Design in Virtual Reality Game Using Arduino Sensors. (D. Cvetkovic, Ed.) *IntechOpen*, 111-127.

Lab-Volt. (2001). *Fluidos. Sensores.* Quebec: Lab-Volta Ltda.

Lidon, M. (2019). *Unity 3D.* Barcelona: Marcombo.

Linowes, J. (2015). *Unity Virtual Reality Projects.* Birmingham: Packt Publishing.

Pallás Areny, R. (2005). *Sensores y acondicionadores de señal* (4ª edición ed.). España: Marcombo.

Peña Millahual, C. A. (2020). *Descubriendo Arduino.* Ciudad Autónoma de Buenos Aires: SIXEDICIONES.

Torrente Artero, O. (2013). *Arduino. Curso práctico de formación.* Madrid: RC Libros.

Yong, J. (2021). Design and Development of "VR + Arduino" Immersion Interactive Experiment Case. *Journal of Physics: Conference Series*.

8. Anexos

Anexo 1. Códigos del sistema

Código Arduino

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

//Declaración de mutex
SemaphoreHandle_t serialMutex;

//Definición de variables
int GSRValue = 0;
int joystickXValue = 0;
int joystickYValue = 0;
int ky039Value = 0;
int ky038Value = 0;
int ky013Value = 0;
float R1 = 10000; // Valor de resistencia en el módulo
float logR2, R2, T;
float c1 = 0.001129148, c2 = 0.000234125, c3 = 0.0000000876741;
//coeficientes del termistor por steinhart-hart

int encoderAValue = 0;
int encoderBValue = 0;
int posicion = 0;
int estadoAnteriorA = LOW;
unsigned long debounceTiempo = 0;
unsigned long debounceDelay = 50;
int ky032Value = 0;
int ky002Value = 0;
int ky026Value = 0;
int ky024Value = 0;
int botonJValue = 0;
int botonEValue = 0;

//Variables para el control de la comunicación en Unity
bool gsrPausado = true;
bool joystickPausado = true;
bool ky039Pausado = true;
bool ky038Pausado = true;
bool ky013Pausado = true;
bool encoderPausado = true;
bool ky032Pausado = true;
bool ky002Pausado = true;
bool ky026Pausado = true;
bool ky024Pausado = true;
/*
```

```
bool monitoreoEPausado = false; //Lectura de los sensores GSR y KY-039
bool estadoMLPausado = false; //Estado movimiento libre
bool estadoPCPausado = false; //Estado panel de control
*/

// Definición de pines analógicos
const int sensorGSR = A0;
const int joystickXPin = A1;
const int joystickYPin = A2;
const int ky039 = A3;
const int ky038 = A4;
const int ky013 = A5;

// Definición de pines digitales
const int encoderA = 2;
const int encoderB = 3;
const int ky032 = 4;
const int ky002 = 5;
const int ky026 = 6;
const int ky024 = 7;
const int botonE = 9;
const int botonJ = 10;

void setup() {
  Serial.begin(9600);

  // Crear un mutex para el puerto serial
  serialMutex = xSemaphoreCreateMutex();
  if (serialMutex != NULL) {
    Serial.println("Serial mutex created.");
  }

  // Configurar pines como entrada
  //Analógicos
  pinMode(sensorGSR, INPUT);
  pinMode(joystickXPin, INPUT);
  pinMode(joystickYPin, INPUT);
  pinMode(ky039, INPUT);
  pinMode(ky038, INPUT);
  pinMode(ky013, INPUT);

  //Digitales
  pinMode(encoderA, INPUT);
  pinMode(encoderB, INPUT);
  pinMode(ky032, INPUT_PULLUP);
  pinMode(ky002, INPUT_PULLUP);
  pinMode(ky026, INPUT_PULLUP);
  pinMode(ky024, INPUT_PULLUP);
  pinMode(botonE, INPUT_PULLUP);
```

```
  pinMode(botonJ, INPUT_PULLUP);

  // Crear tareas para leer cada uno de los sensores
  //Analógicos
  xTaskCreate(taskSensorGSR, "SensorGSRTask", 128, NULL, 1, NULL);
  xTaskCreate(taskJoystick, "JoystickTask", 128, NULL, 1, NULL);
  xTaskCreate(taskKy039, "Ky039Task", 128, NULL, 1, NULL);
  xTaskCreate(taskKy038, "Ky038Task", 128, NULL, 1, NULL);
  xTaskCreate(taskKy013, "Ky013Task", 128, NULL, 1, NULL);

  //Digitales
  xTaskCreate(taskKy026, "Ky026Task", 128, NULL, 1, NULL);
  xTaskCreate(taskKy032, "Ky032Task", 128, NULL, 1, NULL);
  xTaskCreate(taskKy002, "Ky002Task", 128, NULL, 1, NULL);
  xTaskCreate(taskKy024, "Ky024Task", 128, NULL, 1, NULL);
  xTaskCreate(taskEncoder, "EncoderTask", 128, NULL, 1, NULL);
  xTaskCreate(taskBotonEncoder, "BotonEncoderTask", 128, NULL, 1, NULL);
  xTaskCreate(taskBotonJoystick, "BotonJoystick", 128, NULL, 1, NULL);

  // Crear tarea para manejar comandos desde Unity
  xTaskCreate(taskHandleCommands, "CommandHandlerTask", 128, NULL, 1,
NULL);
}

void loop() {
  // El loop no se utiliza en FreeRTOS
}

//////////////////////Funciones Sensores
analógicos//////////////////////////////
void taskSensorGSR(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!gsrPausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer valores del sensor
        GSRValue = analogRead(sensorGSR);
        Serial.print("GSR:");
        Serial.println(GSRValue);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}
```

```
void taskJoystick(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!joystickPausado) { // Solo envía datos si los sensores no están
pausados
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {// Tomar
el mutex para acceder al puerto serial
        // Leer valores del sensor
        joystickXValue = analogRead(joystickXPin);
        joystickYValue = analogRead(joystickYPin);
        Serial.print("JoystickX:");
        Serial.println(joystickXValue);
        Serial.print("JoystickY:");
        Serial.println(joystickYValue);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskKy039(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!ky039Pausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer valores del sensor
        ky039Value = analogRead(ky039);
        Serial.print("ky039:");
        Serial.println(ky039Value);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskKy038(void *pvParameters) {
  (void)pvParameters;
  while (1) {
```

```
    if (!ky038Pausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer valores del sensor
        ky038Value = analogRead(ky038);
        Serial.print("ky038:");
        Serial.println(ky038Value);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskKy013(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!ky013Pausado) { // Solo envía datos si los sensores no están
pausados
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        ky013Value = analogRead(ky013);
        R2 = R1 / ((1023.0 / (float)ky013Value) - 1.0); //calculo del
valor del termistor
        logR2 = log(R2);
        T = (1.0 / (c1 + (c2*logR2) + (c3*logR2*logR2*logR2))); //
Temperatura en Kelvin
        T = (T - 273.15)-5; //convertir Kelvin a Celcius más corrección
de temperatura -5 grados
        // Ahora, convierte la temperatura a un entero
        int temperaturaEntera = (int)T;
        Serial.print("Temperatura:");
        Serial.println(temperaturaEntera);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

//////////////////////Funciones Sensores
digitales//////////////////////////////
void taskKy026(void *pvParameters) {
  (void)pvParameters;

  while (1) {
```

```
    if (!ky026Pausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer el estado del sensor digital
        ky026Value = digitalRead(ky026);
        Serial.print("ky026:");
        Serial.println(ky026Value);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskKy032(void *pvParameters) {
  (void)pvParameters;
  while (1) {

    if (!ky032Pausado) { // Solo envía datos si los sensores no están
pausados
    // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer el estado del sensor digital
        ky032Value = !digitalRead(ky032);
        Serial.print("ky032:");
        Serial.println(ky032Value);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskKy002(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!ky002Pausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer el estado del sensor digital
        ky002Value = !digitalRead(ky002);
        Serial.print("ky002:");
        Serial.println(ky002Value);
        xSemaphoreGive(serialMutex);
      }
```

```
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskKy024(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!ky024Pausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer el estado del sensor digital
        ky024Value = !digitalRead(ky024);
        Serial.print("ky024:");
        Serial.println(ky024Value);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskEncoder(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!encoderPausado) { // Solo envía datos si los sensores no están
pausados
      int estadoActualA = digitalRead(encoderA);
      if (estadoActualA != estadoAnteriorA) {
        if (digitalRead(encoderB) != estadoActualA) {
          // Rotación en sentido horario
          posicion++;
        } else {
          // Rotación en sentido antihorario
          posicion--;
        }
          if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
            Serial.print("Posicion:");
            Serial.println(posicion);
            xSemaphoreGive(serialMutex);
          }
        }
        estadoAnteriorA = estadoActualA;
      }
```

```
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

void taskBotonEncoder(void *pvParameters) {
  (void)pvParameters;
  int ultimoEstado = LOW; // Variable para almacenar el estado anterior
del botón
  bool cambioEstadoEnviado = false; // Bandera para asegurarse de enviar
el mensaje solo una vez

  while (1) {
    if (!encoderPausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer el estado del botón (invertir la lógica)
        int estadoActual = digitalRead(botonE);

        if (estadoActual == HIGH && ultimoEstado == LOW) {
          // Botón pasó de bajo a alto
          Serial.println("botonE:0");
          cambioEstadoEnviado = false; // Restablecer la bandera
        } else if (estadoActual == LOW && ultimoEstado == HIGH &&
!cambioEstadoEnviado) {
          // Botón pasó de alto a bajo y el mensaje no se ha enviado
antes
          Serial.println("botonE:1");
          cambioEstadoEnviado = true; // Establecer la bandera para
evitar mensajes repetidos
        }
        ultimoEstado = estadoActual; // Almacenar el estado actual como
último estado
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }

}

void taskBotonJoystick(void *pvParameters) {
  (void)pvParameters;

  while (1) {

    if (!joystickPausado) { // Solo envía datos si los sensores no están
pausados
      // Tomar el mutex para acceder al puerto serial
```

```
      if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
        // Leer el estado del sensor digital
        botonJValue = digitalRead(botonJ);
        Serial.print("botonJ:");
        Serial.println(botonJValue);
        xSemaphoreGive(serialMutex);
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Leer cada 10 ms
  }
}

//////////////////////////////////////////Tarea de
comunicación//////////////////////////////////////////

void taskHandleCommands(void *pvParameters) {
  (void)pvParameters;
  Serial.println("Lanzo CommandHandlerTask");
  while (1) {
    if (Serial.available() > 0) {
      String command = Serial.readStringUntil('\n'); // Leer el comando
desde Unity
      command.trim(); // Eliminar espacios en blanco extras

      if (command == "gsrpausar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
          gsrPausado = true;
          xSemaphoreGive(serialMutex);
        }
      } else if (command == "gsrreanudar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
          gsrPausado = false;
          xSemaphoreGive(serialMutex);
        }
      }

      if (command == "joystickpausar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
          joystickPausado = true;
          xSemaphoreGive(serialMutex);
        }
      } else if (command == "joystickreanudar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
          joystickPausado = false;
          xSemaphoreGive(serialMutex);
```

```
  }
}

if (command == "ky039pausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky039Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "ky039reanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky039Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "ky038pausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky038Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "ky038reanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky038Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "ky013pausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky013Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "ky013reanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky013Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "encoderpausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    encoderPausado = true;
```

```
    xSemaphoreGive(serialMutex);
  }
} else if (command == "encoderreanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    encoderPausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "ky032pausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky032Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "ky032reanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky032Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "ky002pausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky002Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "ky002reanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky002Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "ky026pausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky026Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "ky026reanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky026Pausado = false;
    xSemaphoreGive(serialMutex);
```

```
  }
}

if (command == "ky024pausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky024Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "ky024reanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky024Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "MonitorEpausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    gsrPausado = true;
    ky039Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "MonitorEreanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    gsrPausado = false;
    ky039Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}

if (command == "EstadoLpausar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky013Pausado = true;
    ky026Pausado = true;
    xSemaphoreGive(serialMutex);
  }
} else if (command == "EstadoLreanudar") {
  // Tomar el mutex para acceder a la variable sensorsPaused
  if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
    ky013Pausado = false;
    ky026Pausado = false;
    xSemaphoreGive(serialMutex);
  }
}
```

```
      if (command == "EstadoPCpausar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
            gsrPausado = true;
            joystickPausado = true;
            ky039Pausado = true;
            ky038Pausado = true;
            ky013Pausado = true;
            encoderPausado = true;
            ky032Pausado = true;
            ky002Pausado = true;
            ky026Pausado = true;
            ky024Pausado = true;
          xSemaphoreGive(serialMutex);
        }
      } else if (command == "EstadoPCreanudar") {
        // Tomar el mutex para acceder a la variable sensorsPaused
        if (xSemaphoreTake(serialMutex, portMAX_DELAY) == pdTRUE) {
            gsrPausado = false;
            joystickPausado = false;
            ky039Pausado = false;
            ky038Pausado = false;
            ky013Pausado = false;
            encoderPausado = false;
            ky032Pausado = false;
            ky002Pausado = false;
            ky026Pausado = false;
            ky024Pausado = false;
          xSemaphoreGive(serialMutex);
        }
      }

    }
    vTaskDelay(10 / portTICK_PERIOD_MS); // Esperar antes de verificar
nuevamente
  }
}
```

Código Unity, control de comunicación serial

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO.Ports;
using System.Threading;

public class SerialManagerScript : MonoBehaviour
{
    //Variables para almacenar datos de los sensores
    private int GSRValue;
    private int JoystickXValue;
    private int JoystickYValue;
    private int ky039Value;
    private int ky038Value;
    private int ky013Value;
    private int ky026Value;
    private int ky032Value;
    private int ky002Value;
    private int ky024Value;
    private int PosicionValue;
    private int BotonEncoderValue;
    private int BotonJoystickValue;

    //Variables del puerto serial
    public string serialPortName = "COM4"; // Nombre del puerto serial
    public int baudRate = 9600; // Velocidad del puerto serial
    private SerialPort serialPort;
    private string receivedData;

    //Hilo para leer el puerto serial
    private Thread serialThread;

    private void Start()
    {
        //Inicializo el puerto serial
        OpenSerialPort();
        //Se lanza un hilo para la lectura de datos del puerto serial
        StartSerialThread();
        //Se inicializa Arduino con los datos necesarios
        SendDataToArduino("EstadoLpausar");
        SendDataToArduino("EstadoPCpausar");
        SendDataToArduino("MonitorEpausar");
        SendDataToArduino("ky032pausar");
        SendDataToArduino("joystickpausar");
        SendDataToArduino("encoderpausar");
        SendDataToArduino("ky024pausar");
        SendDataToArduino("ky002pausar");
        SendDataToArduino("ky038pausar");
        SendDataToArduino("EstadoLreanudar");

        }

    private void OnDestroy()
    {
        CloseSerialPort();
    }

    private void OpenSerialPort()
    {
        if (serialPort != null && serialPort.IsOpen)
        {
```

```
            Debug.LogWarning("El puerto serial se encuentra abierto.");
            return;
        }

        serialPort = new SerialPort(serialPortName, baudRate);
        try
        {
            serialPort.Open();
            Debug.Log("Puerto serial abierto.");
        }
        catch (System.Exception e)
        {
            Debug.LogError("Error al abrir el puerto: " + e.Message);
        }
    }

    void ParseSensorData(string data)
    {
        string[] parts = data.Split(':');
        if (parts.Length == 2)
        {
            string sensorName = parts[0].Trim();
            string sensorValueStr = parts[1].Trim();

            if (int.TryParse(sensorValueStr, out int sensorValue))
            {
                Debug.Log("Sensor: " + sensorName + ", Valor: " +
sensorValue);

                // Almacena el valor del sensor en la variable
correspondiente
                switch (sensorName)
                {
                    case "GSR":
                        GSRValue = sensorValue;
                        Debug.Log("GSR: " + GSRValue);
                        break;
                    case "JoystickX":
                        JoystickXValue = sensorValue;
                        Debug.Log("JoystickX: " + JoystickXValue);
                        break;
                    case "JoystickY":
                        JoystickYValue = sensorValue;
                        Debug.Log("JoystickY: " + JoystickYValue);
                        break;
                    case "ky039":
                        ky039Value = sensorValue;
                        Debug.Log("ky039: " + ky039Value);
                        break;
                    case "ky038":
                        ky038Value = sensorValue;
                        Debug.Log("ky038: " + ky038Value);
                        break;
                    case "Temperatura":
                        ky013Value = sensorValue;
                        Debug.Log("Temperatura: " + ky013Value);
                        break;
                    case "ky026":
                        ky026Value = sensorValue;
                        Debug.Log("ky026: " + ky026Value);
                        break;
                    case "ky032":
                        ky032Value = sensorValue;
```

```
                        Debug.Log("ky032: " + ky032Value);
                        break;
                    case "ky002":
                        ky002Value = sensorValue;
                        Debug.Log("ky002: " + ky002Value);
                        break;
                    case "ky024":
                        ky024Value = sensorValue;
                        Debug.Log("ky024: " + ky024Value);
                        break;
                    case "Posicion":
                        PosicionValue = sensorValue;
                        Debug.Log("Posicion: " + PosicionValue);
                        break;
                    case "botonE":
                        BotonEncoderValue = sensorValue;
                        Debug.Log("botonE: " + BotonEncoderValue);
                        break;
                    case "botonJ":
                        BotonJoystickValue = sensorValue;
                        Debug.Log("botonJ: " + BotonJoystickValue);
                        break;

                }
            }
        }
    }

    // Agrega métodos públicos para acceder a los valores de los sensores
desde otros objetos
    public int GetSensorValue(string data)
    {
        switch (data)
        {
            case "GSR":
                return GSRValue;
            case "JoystickX":
                return JoystickXValue;
            case "JoystickY":
                return JoystickYValue;
            case "ky039":
                return ky039Value;
            case "ky038":
                return ky038Value;
            case "Temperatura":
                return ky013Value;
            case "ky026":
                return ky026Value;
            case "ky032":
                return ky032Value;
            case "ky002":
                return ky002Value;
            case "ky024":
                return ky024Value;
            case "Posicion":
                return PosicionValue;
            case "botonE":
                return BotonEncoderValue;
            case "botonJ":
                return BotonJoystickValue;
        }
        return 0;
    }
```

```
private void CloseSerialPort()
{
    if (serialPort != null && serialPort.IsOpen)
    {
        serialPort.Close();
        Debug.Log("Puerto serial cerrado.");
    }
}

public void SendDataToArduino(string data)
{
    if (serialPort != null && serialPort.IsOpen)
    {
        serialPort.WriteLine(data);
        Debug.Log(data);
    }
    else
    {
        Debug.LogWarning("El puerto serial no está abierto");
    }
}

private void HandleObjectEvent(string message)
{
    switch (message)
    {
        case "MovimientoLibreEntrada":
            SendDataToArduino("EstadoLreanudar");
            break;
        case "PaneldeControlEntrada":
            SendDataToArduino("EstadoPCreanudar");
            break;
        case "PantallaVideosEntrada":
            SendDataToArduino("MonitorEreanudar");
            break;
        case "ZonaRobotMovilEntrada":
            SendDataToArduino("ky032reanudar");
            break;
        case "ZonaBrazoRobot1Entrada":
            SendDataToArduino("joystickreanudar");
            break;
        case "ZonaBrazoRobot2Entrada":
            SendDataToArduino("encoderreanudar");
            break;
        case "ZonaClasificacionEntrada":
            SendDataToArduino("ky024reanudar");
            break;
        case "ZonaVidriosEntrada":
            SendDataToArduino("ky002reanudar");
            break;
        case "ZonaLucesEntrada":
            SendDataToArduino("ky038reanudar");
            break;
        case "MovimientoLibreSalida":
            SendDataToArduino("EstadoLpausar");
            break;
        case "PaneldeControlSalida":
            SendDataToArduino("EstadoPCpausar");
            break;
        case "PantallaVideosSalida":
```

```
                SendDataToArduino("MonitorEpausar");
                break;
            case "ZonaRobotMovilSalida":
                SendDataToArduino("ky032pausar");
                break;
            case "ZonaBrazoRobot1Salida":
                SendDataToArduino("joystickpausar");
                break;
            case "ZonaBrazoRobot2Salida":
                SendDataToArduino("encoderpausar");
                break;
            case "ZonaClasificacionSalida":
                SendDataToArduino("ky024pausar");
                break;
            case "ZonaVidriosSalida":
                SendDataToArduino("ky002pausar");
                break;
            case "ZonaLucesSalida":
                SendDataToArduino("ky038pausar");
                break;
        }
    }

    // Inicia el hilo para la lectura del puerto serie
    private void StartSerialThread()
    {
        serialThread = new Thread(ReadSerialData);
        serialThread.IsBackground = true;
        serialThread.Start();
    }

    // Detiene el hilo de lectura del puerto serie
    private void StopSerialThread()
    {
        if (serialThread != null && serialThread.IsAlive)
        {
            serialThread.Interrupt();
            serialThread.Join();
        }
    }

    // Función que se ejecuta en el hilo aparte para leer el puerto serie
    private void ReadSerialData()
    {
        while (serialPort.IsOpen)
        {
            try
            {
                receivedData = serialPort.ReadLine();
                // Procesa los datos recibidos, por ejemplo, llamando a
ParseSensorData(receivedData);
                ParseSensorData(receivedData);
            }
            catch (System.Exception e)
            {
                Debug.LogWarning("Error al leer datos del puerto serie: " +
e.Message);
            }
        }
    }
}
```

Ejemplo de código Unity de solicitud de datos al manejador de conexiones

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

public class PaneldeControl : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            SendMessageToSerialManager("PaneldeControlEntrada");
            SendMessageToSerialManager("MovimientoLibreSalida");
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            SendMessageToSerialManager("PaneldeControlSalida");
            SendMessageToSerialManager("MovimientoLibreEntrada");
        }
    }

    private void SendMessageToSerialManager(string message)
    {
        // Encuentra el objeto SerialManager en la escena
        GameObject serialManagerObj = GameObject.Find("SerialManager");
        if (serialManagerObj != null)
        {
            // Envía un mensaje al SerialManager para informarle del evento
            serialManagerObj.SendMessage("HandleObjectEvent", message);
        }
    }
}
```

Código Unity para activar el objeto aspersor del entorno

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

public class ActivarAspersor : MonoBehaviour
{
    private bool hijosActivados = false;
    private SerialManagerScript serialManager; // Referencia al
SerialManager
    private int ky026Value = 0;

    void Start()
    {
        serialManager = GameObject.FindObjectOfType<SerialManagerScript>();
// Busca el tipo de objeto SerialManagerScript
        ActualizarEstadoHijos();
    }

    void Update()
    {
        ky026Value = serialManager.GetSensorValue("ky026"); //Lee los datos
del sensor ky026

        if (ky026Value == 1)
        {
            hijosActivados = true; //Enciende los aspersores
            ActualizarEstadoHijos();
        }
        if (ky026Value == 0)
        {
            hijosActivados = false; //Apaga los aspersores
            ActualizarEstadoHijos();
        }
    }

    void ActualizarEstadoHijos()
    {
        foreach (Transform hijo in transform)
        {
            hijo.gameObject.SetActive(hijosActivados);
        }
    }
}
```

Código Unity para activar el objeto luces activas del entorno

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

public class ControlLuces : MonoBehaviour
{
    private int valorSensor; // Valor del sensor de Arduino
    private SerialManagerScript serialManager; // Referencia al
SerialManager
    private bool enZona = false;

    public GameObject Lampara1;
    public GameObject Lampara2;
    public GameObject Lampara3;
    public GameObject Lampara4;
    public GameObject Lampara5;

    private bool lampara1Activada = false;
    private bool lampara2Activada = false;
    private bool lampara3Activada = false;
    private bool lampara4Activada = false;
    private bool lampara5Activada = false;

    // Start is called before the first frame update
    void Start()
    {
        serialManager = GameObject.FindObjectOfType<SerialManagerScript>();
        Lampara1.SetActive(false);
        Lampara2.SetActive(false);
        Lampara3.SetActive(false);
        Lampara4.SetActive(false);
        Lampara5.SetActive(false);
    }

    // Update is called once per frame
    void Update()
    {
        valorSensor = serialManager.GetSensorValue("ky038");

        if (enZona)
        {
            if (valorSensor < 650)
            {
                DesactivarLamparas();
            }
            else if (valorSensor >= 650 && valorSensor <= 700)
            {
                CambiarLamparasConRetardo(ActivarLampara1);
            }
            else if (valorSensor > 700 && valorSensor <= 750)
            {
                CambiarLamparasConRetardo(() =>
                {
                    ActivarLampara1();
                    ActivarLampara2();
                });
            }
            else if (valorSensor > 750 && valorSensor <= 800)
            {
                CambiarLamparasConRetardo(() =>
                {
```

```
                ActivarLampara1();
                ActivarLampara2();
                ActivarLampara3();
            });
        }
        else if (valorSensor > 800 && valorSensor <= 850)
        {
            CambiarLamparasConRetardo(() =>
            {
                ActivarLampara1();
                ActivarLampara2();
                ActivarLampara3();
                ActivarLampara4();
            });
        }
        else if (valorSensor > 850)
        {
            CambiarLamparasConRetardo(() =>
            {
                ActivarLampara1();
                ActivarLampara2();
                ActivarLampara3();
                ActivarLampara4();
                ActivarLampara5();
            });
        }
    }
    else if (!enZona)
    {
        CambiarLamparasConRetardo(DesactivarLamparas);
    }
}
private void ActivarLampara1()
{
    if (!lampara1Activada)
    {
        Lampara1.SetActive(true);
        lampara1Activada = true;
    }
}
private void ActivarLampara2()
{
    if (!lampara2Activada)
    {
        Lampara2.SetActive(true);
        lampara2Activada = true;
    }
}
private void ActivarLampara3()
{
    if (!lampara3Activada)
    {
        Lampara3.SetActive(true);
        lampara3Activada = true;
    }
}
private void ActivarLampara4()
{
    if (!lampara4Activada)
    {
        Lampara4.SetActive(true);
        lampara4Activada = true;
    }
```

```
    }
    private void ActivarLampara5()
    {
        if (!lampara5Activada)
        {
            Lampara5.SetActive(true);
            lampara5Activada = true;
        }
    }
    private void DesactivarLamparas()
    {
        if (lampara1Activada || lampara2Activada || lampara3Activada ||
lampara4Activada || lampara5Activada)
        {
            Lampara1.SetActive(false);
            Lampara2.SetActive(false);
            Lampara3.SetActive(false);
            Lampara4.SetActive(false);
            Lampara5.SetActive(false);

            // Actualiza los estados de las lámparas
            lampara1Activada = false;
            lampara2Activada = false;
            lampara3Activada = false;
            lampara4Activada = false;
            lampara5Activada = false;
        }
    }
    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = true;
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = false;
        }
    }
    private bool cambiandoLamparas = false;
    private float retardoCambioLamparas = 0.5f; // Retardo en segundos

    private void CambiarLamparasConRetardo(System.Action accion)
    {
        if (!cambiandoLamparas)
        {
            cambiandoLamparas = true;
            Invoke("RealizarAccionConRetardo", retardoCambioLamparas);
            accion();
        }
    }

    private void RealizarAccionConRetardo()
    {
        cambiandoLamparas = false;
    }
}
```

Ejemplo de código Unity para activar animación del objeto estación de mecanizado

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem; // Importar el namespace necesario

public class ControlAnimBanda : MonoBehaviour
{
    //Referencia a los objetos a controlar
    public GameObject BrazoRobot; // Referencia al primer objeto
    public GameObject Maquina1; // Referencia al segundo objeto
    public GameObject Maquina2; // Referencia al tercer objeto
    public GameObject Pieza1; // Referencia al cuarto objeto
    public GameObject Pieza2; // Referencia al quinto objeto
    public GameObject Objeto1; // Referencia al sexto objeto
    public GameObject Objeto2; // Referencia al septimo objeto

    private Animator primerAnimator; // Animator del primer objeto
    private Animator segundoAnimator; // Animator del segundo objeto
    private Animator tercerAnimator; // Animator del segundo objeto
    private Animator cuartoAnimator; // Animator del segundo objeto
    private Animator quintoAnimator; // Animator del segundo objeto
    private Animator sextoAnimator; // Animator del segundo objeto
    private Animator septimoAnimator; // Animator del segundo objeto

    private int valorSensor; // Este valor lo obtienes de tus datos de
Arduino
    private SerialManagerScript serialManager; // Referencia al
SerialManager
    private bool enZona;

    private void Start()
    {
        // Obtener los componentes Animator de los objetos
        primerAnimator = BrazoRobot.GetComponent<Animator>();
        segundoAnimator = Maquina1.GetComponent<Animator>();
        tercerAnimator = Maquina2.GetComponent<Animator>();
        cuartoAnimator = Pieza1.GetComponent<Animator>();
        quintoAnimator = Pieza2.GetComponent<Animator>();
        sextoAnimator = Objeto1.GetComponent<Animator>();
        septimoAnimator = Objeto2.GetComponent<Animator>();

        serialManager = GameObject.FindObjectOfType<SerialManagerScript>();

    }

    private void Update()
    {
        valorSensor = serialManager.GetSensorValue("ky024");
        if (enZona == true && valorSensor == 0 &&
(segundoAnimator.GetCurrentAnimatorStateInfo(0).IsName("PuertasQuietasMaquin
a1") &&
            primerAnimator.GetCurrentAnimatorStateInfo(0).normalizedTime >=
1))
        {
            ActivaPieza1();

        }
```

```
        // Ejecutar la primera animación del segundo objeto al presionar la
tecla "M"
        if (enZona == true && valorSensor == 1 &&
(tercerAnimator.GetCurrentAnimatorStateInfo(0).IsName("Puerta1Maquina2Quieta
") &&
            primerAnimator.GetCurrentAnimatorStateInfo(0).normalizedTime >=
1))
        {
            ActivaPieza2();

        }

    }

    private void ActivaPieza1()
    {
        // Activar la cuarta animación en el segundo objeto
        cuartoAnimator.SetBool("ActivaCubo", true);
    }

    private void ActivaPieza2()
    {
        // Activar la cuarta animación en el segundo objeto
        quintoAnimator.SetBool("ActivaPieza", true);
    }

    private IEnumerator DelayedAction()
    {
        Debug.Log("Start of Delayed Action");
        yield return new WaitForSecondsRealtime(2f); // Espera durante 2
segundos en tiempo real
        Debug.Log("Delayed Action Complete");
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = true;
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = false;
        }
    }
}
```

Código Unity para activar el objeto soporte vidrios

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

public class DestruirVidrio : MonoBehaviour
{
    public List<GameObject> objetosCompletos;
    public List<GameObject> objetosDestruidos;
    private Vector3[] objetosDestruidosInitialPositions;

    private bool enZona = false;
    private int valorSensor; // Valor del sensor Arduino
    private SerialManagerScript serialManager; // Referencia al
SerialManager

    private Dictionary<Transform, Vector3> initialPositionsDict = new
Dictionary<Transform, Vector3>();

    private int currentObjectIndex = 0;
    private bool isDestroying = false;
    private float timeSinceLastDestruction = 0f;
    private bool allObjectsDestroyed = false;

    public float restorationDelay = 5f;

    private void Start()
    {
        serialManager = GameObject.FindObjectOfType<SerialManagerScript>();
        InitializeInitialPositions();
        RestoreAllObjects(); // Esto restaura todos los objetos al inicio
    }

    private void Update()
    {
        valorSensor = serialManager.GetSensorValue("ky002");
        if (enZona && !isDestroying && valorSensor == 1)
        {
            StartCoroutine(DestroyNextObject());
        }

        if (allObjectsDestroyed)
        {
            timeSinceLastDestruction += Time.deltaTime;
            if (timeSinceLastDestruction >= restorationDelay)
            {
                RestoreAllObjects();
            }
        }
    }
    private IEnumerator DestroyNextObject()
    {
        isDestroying = true;

        if (currentObjectIndex < objetosCompletos.Count)
        {
            objetosCompletos[currentObjectIndex].SetActive(false);
            objetosDestruidos[currentObjectIndex].SetActive(true);
            currentObjectIndex++;
        }
```

```
        yield return new WaitForSeconds(0.5f);
        isDestroying = false;
        if (currentObjectIndex >= objetosCompletos.Count)
        {
            allObjectsDestroyed = true;
            timeSinceLastDestruction = 0f;
        }
    }

    private void RestoreAllObjects()
    {
        allObjectsDestroyed = false;
        currentObjectIndex = 0;

        for (int i = 0; i < objetosDestruidos.Count; i++)
        {
            objetosCompletos[i].SetActive(true);
            objetosDestruidos[i].SetActive(false);

            Transform[] allTransforms =
objetosDestruidos[i].GetComponentsInChildren<Transform>();

            foreach (Transform childTransform in allTransforms)
            {
                if (initialPositionsDict.TryGetValue(childTransform, out
Vector3 initialPosition))
                {
                    childTransform.position = initialPosition;
                }
            }
        }
    }
    private void InitializeInitialPositions()
    {
        foreach (GameObject objDestruido in objetosDestruidos)
        {
            Transform[] allTransforms =
objDestruido.GetComponentsInChildren<Transform>();

            foreach (Transform childTransform in allTransforms)
            {
                initialPositionsDict[childTransform] =
childTransform.position;
            }
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = true;
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = false;
        }
    }
}
```

Código Unity para controlar brazo robótico con joystick

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

public class ControlBrazoRobotico1 : MonoBehaviour
{
    public Transform[] articulaciones; // Arreglo de referencias a las
articulaciones
    public float velocidadRotacion = 30f; // Velocidad de rotación de la
articulación
    public int valorJoystickMin = 0; // Valor mínimo del joystick
    public int valorJoystickMax = 1024; // Valor máximo del joystick
    private bool enZona = false;

    private int contador = 0;
    private int valorBotonAnterior = 0;

    private SerialManagerScript serialManager; // Referencia al
SerialManager

    void Start()
    {
        serialManager = GameObject.FindObjectOfType<SerialManagerScript>();
    }

    void Update()
    {
        int valorJoystickX = serialManager.GetSensorValue("JoystickX");
        int valorJoystickY = serialManager.GetSensorValue("JoystickY");
        int valorBoton = serialManager.GetSensorValue("botonJ");

        // Mapea el valor del joystick al rango [-1, 1]
        float valorJoystickMapeadoX = Mathf.InverseLerp(valorJoystickMin,
valorJoystickMax, valorJoystickX) * 2f - 1f;
        float valorJoystickMapeadoY = Mathf.InverseLerp(valorJoystickMin,
valorJoystickMax, valorJoystickY) * 2f - 1f;

        // Calcula la rotación en función del valor mapeado y la velocidad
        float rotacion1 = valorJoystickMapeadoX * velocidadRotacion *
Time.deltaTime;
        float rotacion2 = valorJoystickMapeadoY * velocidadRotacion *
Time.deltaTime;

        if (enZona && valorBoton == 1 && valorBotonAnterior == 0)
        {
            // Cambia de articulación aquí
            CambiarArticulacion();
        }

        valorBotonAnterior = valorBoton;

        if (enZona && contador == 0)
        {
            RotarPrimeraArticulacion(rotacion1);
            RotarSegundaArticulacion(rotacion2);
            Debug.Log("contador: " + contador);
        }

        if (enZona && contador == 1)
        {
```

```
            RotarTerceraArticulacion(rotacion1);
            RotarCuartaArticulacion(rotacion2);
            Debug.Log("contador: " + contador);
        }
        if (enZona && contador == 2)
        {
            RotarQuintaArticulacion(rotacion1);
            Debug.Log("contador: " + contador);
        }
        if (contador > 2)
        {
            contador = 0;
        }
    }
    void RotarPrimeraArticulacion(float rotacion)
    {
        articulaciones[0].Rotate(Vector3.up, rotacion);
    }
    void RotarSegundaArticulacion(float rotacion)
    {
        articulaciones[1].Rotate(Vector3.right, rotacion);
    }
    void RotarTerceraArticulacion(float rotacion)
    {
        articulaciones[2].Rotate(Vector3.right, rotacion);
    }
    void RotarCuartaArticulacion(float rotacion)
    {
        articulaciones[3].Rotate(Vector3.up, rotacion);
    }
    void RotarQuintaArticulacion(float rotacion)
    {
        articulaciones[4].Rotate(Vector3.right, rotacion);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = true;
        }
    }
    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "Player")
        {
            enZona = false;
        }
    }
    void CambiarArticulacion()
    {
        contador++; // Incrementa el contador o realiza el cambio deseado
    }
}
```

Código Unity para controlar brazo robótico con encoder

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

public class ControlBrazoRobot2 : MonoBehaviour
{
    public Transform[] articulaciones; // Arreglo de referencias a las
articulaciones
    public float velocidadRotacion = 30f; // Velocidad de rotación de la
articulación
    private int valorEncoderAnterior = 0; // Valor anterior del encoder
    private bool enZona = false;
    private int contador = 0;
    private int valorBotonAnterior = 0; // Valor anterior del botón
    private SerialManagerScript serialManager; // Referencia al
SerialManager
    void Start()
    {
        serialManager = GameObject.FindObjectOfType<SerialManagerScript>();
    }
    void Update()
    {
        int valorEncoder = serialManager.GetSensorValue("Posicion");
        int valorBoton = serialManager.GetSensorValue("botonE");

        // Calcula la diferencia de pulsos desde el frame anterior
        int diferenciaPulsos = (valorEncoder - valorEncoderAnterior)*100;

        // Calcula la rotación en grados (ajusta según tus necesidades)
        float rotacionGrados = diferenciaPulsos * velocidadRotacion *
Time.deltaTime;
        if (enZona && valorBoton == 1 && valorBotonAnterior == 0)
        {
            CambiarArticulacion();
        }
        valorBotonAnterior = valorBoton;
        //valorBotonAnterior = valorBoton;
        if (enZona && contador == 0)
        {
            RotarPrimeraArticulacion(rotacionGrados);

        }
        if (enZona && contador == 1)
        {
            RotarSegundaArticulacion(rotacionGrados);
        }
        if (enZona && contador == 2)
        {
            RotarTerceraArticulacion(rotacionGrados);
        }
        if (enZona && contador == 3)
        {
            RotarCuartaArticulacion(rotacionGrados);
        }
        if (enZona && contador == 4)
        {
            RotarQuintaArticulacion(rotacionGrados);
        }
        if (enZona && contador == 5)
        {
```

```
                RotarSextaArticulacion(rotacionGrados);
            }
            valorEncoderAnterior = valorEncoder;

            if (contador > 5)
            {
                contador = 0;
            }
        }
        void RotarPrimeraArticulacion(float rotacion)
        {
            articulaciones[0].Rotate(Vector3.up, rotacion);
        }

        void RotarSegundaArticulacion(float rotacion)
        {
            articulaciones[1].Rotate(Vector3.up, rotacion);
        }

        void RotarTerceraArticulacion(float rotacion)
        {
            articulaciones[2].Rotate(Vector3.up, rotacion);
        }

        void RotarCuartaArticulacion(float rotacion)
        {
            articulaciones[3].Rotate(Vector3.up, rotacion);
        }
        void RotarQuintaArticulacion(float rotacion)
        {
            articulaciones[4].Rotate(Vector3.right, rotacion);
        }

        void RotarSextaArticulacion(float rotacion)
        {
            articulaciones[5].Rotate(Vector3.right, rotacion);
        }
        private void OnTriggerEnter(Collider other)
        {
            if (other.tag == "Player")
            {
                enZona = true;
            }
        }
        private void OnTriggerExit(Collider other)
        {
            if (other.tag == "Player")
            {
                enZona = false;
            }
        }

        void CambiarArticulacion()
        {
            contador++; // Incrementa el contador
        }
    }
```

Código Unity para controlar robot móvil

```
using System.Collections;
using UnityEngine;
using UnityEngine.AI;

public class RobotController : MonoBehaviour
{
    public Transform[] waypoints; // Arreglo para almacenar los puntos de
destino del robot.
    public float avoidObstacleDistance = 2.0f; // Distancia a la que el
robot detecta un obstáculo.
    public float avoidanceDuration = 1.0f; // Duración de la evasión cuando
el robot se queda atascado.

    private NavMeshAgent agent;
    private int currentWaypointIndex = 0; // Índice del punto de destino
actual.
    private bool isAvoidingObstacle = false; // Bandera para detectar si el
robot está evitando un obstáculo.

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();

        // Verifica si el arreglo waypoints no está vacío y tiene al menos
un punto de destino.
        if (waypoints != null && waypoints.Length > 0)
        {
            // Establece el punto de destino inicial del robot.
            agent.SetDestination(waypoints[currentWaypointIndex].position);
        }
        else
        {
            Debug.LogError("El arreglo de waypoints está vacío o no contiene
ningún punto de destino.");
        }
    }

    void Update()
    {
        if (!isAvoidingObstacle)
        {
            // Si el robot ha llegado al punto de destino actual,
            // establece el siguiente punto de destino en el arreglo de
waypoints.
            if (waypoints != null && waypoints.Length > 0 &&
!agent.pathPending && agent.remainingDistance < 0.2f)
            {
                currentWaypointIndex = (currentWaypointIndex + 1) %
waypoints.Length;

agent.SetDestination(waypoints[currentWaypointIndex].position);
            }
        }
    }

    void OnCollisionEnter(Collision collision)
    {
        // Si el robot colisiona con un objeto, comprueba si es un
obstáculo.
        if (collision.gameObject.CompareTag("Obstacle"))
        {
```

```
            // Si el robot colisiona con un obstáculo, inicia la rutina para
evitar el obstáculo.
            StartCoroutine(AvoidObstacleCoroutine());
        }
    }

    IEnumerator AvoidObstacleCoroutine()
    {
        // Inicia la bandera para evitar obstáculos.
        isAvoidingObstacle = true;

        // Detén el NavMeshAgent para evitar colisiones adicionales mientras
retrocedes.
        agent.isStopped = true;

        // Guarda la posición actual del robot.
        Vector3 originalPosition = transform.position;

        // Retrocede durante el tiempo especificado para evitar el
obstáculo.
        Vector3 obstacleDirection = transform.position - agent.destination;
        obstacleDirection.y = 0f;
        Vector3 avoidancePosition = transform.position +
obstacleDirection.normalized * avoidObstacleDistance;

        // Haz que el robot mire en la dirección opuesta al obstáculo.
        Vector3 lookAtPosition = transform.position +
obstacleDirection.normalized * -1f;
        transform.LookAt(lookAtPosition);

        // Retrocede durante el tiempo de evasión.
        float timer = 0f;
        while (timer < avoidanceDuration)
        {
            transform.position = Vector3.MoveTowards(transform.position,
avoidancePosition, Time.deltaTime * agent.speed);
            timer += Time.deltaTime;
            yield return null;
        }

        // Vuelve a la posición original y reanuda el NavMeshAgent.
        transform.position = originalPosition;
        agent.isStopped = false;

        // Desactiva la bandera para dejar de evitar obstáculos.
        isAvoidingObstacle = false;
    }
}
```

Código Unity para el sensor de temperatura y los colores de la pantalla de jugador

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

public class ValorTemp : MonoBehaviour
{

    public TMP_Text valorSensorText; // Se arrastra el objeto de texto que
se quiere modificar
    private int valorTemp; // Este valor lo obtienes los datos de Arduino
    private SerialManagerScript serialManager; // Referencia al
SerialManager

    void Start()
    {
        serialManager = GameObject.FindObjectOfType<SerialManagerScript>();
        //Se invoca el metodo cada 1s
        InvokeRepeating("ActualizarValorDelSensor", 0f, 1f);
    }

    private void ActualizarValorDelSensor()
    {
        valorTemp = serialManager.GetSensorValue("Temperatura");
        valorSensorText.text = valorTemp.ToString() + " °C";

    }
}

////////////Otro Script control de color///////////////

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

public class CambioColor : MonoBehaviour
{
    public Image fondoImagen; // Arrastra el objeto Image que representa el
fondo desde el Inspector de Unity
    private int valorTemp; // Este valor lo obtienes de tus datos de Arduino
    private SerialManagerScript serialManager; // Referencia al
SerialManager
    public Color colorFrio;
    public Color colorCaliente;
    public float transparenciaDefault = 0.5f; // Valor de transparencia
predeterminado
    public float smoothness = 5f; // Valor de suavizado

    private float currentTransparency = 0f;

    void Start()
    {
        // Busca una instancia de SerialManagerScript en la escena.
        serialManager = FindObjectOfType<SerialManagerScript>();

        if (serialManager == null)
```

```
        {
            Debug.LogError("No se encontró un objeto SerialManagerScript en
la escena.");
        }
    }

    private void Update()
    {
        if (serialManager != null)
        {
            // Supongamos que obtienes la temperatura de alguna manera, por
ejemplo, desde SerialManager
            valorTemp = serialManager.GetSensorValue("Temperatura");

            // Calcula el valor normalizado de la temperatura en el rango 0
a 1
            float valorNormalizado = Mathf.InverseLerp(5, 48, valorTemp);

            // Interpola entre dos colores basados en el valor normalizado
            Color colorFondo = Color.Lerp(colorFrio, colorCaliente,
valorNormalizado);

            // Aplica el color al fondo
            fondoImagen.color = colorFondo;

            // Calcula la transparencia en función de la temperatura
            float targetTransparency;

            if (valorTemp >= 16 && valorTemp <= 24)
            {
                // Si la temperatura está entre 20 y 24 grados, la
transparencia es 0
                targetTransparency = 0f;
            }
            else
            {
                // En cualquier otro caso, usa el valor de transparencia
predeterminado
                targetTransparency = transparenciaDefault;
            }

            // Aplica el suavizado a la transparencia
            currentTransparency = Mathf.Lerp(currentTransparency,
targetTransparency, Time.deltaTime * smoothness);

            // Aplica el color con transparencia al fondo
            Color colorConTransparencia = new Color(colorFondo.r,
colorFondo.g, colorFondo.b, currentTransparency);
            fondoImagen.color = colorConTransparencia;
        }
    }
}
```

Código Unity para activar la pantalla del panel de control

```
using System.Collections;
using System.Collections.Generic;
using TMPro.Examples;
using UnityEngine;
using UnityEngine.InputSystem; // Agregar el namespace para el nuevo sistema
de manejo de entradas.

public class ControlPantallaSensores : MonoBehaviour
{
    public GameObject pantalla;
    public GameObject camara;
    private bool enZona;
    private bool activa;

    private void Update()
    {
        if (enZona)
        {
            if (Keyboard.current.eKey.wasPressedThisFrame)
            {
                activa = !activa;
                pantalla.SetActive(activa);
                camara.SetActive(!activa);

                if (activa)
                {
                    Time.timeScale = 0f; // Detener todas las simulaciones
en el juego
                    Cursor.lockState = CursorLockMode.None; // Liberar el
cursor
                    Cursor.visible = true; // Hacer visible el cursor
                }
                else
                {
                    Time.timeScale = 1f; // Reanudar todas las simulaciones
en el juego
                    Cursor.lockState = CursorLockMode.Locked; // Bloquear el
cursor
                    Cursor.visible = false; // Ocultar el cursor
                }
            }
        }
        else
        {
            pantalla.SetActive(false);
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            enZona = true;
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            enZona = false;
```

```
            pantalla.SetActive(false);
            Time.timeScale = 1f; // Asegurarse de reanudar todas las
simulaciones al salir de la zona
            Cursor.lockState = CursorLockMode.Locked; // Restaurar el
bloqueo del cursor
            Cursor.visible = false; // Restaurar la invisibilidad del cursor
        }

    }

    public void OnPanelButtonPressed()
    {
        activa = !activa;
        pantalla.SetActive(activa);
        camara.SetActive(!activa);

        if (activa)
        {
            Time.timeScale = 0f; // Detener todas las simulaciones en el
juego
            Cursor.lockState = CursorLockMode.None; // Liberar el cursor
            Cursor.visible = true; // Hacer visible el cursor
        }
        else
        {
            Time.timeScale = 1f; // Reanudar todas las simulaciones en el
juego
            Cursor.lockState = CursorLockMode.Locked; // Bloquear el cursor
            Cursor.visible = false; // Ocultar el cursor
        }
    }
}
```