



Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

Asistente Virtual mediante Realidad Aumentada

Autor:
Miguel Bayón Sanz



Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

Asistente Virtual mediante Realidad Aumentada

Autor:
Miguel Bayón Sanz

Tutor:
Irene Lavín Perrino

Resumen

Mediante este proyecto, se presenta la oportunidad al usuario de atender a una pequeña charla impartida por un asistente virtual dotado de forma humana que, aplicando la Realidad Aumentada, se ubicará en el mundo real a través de las imágenes captadas por la cámara del dispositivo móvil. De esta manera, y combinando varias tecnologías de software libre, siendo la interfaz *WebXR* la más destacable, se logrará generar una experiencia inmersiva para el usuario que logre captar su atención y le ofrezca una presentación de la web y de la empresa para la que se ha generado distinta a lo que puede encontrar en otros lugares, siendo necesario únicamente un dispositivo *Android* y el navegador *Google Chrome* para los dispositivos de dicho sistema operativo.

Abstract

Through this project, the user will have the opportunity to attend a short talk given by a virtual assistant with a human form that, applying Augmented Reality, will be located in the real world through the images captured by the camera of the mobile device. In this way, and combining several free software technologies, being the *WebXR* interface the most remarkable one, it will be possible to generate an immersive experience for the user that will capture his attention and will offer him a presentation of the web and the company for which it has been generated different from what he can find in other places, being necessary only an *Android* device and the *Google Chrome* browser for the devices of that operating system.

Índice general

Resumen	1
1. Introducción	5
1.1. Motivación	5
1.2. Objetivos	6
1.3. Metodología	7
1.4. Conceptos de Realidad Aumentada	9
1.5. Tecnologías utilizadas	12
2. Sistema básico	13
2.1. Estructura	13
2.2. Sesión WebXR	15
2.3. Procesado iterativo	19
3. Implementaciones sobre el sistema de Realidad Aumentada	23
3.1. Modelo en 3D y Three.js	23
3.2. Cálculo de superficies, <i>Hit Test Results</i>	33
3.3. Animación de modelos	37
3.4. Sonido espacial	41
3.5. Funcionamiento final de la Realidad Aumentada	44
4. Diseño y modelado en 3D	49
4.1. Creación mediante <i>MakeHuman</i>	50
4.2. Edición con <i>Blender</i>	54
5. Implementación y puesta en marcha	59
5.1. Alojamiento y ejecución	59
5.2. Pruebas de campo	60
5.3. Problemas y errores	61
6. Consideraciones y futuros pasos	63
6.1. Pasos fallidos	63
6.2. Futuros pasos	64

Capítulo 1

Introducción

1.1. Motivación

El *Trabajo de Fin de Grado* supone el último paso para obtener la titulación cursada, y se muestra de manera muy intimidante al alumno. Por mucho que otros compañeros de años anteriores intenten suavizar su dificultad o la necesidad de hacer algo grande, uno no deja de verlo como la culminación de varios años de aprendizaje en un estudio superior. Por eso, fue difícil elegir algo que realizar.

Durante varios años, estuve dispuesto a llevar mi propio proyecto con varias ideas que fui reservando para más adelante, e incluso fueron varias las revisiones que hice a las ideas preparadas para distintos trabajos por distintos profesores. Sin embargo, a medida que se iba acercando la fecha de comenzar este proyecto, iba creciendo la necesidad de comenzar a trabajar y tener mis propios ingresos. Así, y teniendo el *Trabajo de Fin de Grado* en mente, comencé a buscar empleo planteando a las empresas que me concedían entrevista dos condiciones: media jornada para poder terminar mis estudios y que me ofrecieran un proyecto válido para un *TFG* para poder terminar mi carrera.

Fue así como entré en *SilverStorm*, ahora *Thirdera* después de su adquisición por parte de esta última. Ellos me concedieron la flexibilidad necesaria para compaginar empleo y estudios, así como facilidades a la hora de realizar los exámenes de la Universidad. Además, al entrar en el equipo de innovación, sabía que me podrían ofrecer un proyecto que se distinguiere de los que ofrecían al resto de estudiantes que trabajaban en la empresa.

Tras un tiempo, y después de comentarles que quería comenzar a desarrollar mi *Trabajo de Fin de Grado*, me ofrecieron el proyecto que se explicará en esta memoria. Por aquel entonces, la página web de *SilverStorm* desarrollada en *WordPress* contaba con un estilo sencillo y familiar para que los usuarios que entrasen conociesen la empresa y pudieran obtener la información que desearan de ella: a qué nos dedicamos, cómo y con qué trabajamos, qué clientes tenemos o hemos tenido, de qué ofertas de trabajo disponemos, etc. No obstante, esta no lograba destacar de otras webs corporativas: sus animaciones eran aquellas que ofrecía *WordPress*, los dibujos utilizados eran de una colección de imágenes de stock retocadas y su sencillez implicaba una cierta monotonía.

La idea del proyecto era añadir un pequeño detalle a esta página que hiciera que esta destacara un poco más. La propuesta consistía en generar un pequeño avatar que pudiese contar, en cada una de las secciones, lo que el usuario podía encontrar. Este avatar aparecería en el plano real mediante *Realidad Aumentada*, una tecnología en pleno auge [2] que reforzaría la idea de que la empresa se mantiene en contacto con las últimas tendencias tecnológicas. Además, para facilitar el uso de la cámara para ofrecer imágenes del mundo real, era imprescindible que el usuario accediese a esta experiencia a través del móvil.

1.2. Objetivos

Debido a que el trabajo con estas tecnologías era nuevo para nuestro equipo y carecíamos de experiencia al respecto, el primer punto a abordar fue la investigación sobre estas tecnologías. En primer lugar, se tomaron como referencia algunas aplicaciones publicitarias que utilizaban unas versiones algo más rudimentarias de *Realidad Aumentada* donde, escaneando un código QR, aparecía en pantalla una persona comentando promociones sobre la imagen grabada por la cámara del móvil. Nuestra intención era investigar qué tecnología se usó en ese caso o en otros casos donde el resultado fuese mejor que el de partida.

Por supuesto, y dado que el proyecto mayoritariamente trabajaría sobre la tecnología utilizada, el siguiente objetivo sería aprender a utilizarla y **generar una estructura eficaz** y escalable que permitiese futuras mejoras.

Para complementar la experiencia inmersiva de la aplicación, se añadiría **sonido espacial** al mismo sistema. Esto permitiría que un usuario con auriculares pudiera reconocer, sin mirar a la pantalla, la dirección de la que procedía el sonido y la distancia que le separaba del origen de este.

También tendríamos que **desarrollar un modelo en 3D** que funcionase como nuestro avatar. Para ello, habría que buscar herramientas de creación, diseño y modelado en 3D, además de aprender a utilizarlos para obtener un resultado satisfactorio.

Para que diese la impresión de que el modelo fuese el que dijera todo lo mencionado en la grabación, sería necesario **sincronizar la voz con los gestos faciales** y los labios del modelo. Para esta «sincronización labial», sería más eficiente buscar una aplicación o herramienta que fuese fácil de configurar e hiciese gran parte del trabajo. Por lo tanto, este objetivo se podría dividir en buscar una herramienta de sincronización labial, aprender a utilizarla y desplegar su funcionalidad sobre el proyecto.

Sería necesario también **generar una animación** para el resto del cuerpo del modelo 3D, de manera que este no resultara demasiado estático al hablar y diese una impresión algo más realista.

El sistema tiene que ser **accesible a través de la web**, por lo que sería necesario también alojarlo en un servidor que almacenase toda la funcionalidad, así como los protocolos necesarios para que la conexión fuese fiable y segura.

A mayores, otro punto a analizar en el proyecto era la accesibilidad del usuario a esta aplicación. Debido a la finalidad de la web, con obvias intenciones comerciales y laborales, el equipo encargado de desarrollar originalmente esta consideró que la mayor parte de los usuarios que visitaban la página accedían desde ordenadores, siendo una menor proporción los usuarios que accedían desde móviles o tablets. Por esta razón, se consideró que la mejor forma de hacer que un usuario pasara de un ordenador a un móvil era introduciendo un código QR que enlazase directamente a este sistema. En caso de que el usuario navegase desde el móvil, el mismo código funcionaría a la vez como un botón con la capacidad de redirigir.

1.3. Metodología

Debido a la capacidad de adaptación ante nuevos requisitos y a la posibilidad de generar nuevas entregas en un tiempo relativamente corto utilizando metodologías ágiles, se ha optado por esta preferiblemente sobre otras [26]. De esta manera, las partes interesadas serían capaces de ver en períodos cortos de tiempo avances en el proyecto que supongan un cambio importante. Es importante mencionar que, debido a que el equipo de desarrollo contaba solamente con una persona, no se ha aplicado el marco de trabajo *Scrum*, puesto que este está planteado para equipos más grandes.

Para centrar el trabajo en el menor número de tareas posibles al mismo tiempo, y favorecer la visibilidad del estado de las tareas y del proyecto, se ha decidido utilizar una adaptación de la metodología *Kanban* [28], una subcategoría de gestión ágil de proyectos centrada en el uso de los tableros homónimos. La idea de usar esta metodología surgió después de asistir a una charla de Pablo Santos en la Escuela de Ingeniería Informática en la Universidad de Valladolid [1], donde nos explicó los diferentes proyectos que dirigió, así como las metodologías que aplicó para liderarlos. En el momento de la charla, explicó cómo la metodología *Kanban* le permitía dar más visibilidad a las tareas en proceso, así como ver el flujo de trabajo, entre otros datos.

Según David Anderson, el ingeniero de Microsoft que aplicó esta metodología por primera vez a proyectos informáticos, los tableros *Kanban* contienen cinco elementos característicos: señales visuales, columnas, límites de trabajo en curso, punto de compromiso y punto de entrega.

- **Señales visuales:** todo el trabajo, junto con su estado actual, debe ser visible de un solo vistazo. Esta metodología utiliza las tarjetas como representación de tareas únicas que, en combinación con las columnas, nos revelarán el flujo de trabajo y el estado actual de este último, todo esto sin abandonar la filosofía de «información visual» de *Kanban*.
- **Columnas:** representan el estado de la tarea en el flujo de trabajo. Los tableros más sencillos suelen utilizar 3 columnas (*pendiente*, *en curso*, *finalizado*), aunque cada equipo puede organizar

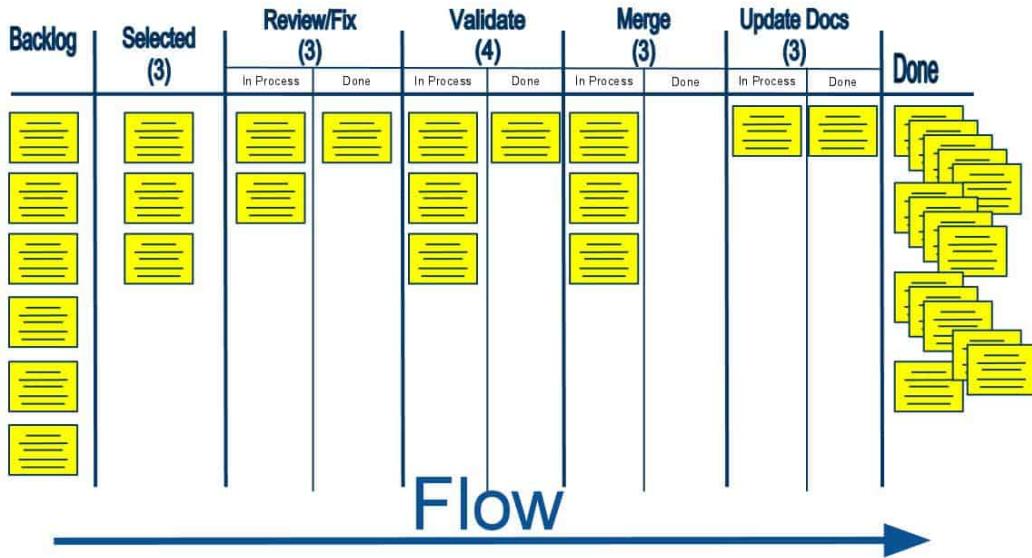


Figura 1.1: Ejemplo de tablero *Kanban*. En este, las tarjetas pasan por varias columnas de trabajo en curso antes de finalizar.

su flujo de trabajo a placer. En su charla, Pablo Santos llegó a indicar que cada empleado tenía su propia columna de «trabajo en curso» para visibilizar de manera más eficaz con qué tarea se encontraba cada empleado.

- **Límites de trabajo en curso:** las columnas que indican el trabajo en curso deben tener un límite establecido de tarjetas en ella. La metodología establece que, cuando una de estas columnas alcanza su límite de tarjetas, el equipo debe aplicar todo su esfuerzo en dichas tarjetas para lograr que avancen. Esta es la forma visual que tiene *Kanban* de mostrar al equipo que se está intentando asumir demasiado trabajo y de avisar de posibles cuellos de botella.
- **Punto de compromiso:** *Kanban* trabaja con una pila de tareas por desarrollar, comúnmente llamada *backlog*, que va creciendo según van apareciendo nuevas necesidades o requisitos. Cada vez que una de esas tareas sale de la columna de tareas pendientes, el miembro del equipo se «compromete» a terminar esa tarea en el menor tiempo posible. El momento de inicio de cada tarea se denomina por esto «punto de compromiso»
- **Punto de entrega:** es el momento en que una tarjeta se convierte en un trabajo completado y, por tanto, está ya en manos del cliente o del producto final. El tiempo entre el punto de compromiso y el punto de entrega se denomina «plazo», y la intención de la forma de trabajar con *Kanban* es que ese plazo sea el menor posible.

Al ser un proyecto de innovación dentro de la empresa, los requisitos iban surgiendo según íbamos encontrando nueva información al respecto. De esta manera, durante reuniones semanales se comentaba el avance del proyecto, así como las nuevas tareas a incorporar junto con su prioridad, procurando siempre no solapar unas tareas con otras.

1.4. Conceptos de Realidad Aumentada

Antes de comenzar a explicar en qué consiste este proyecto, es importante fijar algunos conceptos de *Realidad Aumentada* tal y como se van a utilizar en este tema, de manera que no queden ambigüedades a partir de este punto.

1.4.1. Realidad

El concepto más importante y a la vez básico a dejar claro es el concepto de *Realidad*. Para nuestro sistema, se considera *Realidad* al conjunto de objetos, información y estímulos, principalmente visuales, que componen nuestro entorno y que percibimos a través de nuestros sentidos. Así, si nos encontrásemos en nuestra casa, podríamos percibir varios elementos a través de la *Realidad*: una televisión, un sofá, una temperatura cálida, una iluminación baja, el ruido de un ventilador...

1.4.2. Realidad Aumentada y Realidad Virtual

La *Realidad Aumentada* y la *Realidad Virtual* son dos conceptos que a menudo se confunden por ser relativamente nuevos y por sus puntos en común. Sin embargo, los resultados pueden llegar a ser muy distintos o incluso opuestos en algunos casos. En ambos existe información generada a través de sistemas informáticos, pero las diferencias son ampliamente notables [20].

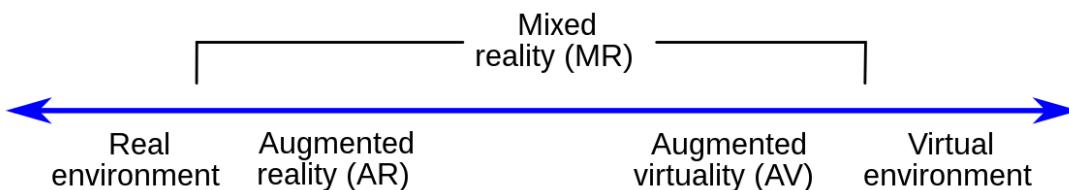


Figura 1.2: Espectro entre un entorno real y un entorno virtual.

La diferencia más destacable es el concepto de *Realidad* anteriormente mencionado, así como su uso: en la *Realidad Virtual*, todos los elementos son generados digitalmente, de manera que la *Realidad* percibida es totalmente distinta a nuestro entorno «real». Para esto, comúnmente se hace uso de las gafas de *Realidad Virtual*, que aíslan la visión del usuario para que solo pueda captar la información generada y transmitida por las gafas.

Sin embargo, en el caso de la *Realidad Aumentada*, la intención es ampliar en tiempo real los datos captados de la *Realidad*, por lo que la base en estas tecnologías siempre serán imágenes e información de nuestro entorno. A esto se le debe sumar todo aquello que añada el sistema para «ampliar» la *Realidad*: figuras, texto, imágenes, etc. Estas últimas se encontrarán superpuestas sobre lo captado del entorno de manera fidedigna y que ofrezca más valor a lo naturalmente captado.

Otra gran diferencia se puede encontrar en los dispositivos utilizados para aplicar ambas tecnologías. Como se comentó antes, la *Realidad Virtual* utiliza unas gafas que aíslan al usuario de la *Realidad*.



Figura 1.3: A la izquierda, las Meta Quest 2, las gafas de *Realidad Virtual* de Meta. A la derecha, las Microsoft Hololens, las gafas de *Realidad Aumentada* de Microsoft.

Pero cabe mencionar también que estas gafas son de uso exclusivo para dicha tecnología, así como su elevado coste, ya que son una tecnología que, pese a que ya lleva varios años de desarrollo y mejora, aún necesita asentarse correctamente en el mercado. La *Realidad Aumentada*, en cambio, se apoya generalmente en dos tecnologías dependiendo de su uso: por un lado, para los usos más cotidianos (aunque a veces también se encuentran en este grupo usos profesionales), se suele implementar en dispositivos móviles, donde la cámara capta las imágenes del entorno y el propio dispositivo móvil añade la información pertinente; por otro lado, y para usos exclusivamente profesionales, muchas empresas han comenzado a utilizar gafas de *Realidad Aumentada* donde, mediante un juego de espejos, el usuario es capaz de ver información añadida a su entorno.

Por último, y aunque ya se ha adelantado algo, se podría mencionar también como distinción que, por lo menos hasta el día de hoy, la *Realidad Virtual* se está especializando más en el área lúdica, al ser los desarrolladores de videojuegos los principales interesados en esta tecnología, aunque no se descarta que en el futuro pueda aplicarse para motivos más profesionales. La *Realidad Aumentada*, aunque también se utiliza para este fin, ha conseguido entrar en el área profesional como herramienta para varios sectores, como es el del marketing, donde aplicaciones como la de Ikea permiten al usuario colocar muebles de la tienda en su propia casa para probarlos y, así, impulsar las ventas.

1.4.3. Sonido espacial

Para ampliar la sensación de que los elementos que añadamos mediante nuestra aplicación forman parte de nuestro entorno, las voces de los asistentes virtuales dispondrán de una propiedad denominada sonido espacial, que es la propiedad por la cual somos capaces de identificar la posición, orientación y distancia del origen de un sonido [35].

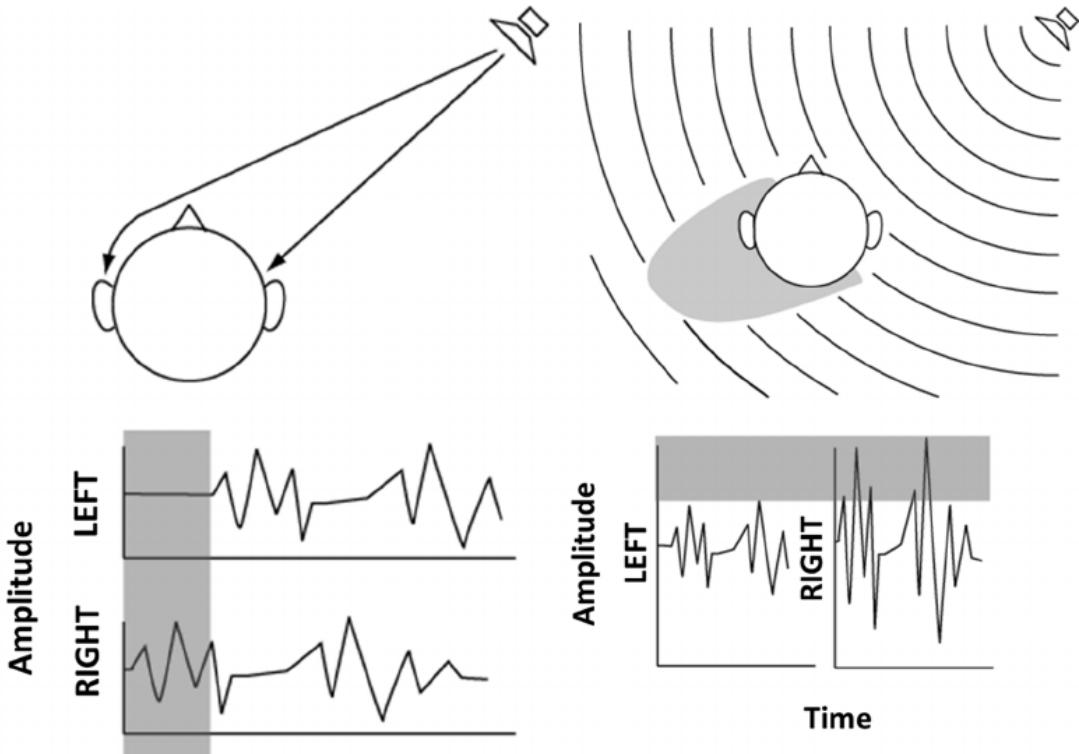


Figura 1.4: A la izquierda, una representación de la diferencia de tiempo interaural. A la derecha, una representación de la diferencia de nivel interaural.

Utilizando como ejemplo el sonido emitido por nuestra televisión. Si cerráramos los ojos, seríamos capaces de interpretar varios datos únicamente por el sonido: no solo de dónde procede o a cuánta distancia está, sino que también podríamos percibir información de nuestro entorno según el eco que produce, como por ejemplo si estamos en una habitación o al aire libre o el tamaño de la habitación en el primer caso.

Toda esta información se percibe gracias a cómo el cerebro interpreta los fenómenos físicos a través de los cuales se transmite el sonido [13, 35]. Por ejemplo, somos capaces de interpretar si un sonido viene desde nuestra derecha o desde nuestra izquierda a través de la *diferencia de tiempo interaural*: si un sonido tarda más en llegar a nuestro oído derecho que a nuestro oído izquierdo, quiere decir que el sonido procede de nuestra izquierda. Además, la diferencia de tiempo entre ambos oídos nos indicará también si el sonido está totalmente a nuestra izquierda o, en cambio, está situado en un ángulo distinto.

Otra forma que tiene nuestro cerebro de detectar la posición de un sonido es utilizando los cambios sutiles que se generan en las altas frecuencias cuando las ondas son interceptadas por la forma de nuestra oreja o de nuestra cabeza, conocidos como *diferencia de nivel interaural*. Por ejemplo, un ruido no nos llegará de la misma manera si se origina delante de nosotros o detrás de nosotros. En el primer caso, el sonido será parcialmente tapado por el trago de la oreja. Pero en el segundo caso, el sonido será interceptado por gran parte del pabellón auricular. En el segundo caso, las frecuencias más altas pueden llegar a perderse, por lo que nuestro cerebro interpretaría que el origen del ruido se

ha producido detrás de nosotros, mientras que en el primer caso, al mantenerse más altas frecuencias, interpretaríamos que la posición se genera delante de nosotros. Esto, en combinación con la inferencia anteriormente comentada, nos permitiría ubicar un sonido en el plano horizontal.

Por último, cabe mencionar que estas técnicas usadas por nuestro cerebro para ubicar sonidos en el plano horizontal son muy similares a las que utiliza para ubicarlos en el vertical o incluso para determinar la distancia a la que está dicho sonido, ya esté estático o en movimiento.

1.5. Tecnologías utilizadas

Para elaborar nuestra aplicación de *Realidad Aumentada*, nos hemos apoyado principalmente en tres herramientas ya existentes, cuyo uso se explicará con detalle más adelante.

El primero y más importante es *WebXR* [54]. Esta es una interfaz para generar entornos tanto de *Realidad Aumentada* como de *Realidad Virtual* y que facilita al desarrollador la interacción con el hardware. Aunque su uso se preparó para generar estos entornos en dispositivos de *Realidad Virtual* exclusivamente, más tarde se actualizó para poder ser utilizado en el navegador *Google Chrome* (versión 79) para móviles *Android*. Gracias a *WebXR*, podremos acceder a aspectos básicos para esta tecnología, como el acceso a la cámara del dispositivo, la ubicación y orientación del mismo, la creación de una sesión interactiva...

Para poder modificar y manipular los modelos en 3D que comentaremos en el desarrollo de esta memoria, hemos recurrido a la librería *Three.js* [37], una librería centrada en la generación de modelos y gráficos 3D para navegadores web para *JavaScript*. Esta puede utilizar varios formatos de modelos 3D como FBX, Collada o OBJ, pero la documentación recomienda encarecidamente utilizar el formato *glTF* por estar más centrado en su uso en tiempo de ejecución: es un formato muy compacto y rápido de cargar. *WebXR* se apoya en esta librería para mostrar estos modelos en imágenes del mundo real (*Realidad Aumentada*) o en entornos totalmente generados (*Realidad Virtual*).

La última librería utilizada es *Resonance Audio* [30], la cual permite ampliar las funcionalidades básicas de sonidos que contiene HTML en conjunto con *JavaScript*. En concreto, *Resonance Audio* permite simular sonidos espaciales recreando las características que permiten a nuestro cerebro posicionar un sonido a nuestro alrededor. Además, dispone de personalizaciones para el sonido emitido, como la simulación de eco en interiores dependiendo de los materiales de las paredes, techo y suelo que rodean al foco del sonido.

Capítulo 2

Sistema básico

2.1. Estructura

Debido a la clara repartición de tareas en cada sección del código, se ha pretendido aplicar una separación tipo Modelo-Vista-Controlador en la aplicación, de manera que todo esté visiblemente separado y se pueda utilizar un sistema de orientación a objetos en toda la parte escrita en *JavaScript*.

La primera sección a comentar sería el Modelo. En esta aplicación, tal y como se ha planteado el desarrollo y teniendo en cuenta qué librerías externas se han incorporado y cómo se han utilizado, se ha tratado a estas como si fueran el propio Modelo. Así, nuestra información es en sí las librerías externas añadidas, que serán además las que se encarguen de cargar los datos estáticos de servidor: los modelos en 3D y las grabaciones de las voces de estos.

La segunda sección sería la Vista. En este caso, la Vista de esta aplicación constaría únicamente de un HTML llamado *index*, que sería además la *landing page* de la aplicación. Sobre esta misma página será sobre la que se colocará más tarde un «*canvas*», que es el elemento HTML que permite mostrar elementos gráficos de manera dinámica a través de *JavaScript*. Utilizando este último podremos mostrar las imágenes captadas a través de la cámara del dispositivo móvil y mezclarlas con los modelos en 3D para generar nuestra *Realidad Aumentada*.

Por último, la sección restante sería el Controlador. En este, se procesan todos los eventos generados por el usuario, además de tratar toda la información que se recibirá de los sensores del dispositivo para poder generar las imágenes de manera verosímil. En este se ubicará un bucle que se ocupará de obtener la imagen a mostrar, mezclarla con el modelo en 3D, renderizar la imagen y situarla en la vista, todo esto de la manera más fluida posible para que el espectador no tenga impresiones que entorpezcan la experiencia de usuario.

Además, esta almacenará también información de la Sesión necesaria para el correcto procesamiento del sistema, tales como los modelos en 3D cargados, la referencia al objeto de *JavaScript* con el

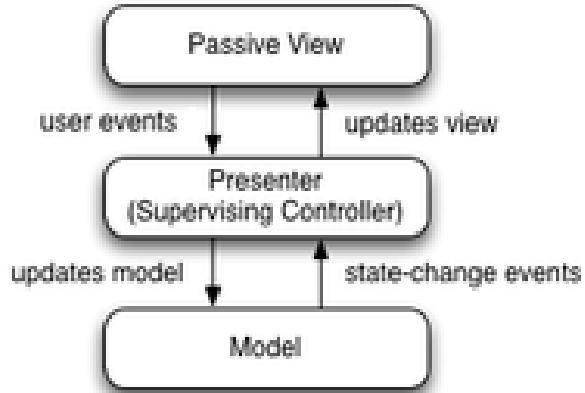


Figura 2.1: Representación gráfica del patrón Modelo-Vista-Presentador. Imagen obtenida de Wikipedia.

«*canvas*» antes mencionado, la posición del usuario y otros datos necesarios para el funcionamiento del sistema que serán mencionados a lo largo de esta memoria.

Es importante mencionar que, a pesar de que se ha tratado a este sistema como un Modelo-Vista-Controlador y se le ha presentado como tal, sería más justo tratar a la aplicación como si utilizara el patrón Modelo-Vista-Presentador [36] por varias razones.

La primera es que no tenemos constancia de cómo almacenan estas librerías las pocas referencias que se le entregan a la Vista y, aún manteniendo esta referencia como si lo tratara como la Vista en este tipo de patrones, no parece realizar cambios sobre esta de manera directa.

La segunda es que las actualizaciones se realizan directamente desde el Controlador, siendo necesario realizar desde este último acciones explícitas para actualizar la apariencia de la vista, como por ejemplo el renderizado de la imagen a mostrar en pantalla.

Por último y más importante, es necesario que todos los cambios estén revisados en el bucle que se genera en el Controlador, el cual se tratará más adelante. Esto obliga a este último a solicitar información continuamente a las librerías, hacer cálculos y mostrarlos en pantalla, mientras está atento a los eventos de los usuarios.

Por todo esto, y dado que la lógica de negocio está separada de la interfaz de usuario, la estructura se corresponde con el patrón Modelo-Vista-Presentador. Sin embargo, dado que desde el inicio del proyecto se trató cada parte como el clásico Modelo-Vista-Controlador, y dado que el Modelo-Vista-Presentador proviene de este último, cada parte será nombrada como si lo fuera, aún siendo conscientes de las diferencias.

2.2. Sesión WebXR

La primera herramienta que será necesario inicializar es una instancia de Sesión de *WebXR*. La Sesión de *WebXR*, llamada *XRSsession* [60], es el objeto de la librería *WebXR* que provee de las funcionalidades más básicas, así como de utilidades extra que serán implementadas a lo largo del desarrollo de la aplicación. Esta interfaz pretende representar la propia actividad que llevará a cabo el usuario mientras observa la *Realidad Aumentada*, por lo que tendrá funciones orientadas a iniciar, mantener y terminar esta actividad.

Para ser inicializado, el objeto Sesión tiene varios requisitos que se deben cumplir [57] para evitar problemas de seguridad. Los dos requisitos mencionados por la documentación de *ARCore* son:

- El servidor que ofrezca el servicio debe tener una conexión a través de un entorno seguro. *WebXR* considerará segura una comunicación que se transmita a través del protocolo *HTTPS* y/o que se ubique en *localhost*.
- El cliente debe utilizar un navegador compatible con *WebXR*. A fecha de este documento, solo existen tres navegadores para dispositivos móviles que cumplan este requisito [9]: *Google Chrome* para *Android* a partir de la versión 79, *Opera* para *Android* a partir de la versión 57 y *Samsung Internet* a partir de la versión 11.2. Además, el dispositivo que lo lance debe soportar la aplicación *ARCore*.

Además de esto, la Sesión se debe iniciar a través de lo que el sistema debe interpretar que es un evento de usuario. Esto quiere decir que la Sesión no se puede generar automáticamente o nada más cargarse la página, sino que debe ser el usuario pulsando un botón o realizando una acción consciente el que lance la solicitud de inicio de Sesión de *Realidad Aumentada*. Los dos primeros requisitos expuestos se comprueban a través de la función *isSessionSupported*:

```
1 if (!navigator.xr) window.polyfill = new WebXRPolyfill();
2
3 navigator.xr.isSessionSupported("immersive-ar").then(
4     this.#checkSessionSupported
5 );
```

Código 2.1: Uso de *isSessionSupported* en la aplicación.

Esta función (línea 3) está integrada en el parámetro *xr*, que está a su vez contenido en *Navigator*, objeto que contiene información acerca del navegador utilizado por el usuario. Este último contendrá el parámetro mencionado únicamente si cumple con el segundo requisito de la librería *WebXR*. En caso contrario, al no existir, podría generarse un error. Para estos casos, se utiliza *WebXRPolyfill* [56], una librería utilizada para la retrocompatibilidad en los navegadores compatibles, pero que también añade el mencionado parámetro y, con este, la funcionalidad que comprueba si se soporta la Sesión.

La línea 3 del código anterior termina en una llamada asíncrona mediante el uso de *.then()* a la función privada *checkSessionSupported*. Esta función es la que se encarga de avisar al usuario en caso

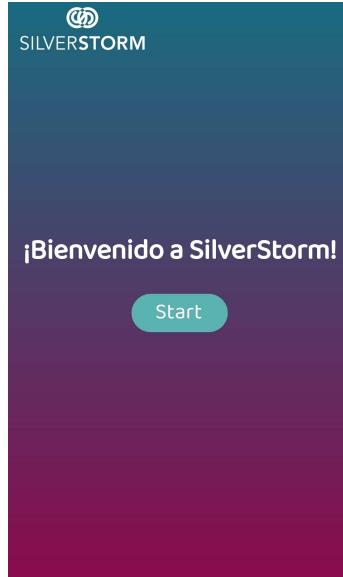


Figura 2.2: Imagen de la *landing page* definitiva de la aplicación web.

de que la Sesión no se pueda levantar. Sin embargo, en el caso afirmativo, se ocupa de preparar el entorno para el momento en que el usuario comience la sesión de *Realidad Aumentada*, cuyos detalles se explicarán más adelante.

```

1 #checkSessionSupported = (isSupported) => {
2     if (isSupported) {
3
4         // Prepara el entorno
5
6     } else {
7
8         document.getElementById("arButton").disabled = true;
9         alert("Tu navegador no permite una sesi\xF3n de Realidad Aumentada.");
10    }
11 }
12 
```

Código 2.2: Control de la Sesión, dependiendo de si es o no soportada.

Esta estructura de funciones, tan común en *JavaScript*, es la que se utilizará en este proyecto para las funciones *callback* [10], funciones pasadas a otras funciones en forma de argumento de forma que puedan ser llamadas una dentro de la otra. Así, en este caso, la función de la librería calculará si el sistema soporta o no la Sesión y, una vez lo sepa, llamará a nuestra función (*checkSessionSupported*) pasándole como argumento el resultado de sus cálculos.

Teniendo en mente estas condiciones, se planteó que la Vista debía ser una interfaz sencilla que contuviese un botón que debiera accionar el usuario para iniciar la Sesión, siendo en las versiones más prematuras un botón simple sin ningún tipo de customización. En las versiones más avanzadas, se generó un fichero *CSS* que personalizara la *landing page* de manera que, además de tener los colores de la empresa, tuviese un aspecto más amigable para el usuario. Antes de que este botón pudiera ser

pulsado, el sistema debería haber calculado previamente si el sistema puede soportar la sesión, así que con la carga de la Vista debía cargarse también el archivo Main.js, que consta únicamente de una línea de código:

```
1 window.arController = new Controller();
```

Código 2.3: Inicialización del Controlador.

Esta línea lanzará el Constructor del Controlador, que contendrá la construcción del *WebXRPolyfill* y la consulta al parámetro *xr* con la función *isSessionSupported*. De esta manera, el sistema está previamente preparado a cualquier acción del usuario, favoreciendo la carga de objetos más pesados de manera asíncrona mientras el usuario reacciona antes de pulsar el botón de lanzamiento de Sesión de *Realidad Aumentada*.

Ahora, una vez se ha comprobado que el sistema es compatible con *Realidad Aumentada*, el usuario puede ejecutar su acción consciente para iniciar la Sesión. Es importante tener en cuenta que, cuando se construye el objeto, también puede detectar si la acción es consciente o no, por lo que la construcción de este también debe estar sujeta a un evento de usuario. Debido a esto, la construcción de este objeto no se puede incrustar en el mismo sitio que hemos incrustado la carga de objetos más pesados, junto al lanzamiento de la página, sino que debe ir en la función que se lance al pulsar el botón que se muestra en la Figura 2.2.

Este botón lanzará la función asíncrona *startLoop* (cuyo significado revelaremos más adelante), que cargará los datos restantes para poder ejecutar el grueso de la aplicación, siendo la Sesión uno de ellos, como ya se ha comentado. En esta aplicación, la sesión se lanza de la siguiente manera:

```
1 // Constantes
2 #GLOPTIONS = { xrCompatible: true };
3 #SESSIONOPTIONS = { requiredFeatures: ['hit-test'] };
4 // ...
5
6 async startLoop() {
7     this.#canvas = document.createElement("canvas");
8     document.body.appendChild(this.#canvas);
9     this.#gl = this.#canvas.getContext("webgl", this.#GLOPTIONS);
10
11    // ...
12
13    this.#session = await navigator.xr.requestSession("immersive-ar", this.#SESSIONOPTIONS);
14    this.#session.updateRenderState({
15        baseLayer: new XRWebGLLayer(this.#session, this.#gl)
16    });
17
18    // ...
19}
```

Código 2.4: Función de preparación del bucle.

En el código se puede observar que, en la línea 13, no se está construyendo un objeto utilizando el constructor clásico *new*, sino que se está solicitando un objeto *XRSession* (Sesión de *WebXR*) al parámetro *xr* previamente mencionado mediante la función *requestSession* [63]. Esta es la manera que tiene *WebXR* de asegurarse de que la sesión se lanza mediante un evento de usuario, debido a que es precisamente esa función la que recoge la información necesaria y hacer las comprobaciones correspondientes al respecto.

También se puede observar que la función recibe dos parámetros: la cadena de texto “*immersive-ar*” en primer lugar y *this.#SESSIONOPTIONS* en el segundo. El primero es un parámetro obligatorio que aclara a la función qué tipo de Sesión se pretende lanzar (*Realidad Aumentada*, *Realidad Virtual* o *inline*, una opción que no nos ocupa ahora), siendo el valor presente en el código el necesario para indicar que queremos solicitar una Sesión de *Realidad Aumentada*. El segundo valor, opcional en este caso, sirve para especificar la configuración que se establecerá en nuestra sesión, en caso de que no queramos utilizar los valores por defecto. En esta aplicación se establece una única opción, definida en la constante de la línea 2, que indica las características que serán requeridas durante la sesión. En este caso, se indica que será necesaria la característica *Hit Test* o, como lo llamaremos para esta aplicación, «cálculo de superficies», aunque eso es algo que explicaremos más adelante.

Justo después de solicitar la Sesión, se utiliza a esta misma para lanzar la función *updateRenderState* [62] en la línea 15. Esta función solicita cambios de configuración a partir del siguiente fotograma que se cargue. Sin embargo, como a estas alturas aún no se ha cargado ningún fotograma, se entiende que estos «cambios en la configuración» son para el primero y para los siguientes fotogramas (es decir, para todos, dado que esa configuración no va a cambiar en esta aplicación).

La configuración que se solicita cambiar para los fotogramas es la que se prepara en las líneas 7, 8 y 9 junto con la constante de la línea 2: en la línea 8, estamos generando un *canvas*, que es un objeto de HTML generado a partir de *JavaScript* cuya finalidad es albergar objetos gráficos, como ya se mencionó en la sección 2.1. Para insertar este objeto en nuestra Vista, se ejecuta la acción de la línea 8, momento a partir del cuál este objeto ocupará la pantalla completa de nuestro dispositivo móvil. Si no añadiésemos más funcionalidad, veríamos que nuestra pantalla no mostraría nada más que una pantalla en negro. Esto se debe a que aún no hemos insertado ningún objeto gráfico en nuestro *canvas*, para lo que aún quedan unos pasos.

Por último, en la línea 9 se solicita al objeto *canvas* un «contexto» a través de la función *getContext*. El contexto de un *canvas* es una interfaz a través de la cuál se puede interactuar y generar gráficos e imágenes desde el código de *JavaScript* [52], por lo que nos resulta imprescindible para trabajar con este «lienzo». Primero, a la función le indicamos qué tipo de imágenes vamos a utilizar para que nos devuelva el objeto de contexto correcto (en nuestro caso, el contexto va a ser la inserción de imágenes en 3D). Eso lo hacemos mediante el primer parámetro (“*webgl*”), mientras que en el segundo, tal y como ocurrió al solicitar la sesión de *WebXR*, le establecemos una serie de opciones que queremos que sean distintas a las predeterminadas. En este caso, queremos activar la opción «*xrCompatible*», que

indica que indica al *canvas* que el contexto que nos devuelva tiene que ser compatible con *WebXR* [19]. Esta configuración es la que se almacena en la constante de la línea 2.

Una vez entendido esto, solo queda añadir esta configuración a la función *updateRenderState*. A través del objeto *XRWebGLLayer*, generaremos un objeto que enlace el contexto del *canvas* (es decir, la interfaz para poder dibujar sobre este) y la Sesión *WebXR* que mantendremos para aplicar nuestra *Realidad Aumentada* [66]. Esto lo asociaremos con la opción «*baseLayer*» a la misma Sesión a partir del momento en que comiencen a lanzarse los fotogramas con la sencilla intención de indicarle a la Sesión cuál será el objeto que le renderizará las imágenes obtenidas desde la cámara del dispositivo. Esto significa que el contexto, a través del enlace creado con el objeto *XRWebGLLayer*, obtendrá la información de la cámara del dispositivo y lo traducirá en la colección de píxeles que plantaremos en el «lienzo».

De esta manera tendríamos iniciada la forma más básica de Sesión, aunque aún no seríamos capaces de ver nada a través de la pantalla de nuestros dispositivos. Para esto, vamos a necesitar lanzar un bucle en el que procesemos y mostremos los fotogramas que captamos con la cámara de nuestro móvil.

2.3. Procesado iterativo

Para poder mostrar por pantalla cada fotograma, así como hacer los cálculos relacionados con las figuras que vamos a insertar llegado el momento, es necesario crear un bucle que dure tanto tiempo como el usuario vaya a permanecer en la Sesión. En cada una de las iteraciones, la aplicación solicitará al hardware del dispositivo móvil la imagen captada a través de su cámara.

Este bucle funcionará de una manera un tanto especial: a diferencia de los bucles *for* o *while*, donde se ejecuta un código de manera secuencial y sin tener nada más en cuenta que lanzar la siguiente iteración cuando la anterior haya terminado, en este bucle se «solicita» al navegador que ejecute una función *callback* en cuanto esté preparado, lo que permite que la aplicación se adapte a la capacidad del dispositivo, debido a que no todos serán capaces de procesar igual de rápido la misma información. En esta función *callback*, que definiremos nosotros, será en la que se le indicará al sistema que muestre por pantalla la información que queramos. Esto, a efectos prácticos, es lo que determinará los fotogramas por segundo: si un sistema, desde que se solicita la ejecución de la función hasta que muestra el fotograma en pantalla, tarda 0,025 segundos de media, entonces mostrará en un segundo alrededor de 40 fotogramas (es decir, 40 fotogramas por segundo).

Esta capacidad para solicitar la ejecución de una función *callback* cuando el sistema esté disponible la tiene la función *requestAnimationFrame* de la Sesión de *WebXR* [61]. Por esto, es imprescindible que la Sesión se haya iniciado previamente. En esta aplicación, la función se lanza en dos puntos dentro del Controlador.

```

1 async startLoop() {
2
3     // ...
4     // Solicitud del primer fotograma
5     this.#session.requestAnimationFrame(this.#loopFunction);
6
7 }

```

Código 2.5: Función que solicita la carga del siguiente fotograma.

La primera llamada a dicha función se realiza al final de la función *startLoop*, que es la función llamada por el botón de la Vista y que, como su propio nombre indica, es la que inicializa este bucle, haciendo ciertas preparaciones como el inicio de la Sesión, como ya sabemos. Al lanzar la función *requestAnimationFrame*, es necesario indicarle cuál va a ser la función *callback* que se va a poner «en cola» de ejecución. En nuestro caso, esta función es *loopFunction*, donde ejecutaremos nuestro procesamiento de imágenes:

```

1 #loopFunction = (time, frame) => {
2
3     // ...
4     // Solicitud de siguiente fotograma
5     this.#session.requestAnimationFrame(this.#loopFunction);
6
7     // Enlazado de imagen de camara con modelo
8     this.#gl.bindFramebuffer(this.#gl.FRAMEBUFFER, this.#session.renderState.
9         baseLayer.framebuffer);
10    // ...
11 };

```

Código 2.6: Función bucle con el contenido mínimo

Para poder generar el bucle que se encargará de procesar y mostrar la imagen, es necesario que en algún punto se solicite de nuevo al navegador que se vuelva a lanzar esta misma función. Para ello, el mejor sitio es la propia función *loopFunction*, que solicitará al navegador que se vuelva a lanzar cuando este esté preparado (línea 5), formándose así un bucle infinito, en el que se realizará el procesado que definamos en la función por cada iteración y que terminará cuando el usuario decida, como se explicará más tarde.

En esta función *callback*, podemos ver que se reciben dos parámetros generados desde la Sesión al llamarla: *time* y *frame* [61]. El parámetro *time* es una variable de tipo *Double* que representa la diferencia de tiempo en milisegundos desde que se ha iniciado la Sesión hasta que se ha ejecutado la función indicada en *requestAnimationFrame*. Esta es una herramienta útil para calcular la posición idónea en la animación de un modelo en 3D, pero en esta aplicación se utiliza una herramienta distinta por conveniencia, al pertenecer a la librería *Three.js*, de la que se tratará más adelante. Por otro lado, el segundo parámetro, *frame*, es el objeto en que *WebXR* nos almacena información que, más tarde, nos será crucial para poder hacer los procesamientos que queremos, como por ejemplo la posición y orientación del usuario en el momento de la iteración.

Ahora, antes de comenzar cualquier procesamiento, vamos a decir al sistema cómo obtener la imagen obtenida a través de la cámara en nuestro *canvas*. Esto se realiza mediante la función de la línea 7, [53], que pertenece al contexto del *canvas* e que implanta en su *Framebuffer* la información que le indiquemos. El *Framebuffer* es un espacio de memoria utilizado para almacenar la información necesaria para generar la salida visual en la pantalla. Entre otros datos, en el *Framebuffer* se almacenará información como los valores de color, de transparencia y brillo de cada píxel [12], así como la proporción de la imagen a generarse. En este caso, el *Framebuffer* destino pertenece al contexto del *canvas*, por lo que al almacenar en este la información de una imagen, el contexto se encargará de pintarlo en el «lienzo».

La función *bindFramebuffer* recibe dos argumentos. En el primero, se indica el *buffer* que se va a utilizar mediante una enumeración contenida en el mismo contexto. El objeto ofrece varias opciones según el tipo de operación que se pretenda utilizar, pero nosotros utilizaremos `#gl.FRAMEBUFFER`, que es el único *buffer* que existe para *WebGL*, siendo el resto para contextos *WebGL 2*. En el segundo argumento, se indica de dónde debe extraer la información sobre la imagen que se generará en el lienzo. En este caso, lo extraeremos del objeto Sesión. Este contiene una propiedad de solo lectura que provee información sobre la imagen a renderizar llamado *renderState*, de donde podremos adquirir el objeto *baseLayer* que contendrá su propio *Framebuffer* para almacenar la imagen captada por la cámara del dispositivo. De esta manera, estaremos indicando al sistema que pinte en el «lienzo» la imagen captada por la cámara.

Con esto, ya tendríamos una aplicación capaz de captar las imágenes de la cámara del dispositivo móvil y que nos ofrecería información del dispositivo como la posición u orientación del mismo. Ahora, podemos trabajar en insertar los modelos en 3D para generar nuestra *Realidad Aumentada* básica.

Capítulo 3

Implementaciones sobre el sistema de Realidad Aumentada

3.1. Modelo en 3D y Three.js

A partir del sistema básico generado en el capítulo anterior, podemos empezar a añadirle complejidad a nuestro sistema. El primer elemento que añadiremos es un modelo en 3D que utilizaremos como base en esta aplicación.

Para la carga, inserción, control y movimientos de los modelos en 3D se ha utilizado en esta aplicación la librería *Three.js*. *Three.js* es una librería ligera preparada para su uso en aplicaciones web compatible con *JavaScript* [58]. Además, este está especializado en el control de modelos en 3D en formato *GLB* y *glTF*, que son dos formatos basados en *JSON* y que están creados para ser óptimos en tiempo de ejecución (el formato *GLB* es el equivalente en binario a *glTF*) [41].

La librería *Three.js* basa su funcionalidad en el uso de Escenas o *Scenes*, el objeto que utiliza esta librería para almacenar y presentar sus modelos en 3D, la iluminación o focos de luz que los alumbrarán y las cámaras virtuales que observarán a estos mismos [43]. Aplicando un símil sencillo, la Escena de *Three.js* podría equivaler a una escena de cine, donde los actores equivaldrían a los modelos en 3D, los focos equivaldrían a la iluminación y las cámaras equivaldrían a las cámaras virtuales. Este último concepto es muy importante, debido a que afectará continuamente al renderizado de los modelos en 3D.

La cámara virtual es la que simula el punto de vista del espectador, ya sea en *Realidad Aumentada*, en *Realidad Virtual* o en otras virtualizaciones que utilicen modelos en 3D como son las simulaciones, los videojuegos o las películas de animación [50]. A alto nivel, una cámara virtual, además de la posición del espectador, definiría también lo que este va a poder observar de manera que, cuando se va a renderizar una imagen, solo tiene que generarse la parte visible de los modelos que entren en el rango de visión de la cámara virtual.

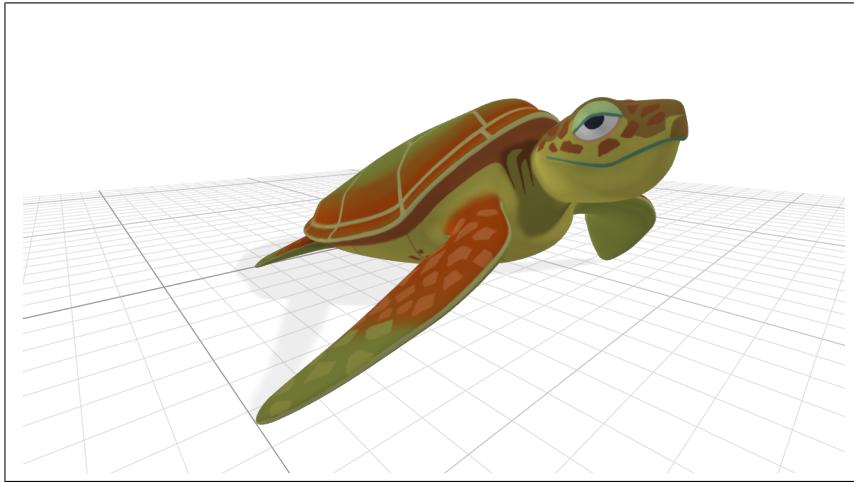


Figura 3.1: Ejemplo de renderizado de un modelo en 3D extraído de la biblioteca de modelos 3D de la aplicación de Windows *Visor 3D*.

Visualizando esto último con un ejemplo, en la figura 3.1 se ha renderizado un modelo 3D de una tortuga marina. En este renderizado, la cámara virtual se encuentra frente al modelo, pero ligeramente ladeado hacia la parte derecha de la tortuga. Debido a la posición de la cámara, no es necesario que se renderice la pata izquierda trasera de la figura, puesto que lo tapa el resto del cuerpo. Sin embargo, si la cámara se encontrase detrás de la tortuga, sería necesario renderizar ambas patas traseras y la cola, pero no se renderizaría la cara de la figura. De la misma manera, si la cámara no estuviese apuntando hacia la tortuga, no habría ningún modelo 3D que renderizar.

Aunque este parezca un concepto obvio, es necesario explicarlo, debido a que será necesario establecer la posición y orientación de la cámara virtual durante el desarrollo. Esto es porque la cámara virtual coincidirá con la posición de la cámara del dispositivo móvil. Cuando la cámara apunte hacia una figura en 3D en la *Realidad Aumentada*, se incrustará en la pantalla del móvil la parte de la figura que se vea desde la cámara virtual (es decir, se debe ver por la pantalla lo que se vería a través de la cámara si esta figura existiese en nuestra Realidad).

Una vez explicado esto, es necesario volver al código. Siguiendo el orden de carga, el primer lugar donde nos detendremos es en la función *callback checkSessionSupported*, vista en la sección 2.2, que es la función que se lanza desde el constructor del Controlador y que inicializaba los elementos más pesados antes de que el usuario pulsase el botón de iniciar la Sesión de *Realidad Aumentada*. En esta, ya habíamos visto las acciones que se realizaban en caso de que el navegador no fuese compatible con *WebXR*, pero ahora vamos a ver algunas de las que se ejecutan en el caso positivo. Ampliando el código 2.2 al que hacemos mención:

```

1 // Constantes
2 #RETICLELINK = "http://url_donde_se_aloja_el_modelo_3d";
3
4 #checkSessionSupported = (isSupported) => {
5     if (isSupported) {
6
7         // Scene y Loader
8         this.#scene = new THREE.Scene();
9         this.#loader = new THREE.GLTFLoader();
10        // ...
11
12        // Modelo en 3D: Reticula
13        this.#loader.load(this.#RETICLELINK, function(gltf) {
14            // Figura
15            this.#reticle = gltf.scene;
16            this.#reticle.visible = false;
17            this.#scene.add(this.#reticle);
18        }.bind(this));
19        // ...
20
21    } else {
22
23        document.getElementById("arButton").disabled = true;
24        alert("Tu navegador no permite una sesi\xF3n de Realidad Aumentada.");
25    }
26}
27};

```

Código 3.1: Carga de elementos de Three.js si se soporta la sesión.

Aquí ya podemos ver tres elementos de cierto peso que se están cargando previamente antes de iniciar la Sesión: la Escena, una instancia de *GLTFLoader* y, desde una función de esta última, un modelo 3D al que llamaremos a partir de ahora, «retícula», sencillamente porque es el nombre del modelo original alojado en la librería de modelos de ejemplo de *WebXR* en GitHub. Este modelo en 3D, que podemos ver en la figura 3.2 será el que utilicemos como base para el funcionamiento de esta aplicación, haciendo las veces de «puntero» más adelante, cuando queramos señalar un punto en el suelo para ubicar el avatar que desarrollemos.

En primer lugar, inicializamos la Escena en la línea 8 para poder introducir la retícula más adelante (volviendo al símil anteriormente utilizado, es como si el actor «retícula» entrara en escena). Sin embargo, esto no lo podemos hacer hasta que se haya cargado el modelo. Para esto, utilizamos el objeto *GLTFLoader* de *Three.js* [16], que es inicializado en la línea 9 para poder ser usado a continuación. Este objeto contiene la función *load()*, que permite cargar un modelo 3D (en este caso, en formato *glTF* o *GLB*, dado que este objeto es una especialización del objeto *Loader* y solo permite cargar archivos de dichos formatos) y lanzar una función *callback* una vez obtiene un resultado. Mediante la función de *GLTFLoader*, *Three.js* hace la traducción del contenido del fichero a objetos de *Three.js* que sean manejables desde código. Aunque la documentación no sea muy extensa sobre el objeto que

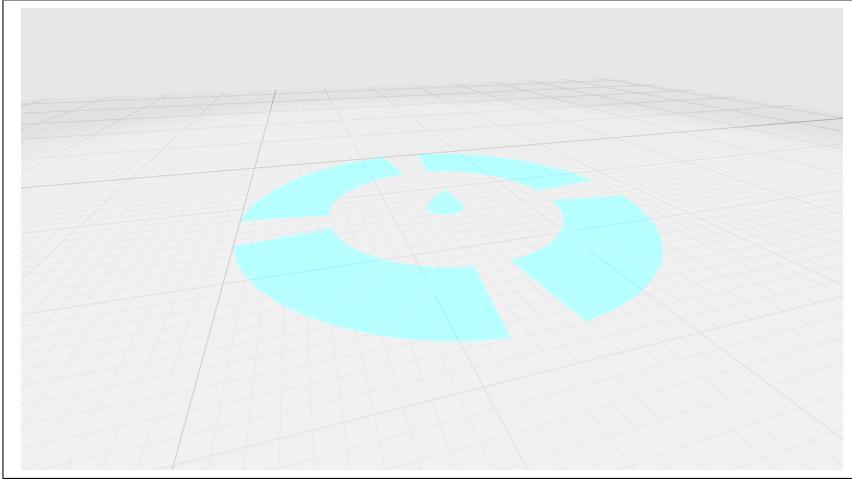


Figura 3.2: Muestra de la retícula utilizada en esta aplicación. Modelo obtenido del repositorio *webxr* del perfil de *Immersive Web at W3C*, en GitHub.

ofrece esta función, sí se puede encontrar que, mediante la función *callback* que creemos para esto, recibiremos un objeto (que en la línea 13 hemos llamado *gltf*) que contiene los siguientes atributos [22]:

- *animations*: devuelve un array con las animaciones (*THREE.AnimationClip*) que contenga el archivo con el modelo 3D. Estas animaciones, aunque ya entraremos más en detalle más adelante, no son más que conjuntos de posturas que, agrupados de la manera correcta, dan la sensación de movimiento si se van recorriendo a una velocidad adecuada.
- *scene*: aunque el sistema llame al atributo también «Escena», en verdad el objeto contenido es de tipo *THREE.Group*. Este objeto se puede definir como un conjunto de mallas de triángulos, que son los que formarán las superficies de los objetos tridimensionales. Al recoger todas las mallas del archivo de manera unida como un solo grupo, estaremos recogiendo en verdad el modelo 3D al completo.
- *cameras*: en caso de que el archivo contenga una cámara virtual preestablecida, podremos encontrarlo en este atributo, que nos devolverá un array de objetos *THREE.camera*. Estas cámaras no nos servirán para la *Realidad Aumentada*, porque nosotros queremos solamente una cámara que se encuentre en todo momento en el punto de vista de la cámara del dispositivo móvil. Este atributo es mucho más útil cuando se pretende utilizar la librería para aplicaciones web de visualización de este tipo de figuras tridimensionales, como pueda ser *Babylon.js Sandbox*.

Estos son los atributos más importantes, aunque también contiene otros que son de menor importancia para el desarrollo hasta ahora de este proyecto:

- *asset*: objeto de *JavaScript* sin tipo (*Object*) que contiene metadatos del archivo, comúnmente generados de forma automática por la herramienta de diseño utilizada para la construcción del modelo.

- *parser*: objeto *GLTFParser* que contiene la información del objeto que se ha utilizado internamente para transformar el archivo.
- *scenes* (no confundir con *scene*): de manera similar a *scene*, contiene un array de objetos de tipo *THREE.Group*. Esto se debe a que los archivos *glTF* pueden contener varios grupos separados de mallas, aunque no será el caso en este proyecto.
- *userData*: objeto sin tipado que contiene información customizada por parte del creador del archivo. Esta no está estandarizada, por lo que la información podría estar de cualquier manera.

Como se puede comprobar, son muchos los elementos que se cargan de un solo archivo, aparte del propio modelo en 3D. Esto se debe a que, como se verá más adelante, al diseñar un modelo en 3D se pueden añadir distintas propiedades preestablecidas. Más adelante, en la sección 3.3, veremos como, desde el mismo archivo del modelo, podremos acceder también a las animaciones predefinidas que se encuentren contenidas en este. Sin embargo, durante este diseño también se pueden preestablecer otros elementos y guardarlos en el mismo archivo como iluminaciones y cámaras virtuales.

Una vez sabemos esto, y volviendo al código 3.1, podemos ver que, en la línea 15, se está almacenando en el atributo privado *reticle* del Controlador el grupo de mallas que formarán el modelo 3D de la retícula. Después de esto, se establece la propiedad *visible* de este modelo a *false* para hacerlo invisible hasta que nos interese, y se añade (esta vez sí) a la Escena de nuestra *Realidad Aumentada*. De esta manera, la figura se estará cargando desde un inicio para ser mostrada en cuanto esté preparada, teniendo en cuenta que todavía no se ha solicitado iniciar la Sesión de *Realidad Aumentada*.

En el momento en que el usuario pulse el botón, con estos elementos pesados ya cargados o, como mínimo, en proceso de ser totalmente cargados, se comenzará a preparar desde la función *startLoop* el bucle de la función *callback loopFunction*, donde también se cargarán algunos elementos de *Three.js* a mayores de los que vimos en los códigos 2.4 y 2.5 de la sección 2.3. Dado que este código puede quedar muy largo si se incluyen todos los elementos previamente vistos, todo lo que se haya explicado con anterioridad quedará resumido mediante un comentario de código:

```

1  /** Opciones del renderizador de la RA. */
2  #RENDEREROPTIONS = {
3      alpha: true
4  };
5
6  async startLoop() {
7      // Carga de Canvas y GL
8      // ...
9
10     // Carga de Renderer
11     this.#RENDEREROPTIONS.canvas = this.#canvas;
12     this.#RENDEREROPTIONS.context = this.#gl;
13     this.#renderer = new THREE.WebGLRenderer(this.#RENDEREROPTIONS);
14     this.#renderer.autoClear = false;

```

```

15
16 // Carga de Camera
17 this.#camera = new THREE.PerspectiveCamera();
18 this.#camera.matrixAutoUpdate = false;
19
20 // Carga de Session
21 // ...
22
23 // Carga de Reference space
24 this.#referenceSpace = await this.#session.requestReferenceSpace('local');
25
26 // ...
27
28 this.#session.requestAnimationFrame(this.#loopFunction);
29 }

```

Código 3.2: Carga de elementos de Three.js antes de lanzar el bucle.

Dentro de esta función, podemos ver que se están preparando tres objetos distintos, de los cuales dos de ellos son parte de la librería *Three.js*. En primer lugar, se prepara el Renderizador, que es el objeto que se encargará de convertir el modelo 3D en imágenes que se insertarán en cada uno de los fotogramas que se muestren por pantalla [51]. Es muy importante tener en cuenta que el Renderizador superpondrá las imágenes de los modelos en 3D sobre las imágenes captadas por la cámara sin tener en cuenta el ángulo o la posición. Será trabajo del sistema que desarrollemos hacer que concuerde la posición del modelo con la superficie sobre la que se vaya a colocar. Para entender esto, es mejor ver las imágenes por separado:

- Primero, mediante *WebXR*, captaremos las imágenes de la cámara del dispositivo. *WebXR* también nos ofrecerá información que utilizaremos para calcular la pose en la que tendremos que colocar los modelos 3D. En el ejemplo de la figura 3.3, la imagen de la cámara correspondería con la fotografía de la izquierda.
- Despues, mediante *Three.js*, colocaremos el modelo 3D en la posición adecuada usando la información del punto anterior. Además, también tendremos que colocar la cámara virtual de manera que apunte al modelo en el ángulo adecuado. En el ejemplo, la figura estaría colocada de tal forma que coincida la distancia entre la cámara virtual y la tortuga y la distancia entre el suelo y el dispositivo móvil. También coincidirá el ángulo de incidencia de ambas cámaras para que parezca que se observa desde el mismo sitio al mismo punto.
- Por último, se superpondrán las imágenes. Primero se colocará la imagen de la cámara sobre la pantalla, y nuestro sistema renderizará al modelo en la posición adecuada usando la librería *Three.js* para que nosotros superpongamos la imagen renderizada sobre la imagen captada por el dispositivo. De esta manera, dará la impresión de que la figura se encuentra en dicho fotograma. Esta acción se realizará fotograma por fotograma.

Teniendo esto en cuenta, es importante que la renderización de la figura se realice con un fondo transparente o *alfa*: cuando solicitemos al renderizador que nos genere una imagen, tendremos que

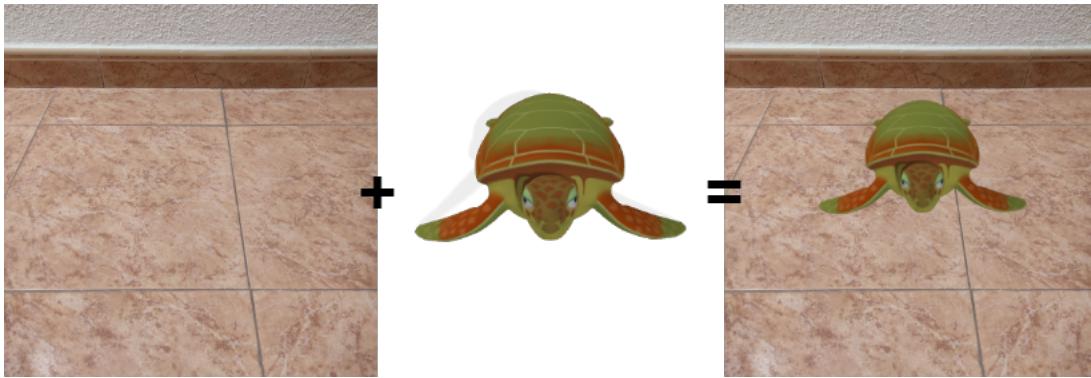


Figura 3.3: Ejemplo de superposición de imágenes en una *Realidad Aumentada*. La pose de la figura en 3D se calcula en relación con la imagen de la cámara para que su posición resulte fidedigna.

indicarle el tamaño que tendrá dicha imagen para que concuerde con el tamaño de la pantalla. Este, generalmente, coincidirá con el tamaño de la imagen que capte la cámara, por lo que tendremos dos imágenes con el mismo tamaño solapándose en pantalla. Si queremos que la superposición tenga algún efecto visual, lo ideal es que el fondo de la imagen superior tenga un fondo transparente de forma que, si lo colocamos sobre cualquier otra imagen (como es el caso), se pueda observar sin problema la imagen inferior alrededor de la superior, como en el ejemplo de la figura 3.1.

Parte de esta información se establece en el código 3.2, a través de las opciones que se le indican al renderizador al ser inicializado en la línea 14. Antes de ser inicializado, se genera el objeto *RENDEREROPTIONS* con elementos estáticos (fuera de la función) en la línea 2, donde se le indica que el fondo sea transparente a través de la propiedad de la línea 3. A mayores, dentro de la función se establecen otras propiedades que necesariamente se necesitan en tiempo de ejecución, como son en este caso los objetos *canvas* y *gl* (el contexto del *canvas*), de los que ya tratamos previamente en la sección 2.2. Esto se hace para que, al renderizar, se apliquen directamente los cambios sobre el *canvas* que creamos para añadir los gráficos.

A continuación, se inicializa en la línea 17 la cámara virtual que utilizaremos durante la Sesión. A mayores de la propia inicialización, se le modifica el valor de una de sus propiedades en la línea siguiente: *matrixAutoUpdate*. Esta propiedad heredada del objeto *Object3D* (objeto del que heredan también los objetos de Mallas anteriormente tratados), cuando se establece como verdadero, establece la posición, rotación y escala del elemento en cuestión en cuanto siente un cambio en estos, a ser posible en el fotograma en el que se realiza el cambio. Sin embargo, nosotros estableceremos el valor como false para poder indicar el fotograma exacto en el que queremos dicho cambio, evitando así que el sistema haga estos cambios en momentos inesperados.

Finalmente, en la línea 24, se inicializa un objeto muy importante que, aunque no forme parte de la librería *Three.js*, es necesario explicarla. El espacio de referencia, o Reference Space, es una definición de los sistemas de coordenadas que serán utilizados, en nuestro caso, dentro de la propia Sesión de *Realidad Aumentada* [55]. Lo que esto quiere decir es que, a través de los sensores de movimiento del

dispositivo móvil, el sistema traducirá (o intentará traducir) en todo momento la información recibida en datos de movimiento y posición del mismo, y utilizará esto para calcular en todo momento dónde está el punto en el espacio que ha calculado como referencia, o dicho de otro modo, la coordenada $(0, 0, 0)$. Dicho de una manera resumida, el espacio de referencia es un sistema de coordenadas que tiene como punto 0 el lugar donde se inicia la Sesión, y el sistema mediante los sensores del dispositivo tratarán de calcular dónde está en todo momento el punto 0 y, a partir de este, tratarán de calcular las coordenadas del mismo dispositivo. Una vez sepa el dispositivo «dónde está», será capaz de calcular también donde están otros objetos a los que se les ha dado también coordenadas en el espacio.

Será este espacio de referencia el que utilicemos para colocar objetos a nuestro alrededor y poder mantener la referencia de su ubicación y pose. Para ello, primero se define qué tipo de espacio de referencia se va a utilizar. La documentación de *WebXR* define varios tipos de espacios de referencia, pero recomienda concretamente *local* para casos en los que el usuario no se vaya a alejar mucho del punto de origen y esté sentado (o cerca del suelo). En nuestro caso, no podemos saber si el usuario va a estar de pie o sentado, lo que podría suponer un debate entre usar *local* o usar *local-floor*, pero la misma documentación también nos indica que no existe una gran diferencia entre los dos tipos, al indicarnos que la diferencia sustancial entre ambos es que en el primero el punto 0 se genera cerca del punto donde estaba el dispositivo cuando se inició la Sesión y en el segundo se genera procurando acercar la coordenada *y* cerca del suelo, lo que para este sistema no va a suponer una variación significativa. Por otro lado, la documentación sí establece las similitudes, al definir a ambos como los tipos de espacio de referencia en los que el usuario se mantendrá cerca del origen o se desplazarán a poca distancia del mismo, por lo que el sistema se optimizará para mantener con la mayor precisión posible el punto 0 y mantenerlo estable en relación a las variaciones de posición del mismo usuario.

Por último, y para terminar con los espacios de referencia, cabe mencionar que el espacio de referencia se solicita a través de una función del objeto que contiene nuestra Sesión. De esta manera, la librería facilita la comunicación del sistema con los detectores de posición y movimiento del móvil al devolver una interfaz que contiene toda la información necesaria. Esta llega a través de una promesa para los casos en que sea necesario recibir la interfaz de manera asíncrona. Sin embargo, en el momento en que se solicita, nuestro sistema está preparando el bucle de renderizado, el cual es un momento crítico donde muchos elementos tienen que estar preparados para que nada falle. Por lo tanto, se esperará a que esta promesa finalice y devuelva el espacio de referencia utilizando *await*, de manera que el sistema no pase a la siguiente orden hasta que finalice la función *requestReferenceSpace*. Una vez termine, podremos solicitar el primer fotograma a través de la función de la línea 28 explicada previamente en la sección 2.3.

Una vez solicitado el primer fotograma, se comenzará a ejecutar la función *callback* para procesado de fotogramas en bucle, hasta que el usuario decida pararlo. En esta función, las acciones que realizaremos relacionadas con la librería *Three.js* estarán orientadas al posicionamiento de los modelos en 3D que mostremos usando *Realidad Aumentada* y a su renderizado. Como se hizo antes, se dejará comentado el código ya visto para evitar que quede demasiado extenso:

```

1 #loopFunction = (time, frame) => {
2     // ...
3
4     // Solicitud de siguiente fotograma y enlazado de imagen de camara con modelo
5     // ...
6
7     // Calculo de posicion del usuario
8     const pose = frame.getViewerPose(this.#referenceSpace);
9     if (pose) {
10         const view = pose.views[0];
11
12         // Establecimiento de las proporciones de las imagenes renderizadas
13         const viewport = this.#session.renderState.baseLayer.getViewport(view);
14         this.#renderer.setSize(viewport.width, viewport.height);
15         // ...
16
17         // Posicion virtual del usuario para la vista
18         this.#camera.matrix.fromArray(view.transform.matrix);
19         this.#camera.projectionMatrix.fromArray(view.projectionMatrix);
20         this.#camera.updateMatrixWorld(true);
21         // ...
22
23         // Posicion de la reticula
24         this.#reticle.position.set(0.0, 0.0, 0.0); // stub
25         this.#reticle.updateMatrixWorld(true); // stub
26         // ...
27
28         // Aplica las imagenes
29         this.#renderer.render(this.#scene, this.#camera);
30     }
31 };

```

Código 3.3: Uso de la librería Three.js durante el bucle de renderizado.

En el bucle, utilizaremos el objeto *frame* descrito por primera vez en la sección 2.3 para obtener datos del dispositivo que nos servirán para calcular cómo se debe renderizar la figura 3D. En primer lugar, debemos conseguir la *pose* del móvil, que es la posición y orientación de este mismo. Para esto, es necesario utilizar el espacio de referencia que creamos en el código 3.2, y se obtendrá mediante la función *getViewerPose* de la clase *XRFrame* (lo que es nuestro objeto *frame*) utilizada en la línea 8. Esta función nos devolverá un objeto de la clase *XRVieviewerPose* [65], que representa la posición y orientación de la vista del espectador dependiendo del tipo de sesión que esté utilizando. En caso de que el usuario haya iniciado una sesión de *Realidad Virtual*, el objeto generará dos posiciones distintas, una por cada ojo, para que por cada vista se pueda realizar la renderización correcta y la sensación de tridimensionalidad sea buena. En cambio, la sesión de *Realidad Aumentada* solo tiene un «ojito», que es el objetivo de la cámara, por lo que la posición que devuelve tiene que representar la posición de este objetivo.



Figura 3.4: Ejemplo de uso de las *Google Cardboard* para visualizar contenido en *Realidad Virtual*. Imagen extraída de A Principal's Reflections (Blogspot).

Una vez tenemos esto, procedemos a utilizar su información. En la línea 9 nos aseguramos de que la función *getViewerPose* nos haya devuelto el objeto. Si es así, podremos obtener de él el objeto *XRView* [64], que contendrá la posición y orientación de nuestro objetivo, lo que nos permitirá calcular su «punto de vista»: qué puede ver, desde qué ángulo lo ve, a qué distancia lo ve, etc. Además, en la línea 13, solicitaremos a la sesión que nos dé la información sobre la proporción de imagen que debemos utilizar. La función usada (*getViewport*), recibe la vista porque, en los casos de sesiones de *Realidad Virtual*, cada vista devolverá información distinta: el objeto *XRViewport* contiene el ancho y el alto de la imagen que se generará, así como la posición de esta (punto *x* y punto *y* en la pantalla). Esto es porque, en las sesiones de *Realidad Aumentada* en los dispositivos móviles, a diferencia de la tecnología 3D utilizada en cines, se separa la vista de cada ojo en cada mitad de la pantalla del móvil para utilizar gafas de tipo *Google Cardboard*, que son lentes muy sencillas que aislan a cada ojo de la imagen que debe ver el otro y que, además, aumentan la imagen para que sea natural al ojo humano. Dado que este no será el caso, solo nos interesarán el ancho y el alto de la imagen proporcionados por el *viewport* para utilizarlo en el renderizador de *Three.js*, de manera que todas las imágenes que se generen tengan las mismas proporciones. Esto se lo explicitaremos en la línea 14 utilizando la función *setSize* sobre el objeto *renderer* que se inicializó en el código 3.2.

A continuación, se establecen las propiedades de la cámara virtual [25]: mediante la función usada en la línea 18, se establece la posición del objetivo, así como su orientación; en la línea 19 se establecerán las coordenadas de proyección, que serán las que establezcan cuánto y cómo reducirán su tamaño las figuras en relación a la distancia a la que estén [64]; y en la línea 20 se fuerza a la cámara a actualizar estos cambios, de manera que los tengamos en cuando se genere la próxima renderización con el objeto *renderer* al final de la función.

En último lugar, en la línea 24, se establece la posición de la figura [24]. En este caso, como ejemplo, se establece la posición en $(0, 0, 0)$, de manera que, si se iniciara la sesión con esta configuración, la figura estaría estática en el aire, muy cercana al punto donde se encontraba el móvil cuando se inició la sesión. En la siguiente sección, se realizará la configuración para que la retícula se encuentre siempre en el suelo y funcione como «puntero» para saber dónde se ubicarán otras figuras. Una vez establecidos los datos de esta figura, también actualizaremos sus datos usando la función *updateMatrixWorld*.

Para finalizar el procesado y juntar la parte *real* con la *virtual*, simplemente será necesario pedirle al *renderer* que renderice las figuras [51] para que esta, de manera automática al contener la referencia al canvas y a su contexto desde su inicialización en el código 3.2, las añada, finalizando así la formación del fotograma con nuestra figura añadida.

Ahora, hasta este punto, el comportamiento debería ser el siguiente: el usuario entraría en la página, pulsaría el botón de iniciar sesión de *Realidad Aumentada* y se mostraría la imagen obtenida por cámara. Cuando el sistema termine de cargar la retícula, esta se mostrará de manera fija en el espacio, muy cercano al punto donde el usuario inició la sesión. Con esto ya montado, es el momento de aumentar la funcionalidad, comenzando por el cálculo de superficies.

3.2. Cálculo de superficies, *Hit Test Results*

La siguiente característica a desarrollar es la capacidad del usuario para ubicar el avatar en el lugar que desee, estando siempre encima de una superficie, de manera que dé la sensación de estar esta reposando sobre el plano que hayamos seleccionado. Para ello se utilizará las herramientas proporcionadas por *WebXR* en conjunto con *Three.js* para poder encontrar superficies y apuntar hacia estas.

En este punto, será de gran importancia la retícula, la figura que se ha estado usando como ejemplo de modelo 3D para la carga y la inserción en este sistema de *Realidad Aumentada*. Esta figura se utilizará como puntero para que el usuario sepa en todo momento hacia dónde está apuntando, situándose esta en el punto de la superficie que se muestre exactamente en el centro de la pantalla. Para encontrar las coordenadas de dicho punto en la superficie, se utiliza *Hit Test*.

Hit Test es una técnica que consiste en encontrar intersecciones entre superficies de objetos 3D y un rayo imaginario que, en el caso de esta aplicación, surgirá desde el dispositivo móvil. En el caso de los sistemas que utilizan *Realidad Virtual*, el cálculo se realiza teniendo en cuenta que se conocen previamente los objetos que generarán las intersecciones, pero a la hora de utilizar *Realidad Aumentada*, este proceso resulta algo más complejo, al ser necesario encontrar la forma que tienen los objetos de nuestro alrededor para poder realizar los cálculos. Afortunadamente, la interfaz *WebXR* cuenta con las herramientas adecuadas para hacer estos cálculos internamente, de forma que los desarrolladores solo tengan que ocuparse de tratar la información que devuelvan los resultados del *Hit Test*.

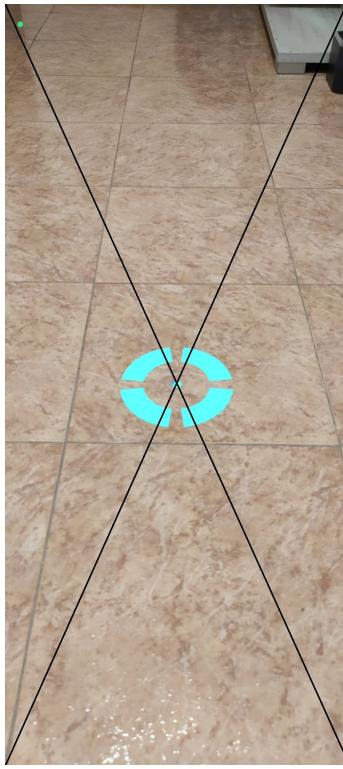


Figura 3.5: *Hit Test* devuelve la posición de la superficie que se encuentre en el centro de la pantalla.

Es importante tener en cuenta que los resultados del *Hit Test* pueden depender de la calidad de los sensores y cámaras del dispositivo. Además, la experiencia puede variar en diferentes entornos y condiciones de iluminación, por lo que las coordenadas obtenidas a través de esta técnica puede variar ampliamente dependiendo de cada casuística.

Conociendo ya esto, vamos a su aplicación en el sistema desarrollado. Para comenzar, es necesario indicar a la Sesión, cuando esta esté siendo creada, que se va a utilizar las opciones de *Hit Test* y que se requerirá cargar la configuración necesaria para su utilización. Para ello, se utiliza el objeto *SESSIONOPTIONS* que contiene información sobre cómo deberá ser la Sesión creada. En este caso, como ya se mostró anteriormente en el código 2.4, la opción a indicar es *requiredFeature*, y dentro de esta indicaremos mediante String que queremos que la sesión use '*hit-test*'.

Después de esto, dentro de la función *startLoop*, tendremos que obtener un espacio de referencia que pueda utilizar el sistema para poder calcular las intersecciones. En este caso, se utilizará un espacio de referencia de tipo *viewer* [55], ya que es un espacio de referencia centrado en ubicar el suelo. Con esto, podemos solicitar a la sesión que nos ofrezca un *Hit Test Source*. El *Hit Test Source* nos permitirá, cuando nos encontramos en el bucle, encontrar en nuestro espacio de referencia las coordenadas del punto de la superficie al que apuntamos con la cámara, tal y como se hace en la imagen 3.5 [59]. En el código 3.4, en la línea 18, justo después de la solicitud del espacio de referencia que va a utilizar y junto al otro espacio de referencia utilizado por la aplicación, se muestra la solicitud de dicho objeto, que se realiza a través de una llamada asíncrona, razón por la cual se utiliza el comando *await*.

```

1  async startLoop() {
2
3      // ...
4
5      // Session
6      this.#session = await navigator.xr.requestSession("immersive-ar", this.#SESSIONOPTIONS);
7      this.#session.updateRenderState({
8          baseLayer: new XRWebGLLayer(this.#session, this.#g1)
9      });
10
11     // Reference space
12     this.#referenceSpace = await this.#session.requestReferenceSpace('local');
13
14     // Viewer space
15     this.#viewerSpace = await this.#session.requestReferenceSpace('viewer');
16
17     // Hit test source
18     this.#hitTestSource = await this.#session.requestHitTestSource({ space: this.#viewerSpace });
19
20     //...
21
22     this.#session.addEventListener("select", this.#onTouchEvent);
23
24     // ...
25
26 }

```

Código 3.4: Solicitud de Hit Test Source a la Sesión y creación de evento.

Antes de terminar con la función *startLoop*, prepararemos un evento que servirá para aplicar la funcionalidad de la aplicación que funciona gracias a *Hit Test*. Como vemos en la línea 22 del código 3.4, mediante la función *addEventListener* de la Sesión, solicitaremos que se fije un evento de tipo «*select*», lo que significa que cada vez que se pulse la pantalla con la sesión iniciada, se lanzará la función de tipo *callback* especificada en el segundo parámetro llamada *onTouchEvent*. Concretamente, como respuesta a este evento, el sistema lanzará el siguiente código:

```

1 #onTouchEvent = (event) => {
2     if (this.#model) {
3         this.#scene.remove(this.#model)
4         this.#model.position.copy(this.#reticle.position);
5         this.#scene.add(this.#model);
6
7         // ...
8     }
9 };

```

Código 3.5: Respuesta al evento de tipo «*select*».

En el código 3.5, lanzado cada vez que el usuario toca la pantalla, eliminaremos del Escenario la retícula mediante la línea 3. A efectos prácticos, esto significa que la retícula desaparecerá por completo de nuestra *Realidad Aumentada*, siendo imposible volver a encontrarla. Sin embargo, esto solo ocurrirá si se cumple la condición de la línea 2, que comprueba si se ha cargado la variable *model*. Esta variable contiene otra figura 3D, que en este caso es el modelo de nuestro asistente virtual. Esta se comenzará a cargar a la vez que la retícula, pero debido a que este modelo 3D pesa mucho más y su carga se realiza de manera asíncrona, es necesario comprobar que se ha terminado de cargar antes de utilizarlo. Por esta razón, se lanza la comprobación de la línea 2: debido a que *JavaScript* interpreta las variables vacías como *false* y cualquier contenido (no booleano) como *true* [8], al leer esa condición, el sistema solo pasará por el condicional si la variable ha sido cargada. Esta carga asíncrona se realiza en la función *checkSessionSupported()* y se explicará con más detalle en la sección 3.2.

Una vez se evalúa afirmativamente la condición de la línea 2, como ya se ha dicho, se quitará la retícula del Escenario. Pero para continuar con la funcionalidad de la aplicación, se procede también con las otras dos líneas: nuestra aplicación debe mostrar el modelo cargado en la variable *model* en el mismo punto exacto en el que se encontraba la retícula en el momento en que el usuario pulsó sobre la pantalla, por lo que el asistente virtual deberá tener exactamente la misma que la retícula con respecto a la posición. Para eso, se usa la función de la línea 4, que se encarga de copiar completamente dicha información. Por último, como queremos que aparezca el asistente virtual en nuestra pantalla, utilizaremos la función de la línea 5 para que esta sea añadida a la Escena y, por tanto, el sistema se encargue de renderizarlo en pantalla. Esta es la manera en que intercambiaremos un modelo por otro.

Una vez obtenido todo lo necesario para iniciar nuestro *Hit Test Source*, comenzaremos con la implementación en el bucle de la función *loopFunction*. En el código 3.3, se sustituyó contenido del script original por las líneas 24 y 25 para poder explicar mejor su funcionamiento, pero ahora que se ha explicado lo necesario para llegar a esto, mostraremos en el código 3.6 el contenido que originalmente va en esas líneas:

```

1 // ...
2 // Calculo de superficies
3 const hitTestResults = frame.getHitTestResults(this.#hitTestSource);
4 if (hitTestResults.length > 0 && this.#reticle) {
5
6     const hitPose = hitTestResults[0].getPose(this.#referenceSpace);
7
8     // muestra la reticula solo si esta cargado el modelo y no esta hablando
9     if (this.#model && !this.#clipAction.isRunning())
10         this.#reticle.visible = true;
11     else
12         this.#reticle.visible = false;
13

```

```

14     this.#reticle.position.set(hitPose.transform.position.x, hitPose.transform.
15         position.y, hitPose.transform.position.z)
16     this.#reticle.updateMatrixWorld(true);
17 }
18 // ...

```

Código 3.6: Obtención de valores captados a través del Hit Test Source.

En primer lugar, en la línea 3, se solicita al objeto *frame* que devuelva el resultado de los cálculos usando el *Hit Test Source* mediante la función *getHitTestResults*. Los resultados serán unas coordenadas que indicarán la posición aproximada del punto de la superficie a la que se esté apuntando con la cámara del dispositivo. Una vez realizada esta función, se comprueba si ha devuelto algún valor mediante la condición de la línea 4. Esto se hace porque es posible que comience el bucle pero el sistema todavía no haya captado correctamente el sistema de referencia y, por tanto, todavía no pueda obtener resultados de superficies. Además, al igual que se hizo en el código 3.5, también se comprueba que se haya terminado de cargar la retícula para poder mostrarla en pantalla. Si ambas condiciones se dan, se pasa a dar a la retícula sus coordenadas: en la línea 6 se obtiene, del objeto «resultados», las coordenadas que queremos asignar a cualquier modelo tridimensional a través de la función *getPose*. En las líneas 9 y 11 se establecen las condiciones que indicarán si la retícula es o no visible durante la sesión del usuario y, finalmente, a través de la función de la línea 14, se establece la posición de la retícula cogiendo las coordenadas recibidas anteriormente del objeto «resultados». Esta acción se realizará en cada iteración para asegurar que la retícula aparece siempre en la posición hacia la que está apuntando el usuario.

Con todo esto, el usuario ya podría ubicar objetos en superficies del mundo real a través de la *Realidad Aumentada*. El siguiente punto será animar dichos objetos.

3.3. Animación de modelos

La creación de animaciones de cada modelo, en el caso de este *TFG*, se hace durante el propio diseño del modelo, punto que será tratado en el capítulo 4. Sin embargo, es importante conocer esto, ya que algunos formatos de modelos en 3D como *GLB* o *glTF* permiten almacenar dichas animaciones de manera que, al cargar los modelos, se puede utilizar también estas animaciones. En nuestra situación, son los modelos humanos los que contienen animaciones, debido a que la retícula no necesita ninguna.

Para comenzar con la animación de modelos, es necesario volver otra vez a la carga de los modelos, en la función *checkSessionSupported*. En este caso, trataremos concretamente la animación de los modelos humanos que, como se ve en el código 3.7, es algo distinto a la carga de la retícula.

```

1 #MODELLINKARRAY = ["resources/model/stormy-male.glb", "resources/model/stormie-
   female.glb"];
2 #ANIMATIONLOOP = false;
3 // ...
4
5 #checkSessionSupported = (isSupported) => {
6     if (isSupported) {
7         // Model
8         this.#modelSelected = Math.floor(Math.random() * this.#MODELLINKARRAY.
   length);
9         this.#modelLink = this.#MODELLINKARRAY[parseInt(this.#modelSelected)];
10
11    // Scene y Loader
12    this.#scene = new THREE.Scene();
13    this.#loader = new THREE.GLTFLoader();
14    this.#clock = new THREE.Clock(this.#CLOCKAUTOSTART);
15
16 // ...
17
18    // Modelo
19    this.#loader.load(this.#modelLink, function(gltf) {
20        // Figura
21        this.#model = gltf.scene;
22        this.#model.name = this.#MODELNAME;
23        this.#model.scale.set(this.#MODELSIZE.x, this.#MODELSIZE.y, this.#.
   MODELSIZE.z);
24
25        // Animacion
26        this.#mixer = new THREE.AnimationMixer(this.#model);
27
28        let jsonAnimations = gltf.animations;
29        this.#clipAction = this.#mixer.clipAction(jsonAnimations[this.#.
   MODELANIMATIONNUMBER]);
30        if (!this.#ANIMATIONLOOP) {
31            this.#clipAction.setLoop(THREE.LoopOnce); // playing the clip
   once
32            this.#clipAction.clampWhenFinished = true;
33            this.#clipAction.enable = true;
34        }
35    }.bind(this));
36 }
37 // ...
38 };

```

Código 3.7: Carga de modelo humano.

Una peculiaridad de la carga de los modelos humanos es que, de los dos modelos que tenemos, se cargará uno de ellos de manera aleatoria, sin aplicar ningún tipo de decisión más allá del uso de un número calculado aleatoriamente. Esto se realiza de la siguiente manera: en la línea 1, se almacena la constante con las ubicaciones de los dos modelos en un *array* de cadenas de texto. Más tarde, justo antes de realizar las cargas, se decide qué modelo se va a utilizar, usando para ello en la línea 7 la librería *Math* para calcular un número entero de manera aleatoria y que permita seleccionar, en la línea 8, la posición del *array* (en este caso, la posición será 0 o 1).

Una vez calculado qué modelo se va a utilizar, se pasa a la función *load* ya comentada en la sección 3.1 donde, a través de una función *callback*, se obtiene un objeto con el que se podrá manipular el modelo 3D. Ya obtenido este, se comienza a configurar: se almacena la Escena devuelta, se le asigna un nombre reconocible a su atributo *name* y se le da el tamaño deseado en el atributo *scale*.

Una vez tenemos el objeto del modelo, comenzamos con el trabajo de tratamiento de la animación. Para ello, utilizamos un objeto de la librería *Three.js*, *AnimationMixer* [39], que contiene utilidades básicas de control de la animación, como la velocidad de reproducción de esta misma o, la más importante, la capacidad de buscar el momento exacto de la animación que debe mostrarse en relación al tiempo. De este objeto obtendremos también el objeto *AnimationAction* [38] almacenado en la variable *clipAction* en la línea 29, que permite un control de la animación más parecido a los controles globales de reproducción de vídeo: pausar, reanudar, establecer un bucle, establecer la duración de la animación, etc. Para poder utilizar este último, tendremos que pasarle la animación que se va a usar. Una vez obtenido, podremos configurarlo como se hace a partir de la línea 31, donde en este caso se establece que la animación no se ejecute continuamente en bucle, se indica que la animación se desactive al terminar y, finalmente, se activa dicha animación.

Por último, antes de lanzar la función *load* de la línea 19, se prepara un objeto llamado *Clock* [40]. Este objeto sirve para contar los segundos que lleva la animación reproduciéndose y, en el momento de renderizar la figura, *AnimationMixer* la actualice para que se encuentre en la posición en la que debe estar en el tiempo indicado por el reloj. Este objeto es necesario debido a que, tal y como se comentó anteriormente en la sección 2.3, cada fotograma se lanza únicamente cuando el sistema está preparado para lanzarlo en lugar de lanzarse todos los fotogramas posibles uno tras otro. Por lo tanto, debe haber un control que tenga en cuenta los segundos de ejecución que se llevan durante cada iteración y que, con esto, sepa qué instante exacto de la animación debe renderizarse.

Una vez están estos objetos preparados, se puede comenzar a utilizar en el propio trabajo de renderizado. El primer lugar en el que se utiliza es en *onTouchEvent*:

```

1 #onTouchEvent = (event) => {
2     if (this.#model) {
3         this.#scene.remove(this.#model);
4         this.#model.position.copy(this.#reticle.position);
5         this.#scene.add(this.#model);
6     ...
7         if (!this.#clipAction.isRunning()) {
8             this.#clipAction.setDuration(this.#audioElement.duration);
9             this.#clipAction.reset();
10            this.#clipAction.play();
11        }
12    }
13};
```

Código 3.8: Evento *onTouchEvent* junto con las acciones relacionadas con la animación.

Aquí, cuando el usuario pulsa sobre la pantalla, después de ejecutarse las acciones comentadas anteriormente en el código 3.5, se comprueba primero que el modelo esté ejecutando su animación (línea 10) y, de no ser así, comienza con esta (línea 13). Antes de reproducirse, se realizan dos acciones: la primera, en la línea 11, establece la duración de la animación en relación con el audio que sonará a su vez. Esto se hace para, por un lado, fijar la velocidad a la que se ejecutará la animación (y asegurarse de que siempre va a ser la misma) y, por otro lado, para que cuadre con el mismo audio, ya que la animación se ha diseñado en relación a esta, como se comentará más adelante. La segunda acción, en la línea 12, se asegura de que, en caso de que la animación se haya detenido porque ya se había ejecutado y terminado por completo, la próxima vez que se inicie la animación se ejecute desde el punto de inicio en lugar de iniciarse desde el final o desde algún punto intermedio. De esta manera, el evento se ocuparía de iniciar la animación tanto si el usuario acaba de entrar en la *Realidad Aumentada* como si ha terminado la animación y quiere volver a reproducirla.

Por último, durante el bucle de renderizado, para actualizar la posición de la animación a la conveniente en el momento de ser renderizado, sería tan sencillo como ejecutar el siguiente fragmento de código al inicio del método:

```

1 #loopFunction = (time, frame) => {
2
3     let delta = this.#ANIMATIONSPEED * this.#clock.getDelta();
4     if (this.#mixer)
5         this.#mixer.update(delta);
6
7 // ...
8 };

```

Código 3.9: Actualización de la posición del modelo 3D antes de renderizar.

La función de la línea 5 se asegura, sin necesidad de ejecutar más acciones, de actualizar la posición del modelo. Para poder ser ejecutado, se extrae en la línea 3 el valor *delta* del objeto *Clock*, que es el tiempo en segundos que ha pasado desde que se inició el reloj. Además, en la línea 4 se comprueba que este código se ejecute únicamente si de manera previa se ha generado el objeto *mixer* durante las acciones asíncronas en la preparación del bucle, como se vio en el fragmento de código 3.7.

Con todo esto, ya dispondríamos de una aplicación de *Realidad Aumentada* que, a partir de las acciones del usuario, será capaz de posicionar modelos 3D con movimiento en superficies reales. Por último, quedaría añadir la última capa de verosimilitud mediante el sonido espacial.

3.4. Sonido espacial

Para aplicar el sonido espacial al sistema, como se mencionó en la sección 1.5, se ha utilizado la librería *Resonance Audio* [30]. Esta librería está especializada en la simulación de sonidos espaciales, de manera que un usuario con auriculares podría reconocer de dónde provienen las voces que escuchará a través de la aplicación. Otra funcionalidad que, además, se percibe sin necesidad de auriculares, es la «pérdida de volumen» en relación con la distancia al origen del sonido, con lo que seremos capaces de escuchar mejor a nuestro modelo si este está ubicado en nuestros alrededores, mientras que si se ubica a una gran distancia no seremos capaces apenas de escucharlo.

El uso de *Resonance Audio* es bastante similar al uso de la librería *Three.js* en cuanto al uso de modelos y animaciones, debido a que ambos requieren ubicar un elemento en el espacio y, además, reproducirlo. En el caso del audio, el proceso de carga de archivos es mucho más sencillo.

```
1  async startLoop() {  
2  
3      // ...  
4  
5      this.#audioContext = new AudioContext();  
6  
7      this.#resonanceAudioScene = new ResonanceAudio(this.#audioContext);  
8      this.#resonanceAudioScene.output.connect(this.#audioContext.destination);  
9      this.#resonanceAudioScene.setRoomProperties(this.#ROOAMDIMENSIONS, this.#  
10         ROOMMATERIALS);  
11  
12      this.#audioElement = document.createElement('audio');  
13      this.#audioElement.src = this.#audioLink;  
14      this.#audioElement.loop = this.#AUDIOLOOP;  
15  
16      this.#audioElementSource = this.#audioContext.createMediaElementSource(this.#  
17          audioElement);  
18  
19      this.#audioSource = this.#resonanceAudioScene.createSource();  
20      this.#audioElementSource.connect(this.#audioSource.input);  
21      this.#audioSource.setPosition(0.0, 0.0, 0.0);  
22  
23      // ...  
24      this.#session.requestAnimationFrame(this.#loopFunction);  
25  }
```

Código 3.10: Preparación del sonido espacial.

El proceso de carga y configuración del audio comienza en la función *startLoop*, donde se van preparando los objetos de la librería y especificando las opciones que se van a utilizar. Para empezar, se requiere del objeto *AudioContext* [4], que forma parte de la funcionalidad básica de *JavaScript* y permite cargar y reproducir elementos de audio. Este será inicializado en la línea 5 del código

3.10 para después poder inicializar en la línea 7 el propio objeto *ResonanceAudio* [31] incluyendo al primero como argumento del constructor. Mediante el uso de este último, la aplicación será capaz de generar el efecto de distancia y movimiento del sonido espacial. Para asignarle una salida de audio, utilizaremos la función de la línea 8.

En la línea 9, se establecen las propiedades de la habitación: largo, ancho, altura del techo y materiales de cada una de las superficies [32] (suponiendo que la habitación está formada por cuatro paredes paralelas, suelo y techo y omitiendo formas que no sean hexaédricas). En este caso, dado que no queremos simular el efecto de eco, se establece como dimensiones 0x0x0, que es la forma de decirle al sistema que la habitación no tiene una forma definida; y como materiales de paredes y techo se asigna el valor «*transparent*», lo que significa que no hay un material sólido en dichas estructuras, mientras que para el suelo se establece «*marble*».

A continuación, en la línea 11, se crea un *HTMLAudioElement* [18] a través de la función *createElement* del propio documento. Este elemento es una interfaz que permite insertar un audio en un documento HTML y que, a su vez, ofrece funciones de manipulación del mismo. Sobre este, estableceremos dos propiedades: la primera, *src* (línea 12), servirá para ubicar el origen del archivo de audio que se reproducirá en la aplicación; la segunda, *loop* (línea 13), indicará a la aplicación si, al terminar el audio, debe volver a reproducirse desde el principio automáticamente.

Finalmente, mediante el código de las líneas 15 y 17, terminamos de «conectar» el *HTMLAudioElement* con el objeto de *Resonance Audio* para que el primero sea el origen del sonido del segundo, y mediante la función de la línea 19, establecemos una posición provisional para el origen del sonido. De esta manera, tenemos preparada la configuración del sonido espacial, con lo que se puede comenzar a desarrollar la funcionalidad aplicada a esta aplicación.

En primer lugar, se definirá lo que ocurre en los casos en los que emisor y receptor del sonido se muevan. Esto se puede ver en el código 3.11, donde la funcionalidad se divide en dos partes: en la primera, representada por las filas 12, 13 y 14, se calcula la posición del usuario (que se extrae de *view.transform.matrix*) y se inserta en el objeto *THREE.Matrix4* [42], objeto que almacena una matriz de 4x4 y que representa la posición, orientación y rotación de una figura. Esta posición del usuario nos servirá para establecer la posición del receptor utilizando la función *setListenerFromMatrix*, de manera que *Resonance Audio* sepa desde dónde se está escuchando el audio reproducido. En la segunda parte, representada por la fila 17, se establece el origen del sonido utilizando la posición del modelo.

A pesar de que en el bucle se estén estableciendo coordenadas, hay que tener en cuenta que todo esto no afecta en la reproducción o pausa de los sonidos cargados. Por lo tanto, durante varias iteraciones de este, *Resonance Audio* tendrá definidas estas coordenadas a modo de «preparación» para cuando comience la reproducción.

```

1 #loopFunction = (time, frame) => {
2
3 // ...
4
5     const pose = frame.getViewerPose(this.#referenceSpace);
6     if (pose) {
7         const view = pose.views[0];
8
9         // ...
10
11         let userPosition = new THREE.Matrix4();
12         userPosition.elements = view.transform.matrix;
13         this.#resonanceAudioScene.setListenerFromMatrix(userPosition);
14
15         // ...
16
17         this.#audioSource.setFromMatrix(this.#model.matrix);
18         this.#renderer.render(this.#scene, this.#camera)
19     }
20 };

```

Código 3.11: Control del sonido espacial en el bucle.

La parte del código donde sí se comienza con la reproducción del sonido es, al igual que con la animación, en el evento *onTouchEvent*. Estas dos acciones se hacen a la vez para que el audio y la animación estén en todo momento sincronizados.

```

1 #onTouchEvent = (event) => {
2     if (this.#model) {
3 // Aqui iria parte del control de la animacion de modelo
4
5         this.#audioElement.play();
6
7 // Aqui iria otra parte del control de la animacion de modelo
8     }
9 };

```

Código 3.12: Control del sonido espacial en el evento onTouchEvent.

En este caso, solo habría una línea que comentar del código 3.12, que es la línea 5, donde se lanza la función *play* para reproducir el audio. Cabe decir que esta función no hace nada si el audio ya se está reproduciendo, por lo que, al igual que el resto del evento, está planteado para lanzar el audio solo si este ya ha terminado o todavía no se ha reproducido.

Por último, en el caso de que el usuario cierre la sesión de *Realidad Aumentada*, necesitaríamos indicar al sistema que debe finalizar también el audio, en caso de que este siga reproduciéndose. Para

esto utilizaremos el evento *onSessionEnd*, asociado a la sesión de *Realidad Aumentada* a la par que el evento *onTouchEvent*. Tal y como vemos en el código 3.13, las dos acciones que se realizan al cerrar la sesión serían, primero, pausar la reproducción y, segundo, establecer el tiempo en cero para que, si el usuario vuelve a iniciar la sesión, la reproducción siga por el mismo sitio.

```
1 #onSessionEnd = (event) => {
2   /* the session has shut down */
3   this.#audioElement.pause();
4   this.#audioElement.currentTime = 0;
5 };
```

Código 3.13: Control del sonido en el cierre de la sesión de Realidad Aumentada.

Con esto, ya podríamos simular el sonido espacial, con lo que tendríamos cubierta la última capa de *Realidad Aumentada* que queremos desarrollar.

3.5. Funcionamiento final de la Realidad Aumentada

El caso de uso principal del usuario será, generalmente, el siguiente:



Figura 3.6: El usuario accede a la landing page.

Nada más entrar en la aplicación, como se explicó anteriormente, el usuario podrá ver la *landing page*, una página estática sencilla donde aparecerá un botón que servirá para cargar los modelos necesarios y lanzar la sesión de *Realidad Aumentada* (figura 3.6).



Figura 3.7: Se solicitan permisos al usuario.

Al pulsar sobre el botón, el navegador avisará al usuario de que se pretenden utilizar funciones asociadas a la *Realidad Aumentada* (figura 3.7). En concreto, el mensaje avisa al usuario sobre la posibilidad de crear mapas 3D del entorno y el seguimiento de la posición de la cámara. Si el usuario ha dado estos permisos previamente, el sistema no lo volverá a preguntar.



Figura 3.8: La aplicación muestra las imágenes de la cámara y coloca la retícula sobre la superficie detectada.

Una vez el usuario conceda los permisos, la aplicación accederá a la cámara, donde utilizará las imágenes recibidas para calcular superficies (figura 3.8). Una vez detecte una superficie horizontal, el sistema mostrará la retícula.



Figura 3.9: Al pulsar sobre la pantalla, aparece el modelo y comienza su «discurso».

Una vez se muestre la retícula, y no antes, el usuario podrá mostrar al modelo en 3D al pulsar sobre la pantalla, apareciendo allá donde estuviese la retícula (figura 3.9). Además, el modelo comenzará a hablar y moverse y la retícula desaparecerá al realizar esta acción. Su voz aumentará o disminuirá en relación a la distancia entre la cámara y el modelo, y el sonido se percibirá por los auriculares en relación a la posición del modelo con respecto a la cámara.



Figura 3.10: La selección del modelo es completamente aleatoria.

El modelo utilizado se seleccionará de manera completamente aleatoria, pudiendo aparecer uno de dos diseños distintos (figura 3.10). Cada uno tendrá su propia voz, aunque las animaciones son similares.



Figura 3.11: El modelo podrá ser desplazado de lugar.

Una vez cargado el modelo y mostrado en la sesión, el usuario podrá volver a pulsar sobre la pantalla para mover al modelo, sin que esto interrumpa la animación o la voz del modelo (figura 3.11).



Figura 3.12: La retícula vuelve a aparecer al terminar la animación.

Una vez termine la animación y el discurso volverá a aparecer la retícula, indicando al usuario que puede volver a pulsar sobre la pantalla para que vuelva a comenzar el discurso y la animación (figura 3.12). Para cerrar la sesión de *Realidad Aumentada*, el usuario podrá pulsar el botón de «volver» del móvil, con lo que, de manera previa, se finalizarán la animación y el audio.

Capítulo 4

Diseño y modelado en 3D

Para diseñar los dos modelos tridimensionales con los que contamos, tuvimos que hacer una pequeña investigación sobre herramientas de diseño y modelado en 3D, ya que nuestra empresa no está especializada en este tipo de trabajos. Además, nuestro equipo de Marketing no podía ocuparse de estos desarrollos debido a que solo trabaja con diseños bidimensionales y a que el tiempo necesario a invertir en formaciones y en el propio modelado era considerablemente alto, por lo que no pudimos contar con su ayuda. La mayor parte de las webs que hacen comparaciones entre distintas herramientas de diseño y modelado 3D mostraban las mismas herramientas, que son las que barajamos en un principio: *Blender* [5], *Unreal Engine* [46] y *Unity* [44].

- **Blender** es una de las herramientas más famosas que existen hoy en día para diseño de modelos 3D por su potencial y por ser gratuita. Se trata de una aplicación de código libre muy extendida que dispone también de un manual de uso y tutorial [6] a disposición de todos los usuarios, además de una de las comunidades más grandes para resolución de dudas. Sus funcionalidades cubrían todas nuestras necesidades (modelado, animación, creación...), aunque la interfaz no es considerada de las mejores.
- **Unreal Engine** era otra de nuestras opciones, debido a que cuenta con una gran trayectoria tanto en series y películas de animación como en videojuegos, aunque también se utiliza en ocasiones para arquitectura y diseño. Esta aplicación es una de las más potentes del mercado y cuenta además con cursos online, documentación y tutoriales [48], así como foros para consulta de dudas [11]. Esta aplicación tiene como gran desventaja que la cantidad de funcionalidades al estar adaptado a esta gran variedad de usos hace de ella que sea más compleja de utilizar. Además, para poder exportar los modelos en el formato que necesitamos (*glTF* o *GLB*), requeríamos de un plugin a mayores [47].
- Por último, habíamos barajado también la posibilidad de utilizar **Unity**, una herramienta inicialmente creada para diseño de videojuegos, pero que también cubre todas las necesidades de modelado y animación que necesitábamos. Esta aplicación es algo menos potente que la anterior opción, pero también es una de las herramientas más utilizadas del mercado, pudiendo darnos una solución más que aceptable. *Unity* también requiere de una extensión para poder exportar modelos a *GLB* o a *glTF* [14].

Finalmente, de entre las distintas herramientas, nos decantamos por *Blender*, siendo una de las principales razones la económica: la libertad de uso de *Blender* está definida por una licencia *GNU General Public License* [7, 17], por lo que no tendríamos ningún problema al utilizarlo como empresa. Las otras dos herramientas, al contrario, requieren de licencias si se van a utilizar en entornos profesionales [45, 49], por lo que podría suponernos un gasto a mayores para una diferencia que, por nuestra baja experiencia en esta materia, no sabríamos encontrar. Además, encontramos a un compañero en nuestra empresa que había usado *Blender* anteriormente para crear pequeños videojuegos a modo de afición y que podía aconsejarnos y ayudarnos a la hora de usar dicha herramienta, punto que sería definitivo a la hora de tomar la decisión.

Una vez decidimos la herramienta a utilizar, contemplamos también la idea de utilizar una aplicación a mayores que estuviese especializada en generar figuras humanas desde cero, debido a que su creación a través de las herramientas anteriormente expuestas es muy costosa y requiere de personal más experimentado para obtener unos resultados que nos pareciesen aceptables para el proyecto.

Inicialmente, optamos por buscar herramientas que generasen modelos 3D a partir de múltiples fotografías de una misma persona desde distintos ángulos: aplicaciones como *Meshroom* [3] están orientadas a generar, no solo figuras humanas, sino también elementos del entorno como árboles, estatuas, edificios, etc. mediante fotogrametría, concepto que ellos mismos definen como la ciencia de tomar medidas a partir de fotografías.

También valoramos la posibilidad de utilizar un banco de modelos 3D previamente generados y que pudiéramos modificar a nuestro placer, como lo que ofrece *Renderpeople* [29]. Esto, en combinación con *Blender*, nos permitiría modificar los aspectos necesarios del modelo y ahorrarnos un costoso trabajo de generación desde cero.

Sin embargo, y también bajo recomendación del mismo compañero antes mencionado, finalmente nos decantamos por usar la aplicación *MakeHuman* [23]: una herramienta de código abierto y libre uso orientado a la generación de modelos 3D humanoides que permite un amplísimo abanico de características a personalizar en estos: desde aspectos básicos como su altura hasta detalles minúsculos como el tamaño del lóbulo de la oreja. Todo esto, además, lo ofrece mediante una interfaz completamente sencilla de utilizar, a lo que además hay que sumar que permite utilizar, también de manera sencilla, texturas personalizadas para el personaje, por lo que podríamos dotarlo de elementos que pudiéramos hacer completamente nuestros.

Uniendo todo esto, podemos comenzar con el trabajo de modelado.

4.1. Creación mediante *MakeHuman*

La herramienta *MakeHuman* nos ofrece, desde un inicio, una serie de pestañas que nos van informando de qué conjuntos de características podemos modificar. Para el desarrollo de este modelo, nos

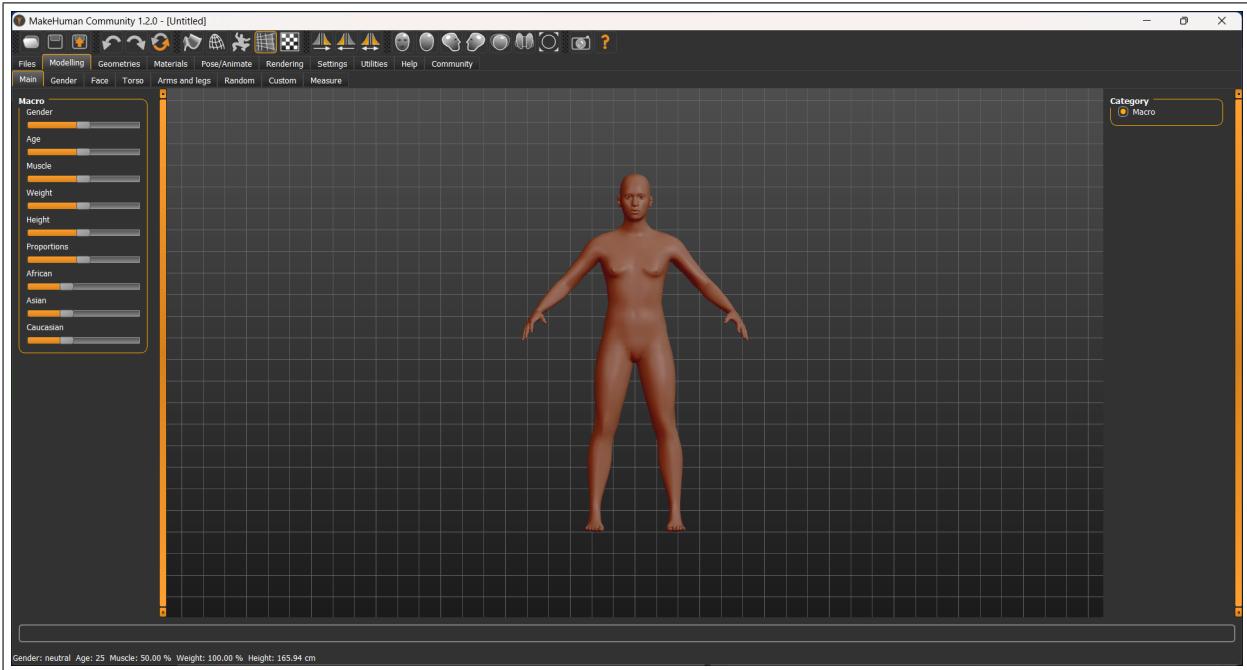


Figura 4.1: Pantalla inicial de *MakeHuman*.

vamos a centrar en solo algunas de estas pestañas. La aplicación ofrece las siguientes opciones:

- **Modelado:** para definir las proporciones y características de la figura humana del modelo.
- **Geometrías:** para modificar la forma de algunas de las estructuras del modelo, como dentadura, ojos, ropa, pelo, etc.
- **Materiales:** para modificar las texturas de las que estarán formadas las mallas que componen la figura.
- **Pose/Animación:** para establecer las opciones relacionadas con estas, como el esqueleto o la posición en la que reposa la figura.

En el caso de la aplicación desarrollada, interesaba tener dos modelos 3D distintos, siendo una figura una mujer y otra figura un hombre. Para esto, a través de la primera opción del **modelado**, *MakeHuman* permite seleccionar conceptos más globales, como el sexo, la edad o la musculatura. A través del resto de categorías dentro de la pestaña principal, la herramienta permite personalizar el resto de secciones del cuerpo, como son la cara, el torso, los brazos y las piernas, cada una de estas con sus propias subcategorías.

En la pestaña de **geometrías**, se establecen algunos detalles alternos a las características corporales. Concretamente, se definen formas que van por encima del cuerpo (camisetas, sombreros, pantalones, etc.) o aquellas estructuras corporales que, por su variabilidad, se diseñan paralelamente a este (ojos, pelo, dientes, etc.). En este último caso, a pesar de que *MakeHuman* ofrece muchas opciones, los modelos finalmente utilizados fueron creados con algunos de los recursos generados por la comunidad,

concretamente, los ojos, dientes y lengua utilizados. Esta decisión se tomó debido a la necesidad de reducir el peso de la figura todo lo posible para su uso en la web, razón por la cual se optó por escoger recursos de baja resolución para partes del cuerpo que, generalmente, no van a ser visibles o necesitan poco detalle para el uso dado. Estos últimos, sin embargo, no podían omitirse debido a que la figura, al hablar, iba a mostrar tanto lengua como dientes. Para los ojos, por otro lado, se utilizó también una geometría de baja resolución debido a que la diferencia entre esta opción y la de alta resolución no era distinguible en la aplicación de *Realidad Aumentada*.



Figura 4.2: Detalle facial del modelo.

Para descargar los recursos que publica la comunidad, existen dos opciones: a través de la web de la aplicación o a través de la propia aplicación, en la pestaña «Community», donde el recurso seleccionado se descargará en la carpeta indicada. Sin embargo, este segundo método es muy lento debido a la propia aplicación, por lo que se optó por descargar los recursos directamente de la web.

Mediante la pestaña de selección de **materiales**, previa elección de ropa de los modelos en la anterior pestaña, se retocaron dos elementos: la piel del modelo, donde usamos una de las opciones que venían en la aplicación evitando así la piel seleccionada por defecto ya que resulta poco natural; y la textura de la ropa del modelo, donde utilizamos una opción personalizada por nosotros. Para este último, se modificó la propia textura original de la ropa que se seleccionó en la aplicación, siendo este un archivo en formato .png que se puede modificar fácilmente mediante aplicaciones como *GIMP*. En esta modificación probamos varios colores para la camiseta e introdujimos el logotipo de la empresa

para darle un aire más «corporativo».



Figura 4.3: Modificación de la textura de la ropa del modelo.

Finalmente, a través de **Pose/Animación**, se establece el elemento principal para la posterior animación a través de *Blender*: el esqueleto. *MakeHuman* ofrece cuatro opciones distintas de esqueletos, donde cada una de estas opciones tiene más o menos huesos, dependiendo del uso que se le quiera dar. Las opciones más básicas controlan únicamente las extremidades, siendo un esqueleto que podría ser útil para animaciones muy básicas. Sin embargo, en este caso, queremos controlar tanto extremidades como músculos faciales, y para este caso, existen dos opciones de esqueletos: «*Default*» y «*Default no toes*». Dado que la última opción descarta una serie de huesos que no se van a utilizar en la animación de nuestra aplicación, será la opción que escogeremos, puesto que es algo más ligera.

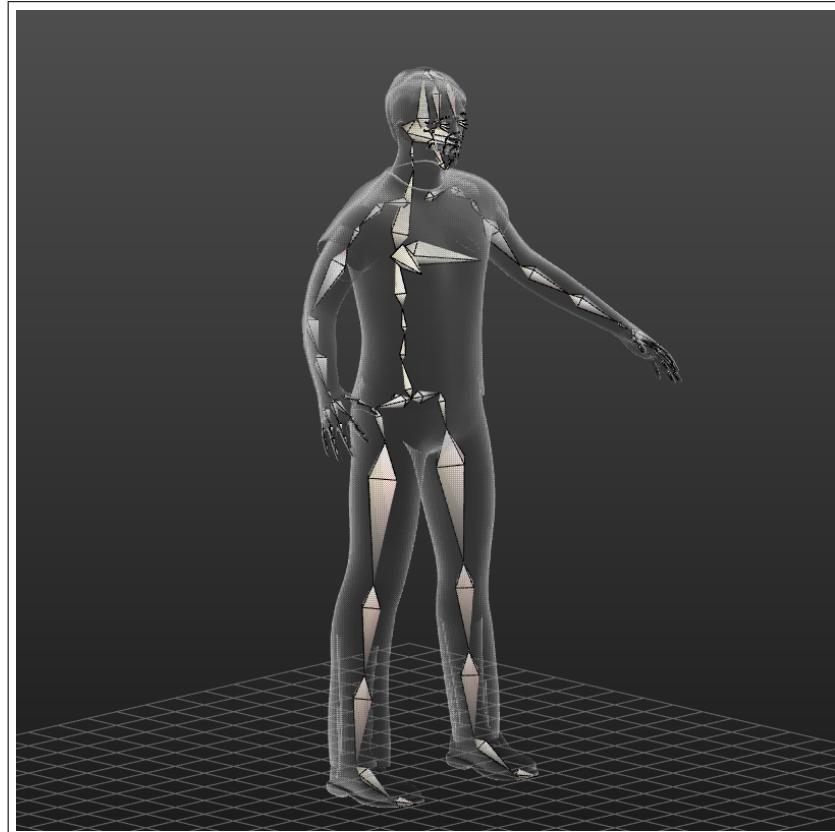


Figura 4.4: Detalle de los huesos del modelo.

Para finalizar, en esta última pestaña se le establece también al modelo una posición de reposo y una expresión facial que ofrezcan al personaje una cierta naturalidad, terminando así el diseño del modelo y el trabajo con *MakeHuman*.



Figura 4.5: Pose escogida para el modelo.

4.2. Edición con *Blender*

Para poder continuar el trabajo en *Blender*, primero es necesario exportar desde *MakeHuman* el trabajo anterior a un formato que puedan entender ambas aplicaciones. *MakeHuman* permite exportar en los formatos *Collada*, *Filmbox*, *MakeHuman Exchange*, *Wavefront obj*, *Ogre3D* y *Estereolitografía*, de los cuales *Blender* solo es capaz de utilizar tres para la importación sin usar plugins: *Collada*, *Wavefront obj* y *Estereolitografía*. Después de algunas pruebas, se llegó a la conclusión de que el mejor formato para este intercambio entre aplicaciones sería *Collada*, ya que parecía tener menos errores gráficos y mantener la mayor información posible en huesos y posición de la figura. Una vez se ha exportado la figura, solo es necesario importarlo en *Blender* usando la opción para importar archivos *Collada* en *Blender*.

4.2.1. Errores gráficos

Una vez la figura se encuentra preparada para ser modificada en la nueva herramienta, se puede comenzar con las ediciones de errores gráficos. En el momento de importación, se detectaron dos

errores en el modelo: por un lado, la textura del pelo, al reducirse la calidad en la exportación, perdía también forma y quedaba «separado» de la cabeza; por otro lado, *MakeHuman* prepara el modelo para que aquellas texturas que queden ocultas por diferentes superficies del mismo modelo se borren para ahorrar espacio, siendo sustituidas por texturas transparentes o «*alpha*», pero al modificarse la forma, algunas de estas texturas transparentes quedaban al descubierto, pudiéndose ver lo que había por dentro y por detrás del modelo a través de las transparencias generadas.

Para el error de forma en el pelo, la solución pasó por un remodelaje manual, utilizando las propias herramientas de modelaje que ofrece *Blender*. Principalmente se usaron 3 opciones de modelaje: *draw sharp*, *clay thumb* y *crease*, tres formas distintas de incidir la forma del modelo, cada uno de una manera más o menos perfilada. Estas opciones se ocupan de modificar la malla de aristas del modelo, «aplastándolas» o desplazándolas según la opción elegida.

En cuanto al error gráfico de texturas transparentes, la resolución pasó por configurar los canales *alpha* de los materiales, debido a que la configuración utilizada por *MakeHuman* no es la misma que la que utilizan el resto de procesadores de modelos 3D. Para esto, a través del menú *Layout*, se debe seleccionar la opción *Material Properties* para poder modificar las propiedades *Blend Mode* y *Shadow Mode*, a los que se les asignará el valor *Alpha Hashed*. De esta manera, las texturas *alpha* serán interpretadas correctamente y el pelo se verá «pegado» a la cabeza.

4.2.2. Animaciones

Después de solucionar los errores gráficos, se puede comenzar con las animaciones del modelo. Estas se dividen en la animación facial y la animación corporal.

Para esta sección, se utilizará mucho el término «postura», tanto para el modelo como para «huesos» del modelo. Es necesario entender dos cosas:

- El hueso, en modelado 3D, es el objeto que, al moverse, desplaza consigo un conjunto de aristas de la textura del modelo
- La postura del modelo es la forma en que está dispuesto el conjunto de sus huesos. De la misma manera, la postura de un hueso será cómo estará colocado este en relación a los huesos a los que está unido, teniendo en cuenta principalmente su posición y ángulo.

Para la animación facial, se añadió un plugin a *Blender* llamado *Rhubarb* [33] dedicado a la sincronización labial con audios. Esto se hizo a través de la opción *Add-ons* en el menú de preferencias, donde se da la opción de buscar el plugin y las opciones de instalación, aspecto muy importante ya que la configuración predeterminada está planteada para sincronización labial con textos en inglés.

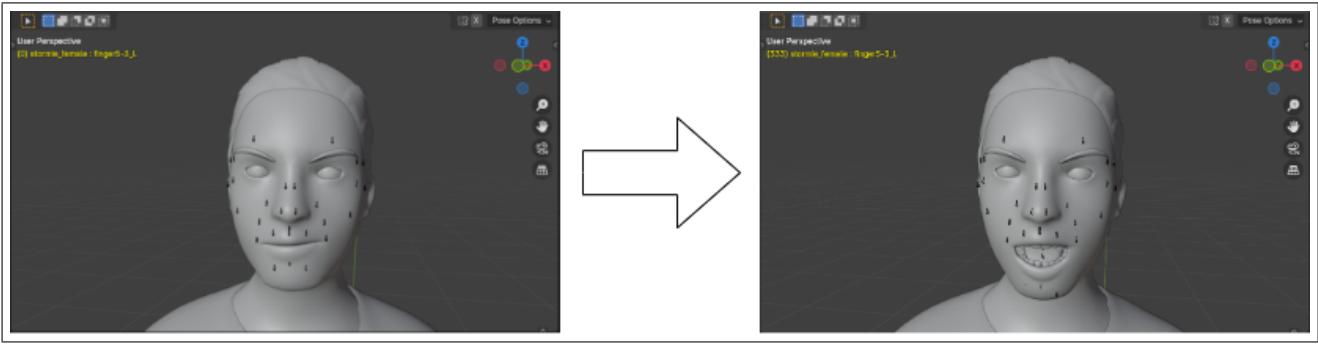


Figura 4.6: Animación generada a través de *Rhubarb*.

Para poder generar la sincronización labial mediante *Rhubarb*, es necesario hacer uso de las «librerías de posturas»: *Blender* permite almacenar posturas de un modelo para su posterior uso junto con un nombre para estos, de manera que pueden ser utilizados más tarde para diferentes usos. *Rhubarb* aprovecha esto para que el usuario almacene las formas que adquirirá la boca del modelo al pronunciar algunos de los fonemas más frecuentes, descritos estos en su documentación [34]. Una vez hecho esto, a través de las herramientas facilitadas por el plugin, se debe asociar cada postura creada con cada conjunto de fonemas mencionado en la documentación.

Finalizado el primer paso, lo siguiente consiste en proporcionarle al plugin tanto el audio que «pronunciará» el modelo como el texto que contiene el diálogo del mismo audio. Después de esto, el plugin generará la animación completa para todo el diálogo, dando un resultado muy fidedigno. En caso de que las animaciones faciales se encuentren a destiempo o no resulten fidedignas, se pueden hacer ajustes a través de las propias herramientas de *Blender*.

Por último, se generaron ciertas animaciones corporales para dotar de cierta expresividad y dinamismo al modelo. Puesto que generar animaciones muy realistas es un trabajo muy costoso, se decidió aplicar el conocimiento adquirido para utilizar la herramienta *Rhubarb* y aplicar el uso de librerías de posturas en la generación de animaciones corporales.

La ventaja de las posturas grabadas en la librería de posturas es que estas posturas se pueden guardar sobre una serie de huesos del modelo y, después, se puede aplicar sobre los mismos huesos o sobre un subconjunto de los huesos del conjunto original. Esto significa que si, por ejemplo, se ha guardado una postura de un brazo completo (desde el hombro hasta los dedos), después esta misma postura se puede volver a aplicar sobre el brazo completo o solo sobre los dedos. Esto permite generar una serie de posturas y explotarlas a la hora de generar animaciones.

Una vez se han conseguido una serie de poses válidas, se puede comenzar con la animación. Esto se hace estableciendo la postura que va a tener un hueso en un momento concreto (si no se especifica nada en ningún momento, el hueso va a permanecer en la misma postura hasta el final). Una vez insertado en la animación mediante la opción *Insert keyframe*, el mismo *Blender* rellena los fotogramas

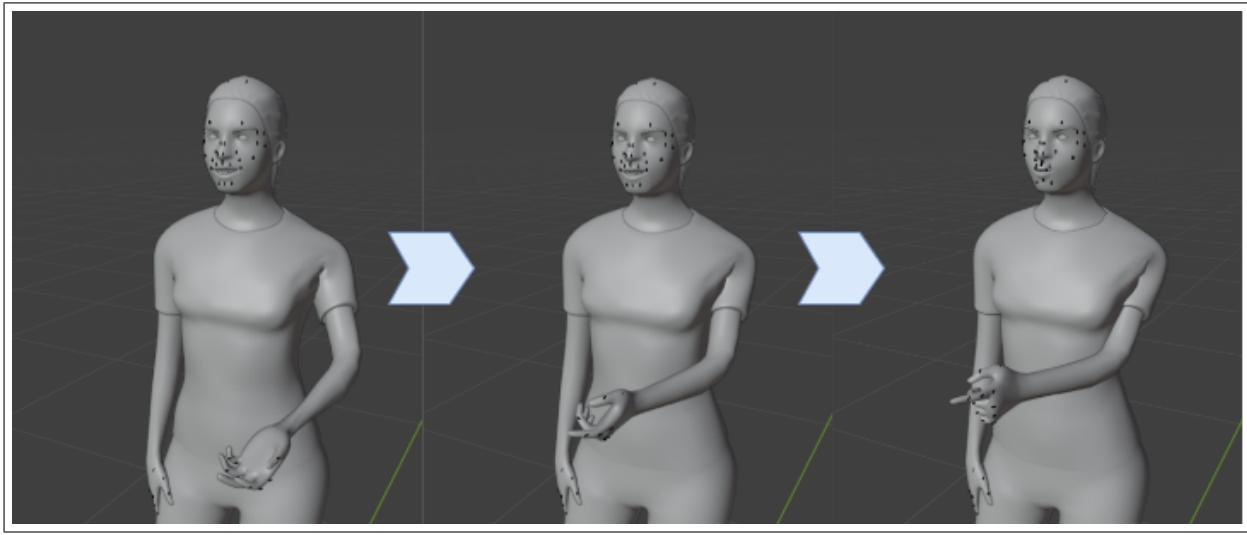


Figura 4.7: Entre dos posturas grabadas (primera y tercera) se generan posturas intermedias (segunda).

intermedios entre dos posturas distintas con las posturas que adoptará la figura entre una posición y otra para generar la sensación de continuidad y fluidez.

Con las animaciones ya terminadas, finalmente se puede exportar para poder ser utilizado en la aplicación. Para esto, basta con pulsar sobre *File, Export* y utilizar la opción *glTF 2.0*. Aquí, nos aseguraremos de que se exporta todo en un solo archivo seleccionando el formato *glTF Binary (.glb)*. Una vez contamos con el archivo, se puede ubicar en el servidor en el destino indicado para los modelos 3D.

Capítulo 5

Implementación y puesta en marcha

5.1. Alojamiento y ejecución

Para poder ejecutar la aplicación web desde cualquier dispositivo, fue necesario alojar el sistema en un servidor. Dado que la empresa *SilverStorm* era la primera interesada en el proyecto y ofrecía los recursos necesarios, permitió el uso de dos servidores de pequeña capacidad con sistema *Linux* proporcionado por *Amazon Web Services*: uno para desarrollo y demostraciones internas y otro para su posterior uso en producción. Además, para poder cumplir los requisitos de *WebXR*, que requería del uso de conexiones seguras a través de *HTTPS* [57], era necesario utilizar certificados SSL. Esto se consiguió utilizando un certificado autofirmado para el entorno de desarrollo, y un certificado para la instancia de producción firmado por *RapidSSL*, entidad utilizada para otras webs de la empresa.

Para alojar la aplicación en los servidores, se hizo uso de los comandos *git* a través de la consola de *Linux*, de manera que se pudiese acceder siempre a la última versión del repositorio. Además, para asegurar que el contenido adecuado se traspasaba siempre al servidor correcto, se mantuvo la rama principal del repositorio como única rama a utilizar en el servidor de producción, siendo esto distinto en el servidor de desarrollo, donde se utilizaba la rama de desarrollo que estuviese activa en el momento, debido a que se creaban nuevas ramas para cada tarea. Una vez se descargaba el contenido en el servidor, mediante un script de *Bash*, se movía el contenido descargado del repositorio para sobreescribir el antiguo contenido, que quedaba copiado en una carpeta aparte para ser recuperado en caso de problemas.

Una vez estuvo configurado el servidor para alojar correctamente la aplicación, se comenzó el proceso para que el usuario pudiese acceder a esta. Este consistía en ubicar un código QR en la página web de *SilverStorm* que apuntase a la *landing page* de la aplicación. Además, en caso de que el usuario accediese a la web desde un móvil, el QR también debía actuar como un hiperenlace, de manera que el usuario pudiese pulsar sobre este y acceder directamente a la aplicación de *Realidad Aumentada*. Para la generación del código QR, se utilizó la aplicación web gratuita *QR Code Generator* [27], sin embargo, debido a que es un proceso simple, actualmente existen múltiples plataformas capaces de hacer la misma tarea con resultados similares o incluso mejores de manera también gratuita.



Figura 5.1: El código QR insertado en la sección *¿Quiénes somos?*.

La ubicación del código QR en la web fue sencillo, debido a que esta está creada a través de *WordPress*. Sin embargo, es necesario mencionar que, debido a que la empresa fue posteriormente adquirida por *Thirdera*, esta web fue dada de baja, dado que todo el contenido importante sería migrado a sus webs, por lo que actualmente no se puede visitar el dominio.

Con el código QR instalado en la web, el procedimiento para iniciar la aplicación de *Realidad Aumentada* es muy sencillo: El usuario, navegando por la web desde su ordenador, vería el código QR. Este escanearía el código QR con el móvil, lo cual le redirigirá a la página principal de la aplicación, donde podrá pulsar el botón que carga la aplicación, continuando con el proceso que se comentó en el punto 3.5. Si el usuario utilizara su móvil en lugar del ordenador para navegar por la web de la empresa, el camino sería muy similar: una vez llegue al código QR, pulsaría sobre este, lo cual le redirigirá en el mismo navegador a la aplicación de *Realidad Aumentada*, pudiendo así pulsar sobre el botón e iniciar el proceso del punto 3.5.

5.2. Pruebas de campo

Tanto durante el desarrollo de la aplicación como después de ser terminado el despliegue de la misma, hubo pruebas continuas para comprobar su la calidad y controlar posibles problemas y mejoras. Uno de los grandes problemas que se dieron desde el inicio fue la calidad de los resultados de *Hit Test*, encontrando una gran variabilidad en relación con el dispositivo utilizado y con el entorno.

En primer lugar, haciendo pruebas con distintos dispositivos móviles de compañeros de trabajo y de amigos y familiares, se encontró que los dispositivos móviles con cámaras de mejor calidad eran los que mejores resultados obtenían del *Hit Test*, presumiblemente por obtener mejores imágenes de

las superficies, lo que ayudaría al sistema a interpretar y calcular los planos que después tendría que representar. Los dispositivos con cámaras de menor calidad tardaban más en encontrar superficies yo no las reconocían bien.

En segundo lugar, también se detectó que en las pruebas existían problemas inherentes al entorno. Por un lado, el material del que estuviese hecha la superficie podía generar problemas al sistema a la hora de interpretarlo debido a imperfecciones, reflejos o brillos. Esto se comprobó en suelos de baldosa donde, dependiendo del lugar, tenían diferente reflejo debido al desgaste: las zonas más desgastadas permitían al sistema calcular mejor la superficie, mientras que las que conservaban mejor el brillo y reflejaban más la luz hacían que el sistema tardase más en detectarlo. Esta diferencia llegaba a ser de 5 segundos aproximadamente en superficies «fáciles» a 2 minutos en los casos más generales en superficies «difíciles». Esto venía a su vez afectado por la cantidad de luz en el ambiente, siendo lo óptimo una cantidad de luz moderada y, a poder ser, natural, debido a que esta iluminación tiende a ser más uniforme, aunque una luz artificial moderada también es favorable para esta aplicación. En los casos de luz escasa, el sistema interpreta mal las superficies o, directamente, no las encuentra, mientras que con una luz demasiado fuerte el sistema puede no ser capaz de encontrar ninguna superficie, dependiendo del material de esta.

Las pruebas más reproducidas fueron las siguientes:

1. Utilizando un Xiaomi Redmi Note 9 Pro, con una cámara de 64MP, en una habitación con iluminación natural moderada y sobre un suelo de baldosa, el sistema tarda aproximadamente 5 segundos en encontrar la superficie.
2. Utilizando un Samsung Galaxy A10, con una cámara de 13MP, en una habitación con iluminación artificial alta y sobre un suelo laminado de madera, el sistema tarda aproximadamente 40 segundos en encontrar la superficie.

5.3. Problemas y errores

A lo largo del desarrollo de la aplicación, han ido surgiendo múltiples errores, algunos más fáciles y otros más difíciles de depurar. La depuración de estos errores resultó generalmente complicada debido a que la aplicación debe ser abierta siempre a través de móvil, lo que obliga a configurar el móvil y conectarlo a través de cable a un ordenador para poder utilizar la consola del navegador. Además, los errores de programación que se encontraran dentro del bucle principal serían especialmente problemáticos, porque no existe forma perfecta de ver la información necesaria de manera sencilla.

Sin embargo, la mayoría de los problemas que más bloquearon el desarrollo de la aplicación no estaban relacionados con *bugs* surgidos durante el proceso. Después de un análisis de los problemas, estos son los considerados como más graves:

- La documentación de la librería *Three.js* resulta, en la gran mayoría de casos, insuficiente o deficientemente explicada. Esto ha afectado múltiples veces al avance del desarrollo, debido a que hay objetos que pertenecen a la librería pero que no se explica qué representan o cómo funcionan. Un caso muy particular es la función de carga de modelos *glTF*, que solo viene explicada a través de un ejemplo, pero no hay ningún detalle sobre los objetos que devuelve dicha función. La falta de detalles en esta documentación ha obligado a descubrir muchos aspectos de esta librería a base de ensayo y error o a través de foros no oficiales y de dudosa calidad en Internet.
- De manera similar, la documentación de *Resonance Audio* también ha resultado ser muy escasa, constando solo de una página con el objeto principal y una lista de atributos y funciones de esta con una descripción muy básica y sin detallar, obligando prácticamente a basarse en los ejemplos que añade la página para poder obtener información mínima útil.
- Un problema relacionado con el código y que requirió de amplia y constante depuración fue la compatibilidad entre *Three.js* y *Resonance Audio* a través de los objetos de representación de posiciones en tres dimensiones. Cada librería utiliza su propio objeto de posición y requiere de sus propias conversiones, lo que obligó a buscar la forma de compatibilizarlo, pero la dificultad para depurar objetos que se encuentran en el bucle de procesado de imágenes y la escasez de documentación al respecto hizo que los pasos para hacer compatible la información de ambas librerías fuera extremadamente lenta.
- A pesar de que *WebXR* sí cuenta con documentación fiable y muy extensa a través de la web de desarrolladores de *Mozilla*, la explicación para lanzar un sistema básico por parte de la web oficial de Google resulta muy escasa en muchos aspectos, omitiendo información que puede ser crucial en desarrollos más complejos. Un ejemplo de esto es la totalmente inexistente explicación de cuándo se solicita la imagen a la cámara y cómo se envía al *canvas* generado. Además, *Google* dispone de varios tutoriales distintos para lanzar un sistema básico a través de *WebXR* donde se ve información distinta y, a veces, contradictoria, lo que tampoco ayudó a lanzar una primera versión de la aplicación.

En todos estos casos, la solución final fue desarrollos a base de ensayo y error apoyándose en blogs y webs de desarrolladores independientes de los que se podían extraer datos que podrían ser importantes.

Capítulo 6

Consideraciones y futuros pasos

6.1. Pasos fallidos

Es importante remarcar que este trabajo forma parte de una colaboración con la empresa *Thirdera*, originalmente *SilverStorm*, y que algunos recursos fueron ofrecidos por esta misma para favorecer el curso del desarrollo y su correcta finalización, así como asegurar el producto mínimo viable.

Dicho esto, también hay que tener en cuenta que la empresa no está especializada en desarrollos de este tipo, y por lo tanto todos éramos nuevos usando las tecnologías aplicadas en el proyecto. Por eso, pese a haber gestionado riesgos y organizado el trabajo, inevitablemente surgieron problemas que no pudimos abordar por diferentes motivos. Los más importantes fueron los siguientes:

- Aunque se pensó desde un primer momento en la compatibilidad entre distintos dispositivos móviles, lo cierto es que fue un punto que no se consiguió. Originalmente se pensó que el uso de *WebXRPolyfill* permitiría utilizar la *Realidad Aumentada* en diferentes navegadores debido a que hacen mención a compatibilidad en navegadores en su documentación [56] (a pesar de que no hacen mucho énfasis en ello). Pero, finalmente, resultó que esta actuaba como una «extensión» para aplicar retrocompatibilidad en caso de que el usuario utilizara un navegador *Google Chrome* para móviles con una versión antigua. La compatibilidad con otras plataformas fue finalmente descartada debido a que no encontramos una solución alternativa.
- A la hora de generar las animaciones corporales de los modelos, se planteó utilizar la herramienta web *Kinetix*: una aplicación que capta los movimientos de una persona, los traduce a movimientos en el esqueleto de un modelo 3D y devuelve el modelo con la animación completa [21]. Por desgracia, esta aplicación no reconoce correctamente los esqueletos de modelos 3D que no hayan sido generados por la misma aplicación, por lo que el resultado finalmente era totalmente incorrecto y no se podía considerar utilizable. Debido al consumo de tiempo que implicaba buscar otra herramienta y aprender a utilizarla, se cambió este paso por el desarrollo «a mano» de las animaciones corporales del modelo a través de *Blender*.
- Cuando se generaron originalmente los modelos, se crearon directamente con la pose de reposo que tienen actualmente cuando no se están moviendo. Sin embargo, en numerosas webs de

diseño de modelos 3D se recomienda encarecidamente generarlos en pose A o en pose T, es decir, con todas las extremidades estiradas y con los brazos estirados hacia las piernas (pose A) o estirados en pose de cruz (pose T). Esto se hace para que los movimientos de los huesos sean mucho más precisos a la hora de mover cada miembro. Además, en este caso, hubiera favorecido que las mallas de los modelos se deformaran menos, ya que *MakeHuman*, al generar un modelo con una pose concreta, también reestructura las mallas para acomodarlas a la postura elegida. El elegir una postura distinta a las poses A y T como postura de reposo ha generado algunas deformidades en brazos y dedos a la hora de moverlos y la imposibilidad de mover de manera normal los labios de los modelos.

- Una de las características que contiene *WebXR* y que no aparecía en ninguno de los tutoriales de primeros pasos de Google era las capas de *renderState*. Estas capas están preparadas para que cada una contenga una imagen, de manera que se superpongan las distintas capas y se genere el efecto de *Realidad Aumentada*. Esto favorecería un desarrollo más limpio y organizado del código, pero no se utilizó debido al desconocimiento del mismo. En su lugar, se utilizó una sola capa en la que se implantaban todas las imágenes de manera ordenada.

6.2. Futuros pasos

Durante el desarrollo y después de finalizar el mismo, se encontraron muchos puntos que necesitan una corrección o que pueden mejorarse. Algunos de estos requieren de mejoras en las propias interfaces utilizadas, por lo que su mejora pasaría por esperar a actualizaciones o sustituir completamente la interfaz, pero otras son puntos mejorables que se han ido encontrando durante el proyecto y posponiendo hasta después de obtener un producto mínimo viable. Algunos de los puntos encontrados son los siguientes:

- En el código original, se utiliza la opción *preserveDrawingBuffer* como una de las opciones a añadir al renderizador. Esto se utilizó debido a que viene así explicado en varios tutoriales, pero en un issue del repositorio de GitHub de *WebXR* indican que esta opción es totalmente ignorada por la herramienta, por lo que no tiene ningún sentido mantener esa opción en el objeto [15]. Este hilo es el único lugar donde se indica esta información, hecho por el cuál costó tanto encontrar este dato.
- La detección de superficies resulta muy imprecisa y depende, como se comentó en la sección 5.2, de la iluminación y del tipo de superficie. Según parece, la detección de superficies se puede configurar, por lo que es posible que se pueda mejorar manualmente, aunque esto requiere de mucha investigación.
- Como se comentó en la sección anterior, la aplicación no es multiplataformas, sino que funciona solo en determinados navegadores específicos determinados por la propia librería *WebXR*. En futuros pasos, sería ideal ampliar esta compatibilidad, ya sea configurando el sistema actual de la manera adecuada o cambiando de librería de *Realidad Aumentada*.

- Los pasos para añadir nuevos modelos, animaciones y voces se encuentran muy medidos y son reproducibles. Sin embargo, sería ideal simplificarlos aún más para poder minimizar el esfuerzo de generar nuevo contenido para la aplicación.
- A pesar de que el código del sistema está repartido para repartir responsabilidades, hubiera sido posible diferenciarlo más de manera que los diferentes trabajos, especialmente en el bucle de procesado, estuviesen mucho más diferenciados y fuese más legible. Esto requeriría de una refactorización intensa, pero que puede afectar positivamente en el mantenimiento de la propia aplicación.
- Algunos sistemas basados en *WebXR* son capaces de detectar objetos que se encuentran entre el modelo y la cámara. Al reconocer esto, dichos sistemas son capaces de «ocultar» el modelo detrás del objeto, cosa que el sistema actual no es capaz de hacer. Sería ideal encontrar la manera de que los modelos se puedan «esconder» detrás de objetos reales.
- La orientación inicial del modelo, a día de hoy, depende del momento de carga del mismo: este mirará siempre hacia el punto en que se encontraba el usuario en el momento exacto de carga. El sistema sería mucho más vistoso si, cada vez que se pulsara sobre la pantalla, el modelo se girase hacia la cámara del usuario.
- Como último punto detectado, la accesibilidad de la aplicación mejoraría si esta mostrase al usuario las instrucciones para su uso en el momento correcto. Es decir: que la aplicación mostrase mensajes como «mueva la cámara para detectar superficies», «toque la pantalla para comenzar la animación» o «toque de nuevo la pantalla para reiniciar la aplicación» en los momentos indicados para que, en todo momento, el usuario sepa cómo utilizar este sistema.

Bibliografía

- [1] P. Santos. *¿Cómo se organizan los equipos de ingeniería de software modernos?* Organizado por la Escuela de Ingeniería Informática de la Universidad de Valladolid y patrocinado por Unity3D. 8 de nov. de 2021. URL: <https://www.inf.uva.es/2021/11/08/organizacion-de-roles-en-empresas-tecnologicas-y-gestion-de-equipos-y-proyectos-el-caso-de-unity-3d-actividad-coordinada-con-las-asignaturas-de-planificacion-y-gestion-de-proyectos/>.
- [2] Jianghao Xiong et al. «Augmented reality and virtual reality displays: emerging technologies and future perspectives». En: *Light: Science & Applications* 10.1 (oct. de 2021), pág. 216. ISSN: 2047-7538. DOI: 10.1038/s41377-021-00658-8. URL: <https://doi.org/10.1038/s41377-021-00658-8>.
- [3] *AliceVision: Meshroom*. URL: <https://alicevision.org/#meshroom>.
- [4] *AudioContext*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/AudioContext>.
- [5] *Blender*. URL: <https://www.blender.org/about/>.
- [6] *Blender 3.0 - Manual de Referencia*. URL: <https://docs.blender.org/manual/es/3.0/>.
- [7] *Blender: Blender is Free Software*. URL: <https://www.blender.org/about/license/>.
- [8] *Boolean*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean.
- [9] *Browser compatibility*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API#browser_compatibility.
- [10] *Callback function*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Callback_function.
- [11] *Epic Developer Community Forums: Unreal Forum*. URL: <https://forums.unrealengine.com/categories?tag=unreal-engine>.
- [12] *Framebuffer*. URL: <https://en.wikipedia.org/wiki/Framebuffer>.
- [13] *Fundamental Concepts*. URL: <https://resonance-audio.github.io/resonance-audio/discover/concepts.html>.
- [14] *GitHub: KhronosGroup/UnityGLTF*. URL: <https://github.com/KhronosGroup/UnityGLTF/>.
- [15] *GitHub: WebXR - Clarify semantics of preserveDrawingBuffer*. URL: <https://github.com/immersive-web/webxr/issues/891>.

- [16] *GLTFLoader*. URL: <https://threejs.org/docs/#examples/en/loaders/GLTFLoader>.
- [17] *GNU General Public License*. URL: <https://www.gnu.org/licenses/gpl-3.0.html>.
- [18] *HTMLAudioElement*. URL: <https://developer.mozilla.org/es/docs/Web/API/HTMLAudioElement>.
- [19] *HTMLCanvasElement: getContext() method*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/getContext>.
- [20] IAT. *Principales diferencias entre realidad aumentada y realidad virtual*. URL: <https://iat.es/tecnologias/realidad-aumentada/diferencias-realidad-virtual/>.
- [21] *Kinetix*. URL: <https://kinetix.tech/>.
- [22] *Load 3D Models in glTF Format: Data Returned by the GLTFLoader*. URL: <https://discoverthreejs.com/book/first-steps/load-models/#returned-glft-data>.
- [23] *MakeHuman Community*. URL: <http://www.makehumancommunity.org/>.
- [24] *Object3D*. URL: <https://threejs.org/docs/#api/en/core/Object3D>.
- [25] *PerspectiveCamera*. URL: <https://threejs.org/docs/index.html?q=perspe#api/en/cameras/PerspectiveCamera>.
- [26] Shelley Pursell. *Metodología Agile: qué es y cómo aplicarla a tu proyecto*. URL: <https://blog.hubspot.es/marketing/metodologia-agile>.
- [27] *QR Code Generator*. URL: <https://es.qr-code-generator.com/>.
- [28] Dan Radigan. *Kanban: Aplicación de la metodología kanban en el desarrollo de software*. URL: <https://www.atlassian.com/es/agile/kanban>.
- [29] *Renderpeople*. URL: <https://renderpeople.com/>.
- [30] *Resonance Audio*. URL: <https://resonance-audio.github.io/resonance-audio/>.
- [31] *Resonance Audio Unity SDK API Reference: ResonanceAudio*. URL: <https://resonance-audio.github.io/resonance-audio/reference/web/ResonanceAudio>.
- [32] *Resonance Audio Unity SDK API Reference: Utils*. URL: <https://resonance-audio.github.io/resonance-audio/reference/web/Utils>.
- [33] *Rhubarb Lip Sync*. URL: <https://github.com/DanielSWolf/rhubarb-lip-sync>.
- [34] *Rhubarb Lip Sync: mouth shapes*. URL: <https://github.com/DanielSWolf/rhubarb-lip-sync#mouth-shapes>.
- [35] *Sound localization*. URL: https://en.wikipedia.org/wiki/Sound_localization#Human_auditory_system.
- [36] *The Model-View-Presenter (MVP) Pattern*. URL: [https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff649571\(v=pandp.10\)?redirectedfrom=MSDN](https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff649571(v=pandp.10)?redirectedfrom=MSDN).
- [37] *Three.js*. URL: <https://threejs.org/>.
- [38] *Three.js: AnimationAction*. URL: <https://threejs.org/docs/index.html#api/en/animation/AnimationAction>.

- [39] *Three.js: AnimationMixer*. URL: <https://threejs.org/docs/#api/en/animation/AnimationMixer>.
- [40] *Three.js: Clock*. URL: <https://threejs.org/docs/#api/en/core/Clock>.
- [41] *Three.js: Loading 3D models*. URL: <https://threejs.org/docs/index.html?q=load#manual/en/introduction>Loading-3D-models>.
- [42] *Three.js: Matrix4*. URL: <https://threejs.org/docs/index.html#api/en/math/Matrix4>.
- [43] *Three.js: Scene*. URL: <https://threejs.org/docs/index.html?q=sce#api/en/scenes/Scene>.
- [44] *Unity*. URL: <https://unity.com/>.
- [45] *Unity: plans and pricing*. URL: <https://unity.com/pricing#plans-individualsand-teams>.
- [46] *Unreal Engine*. URL: <https://www.unrealengine.com/>.
- [47] *Unreal Engine Marketplace: glTF Exporter*. URL: <https://www.unrealengine.com/marketplace/en-US/product/gltf-exporter>.
- [48] *Unreal Engine: Learn Unreal Engine*. URL: <https://www.unrealengine.com/en-US/learn>.
- [49] *Unreal Engine: Licensing options*. URL: <https://www.unrealengine.com/en-US/license>.
- [50] *Viewpoints and viewers: Simulating cameras in WebXR. Virtual cameras*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API/Cameras#virtual_cameras.
- [51] *WebGLRenderer*. URL: <https://threejs.org/docs/#api/en/renderers/WebGLRenderer>.
- [52] *WebGLRenderingContext*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext>.
- [53] *WebGLRenderingContext: bindFramebuffer() method*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/bindFramebuffer>.
- [54] *WebXR*. URL: <https://immersiveweb.dev/>.
- [55] *WebXR Device API - Spatial Tracking: Reference spaces*. URL: <https://github.com/webxr/spatial-tracking-explainer.html#reference-spaces>.
- [56] *WebXR Polyfill*. URL: <https://github.com/immersive-web/webxr-polyfill#webxr-polyfill>.
- [57] *WebXR requirements*. URL: <https://developers.google.com/ar/develop/webxr/requirements>.
- [58] *Wikipedia: Three.js*. URL: <https://es.wikipedia.org/wiki/Three.js>.
- [59] *XRHitTestSource*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRHitTestSource>.
- [60] *XRSession*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRSession>.
- [61] *XRSession: requestAnimationFrame() method*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRSession/requestAnimationFrame>.
- [62] *XRSession: updateRenderState() method*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRSession/updateRenderState>.

- [63] *XRSSystem*: *requestSession()* method. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRSSystem/requestSession>.
- [64] *XRView*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRView>.
- [65] *XRViewerPose*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRViewerPose>.
- [66] *XRWebGLLayer*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XRWebGLLayer>.