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 Pokémon 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 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

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.

1.1. 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

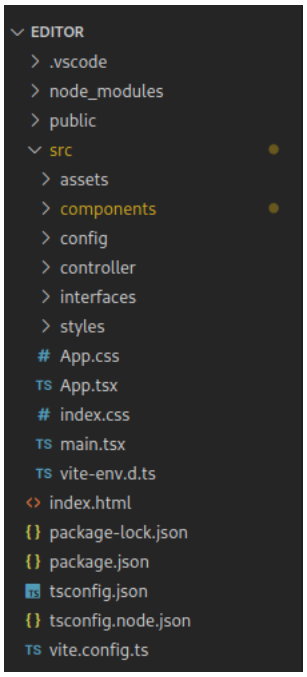

Figura 1.1: Estructura de directorios final de la aplicación web de React.

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.

1.1.1. Renderización de entorno 3D

Lo primero que se realizó antes de el componente en sí con toda la interfaz, fue implementar el entorno 3D interactuable. Para ello se hizo la clase **EditorScene.ts**, que hereda de la clase *Scene* de la librería de Three.js. *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 *PerspectiveCamera* 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 se va a ver por pantalla. Esta información se transmite al renderizador, que se encarga

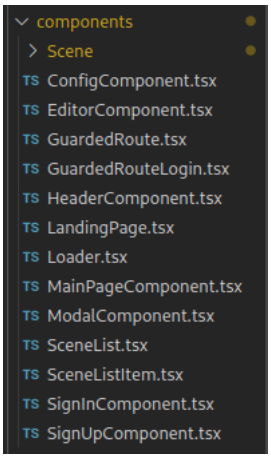

Figura 1.2: Lista de componentes del proyecto de la apliación web.

de aplicar cálculos para convertir el espacio 3D de una representación abstracta a una imagen en dos dimensiones que se puede mostrar 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.

1.1.2. 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 clic derecho del ratón se rota la cámara sobre un punto, y arrastrando con la rueda del ratón pulsada de desplaza. Así queda el clic izquierdo libre para las interacciones con los objetos.

El usuario debe poder seleccionar un objeto de la escena colocando el ratón sobre y haciendo clic. 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 clic. De esta forma el rayo ensarta el punto en el que se ha pulsado, 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 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 automáticamente 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.

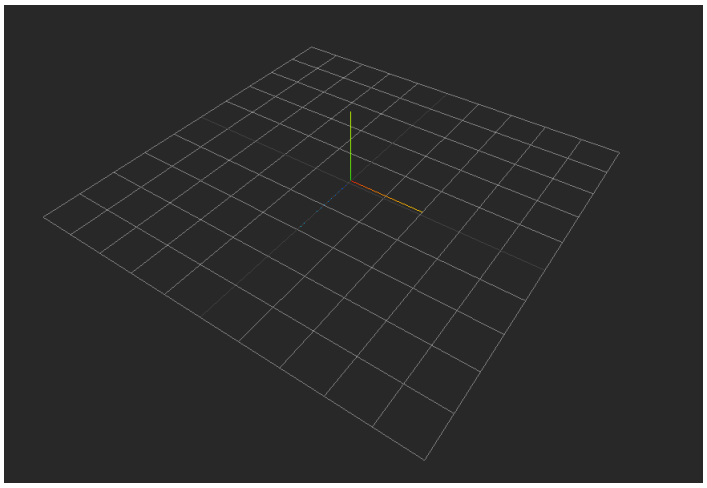

Figura 1.3: Grid del editor de escenas para tener de referencia como suelo.

Sin embargo, esto ocasiona 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 clic 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 clic 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 clic en otro objeto mientras se pulsa la tecla *Control*. Este es un esquema de control usado tanto en software del sector como en los gestores 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 "engancharse" 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 clic directamente sobre él para manipular el objeto.

Aquí apareció otro problema. La clase solo soporta un objeto .^{en}ganchado.^{a}l 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 coordenadas 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.

1.1.3. 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 automáticamente 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 cuál es la seleccionada. Para ello se empleó el 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.

1.1.4. 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. Después 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 utilizarse 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 manera 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.

1.1.4.1. 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.

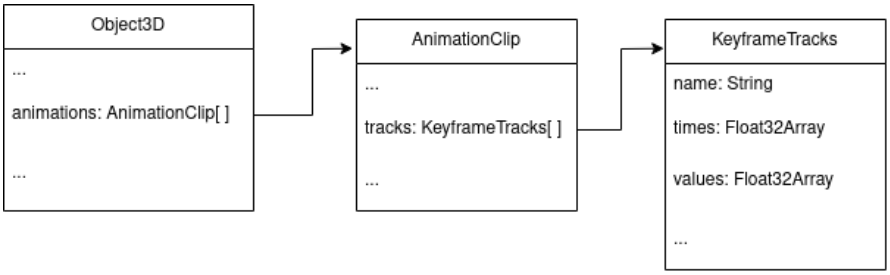

Figura 1.4: Esquema de las estructuras de datos de *Object3D*, *AnimationClip* y *KeyframeTrack* para la reproducción de animaciones.

GLTFExporter al llevar a cabo 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 asignárselas 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 ve 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 empleada para almacenarse en la base de datos como un único archivo.

1.1.5. 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*.

1.1.6. Interfaz del componente editor

Ahora se discutirá el **EditorComponent.tsx** en sí, 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 diseñó 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 lanzadas por Google para interfaces web y móviles. Es lo que se acabaría empleando 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

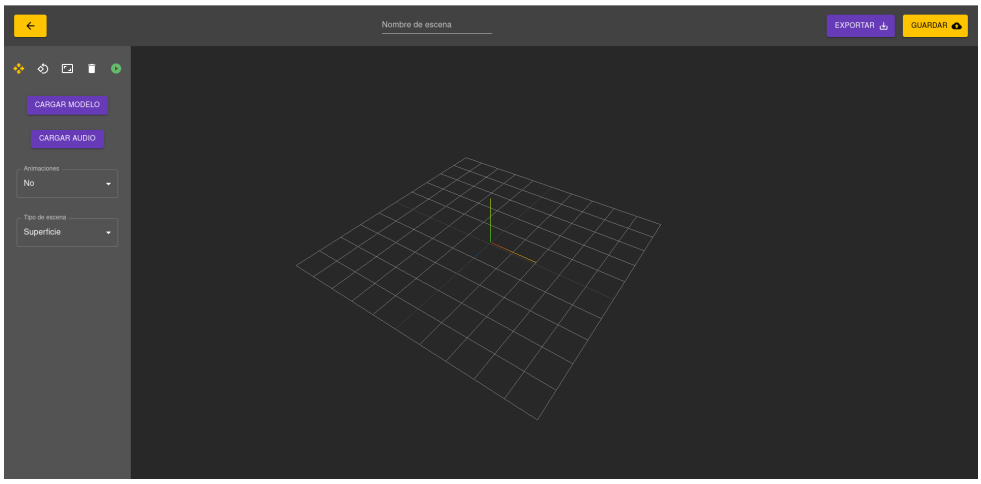

Figura 1.5: Interfaz de usuario final para el editor de escenas.

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 emplean 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 clic, este tendría que tener el número de pulsaciones 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 utilizan (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 utilizar 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 clicados 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 arrastrando 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.

1.1.7. 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.

1.1.8. 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.

1.1.9. 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 hizo 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 conecta-

dos. Se muestra el listado de escenas para poder editarlas o eliminarlas además de opciones para iniciar una nueva 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.

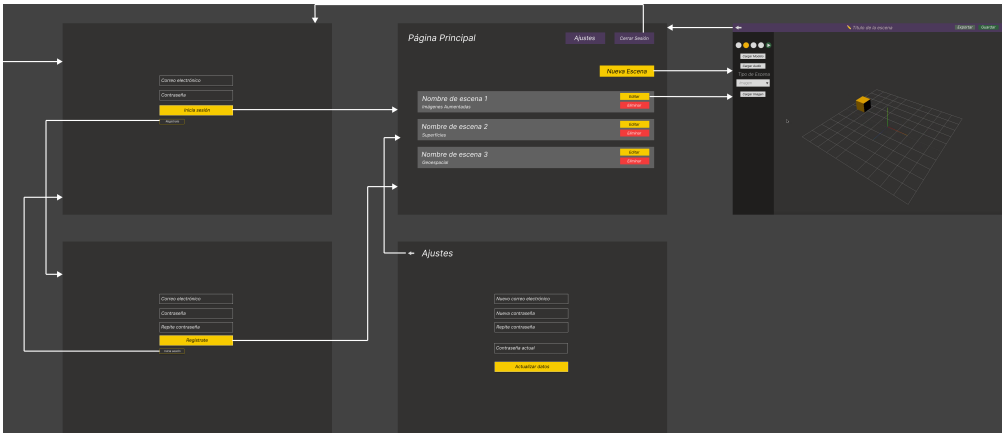

Figura 1.6: Mockup de la UI de la aplicación web realizado en *Figma*

1.1.10. Rutas protegidas

Al ser accesible cualquier página de la aplicación surgía un nuevo problema. En el planteamiento propuesto, 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 comprueba 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.

1.1.11. 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 emplearon 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 reescribir estos archivos.

1.1.12. 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* [4]. 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 introdujo 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.

1.1.13. 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*).

1.2. 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 para reproducir. Por tanto, el registro de usuarios se mantendría únicamente 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.

1.2.1. 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 está 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 aumentada.
- **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.

1.2.2. Escenas de imágenes aumentadas

Todos 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 cómodamente 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 cargar.

Como en este caso se iniciará una escena de imágenes aumentadas, se le pasa la función *initializeSceneViewSession* donde se genera 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 activará 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.

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 automáticamente.
- **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*.

1.2.3. 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 verticales, 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 únicamente 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.

1.2.4. Escenas geoespaciales

Para emplear escenas geoespaciales era necesario habilitar la API Geoespacial [3] 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 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.

1.2.5. 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.

1.2.6. 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 escenas 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 genera los elementos que se encuentran en un instante determinado en la pantalla del dispositivo. Si el usuario se desplaza por el menú, cargará los elementos nuevos que hayan entrado en pantalla y descartará los anteriores. Con esto se consigue que si un usuario tiene un numero muy grande de escenas, el rendimiento de la aplicación no disminuya.

1.3. 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.

1.3.1. Configuración de la base de datos

Lo primero 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 convencionales 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 planteó otra colección para almacenar información de los usuarios, pero se descartó en favor del uso de *Authentication*. Los documentos son asignados automáticamente con un código identificador.

1.3.2. 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 automáticamente *ids* de usuario que los identifica inequívocamente. Acciones como registrarse como usuario, iniciar sesión, cambiar datos o gestionar los tokens autentificadores se realiza de forma automática y transparente al programador a través de simples llamadas a la librería de *Firebase*. Es por eso que se decidió optar usar *Authentication* de mantener la información

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

1.3.3. 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 fácilmente al obtener el JSON en la aplicación.

1.3.4. Servidor Node.js

Al igual que con la aplicación web, se usó *Vite* para iniciar un proyecto base de JavaScript con *Node.js* [2] y *Express* [1] 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.

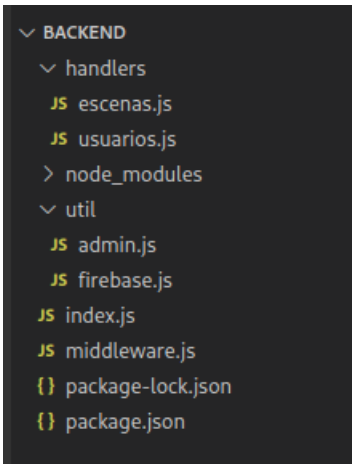

Figura 1.7: Estructura de archivos del proyecto de Node.js para el backend

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.

1.3.5. 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 de las que se ocupa 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 hacen a la base de datos de escenas. Si en un futuro se decidiera sustituir esa base de datos por una opción más conveniente y queriendo mantener el *Authentication* de Firebase para la parte de usuarios, sería posible y fácilmente 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 estándar 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

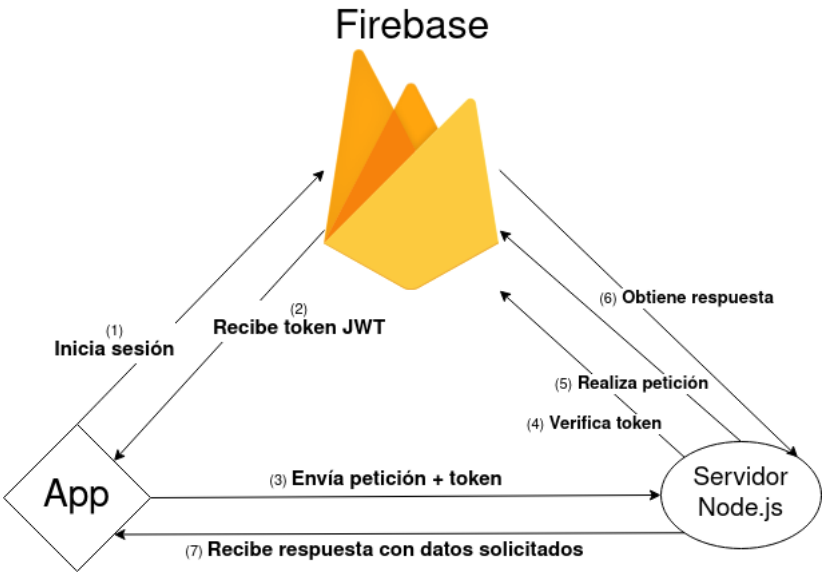

Figura 1.8: Diagrama de comunicaciones del servidor

desde el cliente. En la implementación llevada a cabo, 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 verificación 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.

Bibliografía

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

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

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

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