Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Simulador de robótica educativa basado en Unity para el fomento del pensamiento computacional

Unity-based educational robotics simulator for computational thinking.

Edwin Plasencia Hernández

La Laguna, *11* de *julio* de 2024

D. **Eduardo Manuel Segredo González,** profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Rafael Arnay del Arco,** profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

“Simulador de robótica educativa basado en Unity para el fomento del pensamiento computacional”

ha sido realizada bajo su dirección por D. **Edwin Plasencia Hernández**.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de julio de 2024.

Agradecimientos

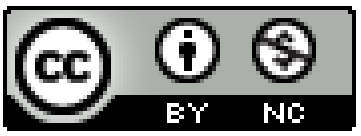
A mis tutores del Trabajo de Fin de Grado, Eduardo Manuel Segredo González y Rafael Arnay del Arco, por su apoyo y orientación a lo largo del proyecto.

A mi madre y mi hermano, por estar ahí siempre para apoyarme y hacerme sentir querido, por estar en los momentos de estrés, confiar en mí y apoyarme.

A mi profesora de Interfaces Inteligentes, Isabel Sánchez Berriel, por enseñarme todo lo que sé de Unity y ser excelente en lo que hace.

A todos los profesores que me han dado clase a lo largo de mi estancia en la Universidad de La Laguna, por darme la oportunidad de llegar a donde he llegado, enseñándome todo lo que saben día a día.

Licencia



Resumen

El objetivo de este trabajo ha sido la actualización y mejora de un simulador de robótica educativa controlado a través de un lenguaje de programación visual basado en bloques, para la solución de problemas en un espacio tridimensional usando unos robots programables por el usuario.

En esta memoria se explica cómo se ha mejorado la aplicación con la adición de un editor de niveles para que el usuario pueda crear sus propios desafíos, cómo se han implementado elementos interactivos para una mayor complejidad y profundidad de niveles, el proceso de refactorización e implementación de control múltiple simultáneo de robots, la inserción de un sistema de comunicaciones entre robots y un sistema de calificación de resultados así como importación y exportación de desafíos para una distribución más sencilla de niveles personalizados.

El simulador puede medir la eficiencia obtenida por el usuario para completar los desafíos, midiendo la cantidad de código empleado y la velocidad de completitud de los niveles.

Este simulador ha sido diseñado en la plataforma Unity-2022.3.27f1 con el objetivo de exportarlo e insertarlo en una página web para el fácil acceso de futuros alumnos y profesores, con el objetivo de educar y fomentar el pensamiento computacional.

Palabras clave: Simulador, Robótica educativa, Sensor, Desafío.

Abstract

The objective of this project has been the update and improvement of an educational robotics simulator controlled through a visual programming language based on blocks, for the solution of problems in a three-dimensional space using user-programmable robots.

This report explains how the application has been improved with the addition of a level editor so that the user can create his own challenges, how interactive elements have been implemented for greater complexity and depth of levels, the refactoring process and implementation of simultaneous multiple control of robots, the insertion of a communication system between robots and a results scoring system as well as import and export of challenges for easier distribution of customized levels.

The simulator can measure the efficiency obtained by the user to complete the challenges, measuring the amount of code used and the speed of completion of the levels.

This simulator has been designed on the Unity-2022.3.27f1 platform with the objective of exporting and embedding it in a web page for easy access by future students and teachers, with the goal of educating and encouraging computational thinking.

Keywords: Simulator, Educational Robotics, Sensor, Challenge.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1 Introducción

El trabajo de fin de grado que se ha realizado, consiste en un simulador de robots, los cuales son controlables por el jugador, que debe usar un lenguaje de programación visual basado en bloques para establecer el comportamiento de cada uno.

El jugador deberá controlar a los robots para lograr un objetivo, que es el completado de un nivel o desafío. Dichos niveles pueden ser completados llevando a un robot a un punto específico del espacio tridimensional del nivel, más comúnmente denominado como meta, bandera o final.

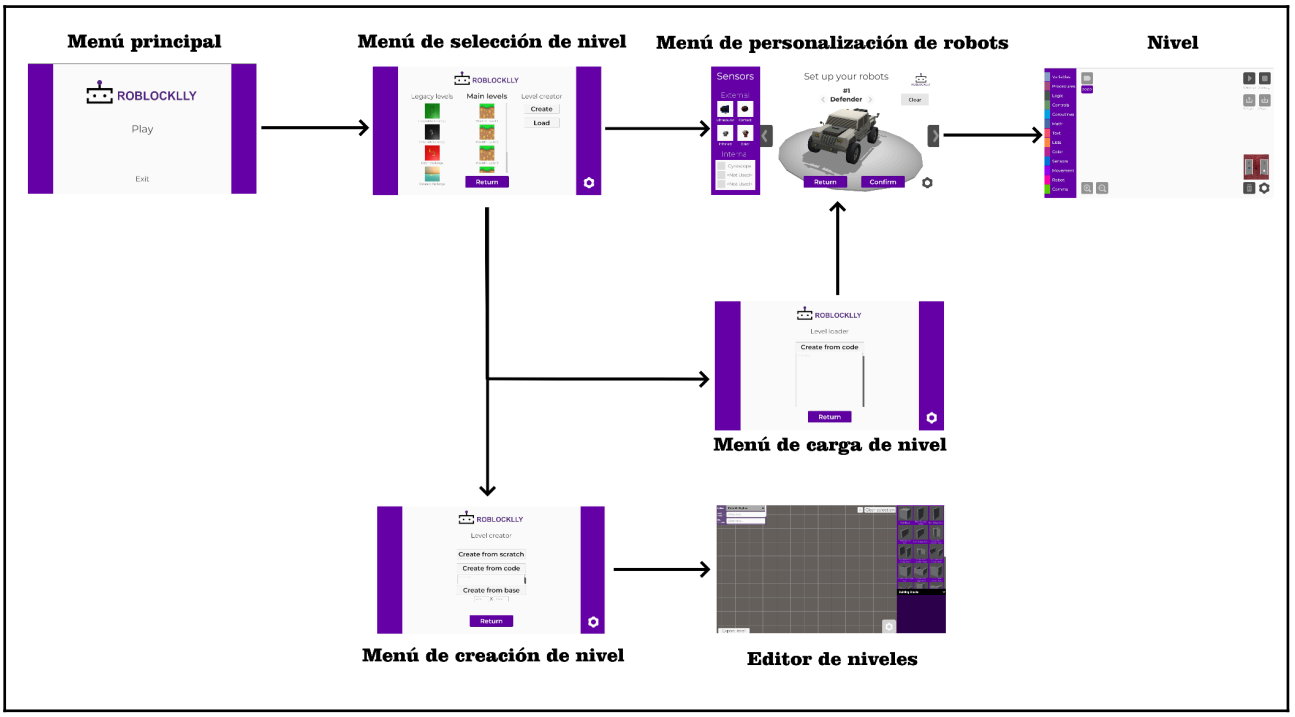


Figura 1.1: Esquema de organización de la aplicación

En la Figura 1.1, podemos ver un esquema del funcionamiento del proyecto. Primero, podemos seleccionar un nivel del menú de selección de nivel, así como cargar uno personalizado o crear uno desde cero. Luego, una vez se ha seleccionado el nivel, podemos seleccionar nuestros robots y modificarlos instalando sensores. Por último, tenemos la zona de trabajo de los niveles, donde podemos programar el funcionamiento de los robots y jugar los desafíos implementados en el proyecto.

Por otra parte, si seleccionamos cargar un nivel, tendremos el menú de carga para introducir nuestro código de generación y si vamos a la pantalla de creación, podremos acceder al editor de niveles, donde crearemos nuestro desafío desde cero y lo podremos compartir con otros usuarios.

Esta aplicación se ha desarrollado en el entorno gráfico de Unity [1] y aunque el proyecto se creó originalmente en la versión 2020 del producto, fué finalizado en la versión 2022 del mismo. Unity es un motor de videojuegos multiplataforma programado en C++ y C#, reconocido mundialmente por su uso gratuito para el desarrollo de juegos, aplicaciones y experiencias 3D en tiempo real, usado en el ámbito del entretenimiento, automóviles, cine, arquitectura y mucho más. Pueden implementarse escenas, que son espacios, ya sean 2D o 3D, que pueden llenarse con imágenes, audio, animaciones, modelos 3D y más, que se pueden modificar mediante código para lograr prácticamente cualquier objetivo y representar detalladamente la visión del diseñador/programador/desarrollador.

Para controlar a los robots, como ya se ha mencionado, se hace uso de un lenguaje de programación visual basado en bloques llamado Ublockly [2]. En este tipo de lenguajes, los bloques que pueden ordenarse y conectarse entre sí, con código embebido en los mismos que provoca un comportamiento predefinido al ser activados. Dichos bloques son activados o ejecutados de arriba hacia abajo en orden y pueden recibir y/o devolver valores o instrucciones. Cabe destacar que no contienen código de un lenguaje de programación clásico como pudiese ser C o JavaScript, sino que tienen un texto de simple lectura con la explicación de lo que hace dicho bloque, para una comprensión veloz y sencilla del mismo. Esto es ideal para un aprendizaje rápido del uso de dichos bloques, sobre todo para los posibles estudiantes a los que está dirigido este proyecto, ya que, como se destaca en el título de esta memoria, el objetivo de la aplicación es estimular el pensamiento computacional.

Jeannette Wing, informática teórica estadounidense, publicaba en 2006 “Computational Thinking” [3], en el que explicaba cómo el pensamiento computacional es una metodología para resolver problemas, diseñar sistemas y comprender el comportamiento humano basada en los principios fundamentales de la informática, tales como la descomposición, el reconocimiento de patrones, la abstracción y la elaboración de algoritmos. Se trata pues de un conjunto de habilidades que todos, no solo los ingenieros informáticos, deberían aprender y usar.

Volviendo al tema anterior, un ejemplo de bloque que podemos encontrar en el proyecto es el bloque de operaciones, éste recibe a su vez dos bloques numéricos que representan un valor matemático y un símbolo de operación. Dados estos bloques y el símbolo, al ejecutarse, el bloque devolverá el valor de la operación entre los bloques numéricos, que puede ser usado a su vez por otro bloque para realizar otra operación o funcionalidad.

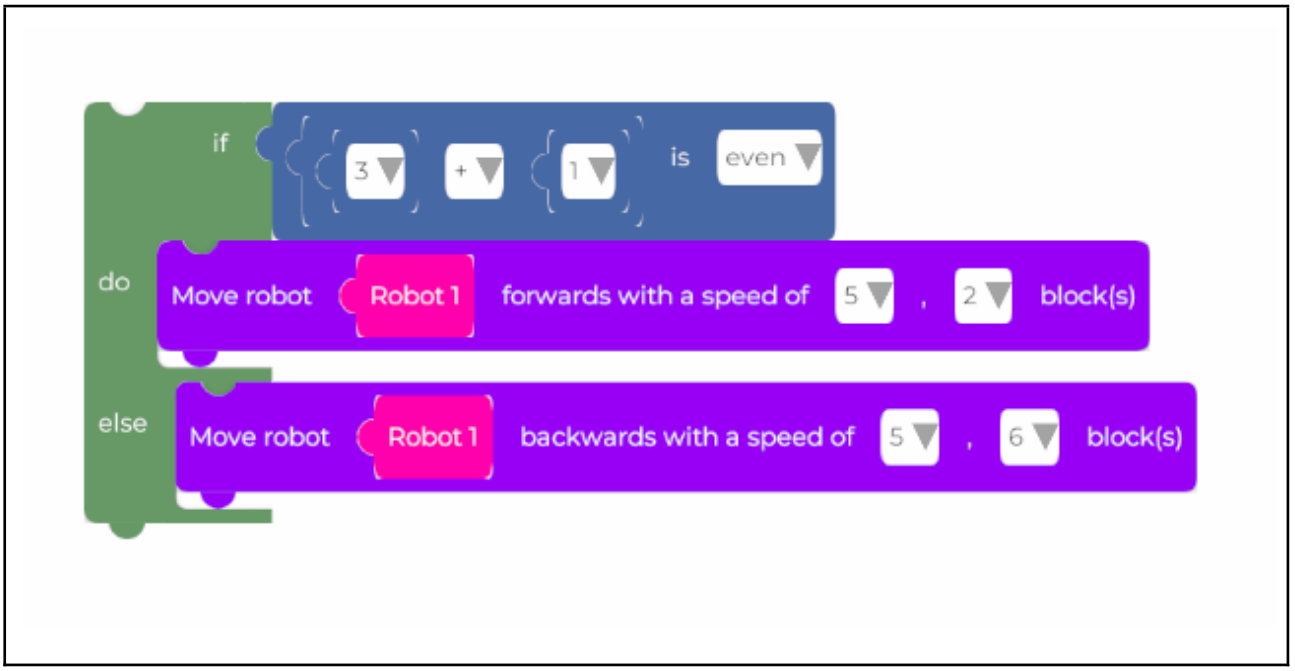


Figura 1.2: Ejemplo de programa simple

Un ejemplo de programa simple se puede observar en la Figura 1.2. Si analizamos el programa desarrollado, veremos que realiza las siguientes instrucciones:

- Primero calcula la operación 3 + 1
- Si dicha operación da como resultado un número par:
    - Mover al robot número 1 hacia delante con una velocidad de 5 unidades, 2 bloques de distancia.
- En caso de que el resultado sea impar:
    - Mover al robot número 1 hacia detrás con una velocidad de 5 unidades, 6 bloques de distancia.

La dificultad e interés del simulador, viene de la optimización y eficiencia máxima que pueda obtener el jugador a la hora de completar niveles con el uso del menor número de bloques posibles y la finalización más rápida del desafío, es decir, encontrando el camino más corto desde el punto de partida hasta el objetivo.

Este lenguaje, aunque sencillo, permite al jugador crear todo tipo de algoritmos y comportamientos para lograr su objetivo. Sin embargo, sin poder obtener información sobre su entorno, los robots serían incapaces de realizar funciones complejas como resolver un laberinto sin la ayuda de un humano que le diga qué caminos tomar.

Para resolver este problema, los robots pueden ser modificados por el jugador con uno o varios sensores. Más específicamente, se pueden usar sensores de detección de infrarrojos, sensores de detección de color, sensores de contacto, sensores de ultrasonido y sensores de rotación o giroscopios. Éstos sensores pueden instalarse en los laterales, parte frontal y parte trasera del robot y cada uno permite obtener información relevante sobre el entorno, como por ejemplo, si está en contacto con una pared o si está siguiendo el camino de color del suelo correctamente, así como calcular la distancia a la pared más

cercana con el sensor de ultrasonido o saber si se encuentra inclinado con respecto al horizonte con un giroscopio.

Este proyecto tiene como referencia otros programas similares del mercado, de entre estos, se incluyen Open Roberta Lab [4], Colobot [5], Erebus [6] y RoboMind [7] entre otros varios.

Open Roberta Lab es una plataforma de código abierto que permite a los usuarios crear programas de control para placas electrónicas para mini-robots reales. Erebus es otro similar pero puramente virtual que contiene robots con sensores por igual, Colobot tiene soporte para modificaciones y modos de juego alternativos y RoboMind es parecido pero está principalmente diseñado en un motor gráfico 2D.

Este proyecto recibe un poco de inspiración de cada uno de ellos. Por ejemplo, implementa sensores como en Erebus y contiene niveles con una organización similar a los de RoboMind. Sin embargo, tiene elementos originales que lo diferencian claramente del resto, como sensores exclusivos no existentes en Erebus o elementos interactivos no vistos en ninguno de los anteriores comentados, un sistema de comunicaciones entre robots o el control simultáneo de múltiples de ellos. La Tabla 1.1, muestra una comparativa entre todos los programas comentados:

Tabla 1.1: Comparativa entre aplicaciones similares del mercado

Programa	Sensores	Múltiples robots	Elementos interactivos	Editor de niveles	Sistema de comunicación
RoblocklIy	Sí	Sí	Sí	Sí	Sí
OpenRobertaLab	Sí	Sí	No	No	Sí
Colobot	Sí	Sí	Sí	Sí	Sí
Erebus	Sí	Sí	No	No	Sí
Robomind	No	No	No	No	No

Para cumplir con el objetivo general de desarrollo de la aplicación, se propusieron los siguientes objetivos específicos a completar:

- Un **sistema de control múltiple**, para poder controlar varios robots al mismo tiempo.
- Un **sistema de comunicación**, que permita a los robots comunicarse entre sí.
- Un **editor de niveles**, para poder desarrollar desafíos jugables por el mismo usuario.

Sin embargo, al momento de finalización del proyecto, también se desarrolló:

- Un **sistema de elementos interactivos**, para la interacción con el mundo de los robots.
- Un **sistema de calificación**, para valorar el resultado final del programa del usuario.
- Un **sistema de importación y exportación de niveles**, para poder compartir un nivel con otros usuarios de manera sencilla y rápida.

Los puntos de valoración para cada objetivo son los siguientes:

Para el sistema de control múltiple, se espera que el jugador pueda mover independientemente dos o más robots con el uso de los bloques de programación. A su vez, los sensores de dichos robots deben ser accesibles al usuario correctamente y deben medir los valores relativos al robot al que están instalados, por tanto, cada robot debe poder instalar sensores independientemente al resto, de manera que uno pueda contener, por ejemplo, un sensor de contacto mientras que otro contenga uno de ultrasonido o de color.

El sistema de comunicación debe dar al usuario la capacidad de manejar automáticamente el envío y recepción de información de los robots, de manera que se pueda desarrollar un programa, con bloques de código asignados a cada robot y que permita la comunicación entre ellos para avisar de acciones o eventos. En términos generales, los robots deben poder comunicarse entre sí, siendo capaces de poder elegir el o los robots a los que va dirigida esa información.

El editor de niveles debe permitir al usuario crear un nivel jugable con, al menos, todos los elementos del entorno perceptible por los sensores del robot. También, permite establecer los elementos mínimos necesarios para poder hacer que un desafío sea completo, es decir, puntos de aparición de robots y metas, así como suelos y paredes para que los robots puedan moverse y evitar caerse al vacío.

Los elementos desarrollados aparte de los objetivos generales no tenían puntos de valoración definidos como tal, sin embargo, yo mismo propuse los mismos a la hora de desarrollarlos:

- Para el sistema de elementos interactivos, implementar a su vez al menos un elemento interactivo pasivo, como puede ser una puerta o ascensor, y un elemento activo, como por ejemplo, un botón o una palanca con lo que activar los interactivos pasivos.

- Implementar el sistema de calificación de manera que los usuarios puedan valorar su desempeño en relación a la calificación preestablecida del nivel.

- Con el sistema de importación y exportación de niveles, se propuso que la forma de compartir los niveles fuese lo más sencilla e intuitiva posible. Debido a limitaciones que se comentarán más adelante, la implementación más adecuada no pudo realizarse, sin embargo, acabó siendo sencillo e intuitivo de todas maneras.

Por último, como objetivo general, se estableció que la implementación de todos y cada uno de los objetivos comentados estuviesen integrados de la manera más general, intuitiva y abierta posible, para que el siguiente desarrollador que trabaje en el proyecto pueda avanzar e implementar nuevas funcionalidades de forma rápida y sencilla.

Capítulo 2 Antecedentes

El proyecto fué recibido con muchos elementos ya desarrollados, principalmente, los menús principales ya estaban implementados, los robots estaban incluídos en el proyecto, se podían jugar cuatro niveles por defecto, los sensores y sus funcionalidades estaban implementadas y los bloques de código de control de robots y de lectura de sensores también estaban incorporados.

2.1 Escenas

Los proyectos de Unity se componen de escenas. Una escena es un contenedor que agrupa todos los elementos y configuraciones necesarias para una parte específica de un juego o aplicación. Principalmente pueden encontrarse dos tipos de escenas en el proyecto: escenas de menús y escenas de desafíos.

En cuanto a las escenas de menús, la parte gráfica de los mismos, es decir, los botones, textos, imágenes, etcétera, estaban bien diseñados y eran intuitivos. No hay demasiado que comentar al respecto, sin embargo, sí que cabe destacar que dos escenas que tenían problemas importantes eran la escena de selección de reto y la escena de selección de sensores.

En cuanto a la escena de selección de reto, que puede verse en la Figura 2.1, podía apreciarse como los cuatro botones de los desafíos ya implementados ocupaban toda la pantalla, esto es, no dejaban espacio para nuevos niveles. Para un proyecto como este, que se pretende llevar a estudiantes para practicar y educar, es inconcebible que solo incluya cuatro niveles y podría haberse deducido que muchos otros se iban a implementar con el tiempo y ayuda de futuros desarrolladores, por lo que es un problema que se decidió remediar y se comentará más adelante.

Figura 2.1: Escena de selección de reto

La otra escena problemática a primera vista era la escena de selección de sensores, visible en la Figura 2.2. Además de poder apreciarse cómo el robot no se encuentra centrado en la plataforma de giro, el sistema de conexión de sensores tenía problemas graves. En primer lugar, se aprecia como los sensores pueden atravesar el robot a la hora de instalarlos y en segundo lugar, la conexión de los mismos al robot no era consistente, en muchas ocasiones el sensor se instalaba en un lugar en el que el usuario no pretendía. Por ello, se decidió también remediar dicho problema en la versión del proyecto más reciente.

Figura 2.2: Escena de selección de sensores

Al contrario que en las escenas de los menús, que los problemas son mínimos y algunos subjetivos, la representación del espacio de trabajo a la hora de programar un nivel sí tenía problemas evidentes. No solo se sentía la falta de espacio en la zona de programación, sino que los bloques de código no se podían leer debido a la falta de espacio también en la zona de bloques, además, considero que los botones de ejecución, pausa y detención que manejan el código eran poco intuitivos. No solo eso, sino que también podían usarse para “hacer trampas”, deteniendo, pausando y reiniciando el código como se quisiese sin reiniciar la posición y movimiento del robot. Por último, cabe señalar que el sistema de cámaras era mínimo, con una única cámara para ver el nivel desde arriba, y que los bloques y botones de todo el proyecto en general estaban escritos en español e inglés, no siguiendo ningún convenio.

2.2 Funcionamiento

Mientras que el funcionamiento principal de los bloques y los menús ya estaba implementado, esto no significa que funcionasen correctamente o que otros elementos no fuesen mejorables.

En primer lugar, el sensor de giroscopio funcionaba de una manera peculiar. Al contrario que lo que uno pudiera pensar, dicho sensor no devolvía el valor de orientación o inclinación del vehículo, sino que devolvía la distancia a una esfera en particular, dentro de en una escena concreta, dicha implementación hacía que el giroscopio perdía su utilidad si no se usaba en un desafío en específico.

Por otra parte, los robots no tenían físicas ni colisiones, lo que hacía perfectamente posible que atravesaran paredes para llegar a la meta sin seguir el camino establecido.

2.3 Scripts

A primera vista podría parecer que el proyecto, aunque con algunos pequeños detalles a mejorar, era sólido y robusto, sin embargo, esto no podría estar nada más lejos de la realidad. Lo cierto es que los scripts de comportamiento de los elementos de las escenas, tanto menús como niveles, pasando por los robots e incluso los sensores, estaban programados de una manera muy poco eficiente, muy poco legible y no seguían ningún principio de la buena programación, es decir, muchas clases definidas realizaban muchas tareas a la vez, otras no realizaban tareas importantes o eran totalmente innecesarias, otras contenían código que afectaba al rendimiento de manera grave y ninguna contenía comentarios o explicaciones de lo que hacían más que un nombre genérico de archivo.

Combinando esto con el hecho de que muchas de las implementaciones de funcionalidades en los scripts estaban programadas de manera poco intuitiva o útil, no es difícil ver que iba a costar muchísimo entender muchas partes del código de la aplicación durante el desarrollo. No solo eso, sino que el proyecto contenía muchos archivos de scripts, los cuales no eran utilizados, desordenados por las carpetas de organización de la aplicación, provocando que costase aún más encontrar los scripts utilizados realmente y, por consiguiente, los importantes.

Al principio se decidió aprovechar todo lo ya desarrollado por los alumnos de la universidad que trabajaron en el proyecto, sin embargo, cuando se descubrió que había tomado más de la mitad del tiempo de programación, desarrollo y diseño en intentar entender qué hacía cada script en el proyecto, consideramos que sería imposible terminar todos los objetivos dentro del tiempo otorgado para finalizar el TFG en este curso. Por ello, se decidió refactorizar parte del proyecto para arreglar los graves problemas que habían, pero finalmente se terminó rehaciendo el proyecto desde cero, sólo reutilizando los modelos 3D ya incluidos en el proyecto, estos siendo los robots y los sensores, y los logos del proyecto. En cuanto a menús, scripts, comportamiento, funcionamiento y todo lo demás, lo único que se mantuvo fue la idea, el tema y el diseño general.

Capítulo 3 Refactorización

Para poder trabajar de manera eficiente, óptima y rápida, y poder terminar el trabajo de fin de grado a tiempo, con un nivel de calidad lo suficientemente alto y todos los objetivos cumplidos, se decidió refactorizar el proyecto al completo, efectivamente empezando desde cero.

Para llegar al punto de progreso al que estaba el proyecto anterior, se siguió un orden de desarrollo que será explicado en este capítulo, así como las acciones realizadas para arreglar los problemas encontrados y las mejoras implementadas.

3.1 Organización

Primero se decidió reorganizar las carpetas del proyecto, de manera que se pudiese encontrar cada archivo y elemento que se buscase sin problema. Si comparamos la raíz del proyecto anterior y el nuevo, que podemos ver en la Figura 3.1, veremos cómo los archivos vacíos, sin uso o inútiles han sido eliminados, las carpetas han sido renombradas con nombres más significativos y se han dividido los assets del proyecto en un mayor número de ellas. Además, las carpetas de assets importadas se han colocado todas en un directorio organizado.

Figura 3.1: Comparativa de organización de proyectos.
Arriba: versión antigua. Abajo: versión actual

Para la creación de escenas organizadas, se decidió llevar un convenio de directorios como en la Figura 3.2:

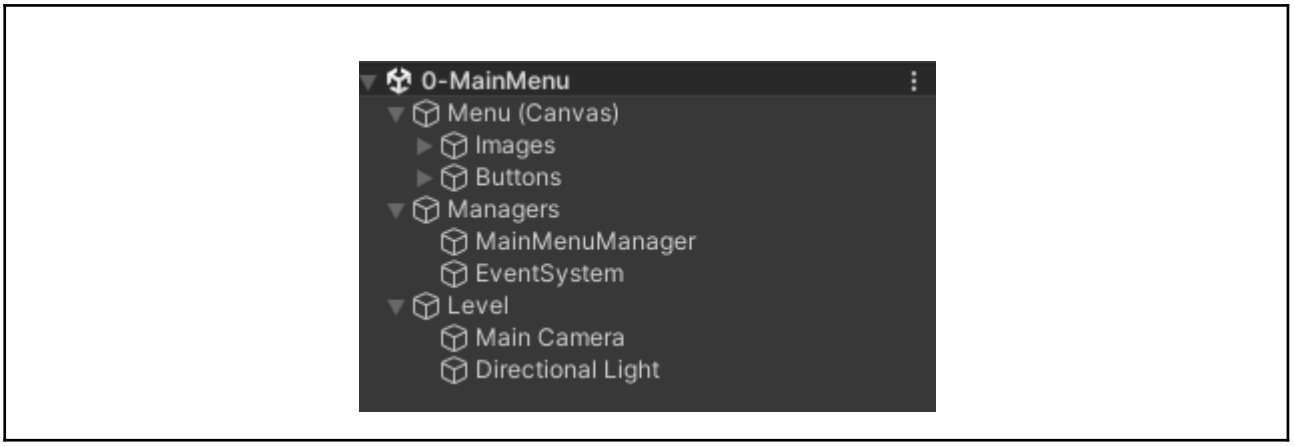


Figura 3.2: Organización de escena

De esta forma, cualquier desarrollador que vea la escena será capaz de discernir el funcionamiento general de la misma, así como entender qué es cada objeto o asset. Además, para el manejo de menús, decidimos implementar un objeto llamado “Menu Manager”, que es es instanciado en cada escena con un menú para aportarle la funcionalidad deseada. En el proyecto anterior, esta clase manejaba absolutamente todo lo referente a la escena.

La división general de escenas es la siguiente: Un directorio de menús contendrá todos los canvas o lienzos que representan los menús de la aplicación, luego, en una carpeta “Level”, se almacenará todo lo relacionado con la escena, es decir, luces, cámaras, robots, modelos 3D, etcétera, y por último en un directorio “Managers” se almacenarán todas las instancias de objetos con los scripts necesarios para implementar las funcionalidades y comportamientos para la escena, tales como el control de menús de dicha escena o la selección de sensores en los robots.

3.2 Clases y Scripts

Después de un pequeño análisis, se vió cómo todos los niveles en el proyecto anterior contenían un manager personalizado, no solo eso, todos los robots contenían scripts específicos para cada uno de ellos también, esto hacía que la cantidad de scripts necesarios para que la aplicación funcionase correctamente fuese ingente debido a que se necesitaba un script por nivel.

En el caso en el que quisiéramos añadir más niveles, esto significaba que debíamos crear un script desde cero para el mismo y como queremos que el usuario sea capaz de usar un editor de niveles para crear su propio nivel, no podemos mantener este sistema. Por ello, y buscando completar el objetivo general comentado anteriormente, propusimos diseñar los scripts y las clases de la manera más genérica posible, para lograr reducir el gran número de ellos necesario para el funcionamiento del programa y hacer más fácil la expansión e inclusión de nuevas funcionalidades en el proyecto, incluyendo el uso de polimorfismo y herencia, entre otros.

1. Para los menús, debido a que cada uno es independiente y contiene botones y elementos distintos entre sí, se tuvo que mantener una clase por escena, exclusivamente dedicada al mismo.

2. Para la escena de selección de nivel, visible en la Figura 3.3, se modificó la posición de los botones de los niveles ya creados para dejar espacio para los niveles nuevos a implementar y los botones de creación y carga de niveles, ya que vamos a implementar un editor de niveles también.

   Para lograr tener espacio ilimitado para añadir niveles, elegimos usar un ScrollView en el que almacenar los botones de carga de los desafíos, esto deja al usuario usar la rueda del ratón para navegar hacia arriba y abajo y elegir el desafío que prefiera. Además, se programó la clase de manera que toda la información para cargar el nivel siguiente estuviese contenida en los propios botones, es decir, que no haría falta modificar la clase para añadir funcionalidad a un botón nuevo para cargar el desafío correcto, como así era en el proyecto anterior,, sino que con la creación de un botón y la llamada a un evento por el mismo fuera suficiente.

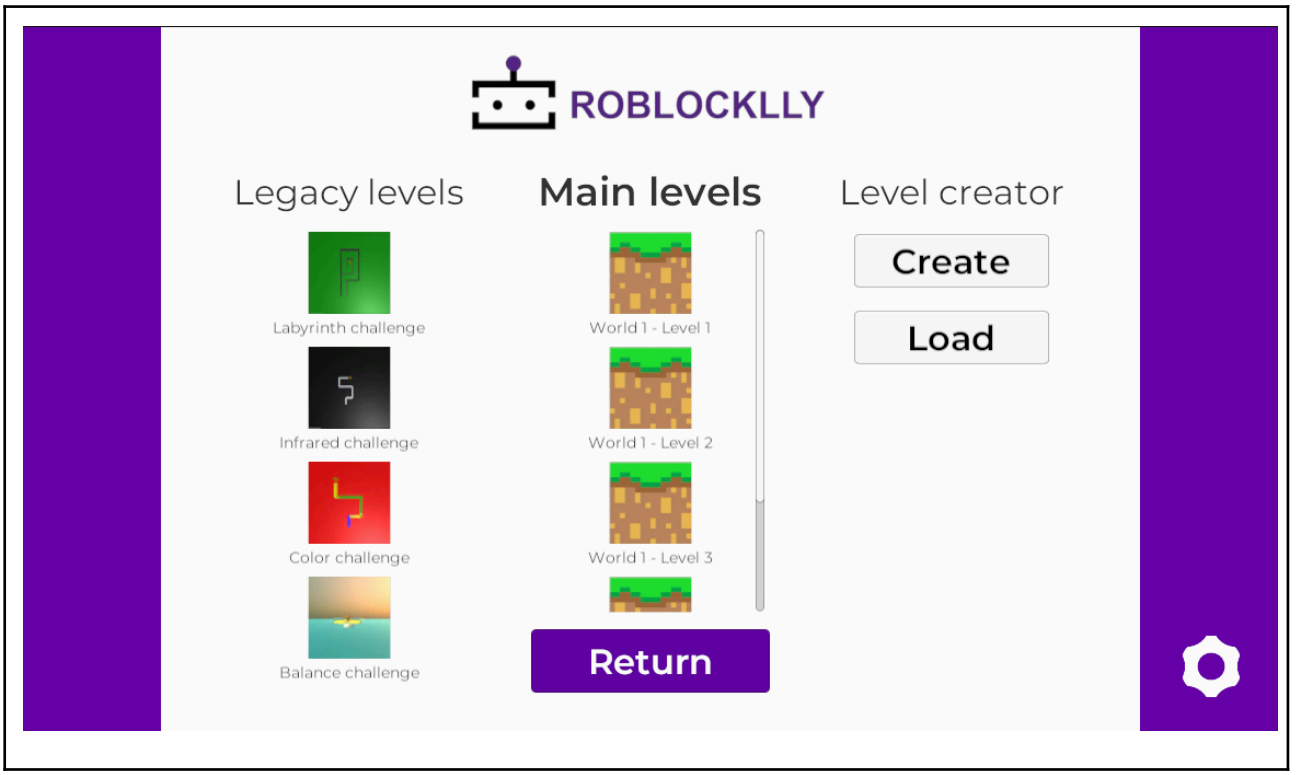


Figura 3.3: Menú de niveles nuevo

3. Debido a que uno de los objetivos es implementar el control de dos o más robots simultáneamente, la escena de selección de robot no era lo suficientemente conveniente, ya que en dicha escena sólo es posible elegir un único robot, lo que significaría que en un nivel con dos o más robots, todos los robots compartirían el mismo modelo.

   Para mejorar este comportamiento y dar al usuario más libertad a la hora de elegir el modelo de cada uno de sus robots, así como poder asignar a cada robot por separado sensores personalizados, se decidió combinar las escenas de sensores y selección de robot en una única, combinando ambas funcionalidades e implementando las nuevas al mismo tiempo.

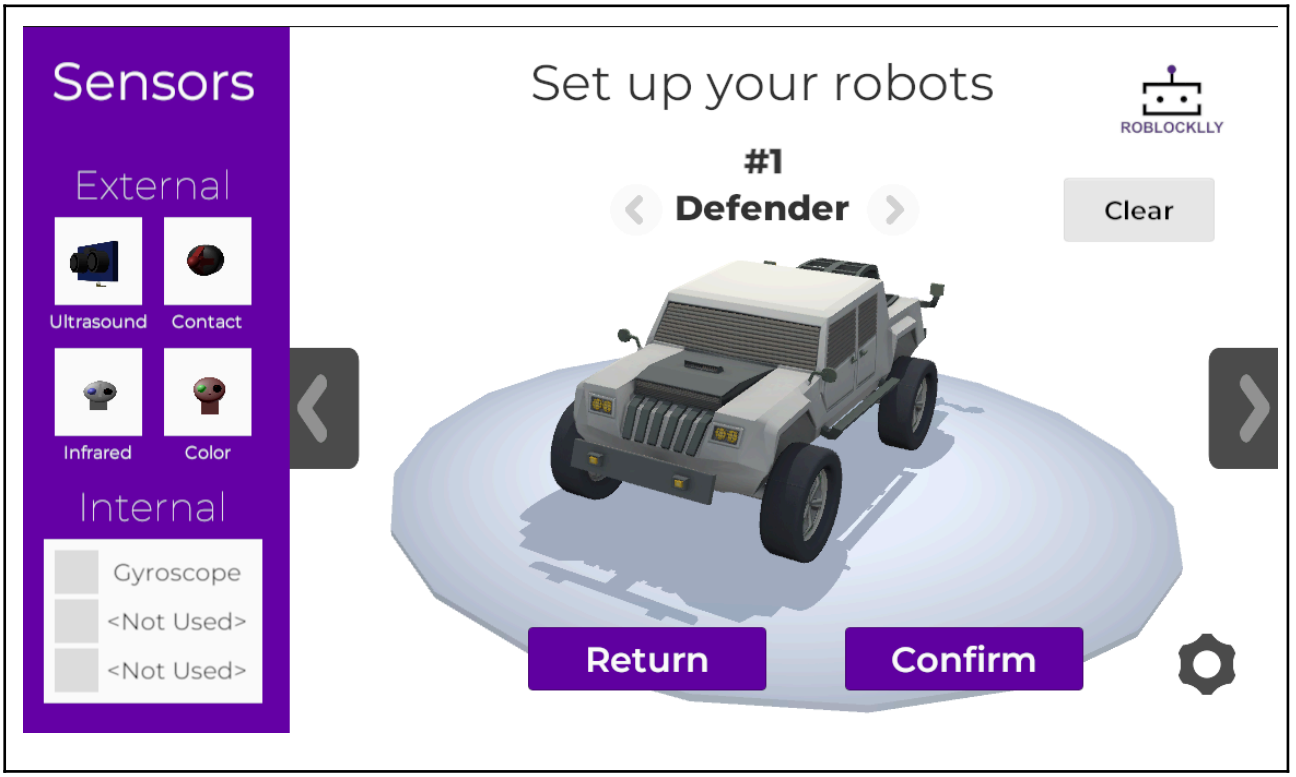


Figura 3.4: Escena combinada de robots y sensores

Como podemos apreciar en la escena de la Figura 3.4, ahora vemos dos botones grises en los laterales, dos botones gris claro encima de la plataforma del robot y un botón con el texto “Clear” además del panel de sensores. En la nueva escena, los botones de los extremos sirven para cambiar entre robots seleccionados mientras que los botones de encima de la plataforma sirven para cambiar el modelo del robot elegido, el botón “Clear” elimina los sensores actualmente instalados en el robot para añadirlos de nuevo y el panel de sensores se encuentra en la parte lateral izquierda para clicar y arrastrar cada uno al robot.

Puede verse cómo se añadieron dos botones extra para sensores internos sin utilizar, de manera que el siguiente desarrollador que trabaje en el proyecto pueda implementar nueva funcionalidad sin tener que preocuparse por la instalación del sensor en el robot.

Para la instalación y selección de robots y sensores, se decidió usar dos managers y un sistema de “snap” o conexión. Debido a la distinta forma y tamaño de cada modelo de robot, podemos esperar que los sensores se instalen en lugares distintos del mismo, por tanto, en el proyecto anterior se creó un script de instalación por cada modelo, sin embargo, esto es tremendamente ineficiente, con lo que decidimos cambiar la implementación.

Para instalar los sensores en el robot, primero se creó una clase “Robot Manager”, dicha clase contiene los sensores instalados en el robot y es capaz de comunicarse con los mismos para obtener la información relevante de su entorno. Para poder usar la misma clase para cada modelo, se añadieron discos de conexión a los modelos y vincularlos a la clase, es decir, zonas de conexión de sensores o “snaps” para conectar automáticamente a ese lugar del modelo cada sensor independientemente de la forma del robot.

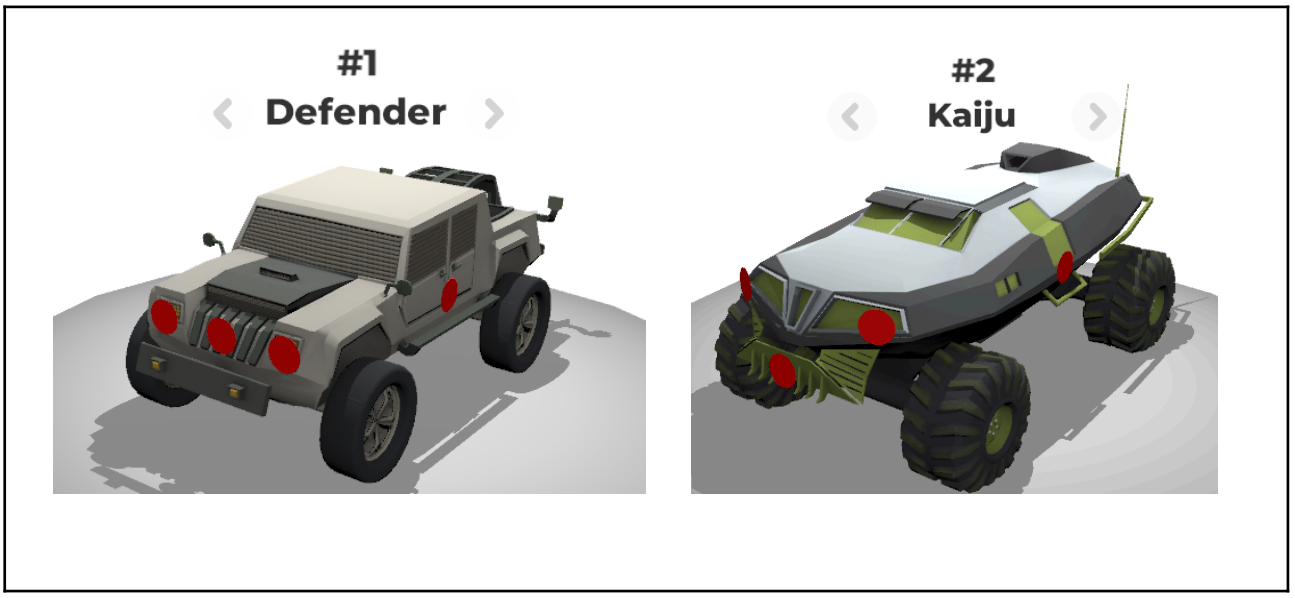


Figura 3.5: Zonas de conexión visibles

Las zonas de conexión, que pueden apreciarse en forma de círculos rojos en la Figura 3.5, son visibles cuando el jugador selecciona un sensor a instalar, de manera que pueda ver dónde colocarlo en el modelo. Para la instalación de los mismos, anteriormente se usaba un sistema de colisiones en donde se calculaba si un sensor colisionaba con una “hitbox” invisible en el modelo. Sin embargo, como esto no era consistente y en muchas ocasiones el sensor colisionaba con varias zonas al mismo tiempo, se cambió el sistema de instalación y en su lugar se utilizó un “Raycast”, ésto es, lanzar un rayo invisible hacia donde apunta el jugador con el sensor y si impacta con una de las zonas, instalarlo ahí.

Por otro lado, se tuvo que rehacer la lógica de los sensores desde cero, ya que la clase de manejo de sensores del robot tenía funciones definidas específicamente para cada sensor, la cual tendría que ser modificada cada vez que se añadiese un nuevo sensor al proyecto. Para lograr que la instalación, uso y control de los mismos fuese de la forma más sencilla posible, se convirtió la clase que contiene el funcionamiento de cada sensor en una clase hija, heredada de una clase padre que define un sensor genérico. De esta forma, en la posible inserción de un nuevo sensor en el proyecto, lo único que habría que hacer es heredar dicha clase para implementar el funcionamiento y nada más, ya que la clase robot tiene una interfaz genérica a la que comunicarse y obtener la información relevante que el sensor emita o genere.

4. En Unity hay un problema que nos dificulta implementar lo que queremos actualmente, y eso es el hecho de que los objetos no se mantienen entre escenas, es decir, un robot instanciado en la escena de selección de sensores no aparecerá en el nivel si cambiamos a él. Para remediar ésto, en el proyecto anterior se usó una clase “Finder”, que se encargaba de realizar muchísimas acciones y de entre ellas, guardar referencias a objetos que debieran ser mantenidos entre escenas. Ya que nos pareció una buena idea en una mala implementación, se decidió reutilizarlo y crear una clase llamada “VaultManager”, como su nombre indica, dicha clase maneja el comportamiento de un “Vault” o baúl, término que se usa para describir un contenedor que almacena todos los objetos que deben ser mantenidos entre escenas, esta clase se encarga de mantener y/o eliminar un objeto de la escena solo con el uso de la función “addObject” y “removeObject”, y contiene una implementación para otras dos funciones útiles, “getObject” y “checkForObject”. Estas dos últimas funciones se encargan de devolver una referencia al objeto especificado del contenedor y de devolver una confirmación si el objeto que se busca se encuentra en el contenedor respectivamente.

   Para que esta clase se mantenga entre escenas y pueda ser utilizada por cualquier otro script, se instancia y se vuelve persistente entre escenas a sí misma, añadiendo a un elemento estático de su clase la referencia a la instancia, para que los scripts puedan acceder a la variable estática “vaultInstance” de la clase “VaultManager” y obtener una referencia a la instancia en su escena.

5. En el proyecto anterior, se hacía uso de una clase de estadísticas, que almacenaba los resultados de finalizar un nivel para su posterior visualización, para implementarla en el nuevo proyecto, decidimos modificarla para hacerla más útil y accesible.

   En primer lugar, se cambió la información almacenada por resultado para incluir el número de bloques usado en la solución y el tiempo que tarda en completarse el nivel, a diferencia de lo que se usaba anteriormente, que era el tiempo completo de programación, además de una nota o “score”, esta nota puede valer “S+”, “S”, “A,” “B”, “C”, “D” o “F” respectivamente, siguiendo un sistema de calificación que compara el resultado obtenido con el resultado diseñado por el desarrollador del nivel. Para ello, la clase “StatsManager”, que fue la que se implementó, almacena una lista con los resultados de los niveles jugados y los resultados de desarrollador o “dev stats” para cada uno de ellos. Para no tener que modificar dicha clase con la

implementación de niveles nuevos como era el caso anteriormente, se añadieron funciones públicas para acceder a dichos valores y poder modificarlos por parte de otros scripts. Cuando un nivel se completa, se puede observar en un pequeño panel emergente el resultado del jugador, podemos ver un ejemplo en la Figura 3.6 a continuación.

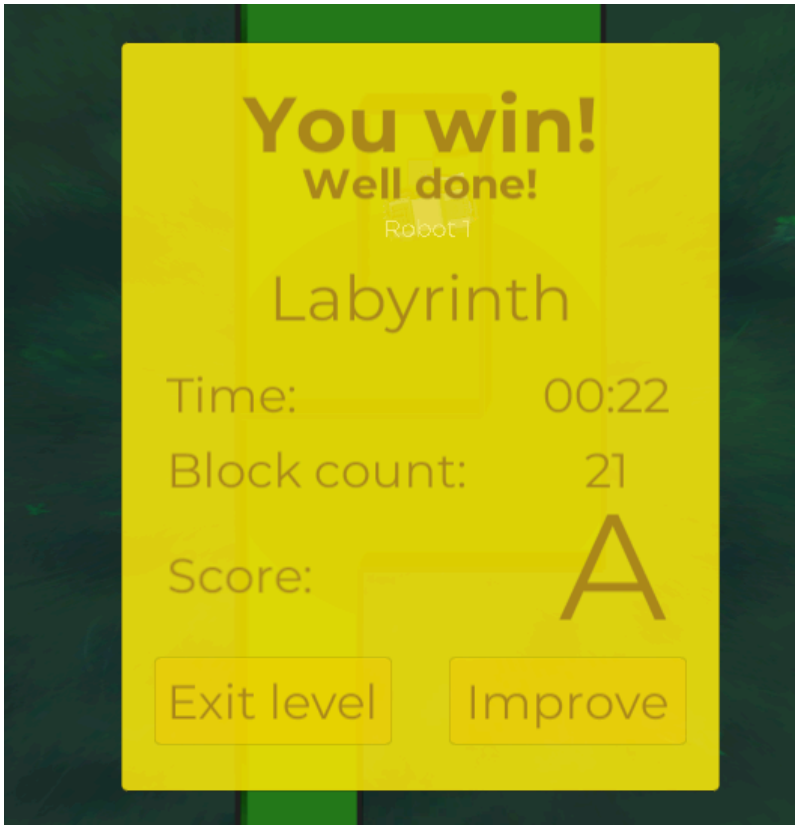


Figura 3.6: Resultados de un desafío

Para la visualización de los resultados fuera del nivel, se diseñó una clase “SettingsManager”, esta se encarga de manejar el funcionamiento de los botones del menú de opciones, junto con los del menú de estadísticas. En la siguiente Figura 3.7, podemos observar que los botones de estadísticas por nivel se generan dinámicamente una vez completamos un desafío y contienen los datos del resultado obtenido.

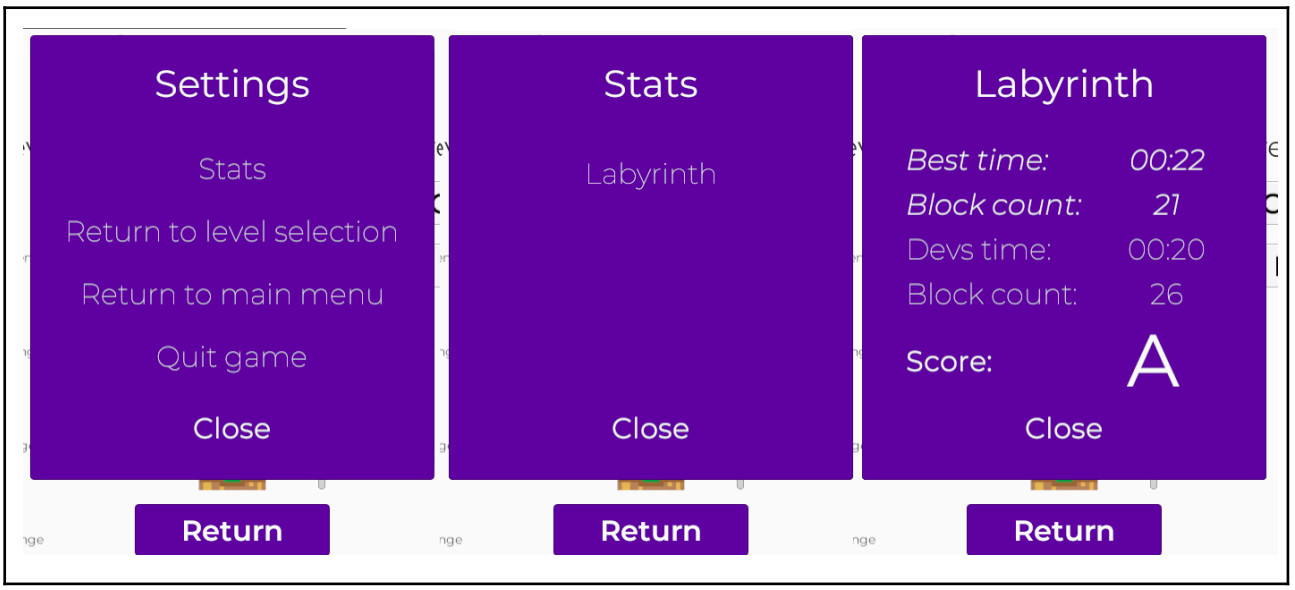


Figura 3.7: Menú de opciones

6. Un problema grave que se tuvo que resolver fue la falta de colisión y físicas de los robots y el mundo, ninguno de los modelos contenían colisiones y eran posibles ser atravesados, convirtiendo la dificultad del nivel en nula. Para arreglarlo, tuvimos que modificar todos los modelos para añadir colisiones correctamente, añadir “Rigidbodies” de manera que pudiesen recibir y ejercer fuerzas como la fuerza de la gravedad, y desarrollar un nuevo script de movimientos llamado “RobotMotionManager”. Este script se encarga de almacenar toda la lógica con respecto al movimiento del robot, desde la rotación hasta la traslación, manejando también el movimiento correcto de las ruedas para que giren proporcionalmente a la velocidad del robot. Como con todo lo demás, se diseñó el script para poder ser reutilizado en cada robot y no tener varios scripts, uno por modelo.

7. Por último, pero no por ello menos importante, se diseñaron todos los textos, botones y bloques en inglés para seguir un convenio constante, al contrario de como estaba en el proyecto anterior, que contenía elementos en español e inglés simultáneamente.

En el siguiente diagrama de clases en la Figura 3.8, podemos observar un esquema general con las clases principales de la aplicación, ya comentadas:

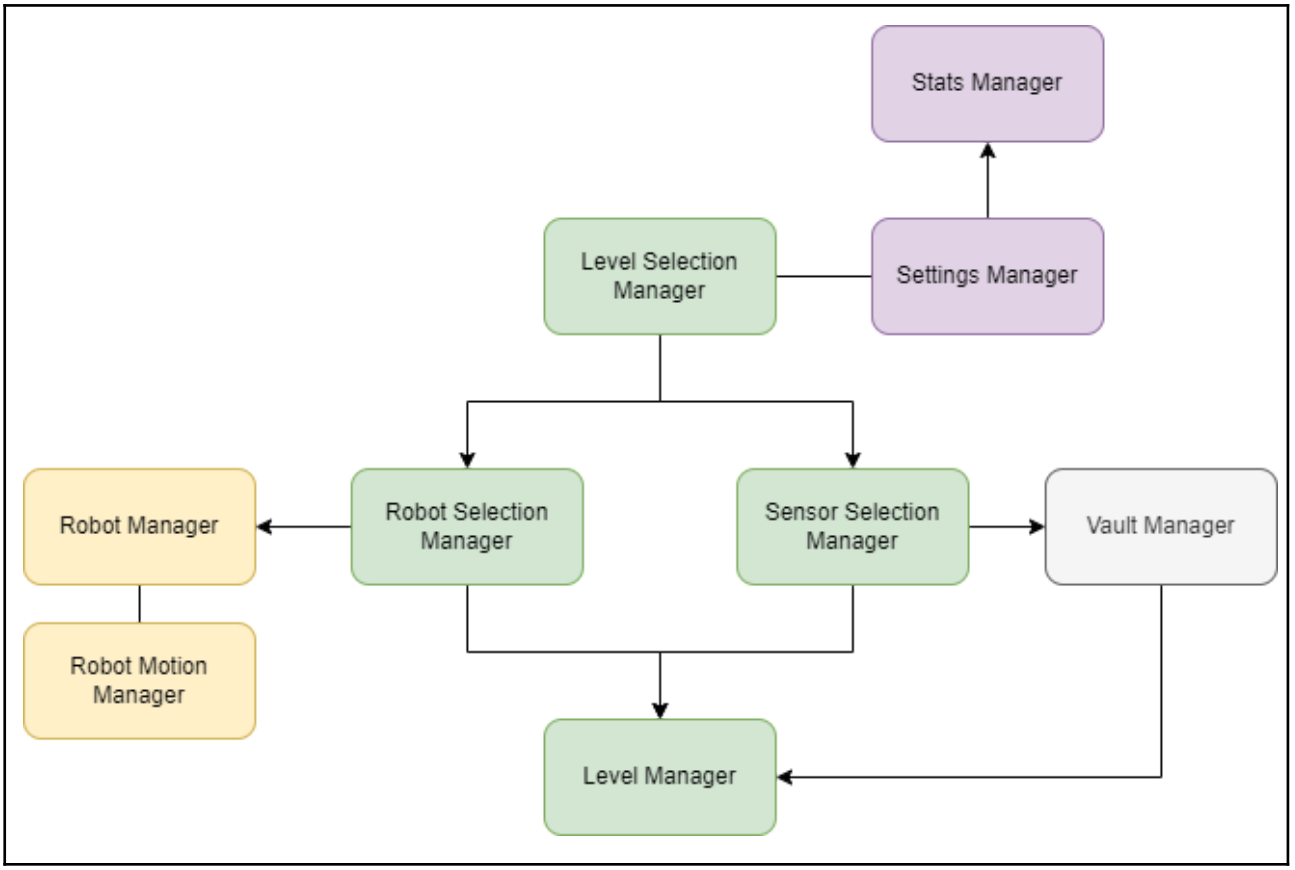


Figura 3.8: Diagrama de clases

3.3 Integración de uBlockly

Una vez reimplementadas, arregladas y mejoradas todas las funciones anteriores del proyecto, se tuvo que reimportar la librería uBlockly, de manera que el sistema de programación visual basada en bloques volviese a funcionar.

Durante la reimplementación, nos dimos cuenta de que la librería ofrecía muchísimas opciones de personalización, permitiéndonos arreglar muchos de los problemas que tenía el anterior proyecto, como la falta de visión de los bloques o el pequeño espacio de trabajo a usar.

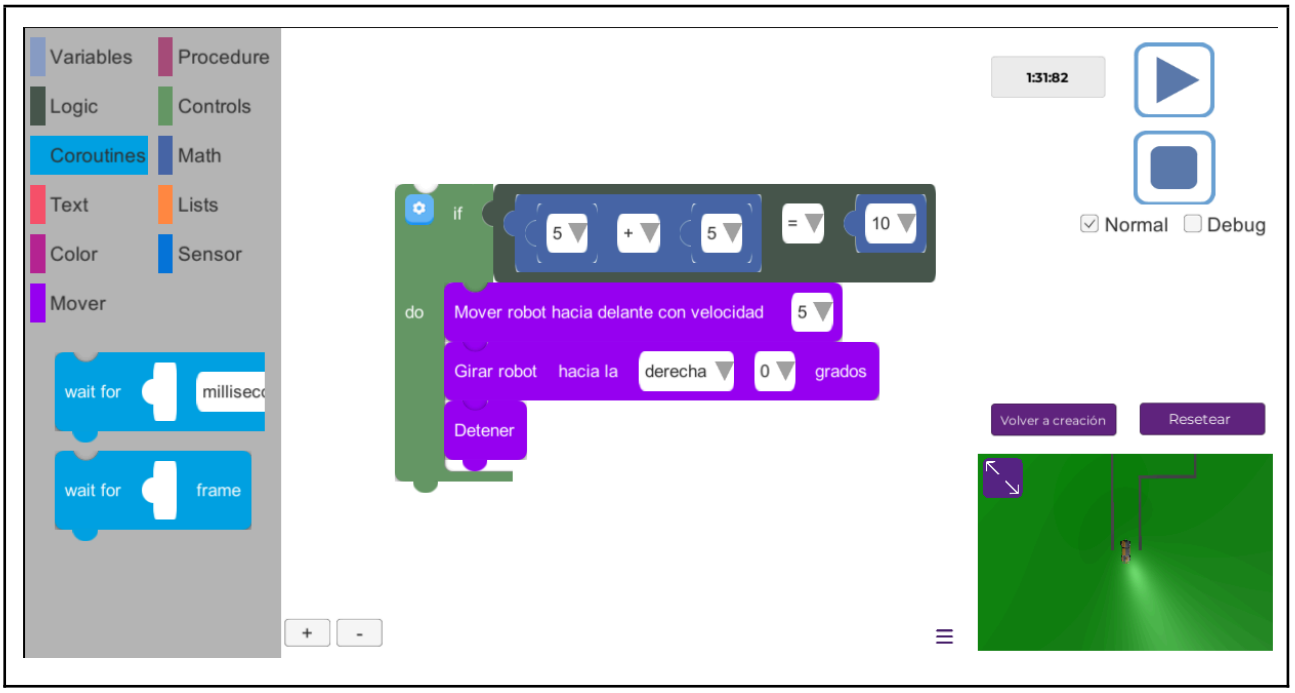


Figura 3.9: Espacio de trabajo uBlockly antiguo

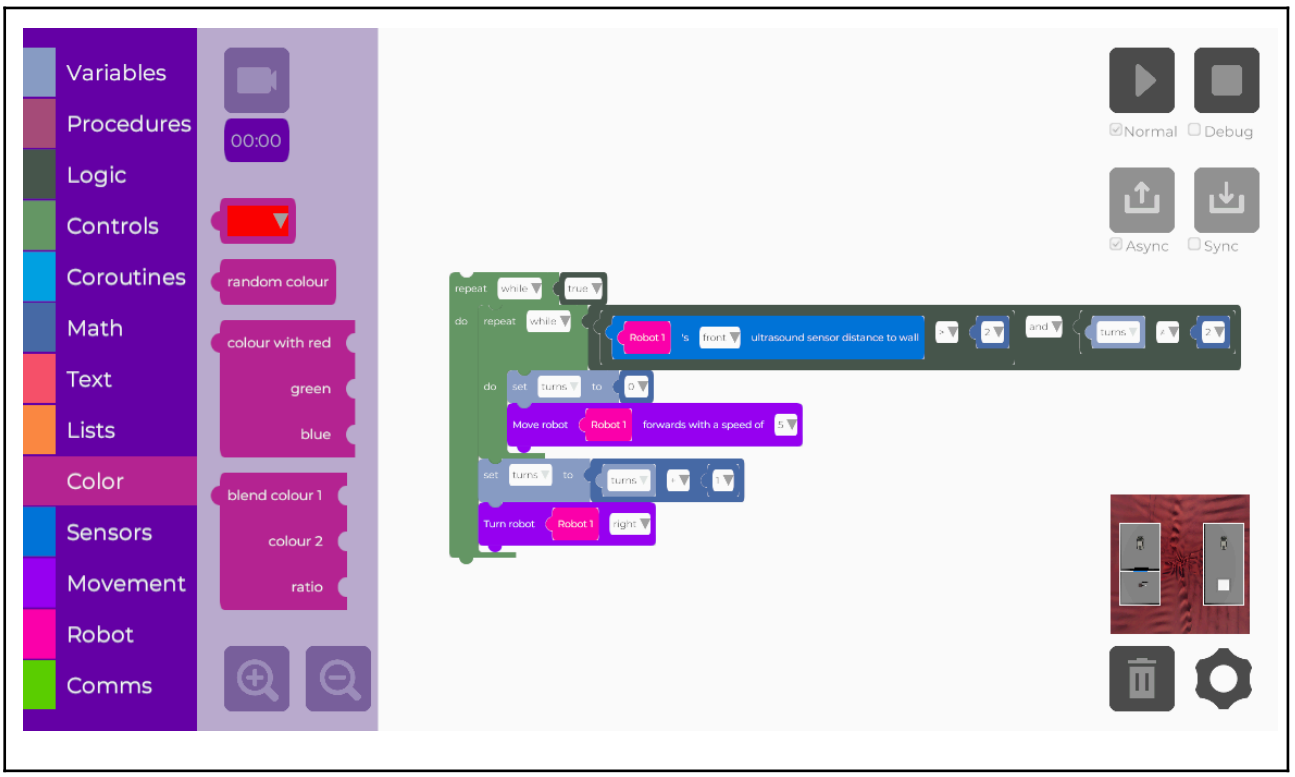


Figura 3.10: Espacio de trabajo uBlockly nuevo

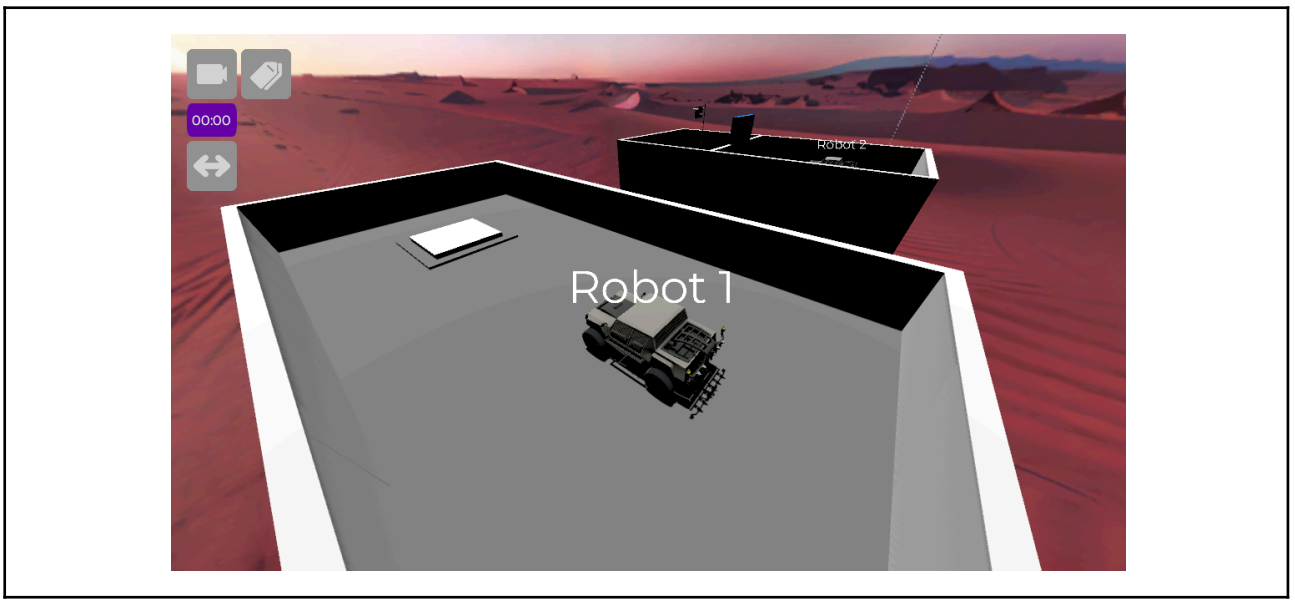


Figura 3.11: Cámara de robot

Como podemos ver en la Figura 3.10, así es como ha quedado la zona de trabajo después de la reimplementación en comparación a la Figura 3.9, que representa el estado antes de la misma. Podemos apreciar como se ha reducido el tamaño de la cámara, como se ha agregado un botón para cambiar entre la zona de trabajo, una cámara de vista de pájaro y una cámara de robot orientable, la cual puede, a su vez, ser cambiada entre robots (Figura 3.11) con el botón de las flechas horizontales, así como un botón para alternar la visibilidad del nombre de cada robot en pantalla.

También se ha cambiado el menú de selección de bloques para que ocupe menos espacio cuando no está abierto y se pueda leer todo el bloque cuando se abra y, por último, se ha cambiado el diseño de los botones para que todo siga una misma temática. Además, se eliminó el botón de resetear y se vinculó directamente al botón de detención del código para que el usuario no pueda hacer trampas, se ocultó el botón de volver a creación en un menú de opciones para ocupar menos espacio y se implementó un sistema de guardado y carga de programas y un botón de borrado para limpiar el espacio de trabajo.

Una vez mejorado el espacio de trabajo, se dedicó tiempo de desarrollo en implementar los bloques de sensores y movimiento que ya habían sido diseñados para que funcionasen correctamente con los nuevos scripts de sensores y de movimiento de robots. Después de comentarlo con los tutores, también se implementaron unos bloques extra para permitir mover el robot hacia delante y hacia detrás un número de “bloques” de distancia. Ya que se va a desarrollar un editor de niveles y dicho editor trabajará con bloques, consideramos buena idea que los robots pudieran moverse siguiendo esta unidad de medida.

Capítulo 4 Múltiples robots

Al fin, con el proyecto reescrito y reestructurado, pudo ponerse a trabajar en los objetivos iniciales del proyecto, siendo el primero, la implementación de control de múltiples robots.

En primer lugar, se modificaron los botones del menú de selección de nivel para que pudiesen almacenar la información de cuántos robots serán utilizados por el nivel, esto es, para que el menú selector de robots sea capaz de enseñar el número de robots correcto a elegir para el desafío pertinente. Una vez hecho, fue cuando se realizó la modificación del menú de selección de robots y sensores y, haciendo uso del “VaultManager”, el programa puede almacenar los robots elegidos por el usuario e instanciarlos en la escena posterior: el nivel.

Una vez llegados al nivel y considerando que cada desafío usaba un script específico, se decidió rehacer esta implementación y crear un script global que maneje cualquier tipo de nivel. Para ello, dicho script, llamado “LevelManager”, recibe los puntos de aparición del nivel, los puntos de finalización del nivel o banderas, el nombre del desafío y los bloques usados y tiempo empleado en completar el nivel por el desarrollador. Con todo esto, el script de nivel es capaz de colocar los robots en los puntos de aparición, comprobar cuándo se completa el nivel, gracias a las banderas, y es capaz de comunicarse con el “StatsManager” en el completado de un nivel para dar los resultados y calcular la calificación.

Con los niveles preparados, los menús adaptados y los managers en funcionamiento, lo último que se tuvo que hacer fue modificar el comportamiento de los bloques de movimiento y sensores que los contribuidores al proyecto anteriores habían implementado para tener en cuenta un mayor número de robots. Por desgracia, dichos bloques no contenían código reutilizable, por lo que tuvimos que rehacerlos desde cero también.

Para implementar los bloques de movimiento y sensores, tuvimos que definirlos de nuevo en la librería de uBlockly y además tuvimos que definir una nueva sección o tipo de bloque también, esto es, los bloques de robot.

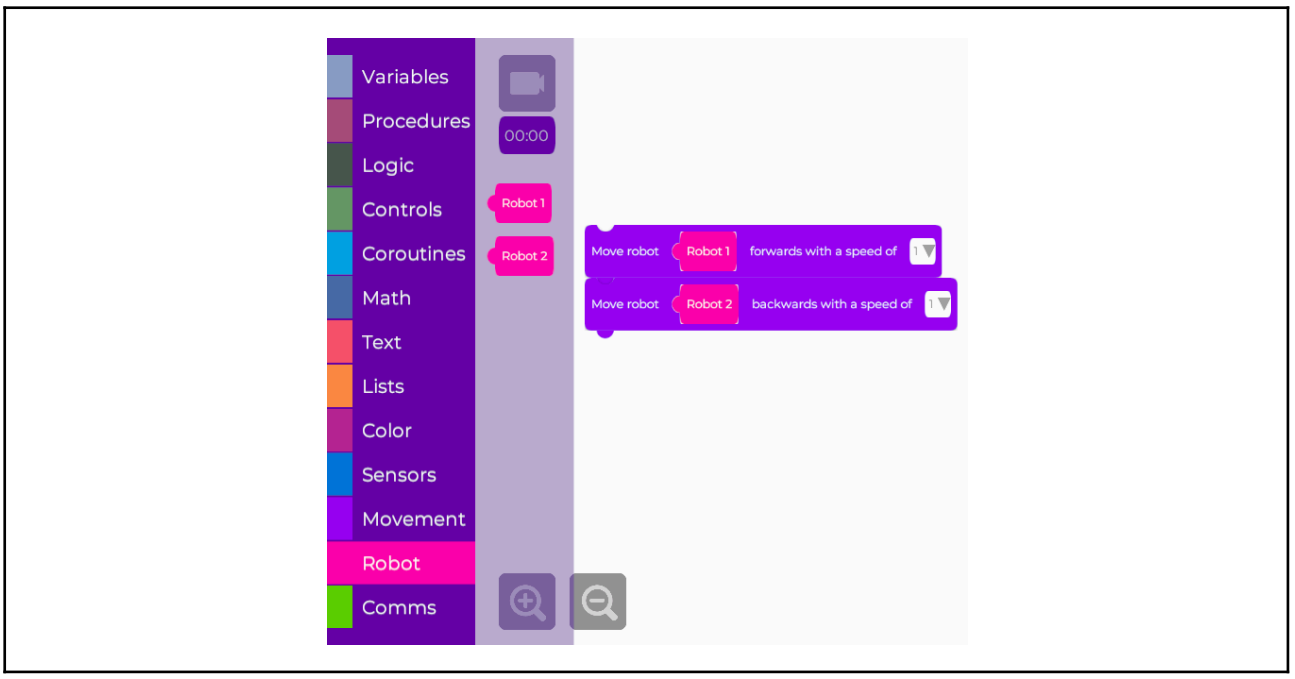


Figura 4.1: Bloques de código de robot generados dinámicamente

Como se puede apreciar en la Figura 4.1, los bloques de robot sirven para especificar a qué robot se refiere cuando se realiza una acción, ya sea la de movimiento del mismo o la de selección de sensor, no solo eso, estos bloques se generan dinámicamente en tiempo de ejecución dependiendo del número de robots que haya en una escena, lo que significa que un desafío con cinco robots, tendrá en su menú de bloques disponibles cinco bloques de robot. Cada robot puede diferenciarse entre sí haciendo uso del botón de etiquetas, que permite ver los nombres de cada robot encima de los mismos. Puede verse un ejemplo en la Figura 3.11.

Por último, para que la lógica de los robots, las colisiones y el escenario funcionasen correctamente, se establecieron en las opciones del proyecto las colisiones correctas entre cada elemento, de manera que los robots no pudiesen atravesar elementos del escenario pero que no pudieran colisionar con sensores o puntos de conexión tampoco, entre otras cosas.

Con esto implementado, ahora los niveles pueden ser resueltos con el control de dos o más robots.

Capítulo 5 Comunicación entre robots

El segundo objetivo a cumplir era lograr la comunicación entre los robots, para ello, sería necesario añadir los bloques pertinentes y establecer un sistema de comunicaciones funcional.

En primer lugar, se consideraron varias formas de comunicación, y al final nos decantamos por el siguiente sistema: cada robot podrá enviar una señal específica a un robot específico, a un grupo de robots o a todos los robots. Los demás robots, serán capaces de leer los avisos que les han llegado a ellos mismos, a un grupo en el que se encuentran o a todos en general. A su vez, serán capaces de leer los avisos o comunicaciones sin límite de tiempo ya que se almacenarán en una lista ordenada por elementos más recientes. Por último, serán capaces de limpiar estas listas o eliminar avisos de la misma para no leerlos varias veces. Para asistir al usuario a la hora de recordar o saber qué robot se encuentra en qué grupo, ya que dichos robots pueden cambiar de grupo de manera dinámica en tiempo de ejecución, implementamos que cada uno de ellos obtuviese una luz con movimiento y color, parecida a las usadas por coches policiales, dependiendo del grupo al que formen parte. Puede verse un ejemplo en la Figura 5.2.

Con esta implementación, los robots son efectivamente capaces de enviar, recibir, leer y eliminar avisos de otros robots para comunicarse entre sí.

Para añadir toda esta funcionalidad, primero se implementó el sistema de comunicaciones con un objeto “CommsManager” que estará en todos los niveles por igual. Esta clase almacena una lista global con todos los avisos enviados por los robots, dichos avisos contienen la información de su tipo (si va a un grupo de color, es un broadcast o va dirigido a un robot en específico), su valor o nombre, su remitente y su receptor o grupo al que va dirigido. Los robots, haciendo uso de los bloques que se pueden apreciar en la Figura 5.1, que son los que fueron implementados para el sistema de comunicaciones en uBlockly, serán capaces de contactar con esta clase y obtener la información necesaria sobre el protocolo de comunicaciones.

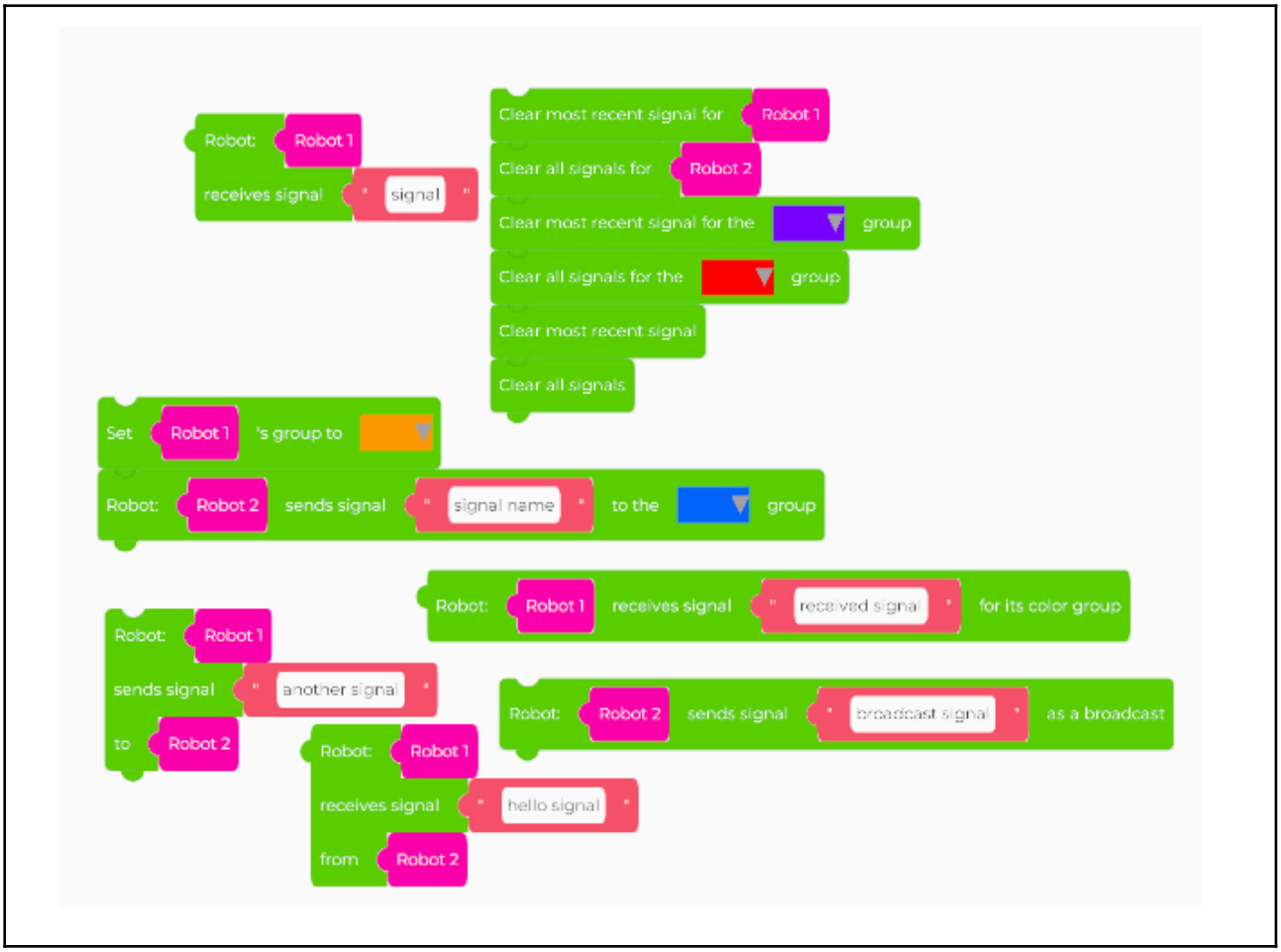


Figura 5.1: Bloques de comunicación

Figura 5.2: Luces de color por grupo

Capítulo 6 Elementos interactivos

Justo después de implementar el sistema de comunicaciones, se consideró empezar a trabajar en el editor de niveles, sin embargo, sin nada con lo que interactuar entre el robot y el entorno más que paredes, se sentía que el simulador carecía de algo. Por lo que se propuso integrar un sistema de elementos interactivos.

Para esto, se creó un script principal para elementos interactivos pasivos y otro para elementos interactivos activos siendo, estos últimos, elementos que son modificados por una señal y elementos que envían señales respectivamente. Como siempre, estos scripts son genéricos y por tanto heredables para especificar funcionamiento específico, pero compartiendo una interfaz pública sencilla para que cualquier elemento interactivo activo pueda controlar uno o más pasivos de manera simple.

Desde el punto de vista del desarrollador, todo lo que se debe hacer para integrar un nuevo elemento activo o pasivo, es heredar la clase padre en un script y agregar a ese, la funcionalidad nueva del elemento. No se tendrá que considerar la comunicación por el mismo en los elementos interactivos ya implementados ni nada por el estilo.

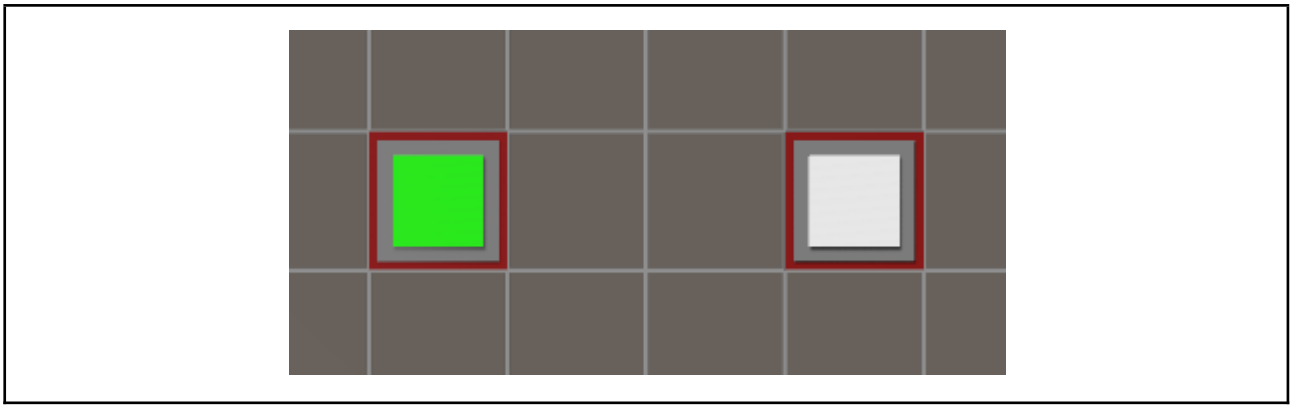


Figura 6.1: Placa de presión vista desde arriba, presionada y neutral sobre bloques rojos

En el proyecto se acabaron implementando tres elementos interactivos: una placa de presión, una puerta elevadora y una plataforma horizontal. La placa de presión, apreciable en la Figura 6.1, contiene cuatro posibles listas de interacciones, una lista de interacciones de activación, otra de desactivación, una de mantener y una de cambio. Estas listas establecen el modo de interacción entre la placa y los elementos pasivos a activar. La lista de elementos a activar, activa los elementos pasivos en ella cuando la placa de presión es pulsada, la lista de elementos a desactivar, desactiva los elementos en ella, la lista de mantener, se encarga de activar los elementos pasivos cuando la placa de presión es pulsada y de desactivarlos cuando se deja de pulsar, y la lista de cambio se encarga de activar o desactivar con cada pulsación consecutiva de la placa de presión por el robot. Con todas estas posibilidades, el desarrollador del nivel puede crear desafíos con mucha libertad y detalle.

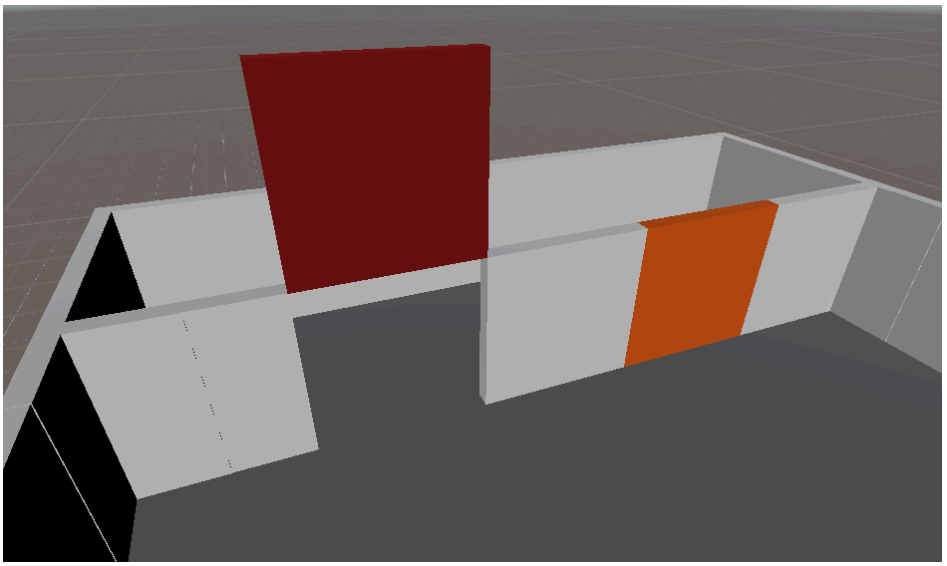


Figura 6.2: Puerta abierta y cerrada

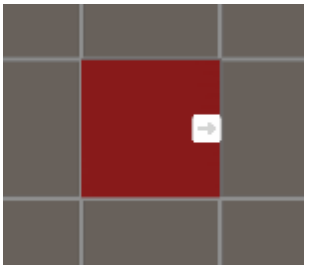


Figura 6.3: Plataforma vista desde arriba, nótese la flecha que indica la orientación de la misma

La puerta elevadora, visible en la Figura 6.2, como su nombre indica, es un elemento interactivo pasivo que se asemeja a una puerta, dicha puerta puede activarse o desactivarse, por tanto abriéndose o cerrándose, con la señal de un elemento interactivo pasivo.

Por otro lado, la plataforma horizontal móvil, visible en la Figura 6.3, que contiene configuraciones para su velocidad, distancia entre paradas y número de paradas, permite que el usuario que la usa especifique la velocidad a la que se mueve cuando recibe una señal de un elemento interactivo activo, el número de lugares en donde parará la plataforma y la distancia en bloques entre cada uno de esos puntos.

Por último, aunque no es considerado un elemento activo o pasivo, se introdujo el comportamiento interactivo de una bandera de finalización o meta, esto es, una bandera que se coloca al final del nivel, al contrario que el modelo de moneda que se usaba en el proyecto anterior. Dicha bandera contiene el comportamiento necesario para avisar al “LevelManager” de que un robot ha colisionado con ella y por tanto finalizar el desafío, pero también he incluido la característica de que la bandera pueda ser una bandera con contador, lo que proporciona la posibilidad de que un robot deba permanecer en la bandera durante un tiempo determinado antes de poder ser considerado una victoria.

Capítulo 7 Editor de niveles

El último objetivo a cumplir para completar el TFG, consistía en la creación de un editor de niveles o desafíos, este editor debe permitir al jugador crear sus propios niveles y compartirlos con otros usuarios, y estos usuarios deben poder importar los niveles del primer jugador y jugarlos.

Como primera idea de editor, se pensó en integrar un sistema de construcción en 3D basado en bloques, al igual que en videojuegos populares como Minecraft, sin embargo, debido a que el proyecto no está pensado para alojar niveles con distintas alturas, ya que, al momento de finalización de este proyecto, no se encuentra una verdadera utilidad a tener rampas o pisos en un simulador de este estilo, se acabó decidiendo realizar un editor basado en bloques con vista desde arriba.

Para implementar el editor, primero consideramos la integración de un sistema de rejilla para colocar los bloques de construcción. Para ello, se realizaron tres acciones: primero, añadir una rejilla visual para el jugador, segundo, añadir una rejilla de nivel, para almacenar los bloques utilizados en una matriz manejable por script y, por último, establecer unos controles intuitivos para la construcción en dicha rejilla.

Para crear la rejilla gráfica, se hizo uso del “Shader Graph” de Unity. Esta es una herramienta muy útil para crear shaders y aplicarlos a un material. En primer lugar, se estableció una malla plana en el fondo de la escena, luego, se aplicó un material “GridMaterial” que contendría la textura de la rejilla necesaria para su representación en el fondo, por último, se creó un shader en el “Shader Graph” de Unity y se aplicó al material de la malla del fondo. Puede verse el resultado en la Figura 7.2.

Siguiendo la Figura 7.1, que contiene una captura de pantalla del “Shader Graph” de Unity con el shader “GridShader” a implementar, podemos ver cómo funciona concretamente. Como siempre, todo lo desarrollado está pensado para ser lo más fácilmente modificable y entendible posible, por lo que el shader usa muchas variables que se pueden editar para cambiar el efecto en el material final. En primer lugar, obtenemos la escala del objeto al que está aplicado el material, luego, obtenemos los componentes “x” y “z” para calcular la escala correcta del shader. Ya que el material será establecido en un plano, no necesitamos preocuparnos por la componente “y”. A continuación, multiplicamos la escala del objeto por una variable “DefaultScale”, que usaremos para establecer la escala a la que queremos que se ajuste nuestra rejilla multiplicada a su vez por una variable “Size” que nos servirá para establecer el tamaño de cada casilla. No se ha comentado antes, pero para la óptima creación del nivel y teniendo en cuenta el tamaño de los modelos de

robots en el proyecto, se decidió usar bloques de 16x16 unidades de distancia de Unity para tener suficiente espacio sin parecer demasiado grandes.

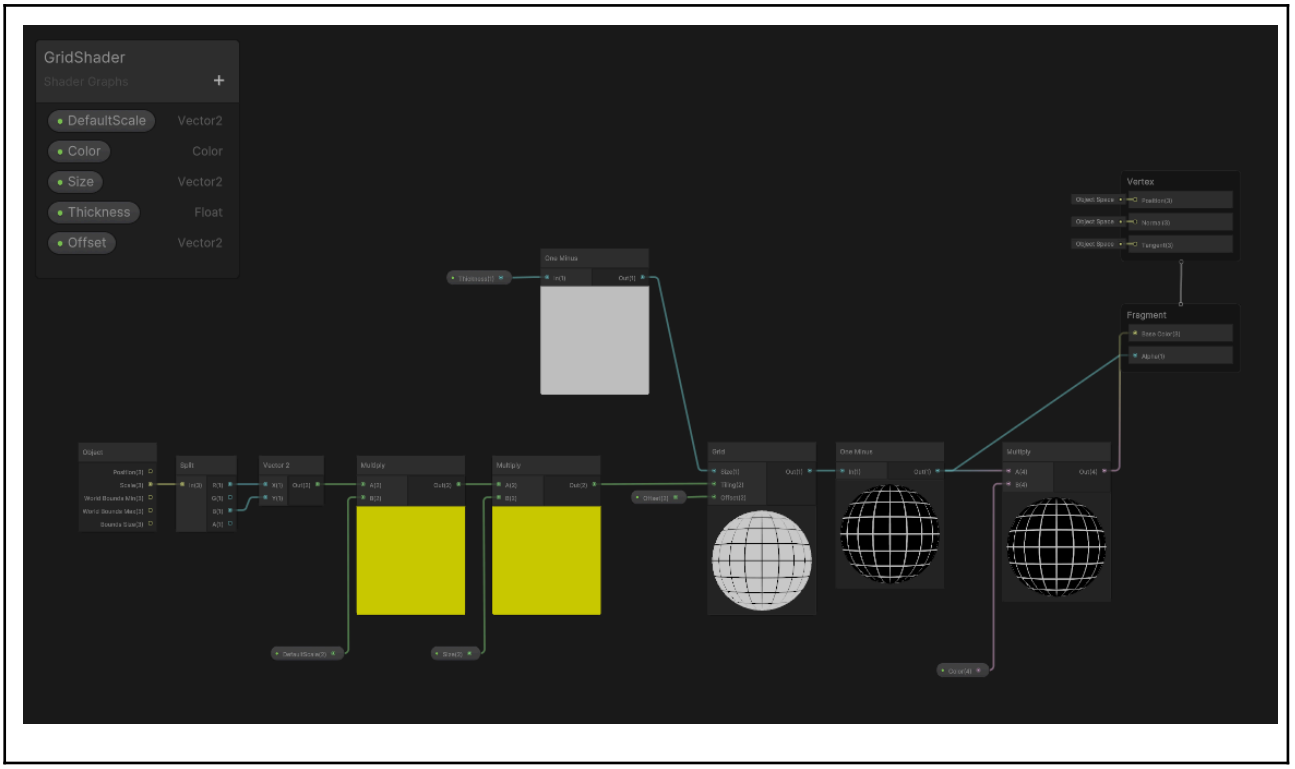


Figura 7.1: Shader graph

Una vez nuestro shader se ajusta a las proporciones y tamaño que queremos, tenemos que contar con el grosor de los bordes de la rejilla, así como el posible posicionamiento en el que queramos colocarla, junto con el color de la rejilla óptimo. Para resolver todo esto, primero pasamos los valores de escala que hemos calculado a un elemento grid, luego, establecemos el valor “Size”, que no es más que el grosor de la rejilla a uno menos una variable que denominaremos “Thickness” o grosor en inglés y una variable “Offset” que usaremos para posicionar la rejilla.

Invirtiendo el resultado, deberíamos obtener una rejilla funcional, sin embargo, queremos también establecer un color, para ello, simplemente multiplicamos el resultado del módulo grid por un color que almacenaremos en la variable “Color” y por último pasamos todo lo calculado al bloque “Fragment” del shader en la entrada “Base Color”, mientras que en “Alpha”, que será la entrada que determinará qué partes de la textura son renderizadas y qué partes no, pasamos el valor de uno menos la rejilla que hemos creado para que solo los bordes sean dibujados en el material. Podemos ver el resultado de dicha rejilla, con un tamaño de 16x16 y color gris en la siguiente Figura 7.2.

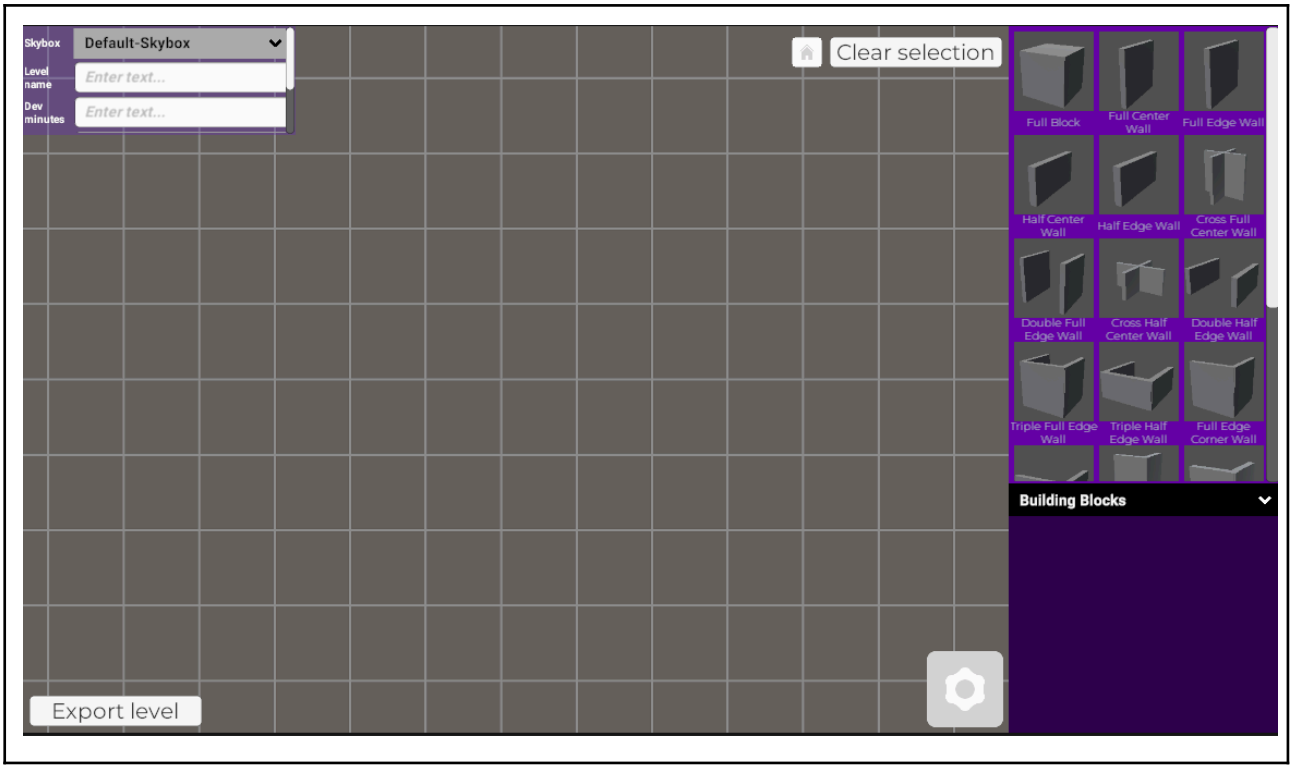


Figura 7.2: Editor de niveles vacío

Con la rejilla visual terminada, debemos, a continuación, implementarla en el espacio real, es decir, darle funcionalidad para que se puedan instanciar en el mundo bloques de construcción en las posiciones correctas. Convenientemente, Unity contiene un componente “Grid” que hace justo eso, ordena en un espacio de la escena elementos con una distancia, tamaño y posición determinadas.

Después de crear un objeto vacío y atribuirle el componente “Grid”, hace falta hacer uso del mismo a través de un script pero, en primer lugar, hace falta tener bloques que colocar, con lo que decidí crear distintos tipos de bloques de construcción.

Para crear diferentes bloques como las paredes se usó el plugin de Unity ProBuilder, que es una herramienta que permite editar y crear modelos 3D directamente en el programa. Ya que se buscaba permitir al usuario usar la mayor creatividad posible, se decidió incluir bastantes bloques en el editor, de manera que casi cualquier nivel o idea pudiese crearse con los bloques otorgados. Añadimos un total de 19 bloques de construcción, 4 elementos interactivos y 5 elementos misceláneos.

En el apartado de bloques de construcción, se añadió un bloque completo de 16x16, paredes altas y bajas que pueden ser colocadas en el borde de un bloque, paredes altas y bajas que pueden ser colocadas en el centro de un bloque, paredes dobles ya sean en cruz o paralelas, paredes esquinadas en el borde o en el centro, paredes triples altas y bajas de borde y paredes triples altas y bajas centrales de borde o de centro, casi todas pueden apreciarse en la Figura 7.2. Con esta gran cantidad de bloques, el usuario puede

crear una amplia variedad de niveles.

En la sección de elementos interactivos, se añadió la placa de presión anteriormente comentada, la puerta de elevación, la plataforma horizontal y la bandera de finalización. Y por último, en la sección de misceláneos, añadimos los puntos de aparición y cuatro posibles caminos de color, esto es, elementos similares a alfombras que pueden colocarse encima de los bloques para formar caminos que el robot pueda tomar usando un sensor de infrarrojos o de color. Pueden apreciarse dichos caminos en la Figura 7.3.

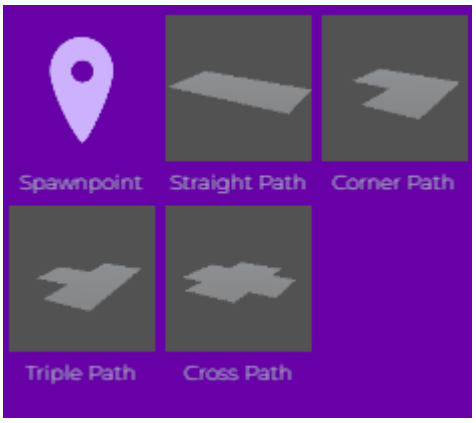


Figura 7.3: Sección miscelánea del editor

Cuando se incluyeron en el proyecto los bloques de construcción necesarios, pudimos ponernos a implementar el sistema de construcción, que es justo lo que hicimos. Para ello, se creó un nuevo objeto con un script “LevelCreatorManager”. Este script se encarga de manejar el sistema de construcción al completo, es decir, es capaz de guardar la información del nivel construido y es capaz de instanciar los elementos de dicha rejilla en el mundo.

Para lograr la funcionalidad completa del sistema, el script crea una matriz tridimensional de objetos, en donde almacenará los objetos instanciados en la escena. Cabe destacar que la matriz tiene un valor de dos de altura para dar una cierta libertad a la hora de construir en la misma al jugador, pero una altura mayor puede implementarse sin problema aunque no se consideró durante el desarrollo del proyecto. El script, al mismo tiempo, maneja los botones del menú, de manera que cada botón del menú entregue al script el bloque que se quiere colocar. Actualmente, el menú contiene cuatro secciones diferenciadas que pueden verse en la Figura 7.4.

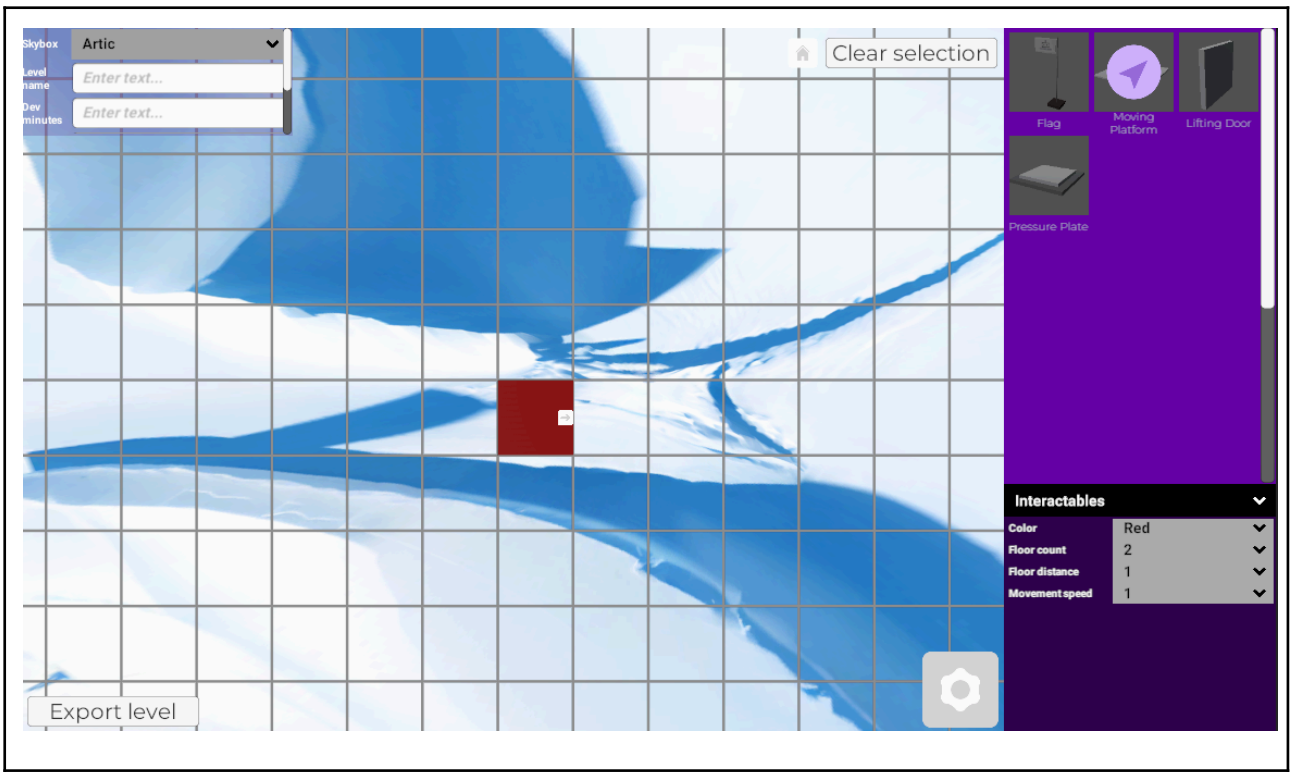


Figura 7.4: Editor con una plataforma seleccionada y un fondo de ártico elegido

En primer lugar, en la esquina superior derecha, podemos observar la zona de selección. Aquí, se almacenan todos los objetos de una sección dada, como puede ser la de elementos interactivos, que es la que aparece en la imagen. Haciendo uso de la barra negra de selección podemos cambiar entre secciones, mientras que si hacemos clic en los botones de cada bloque de construcción, los seleccionaremos avisando al usuario con un pequeño icono morado.

Desde el punto de vista del código, los objetos no están programados dentro de la clase “LevelCreatorManager”, sino que, como con todo lo demás, está diseñado para que un desarrollador pueda arrastrar y soltar cualquier prefab en la clase y agregarlo al editor para ser usado automáticamente. Podemos apreciar también una barra de “scroll” para mover el menú en el caso de que hubiesen más elementos que los que quepan y un botón “Clear selection”, que como su propio nombre indica, permite deseleccionar un bloque a colocar.

En la esquina superior izquierda podemos ver la segunda zona implementada, la zona de opciones. Esta zona es manejada por otro script llamado “LevelOptionsManager” y se encarga de recibir la información almacenada en dicha sección para usarla a la hora de exportar un nivel. En ella se guarda la información del nombre del nivel, el tiempo que tardó el desarrollador en terminar el nivel y el número de bloques que usó. Todo esto para crear el récord a batir por el jugador. También se especifica el “skybox” o fondo de nivel que tendrá el desafío.

Haciendo uso de varios plugins gratuitos de la tienda de assets de Unity, se incluyeron en el proyecto varios skyboxes para que el jugador pudiese elegir el que más quiera. Al momento de finalización del proyecto, la aplicación contiene 27 posibles skyboxes a seleccionar, desde desiertos hasta bosques pasando por valles o el ártico, que es justo el que se ve en la Figura 7.4.

La tercera zona implementada y la que más hemos comentado hasta ahora es la central, dicha sección es la zona de construcción y contiene el nivel principal con el que el usuario puede interactuar para editar. Para implementar el sistema de construcción, se hace uso de la clase “LevelCreatorManager” como ya hemos comentado antes, gracias a los eventos lanzados por los controles, podemos leer los valores del teclado y el ratón y por tanto crear unos controles personalizados e intuitivos para la construcción.

Con un elemento seleccionado, podemos hacer clic izquierdo sobre la rejilla para colocar el bloque, dicho bloque será instanciado por la clase y almacenado en la matriz interna del manager. Con clic derecho sobre un objeto ya colocado en la rejilla, el usuario será capaz de eliminarlo, borrando por completo la instancia del editor y de la matriz interna del script.

Haciendo uso de la rueda del ratón, el usuario puede también cambiar la orientación del bloque que va a colocar, de manera que paredes y elementos con orientación, como la plataforma horizontal, puedan colocarse como se prefiera.

Por último, haciendo clic en el botón central del ratón y arrastrando, puede moverse la rejilla para acceder a otras partes del nivel, cabe destacar que se implementó un botón extra de reset que puede verse con el icono de una casa en la esquina superior derecha, para volver al punto 0,0 de la rejilla de forma rápida. Un botón con “Export level” de texto puede verse en la esquina inferior izquierda, sin embargo hablaremos de ello más adelante. Podemos observar un ejemplo de nivel terminado en la siguiente Figura 7.5.

Figura 7.5: Ejemplo de nivel finalizado con el editor, el robot 1 debe seguir el camino rojo para poder abrir la puerta roja, el robot 2 debe seguir el camino verde para abrir la puerta verde y que el robot 1 pueda pulsar la placa amarilla, por último y con la puerta amarilla y roja abiertas, el robot 2 puede llegar a la bandera naranja y finalizar el nivel.

La última zona desarrollada y la más compleja es la zona de configuración, puede discernirse en la Figura 7.4 por el fondo púrpura oscuro en la parte inferior derecha del editor. Esta zona contiene todo lo necesario para poder configurar los elementos a colocar en la rejilla.

En la zona de configuración, puede establecerse el color de los bloques de construcción de entre una selección de 16 colores. También puede establecerse el contador de la bandera de finalización de nivel y puede configurarse cada elemento interactivo individualmente.

Para poder implementar dicha funcionalidad, cualquiera pensaría que está establecido desde dentro del “LevelCreatorManager”, pero como con todos los demás scripts, nuestro objetivo fue dar al desarrollador la mayor sencillez de implementación de nuevos elementos. Por ello, se creó una clase genérica de la que heredar llamada “GenericBlock”. Esta clase sirve como interfaz entre la clase del creador de niveles y cada bloque. De

dicha clase heredan también otras dos clases genéricas que usé para implementar el funcionamiento de los bloques de las diferentes secciones, un “GenericBuildingBlock” y un “GenericActiveInteractable”, junto con un “GenericPassiveInteractable”.

Desde la clase de “LevelCreatorManager”, el programa solo tiene que instanciar cada bloque genérico y preguntar por los elementos configurables a representar en el menú, mientras que las clases de cada bloque pueden albergar código personalizado para cada ocasión.

Para enseñar en el menú los posibles valores de cada configuración, se almacena en cada bloque una lista con opciones y sus posibilidades y una lista con las mismas opciones y las posibilidades elegidas realmente. Por ejemplo, un bloque genérico de “GenericBuildingBlock”, contiene una única opción llamada Color junto con todos los posibles colores y una lista con la opción Color y su color elegido que por defecto es blanco. El manager del nivel es capaz de conversar con la interfaz de “GenericBlock” y obtener y establecer estas opciones según la selección del usuario, de esta manera, cada bloque puede tener su propio código que maneje su funcionamiento al seleccionar una opción u otra.

En las clases que se implementaron, se especificaron tres tipos de bloque, un bloque de construcción, que como se explicó anteriormente almacena el color que usará, un bloque activo como puede ser la placa de presión, que obtiene todos los elementos pasivos de la escena y permite establecer el modo de funcionamiento de cada uno según se presione la placa o no, y un bloque pasivo que por defecto solo tiene la propiedad color como en los bloques de construcción, pero que se heredó para hacer una clase específica de la plataforma de movimiento, dicha implementación puede verse en la Figura 7.4 y como puede apreciarse, tiene las opciones de configuración de velocidad, color, distancia entre celdas y número de pisos.

Antes de entrar en el sistema de exportación e importación, se debe comentar que se implementó en las últimas semanas de desarrollo una pantalla de creación automática de niveles, en esta pantalla, podemos discernir tres botones principales, un botón de inicio en blanco, un botón de inicio por código y un botón de inicio por matriz.

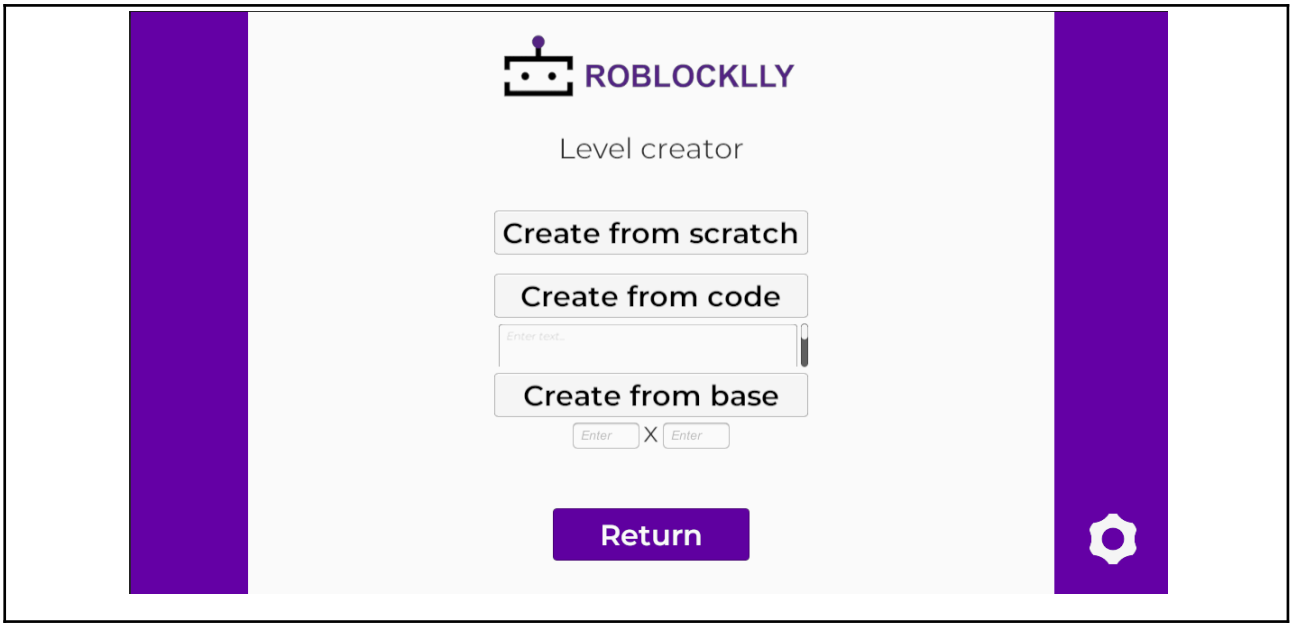


Figura 7.6: Botones de creación

En la Figura 7.6, podemos ver los botones comentados y varios campos de entrada. En primer lugar, el botón de inicio en blanco permite iniciar el editor con una escena vacía. Por otro lado, el botón de inicio desde código recibe un código de exportación y genera el nivel a editar a partir del mismo. En último lugar, el botón de inicio desde base recibe un valor “m” y un valor “n” y genera una matriz de bloques de tamaño (m x n) para empezar a trabajar directamente.

Por último y sin duda lo más complejo de todo el TFG ha sido el sistema de exportación e importación de niveles.

Con el objetivo de poder compartir los niveles desarrollados por los jugadores, se me pidió que se pudiesen exportar e importar en la aplicación, para ello, en un principio se consideró devolver los datos del nivel en un archivo, sin embargo, dado que la aplicación está desarrollada para ser implementada en un entorno web, la descarga de archivos es complicada y con la falta de tiempo para terminar el TFG nos decidimos por una opción más sencilla pero igualmente efectiva.

Para exportar los niveles se hace uso de la librería de Newtonsoft.Json, de esta manera, sería posible exportar la organización de cada nivel como una cadena de texto en formato Json.

Debido a la falta de compatibilidad entre los elementos “GameObject” de Unity y la librería de Json, no fue sencillo exportar los valores en la matriz de “LevelCreatorManager”, con lo que se tuvo que implementar desde cero el sistema de exportación de los bloques.

Como describir cada bloque en formato Json sería impensable y muy lento para desarrolladores que quieren añadir nuevos bloques, se decidió una vez más usar la clase genérica “GenericBlock” para obtener la cadena Json automáticamente. Dicha función de la clase, obtiene el nombre del bloque, la posición en la rejilla, la rotación y sus opciones y las codifica en formato Json, luego, desde la clase “LevelExportManager”, que es una nueva clase que maneja el sistema de exportación, se accede a dicha función y se obtiene la cadena Json de cada bloque de la matriz almacenada en “LevelCreatorManager”. Además, la clase también se comunica con “LevelOptionsManager” para obtener los valores de skybox, nombre del nivel y récord del desarrollador y los codifica también en formato Json. Por último, la función de ExportLevel(), que es la que es llamada por el botón de Export level de la zona de construcción del editor de niveles que puede verse en las Figuras 7.4 y 7.5, crea una cadena Json final en la que guarda los datos del entorno, los puntos de aparición, las banderas, el récord y los bloques del nivel todo por separado y lo enseña por pantalla para que el usuario pueda copiarlo y compartirlo.

Como resultado, un ejemplo simple de nivel con pocos bloques, dos puntos de aparición y dos banderas junto con elementos interactivos puede verse en el apéndice 1.

Para finalizar el TFG, se implementó un sistema de importación también, para simplificar el proceso de importación y mejorar la eficiencia, se incluyó el proceso de importación en la clase de “LevelManager”, de esta forma, si se recibe un código de la escena anterior, se creará el nivel de manera dinámica para que el jugador lo juegue. Puede verse la pantalla de carga en la Figura 7.7

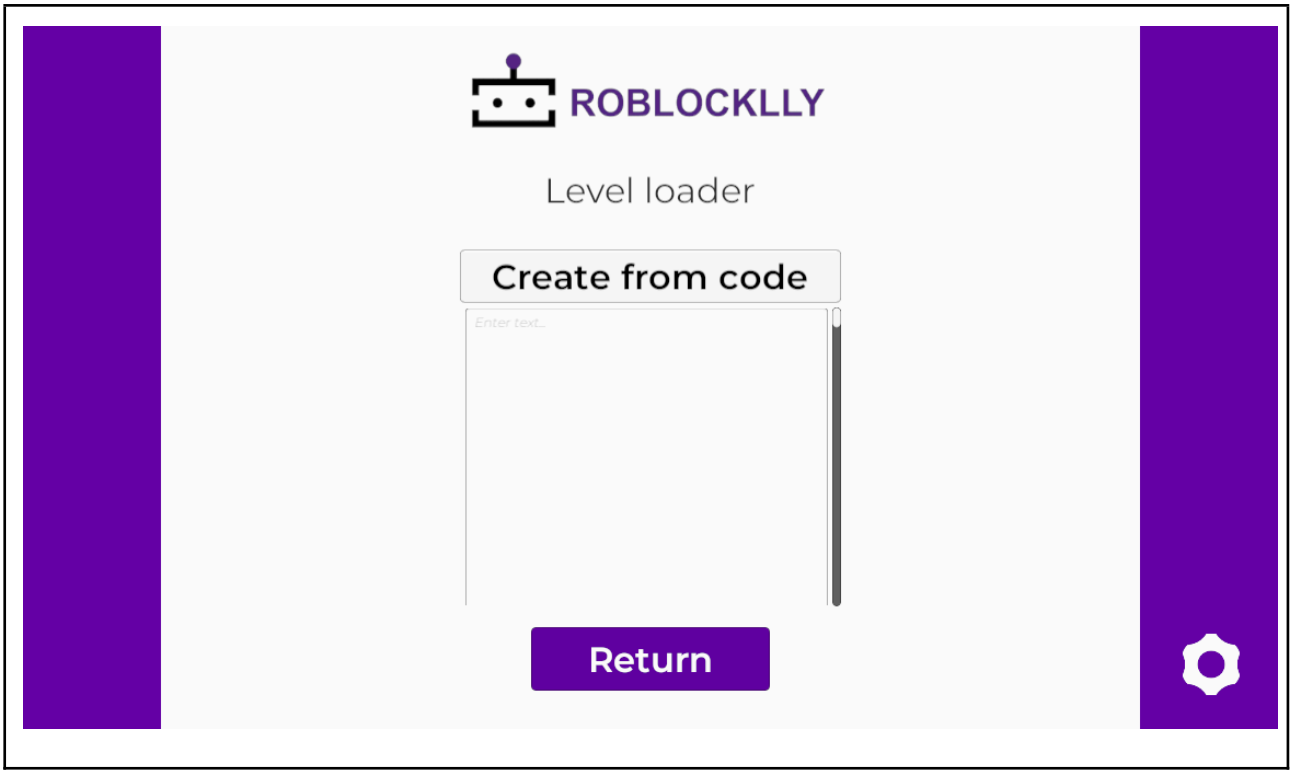


Figura 7.7: Menú de carga

Para la correcta importación del nivel, el script lee primero los valores de entorno y establece correctamente el skybox y el récord del desarrollador, luego, accede a la sección de banderas e instancia cada una con su correspondiente funcionamiento, a continuación, hace lo mismo con la sección de puntos de aparición. Primero los guarda en sus variables para usarlos a la hora de resetear las posiciones de los robots y luego instancia los mismos en dichas posiciones. Por último, el script accede a la sección “level”, que contiene los bloques y elementos interactivos y los instancia cada uno con sus determinadas opciones gracias a las clases genéricas.

Al finalizar, el nivel es completamente jugable y devuelve el control al jugador, que ahora es capaz de usar la zona de trabajo de uBlockly para calcular la solución al desafío. Como con los niveles originales, el resultado del nivel se calcula correctamente con los valores del récord del desarrollador.

Veamos en las siguientes Figuras 7.8 y 7.9 los códigos de una posible solución para el desafío de la Figura 7.5:

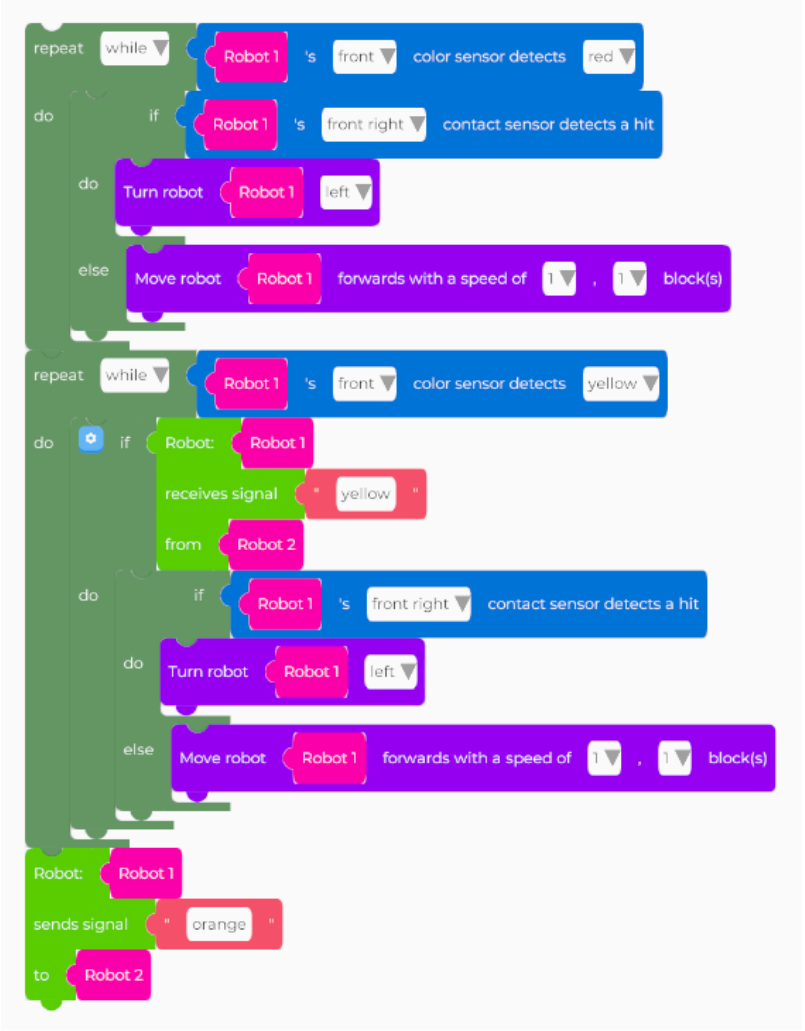


Figura 7.8: Código del Robot 1

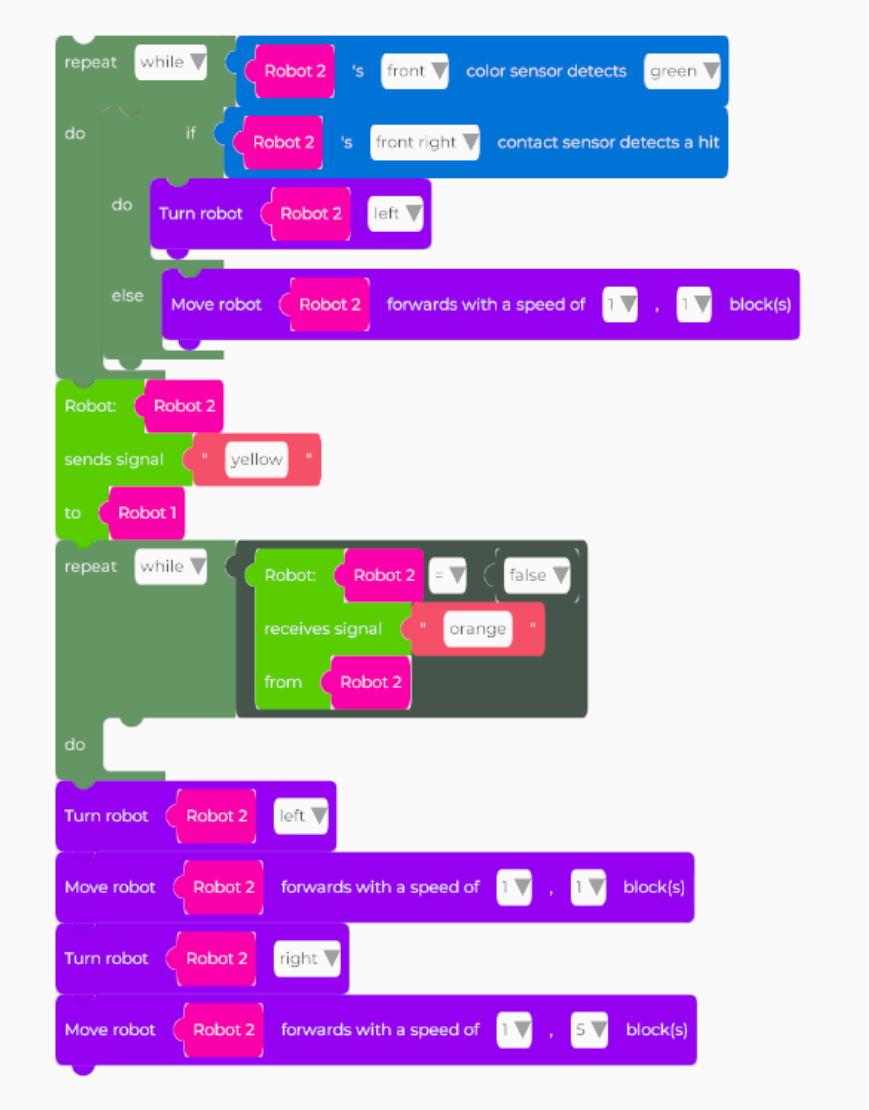


Figura 7.9: Código del Robot 2

En el código del robot 1, vemos cómo se llevará al robot hasta la placa de presión roja haciendo uso del sensor de colisión y del sensor de color, una vez se encuentre encima de la placa de presión y vea el color amarillo, esperará a la señal “yellow” del robot 2. Durante este periodo, el robot 2 habrá llegado a la plataforma verde siguiendo los mismos pasos que el robot 1 y, al haber llegado, esperará por la señal “orange” y enviará al robot 1 la señal “yellow”.

Al recibir la señal “yellow”, el robot 1 seguirá su programa, continuando por el camino amarillo hasta llegar a la plataforma, una vez llegue, parará ya que no encuentra más camino amarillo y lanzará el mensaje “orange” al robot 2 para avisarle de que las puertas están ambas abiertas.

El robot 2, al recibir al fin el mensaje “orange”, seguirá un camino preestablecido para llegar a la bandera naranja final y completar el nivel.

Capítulo 8 Conclusiones y líneas futuras

Este TFG consistió en el desarrollo y expansión de una aplicación de programación visual basada en bloques para el control de robots y la realización de niveles en un entorno tridimensional. Este objetivo se ha cumplido satisfactoriamente, implementando diversas funcionalidades y herramientas que han enriquecido el proyecto de gran manera.

Entre los objetivos completados del proyecto se encuentran la implementación del control de múltiples robots de forma simultánea, un sistema de comunicación entre robots, y la integración de un sistema de elementos interactivos, así como una placa de presión, una puerta y una plataforma de movimiento horizontal. Además, se ha desarrollado un editor de niveles que permite a los usuarios crear sus propios desafíos y compartirlos con otros jugadores y un sistema de calificación para fomentar la motivación de los participantes.

Durante el desarrollo del proyecto, se presentaron varios desafíos, donde tuve que irremediablemente refactorizar el proyecto original por completo. Estos desafíos fueron abordados con éxito y la implementación de nuevas escenas, clases y scripts, que mejoraron la estructura y funcionalidad del programa, serán muy útiles para el trabajo de futuros alumnos de la Universidad de La Laguna que busquen extender este TFG aún más, encontrando gran cantidad de documentación del funcionamiento de cada clase en los archivos con el código del script e implementaciones intuitivas y abiertas que servirán de base para construir encima sin problemas. Como posible línea futura o en el caso en el que yo continuase trabajando en este proyecto, consideraría añadir un sistema de multijugador, muchos más elementos interactivos ahora que la interfaz está disponible para los desarrolladores y efectos de sonido para los robots y el mundo, ya que creo que añadiría mucho carácter al proyecto.

El impacto de este proyecto es significativo en el ámbito de la educación, ya que funciona como una herramienta para que los estudiantes aprendan conceptos de programación y robótica de manera interactiva.

Como conclusión, me gustaría comentar que me ha encantado trabajar en este proyecto, la idea y ejecución me han parecido de lo más interesantes y me he encontrado en muchas ocasiones pasándolo bien durante el desarrollo de la aplicación, como si de un hobby se tratase. Al mismo tiempo, el proyecto ha sido desafiante y espero que otros

alumnos de la universidad decidan decantarse por trabajar en este TFG ya que me ha parecido una experiencia gratificante y me ha enseñado muchos elementos y conceptos que no conocía sobre la robótica, la programación y la plataforma de Unity.

Capítulo 9 Summary and Conclusions

This undergraduate thesis project consisted of the development and expansion of a visual programming application based on code blocks for robot control and the creation of levels in a three-dimensional environment. This objective has been successfully accomplished, implementing various functionalities and tools that have greatly improved the project.

Among the main objectives of the project, one of them was the implementation of the control of multiple robots simultaneously, a communication system between robots, and the development of an interactive object system, as well as a pressure plate, a door and a horizontal movement platform to go along with it. In addition, a level editor was developed that allows users to create their own challenges and share them with other players and a ranking system for further competition among participants.

During the development of the project, several challenges arose, where I had to irremediably refactor the original project completely. These challenges were successfully addressed and the implementation of new scenes, classes and scripts that improved the structure and functionality of the program, will be very useful for the work of future students of the University of La Laguna who seek to extend this project even more, finding a lot of documentation of each class in the script files and intuitive and open implementations that will serve as a basis to build on top of it without problems. As a possible future of the project or in the event that I continue to work on it, I would consider adding a multiplayer system, many, many more interactive elements now that the interface is available to the developers and sound effects, as I think it would add a lot of charisma to the simulator.

The impact of this project is significant in the field of education, as it works as a tool for students to learn programming and robotics concepts in an interactive way.

In conclusion, I would like to comment that I have loved working on this project, I found the idea and execution very interesting and I have found myself on many occasions having fun during the development of the application, as if it were a hobby. At the same time, the project has been challenging and I hope that other students at the university decide to work on this project because I found it a rewarding experience and it has taught me many things and concepts that I did not know about robotics, programming and the Unity development platform.

Capítulo 10 Presupuesto

La Tabla 10.1 contiene el tiempo aproximado de trabajo y coste.

Tabla 10.1: Tabla de tiempos

Sección	**Tiempo**	**Costo - 35€ la hora**
Desarrollo de la memoria	10 horas - desarrollo irregular a lo largo de 3 días	350€
Lectura, análisis y estudio del/para el proyecto	60~ horas - a lo largo de todo el desarrollo del proyecto	2.100€
Refactorización de la aplicación por completo	66 horas - 14 horas durante 3 días y 24 restantes a lo largo del proyecto para arreglar problemas emergentes y mejoras puntuales	2.310€
Control múltiple de robots	48 horas - 12~ horas en 4 días no consecutivos	1.680€
Sistema de comunicación y luces	9 horas - todo en un solo día	315€
Implementación de elementos interactivos	31 horas - 6~ horas durante 4 días no consecutivos y unas pocas extra para arreglar errores emergentes	1.085€
Sistema de calificación y récords	5 horas - todo en un solo día	175€
Editor de niveles	50~ horas - 8 horas al día por una semana y unas cuantas horas para arreglos y optimizaciones	1.750€
Exportador e importador de niveles	40 horas - 8 horas al día durante una semana	1.400€
Mejoras de último momento/añadidos pequeños o detalles	8~ horas - a lo largo de todo el proyecto, normalmente después de cada reunión con los tutores.	280€
Agregado	**327 horas, aproximadamente equivalentes a trabajar 5.5 horas por día laborable durante 3 meses.**	**11.445€**

Apéndice 1 - Código Json de ejemplo

Si analizamos el siguiente Json, que contiene los datos de un nivel exportado, podemos observar varios contenedores importantes.

El primer contenedor “environment” contiene todas las variables relacionadas al nivel como el nombre del desafío, el récord del desarrollador y el skybox a utilizar.

Justo debajo, el contenedor “spawnpoints” contiene todos los puntos de aparición a usar para los robots del nivel.

A continuación, el contenedor “flags” contiene todas las banderas a instanciar donde el robot pueda terminar el desafío, cada una de estas banderas y puntos de aparición contienen a su vez su posición en la rejilla, su rotación en cuaterniones y su nombre, además de posibles opciones que pudiesen tener como su contador, si es una bandera.

Por último, el contenedor “level” contiene todos los elementos del nivel, es decir, bloques, elementos interactivos y elementos misceláneos como los caminos. Cada uno de los bloques contiene a su vez la posición en la rejilla y orientación como con los puntos de aparición y banderas, pero también contienen todas y cada una de las opciones a establecer. Por ejemplo, el color de un bloque genérico o la configuración de una plataforma de movimiento horizontal.


```
{
  "environment": {
    "skybox": "Forest",
    "level_name": "Test level",
    "dev_minutes": "0",
    "dev_seconds": "21",
    "dev_blocks": "12"
  },
  "spawnpoints": [
    {
      "name": "Spawnpoint 0",
      "position": "(1, 0, 0)",
      "rotation": "(0.00000, 0.70711, 0.00000, -0.70711)",
      "options": []
    },
    {
      "name": "Spawnpoint 1",
      "position": "(-1, 0, 0)",
      "rotation": "(0.00000, 0.70711, 0.00000, 0.70711)",
      "options": []
    }
  ],
  "flags": [
    {
      "name": "Flag 0",
      "position": "(0, 0, 0)",
      "rotation": "(0.00000, 0.70711, 0.00000, 0.70711)",
      "options": [
        {
          "Timed": "False"
        },
```

```
      {
        "Seconds for win": "0"
      }
    ]
  },
  {
    "name": "Flag 1",
    "position": "(-1, 0, -1)",
    "rotation": "(0.00000, 0.70711, 0.00000, 0.70711)",
    "options": [
      {
        "Timed": "True"
      },
      {
        "Seconds for win": "5"
      }
    ]
  }
],
"level": [
  {
    "name": "Full Block 3",
    "position": "(0, 0, -1)",
    "rotation": "(0.00000, 0.00000, 0.00000, 1.00000)",
    "options": [
      {
        "Color": "White"
      }
    ]
  },
  {
    "name": "Full Block 5",
    "position": "(0, 0, 1)",
    "rotation": "(0.00000, 0.00000, 0.00000, 1.00000)",
    "options": [
      {
        "Color": "White"
      }
    ]
  },
  {
    "name": "Full Block 6",
    "position": "(1, 0, -1)",
    "rotation": "(0.00000, 0.00000, 0.00000, 1.00000)",
    "options": [
      {
        "Color": "White"
      }
    ]
  },
  {
    "name": "Moving Platform 0",
    "position": "(1, 1, -1)",
    "rotation": "(0.00000, 0.70711, 0.00000, 0.70711)",
    "options": [
      {
        "Color": "Red"
      },
      {
        "Floor count": "3"
      },
      {
        "Floor distance": "2"
      },
      {
        "Movement speed": "10"
      }
    ]
  },
  {
    "name": "Lifting Door 0",
    "position": "(0, 1, 1)",
    "rotation": "(0.00000, 0.00000, 0.00000, 1.00000)",
    "options": [
```

```
        {
          "Color": "Yellow"
        }
      ]
    },
    {
      "name": "Pressure Plate 0",
      "position": "(0, 1, -1)",
      "rotation": "(0.00000, 0.00000, 0.00000, 1.00000)",
      "options": [
        {
          "Moving Platform 0": "Activate"
        },
        {
          "Lifting Door 0": "Hold"
        }
      ]
    }
  ]
}
```

Bibliografía

[1] “Plataforma de desarrollo en tiempo real de Unity | Motor de 3D, 2D, VR y AR”. Unity. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://unity.com/es

[2] “GitHub - imagicbell/ublockly: Reimplementation of google blockly for Unity”. GitHub. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://github.com/imagicbell/ublockly

[3] CMU School of Computer Science. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://www.cs.cmu.edu/~15110-s13/Wing06-ct.pdf

[4] “Open Roberta Lab”. Open Roberta Lab. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://lab.open-roberta.org/

[5] “Official Colobot: Gold Edition website - International Colobot Community”. Official Colobot: Gold Edition website - International Colobot Community. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://colobot.info/

[6]“Erebus - Educational Rescue Simulation Platform”. Erebus. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://erebus.rcj.cloud/

[7] “RoboMind.net - Welcome to RoboMind.net, the new way to learn programming.” [En línea]. Disponible: https://www.robomind.net/es/“

[8] Json.NET - newtonsoft”. Json.NET - Newtonsoft. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://www.newtonsoft.com/json

[9] “Robotics simulator – edasim”. Edasim – Integrating Ideas. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://edasim.com/robotics-simulator/

[10] “Simulate FIRST LEGO league & WRO | virtual robotics toolkit”. Simulate FIRST LEGO League & WRO | Virtual Robotics Toolkit. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://www.virtualroboticstoolkit.com/

[11] “RoboSim”. RoboSim. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://robosim.stemtown.com/

[12] “MDN web docs”. MDN Web Docs. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://developer.mozilla.org/en-US/

[13] “GitHub - Computational-Thinking/RoblockLLy-Source: Source code of RoblockLLy, an educational robotics simulator based on Unity and UBlockly”. GitHub. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://github.com/Computational-Thinking/RoblockLLy-Source

[14] “Unity asset store - the best assets for game making”. Unity Asset Store - The Best Assets for Game Making. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://assetstore.unity.com/

[15] “Unity - manual: Unity user manual 2022.3 (LTS)”. Unity - Manual: Unity User Manual 2022.3 (LTS). Accedido el 10 de julio de 2024. [En línea]. Disponible: https://docs.unity3d.com/Manual/index.html

[16] “Unity documentation”. Unity Documentation. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://docs.unity.com/

[17] “Empowering the world to develop technology through collective knowledge – Stack Overflow”. Empowering the world to develop technology through collective knowledge – Stack Overflow. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://stackoverflow.co/

[18] “Bevor Sie zu YouTube weitergehen”. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://www.youtube.com/c/unity

[19] “Game dev beginner - learn unity game development”. Game Dev Beginner. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://gamedevbeginner.com/

[20] “Personal cloud storage & file sharing platform - google”. Personal Cloud Storage & File Sharing Platform - Google. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://drive.google.com/

[21] “GitHub: Let’s build from here”. GitHub. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://github.com/

[22] “Microsoft Learn: Adquirir conocimientos que le abran las puertas en su carrera profesional”. Microsoft Learn: Build skills that open doors in your career. Accedido el 10 de julio de 2024. [En línea]. Disponible: https://learn.microsoft.com/es-es/