TRABAJO FIN DE GRADO

GRADO EN INGENIERIA INFORMATICA

EditAR (working tittle)

Editor de escenas de Realidad Aumentada

Autor
Pablo Millán Cubero

Director
Francisco Javier Melero Rus

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Granada, Julio de 2023

EditAR
Editor 3D web para escenas de Realidad Aumentada

Pablo Millán Cubero

Palabras clave: *software libre, informática gráfica, Typescript, React, Firebase, NodeJS, Express, Android Studio, Kotlin, Sceneview, GLB*

Resumen

Las tecnologías de Realidad Aumentada nos permiten combinar elementos virtuales y reales en tiempo real pudiendo así crear experiencias aplicables a muchos campos; filtros con maquillaje para fotos en redes sociales, previsualizar cómo quedaría un mueble en tu propio salón, dibujar en un partido de fútbol la trayectoria que recorrió la pelota en una repetición, o incluso en videojuegos como Pokemon GO.

Con el avance de los teléfonos móviles de todas las gamas en cuanto a cámaras y capacidad de procesamiento en la última década estas experiencias han pasado a estar al alcance de todo el mundo que posea un dispositivo básico con resultados muy vistosos. Sin embargo en la mayoría de experiencias de Realidad Aumentada el usuario ocupa el rol de consumidor pasivo del contenido que generan los desarrolladores.

Se propone entonces desarrollar una aplicación web en el que el usuario pueda cargar múltiples modelos 3D con textura y animaciones, aplicarles a estos transformaciones como translaciones, rotaciones y escalado, reproducir animaciones y cargar pistas de audio. Todo ello desde un sencillo e intuitivo editor que puede usar cualquiera en el que no haga falta tener conocimientos previos. La escena 3D que se cree podrá posteriormente descargarse o guardarse en un servidor web asociado a una cuenta de usuario. El usuario podrá además, desde una aplicación Android cargar sus escenas creadas y reproducirlas, pudiendo visualizarlas en su entorno a través de la cámara del dispositivo móvil.

EditAR
Web editor for Augmented Reality scenes

Pablo Millán Cubero

Keywords: *software libre, informática gráfica, Typescript, React, Firebase, NodeJS, Express, Android Studio, Kotlin, Sceneview, GLB*

Abstract

Augmented Reality technologies let us merge virtual elements with camera footage in real time, letting us create experiences for many fields; makeup filters for social network photos, preview new furniture for your livingroom, draw the trayectory of the ball in a football match replay, or even in videogames like Pokemon GO.

With the improvement of smartphone's cameras and processing power in the last decade, this experiences are know within everybody's reach if you have a basic device. However, the mayority of Augmented Reality experiences are based in the pasive comsumption of content pregenerated by developers.

The proposed web application lets the user load multiple 3D models with textures and animations, apply transformations like translations, rotations and scales, and play audio tracks. All this in an easy and intuitive editor that anyone can use without previous knowledge. The resultant 3D scene can be either downloaded or uploaded to a web server asociated to an user account. Aditionaly, the user can load the created scenes within an Android app and play them with their device's camera, placing it in their environment.

D. **Tutora/e(s)**, Profesor(a) del ...

Informo:

Que el presente trabajo, titulado ***Chief***, ha sido realizado bajo mi supervisión por **Estudiante**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a Junio de 2018.

El/la director(a)/es:

(nombre completo tutor/a/es)

Pablo Millán Cubero

Agradecimientos

A mis padres por permitirme estudiar esta carrera en Granada y apoyarme.

A mis abuelos, en especial a mi abuela Elena, que tanto presume de "su nieto el ingeniero".

A mis tíos "los franceses".

A mis amigos del Krustáceo Krujiente, Juan, Antonio, y mis compañeros de trinchera Pepe, Finn, Moisés y Pablo.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción y fundamentos

En este capítulo se dará una primera introducción a la *Realidad Aumentada* y a la idea del proyecto, además de describir una serie de conceptos y tecnologías fundamentales para entender el trabajo y seguir con la lectura del resto del documento.

1.1. Descripción general y motivación

El término de Realidad Aumentada comprende al conjunto de técnicas y tecnologías que permiten la superposición e interacción de elementos virtuales sobre la realidad física a través de un dispositivo electrónico. Esta tecnología se ha aplicado a una gran variedad de ámbitos, tales como en la creación de filtros faciales para fotos en redes sociales como Instagram, dibujar la trayectoria que ha seguido el balón en la repetición de una jugada de fútbol, muestra de distintos gráficos y maquetas en los telediarios, previsualización de mobiliario en una habitación e incluso videojuegos como Pokémon GO.

El avance en las cámaras y procesadores de los teléfonos móviles en la última década ha facilitado que casi cualquier persona tenga en el bolsillo un dispositivo capaz de reproducir experiencias muy vistosas de Realidad Aumentada. Sin embargo, en la mayoría de casos se trata de aplicaciones en las que el usuario consume pasivamente contenido pre generado por los desarrolladores, sin la capacidad de crear ellos mismos estas experiencias.

Se propone entonces desarrollar en primer lugar una aplicación web que permita al usuario la composición de textitescenas 3D para uso en Realidad Aumentada. Cuando hablamos de escena nos referimos una composición de uno o más modelos 3D con una disposición concreta. Por ejemplo podríamos colocar el modelo de un tobogán junto al de un columpio y un balancín, formando el

conjunto la escena de un parque. La aplicación consistirá principalmente de un editor sencillo e intuitivo que permita cargar modelos 3D con texturas y aplicarles transformaciones para colocarlos en una escena. Adicionalmente, se podrá añadir a la escena una pista de audio y reproducir animaciones de los modelos. Se podrá guardar la escena en un servidor web en una colección de escenas generadas asociadas a una cuenta de usuario, desde la cual se podrán modificar o eliminar.

Adicionalmente, el usuario podrá iniciar sesión en una app móvil Android para acceder a su lista de escenas creadas y podrá reproducir cualquiera de ellas en Realidad Aumentad con su dispositivo móvil.

Las principales motivaciones para elegir este como mi trabajo de fin de grado fue mi alto grado de interés por los gráficos 3D, mis conocimientos previos en tecnologías que acabaría usando como three.js o Android Studio, y que es un proyecto relativamente multidisciplinar, requiriendo del desarrollo de una aplicación web, una aplicación Android y un servidor web que los conecte, montando así un pequeño ecosistema y con suerte dándome una intuición de lo que puede ser realizar un proyecto parecido en un entorno laboral real.

1.2. Fundamentos

A continuación se van a describir una serie de conceptos y tecnologías esenciales para la correcta comprensión del proyecto.

1.2.1. Realidad Aumentada

La *Realidad Aumentada* (*Augmented Reality* en inglés, *RA* de aquí en adelante) hace referencia al conjunto de técnicas y tecnologías que permiten crear una experiencia en la que se combina el mundo real con contenido generado por computadores. Aunque el uso más extendido es únicamente visual y auditorio, puede apelar a más sentidos como el háptico, somatosensorial y olfativo. Se puede decir que un sistema *RA* debe incorporar tres elementos básicos: una combinación del mundo real y el virtual, interacción en tiempo real y un correcto registro de la posición y movimientos de objetos tanto reales como virtuales.

Existen otros términos acuñados que son a grandes rasgos sinónimos de la *RA* como *Mixed Reality*. No debe confundirse con la *Realidad Virtual*, otro ámbito en el que el objetivo es sumergir al usuario en un mundo virtual enajenándolo de sus sentidos con el mundo real, usualmente con unas gafas envolventes diseñadas específicamente para realidad virtual. Todas estas disciplinas están recogidas bajo el término paraguas de la *Extended Reality* (*XR*).

A continuación se van a describir los tipos de escenas que podemos encontrarnos en el contexto de la Realidad Aumentada.

Figura 1.1: Ejemplo de Realidad Aumentada en el videojuego Pokémon Go

- **Posicionamiento en superficies**: Es quizás el ejemplo más sencillo tanto conceptualmente como técnicamente. El dispositivo intentará analizar sus alrededores a través de la cámara e intentará detectar superficies planas como podría ser una el suelo, una mesa o una pared. Colocará objetos en puntos concretos de estas superficies, y al mover el dispositivo esos objetos deberán visualizarse como si estuvieran colgados o colocados en dichas instancias del mundo real. Existe todavía a día de hoy una limitación importante a la hora de realizar estas escenas; si la superficie es demasiado lisa y uniforme, el dispositivo la reconocerá con gran probabilidad como un color plano, y no tendrá la capacidad de identificarla como una superficie, haciendo que no pueda llegar a colocar nada.

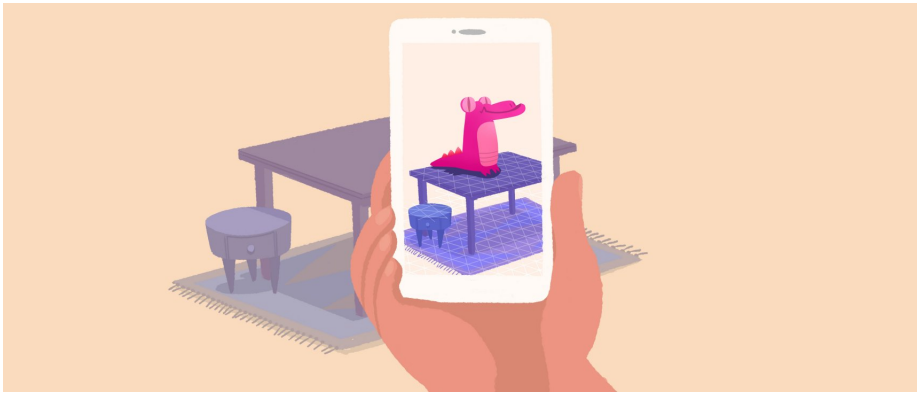

Figura 1.2: Ejemplo de posicionamiento en superficies de la documentación de *ARCore* [6]

- **Imágenes aumentadas**: En este tipo de escena, cada modelo o conjunto de modelos tiene asociada una imagen bidimensional a la que denominamos **marcador** o **activador**. Cuando se inicie, el dispositivo tratará de reconocer esta imagen en el mundo real. Una vez identificada se reproducirán sobre la imagen (tratándola como una superficie) o en una posición relativa a la imagen, los modelos pertinentes que se hayan definido. Es posible definir distintos objetos asociados a distintos marcadores y que solo se reproduzcan los elementos pertinentes.

Figura 1.3: Ejemplo de escena de imágenes aumentadas de la web de *Pictarize Studio* [1]

- **Escenas geoespaciales**: Son escenas en las que los objetos no se posicionan en una superficie, si no en unas coordenadas GPS. El usuario solo podrá ver la escena si se encuentra a una latitud, longitud y altura cercanos a aquella que se definieran para la escena. Para ello logicamente es esencial disponer de conexión a internet y que el dispositivo tenga acceso a las coordenadas actuales del usuario durante la ejecución. Las localización GPS de los dispositivos móviles no suele ser del todo precisa, así que usando solo estas es posible que si queremos, por ejemplo, colocar un modelo 3D que represente cómo era la *Alhambra de Granada* cuando se construyó justo en la puerta del edificio, es posible que algunos usuarios la vean algunos metros fuera de lugar, o incluso flotando en el aire. Para poder tener algo más de precisión, algunas APIs geoespaciales como la de *ARCore* [6] permiten extraer información de *Street View*, el visor en primera persona de *Google Maps* para reconocer a través de puntos clave,

fachadas de edificios y calles que hayan sido captadas previamente por este servicio, resultando así en una mejor aproximación a la verdadera posición del usuario y por tanto un posicionamiento más preciso de las escenas.

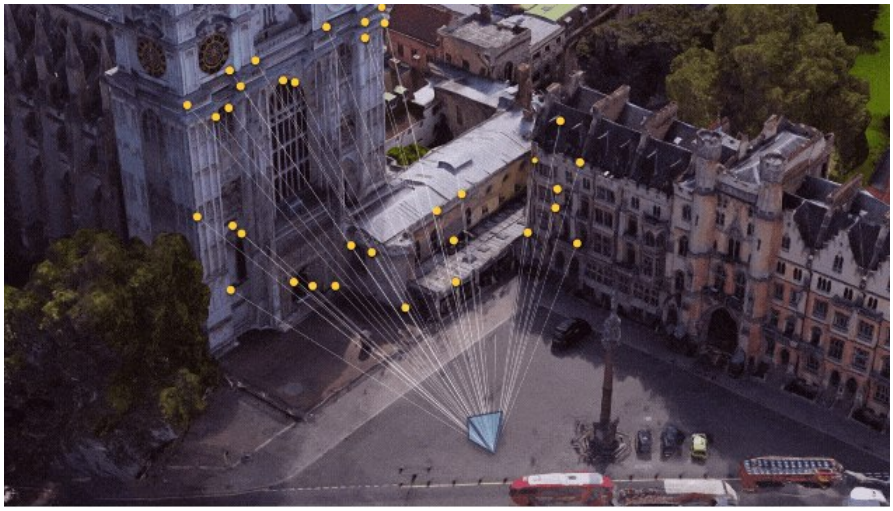

Figura 1.4: Ejemplo de la aproximación por puntos clave para escenas geoespaciales de la documentación de *ARCore* [6]

1.2.2. Visión por computador

Javier González Jiménez define en su libro *Visión por computador* [11] este campo con la siguiente cita:

"La visión por computador, también denominada visión artificial, puede definirse por el proceso de extracción de información del mundo físico a partir de imágenes, utilizando para ello un computador. Desde un punto de vista más ingenieril, un sistema de visión por computador es un sistema autónomo que realiza algunas de las tareas que el sistema de visión humano [...] realiza. La información o tareas que este sistema de visión puede llegar a extraer o realizar puede ir desde la simple detección de objetos sencillos en una imagen hasta ***la interpretación tridimensional de complicadas escenas****. "*

"[...] la información visual consiste en energía luminosa procedente del entorno. Para utilizar esa información es preciso transformarla a un formato susceptible de ser procesado [...] esta misma tarea se realiza con una cámara de vídeo (o algún otro dispositivo similar), que convierte la energía luminosa en corriente eléctrica, que puede ser entonces muestreada y digitalizada para su procesamiento en un computador."

En nuestro caso nos interesa especialmente la sentencia destacada de la cita. Esta disciplina nos permite extraer una idea de la estructura física de un escenario,

permitiéndonos detectar elementos como superficies o esquinas, indispensables si queremos por ejemplo, posicionar un objeto 3D encima de una mesa, y que el objeto parezca que siga apoyado en la mesa si movemos la cámara de lugar, cambiamos el ángulo, o nos alejamos.

Por fortuna la obtención de información a través de imágenes será un proceso completamente transparente para nosotros gracias a las distintas herramientas y tecnologías que ofrecen una API para hacer llamada de estas funciones sin que tengamos que preocuparnos por los detalles técnicos como veremos a continuación.

1.2.3. ARCore

ARCore [6] es un *SDK* (*software development kit*) de Realidad Aumentada desarrollado por Google. Ofrece una API con la que se pueden desarrollar experiencias *AR* en Android, iOS, Unity y Web renderizando modelos 3D con *OpenGL*. Para ello ARCore realiza una serie de tareas a bajo nivel relativas a la visión por computador y por las cuales el usuario no tiene que preocuparse, solo hacer las llamadas necesarias a su API. Estas funciones se detallan a continuación.

- **Seguimiento de movimiento**: Se emplea un proceso llamado *localización y asignación simultáneos*, o ANSM, para que el dispositivo obtenga un contexto sobre el mundo físico que lo rodea. Para ello ARCore detecta características visualmente distintas en las imágenes capturadas (también llamadas **puntos de atributo**) y los usa para calcular el cambio de ubicación del dispositivo. La información visual se combina con la IMU[1] del dispositivo para calcular la pose, es decir, la posición y orientación de la cámara respecto al mundo a lo largo del tiempo.
- **Comprensión ambiental**: Búsqueda de clústeres de puntos de atributos que aparentan pertenecer a superficies horizontales o verticales como pareces o mesas, y las pone a disposición del usuario como *planos* que pueden emplearse para posicionar objetos.
- **Comprensión de profundidad**: ARCore detecta y almacena *mapas de profundidad* con información sobre la distancia entre dos puntos cualquiera pertenecientes a una superficie. Esto puede emplearse para simular interacciones físicas realistas entre objetos virtuales y reales o lograr el efecto de que un objeto virtual aparezca tapado o detrás de uno real.
- **Estimación de luz**: Detecta información sobre la luz de su entorno, su intensidad y color. Con esto se pueden lograr efectos como que un objeto

[1] Un IMU (Unidad de Medida Inercial) es un dispositivo capaz de estimar y reportar información acerca de la velocidad y orientación del mismo a través de acelerómetros y giroscopios.

virtual a la sombra se represente más oscuro que uno expuesto a una fuente de luz.

- **Estimación de luz**: Detecta información sobre la luz de su entorno, su intensidad y color. Con esto se pueden lograr efectos como que un objeto virtual a la sombra se represente más oscuro que uno expuesto a una fuente de luz.

- **Interacción de usuario**: A través de raycasts[2], ARCore puede obtener un *punto de atributo* o modelo posicionado en la escena a partir de un gesto de usuario como tocar la pantalla táctil del dispositivo, lo que permite interacciones como colocar o mover objetos 3D.

- **Puntos orientados**: Los puntos orientados permiten posicionar objetos 3D en superficies que no son superficies planas. ARCore analiza los puntos de atributo vecinos al seleccionado para posicionar el objeto y realizará una estimación de la inclinación, devolviendo un ángulo que servirá para aplicar una pose al objeto para que se vea lo más natural posible.

- **Imágenes aumentadas**: ARCore puede detectar a través de la cámara imágenes 2D definidas previamente y asociar a cada una uno o varios modelos 3D que posicionar sobre estas en el caso de que se enfoquen. Los modelos mantendrán su posición de forma consistente si la cámara se mueve o rota.

1.2.4. SceneView

Sceneform es un *SDK* desarrollado por *Google* que permite importar y visualizar modelos 3D en distintos formatos para renderizar escenas 3D para aplicaciones de *ARCore* o realidad virtual sin necesidad de programar código *OpenGL* de bajo nivel. Cuenta con un *grafo de escena* a través del cual podemos definir la estructura de los modelos de la misma a través de un árbol de nodos, y un motor de físicas de *Filamente*[3].

Lamentablemente, *Sceneform* fue abandonado por *Google*, pero el proyecto ha renacido bajo el nombre de **Sceneview** [9], desligándose de *Google*, con código abierto y mantenido por su comunidad.

1.2.5. Android y Android Studio

Android es un sistema operativo desarrollado por Google, de código abierto y basado en el núcleo de *Linux*. En un principio se diseñó con dispositivos táctiles

[2] text raycast

[3] Filament: Motor de físicas para escenas 3D en tiempo real desarrollado por Google.

en mente como teléfonos inteligentes y *tablets*. Es el que actualmente tiene la mayor cuota de mercado en dispositivos de este tipo, contando en 2022 con el 72,2%[4]. También se emplea en otro tipo de sistemas como televisiones y relojes inteligentes o incluso automóviles.

Android Studio es un entorno de desarrollo integrado o *IDE*, es decir, una aplicación informática que proporciona servicios y recursos para el desarrollador de software como un editor de código, depurador, compilador e interfaz gráfica. En este caso para el desarrollo de aplicaciones en dispositivos Android. Este tipo de herramientas es de gran interés para encarar un desarrollo, ya que aglutina todas las herramientas que podemos necesitar como programadores en un solo entorno.

Kotlin es un lenguaje de programación multiplataforma, estáticamente tipado, con inferencia de tipos y de alto nivel. Está diseñado para ser totalmente inter operable con *Java*, pero usando una sintaxis más concisa. Es además, el lenguaje preferido por Google para el desarrollo de aplicaciones Android y el que se recomienda en toda la documentación.

1.2.6. Typescript

JavaScript (*JS*) es un lenguaje de programación interpretado y compilado *just-in-time*, aunque es más conocido como un lenguaje de *scripting*, es decir, un lenguaje que nos permite incrustar código dentro de páginas web.

Por otro lado, *TypeScript* (*TS*) es un lenguaje construido por encima de JavaScript que añade sintaxis adicional que lo hace **fuertemente tipado**. Es decir, siempre que declaremos una variable debemos especificar su tipo. Si más adelante en el código se intenta asignar un valor no correspondiente a ese tipo, el editor podrá detectarlo y avisar, pudiendo así identificar posibles errores antes siquiera de la ejecución del programa, cosa que con *JavaScript* no es posible.

1.2.7. Archivo JSON

Los archivos JSON (o *JavaScript Object Notation*) es un formato de texto sencillo para el intercambio de datos. Toma como referencia la notación de objetos de *JS*, pero a lo largo de los años y debido a su simplicidad, se ha convertido en la opción por defecto para intercambio de información entre servicios web, en contraposición a su principal alternativa, *XML*.

[4] https://root-nation.com/es/noticias-es/es-cuota-mercado-android-ios-estadisticas-publicadas-

1.2.8. Three.js

Three.js [?] es una librerías *JavaScript* usada para crear y renderizar gráficos 3D animados en páginas web para que puedan se reproducidas en un navegador. Para ello hace uso de *WebGL*, otra librería *JavaScript* para la renderización de gráficos pero a más bajo nivel. *Three.js* se abstrae de conceptos técnicos y de bajo nivel para que el desarrollador no tenga que preocuparse de todo eso y agilice su desarrollo. Además es una librería de código abierto mantenida por su comunidad.

1.2.9. React.js

React [?] es una librería de *JavaScript* para construir interfaces de usuario web de código abierto y mantenido por su comunidad. Opcionalmente, utiliza ficheros de extensión *.jsx* (*JavaScript Syntax Extension*) para facilitar el desarrollo. Estos son, como su nombre indica, una extensión sintáctica de *JS* que permite insertar código HTML, facilitando así la construcción de interfaces de usuario.

1.2.10. Node.js

Node [5] es un entorno de ejecución de *JavaScript* orientado a eventos asíncronos diseñado para crear aplicaciones de red rápidas capaces de manipular grandes cantidades de conexiones concurrentes con una alta escalabilidad para los proyectos.

1.2.11. Express

Express [3] es un *"Web application framework minimalista y flexible para Node.js que provee una serie de herramientas para aplicaciones web y móviles."* como lo definen ellos mismos. Cuenta con una gran variedad de utilidades para llamadas *HTTP* y *middleware*. Cuando hablamos de *middleware* nos referimos a un software a través del cual distintas aplicaciones se comunican entre sí por internet.

1.2.12. Firebase

Firebase [8] es un BAAS[5] desarrollado por *Google* que pone a disposición de los desarrolladores una serie de herramientas y servicios para facilitar la creación

[5] Backend-as-a-service: aplicaciones puestas al servicio de desarrolladores para que estos hagan usos de sus servicios y se centren en desarrollar el frontend de sus apps, siendo innecesario así que creen ni mantengan sus propios backends.

de aplicaciones. Algunas de sus servicios más importantes son:

- **Autenticación**: Soporta autenticación de usuarios usando contraseñas con correo electrónico, número de teléfono, o cuentas de *Google*, *Facebook* y *Twitter* entre otros. Ofrece también encriptación por defecto en la base de datos para contraseñas.
- **Base de datos en tiempo real**: *Firebase* tiene también una base de datos **no relacional** que se actualiza a tiempo real y se mantiene en línea incluso cuando la aplicación no se está ejecutando.
- **Hosting**: Permite *hostear* aplicaciones web de forma fácil y rápida, pudiendo almacenar contenido en una memoria caché y accesible desde cualquier parte del mundo.

1.2.13. Archivo GLB

GLB (*.glb*) es un formato de archivo binario estandarizado para representar datos de objetos 3D como modelos, escenas, iluminación, materiales, jerarquías y animaciones. Sus siglas significan *GL Transsmission Format Binary File*. Fue introducido en 2015 como una versión binaria de los ficheros **GLTF** (*.gltf*), un formato basado en la notación *JSON* muy usado en aplicaciones web y móvil debido a su ligero peso.

Capítulo 2

Descripción del problema

En este capítulo se detallará el problema que se pretende resolver con este proyecto, además de establecer los objetivos que se requieren cumplir. Se realizará una crítica al estado del arte y para finalizar se enunciará una propuesta de producto.

2.1. Problema a resolver

Como se ha comentado en apartados anteriores, existen en el mercado multitud de experiencias de Realidad Aumentada al alcance de los usuarios, aunque en la gran mayoría de ellas los usuarios juegan un rol pasivo. Esto último no hace referencia a que no puedan interactuar con el contenido, (ya hemos visto que la interacción en tiempo real es uno de los rasgos definitorios de la *RA*) sino que consumen contenidos y experiencias que han sido creadas por los desarrolladores de las distintas apps. Suelen ser sistemas cerrados donde el usuario no tendría opción de, por ejemplo, cargar un modelo 3D de su dispositivo para visualizarlo en su salón.

Por otro lado, las herramientas de modelado y animación de objetos 3D como *Blender*[1] o *3ds Max*[2] son herramientas profesionales que requieren de una formación, conocimientos previos y curva de aprendizaje para poder manejarlos con soltura. Incluso si las aplicaciones pudieran efectivamente cargar modelos en las aplicaciones ya mencionadas, estarían limitados a elementos sueltos que pudieran encontrar en internet. Esto es algo que limita mucho tus opciones creativas. Volviendo al ejemplo de la escena del parque que se puso en el capítulo 1, si se pretende recrear hacer la escena de un parque, quizás se encuentre un

[1] link blender
[2] link 3ds max

modelo 3D para ello. Pero, ¿y si se quiere tenga concretamente dos toboganes y un columpio? Es muy poco probable que alguien haya modelado y publicado una escena con exactamente los mismos requisitos. Pero contando con que se puedan encontrar los elementos por separado (un modelo de un tobogán, de un balancín, de un columpio...) lo cual es más realista, y alguna forma de componerlos en una sola escena, se podría construir un parque con una composición cualquiera.

Teniendo todo esto en cuenta se detectan dos necesidades para el usuario. Por un lado, una suerte de editor sencillo, simplificado e intuitivo en el que un usuario sin conocimiento o experiencia previa pueda cargar o crear modelos 3D con los que pueda construir una escena con total libertad creativa. Por otro lado, una manera de que el usuario pueda reproducir cómodamente sus composiciones producidas en un contexto de Realidad Aumentada en un dispositivo con cámara como podría ser un *smartphone*.

Este escenario presenta ciertos retos de diseño e implementación. El software profesional de modelado profesional es complejo por una razón: tienen muchísimas posibilidades. Si se quiere tener una herramienta sencilla inevitablemente el usuario perderá opciones. Por tanto, se debe encontrar un compromiso, un *sweetspot* en el que se intenten cubrir casi todas las necesidades que podría tener una persona sin que el software se vuelva demasiado obtuso. También se debe de tener en cuenta un aspecto técnico importante: qué formato de archivos para modelos soportará el sistema. Existe una gran variedad hoy en día, cada uno con sus peculiaridades, y no siempre compatibles entre ellos, por lo que el software deberá adoptar uno o contar con herramientas de conversión para soportar un conjunto de ellos. También hay que tener en cuenta que el tipo de dispositivo en el que principalmente se tiene interés por reproducir escenas *RA* son teléfonos móviles o tablets debido a sus cámaras integradas y su portabilidad, pero estos se controlan con gestos táctiles, los cuales no son tan precisos como un ratón de ordenador y podría ser una dificultad a la hora de implementar controles precisos para el editor. Por ello deberá buscarse una forma de adaptar este tipo de programa a un control táctil o bien implementar una conectividad ordenador-*smartphone* para poder crear la escena en el primero y reproducirla en el segundo. En este TFG se propondrá y desarrollará una solución para estas necesidades.

2.2. Objetivos

Una vez definidos los retos y problemas a resolver en este contexto, se procede a enunciar los objetivos a los que se pretende llegar con este proyecto:

- **OBJ-1**: Desarrollar un software que permita cargar y crear modelos 3D a los que aplicarles transformaciones como translaciones, rotaciones y escalado para construir una escena.

- **OBJ-2**: El software descrito descrito debe ser simple, sencillo e intuitivo para que lo pueda usar cualquier persona sin que tenga experiencia ni conocimientos previos en la materia.
- **OBJ-3**: El software debe permitir la máxima flexibilidad posible a la hora de producir las escenas teniendo en cuenta su alcance limitado y controles sencillos.
- **OBJ-4**: Desarrollar una aplicación informática que reciba como entrada uno o varios modelos 3D que compongan una escena y reproduzca con ellos una experiencia de realidad aumentada.
- **OBJ-5**: La solución aportada debe ser ligera y lo más rápida posible, para que los usuarios de dispositivos de gama media o baja puedan usarlo sin problemas.
- **OBJ-6**: El sistema debe tener memoria, es decir, un usuario deberá de alguna forma poder conservar las escenas generadas con anterioridad para que el usuario las pueda volver a modificar en un futuro.

2.3. Estado del arte

Existe en el mercado cierta variedad de opciones desarrolladas que cumplen en mayor o menor medida los objetivos que se han planteado en este TFG. Esta sección se dedicará a analizarlas, compararlas y detectar cuales son las fortalezas y carencias de cada una, con la intención de sacar en claro qué se podría aportar con una nueva solución. Después de investigar y analizar las soluciones más relevantes, se encontraron **cinco plataformas** con un propósito y características similares al proyecto que se pretende desarrollar. De estas cinco se descartaron dos, *Aryel*[3] y *Artificio*[4] ya que aunque superficialmente comparten algunos elementos como son al fin de al cabo la creación de escenas *RA*, la primera está enfocada para que se le dé uso en agencias de marketing para diseñar publicidad interactiva, y la segunda es una herramienta para diseñar interiores de viviendas.

2.3.1. Onirix Studio

Onirix Studio [?] es una plataforma gratuita en la que un usuario al darse de alta puede crear multitud de escenas *RA* a través de un editor. Las escenas producidas quedan registradas en la cuenta de usuario, y pueden ser accedidas a través de un menú que permite volver a editarlas o eliminarlas. En cuanto al editor

[3] https://aryel.io/
[4] https://arficio.com

tiene un gran abanico de opciones. El usuario puede añadir modelos a la escena, posicionarlos, rotarlos y escalarlos con total libertad a través tanto de controles gestuales de ratón como de un panel numérico. Cuenta con la posibilidad de modificar la iluminación que proyecta la escena sobre los modelos. Es posible añadir texto que se visualice junto a los modelos. Para añadir *assets* como los modelos, el usuario cuenta con un catálogo personal, con algunos elementos que añade la aplicación por defecto, pero con la opción de que el usuario añada los suyos propios. Una vez los añade los puede usar en cualquier otro proyecto.

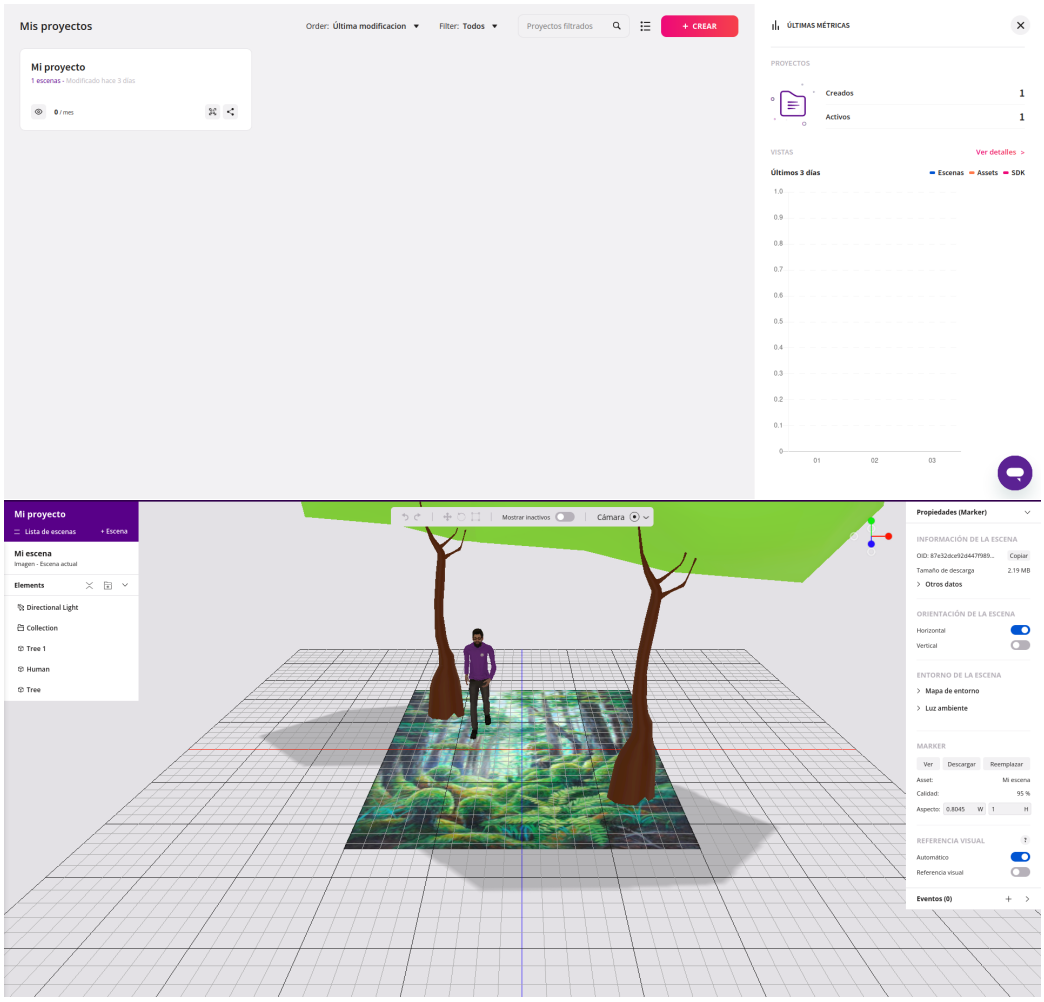

Figura 2.1: Web de creación de escenas RA Onirix Studio

Uno de los puntos más destacables que ofrece este entorno es un sistema al través del cual se pueden definir eventos dinámicos para la escena al cumplirse cierto activador. Por ejemplo, si tenemos un botón pulsable en la escena, se puede añadir una regla que ordena al pulsarse el botón la reproducción de la animación de uno de los modelos, o aparezcan modelos nuevos, desaparezcan

otros, etc.

Para la reproducción tenemos se pueden o bien reproducir en la propia web, o generar un *código QR*[5] escaneable por un *smartphone*. Este *QR* abre en el navegador un enlace a través del cual se reproduce la escena. Por un lado, esta es una solución cómoda y ágil, pero tiene puntos negativos. Para empezar el rendimiento de la cámara y renderización de elementos 3D de un dispositivo móvil se va a ver reducida si es a través de una aplicación web en lugar de una aplicación nativa del teléfono. También hay que tener en cuenta que existe un enorme catálogo de navegadores gratuitos, y aunque la mayoría de los usuarios utilizan los más populares como *Google Chrome*, *Mozilla Firefox*, *Opera* o *Safari*, no se puede asegurar la compatibilidad total con cualquier navegador, o que el rendimiento sea óptimo en todos. Otro punto negativo a destacar de *Onirix Studio* [**?**] es que solo es posible crear escenas de posicionamiento sobre superficie y de imágenes aumentadas.

2.3.2. Blippbuilder

Blippbuilder [**?**] es una web muy similar a la anteriormente comentada, y también se puede emplear sin coste alguno. A través de una cuenta el usuario puede almacenar una colección de escenas RA editables a través de un menú similar. El editor contaba con una estructura parecida, permitiendo también aplicar transformaciones con gestos de ratón. Al igual que la anterior, tiene la opción tanto de cargar recursos desde el ordenador del usuario como de acceder a una librería incluida con la web, opción de añadir texto, manipular la iluminación de la escena. También cuenta con un pre visualizador dentro del editor y con opción para probar la escena desde el navegador de un dispositivo móvil escaneando un código *QR* (es decir, tampoco es nativa).

Como aspecto diferenciador, es posible añadir a la escena geometrías básicas tales como esferas, cubos o cilindros para añadirlos. Además, cuenta con un menú a través del cual se pueden **crear animaciones** para la escena. A través de *keyframes*. Un *keyframe* o *fotograma clave* es una unidad de información que almacena atributos tales como la posición, rotación o escala de un objeto para un instante determinado. Definiendo dos *keyframes* para el mismo objeto en 2 instantes distintos, un software puede interpolar sus posiciones intermedias generando así una animación. Esta animación puede ser reproducida junto a la escena.

En *Blippbuilder* solo es posible la creación de escenas de Realidad Aumentada por superficie.

[5] explicar código qr

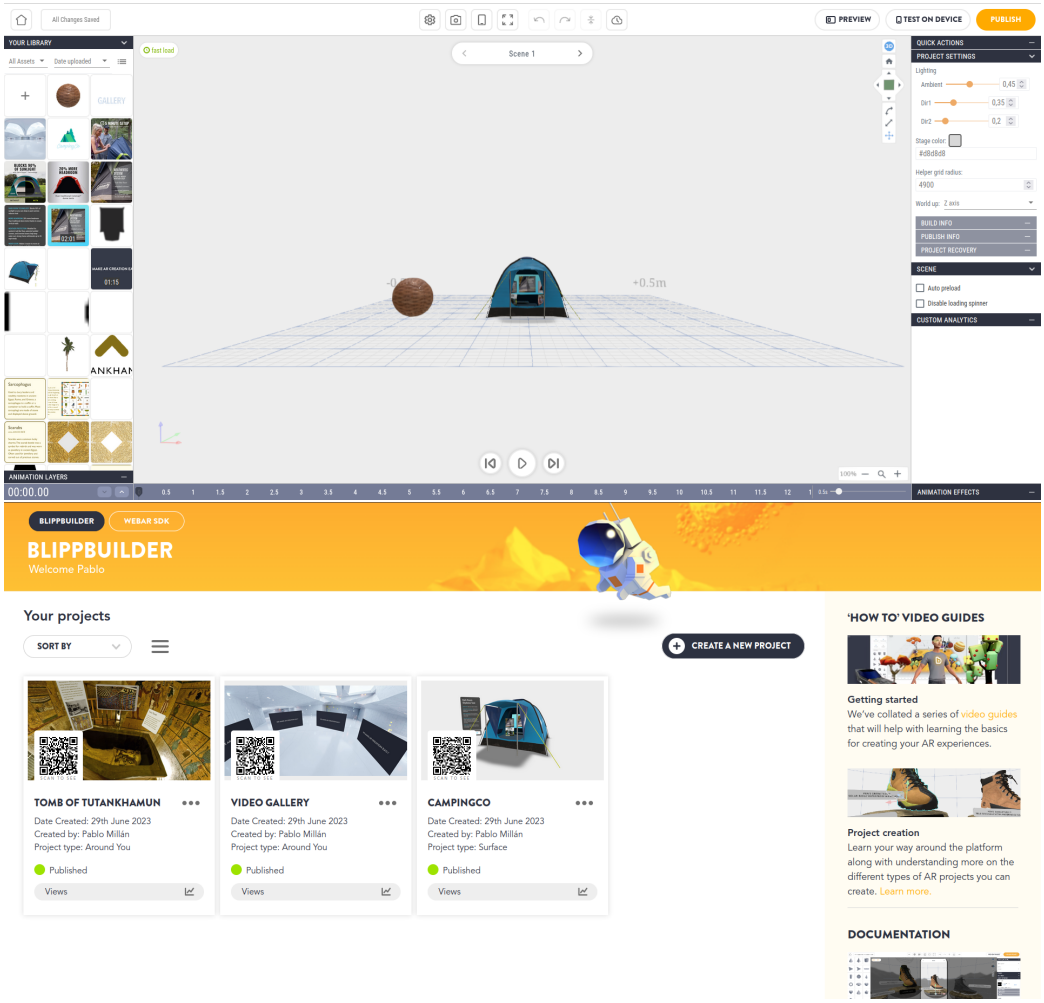

Figura 2.2: Web de creación de escenas RA Onirix Studio

2.3.3. Pictarize

Pictarize [1] de nuevo tiene un funcionamiento muy parecido a lo visto hasta ahora y sin requerir de pagos. Una galería de escenas producidas asociadas a una cuenta de usuario, y un editor para hacer las mismas. Tiene opción para cargar modelos desde el ordenador del cliente y de nuevo controles gestuales con el ratón para manipularlos. Aunque también puede añadir texto, no existe opción para alterar la iluminación. Sin embargo, es posible embeber videos de youtube en forma de textura plana, para reproducirlos al pulsarlos cuando se reproduce la escena. Es posible también asociar una pista de audio a la escena para que se reproduzca junto a las animaciones de los modelos 3D.

Para la reproducción de escenas se cuenta tanto con un pre visualizador

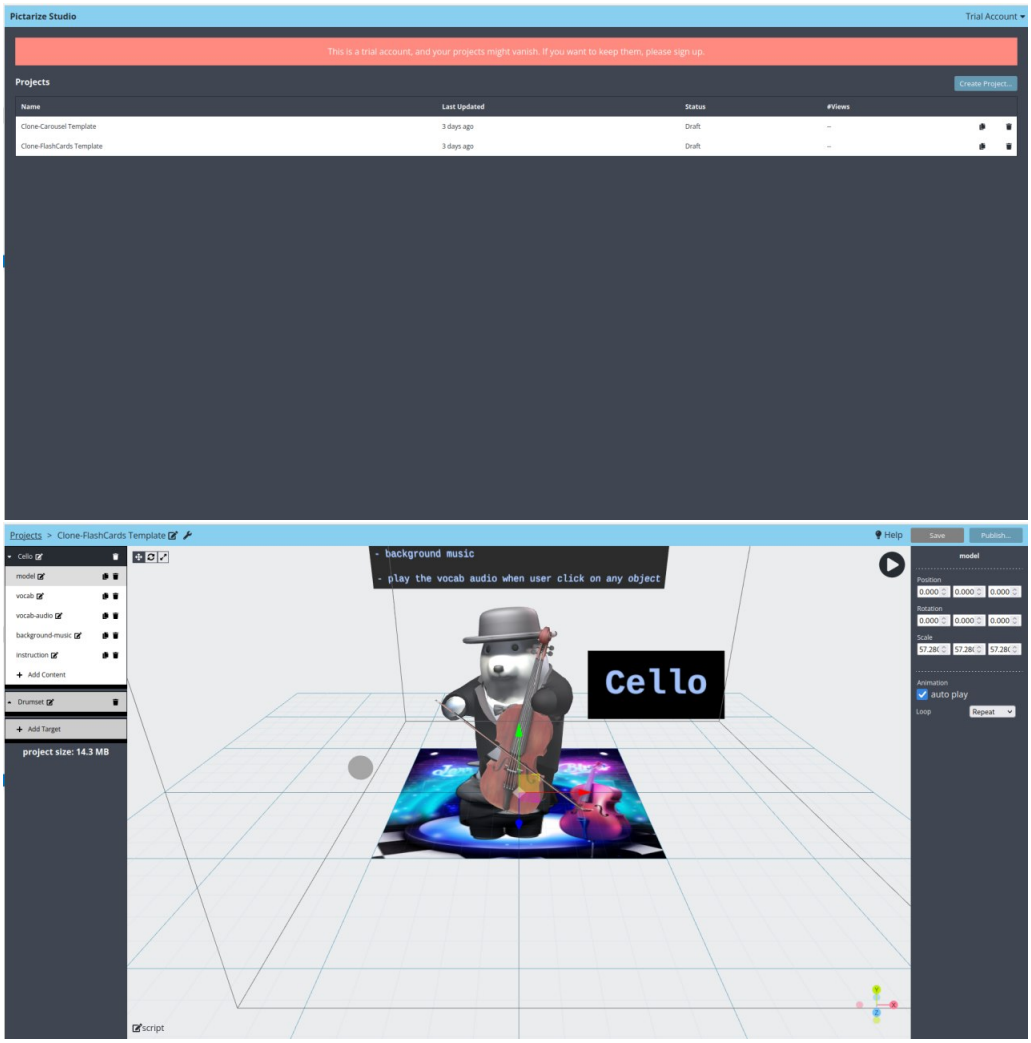

Figura 2.3: Web de creación de escenas RA Onirix Studio

dentro del editor como una opción para verlas desde un *smartphone* a través de un código *QR* y un navegador web, como se ha visto en los dos ejemplos anteriores. Sólo hay opción para escenas de imágenes aumentadas, con la opción de definir distintos activadores para distintos elementos.

2.4. Crítica

Como se ha podido dilucidar a través de estos ejemplos ya hay varias soluciones disponibles y bastante completas aun siendo gratuitas. Sus editores son potentes en cuanto a que permiten un alto grado de expresión por parte del

usuario, permitiendo un gran abanico de posibilidades de forma que el usuario podrá muy probablemente representar la escena que tenía en mente siemrpe que cuente con los modelos adecuados.

Las propuestas existentes analizadas comparten multitud de elementos en común. Todas ofrecen un menú para gestionar una lista de escenas creadas por un usuario, el cual se identifica con una cuenta para la propia web. Esta cuenta usa como identificador un nombre de usuario y correo electrónico. En todas es posible cargar objetos 3D desde el ordenador del cliente.

También permiten aplicar transformaciones de translación, rotación y escalado a los modelos de la escena con gestos del ratón a través de *gizmos*. Estos son unas estructuras visuales interactuables que ayudan a aplicar transformaciones a objetos 3D. Dependiendo de la operación que se quiera realizar, hay distintos tipos de gizmo. Cogiendo el ejemplo del *3D Move Gizmo* de la siguiente figura, si se hiciera click en una de las flechas coloreadas, se permitiría mover el objeto con un desplazamiento del ratón en el eje que señala la misma. Además de controles gestuales permiten introducir manualmente números para representar las transformaciones del objeto a través de un menú. Por ejemplo se puede especificar que un modelo está *'6.2' unidades desplazado en el eje X* o con *una rotación de 90º en el eje Y*.

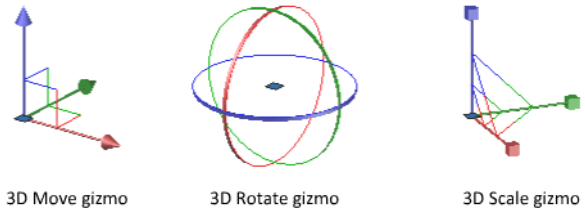

Figura 2.4: Ejemplo de Gizmo en la documentación de Autodesk

Hay otras funciones que si bien no son imprescindibles para el objetivo de crear una escena, dan posibilidades extra de personalización como la adición de texto, manipulación de luces, reproducción de audios, definición de eventos o creación de animaciones con *keyframes*. Hay que tener una cosa en cuenta con este tipo de funcionalidades, y es que si bien aportan valor y posibilidades a la aplicación, también aumentan la complejidad de la aplicación y la curva de aprendizaje que tienen los usuarios. No es descabellado pensar que alguien que no tiene nociones básicas sobre cómo funciona la animación tenga dificultades para manipular una timeline de *fotogramas clave* para crear una pequeña animación. Por tanto, se puede deducir que un porcentaje de los usuarios se sentirán confundidos por funciones como esta o la de definir eventos, o directamente no llegarán a intentar usarla.

También hay presentes algunas carencias en estas webs. Se puede ver que una de ellas permite crear tanto escenas por superficie como de imágenes aumen-

tadas, mientras que en las otras dos opciones solo es posible hacer o imágenes aumentadas o superficie. Y lo que es más, ninguna tiene soporte para generar escenas geoespaciales basadas en la localización.

Otra ausencia a destacar es la imposibilidad en ninguna propuesta de descargar guardar de forma local en el ordenador las escenas que se han creado en un formato de objeto 3D, para que el usuario pueda utilizar ese archivo en cualquier otro software que admita el tratamiento de modelos. Esto se traduce en una imposibilidad de que las escenas salgan del ecosistema de sus respectivas webs.

Por último señalar que en todas las webs analizadas la única opción para reproducir las escenas en un entorno real de Realidad Aumentada en un dispositivo móvil es a través de un navegador web al que se accede escaneando un código QR. Como se ha explicado anteriormente, se trata de una opción cómoda en tanto que es multiplataforma, no hace falta iniciar sesión de nuevo en el segundo dispositivo y en pocos segundos se puede escanear el código y tener cargando la escena, pero el rendimiento y fluidez de la cámara que se obtiene es considerablemente menor que si estuviera corriendo en una aplicación nativa, por lo que no estaría de más tener la opción de ejecutarlo de esta forma.

2.5. Propuesta

Con esta propuesta de proyecto se pretende desarrollar una alternativa completa que intentará cubrir las carencias que se encontraron analizando las webs disponibles en el mercado:

- **Intuitivo**: El editor contará con las herramientas básicas e imprescindibles para poder elaborar una escena 3D con las necesidades del usuario. Esto permitirá mantener una interfaz limpia, legible e intuitiva que permita construir escenas de una forma rápida y que no requiera conocimientos o experiencias previas con software de manipulación de entornos 3D. Así los usuarios no se sentirán perdidos o que tiene más herramientas de las que realmente necesitan.

- **Múltiples tipos de escenas**: Será posible la creación de distintos tipos de escenas de Realidad Aumentada. En concreto permitirá escenas de posicionamiento por superficies, de imágenes aumentadas y geoespaciales en unas coordenadas concretas.

- **Descarga de escenas**: Además de poder almacenar las creaciones en la nube, será posible exportar la escena con formato de objeto 3D para que el usuario pueda emplearla en cualquier otro software que permita archivos de este tipo.

- **Reproducción nativa**: Será posible reproducir escenas en una aplicación nativa para dispositivos móviles, proporcionando un mayor rendimiento y asegurando compatibilidad.

Para eso se desarrollará un servicio completo que contará de tres partes fundamentales. Para empezar, una **aplicación web** que permita la creación y gestión de escenas de realidad aumentada. Por otro lado, se tiene una **aplicación móvil** desde la que reproducir de forma nativa las escenas. Para terminar, un **servidor web** en el que se almacenarán las escenas creadas, que podrán ser recuperadas tanto desde la aplicación web como la aplicación móvil.

En el capítulo 4 se dará una hablará más en detalle del software en una descripción del mismo desde el ángulo de las metodologías ágiles, y en el capítulo 5 se explicará tanto la implementación como las decisiones de diseño y tecnologías empleadas.

Pablo Millán Cubero

Capítulo 3

Estado de la Realidad Aumentada

El software libre y sus licencias [4] ha permitido llevar a cabo una expansión del aprendizaje de la informática sin precedentes.

Capítulo 4

Planificación

4.1. Metodología utilizada

El software es un sistema complejo de crear que depende de muchas partes interconectadas. No es igual de dificultoso programar un pequeño *script* que ordene los ficheros de un directorio de un sistema operativo que construir una aplicación de cierta envergadura que responda a las necesidades de un usuario o un cliente concreto, mucho más si además se debe coordinar un equipo de personas en la concepción del mismo. Existe un gran factor de riesgo e incertidumbre, sobre todo al inicio de un proyecto. Las **metodologías de desarrollo ágil** son una forma de desarrollar software enfocada a equipos pequeños que busca paliar estos problemas a la hora de encarar un proyecto,. En otras metodologías anteriores se redactaban unos requisitos completos y fijos al inicio del proyecto y antes si quiera de comenzar la implementación. Esto era una fuente común de conflictos, ya que es muy común que los requisitos cambien a lo largo del desarrollo. Quizás el cliente cambie de idea, o el equipo encuentre mejores formas de realizar una tarea, o el contexto en el ámbito en el que se desarrolla el producto ha cambiado y hay que adaptar ciertas partes del proyecto, etc. La incertidumbre es un elemento muy presente, y las metodologías ágiles la abraza. Conviven con el cambio inevitable estableciendo unas bases menos sólidas pero más flexibles. Esta forma de trabajar se enunció oficialmente por primera vez en 2001, en el documento de **Manifiesto Ágil**[1]. Esta forma de organizar proyectos se fundamenta en cuatro puntos, extraidos del libro *Métodos Ágiles y Scrum* [2]:

- **Valorar a individuos y sus interacciones**, frente a procesos y herramientas. Aunque todas las ayudas para desarrollar un trabajo son importantes,

[1] El Manifiesto Ágil se puede consultar en esta web oficial. Hay una versión en español en la misma web. https://agilemanifesto.org/

nada sustituye a las personas, a las que hay que dar toda la importancia y poner en primer plano.

- **Valorar más el software (producto) que funciona**, que una documentación exhaustiva. Porque había llegado un punto en el que documentar el trabajo había alcanzado tanta importancia como el objeto del trabajo: el producto.

- **Valorar más la colaboración con el cliente** que la negociación de un contrato. La forma más productiva de sacar adelante un trabajo es establecer un marco de confianza y colaboración con quien nos lo encarga. Sin embargo se estaba poniendo el foco en cerrar un contrato atado que sirviera ante todo como una herramienta de protección, como si el cliente y equipo fueran dos partes enfrentadas, cuando en realidad comparten objetivos e intereses.

- **Valorar más la respuesta al cambio** que el seguimiento de un plan. Se trata de apreciar la incertidumbre como un componente básico del trabajo, por lo que la adaptación y la flexibilidad se convierten en virtudes y no en amenazas. El seguimiento ciego de un plan lleva, salvo contadas excepciones, al fracaso si no se puede corregir la dirección ante los inevitables cambios que van surgiendo.

Este típo de metodologías intentan aplicar un cambio de paradigma: mientras que tradicionalmente se fijaban unos requisitos a partir de los cuales se estimaban los recursos y plazos que se requerirían para su terminación, la forma ágil sería fijar unos recursos y fechas disponibles en las que se disponen para el trabajo, y a partir de ahí estimar los requisitos que va a tener el producto al final. Se eligió optar por esta forma de encarar proyectos ya que en primer lugar el trabajo tenía una fecha de finalización fija, y además no se tenía experiencia en algunos de los campos requeridos para el desarrollo, como podría ser por ejemplo la implementación de interfaces web. Se iba a requerir de un periodo de formación paralela al desarrollo, lo cual con total probabilidad cambiará la visión del desarrollador sobre la estimación de los costes y tiempo del proyecto. Sería contraproducente espablecer desde un inicio unos requisitos fijos si se prevé que van a cambiar, y las metodologías ágiles se ajustaban perfectamente a este escenario.

Una vez definidos los métodos de desarrollo ágil y entendido por qué son convenientes para este proyecto en particular, se debe elegir uno de los que hay disponibles como *Lean Software Development*, *Kanban*, *Pragmatic Programming* o *eXtreme Programming*. En este caso se ha optado por **SCRUM** [12], una de las metodologías más difundidas.

SCRUM proporciona un marco de trabajo para equipos de desarrollo pequeños y autogestionados. Como en este caso el trabajo se realiza de forma

individual los roles que propone este método se han sintetizado en una sola persona. Originalmente se tenía al equipo de desarrollo, el *Product Owner* (dueño del producto o cliente) y el *SCRUM Master*, el encargado de que el equipo sea efectivo.

Se trabaja definiendo una serie de *sprints*, iteraciones de dos semanas de duración las cuales deben presentar un producto entregable al final de cada una, como por ejemplo un software ejecutable. Al comienzo de cada sprint se hace el *sprint planning*, una reunión en la que se planifica el trabajo a desarrollar durante la iteración. Al final de cada sprint se realiza el *sprint review* en el que se valora cómo se ha trabajado y qué aspectos mejorar para el siguiente sprint. Las *daily meetings* que propone SCRUM en las que cada miembro del equipo comenta en qué estado del trabajo se encuentra se ha omitido para este proyecto por ser solo un integrante en el equipo.

4.2. Historias de usuario

Para describir los requisitos del sistema a desarrollar se emplearán *historias de usuario*. Estas son descripciones del sistema que se usan para plasmar los requisitos del mismo desde el punto de vista del usuario. Se redactan al comienzo del desarrollo, durante el *Sprint 0* del que se hablará más adelante, y a partir de estas se redactan las *tareas* del desarrollo.

Antes de redactar las historias es necesario definir definir los distintos perfiles que van a hacer uso del producto. En este caso se tiene un único perfil:

- **Usuario**: Este perfil representa a un consumidor casual, que puede o no tener nociones tanto de Realidad Aumentada como de modelado 3D. Tiene una experiencia como poco básica en el manejo de ordenadores y aplicaciones de los mismos, usando tanto el teclado como el ratón para interactuar con distintos elementos en pantalla. Es usuario de servicios como correo electrónico y está familiarizado con el proceso de iniciar sesión con contraseña. Tiene los recursos para poder encontrar en internet modelos 3D gratuitos y descargarlos en su equipo.

Una vez definido el perfil que hará uso del software se redactarán los *journeys*. Estas son una descripción en forma de narración en la que se detalla el uso que un usuario concreto (que no un perfil de usuario) hace uso del producto desde el inicio hasta al final. Esto servirá para comprender mejor las necesidades de estos y reflejarlas en las historias de usuarios.

- **J-01**: Florentino Pérez es un hombre de 42 años. Escucha sobre la tecnología de Realidad Aumentada en el telediario y le parece de lo más vistosa e innovadora, por lo que suscita su curiosidad y decide probarla. Tras una

breve búsqueda en Google encuentra EditAR entre otras opciones, y decirle probarla porque aparenta ser sencilla. Tras acceder a la web sigue las instrucciones para registrarse como usuario. Una vez hecho, selecciona la opción de crear escena, y empieza a componer una del tipo de posicionamiento en superficies. Carga modelos 3D que ha encontrado en una web de recursos gratuitos y hace click en ellos para manipularlos y colocarlos donde él prefiera. Cuando ha terminado selecciona la opción de guardar la escena. Florentino procede a instalar en su dispositivo móvil la aplicación del visor. En ella inicia sesión con las credenciales que introdujo en pasos anteriores. Aparece un menú con la única escena que ha creado y selecciona la opción de reproducir. Se inicia la cámara de su dispositivo y procede a colocar el modelo de la escena encima de su escritorio. Florentino observa cómo detecta la superficie y coloca el objeto. Incluso aunque mueva la cámara de ángulo sigue dando la sensación de que el objeto está en la mesa. Satisfecho, cierra la aplicación.

- **J-02**: Belkan Díaz es un hombre de unos 20 años. Acaba de adquirir un nuevo y flamante smartphone Xiaomi y está oteando la tienda de aplicaciones para descargar algunas para estrenarlo. De casualidad se topa con la aplicación de EditAR y acaba descubriendo también su web, así que decide probarla. Se registra en el sistema con sus credenciales y pasa a crear una escena de imágenes aumentadas con modelos 3D gratuitos de internet. Cuando ha terminado y la guarda, se da cuenta de que por error se ha registrado con el correo electrónico de su trabajo, lo cual es inapropiado, así que accede al menú de cambio de credenciales, e introduciendo su contraseña logra cambiar su dirección de email. Una vez solucionado eso, accede usando su nuevo correo desde la aplicación móvil y consigue reproducir la escena. Aunque le sorprende el funcionamiento de la realidad aumentada, no le gusta la escena que ha creado para nada, así que vuelve al editor para eliminarla y crear una nueva, la cual sí le convence.

- **J-03**: Antonia Miyamoto es una artista de modelos 3D. Tras haber oido hablar de la aplicación decide probarla con alguno de los modelos que ella misma ha diseñado. Accede a la web y se registra con sus credenciales. Piensa en que le gustaría ver un dragón sobrevolando su casa, así que inicia una nueva escena y la configura para que sea de tipo geoespacial. Introduce las coordenadas y altura de su vivienda, que ha podido obtener en algún servicio como *Google Earth*. Carga el modelo del dragón que modeló días anteriores en *Blender*. Lo coloca en la escena y selecciona que quiere que se reproduzca una animacioń del modelo en la que el dragón bate sus alas, para que de la sensación de que verdaderamente está volando. Una vez se da por satisfecha guarda la escena y cierra el editor. Instala la aplicación móvil e inicia sesión. Selecciona su escena y ve al dragón volando. Al verlo, piensa que podría quedar mucho mejor si mete una gran bola de fuego y

una pista de audio de un gruñido de dragón. Antonia vuelve a abrir el editor y selecciona la escena que ya había creado en una opción que le permite editarla. El editor carga la escena tal como la dejó, lo que le permite añadir nuevos modelos para añadirle fuego. Además carga un archivo mp3 con el rugido. Cuando ve la nueva escena en su dispositivo móvil piensa que la escena ha quedado verdaderamente vistosa, y le gustaría exportarla para usarla en alguno de sus proyectos de *Blender*, así que vuelve a abrir el editor web con la escena creada y hace click en un botoń que le exporta su escena y la descarga a su ordenador en forma de modelo 3D.

Una vez entendidas mejor las necesidades que tendrá el usuario del sistema se pueden redactar las historias de usuario. Para cada historia se definen una id que servirá para identificarla con formato **RA-X**, una descripción, unos requisitos asociados con formato **RF/RNF-X.n** y una prioridad. La prioridad podrá ser *high* (alta) si describen elementos imprescindibles para el correcto funcionamiento del software y el cumplimiento de su propósito, *mid* (media) si son de gran importancia pero la ausencia de alguna no es un impedimento crítico para el producto final, y *low* para las historias que describen elementos preferibles, que aportan valor añadido pero que no son necesarios.

4.2.1. RA-01: Entorno 3D

Como usuario quiero tener un entorno 3D interactivo para configurar mis escenas de Realidad Aumentada.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.2. RA-02.1: Escenas de superficie

Como usuario la posibilidad de crear escenas de Realidad Aumentada de posicionamiento por superficies.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.3. RA-02.2: Escenas de imágenes aumentadas

Como usuario la posibilidad de crear escenas de Realidad Aumentada de imágenes aumentadas.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.4. RA-02.3: Escenas geoespaciales

Como usuario la posibilidad de crear escenas de Realidad Aumentada geoespaciales introduciendo una latitud, longitud y altura concreta.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.5. RA-03: Varias imágenes activadoras

Como usuario quiero, al crear una escena por marcador, poder definir varias imágenes activadoras, y distintos modelos asociados a cada una de ellas.

- Prioridad: **LOW**
- Requisitos: Ninguno

4.2.6. RA-04: Animaciones

Como usuario quiero que los modelos que cargo puedan tener animaciones, y decidir si se reproducen al ver la escena una vez, en bucle o no lo hacen.

- Prioridad: **MID**
- Requisitos: Ninguno

4.2.7. RA-05: Pistas de audio

Como usuario quiero poder cargar audios que se reproduzcan cuando se dispara la escena, y decidir si se reproducen una vez o en bucle.

- Prioridad: **MID**
- Requisitos:
  - **RF-5.1**: Cada escena tendrá asociado un único audio como mucho.

4.2.8. RA-06: Carga de modelos

Como usuario quiero cargar un modelo 3D desde mi ordenador para añadirlo a la escena, así como eliminarlo más adelante.

- Prioridad: **HIGH**
- Requisitos:
  - **RNF-6.1**: Los modelos cargados deben ser de formato GLB con las restricciones para su uso en Filament.

4.2.9. RA-07: Transformaciones

Como usuario quiero poder seleccionar un objeto 3D de la escena haciendo click sobre él y poder moverlo, rotarlo o hacerlo más grande o más pequeño.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.10. RA-08: Selección múltiple

Como usuario quiero poder seleccionar más de un objeto a la vez para poder aplicarles los mismos cambios como moverlos, rotarlos o cambiarlos de tamaño.

- Prioridad: **LOW**
- Requisitos: Ninguno

4.2.11. RA-09: Figuras básicas

Como usuario quiero poder crear figuras geométricas básicas como esferas, cubos, pirámides o cilindros entre otras.

- Prioridad: **LOW**
- Requisitos: Ninguno

4.2.12. RA-09.1: Texturas

Como usuario quiero un menú en el que poder cambiar el color de las figuras básicas que creo.

- Prioridad: **LOW**
- Requisitos: Ninguno

4.2.13. RA-10: Inicio de sesión

Como usuario quiero poder iniciar sesión con mis datos en la aplicación web para que, si accedo desde otro ordenador, pueda recuperar datos mios como las escenas creadas.

- Prioridad: **HIGH**
- Requisitos:
  - **RNF-10.1**: Se almacenará y empleará para el inicio de sesión únicamenet el email y contraseña del usuario

4.2.14. RA-10.1: Cambio de credenciales

Como usuario quiero tener la opción de modificar mi correo electrónico o contraseña en un futuro.

- Prioridad: **MID**
- Requisitos: Ninguno

4.2.15. RA-10.2: Guardado de escenas

Como usuario quiero poder guardar bajo mi cuenta de usuario las escenas que creo para poder volver a abrirlas en un futuro y editarlas.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.16. RA-11.1: Reproducción de imágenes aumentadas

Como usuario quiero poder reproducir en una app móvil en la que reproducir mis escenas de imágenes aumentadas, detectando con la cámara del teléfono la imagen que establecí en el editor como disparador de la escena.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.17. RA-11.2: Reproducción de escenas por superficies

Como usuario quiero poder reproducir en una app móvil en la que reproducir mis escenas por superficies, las cuales podré colocar en sitios como mesas o suelos.

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.18. RA-11.3: Reproducción de escenas geoespaciales

Como usuario quiero poder reproducir en una app móvil en la que reproducir mis escenas geoespaciales, en las cuales podré ver los modelos solo si estoy cerca de ciertas coordenadas que introduje en el editor a la hora de crear la escena.

- Prioridad: **MID**
- Requisitos: Ninguno

4.2.19. RA-12: Inicio de sesión en la app móvil

Como usuario quiero iniciar sesión en la app móvil para acceder a las escenas que previamente he creado

- Prioridad: **HIGH**
- Requisitos: Ninguno

4.2.20. RA-12.1: Listado de escenas

Como usuario quiero tener en la app móvil un listado de todas las escenas que he creado para poder seleccionarla y reproducirla.

- Prioridad: **MID**
- Requisitos: Ninguno

4.2.21. RA-12.1: Filtrado de escenas

Como usuario quiero en el listado de escenas poder filtrarlas por tipo o por nombre, para poder encontrar la que me interesa rápidamente.

- Prioridad: **LOW**
- Requisitos: Ninguno

4.2.22. RA-13: Exportación de escenas

Como usuario quiero poder exportar mi escena y modelos a un formato que permita ser utilizado por aplicaciones móviles de realidad aumentada.

- Prioridad: **HIGH**
- Requisitos:
  - **RNF-13.1**: Las escenas con varios modelos cargados deben exportarse como un solo modelo.

4.3. Sprint 0

El sprint 0 tiene la el objetivo de definir todo el trabajo que se va a realizar y cómo. Se estudia el problema para saber si es viable construir una solución. Esta iteración especial tuvo una duración de una semana. Aquí se realizaron las labores de investigación que se ha persebntado en los capítulo 2 y 3 de este trabajo. También se crea el *product backlog*, la lista de historias de usuario que recoge la visión de los requisitos principales de la aplicación desde la perspectiva de sus usuarios que ya se redactó en el apartado anterior.

Otra actividad esencial de esta etapa es la definición y temporicación de los sprints que se desarrollarán en todo el desarrollo. Se decidió establecer un total de **cinco sprints de dos semanas cada uno** (sin incluir este). Las historias de usuario se dividirían entre los distintos sprints. Para cuando acaben cada uno de ellos, debe resultar una versión entregable del producto que implemente las funcionalidades y requisitos especificados por sus historias de usuario. Para ello, de cada historia de usuario se extraerá al principio de cada sprint una serie de **tareas** que completar para avanzar el desarrollo. Se dividieron las historias de tal forma que las que se encasillen en un mismo sprint, guarden cierta relación entre ellas tematizando así las labores que se desarrollan en cada iteración. También se tuvieron en cuenta la prioridad de las historias, dejando la mayoría de las menos relevantes para sprints finales, por si hubiera complicaciones en alguno de los anteriores, priorizar las funcionalidades más esenciales.

Se definió también la temporización de estos sprints, comenzando el desarrollo el 17 de abril de 2023 y finalizando el 25 de junio de 2023:

- **Sprint 1**: 17/4/23 a 30/4/23
- **Sprint 2**: 1/5/23 a 14/5/23
- **Sprint 3**: 15/5/23 a 28/5/23
- **Sprint 4**: 29/5/23 a 11/6/23

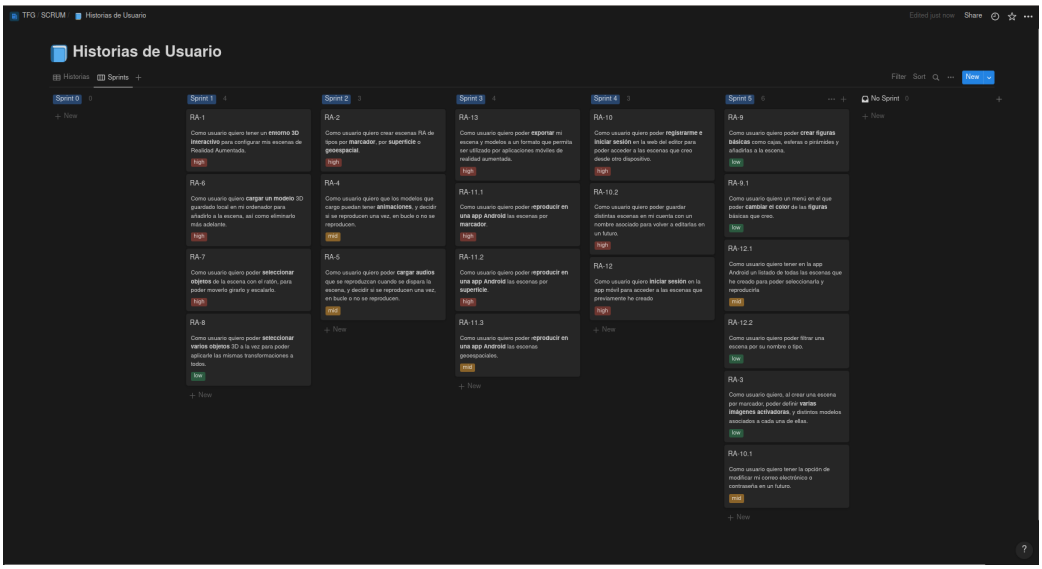

Figura 4.1: Distribución de las historias de usuario por sprints

- **Sprint 5**: 12/6/23 a 25/6/23

Para cada sprint se tendrá un tablero de tareas en el que de un vistazo el desarrollador podrá obtener una idea de en qué estado del desarrollo se encuentra. Para ello se organizará la tabla en columnas, bajo las que se colocarán las distintas tareas según su estado, que pueden ser:

- **Backlog**: Listado de tareas por implementar en el sprint
- **Implementación**: Las tareas que están en proceso de implementación en este momento.
- **Terminada**: Tareas finalizadas por completo.
- **Diseño**: Tareas que están en fase de diseño porque requieren de la elaboración de diagramas o similares.
- **Investigación**: Lista de tareas que requieren alguna labor de investigación adicional que no se había previsto.
- **Bloqueado**: Tareas que no pueden ser implementadas hasta que finalicen otras o que se han descartado por alguna razón.

Otra tarea esencial del sprint 0 es definir las tecnologías y herramientas que se usarán para el desarrollo para tener una visión más amplia del producto final y reducir la incertidumbre del desarrollo, por lo que es lo que se va a proceder a describir.

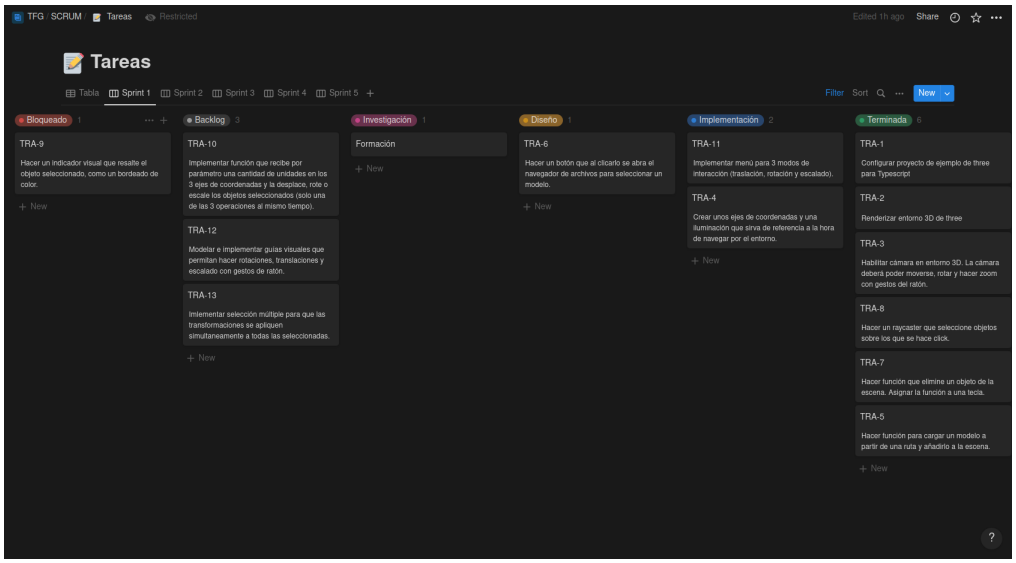

Figura 4.2: Ejemplo de tabla para el sprint 1

Como software de gestión de proyecto se optó por usar Notion [?]. Aquí se gestiona todo lo relacionado con la planificación del trabajo: historias, tareas, sprints, requisitos, tableros, etc. Se contemplaron otras opciones, principalmente *Jira* [?], pero era una herramienta demasiado compleja ya que está contemplada para el trabajo en equipo y en este caso el desarrollador era una única persona, por lo que era un software que venía demasiado grande para el propósito. Notion por otro lado, aunque no esté directamente diseñado para estos porpósitos, permite la creación de tablas, bases de datos y tableros, hiperenlaces a distintos documentos dentro de un espacio de trabajo y almacenamiento en la nube. Como software de gestión de proyectos se empleó *Git y GitHub*, ya que es la plataforma más utilizada y con la que se contaba experiencia previamente, tenía todas las funcionalidades que se requerían para el proyecto. Se creó un repositorio[2] público desde el primer momento con el proyecto íntegro publicado bajo licencia de software libre.

En cuanto a desarrollar aplicaciones web se refiere, *JavaScript* es la única opción existente en cuanto a lenguajes ya que es el único que pueden ejecutar todos los navegadores, así que no hubo muchas dudas en ese aspecto. Aunque no se optó exactamente por *JavaScript*, si no por *TypeScript*, una extensión sintáctica para el lenguaje que añade la opción de introducir tipado estático a los objetos. Se tomó esta decisión porque se consideró que a cambio de invertir un poco más de tiempo en escribir el código, se podría ahorrar tiempo de depuración. Esto es posible ya que gracias a declarar los tipos de las variables, el editor puede detectar errores antes incluso de la ejecución, como por ejemplo intentar añadir

[2] https://github.com/pabloMillanCb/tfg

un objeto *String* a un array de objetos *Object 3D*, evitándose así sorpresas desagradables en la ejecución.

Como se pretende desarrollar un editor que muestre y trabaje con modelos 3D, se necesitará un motor de gráficos por ordenador operable desde *JavaScript* y con soporte para web. Las dos opciones que se tuvieron en cuenta fueron *Three.js* [**?**] y *Babylon.js* [**?**], dos librerías *JavaScript* para la renderización de gráficos 3D. Se optó por coger la primera, ya que tras analizar ambas no se encontró ninguna ventaja significative de una respecto a otra, y para *Three.js* se contaba ya con experiencia, lo que aceleraría el desarrollo.

Para construir la aplicación web se barajaron dos opciones: *Angular* [**?**], un framework completo para *JS* para el desarrollo de web y móvil o *React.js* [**?**], una librería *JS* para el desarrollo frontend. Se escogió esta última ya que, aunque *Angular* es más completo y no necesita librerías externas para realizar ciertas funcionalidades, React es mucho más simple e intuitivo. Su curva de aprendizaje es mucho menos pronunciada, casi inmediata si se tiene conocimiento en *JavaScript* como era el caso. Como no se tenía experiencia en ninguna de las dos herramientas, se optó por esta para reducir el periodo de formación requerido.

Para el visor de las escenas en móvil se decidió hacer una aplicación Android. Se es consciente de el hecho que supone una desventaja respecto a la versión web de otras opciones del mercado ya que en dispositivos iOS no funcionaría, pero se decidió apostar por una aplicación nativa ya que no la ofrece nuinguna otra propuesta. Se escogió Android porque es la plataforma mayoritaria en el mercado móvil. Para el desarrollo de la aplicación se escogió usar *Android Studio* frente a *React Native*. Ambos permiten escribir aplicaciones nativas, pero la primera tiene una mayor integración con *Sceneview*, la librería escogida para las funcionalidades de Realidad Aumentada. Se programará en Kotlin en lugar de Java, ya que es la recomendación oficial de Google y la forma más moderna de hacerlo.

Para las funcionalidades de Realidad Aumentada se optó por ARCore [6], el SDK de Google para esta tecnología. El único problema que se presentaba con esta decisión es que ARCore renderiza objetos 3D a través de OpenGL, una API de bajo nivel que ralentizaría bastante el deasrrollo. Para paliarlo se adoptó Sceneview [9], una librería para Kotlin que envuelve a ARCore y hace el renderizado mucho más simple. ARCore recomienda el uso de modelo 3d en formato *.glb*. Esto es debido a *Filament* [**?**], el motor que implementa por debajo. Es el formato que más garantías de compatibilidad tiene con la tecnología. Por lo que se definió que el sistema trabajaría y admitiría únicamente modelos con esta extensión.

Para las necesidades del proyecto era evidente que haría falta algún tipo de base de datos para almacenar información como los usuarios, las escenas de estos y los recursos como modelos y audio que componen las escenas. Era necesario desarrollar una API REST, una interfaz que emplean dos sistemas informáticos para intercambiar información de manera segura a través de la

web. Para ello se pensó en un primer momento hacerla con PHP y MYSQL, pero finalmente se decidió usar *Firebase* [8]. Esta herramienta viene con muchas utilidades ya implementadas para que el desarrollador pueda hacer uso de estas con llamadas simples a su API, además de no invertir tiempo extra de desarrollo. Trae incorporado un servicio para el registro e inicio de sesión de usuarios **con contraseña encriptada** de forma automática, almacenamiento para archivos binarios y una base de datos no relacional entre otros. Para lo sencillo que se previó que iba a ser la infraestructura del backend (solo se necesitarían tablas para usuarios y escenas, además de una forma de guardar binarios en la base de datos) era una opción perfecta que se acomodaba a las necesidades y reducía mucho la complejidad del desarrollo.

Si bien Firebase ofrece todos esos servicios, seguía haciendo falta una API que lo comunicara con los clientes web y Android. Se decidió usar *Node.js* [5] junto a *Express* [3], ya que es una dupla ampliamente usada para el desarrollo de aplicaciones con Firebase con muchos recursos disponibles en internet. Además, no habría que familiarizarse con lenguajes nuevos, pues se programaría en *JavaScript*

4.4. Sprint 1

Para este sprint se centraron esfuerzos en desarrollar las funcionalidades básicas del editor. El objetivo era que al final del sprint se tuviera un entorno 3D con una cámara la cual pudieras mover y rotar con el ratón, que hubieran varios objetos de prueba que fueran seleccionables con un click del ratón, además de la opción de cargar modelos. También se podrían aplicar transformaciones a través de un menú. Para realizar todo esto se añadieron al sprint la RA-01, RA-06, RA-07 y RA-08. De cada historia se extrae un backlog de tareas que se detalla a continuación.

Tarea: TRA-01	**Horas estimadas: 3**	**Horas totales: 1**	**HU asociada: RA-01**
Configurar proyecto de ejemplo en Three.js con TypeScript			

Tarea: TRA-02	**Horas estimadas: 1**	**Horas totales: 1**	**HU asociada: RA-01**
Renderizar entorno 3D de Three.js			

Tarea: TRA-03	**Horas estimadas: 2**	**Horas totales: 2**	**HU asociada: RA-01**
Habilitar cámara en entorno 3D. La cámara deberá poder moverse, rotar y hacer zoom con gestos del ratón.			

Tarea: TRA-04	Horas estimadas: 3	Horas totales: 0.5	HU asociada: RA-01
Crear unos ejes de coordenadas y una iluminación que sirva de referencia a la hora de navegar por el entorno.			

Tarea: TRA-05	Horas estimadas: 1	Horas totales: 0.1	HU asociada: RA-06
Hacer una función para cargar un modelo a partir de una ruta y añadirlo a la escena.			

Tarea: TRA-06	Horas estimadas: 1	Horas totales: 1	HU asociada: RA-06
Hacer un botón que al clicarlo se abra el navegador de archivos del navegador para seleccionar un modelo y cargarlo.			

Tarea: TRA-07	Horas estimadas: 1	Horas totales: 0.5	HU asociada: RA-06
Hacer función que elimine un objeto de la escena.			

Tarea: TRA-08	Horas estimadas: 2	Horas totales: 2	HU asociada: RA-07
Hacer un raycaster que seleccione objetos sobre los que se hace click.			

Tarea: TRA-09	Horas estimadas: 3	Horas totales: 0.5	HU asociada: RA-07
Hacer un indicador visual que resalte el objeto seleccionado, como un bordeado de color.			

Tarea: TRA-10	Horas estimadas: 2	Horas totales: 2	HU asociada: RA-07
Implementar función que recibe por parámetro una cantidad de unidades en los 3 ejes de coordenadas y la desplace, rote o escale los objetos seleccionados (solo una de las 3 operaciones al mismo tiempo).			

Tarea: TRA-12	Horas estimadas: 8	Horas totales: 8	HU asociada: RA-07
Modelar e implementar guías visuales que permitan hacer rotaciones, translaciones y escalado con gestos de ratón.			

Tarea: TRA-13	Horas estimadas: 1	Horas totales: 2	HU asociada: RA-08
Imlementar selección múltiple para que las transformaciones se apliquen simultaneamente a todas las seleccionadas.			

El transcurso del sprint fue sin demasiadas complicaciones. Se terminaron todas las tareas a excepción de dos. La TRA-09 decidió cancelarse debido a que se pensó que en realidad el *gizmo* que se implementaría para transformar los objetos en sprints posteriores ya era suficiente indicador visual para saber queun objeto estaba seleccionado. La TRA-11 describía un menú para seleccionar

los tres posibles modos de interacción con los objetos (translación, rotación y escalado). Se canceló debido a que a lo largo del sprint se pensó que sería mejor esperar a tener diseños para toda la interfaz e implementarlo todo de golpe, que hacer ahora un menú que con toda probabilidad sería descartado, así que simplemente cada funcionalidad se asignó a una tecla. La gráfica de burndown del sprint es la siguiente:

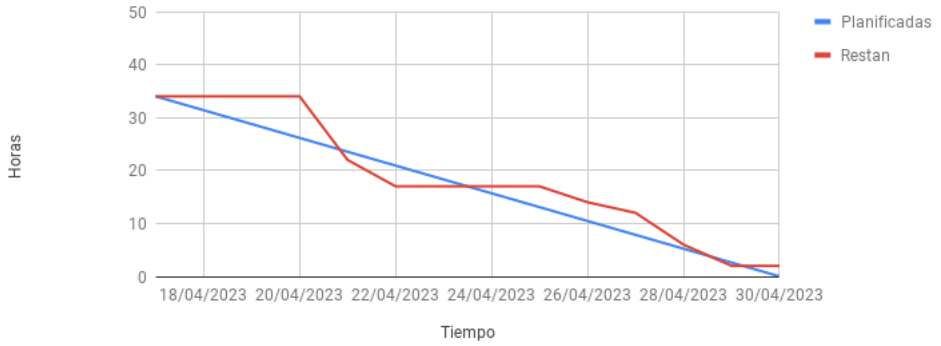

Figura 4.3: Gráfica de la progresión del Burndown en el Sprint 1

4.5. Sprint 2

En este segundo sprint se centró el trabajo principalmente en implementar toda la interfaz de usuario que correspondería a la página del editor, incluso si muchas de las funciones accesibles desde este no estaban implementadas. Esto requirió una formación continua a lo largo del sprint en React, tecnología para la que no se contaba experiencia hasta la fecha. Además se implementaría la reproducción de animaciones para los modelos cargados y de audios cargados a la escena. Las historias de usuario incluidas en este sprint son RA-02, RA-04 y RA-05.

Tarea: TRA-14	**Horas estimadas: 3**	**Horas totales: 1**	**HU asociada: RA-02**
Hacer diseños de la interfaz de usuario del editor.			

Tarea: TRA-15	**Horas estimadas: 15**	**Horas totales: 14**	**HU asociada: RA-02**
Implementar la interfaz de usuario en React			

Tarea: TRA-16	**Horas estimadas: 2**	**Horas totales: 2**	**HU asociada: RA-04**
Implementar función que reproduzca una animación en todos los objeto 3D de la escena si es que tienen.			

Tarea: TRA-17	Horas estimadas: 1	Horas totales: 0.5	HU asociada: RA-05
Implementar una función que cargue un audio desde un archivo seleccionado desde el navegador de archivos del sistema.			

Tarea: TRA-18	Horas estimadas: 1	Horas totales: 0.5	HU asociada: RA-05
Implementar una función que reproduzca el audio cargado de la escena.			

Tarea: TRA-19	Horas estimadas: 2	Horas totales: 4	HU asociada: RA-02
Asignar a cada elemento de la interfaz de usuario su funcionalidad correspondiente del editor si es que ya está implementada.			

Tarea: TRA-20	Horas estimadas: 2	Horas totales: 3	HU asociada: RA-02
Migrar el código del editor en Three.js a React.			

Tarea: TRA-21	Horas estimadas: 1	Horas totales: 1	HU asociada: RA-02
Hacer una página de placeholder que sea la inicial al abrir la aplicación web. Tendrá un único botón que llevará a otra página donde se encuentra el editor.			

Además, durante el transcurso del sprint se añadió una nueva tarea que no se había contemplado al inicio:

Tarea: TRA-22	Horas estimadas: 1	Horas totales: 0.5	HU asociada: RA-02
Mudar la lógica de control de la escena de EditorComponent.tsx a una clase Typescript.			

Para comentar el transcurso de este sprint debemos observar antes la gráfica del burndown.

Como se puede observar, la línea roja del progreso no llega a tocar el eje X de la gráfica, representando que efectivamente quedaron algunas tareas sin completar. Esto se achaca a un problema puntual de salud y de trabajo externo, no a una mala planificación o previsión. Se llega a este razonamiento porque, de las 28 horas planificadas, se realizaron **24,5 netas y 21,5 reales**. No se habían subestimado las tareas en si, si no el tiempo que se tendría disponible para trabajar en el proyecto durante la franja de la iteración. Independientemente de la razón, el proyecto estaba sufriendo un pequeño retraso, y había que paliarlo de alguna manera. Como opciones se presentaban alargar algunos días el sprint, o pasar las tareas sin completar a la siguiente iteración. Se optó por esto último. Las tareas TRA-21 y TRA-16 se terminarían en el Sprint 3.

Figura 4.4: Gráfica de la progresión del Burndown en el Sprint 2

4.6. Sprint 3

Este sprint orbitó alrededor del desarrollo de las funcionalidades básicas para la app Android. Se pretendía tener funcionales los tres tipos de escenas en esta. Además se implementaría la opción de exportar y descargar escenas desde el editor web. Por último se completarían las tareas atrasadas del sprint anterior. Las historias que describen este sprint son RA-13, RA-11.1, RA-11.2, RA-11.3.

Tarea: TRA-22b	**Horas estimadas: 0.5**	**Horas totales: 0.5**	**HUs RA-11.1 RA-11.2 RA-11.3**
Realizar configuración inicial del proyecto de Android Studio.			

Tarea: TRA-23	**Horas estimadas: 2**	**Horas totales: 0.5**	**HUs RA-11.1 RA-11.2 RA-11.3**
Implementar reproducción de sonidos en la app de Android Studio.			

Tarea: TRA-24	**Horas estimadas: 2**	**Horas totales: 2**	**HUs RA-11.1 RA-11.2 RA-11.3**
Implementar escena AR de ejemplo de la documentación de Sceneview.			

Tarea: TRA-25	**Horas estimadas: 3**	**Horas totales: 2**	**HU asociada RA-11.1**
Implementar modo de escenas por marcador en la app Android siguiendo el ejemplo de la documentación.			

Tarea: TRA-26	**Horas estimadas: 3**	**Horas totales: 3**	**HU asociada RA-11.3**
Implementar modo de escenas geoespaciales en la app Android siguiendo el ejemplo de la documentación.			

Tarea: TRA-27	Horas estimadas: 3	Horas totales: 2	HU asociada RA-11.2
Implementar modo de escenas de posicionamiento sobre superficies en la app Android siguiendo el ejemplo de la documentación.			

Tarea: TRA-28	Horas estimadas: 2	Horas totales: 0.5	HUs RA-11.1 RA-11.2 RA-11.3
Implementar la reproducción de animaciones para los modelos en caso de que estos tengan y así se especifique en el JSON de entrada.			

Tarea: TRA-29	Horas estimadas: 8	Horas totales: 6	HU asociada RA-13
Implementar una función que exporte todos los modelos de la escena a un solo modelo glb y lo descargue.			

Tarea: TRA-30	Horas estimadas: 0.5	Horas totales: 1	HU asociada RA-13
Diseñar un archivo JSON para la salida del editor y la entrada de la app. Este deberá indicar los parámetros necesarios para la configuración de una escena, como el tipo de la misma, si tiene audio o animaciones, sus modelos, etc.			

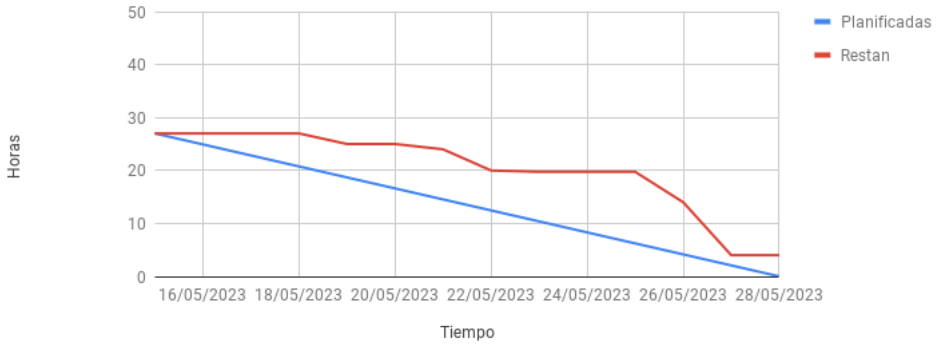

Figura 4.5: Gráfica de la progresión del Burndown en el Sprint 3

En esta iteración se cumplieron la mayoría de los objetivos que se propusieran. La funcionalidad básica de la app Android estaba completa, pero la TRA-29 se había quedado a medias de la implementación. Era un retraso bastante mínimo así que se decidió pasar lo que restaba de la tarea al siguiente sprint de nuevo.

4.7. Sprint 4

Esta iteración se implementarían las funcionalidades de cuentas de usuario e inicio de sesión, y para ello habría que desarrollar un backend al que pudieran

conectarse tanto el editor web como la app y hacer las llamadas que necesitaran a la API para operaciones como crear un usuario o actualizar una escena. Sin embargo, a lo largo de la formación que se realizó en estas tecnologías y antes de redactar las tareas de la iteración, se llegó a la conclusión que se necesitaba más tiempo de desarrollo para implementarlo que lo que se había planeado para este sprint. Sobre todo ocurría que con todo el código que había que preparar tanto en el backend como en el frontend para conectar ambos, a penas daba tiempo a implementar correctamente las interfaces de usuario que se requerían. Además había que recuperar un pequeño retraso del sprint anterior.

Esto no supuso ningún golpe para el proyecto, ya que se anticipó una posible situación similar. El area en el que menos experiencia se tenía a la hora de comenzar el desarrollo era en el desarrollo de backend. Y aunque se hicieron estimaciones, se contemplaba la posibilidad de que pudieran ser erroneas desde un inicio. Es por ello que para el sprint 5 se designaron las historias de usuario de menor prioridad de las cuales podía prescindir el proyecto si se diera el caso como la creación de figuras básicas, la filtración de escenas o las múltiples imágenes activadoras en imágenes aumentadas. Por tanto se tomó la decisión de descartar esas historias y re organizar el Sprint 5. En el 4 se realizaron las siguientes tareas.

Tarea: TRA-31	**Horas estimadas: 1**	**Horas totales: 2**	**HU asociada RA-10**
Crear proyecto base de Node.js con Express.			

Tarea: TRA-32	**Horas estimadas: 1**	**Horas totales: 1**	**HU asociada RA-10**
Crear y configurar proyecto de Google Firebase, obteniendo las claves secretas necesarias e incluyéndolas en el proyecto de Node.js.			

Tarea: TRA-33	**Horas estimadas: 1**	**Horas totales: -**	**HU asociada RA-10**
Implementar función en Node.js para el inicio de sesión de un usuario.			

Tarea: TRA-34	**Horas estimadas: 1**	**Horas totales: -**	**HU asociada RA-10**
Implementar función en Node.js para la creación de una cuenta de usuario.			

Tarea: TRA-35	**Horas estimadas: 1**	**Horas totales: 1**	**HU asociada RA-10.2**
Implementar función en Node.js para la creación de una nueva escena.			

Tarea: TRA-36	**Horas estimadas: 1**	**Horas totales: 1**	**HU asociada RA-10.2**
Implementar función en Node.js para eliminar una escena a partir de su identificador.			

Tarea: TRA-37	Horas estimadas: 1	Horas totales: 1	HU asociada RA-10.2
Implementar función en Node.js para actualizar una escena ya existente.			

Tarea: TRA-38	Horas estimadas: 1	Horas totales: 1	HU asociada RA-10.2
Implementar función en Node.js para obtener todas las escenas de un usuario concreto.			

Tarea: TRA-39	Horas estimadas: 1	Horas totales: 2	HU asociada RA-10.2
Configurar Firebase Storage para poder almacenar objetos binarios como imágenes o modelos 3D y que sean accesibles desde una url para descargarlos.			

Tarea: TRA-40	Horas estimadas: 1	Horas totales: 1	HU asociada RA-10.2
Diseñar las tablas de bases de datos para usuarios y escenas.			

Tarea: TRA-41	Horas estimadas: 2	Horas totales: 2	HU asociada RA-10.2
Implementar las tablas de bases de datos en Google Firebase.			

Tarea: TRA-42	Horas estimadas: 2	Horas totales: 2	HU asociada RA-12
Configurar acceso a Firebase Storage desde la app Android.			

Tarea: TRA-43	Horas estimadas: 10	Horas totales: 8	HU asociada RA-10
Configurar acceso a Firebase Storage y API desde React. Asignar a cada menú de la aplicación su llamada correspondiente.			

Tarea: TRA-44	Horas estimadas: 1	Horas totales: -	HU asociada RA-10
Implementar función en Node.js para la modificación de datos de un usuario.			

Tarea: TRA-45	Horas estimadas: 1	Horas totales: -	HU asociada RA-10
Implementar función en Node.js para la eliminación de un usuario.			

Algo importante a destacar en este Sprint es que las tareas TRA-33, TRA-34, TRA-44 y TRA-45 fueron descartadas durante el desarrollo. Esto se debe a que, como se comentará en el capítulo de Implementación, se decidió optar por otra vía en cuanto a la gestión de cuentas de usuario se refiere, por lo que ya no sería necesario crear estas llamadas en la API. En esta iteración se dispuso de más tiempo, lo cual se reflejó en la compleción de todas las tareas a tiempo.

Figura 4.6: Gráfica de la progresión del Burndown en el Sprint 4

4.8. Sprint 5

Este sprint en un inicio se planificó como un colchón en caso de que la planificación no fuera al ritmo esperado en el resto de sprints. Se desarrollarían las historias RA-9, RA-9.1, RA-12.2 y RA-3, de prioridad LOW, sin las cuales el sistema podía existir sin perder demasiado. Como ya se comentó en anteriores sprints, se descartó la implementación de todas ellas, y en su lugar se continuó con RA-10, RA-10.2 y RA-12 del cuarto sprint, que requerían de más tiempo además de RA-10.1 y RA-12.1 de prioridad MID que sí estaban programadas para esta iteración.

Tarea: TRA-47	**Horas estimadas: 4**	**Horas totales: 4**	**HU asociada RA-12.1**
Implementar menú de selección de escena en la aplicación Android.			

Tarea: TRA-48	**Horas estimadas: 4**	**Horas totales: 4**	**HU asociada RA-10.1**
Implementar menú de selección de escena la aplicación web.			

Tarea: TRA-49	**Horas estimadas: 2**	**Horas totales: 4**	**HU asociada RA-12.1**
Configurar la API de acceso al backend para que sea accesible desde otros dispositivos de la red.			

Tarea: TRA-50	**Horas estimadas: 3**	**Horas totales: 3**	**HU asociada RA-10**
Implementar funciones para el inicio de sesión y registro de un usuario a partir de un correo y contraseña en la aplicación web.			

Tarea: TRA-51	Horas estimadas: 2	Horas totales: 2	HU asociada RA-12
Implementar inicio de sesión con usuario y contraseña en la aplicación Android.			

Tarea: TRA-53	Horas estimadas: 2	Horas totales: 4	HU asociada RA-12.1
Implementar petición GET desde la app Android a la API para obtener las escenas creadas por un usuario.			

Tarea: TRA-55	Horas estimadas: 1	Horas totales: 1	HU asociada RA-10.1
Implementar menú para el cambio de credenciales de un usuario en el editor web.			

Tarea: TRA-56	Horas estimadas: 3	Horas totales: 3	HU asociada RA-10
Proteger las rutas que no sean la página de login del editor web para solo poder acceder a ellas cuando hay una sesión iniciada.			

Tarea: TRA-57	Horas estimadas: 10	Horas totales: 7	HU asociada RA-10.2
Implementar la carga, guardado y borrado de una escena a partir del JSON obtenido por la API con Firebase Storage en el editor web.			

Tarea: TRA-58	Horas estimadas: 2	Horas totales: 1.5	HU asociada RA-10.2
Añadir animación de carga para momentos en los que se esté esperando la respuesta a una petición o la carga de algún recurso desde el editor web.			

Tarea: TRA-59	Horas estimadas: 3	Horas totales: 2	HU asociada RA-10.2
Implementar menús pop-up de confirmación para las acciones de eliminar una escena, abandonar el editor y cerrar sesión desde el editor web.			

El sprint finalizó sin mayores complicaciones, teniendo el producto final listo. A estas alturas se notó la experiencia obtenida en el proceso de planificación, el cual fue preciso. Una vez terminado el producto solo restaba desplegar la aplicación.

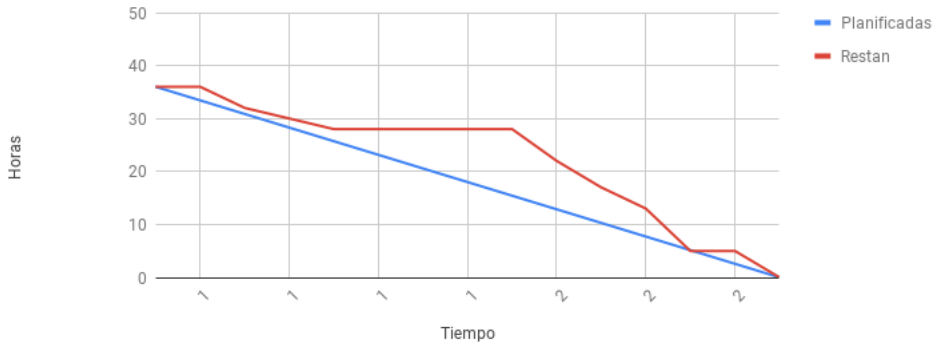

Figura 4.7: Gráfica de la progresión del Burndown en el Sprint 5

Capítulo 5

Implementación

En este capítulo se describirá en detalle la implementación del sistema y su diseño. Se comentarán las dificultades más relevantes que ha planteado el desarrollo y cómo se ha optado por resolverse. El repositorio[1] del proyecto está dividido en cuatro directorios:

- **editor**/: Contiene la aplicación web con el editor 3D.
- **visor**/: Aplicación Android para la reproducción de escenas.
- **backend**/: Servidor web que conecta las aplicaciones con la base de datos.
- **doc**/: Documentación del proyecto.

Se dedicará una sección a cada uno de los tres primeros directorios.

5.1. Aplicación web

Como ya se mencionó en capítulos anteriores, se usaría la biblioteca de *React.js* [?] para desarrollar la aplicación web. Para montar el proyecto se empleó **Vite** [?], que permite crear un proyecto de React + TypeScript de forma automática.

El bloque de construcción básico de las aplicaciones en React son los **componentes**. Estos son elementos funcionales de interfaz, reutilizables, anidables y parametrizables. Se escriben en la extensión de TypeScript *.tsx*, y se encuentran en la carpeta *src/components* del proyecto. Un componente puede ser por ejemplo un botón que haga una acción específica, o la cabecera de una página,

[1] https://github.com/pabloMillanCb/tfg

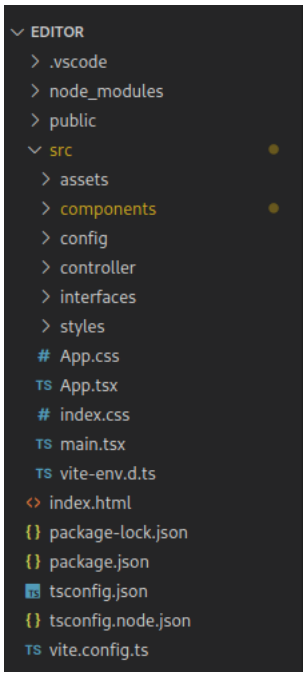

Figura 5.1: Estructura de directorios final de la aplicación web de React

un menú desplegable, etc. El corazón de la aplicación se encuentra en el componente **EditorComponent.tsx**, que gestiona el editor 3D y su interacción con el usuario.

5.1.1. Renderización de entorno 3D

Lo primero que se realizó, antes de implementar el componente en si con toda la interfaz, fue crear el entorno 3D interactuable. Para ello se creó la clase **EditorScene.ts**, que hereda de la clase *Scene* de la librería de Three.js. La clase *Scene* se encarga de gestionar como su nombre indica la escena entendida como el espacio donde habitan los distintos objetos 3D de nuestra aplicación. En Three.js, los espacios 3D se representan con una estructura de datos en forma de árbol. Los objetos son hijos unos de otros, y el objeto especial *scene* es el nodo raíz. Un objeto puede establecerse como hijo de otro objeto a su vez, haciendo que todas las transformaciones que se apliquen al padre también lo hagan en el hijo.

Para poder visualizar un entorno 3D de Three.js se necesita un objeto *Scene*, una cámara en forma de *PerspectiveCámera* y un renderizador *WebGLRenderer*. La cámara se puede entender conceptualmente como si fuera una cámara real. Es un objeto que está en una posición determinada del espacio 3D y apuntando en una dirección. Es la que se encarga de definir qué parte de la escena es la que

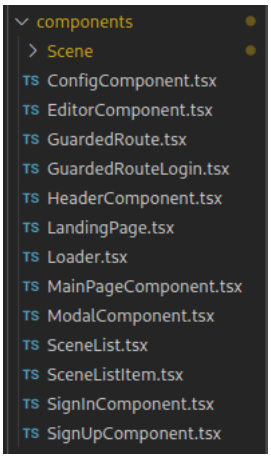

Figura 5.2: Lista de componentes del proyecto de la apliación web.

se va a ver por pantalla. Esta información se transmite al renderizador, que se encarga de aplicar cálculos para convertir el espacio 3D de una representación abstracta a una imagen en dos dimensiones que se puede monstrar por pantalla y que el usuario puede entender. Los tres elementos se inicializan y mantienen desde **EditorSceneController.ts**. Esta clase hace de interfaz e implementa las llamadas necesarias para que el componente *EditorComponent.tsx* pueda realizar cambios en la escena y cámara a través de su interfaz de usuario.

5.1.2. Controles de cámara y selección de objetos

Para la manipulación de la cámara se adoptaron controles estándares presentes en otros programas de edición: arrastrando con el click derecho del ratón se rota la cámara sobre un punto, y arrastrando con la rueda del ratón pulsada de desplaza. Así queda el click izquierdo libre para las interacciones con los objetos.

El usuario debe poder seleccionar un objeto de la escena colocando el ratón sobre y haciendo click. Esto le servirá para, una vez elegido un modelo, poder manipularlo moviéndolo, rotándolo o escalándolo sin afectar al resto de objetos de la escena. Esto presenta el problema de cómo saber cual es el objeto que se ha seleccionado. Para ello se utilizó un **raycaster**. Es un recurso muy usado en la informática gráfica y disponible en la librería de Three.js en el que se dispara una suerte de haz o rayo desde un punto de la escena y en una dirección. Este disparo puede atravesar o no objetos 3D que se encuentre en su camino. En caso afirmativo, devuelve una lista ordenada de los modelos con los que ha colisionado. Para este proyecto se ajustó un raycast desde el punto en el que se encuentra la cámara (es decir, la perspectiva del usuario) y en dirección al punto en el que se hace click. De esta forma el rayo ensarta el punto en el que se ha hecho click en el ratón, y si había un objeto debajo, colisionará con él. En

la clase de raycast que implementa Three se le debe pasar como parámetro una *Scene* o un *Group*, un objeto especial que tiene como hijos una lista de objetos. El rayo solo detectará colisiones con esos objetos. Aparece así otro problema. Si se pasa la *scene* como parámetro se detectaría cualquier objeto que haya en esta, no solo los que haya añadido el usuario.

Se debe distinguir distinguir entre dos tipos de objetos que se encontrarán en la escena: los dinámicos que los añade el usuario y son interactuables por este, y los estáticos, que se crean al iniciarse el espacio 3D automaticamente y tienen de propósito servir de referencia visual, como por ejemplo un *grid* que simboliza dónde está el suelo. Todos estos objetos se deben añadir a la *scene* como hijos de esta usando *scene.add(object)*. Para diferenciar las dos clases de elementos se añade a la escena un *Group* con el nombre de **liveObjects**. Los modelos introducidos por el usuario serán entonces añadidos como objetos a este atributo, y será el parámetro que se pasará al raycaster para que solo reconozca objetos *dinámicos*, solucionando así el problema de reconocimiento. Para eliminar objetos de la escena simplemente habría que desconectar al nodo hijo de su padre.

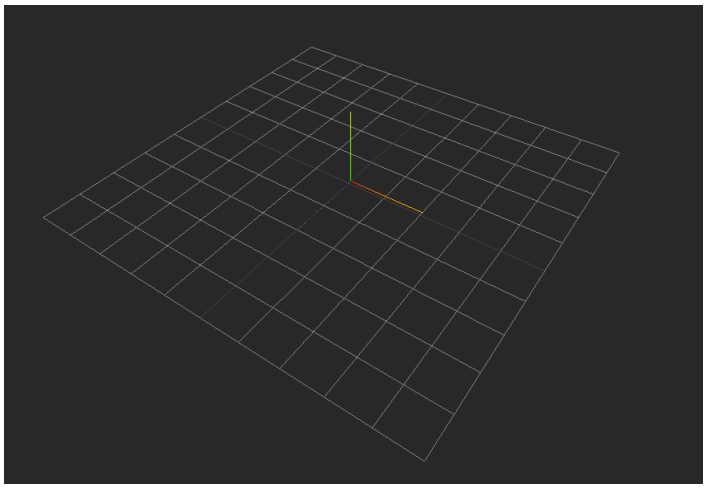

Figura 5.3: Grid del editor de escenas para tener de referencia como suelo.

Sin embargo esto crea otro problema. Se ha comentado anteriormente que un objeto 3D puede ser hijo de otro, y esto es muchas veces el caso con los modelos que se pueden encontrar en la web. Es muy probable que si cargo en la escena a un robot, este esté hecho de múltiples objetos 3D. Uno de estos podría ser su brazo, su cabeza o su cuerpo. Esto es problemático porque si es el caso y hago click en el brazo del robot con la intención de seleccionar el modelo entero, el raycaster solo nos devolverá información sobre el nodo hijo. Para solucionar esto se hizo uso del atributo **name** que tienen todos los *Object3D* de Three. Este atributo es un string con el que se puede asignar un nombre al objeto. En la función que añadía los objetos del usuario al grupo de **liveObjects**, se etiquetan

los *name* de todos los elementos como .ªlive". De esta forma, cuando se obtiene un nodo hijo de un modelo, se puede ir recorriendo el padre de cada nodo hasta encontrar uno cuyo *name* sea igual a .ªlive". Aquí pararía la búsqueda ya que de seguir ascendiendo por sus padres se llegaría a la *scene*, el nodo raíz.

Tras esto el raycaster sería capaz de detectar qué objeto selecciona el usuario. Este se almacenaría en una variable para poder aplicarle las transformaciones pertinentes cuando el usuario lo demande. Si este quisiera dejar de tenerlo seleccionado, deberá hacer click en cualquier punto de la pantalla en el que no haya ningún objeto. Si el raycaster colisiona con un total de 0 modelos, se liberará la variable del objeto seleccionado. Esta variable de la que se habla es en realidad un array de *Object3D*, ya que el usuario tiene la posibilidad de seleccionar múltiples objetos para que las transformaciones que se apliquen afecten a todos simultáneamente. Para ello se hará click en otro objeto mientras se pulsa la tecla *Control*. Este es un esquema de control usado tanto en software del sector como en los gestories de archivos de los sistemas operativos, es un gesto muy reconocido. Si ya había un objeto seleccionado y el usuario está pulsando la tecla mientras elige otro objeto, este se añadirá al array de objetos seleccionados. En caso contrario se vaciaría el array y se incluiría el nuevo objeto seleccionado.

Una vez está resuelta la selección de objetos es necesario implementar la forma en la que el usuario comanda las transformaciones que se aplican sobre estos. Se usarán los **gizmos** comentados en el anterior capítulo para ello. Three incorpora *gizmos* de fábrica bajo la clase *TransformControls*. Esta clase cuenta con tres modos: translación, rotación y escalado. La clase puede .ªgancharse.ª un objeto que se le pase como argumento. Una vez hecho mostrará el gizmo correspondiente en función del modo actual superpuesto sobre el objeto. El usuario podrá hacer click directamente sobre él para manipular el objeto.

Aquí apareció otro problema. La clase solo soporta un objeto .ªganchado.ªl mismo tiempo, por lo que en el caso de tener varios objetos seleccionados no se podrían modificar todos. La primera solución que se intentó dar a esto fue hacer a todos los modelos seleccionados hijos de un nodo temporal vacío el cual se asignaría al *TransformControls*. Esto no funcionó debido a que las transformaciones en three son locales respecto al padre del objeto. Es decir, si se aglutinan los objetos seleccionados en un nodo y se mueven dentro de ese nodo, cuando se devuelven a la escena regresan a su lugar de origen. Esto se arregló con un *dummie*, que es un objeto invisible que sirve como apoyo para hacer ciertas operaciones en la informática gráfica. En este caso, cuando se inicia una selección múltiple se crea un *dummie* en el origen de cordenadas y se asigna a los *TransformControls*. En cada frame, se comprueba cuanto se ha modificado su posición, rotación o escalado y se aplica esa misma variación a todos los objetos de la lista de seleccionados. Con esto la funcionalidad estaba lista.

5.1.3. Reproducción de animaciones

Algunos modelos 3D incluyen animaciones. El usuario cuenta con una opción en su interfaz para poder reproducir la escena, y con ello las animaciones de todos los objetos que se encuentren en esta. Los *Object3D* de Three soportan animaciones, y para poder reproducirlas se hace uso de la clase **AnimationMixer**. Esta recibe un objeto como parámetro y automaticamente extrae toda la información relativa a sus animaciones. Cada animación está identificada con una string que le da nombre como podrían ser "Death", "Walk", Run", etc. Desde *AnimationMixer* se invoca a la función *clipAction(animacion)* y se obtiene un objeto de tipo *AnimationAction*. Desde este último se puede ordenar a las animaciones que se reproduzcan o se detengan.

Como el usuario necesita tener la capacidad de elegir una animación concreta de entre las muchas que puede tener un modelo para que se reproduzca, se necesita registrar en algún sitio cual es la seleccionada. Para ello se hará uso del atributo **userData** de *Object3D*. Este es un objeto genérico JavaScript en el que se puede definir multitud de datos auxiliares relacionados con el modelo. Se define entonces el atributo **animationIndex**, un número entero que va de 0 a n-1, siendo n el número de animaciones que posee ese objeto. Cuando se quiere reproducir la animaciones en la escena, para cada objeto se crea un *AnimationAction* con el índice actual del objeto y se llama al método *animationClip.play()*. Cuando se quiera detener la animación, se ejecutará *animationClip.reset()* para que los modelos vuelvan a su posición inicial y *animationClip.stop()* para que se detengan.

5.1.4. Exportación y carga de escenas

Se procede ahora a hablar sobre la carga y exportación de modelos 3D. Se parte desde la base de que el usuario tiene acceso a archivos *.glb* de forma local en su dispositivo, que lo ha cargado desde la interfaz de la aplicación, se guarda en un objeto JavaScript *File* en la caché y con una url temporal para acceder al recurso (ya se entrará en detalles más adelante sobre esta parte del proceso). Para cargar el modelo se hará uso de la clase de Three.js **GLTFLoader**, que permite convertir un *File* en *Object3D* a partir de su url. Despues se añadiría el *Object3D* como hijo a *liveObjects* y ya sería visible en la escena.

Para la exportación de la escena es estrictamente necesario encontrar la forma de convertirla en un archivo único *.glb*. Para ello se empleó la clase de THREE.js **GLTFExporter**. Esta recibe como argumento un objeto o grupo de objetos y los convierte a un archivo *.gltf*, o *.glb* si se especifica el parámetro *binary* como verdadero. Como resultado se obtiene un buffer con un array de datos *raw*, es decir binarios. Estos datos pueden usarse para crear un objeto *Blob* que se puede descargarse como fichero a través del navegador. Solo es necesario entonces pasarle al **GLTFExporter** el grupo de objetos vivos (los que introdujo

el usuario) *liveObjects* y la clase hará el resto del trabajo. El archivo resultante no solo contendrá el modelado sino las animaciones de todos los objetos de la escena.

Con lo desarrollado hasta ahora sería posible descargar la escena al ordenador como usuario, pero existe un segundo caso en el que se requiere exportar una escena: guardarla en el servidor. Se necesita establecer la forma en la que se van a almacenar los datos de la escena como los objetos que contiene, su posición, rotación y escalado, su índice de animación activa, su lista de animaciones, etc. Para ello se contemplaron dos opciones.

- **Archivo binario único**: Los modelos se exportarían a un fichero similar al descargable por el usuario en el que la escena se convierte en un único modelo. El archivo sería lo que se almacenaría en la base de datos, conteniendo este toda la información sobre los modelos que se tienen y su estado actual. Esto requeriría que fuera posible cargarlo a posteriori en el editor separando todos los modelos para añadirlos de forma individual a la escena y que conserven su independencia.
- **JSON**: Se guardaría en la base de datos una copia de cada modelo en la escena. Adicionalmente, se guardaría un archivo JSON en el que se describen uno por uno la posición, translación y escalado de cada objeto, además de otros parámetros como el *animationIndex* para cada uno de ellos si es que tienen.

La primera opción era la preferible, ya que era una solución mucho más simple y elegante, tanto a la hora de empaquetar los datos como de almacenarlos en la base de datos (solo se guardaría un único archivo con toda la información). Por suerte, cuando *GLTFLoader* carga un archivo *.glb* conserva toda la estructura de nodos del objeto original. Tras hacer algunas pruebas se comprobó que *GLTFExporter*, al recibir un grupo de objetos como entrada para exportar una escena, lo que hace es unirlos todos a un nodo raíz que sigue siendo accesible si volvemos a cargar el modelo exportado con *GLTFLoader*. Por tanto se implementó una función *loadScene* que recorría la lista de hijos del archivo *.glb* recibido y los añadía uno a uno a la escena.

5.1.4.1. Conflicto entre animaciones

Llegados a este punto la escena se cargaba correctamente pero surgió un problema derivado de las animaciones de los archivos exportados. Para explicar la resolución es necesario explicar algunos detalles sobre las animaciones en Three.js.

Las animaciones de un modelo se guardan en un atributo de *Object3D* llamado **animations**, un array de objetos *AnimationClip*. Estos objetos a su vez

tienen entre sus atributos un array de *KeyframeTrack*, objetos que almacenan secuencias de *keyframes* con información sobre las transformaciones que realiza la animación a un objeto. ¿Cómo saben estas estructuras de datos a qué objeto se deben aplicar estos movimientos? **a través del *name*** de los *Object3D* de la escena.

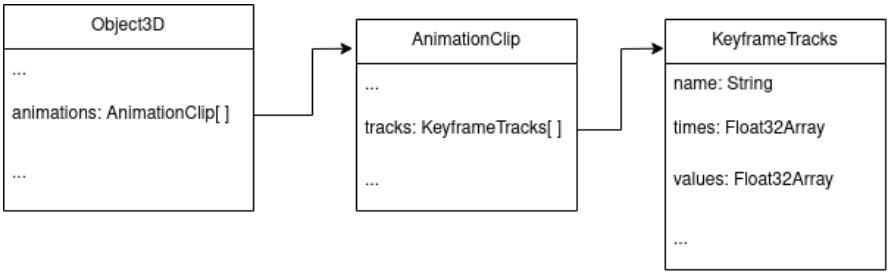

Figura 5.4: Esquema de las estructuras de datos de *Object3D*, *AnimationClip* y *KeyframeTrack* para la reproducción de animaciones.

GLTFExporter al realizar el empaquetamiento de la escena almacena todos los *animations* de los objetos en un único array en el nodo raíz. Por lo tanto al cargar el archivo, no hay forma de saber qué animaciones pertenecían a cada objeto para poder asignarselas posteriormente. Para solucionar esto, antes de añadir un objeto a la escena se crea un array de datos en el atributo *userData* de *Object3D* conteniendo el nombre de todos los *AnimationClip* del modelo. Así, al cargar cada objeto luego, se podría comprobar en la lista general de animaciones del *.glb* y añadir a *animations* las que coincidan en nombre.

Sabiendo todo esto se puede describir el verdadero problema en todo este asunto: **se producían conflictos al exportar y volver a cargar escenas con dos o más modelos idénticos con animaciones**. Este fue un punto crítico del desarrollo, ya que de arreglar esto dependía poder ahorrar luego mucho tiempo de desarrollo en el backend y simplificar el funcionamiento del sistema. Lo que se podía observar al generar este caso concreto es que al cargarse una escena previamente exportada con estas condiciones, solo el primer modelo de todos reproducía las animaciones correctamente.

La razón residía en *GLTFExporter*, que cuando almacena la escena en un objeto tiene un comportamiento muy relevante para este caso: **si dos nodos tienen el mismo nombre, modifica uno de ellos**. Por ejemplo, si hubiera dos nodos llamados *hand*, al segundo lo renombraría como $hand_1$

Conociendo todo esto se puede intuir dónde está el error: si los modelos modelo tenía una animación que apuntaba al nodo *hand*, solo se moverá el primer objeto, ya que los las animaciones del segundo apuntarán también a *hand*, pero el nodo del segundo modelo en realidad se llama $hand_1$, por lo que no se vé afectado por la animación.

Para solucionarlo se hizo uso de otro atributo de *Object3D*, el **id**, una variable numérica que identifica un objeto en una escena. Al cargar nuevos objetos en la escena, se comprueba si coincide con alguno otro. En caso afirmativo, se concatena el *id* del objeto recursivamente a todos sus nodos hijos, y luego se hace lo mismo para el array de *animations*, para que los *KeyframeTrack* apunten a los nodos con el nombre cambiado. Si el *id* de un objeto repetido es *23*, su nodo hijo *hand* pasará a llamarse *23#hand*, evitando así todos los conflictos que puede generar el *GLTFExporter*.

Con esto la exportación y carga de escenas es perfectamente funcional para cualquier caso, y podrá ser usada para almacenarse en la base de datos como un único archivo.

5.1.5. Reproducción de audios

Los audios asociados a las escenas se manejan desde **EditorSceneController.ts**. La clase tiene el atributo *audio* de tipo *HTMLAudioElement*, una clase que permite reproducir audios cargados por *url*. Como se mencionó antes, se da por supuesto que el usuario carga un archivo de audio compatible y este se almacena en la caché accesible desde una url. Con la función *loadAudio* se crea el objeto que gestionará su reproducción con los métodos *playAudio* y *stopAudio*.

5.1.6. Interfaz del componente editor

Ahora se discutirá el **EditorComponent.tsx** en si, el componente que se encarga de renderizar React y que implementa las funcionalidades del editor descrito hasta ahora con la interfaz para que el usuario pueda interactuar con la escena.

Todo lo que a la interfaz se refiere se diseño previamente en **Figma** [?], un software para *mockups* de interfaces de usuario. Todos los elementos como botones, selectores y cajas de texto se implementaron usando *MUI Core*, una librería para React y JavaScript para la creación de interfaces de usuario. Ofrece todos bloques fundamentales que pueden necesitarse para la construcción de aplicaciones web como botones, selectores, scrolls, cajas de texto, iconos, etc. Además adopta *Material*, unas directrices de diseño creadas por Google para interfaces web y móviles. Es lo que se acabaría usando en la aplicación Android, así que de entre todas las librerías que existían con el mismo propósito, se eligió esta para unificar en la medida de lo posible el estilo de las aplicaciones web y de móvil del proyecto.

La interfaz se dividió en tres secciones:

1. La barra superior. Aquí se encontrarían los botones para exportar y guardar la escena, un cuadro de texto para introducir el nombre de la misma, y un

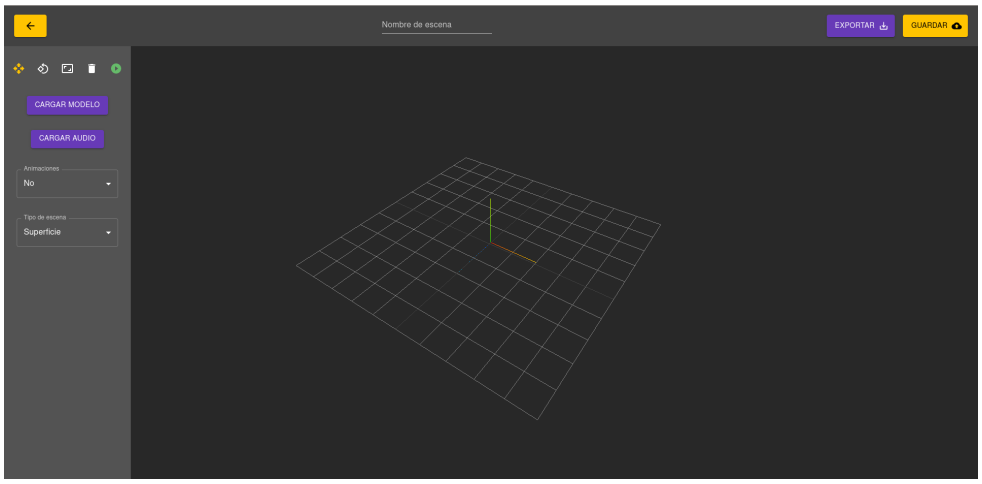

Figura 5.5: Interfaz de usuario final para el editor de escenas.

botón para volver al anterior menú.

2. La barra lateral, un panel de control con todas las opciones para configurar la escena AR.

3. El *canvas*, todo el espacio sobrante entre las dos barras donde se renderizaría el entorno 3D del editor.

En este componente y en el resto de la aplicación se haría uso de los llamados *hooks* de React. Estas son unas funciones JavaScript que entre otras cosas permiten dotar de estado a un componente, ya que de normal no tienen. Por ejemplo, si se quisiera hacer un contador de clicks, este tendría que tener el número de clicks actual guardado. Esto es posible gracias al hook de **useState**, que permite almacenar variables persistentes entre distintas renderizaciones de un componente. Para este proyecto se usa (por poner un ejemplo) para almacenar la instancia de *EditorSceneController* que se crea al iniciar una escena por primera vez y renderizarlo en el *canvas*. Como se ha señalado anteriormente, esta es la puerta de entrada del componente para manipular la escena. Cada vez que se active alguna opción para manipular la escena desde *EditorComponent* lo comunicará llamando a la función pertinente de *EditorSceneController*, como por ejemplo *playAudio()*.

Para el panel de control lateral se tienen las siguientes utilidades:

- **Botones de herramientas**: Con estos botones se puede elegir qué herramienta usar para manipular la escena. Dependiendo de cual se tenga escogida cambiará la forma de manipular un objeto. Cambiará de color en función de la herramienta seleccionada. Se cuenta con las siguientes:

  - Translación: Con esta herramienta aparecerá un *gimzo* de desplazamiento en los objetos seleccionados, con el que se podrá mover de posición.
  - Rotación: De forma similar a la anterior herramienta, con esta se pueden rotar objetos.
  - Escalado: Con esta herramienta se pueden hacer los objetos más grandes o más pequeños en cualquier eje de coordenadas.
  - Papelera: Con esta herramienta seleccionada, los objetos clickados desaparecerán de la escena.

- **Botón de play**: Previsualiza la escena en el editor. Reproduce las animaciones de los modelos de la escena si estos tuvieran y la opción de animaciones estuviera activa y/o el audio si hubiera.

- **Cargar modelo**: Abre el navegador de archivos del dispositivo para seleccionar un archivo *.glb* y añadirlo a la escena. Opcionalmente el usuario puede cargar modelos arrastrándo el archivo a la ventana del navegador.

- **Cargar audio**: Abre el navegador de archivos del dispositivo para seleccionar un archivo de audio para asociarlo a la escena.

- **Animaciones**: Menú desplegable para establecer si reproducir o no las animaciones de los modelos.

- **Tipo de escena**: Menú desplegable para elegir el tipo de escena. Dependiendo del tipo elegido aparecerá abajo una opción adicional para añadir configuración adicional necesaria:

  - Imágenes aumentadas (Marcador): Aparece un botón para cargar una imagen. Esta imagen se pasará a *EditorScene* como objeto *File* y será cargada como textura plana en el suelo de la escena para tenerla como referencia.
  - Geoespacial: Aparecen tres campos numéricos para introducir la latitud, longitud y altura a la que se desea que aparezca la escena.
  - Superficie: No se muestra ningún menú adicional.

5.1.7. Carga de archivos

La carga de archivos como modelos, audio o imagen se gestiona a través de la función *handleLoad*. Esta se llama en un evento *onChange* para los elementos HTML asociados a los botones de carga. Este evento se activará cada vez que se cargue un nuevo archivo y lo pasará como parámetro en forma de objeto *File*. Ahí se comprueba si es uno de los archivos soportados (*.glb* para los modelos,

.mp3 o *.ogg* para los audios y *.jpg*, *.png* o *.svg* para las imágenes). Si la extensión del archivo cae en alguna de estas categorías, se llama a la función de *EditorSceneController* correspondiente: *loadModel*, *loadAudio* o *loadImage*. La clase se encarga de gestionar esas entradas como se ha visto anteriormente.

5.1.8. Gestión de evento asíncrono en el guardado

Para guardar la escena se tienen en la esquina superior derecha dos botones, uno para descargar la escena como modelo (Exportar) y otro para guardarlo en el servidor (Guardar). Se debe tener en cuenta que la función de *EditorScene* que convierte la escena en modelo, *exportScene*, es asíncrona. En el caso de *Exportar* no hay ningún problema, ya que cuando termina el proceso no hay que hacer nada más, es descargado por el navegador, pero en el caso de *Guardar* surgía un inconveniente. Para enviar el archivo a la base de datos (se entrará en maś detalle sobre cómo en la sección del backend) se necesita esperar a que este termine. Por cómo funciona *GLTFExporter*, el resultado de la función *parse* (la que se usa para hacer la conversión) no puede ser devuelto por la función en la que se ejecuta ya que está dentro del contexto de una función que se pasa por argumento a *parse*. Introducir en la clase de *EditorScene* código para hacer peticiones HTTP al backend era posible pero una solución poco elegante ya que ese no es el propósito de la clase

Se optó por hacer en *EditorScene* una versión clónica de *exportScene()* llamada *getBlob(upload: (blob: Blob) =>void)*. Esta recibe como argumento una función, *upload* que a su vez recibe como argumento un *Blob* (equivalente a *File*). Esta función es definida en *EditorComponent* y llamada dentro del contexto de finalización de las labores de *GLTFExporter*. Contiene el código para realizar las peticiones HTTP para enviar la escena al backend. En resumen, desde *EditorScene* se ejecuta código de *EditorComponent* para comunicarse con el servidor cuando la escena está lista para ser enviada.

5.1.9. Navegación entre distintas páginas

React es una librería y no un framework como se mencionó anteriormente. Por ello, de base solo tiene capacidad para soportar *single page applications*. Sin embargo se necesitaba crear una interfaz con distintas páginas: inicio de sesión, configuración de perfil, selección de escena y editor. Para ello se empleó **React Router** [**?**], una librería que permite definir distintas páginas cada una asociada a una url distinta (*web/login*, *web/config*, etc). Se realizó un *mockup* para la navegación del sitio web y sus interfaces.

Las páginas con las que cuenta la aplicación son las siguientes:

- **MainPageComponent.tsx** /: Página principal de los usuarios logeados.

Se muestra el listado de escenas para poder editarlas o eliminarlas además de opciones para crear escena, cerrar sesión o cambiar credenciales.

- **SignInComponent** */login* y **SignUpComponent** */register*: Páginas para crear cuenta e iniciar sesión.
- **ConfigComponent** */config*: Menú para actualizar las credenciales del usuario.
- **EditorComponent** */editor*: Editor de escenas.

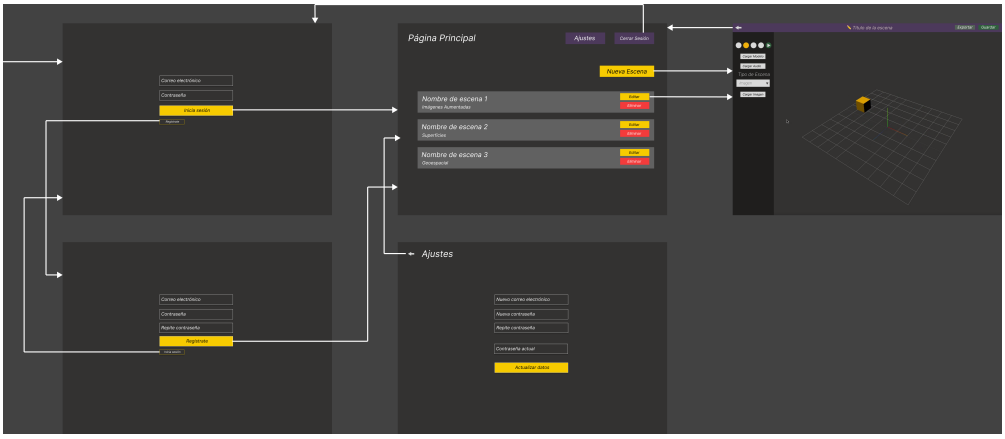

Figura 5.6: Mockup de la UI de la aplicación web realizado en *Figma*

5.1.10. Rutas protegidas

Al ser accesible cualquier página de la aplicación surgía un nuevo problema. En el planteamiento realizado, el usuario debía registrarse o iniciar sesión antes de poder acceder a páginas como su lista de escenas, configuración o el editor (no tendría sentido de otra forma). Se necesitaba entonces *proteger* las páginas sensibles al inicio de sesión bajo la condición de que se hubiera realizado. Esto se implementó con el componente **GuardedRoute.tsx**. Como se mencionó anteriormente, los componentes en React pueden estar anidados. Cada vez que se introduce un componente de alguna de las páginas de la aplicación, como podría ser *EditorComponent* con dirección */editor*, se comprobaba si existe una sesión iniciada. En caso afirmativo renderiza a su hijo, y si no redirige la aplicación a */login* para que el usuario entre en la página de inicio de sesión. Para las páginas de inicio de sesión de registro (cada unas con su propio componente *SignInComponent* y *SignUpComponent*) se seguía el razonamiento contrario: si no existía un usuario logeado se renderizaban, y si no la aplicación se redirige a la página principal.

5.1.11. Contextos compartidos

En react cada componente tiene su propio contexto. Las funciones y variables que se declaran dentro de uno solo pueden ser accedidos por él mismo, a menos que un padre las pase como argumento a un hijo, en cuyo caso este sí podría acceder. Sin embargo es un procedimiento un poco rudimentario si se va a necesitar más que una variable puntual, mucho más si hay componentes anidados a varios niveles y el nodo del fondo necesita hacer uso de algún elemento del padre de todos.

Esto fue un problema cuando se trataba con la gestión del usuario activo y llamadas a la API. El usuario activo se almacena en un objeto de la librería de *Firebase* (se explicará en siguientes apartados). Este objeto debía instanciarse una vez y llamarse desde cualquier punto de la aplicación, lo cual era un problema por lo comentado sobre los contextos. Se tendría que pasar este objeto a cada elemento de la página que lo necesitase por parámetros, lo cual no era elegante.

Para solucionarlo se hizo uso de los *createContext* de React. Estos permiten definir unos componentes especiales cuyos descendientes tienen acceso a las variables y funciones declaradas en el padre. *userController* y *sceneController* se crearon para manejar respectivamente las llamadas de usuarios y escenas. Tienen como hijos al resto de los componentes de la aplicación, por lo que se pueden llamar desde cualquier sitio. Además, encapsulan las funciones de comunicación con el servidor, por lo que si en algún momento se fuera a cambiar el backend del sistema solo habría que re escribir estos archivos.

5.1.12. Animación de carga

Existían muchos momentos en la aplicación que suponían una espera para el usuario, como por ejemplo cuando se obtiene el listado de escena, cuando se carga un modelo o al guardar la escena en el servidor. Para hacer saber al usuario que se están haciendo operaciones y la aplicación está reaccionando a sus órdenes se añadió una animación de carga de la librería de *react-spinners* [10]. Se creó un componente encargado de renderizar esta animación en el centro de la pantalla. En lugar de incluir este componente en cada página que necesitara de un proceso de carga (lo cual sería tedioso), se creó un nuevo *context* de React que envolvía a toda la aplicación, haciendo que desde cualquier punto se pudiera manipular una variable booleana que activaba o desactivaba la animación de carga.

5.1.13. Mensajes de incidencia y avisos

Se introdujeron mensajes en las páginas de inicio de sesión, registro y cambio de datos para alertar al usuario el motivo del error a la hora de realizar una operación. Por ejemplo contraseña o correo incorrecto, clave poco segura, correo

inválido o ya registrado, etc. También se añadieron mensajes de confirmación en acciones irreversibles como al eliminar una escena (*¿seguro que desea eliminarla?*) o al abandonar el editor durante la creación de una escena (*¿seguro? Se eliminarán los cambios no guardados*).

5.2. Reproductor de escenas Android

La aplicación Android se compone de tres elementos básicos: una página de inicio de sesión, una página con el listado de las escenas creadas por el usuario activo, y el visor de Realidad Aumentada en función de la escena seleccionada. Se planteó también en un principio una página de registro de usuario pero se descartó debido a que si una persona accedía al sistema por primera vez desde esta app no tendría ninguna escena creada para reproducir. Por tanto el registro de usuarios se mantendría unicamente en el editor web. Antes de comenzar con la programación se realizaron diseños para las distintas páginas de la aplicación y la navegabilidad entre estas. Estas páginas reciben el nombre de *actividades* en Android Studio y se representan con una clase Kotlin y un archivo *XML* para la interfaz.

5.2.1. Archivo de configuración de escena

Lo primero que hubo que definir es la estructura de datos con la que se representarían las escenas. Este sería el formato usado tanto para almacenarlos en memoria como para almacenarlos en la base de datos. Debido a que la base de datos de *Firebase* funcionaba con objetos JSON se decidió que este fichero sería la forma de codificar las escenas. A continuación se tendría que decidir qué campos tendría que tener el archivo. Se llegó a la siguiente lista:

- **name** *(string)*: Nombre de la escena.
- **uid** *(string)*: Identificador del usuario creador de la escena.
- **scene_type** *(string)*: Tipo de escena. Campos posibles: *augmented_images*, *ground* o *geospatial*.
- **model_url** *(string)*: Enlace de descarga del modelo.
- **loop** *(boolean)*: De ser verdadero, las animaciones y audio se reproducirán en bucle.
- **audio** *(string)*: Enlace de descarga del archivo de audio. Si esstá vacío es que no hay audio asociado.
- **image_url** *(string)*: Enlace de descarga de la imagen marcadora. Solo está relleno si la escena es de imágenes aumentadad.

- **coordinates** *(float array)*: Coordenadas GPS en caso de que la escena sea geoespacial.
- **animations** *(string array)*: Nombre de las animaciones que se reproducen en la escena.

5.2.2. Escenas de imágenes aumentadas

Todas los tipos de escena se reproducen en la misma actividad android, *ArActivity.kt*. Cuando se inicia una escena se abre esta actividad y recibe como parámetro un JSON en forma de *string*. Con la librería *Gson* [**?**] se convierte en una *data class* llamada **sceneParameters**, una clase sencilla que almacena únicamente una lista de atributos. Así se podrán acceder comodamente a lo largo de la ejecución. Lo primero que hace la actividad es comprobar el parámetro *scene_type* para saber qué tipo de escena se va a ejecutar.

Para configurar una escena de imágenes aumentadas en *Sceneview* (la librería usada para manejar *ARCore*), se debe crear un objeto de la clase *ArSceneView*. A este objeto se le deberá indicar las distintas configuraciones en función del tipo de escena que se pretenda iniciar.

Como en este caso se iniciará una escena de imágenes aumentadas, se le pasa la función *initializeSceneViewSession* donde se crea una base de datos de imágenes que deberán ser reconocidas por la aplicación. En este caso solo se tiene una imagen la cual se puede descargar en forma de *Bitmap* con la url de *sceneParameters*. Al realizar esta configuración, automáticamente se iniciará la cámara y se mostrará un recuadro blanco para enmarcar la imagen activadora de la escena.

Ahora se debe definir la función *checkAugmentedImageUpdate* e introducirla en el objeto *ArSceneView*. Esta función se ejecuta en cada fotograma e indicará qué hacer en el caso de encontrar con la cámara una imagen de las definidas previamente. Se escribirán aquí las llamadas necesarias a la biblioteca para enmarcar el modelo de la escena en la imagen. Para saber cómo se siguieron los ejemplos de la documentación.

primero se debe definir qué imágenes dispararán la escena. En este caso es una única imagen la cual se puede descargar en forma de *Bitmap* con la url de *sceneParameters*.

Para cargar el modelo en la escena solo es necesario instanciar un objeto de clase *ArModelNode* con los siguientes parámetros:

- **glbFileLocation**: Enlace de descarga del modelo. Se puede obtener de *SceneParameters*. La clase se encargará de descargarlo automaticamente.
- **applyPoseRotation**: En caso de ser verdadero, aplica una rotación al

modelo de la escena para que siempre se muestre apoyado en la imagen independientemente de la inclinación de esta.

- **scaleToUnits**: Multiplicador de escala que se le aplica al modelo. En un proceso de prueba y error se determinó que, para que el tamaño relativo del modelo a la imagen fuera lo más equivalente posible al del editor web, este parámetro tendría un valor de *0.09*.

Este objeto se almacenará en un atributo de clase de la actividad y será el que se ancle a la posición de la imagen desde la función *checkAugmentedImageUpdate*.

5.2.3. Escenas por superficie

Para este tipo de escenas se debe activar el flag de *planeRenderer.isvisible*. Así se visualizará un patrón de puntos en los lugares que identifique la librería como suelos a través de la cámara. También hay que definir el *planeFindingMode*, que determinará las estrategias de búsqueda de superficies que ejecuta *SceneView* como por ejemplo buscar solo superficies horizontales, solo verticalez, fijar los puntos de anclaje a la mejor posición estimada, etc. Después de un proceso de prueba se determinó que el que mejores resultados obtenía era *HORIZONTAL*.

Se definieron dos botones en la interfaz, *Cargar objeto* y *Colocar objeto*. De primeras solo está el primer botón visible. Al pulsarlo, el modelo de la escena aparece en pantalla y comienza a colocarse a la superficie que esté apuntando el usuario. Si el usuario mueve la cámara, el objeto se desplaza acorde. Al pulsar el primer botón este desaparece y aparece el segundo. Cuando se pulsa este el objeto se queda fijo en la posición en la que se encontrara. Llegado a este punto el usuario puede mover el dispositivo como plazca, el objeto seguirá en el mismo punto.

5.2.4. Escenas geoespaciales

Para emplear escenas geoespaciales era necesario habilitar la API Geoespacial [7] de ARCore en la aplicación. Esta es un servicio de *Google Cloud* que provee entre otras cosas de funciones de geoposicionamiento para aplicaciones que usan ARCore. Para ello simplemente se siguieron las instrucciones indicadas en la página para habilitarlo y usarlo en una sesión de Realidad Aumentada.

Fue necesario habilitar una cuenta de *Google Cloud* para el proyecto. Este es un servicio de pago pero que ofrece una versión gratuita con $300 de presupuesto, suficientes para el desarrollo. Si se fuera a desplegar la aplicación a nivel comercial sería necesario introducir más presupuesto. El saldo se consume según el número de operaciones que se realicen con la API.

De vuelta al código de la aplicación con todo configurado, solo sería necesario crear un objeto de tipo *Anchor* con las coordenadas GPS almacenadas en *SceneParameters* y asignársela al nodo del modelo creado de forma similar a escenas explicadas anteriormente.

Un problema que surgió fue que el usuario no podía visualizar la escena a menos que estuviera muy cerca de esta. Para solventarlo se aumentó el atributo *sceneView.cameraNode.farClipPlane*, que indica la distancia de renderizado máxima. Hay que tener en cuenta que las aproximaciones GPS en dispositivos móviles siempre son aproximadas hasta cierto punto, así que es posible que si se reproduce la misma escena varias veces el objeto aparezca en lugares ligeramente distintos.

5.2.5. Reproducción de audio y animaciones

El audio se gestionó con la clase *MediaPlayer*. Esta se encarga tanto de descargar el archivo a través de la *URL* como de reproducirlo al indicárselo. Para las animaciones, la clase *ArModelNode* contaba con soporte para reproducirlas. Solo era necesario indicar el nombre de las animaciones deseadas, los cuales se sacan de *SceneParameters*.

El momento en el que se reproduce el audio y las animaciones. En imágenes aumentadas comienzan cuando se detecta la imagen activadora en *checkAugmentedImageUpdate*. En el caso de las escenas por superficie es cuando se pulsa el botón de *Colocar objeto*. Con las geospaciales simplemente comienzan cuando la escena se inicia.

5.2.6. Menú de selección de escena

Una vez se inicia sesión se muestra un menú *scrolleable* en el que aparecen todas las escenas creadas por el usuario mostrando su nombre, el tipo de escena y un botón para iniciarla. También hay un botón para cerrar la sesión. Para implementar la lista de escena se hizo uso de un *RecyclerView*. Este es un tipo de interfaz Android que permite la creación de menús con listas extensas de elementos repetidos. En este caso se repetiría la *"tarjeta"* que representa cada una de las escenas en la interfaz. La peculiaridad que tiene *RecyclerView* con otros elementos del estilo, es que solo crea los elementos que se encuentran en un instante determinado en la pantalla del dispositivo. Si el usuario se desplaza por el menú, creará los elementos nuevos que hayan entrado en pantalla y descartará los anteriores. Con esto se consigue que si un usuario ha creado un numero muy grande de escenas, el rendimiento de la aplicación no disminuya.

5.3. Backend

Para el backend se tienen dos elementos. Por un lado el servicio de *Firebase*, hosteado por Google. Por otro lado se hizo un servidor web que hace de intermediario con ambas aplicaciones y *Firebase*. Se van a detallar en los siguientes apartados las configuraciones y desarrollo que se realizó además de justificar las decisiones de diseño.

5.3.1. Configuración de la base de datos

Lo primero que se realizó fue crear un proyecto en *Firebase*, el *backend-as-a-service* que se utilizaría para la base de datos y autenticación de usuarios. Para ello se siguieron las instrucciones ofrecidas en la propia página. Se generaría una *key* o clave que necesitaría cargarse en cualquier programa que hiciera uso directo de los servicios de *Firebase*. El backend está hosteado *24/7* por Google. Estos servicios tienen opciones de pago profesionales y una gratuita, que para propósitos del desarrollo sería suficiente. Esta simplemente limita la capacidad máxima para almacenar datos en la nube, el número de operaciones de lectura y escritura por día y funcionalidades varias de *Google Cloud*.

Se procedió a configurar la base de datos. *Firebase* utiliza una base de datos no relacional, que a diferencia de las convencionables no emplean tablas que necesiten definirse previamente. En su lugar aquí se tienen **colecciones**. Estas son conjuntos de **documentos**, los cuales contienen *campos*. Los documentos tienen en esencia la misma forma que un archivo JSON, y de hecho es el formato que se emplea para el intercambio de información cuando se realizan peticiones a la base de datos. Se creó una colección llamada *escenas* que contendría distintos documentos, cada uno de ellos almacenaría la información sobre una escena, con la misma estructura que se describió en la sección de la aplicación Android. En un inicio se creó otra colección para almacenar información de los usuarios, pero se descartó en favor del uso de *Authentication*. Los documentos una vez creados son asignados automáticamente con un código identificador.

5.3.2. Usuarios del sistema

Uno de los servicios que ofrece *Firebase* es **Authentication**. Este gestiona su propia base de datos de usuario. Los usuarios poseen un correo electrónico que usan para registrarse en el sistema, mantiene todas las contraseñas seguras bajo encriptación y genera automáticamenet *ids* de usuario que los identifica inequivocamente. Acciones como registrarse como usuario, iniciar sesión, cambiar datos o gestionar los tokens autentificadores se realiza de forma automática y trasnparente al programador a través de simples llamadas a la librería de *Firebase*. Es por eso que se decidió optar por esta forma de mantener la información de

los usuarios en lugar de crear una colección en la base de datos.

5.3.3. Almacenamiento de archivos binarios

Otro de los servicios de *Firebase* es **Storage**. Esta interfaz permite almacenar archivos binarios de tamaños elevados. Posteriormente se puede obtener una *URL* a través de la cual descargar el fichero. Estos archivos pueden estar además almacenados en distintas carpetas. Para el proyecto se crearon las carpetas *models*, *audio* e *images*. Como cada escena tendría como mínimo un archivo de modelo, y como mucho otro de imagen y de audio, el nombre de los ficheros almacenados sería la *id* del documento de la escena. Así podrían rescatarse facilmente al obtener el JSON en la aplicación.

5.3.4. Servidor Node.js

Al igual que con la aplicación web, se usó *Vite* para crear un proyecto base de JavaScript con *Node.js* [5] y *Express* [3] para implementar la API a la que pedirían servicio las aplicaciones web y de Android. En el archivo *index.js* se definen todas las llamadas que ofrece la interfaz. Cada llamada tiene una *URL* asociada. Por ejemplo si se quiere subir una nueva escena, se haría desde la aplicación una petición HTTP POST a la dirección en la que esté alojado el servidor seguido de */post/escena*. Algunas llamadas tienen implícito en la dirección un parámetro, señalizado con *":"* en la *URL*. En el caso de */get/escenas/:id*, al hacer la petición se introduce en el *:id* el identificador de usuario del que se quieran obtener las escenas.

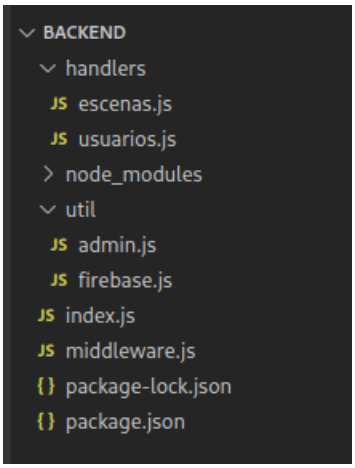

Figura 5.7: Estructura de archivos del proyecto de Node.js para el backend

En la carpeta util se almacenan scripts que cargan objetos *admin* y *db* de *Firebase* a través de los cuales se realizarán las llamadas a la base de datos.

En la carpeta *handlers* se encuentran las funciones correspondientes a las llamadas ya descritas. Aquí es donde se hace la conexión con *Firebase* y se realizan operaciones adicionales. Cada petición tiene dos argumentos:

- **req**, o request, es el objeto con la información de la petición. Aquí se almacenan los parámetros de la *URL* o el *body*, donde viene incluido archivos adjuntos si fuera el caso.
- **res**, o response, es la respuesta que se envía a la máquina que realizó la petición. Si se realizó una llamada para obtener una lista de escenas, esa información se añadirá en este objeto.

5.3.5. Esquema y verificación de peticiones

Se cuenta con tres agentes en la red: *App*, hace referencia a la aplicación web o Android, el *servidor Node.js* y *Firebase*. Como se ha visto la aplicación realiza peticiones al servidor Node.js para obtener escenas. Sin embargo, en el caso de identificarse como usuario la aplicación conecta directamente con Firebase. Esto podría parecer anómalo pero tiene una razón detrás. Todo el tema de gestión de sesión de usuario viene resuelto ya en las librerías de Firebase para Android y JavaScript. Se tiene un objeto *Authenticator* en memoria, y a través de este se puede crear un nuevo usuario, iniciar sesión, cambiar credenciales, comprobar si la sesión activa es válida, etc y todo ello de forma segura. Si se quisiera realizar todos esos cálculos desde el servidor Node.js añadiría muchos pasos extra y probablemente quedaría un resultado más pobre, ya que para este proyecto no se cuenta con la experiencia que puede tener un equipo de ingenieros de Google.

Pero si para el inicio de sesión se prescinde del servidor intermediario, ¿por qué no para todo lo demás? Por dos razones. La primera es que eso supondría programar todas las operaciones que realiza el servidor en una clase para React y Android Studio, lo cual sería redundante: repetiríamos código en dos proyectos distintos. Si en un futuro se quisiera actualizar alguna de estas funciones, tendría que hacerse por partida doble. Por otro lado teniendo el servidor intermedio obtenemos mayor modularidad. Al final su función es únicamente gestionar las peticiones que se realizan a la base de datos de escenas. Si en un futuro se decidiera sustituir esa base de datos por una opción más conveniente, pero manteniendo el *Authenticator* de Firebase para la parte de usuarios, sería posible y facilmente implementable con la estructura que se propone.

Para asegurar que el usuario que realiza las llamadas es realmente de quien dice ser, se hace uso de *JWT* (*JSON Web Tokens*). Es un estandar con el que se puede propagar entre dos partes y de forma segura la identidad de un usuario. En esencia es una cadena de texto codificada que puede enviarse junto a las peticiones para firmarlas verificando la identidad. El servicio *Authentication* de Firebase genera un *JWT* para cada usuario identificado que se puede obtener desde el

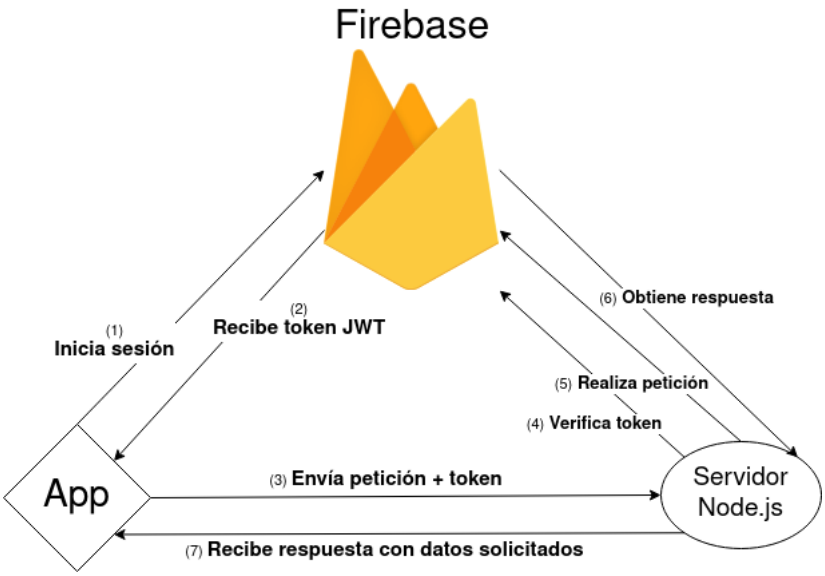

Figura 5.8: Diagrama de comunicaciones del servidor

cliente. En la implementación realizada, las aplicaciones envían al servidor por cada petición el token. Este es recibido por el servidor, y antes de atender la petición recibida, comprueba con *Firebase* que el token es válido. Si es el caso, se atiende a la petición y se envía la respuesta. Los tokens tienen validez durante una hora. Una vez transcurrido ese tiempo, *Firebase* genera uno nuevo el cual vuelve a ser obtenible desde la aplicación.

En el proyecto de Node.js se realiza esta comprobación en la función *decodeToken* dentro del archivo *middleware.js*. Esta comprobación ocurre a cada petición que llega. Si es exitosa comienza a ejecutarse el código que atiende la petición. Si no lo es, devuelve un mensaje de error y termina la comunicación.

Capítulo 6

Conclusiones y trabajos futuros

Bibliografía

[1] Pictarize studio. https://pictarize.com/.

[2] Carmen Lasa Gómez Alonso Álvarez García, Rafael de las Heras del Dedo. Métodos Ágiles y SCRUM, manual imprescindible. .

[3] Express.js. Fast, unopinionated, minimalist web framework for node.js. http://expressjs.com/.

[4] Free Software Foundation. GNU General Public License. http://www.gnu.org/licenses/gpl.html.

[5] Node.js Foundation. Node.js. https://nodejs.org/es/.

[6] Google. Arcore. https://developers.google.com/ar?hl=es-419.

[7] Google. Arcore api, cloud services for arcore. https://console.cloud.google.com/apis/library/arcore?hl=es-419.

[8] Google. Firebase. https://firebase.google.com/?hl=es-419.

[9] Thomas Gorisse. Sceneview, 3d and ar view for android, flutter and react native working with arcore and google filament. https://sceneview.github.io/.

[10] David Hu. React spinners. https://www.davidhu.io/react-spinners/.

[11] Javier González Jiménez. Visión por computador. .

[12] Scrum Manager. Troncal i. scrum master. http://scrummanager.net/files/scrum_manager.pdf.