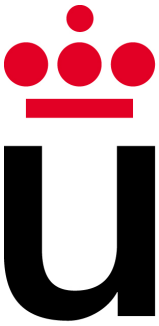


Universidad
Rey Juan Carlos

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

Curso Académico 2022/2023

Trabajo Fin de Grado

Salas compartidas de realidad virtual en navegador web con robots simulados

Autor: Ana Villanueva Ortiz

Tutor: Dr. José María Cañas Plaza

Abril 2023

Agradecimientos

En mi primer lugar, quería agradecer a mi tutor José María la posibilidad que me ha dado para crecer en el mundo del desarrollo web. También quería agradecer a Marta Quintana, integrante del equipo de *Kibotics*, la predisposición y el apoyo que me ha ofrecido en todo momento durante la realización del TFG.

En segundo lugar, dar las gracias a mi familia, sin el esfuerzo de cada uno de vosotros no podría estar aquí. Gracias a mi padres, Ángel y Paloma, vuestra confianza plena en mí ha sido lo que me ha ayudado a seguir esforzándome cada día para alcanzar mis objetivos. Gracias a mi hermano Ángel, el que siempre ha sido mi referente en muchos aspectos de mi vida. Tu pasión por lo estudios y tu paciencia como tutor extraescolar no-remunerado me ha motivado y ayudado a superarme. También, gracias a Neo, quien siempre ha sabido separarme del escritorio de mi habitación para relajarme un par de horas durante nuestras caminatas interminables, aunque también has sabido aceptar los momentos en los que no habia tiempo para descansos.

Por último, Sergio. La persona que más ha sufrido mis episodios de estrés, horas antes de un examen o meses antes de la fecha de entrega de un trabajo. Gracias por confiar en mí, por ser capaz de sacarme del bucle, por todas esas veces que, bajo coacción, me has llevado a casa desde el campus de Fuenlabrada. Gracias por transmitirme todos tus conocimientos de administración de sistemas durante el desarrollo de este trabajo y, sobre todo, gracias por estar ahí.

Resumen

Este Trabajo Fin de Grado está enfocado en la creación de un prototipo de salas compartidas denominado *MiniWebSim*. Este prototipo combina robots simulados y la tecnología *Networked A-frame*, la cual ofrece escenas de realidad virtual compartidas en el navegador.

En particular, el presente trabajo explora alternativas para crear escenas multijugador con varios robots. Hasta ahora el prototipo existente constaba de un usuario maestro que transmitía la escena mediante *WebRTC* a un usuario esclavo gastando un elevado ancho de banda, típico de aplicaciones de videoconferencia. El nuevo prototipo hace uso de la novedosa tecnología *Networked A-frame* (*NAF*). Consta de un servidor centralizado, encargado de facilitar la comunicación entre los dos navegadores en diferentes contextos de conectividad; un navegador maestro encargado del movimiento y las físicas de la escena, así como de las aplicaciones autónomas de ambos robots; y un navegador esclavo en el que únicamente se visualiza la escena compartida. Este prototipo está diseñado para consumir mucho menos ancho de banda en las comunicaciones, puesto que solo se envían las modificaciones en las posiciones de los objetos de la escena compartida, ningún flujo de video.

También ha sido utilizado un nuevo motor de físicas compatible con la tecnología, *Ammo*, y se han implementado conceptos como el modelo maestro-esclavo con físicas centralizadas y mallas de colisión compuestas. La implementación del software ha sido realizada mediante *JavaScript*, mientras que para la introducción de *NAF* ha sido necesario el estudio de esa tecnología para la correcta mezcla con entidades propias del simulador robótico.

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo serán expuestos los tres campos clave en los que se encuadra este TFG: robótica, tecnologías web y realidad virtual. La comprensión de los ámbitos de la robótica y tecnologías web de última generación es fundamental, puesto que la combinación de ambos es necesaria para tener robots simulados en salas compartidas de realidad virtual en navegadores web.

1.1. Robótica

El cometido de la robótica es construir herramientas capaces de recrear comportamientos humanos o animales que ayuden a la vida de las personas. Las partes principales de un robot son las siguientes:

1. **Sensores.** El conjunto de sensores constituye la base del sistema perceptivo del robot. Son los encargados de medir propiedades del entorno o del estado interno del propio robot y convertirlas en datos digitales que pueden ser procesados por un software. Son capaces de detectar obstáculos o cambios de condiciones que darán lugar a diferentes respuestas, dependientes de como hayan sido programados. Algunos de los sensores

más importantes son:

- **Distancia.** Miden la distancia entre un objeto y el sensor. Utilizados en una gran variedad de aplicaciones robóticas, desde robots móviles que necesitan evitar obstáculos hasta robots industriales que necesitan medir la distancia con una precisión elevada. Los sensores basados en ultrasonidos hacen uso de ondas sonoras de alta frecuencia para determinar la distancia a los obstáculos. El sensor de ultrasonido consta de un transmisor que emite una señal de sonido de alta frecuencia y un receptor que recibe la señal reflejada desde el objeto. La distancia se calcula midiendo el tiempo que tarda la señal en viajar desde el transmisor al primer objeto que tiene delante y de vuelta al receptor.

- **Posición.** Miden la posición y orientación del robot. Este tipo de sensor es esencial para el control de movimiento y la navegación autónoma del robot.

- **Cámara.** Capturan imágenes del entorno del robot y las convierten en datos digitales que pueden ser procesados. Estos sensores permiten obtener información visual sobre su entorno, permiten un número elevado de aplicaciones como navegación autónoma, reconocimiento de objetos y la manipulación de objetos.

- **Infrarrojos.** En su configuración más común, los sensores constan de un emisor de luz infrarroja y un receptor que mide la energía infrarroja rebotada en el primer objeto. Son útiles, por ejemplo, para detectar si el robot está posicionado encima de un suelo blanco o una línea negra. Un uso frecuente de estos sensores es en robots autónomos que siguen una línea.

2. **Actuadores.** Dispositivos encargados de convertir la energía en movimiento mecánico, es decir, son componentes capaces de realizar una acción sobre el entorno. Por ejemplo, son considerados actuadores un motor que mueve una rueda, un LED indicativo sobre el estado del aparato o un altavoz que emita sonido. Los actuadores puenden clasificarse

por el tipo de energía utilizada: neumáticos, hidráulicos y eléctricos. Los más utilizados en robótica son los motores eléctricos, cuya fuente de energía es la electricidad. Estos actuadores son populares debido a su alta precisión y capacidad de control, así como por su capacidad de integración con sistemas de control automatizados. Por ejemplo, los motores de corriente continua se controlan en velocidad, por otro lado, los servos se controlan en posición.

3. **Controladores.** El cerebro del robot o la placa controladora es como un pequeño ordenador. Básicamente se trata de un microcontrolador que se encarga de coordinar y supervisar todas las funciones y movimientos del robot, es decir, se encarga de procesar datos procedentes de los sensores y generar movimientos que controlan al robot.

4. **Cuerpo.** Es lo que se denomina el esqueleto del robot, es decir, la estructura física que sostiene y contiene todos los componentes internos del robot, incluyendo sensores, actuadores y controladores.

5. **Software.** Otorga la inteligencia al robot, es decir, le posibilita realizar tareas autónomas. También, permite al usuario controlar un robot de forma simple e intuitiva.

La robótica busca impactar de forma positiva en la sociedad reduciendo los esfuerzos del ser humano en muchos sectores. A continuación se resumen a modo ilustrativo diferentes áreas en las que se emplean robots hoy en día. La Figura 1.1 muestra algunos de ellos.

Por un lado, en la industria militar la robótica juega un papel importante, puesto que se estima la reducción de pérdida de vidas humanas, minimizando la exposición al riesgo. En este sector destacan el robot *BigDog*, encargado del transporte de materiales en cualquier tipo de terreno, y *Packbot*, utilizado para detectar y desactivar bombas. Los vehículos de combate aéreos, más conocidos como drones de combate, han cambiado las operaciones militares.

En medicina es posible facilitar el trabajo a los profesionales reduciendo las tareas mécanicas. Además, serán posibles métodos menos invasivos, intervenciones más rápidas y el acceso

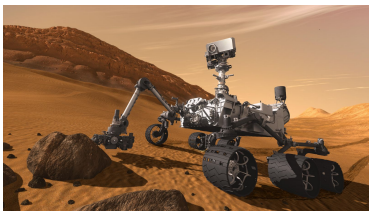
a zonas más delicadas. Sobresalen *ViRob*, nanorrobot de 14mm capaz de desplazarse por el cuerpo humano liberando fármacos en zonas de difícil acceso, *Riba*, robot de asistencia en hospitales que ayuda a personas mayores a realizar tareas cotidianas como levantarles de la cama, y *Da Vinci*, sistema robótico que amplía la capacidad del cirujano para operar de manera menos invasiva y con mejores resultados.

Por otro lado, en el hogar ha sido posible la reducción de tareas domésticas mejorando la calidad de vida. Dos de los más populares son *Roomba*, aspiradora de forma circular que limpia automáticamente los distintos espacios, y *Domótica*, permite la automatización de viviendas, persianas, luz, etc.

La industria automotriz tiene en el punto de mira la construcción de vehículos autónomos, puesto que entre sus beneficios pueden encontrarse una mejora en el medio ambiente y en la seguridad vial. Los vehículos autónomos están constituidos por sensores, cámaras, radares y tecnologías de detección y medición por láser, que recopilan información sobre el entorno. El software analiza la información de los sensores y toma decisiones en tiempo real sobre cómo deben responder los vehículos. Destacan *Tesla*, depende exclusivamente de cámaras, y *Waymo*.

En el sector espacial hay dos áreas de interés: la robótica orbital, especializada en tareas autónomas en escenarios sin gravedad, y los exploradores planetarios, diseñados para operar en la superficie de planetas o lunas. Destacan *Curiosity*, vehículo espacial diseñado para hacer exploraciones en el planeta Marte, y *Dextre*, robot telemanipulador encargado de los trabajos rutinarios dentro y fuera de la estación espacial.

Por último, la robótica en la agricultura permitirá reemplazar el personal faltante y superar la productividad. Despuntan *Vinerobot*, identifica las uvas que están en perfecto estado para recolectarlas, y *RIPPA*, controla las plagas mediante la identificación de malas hierbas y la fertilización de los cultivos.



(a) Sector espacial - Curiosity

(b) Sector hogar - Roomba 860

(c) Sector militar - Packbot

(d) Sector agricola - Vinerobot

Figura 1.1: Robots pertenecientes a sectores diferentes

Los robots pueden ser clasificados cronológicamente, en base a su inteligencia o según su funcionalidad. A continuación, se expone una clasificación en 5 generaciones dada por T. Michael Knasel[1], director del Centro de Aplicaciones Robóticas de Science Application Inc.

- **Primera generación - *Pick and Place.*** Usados para servir de complemento a máquinas industriales. Anclados a una base fija.

- **Segunda generación - *Servocontrol.*** Caracterizados por controles definidos con un controlador de movimiento, capaces de desplazarse por una vía y dotados de programas con trayectoria continua. Empleados fundamentalmente en soldadura y pintura.

- **Tercera generación - *AGVs.*** Disponen de un control más avanzado, con servomecanismos de precisión. Dotados de visión, tacto y movimiento autoguiado. Usados en funciones de acoplamiento, montaje y transporte interno.

[1] `https://ignaciogavilan.com/las-cinco-generaciones-de-robots-segun-michael-knasel/`

- **Cuarta generación - Robots móviles.** Utilizados en la construcción y procesos de mantenimiento. Formados por ruedas o piernas artificiales y dotados de sensores como sistemas de visión, sensores láser o sensores de proximidad.

- **Quinta generación - Basados en inteligencia artificial.** Controladores basados en inteligencia artificial. Dotados de movilidades con diferentes tipos de andadores. Por ahora su uso se reduce al ámbito militar.

1.1.1. Simuladores robóticos

Los sistemas robotizados están sometidos a condiciones muy exigentes de precisión, fiabilidad y velocidad. Los simuladores en robótica permiten crear aplicaciones para un robot sin depender de la máquina física, adelántandose a errores de diseño y ahorrando costos y tiempo. El software de simulación permite mostrar las acciones del robot y las piezas de ensamblaje en un entorno virtual tridimensional meses antes de que se produzcan los prototipos. Los simuladores robóticos también pueden ser utilizados en el ámbito educativo, puesto que facilitan a los centros la enseñanza de conceptos básicos sobre robótica y programación de modo que permite desarrollar habilidades relacionadas con el pensamiento computacional como pensamiento algorítmico, abstracción, reconocimiento de patrones y descomposición.

Algunos de los simuladores robóticos más utilizados son:

***Webots*[2].** Simulador de código abierto cuya principal ventaja es la capacidad del usuario de interactuar con el modelo durante la simulación. Basado en el motor de física ODE (*Open Dynamics Engine*) y el motor de renderizado *OpenGL*. El control del robot se puede escribir en *C, C++, Java, Python, Matlab* y ROS. Este software permite modelar, programar y simular todo tipo de robots.

***CoppeliaSim*[3].** Simulador desarrollado por *Coppelia Robotics*. Tiene un entorno de desa-

[2] `https://cyberbotics.com/`

[3] `https://www.coppeliarobotics.com/`

rrollo integrado, que se basa en una arquitectura de control distribuido, donde cada objeto/-modelo puede controlarse individualmente mediante *script Python, Lua* y *C/C++*. Tiene una programación multienfoque, es decir, puede programarse con 6 lenguajes distintos (*Matlab, Octave, C/C++, Python, Java, Lua*) y 6 paradigmas (API remota, *scripts* embebidos, *plugins*, entre otros). Este simulador utiliza un motor cinemático para cálculos cinemáticos directos e inversos, y varias bibliotecas de simulación física (*MuJoCo, Bullet, ODE, Vortex, Newton Game Dynamics*) para realizar simulaciones de cuerpos rígidos.

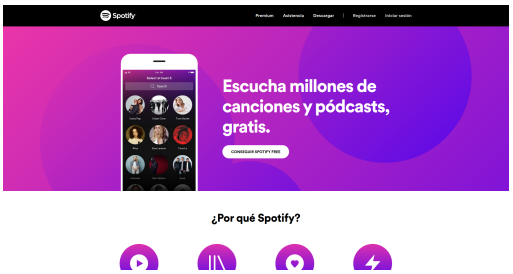
Gazebo[4]**.** Es el simulador de código abierto más extendido. Destaca gracias a su motor de renderizado, sus motores de físicas (*ODE, Bullet*) y su soporte para *plugins* de sensores y actuadores, además su catálogo de robots es muy extenso. Otro hecho importante es su integración con ROS (*Robotics Operating System*)[5], por lo que permite probar el software desarrollado en multitud de robots simulados.

1.2. Tecnologías web

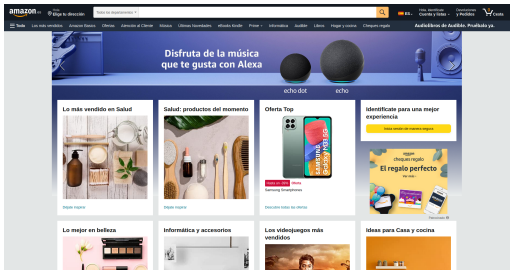
El avance de las tecnologías web es veloz, por ello, es necesario mantenerse al tanto de las tecnologías más actuales. Se ha de distinguir entre tecnologías *backend*, que trabajan en el lado del servidor, y tecnologías *frontend*, que trabajan en el lado del cliente, es decir, en el navegador web. La idea principal de esta separación es marcar las diferentes partes de un sistema software web.

Hoy en día son muy comunes las aplicaciones web que se basan en estas tecnologías. Una aplicación web es un software cliente-servidor que permite realizar funciones determinadas en Internet, como usar el navegador web para leer el correo electrónico, ver contenido audiovisual, comprar productos, entre otras cosas. Algunas de las aplicaciones web con más éxito en los últimos años son *Gmail*, *Spotify*, *Netflix* o *Amazon*.

[4] `https://gazebosim.org/home`
[5] `https://ros.org/`



(a) Dedicada a la distribución de contenido



(b) Dedicada a la distribución de contenido

Figura 1.2: Aplicaciones web

Por un lado, el *frontend* se ocupa de la creación de interfaces de usuario y de establecer comunicaciones con el servidor. *HTML*, *CSS* y *JavaScript*, sus tecnologías principales, permiten al usuario interactuar con el sevidor web, utilizando un navegador como intérprete y como interfaz de usuario.

Por otro lado, el *backend* se ocupa de la implementación de comportamientos de la web en el servidor. Incluye lenguajes como *PHP*, *Python*, *Ruby*, *.NET* o *Java* y las bases de datos sobre las que se puede trabajar son *PostgreSQL*, *SQL*, *MongoDB* o *MySQL*.

La arquitectura cliente-servidor es un modelo en el que el cliente se encarga de recoger los datos que introduce el usuario y el servidor los procesa. El protocolo que permite el diálogo entre servidor web y clientes es el protocolo *HTTP* (*Hypertext Transfer Protocol*). Entre los métodos más utilizados de este protocolo encontramos[6]:

- **GET.** Solicita una representación de un recurso específico. Únicamente utilizada para recuperar datos.

- **HEAD.** Petición de una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.

[6] `https://developer.mozilla.org/es/docs/Web/HTTP/Methods`

- **POST.** Causa a menudo cambios en el estado del servidor. Utilizado para enviar una entidad a un recurso en específico.

- **PUT.** Reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.

- **DELETE.** Borra un recurso en específico.

Figura 1.3: Protocolo *HTTP*

1.2.1. Tecnologías web en el lado del cliente

Las tecnologías web de *frontend* corren en el navegador, es decir, permiten la interacción entre el usuario y el servidor web. Existen varios lenguajes usados en el *frontend*, entre los que destacan *HTML*, *CSS* y *JavaScript*.

- ***HTML.*** Es un lenguaje de marcado de los contenidos de sitios web. Se trata de una tecnología con una evolución a pasos agigantados y, en la actualidad, se encuentra en la versión *HTML5*. Este nuevo estándar tiene una gran cantidad de novedades. Entre las mejoras encontramos que es posible añadir archivos multimedia a la web sin necesidad de *plugins*, incluso incluye una nueva etiqueta que permite crear animaciones 2D. Esta

nueva versión también ha añadido opciones de geolocalización, permitiendo adaptar el idioma de la página web, y es compatible con diseños adaptativos, es decir, la página web tiene la capacidad de reconocer el tipo de dispositivo y adaptarse.

HTML permite crear texto enriquecido mediante el uso de etiquetas, es decir, el texto es guardado en un archivo llamado *HTML* fácilmente encontrable a través de los navegadores. Las etiquetas que lo componen son variadas y tienen diversos significados. Las atributos se encargan de personalizar las etiquetas, es decir, describen las características.

En resumen, *HTML* permite describir el contenido de un sitio web, su información estructurada de párrafos, imágenes, etc. Por todo esto es una de las tecnologías imprescindibles para cualquier desarrollador web.

- ***CSS.*** Se trata de un lenguaje de hojas de estilo creado para controlar la presentación de la página web. Es una de las tecnologías web a la que más recurren los programadores gráficos para desarrollar sus proyectos. Entre sus funciones se encuentra indicar la presentación visual de las estructuras *HTML*, es decir, sirve para acotar y trazar el aspecto visual de las etiquetas generadas por el *HTML*.

- ***JavaScript.*** Lenguaje de programación multiplataforma con el que es posible dar mayor interactividad y dinamismo a las páginas web. Entre sus funciones se encuentra crear animaciones, objetos y *cookies*, localizar errores en formularios y cambiar elementos web de manera intuitiva. Además, también permite desarrollar aplicaciones tan potentes como *YouTube* o *Twitter*. Se trata de una de las tecnologías web más usadas en la actualidad.

1.2.2. Tecnologías web en el lado del servidor

En el lado del servidor se encuentran algunas de las tecnologías que permiten acceder a bases de datos, otorgan escalabilidad, gestionan los servidores y sirven las páginas web

mediante la utilización de plantillas. Existen gran variedad, entre otros podemos encontrar *PHP*, *Node.js*[7] o *Django*[8].

- ***Node.js.*** Se trata de un entorno de ejecución *JavaScript* frecuentemente utilizado en la programación del *backend*. *JavaScript* inicialmente era utilizado solo en el *frontend*; sin embargo, gracias al motor *Node.js*, este lenguaje puede también ser utilizado en el lado del servidor.

- ***Django.*** Es un entorno web de alto nivel programado en *Python*. Entre sus características destacan que es rápido y fácil de utilizar. Además, dispone de una interfaz para el manejo de las bases de datos *SQL* y se puede acceder a ellas en el código mediante instrucciones *Python*.

1.3. Realidad virtual

La realidad virtual consiste en la inmersión sensorial de un usuario en un mundo ficticio 3D, que ha sido generado de forma artificial y puede percibirse gracias a unas gafas de realidad virtual y otros accesorios. Los sistemas de realidad virtual suelen estar compuestos por los siguientes elementos:

- **Gafas de realidad virtual.** Encargadas del aislamiento del mundo real mediante la salida de imágenes, puesto que la vista es uno de los sentidos que más estímulos recibe. Usualmente incorporan dos pantallas, una para cada ojo, o tienen el soporte necesario para incluir una, como un *smartphone*. Las técnicas de visión estereoscópica otorgan profundidad y realismo a las imágenes tridimensionales con la superposición de dos imágenes que crean sensación de profundidad.

- **Procesador.** Es la parte del hardware encargada del funcionamiento de la tecnología.

[7] `https://nodejs.org/en/`
[8] `https://www.djangoproject.com/`

Normalmente se integra a la pantalla, generando los gráficos tridimensionales que ofrecen una percepción real de lo que se observa a través de las gafas de realidad virtual.

- **Sensores de posición.** Dispositivo encargado de monitorizar la ubicación y movimientos 3D del usuario para recrearlos en el entorno virtual, es decir, miden el movimiento de la cabeza de la persona para mostrarle la escena virtual acorde a esos movimientos y giros.

- **Software.** Componente informático encargado de gestionar la ejecución, procesar las imágenes y coordinar al sistema en general.

Además de los componentes básicos, esta tecnología puede incluir más sensores o periféricos opcionales para aumentar el nivel de funcionalidad e interacción. Por ejemplo:

- **Sistemas de audio.** Teóricamente un entorno de realidad virtual puede funcionar sin audio, pero normalmente es incluido para incrementar la inmersión.

- **Sensores de amplio alcance.** Consiste en la distribución de sensores de posicionamiento en varias habitaciones, permitiendo que el sistema ubique a los usuarios y traslade sus movimientos al entorno virtual.

- **Mandos de mano.** Similares a un *joystick* para videojuegos, los entornos de realidad virtual pueden incluir mandos o controles de mano como sistemas de entrada, para gestionar funciones en el entorno virtual de manera rápida. La visión se fusiona con la aplicación sobre la que el usuario interactúa. Con ellos, los controles se hacen tan intuitivos que será muy fácil desenvolverse en estos mundos, ya que los movimientos son naturales.

En un sentido general, esta tecnología no requiere de periféricos especiales, más de los que pudiera encontrar en un teléfono y un software con la capacidad de recrear escenarios, pero puede adaptarse lo suficiente como para cubrir un amplio número de posibilidades.

La capacidad de abstraer al usuario de su realidad y posicionarlo en un entorno simulado provoca que el impacto social de la realidad virtual sea muy extenso. Por ejemplo, en el mundo del entretenimiento se esperan cambios significativos, particularmente en el mundo de los videojuegos, permitiendo una experiencia extremadamente real. La imposibilidad de ser teletransportados a otros lugares ha posibilitado que determinadas ramas aprovechen el potencial de la realidad virtual. Por un lado, el sector inmobiliario ha adaptado la tecnología para que potenciales compradores tengan la capacidad de visitar las propiedades sin perder horas de su día programando una visita con el agente inmobiliario. Por otro lado, en el sector turistico los usuarios pueden visitar lugares exóticos o probar hoteles en el otro lado del mundo, evitando los problemas que puede suponer una estancia real. La educación online también puede verse mejorada gracias a esta tecnología, ya que será posible que los alumnos asistan a las clases desde una ubicación remota. Por todo lo anterior, la aparición de la realidad virtual supone uno de los cambios tecnológicos más importantes de los últimos tiempos, su adopción a nivel usuario supondrá un antes y un después en la forma de vida del ser humano.

1.4. Plataforma educativa *Kibotics*

Las tecnologías web ofrecen nuevas oportunidades para la robótica, entre ellas, la docencia. La robótica educativa es un sistema de enseñanza interdisciplinar que permite a los estudiantes desarrollar sus conocimientos y habilidades de la educación STEM[9] (*Science, Technology, Engineering and Maths*). Durante los últimos años la docencia robótica ha crecido gracias a que es una manera atractiva de introducir a los niños en el mundo de la tecnología.

En consecuencia, han surgido plataformas dedicadas a la docencia robótica, encargadas de impartir cursos de iniciación a la robótica desde edades muy tempranas para atraer y

[9]Método educativo que busca, en primer lugar, potenciar las capacidades de los estudiantes y desarrollar nuevas tecnologías, en segundo lugar, la combinación de estas disciplinas integradas forman una base de pensamiento.

ayudarles a adquirir las habilidades necesarias para afrontar los problemas de los programadores.

Entre las plataformas destacadas encontramos *Lenobotics*[10], *LEGO education*[11], *OpenRoberta*[12], *iRobot*[13], entre otras. El presente trabajo se centra en la plataforma *Kibotics*[14], un entorno web para la docencia robótica y programación que permite la introducción de los más pequeños en el mundo del desarrollo. *Kibotics* fomenta el aprendizaje de estas disciplinas a través de una formación práctica, en la que los estudiantes trabajan a través de la experimentación, puesto que es más atractivo y eficaz.

Kibotics se basa en la utilización del simulador *WebSim* [16] que, a su vez, esta basado en la tecnología *A-frame*[15] de realidad virtual que permite crear mundos tridimensionales. Este simulador contiene diferentes ejercicios para los robots que soporta la plataforma.

(a) Mbot (b) Dron Tello (c) LegoEV3

Figura 1.4: Robots soportados por la plataforma

Estos robots simulados cuentan con sensores y actuadores que permiten a los usuarios solucionar los diferentes ejercicios encontrados en la plataforma.

Existe una gran cantidad de motores de físicas como *Box2D*[16] (simulaciones 2D), *Co-*

[10] https://lenobotics.com/
[11] https://education.lego.com/es-es/
[12] https://www.open-roberta.org/
[13] https://www.irobot.es/
[14] https://kibotics.org/
[15] https://aframe.io/
[16] https://box2d.org/

cos2D[17] (simulaciones 2D), *Cannon.js*[18] (simulaciones 3D) o *Ammo.js*[19] (simulaciones 3D). El presente trabajo se va a centrar en este último, ya que el motor que emplea *A-frame* por defecto, *Cannon.js*, se quedará obsoleto en los próximos años y tiene una menor cantidad de funcionalidades. En el capítulo 4, se explicarán con mayor detalle las peculiaridades de este motor de físicas.

Por el momento, *Kibotics* no ofrece salas compartidas. Anteriormente se construyó un primer prototipo basado en la compartición de las escenas mediante la transmisión de flujo de vídeo *WebRTC* [14], obteniéndose un consumo de ancho de banda elevado y complicaciones en algunos escenarios de conectividad. El presente Trabajo Fin de Grado tiene su motivación principal en explorar una tecnología diferente, *Networked A-frame*[20], para hacer un segundo prototipo de salas compartidas con robots simulados, en las que las entidades de una escena se sincronicen y únicamente los cambios de posición de los robots en la escena sean compartidos entre navegadores.

[17] `http://www.cocos2d.org/`

[18] `http://schteppe.github.io/cannon.js/`

[19] `https://github.com/kripken/ammo.js/`

[20] `https://github.com/networked-aframe/networked-aframe`

Capítulo 2

Objetivos

En este capítulo se explican los objetivos del presente TFG, la metodología seguida y la planificación llevada a cabo durante el proceso de desarrollo.

2.1. Objetivo

El objetivo general que persigue el presente trabajo es explorar tecnologías de salas compartidas de realidad virtual que incluyen robots simulados. Los tres subobjetivos concretos en los que se ha articulado son los siguientes:

- Crear un prototipo *Networked A-frame* con salas compartidas de realidad virtual, el cual incluye objetos distribuidos visibles en múltiples navegadores que se ven afectados por físicas realistas ofrecidas por el motor de físicas *Ammo.js*.

- Cambiar el motor de físicas utilizado hasta ahora en el simulador *WebSim*, *Cannon.js*, por *Ammo.js*. El cambio es necesario dado que el nuevo motor de físicas ofrece muchas funcionalidades nuevas, entre ellas, la compatibilidad con la tecnología *Networked A-frame*. Es importante mantener la combinación con el motor complementario existente

en el simulador para que las fuerzas generadas voluntariamente por la aplicación que controla el comportamiento del robot sean adecuadas. El nuevo motor de físicas permite crear mallas de colisión compuestas que se ajustan en mayor medida a los modelos 3D importados en la escena.

- Crear un prototipo de salas compartidas de realidad virtual con dos robots que combine la tecnología *Networked A-frame* y un simulador robótico. Este nuevo simulador, denominado *MiniWebSim*, hará uso de un menor ancho de banda en las comunicaciones entre todos los componentes del sistema. Debe hacerse uso del concepto maestro-esclavo, dada la experiencia adquirida en los dos subobjetivos anteriores. Además, es necesario implementar los sensores y actuadores de los robots que componen la escena.

2.2. Metodología

El modelo de desarrollo utilizado es el modelo en espiral, muy extendido en el desarrollo de software. Es un modelo que se adapta a las necesidades, permitiendo disponer de flexibilidad ante cambios en los requisitos semanales. Con esto, se ha logrado una reducción del riesgo en el proyecto gracias a la temprana identificación de potenciales riesgos. El desarrollo en espiral principalmente abarca las siguientes cuatro fases:

- **Determinar objetivos, alternativas y restricciones.** Los requerimientos del sistema se definen con el mayor detalle posible, indicándose los objetivos asociados al ciclo actual de desarrollo. Además, se examinan las diferentes alternativas y restricciones para su implementación.

- **Análisis de riesgo.** Se evalúan todas las alternativas propuestas. Los objetivos y restricciones son determinantes para seleccionar la mejor solución.

- **Desarrollo y prueba.** Se realiza todo el desarrollo necesario, utilizando la tecnología y solución seleccionada. Con cada iteración se va creando una mejor versión de la

aplicación.

- **Planificación del próximo ciclo.** Al completar un ciclo se comienza la planificación del siguiente. Si el objetivo del ciclo se alcanzó se plantearía la definición del próximo objetivo, en caso contrario se han de buscar soluciones. Los tres ciclos principales se corresponden con los subobjetivos presentados en el apartado 2.1.

Se estableció una reunión semanal dedicada a compartir los progresos, por parte del alumno, y el direccionamiento de los esfuerzos, por parte del tutor. Paralelamente a las reuniones se ha contado con un canal de *Slack* de comunicaciones en el que se han podido plantear todo tipo de dudas durante el proceso de aprendizaje, en el se pueden encontrar a todos los desarrolladores de la plataforma *Kibotics*. También se ha elaborado un blog[1] actualizado semanalmente, en el que se han ido compartiendo los resultados del trabajo realizado.

2.3. Plan de trabajo

El proceso de elaboración del Trabajo de Fin de Grado se ha dividido en las siguientes fases:

- **FASE 1.** Aprendizaje y primera toma de contacto con las tecnologías web necesarias para la elaboración del trabajo. Especialmente *A-frame*, el sistema de físicas *Ammo.js* y *Networked A-frame*.

- **FASE 2.** Adaptación y análisis del principio maestro-esclavo a escenas *A-frame* multijugador.

- **FASE 3.** Estudio profundo del motor de físicas *Ammo.js* con la finalidad incluir el

[1] `https://github.com/RoboticsLabURJC/2022-tfg-ana-villanueva/tree/main/docs/_posts/training-stage`

sistema de físicas en el simulador *WebSim* y crear mallas de colisión compuestas.

- **FASE 4.** Combinación de la tecnología de la tecnología *Networked A-frame* y el simulador *MiniWebSim* con el objetivo de crear escenas compartidas con robots con sensores y actuadores simulados.

- **FASE 5.** Escritura de la memoria.

Capítulo 3

Herramientas

En este capítulo se presentan los diferentes componentes software en los que se ha apoyado el trabajo actual durante su desarrollo. Principalmente han sido empleados los entornos *A-frame* y *Networked A-frame*, encargados de crear experiencias de realidad virtual y salas compartidas, respectivamente. Estos entornos se construyen sobre las tecnologías web *HTML* y *JavaScript*. También ha sido aprovechada la aplicación de modelado 3D *Blender*, para la creación de modelos 3D, y el simulador robótico *WebSim*.

3.1. Lenguaje *JavaScript*

JavaScript[1], dialecto del estándar *ECMAScript*[2], es un lenguaje de programación interpretado que se utiliza principalmente para crear aplicaciones web interactivas y dinámicas. Para interactuar con una página web se provee al lenguaje *JavaScript* de una implementación del *DOM* (*Document Object Model*) proporcionada por el navegador web. El *DOM* es la estructura que representa el contenido de un documento *HTML* o *XML* en un objeto

[1] `https://developer.mozilla.org/es/docs/Web/JavaScript`

[2] Especificación de lenguaje de programación publicada por Ecma International. Actualmente está aceptado como el estándar ISO/IEC.

que puede ser manipulado por *JavaScript*. Facilita una representación estructurada del documento y define de qué manera los programas pueden acceder, al fin de leer o de modificar, tanto a su estructura, estilo y contenido. Se trata del lenguaje de programación de referencia que entienden de forma nativa los navegadores modernos.

Sus características principales son:

- **Lenguaje del lado del cliente.** Típicamente es ejecutado en la máquina del cliente a través del navegador. También se utiliza en algunos servidores como *Node.js*, pero nació principalmente para el navegador web.

- **Orientado a objetos.** Utiliza objetos como estructuras que permiten organizarse de forma simple y que son reutilizables durante todo el desarrollo. No utiliza clases de forma nativa, sino prototipos.

- **Tipado débil o no tipado.** No es necesario especificar el tipo de dato al declarar una variable. Esta característica supone una gran ventaja a la hora de ganar rapidez programando.

- **Alto nivel.** Sintaxis fácilmente comprensible por su similitud al lenguaje de las personas. Es un lenguaje de alto nivel porque su sintaxis se encuentra alejada del nivel máquina.

- **Lenguaje interpretado.** Se convierten las líneas de código al lenguaje de la máquina mediante un intérprete. Esto tiene muchas de ventajas, como la reducción del procesamiento en servidores web al ejecutarse directamente en el navegador del usuario, y que es apto para múltiples máquinas y procesadores, permitiendo ejecutar en ellas el mismo código *JavaScript*.

- **Muy utilizado por desarrolladores.** Su versatilidad y su infinita capacidad para crear plataformas cada vez más atractivas provoca que *Javascript* sea en la actualidad

uno de los lenguajes más demandados de los últimos años.

3.2. Lenguaje *HTML*

HTML[3] (*HyperText Markup Language*) es un lenguaje de marcado que permite indicar la estructura de un documento mediante etiquetas. Es el componente básico de la web, ofreciendo una gran adaptabilidad, una estructuración lógica y facilidad de interpretación, tanto por humanos como por máquinas.

Las partes principales del documento *HTML* son las siguientes:

1. Línea que contiene la versión de *HTML*.

2. Sección de cabecera declarativa delimitada por la etiqueta `head`.

3. Cuerpo que contiene el contenido real del documento delimitado por la etiqueta `body`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Hello world!</title>
</head>
<body>
    <a-scene>
        <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9"></a-box>
        <a-sphere position="0 1.25 -5" radius="1.25" color="#EF2D5E"></a-sphere>
        <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" color="#FFC65D"></a-cylinder>
        <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4" color="#7BC8A4"></a-plane>
        <a-sky color="#ECECEC"></a-sky>
    </a-scene>
</body>
</html>
```

Figura 3.1: Estructura básica de *HTML*

Es en el cuerpo del *HTML* donde se forma la escena. Entre los componentes destacan:

[3] `https://developer.mozilla.org/en-US/docs/web/html`

- Etiquetas de apertura.
- Etiquetas de cierre.
- Contenido.
- Cero o más atributos.

3.3. Lenguaje *JSON*

JSON[4] (*JavaScript Object Notation*) es un formato de texto sencillo utilizado para el intercambio de datos, ya que es flexible, ligero y transferible a través de las redes. El formato *JSON* se basa en la sintaxis de los objetos y arrays de *JavaScript*, lo que lo hace fácil de entender y manipular por los desarrolladores. La información en *JSON* se almacena en pares de "clave: valor", donde la clave es una cadena que describe el tipo de dato que se está almacenando y el valor puede ser cualquier tipo de dato válido, incluyendo otros objetos *JSON* y arrays.

```
[
    {
        "nombre": "Neo",
        "edad": 16,
        "hobbies": ["comer", "dormir", "jugar"]
    },
    {
        "nombre": "Rebeca",
        "edad": 30,
        "hobbies": ["beber", "dormir", "fiesta"]
}
]
```

Figura 3.2: Estructura de datos simple en formato JSON

[4] `https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/JSON`

3.4. *Blender*

Blender[5] es un programa informático multiplataforma de creación de gráficos tridimensionales. Entre las funciones principales encontramos modelado 3D, iluminación, renderizado y animación. También se pueden realizar acciones relacionadas con la composición de vídeo.

Durante el presente trabajo se ha utilizado la herramienta para la crear los robots de los simuladores. Entre las principales ventajas encontramos la capacidad de *Blender* de exportar los modelos en formato *glTF* (*GL Transmission Format*) basado en el estándar *JSON*, que permite la compresión de escenas y modelos 3D para minimizar el tiempo de ejecución de los programas en los que posteriormente serán utilizados.

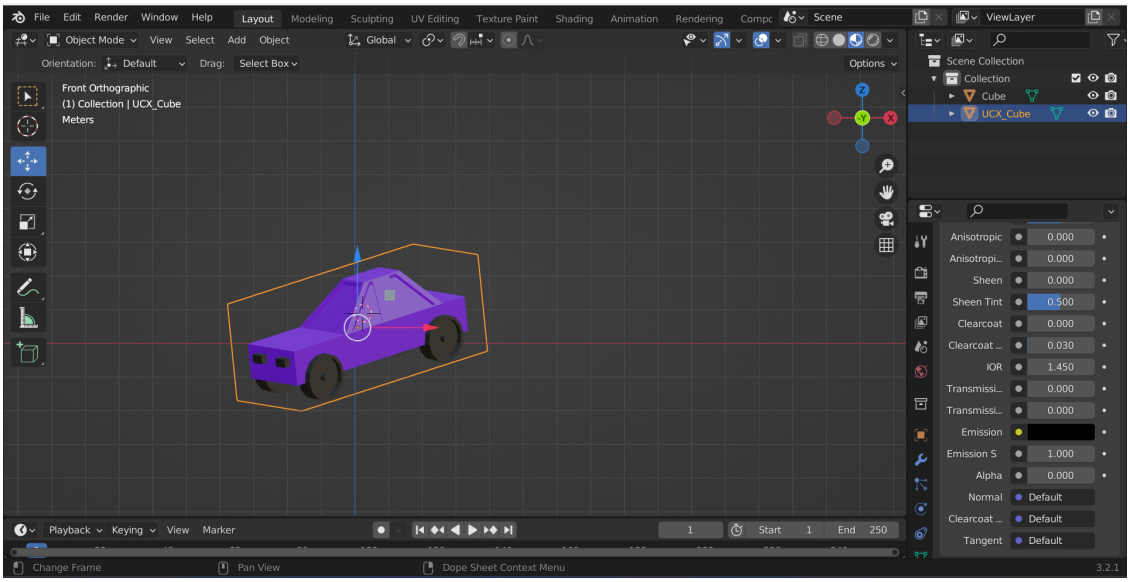


Figura 3.3: Modelo 3D diseñado en *Blender*

[5] `https://www.blender.org/`

3.5. Entorno *A-frame*

A-Frame[6] es un entorno de código abierto para crear experiencias de realidad virtual. Es una estructura de sistema-componente-entidad para *Three.js*[7], donde los desarrolladores pueden crear escenas 3D y *WebVR* usando *HTML*. Sus principales características son las siguientes:

- ***HTML* declarativo.** Se trata de una tecnología basada en *HTML*, por ello, es sencilla y accesible.

- **Realidad virtual sencilla.** *A-Frame* solo necesita las etiquetas *script* y *a-scene* para ser puesto en funcionamiento. Por si mismo se encarga de manejar el modelo 3D, la configuración de realidad virtual y los controles predeterminados.

- **Arquitectura componente-entidad.** *A-Frame* es un potente marco de *Three.js* que proporciona una estructura componente-entidad declarativa, componible y reutilizable. Los desarrolladores tienen acceso ilimitado a *JavaScript*, *API DOM*, *Three.js*, *WebVR* y *WebGL*.

- **Multiplataforma.** Compatible con una gran cantidad de plataformas como *Live*, *Rift*, *Windows Mixed Reality*, *Daydream*, *GearVR* y *Cardboard*.

- **Rendimiento.** Las actualizaciones de los objetos 3D se realizan en la memoria con poca sobrecarga. *A-Frame* está optimizado desde cero para *WebVR*.

- **Inspector visual.** Incorpora un práctico inspector visual 3D.

- **Componentes.** *A-Frame* cuenta con una gran cantidad de componentes. Encontramos desde componentes geométricos y materiales básicos hasta componentes para la realidad aumentada o componentes personalizados.

[6] `https://aframe.io/`

[7] Biblioteca que proporciona muchas funciones y APIs para dibujar escenas 3D en su navegador.

- **Comprobado y escalable.** Ha sido utilizado por empresas como *Google*, *Disney*, *Samsung*, *Toyota*, entre otras. Además, empresas como *Google*, *Microsoft*, *Oculus* y *Samsung* han realizado contribuciones a *A-Frame*.

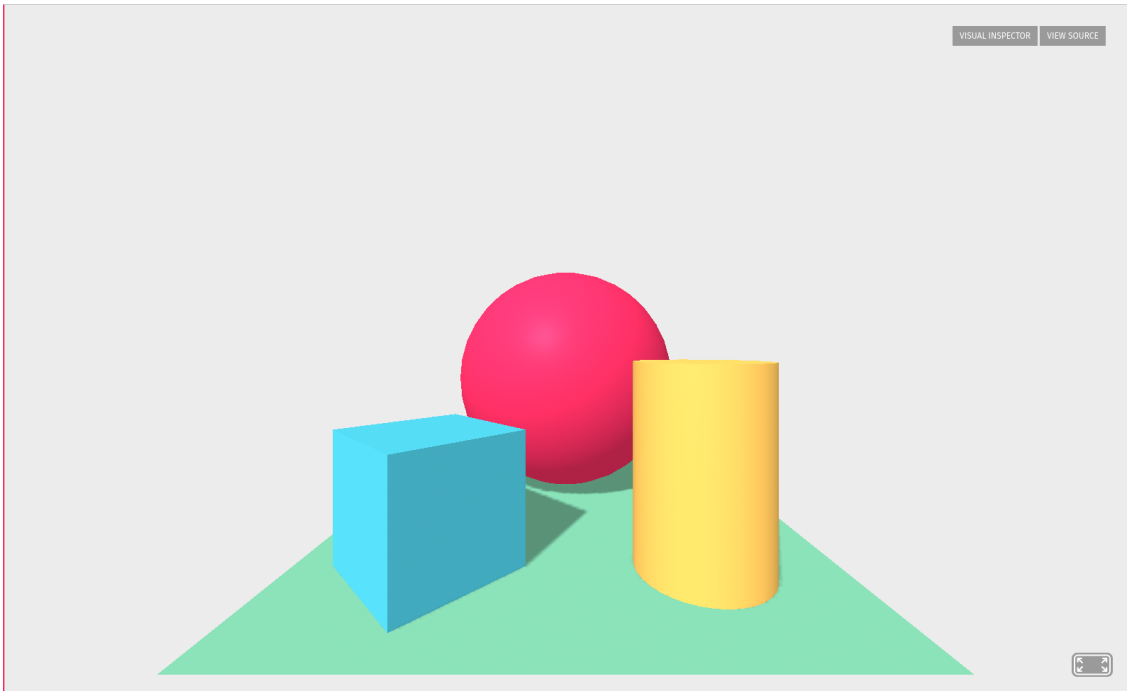


Figura 3.4: Ejemplo de construcción 3D con *A-Frame*

3.5.1. Sistema de físicas de *A-Frame*

Los juegos basados en físicas han tenido una gran proliferación en los últimos años. El motor realiza una simulación física de los objetos en la pantalla, siguiendo las leyes de la cinemática y la dinámica. Es decir, los objetos están sujetos a la gravedad y cada uno de ellos tiene una masa, cuando se produce una colisión entre ellos se produce una fuerza que dependerá de su velocidad y su masa. *A-Frame* soporta dos motores de físicas: *Cannon.js*, utilizado por defecto, y *Ammo.js*.

Para incluir el motor de físicas en una escena basta con añadir el siguiente *script* en la cabecera del *HTML* de la aplicación. Una vez incluido los componentes de las físicas pueden ser utilizados.

```
src="//cdn.rawgit.com/donmccurdy/aframe-physics-system/v4.0.1/dist/aframe-p⌋
↪ hysics-system.min.js"
```

El sistema de físicas de *A-frame* cuenta con dos tipos de cuerpos: dinámicos y estáticos.

- **Cuerpo dinámico.** Objetos con libertad de movimiento, tienen masa, chocan con otros objetos, rebotan y se ven afectados por la gravedad.

- **Cuerpo estático.** Objetos de posición fija con los que otros objetos pueden colisionar, pero los cuerpos estáticos en sí mismos no se ven afectados por la gravedad y las colisiones.

Los sistemas de físicas incluyen mallas de colisión, que son áreas que envuelven a la entidad utilizada para detectar las colisiones entre diferentes objetos del juego. *Cannon.js* ofrece dos tipos de mallas de colisión, simples o compuestas. Estas últimas permiten adaptarse lo máximo posible a la figura de una entidad compleja mediante la unión de varias entidades simples.

Por último, el motor de físicas tiene una serie de parámetros de valor modificable que permiten ajustar el mundo a las características deseadas. En caso de no indicarse el valor de un parámetro se tomará su valor por defecto. Algunos de los parámetros editables que admite una escena son los siguientes:

Propiedad	**Valor por defecto**	**Descripción**
debug	true	Mostrar mallas de colisión
gravity	-9.8	Gravedad
friction	0.01	Fricción
restitution	0.3	Coeficiente de restitución

Cuadro 3.1: Valores por defecto del motor de físicas de *A-Frame*

3.6. Entorno *Networked A-frame*

Networked A-frame[8] es un marco construido sobre *A-frame* para escribir aplicaciones de realidad virtual multiusuario en *HTML* y *JavaScript*. Principalmente funciona sincronizando las entidades y sus componentes entre los usuarios conectados a la aplicación. Entre sus propiedades encontramos:

- **Sensible al ancho de banda.** Solo envía actualizaciones de red cuando la escena cambia.

- **Multiplataforma.** Funciona en todos los navegadores modernos de escritorio y móviles.

- **Extensible.** Cualquier componente *A-frame* puede ser sincronizado sin modificar el código.

- **Soporte para conexiones *WebRTC* y/o *WebSocket*.**

  Ambas son tecnologías clave para crear aplicaciones web modernas y de baja latencia. *WebRTC* (*Web Real-time Communication*) es un protocolo que permite la comunicación en tiempo real *peer-to-peer*[9]. El protocolo de transporte utilizado es UDP (*User Datagram Protocol*), ofreciendo un servicio de comunicación no orientado a conexión, aunque también puede ser utilizado sobre TCP (*Transfer Control Protocol*), protocolo de transporte orientado a conexión.

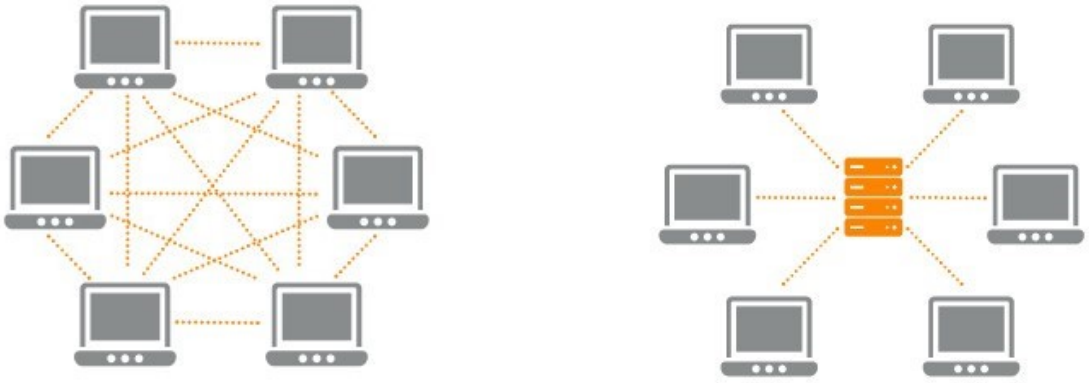
  En *Networked A-frame* el adaptador predeterminado es *wseasyrtc*, perteneciente a la familia *WebSocket*, debido a problemas de conexión inherentes con *WebRTC*. *WebSocket* es un protocolo de comunicación que ofrece una arquitectura cliente-servidor[10], que

[8] `https://github.com/networked-aframe/networked-aframe`

[9] No existe servidor central que controle la red, sino que todos los ordenadores estan conectados entre sí para compartir recursos.

[10] Existe servidor central que controla la red y una serie de dispositivos cliente que se conectan al servidor

hace uso del protocolo de transporte TCP.



(a) Modelo peer-to-peer (b) Modelo cliente-servidor

Figura 3.5: Protocolos de comunicación

	TCP	**UDP**
Velocidad	Más baja	Alta
Fiabilidad	Alta	Más baja
Método de transferencia	Secuencia	Flujo
Detección y corrección de errores	Sí	No
Control de congestión	Sí	No

Cuadro 3.2: Comparativa de los protocolos de comunicación

- **Incluye chat de audio.** Únicamente si el servidor utilizado es *WebRTC*.
- **Incluye transmisión de video.** Únicamente si el servidor utilizado es *WebRTC*.

Networked A-frame hace uso de dos servidores. Primero, el servidor web se ocupa de proporcionar la información de los sitios en formato *HTML*, donde se incluye texto, imágenes,

para realizar tareas específicas.

vídeos y todo tipo de datos. Mediante un explorador web los usuarios pueden visualizar todo esto en sus pantallas. Segundo, el servidor de comunicación es un sistema informático diseñado para manejar una amplia gama de aplicaciones basadas en comunicaciones. Estos dos servidores se conectan permitiendo que los navegadores que usan *Networked A-frame* se comuniquen entre sí sin interferencias.

El marco construido sobre *A-frame* puede ser utilizado con múltiples bibliotecas y servicios de red, siendo el adaptador de comunicaciones una clase que agrega soporte para una biblioteca que permite la comunicación entre navegadores. La configuración predeterminada puede ser suficiente para un proyecto pequeño, pero es importante tener una serie de consideraciones en cuenta, como pueden ser la cantidad de usuarios admitidos en una sala, la arquitectura deseada o la necesidad de transmisión de audio a la hora de escoger el adaptador utilizado en un proyecto.

Adaptador	**Librería utilizada**	**WebSockets o WebRTC**
wseasyrtc	open-easyrtc	WebSockets
easyrtc	open-easyrtc	WebRTC
janus	Janus WebRTC server janus-plugin-sfu	WebRTC

Cuadro 3.3: Adaptadores en funcionamiento

3.7. Simulador robótico *WebSim*

La plataforma *Kibotics*[11] es un entorno web para la docencia de robótica y de programación (*STEM*). Incluye un simulador robótico diseñado para el aprendizaje de conceptos básicos de programación. El entorno ejecuta diferentes ejercicios usando el simulador *Web-*

[11] `https://kibotics.org/`

Sim [16], el cual permite que los usuarios puedan programar de manera sencilla los distintos movimientos de los robots, simplemente han de acceder a la información que recogen sus sensores y enviar órdenes precisas a los actuadores del robot. Los actuadores simulados tienen la capacidad de generar dos tipos de movimiento; autónomo y teledirigido. Los principales sensores simulados son cuatro; sensor posición, sensor distancia, sensor cámara y sensor infrarrojo.

El simulador *WebSim* está diseñado basándose en el uso del entorno *A-frame*. Existen dos motores de físicas combinados, por un lado, el motor de físicas *Cannon.js* que se encarga de la gravedad, la fricción y los choques, propio de *A-frame*, y el motor de físicas complementario [15], propio de *WebSim*, que permite el movimiento autónomo de los robots según ordene el programa software desarrollado por el usuario de la plataforma.

Figura 3.6: Torneos de la plataforma *Kibotics* usando el simulador *WebSim*

Capítulo 4

Salas compartidas de realidad virtual con robots simulados

Este capítulo describe los siguientes puntos clave: la creación de un entorno básico *Networked A-frame*, la implementación del modelo maestro-esclavo, la actualización del motor de físicas en el simulador *WebSim*, la creación de mallas compuestas en el motor de físicas *Ammo.js* y la asimilación de los conceptos principales para combinarlos y dar lugar a un nuevo simulador denominado *MiniWebSim*, que ofrece una sala de realidad virtual compartida entre dos navegadores y en la que hay robots simulados.

4.1. Sala compartida de realidad virtual con objetos móviles distribuidos

El primer intento creado en busca de la implementación de escenas multijugador en el simulador *WebSim* se basaba en la transmisión de la escena 3D contenida en uno de los navegadores, a través de *flujo de vídeo WebRTC* a otro navegador [14]. El resultado obtenido fue un ancho de banda necesario excesivamente elevado. Por ello, en el presente trabajo se

busca construir un prototipo multijugador basado en *Networked A-frame* que necesite un menor ancho de banda.

4.1.1. Entorno básico

En el momento de la realización de este trabajo la última versión de *A-frame* es la 1.3.0, mientras que la actualización más reciente de *Networked A-frame* es la 0.11.0. Para la correcta sincronización de escenas es imprescindible la compatibilidad de las versiones utilizadas.

El servidor web encargado de servir la página en el navegador puede encontrarse en el repositorio de *Networked A-frame*. Se trata de un servidor `easyrtc-server` que se lanzará a través del terminal con el comando `node ./server/easyrtc-server.js`. En el mismo repositorio pueden encontrarse los adaptadores que permiten la comunicación entre navegadores, el desarrollador es la persona encargada de determinar qué adaptador concreto es el adecuado para su proyecto.

La construcción de las escenas en *Networked A-frame* difiere de la producción de las mismas en *A-frame*. La estructura básica utilizada en la definición de escenas en esta novedosa tecnología es la siguiente:

- **Archivo de texto con código *JavaScript*.** En el interior de la etiqueta `head` de un documento *HTML* pueden encontrarse rutas hacia archivos externos o recursos que el desarrollador desea enlazar al documento. La tecnología *Networked A-frame* necesita de las siguientes rutas para un correcto funcionamiento:

    1. **Dependencias *A-frame*.**[1] Ruta que señala a la de red de distribución de contenido (CDN). Los CDN son un conjunto de servidores ubicados en diferentes puntos de una red que contienen copias locales de ciertos contenidos que están almacenados en otros servidores, generalmente alejados geográficamente, que inlcuye el

[1] `https://aframe.io/releases/1.2.0/aframe.min.js`

código *JavaScript* de *A-frame*.

2. **Peticiones AJAX.**[2] Enlace a la red de distribución de contenido que incluye el código *JavaScript* que permite la realización de peticiónes AJAX (*Asynchronous Javascript and XML*). Se trata de una modalidad basada en que la petición *HTTP* se realiza desde *Javascript*, de forma transparente al usuario, descargando la información y pudiendo tratarla sin necesidad de mostrarla directamente en la página.

3. **Servidor de comunicación.**[3] Ruta que incluye el servidor de red del escenario.

4. **Dependencias *Networked A-frame.***[4] Enlace que incluye las diferentes dependencias de *Networked A-frame*.

- **Componente `networked-scene`.** Para que el usuario tenga la capacidad de conectarse a una escena en línea, debe añadirse el componente `networked-scene` en el interior de la etiqueta `scene`. Existen diversas propiedades en este componente con un valor por defecto, tal y como muestra el Cuadro 4.1.

[2] `https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.3.0/socket.io.slim.js`
[3] /easyrtc/easyrtc.js
[4] /dist/networked-aframe.js

Propiedad	**Valor por defecto**	**Descripción**
serverURL	/	Ubicación del servidor WebSocket/-WebRTC
app	default	Nombre de la aplicación
room	default	Nombre único de la habitación
connectOnLoad	true	Conexión automática al servidor
onConnect	onConnect	Llamada cuando se haya dado la conexión con el servidor
adapter	wseasyrtc	Servicio de red
audio	false	Audio activado/desactivado
video	false	Video activado/desactivado
debug	true	Depurador activado/desactivado

Cuadro 4.1: Propiedades del componente *networked-scene*

- **Definición de una entidad.** A la hora de definir una entidad se ha de crear en primer lugar una plantilla en el interior de la etiqueta `assets`, donde podrá encontrarse la definición de la misma mediante la determinación de propiedades como la geometría y el material, entre otras.

```
<a-assets>
  <template id="my-template">
    <a-entity>
      <a-sphere color="#f00"></a-sphere>
    </a-entity>
  </template>
</a-assets>
```

- **Creación de una entidad.** Una vez definida una entidad se ha de crear fuera de la plantilla. El componente `networked` permite la sincronización de la entidad entre navegadores web gracias a las siguientes propiedades; `template` es un selector *CSS* para una plantilla almacenada en la etiqueta `assets` y `persistence` permite que un usuario tome la posesión de las entidades persistentes tras la desconexión del creador, en lugar de eliminarlas.

```
<a-entity id="my"
  networked="template:#my-template;networkId:my;persistent:true;
  owner:scene">
</a-entity>
```

- **Esquema por plantilla.** Para una correcta sincronización entre navegadores, es necesario crear un esquema en el que se indican las propiedades sincronizables. Por defecto, se acompasan la posición y rotación de las entidades.

```
NAF.schemas.add({
  template: '#avatar-template',
  components: [
    'position',
    'rotation',
    'scale',
    {
      selector: '.hairs',
      component: 'show-child'
    },
    {
      selector: '.head',
      component: 'material',
```

```
        property: 'color'
      },
    ]
  });
```

La ejecución de los pasos anteriores dan lugar a una experiencia de realidad multiusuario ejecutándose en un navegador web, tal y como se muestra en la Figura 4.1, con dos objetos en la misma sala de realidad virtual, cada uno de ellos creado por un navegador diferente. Por el momento, las físicas no se encuentran activadas, únicamente se modifica la posición y la rotación de la entidad.

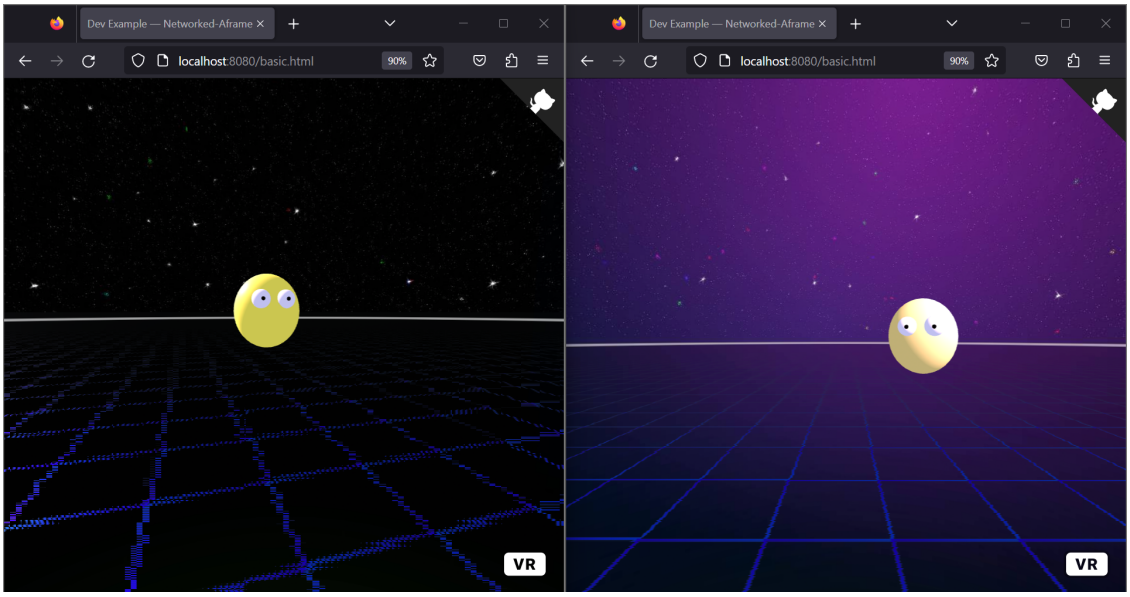


Figura 4.1: Escena multijugador basada en *Networked A-frame*

4.1.2. Estudios del motor de físicas Ammo

Partiendo del prototipo básico de *Networked A-frame* con objetos distribuidos en una sala compartida el siguiente paso natural era incorporar un motor de físicas a esa sala. El controlador *Ammo.js* proporciona muchas funcionalidades para que los objetos tengan movimientos realistas, por ejemplo, caídas provocadas por la fuerza de gravedad. Este controlador

no se trata de una dependencia del proyecto, por ende, se ha de incluir manualmente usando *npm* y *Webpack* o incluyendo una etiqueta en la cabecera del *HTML*, siendo esta última la manera más sencilla y rápida.

```
src="https://cdn.jsdelivr.net/gh/MozillaReality/ammo.js@8bbc0ea/builds/ammo
↪  .wasm.js"
src="https://cdn.jsdelivr.net/gh/n5ro/aframe-physics-system@v4.0.1/dist/afr
↪  ame-physics-system.min.js"
```

Para comenzar a usar el controlador *Ammo.js* en la sala compartida se ha de indicar, en la declaración del sistema de físicas, que el motor utilizado no es el otorgado por defecto en *A-frame*, sino `driver: ammo`. Los componentes del motor de físicas son *ammo-body*, *ammo-shape* y *ammo-constraint*. El primer componente otorga un cuerpo físico válido, pero solo tras añadir el segundo componente se darán colisiones con el resto de objetos. El tercer componente es utilizado para unir distintas entidades mediante bisagras, distancias fijas o puntos de unión fijos.

Estos componentes contienen diversas propiedades, con sus propios valores por defecto, que permiten crear la escena tal y como el desarrollador quiera.

El componente *ammo-body* tiene como propiedad más relevante *type*, entre las opciones a elegir pueden encontrarse:

- ***Dynamic.*** Los cuerpos dinámicos tienen masa, chocan con otros cuerpos, rebotan o se ralentizan durante las colisiones, es decir, los cuerpos se ven afectados por la física de la escena.

- ***Static.*** Otros cuerpos pueden chocar con cuerpos estáticos, pero los cuerpos estáticos en sí mismos no se ven afectados por las físicas ni las colisiones. Por lo general, estos cuerpos no deben moverse después de la inicialización.

- ***Kinematic.*** Similar a un cuerpo estático, excepto que se pueden mover actualizando la posición de la entidad. Además, aplican fuerzas sobre los cuerpos dinámicos cuando se mueven.

Por su parte, el componente *ammo-shape* tiene a *type* como la propiedad más relevante, pero en este caso otorga la forma de la malla de colisión.

- ***Primitivas.*** Las figuras destacadas son; recuadro, cilindro, esfera, cápsula y cono.

- ***Hull.*** Envuelve un modelo en un casco convexo, como una envoltura retráctil. No es tan eficaz como las primitivas, pero sigue siendo muy rápido.

- ***Hull Approximate Convex Decomposition.*** Característica experimental que genera múltiples cascos convexos para aproximar cualquier forma convexa o cóncava.

- ***Volumetric Hull Approximate Convex Decomposition.*** Característica experimental que genera múltiples cascos convexos para aproximar cualquier forma convexa o cóncava, pero con un algoritmo diferente.

- ***Mesh.*** Crea una forma de colisión cóncava 1:1 con los triángulos de las mallas de la entidad. Solo se puede usar en cuerpos estáticos. Esta es la forma de menor rendimiento, sin embargo, pueden funcionar muy bien para entornos estáticos.

- ***Heightfield.*** Similar a una primitiva, pero debe proporcionarse una matriz de alturas y la distancia entre los valores.

Finalmente, el componente *ammo-constraint* permite la creación de mallas de colisión que se ajustan lo máximo posible a los modelos 3D, aunque debe tenerse en cuenta que para un funcionamiento ideal, con dos entidades dinámicas, se ha de combinar con el concepto de filtrado de colisiones.

4.1.3. Configuración maestro-esclavo

Con respecto a la inclusión del motor de físicas en escenas multijugador se barajaron dos posibilidades. Por un lado, que ambos navegadores puedan otorgar físicas a las entidades y, por otro lado, que únicamente un navegador pueda hacerlo.

- **Modelo maestro-maestro.** Cada navegador otorga a su propia entidad las físicas. En este escenario ambos navegadores han de envíar y recibir datos de manera constante, provocando cortes en la fluidez de las escenas.

- **Modelo maestro-esclavo.** Únicamente uno de los navegadores, el maestro, puede otorgar a las entidades del escenario parámetros pertenecientes al motor de físicas. La selección de quién hace el papel de maestro puede realizarse de diversos modos; puede ser el primero que inicie la escena o el primero que emita un evento determinado. Que solo uno de los navegadores tenga esta capacidad da lugar a una visualización de la escena fluida y sin grandes latencias en ambos navegadores. El navegador esclavo recibe las actualizaciones de posición de los objetos y visualiza localmente la escena actualizada.

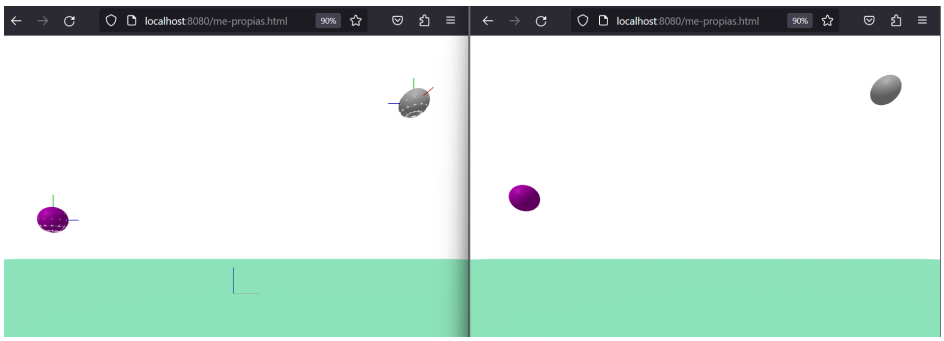


Figura 4.2: Modelo maestro-esclavo

Para determinar el rol de los navegadores se hace uso de dos variables, una variable compartida y una variable local. La selección es realizada a través de un evento de teclado, una vez el usuario pulse la tecla dedicada a la selección del papel del navegador

se comprobarán los valores de las variables. La variable compartida es inicializada como `undefined`, la variable local es inicializada como `null`. Una vez generado el evento de teclado programado para la selección del rol se comprobará el valor de las variables, si la variable compartida es igual a `undefined` su valor cambiará a `defined` y la variable local cambiará a `m`, por ende, cuando el siguiente usuario ejecute el evento de teclado `keydown` se observará que la variable compartida es igual a `defined`, mientras que la variable local es igual a `null`. Es importante comprobar el valor de la variable local para que el maestro no pase a ser esclavo.

```
switch (name){
case 'm':
    if (this.master.getAttribute('text').value == "undefined"){
        local = "m";
        this.master.setAttribute('text',{
            value: 'defined'
        });
        NAF.utils.takeOwnership(this.master);
    }else if (this.master.getAttribute('text').value
    != "undefined" && local == null){
        local = "s";
    }
break;
```

4.1.4. Validación experimental

El objetivo de la implementación del modelo maestro-esclavo busca evitar cortes en la escena e incongruencias en el sistema de físicas. Las físicas son ejecutadas únicamente en uno de los navegadores, el considerado maestro.

Para esta validación, han sido creados dos escenarios[5]: el primer escenario hace uso del modelo maestro-maestro, mientras que el segundo hace uso del modelo maestro-esclavo. Los modelos han sido observados en múltiples ocasiones, y se ha percibido que el primero presenta *lag* en la comunicación entre el servidor y el cliente, mientras que el segundo no. Por tanto, el uso del modelo maestro-esclavo es una mejor opción, más efectiva y eficiente para garantizar una experiencia de juego fluida y sin interrupciones.

4.2. Simulador robótico *WebSim*

El simulador *WebSim* [16] empleado en *Kibotics* hace uso del motor de físicas *Cannon.js* para materializar los movimientos de los cuerpos dinámicos de una escena incluyendo robots simulados. La principal ventaja del sistema de físicas *Cannon.js* es su completa disponibilidad en red y que su tamaño de archivo es considerablemente más pequeño que el de otros motores de físicas. Su principal desventaja es que se prevé que quede obsoleto en los próximos años y que el motor de físicas utilizado por defecto en *A-frame* pase a ser *Ammo.js*. El simulador *WebSim* también hace uso de un motor complementario [15] que se encarga de materializar las fuerzas autónomas de los robots de la escena y que coexiste con *Cannon.js*, ya que el sistema de físicas de *A-frame* se encarga solamente de materializar la fricción, la gravedad y las colisiones. Este apartado busca la introducción del nuevo motor de físicas en el simulador *WebSim*.

4.2.1. Paso de *Cannon.js* a *Ammo.js*

A-frame puede hacer uso de ambos sistemas de físicas, pero *Networked A-frame* solo consiente el empleo del motor de físicas *Ammo.js*. En el presente apartado se muestra la adaptación del código fuente del simulador *WebSim* para extender la implementación del motor complementario en conjunción con el futuro sistema de físicas por defecto, *Ammo.js*.

Dado que *A-frame* se desarrolla a partir de un archivo *HTML*, la escena y sus entidades

[5] `https://youtu.be/mD311D8y7ws`

pueden ser controladas utilizando *JavaScript* y *DOM API*. Es posible acceder a las propiedades a través de diversos métodos; `getAttribute(componentName)`, `body.componentName` y `object3D.componentName`. Únicamente el primer método es utilizable en ambos motores de físicas, ya que el segundo solo es válido en *Cannon.js* y el tercero sólo funciona en *Ammo.js*. Para un mejor acceso a las utilidades es recomendable acceder directamente al nivel *Three.js* vía *Object3D*. El simulador *WebSim* hace uso del segundo método, inválido en el nuevo motor. En consecuencia, es imprescindible modificar el código fuente del simulador e implementar el tercer método mencionado.

Cannon.js	**Ammo.js**
entity.body.componentName	entity.Object3D.componentName
entity.getAttribute(componentName)	entity.getAttribute(componentName)

Cuadro 4.2: Obtención de propiedades con Cannon.js y Ammo.js

El simulador *WebSim* posee cuatro sensores róboticos principales; distancia, cámara, infrarrojos y posición. El sensor distancia hace uso de la técnica *raycasting*, que consiste en extender una línea desde un punto de origen en una dirección específica y detectar si esa línea se cruza con otras entidades, permitiendo obtener la distancia entre el robot y la entidad con la que se produce la colisión. El sensor infrarrojo activa una cámara central del robot emulando un sensor infrarrojo mirando hacia el suelo. Y el sensor posición devuelve las coordenadas exactas en la que se encuentra el robot. De los cuatro sensores el único que ha sido modificado ha sido el de posición debido a que obtiene las coordenadas utilizando `entity.body.position` en lugar de `entity.object3D.position`.

```
getPosition() {
    let x = this.robot.object3D.position.x;
    let y = this.robot.object3D.position.y;
    let z = this.robot.object3D.position.z;
    let rot = THREE.Math.radToDeg(this.robot.object3D.rotation.y);
```

```
    return {x: x, y: y, z: z, theta: rot};
}
```

Una vez adquiridos los conocimientos necesarios para comprender el funcionamiento del simulador *WebSim* han sido localizados aquellos fragmentos de código que hacen uso del motor de físicas de *A-frame* y del motor complementario. El simulador en su despliegue local incluye los siguientes botones que hacen uso, de manera directa o indirecta, de ambos motores.

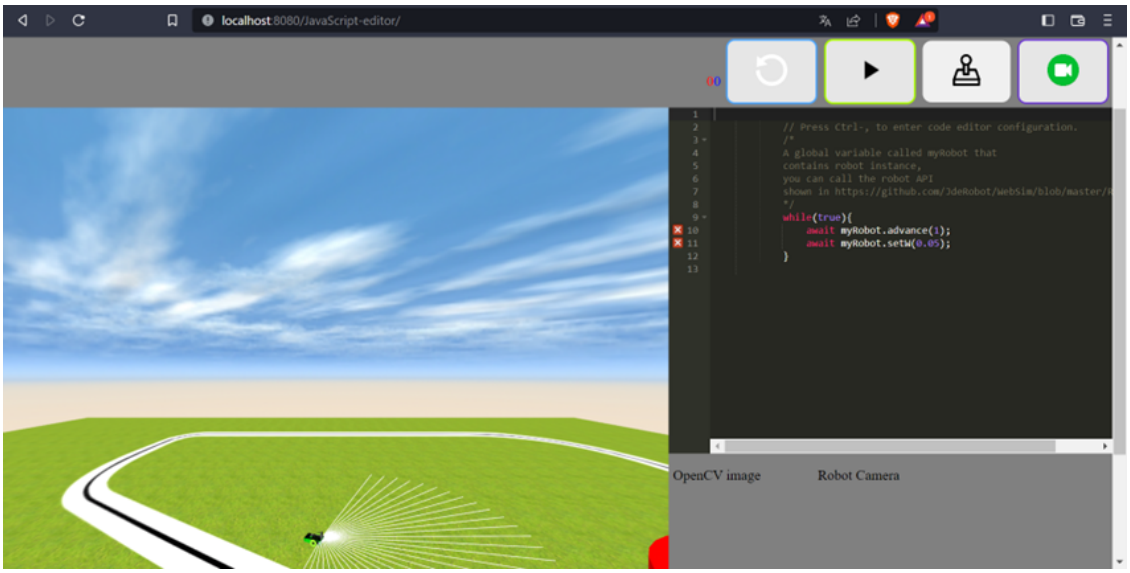


Figura 4.3: Simulador *WebSim* renderizado en un navegador

- **Botón Reset.** Restablece la posición inicial de las diferentes entidades.
- **Botón Play.** Invoca a la función programada por el usuario a través del editor, permitiendo el movimiento del robot.
- **Botón Teleoperator.** Habilita la capacidad del usuario de modificar la posición del robot por medio de eventos del teclado.
- **Botón Camera.** Inicia la retransmisión de la camara que porta el robot.

Reset es el botón más sencillo de acoplar al nuevo motor de físicas. En *Ammo.js* las entidades dinámicas no pueden ser trasladadas mediante cambios en el componente posición, puesto que al tratarse de entidades afectadas por las físicas esta acción es incoherente. Por ello, con anterioridad a la modificación de los componentes posición y rotación ha de paralizarse temporalmente la aplicación de las físicas en la entidad. Una vez sea posicionada en la ubicación original se le devuelve a la entidad su dinamismo.

```
async function resetElements() {
    arrayInitialPos.forEach((element) => {
        pauseSimulation();
        var el = document.getElementById(element.id);
        el.setAttribute('ammo-body', {
            type: 'kinematic',
        })
        el.object3D.position.set(element.position.x, element.position.y,
        ↪  element.position.z)
        el.object3D.rotation.set(element.rotation.x, element.rotation.y,
        ↪  element.rotation.z)
        setTimeout(() => {
            el.setAttribute('ammo-body', {
                type: 'dynamic',
            })
        }, 200)
        playSimulation();
    });
    arrayInitialPosObjects.forEach((element) => {
        pauseSimulation();
        var el = document.getElementById(element.id);
        el.setAttribute('ammo-body', {
```

```
            type: 'kinematic',
        })
        el.object3D.position.set(element.position.x, element.position.y,
        ↪  element.position.z)
        el.object3D.rotation.set(element.rotation.x, element.rotation.y,
        ↪  element.rotation.z)
        setTimeout(() => {
            el.setAttribute('ammo-body', {
                type: 'dynamic',
            })
        }, 200)
        playSimulation();
    });
await sleep(0.05);
}
```

Los botones *teleoperator* y *play* están altamente relacionados, puesto que ambos hacen uso de las mismas funciones a la hora de provocar movimiento en la entidad. Las funciones implicadas son; `setVelocity`, `auxiliaryPhysics` y los diferentes controladores. La función `auxiliaryPhysics` materializa el motor de físicas complementario, ejecutándose cada 20 milisegundos, y los controladores se encargan de la traducción de las velocidades deseadas, que le llegan al motor complementario en cada momento, a la fuerza autónoma que se aplicará al robot.

```
setVelocity() {
    var robot;
    if (this.robot.object3D.position.y > 1) {
        robot = document.querySelector("#" + this.myRobotID);
        robot.setAttribute('animation-mixer', "clip:*;timeScale:1.5");
```

```
    } else {
        robot = document.querySelector("#" + this.myRobotID);
        robot.setAttribute('animation-mixer', "clip:None");
    }
    this.timeoutMotors = setTimeout(this.setVelocity.bind(this), 50);
}
```

Ambos motores de físicas tienen su propia API (*Application Programming Interfaces*). Las APIs ofrecen recursos para aprender a trabajar con las físicas en *JavaScript*. La función de *WebSim*, `auxiliaryPhysics`, hace uso de `angularVelocity()` y `velocity()`, dos funciones pertenecientes a la API de *Cannon.js* que deben ser sustituidas por `setAngularVelocity()` y `setLinearVelocity()`, pertenecientes a la API de *Ammo.js*. El nuevo motor de físicas hace uso de *Emscripten*, un compilador denominado *source to source* que devuelve como salida un archivo *JavaScript* que puede procesarse en navegadores web, por ello, se han de usar tipos de datos nativos para poder hablar con él. Además, es apropiado crear un vector temporal `btVector3` para reducir la sobrecarga de asignación y recolección de basura. Las nuevas funciones implementadas son:

```
...
var tbv30 = new Ammo.btVector3();  tbv30.setValue(velocity.x(),
this.commandedVelocityY, velocity.z());
this.robot.body.setLinearVelocity(tbv30);
...
var tbv30 = new Ammo.btVector3();
tbv30.setValue(this.commandedVelocityXZ * Math.cos(rotation.y * Math.PI /
↪  180),velocity.y(),this.commandedVelocityXZ * Math.sin(-rotation.y *
↪  Math.PI / 180));
this.robot.body.setLinearVelocity(tbv30);
...
```

```
tbv30.setValue(0,this.commandedVelocityW,0);
this.robot.body.setAngularVelocity(tbv30);
...
```

En *WebSim* existen 4 controladores PD que se ejecután en función del tipo de robot, terrestreo o aéreo, y del movimiento que ejecute, lineal, angular, vuelo o suspensión en el aire. El drone hace uso de `controllerPDVerticalVel()`, calcula la fuerza que se debe ejercer para alcanzar la velocidad objetivo, y `controllerPDVerticalPos()`, calcula la fuerza que se debe ejercer el robot para mantener la posición de referencia. El robot terrestre hace uso de `controllerPDHorizontal()`, calcula la fuerza que se debe ejercer para alcanzar la velocidad objetivo, y `controllerPDAngular()`, calcula la fuerza que debe ejercer el robot para alcanzar la velocidad de giro objetivo. En estas funciones es necesario incluir los métodos propios del motor de físicas *Ammo.js* para obtener los componentes y modificar la función de la API utilizada en *Cannon.js* por `getAngularVelocity()`, propia de *Ammo.js*.

```
controllerPD() {
    ...
    this.currentErrorY = this.velocity.y - this.robot.object3D.velocity.y;
    ...
}
```

El componente de la gravedad también difiere en los diferentes motores de físicas. Por un lado, en *Cannon.js* se realiza una definición global de la gravedad, es decir, todas las entidades se ven afectadas por la gravedad definida en la etiqueta `scene`. Por otro lado, en el sistema de físicas *Ammo.js* cada entidad define su propia gravedad. Por ende, ha de eliminarse la definición global de la gravedad en el simulador *WebSim*.

```
export function parser(json) {
    return new Promise(async function (resolve, reject) {
        ...
```

```
        //scene.systems.physics.driver.world.gravity.y = ...
        //scene.systems.physics.driver.world.gravity.y = ...
    })
}
```

Por último, ha sido necesaria la creación de un nuevo componente. Una vez modificado el motor de físicas se ha observado que en caso de que las propiedades de las físicas no fueran dadas, el robot no obtenía sus valores por defecto. Además, no se le atribuían las propiedades que permiten a la entidad tener el comportamiento de un robot. Para la resolución de este problema se ha creado un nuevo componente en el que se incluyen los valores por defecto de las físicas de la entidad y se incluye a la entidad en `arrayLoadedBodyRobots`, otorgando de este modo a la entidad las propiedades y capacidades de una entidad considerada robot.

```
const gltfPhysicsObj = {
  schema: {
    id: {type: 'string', default: 'object'},
    model: {default: ''},
    body: {type: 'string', default: 'dynamic'},
    shape: {type: 'string', default: 'hull'},
    fit: {type: 'string', default: 'all'},
    halfExtents: {type: 'vec3', default: "0 0 0"},
    offset: {type: 'vec3', default: "0 0 0"},
    restitution: {type: 'float', default: "0.9"},
  },
  init() {
    const gltfmodel = document.getElementById(this.data.id)
    gltfmodel.setAttribute('gltf-model', this.data.model)
    gltfmodel.setAttribute('shadow', {receive: false})
    gltfmodel.setAttribute('ammo-body', {type: this.data.body})
```

```
    gltfmodel.addEventListener('model-loaded', () => {
      gltfmodel.setAttribute('ammo-shape',
      {         type: this.data.shape,
                fit: this.data.fit,
                halfExtents: this.data.halfExtents,
                offset: this.data.offset})
    });
    gltfmodel.addEventListener('body-loaded', () => {
      try {
        var sceneEl = document.querySelector('a-scene');

        ↪  sceneEl.querySelector("#"+toString(this.data.id)).body.setRestitution(this.d
        restitution);
        sceneEl.querySelector("#a-plane").body.setRestitution(0.8);
      } catch (e) {}
      var exists = arrayIds.includes(this.data.id);
      var exitRobot = arrayLoadedBodyRobots.includes(this.data.id);
      if (exists && !exitRobot) {
        var robotID = this.data.id;
        console.log("Body for robot with ID -->", robotID, "loaded.");
        arrayLoadedBodyRobots.push(robotID);
        arrayLoadedBodyRobots.sort();
      }
    })
  },
}
export {gltfPhysicsObj}
```

4.2.2. Mallas compuestas en el motor de físicas *Ammo.js*

El sistema de físicas *Ammo.js* define la malla de colisión de la entidad a través del componente *ammo-shape*, componente que puede ser añadido y eliminado en cualquier momento. Sin su presencia la entidad no tendrá la capacidad de colisionar con el resto de cuerpos que se encuentran en la escena. A la hora de definir una malla de colisión el tipo con mayor rendimiento computacional es la primitiva, pudiéndose adaptar a la figura de la malla existente en la entidad o seleccionando manualmente su tamaño.

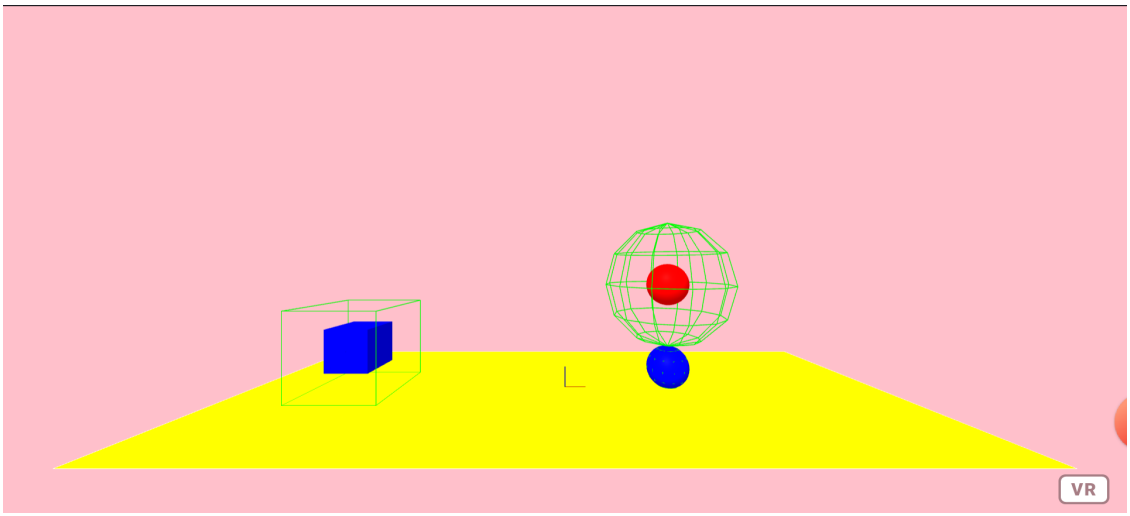


Figura 4.4: Malla de colisión automática y manual

Los modelos 3D importados desde otros programas, como por ejemplo *Blender*, no contienen mallas intrínsecas, por ello, es necesario indicar la figura y dimensión de la malla que define el espacio físico ocupado por el modelo gráfico, para ello es empleada la propiedad *halfExtents*. Mediante la definición manual del tamaño de las primitivas puede encapsularse un modelo 3D en el interior de una malla de colisión, aunque lo ideal es adaptar lo máximo posible la malla de colisión al modelo importado.

El método de *filtrado de colisiones* de *Ammo* permite la creación de mallas compuestas a partir de la combinación de primitivas. Para hacer uso del filtro de colisiones se han de tener en cuenta dos propiedades del componente *ammo-body*, *collisionFilterGroup* y *collisionFil-*

terMask. Para poner correctamente en funcionamiento este método es necesario tener un conocimiento básico de operaciones binarias, dado que dos de las tres operaciones binarias básicas son utilizadas. La operación AND realiza la función booleana de producto lógico, mientras que la puerta lógica OR realiza la operación de suma lógica.

El funcionamiento del método es el siguiente; la operación binaria OR es realizada entre las entidades involucradas en la colisión, utilizando como operandos sus respectivos *collisionFilterGroup*. El valor obtenido de la operación anterior es utilizado para realizar la operación binaria AND junto al valor del componente *collisionFilterMask*. Si el resultado de esta última operación es igual a cero no se dará interacción entre las entidades, es decir, no se producirá la colisión. En función del efecto deseado el programador ha de otorgar un valor u otro a estos dos componentes.

En la siguiente figura puede observarse un modelo 3D formado por diferentes primitivas, cada una con su propia malla de colisión.

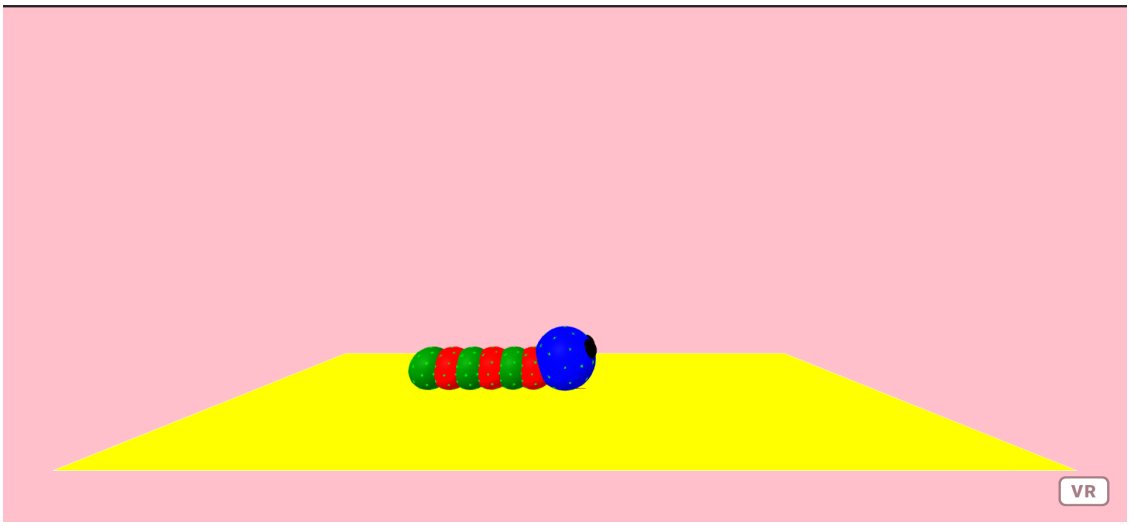


Figura 4.5: Entidades dinámicas bajo el efecto del filtrado de colisiones

Las mallas de colisión de las entidades que componen el gusano tienen un *collisionFilterGroup* y un *collisionFilterMask* igual a 1 y 2, respectivamente, mientras que el suelo

tiene asignados los valores 2 y 3. Las operaciones binarias hacen uso de únicamente números binarios, por tanto, los números impares actuarán como un 1 y los pares como un 0.

- **¿Chocarán las entidades que conforman al gusano?**

  En primer lugar, se ha de realizar la operación binaria OR entre los componentes *collisionFilterGroup* de las entidades involucradas en la colisión. El resultado obtenido es utilizado como uno de los operandos de la operación binaria AND junto a *collisionFilterMask*. Por tanto, el resultado es igual a 0 y no se dará la colisión entre las entidades.

```
collisionFilterGroup: 1
collisionFilterGroup: 1
Operacion binaria OR = 1 + 1 = 1
collisionFilterMask: 2
Operacion binaria AND = 1 * 0 = 0
```

- **¿Chocará el gusano con el suelo?**

  En primer lugar, se ha de realizar la operación binaria OR entre los componentes *collisionFilterGroup* de las entidades involucradas en la colisión. El resultado obtenido es utilizado como uno de los operandos de la operación binaria AND junto a *collisionFilterMask*. Por tanto, el resultado es igual a uno y se dará la colisión entre las entidades.

```
collisionFilterGroup: 1
collisionFilterGroup: 2
Operacion binaria OR = 1 + 0 = 1
collisionFilterMask: 3
Operacion binaria AND = 1 * 1 = 1
```

El filtrado de colisiones simplemente evita la colisión entre aquellas entidades cuyas operaciones lógicas den un resultado distinto de 1. En caso de que estas entidades se vean perturbardas por una entidad externa no se mantendrán unidas y se verán afectadas de manera diferente por las físicas del escenario. Existen dos maneras de que las entidades se mantengan unidas; definiendo únicamente a una de las entidades como dinámica o utilizando el componente *ammo-constraint*. Que solo una de las entidades se vea afectada por las físicas puede dar lugar a incoherencias, ya que las colisiones se verán materializadas tan solo cuando la entidad dinámica se vea afectada, mientras que al utilizar el componente del sistema de físicas *Ammo.js* las entidades se mantienen unidas mediante bisagras, distancias fijas o puntos de fijación independientemente de si son entidades dinámicas o estáticas.

(a) *Ammo-constraint* no incluido (b) *Ammo-constraint* incluido

Figura 4.6: Entidades dinámicas bajo el efecto del filtrado de colisiones

El concepto *parent-child* debe tenerse en cuenta también a la hora de crear mallas de colisión compuestas. Las entidades pueden tener una relación *parent-child* en la que la entidad hija recibe por herencia las propiedades de su entidad padre. Este concepto permite posicionar de forma sencilla entidades alrededor de aquella que es considerada padre, ya que como posición origen se heredan los parámetros de la entidad padre.

4.2.3. Validación experimental

Los cambios ejecutados en el código fuente del simulador *WebSim* han de ser imperceptibles para los usuarios, puesto que se busca modificar el software que simula y calcula el comportamiento físico de los objetos, manteniéndose el comportamiento del simulador. A pesar de que el nuevo motor de físicas no ofrezca cambios en las escenas a primera vista,

permite la posibilidad de crear experiencias de realidad virtual inmersivas y aumentar el rendimiento en escenarios con muchos objetos, creando experiencias de realidad virtual más complejas y detalladas sin comprometer el rendimiento. Puede observarse en el vídeo de validación experimental[6] cómo el uso de diferentes motores de físicas no afecta al funcionamiento básico del simulador.

Respecto a las mallas de colisión compuestas el resultado obtenido es satisfactorio. El objetivo principal es que estas mallas de colisión se adapten lo máximo posible a la figura del modelo 3D, importado o propio de *A-frame*, otorgando un mayor realismo a las escenas, evitando incoherencias en las físicas de la escena. El vídeo de validación experimental[7] muestra la importancia de la presencia de mallas de colisión compuestas, puede observarse cómo el movimiento de la pelota tiene un mayor realismo.

4.3. Sala de realidad virtual compartida con robots simulados

La plataforma *Kibotics* emplea un fichero *JSON* para configurar los diferentes escenarios *A-frame*. Este fichero de configuración es analizado para construir el mundo *A-frame* a través de *JavaScript* y *DOM API*. Integrarlo en la plataforma operativa *Kibotics* obligaba a mucho desarrollo no especifíco de las salas compartidas, puesto que las escenas multijugador no son definidas del mismo modo. Por tanto, se optó por hacer un prototipo fuera de la plataforma, pero que mantenga tecnologías integrables en ella. Este nuevo simulador denominado *MiniWebSim* combina lo aprendido en las dos sesiones anteriores.

[6] `https://www.youtube.com/watch?v=03swBq7aSx0`
[7] `https://www.youtube.com/watch?v=uGx3tKeLbQ4`

4.3.1. Escenario y robots con mallas de colisión compuesta

MiniWebSim contiene únicamente un escenario, el cúal es cargado automáticamente accediendo a la URL (*Uniform Resource Locators*) en la que se encuentra el recurso. La escena creada mediante *HTML* hace uso del motor de físicas *Ammo.js* y está constituida por dos robots, con sus respectivas entidades *raycaster*.

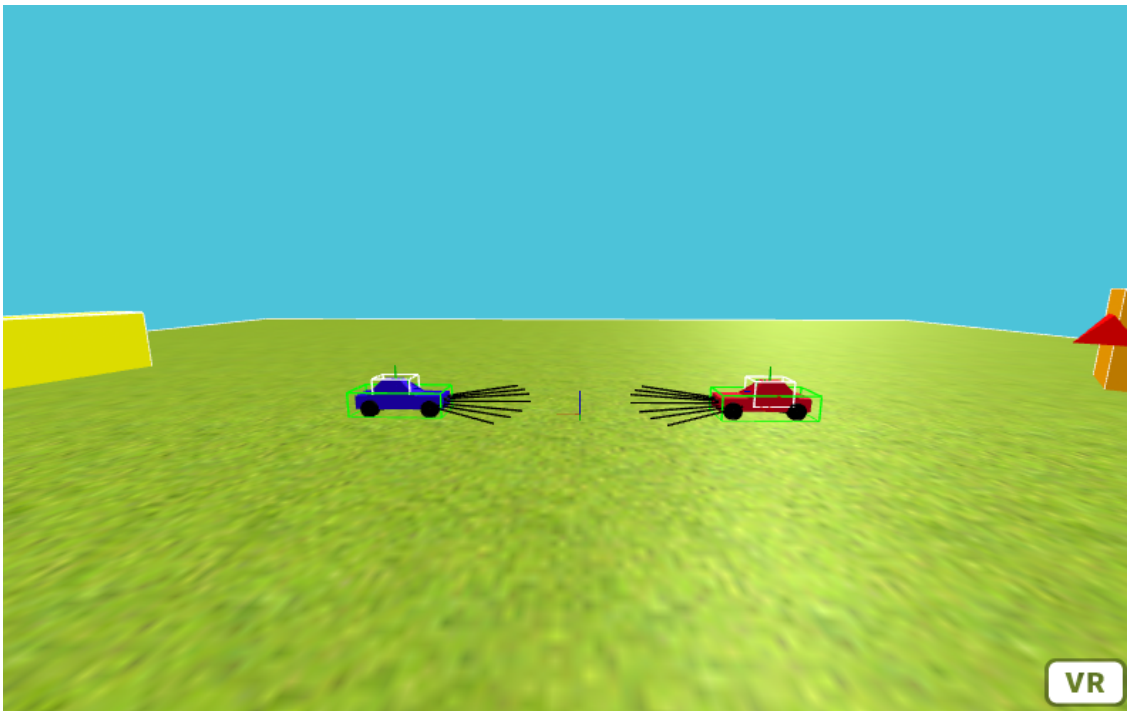


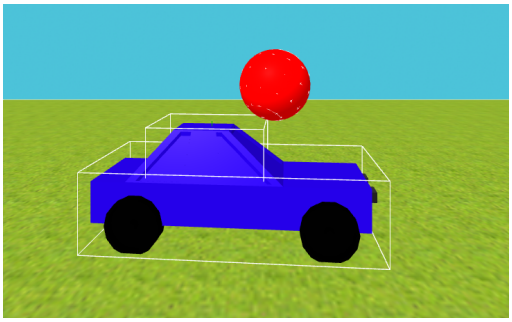
Figura 4.7: Escena MiniWebSim

El simulador *MiniWebSim* hace uso de modelos 3D importados en la escena que representan un automóvil. Computacionalmente son más eficientes las mallas de colisión basadas en una única primitiva, pero aportan un menor realismo a la escena, mientras que la combinación de primitivas ofrece un mayor ajuste de la malla de colisión en el modelo. Por ello, los modelos 3D creados para el nuevo simulador robótico hacen uso de mallas de colisión compuestas.

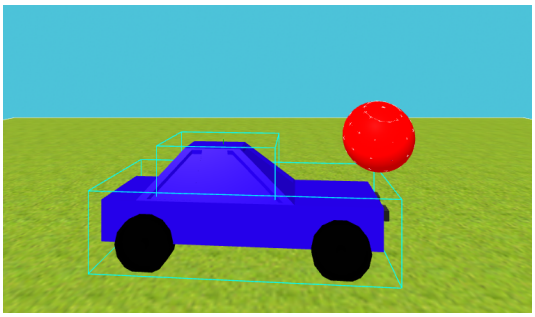
Ese modelo 3D de coche es dividido en dos entidades rectangulares. La primera entidad

envuelve la base, mientras que la segunda entidad guarda el techo del vehículo. La definición de las entidades se basa en el concepto de herencia, facilitando su definición. En el interior de la entidad base es definida la entidad techo, por ende, esta última hereda propiedades como la posición.

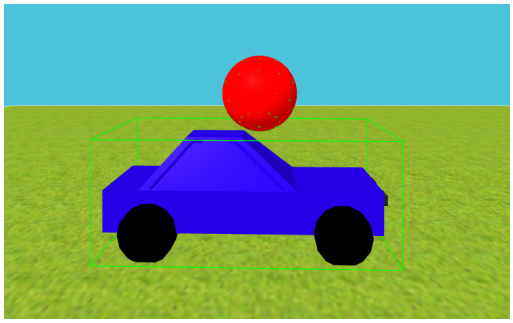
La malla compuesta de las entidades esta conformada por una entidad dinámica, la base, y una entidad estática, el techo.



(a) Malla de colisión compuesta



(b) Malla de colisión compuesta



(c) Malla de colisión simple

Figura 4.8: Modelo 3D con malla de colisión compuesta y simple

4.3.2. Sensores simulados

El simulador *MiniWebSim* adapta dos de los sensores encontrados en *WebSim*, el sensor posición y el sensor distancia. El sensor posición sigue la implementación encontrada en el simulador de referencia, mientras que el sensor distancia ha sido modificado. El sensor distancia está apoyado en el concepto *raycasting*, método basado en la extensión de una línea

desde un origen a una dirección determinada que permite verificar si esa línea se cruza con otras entidades. Como el robot tiene varios sensores de distancia en diferentes orientaciones la entidad robot es padre de diversas entidades *raycaster* que se encargan de determinar la distancia a la que se encuentra la entidad con la que su haz representativo ha colisionado.

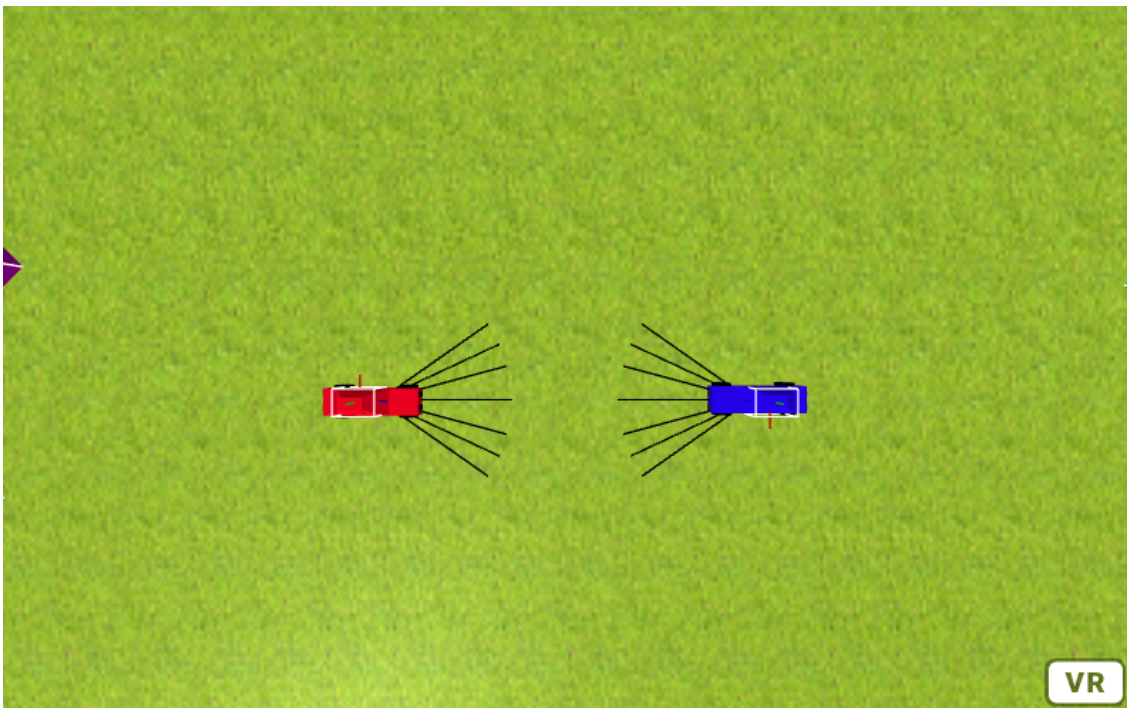


Figura 4.9: Entidades raycaster

Se ha registrado un componente *A-frame* encargado de captar el evento emitido por las entidades *raycaster* cuando un haz se topa con otra entidad. Los componentes registrados son propiedades que necesitan ser añadidas a la entidad para ser utilizadas. Por otro lado, se ha creado la función periódica `getDistance()` encargada de mostrar por pantalla la variable distancia definida en los componentes registrados.

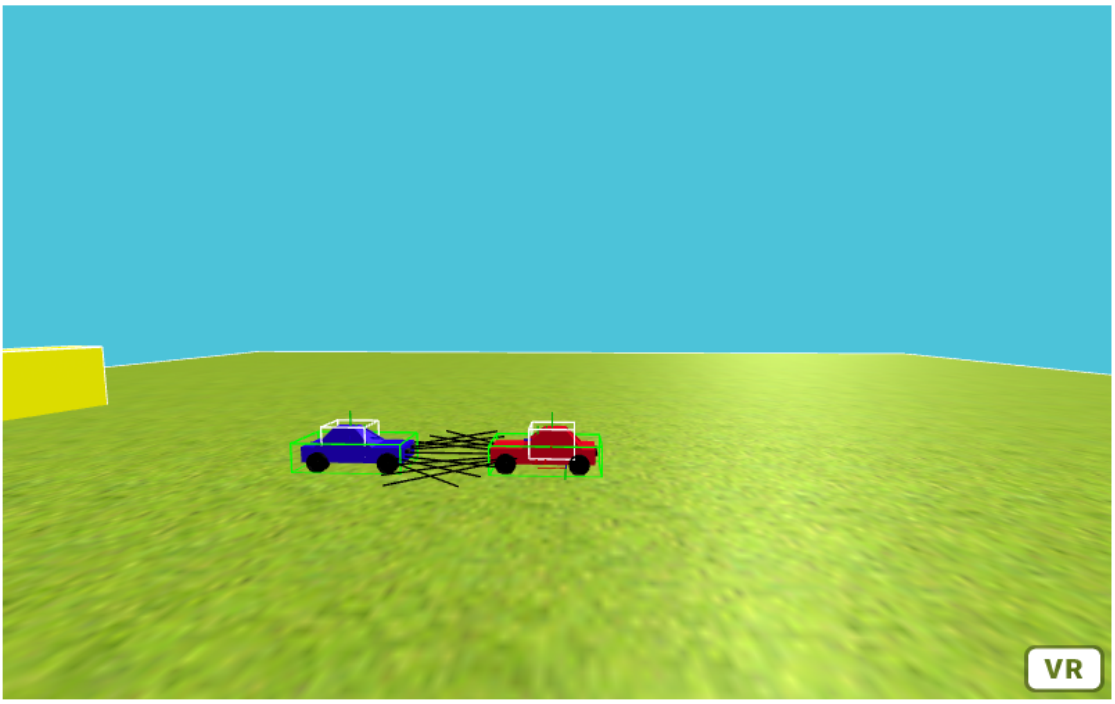


Figura 4.10: Valores obtenidos por los sensores

4.3.3. Motores eléctricos simulados

La movilidad de un robot en el simulador *MiniWebSim* se resume en dos tipos de movimientos, movimiento rectilíneo y movimiento angular. El movimiento rectilíneo es un movimiento cuya trayectoria es una línea recta. Si el móvil no cambia de sentido, la única variación que puede experimentar la velocidad es la de su módulo. El movimiento angular es un movimiento de rotación en torno a un eje o un punto de referencia.

El movimiento rectilíneo es provocado por la fuerza que actúa sobre los cuerpos. En *Ammo.js* existen dos fuerzas aplicables sobre las entidades de una escena, la fuerza y el impulso. La principal diferencia es que una fuerza se aplica a un cuerpo durante un período

de tiempo, mientras que un impulso es una fuerza que se aplica en un instante. El sistema de físicas utilizado ofrece en su API funciones relacionadas con la fuerza y el impulso.

- `void applyImpulse()`
- `void applyCentralImpulse()`
- `void applyForce()`
- `void applyCentralForce()`
- `void applyCentralLocalForce()`

Estas funciones son válidas para provocar un movimiento rectilíneo sobre las entidades, su selección depende completamente del efecto buscado por el programador. El impulso[8] es ideal para movimientos extremadamente rápidos, como podría ser el movimiento de una bala, mientras que la aplicación de una fuerza[9] constante es ideal para el movimiento del robot. Además, es importante generar una fuerza o impulso sobre la propia entidad, es decir, se han de usar referencias relativas a los robots. Por todo lo anterior, el movimiento rectilíneo de los robots encontrados en la escena es provocado por `void applyCentralLocalForce()`. Por otro lado, el movimiento angular es definido por la velocidad angular. La API del motor de físicas ofrece la función `void setAngularVelocity()` que ofrece el movimiento de rotación buscado.

Los actuadores del simulador *MiniWebSim* son puestos en marcha a través de los siguientes eventos `keydown`[10] que se generan al pulsar la tecla correspondiente:

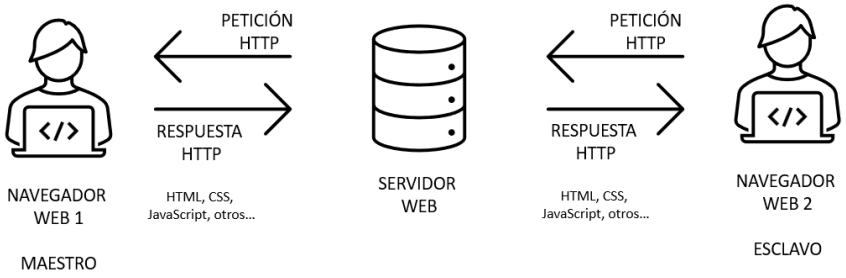
- **l.** Avance.

[8] `https://www.youtube.com/watch?v=bwFibCdM9Bc`
[9] `https://youtu.be/sqiDsOJo30w`
[10] `https://www.youtube.com/watch?v=TSXV7yFQwEk`

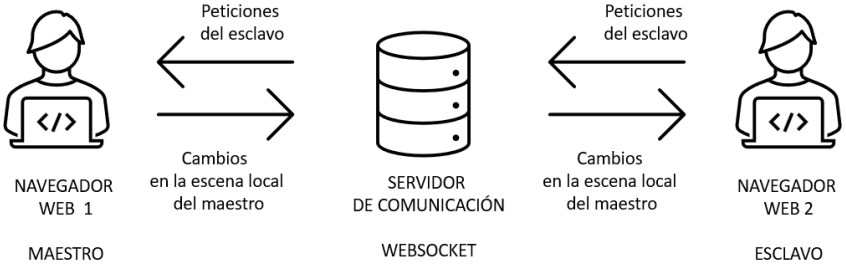
- **j.** Retroceso.
- **k.** Giro derecha.
- **i.** Giro izquierda.
- **g.** Iniciación movimiento autónomo.
- **p.** Finalización movimiento autónomo.

4.3.4. Configurar maestro-esclavo de la sala compartida

Para la visualización fluida de escenas, el simulador *MiniWebSim* hace uso del modelo maestro-esclavo, aprovechándose lo aprendido y descrito en la sección 4.1. Únicamente uno de los navegadores, el maestro, ejecutará las físicas de la escena.



(a) Servidor web que proporciona la página con la escena



(b) Servidor de comunicaciones que hace de intermediario entre los navegadores

Figura 4.11: Arquitectura *MiniWebSim*

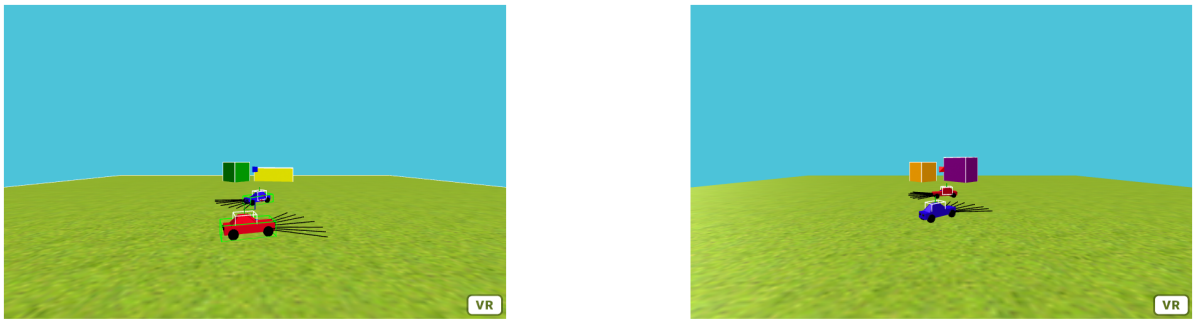
El rol del usuario se determina a través del evento de teclado `keydown 'm'`, haciéndose uso de una entidad de texto compartida y una variable local. El papel de maestro o esclavo es obtenido por los navegadores en base al contenido en la entidad de texto y la variable local.

Rol	**Entidad de texto**	**Variable local**
Maestro	undefined	null
Esclavo	defined	null

Cuadro 4.3: Determinación de roles maestro-esclavo

En el momento en el que el rol de maestro es adquirido por un navegador, los componentes del sistema de físicas *Ammo.js* son inicializados. Después está pendiente de si se pulsa alguna tecla para mover su robot, llamémosle robot 1, en la sala compartida. Las físicas de una escena no deben ser determinadas durante la creación de la escena en el *HTML*, puesto que de este modo ninguno de los navegadores adquiere el control de la escena.

El navegador maestro ejecuta periódicamente la función `getSlave()`, obteniendo los eventos `keydown` producidos por el navegador esclavo mediante pulsaciones en su teclado asociado, para luego provocar el movimiento indicado para el robot 2.



(a) Visión maestro

(b) Visión esclavo

Figura 4.12: Escena *MiniWebSim*

4.3.5. Validación experimental

La base del nuevo simulador *MiniWebSim* es la sala de realidad virtual compartida basada en *Networked A-frame* desarrollada en el apartado 4.1. A esta sala se le ha añadido el motor de físicas *Ammo.js* y robots simulados rodeados de mallas de colisión compuestas, con sus propios sensores y actuadores ejecutados en el navegador maestro.

El simulador robótico hace uso de dos sensores; sensor posición y sensor distancia. El primer sensor determina la ubicación global del robot, mientras que el sensor distancia mide la distancia con el robot rival. Los sensores programados proporcionan buenos resultados, puesto que cumplen su función de manera precisa y confiable.

(a) Navegador maestro (b) Navegador maestro

Figura 4.13: Sensores

El simulador *MiniWebSim* ofrece cuatro movimientos; giro a la izquierda, giro a la derecha, línea recta hacia delante, línea recta hacia atrás. Todos estos movimientos son ejecutados en el navegador maestro, permitiendo que los usuarios se encuentren en igualdad de condiciones. Además, los movimientos son llevados a cabo de manera satisfactoria, tal y como puede observarse en el vídeo presentación del nuevo simulador robótico[11].

[11] `https://www.youtube.com/watch?v=IdKiL7mbAgM`

(a) Navegador maestro

(b) Navegador maestro

Figura 4.14: Actuadores

4.3.6. Distintos escenarios de red

Los experimentos realizados se dividen en dos vertientes, por un lado, aquellos en los que el servidor utiliza una IP pública y, por otro lado, aquellos en los que el servidor hace uso de un IP privada.

Dirección IP pública	**Dirección IP privada**
Alcance externo (global)	Alcance interno (local)
Se utiliza para comunicarse por Internet, fuera de su red privada	Se utiliza para comunicarse con otros dispositivos de la casa u oficina, dentro de su red privada
Un código numérico exclusivo que nunca se reutiliza para otros dispositivos	Un código numérico no exclusivo que pueden reutilizar otros dispositivos de otras redes privadas
La asigna y controla su proveedor de servicios de Internet	Se asigna a su dispositivo específico dentro de una red privada
No es gratuita	Gratuita

Cuadro 4.4: Resumen de las diferencias entre direcciones IP privadas y públicas

Las direcciones IP públicas se utilizan al interactuar con Internet, mientras que las pri-

vadas operan con una red local. Dentro de esa red, es probable que haya varios dispositivos diferentes. El router asigna a cada uno una dirección IP privada exclusiva, para así poder enviar datos al dispositivo concreto que los solicita. Los dispositivos en la misma red utilizan direcciones IP privadas para comunicarse directamente.

Si el servidor tiene una IP pública los usuarios podrán acceder a la escena independientemente de la red en la que se encuentren, mientras que si su IP es privada únicamente podrán acceder aquellos usuarios que se encuentran en la misma subred, ya que el resto no tienen la capacidad de localizar el dispositivo. A su vez, se han dividido en aquellos que hacen uso de un adaptador *WebSocket* y los que usan un adaptador *WebRTC*.

En primer lugar, independientemente del adaptador utilizado el resultado es similar, no se perciben cambios en el funcionamiento ni en el rendimiento más allá de la aparición de un mensaje de error fuera del canvas de la escena cuando se hace uso de *WebRTC*. En segundo lugar, es destacable la mejora que se da en la fluidez en el esclavo, independientemente de su IP, cuando el maestro contiene una IP pública. Por tanto, el resultado mejora cuando el servidor y el maestro tienen una IP pública, en el resto de casos la fluidez es ligeramente inferior en el esclavo, aunque sigue siendo aceptable.

***Websocket* - Servidor público**

Maestro	**Esclavo**	**Conexión**	**Calidad**	**Latencia**
Privado	Privado	Apta	Media	Media
Público	Público	Apta	Alta	Baja
Público	Privado	Apta	Alta	Baja
Privado	Público	Apta	Media	Media

Cuadro 4.5: Tabla de resultados red pública con adaptador WebSocket

***WebRTC* - Servidor público**

Maestro	**Esclavo**	**Conexión**	**Calidad**	**Latencia**
Privado	Privado	Apta	Media	Media
Público	Público	Apta	Alta	Baja
Público	Privado	Apta	Alta	Baja
Privado	Público	Apta	Media	Media

Cuadro 4.6: Tabla de resultados red pública adaptador WebRTC

***WebSocket* - Servidor privado**

Maestro	**Esclavo**	**Conexión**	**Calidad**	**Latencia**
Privado	Privado	Apta	Media	Media
Público	Público	No apta	-	-
Público	Privado	No apta	-	-
Privado	Público	No apta	-	-

Cuadro 4.7: Tabla de resultados red privada adaptador WebSocket

***WebRTC* - Servidor privado**

Maestro	**Esclavo**	**Conexión**	**Calidad**	**Latencia**
Privado	Privado	Apta	Media	Media
Público	Público	No apta	-	-
Público	Privado	No apta	-	-
Privado	Público	No apta	-	-

Cuadro 4.8: Tabla de resultados red privada adaptador WebRTC

4.3.7. Consumo de ancho de banda

En este apartado se realiza un análisis del consumo de ancho de banda en el navegador maestro de los dos sistemas de salas compartidas; En primer lugar, el sistema en el que se realiza la compartición de la escena a través de la compartición de pantalla mediante la transmisión de flujo de vídeo *WebRTC* [14], basado en el Trabajo Fin de Máster de Pablo Moreno Vera[12] [14]. Y en segundo lugar, el prototipo creado en el presente trabajo basado en la tecnología *Networked A-frame*.

Para medir el volumen de datos intercambiados en ambos sistemas ha sido utilizada la aplicación *WireShark*[13], un software de análisis de protocolos de red que permite capturar y analizar el tráfico de red en tiempo real. Los paquetes *Networked A-frame* son localizados mediante la IP del servidor público que ofrece la escena, mientras que los paquetes de flujo de video son localizados mediante la filtración de paquetes multimedia.

Experimento	**Duración**	**Bytes totales**	**Ancho de banda**
Flujo de vídeo *WebRTC* Videoconferencia	46 s	18 MB	3.27 Mbps
Flujo de vídeo *WebRTC* Compartición de pantalla	55 s	10 MB	1,45455 Mbps
Networked A-frame	52 s	818 KB	0,12585 Mbps

Cuadro 4.9: Tabla de resultados del análisis del ancho de banda

Por un lado, se observa en el Cuadro 4.9 que el ancho de banda durante la transmisión de flujo de vídeo al compartir pantalla es 2,22 veces menor al de una videoconferencia, esto es debido a que los ratios de compresión en las videoconferencias son mucho menores. Por otro lado, se verifica que la aplicación basada en transmisión de flujo de vídeo consume

[12] `https://gsyc.urjc.es/jmplaza/students/tfm-kibotics-torneos-pablo_moreno-2020.pdf`
[13] `https://www.wireshark.org/`

aproximadamente 11 veces más ancho de banda en términos de datos descargados y cargados que la aplicación basada en *Networked A-frame*. Puede observarse cómo *Networked A-frame* es sensible al ancho de banda y que únicamente se transmiten los cambios de posición en la escena, por tanto, su ancho de banda depende de la cantidad de cambios de posición, siendo las posiciones de los objetos de la escena datos poco voluminosos.

Por lo tanto, en caso de disponer de un ancho de banda limitado, el primer sistema tendría un impacto mucho mayor en la capacidad de la red para manejar el tráfico, pudiendo afectar a la calidad del servicio o incluso impedir su uso para otras aplicaciones o usuarios que compartan la misma red. En cambio, el sistema desarrollado en este TFG consume significativamente menos ancho de banda y puede ser más adecuado para entornos con recursos de red limitados.

Capítulo 5

Conclusiones

En este capítulo se recogen las conclusiones y valoraciones finales a las que se ha llegado tras la realización de este Trabajo Fin de Grado. De igual forma, se expondrán algunas líneas futuras para mejorar y completar la implementación realizada.

5.1. Balance global y valoración del resultado

El objetivo principal del presente Trabajo Fin de Grado era la experimentación con salas compartidas de realidad virtual que incluyen robots simulados. Esta misión global se ha dividido en tres subobjetivos. Los tres se han conseguido satisfactoriamente.

El primer subobjetivo era la implementación de una sala virtual compartida con objetos distribuidos, no robots, que se ven afectados por físicas realistas. Este subobjetivo ha sido alcanzado satisfactoriamente, tal y como se ha descrito en el apartado 4.1.3 gracias a la tecnología *Network A-frame*. Las físicas realistas se aplican en uno de los navegadores debido a la puesta en funcionamiento del modelo maestro-esclavo. Esta sala es vista simultáneamente desde dos navegadores web separados.

El segundo subobjetivo era la renovación del motor de físicas del simulador robótico *WebSim*, pasando de *Cannon.js* a *Ammo.js*. Este subobjetivo ha sido alcanzado con éxito, tal y como se describe en el apartado 4.2, tras realizar un estudio exhaustivo del motor de físicas por defecto de *A-frame* y del nuevo motor que se quiere implementar. También han sido construidas mallas de colisión compuestas que se ajustan mejor a los objetos de geometría difícil que las actuales de *Cannon.js*.

Por último, el tercer subobjetivo era la combinación de los dos desarrollos anteriores, es decir, crear una sala compartida de realidad virtual basada en *Networked A-frame*, en la que hay robots con sensores y actuadores simulados, con modelos 3D rodeados por mallas compuestas y que cumplan el modelo maestro-esclavo. Ha sido alcanzado con éxito, tal y como se describe en el apartado 4.3.

Este prototipo operativo abre la puerta a juegos distribuidos, entre dos participantes remotos, con robots simulados en igualdad de condiciones, puesto que el movimiento de los dos robots se produce en un navegador, el maestro, y la misma escena puede visualizarse en un segundo navegador al mismo tiempo. El sistema se ha validado experimentalmente en varias condiciones de conectividad con resultados satisfactorios. Las comunicaciones para sincronizar los dos navegadores intercambian de modo continuo los cambios en las posiciones de los objetos de la sala compartida. Este planteamiento es mucho más eficiente que el establecimiento de un canal de vídeo que comparta el canvas que emite uno de los navegadores, tal y como se indica en el apartado 4.3.7 se ahorra un 90 % de ancho de banda, haciendo este sistema usable en escenarios de conectividad doméstica.

También se ha alcanzado el objetivo personal de una mejora de conocimientos en el campo del desarrollo web. Gracias a ello, se han adquirido conocimientos en tecnologías tan diversas como *JavaScript*, *HTML*, *CSS*, *A-frame*, *Networked A-frame*, *WebSocket*, *WebRTC*, *Node.js*, *Blender*, entre otros.

5.2. Líneas futuras

La implementación del nuevo simulador *MiniWebSim*, que permite poner en marcha escenas multijugador efectivas, ha supuesto un gran avance en términos de partidas online. No obstante, aún existen múltiples líneas abiertas que ayudarán a mejorar aún más el sistema.

- **Robots programables.** Los robots simulados del prototipo conseguido estan básicamente teleoperados localmente por teclado. Para hacer más atractivas las partidas multijugador es interesante incorporar un editor que permita a los usuarios programar los movimientos autónomos y velocidades del robot que le corresponda.

- **Chat de audio.** La adición de un chat de audio en vivo en videojuegos multijugador da lugar a un mayor dinamismo, ya que la comunicación es directa e implica una mayor diversión. *Networked A-frame* también ofrece la capacidad de transmitir audio en función del adaptador utilizado, también podría tenerse en cuenta transmitir audio usando *WebRTC*.

- **Sonido compartido.** El sonido en videojuegos es esencial para que el usuario tenga una experiencia inmersiva. Diseñar engloba tener en cuenta la banda sonora de un título, con sus diferentes temas, y toda clase de sonidos encontrados en un videojuego, desde el dado en la aceleración de un coche, hasta el sonido de una colisión.

- **Exploración de la entidad *raycaster* en NAF.** El componente *raycaster* ha sido utilizado para materializar el sensor distancia en el presente trabajo. Se trata de un rayo que se emite desde una entidad y detecta los objetos que están en su línea de visión. A pesar de que la propiedad distancia funciona correctamente su visualización vía *Networked A-frame* falla y los rayos penetran dentro de los objetos gráficos.

- **Modificación del giro del robot.** La ejecución actual del giro es ejercida desde el centro de masas de la entidad de *A-frame*, para aumentar el realismo en la escena podría ejercerse un giro desde las ruedas frontales del robot y no desde el centro de

masas.

Bibliografía

[1] HTTP. Documentación oficial de HTTP. https://developer.mozilla.org/es/docs/Web/HTTP/Methods.

[2] Blog del TFG. https://github.com/RoboticsLabURJC/2022-tfg-ana-villanueva/tree/main/docs/.

[3] JavaScript. Documentación oficial de JavaScript. https://developer.mozilla.org/es/docs/Web/JavaScript.

[4] HTML. Documentación oficial de HTML. https://developer.mozilla.org/en-US/docs/web/html.

[5] JSON. Documentación oficial de Json. https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/JSON.

[6] Blender. Documentación oficial de Blender. https://www.blender.org/.

[7] A-frame. Documentación oficial de A-frame. https://aframe.io/.

[8] Networked A-frame. Documentación oficial de Networked A-frame. https://github.com/networked-aframe/networked-aframe.

[9] Cannon. Documentación oficial de Cannon.js. https://github.com/n5ro/

aframe-physics-system.

[10] Ammo. Documentación oficial de Ammo.js. https://github.com/n5ro/aframe-physics-system/blob/master/AmmoDriver.md.

[11] Kibotics. Documentación oficial de Kibotics. https://kibotics.org/.

[12] Canal de Youtube oficial. https://www.youtube.com/@anea2415.

[13] WireShark. Documentación oficial de WireShark. https://www.wireshark.org/.

[14] Pablo Moreno Vera. Torneos de programación de robots en una plataforma online (2020). https://gsyc.urjc.es/jmplaza/students/tfm-kibotics-torneos-pablo_moreno-2020.pdf.

[15] Natalia Monforte Rodríguez. Motor de físicas mejorado para simulador robótico basado en tecnologías web (2020). https://gsyc.urjc.es/jmplaza/students/tfg-kibotics-motor_fisicas-natalia_monforte-2020.pdf.

[16] Ruben Álvarez Martín. Mejoras en entorno de robotica educativa para niños (2019). https://gsyc.urjc.es/jmplaza/students/tfg-kibotics-ruben_alvarez-2019.pdf.