UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Un Simulador Web para aprender las preferencias y necesidades del humano en sistemas human-in-the-loop

Trabajo Fin de Máster

Máster Universitario en
Ingeniería y Tecnología de Sistemas Software

Departamento de Sistemas Informáticos y Computación

Autor/a: Diago Sans, Marc

Tutores: Albert Albiol, Manoli

Gil Pascual, Miriam

Pelechano Ferragud, Vicente

Curso 2019-2020

Resumen

Los sistemas autónomos pueden reducir de forma importante la carga de trabajo de los humanos. Sin embargo, no todos los sistemas pueden realizar todas las tareas de forma completamente autónoma, por lo que en muchas ocasiones es necesario que el humano ayude a completar su funcionalidad ('human-in-the-loop'). A los sistemas con participación del humano los llamamos sistemas HiL. Cuando se diseñan sistemas HiL un aspecto clave es realizar un diseño adecuado de cómo se va a realizar el trabajo colaborativo entre humano y sistema. Esto significa fundamentalmente especificar cual va ser el trabajo que debe realizar el humano, cual va a ser el trabajo que va a realizar el sistema, cual es el flujo de control entre las actividades que realizan cada uno, y qué interacciones se van a llevar a cabo para establecer la comunicación entre ambos. Esta tarea no es sencilla y requiere conocer las necesidades y preferencias de los usuarios. Esto es especialmente importante en los sistemas HiL para conseguir una colaboración entre el sistema y el usuario capaz de involucrar al humano de forma adecuada ante situaciones donde no es posible alcanzar la autonomía, procurando garantizar una correcta integración humano-sistema. Imaginemos un robot que transporta mercancías en un almacén. Este robot tiene que realizar algunas tareas, como la carga y descarga, en colaboración con el humano. Si la forma de cooperación para la carga y descarga no se adecúa a la forma de trabajar del humano (por ejemplo, el robot le pide realizar una carga al humano cuando este está ocupado) o no consigue comunicarse de forma eficiente con él (por ejemplo, el humano no sabe si el robot está o no preparado para cargar), el humano puede cometer errores durante la carga o descarga, o incluso no aceptar el robot como colaborador de trabajo.

Establecer estas necesidades y preferencias no siempre es posible cuando se aborda el diseño de sistemas HiL, lo que origina sistemas que no consiguen una adecuada participación del humano y conducen a una mala experiencia de usuario. El presente TFM propone la construcción de un simulador de soluciones HiL para que los usuarios puedan probar las soluciones antes de ser implementadas, se recoja el feedback de estos usuarios y se analice de forma automática para mejorar los diseños. El simulador propuesto es un simulador genérico que se configurará de forma fácil para cada solución HiL concreta. Este simulador se desarrollará como una aplicación web, de forma que pueda ser probado por un gran número de usuarios. El feedback de estos usuarios se almacenará en la nube y posteriormente se analizará a través de herramientas de análisis automáticas. Estas herramientas procesarán los datos y proporcionarán informes que ayuden a identificar las necesidades y preferencias de los usuarios y a partir de ellas mejorar los diseños. Los nuevos diseños conllevarán la modificación del simulador, y la vuelta al ciclo de prueba.

Palabras clave: human-in-the-loop; simulador web; interacciones usuario-sistema; diseño de interacciones; preferencias de usuario; feedback de usuario; análisis de feedback

Abstract

Autonomous systems can significantly reduce the workload of humans. However, not all systems can perform all tasks completely autonomously, so in many cases it is necessary for humans to help complete their functionality ('human-in-the-loop'). We call systems with human participation HiL systems. When designing HiL systems, a key aspect is to make an adequate design of how the collaborative work between human and system is going to be carried out. This fundamentally means specifying what work the human must do, what work the system is going to do, what is the control flow between the activities that each one performs, and what interactions are going to be carried out. out to establish communication between the two. This task is not easy and requires knowing the needs and preferences of users. This is especially important in HiL systems to achieve a collaboration between the system and the user capable of involving the human in an adequate way in situations where autonomy is not possible, trying to guarantee a correct human-system integration. Let's imagine a robot that transports goods in a warehouse. This robot has to perform some tasks, such as loading and unloading, in collaboration with the human. If the form of cooperation for loading and unloading does not suit the way of working of the human (for example, the robot asks the human to carry out a load when he is busy) or does not manage to communicate efficiently with him (for example , the human does not know whether or not the robot is ready to load), the human may make mistakes during loading or unloading, or even not accept the robot as a work partner.

Establishing these needs and preferences is not always possible when approaching HiL system design, leading to systems that do not get adequate human participation and lead to poor user experience. This TFM proposes the construction of a HiL solutions simulator so that users can test the solutions before being implemented, collect feedback from these users and automatically analyze it to improve designs. The proposed simulator is a generic simulator that will be easily configured for each specific HiL solution. This simulator will be developed as a web application, so that it can be tested by a large number of users. The feedback from these users will be stored in the cloud and subsequently analyzed through automatic analysis tools. These tools will process the data and provide reports that help identify the needs and preferences of users and, based on them, improve the designs. The new designs will entail modifying the simulator, and a return to the test cycle.

Key words: human-in-the-loop; web simulator; user-system interactions; interaction design; user preferences; user feedback; feedback analysis

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En este documento se expone el trabajo de fin de máster correspondiente al Máster Universitario en Ingeniería y Tecnología de Sistemas Software de la Universitat Politècnica de Valencia.

En este capítulo se describirán los motivos para el desarrollo de este trabajo, los objetivos que se esperan lograr, la metodología empleada para el desarrollo del proyecto, la planificación seguida y la estructura utilizada durante el proceso de desarrollo de esta memoria.

1.1 Motivación

El mundo inteligente del futuro se está diseñando como complejos ecosistemas compuestos por una amplia variedad de dispositivos y servicios distribuidos. Ante esta situación, aumenta la necesidad de desarrollar sistemas que sean acapaces de adaptarse automáticamente en tiempo de ejecución.

Esta auto-adaptación surge como una solución para la gestión de este tipo de sistemas software que permitan hacer realidad sistemas autónomos.

Aunque las soluciones completamente autónomas han resultado satisfactorias en muchos dominios, la capacidad de estos sistemas de proporcionar servicios de confianza en presencia de cambios inesperados se ve afectada por su creciente complejidad, así cómo la naturaleza impredecible de los entornos en los que tienen que operar.

Como mecanismo para dar solución a este problema, parece interesante hacer partícipe al humano para que ayude al sistema en situaciones que resultarían difíciles de resolver al sistema autónomo. Estos mecanismos se llaman sistemas *Human-in-the-Loop*.

Los sistemas *Human-in-the-Loop* describen el proceso dónde una máquina o sistema software que es capaz de resolver un problema necesitando la intervención de un humano, creando un continuo bucle de retroalimentación entre ambos permitiendo que el sistema sea más eficiente constantemente.

Los sistemas HiL se centran en la especificación del control y interaciones necesarias para funcionar de forma colaborativa entre los humanos y los sistemas

para completar correctamente una funcionalidad particular, a este concepto se le llama tarea colaborativa.

La recopilación de *feedback* de los usuarios a través de la validación de prototipos es un elemento fundamental en todos los procesos de diseño centrado en humanos. Sobre esa base, nuestro enfoque propone el uso de prototipos para que los diseñadores validen los diseños de colaboración del sistema humano con los usuarios y utilicen sus comentarios para lograr diseños mejorados.

Los prototipos HiL son un método dónde los diseñadores pueden validar el flujo de interacciones, los recursos de atención de estas y el *feedback* o información que se le proporciona al humano.

Estos prototipos permiten simular diferentes configuraciones de contexto para comprobar el comportamiento de los humanos en los distintos contextos. Además, este tipo de validación no requiere de una evaluación *in situ* de un contexto de uso real con un prototipo completamente funcional. Una estructura mínima para simular el entorno del prototipo es suficiente para representar los aspectos que permitirían a los diseñadores y expertos verificar si el diseño es adecuado antes de definir otras partes del sistema.

Esto supone un enfoque que busca permitir a los diseñadores validar sus sistemas o prototipos HiL con los usuarios y utilizar sus comentarios para mejorar los diseños.

La validación de los prototipos, por parte del usuario, proporciona un valioso *feedback* que debe analizarse. Los diseñadores deben prestar especial atención a la forma en que los usuarios interactúan con el prototipo para determinar las preferencias y necesidades de los usuarios y adaptarlos en consecuencia.

Para dicha validación, se propone los siguientes pasos:

1. **Generar un prototipo HiL**. El prototipo HiL es generado gracias a un *framework software*, nos referiremos a partir de ahora a este *framework* cómo *framework HiL*. Esto nos permite generar prototipos basados en tareas colaborativas en base al diseño HiL implementado en dicho *framework*.

2. **Validar el prototipo HiL**. Debido a que un prototipo HiL se ejecuta en un entorno real complejo, se ha propuesto una infraestructura de simulación para reproducir visualmente la ejecución del prototipo generado por el *framework HiL*. Esta infraestructura de simulación, al cual se le ha nombrado Simulador HiL, permite recoger información de las interacciones de los usuarios mientras estos validan el sistema.

3. **Inferir las preferencias de los usuarios**. Se ha desarrollado una herramienta IA que continuamente recibe *feedback* de las interacciones de los usuarios a través del simulador HiL. A partir de esta información, la herramienta IA, llamada dentro del sistema cómo la red neuronal, inferirá en modelos de predicción con la información obtenida del simulador mediante técnicas de *machine learning*.

4. **Adaptar el prototipo HiL**. La inferencia de la red neuronal se usa para autoadaptar el prototipo HiL a las preferencias del usuario en tiempo de ejecución.

Este proyecto se centra principalmente en el punto número 2, diseñar y desarrollar un simulador capaz de representar gráficamente estas tareas colaborativas de los prototipos HiL y de auto configurarse para representar las mejoras de la red neuronal al prototipo o sistema HiL que se esté validando.

1.2 Objetivos

Este proyecto se centra en el diseño y desarrollo de una infraestructura de simulación basada en la propuesta de la validación de prototipos HiL descrita en la motivación.

Esta infraestructura de simulación, o simulador HiL, será parte de un sistema mayor formado por un *framework HiL*, capaz de implementar tareas colaborativas, y una red neuronal capaz de analizar datos de interacciones y proponer mejoras al sistema. Todo este sistema ha sido desarrollado por el grupo TaTAmI de la Universitat Politécnica de Valéncia.

De esta forma, el objetivo principal de este proyecto es el de desarrollar un simulador generalista para cualquier tipo de situación al que un sistema autónomo se deberá enfrentar, permitiendo las pruebas de las tareas dentro de un entorno seguro y configurable.

Para lograr este objetivo principal se persiguen los siguientes sub-objetivos:

- Cohexistir con los dos sistemas adicionales dentro de la validación de prototipos HiL.
- El simulador deberá poder comunicarse con el *framework HiL* y representar gráficamente cualquier tarea colaborativa que este implemente.
- Generar una recogida de información de las interacciones del usuario para poder dotar de datos de prueba la red neuronal. De esta forma, podrá mejorar sus resultados y proporcionar mejores autoconfiguraciones del prototipo o sistema HiL.

Para lograr estos objetivos y sub-objetivos, se ha propuesto utilizar el motor gráfico *Unity3D*, así cómo el protocolo de comunicación, frecuente en sistemas pertenecientes a la internet de las cosas, *MQTT*.

1.3 Metodología

Se ha decidido utilizar una metodología de desarrollo iterativo e incremental. En este tipo de metodología el proyecto se divide en bloques llamados iteraciones. De esta forma, se podían ir haciendo pruebas y correcciones del diseño del simulador sin tener que esperar al desarrollo completo.

En la primera iteración se desarrollará un simulador para el caso del coche HiL propuesto. En la segunda iteración el simulador incrementará su funcionalidad para poder aceptar cualquier otra solución HiL que siga la especificación del

framework. Y finalmente, la tercera iteración permitirá la conexión con una red neuronal para reconfigurar los sistemas de interacción en tiempo de ejecución.

1.4 Planificación

La planificación aplicada en este proyecto es la siguiente:

- **Análisis de requisitos:** Periodo de tiempo dedicado a determinar las necesidades del sistema, tanto funcionales cómo no funcionales.
- **Diseño:** Esta fase consiste en el diseño funcional y de la interfaz de usuario de la solución propuesta.
- **Implementación inicial:** Desarrollo e implementación de la primera iteración del proyecto, definiendo las bases en las que las siguientes iteraciones se implementarán.
- **Pruebas:** En todas las fases de pruebas, el simulador se probará con distintos perfiles de usuarios para comprobar si cumple los requisitos.
- **Generalización del simulador:** En esta fase se implementa la segunda iteración.
- **Refactorización de los sistemas de interacción:** Esta fase se desarrolla junto a la integración con IA. El objetivo es refactorizar los sistemas de interacción del usuario con el simulador para que permitan la implementación de la tercera iteración.
- **Integración con IA:** El objetivo principal de esta fase es el de implantar la tercera iteración del desarrollo.
- **Corrección de errores:** Esta fase consiste en corregir todos los errores menores y mayores identificados durante las pruebas, además de realizar los últimos ajustes antes de su puesta en marcha.
- **Puesta en marcha en web:** En esta fase final, el simulador se desplegará en un servidor web para que pueda ser utilizado.

En cuanto a la planificación temporal, se optó por una tabla de marcas de tiempo estimados de inicio y fin para cada una de las tareas del proyecto. De esta forma, se puede observar la dedicación estimada a cada fase. La Tabla 1.1 describe dicha planificación temporal.

Actividad	Inicio	Fin
Análisis de requisitos	01/01/2020	10/02/2020
Diseño		
Diseño de la interfaz	11/02/2020	25/02/2020
Diseño funcional	20/02/2020	05/03/2020
Implementación inicial		
Implementación de la interfaz	06/03/2020	15/03/2020
Implementación lógica y funcional	13/03/2020	09/04/2020
Pruebas	09/04/2020	14/04/2020
Correcciones respecto a pruebas	14/04/2020	10/05/2020
Segundas pruebas	10/05/2020	13/05/2020
Generalización del simulador	14/05/2020	10/06/2020
Integración con IA		
Sistema configurable en ejecución	14/05/2020	26/06/2020
Exportación del estado de la simulación	27/05/2020	06/06/2020
Recepción y configuración del sistema	11/06/2020	26/06/2020
Terceras pruebas	26/06/2020	03/07/2020
Corrección de errores	26/06/2020	03/07/2020
Puesta en marcha en web	03/07/2020	10/07/2020

Tabla 1.1: Tabla de planificación estimada

1.5 Estructura de la memoria

El documento ha sido dividido en los siguientes ocho capítulos:

- **Introducción:** Se indica la motivación para la realización de este proyecto, los objetivos a cumplir, el impacto esperado de estos objetivos, la metodología y planificación del trabajo y las convenciones.
- **Estado del arte:** Se describe la situación del estado del arte relativo al proyecto.
- **Análisis del problema:** Se especifican las posibles soluciones y la solución propuesta al proyecto, así cómo los casos de uso, actores y requisitos del sistema.
- **Solución propuesta:** Se detalla el diseño, arquitectura y tecnologías utilizadas en la realización del simulador, detallando como ha ido evolucionando a través del desarrollo iterativo, además de prestar atención a los problemas surgidos durante el proceso.
- **Implantación:** Este capitulo detalla la etapa en el que el simulador se ha realizado y se ha llevado a explotación.
- **Pruebas:** Se detallan las pruebas y validaciones realizadas para el prototipo del videojuego, usando como modelo el caso del coche autónomo.
- **Caso del coche autónomo:** En este capítulo se describe las decisiones de diseño y configuraciones realizadas para el caso de prueba del coche autónomo desarrollado cómo ejemplo de funcionamiento del simulador.

- **Conclusiones:** Conclusión de la memoria indicando los objetivos cumplidos, trabajos futuros y posibles iteraciones adicionales para la mejora del proyecto.

Adicionalmente a los capítulos listados anteriormente, se han incluido dos apartados finales de bibliografía y un apéndice con la información adicional agregada a la memoria.

1.6 Convenciones

En este documento, se han utilizado distintas convenciones:

- Las siglas se han puesto en cursiva y se han definido en el apéndice de glosario.
- Las palabras que se han escrito en otros idiomas, sea el motivo que es la forma más utilizada aunque disponga de una traducción o que directamente no tenga traducción, también se han escrito en cursiva y se han definido en el apéndice de glosario.
- Los bloques de código o archivos del sistema, se han incluido en forma reducida, enfatizando la parte de la que se está hablando. Cualquier archivo que requiriera ser mostrado completamente se ha incluido cómo un apéndice al final de la memoria.

CAPÍTULO 2
Estado del arte

La internet de las cosas, o *IoT*, ha sufrido una transformación fundamental en las últimas décadas, partiendo de la tecnología heredada de identificación por radiofrecuencia, las redes inalámbricas de sensores, hasta su forma actual, una red cada vez más interconectada y heterogénea. [1]

La *IoT* a día hoy en día ya es una fusión de numerosas herramientas y dispositivos en red, equipados con inteligencia computacional avanzada y grandes capacidades de comunicación.

En general, los principios de la *IoT* contemporánea se superponen a dominios adyacentes, incluida la informática móvil o incluso la robótica, con aplicaciones que van desde redes sociales basadas en teléfonos inteligentes hasta reducir el tráfico y la contaminación en las ciudades inteligentes.

Sin embargo, la *IoT* era una multitud de máquinas interconectadas e inteligentes que se comunican entre sí y se adaptan de manera autónoma sin la participación de una persona.

De hecho, los sistemas modernos de *IoT* aún desconocen ampliamente el contexto humano y, en cambio, consideran que las personas son un elemento externo e impredecible.

Por lo tanto, las futuras aplicaciones de *IoT* necesitarán involucrar íntimamente a los humanos, para que las personas y las máquinas puedan operar sinérgicamente.

Este concepto es conocido cómo *Human-in-the-Loop*. El HiL abre la puerta a las plataformas IoT orientadas a las personas, que son conscientes del contexto, la movilidad e incluso el estado de ánimo de la gente, por lo que tienen una manipulación más eficiente e intuitiva.

Gracias a la aparición de la Industria 4.0, se ha disparado la integración de estos sistemas *Human-in-the-Loop*. La lógica de la Industria 4.0 prevé humanos y máquinas como partes indistinguibles de un cuerpo heterogéneo más grande de entidades autónomas y cooperativas distribuidas. [2] [3]

Con esto, los sistemas auto-adaptativos han emergido y superado las limitaciones de la supervisión de las personas, mediante sistemas con mecanismos capaces de auto adaptar su estructura y comportamiento en tiempo de ejecución. [4]

Sin embargo, esto ha llevado a que muchos expertos consideren que los participantes humanos son fácilmente influenciados por factores externos, como nivel de conocimientos del sistema que controlan, estrés o fatiga. Esto puede determinar el éxito de una tarea en particular, el tiempo que llevará o si estarían dispuestos a realizarla desde un principio.

De esta forma, los futuros sistemas autónomos deberán ajustarse a la situación del usuario, sus hábitos y interacciones con el entorno. Estas aplicaciones adaptativas deberán cambiar autónomamente de acuerdo a los contextos dónde se encuentre el ususario. [5]

De la necesidad de diseñar sistemas auto adaptativos y autónomos nacen propuestas cómo el *framework HiL*. Un sistema que permite la especificación de tareas colaborativas.

El concepto de una tarea colaborativa representa una funcionalidad que requiere la interacción entre un humano y un sistema. Las tareas colaborativas requieren a un humano con unas capacidades y conocimientos específicos para poder realizarlas.

Estos humanos son representados con el concepto de un perfil humano. Y son las personas las cuales encajan con los requisitos asociados a este perfil, los únicos que deberían asistir al sistema para realizar una tarea.

Estos perfiles humanos pueden realizar acciones para proporcionar información o identificar situaciones del contexto al sistema de la tarea que se está realizando. Estas notificaciones pueden hacerse mediante sensores o mecanismos de interacción que el sistema disponga.

Gracias a propuestas cómo el *framework HiL* ahora pueden ser especificadas tareas de sistemas autónomos y auto adaptativos. La importancia de que estas tareas puedan ser especificadas en un lenguaje de especificación especifico crea la posibilidad de la aparición de simuladores HiL para probar estas tareas antes de su puesta en marcha.

Distintos simuladores han sido protitipados con la idea de probar estas tareas antes de su implementación. [6] [7] [8]

2.1 Crítica al estado del arte

En cuanto al proyecto se refiere, la crítica al estado del arte se centrará en el estado de los simuladores de sistemas autónomos HiL.

El principal inconveniente de estos simuladores se encuentra a que están centrados en dominios específicos.

J. Grono habla de un simulador de generadores, capaz de simular ondas reales y cómo varios generadores se sincronizan entre ellos. [6] Este simulador presenta una interfaz gráfica con instrumentos virtuales y controles para poder operar con el.

Este simulador de generadores no reproduce ninguna tarea en concreto, se limita a simular las acciones del usuario y la funcionalidad detrás de este calcula y muestra el *output*.

En cuanto al simulador propuesto por T. Gyorgy et al. presentan un simulador de un coche eléctrico capaz de simular la conducción de dicho coche por una ciudad. [8] Prediciendo cómo el coche reaccionará con las calles y pendientes. Centrándose en cómo el coche reaccionará y no en cómo el humano interactua con el este.

En cuanto al simulador propuesto por W. Lei et al. si que utilizan dispositivos hardware para analizar el comportamiento humano en el sistema, pero este se limita únicamente con el dominio de un coche. [7]

La crítica al estado del arte muestra que hay iniciativas por estos simuladores HiL, pero carecen de simuladores generalistas para cualquier tipo de dominio o sistema.

2.2 Propuesta

En este trabajo se ha propuesto un simulador que permita probar en un entorno controlado sistemas autónomos *HiL* antes de ser probados con dispositivos reales. El simulador propuesto será genérico, permitiendo cualquier configuración de situaciones, para poder garantizar que cualquier sistema pueda ser probado.

El simulador busca que los expertos y diseñadores de sistemas autónomos prueben y obtengan información de sus sistemas antes de ser puestos en marcha en dispositivos o maquinarias reales. Para esto, el simulador genera registros de todas sus ejecuciones. También permitirá la conexión a una red neuronal a la cual le transmitirá los mismos registros para que esta pueda analizarlos y realizar cambios en caliente de la configuración del sistema, de forma que este pueda auto mejorarse.

El proyecto se ha desarrollado mediante Unity3D, un motor gráfico utilizado normalmente en videojuegos pero que permite aplicaciones mas allá de estos. Unity3D se ha configurado mediante la *platform* WebGL, la cual permite exportar los proyectos para ser utilizados en entornos web. Además, toda la configuración del simulador con el resto de sistemas que lo componen se realiza mediante el protocolo *MQTT*.

CAPÍTULO 3
Análisis del problema

Al igual que en el desarrollo de otras aplicaciones, antes de empezar a trabajar en un proyecto software, hay que realizar una investigación y análisis de posibles soluciones, así cómo determinar los casos de uso y requisitos del sistema. Esto es necesario para realizar un correcto desarrollo del sistema, evitar mal interpretaciones de objetivos y de esta forma, comprobar que el desarrollo ha cumplido los objetivos una vez finalizado el proyecto.

3.1 Identificación y análisis de soluciones posibles

Antes de empezar con el diseño del simulador, se hizo un estudio de simuladores existentes para identificar que soluciones podrían ser adecuadas para nuestro proyecto. Se evaluaron también costes de tiempo y esfuerzo de los distintos simuladores para poder evaluar la viabilidad del desarrollo de estos.

Durante el *brainstorming*, aprovechando los conocimientos de videojuegos del autor, se habló de integrar una experiencia *3D* al simulador, planteando incluso el uso de las gafas de realidad virtual *Oculus Rift* de *Facebook*. Pero un sistema así sería demasiado complejo de desarrollar por una sola persona y llevaría demasiado tiempo, al igual de que era insostenible alojar un simulador tan pesado en un servicio en la red.

A su vez, se consideró un simulador mucho más básico, con imágenes estáticas y planos generales. El grupo ya disponía de un simulador similar con el que hacían pruebas y la idea planteada consistía en una versión mejorada de este. Imágenes como botones, que al pulsarlas se abrían *pop-ups* con mecanismos de interacción, un sistema altamente configurable, pero que a la vez era demasiado simple y no tan atractivo a la vista del usuario.

Figura 3.1: Boceto del simulador básico

Las propuestas anteriores y sus variantes cercanas fueron descartadas, unas por complejidad y alto coste de desarrollo, y otras por simpleza y poca mejora respecto al sistema existente. El simulador debería centrarse en permitir que el usuario pueda probar de una forma realista como sería la interacción con el sistema autónomo HiL. Este objetivo hizo que el desarrollo se centrara en buscar de que forma la primera opción podría simplificarse y no ser tan pesada computacionalmente.

Finalmente, se propuso juntar las ideas clave de todas las propuestas. Al igual que en la propuesta de las gafas de realidad virtual, la idea de plantearlo como un videojuego volvió al debate. Los *serious games* son videojuegos que sustituyen el propósito de la diversión o entretenimiento para ofrecer un entorno educativo o de formación. No se quería llegar al punto de que el simulador tenga objetivos o sea una prueba mental, pero si utilizar componentes de los videojuegos cómo el *game feel*[9].

Por lo que se propuso un simulador utilizando *UnityWebGL* para poder añadir distintos elementos gráficos *2D* sin aumentar el coste de recursos necesarios para ejecutarlo. Gracias al componente *Animator* del motor gráfico, es posible generar distintas animaciones básicas para los elementos gráficos y que la simulación no de una sensación de estaticidad. Además, la capacidad de generar diversos canales de audio, es posible crear distintos tipos de sonido que pueden reproducirse de forma simultanea, cómo *Text-to-speech* y sonidos del sistema, entre otros.

Este simulador será capaz de conectarse con un *framework* que implementa tareas colaborativas de sistemas HiL para representar gráficamente cómo estas se desarrollan y permitiendo a los usuarios interactuar para probarlas en un entorno simulado.

Adicionalmente, será capaz de conectarse a una red neuronal capaz de analizar las interacciones de los usuarios y podrá sugerir cambios a la tarea simulada.

3.2 Especificación de requisitos

En esta sección se definen los requisitos software pertenecientes al simulador. Los requisitos han sido generados a partir de reuniones con los tutores y la investigación previa al desarrollo.

La siguiente tabla representará la forma en la que se definen los requisitos:

Identificador	
Nombre	
Descripción	
Prioridad	

Tabla 3.1: Tabla de ejemplo de requisitos

Cuya descripción de cada uno de los campos es el siguiente:

- *Identificador*: Código identificativo único de cada requisito.
- *Nombre*: Nombre descriptivo del requisito.
- *Descripción*: Definición simple y concisa del requisito especificado.
- *Prioridad*: Urgencia para realizar el requisito.

En cuanto a los requisitos han sido divididos en funcionales y no funcionales.

3.2.1. Requisitos funcionales

Esta sección consiste en la enumeración de los distintos requisitos funcionales del sistema utilizando la tabla especificada anteriormente:

RF-01	
Nombre	Iniciar simulación de tarea
Descripción	El simulador deberá permitir iniciar una nueva simulación de una tarea.
Prioridad	Alta

Tabla 3.2: RF-01: Iniciar simulación de tarea

RF-02	
Nombre	Interactuar con elementos mediante el ratón.
Descripción	El usuario podrá interactuar con los elementos del simulador mediante el ratón.
Prioridad	Alta

Tabla 3.3: RF-02: Interactuar con elementos mediante el ratón

RF-03

Nombre	Interactuar con elementos mediante el teclado.
Descripción	El usuario podrá interactuar con los elementos del simulador mediante el teclado.
Prioridad	Media

Tabla 3.4: RF-03: Interactuar con elementos mediante el teclado

RF-04

Nombre	Interactuar con elementos mediante voz.
Descripción	El usuario podrá interactuar con los elementos del simulador mediante voz.
Prioridad	Baja

Tabla 3.5: RF-04: Interactuar con elementos mediante voz

RF-05

Nombre	Conectarse al *framework HiL*.
Descripción	El simulador deberá conectarse al *framework HiL* que almacena la información del sistema autónomo y sus tareas.
Prioridad	Alta

Tabla 3.6: RF-05: Conectarse al framework HiL

RF-06

Nombre	Generar listado de tareas.
Descripción	A partir de la conexión con el *framework HiL*, el simulador generará una lista con todas las tareas del sistema cargado. Esta lista será compuesta con botones que hacen una llamada para iniciar dicha tarea, funcionamiento contenido en el *RF-01*.
Prioridad	Alta

Tabla 3.7: RF-06: Generar listado de tareas

RF-07

Nombre	Generar elementos del simulador.
Descripción	A partir de un archivo de configuración, el simulador leerá el contenido dónde se encuentra toda la información necesaria para la generación de todos los elementos del simulador, desde los elementos *background* hasta *feedback*.
Prioridad	Alta

Tabla 3.8: RF-07: Generar elementos del simulador

RF-08

Nombre	Notificar interacción al *framework HiL*.
Descripción	Cuando el usuario interactue con alguno de los elementos, dicha interacción será notificada al *framework HiL*.
Prioridad	Alta

Tabla 3.9: RF-08: Notificar interacción al *framework HiL*

RF-09

Nombre	Generar registro.
Descripción	El simulador registrará todas las interacciones del usuario en el simulador y generará un archivo de texto con dicha información.
Prioridad	Media

Tabla 3.10: RF-09: Generar registro

RF-10

Nombre	Conectarse a la red neuronal.
Descripción	El simulador deberá conectarse a la red neuronal.
Prioridad	Media

Tabla 3.11: RF-10: Conectarse a la red neuronal

RF-11

Nombre	Notificar interacción a la red neuronal.
Descripción	Cuando el usuario interactue con alguno de los elementos, esta interacción será notificada a la red neuronal.
Prioridad	Media

Tabla 3.12: RF-11: Notificar interacción a la red neuronal

RF-12

Nombre	Registrar *heatbeat*.
Descripción	Cada cinco segundos, el simulador registrará en el archivo de log y a la red neuronal del estado actual del simulador.
Prioridad	Media

Tabla 3.13: RF-12: Registrar heatbeat

RF-13

Nombre	Efectos de sonido.
Descripción	El simulador podrá usar efectos de sonido por defecto o insertados mediante la configuración del *framework HiL* para los elementos *feedback*, de ser necesario.
Prioridad	Baja

Tabla 3.14: RF-13: Efectos de sonido

RF-14

Nombre	*Text-to-speach*.
Descripción	El simulador podrá usar sonido *text-to-speach* para los elementos *feedback*, de ser necesario.
Prioridad	Baja

Tabla 3.15: RF-14: Text-to-speach

RF-15

Nombre	Efectos de animación en elementos.
Descripción	Los distintos elementos interactuables, de *background* o de *feedback* podrán hacer uso de animaciones previamente generadas.
Prioridad	Baja

Tabla 3.16: RF-15: Efectos de animación en elementos

3.2.2. Requisitos no funcionales

En cuanto a los requisitos no funcionales del simulador, se definen en las siguientes tablas:

RNF-01

Nombre	Compatibilidad con navegadores web.
Descripción	El simulador debe ser compatible con los navegadores web más usados, cómo Chrome, Firefox, Opera, Safari y Edge.
Prioridad	Alta

Tabla 3.17: RNF-01: Compatibilidad con navegadores web

RNF-02

Nombre	Compatibilidad con formato de audios.
Descripción	El simulador debe permitir el uso de los principales formatos de audio, cómo MP3, WAV, MIDI, OGG y ACC.
Prioridad	Media

Tabla 3.18: RNF-02: Compatibilidad con formato de audios

RNF-03

Nombre	Compatibilidad con formato de vídeos.
Descripción	El sistema debe permitir el uso de los principales formatos de vídeo, cómo AVI y MP4.
Prioridad	Media

Tabla 3.19: RNF-03: Compatibilidad con formato de vídeos

RNF-04

Nombre	Control de error de conexión.
Descripción	El simulador deberá controlar los errores en caso de fallos en la conexión con el *framework HiL*. Intentará reconectarse o detendrá la ejecución de la tarea de no recibir respuesta al pasar un periodo de tiempo.
Prioridad	Alta

Tabla 3.20: RNF-04: Control de error de conexión

3.3 Casos de uso

A continuación se detallan los casos de uso que intervienen en el simulador. La tabla que los define es la siguiente:

Identificador	
Nombre	
Descripción	
Actor	
Precondición	
Flujo	
Postcondición	
Requisitos asociados	

Tabla 3.21: Tabla de ejemplo de los casos de uso

El significado de los campos es el siguiente:

- *Identificador*: Código identificativo único de cada caso de uso.
- *Nombre*: Nombre descriptivo del caso de uso.
- *Descripción*: Definición simple y concisa de lo que especifica el caso de uso.
- *Actor*: Rol que realiza el caso de uso.
- *Precondición*: Condicion requerida para poder realizar la funcionalidad especificada.
- *Flujo*: Funcionalidad del caso de uso paso por paso.
- *Postcondición*: Estado del sistema tras ejecutar el caso de uso.
- *Requisitos asociados*: Requisitos funcionales asociados al caso de uso.

Debido a que el sistema solo dispone de un actor, el usuario, todos los casos de uso serán para el mismo rol. En la *Figura 3.2* están definidos los casos de uso que el usuario podrá realizar en la simulación.

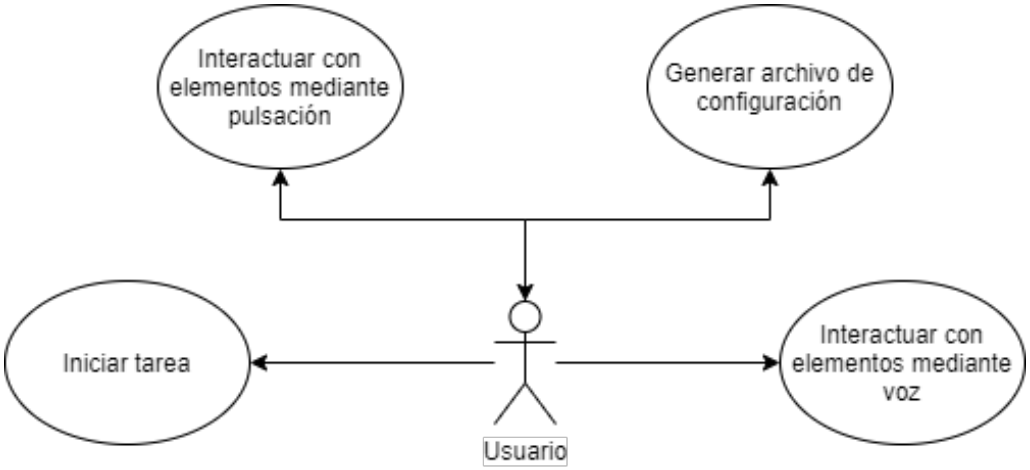

Figura 3.2: Casos de uso del simulador

Estas funcionalidades consisten en iniciar tareas del sistema autónomo y las distintas formas de interactuación con la simulación. Se ha buscado una funcionalidad simple para evitar que el tiempo de aprendizaje sea elevado.

Los casos de uso son los siguientes:

CU01

Nombre	Iniciar tarea.
Descripción	El usuario podrá iniciar una tarea del sistema autónomo.
Actor	Usuario.
Precondición	Ninguna.
Flujo	1. El usuario quiere iniciar una tarea colaborativa. 2. El usuario elije una tarea colaborativa de la lista de tareas disponibles. 3. El sistema detiene cualquier tarea que se estuviera ejecutando, volviendo a un estado ocioso. 4. El sistema inicia la simulación de la tarea.
Postcondición	Tras unos segundos de *delay*, inicia la simulación de la tarea.
Requisitos asociados	RF-01, RF.05, RF-06, RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13, RF-14, RF-15

Tabla 3.22: CU01: Iniciar tarea

CU02

Nombre	Interactuar con elementos mediante pulsación.
Descripción	El usuario podrá interactuar con los elementos del simulador.
Actor	Usuario.
Precondición	Hay una tarea en simulación.
Flujo	1. El usuario quiere interactuar con el simulador. 2. El usuario utiliza el ratón o las teclas para interactuar con la simulación. 3. El sistema registra la interacción y notifica al *framework* y a la red neuronal de ello.
Postcondición	La interacción es notificada al *framework* y registrada en los *logs* y la red neuronal.
Requisitos asociados	RF-02, RF-03

Tabla 3.23: CU02: Interactuar con elementos mediante pulsación

CU03

Nombre	Interactuar con elementos mediante voz.
Descripción	El usuario podrá interactuar con la simulación mediante comandos de voz.
Actor	Usuario.
Precondición	Hay una tarea en simulación.
Flujo	1. El usuario quiere interactuar con el simulador mediante voz. 2. El usuario utiliza un micrófono para introducir instrucciones mediante voz para interactuar con la simulación. 3. El sistema registra la interacción y notifica a la red neuronal y al *framework*.
Postcondición	La interacción es notificada al *framework* y registrada en los *logs* y la red neuronal.
Requisitos asociados	RF-04

Tabla 3.24: CU03: Interactuar con elementos mediante voz

CU04

Nombre	Generar archivo de configuración.
Descripción	El usuario podrá generar un archivo de configuración y cargarlo en el simulador.
Actor	Usuario.
Precondición	Ninguna.
Flujo	1. El usuario quiere modificar o crear una configuración para el simulador. 2. En la carpeta dónde se encuentra el ejecutable del simulador, el usuario puede modificar el archivo *conf.txt*. 3. El simulador deberá reiniciarse para que los cambios tengan efecto.
Postcondición	Una vez reiniciado, el simulador deberá mostrar los cambios realizados en su configuración.
Requisitos asociados	RF-07

Tabla 3.25: CU04: Generar archivo de configuración

CAPÍTULO 4
Solución propuesta

En este capítulo se detallará el diseño, arquitectura y tecnología de la solución propuesta. En primer lugar se hablará de la arquitectura del sistema propuesto, definiendo cada uno de los subsistemas que lo componen.

Posteriormente, se explicará en detalle el diseño del simulador. En este apartado se detallará la tecnología utilizada, así cómo los componentes y implementaciones realizadas para garantizar que se cumplan todos los requisitos propuestos. Adicionalmente, se detallarán las iteraciones realizadas y la evolución que ha tenido en cada etapa.

4.1 Arquitectura del sistema

Este simulador imita el comportamiento de un sistema HiL, permitiendo a los usuarios interactuar en este sistema. La implementacióndel sistema HiL se consigue mediante el *framework HiL*, definido en detalle en el siguiente apartado, desarrollado por el grupo *TaTAmI* del centro *PROS* en el *DSIC*.

Además, el simulador es capaz de conectarse a una red neuronal auxiliar capaz de analizar las interacciones de los usuarios con la simulación y proponer mejoras a las tareas que sean simuladas.

Por lo que nuestro sistema está compuesto por tres piezas fundamentales: el *framework HiL*, el simulador HiL y la red neuronal.

4.1.1. Framework HiL

Como se ha explicado en la introducción de la sección, el simulador necesita comunicarse con un *framework* específico para poder funcionar. Este *framework* utiliza una arquitectura *OSGI*, o *Open Services Gateway Initiative*, la cual fue necesaria adaptarla para que pudiera transmitir los mensajes aceptados por el simulador.

OSGI es una arquitectura Java que sirve para crear un entorno de *software* para gestionar el ciclo de vida. Uno de los conceptos clave de Java *OSGI* son los *bundle*, un componente parecido a un *jar* tradicional de Java, que contiene interfaces y un archivo de tipo *MANIFEST*. Estos *bundles* permiten que nuevas funcionalida-

des sean añadidas sin necesidad de conocer las demás y que los componentes se relacionen de forma idependientes. Gracias a que los *bundles* pueden quitarse y ponerse en caliente, permite que varias versiones de un componente funcionen a la vez y que se usen según su necesidad.

El *framework HiL* facilita la implementación de tareas *HiL* y para esto contiene una parte lógica que debe comunicarse con una parte gráfica. La parte gráfica del *framework HiL* se refiere al simulador descrito en este proyecto. Mientras que la parte lógica contiene una instancia del *framework HiL*. Esta instancia se crea de forma automática a partir de la especificación de las tareas, que están especificadas en *JSON*.

La parte lógica es la encargada de comunicarse con la parte gráfica y notificar de que debe representarse en el simulador, indicando que elementos se encuentran activos y deteniendo la ejecución si el tiempo dedicado a la tarea finaliza.

La comunicación entre ambos sistemas se realiza mediante *MQTT*, o *Message Queuing Telemetry Transport*, un protocolo de red bastado en publicación y suscripción, o pub-sub, utilizado generalmente para el intercambio de mensajes entre distintos dispositivos. A diferencia de *HTTP*, *MQTT* mantiene abiertas las conexiones y las reutiliza.

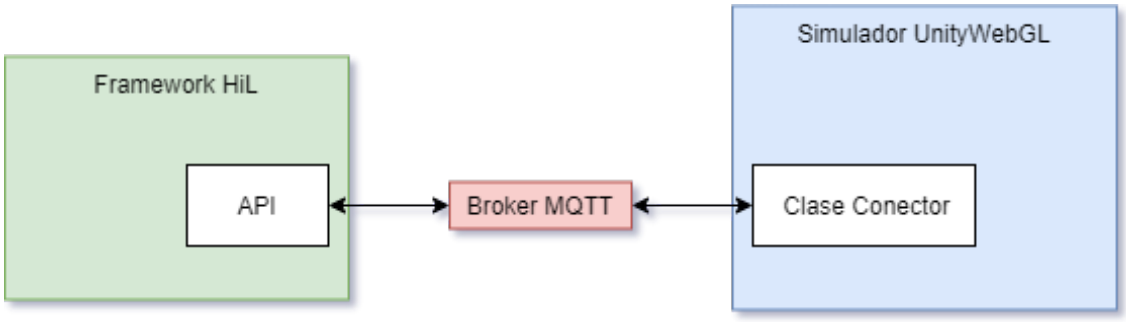

Figura 4.1: Diagrama de conexión entre el framework y el simulador

Como puede verse en la *Figura 4.1*, el protocolo requiere de un *broker* entre ambos sistemas para poder filtrar los mensajes. Mediante el uso de *topics* organizados jerárquicamente, a los cuales los clientes pueden subscribirse, los mensajes pueden ser filtrados para notificar un componente concreto en vez de que todos los clientes sean notificados. Los mensajes *MQTT* consisten simplemente en un *topic* y en un cuerpo del mensaje.

La información relevante del *framework* en el archivo de configuración es la siguiente:

	Descripción	Ejemplo
Dirección del servidor	Dirección y puerto dónde el *framework* se ejecuta.	192.168.1.1:800
Listado de tareas	Lista con el mensaje que debe recibir el *framework* para iniciar una tarea.	*TakeOver*
***Topic* de tareas**	*Topic* al que hay que notificarle al *framework* para lanzar una tarea.	*frameworkhil/tasks/*

Tabla 4.1: Información del framework

Es importante destacar que las tareas introducidas correspondan con el nombre de la tarea del *framework*. La parte correspondiente en el archivo *JSON* sería la siguiente:

```
{
  "framework_URL": "localhost:800",
  "tasks": [
    "TakeOver",
    "AutoPilot"
  ],
  "task_topic": "frameworkhil/tasks/"
}
```

4.1.2. Simulador HiL

El simulador tiene como objetivo principal servir a los expertos del dominio y diseñadores de la interacción como una herramienta que les ayudará a observar y probar el comportamiento en ejecución de las tareas HiL diseñadas. Esto permitirá iterar y mejorar su diseño antes de su implementación en un sistema real.

Adicionalmente, el simulador permitirá recoger datos de uso que podrán usarse junto a una red neuronal para la mejora del sistema simulado.

Este simulador permitirá representar y modificar de forma sencilla el contexto del humano y el sistema, además de simular gráficamente las acciones del sistema. Esto puede depender del sistema simulado en concreto o el dominio del mismo. Desde un robot de carga que se mueve, un coche que calcula una ruta o incluso una máquina industrial.

A parte de las acciones del sistema, el simulador podrá simular gráficamente las acciones del humano dentro del dominio. Permitiría ver cómo un conductor sujeta el volante del coche, cargar un robot de carga o realizando un mantenimiento de una máquina industrial.

En la sección siguiente del diseño del simulador, se entrará en detalle de todas las opciones que el simulador tiene disponible.

4.1.3. Red neuronal

Para la autoconfiguración del simulador y su automejora, se implementa la capacidad de comunicarse con una red neuronal externa a la cual el sistema irá notificando periódicamente con el fin de nutrir de información la red y que pueda optimizarse y intentar mejorar la simulación y la satisfacción del usuario.

Al igual que con el *framework HiL*, el sistema se comunica mediante el protocolo *MQTT*, utilizando el *topic* $ia/input$. Similar al *log*, un ejemplo del cuerpo del mensaje *MQTT* que se envía a la red neuronal sería el siguiente:

```
1 {"Action":"takecontrol","TimeStamp":"06:59:24.2922685","
    Context":{"HandsOnWheel":true,"Seat":"driverSeat","
    AttentionLevel":"high","RadioVolume":"low"}}
2 {"Action":"confirmcontrol","TimeStamp":"06:59:26.2794196"
    ,"Context":{"HandsOnWheel":true,"Seat":"driverSeat","
    AttentionLevel":"high","RadioVolume":"low"}}
```


A partir de esos datos, el simulador siempre deja una conexión abierta y escuchando el *topic ia/output* para autoconfigurarse con la información que se le proporcione. Cuando un mensaje desde ese *topic* llega al simulador, este es capaz de modificar los valores de los eventos de entrada y salida de los objetos *Interactuables* y *Feedback*, permitiendo así cambiar que sistemas de interacción pueden ser activados en la simulación.

Con la adición de la red neuronal al sistema, el diagrama de conexiones quedaría de la siguiente forma:

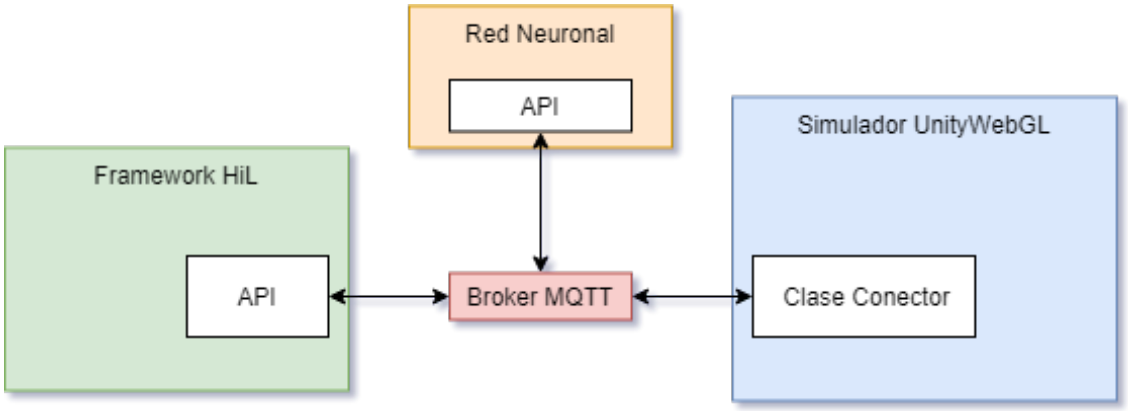

Figura 4.2: Diagrama de conexión entre el framework, el simulador y la red neuronal

4.2 Diseño del simulador web

El simulador está compuesto por una serie de elementos configurables para poder personalizar el simulador a representaciones del distintos sistemas o para cambiar la interacción de estos. También puede configurarse en caliente mediante instrucciones recibidas desde una red neuronal externa.

Para que el simulador funcione se debe cargar un archivo de configuración que contiene toda la configuración de los elementos que intervienen en la simulación, se ha añadido un ejemplo de este en los apéndices. En las siguientes secciones se entrará más en detalle en cada uno de los componentes del sistema y qué parte de la configuración corresponde cada uno.

4.2.1. Tecnología utilizada

Para la realización de este proyecto, se ha decidido utilizar el motor gráfico *Unity3D* ya que es muy versátil y dispone de muchas funcionalidades avanzadas para dar libertad a la hora de crear contenido.

Este motor gráfico es usado principalmente para el desarrollo de videojuegos, especialmente en aquellos de menor o presupuesto medio. Esto se debe a que *Unity3D* es de uso gratuito aceptando que la marca de agua del motor aparezca al iniciar la aplicación.

Si se desea optar por modificar las marcas del arranque, por defecto aparecen el logo de *Unity* y otro logo que tu decidas de forma estática, *Unity3D* dispone de dos versiones de pago, las cuales te proporcionan mejor asistencia técnica, y en el caso de *Unity3D pro* la capacidad de sugerir cambios personales del motor para suplir carencias de este.

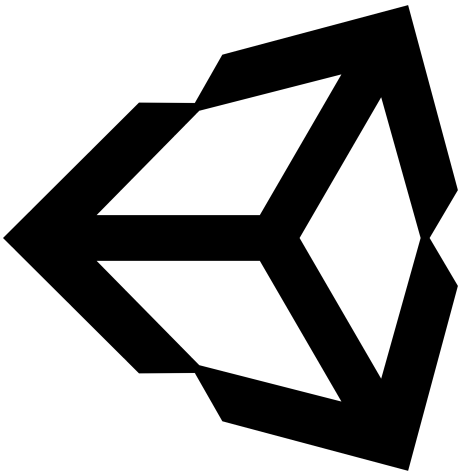

Figura 4.3: Logo de Unity

Unity3D fue desarrollado en 2005 por *Unity Technologies* y se ha posicionado como uno de los motores gráficos más utilizados de los últimos años, teniendo por delante únicamente a *Unreal Engine*.

Actualmente se encuentra en la versión *2020.1 TECH* la cual permite el uso de las nuevas herramientas desarrolladas para coexistir con el motor y que añaden un sin fin de funcionalidades adicionales, para separarse de los videojuegos y ofrecer oportunidades a cualquier tipo de desarrollo que necesite una representación gráfica, sea *2D*, *3D*, realidad virtual o realidad aumentada [1].

Una de las características más importantes de *Unity3D* es la capacidad de cambiar su *platform*. Una *platform* es una pseudo-distribución de *Unity3D* habilidatada para compilar para un sistema operativo o tecnología en concreto. Estas *platforms* pueden almacenar datos de configuración de forma idependiente o utilizar la genérica de *Windows*. Esto permite que por ejemplo, si estás usando una *platform* de *Unity Android OpenJDK* bajar la resolución de las texturas únicamente para esa *platform* sin la necesidad de crear un proyecto nuevo.

La última versión de *Unity3D* dispone de las plataformas *Windows, GNULinux* y *OS X* de forma nativa, pudiendo alternar de *platform* en cualquier momento. Las siguientes *platforms* están disponibles desde la gestión de módulos durante la instalación del motor o desde la opción de actualizar:

- *SteamOS*
- *Android*

  Android SDK & SNK

[1] `https://unity.com/es/products`

OpenJDK

- *iOS*
- *Windows Phone*
- *Samsung Smart TV*
- *WebGL*
- *LuminOS*

Mientras que estos son las *platforms* gratuitas, *Unity3D* dispone de de muchas otras, las cuales requieren solicitarlas con una licencia de desarrollo para estas. Por lo general, las *platforms* de pago son únicamente para consolas de videojuegos.

En cuanto a herramientas externas, *Unity* es compatible con la mayoría de los formatos de arte, sonido y vídeo más utilizados de la actualidad, por lo que no pone obstáculos a la hora de utilizar la herramienta externa que se desee para trabajar en estos apartados.

En cuanto a lenguajes, *Unity* es compatible tanto con JavaScript como con `C#`, y ofrece plugins de compatibilidad con los *IDE MonoDevelop* y *Microsoft Visual Studio*. A pesar de esto, existe la posibilidad de utilizar cualquier otro *IDE* del mercado abriendo la carpeta de *scripts* dónde se alojan todos los archivos de código del motor.

En cuanto a la *platform* usada para el desarrollo del simulador, se ha utilizado *UnityWebGL* cómo se ha indicado repetidas veces en la memoria. A parte de la capacidad de compilar el simulador para funcionar con esta especificación, ha permitido modificar la codificación y resolución de las texturas para conseguir que estas ocuparan el menor espacio en disco posible.

Respecto a la tecnología de mensajes, se ha optado por utilizar *MQTT* cómo se explicó en el apartado del diseño de la solución.

Figura 4.4: Logo de MQTT

Esto se debe a que *MQTT* permite crear sistemas escalables, es asíncrono y desacopla los clientes, permitiendo reestructurar el *framework HiL* o directamente cambiarlo por otro, que el simulador seguirá funcionando de la misma forma.

Una ventaja de esta tecnología es que es muy sencilla y ligera en cuanto a coste de recursos, permitiendo un intercambio de información rápido entre los componentes del sistema completo, además de no causar problemas de rendimiento.

Este protocolo requiere un ancho de banda mínimo, al estar simplemente intercambiando documentos de texto de poca información lo cual hace que no se requiera de una conexión a internet alta, de querer alojar el *framework*, el simulador y la red neuronal en dispositivos distintos.

4.2.2. Capas del simulador

A continuación se describirán por separado las distintas capas del simulador. Estas capas, ordenadas de delante a atrás son *Feedback*, *Interactuable*, *Background*, *vídeo* y variables de contexto. Estas capas disponen de distintas opciones y pueden incluirse tantas cómo se quiera, a excepción de la capa de vídeo, ya que por motivos de seguridad de *UnityWebGL*, sólo se permite uno.

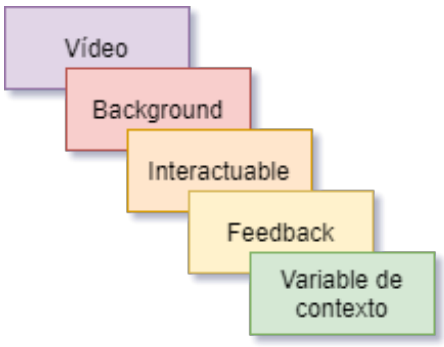

Figura 4.5: Orden de las capas de objetos del simulador

Las capas de objetos es un mecanismo utilizado por todas las versiones de *Unity3D*, incluido *UnityWebGL*, para definir en que áreas se pueden renderizar los objetos. Esto sirve para que cuando dos o varios objetos ocupen el mismo espacio, el motor sepa cuales tienen prioridad y cuales deben quedarse en un plano más atrás.

En las siguientes secciones, al explicar los distintos elementos que componen la simulación, estos contienen atributos de información llamados *offset*, *size* y *layer*, estos atributos son utilizados por *UnityWebGL* para gestionar de que forma se muestran los elementos en la interfaz.

En el caso del *offset*, que es la posición en el espacio dentro del simulador, hay que destacar que el motor sitúa el punto por defecto $x = 0, y = 0$ en el centro de la ventana. Por lo que si deseas situar un elemento a la izquierda, el valor de la x deberá ser negativo. Además que cada valor del *offset* representa un píxel.

En cuanto al *size*, representa el tamaño del elemento en la simulación, este valor se encuentra en porcentaje y no puede deformar una figura rompiendo la proporción *x*,*y* que disponga.

Por último, con el atributo *layer*, llamado así el orden de capas que utiliza *UnityWebGL*, se utiliza para cuando hay más de un elemento en el mismo lugar, siendo 250 lo más atrás y 0 lo más delante. Es importante no confundir el atributo *layer* con el orden de capas de los elementos. Por ejemplo, un elemento *Interactuable* de *layer* 0 siempre estará detrás de un elemento *Feedback* aunque este tenga un valor *layer* de 250.

Vídeo

Figura 4.6: Capa de vídeo

Esta es la capa que va más al fondo a la hora de representarlo gráficamente. Cómo se ha dicho anteriormente, el simulador funciona sobre *UnityWebGL* y este es muy restrictivo con el peso y resolución de los vídeos. Para evitar problemas de peso con el simulador, el archivo de vídeo debe proporcionarse a través de un enlace *url*.

	Descripción	**Ejemplo**
Dirección del vídeo	Enlace *url* a la localización del vídeo.	$http://host/video.avi$
Offset	Posición del vídeo en el espacio del simulador.	$x = 150, y = -50$
Size	Tamaño del vídeo.	0.5

Tabla 4.2: Datos de la capa Vídeo

En cuanto a la parte del archivo, un ejemplo sería el siguiente:

```
{
    "video":{
        "url": "http:/host/video.avi",
        "offset": {
            "x": 150,
            "y": -50
        },
        "size": 0.5
    }
}
```

Background

Figura 4.7: Capa de *background*

En cuanto a la capa de *Background*, esta se encuentra entre el *Vídeo* y los *Interactuables*. Consiste en una imagen estática en la pantalla. Esta capa está pensada, y es recomendada, para que exista una única imagen, por lo que los atributos *offset*, *size* y *layer* son opcionales.

De la misma forma cómo se verá en las siguientes capas, *Background* dispone de un repertorio de imágenes por defecto, de poco peso al estar comprimidas en el simulador, que pueden usarse a libre disposición.

Los datos relevantes a la hora de crear objetos de esta capa son los siguientes:

	Descripción	**Ejemplo**
***ID* del objeto**	*ID* de referencia del objeto, de dejar el campo en blanco, el motor asignará uno por defecto.	bgr001
Selector de *Background*	Valor numérico de la imagen a usar. El valor 0 significa que se va a utilizar una imagen propia.	
Dirección del *Background*	Enlace *url* a la localización de la imagen de *Background*.	*http* : *//host/bgr.png*

Tabla 4.3: Datos de la capa *Background*

Cómo se puede comprobar en la tabla, el campo del selector se pide un número para seleccionar una imagen, esto se debe a que dentro del simulador, existe un array de *prefabs* ya instanciados por el motor, siendo el valor 0 un *prefab* especial que carga la imagen desde una *url* y en caso de no especificar una url usaríamos un *Background* transparente.

De utilizar un *Background* por defecto, el campo *url* no es necesario incluirlo. A continuación se muestran dos ejemplos de configuración mostrando las dos opciones.

```
{
    "bgr001":{
        "id": "bgr001",
        "selector": 0,
        "url": "http:/host/bgr.png"
    },
    "bgr002":{
        "id": "bgr001",
        "selector": 1
    }
}
```

Puesto que los objetos del simulador se cargan durante el arranque del sistema, no es posible visualizar las distintas imágenes almacenadas. Además, si se utiliza en el selector un índice a una posición del array inexistente, este devolverá un *null* y no instanciará dicho objeto, advirtiendo al usuario por consola y continuando la ejecución sin detenerse.

Interactuable

Figura 4.8: Capa de *interactuables*

Respecto a la capa de *Interactuables*, es junto a la de *Feedback*, las capas más personalizables y complejas a la hora de especificar en el archivo. Estas capas disponen de distintos atributos opcionales, así cómo modos de comportamiento para intentar poder representar cualquier elemento que se desee.

Al igual que con los *Backgrounds*, la capa de *Interactuable* también dispone de un array selector de imágenes que funciona de la misma manera, utilizando el valor 0 si se quiere utilizar una imagen propia.

A continuación se muestra una tabla con los datos usados por los *Interactuables* exceptuando la información de los modos de comportamiento, que se explicarán más adelante.

	Descripción	**Ejemplo**
***ID* del objeto**	*ID* de referencia del objeto, de dejar el campo en blanco, el motor asignará uno por defecto. El ID se utilizará para que otros objetos puedan referenciarse con este.	int001
Selector de *Interactuable*	Valor numérico de la imagen a usar. El valor 0 significa que se va a utilizar una imagen propia.	
Dirección del *Interactuable*	Enlace *url* a la localización de la imagen del *Interactuable*.	$http://host/int1.png$
Offset	Posición del *Interactuable* en el espacio del simulador.	$x = 0, y = 100$
Size	Tamaño del *Interactuable*.	1
Layer	Posición del *Interactuable* en la jerarquía de dicha clase.	5
Comportamiento	Forma en la que se interacciona con el objeto, más adelante se explica su funcionamiento.	
Evento de entrada	Indica que evento escucha el *Interactuable* para activarse, en caso de no especificar este campo se toma por defecto el inicio de la tarea.	HiLAction1
Evento de salida	Indica que evento escucha el *Interactuable* para desactivarse, en caso de no especificarlo se toma por defecto el final de la tarea.	HiLAction2

Tabla 4.4: Datos de la capa *Interactuable*

Cómo puede verse en la tabla, la capa *Interactuable* dispone de dos campos llamados *Evento de entrada* y *Evento de salida* que hacen referencia a la escucha de eventos. Estos eventos son los enviados por el *framework* para avisar al simulador

por que etapa se encuentra. Un ejemplo de uso de esto sería que durante un momento, el simulador requiriera pulsar una confirmación desde una pantalla cuyo botón solo está disponible durante un momento.

En cuanto a los modos de comportamiento, se han definido tres distintos: *switch*, *exclusive* y *button*.

El modo de comportamiento *switch* define un comportamiento que puede considerarse *true* o *false*, cómo podría ser estar sujetando una palanca o el estado de un interruptor. Es una forma simplificada del *exclusive* que sólo devuelve valores booleanos al *framework*.

A este modo se le pueden asignar dos imágenes para cuando el valor está en *true* y otra, o la misma, para cuando está en *false*. Si alguna de las dos se deja en blanco, el simulador ocultará la imagen del estado contrario. Por ejemplo, en el caso del coche autónomo, el volante dispone de una imagen de unas manos para cuando está el valor a *true* y un espacio en blanco para cuando está en *false*. Para definir una de estas imágenes se pueden utilizar objetos *Feedback* o incluso otros elementos *Interactuable*.

	Descripción	**Ejemplo**
Variable	Nombre de la variable que modifica en el *framework*.	*HandsOnWheel*
ID true	*ID* del objeto *Feedback* asociado a el estado *true*.	fdb005
ID false	*ID* del objeto *Feedback* asociado a el estado *false*.	fdb006

Tabla 4.5: Datos del comportamiento *Switch*

Respecto al modo de comportamiento *exclusive*, funciona de la misma forma que un *radio button*, dónde hay una secuencia de opciones pero solo uno puede ser *true*. En este caso, se debe especificar una variable del grupo, la cual deberá corresponder a la variable del *framework* que este grupo modifique. Además, se tendrá que especificar el valor que asignará este *Interactuable* a la variable del grupo cuando el mismo entre en el estado *true*.

Al igual que en el caso de *switch*, se le pueden asignar dos objetos *Feedback* para los estados *true* y *false*.

	Descripción	Ejemplo
Variable del grupo	Nombre de la variable que modifica en el *framework*.	*LuzSemaforo*
Valor	Valor que asignará a la variable cuando cambie al valor *true*.	*Rojo*
ID true	*ID* del objeto *Feedback* asociado a el estado *true*.	fdb007
ID false	*ID* del objeto *Feedback* asociado a el estado *false*.	fdb008

Tabla 4.6: Datos del comportamiento *Exclusive*

Por último, el comportamiento *button* consiste en un botón que envía un mensaje de vuelta al *framework*. Este tipo de comportamiento no dispone de estados *true* o *false*, por lo que no se le puede asignar una animación. En el *array* de *Interactuables* hay disponible una selección de botones de *UnityWebGL* que incluyen las animaciones por defecto del motor gráfico para este tipo de objetos.

Adicionalmente, se puede especificar, de forma opcional, un texto para el botón que aparecerá en la posición $x = 0, y = 0$ relativa a este objeto.

Los *Interactuables button* no hacen asignaciones a variables, únicamente envían un mensaje al *framework*.

	Descripción	Ejemplo
Valor	Valor que enviará al *framework* cuando sea pulsado.	*Confirm*
Texto	Cadena de texto del botón.	Pulsa para confirmar

Tabla 4.7: Datos del comportamiento *Button*

A continuación se muestra un ejemplo del archivo de configuración que contiene la definición de uno de estos objetos. Para más información sobre como definir objetos de este tipo, se puede consultar el archivo de configuración en el anexo.

```
{
    "int001":{
        "id": "int001",
        "selector": 0,
        "url": "http://host/int1.png",
        "offset": {
            "x": 150,
            "y": -50
        },
        "size": 1,
```

```
        "layer": 0,
        "exclusive": {
            "group": "LuzSemaforo",
            "value": "Rojo",
            "true": "fdb007",
            "false": "fdb008"
        },
        "in": "HiLAction1",
        "out": "HiLAction2"
    }
}
```

Feedback

Figura 4.9: Capa de *feedback*

La capa de *Feedback* es la capa más externa que se sitúa por encima de los otros elementos del simulador. Representan las interacciones del sistema con el usuario. Al igual que con el comportamiento de los objetos *Interactuable*, se ha buscado integrar el máximo tipo de elementos para poder dar *Feedback* al usuario.

Los objetos *Feedback* se dividen en cuatro tipos: *Text-to-Speech*, *SFX*, imagen y texto. Cada uno especificado dentro de la configuración cómo un objeto diferente, debido a lo distintos que son entre ellos. Cada objeto *Feedback* solo puede ser de un tipo, por lo que si se desea que se emita un sonido, se muestre un texto y este sea leído por el *Text-to-Speech*, se deberán definir tres objetos distintos.

En cuanto a su configuración, todos los objetos *Feedback* comparten una cabecera simple la cual está compuesta por el *ID* y los eventos de entrada y salida, ya que el resto de configuración es totalmente distinta dependiendo del tipo que sean. En caso de estar definiendo un objeto *Feedback* utilizado por un objeto *Interactuable*, no se deberá especificar nada en los eventos de entrada y salida, a excepción de que se quiera que el elemento sólo se ejecute durante determinados momentos de la simulación.

	Descripción	Ejemplo
***ID* del objeto**	*ID* de referencia del objeto, de dejar el campo en blanco, el motor asignará uno por defecto.	fdb001
Evento de entrada	Indica que evento escucha el *Feedback* para activarse.	HiLAction1
Evento de salida	Indica que evento escucha el *Feedback* para desactivarse.	HiLAction2

Tabla 4.8: Datos de la capa *Feedback*

Los objetos *Text-to-Speech* utilizan la API nativa del navegador para convertir una cadena de texto a voz. Estos objetos no disponen de imagen, por lo que no contienen campos de *size* o *offset*, pero pueden ser usados por los objetos *Interactuables*. Puesto que utilizan el navegador para ser leídos, depende de la API del navegador que puedan ser reproducidos varios a la vez.

	Descripción	Ejemplo
Texto	Cadena de texto a leer.	Iniciando
Reproducción	Opción que permite definir si la reproducción es única o repetitiva.	*unique / repeat*
Intervalo	En caso de usar una reproducción de repetición, hay que especificar los segundos de intervalo en *float*	0.5f

Tabla 4.9: Datos de los objetos *Feedback Text-to-Speech*

En cuanto a los *SFX*, o *Sound Special Effect*, son los objetos que emiten sonidos básicos en el sistema. A diferencia de los *Text-to-Speech*, el simulador solo dispone de un canal de *SFX*, por lo que solo uno podrá ser reproducido al momento.

	Descripción	Ejemplo
Selector de *SFX*	Al igual que con las imágenes, el simulador dispone de sonidos por defecto.	0
Dirección del *SFX*	Enlace *url* si se ha seleccionado 0 en el selector para usar un archivo de sonido propio.	$http://host/sfx.wav$
Reproducción	Opción que permite definir si la reproducción es única o repetitiva.	*unique / repeat*
Intervalo	En caso de usar una reproducción de repetición, hay que especificar los segundos del intervalo	0.2

Tabla 4.10: Datos de los objetos *Feedback SFX*

Ambos objetos relacionados con el sonido pueden ser configurados cómo únicos o de repetición, cómo puede verse en las tablas. En el caso de los únicos, estos se reproducirán cada vez que el objeto sea activado, mientras que los de repetición, se ejecutarán en un bucle en intervalos de tiempo especificados en la configuración.

Respecto a los objetos de tipo imagen, son muy similares a lo visto en los otros elementos. Son iguales que los objetos *Interactuables*, exceptuando que estos no tienen modos de comportamiento. Los datos en referencia a estos objetos son los siguientes.

	Descripción	Ejemplo
Selector de imagen *Feedback*	Valor numérico de la imagen a usar. El valor 0 significa que se va a utilizar una imagen propia.	
Dirección de la imagen *Feedback*	Enlace *url* a la localización de la imagen de la imagen *Feedback*.	$http://host/fdb1.png$
Offset	Posición del *Interactuable* en el espacio del simulador.	$x = 0, y = 100$
Size	Tamaño del *Interactuable*.	1
Layer	Posición del *Interactuable* en la jerarquía de dicha clase.	5

Tabla 4.11: Datos de los objetos *Feedback* imagen

Y por último, los objetos *Feedback* de tipo texto pueden ser definidos utilizando distintas opciones de configuración. El tamaño puede definirse de forma concreta o dejar que el motor adapte el tamaño de la fuente para que encaje con el *best fit* en la caja de texto del objeto. En cuanto a la posición, en el caso de los objetos de tipo texto tienen la posibilidad de especificar un *ID* de cualquier otro objeto para que este lo use de ancla, utilizando así el *offset* relativo al ancla y no al canvas general del simulador.

De forma especial, los objetos de tipo texto disponen de un campo de texto, o *textbox*, que es el área dónde el texto puede ser escrito. En caso de que se utilice un tamaño de fuente automática, el motor adaptará el texto al tamaño del area, si por el contrario se define una y la cadena de texto no cabe en el área, las letras que no estén dentro de la caja de texto no serán visibles. Las coordenadas del *textbox* son relativas al propio objeto, siendo $x = 0, y = 0$ el centro del objeto texto.

	Descripción	**Ejemplo**
Texto	Cadena de texto a mostrar.	
Font-size	Puede definirse cómo un número o cómo la palabra especial *auto*.	*auto*
Offset	Posición del *Interactuable* en el espacio del simulador.	$x = 0, y = 100$
Anchor	*ID* del objeto al cual está anclado el texto.	fdb001
Textbox	Área de renderizado del texto. Se dan dos puntos, cuyas coordenadas son relativas a si mismo.	$(x = -20, y = -20)(x = 20, y = 20)$

Tabla 4.12: Datos de los objetos *Feedback* texto

A continuación se presentan dos ejemplos de objetos *Feedback*, uno de *SFX* y otro de texto. Al igual que con los objetos *Interactuable*, en el apéndice se puede consultar un ejemplo de archivo de configuración.

El siguiente código corresponde a un objeto *SFX*.

```
{
    "SFX": {
        "sfx001": {
            "id": "fdb001",
            "in": "HiLAction1",
            "out": "HiLAction2",
            "selector": 1,
            "reproduction": "repeat",
            "interval": 0.2
        }
    }
}
```

Y un ejemplo de objeto texto, sería el que se encuentra a continuación.

```
{
    "Text": {
        "txt001":{
            "id": "txt001",
            "in": "HiLAction1",
            "out": "HiLAction3",
            "text": "Esta sucediendo la accion.",
            "font_size": "auto",
            "offset": { "x": 0, "y":100 },
            "anchor": "fdb001",
            "textbox":{
                "p1": { "x": -20, "y": -20 },
                "p2": { "x": 20, "y": 20 }
            }
        }
    }
}
```

Variables de contexto

Figura 4.10: Variables de contexto en el simulador

Las variables de contexto son un tipo de variables que se usan para definir aquellos datos que no son tan fáciles de representar gráficamente. Cómo por ejemplo que el usuario está atento o despistado.

Estas variables son representadas en la esquina superior derecha, indicando el nombre de la variable de contexto, su icono y su valor. Al igual que con las imágenes de los elementos del simulador, hay disponible un array de iconos por defecto.

	Descripción	Ejemplo
ID	Identificador de la variable de contexto.	ctx001
Variable	Nombre de la variable que va a modificar en el *framework*.	*enginetemp*
Nombre público	Nombre que se muestra en el cajón de variables de contexto.	Temperatura del motor
Valores	Listado de los valores que puede tener la variable de contexto.	Baja, Media, Alta
Control	Tipo de *inputs* para modificar la variable de contexto. El tipo de controles disponibles se detalla más adelante.	Horizontal

Tabla 4.13: Datos de las variables de contexto

Los controles, o *inputs*, de estas variables son limitados a día de hoy, puesto que solo se pueden usar los definidos en el sistema, sin la posibilidad de crear adicionales. A pesar de esto, se recomienda usar las menos variables de contexto posibles, pues estas no consiguen integrarse realmente en el sistema y pueden perjudicar a la hora de conseguir todo el *feedback* posible.

El listado de controles es el siguiente:

- **Horizontal**: Control con las teclas de dirección horizontales o con *A* y *D*.
- **Vertical**: Control con las teclas de dirección vertical o con *W* y *S*.
- ***MouseWheel***: Control con la rueda del ratón.
- ***AxysL***: Control con el *joystic* izquierdo de un *controller* de *XBOX*.
- ***AxysR***: Control con el *joystic* derecho de un *controller* de *XBOX*.

Un ejemplo de una variable de contexto en el archivo de configuración sería la siguiente.

```
{
    "context": {
        "ctx001":{
            "id": "cxt001",
            "var": "enginetemp",
            "name": "Temperatura del motor",
            "value": ["Baja", "Media", "Alta"],
            "input": "Horizontal"
        }
    }
}
```

4.2.3. Iteraciones

En esta sección se resume la arquitectura que ha tenido el sistema a lo largo de las tres iteraciones y cual es su estado al final de estas. En cada subsección de cada iteración, se entrará en detalle en los cambios más drásticos del sistema y se comentarán aquellos que han tenido menos impacto, de forma que pueda verse el cambio de todo el simulador a través de las iteraciones.

UnityWebGL tiene la capacidad de instanciar y controlar distintos objetos sin que estos necesiten estar conectados entre si, ni saber de la existencia de otros. Estos objetos especiales se llaman *GameObjects*, y todos pertenecen al hilo de ejecución principal de *Unity*. Los *GameObjects* son estructuras compuestas por módulos o otros *GameObjects*, los cuales pueden tener vinculados la cantidad de *scripts* que se desee, o incluso no tener ninguno.

Todo *GameObject* tiene una representación en un punto del canvas de *Unity*, cuya representación se le atribuye el nombre de *Transform*. Los atributos comentados en los elementos del simulador, cómo el *offset*, el *size* y el *layer*, son usados por este módulo *Transform* para poder representar los *GameObjects* correctamente.

La estructura y funcionamiento de los *GameObjects* permite que un patrón de diseño como el *Observer* sea muy práctico a la hora de desarrollar un sistema en el cual todos sus elementos interactuen entre si, ya que una de las dificultades a la hora de desarrollar en *Unity*, es que las referencias que tienen unos *GameObjects* con otros son sensibles a romperse. Mientras que usando un patrón de diseño *Observer*, aseguras que todos los *GameObject* reciban la información que desees transmitirles, sin necesidad de generar relaciones entre ellos.

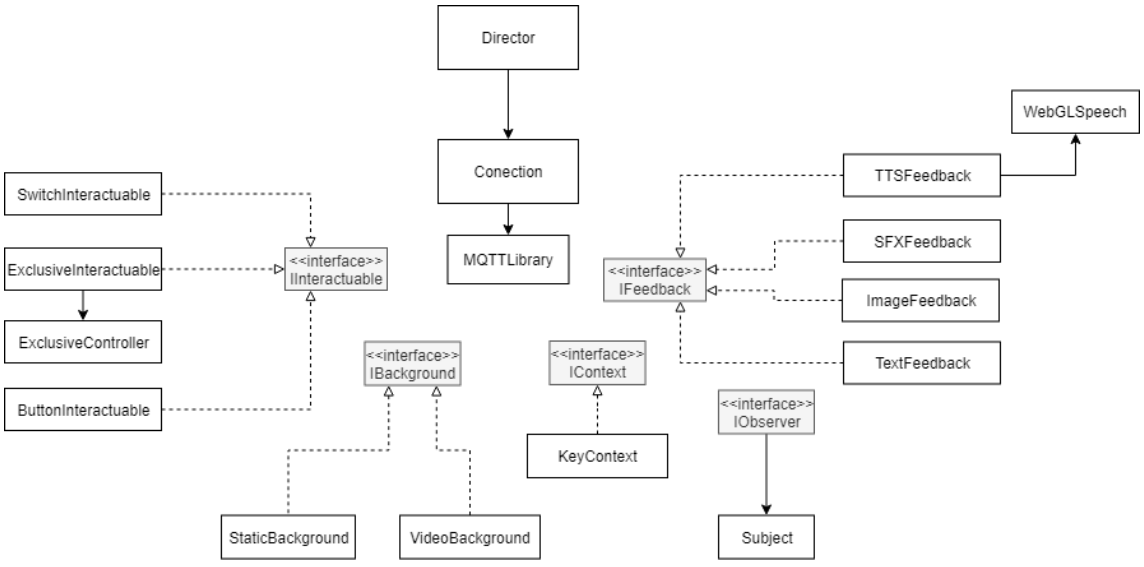

Figura 4.11: Diagrama de clases del simulador

En la *Figura 4.8* se puede ver como las distintas clases que componen el sistema no tienen relación entre ellas, a excepción de las interfaces con las clases que las implementan o el *Director* y el *Conection*. Las clases han sido desarrolladas cómo módulos de *GameObjects* para poder crear unos archivos llamados *prefab*, los cuales son modelos de *GameObjects* para poder duplicar e instanciar de forma paralela.

Los *prefabs* son *GameObjects* con los *scripts* y módulos *Unity* necesarios para que funcionen correctamente, y al instanciarlos, estos requieren que se les proporcione todos los valores de sus módulos. Dicha información se extrae de el archivo de configuración que ha sido explicado en el apartado de elementos del simulador.

Para la organización de los elementos en la simulación, se hace uso de la *Hierarchy*, la organización que utiliza *Unity* para generar y gestionar los *GameObjects* que componen el canvas.

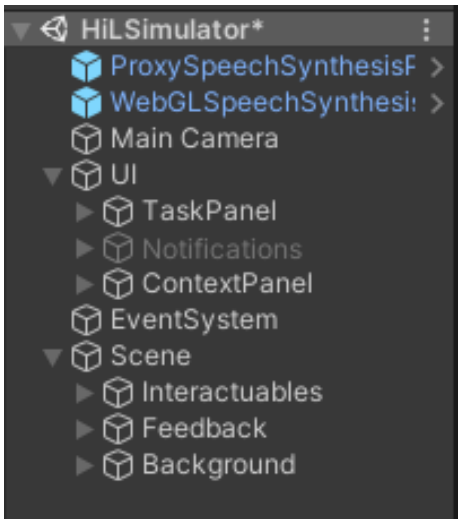

Figura 4.12: *Hierarchy* del simulador en *UnityWebGL*

Es una práctica muy utilizada, el crear *GameObjects* vacíos en la *hierarchy* para utilizarlo cómo carpetas, además de muy práctico ya que de querer eliminar y crear todos los elementos de un tipo, simplemente habría que llamar la función `DestroyAllChilds()` del *GameObject* padre.

Como puede verse en la *Figura 4.9*, los *GameObjects* padres cómo *TaskPanel*, *ContextPanel*, *Interactuables*, *Feedback* y *Background* anidarán objetos del tipo correspondiente. Mientras que *MainCamera* es el *GameObject* encargado de renderizar el canvas de *Unity* y *EventSystem* de capturar todos los *inputs* realizados. Los objetos con nombre en azul son *prefabs* externos al proyecto, en este caso, los dos *prefabs* en la *hierarchy* son los encargados de comunicarse con la *API* del navegador para capturar la voz del usuario y de llamar al *Text-to-Speech*.

Una vez detallado brevemente la forma en la que *UnityWebGL* gestiona los objetos, se verá, cómo se ha descrito en la introducción de la sección, el avance del proyecto a través de las distintas iteraciones que ha tenido.

Primera iteración

La primera iteración corresponde a una versión temprana y de prueba del caso del coche autónomo propuesto. No se pretendía que fuera una versión generalista del simulador, si no una prueba para ver si la funcionalidad era la que se buscaba.

Durante esta iteración, los elementos que componen la simulación fueron creados *hard-coded*. No se partía de ningún archivo de configuración. Dichos elemen-

tos no eran configurables en caliente, y respondían únicamente a eventos concretos del *framework*.

En esta primera iteración, fue cuando las clases *Director* y *Conection* fueron diseñadas y implementadas. Estas dos clases son la parte principal del simulador y permiten que el resto de clases implementadas tanto en esta iteración, cómo en las futuras, funcionen.

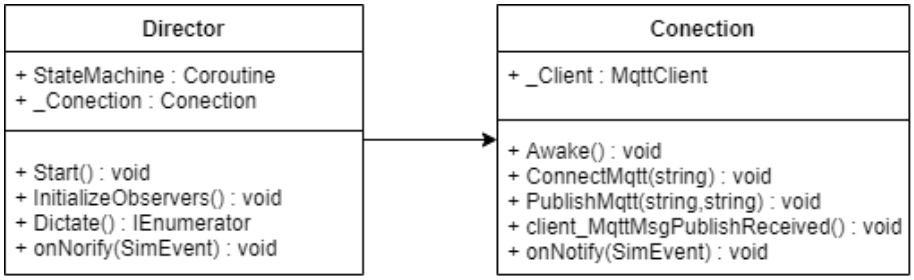

Figura 4.13: *Director* y *Conection*

La *Figura 4.10* contiene las definiciones de las clases *Director* y *Conection*. Como puede verse, *Director* únicamente contiene una máquina de estados y una referencia a *Conection*, ya que es la clase *Director* quien la inicializa.

Para la gestión de la máquina de estados del simulador se ha utilizado una *Coroutine*, un tipo de variable similar a un *thread* de Java, que es capaz de almacenar funciones *IEnumerator*.

La principal característica de las *Coroutines* es la capacidad de detener su propio hilo de ejecución sin que *Unity* se quede bloqueado esperando a que reanude. Mediante la orden `yield return`, se pueden definir pausas en el código cómo `yield return new WaitEndOfFrame()`, para que espere hasta el siguiente frame, o `yield return new WaitUntil(() => variable)`, para que la ejecución de la *Coroutine* se detenga hasta que el valor de la variable sea *true*.

La *Courutine StateMachine* tiene almacenada la función `Dictate()`, la cual contiene una máquina de estados, y sirve para escuchar ciertos tipos de eventos creados por el patrón de diseño *Observer* para ir actualizando el estado de la tarea y notificando a la red neuronal, al *framework* o a los elementos de la simulación de los cambios. *StateMachine* va alternándose entre los estados de inicialización de tareas, espera de cambios, notificar cambios y finalizar tarea.

En cuanto a la clase *Conection*, esta es la encargada de hacer de intermediario entre la librería de *MQTT* y el simulador. La clase *Director* durante el `Start()` proporciona a la clase *Conection* de los datos del *broker* y los topics a los que tiene que suscribirse.

Es importante destacar que siempre que el *framework* o la red neuronal envían algún mensaje mediante *MQTT*, la clase *Conection* generará un evento con dicho mensaje para notificar a la clase *Director*.

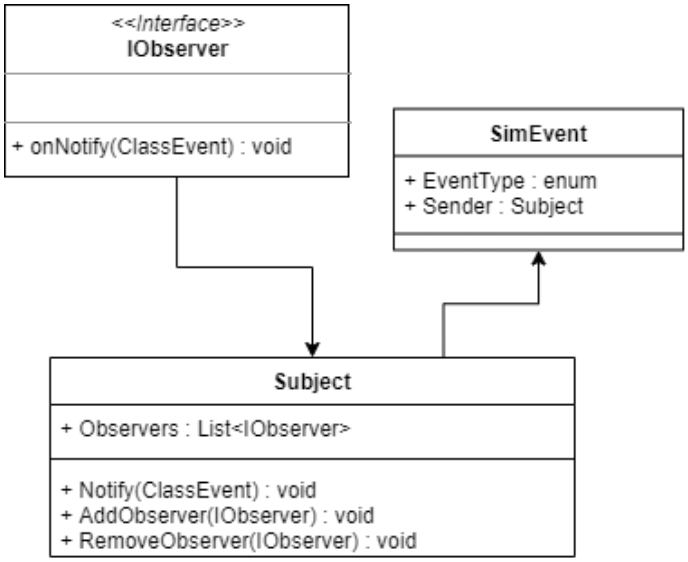

Figura 4.14: Implementación del patrón *Observer*

En cuanto al patrón *Observer*, es una implementación básica del patrón dónde a la clase de eventos se le ha llamado *SimEvent* para evitar que hiciera conflicto con la clase *Event* nativa de *Unity*.

Los objetos *Interactuables*, las variables de contexto, *Director* y *Conection* implementan y heredan tanto de *IObserver* como de *Subject*. Mientras que los objetos *Feedback* son exclusivamente observadores, debido a que su funcionalidad es reaccionar a eventos y no generarlos.

Además de las clases presentadas, se crearon otras clases que se les denominó *clases de utilidad*, las cuales son efectos gráficos o animaciones que no tienen ningún impacto en el desarrollo del simulador, por ello no han sido incluidas en el diagrama de clases de la *Figura 4.8*. Ejemplos de estos sería una clase módulo que resalta el borde de los elementos *Interactuables* al pasar el cursor por encima.

Segunda iteración

La segunda iteración requirió deshacer todos los elementos *hard-coded* para sustituirlos por estructuras instanciables y genéricas.

Para esto se crearon cuatro interfaces: *IInteractuable*, *IFeedback*, *IContext* y *IBackground*. De esta forma, la cantidad de elementos disponibles para el simulador será escalable en cualquier momento.

La interfaz *IInteractuable* y sus implementaciones eran muy importantes para el simulador, ya que son el método dónde el usuario interactua con el sistema. Cada una de las implementaciones corresponde a cada uno de los elementos *Interactuable* descritos en el diseño de la solución.

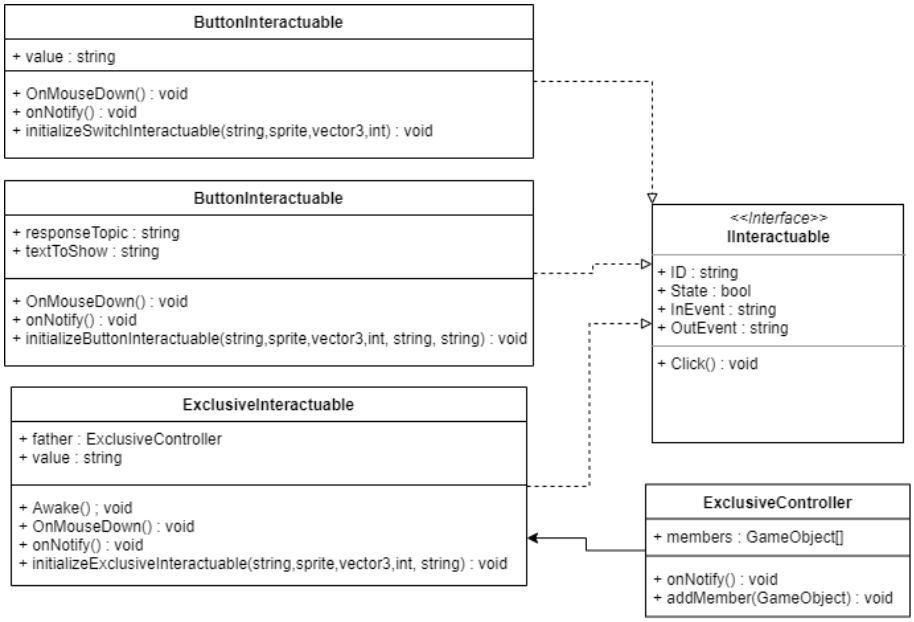

Figura 4.15: *IInteractuable* y las implementaciones

La interfaz únicamente proporciona las variables *ID*, *State*, *InEvent* y *OutEvent*, y el método `Click()`. Dicho método es utilizado dentro de la función nativa de *Unity* `OnMouseDown()` el cual es ejecutado cuando se detecta una acción del ratón encima del renderizado.

Para definir zonas interactuables de un renderizado, el *GameObject* debe disponer de uno de los muchos módulos derivados del *Collider2D*. Los distintos módulos permiten crear distintas zonas poligonales de detección de colisiones.

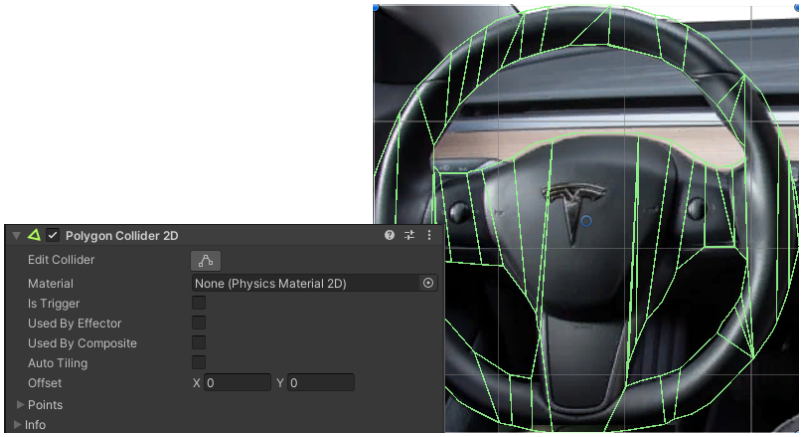

Figura 4.16: Ejemplo de *PolygonCollider2D*

Para el caso del proyecto, se ha optado por usar el módulo de *PolygonCollider2D*, ya que permite la creación automática de polígonos irregulares de forma automática. De esta forma, cualquier imagen introducida adquirirá una zona de colisión interactuable que se adaptará a su forma.

Todos los *IInteractuable* disponen de un método *initialize* que es llamado en el `onNotify()` si reciben el evento de crear una instancia de estos. El método hace una copia de si mismo y asigna todos los valores correspondientes proporcionados en el mensaje del evento. Una vez un *Interactuable* dispone de *ID*, la llamada por evento de instanciar un objeto se deshabilita.

En el caso de los *ExclusiveInteractuable* al hacer una copia, el método `Awake()` nativo de *Unity*, permite que una porción del código se ejecute durante la creación del *GameObject*. Este método busca en la *hierarchy* de *Interactuable* si existe un *GameObject* con el nombre del grupo al que pertenece. Si existe, se mueve en la *hierarchy* para convertirse en un hijo, en caso contrario, crea un *ExclusiveController* y se mueve para ser hijo suyo.

ExclusiveController contiene el listado de todos los *ExclusiveInteractuable* que pertenecen a su cargo, de forma que cuando uno de ellos hace la llamada de `Click()`, pone el *State* de todos sus hijos, menos el que ha hecho la llamada del método, a *false*. Esto puede conseguirse mediante la siguiente instrucción:

```
1 foreach(var ei in Father.gameobject.
    GetComponentsInChildren<ExclusiveInteractuable>())
2     ei.State = this.ID.equals(ei.ID);
```

Los *ButtonInteractuable* únicamente necesitan el *topic* al cual tiene que responder. El *textToShow* es simplemente un *string* que aparecerá en el centro del renderizado del botón si no es un campo vacío.

La interfaz *IFeedback* consiste únicamente en una variable *ID* y un método `Play()` que se utilizará para activar el *Feedback* cuando sea necesario. Las implementaciones de esta interfaz tienen muy pocos elementos en común, debido a que sus funcionalidades son totalmente distintas entre ellas.

En el caso de la clase *ImageFeedback*, requiere únicamente de una imagen para funcionar, además de los parámetros comunes en el resto de imágenes del simulador.

La configuración de un *TextFeedback* implica la modificación de su *textbox*. Gracias al módulo avanzado *TextMeshPro GUI*, disponible en los paquetes extras de *Unity* y sobrescribe el módulo por defecto *TextUI*, permite la modificación de muchos parámetros del texto.

El módulo avanzado de texto, además, permite que el texto introducido acepte cualquier comando de *HTML*, permitiendo etiquetas cómo `<b>...</b>`, para convertir el texto en negrita, o incluso `<color=...>...<\color>`, para dar color al texto. Por lo que gracias a esto, se le puede dar formato a la cadena introducida sin que sean necesarios parámetros adicionales al instanciar el *GameObject*.

Respecto a los *GameObjects* que contengan un módulo de la clase *SFXFeedback* utilizan el método `Start()` para guardar una referencia al módulo *SoundManager*, alojado al *GameObject* llamado *MainCamera*.

Dado que cualquier proyecto de *Unity* necesita un *GameObject* de tipo *Camera* es una buena idea alojar en este los módulos indispensables. El módulo *Camera* dispone de la directriz `DontDestroyOnLoad` haciendo que el *GameObject* que contenga un módulo con dicha directriz nunca se destruya.

SoundManager es un módulo nativo que permite reproducir archivos de audio a través de un único canal de sonido. Por lo que si se hacen dos llamadas a distintos *SFXFeedback*, sólo el que se ha ejecutado más tarde se reproducirá, ya que cuando el *SoundManager* recibe un archivo a reproducir, este expulsa al anterior.

Al contrario de la limitación de canales de sonido del *SoundManager* pueda tener. *TTSFeedback* utiliza una librería avanzada de *Unity* para acceder a la *API* del *Text-to-Speech* del navegador dónde la aplicación *UnityWebGL* se esté ejecutando, por lo que el límite de canales de audio lo determina el propio navegador.

Cómo se ha comentado en la sección de las capas del simulador, las variables de contexto sirven para representar en forma de icono y texto aquellos detalles o situaciones de la simulación que no pueden representarse gráficamente de forma sencilla.

Dado que la intención es que todo, por complicado que pueda ser, esté representado en el simulador, la intención es que este tipo de objetos caigan en desuso. A pesar de esto, se han implementado y dejado disponibles por si en un caso especifico es necesario su uso.

Se trata de la implementación más compleja de instanciar debido a que en la configuración se incluyen métodos de control de estas variables. *UnityWebGL* no puede verificar que el tipo de *input* introducido en el archivo de configuración exista en el sistema, así que si no se introduce correctamente, se le asignarán los controles *Horizontal* por defecto.

Unity no limita la cantidad de acciones que puede realizar un tipo de control, por lo que varias variables de contexto pueden compartir *input*. Esto daría la situación de que dos o más variables serían modificadas a la vez y que su control no fuera preciso.

En la creación de un objeto *KeyContext* se crea una variable de tipo *enum*, la cual es la versión del *enumerator* de `C#`, con las claves que la variable puede tener.

Para la modificación de las claves de la variable, se usa el siguiente código, el cual impide que, por ejemplo, de la clave *Baja* pase a *Alta*.

```
1 State = State == 0 ? State = KeyLevel.Next() : State.
    Prior();
```

Siendo *State* en este caso una variable de tipo *enum*, fue necesaria la creación de una extensión para permitir encontrar el valor para cuando se intenta acceder a la siguiente posición del último valor o a la posición anterior del primer valor. Siendo `Next()` y `Prior()` dos métodos muy similares, se presenta a continuación un ejemplo de uno de ellos.

```
1  public static T Next<T>(this T src) where T : struct
2      {
3          if (!typeof(T).IsEnum) throw new
               ArgumentException(String.Format("Argument {0}
               is not an Enum", typeof(T).FullName));
4          T[] Arr = (T[])Enum.GetValues(src.GetType());
5          try
6          {
7              int j = Array.IndexOf<T>(Arr, src) + 1;
8              return Arr[j];
9          }
10         catch (NullReferenceException)
11         { return Arr[Arr.Length - 1]; }
```

```
12          catch (IndexOutOfRangeException)
13          { return Arr[Arr.Length - 1]; }
14      }
```

A diferencia de el resto de clases vistas anteriormente, *KeyContext* hace uso del método nativo `Update()` de *Unity*, el cual es ejecutado en cada *frame* del renderizado.

Al contrario que con las implementaciones de *IContext*, *IBackgroud* es la más simple. Compuesto únicamente por un módulo imagen al cual *StaticBackground* y *VideoBackground* modificarán en su creación mediante el método `Start()`.

En el caso especial de *VideoBackground*, este hace uso de una de las funcionalidades avanzadas de *Unity*, la cual se encuentra en estado *Beta* para la plataforma *UnityWebGL*. Esta funcionalidad permite usar un archivo o *url* de un vídeo para usarlo como textura de un modelo *2D*.

Se tomó la decisión que para evitar que el simulador fuera muy pesado, restringir que los vídeos solo pudieran ser cargados mediante *url* y no disponer de opciones por defecto de estos.

Muchos navegadores impiden, por seguridad, que un vídeo se reproduzca automáticamente dentro de una aplicación *WebGL*, así que es posible que hasta que no se haga un click en pantalla, el vídeo no cargue correctamente.

Tercera iteración

La tercera iteración sirve para habilitar una conexión con una red neuronal y autoconfigurar la simulación con las órdenes que esta le proporcione.

Para alcanzar este objetivo de autoconfiguración, se le añadió *Conection* un nuevo *topic* al cual subscribirse, llamado *io/output*, así cómo un nuevo caso dentro del método `client_MqttMsgPublishReceived()` para que aceptara el nuevo tipo de mensaje *MQTT* y que notificara a *Director* de los cambios que tenía que recibir.

```
1  void client_MqttMsgPublishReceived(object sender,
       MqttMsgPublishEventArgs e)
2  {
3      string msg = System.Text.Encoding.UTF8.GetString(e.
           Message);
4      switch (e.Topic){
5          case "ia/output":
6              JObject newCOnfig = JObject.Parse(msg);
7              Notify(new SimEvent(this, SimEvent.EventType.
                   IAOUTPUT, JObject.Parse(msg)));
8              break;
9              //...
10     }
11 }
```

Además, en la clase *SimEvent* del patrón *Observer* se añadió al *enum EventType* un nuevo evento interno para cuando un objeto *Feedback* o *Interactuable* debiera ser modificado.

Para alcanzar la autoconfiguración de cuando se activan los elementos afectados por la modificación, las variables *InEvent, OutEvent*, y en el caso de los *Interactuables* aquella variable que almacene la respuesta que enviar al *framework*, por propiedades.

Las propiedades de `C#` permiten definir el acceso a *get* y *set* de dicha propiedad público, pero ocultando su implementación. De esta forma, a pesar de hacer una asignación usando `=`, se ejecutará el contenido de la función almacenada en el *set* de la propiedad.

Esta decisión fue tomada ya que las clases comprueban si deben activarse comparando su atributo *InEvent* al recibir un `Notify()` del *Director*. Esto puede dar el caso de que la red neuronal tarde más en notificar del cambio de *InEvent* que la notificación del *Director*, por lo que se modificó el *set* de las propiedades.

Esta modificación comprueba si la acción o evento actual ya ha sido modificado y concuerda con el nuevo valor que la red neuronal ha proporcionado. En caso negativo, simplemente se modifica la variable *InGame* para que se active cuando es debido. Mientras que en caso afirmativo, se lanza el método `SetActive(bool)` nativo de los *GameObjects*, el cual hace que se ejecuten los métodos `Awake()` y `Start()`.

Cómo se puede ver en el código siguiente, esta sería un ejemplo de cómo se ha implementado una propiedad:

```
1 String InEvent { get => InEvent; set => value.Equals(
    Director.GetInstance().Action) ? gameobject.SetActive(
    true) : InEvent = value;}
```

CAPÍTULO 5
Implantación

En este capítulo se detalla la etapa dónde el prototipo se ha llevado a la explotación.

Se describirá en este capítulo, únicamente la tercera y última de ellas, ya que es la que contiene todos los casos de uso y requisitos del sistema.

5.1 Puesta en marcha

Una vez el simulador ha sido compilado, *UnityWebGL* generará el directorio *HiLSimulator* cuyo contenido es el que se puede ver en la *Figura 5.1*.

La carpeta *Build* contiene todos los archivos necesarios para que el simulador pueda funcionar correctamente. Dichos archivos son compilados y encriptados por *Unity* por lo que no se debería de poder acceder a su información.

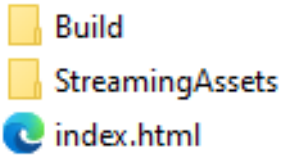

Figura 5.1: Contenido de la carpeta HiLSimulator

En cuanto a la carpeta *StreamingAssets* es usada por el motor para aquellos archivos que se desee cargar dentro de la aplicación. Cuando en el archivo de configuración se introduce el enlace de un *asset*, o elemento multimedia, se realiza una copia temporal en esta carpeta. En el caso de *UnityWebGL*, los archivos de esta carpeta se localizan a través de *Aplication.streamingAssetsPath* y son cargados a través de un *UnityWebRequest*.

El archivo *index.html* es un archivo *HTML* común que puede ser cargado desde un navegador web. Ya sea mediante la carga del archivo desde una carpeta o subiéndolo a un servidor.

La aplicación de *UnityWebGL* es puesta en marcha a través de la siguientes líneas:

```
<title>Simulador HiL Unity | WebGL Player</title>
<script src="Build/UnityLoader.js"></script>
<script>
    UnityLoader.instantiate("unityContainer", "Build/
       Builds.json");
</script>
```

El simulador no requiere que ninguno de los otros dos sistemas estén en ejecución al lanzarlo. Al iniciarse, el simulador enviará una petición *MQTT* mediante un *topic request* y esperará la respuesta del *framework HiL* para poder empezar a iniciar tareas.

A diferencia del *framework HiL*, no es necesario que la red neuronal esté en funcionamiento en cualquier punto de la ejecución, se trata de un sistema auxiliar y el simulador puede lanzar tareas y registrar interacciones a pesar de no estar la red neuronal disponible.

CAPÍTULO 6
Pruebas

En este capítulo corresponde a las distintas pruebas que se han ido realizando durante el desarrollo para validar que los objetivos se han ido cumpliendo.

Se han realizado pruebas al final de cada una de las iteraciones del desarrollo para verificar que eran válidas y sin errores.

6.1 Validación

Además de las tutoras de este proyecto, el compañero del grupo de investigación Antoni Mestre, y los antiguos compañeros del grado y máster Daniel Fenollar, Carlos Longares, Josue Navarro y Iris López, participaron como alfatesteadores, o *alpha testers*.

Los participantes se encargaron de probar el sistema tras cada iteración, en lo que se llamaron versiones *alfa*. Debido al problema de salud pública en la que se encontraba el país, las pruebas se realizaron online, a través del servidor privado del alumno.

Una versión *alfa* de una aplicación se refiere a una fase dónde el producto todavía es inestable y que puede que contenga errores o faltas de funcionalidad, pero satisface la mayoría de objetivos.

Las pruebas se realizaban en un entorno informal y no se utilizaban cuestionarios, los *testers* reportaban todos los errores y problemas directamente al autor en el cara a cara o por escrito en algún programa de mensajería instantánea. De acuerdo con Nielsen, con cinco o seis usuarios realizando pruebas, deberían encontrarse la mayoría de problemas con usabilidad[10], esto reforzó la idea de reducir los *alfa testers* a compañeros y amigos.

En general, la mayor parte de los problemas reportados estaban centrados en el apartado visual del simulador. Es importante que el usuario se sienta cómodo con todos los elementos en pantalla.

Durante las pruebas de la primera iteración, los elementos no disponían de animaciones prácticamente. Por limitaciones del sistema dónde se reproducía, y que el componente vídeo se encuentra en fase *beta*, la simulación no disponía de capa de vídeo funcional. Esto supuso que los *testers* reportaran que todo se veía muy estático.

En la siguiente iteración, con una capa de vídeo ya funcional, el resto de sistemas ya se sentían más integrados. Los componentes *Text-to-Speech* fallaban ya que al depender del navegador y la terminal que ejecutaba la aplicación era muy limitada, algunos mensajes de voz se superponían.

Este fallo fue reportado y solucionado en la siguiente fase de corrección de errores cambiando la forma en la que interactuaban las clases *TTSFeedback* con la API del navegador, sacando las llamadas *Text-to-Speech* del hilo principal de *Unity* y evitando que estas dependieran del rendimiento del terminal.

Al finalizar el desarrollo de la tercera y última iteración, la aplicación se desplegó en un servidor personal del autor. La versión desplegada podría decirse que se consideraba una versión *beta*. La aplicación puede probarse por cualquier usuario mientras el servidor esté abierto.

Durante estas pruebas, se notificaron problemas de conexión y con los navegadores. Se determinó que los problemas de conexión eran debidos a que el servidor no estaba preparado para recibir muchas peticiones, por lo que era externo a la aplicación. Mientras que los problemas con los navegadores eran debido a la seguridad de estos.

En el momento de que esta memoria es escrita y presentada, el simulador está disponible en su versión *beta* en servidor privado del autor para ser probado.

A pesar de que podría considerarse que es una versión candidata a definitiva, o *RC*, se ha decidido dejarla en versión *beta* a que se siga probando y sacando problemas de usabilidad. De esta forma aumentando la cantidad de usuarios para realizar las pruebas, y evitando que se tratara de uno de los casos descritos por Faulkner[11].

CAPÍTULO 7
Caso del coche autónomo

Este capítulo se centra en el caso del coche autónomo propuesto como ejemplo de funcionamiento del simulador. Poniendo atención a la definición del caso propuesto, los elementos elegidos para ser utilizados y un ejemplo de una tarea.

7.1 Definición del caso propuesto

El caso propuesto corresponde al de un coche autónomo. La idea principal de este es de hacer pruebas sobre las distintas tareas que un coche autónomo debería realizar y cómo el usuario se relaciona con él.

Este caso propuesto ya está implementado dentro del *framework HiL*, con sus tareas, para ser probado. El coche autónomo utiliza distintos mecanismos de interacción con el usuario, diferenciando los *outputs* y los *inputs*.

Los mecanismos de interacción de *output* corresponden a cómo el coche se comunica con el conductor, y son los siguientes:

- Mecanismos visuales
  - Texto por la pantalla del coche
  - Iconos mostrados en la pantalla del coche
  - Luces en el velocímetro, pantalla o otras localizaciones del coche
- Mecanismos auditivos
  - Sonidos básicos, como por ejemplo una alarma
  - Voz anunciando alguna petición o confirmación
- Mecanismos táctiles
  - Vibración, ya sea del volante o del asiento

En cuanto a los mecanismos de interacción *input*, estos son los cuales el conductor interactua y notifica al coche. Dichos mecanismos disponibles son los siguientes:

- Introducción de texto

- Orden por voz
- Detección de que el conductor está tocando algún elemento del coche, como por ejemplo estar sujetando el volante
- Pulsar botones
- Confirmar mediante pulsación la pantalla del coche

7.2 Elementos del coche autónomo

En base a los mecanismos de interacción descritos en la sección anterior, se tuvieron que diseñar y definir los distintos elementos que se usarían en el simulador.

Utilizando la clasificación de elementos descrita en las capas del simulador, se han definido todos los diversos mecanismos de interacción, tanto del sistema cómo del humano.

De acuerdo a la definición del caso y al diseño de capas del simulador, se definieron los siguientes elementos *Interactuables*:

- **Volante**: Se implementó cómo un *SwitchInteractuable* con un elemento *ImageFeedback* con una imagen de unas manos asociado a el, para representar que el humano estaba sujetando el volante.
- **Asientos**: Ambos asientos se configuraron cómo *ExclusiveInteractuable*, asociados a su estado *true* un *SwitchInteractuable* para el conductor. Ambos asientos simbolizan si el humano está sentado en el asiento del conductor o del copiloto.
- **Conductor**: El conductor, cómo se ha comentado en los asientos, es un *SwitchInteractuable* con dos *ImageFeedback* asociado a ambos estados *true* y *false*, representando si el humano está distraído o mirando al frente.
- **Botón del volante**: Este elemento se definió cómo un *ButtonInteractuable* en la misma posición del volante y con una *layer* superior, para aparecer delante de el. Este elemento envía un mensaje al *framework* notificando que el botón del volante ha sido pulsado.
- **Botón en pantalla del coche**: Este elemento se definió al igual que con el botón del volante, cómo un *ButtonInteractuable*, aunque no se usa para la ejecución actual del *framework*, pero está disponible por si la red neuronal considera que hay que utilizarlo. Envía un mensaje al *framework* notificando que ha sido pulsado.
- **Botón de volumen**: Este objeto, situado en la pantalla del coche, simboliza el volumen de la radio o sistema de comunicación del coche. Es un *SwitchInteractuable* con dos imágenes *ImageFeedback* para los estados silenciado o volumen alto. Este elemento antes era una variable de contexto ya que en las primeras iteraciones la pantalla del coche era demasiado pequeña. Notifica al *framework* el cambio de contexto del volumen de la radio.

- **Detección de voz**: A pesar de no tener un elemento en las capas del simulador en si, está definido en forma de prueba y solo disponible para el caso propuesto, un detector de voz utilizando la API del navegador y con un diccionario básico de órdenes. Este detector no se utiliza en ninguna de las tareas, ya que no se encuentra ninguna tarea que lo requiera, pero estando disponible para la red neuronal. Actualmente el diccionario solo permite órdenes muy básicas.

Los elementos *Interactuables* hacen referencia a todas las acciones que el humano puede realizar en la simulación. Mientras que los objetos *Feedback* hacen referencia a los mecanismos de interacción del sistema. Estos elementos son los siguientes:

- **Texto en la pantalla del coche**: La pantalla del coche tiene asociados distintos elementos de la clase *TextFeedback*, estos son usados para las distintas afirmaciones o peticiones de confirmación que el sistema desea comunicar. Cada uno de los mensajes se considera un elemento distinto.

- **Iconos en la pantalla del coche**: Al igual que en el elemento anterior, representan los iconos que se mostrarán en la pantalla del coche si el *framework* considera que no es necesario texto y que solo imágenes se deben representar. No sólo están las afirmaciones del texto de la pantalla en icono, también se encuentra aquí el icono de cuando alguno de los textos contiene un *Text-to-Speech*. Estos elementos son del tipo *ImageFeedback*.

- **Sonidos del coche**: El caso contiene distintos elementos *SFXFeedback* para cada uno de los sonidos del coche, desde alarmas, hasta sonidos de cuando una notificación necesita ser mostrada. Al igual que con la pantalla del coche, cada uno de los sonidos es un objeto distinto instanciado.

- **Voz del coche**: Algunos de los textos o iconos mostrados en la pantalla del coche pueden ir acompañados de voz, si el *framework* lo solicita. Cada uno de los textos o iconos que requieran esta función, deberán tener un objeto *TTSFeedback* adicional.

Adicionalmente a todos los elementos descritos anteriormente, el simulador dispone de un elemento *StaticBackground* representando la tapicería interna del coche, con el velocímetro, la pantalla, retrovisores, etc. Así cómo un *VideoBackground* donde se muestra un vídeo de el coche circulando por una carretera. En caso de no querer utilizar el vídeo por ahorro de recursos, se encuentra también un segundo *StaticBackground* que representa una carretera, aunque esté fondo es estático y no tiene ninguna animación.

7.3 Especificación de la tarea

Para verificar que el simulador funciona correctamente, se han utilizado las tareas colaborativas implementadas en el *framework HiL*.

Estas tareas disponen de distintos atributos y ordenes que el *framework* traduce y genera llamadas de eventos para satisfacerlas. Estos eventos son transmitidos al simulador para que sean representados.

Una de las tareas utilizadas cómo pruebas es la tarea *TakeOver* la cual simboliza que el humano toma el control del coche ante una situación de no emergencia.

La tabla que define la tarea es la siguiente:

Nombre	Transferencia del control al humano delante de una situación de no emergencia.
Descripción	El coche transfiere el control del coche al humano.
Perfil humano	Conductor
Precondición	▪ El humano está en el asiento del conductor. ▪ El humano está atento a la conducción. ▪ El humano está con las manos al volante.
Acciones preparatorias	▪ EL sistema avisa al conductor que se siente al asiento del conductor. ▪ El sistema avisa al conductor de que esté atento a la conducción. ▪ El sistema avisa al conductor de que ponga las manos al volante.
Restricciones	La tarea se debe de realizar en menos de un minuto.
Acciones	1. **Sistema**: Notificar al humano. 2. **Humano**: Confirmar aceptación de control. 3. **Sistema**: Transferir control. 4. **Sistema**: Informar del resultado de la transferencia.
Fallback plan	Reducir velocidad y aparcar en la acera.

Tabla 7.1: Tarea TakeOver

En la tabla de la especificación de la tarea colaborativa se encuentran distintos campos. El campo de nombre es simplemente el nombre identificativo de la tarea. Este nombre varía del nombre interno del *framework*, siendo el nombre interno *TakeOver*. El campo de descripción es una descripción simple de el objetivo de la tarea.

El perfil humano representa el contexto y dominio del humano en la tarea, siendo en este caso, una tarea centrada en el conductor. Otro ejemplo de perfil humano en este dominio sería el copiloto.

Las precondiciones son estados del contexto que deben cumplirse para que la tarea se inicie. Con que una de estas precondiciones no se cumpla, la tarea no iniciará.

Las acciones preparatorias es el mecanismo que tiene el sistema para alertar al perfil humano de que alguna de las precondiciones no se está cumpliendo. Solo las acciones preparatorias asociadas a las precondiciones no cumplidas son las que se ejecutarán.

Las restricciones son condiciones que deben cumplirse durante la ejecución de la tarea, si alguna de las restricciones no se cumple, la tarea fallará. En este caso, se pone un tiempo límite a la ejecución de un minuto. Las restricciones son comprobadas en todo momento desde que la tarea se lanza en el *framework*. Esto significa que las restricciones deben cumplirse aunque el sistema se encuentre en las acciones preparatorias.

Las acciones describen el flujo de eventos de la tarea. Este caso solo necesita de la interacción del humano para confirmar el control del coche, siendo este punto el que la red neuronal podría mejorar. Por lo general, el *framework* utiliza la confirmación mediante el botón del volante.

Por último, el *fallback plan* es el plan de acción del sistema cuando alguna de las restricciones no se cumple. En el caso de esta tarea, representará que el coche se detiene.

7.4 Ejecución de la tarea

La tarea a ejecutar, descrita en la sección anterior, fue implementada en el *framework HiL* y la configuración de los elementos del dominio fueron correctamente importados al simulador.

Una vez el *framework HiL* está en funcionamiento, se enciende el simulador, el cual se conecta al broker de la dirección proporcionada en el archivo de configuración.

La primera acción del simulador es hacer una petición al tópic *frameworkHiL/humanPropertiesStatus/request*, el cual hará que el *framework HiL* proporcione la información del contexto actual.

Con el estado del contexto del *framework* recibido, el simulador sitúa todos los elementos en los estados correspondientes y muestra la lista de tareas a simular.

Figura 7.1: Simulación esperando que tarea simular

En la *Figura 7.1* se puede ver que la tarea *Take Over* está lista para iniciarse. Así cómo el estado del contexto, estando el usuario distraído y en el asiento del copiloto y un volumen de radio medio.

Cuando una tarea es elegida, el *framework* comprobará que todas las precondiciones se cumplen. En el caso del contexto actual, se da la situación de que ninguna lo hace.

Figura 7.2: El simulador espera a que se cumplan las precondiciones

Para que el simulador avance a su siguiente acción, se debe cumplir que el usuario se encuentre en el asiento del conductor, que tenga las manos al volante y que esté atento. Para que el usuario se considere atento, el *framework* requiere que esté mirando al frente y la radio esté apagada.

El coche avisa al usuario mediante texto en la pantalla del coche y la lectura de este texto por voz. La red neuronal podría determinar que en vez de texto se usaran iconos, o que en vez de leer el texto por voz se usaran sonidos simples.

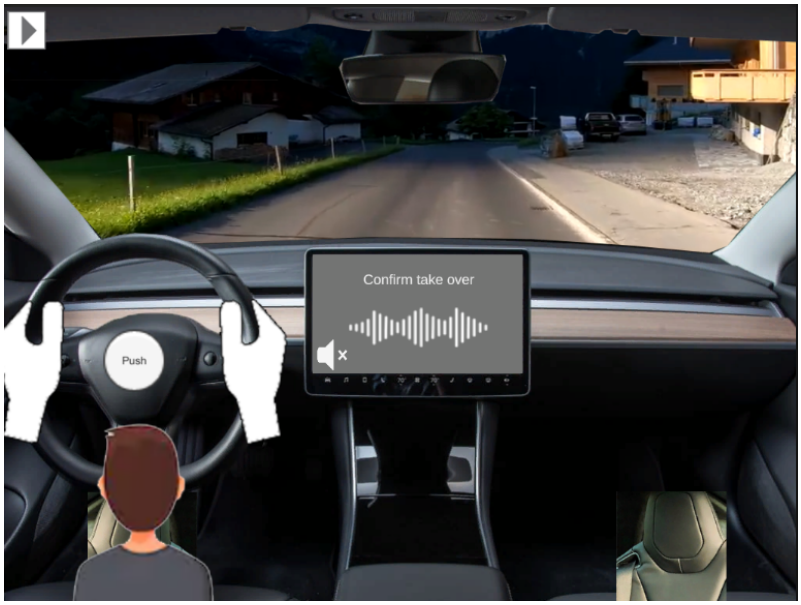

Figura 7.3: El coche espera la confirmación para dar el control

Cómo puede verse en a *Figura 7.3*, una vez que las precondiciones se cumplen, la simulación pasa a realizar las acciones. Se puede observar cómo el coche solicita una confirmación al humano para darle el control del coche.

En este punto, se ha activado el botón del centro del volante. En este punto, la red neuronal podría indicar que podría usarse un mecanismo de interacción distinto, cómo un botón en la pantalla del coche, por ejemplo.

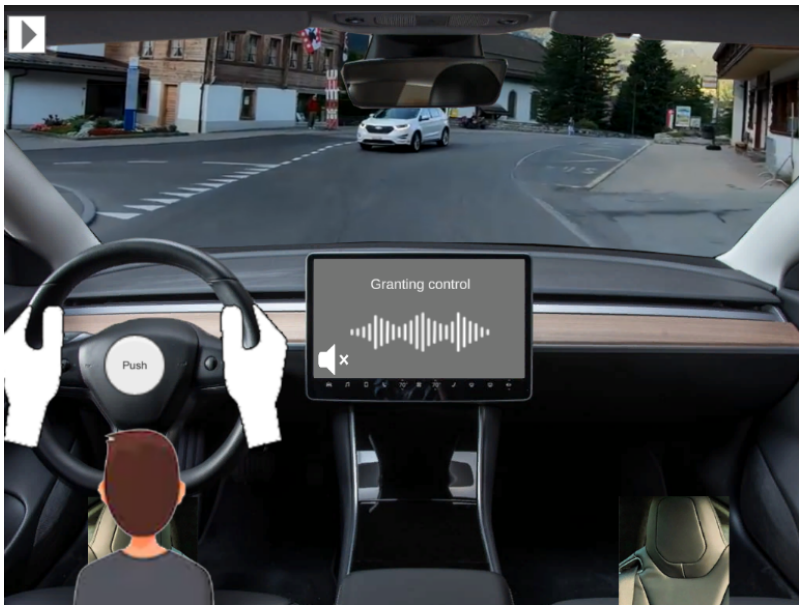

Figura 7.4: El coche anuncia que le va a dar el control al humano

Una vez confirmado que se quiere obtener el control del coche, este anunciará que ha iniciado la transferencia de control. Esta acción transiciona directamente

con otra acción por parte del sistema, cómo puede verse en la especificación de la tarea. Esta transición simplemente consiste en la espera de unos segundos.

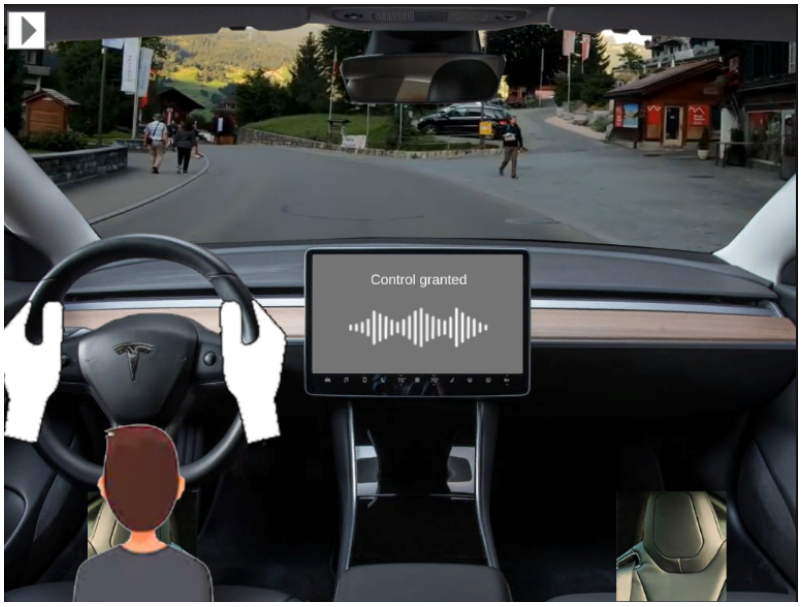

Figura 7.5: El coche notifica que ha dado el control al humano

Una vez el tiempo de transferir control ha pasado, el coche anunciará que se ha transferido correctamente. El botón del centro del volante se deshabilitará y la tarea finalizará correctamente. En este momento se generará un archivo *log* con todas las interacciones y eventos ocurridos durante la simulación, de la misma forma que se ha podido ver en el capítulo de la solución propuesta.

CAPÍTULO 8
Conclusiones

Este capítulo detalla las conclusiones a las que se ha llegado tras la realización del proyecto, así como la relación con algunas de las asignaturas y los trabajos futuros posibles.

La realización de este proyecto ha reforzado los conocimientos de *Unity* al utilizar un motor comúnmente utilizado en videojuegos para la realización de un simulador auxiliar de un sistema existente. Con esto también se pretendía demostrar la utilidad de dicha herramienta fuera del dominio al que se supone que está dedicada.

Al enfrentarse al desarrollo de un sistema tan general, se tuvo que abrir la mente y diseñar una arquitectura que pudiera simular la mayoría de las situaciones. Que los elementos que componen el sistema fueran capaces de interconectar entre ellos y que este fuera configurable en caliente.

El hecho de que el simulador fuera para cualquier tipo de dominio, también supuso el principal contratiempo. Se tuvieron que tener en cuenta muchos factores que iban apareciendo sobre el tiempo. Cómo cuantos tipos de objetos con los que interactuar habían.

Uno de los grandes inconvenientes a la hora de desarrollar todo el simulador, fue que al ser todo objetos instanciados a partir de un archivo y que además estos podían activarse y desactivarse, las relaciones entre los *GameObjects* se rompían con facilidad, por lo que fue necesario optar por patrones de diseño específicos para intentar prevenir este tipo de incidentes.

El resultado obtenido al final de esta tercera iteración muestra un simulador funcional capaz de representar muchas situaciones de los sistemas autónomos, pero cómo todo sistema, está lejos de ser perfecto. Nuevos tipos de mecanismos de interacción, tanto humanos como del sistema, serán necesarios en un futuro para cubrir todas las situaciones.

A la vez, el archivo *JSON* de configuración también presenta complicaciones a la hora de definir el sistema, ya que no puedes ver en tiempo real dónde se ubican estos elementos. Y si el formato del archivo falla, nada se verá representado en el simulador.

A pesar de esto, se considera que los requisitos y casos de uso se han cumplido y el simulador permite que los sistemas puedan ser probados por expertos y

diseñadores, y la conexión con la red neuronal les permitirá mejorar los sistemas gracias a los datos extraídos de la simulación.

Finalmente, los plazos fueron demorados debido a la situación de salud pública en la que se encuentra el país, pero a pesar de esto, el proyecto finalizó su tercera iteración y pudo ser probado desde un servidor web privado satisfactoriamente.

8.1 Relación del trabajo desarrollado con los estudios cursados

El proyecto realizado hace uso de teoría y prácticas que han sido impartidas por las asignaturas cursadas durante el máster. Las asignaturas que tienen especial relación son las siguientes:

- **SUA**: Durante la asignatura se estudió en especial el funcionamiento de Java *OSGI*, tecnología con la que fue desarrollado el *framework HiL*. Esto permitió un aprendizaje rápido de cómo el *framework* funciona. Además, durante la asignatura, dos de las prácticas eran sobre el caso propuesto del coche autónomo.

- **IoT**: Esta asignatura introdujo los principios de la *IoT*. El *Human-in-the-Loop* fue parte del temario, además de que se realizaron prácticas con el protocolo *MQTT*.

- **DIM**: En esta asignatura se hizo hincapié en las interfaces de usuario y todo lo relacionado con distintos métodos de interacción, cómo el *Text-toSpeech*.

8.2 Trabajos futuros

El simulador HiL, cómo se ha explicado anteriormente, dista de estar perfecto. Existen posibles mejoras de cara siguientes iteraciones del sistema, así cómo refinamientos de cómo los elementos se instancian y se relacionan entre ellos.

El autor, de entre todas las posibles mejoras, destaca las siguientes de cara a futuros trabajos del simulador:

- Mejora de las instancias de elementos. Muchos de los elementos *Feedback* podrían refactorizar para no depender de tantos objetos concurrentemente y juntar, por ejemplo, todos los objetos *TextFeedback* bajo el mismo *GameObject*. Esto puede ser un arma de doble filo, ya que cuantos más elementos se junten en un mismo *GameObjects*, más vulnerable es a fallos.

- Mejora en la detección por voz y sus diccionarios. Una ampliación del diccionario o permitir que se puedan insertar diccionarios mediante la carga de archivos podría ser útil de cara a futuras versiones.

- Aprovechando el abanico de *platforms* que dispone *Unity3D*, se podría plantear el diseñar versiones para otros dispositivos, cómo por ejemplo móviles, y de esta forma ofrecer más opciones a los usuarios.
- Mejora del archivo de configuración. El simulador funciona asumiendo que el usuario carga en el sistema un documento bien estructurado. Haría falta una herramienta externa para la creación de este documento o un posible editor interno del simulador.
- Actualización de las funcionalidades beta del motor una vez salgan en versión *Release*. En especial, la funcionalidad de vídeo es la que más vulnerabilidades presenta y sería prioritario actualizarla una vez la actualización esté disponible.
- Creación de nuevos tipos de elementos. De la forma en la que se han estructurado las capas del simulador, se ha facilitado la creación de elementos de tipo *Interactuable* y *Feedback*, por lo que una futura incorporación de nuevos mecanismos de interacción podría ser interesante.

Bibliografía

[1] E. Rubio-Drosdov, D. Díaz-Sánchez, P. Arias-Cabarcos, F. Almenárez, A. Marín (2015) *Towards a seamless human interaction in IoT.* International Symposium on Consumer Electronics (ISCE), Madrid, 2015, pp. 1-2, doi: 10.1109/ISCE.2015.7177781

[2] W. Yang, W. Li, J. Cao, Q. Wang, Y. Duan (2018) *Industrial Internet of Things: A Swarm Coordination Framework for Human-in-the-Loop.* IEEE International Conference on Systems, Man, and Cybernetics (SMC), Miyazaki, Japan, pp. 2754-2759, doi: 10.1109/SMC.2018.00470

[3] C. Garrido-Hidalgo, D. Hortelano, L. Roda-Sanchez, T. Olivares, M. C. Ruiz, V. Lopez (2018) *IoT Heterogeneous Mesh Network Deployment for Human-in-the-Loop Challenges Towards a Social and Sustainable Industry 4.0.* IEEE Access, vol. 6, pp. 28417-28437, doi: 10.1109/ACCESS.2018.2836677

[4] J. Cámara, G. Moreno, D. Garlan (2015) *Reasoning about Human Participation in Self-Adaptive Systems.* IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Florence, 2015, pp. 146-156, doi: 10.1109/SEAMS.2015.14

[5] Evers, C., Kniewel, R., Geihs, K., Schmidt, L. (2014) *The user in the loop: Enabling user participation for self-adaptive applications.* Future Generation Computer Systems, vol. 34, no. 0, pp. 110–123

[6] A. J. Grono (2001) *Synchronizing generators with HITL simulation.* IEEE Computer Applications in Power, vol. 14, no. 4, pp. 43-46, doi: 10.1109/67.954527

[7] W. Lei, Y. Yalian, P. Zhiyuan, Y. Guo, H. Xiaosong (2014) *Research on hybrid electrical vehicle based on human-in-the-loop simulation.* IEEE Conference and Expo Transportation Electrification Asia-Pacific (ITEC Asia-Pacific), Beijing, pp. 1-5, doi: 10.1109/ITEC-AP.2014.6940738

[8] T. Gyorgy and D. Fodorean (2018) *Human-in-the-Loop Simulation of an Electric Vehicle Drivetrain.* XIII International Conference on Electrical Machines (ICEM), Alexandroupoli, pp. 1545-1550, doi: 10.1109/ICELMACH.2018.8506855

[9] Swink, S. (2009) *Game feel: a game designer's guide to virtual sensation.* Morgan Kaufmann Publishers, Burlington, MA. ISBN 978-0123743282

[10] J. Nielsen *Why You Only Need to Test with 5 Users* Nielsen Norman Group, March 19, 2000

[11] L. Faulkner (2003) *Beyond the five-user assumption: Benefits of increased sample sizes in usability testing* Behavior Research Methods, Instruments, & Computers, vol 35, num. 3, p. 379-383

APÉNDICE A
Glosario

- **2D**: Siglas correspondientes a dos dimensiones.
- **3D**: Siglas correspondientes a tres dimensiones.
- **Anchor**: Utilizado por *Unity3D* para representar el punto donde un objeto está anclado en el espacio del canvas.
- **Animator**: Módulo componente para *GameObjects* de *Unity* encargado de contener toda la información de las animaciones.
- **API**: Interfaz de programación de aplicaciones, o *Application Programing Interface*. Conjunto de subrutinas, funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro software.
- **Array**: Listas de objetos de alto nivel.
- **Asset**: Se entiende cómo *asset* a todo componente externo al motor *Unity*, desde imágenes, escenas, *prefabs* o *scripts*.
- **Axys**: Nombre que reciben los ejes de los *joystic*.
- **Best fit**: Opción de los textos de la librería nativa de *Unity3D TextMeshPro* para que una cadena de texto adapte su tamaño para cubrir el mayor espacio de la *textbox*.
- **Brainstorming**: También llamado lluvia de ideas, es una herramienta de trabajo grupal para facilitar el surgimiento de ideas sobre un tema o problema determinado.
- **Broker**: Bróker de mensajería, o *message broker*. Programa intermediario que traduce los mensajes de un sistema a otro a través de un protocolo de comunicación.
- **Build**: Nombre que reciben los programas compilados de *Unity3D*.
- **Bundle**: Componente característico de *OSGI* que contiene interfaces y un archivo *MANIFEST*.
- **Camera** o **MainCamera**: Componente esencial de *Unity* que permite renderizar un canvas.

- **Collider2D**: Conjunto de módulos de *Unity3D* para generar mapas de colisiones en los *GameObjects*.
- **Controller**: Nombre que reciben los dispositivos hardware para la inserción de *inputs* en *Unity3D*. Desde teclados hasta mandos de consola.
- **Coroutine**: Similar al *thread* de JAVA. Método que permite la creación de un hilo de ejecución paralelo.
- **Delay**: Retraso para la ejecución de una tarea o función.
- **DSIC**: Departamento de Sistemas Informáticos y Computación.
- **enum**: Tipo de variable similar a los *Enumeration* de C++.
- **Event**: Clase nativa de *Unity3D* para crear eventos internos del motor.
- **EventSystem**: *GameObject* creado por defecto en un proyecto encargado de gestionar los *inputs* y el sistema de eventos.
- **Font-size**: Parámetro para determinar el tamaño de la fuente de una cadena de texto.
- **Frame**: Fotograma generado por el motor. El hilo principal de *Unity* genera un fotograma al final de cada ciclo.
- **Framework HiL**: Sistema creado por el grupo *TaTAmI* para la implementación de tareas colaborativas en sistemas HiL.
- **Game feel**: Concepto descrito por S. Swink para representar la sensación que produce jugar o utilizar una aplicación.
- **GameObject**: Característica principal del motor *Unity3D*. Es un objeto modular capaz de contener componentes nativos o desarrollados por un programador.
- **Hard-coded**: Práctica que consiste en insertar datos directamente en el código fuente del programa.
- **Hierarchy**: Componente de *Unity3D* dónde se almacenan todos los *GameObjects*, así cómo su organización jerárquica.
- **HiL** o **HitL**: Siglas de *Human-in-the-loop*.
- **HTML**: *HyperText Markup Language*. Un lenguaje de marcas.
- **HTTP**: *HyperText Transfer Protocol*. Un protocolo de comunicación.
- **human-in-the-loop**: Modelo de un sistema que requiere de la interacción humana.
- **ID**: Siglas de Identificador.
- **IDE**: Entorno de desarrollo integrado, o *Integrated Development Environment*. Aplicación informática que facilita al desarrollador o programador el desarrollo de software.

- **Inputs**: Palabra clave de *Unity3D* para referirse a los métodos de introducción de controles en la aplicación.
- **IoT**: Siglas de *Internet of Things*.
- **JSON**: *JavaScript Object Notation*. Formato de texto simple para el intercambio de datos.
- **Layer**: Palabra clave de *Unity3D* para determinar en que orden se renderizan los objetos en el canvas.
- **MQTT**: *Message Queuing Telemetry Transport*. Protocolo de comunicación de paquetes simples usado principalmente en aplicaciones *IoT*.
- **Offset**: Palabra clave de *Unity3D* para determinar la ubicación de un objeto en el canvas.
- **OSGI**: *Open Services Gateway Initiative*. Arquitectura Java que sirve para crear un entorno de *software* que gestiona el ciclo de vida.
- **Output**: Palabra clave de *Unity3D* para referirse a los métodos en los que la aplicación puede transmitir información al usuario.
- **Platform**: Módulo de *Unity3D* que permite la compilación y la configuración especifica del sistema a una plataforma determinada.
- **Prefab**: Configuración de un *GameObject* guardada para facilitar que se instancie.
- **PROS**. Siglas del Centro de Investigación en Métodos de Producción de Software.
- **SFX**: Siglas de *Sound Special Effects*.
- **Size**: Palabra clave de *Unity3D* para determinar el tamaño de un objeto en el canvas.
- **TaTAmI**: Siglas del grupo de Techniques and Tools for Ambient Intelligence.
- **Text-to-Speech**: Sistema que permite traducir cadenas de texto a voz.
- **Textbox**: Módulo de *Unity3D* que permite delimitar una zona de renderizado de texto.
- **Topics**: Cabecera utilizada por *MQTT* para determinar a que sistemas deben recibir un mensaje.
- **Transform**: Módulo de *Unity3D* que contiene toda la información sobre la representación de un objeto en el canvas.
- **Unity3D**: Motor gráfico utilizado principalmente en el desarrollo de videojuegos.
- **UnityWebGL**: *Platform* de *Unity3D* para la realización de proyectos basados en la tecnología *WebGL*.

APÉNDICE B
Archivo de configuración

Debido a la longitud del archivo, solo se ha adjuntado una parte de este.

```
{
    "framework_URL": "127.0.0.1:1883",
    "tasks": [
      "TakeOver"
    ],
    "task_topic": "frameworkHiL/humanPropertiesStatus/
       request",
    "elements":{
        "video":{
            "url": "http://nothumangames.com/Pruebas/
               StreamingAssets/video.mp4",
            "offset": {
                "x": 0.86,
                "y": 3
            },
            "size": 2
        },
        "backgrounds":{
            "bgr01":{
                "id": "bgr01",
                "selector": 1
            }
        },
        "interactuables":{
            "intWheel":{
                "id": "intWheel",
                "selector": 1,
                "offset":{
                    "x": 0,
                    "y": 0
                },
                "size": 1,
                "layer": 1,
                "switch": {
```

```
                    "value": "HandsOnWheel",
                    "true": "fdbHands"
                },
                "in": "",
                "out": ""
            },
            "intPilot":{
                "id": "intPilot",
                "selector": 2,
                "offset":{
                    "x": 0,
                    "y": 0
                },
                "size": 1,
                "layer": 1,
                "exclusive": {
                    "group": "human.location",
                    "value": "driverSeat",
                    "true": "intDriver"
                },
                "in": "",
                "out": ""
            },
            "intCopilot":{
                "id": "intCpilot",
                "selector": 3,
                "offset":{
                    "x": 0,
                    "y": 0
                },
                "size": 1,
                "layer": 1,
                "exclusive": {
                    "group": "human.location",
                    "value": "copilotSeat",
                    "true": "intDriver"
                },
                "in": "",
                "out": ""
            },
            "intDriver":{
                "id": "intDriver",
                "selector": 0,
                "url": "",
                "offset":{
                    "x": 0,
                    "y": 0
                },
                "size": 1,
```

```
                "layer": 1,
                "switch": {
                    "value": "human.attention",
                    "true": "fdbForward",
                    "false": "fdbDistracted"
                },
                "in": "",
                "out": ""
            },
            "intButtonWheel":{
                "id": "intButtonWheel",
                "selector": 4,
                "offset":{
                    "x": 0,
                    "y": 0
                },
                "size": 1,
                "layer": 1,
                "button": {
                    "value": "frameworkHiL/tasks/TO/
                       interaction/response",
                    "text": "Push"
                },
                "in": "HilAction2",
                "out": "HilAction3"
            },
            "intButtonScreen":{
                "id": "intButtonScreen",
                "selector": 5,
                "offset":{
                    "x": 0,
                    "y": 0
                },
                "size": 1,
                "layer": 1,
                "button": {
                    "value": "frameworkHiL/tasks/TO/
                       interaction/response",
                    "text": "Push"
                },
                "in": "NONE",
                "out": ""
            }
        },
        "feedback":{
            "fdb001": "..."
        }
    }
}
```