ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Trabajo Fin de Grado

Editor de Escenas Basado en A-Frame

Autor : Miguel Hidalgo Pérez

Tutor : Dr. Jesús M. González Barahona

Curso Académico 2020/2021

Trabajo Fin de Grado

Editor de Escenas Basado en A-Frame

Autor : Miguel Hidalgo Pérez

Tutor : Dr. Jesús M. González Barahona

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 2021, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2021

Dedicado a

mi familia

Agradecimientos

Dedicado a mi familia, especialmente a mi padre que me ha ayudado y permitido llegar hasta aquí y espero poder devolverle una parte de todo lo que me ha dado. También a mis amigos que siempre me han recordado que a pesar de los problemas se pueden cumplir los objetivos que te propongas.

Resumen

Este proyecto consiste en el desarrollo de un editor de escenas 3D inmerso en realidad virtual ejecutado en un navegador web, por lo que será compatible con todos los dispositivos que dispongan de uno como gafas de realidad virtual, dispositivos móviles o de sobremesa.

El objetivo es construir una versión inspirada en los editores tradicionales de escenas 3D y tener una base para seguir implementando más funcionalidad.

Es interesante debido a que el usuario en este caso forma parte de la propia escena y actualmente en el mercado no parece haber muchos editores inmersivos. Por otro lado la realidad virtual es campo en continuo crecimiento con muchas posibilidades que explorar.

Recordamos que las escenas 3D son usadas por múltiples sectores y disciplinas como pueda ser la simulación en ingenierías, arquitectura, educación, etc... donde la realidad virtual puede a introducirse para crear experiencias más atractivas o realistas.

Se ha utilizado un *framework* web llamado A-Frame, que permite construir aplicaciones de realidad virtual en soporte Web a partir de su API mediante componentes HTML, donde podemos encontrar entidades con las que enriquecer la escena con distintos elementos 3D y comportamientos.

A-Frame es una tecnología que refleja nodos HTML del DOM en un Canvas 3D y para ello se apoya en una librería de software libre con bastante comunidad llamada ThreeJS mas a bajo nivel usando como motor gráfico WebGL. WebGL es una implementación para navegadores de OpenGL, API de software libre que hace uso del hardware gráfico para pintar figuras 3D.

El desarrollo del proyecto ha sido implementado en Javascript y HTML5 haciendo operaciones con el DOM en tiempo de ejecución para manipular sus atributos o insertar nuevos elementos que interpretará A-Frame.

Los elementos del editor están modularizados y apificados para poder configurarlo y extenderlo.

Finalmente se ha publicado el proyecto para la comunidad y así los desarrolladores que quieran puedan apoyarse en el y utilizarlo como deseen.

Summary

This project is about development of a 3D scene editor immersed in virtual reality running in a web browser, it will be compatible with all devices that have one such as virtual reality glasses, mobile or desktop devices.

The goal is to build a version inspired by traditional 3D scene editors and to have a base to continue implementing more functionality.

It is interesting because the user in this case is part of the scene itself and currently there do not seem to be many immersive editors on the market. On the other hand, virtual reality is a growing field with many possibilities to explore.

We remember that 3D scenes are used by multiple sectors and disciplines such as simulation in engineering, architecture, education, etc ... where virtual reality can be introduced to create more attractive or realistic experiences.

We have used a web framework called A-Frame, which allows to build virtual reality applications in Web support from its API through HTML components, where we can find entities with which we can enrich the scene with different 3D elements and behaviors.

A-Frame is a technology that reflects HTML nodes of the DOM in a 3D Canvas and for this it relies on a free software library with enough community called ThreeJS but at a low level using WebGL as a graphics engine. WebGL is a browser implementation of OpenGL, a free software API that makes use of graphics hardware to paint 3D figures.

The development of the project has been implemented in Javascript and HTML5 doing operations with the DOM at runtime to manipulate its attributes or insert new elements that A-Frame will interpret.

Application elements are modularized and modified to be able to configure and extend it.

Finally, the project has been published for the community so that developers can use it as they wish.

Acrónimos

HTML: Hyper Text Markup Language

HTTP: Hyper Text Transfer Protocol

JSON: JavaScript Object Notation

JS: JavaScript

DOM: Document Object Model

API: Application Programming Interface

VR: Virtual Reality

Índice general

1. Introducción **1**

1.1. Contexto . . . 2

1.2. Motivación . . . 3

1.3. Objetivo general . . . 4

1.4. Objetivos específicos . . . 4

1.5. Estructura de la memoria . . . 5

1.6. Disponibilidad del Software . . . 5

2. Tecnologías Utilizadas **7**

2.1. HTML5 . . . 7

2.2. JavaScript . . . 9

2.3. Typescript . . . 10

2.4. JSON . . . 11

2.5. NodeJS . . . 12

2.6. npm . . . 13

2.7. Webpack . . . 13

2.8. OpenGL . . . 14

2.9. Realidad Virtual . . . 15

2.10. WebXR . . . 16

2.11. Three.js . . . 17

2.12. A-Frame . . . 21

2.13. Git . . . 23

2.14. Tecnologías Similares . . . 24

3. Diseño e implementación **29**

3.1. Metodología SCRUM 29

3.2. Sprint 0. Estudio previo. 30

3.3. Sprint 1. Mover y arrastrar figuras 35

3.4. Sprint 2. Replicar figuras 45

3.5. Sprint 3. Editando figuras 48

3.6. Sprint 4. Multiselección 54

3.7. Sprint 5. Extras 57

4. Resultados **63**

4.1. Demo 64

4.2. Manual de usuario 66

4.2.1. Ejemplo Escena Cambiada 71

4.3. Estructura 72

4.4. Manual técnico 75

4.4.1. Componentes Visuales 75

4.4.2. Componentes de comportamiento 76

4.4.3. Helpers 77

4.4.4. Modelos 80

4.4.5. Servicios 82

5. Conclusiones **85**

5.1. Consecución de objetivos 85

5.2. Aplicando Conocimientos 86

5.3. Lecciones Aprendidas 87

5.4. Trabajos futuros 88

Índice de figuras

2.1. De Typescript a Javascript 11
2.2. Ejemplo JSON básico 11
2.3. Arquitectura NodeJs 12
2.4. Qué es Webpack 14
2.5. Dispositivo de Realidad Virtual 16
2.6. Realidad Aumentada 16
2.7. Escena Three.js 18
2.8. Ejemplo de cámara. Forma geométrica ”*frustum*” 19
2.9. Ejemplo de escena con ”*threejs*” 20
2.10. Ejemplo código HTML de A-Frame 21
2.11. Ejemplo del inspector de A-Frame 22
2.12. Blender ejemplo 24
2.13. 3DS Max ejemplo 25
2.14. Unity ejemplo 26

3.1. Primera escena con A-Frame (resultado) 31
3.2. Antes de accionar la palanca 34
3.3. Después de accionar la palanca 34
3.4. Primera arquitectura del proyecto 36
3.5. Resultado Sprint 1 45
3.6. Resultado Sprint 2 47
3.7. Resultado Sprint 3 51
3.8. Selección figuras 54
3.9. Operación grupal 54

3.10. Edición luz baja 58
3.11. Edición luz alta 58
3.12. Inicio Timelapse 59
3.13. Final Timelapse 59
3.14. Cambio de escena 60
3.15. Gravedad 61

4.1. Escena inicial demo 64
4.2. Figura insertada 65
4.3. Figura con menú 65
4.4. Figura con menú 72
4.5. Estructura final del proyecto 74

Capítulo 1

Introducción

El proyecto consiste en el desarrollo de un editor de escenas en 3D inmersivo en Realidad Virtual mediante tecnologías soportadas por navegadores web. Para ello se ha recurrido a un *framework* web llamado A-Frame[1] que nos permite generar modelos 3D a partir de componentes HTML.

Esta tecnología se apoya en Three.js[2] y WebGL[3] para la generación de gráficos. Gracias a A-Frame nos abstraemos de trabajar con la generación de modelos 3D a bajo nivel lo cual es relativamente complejo y podremos centrarnos en otros retos como la lógica de la aplicación a mas alto nivel.

También nos ayuda en la adaptación de las aplicaciones que construyamos para adaptarlas a distintos dispositivos e interactuar con sus periféricos, lo cual hace que sea multiplataforma pudiendo correr en distintos dispositivos como gafas vr, móviles o escritorio.

El objetivo de este editor es poder construir una escena donde se pondrán introducir distintas figuras con propiedades editables que podremos desplazar a lo largo del espacio y ver su interacción con la luz y físicas.

El campo de la realidad virtual todavía se encuentra en constante crecimiento y desarrollo por parte de empresas y comunidades que apuestan por ello. Aún es un terreno por explorar con muchas posibilidades y casos de uso, añadiendo además que en web es un terreno más virgen incluso en el que no hay tantas tecnologías estandarizadas. Todo esto hace atractivo enfrentarse

[1] `https://aframe.io/`
[2] `https://threejs.org/`
[3] `https://www.opengl.org/`

al reto e investigar las posibilidades.

Vamos a proceder a contextualizar más el proyecto, motivación, objetivos e implementación.

1.1. Contexto

La web y los navegadores son de las tecnologías que mas han evolucionado a lo largo de los años. La web fue concebida como protocolo de texto para presentar información en linea. Fue derivando en foros y paginas con hipertexto que se convertirían en un vehículo comercial mas allá de una gran fuente de información.

Con el tiempo las necesidades siguieron creciendo y fue necesario tener lenguajes de programación para crear contenido dinámico naciendo así el desarrollo web, mercado que a día de hoy es enorme, pues es multiplataforma y muy sencillo de distribuir, aplicable prácticamente a cualquier sector o negocio que se nos ocurra.

En sus inicios el contenido se generaba en el lado del servidor mediante tecnologías tan conocidas como PHP, ASP o JSP. Más adelante al mejorar el hardware de las máquinas clientes donde corrían los navegadores, los desarrolladores se platearon generar todo el contenido en el lado del cliente.

Con esta estrategia aumentaban las posibilidades, se podía generar contenido dinámico sin esperar que el servidor mandara toda la página entera para un pequeño cambio y navegar. Por otro se podía repartir la carga y ser mas eficiente pues los servidores podían ocuparse de otro tipo de tareas como cálculo o transacciones con bases de datos y centrarse en atender peticiones y operaciones de los usuarios.

Para ello el conocido lenguaje Javascript inicialmente concebido para animaciones y cálculos sencillos, transmutó en una herramienta para construir y generar todo el contenido en el cliente, acercándose a los lenguajes de backend. También se convirtió en la API del navegador para explotar las tecnologías del estándar de HTML5 como sockets, vídeo, animaciones, motores gráficos y contenido en tiempo real.

Respecto al pintado de gráficos 3D, el primer escalón fue el canvas3D, WebGL apareció posteriormente por parte de Mozilla Foundation en versiones más modernas de navegadores aprovechando el hardware gráfico de la máquina. Es a partir de aquí donde entran las comunidades aportando librerías como Three.js que nos abstraerán nos ayudaran en un nivel mas alto

en la definición de elementos que se hacía a bajo nivel como polígonos, luces, animaciones, cámaras, etc... para centrarnos preocuparnos de capas superiores.

Existen otras tecnologías de desarrollo como Unity o Unreal que contienen su propio editor y motor gráfico entre muchas herramientas, los cuales tradicionalmente han estado orientados a construir aplicaciones de escritorio con lenguajes de backend, aunque a día de hoy contemplan el desarrollo multiplataforma contando con múltiples tecnologías y lenguajes.

1.2. Motivación

La llegada de los gráficos 3D a los navegadores ha supuesto un punto de inflexión en el desarrollo de algunos sectores como pueden ser los videojuegos, explotación de datos, simulaciones o demos inmersivas para educación por ejemplo.

Recordando que la distribución y disponibilización de las aplicaciones web es mucho mas sencilla que instalar una aplicación en un SO.

La motivación de este proyecto era construir un editor de escenas 3D que tratara de emular a editores clásicos de escritorio como pueda ser 3D studio, Autocad o Unity (entre otras muchas de las funciones y herramientas que incluyen), pero a través de un sistema inmersivo en VR a través de periféricos como las gafas y los mandos donde parece que formemos parte de la escena y no la veamos en tercera persona.

Respecto a mi propia motivación personal, me resulta muy atractivo el desarrollo de frontend puesto que ofrece un resultado vistoso y relativamente inmediato. Los problemas a los que te enfrentas y las capas de ingeniería actuales que no son pocas, son diferentes del backend pero son retos igualmente complejos y estimulantes.

La representación en 3D por otro lado era un valor añadido, es muy vistoso y un campo por explorar en navegadores. Siempre me he criado con videojuegos y tecnología alrededor por lo que ha sido todavía mas entretenido.

Por todo esto me embarqué en este proyecto y tratar de aterrizarlo en algo enseñable y utilizable.

1.3. Objetivo general

Este proyecto consiste en crear un editor de escenas en 3D para navegadores web que puede ser explotado en dispositivos de VR. Las escenas que se construyan con este editor podrían ser potencialmente explotables y útiles para otras aplicaciones.

1.4. Objetivos específicos

En este apartado se realiza una breve descripción de los diferentes objetivos.

1. La aplicación será desarrollada con A-Frame y three.js según se ha acordado con el tutor.

2. Esta aplicación puede ser ejecutada en cualquier navegador sin necesidad de instalar nada. Esta demo en concreto es una versión escritorio.

3. La escena es editable y exportable.

4. El rendimiento es un elemento a tener en cuenta, la escena debería soportar una volumetría razonable de elementos.

5. La escena generada debe poder ser usable en otros entornos con A-Frame.

6. La aplicación se debe distribuir y poner al servicio de la comunidad. Se debe dar fácil acceso y ser subida a un repositorio de código donde se puedan recibir sugerencias para implementar y resolver posibles problemas.

7. Debe ser fácilmente escalable, es decir, deberá estar desarrollada de tal manera que sea sencillo implementar nuevos desarrollos o modificaciones en el código [6].

8. Un objetivo importante es que el código deberá ser mantenible tanto por su autor como por la comunidad. Se debe publicar dicho software en una plataforma donde, como se menciona anteriormente, se pueda contribuir a resolver incidencias y mantener actualizado de manera sencilla las versiones de los framework y librerías que utilizará el proyecto.

9. Se creará una web con una demo y ejemplos de uso de la aplicación.

1.5. Estructura de la memoria

En esta sección se describe la estructura de la memoria para una mejor compresión de la misma:

- En el primer capítulo se hará introducción al proyecto, donde contextualizaremos para facilitar el seguimiento y la comprensión. Después, hablaremos de la motivación del proyecto a nivel personal y profesional. A continuación describiremos el objetivo principal que perseguimos con este trabajo para continuar con los más específicos en los cuales profundizaremos. Por último describiremos la estructura de la memoria.
- En el segundo capítulo hablaremos de tecnologías utilizadas, donde haremos un repaso así como una breve descripción de todas las tecnologías que se han utilizado.
- En el tercer capítulo desarrollaremos el diseño e implementación. Se entrará al detalle de como está construida la aplicación, su arquitectura y sus distintos componentes.
- En el cuarto capítulo hablaremos de los resultados obtenidos, donde realizaremos un análisis del funcionamiento de esta aplicación. Además se podrá ver un amplio abanico de casos de uso donde se verán los resultados y capacidades del proyecto.
- Por último, tendremos las conclusiones, donde haremos un resumen de los distintos conceptos aprendidos y aplicados. Podremos ver si se han alcanzado los objetivos propuestos y expondremos líneas futuras de investigación y mejora de esta aplicación.

1.6. Disponibilidad del Software

Para poder consultar y mantener el proyecto se ha creado un repositorio de software libre en el portal de GitHub. Este proyecto de software libre cuenta con una licencia de Apache 2.0. Todos los aportes a lo largo del tiempo así como las incidencias se pueden consultar en:

`https://github.com/hpmiguel/aframe-editor-scene`

Además esta librería cuenta con una página web donde podemos ver las *demos* y casos de uso. También se tiene un enlace directo al código fuente para consultar los componentes y la

escena. Se puede acceder a través del siguiente enlace:

`https://hpmiguel.github.io/profile-web-site/`

Capítulo 2

Tecnologías Utilizadas

En este capítulo se describirán las tecnologías que se utilizan en este trabajo. Se hará un repaso tanto de los lenguajes básicos de programación web utilizados, librerías y las herramientas de empaquetado y distribución de la aplicación.

También se mencionarán las librerías de realidad virtual y los motores gráficos en los que estas se apoyan, los cuales facilitan enormemente el trabajo de desarrollo.

Empezaremos por las librerías usadas en el proyecto y más tarde repasaremos tecnologías similares en el mercado.

2.1. HTML5

En 1993 se lanza el lenguaje de marcado HTML base para definir las web. Consiste en una estructura básica de etiquetas que son interpretadas por los navegadores para generar contenido. El encargado del mantenimiento y estandarización es el Consorcio de la *World Wide Web* (W3C).

Mas adelante, en 2014, se publica HTML5. Esta nueva versión supuso un punto de inflexión en las capacidades de los navegadores que comentaremos a continuación.

Por un lado podemos hablar de los cambios estructurales a nivel del documento que incluyen

nuevas etiquetas para definir secciones concretas de las como la cabecera <header>, el pie <footer> y la sección <section> que ayudarán tanto en maquetación como semántica de los campos.

A nivel de estilo va de la mano con el nuevo estándar CSS3 que ofrece una nueva gran variedad de opciones para hacer maquetación y diseños mas ambiciosos con técnicas como degradados, sombras o animaciones.

En cuanto a términos de semántica cabe mencionar brevemente el proyecto de la web semántica [1] en el que influyó HTML5. La web semántica fue un estándar creado para la definición y estructuración de conocimiento como ontologías. Para ello se definen tripletas en formato RDF/OWL y se realizan peticiones de consulta mediante SPARQL. Un ejemplo de la web semántica es la dbpedia [2] donde podremos hacer búsquedas estructuradas con esta tecnología.

Por otro lado aparecen nuevos elementos que van ligados a las capacidades tecnológicas del navegador que realmente supuso el gran cambio en el desarrollo web de los cuales mencionaremos algunos de los mas interesantes.

Aparece el Canvas que sirve para pintar elementos en dos y tres dimensiones en el cual se apoyará nuestro proyecto.

El tag de vídeo y audio que permiten que el navegador de forma nativa consuma elementos multimedia sin la necesidad de tecnologías satélite utilizadas hasta el momento como ha podido ser el conocido y difunto Flash.

También aparece SVG para gráficos vectoriales donde se interpreta un lenguaje vectorial para generar imágenes en la resolución que disponga el dispositivo.

Aparecen nuevas implementaciones de comunicaciones que permiten desarrollar aplicaciones en tiempo real como las vídeo conferencias a través de tecnologías como WebRTC o WebSockets inspirados en protocolos de backend.

La gestión de hilos de ejecución con WebWorkers o las caches de peticiones con WebServices y la posibilidad de trabajar offline subiendo después los datos.

Estados persistentes almacenados en caches del navegador LocalStorage y SessionStorage ligados al dominio y a la sesión para recuperar el estado de la aplicación aunque se refresque la

[1] https://es.wikipedia.org/wiki/Web_sem%C3%A1ntica

[2] https://es.dbpedia.org/

página.

APIs para acceder a periféricos o conexiones como el GPS, acelerómetros, Bluetooth, etc...

Como vemos el estándar trata de cubrir las necesidades que van surgiendo dotando a los navegadores de nuevas capacidades para aproximarse al desarrollo de aplicaciones nativas en las distintas plataformas, lo cual ha supuesto una gran tendencia de uso en el desarrollo de interfaces para todo tipo de dispositivos.

2.2. JavaScript

Con el objetivo de lograr que las paginas web fuesen dinámicas y tener funcionalidades, apareció JavaScript en 1995. Es un lenguaje de alto nivel orientado a objetos, con tipado dinámico no estricto y prototipos en vez de clases para la herencia.

Se han definido varios estándares desde su aparición asentándose ECMAScript. Todos los navegadores incluyen una implementación de dicho estándar más o menos moderna. En este proyecto utilizamos la última versión publicada en 2015 ECMAScript 6 (ES6).

Este lenguaje nació como tecnología para construir frontend, es decir aplicaciones cliente con interfaz gráfica. Con los años al evolucionar y crecer su uso han acabado portándose sus interpretes para poder usarse como tecnología de backend de la que hablaremos mas adelante. Respecto al lado del cliente que es el contexto de nuestra aplicación, lo interesante es la interacción con el DOM [4] para el pintado. El DOM (Document Object Model) es la representación en memoria del árbol de elementos de la página que tiene una API en javascript para su interacción pudiendo acceder a el o modificar la página en tiempo real.

Por último cabe destacar la infinidad de *frameworks* y librerías que surgen a diario actualmente, pues es un mercado enorme donde crecen las necesidades mas allá de los estándares.

Un *framework* puede servirnos para definir el diseño y la arquitectura de una aplicación web a través del uso de patrones de diseño [7] como por ejemplo, el conocido Modelo Vista Controlador (MVC) donde podremos separar la lógica de la vista respecto al modelo de datos y

la lógica de negocio.

Hoy en día existen muchos frameworks para este propósito, dentro de los más conocidos podríamos destacar Angular[3], un MVC (Modelo Vista Controlador) completo y cerrado, React[4] y Vue.js[5] que ambos son prácticamente para la vista, los cuales disponen de un ecosistema modular donde según las necesidades el desarrollador podrá importar esas librerías en sus proyectos.

2.3. Typescript

Typescript [10] es un lenguaje de alto nivel creado por Microsoft en 2012 con el objetivo de estandarizar y llevar la potencia de los lenguajes tipados orientados a objetos al desarrollo de aplicaciones frontend, aunque también puede usarte como lenguaje de backend. Es un superconjunto construido sobre Javascript, el cual permite tener código Javascript nativo o importar librerías nativas.

Su interprete transpila al estándar de Javascript que se necesite abstrayendo al usuario de la compatibilidad con el navegador. Aporta muchas herramientas que se han echado en falta por parte de las comunidades como el tipado estricto para las interfaces, lo cual ahorra muchos errores en tiempo de ejecución, la orientación a objetos con herencia y clases (sin prototipos) y operadores de alto nivel.

Trae también cuenta con herramientas de mas nivel de abstracción propio de los frameworks actuales de backend, como decoradores o inyección de dependencias, los cuales permiten implementar multitud de patrones de una forma mas sencilla y robusta.

Cabe destacar que una de sus ventajas es la definición de módulos que va a combinar bien el empaquetado y distribución de librerías de lo que hablaremos más adelante.

En la figura podemos ver un ejemplo de transpilación de Typescript a Javascript para ver mas claro el concepto.

[3] `https://angular.io/docs`
[4] `https://reactjs.org/`
[5] `https://vuejs.org/`

Figura 2.1: De Typescript a Javascript

2.4. JSON

Cabe destacar también este conocido formato de objetos utilizado como vehículo en la gran mayoría de configuraciones y APIs actuales. Antiguamente en las aplicaciones SOAP se usaba XML que es un sistema de marcado similar HTML pero con el propósito de almacenar campos y valores.

JSON nace con el mismo propósito pero mas cercano a la notación de un objeto en Javascript, es un formato de texto para la distribución de datos, como vemos en la Figura 2.2. Es un formato muy sencillo de leer y de *parsear*, es decir, de obtener su información.

Figura 2.2: Ejemplo JSON básico

Los tipos primitivos de datos que soporta JSON son: números, texto, *booleanos*, *arrays* y

objetos de nuevo formados por tuplas clave-valor.

2.5. NodeJS

NodeJs es un servidor web desarrollado en C++ en el año 2009. Esta plataforma nos permite desarrollar en el lado del servidor con el lenguaje Javascript y el estándar ES6.

Es un servidor donde las características destacables son que es *monohilo* y asíncrono. Como vemos en la Figura 2.3, tenemos un único hilo donde por cada petición se delega el trabajo y se da respuesta de manera asíncrona, es decir, no es bloqueante y se ejecutará de manera concurrente recibiendo dichas respuestas mediante funciones *callback*.

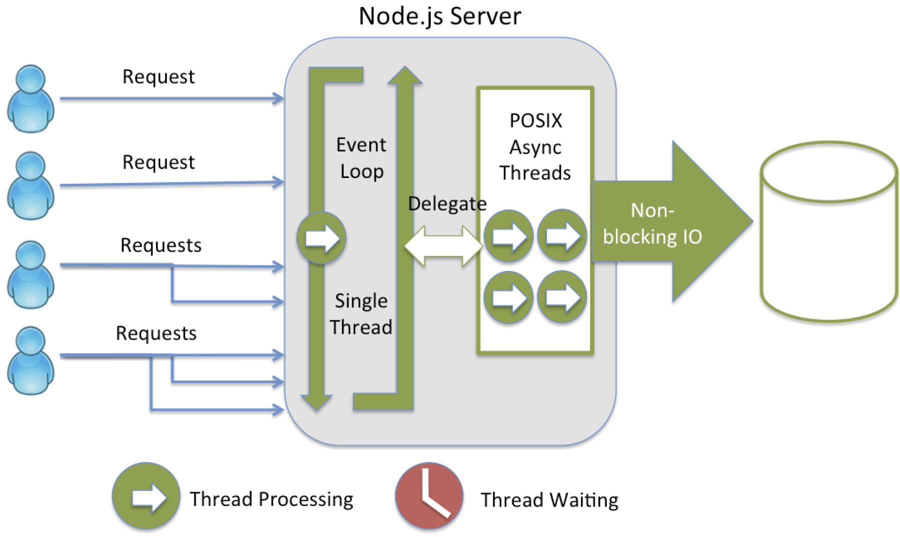


Figura 2.3: Arquitectura NodeJs

Utiliza como motor interno el v8 desarrollado por google y utilizado en el conocido navegador Chrome [6].

Cabe destacar que NodeJs se utiliza como servidor web en infinidad de compañías y tiene una fuerte comunidad detrás apoyando su desarrollo. Por ejemplo Chrome utiliza el motor v8 de node.

[6] `https://v8.dev/`

En este proyecto lo hemos usado para empaquetar los módulos de nuestra aplicación y distribuirla apoyándonos en Webpack, tecnología de la que hablaremos más adelante, además de servidor de desarrollo.

2.6. npm

Al igual que las tecnologías de backend de alto nivel, Node cuenta con un gestor de dependencias llamado npm (Node package manager), desarrollado en 2014, el cual se nutre de un archivo llamado package.json donde además de las dependencias se define metadatado del proyecto como las tareas y la configuración de la aplicación. Esta tecnología es análoga a Maven en Java por ejemplo.

Gracias a esta herramienta se facilita el poder crear, compartir, reutilizar y contribuir a los distintos módulos desarrollados por la comunidad.

2.7. Webpack

Webpack es un empaquetador de módulos definidos en Javascript y otros lenguajes generalmente enfocados en el desarrollo web de frontend. El paradigma de la programación modular nace para facilitar la forma de importar y exportar código con funcionalidades concretas para así evitar cargar de forma asíncrona scripts probablemente muy grandes y tener mecanismos para particionar y organizar el código de una forma más elegante y eficiente.

Como podemos ver en la Figura de su arquitectura 2.4 es una herramienta que nos permite empaquetar aplicaciones *JavaScript* modernas que conformadas por elementos de varias naturalezas módulos, imágenes, fuentes, hojas de estilos. Pero su único propósito no es el em-

paquetado, también se usa para ejecutar tareas, convertir formatos, servidor de desarrollo con carga en caliente para facilitar el desarrollo, etc...

Una de las partes mas interesantes son los mecanismos de ingeniería de rendimiento para el empaquetado como la ofuscación de código, carga perezosa bajo demanda o treeshaking que te permiten ignorar lo que realmente no estés usando y muchas otras técnicas con el objetivo de aligerar peso en las aplicaciones web, ya que el tráfico por usuario es un punto crítico en las redes.

Actualmente las capas de ingeniería de front son muchas para muchos propósitos y cada vez mas complejas pues no paran de surgir necesidades y se trata de acercar a los lenguajes de backend por lo que sin herramientas de empaquetado por módulos como Webpack, Parcel o Rollup perderíamos mucha automatización y rendimiento en el empaquetado.

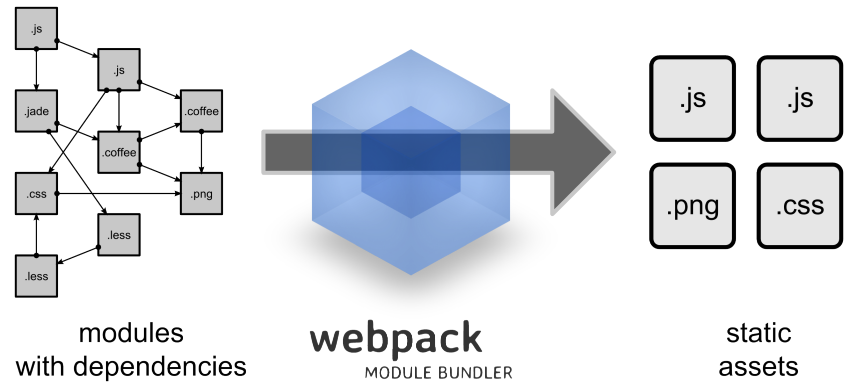
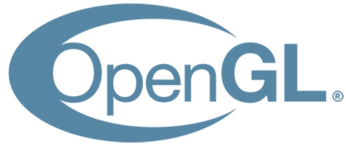


Figura 2.4: Qué es Webpack

2.8. OpenGL



OpenGL es una librería de gráficos de código abierto lanzada en el año 1994. Consiste en una API en varios lenguajes y multiplataforma con el objetivo de pintar gráficos en 2D y 3D. Nació con el objetivo de democratizar y estandarizar el desarrollo de aplicaciones de este sector que contaba con software privado prácticamente.

Tiene distintas aplicaciones y usos en varios negocios, el más inmediato y uno de los mercados mas grandes el desarrollo de videojuegos, pero también tiene su hueco en simulaciones, educación, demos para industria.

Respecto a su integración en el mundo web podemos hablar de WebGL.

WebGL [8] o *Web Graphics Library* es un estándar que define una API escrita en JavaScript que implementa OpenGL por debajo con el objetivo de trasladar el pintado de gráficos en 3D al navegador.

Esta tecnología se apoya en un elemento de HTML5 que mencionamos anteriormente llamado Canvas 3D o accesible mediante el DOM y Javascript.

Respecto a la parte gráfica y hardware, permite aceleración por GPU y procesamiento de imágenes y efectos. La gestión de la memoria se realiza mediante Javascript.

En cuanto a los sombreados o shaders se definen en un lenguaje nativo llamado GLSL y se le transmiten a la API de WebGL como cadenas de texto, instrucciones que son traducidas y compiladas a código GPU.

2.9. Realidad Virtual

Entendemos por realidad virtual un escenario que simula la realidad generada artificialmente mediante dispositivos digitales que surge a finales de la década de 1980.

En la Figura 2.5 podemos ver un ejemplo de dispositivo de realidad virtual. El dispositivo principal suelen ser unas gafas para realizar una inmersión en el mundo virtual y abstraerse de la realidad convencional. Además, como vemos en la imagen, estos dispositivos cuentan con mas periféricos como mandos para desplazarse o realizar acciones dentro de este mundo virtual.

También se ha podido simular con dispositivos menos específicos como pueda ser un teléfono móvil con acelerómetros.

Desde un inicio, esta tecnología ha estado orientada a la industria de los videojuegos. Pero no ha tardado en expandirse y tener aplicación en el mundo de la medicina, enseñanza, simulación, industria y turismo.

Figura 2.5: Dispositivo de Realidad Virtual

También cabe mencionar la Realidad Aumentada, donde tendríamos modelos virtuales superpuestos con la realidad como podemos ver en la Figura 2.6.

Figura 2.6: Realidad Aumentada

2.10. WebXR

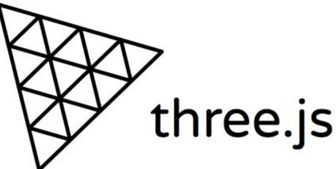
WebXR (Extended Reality) es una API [1] de navegador para obtener información sobre los periféricos de realidad virtual, aumentada o mixta y poder interactuar con ellos.

Sus principales objetivos son detectar de manera automática dispositivos de realidad virtual y obtener sus características principales, como posición y orientación del dispositivo y mostrar imágenes con una latencia razonable.

En el caso de VR anteriormente existía un estándar llamado WebVR con los mismos objetivos pero solo en el escenario de realidad virtual que ha sido deprecado en favor de WebXR.

Estas tecnologías se apoyan en WebGL para el pintado de gráficos 3D aportando las referencias y configuración la cámara y los dispositivos controladores.

2.11. Three.js



Three.js[7] es una librería [2] escrita en JavaScript para mostrar contenido 3D en páginas web. Nos provee la capacidad de generar escenas, modelos, sonido, vídeos, luces, sombras, animaciones, partículas y muchos otros tipos de visualizaciones.

Se apoya en WebGl para el pintado aunque su definición se encuentra a muy bajo nivel aunque nos permite abstraernos del pintado de ciertas cosas relativamente complejas como pueden ser materiales, luces, sombras, animaciones, etc.

En la Figura 2.7 podemos ver la arquitectura general de como se compone dicha escena. Por un lado tenemos la figura geométrica compuesta por vértices y superficies. Esta figura geométrica puede se le puede añadir un material que está compuesta por una o varias imágenes que a su vez componen lo que conocemos por textura. Si sumamos esta figura geométrica y su material obtenemos lo que denominamos una malla. La malla, las luces y la cámara componen nuestra escena 3D.

[7] https://threejs.org

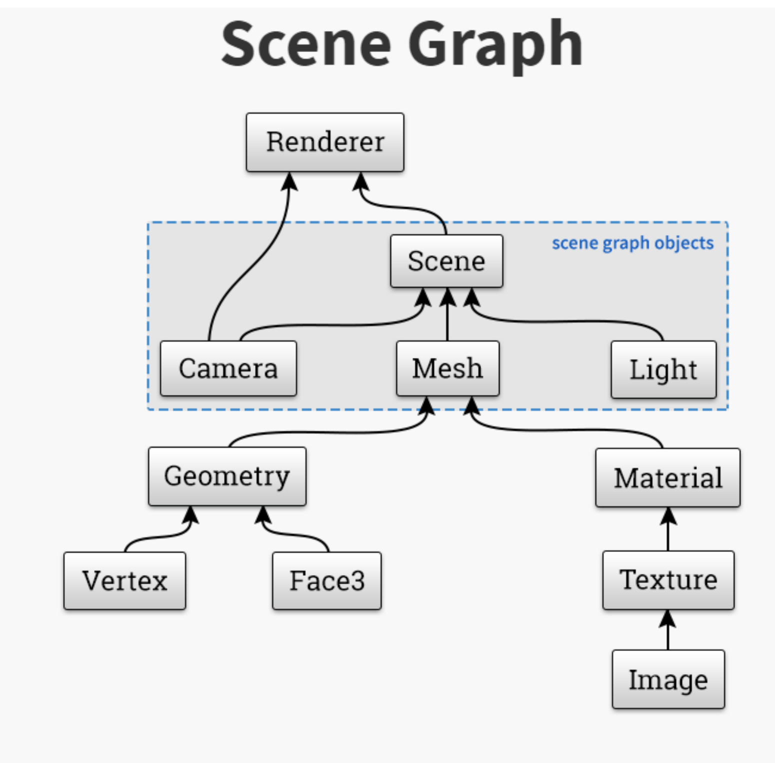


Figura 2.7: Escena Three.js

Por último, de manera transparente para el desarrollador, Three.js invoca a WebGL para representar nuestra escena que se podrá visualizar en cualquier navegador web.

Para comprender mejor el funcionamiento o para que sirve una cámara en una escena 3D tenemos la Figura 2.8. La cámara es un punto de perspectiva desde donde se visualiza la escena 3D. En Three.js tenemos cuatro parámetros básicos para definir una cámara:

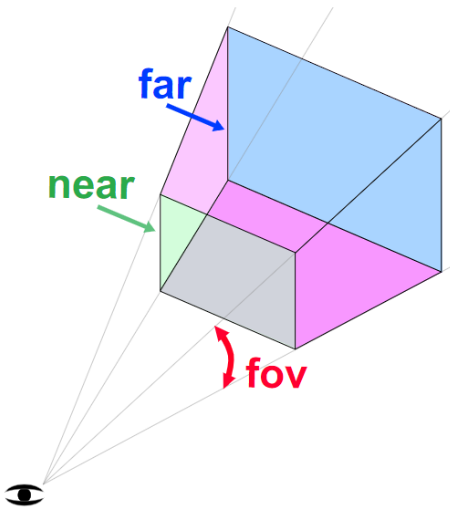


Figura 2.8: Ejemplo de cámara. Forma geométrica ”*frustum*”

- El ”fov” es la abreviatura de *field of view* o campo de visión. Es el ángulo de visión de la cámara y se utilizan grados como medida.

- El segundo parámetro es el ”aspect”. El cual define el aspecto de visualización del lienzo o el ratio de píxeles. Por ejemplo un Canvas de 400 píxeles de ancho por 200 de alto, tendría un ”aspect” de dos.

- Por último tenemos los parámetro ”far” y ”near” que definen la distancia de lo que vamos a poder visualizar en la escena. Si algún objeto quedara fuera de este rango no se visualizará.

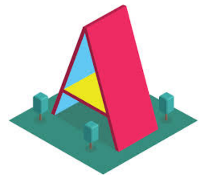
Estos cuatro ajustes definen un ”*frustum*”, es decir, el nombre de una forma 3D en forma de pirámide con la punta cortada. Al fin y al cabo esta definición es la que utiliza la cámara para visualizar la escena.

Ahora veremos un ejemplo de como definir una escena básica en Javascript en la Figura 2.9.

```
<!DOCTYPE html>
<html>
<head>
   <title>Iniciando con Three.js</title>
   <style>canvas { width: 100%; height: 100% }</style>
</head>
<body>
   <script src="three.js"></script>                <!--Incluyendo la biblioteca-->
   <script>

   //Escena
   var scene = new THREE.Scene();                  // Creando el objeto escena, donde se añadirán los demás.

   //Cámara
   var camera = new THREE.PerspectiveCamera(
    75,                                         // Ángulo de "grabación" de abajo hacia arriba en grados.
    window.innerWidth/window.innerHeight,       // Relación de aspecto de la ventana de la cámara(Ejemplo: 16:9).
    0.1,                                        // Plano de recorte cercano (más cerca no se renderiza).
    1000                                        // Plano de recorte lejano  (más lejos no se renderiza).
    );

   camera.position.z = 5;  //Enviar la cámara hacia atrás para poder ver la geometría. Por defecto es z = 0.

   //Renderizador
   var renderer = new THREE.WebGLRenderer({antialias:true}); // Utilizar el renderizador WebGL.
   renderer.setSize(window.innerWidth, window.innerHeight);  // Renderizador del tamaño de la ventana.
   document.body.appendChild(renderer.domElement);           // Añadir el renderizador al elemento DOM body.

  //Geometría
   var geometry = new THREE.CubeGeometry(1,1,1);   // Crear geometría cúbica con dimensiones(x, y, z).
   var material = new THREE.MeshLambertMaterial({color: 0xFF0000}); // Crear el material para la
                                                        // geometría y darle color rojo.
   var cube = new THREE.Mesh(geometry, material);  // Crear una malla que agrupará la geometría
                                    // y el material creados anteriormente.
   scene.add(cube);                                // Añadir la malla al objeto escena.

   //Luz (requerida para el material MeshLambertMaterial)
   var light = new THREE.PointLight( 0xFFFF00 ); //  Luz proveniente de un punto en el espacio,
                                      //  semejante al sol.
   light.position.set( -10, 5, 10 );             //  Localización de la luz. (x, y, z).
   scene.add( light );                           //  Añadir la luz al objeto escena.

   // Función para renderizar
   var render = function () {
    requestAnimationFrame(render);           // la renderización ocurrirá continuamente si la escena está visible.

    cube.rotation.x += 0.03;                 //Velocidad de rotación en el eje x
    cube.rotation.y += 0.03;                 //Velocidad de rotación en el eje y

    renderer.render(scene, camera);          //Renderizar escena cada vez que se ejecuta la función "render()".
   };

   render();

   </script>
</body>
</html>
```

Figura 2.9: Ejemplo de escena con ”*threejs*”

2.12. A-Frame



A-Frame[8] es un *framework web* que permite construir experiencias de realidad virtual o realidad aumentada en el navegador.

Esta tecnología reconoce componentes HTML con etiquetas propias, normalmente precedidas de 'a-', que usará trasladar esa entidad al Canvas 3D apoyándose en tecnologías a más bajo nivel.

Se basa en una filosofía entidad-componente para construir elementos en la escena e interactuar con ellos. A-Frame nos sitúa en un un nivel de abstracción que simplifica el desarrollo de gráficos 3D sin tener que usar Three.js a mas bajo nivel.

A-Frame es mantenido por la comunidad de WebVR y Supermedium[9], el cual es un navegador web puramente diseñado para WebVR y usado con dispositivos de realidad virtual.

Entre sus mayores características encontramos:

- A-Frame provee de un arquetipo 3D, configuración para VR y los controles predeterminados. Sin necesidad de instalación o compilación, únicamente es necesario cargar el script de la librería y crear el componente <a-scene>.

- Como vemos en la Figura 2.10, se puede partir de un HTML sencillo de entender.

```html
<html>
  <head>
    <script src="https://aframe.io/releases/0.9.2/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>
      <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9"></a-box>
      <a-sphere position="0 1.25 -5" radius="1.25" color="#EF2D5E"></a-sphere>
      <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" color="#FFC65D"></a-cylinder>
      <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4" color="#7BC8A4"></a-plane>
      <a-sky color="#ECECEC"></a-sky>
    </a-scene>
  </body>
</html>
```

Figura 2.10: Ejemplo código HTML de A-Frame

[8] https://aframe.io/

[9] https://supermedium.com/

- Utiliza una arquitectura basada en entidad-componente. Provee de una interfaz para declarar y acceder a objetos de Three.js o usar la API de WebGl.

- Es una librería VR diseñada para ser multidispositivo y multiplataforma. Es compatible con Vive, Rift, Windows Mixed Reality, Daydream, GearVR y Cardboard con soporte para sus controladores.

- El rendimiento es otro punto fuerte de esta librería. A-Frame está optimizada para WebVR. Los componentes no manipulan el motor 3D del navegador y las actualizaciones se realizan en memoria llegando a alcanzar los 90 *fps* en escenas con infinidad de elementos.

- Además cuenta con utilidades de desarrollo como el inspector de elementos que vemos en la Figura 2.11 . Con una interfaz similar a la de otros editores de escenas permite inspeccionar y editar la escena.

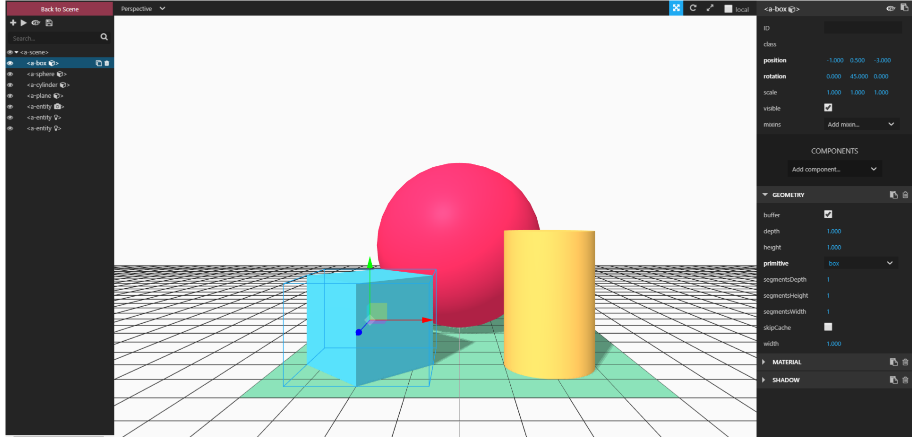


Figura 2.11: Ejemplo del inspector de A-Frame

- La API de A-Frame nos permite manipular elementos tales como geometría, material, colores, luces, animaciones, modelos, difusores de rayos, sombras. La comunidad ha aportado otros como escenarios, ambiente, partículas, físicas, fluidos y nuevas interacciones.

- Ha sido adoptada por grandes empresas de distintos sectores como Google, Disney, Samsung, Toyota, Ford, Chevrolet, CERN, NPR. Entre las que algunas han realizado contribuciones para mejorar A-Frame, por lo que su roadmap y su mantenimiento tienen objetivos sean ambiciosos y para casos de uso reales.

2.13. Git

Como herramienta de control de versiones, en este proyecto se utiliza Git [5]. Este *software* fue diseñado por Linus Torvalds (Creador del *kernel* de *Linux*) en 2005, cuyo propósito era crear un sistema capaz de llevar el registro de cambios en archivos y coordinar el trabajo que realicen varias personas simultáneamente sobre dichos ficheros.

Respecto a los sistemas de control de versiones más tradicionales como Subversion o Mercurial, introdujo una nueva filosofía de reflejar un registro de cambios en local que puede ser sincronizado con el servidor. Pudiendo el usuario mantener diferentes estados sin necesidad de preocuparse de reflejarlo inmediatamente en el servidor o perder cambios al cambiarse de rama.

Con el tiempo se ha consolidado en el mercado como el cliente de control de versiones estándar.

Respecto a las tecnologías servidoras se han creado muchos dominios con pequeñas funcionalidades fuera del contexto de git con el objetivo de administrar los repositorios de código y la colaboración, tan famosos como pueden ser Github actualmente de Microsoft, Bitbucket de Atlassian o gitlab.

Gracias a estos servicios podemos ver publicar el código o mantenerlo de forma privada como propiedad intelectual, gestionar permisos para usuarios y colaboración y añadir un concepto muy interesante llama 'pull request' donde un usuario crear una rama partiendo de un commit de otra rama proponiendo funcionalidad nueva que podrán revisar compañeros o responsables del repositorio en la comunidad.

2.14. Tecnologías Similares

El mercado del diseño 3D ha evolucionado mucho a lo largo de los años desde junto con el hardware que ha sido el motor para permitir tener productos más ambiciosos en todo tipo de sectores, videojuegos, películas de animación, arquitectura, simulación y diseño en ingeniería e industria, etc.

Por lo que existe muchas aplicaciones algunas libres y otras privadas de las cuales hablaremos de las más conocidas que han servido como fuente de inspiración en las funcionalidades más básicas.

Blender

Es un programa multiplataforma de software libre dedicado al modelado, animación e iluminación en gráficos 3D. También permite editar vídeo y automatizar tareas escritas en Python. Es muy conocido y usado por la comunidad debido a que se trata de software libre, especialmente para modelar.

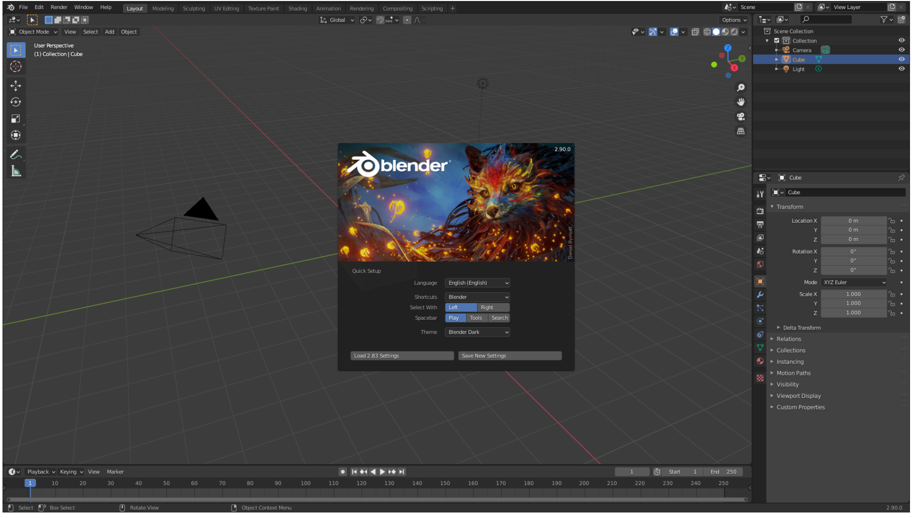


Figura 2.12: Blender ejemplo

Autodesk 3DS Max

Es un programa de diseño 3D propiedad de Autodesk publicado en 1990, empresa conocida por software para ingeniería y arquitectura desarrolladora del conocido Autocad. Es de los programas más usados en el sector sobretodo en publicidad, películas y videojuegos.

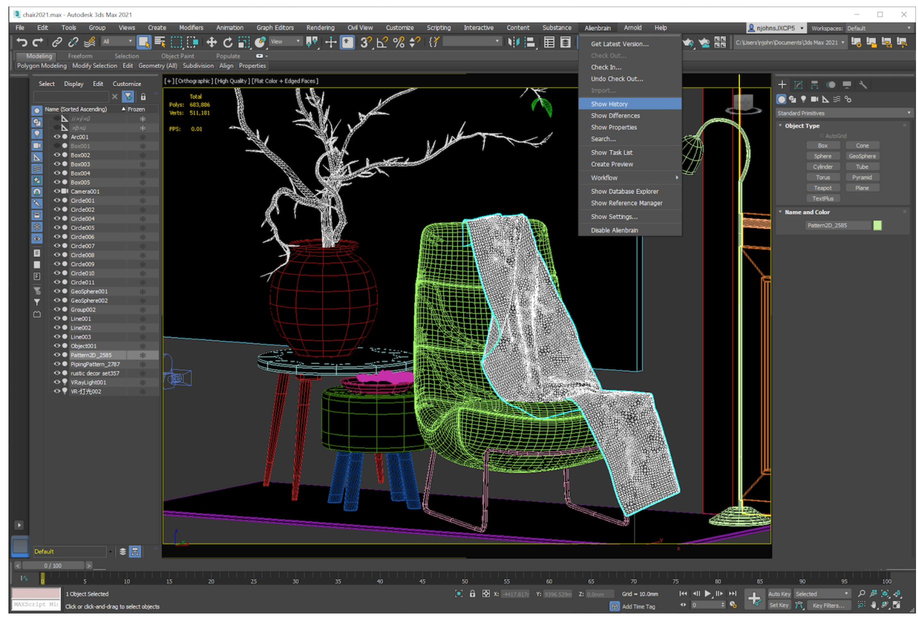


Figura 2.13: 3DS Max ejemplo

Unity

Es un motor gráfico multiplaforma integrado en un IDE de desarrollo con un editor de escenas, fue publicado en 2005. A día de hoy es un referente en el mundo del desarrollo de videojuegos multiplataforma por las facilidades que ofrece reutilizando assets de la comunidad, la posibilidad de monetizarlos y la simplificación que ha supuesto en el empaquetado y la distribución de aplicaciones para todo tipo de dispositivos y stores digitales. Está escrito en C++ y permite desarrollar en C# que es el estándar en .NET y Microsoft. Respecto al desarrollo web, se puede escribir también el fuente en JS, antes contaba con un plugin que corría en los navegadores, pero actualmente corre sobre OpenGL y empaqueta mucha de las operaciones como un WASM para ejecutar directamente como ensamblador del navegador a bajo nivel.

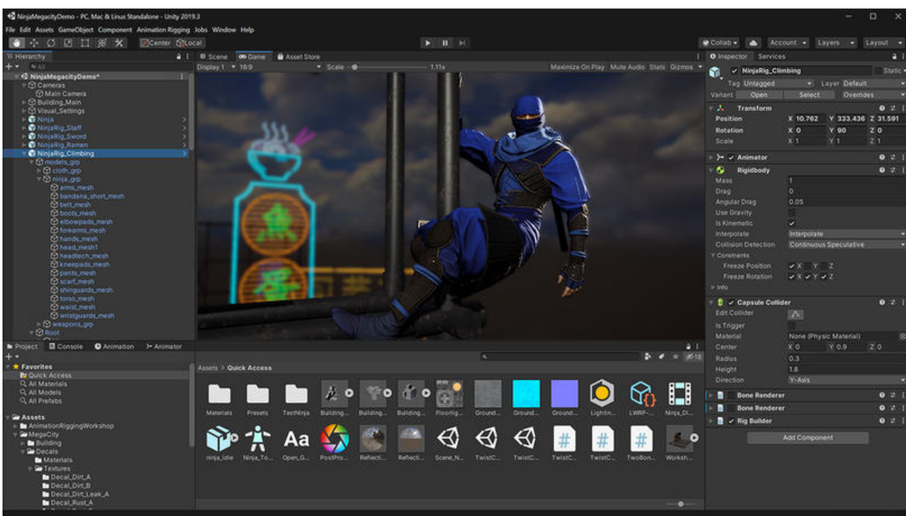


Figura 2.14: Unity ejemplo

Tras mencionar estas aplicaciones que son de las más conocidas por su repercusión y lo que han aportado a la industria y la comunidad, podemos enumerar muchas otras por su importancia y presencia en el mercado como Catia, Maya, Houdini, Sketch up, cualquier aplicación de Adobe o Autodesk e incluso videojuegos que sirven para editar escenas como el conocido Minecraft, que en su caso es una gran fuente de inspiración pues ofrece una vista en primera persona.

Respecto a otros proyectos de editores en VR por ejemplo el propio Supermedium tienen un ejemplo de editor con un mando muy interesante [10] [11].

Otros ejemplos de la comunidad [12].

Por supuesto recordar el trabajo de otros compañeros sobre editores que también ha llevado mi tutor.

Aun que no sean editores de escenas dentro de la realidad virtual, me parece interesante comentar frameworks que son competencia de A-Frame y conocidos en la comunidad.

- **Babylonjs:** A mas bajo nivel que A-frame, con una comunidad muy fuerte [13].
- **Primrose** [14]

[10] `https://supermedium.com/supercraft/`
[11] `https://supermedium.com/craft/`
[12] `https://github.com/thedart76/aframe-webvr-scene-editor`
[13] `https://www.babylonjs.com`
[14] `https://www.primrosevr.com`

- **React 360** [15]

En esta web se muestra el estado del arte del desarrollo de aplicaciones WebVR, hablando de los navegadores compatibles y sus capacidades, frameworks de desarrollo y los editores y herramientas de la comunidad [16].

También quería mencionar este pequeño proyecto de la comunidad porque me parece interesante y admirable, Guri VR [17].

En cualquier caso como podemos ver es un terreno por explorar posibilidades y donde la comunidad está empezando a invertir esfuerzos pero todavía no existen herramientas estandarizadas y rodadas por empresas y sectores.

[15] https://github.com/facebookarchive/react-360
[16] https://createwebxr.com/webVR.html
[17] https://gurivr.com/guide

Capítulo 3

Diseño e implementación

En este apartado hablaremos del diseño e implementación de este proyecto. Comenzaremos describiendo la metodología utilizada y las distintas etapas por las que ha pasado. Más adelante profundizaremos en cada una de dichas etapas, describiendo el desarrollo que se ha realizado de manera incremental para este trabajo.

3.1. Metodología SCRUM

La metodología de trabajo que se ha seguido es una inspiración de la metodología SCRUM [9], para estructurar el desarrollo de forma superficial en sprints y marcar objetivos iterativos.

SCRUM es un marco de trabajo con el objetivo de ayudar a gestionar proyectos y equipos que adquieren cierta complejidad, clasificada como metodología ágil. Esta sistema estandariza las fases del desarrollo como puedan ser la toma de requisitos, el contacto con cliente, fase de refinamiento, implementación, verificación y mantenimiento.

El objetivo es lograr que el equipo trabaje en pequeños incrementos tangibles y alcanzables que aporten valor al producto que se desarrolla y gestionar los recursos disponibles de la forma más eficiente posible.

Existen diferentes roles en este proceso empezando por el *SCRUM Master* que trata de guiar al equipo y asegurarse de que se cumplen la lineas de trabajo definidas, detectando bloqueos y ayudando en la medida de los posible para que el trabajo evolucione favorablemente.

El resto de roles involucrados son los desarrolladores, product owner o project manager que deciden que objetivos abordar en el proyecto y la figura del cliente para la toma de requisitos.

En este caso yo he ejercido de desarrollador y mi tutor de proyecto hacía las veces de 'cliente' delimitando los objetivos de los supuestos sprints o incrementos, mediante reuniones periódicas con seguimiento de los avances.

A continuación se describe brevemente cada una de las etapas o *sprints* por las que ha pasado este proyecto:

- **Sprint 0:** Esta es la fase previa donde se realizará un estudio de los requisitos del proyecto y las tecnologías a utilizar. La idea principal es familiarizarse con la librería A-Frame utilizando la documentación oficial y practicar mediante ejemplos básicos, en este caso crear una escena con acciones y movimientos.

- **Sprint 1:** En esta primera etapa se tiene como objetivo interactuar con figuras básicas y poder desplazarlas a través de la escena. Para ello crearemos un repositorio de código inicial donde se va a usar Webpack como empaquetador y servidor de nuestra librería.

- **Sprint 2:** En el segundo paso realizaremos el trabajo de aprender a clonar y reproducir figuras en otros puntos a mediante el DOM para insertar figuras en la escena.

- **Sprint 3:** En la tercera fase se realizará el grueso de la edición de una figura, creando menús para modificar cada una de las propiedades.

- **Sprint 4:** En esta ultimo sprint abordaremos la multiselección de figuras y como aplicar operaciones en bloque, por otro lado se incluirá funcionalidades extra a la escena para consolar mas la edición de la misma.

3.2. Sprint 0. Estudio previo.

En esta fase previa comenzaremos por familiarizarnos con las tecnologías que vamos a utilizar en el proyecto.

Partiendo de unos conocimientos básicos de HTML, Javascript y los navegadores abordaremos directamente el estudio de el *framework* A-Frame que es la barrera. Como con cualquier tecnología desconocida comenzaremos con la web oficial de A-Frame[1] donde se puede encontrar la documentación[2] y un quickstart para comenzar a crear escenas 3D.

[1] https://aframe.io/

[2] https://aframe.io/docs/1.2.0/introduction/

Comenzamos creando un proyecto demo. Por comentarlo utilizaremos el IDE WebStorm[3] de la compañía Jetbrains para comenzar con el desarrollo. Crearemos un proyecto HTML vacío que constará de los siguientes componentes: última versión de la librería de A-Frame[4] mimificada y el siguiente fichero HTML con el código que vemos a continuación.

```
<html>
  <head>
    <script src="https://aframe.io/releases/1.2.0/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>
      <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9"></a-box>
      <a-sphere position="0 1.25 -5" radius="1.25" color="#EF2D5E"></a-sphere>
      <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" color="#FFC65D"></a-cylinder>
      <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4" color="#7BC8A4"></a-plane>
      <a-sky color="#ECECEC"></a-sky>
    </a-scene>
  </body>
</html>
```

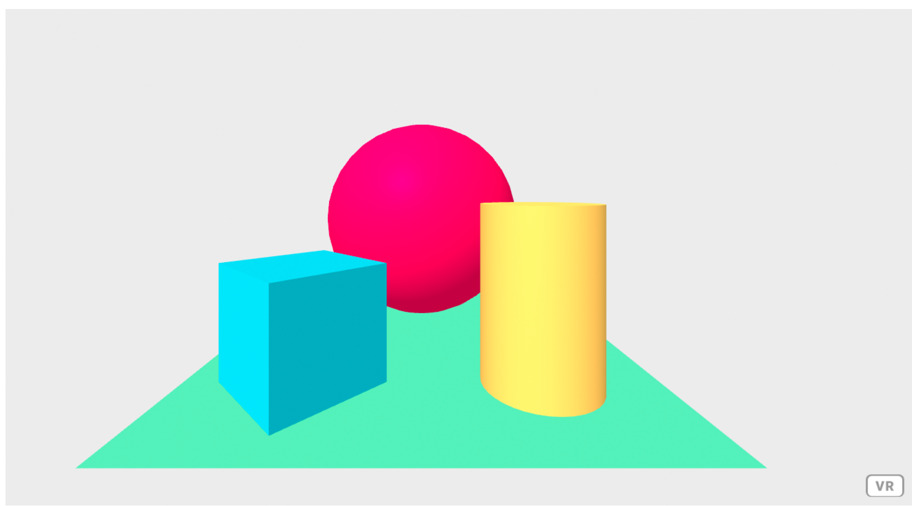


Figura 3.1: Primera escena con A-Frame (resultado)

Como se puede ver en el código, A-Frame interpreta distintas etiquetas de marcado como:

[3]https://www.jetbrains.com/webstorm/
[4]https://aframe.io/releases/1.2.0/aframe.min.js

– <a-scene>: La escena es el objeto raíz global y todas las entidades están contenidas dentro de la escena. Que es lo que el framework de A-Frame interpretará y trasladará al Canvas 3D.

  Figuras:

– <a-box>: Es una de las formas geométricas primitivas de la librería, un prisma en definitiva.

– <a-sphere>: Otra forma primitiva para crear una forma esférica a partir de un radio.

– <a-cylinder>: Esta etiqueta crea cilindros y superficies curvas.

– <a-plane>: Una primitiva para crear superficies planas.

  Elementos de Escena:

– <a-sky>: El cielo agrega un color de fondo o una imagen de 360 ??° a una escena. En este caso el cielo se representa como una esfera por dentro con un color y una textura.

– <a-camera>: Componente que define la posición y orientación inicial de la vista en la escena, así como la forma de desplazarse y como modificar la orientación o la posibilidad de interactuar con elementos al mirarlos.

El resultado de esta escena de ejemplo podemos verla en la imagen 3.1.

Tras jugar un poco definiendo propiedades como la posición, tamaños o colores, antes de ponernos con tareas del editor el tutor me sugirió el reto de tratas de interactuar con las escena y desplazar elementos, por lo que nos embarcamos en una demo sencilla donde tratara de accionar una palanca para desplazar una plataforma y crear un puente entre dos superficies.

Aproveché para empezar a modelar elementos y sus interacciones en la escena con orientación a objetos.

```
1 class Lever {
2
3     _componentId = 'a-lever';
4     _entityRef = {};
5     onLoad;
6
```

```
    constructor (attrs) {
        // Promise resolve when is inserted on scene
        this.onLoad = new Promise((resolve) => {
            LeverAction.register(this.switching);
            this._append(attrs, resolve);
        });
    }

    _append = (attrs, resolve) => {
        const sceneEl = document.querySelector('a-scene');
        this._entityRef = document.createElement('a-cylinder');
        ...
        this._entityRef.setAttribute(LeverAction.componentId, '');
        sceneEl.appendChild(this._entityRef);
        setTimeout(() => resolve('inserted!'), 0);
    }

    isActivated = () => this._entityRef.getAttribute('rotation').z === -45
    _rotate = (direction) => { ... }
    switching = () => { ... }
}
```

Afrontando esta tarea aprendí a interactuar con eventos, en este caso mirando fijamente con la cámara, evento 'fused' para A-Frame, a la palanca, lo cual te obliga a implementar entidades con comportamiento que es de las piezas mas interesantes y útiles de la librería.

```
class LeverAction {
    static componentId = 'lever-action';

    static register = (callback) => {
        console.log('register lever-action');
        AFRAME.registerComponent(this.componentId, {
            init: function () {
                let el = this.el;
                console.log('init lever-action');
                this.el.addEventListener('click', function (evt) {
                    console.log('lever actioned!!!');
                    callback();
                });
            }
        });
```

```
16        }
17 }
```

El resultado de la escena es el que se ve en las imágenes 3.2 3.3.

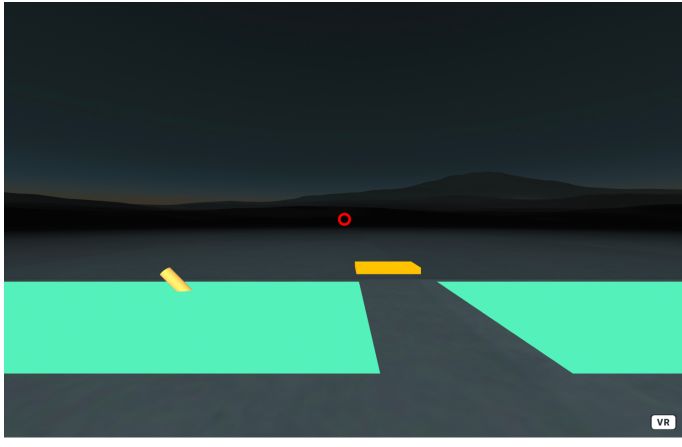


Figura 3.2: Antes de accionar la palanca

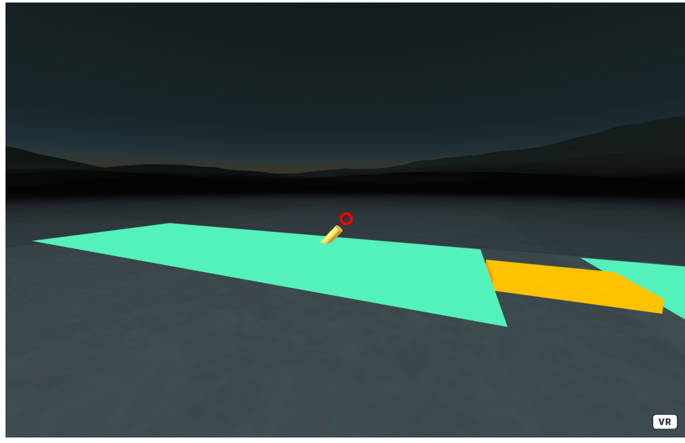


Figura 3.3: Después de accionar la palanca

3.3. Sprint 1. Mover y arrastrar figuras

El objetivo de este primer sprint va a ser construir una escena con elementos draggables, hemos decidido apoyarnos en una librería llamada superhands[5] que nos ayudará con este propósito.

Para construir nuestra aplicación vamos a empezar con los requisitos y bases para ello por lo que comenzaremos creando un repositorio de código público en GitHub donde subiremos los primeros desarrollos de la aplicación. Como suele hacerse en cualquier repositorio de desarrollo de software, tendremos dos ramas de desarrollo, develop y master.

Master representa un estado estable y productivo, develop por el contrario puede ser un estado intermedio probablemente no productivo en la mayoría de los momentos. Así podremos ir desarrollando y probando distinta funcionalidad y solo subiremos a nuestra rama principal cuando tengamos el código probado y listo para producción.

Como ahora nos enfrentamos a un proyecto real con el objetivo de ser productivo y distribuible, definiremos una estructura básica de proyecto frontend que incluye la configuración del empaquetado mediante Webpack.

Con esta información, se genera un nuevo proyecto llamado a-frame-editor-scene con la arquitectura que vemos en la Figura 3.4

[5] https://github.com/wmurphyrd/aframe-super-hands-component

```
sprint1-dragging
├── images
│   └── favicon.png
├── package.json
├── README.md
├── src
│   ├── components
│   │   ├── figures-palette
│   │   │   └── figures-palette.ts
│   │   └── selectable-figure
│   │       └── selectable-figure.ts
│   ├── index.html
│   └── main.ts
├── tsconfig.json
└── webpack.config.js
```

Figura 3.4: Primera arquitectura del proyecto

Para poder configurar el empaquetado y el compilado tendremos que aprender a usar varias tecnologías de las que hemos hablado en la sección del estado del arte.

Empezaremos por el *package.json* donde se definiremos las tareas automatizadas y la gestión de dependencias que necesitaremos tanto en la parte servidora como en la cliente del navegador. Veremos el fichero a continuación.

```
{
  "name": "aframe-editor-scene",
  "description": "aframe front app for build scenes interactively",
  "version": "1.0.0",
  "license": "MIT",
  "scripts": {
    "start": "webpack serve --config=webpack.config.js --env development=true --host 0.0.0.0 --progress --hot --inline --port 9999",
    "build": "webpack --config=webpack.config.js --env production=true"
  },
  "dependencies": {
    "aframe": "^1.0.4",
    "aframe-event-set-component": "^5.0.0",
    "aframe-environment-component": "^2.0.0",
    "lodash-es": "^4.17.15",
```

```
15      "rxjs": "^6.6.3",
16      "super-hands": "^3.0.0"
17    },
18    "devDependencies": {
19      "webpack": "^5.4.0",
20      "webpack-cli": "^4.2.0",
21      "webpack-dev-server": "^3.11.0",
22      "typescript": "^4.0.5",
23      "tslint": "^6.1.3",
24      "@types/lodash": "^4.14.165",
25      "babel-cli": "^6.26.0",
26      "babel-preset-env": "^1.7.0",
27      "html-loader": "^1.3.2",
28      "ts-loader": "^8.0.11",
29      "file-loader": "^6.2.0",
30      "html-webpack-plugin": "^4.5.0",
31      "clean-webpack-plugin": "^3.0.0",
32      "terser-webpack-plugin": "^5.0.3"
33    },
34    "keywords": [
35      "aframe",
36      "aframe-editor-scene"
37    ]
38  }
```

Las tareas definidas son para desplegar en local la aplicación o para empaquetarla, suele haber muchos mas tipos de tareas como lanzar tests o parsear la aplicación en busca de errores de sintaxis lo que comúnmente se llama 'linting'. Por otro lado vemos las dependencias, las que son usadas para empaquetar y desarrollar y las que son propias de la aplicación, que serán descargadas en el directorio *src/nodemodules*.

El comando de entrada para interactuar con la definición de este fichero será npm, que recordamos como explicamos anteriormente es el gestor de dependencias de node. Cuenta con alias tales como 'npm start' que hará referencia a la tarea de 'start' o 'npm install' que resolverá las dependencias. Para el resto será 'npm run xxx'.

Continuaremos con Webpack, que como dijimos es la tecnología utilizada para empaquetar y distribuir módulos de Javascript.

Donde podemos ver su documentación para entender los conceptos y como configurarlo. [6]

La configuración del proceso de construcción y empaquetado será definida en un fichero llamado *webpack.config.js* que mostramos a continuación:

```
const path = require('path');
const basePath = __dirname;
const distPath = 'dist';

function getWebpackConfig(env) {
    let mode, devtool, bundleName;
    if (env.production) {
        mode = 'production';
        devtool = false;
        bundleName = '[name].[contentHash].js';
    } else if (env.development) {
        mode = 'development';
        devtool = 'eval-source-map';
        bundleName = 'main.bundle.js';
    }
    return {
        mode: mode,
        resolve: {
            extensions: [ '.ts', '.js' ]
        },
        entry: {
            app: ['./src/main.ts']
        },
        output: {
            path: path.join(basePath, distPath),
            filename: bundleName
        },
        devtool: devtool,
        module: {
            rules: [
                {
                    test: /\.ts?$/,
                    use: 'ts-loader',
                    exclude: /node_modules/
                },
```

[6]https://webpack.js.org/guides/getting-started/

```
36                  {
37                      test: /\.html$/,
38                      // Exports HTML as string, require references to static
    resources
39                      use: ["html-loader"]
40                  }
41              ]
42          },
43          plugins: [
44              new HTMLWebpackPlugin({
45                  filename: 'index.html',
46                  template: './src/index.html',
47                  inject: 'head'
48              }),
49              // Deletes the dist folder, so the new .js files wont stack and pollute
    the folder
50              new CleanWebpackPlugin()
51          ],
52          optimization: {
53              usedExports: true,
54              minimizer: [new TerserPlugin(), new HTMLWebpackPlugin({
55                  template: "./src/index.html",
56                  // Injects file in the head of the html
57                  inject: 'head',
58                  // Settings for the html file itself
59                  minify: {
60                      removeAttributeQuotes: true,
61                      collapseWhitespace: true,
62                      removeComments: true
63                  }
64              })]
65          }
66      };
67  }
```

En un primer vistazo podemos ver que cuenta con varias secciones que iremos viendo rápidamente.

Empezaremos detectando si el transpilado de la aplicación se hará para desarrollo o para un entorno productivo pues no se abordará de la misma manera.

Para desarrollo no realizaremos técnicas de optimizado ni empaquetaremos, uno de los de-

talles más significativos es que no se ofuscará el código precisamente para que sea legible en depuración. Otra cosa a destacar es que se servirá mediante un servidor de estáticos internos con filewatchers sobre el fuente, esto quiere decir que si realizamos un cambio en caliente se retranspilará y se servirá de nuevo la aplicación con poca latencia.

Respecto a la versión productiva empaquetaremos con todas las técnicas de optimización generando una salida final denominada bundle, en el directorio dist, abreviatura de 'distribuible'.

Lo siguiente que veremos son los ficheros que tratará de procesar y los puntos de entrada a la aplicación que serán una vista, es decir un HTML que en nuestro caso es el *src/index.html* y por otro lado el script inicial con la lógica de la aplicación, que en nuestro caso es *src/main.ts*.

Respecto al script, en nuestro caso partiremos de *src/main.ts*, el cual importará al resto de módulos según vaya necesitando. Su salida será un bundle.js, que será el código ofuscado y optimizado, más algunos chunks, que son scripts más pequeños con otras partes de código que puedan separarse y sean susceptibles de pedirse bajo demanda con carga perezosa cuando la aplicación requiera de esa funcionalidad con el objetivo de minimizar el tráfico.

Después vemos que hay varias reglas a aplicar según la naturaleza del fichero o de la operación, las cuales requieren de módulos, que son extensiones de funcionalidad que no forman parte de Webpack y realiza la comunidad. Estas extensiones se descargan como dependencias del proyecto.

Cabe destacar que para este proyecto hemos decidido utilizar el lenguaje Typescript, como describimos anteriormente es un lenguaje definido encima de Javascript para ayudar al desarrollador, es código fuente, no es Javascript nativo interpretable por el navegador por lo que tendrá que ser transpilado por un transpilador de Typescript que llamaremos mediante Webpack.

El transpilador necesita definir cierta configuración para saber que decisiones tomar a la hora de transpilar, por ejemplo el estándar de Javascript que usará pues no todos lo navegadores implementan los más modernos. Este fichero es *src/tsconfig.json*. En Webpack se hará referencia en la parte de reglas para los ficheros ts mediante el plugin 'ts-loader'.

Ahora que ya está toda la parte de configuración podemos empezar con el fuente de nuestra aplicación.

Si miramos el index.html se definen varias etiquetas que A-Frame interpretará en tiempo de ejecución en el navegador.

En el *Sprint 0* ya vimos las básicas que usaremos ahora.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>aframe-vr-editor-scene</title>
    <meta name="description" content="aframe-vr-editor-scene">
    <link rel="shortcut icon" type="image/png" href="images/favicon.png"/>
  </head>
  <body>
    <a-scene>
      <a-assets>
        <img id="groundTexture" src="https://cdn.aframe.io/a-painter/images/floor.
    jpg" crossorigin="anonymous">
        <img id="skyTexture" src="https://cdn.aframe.io/a-painter/images/sky.jpg"
    crossorigin="anonymous">
      </a-assets>

      <!-- Camera. -->
      <a-entity id="rig" position="-1 0.5 -2">
        <!-- Mouse camera-->
        <a-entity id="camera-mouse" camera wasd-controls
                  cursor="rayOrigin:mouse"
                  raycaster="objects: .selectable"
                  super-hands="colliderEvent: raycaster-intersection;
                               colliderEventProperty: els;
                               colliderEndEvent:raycaster-intersection-cleared;
                               colliderEndEventProperty: clearedEls;"
        ></a-entity>

      </a-entity>

      <!-- Background sky. -->
      <a-sky height="2048" radius="30" src="#skyTexture" theta-length="90" width="
    2048"></a-sky>

      <!-- BackGround floor. -->
      <a-circle src="#groundTexture" rotation="-90 0 0" radius="32"></a-circle>

    </a-scene>
  </body>
</html>
```

Lo mas destacable es el cambio de cámara que ahora es sensible al click del ratón gracias a 'rayOrigin:mouse', pues vamos a desarrollar en una primera implementación la versión de escritorio y nos enfrentamos a eventos de drag donde arrastrar.

Por otro lado podemos ver el tag de super hands, es una librería de terceros con el fin de abstraer y simplificar la interacción con las escenas y los eventos a partir de cualquier periférico y en este caso lo usaremos para detectar el evento de drag. Esta sería la única parte de la escena definida estáticamente, el resto se definirá en tiempo de ejecución de forma programática.

Vamos entonces a ver el fichero main.ts donde veremos los módulos y dependencias a importar y la definición dinámica de la escena.

```
1  import 'aframe';
2  import 'super-hands';
3
4  import { FiguresPalette } from "./components/figures-palette/figures-palette";
5
6  document.addEventListener("DOMContentLoaded", function(event) {
7
8      console.log("DOM fully loaded");
9
10     // Render figures palette
11     const figuresPalette = new FiguresPalette(
12         {
13             position: "0 0.5 -2",
14             rotation: "0 0 0"
15         }
16     );
17
18 });
```

Lo primero dado que es el entrypoint de código es resolver las dependencias de aframe y superhands, si estuviesen exportadas como módulos se podrían cargar, pero hacemos un import por defecto para cargar el script y que lo empaquete Webpack.

Después se espera a que se cargue el DOM para construir dinámica el objeto de la paleta de figuras la cual en su constructor que veremos mas adelante irá construyéndolas e insertándolas en el DOM para que A-Frame las pinte posteriormente con las correspondientes propiedades para que sean draggables.

A continuación veremos el código, donde repasaremos brevemente las propiedades que per-

mitirán interactuar con ellas.

```
import "aframe-event-set-component";
import "aframe-environment-component";

class FiguresPalette {

    private componentId: string = 'figures-palette';

    private entityRef: HTMLElement;

    private figures = [
        {
            primitive: 'a-cone',
            color: 'red',
            'radius-bottom': 0.3
        }
        ...
    ];

    constructor (attrs) {
        this.appendPalette(attrs);
        setTimeout(() => this.appendFigures(), 0);
    }

    private appendPalette (attrs) {
        const sceneEl = document.querySelector('a-scene');
        this.entityRef = document.createElement('a-entity');
        sceneEl.appendChild(this.entityRef);
    }

    private setInteractionProperties(figEl) {
        figEl.setAttribute('hoverable', '');
        figEl.setAttribute('grabbable', '');
        figEl.setAttribute('draggable', '');
    }

    private setInteractionBehaviour(figEl) {
        figEl.setAttribute('event-set__hoveron', '_event: hover-start; material.
    opacity: 0.7; transparent: true');
        figEl.setAttribute('event-set__hoveroff', '_event: hover-end; material.
```

```
opacity: 1; transparent: false');
39         figEl.setAttribute('event-set__dragdrop', 'event: drag-drop');
40         figEl.setAttribute('event-set__dragon', '_event: dragover-start; material.
    wireframe: true');
41         figEl.setAttribute('event-set__dragoff', '_event: dragover-end; material.
    wireframe: false');
42     }
43 
44     public appendFigures () {
45         const {x, y, z} = this.entityRef.getAttribute('position') as any;
46 
47         this.figures.forEach((fig, i) => {
48             // Initializing fig html element
49             const figEl = document.createElement(fig.primitive);
50 
51             // Getting initial coords
52             const figCoords = (x + 2) - (i + 1) + ` ${y} ${z}`;
53 
54             // Setting basic props
55             figEl.setAttribute('position', figCoords);
56             let figProps = Object.keys(fig);
57             figProps.forEach((key) => {
58                 if (key !== 'primitive') figEl.setAttribute(key, fig[key]);
59             });
60 
61             // Setting interaction props and events
62             figEl.setAttribute('class', 'selectable');
63             this.setInteractionProperties(figEl);
64             this.setInteractionBehaviour(figEl);
65 
66             this.entityRef.appendChild(figEl);
67         });
68     }
69 }
```

Lo primero a destacar es la necesidad de importar 'aframe-event-set-component' que nos permitirá enlazar eventos a las figuras mediante atributos en el HTML que añadiremos en las funciones 'setInteractionProperties' y 'setInteractionBehaviour', donde básicamente definimos como se comportará físicamente la figura ante los eventos de arrastrar y poner el cursor sobre ellas.

El resto es definir las propiedades básicas de las figuras e insertarlas como hijas de la entidad de la paleta de las figuras.

El resultado podemos verlo en la Figura 3.5.

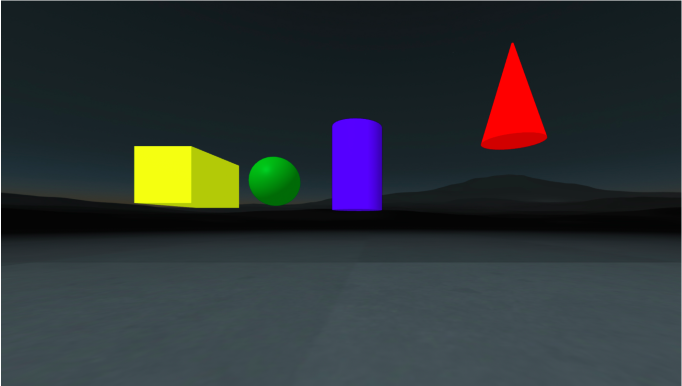


Figura 3.5: Resultado Sprint 1

3.4. Sprint 2. Replicar figuras

En esta segundo sprint nos enfrentaremos al reto de replicar una figura en otra entidad.

Empezaremos por definir una entidad para hacer a otra, en este caso las figuras, sensibles a los eventos de click.

```
import { registerComponent } from 'aframe';

export const selectableFigureAttr = 'selectable-check';

export function registerSelectableFigure(idDest: string) {
    const tableDest = document.querySelector(`#${idDest}`);
    const selectableFigureComponent = {
        dependencies: ['raycaster'],
        init: function () {
            const figSelected = this.el;

            this.el.addEventListener('click', function (evt) {
                _duplicateFigure(figSelected, tableDest);
            });
        }
    };
    registerComponent(selectableFigureAttr, selectableFigureComponent);
```

```
}

function _duplicateFigure(figEl: HTMLElement, parent: HTMLElement) {
    // Cloning figure
    const clonedFigureEl = figEl.cloneNode() as HTMLElement;

    // Append to dest
    parent.appendChild(clonedFigureEl);
}
```

Como podemos ver al principio se crea la función ’registerSelectableFigure’ para registrar el componente que se encargará de escuchar los eventos de click e invocar a la función que clonara la figura. Esta función acudirá al elemento recuperado del DOM en el evento de click y llamará al método ’cloneNode’ perteneciente al tipo ’HtmlElement’ que nos devolverá un nodo replicado que podremos insertar en la referencia que hayamos pasado a la función.

Como haremos dentro de la paleta de Figuras en este caso.

```
constructor (attrs, initialFigures, idDest) {
    this.figures = initialFigures || new Array<Figures>();
    this.appendPalette(attrs);
    setTimeout(() => {
        // registering component to convert into selectable figures (Custom
    behaviour)
        registerSelectableFigure(idDest);

        // Getting palette coords
        const {x, y, z} = this.entityRef.getAttribute('position') as any;

        // Insert palette figures
        this.figures.forEach((fig, i) => {
            const figCoords =  `${(x + 2) - (i + 1)} ${y} ${z}`;
            appendFigure(fig, figCoords, this.entityRef);
        });
    }, 0);
}
```

Donde el id pasado en construcción será el de la superficie de clonado.

Podemos ver el resultado en la Figura 3.6.

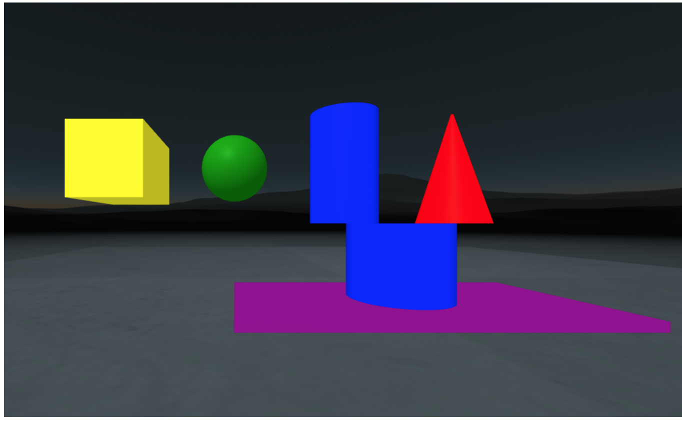


Figura 3.6: Resultado Sprint 2

3.5. Sprint 3. Editando figuras

En esta sección abordaremos el problema de editar las propiedades de las figuras estudiando la API de A-frame para ello y por otro lado tendremos que enfrentarnos a incluir interfaces de usuario para ello acudiendo a una librería de terceros llamada 'aframe-gui' [7].

Nos planteamos el reto de editar las propiedades más básicas como, color, opacidad, tamaño y otras un poco más avanzadas como sombras y materiales.

Según la naturaleza del parámetro nos planteamos opciones diferentes para editarlas, botones, barras, conmutadores, ...etc.

La primera toma de contacto fue modelar la construcción de un panel donde insertar todo este tipo de elementos.

Por lo que creamos en una nueva función donde podremos pasar propiedades según la documentación de los componentes partiendo de unas básicas.

```
export function createContainer(props?): HTMLElement {
    const containerGui = document.createElement('a-gui-flex-container');

    // Style properties
    const defaultProps = {
        'flex-direction': 'column',
        'justify-content': 'center',
        'align-items': 'normal',
        'component-padding': '0.1',
        opacity: '0.7',
        width: '0.8',
        height: '0.25',
        position: '0 0 0.05 0'
    }

    setHtmlTags(containerGui, defaultProps);
    setHtmlTags(containerGui, props);

    return containerGui;
}
```

Este panel será definido con una clase donde será invocada esta función y las necesarias para ir insertando los controles que contiene.

[7] https://github.com/rdub80/aframe-gui

```
export class EditMenuFigure {

    private entityRef: HTMLElement;
    private readonly figure: Figure;

    constructor(fig: Figure) {
        this.figure = fig;

        this.createMenuContainer();

        // Add controls properties
        addControlCloseMenu(this.entityRef, fig);
        addControlEditColor(this.entityRef, fig);
        addControlEditOpacity(this.entityRef, fig);
        addControlEditSize(this.entityRef, fig);
        addControlEditWireframe(this.entityRef, fig);
        addControlEditMaterial(this.entityRef, fig);
        addControlEditShadow(this.entityRef, fig);
    }

    private createMenuContainer() {
        this.entityRef = createContainer({
            width: '1.3',
            height: '3.1',
            position: '0 2 0'
        });
        this.figure.htmlRef.appendChild(this.entityRef);
        this.entityRef.setAttribute('visible', 'false');
    }

}
```

La clase será llamada en la función que inserta cada figura pasándosela como referencia para poder insertarse en el DOM como hijo de ella, oculta inicialmente hasta mostrarla en el evento de doble click, para el cual hemos tenido que cambiar el comportamiento.

```
export function registerSelectableFigure() {
    const tableDest = document.querySelector(`a-plane`);
    const selectableFigureComponent = {
        dependencies: ['raycaster'],
        init: function () {
```

```
6               const figSelected = this.el;
7               let lastClick = null;
8
9               // Double click
10              this.el.addEventListener('click', function (evt) {
11                  evt.stopPropagation();
12
13                  // Detect double click manually implementation
14                  if(evt instanceof MouseEvent) {
15                      if (!lastClick) {
16                          lastClick = new Date().getTime();
17                      } else {
18                          const thisClick = new Date().getTime();
19                          const isDblClick = thisClick - lastClick < 400;
20                          if (isDblClick) {
21                              // hide figure menu
22                              const menuRef = figSelected.childNodes[0];
23                              menuRef.setAttribute('visible', 'true'); // Changing
    visibility
24                          }
25                          lastClick = null;
26                      }
27                  }
28              });
29          }
30      };
31      registerComponent(selectableFigureAttr, selectableFigureComponent);
32  }
```

Podemos ver el resultado en la Figura 3.7.

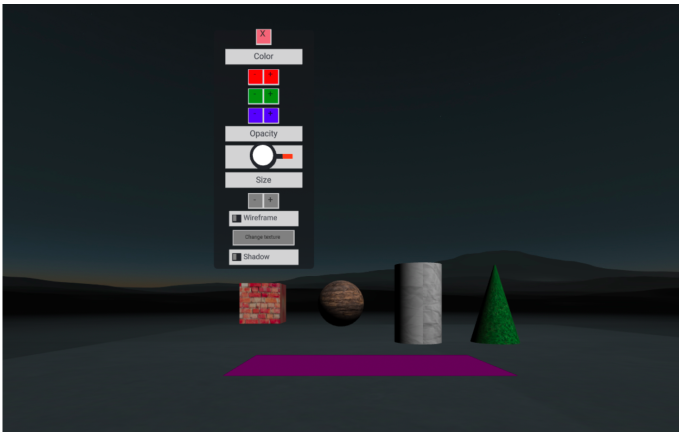


Figura 3.7: Resultado Sprint 3

Repasando rápidamente de forma individual cada propiedad modificable:

- El color ha sido separado en formato RGB con botones que incrementan o decrementan en 25 la intensidad.
- La opacidad es un slider analógico donde elegir un valor en tanto por 1.
- El tamaño también se escala con un multiplicador de 1.2 donde gana el 20 % del tamaño.
- La textura es una imagen que se adhiere a la figura y la cambiamos con un botón que la elige entre una lista interna de imágenes.
- La sombra proyectada se activa con un conmutador que retorna valores binarios.

Quedando entonces el modelo de la figura de la siguiente forma.

```
class Figure {
    id?: string;
    htmlRef?: HTMLElement;
    primitive?: string;
    color?: string = 'white';
    material?: any;
    shadow?: boolean = false;
    opacity?: number = 1;
    wireFrame?: boolean = false;

    setColor?(color: string) {
        this.color = color;
```

```
13          this.htmlRef.setAttribute('color', color);
14      }
15
16      setMaterial?(material: any) {
17          this.material = material;
18          const materialAttr: string = material && propsInLine(material);
19          this.htmlRef.setAttribute('material', materialAttr);
20      }
21
22      setOpacity?(percent: number) {
23          this.opacity = percent
24          this.htmlRef.setAttribute('opacity', percent.toString());
25      }
26
27      resize?(scaleFactor: number) {
28          (this.htmlRef as any).object3D.scale.multiplyScalar(scaleFactor);
29      }
30
31      setWireframe?(wireFrame: boolean) {
32          // const wireframeStatus = this.htmlRef.getAttribute('wireframe') === "true
    ";
33          this.wireFrame = wireFrame;
34          this.htmlRef.setAttribute('wireframe', String(wireFrame));
35      }
36
37      setShadow?(shadow: boolean) {
38          // const shadowStatus = this.htmlRef.getAttribute('shadow')['receive'];
39          this.shadow = shadow;
40          this.htmlRef.setAttribute('shadow', `receive: ${shadow}`);
41      }
42
43  }
```

Y Cada uno de los elementos de GUI usados para no mostrar el código de todos serán: ’a-gui-label’, ’a-gui-button’, ’a-gui-slider’, ’a-gui-toggle’.

Respecto a la funcionalidad de los menús veremos un ejemplo de como escuchar un evento según la documentación de su librería, donde tendremos que incluir una callback en el objeto Window que será invocada al dispararse el evento ’onClick’ por ejemplo.

```
1 function addButtonResize(operation: string, container: HTMLElement) {
2     // Create Button
```

```
3     const buttonControl = createButton({
4         value: operation === 'increase' ? '+' : '-',
5         'background-color': 'grey'
6     });
7
8     // Interaction
9     const customAction = operation + 'size';
10    buttonControl.setAttribute('onclick', customAction);
11
12    window[customAction] = function (event) {
13        const factor = 0.2;
14        const op = operation === 'increase' ? (1+factor) : (1-factor);
15        _figure.resize(op);
16    }
17
18    container.appendChild(buttonControl);
19 }
```

Por otro lado cabe resaltar que para la sombra nos hemos visto en la necesidad de incluir una luz en la escena que también se ha definido como modelo.

```
1 new LightScene({
2     type: 'directional',
3     castShadow: true,
4     intensity: 0.98,
5     shadowCameraVisible: false,
6     position: "1 2 1.8"
7 });
```

Y con esto habríamos repasado de forma rápida los problemas a los que nos hemos enfrentado en este sprint.

3.6. Sprint 4. Multiselección

En este sprint nos enfrentamos al último reto de seleccionar más de una figura para realizar operaciones sobre ellas en grupo. Aunque en un primer momento se nos ocurrió trabajar con los colliders para detectar colisiones y dibujar un prisma que interseccionara con las figuras para seleccionarlo, empezamos por una primera aproximación mas alcanzable.

Esta consistió en la posibilidad de cambiar a un modo multiselección donde el click sirviese para incluir en una lista a la figura seleccionada a través de la cual se realizarían las modificaciones sobre sus nodos.

Empezamos por crear un panel lateral donde incluir este botón con el modo multiselección para alternar el funcionamiento del doble click sobre la figura que hasta ahora la insertaba en la escena.

Tras activarlo, el doble click seleccionará las figuras dejándolas de color cyan con lo que podremos hacer la operación incluida de prueba de reescalarlas todas en el mismo menú como podemos ver en las imágenes 3.8 3.9.

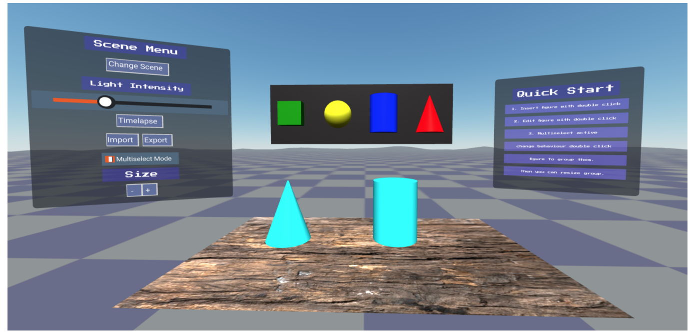


Figura 3.8: Selección figuras

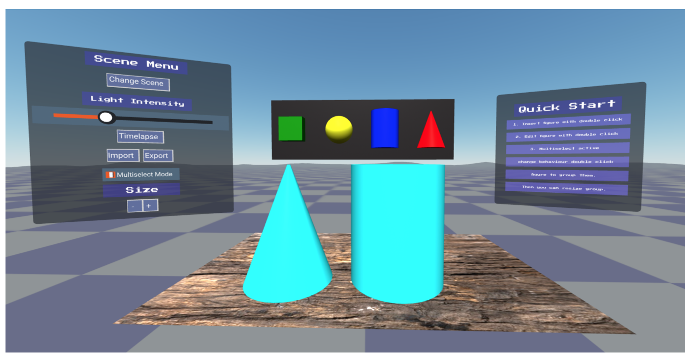


Figura 3.9: Operación grupal

Para ello hemos implementado un estado global con el patrón singleton para tener una única referencia desde cualquier contexto de la aplicación pudiendo acceder a las figuras seleccionadas como veremos en el código.

Definición del estado global.

```
export class GlobalState {

    private static instance: GlobalState;

    private lightScene: LightScene;

    private sceneFigures: Array<Figure> = new Array<Figure>();

    private multiselectEnable: boolean = false;
    private selectedFigures: Array<Figure> = new Array<Figure>();

    private constructor() {}

    static getInstance(): GlobalState {
        if (!GlobalState.instance) {
            GlobalState.instance = new GlobalState();
        }
        return GlobalState.instance;
    }

    // Getters and Setters
....
```

Para marcar las figuras tendremos la siguiente función que añadirá la figura marcada al estado y le cambiará el color.

```
export function markFigureAsSelected(figEl: HTMLElement) {
    const figuresScene: Array<Figure> = globalState.getSceneFigures();
    const selectedFigures: Array<Figure> = globalState.getSelectedFigures();

    const alreadyMarked = Boolean(selectedFigures.find(sel => sel.htmlRef.innerHTML
    === figEl.innerHTML));
    const figModel: Figure = figuresScene.find(fig => fig.htmlRef.innerHTML ===
    figEl.innerHTML);

    if (!alreadyMarked) {
        selectedFigures.push(figModel);
        figEl.setAttribute('color', 'cyan');
    } else {
        globalState.deselectFigure(figEl);
        figEl.setAttribute('color', figModel.color);
    }
}
```

Para acceder al estado y recuperar las figuras para operar con ellas podemos hacer lo siguiente.

```
const globalState = GlobalState.getInstance();

const figuresScene: Array<Figure> = globalState.getSceneFigures();
selectedFigures.forEach(fig => fig.resize(op));
```

3.7. Sprint 5. Extras

Por mejorar la experiencia y la funcionalidad de la aplicación se añadieron algunas operaciones de última hora como

- Exportar e importar la escena como HTML.
- Edición de la luz de la escena.
- Simulación del movimiento del sol a lo largo de un día (Timelapse).
- Cambio de escena.
- Gravedad.

Explicaremos brevemente algunos de los retos a los que nos hemos enfrentado haciéndolos.

Exportar Importar

El objetivo aquí era exportar el DOM de las figuras en escena en formato HTML a un fichero a través del propio navegador, donde tuvimos que acceder a cada uno de los elementos y clonar los nodos para obtener su propiedad 'innerHtml'.

```
export function exportSceneAsHtml() {
    // Getting html scene
    const sceneElements = Array.from(Jquery('#'+clonePodiumId).children()) as Array<
    HTMLElement>;

    let htmlToExport = '';
    sceneElements.forEach(node => {
        const htmlStrNode = getInnerHtml(node);
        htmlToExport += htmlStrNode;
    });

    //Building downloadable file
    const a = document.createElement('a') as unknown as HTMLAnchorElement;
    const fileName = `scene_${new Date().toISOString()}.html`;
    const blob = new Blob([htmlToExport], { type: 'text/html' });
    a.href = URL.createObjectURL(blob);
    a.download = fileName;
    a.click();
}
```

En la acción de importar se tuvo que implementar un buffer de carga del fichero para leerlo como un una cadena de texto que insertar en el DOM y posteriormente enriquecer los nodos con funciones para añadirles los eventos como el de sacar el menú.

```
document.getElementById('inputSceneFile').addEventListener('change', function() {
    var fr = new FileReader();
    fr.onload = function(){
        const fileContent: string = fr.result.toString();
        replaceScene(fileContent);
    }
    fr.readAsText((this as any).files[0]);
});
```

Edición de la Luz

Simplemente se añadió un control al menú para interactuar con el objeto de la luz y modificarle la intensidad. Como podemos ver en las Figuras 3.10 3.11.

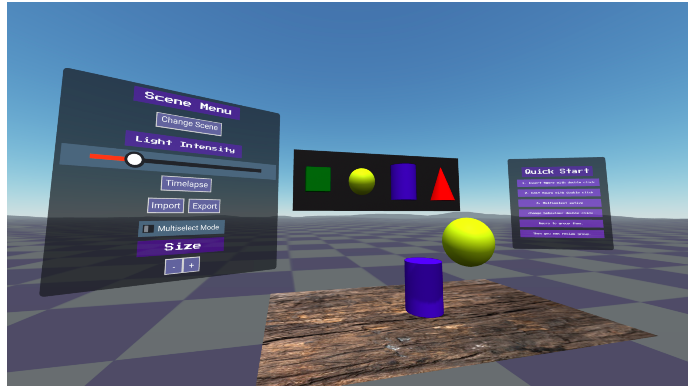


Figura 3.10: Edición luz baja

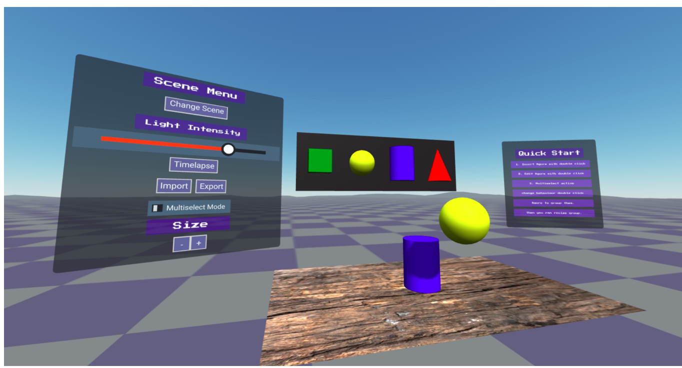


Figura 3.11: Edición luz alta

Respecto al timelapse se creo una función para interpolar posiciones a lo largo de 5 segundos y simular el movimiento del sol a lo largo de un día como podemos ver en las figuras 3.12 3.13.

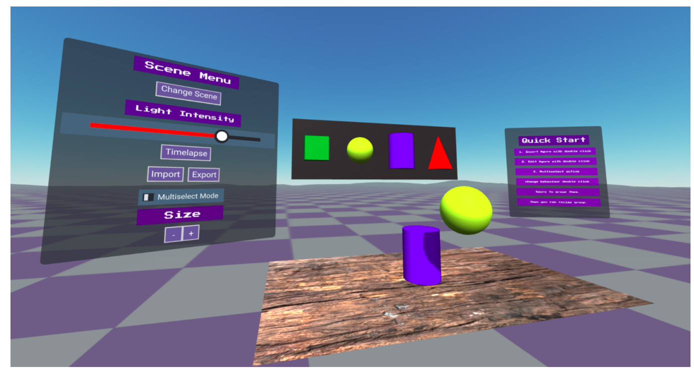


Figura 3.12: Inicio Timelapse

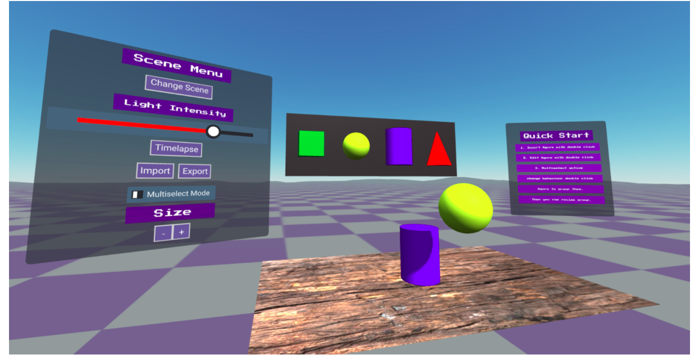


Figura 3.13: Final Timelapse

Cambio de Escena

Se añadió un botón para rotar igual que en las texturas por una lista de fondos que puede cambiar el propio componente de aframe-environment como podemos ver en la Figura 3.12.

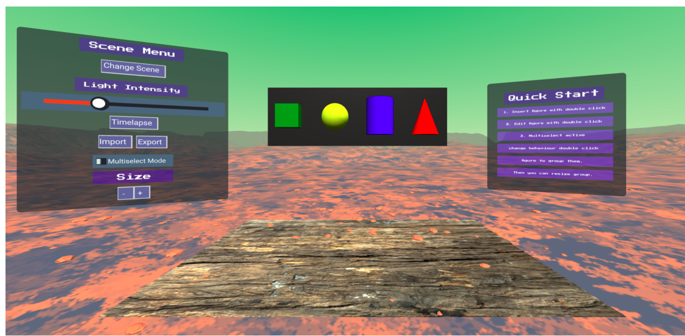


Figura 3.14: Cambio de escena

Gravedad

Aquí acudimos a una de las demos que tenía mi tutor y se incluyo en el modelo de la figura las propiedades necesarias para el comportamiento de las físicas, siendo una figura estática y otras dinámicas sensibles a la gravedad.

La librería en la que se basan las físicas es ammo.js una interfaz Javascript que se apoya en un WASM, para introducir en esta tecnología, los navegadores modernos han habilitado la posibilidad de interpretar un lenguaje ensamblador llamado WebAssembly y a través de un transpilador es posible compilar código de otros lenguajes como C o Python a este ensamblador que entenderá el navegador. Es decir se ha portado una librería de escritorio al navegador.

Esta librería es servida a través del servidor de estáticos y la carga el navegador bajo demanda a través de una función que hemos implementado en el proyecto.

```
const physicsActive: boolean = true;

// Asynchronous loading physics libs
if (physicsActive) loadPhysicsLibs();

export function loadPhysicsLibs() {
    loadScript('ammo.wasm');
    loadScript('aframe-physics-system.min');
}
```

```
// Surface physics
const floor = new Plane({
    id: clonePodiumId,
    height: 3,
    width: 5,
    rotation: '-90 0 0',
```

```
    material: {
        src: textures.WOODEN,
        roughness: 1
    },
    physics: {
        body: 'static',
        shape: 'box'
    }
});

// Gravity physics
export function setPhysicsHtml(figEl: HTMLElement, physics: Physics) {
    if (physics) {
        physics?.body && figEl.setAttribute('ammo-body', `type: ${physics.body}`);
        physics?.shape && figEl.setAttribute('ammo-shape', `type: ${physics.shape}`)
    ;
    } else {
        figEl.removeAttribute('ammo-body');
        figEl.removeAttribute('ammo-shape');
    }
}
```

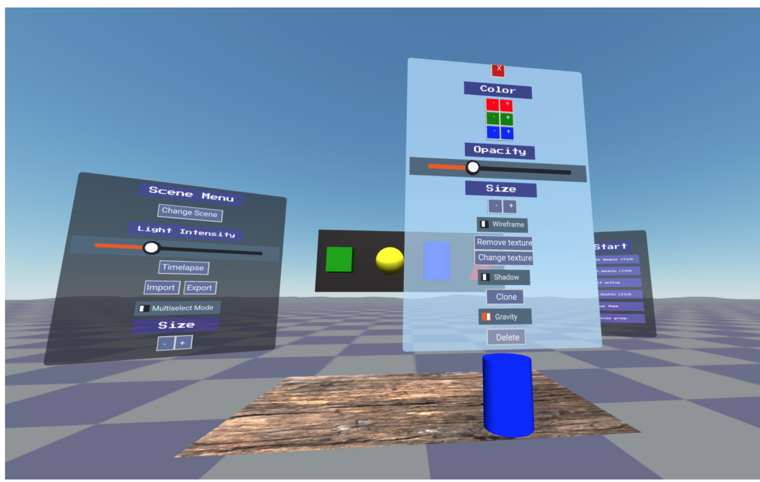


Figura 3.15: Gravedad

Capítulo 4

Resultados

En esta sección revisaremos el estado final de la aplicación y los resultados obtenidos tras el ultimo sprint.

Repasando la funcionalidad de forma conjunta tenemos un aplicación donde podemos insertar figuras en la escena a través del panel central, las cuales pueden ser arrastradas por el escenario y editar sus propiedades como color, opacidad, material, ...etc

Tras tener figuras en la escena, podemos hacer uso de las opciones del panel izquierdo el cual controla la luz de la escena, permite importar y exportar las figuras y por ultimo le multiselección para realizar operaciones sobre un grupo, en este caso el reescalado grupal.

Ahora vamos a ver un demo de la aplicación como usuarios y más tarde veremos como modificar la escena y la estructura del proyecto final.

4.1. Demo

Según abramos la aplicación nos encontramos con esta escena 4.1.

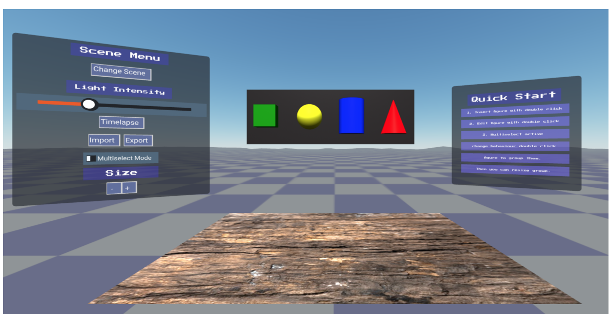


Figura 4.1: Escena inicial demo

Donde veremos 4 elementos:

- **Menú lateral izquierdo**, es el menú de operaciones globales de la escena, donde podremos realizar varias operaciones que explicaremos más adelante.

- **Paleta central de figuras**, es una paleta de figuras clickables para añadirlas en la escena mediante doble click.

- **Panel lateral derecho**, es una pequeña descripción de la escena para saber como operar con ella.

- **Suelo**, es la zona donde se insertarán las figuras y superficie sensible a las físicas gravitacionales.

La interacción que tenemos con la cámara de la escena sera mediante teclado y ratón, si arrastramos con el ratón rotamos la dirección de la cámara y si nos desplazamos con el común cursor de los caracteres ’awsd’, donde ’w’ es arriba, ’a’ es izquierda, ’s’ abajo y ’d’ derecha.

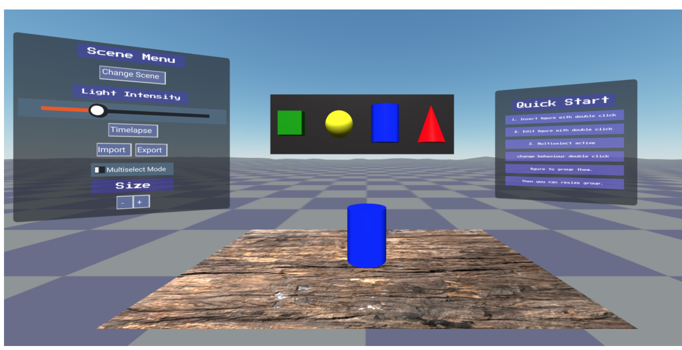


Figura 4.2: Figura insertada

El primer paso natural será añadir una figura a la escena, para ello haremos doble click en la figura y veremos como aparece en escena 4.2.

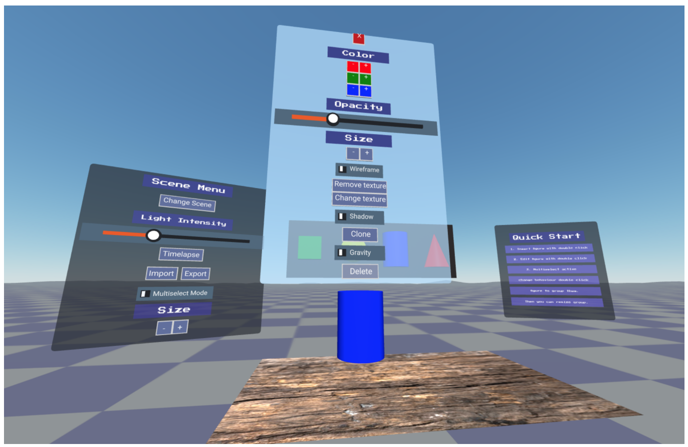


Figura 4.3: Figura con menú

Tras tener la figura en escena podemos interactuar de la siguientes maneras con ella. Bien podemos arrastrarla y moverla a lo largo de la escena dejando pulsado el click o bien hacer doble click para desplegar el menú de operaciones sobre ella, donde de forma intuitiva veremos que podemos cambiar propiedades como los colores, el material o la opacidad como se puede ver en la Figura 4.3.

Por comentarlo, si se tiene activada la gravedad la figura no se puede arrastrar porque la física predomina sobre ese desplazamiento. Por otro lado si se activa la gravedad estando la figura fuera del tablero caerá al vacío.

Respecto a la sombra, sirve para hacerse opaco y reflejar sombras de otras figuras, la forma más sencilla de verla es poner una figura delante de otra y activar la sombra en ambas.

Ahora pasaremos a explicar las operaciones del menú de la escena ubicado a la izquierda:

- **Change scene**, este botón cambiará el fondo de la escena para ubicarnos en una temática diferente, inicialmente partimos de la escena más básica para la demo para que no molesten ni distraigan los relieves.
- **Control intensidad luminosa**, es una barra analógica para controlar la luminosidad de la luz inicial insertada en la escena.
- **Timelapse**, este botón generará una simulación del recorrido del sol en un día a lo largo de 5 segundos para jugar con la luz de la escena y ver como las sombras de las figuras se desplazan.
- **Importar y exportar**, estos botones nos permiten exportar e importar las figuras de la escena.
- **Modo multiselección**, este botón cambia el comportamiento del doble click sobre las figuras, el cual agrupara figuras para aplicarles una operación.
- **Reescalado multiselección**, teniendo figuras seleccionadas se pueden reescalar el grupo.

Y con esto habría terminado el recorrido por la demo.

Ahora podemos repasar la implementación y la estructura de la escena para entenderla y poder hacer cambios.

4.2. Manual de usuario

En este apartado describiremos el funcionamiento de la aplicación y sus componentes con el objetivo de entender mejor la escena y tener la posibilidad de modificarla.

Si observamos el código principal de la aplicación donde se describe la escena.

```
1 // Global libs imports
2 import 'aframe';
3 import 'aframe-event-set-component';
4 import 'aframe-environment-component';
5 import 'super-hands';
6 import 'aframe-gui';
```

```
7
8  // Modules imports
9  import { Box, Cone, Cylinder, Figure, Plane, Sphere } from './models/figure';
10 import { SceneRef } from './services/scene-ref';
11 import { appendFigure } from './helpers/figure-helper';
12 import { clonePodiumId, textures } from './utils/constants';
13 import { LightScene } from './components/light-scene/light-scene';
14 import { FiguresPalette } from './components/figures-palette/figures-palette';
15 import { GlobalMenu } from "./components/global-menu/global-menu";
16 import { registerSelectableFigureScene } from "./components/behaviour-components/
       selectable-figure-scene/selectable-figure-scene";
17 import { loadPhysicsLibs } from "./utils/script-loading";
18 import { defineImportEvent } from "./utils/export-scene";
19 import { InfoMenu } from "./components/info-menu/info-menu";
20 import { GlobalState } from "./services/global-state";
21
22 document.addEventListener("DOMContentLoaded", function(event) {
23
24     console.log("DOM fully loaded");
25
26     // Building scene
27
28     const physicsActive: boolean = true;
29
30     // Asynchronous loading physics libs
31     if (physicsActive) loadPhysicsLibs();
32
33     defineImportEvent();
34
35     const scene = SceneRef.getInstance().getSceneEl();
36     const globalState = GlobalState.getInstance();
37
38     const lightScene = new LightScene({
39         type: 'directional',
40         castShadow: true,
41         intensity: 0.9,
42         shadowCameraVisible: false,
43         position: "0 3 4"
44     });
45     globalState.setLightScene(lightScene);
46
```

```
47      const floor = new Plane({
48          id: clonePodiumId,
49          height: 3,
50          width: 5,
51          rotation: '-90 0 0',
52          material: {
53              src: textures.WOODEN,
54              roughness: 1
55          },
56          physics: {
57              body: 'static',
58              shape: 'box'
59          }
60      });
61      appendFigure(floor, '0 0.01 1.5', scene);
62      registerSelectableFigureScene();
63
64      const initialFigures: Array<Figure> = [
65          new Cone({
66              'radius-bottom': 0.3,
67              height: 0.8,
68              color: 'red'
69          }),
70          new Cylinder({
71              radius: 0.3,
72              height: 0.8,
73              color: 'blue'
74          }),
75          new Sphere({
76              radius: 0.3,
77              color: 'yellow'
78          }),
79          new Box({
80              height: 0.5,
81              width: 0.5,
82              depth: 0.5,
83              color: 'green'
84          })
85      ];
86
87
```

```
88     // Render figures palette
89     new FiguresPalette(
90         {
91             position: "0.2 2.3 0",
92             rotation: "0 0 0"
93         },
94         initialFigures
95     );
96 
97     // Global Menu
98     new GlobalMenu({
99         width: '2.6',
100        height: '2.6',
101        position: '-3.6 2 1.5',
102        rotation: '0 20 0'
103        // 'panel-color': '#93b2e8'
104    });
105
106    // Info Menu
107    new InfoMenu({
108        width: '1.8',
109        height: '1.8',
110        position: '3 1.8 1.5',
111        rotation: '0 -20 0'
112        // 'panel-color': '#93b2e8'
113    });
114
115 });
```

Como vimos anteriormente nada mas empezar hacemos los imports de las librerías de A-Frame que necesitamos:

- **aframe**, es el core de A-Frame.
- **aframe-event-set-component**, la librería para definir eventos a las entidades a través de atributos de HTML.
- **aframe-environment-component**, controla la configuración de la escena, fondo, distancias...etc.

- **super-hands**, es un superset desarrollado por supermedium para controlar eventos de periféricos y traducirlos a eventos de navegador como arrastrar, estirar, clicks..etc.
- **aframe-gui**, una librería de terceros que es para construir interfaces en A-Frame con la que definiremos los menús y botones.

Las siguientes dependencias son los módulos definidos relativos a nuestro proyecto en Typescript donde puede haber desde funciones a clases de las figuras que iremos viendo repasando el código de la escena.

Lo siguiente que nos encontramos es un manejador del evento del DOM cargado para realizar operaciones a partir de que se haya interpretado el index.html y ya esté todo cargado en memoria. Tras esto podremos empezar a escribir código para configurar la escena.

Lo primero con lo que nos encontramos es una constante booleana 'physicsActive' para controlar si se quieren descargar bajo demanda en tiempo de ejecución las librerías relativas a las físicas a través de una función que se ha definido en un utils del proyecto.

Tras esto se instancia la referencia al elemento del DOM de la escena para operar sobre ella e insertar elementos, la constante scene. Y por otro lado el estado global de la aplicación donde se almacenan cosas como los modelos de las figuras en la escena o las que están seleccionadas. Sin esto no funcionarán operaciones desde los menús pues el estado es compartido.

A continuación se define la luz de la escena como un modelo nuestro que también es cacheada en el estado global.

Luego definimos el suelo de madera sobre el que se insertan las figuras y funcionan las físicas gravitacionales, otorgándole un id para que pueda ser recuperado del DOM a través de otras funciones.

La función 'appendFigure' es la piedra angular de la inserción de elementos en la escena, pues a partir de un tipo figura, unas coordenadas y una entidad destino realizar este proceso. Es usada de forma interna por otras clases pero se puede utilizar como una utilidad en cualquier punto.

No menos importante es la función 'registerSelectableFigureScene' que registra el comportamiento con los eventos necesarios para poder abrir el menú de las figuras a través de una nueva entidad para A-Frame.

Lo siguiente que haremos es definir las figuras de que irán en la paleta desde la cual podre-

mos empezar a insertarlas en la escena. Esta paleta también se define a través de su propia clase y recibirá como parámetro la lista de figuras que serán insertables.

Para terminar nos encontramos con la definición de los menús de ayuda y operaciones globales, donde definiremos su tamaño y colocación.

4.2.1. Ejemplo Escena Cambiada

Vamos a ver algunos de los elementos que podríamos cambiar:

- **La cámara**, en el index.html poniendo ’<a-entity id=rig”position=”0 1.6 12”>’.
- **La escena**, en el index cambiando el campo preset ’<a-entity id=”background-scene.^{en}vironment=”prese forest; groundColor: #445; grid: cross”><a-entity>’.
- **La luz**, podemos por ejemplo bajar la intensidad para que parezca de noche poniéndole al objeto lightScene la propiedad ’intensity: 0.2’.
- **Suelo**, podemos cambiar el material por ejemplo a ladrillos modificando en el objeto ’floor’ en el campo ’src’ por ’textures.WALLBRICK’ y el tamaño con ’height’ y ’width’.
- **Figuras paleta**, modificamos el objeto ’initialFigures’ para que aparezcan 4 esferas de colores diferentes.
- **Figura en escena**, dejamos insertada una figura en la escena.

```
const figureScene: Figure = new  Sphere({
        radius: 0.3,
        material: {
            src: textures.STONE,
            roughness: 1
        }
    });
appendFigure(figureScene, '0 0.4 1.5', scene);
```

Quedando el siguiente resultado 4.4.

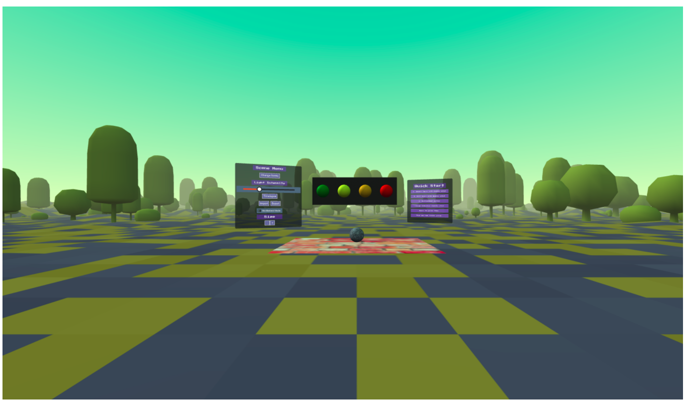


Figura 4.4: Figura con menú

4.3. Estructura

La estructura final que nos ha quedado es la siguiente:

- **assets:** Directorio donde se almacenan recursos estáticos como fuentes, iconos, imágenes, etc.

- **node_modules:** En esta carpeta se almacenan todas las dependencias del proyecto reflejadas en el archivo *package.json* o aquellas que se utilicen de forma indirecta (dependencias transitivas).

- **package.json:** Como hemos mencionado anteriormente contiene el metadatado del proyecto. Además, se utiliza como gestor de dependencias y definición de las tareas de compilación, despliegue, tests, etc.

- **src:** Directorio donde se almacenan el código fuente de la aplicación, del cual detallaremos más la estructura.

    - **components:** En este directorio se encuentran todos los componentes de la aplicación que tienen un modelo asociado. Menús, entidades de comportamiento, paleta de figuras inicial o la luz de la escena.

    - **helpers:** En este directorio tendremos ficheros con funciones auxiliares para realizar operaciones sobre menús y figuras.

- **models:** En este directorio tendremos ficheros con los tipados y las clases para modelar, en este caso lo usaremos para las figuras.

- **models:** En este directorio tendremos ficheros con servicios consumibles desde cualquier punto de la aplicación, en este caso como no tenemos frameworks para este tipo de operaciones, serán objetos implementados con el patrón singleton que usaremos como estado de la aplicación.

- **utils:** En este directorio tendremos ficheros con funciones auxiliares para propósitos autónomos como la exportación e importación con sus funciones de ficheros, definición de constantes, o carga dinámica de librerías.

- **vendor:** En este directorio tendremos ficheros con librerías de terceros previamente descargadas que no podremos importar a través de npm debido a que no tienen módulos de ESM o no están subidas al registry.

- **index.html:** Este fichero es utilizado como la web de partida donde dejaremos definida la escena de forma estática.

- **main.jts:** Este archivo es el punto de partida a nivel de código donde se insertarán elementos en la escena dinámica para construir el editor y se apoyará en el resto de código definido en los directorios anteriores.

- **README.md:** Este archivo sirve para describir el funcionamiento de la aplicación y como arrancarla para un primer vistazo.

- **tsconfig.json:** Este fichero sirve para configurar la transpilación de Typescript cuando Webpack compile la aplicación.

- **webpack.config.json:** Este fichero sirve para configurar la compilación y empaquetado de la aplicación.

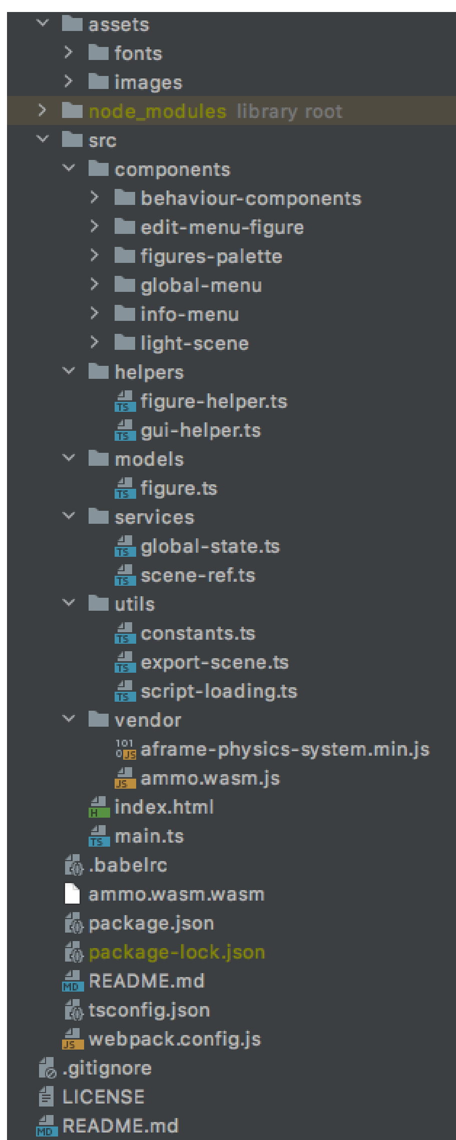


Figura 4.5: Estructura final del proyecto

4.4. Manual técnico

En esta sección hablaremos de los componentes que forman parte de la aplicación desde un punto de vista técnico explicando las interfaces y la implementación como la documentación de cualquier API.

4.4.1. Componentes Visuales

Empezaremos por los componentes que tienen un comportamiento visual con A-Frame.

FiguresPalette

Es la paleta de figuras clickables para insertarlas en la escena.

Solo tiene publico su constructor para definirlo inicialmente:

constructor(props: any, initialFigures: Array<Figure>)

donde props será un objeto libre con campos que tengan sentido que acaben como atributo HTML para que los interprete A-Frame posteriormente como la posición inicial.

initialFigures, será la lista de figuras seleccionables para insertar en la escena que se pintará.

GlobalMenu

Es el menú de operaciones globales de la escena, que a su vez tiene mas componentes internos con los botones y controles que lo componen.

Solo tiene publico su constructor para definirlo inicialmente:

constructor(props: any)

donde props será un objeto libre con campos que tengan sentido que acaben como atributo HTML para que los interprete A-Frame posteriormente como la posición inicial o la rotación.

LightScene

Es la interfaz para incluir una luz en la escena customizable.

Su API es la siguiente:

setIntensity(intensity: number)

Define la intensidad luminosa en tanto por 1.

setPosition(pos: string)

Cambia la posición de la luz en la escena con una cadena en formato 'x y z'.

timelapse()

Esta función hace moverse a la luz simulando la rotación a lo largo de un día interpolando posiciones en un intervalo de 5 segundos.

4.4.2. Componentes de comportamiento

Estos son los componentes que registran entidades sensibles a eventos con determinados comportamientos.

registerSelectableFigureScene()

Es una función que se invoca una vez y a partir de entonces queda registrado el comportamiento que tienen las figuras seleccionables, pudiendo desplegar el menú en ellas a partir del doble click o el modo multiselección.

4.4.3. Helpers

Ficheros con funciones dedicadas a enriquecer operaciones de forma externa al propio modelo, agrupándolas por tipo de componente.

Figuras

Dentro del fichero figure-helper.ts podemos encontrar las siguientes funciones de uso publico:

appendFigure(fig: Figure, figCoords: string, parent: HTMLElement, behaviour: FigureBehaviour = {draggable: false, hoverable: false, custom: ''})

Esta función permite insertar una figura en el elemento entidad deseado, pasándole unas coordenadas sobre el mismo y definiendo las características que tendrá a nivel de comportamiento frente a los eventos.

duplicateFigure(figEl: HTMLElement, parent: HTMLElement)

Esta función sirve para duplicar una figura a partir de su nodo HTML en el DOM e insertarla dentro de otro elemento perteneciente a la escena.

Interfaces de Usuario (Menús)

Aquí hablaremos de las funciones que permiten insertar todos los controles de la librería de 'aframe-gui' que han sido necesarios para construir nuestros menús.

Lo primero será referenciar la documentación de la librería por si quisiéramos consultar las propiedades de estilos, eventos o comportamientos de los componentes sobre los que se ha creado esta API para simplificar su inserción en los menús [1].

[1] `https://github.com/rdub80/aframe-gui`

createContainer(props?): HTMLElement

Sirve para crear un contenedor equivalente a un div en HTML donde agrupar elementos con una maquetación en concreto, pudiendo definir en las props estilos que se asociarán como atributos HTML por ejemplo margenes, ancho, relleno, formato columna o fila, ...etc.

createLabel(text: string, props?: any): HTMLElement

Crea un titulo con texto, el primer argumento es el mensaje, el segundo las propiedades de estilo.

createInputText(props?): HTMLElement

Crear una entrada de texto de formulario, el argumento son las propiedades de estilo.

createButton(props?): HTMLElement

Crea un botón al que posteriormente se le puede asociar un evento, como argumento tiene las propiedades de estilo.

Un pequeño ejemplo de como asociar un evento al botón

```
const buttonElement = createButton({});

buttonElement.setAttribute('onclick', 'customActionName');

window[customAction] = function (event) {
  // code executed on click
}
```

createSlider(props?): HTMLElement

Crea un slider que ofrece valores analógicos en tanto por 1, al que posteriormente se le puede asociar un evento, como argumento tiene las propiedades de estilo.

Ejemplo de como asociar un evento y recuperar el valor del slider

```
const sliderElement = createSlider({
    percent: '0.3'
});

sliderElement.setAttribute('onclick', 'customActionName');

window[customAction] = function (event, percent) {
    // percent return value changed on slider
}
```

createToggle(props?): HTMLElement

Crea un toggle button para conmutar entre valores booleanos como activo o inactivo, al que posteriormente se le puede asociar un evento, como argumento tiene las propiedades de estilo.

Ejemplo de como asociar un evento

```
const toggleElement = createSlider({
    percent: '0.3'
});

toggleElement.setAttribute('onclick', customAction);

window[customAction] = function (event) {
    // only way to recover state is getting from HTML
}
```

Nota. Para recuperar el valor que haya cambiado nos vimos obligados a recurrir a selectores del DOM para comprobar los cambios, no parece que retorne el valor en la callback del evento.

4.4.4. Modelos

Ahora vamos a hablar de las clases y los modelos de datos.

Figure

Es la representación de las figuras básicas que es capaz de representar A-Frame a través de HTML.

Contiene estos atributos de clase que son comunes a todas las figuras.

```
id?: string;
htmlRef?: HTMLElement;
primitive?: string;
color?: string = 'white';
material?: any;
shadow?: boolean = false;
opacity?: number = 1;
wireFrame?: boolean = false;
physics?: Physics = null;

type Physics = {
    body: string;
    shape: string;
}
```

Y contiene métodos de clase que actualizaran tanto el estado del modelo de datos en memoria como la representación a través del DOM.

setColor?(color: string)
Configura el color a partir de una cadena que puede ser notación hexadecimal o su nombre en ingles como en se hace en CSS.

setMaterial?(material: any)
Configura el material pasándole un objeto que tiene como campos

```
{
  src: pathImageResource,
  roughness: 1
}
```

donde ’src’ será la ruta de la imagen estática que añadas a tu servidor de estáticos y ’roughness’ la opacidad en tanto por 1.

setOpacity?(percent: number)

Configura la opacidad de la figura en un número en tanto por 1.

resize?(scaleFactor: number)

Reescala el tamaño de la figura con un multiplicador.

setWireframe?(wireFrame: boolean)

Activa el modo rejilla de la figura

setShadow?(shadow: boolean)

Activa o desactiva la proyección de sombras.

setPhysics?(physics: Physics)

Configura las físicas pasándo un objeto de este tipo.

```
type Physics = {
    body: string;
    shape: string;
}
```

Donde ’body’ define la movilidad de la figura (’dynamic’ o ’static’) y ’shape’ la forma real del comportamiento de la física independientemente de la pintada realmente como ’box’.

4.4.5. Servicios

En la categoría de servicios tendremos clases que usaremos como singleton ya que realmente no usaremos los decoradores de servicio ni lo inyectaremos a través de constructores en los módulos, pero a efectos prácticos hace las veces de ello.

Global State

Aqui tendremos la referencia al estado general de la aplicación donde almacenaremos cosas como la luz de la escena, las figuras o figuras seleccionadas.

getInstance(): GlobalState

Cada vez que se invoque a esta función te devolverá una referencia de instancia única debido a que es una clase estática y se queda definido en tiempo de ejecución la primera vez que se invoca. Esto sirve para llamarlo desde distintos ficheros y ámbitos de funciones sin tener que preocuparte de pasarla por referencia a las funciones.

Getter y Setter Luz

getLightScene()

setLightScene(lightScene: LightScene)

Para cachear y recuperar la luz de la escena.

Figuras de la escena

getSceneFigures()

setSceneFigures(figures: Array $<$Figure$>$)

Para cachear y recuperar las figuras en escena.

removeSceneFigure(figEl: HTMLElement)

Elimina una figura de la escena.

Getter y Setter modo multiseleccion

getMultiselectEnable()

setMultiselectEnable(enable: boolean)

Para modificar y recuperar si está activo el modo multiselección en la escena.

Figuras seleccionadas de la escena

getSelectedFigures(): Array<Figure>

setSelectedFigures(selFigs: Array<Figure>)

Para cachear y recuperar las figuras seleccionadas de la escena.

deselectFigure(figEl: HTMLElement)

Elimina una figura de la selección.

Y para terminar un reset del estado.
resetState()
El cual vacía las figuras seleccionadas y en escena.

Capítulo 5

Conclusiones

5.1. Consecución de objetivos

El objetivo principal de este proyecto, ha sido el de crear una aplicación con aspiraciones a parecerse a un editor 3D del mercado actual dentro de un escenario de realidad virtual a través de un navegador web.

A nivel de objetivos específicos se ha logrado conseguir la edición básica de propiedades de figuras como color, tamaño, texturas, luz, sombras, transparencia y otras más concretas como físicas. También otras operaciones básicas a nivel de escena, como replicar una figura, exportarla en HTML o la multi-selección.

Con esto tendríamos la base de a nivel de componentes y de funcionalidad básica de la escena para poder trabajar en lineas que lo acerquen a un editor actual enriqueciéndolo con más funcionalidades e interfaces donde puede mejorar la experiencia de usuario [3].

Como objetivo personal he cumplido con el reto de implementar la aplicación con Typescript que es un lenguaje tipado de mas alto nivel con el que actualmente se desarrollando la gran mayoría de proyectos frontend y frameworks modernos con el objetivo de escalar mejor y detectar errores en modelos e interfaces, además de toda la funcionalidad extra que aporta.

Aunque ha supuesto un punto más de complejidad en algunos momentos, también he podido apreciar la agilidad y la robustez en varios puntos del desarrollo. De hecho no he visto proyectos de A-Frame implementados en Typescript y podría ser un pequeño ejemplo para la comunidad.

El hecho de haberme enfrentado a un proyecto con tecnologías con las que no estaba familiarizadas haciéndolo de cero y conocer un poco más el sector en auge que es el VR ha sido

bastante enriquecedor a nivel personal y profesional.

5.2. Aplicando Conocimientos

Haciendo retrospectiva de la las asignaturas cursadas a lo largo del Grado he podido sacarle partido a varios de los conocimientos adquiridos en el área del desarrollo de software, sirviendo de base para aventurarme a conocer nuevas tecnologías.

- **Fundamentos de la Programación**: Asignatura base donde se aprenden los conceptos básicos y los pilares fundamentales de la programación estructurada. Definición de variables y funciones, algoritmia básica, estructuras de control de flujo, ficheros, uso de memoria, etc. Donde en frontend se saca el mismo partido que en tecnologías en el lado servidor.

- **Programación de Sistemas de Telecomunicación**: De esta asignatura se aplican los conceptos aprendidos sobre modularidad, encapsulación, definición de interfaces. Son las bases para definir APIs y servicios.

- **Sistemas Operativos**: En esta asignatura se comprende el funcionamiento de un sistema operativo a bajo nivel así como su uso y diseño. A nivel de comandos en la shell es donde más a aportado sin olvidar la base que aporta en cualquier cosa relacionada con la informática.

- **Servicios y Aplicaciones Telemáticas**: Sin ninguna duda la asignatura más cercana al proyecto donde se adquiere conocimiento de diseño y desarrollo de aplicaciones frontend. Aprendimos a definir ficheros JSON y explotarlos, definición de HTML en el navegador, que es el DOM y como interactuar con el mediante JQuery, tecnologías AJAX, CSS, políticas de la máquina Javascript, REST.

- **Ingeniería de Sistemas de Información**: De esta asignatura se aplican conceptos como toma de requisitos, pruebas de software, mantenimiento, control de versiones con Git, metodologías ágiles.

5.3. Lecciones Aprendidas

Tras haber desarrollado esta aplicación vamos a comentar algunos de los retos tecnológicos a los que nos hemos enfrentado.

A pesar de los conocimientos de desarrollo de aplicaciones web adquiridos a lo largo de la carrera, el desarrollo web ha evolucionado mucho en los últimos años y se ha vuelto ciertamente complejo con muchas capas de ingeniería según han ido apareciendo necesidades en la comunidad.

Por lo que hemos tenido que comprender como funcionan los módulos en Javascript y el funcionamiento básico de NodeJS para poder interpretarlos y resolver librerías subidas en ESM.

También hay que conocer funciones y eventos nativos del navegador que por suerte la documentación de Mozilla de MDN es bastante buena y útil [1].

Por otro lado a pesar de haber muchos ejemplos y documentación sobre Webpack ha sido difícil aprender a configurarlo con sus plugins e incluir Typescript en el transpilado inyectando A-Frame como dependencia que no está concebida de este modo, como explicamos en el primer sprint.

También el reto de sentarse a entender librerías de terceros con menos documentación como ha podido ser 'aframe-gui' para construir las interfaces, donde muchas veces tienes que leer incluso el código fuente para comprender el funcionamiento o los fallos que tienen algunas funciones y como solucionar ciertos problemas que aparecen a lo largo del desarrollo.

Por supuesto el hecho de enfrentarse a un proyecto desde cero del cual no sabes nada en una disciplina en la que no se enfoca la carrera y poco se parece el desarrollo respecto al de otros lenguajes de backend con el que es más fácil sentirse familiarizado.

Javascript es un lenguaje orientado a eventos con mucho azúcar sintáctico y es fácil cometer errores si no sigues unas pequeñas guías de estilo. En este sentido Typescript ha ayudado mucho a jerarquizar y tipar todas las interfaces y modelos para detectar problemas y cambios.

El desarrollo del proyecto ha pasado por etapas más ágiles que otras según la resistencia que me han ofrecido ciertos problemas, pero en general no es una aplicación sencilla de desarrollar y ha llevado un tiempo considerable, varias semanas por sprint.

También ha sido muy interesante investigar el mundo de WebAssembly del que se ha hecho

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

uso con el código de las físicas a pesar de no haberlo desarrollado o empaquetado yo mismo me ha servido para estudiar la API y entender la gran revolución que está suponiendo en las capacidades de los navegadores, pues las aplicaciones son bastante sorprendentes como Vocoders, Tracking de objetos en imágenes o aplicaciones y librerías de backend usándose en el navegador que están sirviendo para migrar aplicaciones de escritorio a web.

Respecto a la parte menos técnica, ha sido una gran experiencia enfrentarme a escribir una memoria y preparar una demo para una defensa ante un tribunal que siempre es experiencia para futuros proyectos académicos o laborales.

5.4. Trabajos futuros

Para acercar más este editor de escenas a la funcionalidad que pueden tener los existentes en el mercado comentados anteriormente se podría avanzar en las siguientes ideas:

- A nivel de físicas se podrían incluir funcionalidades más ambiciosas pero probablemente útiles como son el **rol del alfarero** donde se podría ejercer fuerza sobre una figura para contorsionarla y poder moldearla al gusto, ejemplo doblar un cilindro como si fuese una barra o alargarlo estirándolo.

- Otra posibilidad a nivel de físicas podría ser volviendo a roles artesanos **la función del herrero**, se podría tener un instrumento por ejemplo un pico como en el conocido juego de minecraft con el que se pudiera desgastar una superficie poniendo física de partículas fragmentándose la figura en más trozos o perdiendo volumen en el punto de la erosión, quizá también con físicas dejándola caer sobre una superficie.

- En cuanto al aspecto **colaborativo** también se barajó la posibilidad de mediante tecnologías de tiempo real con sockets poder editar una escena con varios usuarios sincronizando el estado igual que hace actualmente el editor de word de google.

- La multiselección por ejemplo se podría hacer con detectores de **colisiones** usando un prisma para chocara con las figuras de la escena.

- Incluir un **buscador de modelos 3D** que tuviéramos en un repositorio o en el servidor de estáticos para insertar elementos como se hace en Unity por ejemplo.

- Trabajar con los **periféricos** con más grados de libertad como son por ejemplo las gafas de valve donde puedes usar las manos con guantes y agarrar objetos, pintar sobre ellos o cualquier tipo de operación que se ocurra.

Bibliografía

[1] Rakesh Baruah. *AR and VR Using the WebXR API*. Apress, 2020.

[2] Jos Dirksen. *Learning Three.js: The JavaScript 3D Library for WebGL*. Packt Publishing, 2013.

[3] Steve Krug. *Don't Make Me Think*. New Riders, 2013.

[4] Cody Lindley. *DOM Enlightenment*. O'Reilly Media, 2013.

[5] Jon Loeliger. *Version Control with Git*. O'Reilly Media, 2012.

[6] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[7] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, 2012.

[8] Tony Parisi. *Programming 3D Applications with HTML5 and WebGL*. O'Reilly Media, 2014.

[9] Kenneth Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Signature, 2012.

[10] Vilic Vane. *TypeScript Design Patterns*. Packt Publishing, 2016.